NUMÉRO D'ORDRE : 2797 ANNÉE : 2000

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Djemai KEBBAL

Tolérance aux fautes et ordonnancement adaptatif dans les systèmes distribués hétérogènes

Thèse soutenue le 14 Novembre 2000 devant la commission d'examen :

Président :	M. Bernard Toursel	Université de Lille 1
Rapporteurs :	M. Guy Bernard	INT-Evry
	M. Gaétan Hains	Université d'Orléans
Directeurs :	M. Jean Marc Geib	Université de Lille 1
	M. El Ghazali Talbi	Université de Lille 1
Examinateurs :	M. Pierre Sens	Université de Paris 6

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. 03 20 43 47 24 Télécopie 03 20 43 65 66

Remerciements

Je tiens à remercier très vivement :

Monsieur Bernard Toursel, Professeur à l'Ecole Universitaire D'Ingénieurs de Lille (EUDIL), pour avoir accepté de présider le jury de cette thèse.

Monsieur Guy Bernard, Professeur à l'Institut National des Télécommunications à Evry, tant pour m'avoir fait l'honneur de rapporter cette thèse, que pour ses remarques et critiques très constructives.

Monsieur Gaétan Hains, Professeur au Laboratoire d'Informatique Fondamentale d'Orléans, de l'Université d'Orléans, pour avoir accepté d'être rapporteur de cette thèse. Ses remarques et conseils m'ont été d'une grande utilité et m'ont aidé à améliorer la qualité de la thèse.

Monsieur Jean-Marc Geib, Professeur à l'Université des Sciences et Technologies de Lille, et Monsieur El-Ghazali Talbi, Maître de Conférences à l'Université des Sciences et Technologies de Lille, pour m'avoir accueilli dans leur équipe et mis les moyens nécessaires pour l'avancement de mes travaux de thèse. Leur conseils et critiques constructifs m'ont permis de finaliser mon travail de recherche.

Monsieur Pierre Sens, Maître de Conférences à l'Université de Paris 6, pour avoir accepté d'examiner ce travail de thèse et pour ses remarques qui ont enrichi ce document.

Je remercie chaleureusement les membres de l'équipe OPAC pour leur soutien. Je remercie particulièrement Mademoiselle Laetitia Jourdan pour m'avoir aidé à compléter les démarches administratives. Mes vifs remerciements vont également à Madame Clarisse Dhanens, Maître de Conférences à l'Ecole Universitaire D'Ingénieurs de Lille (EUDIL) et membre de l'équipe OPAC pour son aide précieuse pour la lecture de la thèse. Ses remarques m'ont aidé à mettre au point ce document.

Je remercie tous mes collègues et amis qui m'ont soutenu tout au long de ce travail de thèse. Mes remerciements vont également à ma famille qui a tant pensé à moi et à mon avenir.

Table des matières

Introduction

Les réseaux de stations de travail (NOWs) et les clusters de processeurs (Alpha Farm, IBM SP2/SP3, etc) suscitent de plus en plus l'intérêt de la communauté "parallélisme". En effet, ces plateformes sont de plus en plus utilisées pour le calcul parallèle et distribué. Cette prolifération rapide est dûe au progrès réalisé dans le domaine des technologies informatiques. D'une part, le rapport performance/coût des stations de travail est en constante hausse. D'autre part, les technologies des réseaux de communication (ATM, Myrinet, etc) ne cessent d'évoluer. Par ailleurs, les besoins en puissance de calcul des applications parallèles ne cessent d'augmenter. Ceci est dû au progrès réalisé dans différents domaines (télécommunication, météorologie, génétique, data-mining, etc).

Problématique

Aujourd'hui, les clusters de processeurs et les réseaux de stations offrent des puissances de calcul comparables à celles offertes par les machines dites massivement parallèles (MPPs). L'interconnexion de ces plateformes permet de délivrer des puissances de calcul importantes.

Plusieurs études ont montré que les réseaux de stations et les clusters de processeurs sont généralement sous-utilisés (de 50 à 95% selon les études). La propriété d'être *multi-utilisateur* permet à plusieurs utilisateurs d'exploiter le potentiel de leurs ressources. L'intégration de ces plateformes donne naissance à des plateformes hybrides (*méta-systèmes*). Le réseau du LIFL (Laboratoire d'Informatique Fondamentale de Lille) est un exemple de méta-système (figure 0.1). Les méta-systèmes sont caractérisés par un ensemble de propriétés rendant difficile leur exploitation :

- **défaillances :** le nombre important de nœuds d'un méta-système et la nature des nœuds et des réseaux font que le risque de défaillance est très important. De plus, l'intervention humaine est une source importante de défaillance ;

- **dynamicité :** la charge des nœuds de la configuration évolue dynamiquement d'une manière imprévisible à cause de l'aspect multi-utilisateur des méta-systèmes. Le système doit réagir dynamiquement à de tels changements en agissant sur les applications qu'il contrôle. L'aspect propriétaire doit être également pris en compte dans l'exploitation de ressources ;

- **hétérogénéité :** les réseaux de stations se caractérisent généralement par une forte hétérogénéité matérielle et logicielle (architecture matérielle, système d'exploitation, puissance relative des processeurs, etc). L'inconvénient d'une telle situation est de limiter la flexibilité des systèmes opérant dans de tels environnements en compliquant certaines opérations (migration, etc) ;

- **multi-application :** les méta-systèmes sont généralement exploités en mode multi-application. Les applications parallèles et séquentielles évoluent ensemble, en compétition pour l'utilisation des mêmes ressources. Ces applications font parfois usage de mécanismes de régulation de charge intra-application, sans considération des autres applications. Une vue globale du système, à travers un ordonnancement multi-application, est cependant indispensable.

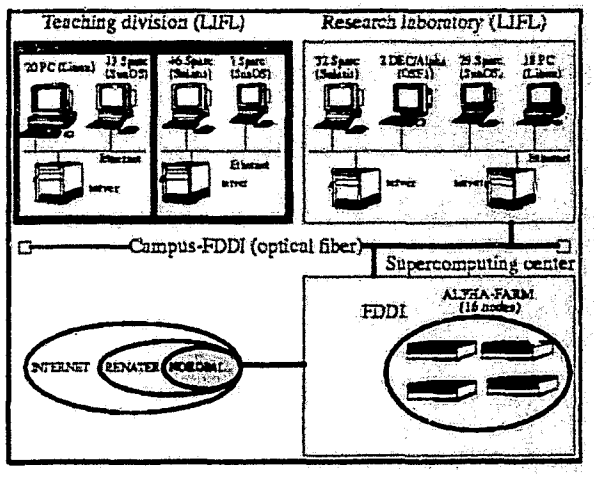


FIG. 0.1 - *Méta-système utilisé au Laboratoire d'Informatique Fondamentale de Lille.*

Les applications que nous visons sont issues des domaines de l'optimisation combinatoire et du calcul scientifique en particulier. Les propriétés principales de ces applications sont :

- **longue durée de vie :** ces applications sont caractérisées par de très longues durées d'exécution. Ceci augmente en particulier le risque de défaillances ;

- **calcul intensif :** les applications considérées font souvent peu d'opérations d'entrées/sorties. En revanche, les temps de communication sont relativement importants pour certaines applications (applications du calcul scientifique) ;

- **potentiel de parallélisme important :** les applications d'optimisation combinatoire et de calcul scientifique possèdent une granularité moyenne voire grosse et un degré de parallélisme important.

L'exploitation des méta systèmes est confrontée à de nombreux problèmes liés à leurs propriétés sus-citées. De nombreux systèmes ont été développés dans le but d'offrir une virtualisation de ces plateformes (PVM, MPI, LINDA, etc). Cependant, ils n'offrent pas d'outils pour l'exploitation **efficace** des ressources.

Objectifs et cadre du travail

Ce travail de thèse rentre dans le cadre du projet MARS (Multi-user Adaptive Resource Scheduler) développé au LIFL dans l'équipe OPAC (Optimisation PArallèle Coopérative). L'objectif du projet MARS est de définir un cadre méthodologique et un support d'exécution pour des applications génératrices d'un parallélisme massif. Les plateformes matérielles considérées sont essentiellement les réseaux de stations et les clusters de processeurs.

Notre objectif est de concevoir et de réaliser un système pour l'ordonnancement de plusieurs applications parallèles dans lequel la tolérance aux fautes occupe une place prépondérante. Les aspects suivants sont développés :

Ordonnancement et gestion de ressources : l'objectif est de concevoir un système ayant une vue globale du méta-système afin d'assurer une gestion efficace des ressources et de répondre aux besoins des applications parallèles. D'une part, cette partie vise à fournir les moyens permettant de développer les applications adaptatives et d'optimiser leur exécution (interfaces de développement, ordonnancement, etc). D'autre part, le système doit pouvoir partager les ressources efficacement entre plusieurs applications parallèles adaptatives, à travers un mécanisme d'ordonnancement multi-application. L'ordonnancement doit pouvoir respecter à la fois les contraintes de l'environnement (charge, propriétaire, etc) et des applications (besoins particuliers, temps de réponse, etc).

Tolérance aux fautes : la tolérance aux fautes est un aspect très important des méta-systèmes. En effet, le risque de défaillance important de ces plateformes et les durées de vie considérables des applications font que cette composante doit être prise en compte. La tolérance aux fautes est considérée à deux niveaux. D'une part, les applications parallèles doivent disposer d'un mécanisme de tolérance aux fautes non-coûteux, leur permettant de s'affranchir du problème de défaillance. D'autre part, le système doit pouvoir détecter certaines défaillances et informer les applications. De plus, il doit pouvoir, avec la complicité des applications, procéder à des recouvrements automatiques et veiller au bon fonctionnement du système.

Applications : la validation du projet est faite à travers plusieurs applications d'optimisation combinatoire (recherche tabou, recuit simulé, etc) et de calcul scientifique (Gauss, Gauss-Jordan, multiplication de matrices, etc). Ces applications, caractérisées par de longues durées d'exécution, ont montré l'intérêt de la tolérance aux fautes et de l'ordonnancement adaptatif.

Organisation de la thèse

La thèse est organisée en trois parties et trois annexes :

- la **première partie** est consacrée à la tolérance aux fautes en général, et aux algorithmes de sauvegarde/reprise en particulier. Le *chapitre 1* présente un état de l'art sur la tolérance aux fautes et les algorithmes de sauvegarde/reprise. Dans le *chapitre 2*, nous abordons les applications parallèles adaptatives et nous décrivons la conception et la mise en œuvre d'un algorithme de sauvegarde/reprise que nous avons développé pour cette classe d'applications. Le *chapitre 3* est consacré à l'intégration du mécanisme de tolérance aux fautes développé dans le système MARS. Ainsi, nous discutons de la détection des défaillances et de la reprise des applications ;
- la **seconde partie** traite les problèmes d'ordonnancement et de régulation de charge. Cette partie est scindée en deux chapitres. Le *chapitre 4* est consacré aux problèmes liés à l'exécution d'applications parallèles adaptatives. Dans ce chapitre, nous présentons un modèle pour la construction et l'ordonnancement d'une application adaptative. L'algorithme d'ordonnancement adopté est ensuite détaillé et évalué. Le *chapitre 5* est consacré à l'ordonnancement multi-application. L'objectif est de pouvoir exécuter plusieurs applications parallèles simultanément. Le système développé est discuté et l'algorithme d'ordonnancement multi-application est présenté et ses performances évaluées ;
- dans la **troisième partie**, nous nous intéressons aux applications parallèles adaptatives. Le *chapitre 6* est consacré à la présentation de la méthodologie de développement d'applications adaptatives. Dans le *chapitre 7*, un ensemble d'applications, issues des domaines de l'optimisation combinatoire et du calcul scientifique développées ou portées pour valider le système, sont étudiées. La mise en œuvre parallèle adaptative ainsi que quelques résultats de performances seront présentés.

L'annexe A présente un résumé des fonctions de l'API (Application Programmer Interface) de la bibliothèque de programmation parallèle adaptative et des primitives de gestion de la tolérance aux fautes. Dans l'annexe B, nous présentons les détails de la détermination de la période optimale de sauvegarde. L'annexe C présente une table des symboles utilisés dans la thèse, en particulier dans les chapitres 4, 5 et 7.

Mots clés et terminologie

Tolérance aux fautes, sauvegarde/reprise, détection et recouvrement des défaillances, ordonnancement d'applications parallèles adaptatives, ordonnancement multi application, régulation de charge, optimisation combinatoire, calcul scientifique.

Dans tout ce qui suit, le terme nœud sera utilisé pour désigner une station de travail, un nœud ou un processeur d'une MPP.

Première partie

Tolérance aux fautes

Chapitre 1

Tolérance aux fautes

Dans les systèmes et les procédés industriels, la faute et la défaillance sont naturelles, résultant généralement des imperfections dans la conception, la mise en œuvre et dans l'exploitation du système. Ainsi, la sûreté de fonctionnement a toujours constitué une composante importante dans le cycle de vie et une mesure de qualité de tels systèmes. Ce chapitre évoquera ce sujet dans les systèmes informatiques. Nous nous focaliserons sur une méthode de sûreté de fonctionnement qui considère que la présence de fautes est inévitable : il s'agit de *la tolérance aux fautes*. Cette approche met l'accent sur les méthodes et les moyens permettant de faire face aux fautes au moment même de leur apparition. La tolérance aux fautes a fait son apparition à l'aube de l'informatique. Ceci s'explique par le fait que la fiabilité n'était pas le point fort des premiers systèmes informatiques. Les premiers travaux effectifs dans le domaine remontent aux premières années de l'apparition des systèmes informatiques [Sha48, Ham50, Neu56].

La section suivante présente quelques notions de sûreté de fonctionnement. Dans la section 1.2, nous présentons une synthèse des techniques de tolérance aux fautes et en particulier les méthodes dites de *reprise par recouvrement arrière*. La section 1.3 est consacrée à une classe d'approches de tolérance aux fautes appelée la *sauvegarde/reprise*. Ces approches sont des techniques de reprise par recouvrement arrière utilisant l'abstraction de *processus fiable* et de *mémoire stable*. Cette étude permet, en particulier, de mettre le point sur la complexité des algorithmes de sauvegarde/reprise. Dans le chapitre suivant, nous proposons un algorithme de sauvegarde/reprise présentant des propriétés intéressantes pour une classe d'applications parallèles : les *applications adaptatives*.

1.1 Sûreté de fonctionnement et tolérance aux fautes

Dans le rapport ARAGO15 de l'*Observatoire Français des Techniques Avancées* [dTA94], le terme **faute** est défini comme étant la cause adjugée ou supposée d'une défaillance. Cette cause peut être accidentelle ou intentionnelle.

La sûreté de fonctionnement[1] d'un système se mesure par sa capacité à fonctionner correctement[2] en présence ou non de fautes. Dans [Lap94b], Laprie définit cette notion comme étant la propriété qui permet aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre. Dans [Con96], cette notion est définie comme étant la qualité du service qu'un système délivre. Cette propriété peut être assurée en utilisant plusieurs approches : éviter les fautes, tolérer les fautes, etc. La tolérance aux fautes est donc un moyen de sûreté de fonctionnement d'un système.

La tolérance aux fautes d'un système est sa capacité à fonctionner en présence de fautes. Le degré ou la qualité de la tolérance aux fautes peut varier d'une garantie de service totale à l'arrêt propre du système en passant par un fonctionnement en mode dégradé.

En réalité, la tolérance aux fautes n'est pas le seul moyen pour assurer la sûreté de fonctionnement d'un système. Laprie définit et traite trois aspects de la sûreté de fonctionnement [Lap94b]. Il s'agit de ses attributs, des moyens utilisés et de ses entraves. Ainsi, il cite plusieurs approches pour construire des systèmes sûrs de fonctionnement dont les utilisations peuvent être combinées.

1.1.1 Moyens de la sûreté de fonctionnement

Les moyens de la sûreté de fonctionnement définissent les méthodes et les approches utilisées pour assurer cette propriété. Les approches les plus connues sont :

- **la prévention des fautes** qui se penche sur les moyens permettant d'éviter l'occurence des 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 des fautes ou leur impact. 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 vis-à-vis des fautes [HTI97] ;

- **la tolérance aux fautes** qui essaye de fonctionner en dépit des fautes. Le degré de tolérance aux fautes se mesure par la capacité du système à continuer à délivrer son service en présence de fautes.

1. En anglais reliability ou dependability.
2. Conformément à sa spécification.

1.1.2 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 doit vérifier la sûreté de fonctionnement du système. Ces propriétés définissent des aspects d'évaluation de la qualité du service d'un système. Laprie définit six attributs de la sûreté de fonctionnement [Lap94b] :

- **disponibilité :** c'est la propriété requise par la plupart des systèmes sûrs de fonctionnement. Il s'agit de la fraction de temps durant lequel le système est utilisé pour des fins utiles (en excluant les temps de défaillances et de réparations) ;
- **fiabilité :** cet attribut évalue la continuité du service i.e. le taux en temps de fonctionnement sans faute du système ;
- **sécurité-innocuité :** similaire à la fiabilité mais par rapport aux fautes catastrophiques ;
- **confidentialité :** cet attribut évalue l'aptitude 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 [Jac91] ;
- **maintenabilité :** cette propriété décrit la souplesse du système vis-à-vis des modifications apportées en vue de sa maintenance.

L'importance des attributs est étroitement liée à l'application et à ses besoins. Dans [Sie91], quatre classes d'applications ont été définies :

- les applications *commerciales à usage général* qui donnent une importance modérée à la fiabilité et tolèrent les défaillances temporaires ;
- les application *hautement disponibles* qui mettent l'accent sur la sécurité-innocuité (les systèmes de réservation de places) ;
- les applications à *longue durée de vie* qui font prévaloir la maintenabilité et la fiabilité ;
- les applications *critiques* qui donnent une grande importance à tous les attributs (pilotage de fusées, etc).

1.1.3 Les entraves à la sûreté de fonctionnement

Ce sont les causes et les conséquences de la non-sûreté de fonctionnement. Il s'agit des *fautes*, *erreurs*, et *défaillances*. Une *défaillance* d'une partie ou de la totalité du système survient lorsque le service délivré par celui-ci n'est pas conforme à sa spécification. Une *erreur* est la partie de l'état du système susceptible de provoquer sa défaillance. Une *faute* est la cause supposée d'une erreur [Lap94b]. Lorsqu'une faute arrive à mettre un ou plusieurs composants du système dans un état inconsistant, une erreur est alors provoquée. Lorsqu'un service utilise cette partie inconsistante du système, le système se comporte d'une manière non conforme à ses fonctions ce qui provoque sa défaillance.

1.1.3.1 Fautes

La faute peut être caractérisée par sa nature, son origine et son étendue temporelle [Lap94b]. La nature d'une faute précise la manière dont elle a été provoquée: *intentionnelle* ou *accidentelle*. L'origine d'une faute relève de la cause de son apparition. La faute peut être liée à la *création et au fonctionnement* du système (conception, mise en œuvre, exploitation, etc), *phénoménologique* liée à l'influence de l'environnement [Avi78] (physique, humaine) ou liée aux *frontières* du système (interne, résultant de la propagation d'effets d'autres fautes: logique maligne, ou externe résultant des interférences avec le monde externe: chaleur, électromagnétisme) [Lap94b]. L'étendue temporelle d'une faute caractérise sa persistance ou sa durée: *temporaire* ou *permanente*. Une autre caractéristique peut être attribuée à une faute: son étendue ou sa portée dans le système [Nel90, Con96]. Lorsque la faute provoque la défaillance totale du système, on parle d'une faute *globale*.

1.1.3.2 Erreurs

Une erreur est la conséquence d'une faute. Elle reflète l'état des parties affectées du système. Dès que le système utilise un état erroné, une défaillance survient. Les erreurs sont classées selon leur type. La forme la plus courante d'une erreur est la non correspondance d'un service à sa spécification. Powell [Pow88] parle également du type d'erreurs en relation avec le temps. Ces erreurs sont caractérisées par le fait que le service demandé ne respecte pas ses contraintes temporelles. Ainsi, le service peut être en retard ou en avance. Powell [Pow88] définit plusieurs classes d'erreurs suivant que le service est toujours en avance, toujours en retard ou encore arbitraire. L'omission du système de rendre le service est considérée également comme un type d'erreur. Dans [Con96], un arrêt (crash) est défini par le fait que le système omet définitivement de délivrer des services. L'erreur dépend également de l'activité du système qui peut l'éliminer avant de provoquer des défaillances. Enfin, l'interprétation et la définition d'une défaillance diffèrent d'un utilisateur à l'autre (généralement, une défaillance est définie par rapport à un taux d'erreurs fixé par l'utilisateur).

1.1.3.3 Défaillances

La défaillance correspond à la perception de l'état erroné du système par l'utilisateur. En d'autres termes, le service qu'il délivre ne correspond plus à ce qu'il doit être. Ceci inclut également les cas où le service n'est pas délivré (omission) ou lorsque sa délivrance ne respecte pas les contraintes temporelles associées au service. En conséquence, une défaillance peut être caractérisée par son *domaine* dont les éléments sont ceux définis pour les erreurs : défaillance par valeur ou temporelle [Pow88]. La défaillance peut être caractérisée également par la façon dont elle se manifeste pour chaque utilisateur (cohérente lorsqu'elle se manifeste de la même manière à tous les utilisateurs ou byzantine si chaque utilisateur la perçoit différemment des autres utilisateurs [PSL80, LSP82]). Lorsqu'elle survient, une défaillance ne s'arrête pas au niveau du système mais le dépasse pour affecter son environnement.

Powell [Pow88] définit un graphe reliant les modes de défaillances aux domaines des erreurs qui les ont provoquées. Le graphe schématise la sévérité des défaillances suivant les deux catégories d'erreurs. La valeur la plus faible correspond à un système sans défaillance et la valeur la plus forte correspond aux défaillances provoquées par des erreurs arbitraires (par valeur et temporelles).

Dans le domaine des systèmes distribués, une classe de défaillances est souvent sollicitée. Il s'agit des défaillances par arrêt qui sont liées à la fois au domaine des erreurs par valeur et temporelles. Ces défaillances sont constatées par la rupture de l'activité du système par l'utilisateur. On parle alors des *systèmes à arrêt sur défaillances*. Si le système s'arrête après la délivrance d'une valeur correcte mentionnant son arrêt, il est qualifié de *système figé sur défaillances (fail-stop)*. En revanche, si l'arrêt est constaté par l'absence d'événements tels que l'envoi de messages dans un système réparti, on parle de *systèmes à silence sur défaillances (crash failure)* [PBS+88, Lap94b]. Ce sujet sera traité dans le chapitre 3.

Notre objectif est la sûreté de fonctionnement dans les systèmes distribués ayant comme plateforme matérielle les réseaux de stations de travail en particulier. Le moyen utilisé est la tolérance aux fautes et les attributs de la sûreté de fonctionnement recherchés sont modérément la fiabilité et la disponibilité. Les défaillances considérées sont généralement celles liées au mode *silence sur défaillances*. Le choix de ce mode de défaillance est motivé par sa simplicité et son réalisme, étant donnée la difficulté, voire l'impossibilité, de mettre en œuvre un modèle figé sur défaillance dans les systèmes distribués (voir chapitre 3).

1.2 Techniques de tolérance aux fautes

Les motivations principales pour l'adoption de la tolérance aux fautes sont économiques. Par conséquent, l'attribut le plus recherché est la disponibilité. Pour les systèmes où la sécurité-innocuité est indispensable, la motivation est également d'ordre réglementaire [Lap94a]. Par ailleurs, le coût des défaillances varie suivant les utilisa-

teurs et dépend de multiples facteurs. Laprie affirme que les coûts liés à la non-sûreté de fonctionnement des systèmes informatiques sont parmi les plus grandes sources de sinistres industriels (ils représentent environ 7% en moyenne du chiffre d'affaires des 500 premières sociétés d'informatique en France).

La tolérance aux fautes est assurée selon deux approches : le traitement d'erreurs et le traitement des fautes [AL81]. Le traitement des fautes a pour objectif d'éviter la propagation des fautes. Le traitement des erreurs vise à empêcher qu'une erreur ne provoque la défaillance d'un ou de plusieurs composants. Le traitement d'erreurs peut être réalisé selon deux approches différentes [Lap94b] :

- *le recouvrement d'erreurs* qui consiste à remplacer l'état erroné par un état dépourvu de fautes [AL81]. Le recouvrement peut être effectué par *reprise* où des points de reprise sont prévus afin de pouvoir restaurer le système dans un état antérieur survenu avant l'erreur. La seconde alternative consiste à *poursuivre* (éventuellement en mode dégradé) en mettant le système dans un nouvel état[3] ;

- *la compensation d'erreurs* dans laquelle la redondance assure la délivrance de service malgré les erreurs apparues.

Le traitement des fautes, quant à lui, est effectué en plusieurs étapes [Lap94b] : tout d'abord un *diagnostic de faute* est entrepris. Cela consiste à déterminer la nature et la localisation de la faute. Par la suite, la place est laissée à l'étape de *passivation de la faute*. Il s'agit de l'opération principale dans le processus de traitement de fautes, qui consiste à empêcher une nouvelle activation de la faute en retirant les composants considérés comme fautifs du système[4]. En fait, l'étape de passivation n'est entreprise que si le traitement d'erreurs n'a pas pu éliminer la faute ou si la probabilité de sa récurrence n'est pas négligeable.

Une faute est considérée *douce* tant que l'étape de passivation n'a pas eu lieu. Une fois la passivation entreprise, la faute devient *dure* [Lap94b]. On distingue également la notion de fautes *indépendantes/corrélées* et de défaillances *séquentielles/simultanées*. Globalement, la tolérance aux fautes dans les systèmes informatiques peut être assurée au niveau matériel et/ou logiciel.

1.2.1 Techniques de tolérance matérielle aux fautes matérielles

La tolérance matérielle aux fautes matérielles peut être assurée au niveau microscopique (circuits élémentaires) ou au niveau macroscopique (unités de traitement).

3. Le recouvrement par reprise est appelé également recouvrement arrière et le recouvrement par poursuite est appelé recouvrement avant [DVCL93].

4. Cela peut provoquer la reconfiguration du système.

Au niveau des composants matériels élémentaires, les techniques de tolérance aux fautes reposent généralement sur les techniques de redondance et d'autotest [CC94, Cro78]. La tolérance aux fautes matérielles peut être garantie à un niveau plus élevé également. La redondance est de nouveau l'approche la plus utilisée. Les approches varient de la réplication d'un ensemble d'organes (bus, disques de sauvegarde, processeurs, etc) jusqu'à prévoir des systèmes de rechange complets. Dans le cas de processeurs multiples, la notion de *"secondaires" (backup)* est utilisée. Il s'agit de prévoir une ou plusieurs machines redondantes qui serviront de répliques à la machine *primaire* [DVCL93].

1.2.2 Techniques logicielles de tolérance aux fautes logicielles

Les techniques de tolérance aux fautes logicielles diffèrent suivant le modèle d'application et le type du système sous-jacent. Dans le domaine des systèmes distribués, on trouve des méthodes basées sur les *groupes* dans lesquels, la communication est effectuée par diffusion de messages entre les membres du groupe. La cohérence du système est maintenue en utilisant des algorithmes de gestion de groupe et le traitement d'erreurs est effectué par recouvrement ou par compensation si le groupe comporte des répliques. Dans cette catégorie, on trouve généralement les méthodes de réplication (modèle des primaires/secondaires, machines à états, etc).

Dans les modèles d'applications composées de processus communiquant par envoi de messages point-à-point, une seconde classe d'approches est utilisée. En effet, le traitement d'erreurs est généralement effectué par recouvrement arrière à partir des fichiers d'états (sauvegardes) construits au cours de l'exécution de l'application. La cohérence est assurée par des algorithmes de définition d'états globaux cohérents.

Le modèle des transactions, ou *"des objets et des actions"*, vise à assurer l'intégrité des données [MS92]. Ainsi, toute opération ou tout service est exprimé en terme d'actions atomiques dont l'exécution est indivisible. Ces modèles utilisent le recouvrement arrière pour annuler l'effet d'une transaction et communiquent par envoi de messages ou par appels de procédures à distance.

A- Abstractions du système d'exploitation pour la tolérance aux fautes logicielles : Dans tous ces modèles, la tolérance aux fautes est réalisée en utilisant un ensemble commun d'outils de base qui peuvent être des services du système d'exploitation. Mishra et Shlichting identifient un ensemble de ces abstractions qu'ils classifient en deux catégories principales. La première comporte les abstractions qui fournissent des fonctionnalités similaires à celles fournies par les systèmes d'exploitation ou le matériel traditionnel (mémoire stable, actions atomiques, processus fiables, RPCs). L'autre catégorie englobe les mécanismes qui assurent la cohérence des informations fournies à l'ensemble des composants du système distribué (temps global, composition de groupe, diffusion). Ces fonctionnalités sont généralement fournies par les mécanismes de tolérance aux fautes [MS92].

Le temps global commun est une notion fondamentale dans la théorie des sys-

tèmes distribués [Lam78]. Il sert d'outil pour ordonner certains événements apparaissant dans le système distribué. Le temps peut être physique ou logique, et l'horloge physique utilisée globale ou locale à chaque nœud [Lam78, RSB90, Cri89, LMS85].

L'action atomique est une abstraction fournie généralement par les systèmes distribués tolérants aux fautes. Le principe est de garantir l'exécution complète de l'action ou d'annuler ses effets en cas de non terminaison et ceci, malgré la concurrence[5] et les fautes. Cette notion est sollicitée dans les systèmes transactionnels où des données de longue durée de vie, stockées en mémoire stable, sont sujettes aux accès concomitants et aux fautes des processeurs [MS92]. De plus, cette notion est utilisée dans d'autres mécanismes tels que la validation des *"points de reprise temporaires"* et la diffusion atomique. Les actions atomiques doivent respecter les propriétés dites **ACID** : *atomicité, cohérence, isolation* et *durabilité* [LMP94, Ske82, BSW79, SM77, HY86, BHG87].

L'appel de procédure à distance (RPC, Remote Procedure Call) est une abstraction fournissant à la fois un moyen de communication inter-processus dans un système distribué et une sémantique bien connue des langages de programmation : l'appel procédurale. Le RPC diffère de l'appel de procédure classique puisque l'appelant (*client*) et l'appelé (*serveur*) sont deux processus différents qui peuvent se trouver sur des sites différents. Les paramètres de la procédure sont acheminés au serveur à travers le réseau de communication. Ce dernier invoque le service demandé et réenvoie le résultat. La synchronisation est assurée en contraignant le client à se bloquer jusqu'à la réception du résultat [MS92]. La présence des fautes peut donner naissance à des procédures *orphelines*, qui altèrent l'ordre des appels et provoquent ainsi la révision des schémas de communication et d'exécution précédents [PS88].

La diffusion permet d'envoyer un même message à tous les membres d'un groupe. La diffusion relève ainsi des services de communication et de groupes. En effet, le mécanisme de groupe fournit une information (composition du groupe) au mécanisme de la diffusion. Cela est encore plus important en présence de fautes où les groupes changent de membres dynamiquement. Suivant les types d'applications supportées et leur contraintes, le mécanisme de diffusion peut ou doit vérifier un sous-ensemble des propriétés suivantes [MS92] : *fiabilité* [CT96], *atomicité*[6] [Sch90, CASD85, Anc93, Bir96], *ordre (total, partiel, etc)* [Bir93, Lam78], *terminaison*. Les approches utilisées pour réaliser ce service diffèrent généralement suivant les propriétés recherchées, le mode de défaillance, le modèle du réseau de communication, le nombre de fautes simultanées toléré, etc [Bir93, CT96, CASD85, GMS91, Bir96].

Le processus fiable est un processus s'exécutant correctement même après avoir été interrompu par une défaillance et avoir procédé à un recouvrement [MS92]. Les processus fiables sont généralement réalisés via des techniques de *recouvrement ar-*

5. Plusieurs chercheurs français [dTA94] préfèrent le terme concomitance qui traduit mieux le terme anglais counccurency et exprime mieux la simultanéité.

6. Il a été démontré que la diffusion atomique est équivalente au problème du consesus [CT96]. Par conséquent, elle ne possède pas de solution déterministe dans un environnement asynchrone même avec la défaillance d'un seul processeur et par arrêt (crash failure) [FLP85].

rière basées sur la sauvegarde de l'état du processus sur un *espace de stockage stable*. La *mémoire stable* est une abstraction fournie par le matériel, ou, par le matériel et le logiciel si des procédures d'accès atomique sont fournies. La mémoire stable est l'espace de stockage demeurant inchangé après une défaillance (disques, bandes, etc). Sa mise en œuvre peut être effectuée en utilisant une unité stable d'une machine ou en dupliquant la sauvegarde sur plusieurs machines. L'accès est alors effectué en utilisant des protocoles d'accès atomique. Les mécanismes de *sauvegarde/reprise (checkpoiting/roll-back)* sont basés sur ces deux abstractions.

B- Paradigmes de programmation tolérante aux fautes : La figure 1.1 illustre les dépendances entre les paradigmes de programmation tolérante aux fautes et les abstractions présentées avant [MS92].

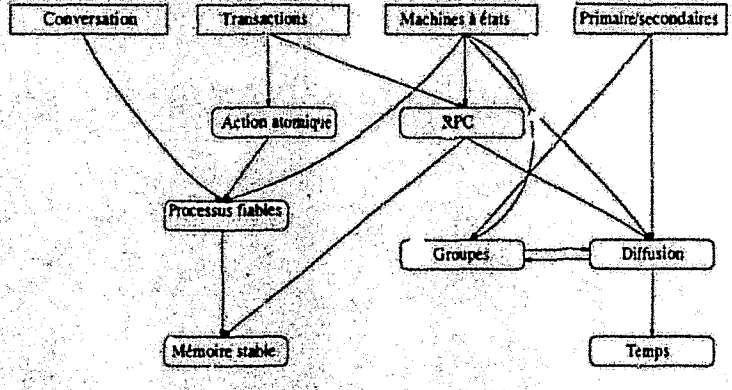


FIG. 1.1 - *Dépendances entre les paradigmes de programmation et les abstractions du système d'exploitation.*

Dans ce qui suit, nous qualifions de *tolérance aux fautes logicielles* toutes les techniques mettant en œuvre le logiciel pour tolérer toutes sortes de fautes (logicielles et/ou matérielles). Les techniques de tolérance aux fautes logicielles reposent sur des approches de génie logiciel dans lesquelles, des paradigmes de programmation tolérante aux fautes sont employés dans le développement des applications. Ces paradigmes consistent généralement en la diversification fonctionnelle [Avi85] et les exceptions [CL94].

La méthode des exceptions (approche *"fail-fast"*) relève de la programmation défensive. La détection d'erreurs est effectuée par des assertions exécutables sur les données et les résultats. Le traitement d'erreurs est effectué par le traitement d'exceptions. Cette approche vise principalement à empêcher la propagation des erreurs afin d'éviter qu'une sollicitation d'un composant défaillant ne provoque la défaillance d'autres composants du système.

La diversification fonctionnelle, quant à elle, consiste généralement à établir des exemplaires multiples (*variantes*) du système tolérant aux fautes [CA78]. Afin de

pouvoir tolérer les fautes de conception, les variantes doivent être supervisées par un *décideur* qui doit fournir un résultat supposé correct. Pour cela, les points de décision, les données sur lesquelles la décision est effectuée et la manière dont les décisions sont prises doivent être spécifiés [CL94]. La diversification fonctionnelle englobe deux classes d'approches : les blocs de recouvrement et la réplication.

B.1- Blocs de recouvrement : Dans les blocs de recouvrement [Ran75], l'application est organisée en un ensemble d'alternatives appelées les *alternants* et un test d'*acceptation* est spécifié. Les alternants sont essayés un à un jusqu'à la satisfaction du test d'acceptation. Si toutes les alternatives sont essayées sans succès, le bloc de recouvrement est considéré défaillant.

B.2- Réplication : Les techniques de réplication sont utilisées notamment dans le domaine des serveurs afin d'augmenter la disponibilité et le degré de tolérance aux fautes, en particulier, dans les systèmes critiques. Ces méthodes consistent à exécuter plusieurs exemplaires du composant logiciel appelés *répliques* [GS97]. Cependant, elles diffèrent suivant la manière dont ces répliques sont organisées, mises à jour et la façon dont les décisions sont prises. Dans la réplication *passive (modèle du primaire et des secondaires)*, une réplique particulière appelée *primaire* reçoit les entrées (les messages) et produit les résultats [CA78]). Les secondaires ne font que suivre en mettant à jour leur état à partir d'un fichier d'état produit régulièrement par le primaire. Dans la réplication active (*machines à états* [Sch90]), les messages d'entrées sont diffusés à toutes les répliques et les résultats sont généralement comparés [GS97]. Dans la réplication semi-active [LMP94], les messages sont diffusés à toutes les variantes (secondaires). En revanche, seule la réplique primaire est habilitée à produire des résultats.

Ces approches logicielles sont en réalité des méthodes de génie logiciel dans lesquelles, la transparence de la tolérance aux fautes n'est pas entièrement respectée. Les méthodes de sauvegarde/reprise basées sur les abstractions de processus fiables et de mémoire stable peuvent être vues comme un service fourni par le système d'exploitation ou par des librairies complémentaires. Outre la transparence de la plupart des techniques de cette approche, l'efficacité, le coût peu élevé en terme de matériel (pas de redondance ni de matériel spécifique) et la non sensibilité aux fautes transitoires constituent les motivations principales pour l'adoption de ces approches dans le domaine des *"applications commerciales à usage général"*.

1.3 La sauvegarde/reprise

Nous qualifions le mécanisme de constitution de points de reprise, utilisés pour effectuer des reprises en cas de défaillances, par *sauvegarde/reprise (checkpointing/rollback)*. La figure 1.2 illustre une séquence d'utilisation typique du mécanisme de sauvegarde/reprise. Périodiquement, l'application enregistre son état sur un support de stockage stable. Dès l'occurence d'une défaillance (croix sur la figure), l'application

effectue un recouvrement arrière par reprise en reconstruisant son état à partir d'un état antérieur sauvegardé précédemment en mémoire stable (les points de reprise sont appelés également lignes de recouvrement [DVCL93]).

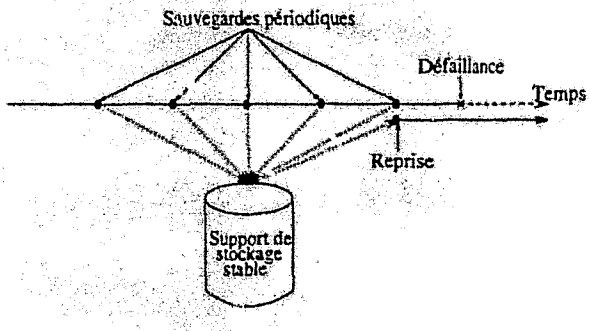


FIG. 1.2 - *Sauvegarde/reprise typique d'une application.*

Outre la reprise sur erreur, l'utilisation du mécanisme de sauvegarde/reprise est très réputée à la fois pour les applications séquentielles et pour les applications distribuées. Parmi les domaines d'utilisation, citons : la migration de processus [MDP+96], le débogage distribué [FPR95, FB89], le déplacement d'applications (job swapping) [TL95], la résolution des interblocages [KS94] et bien d'autres utilisations.

1.3.1 Le modèle du système distribué

Nous nous plaçons dans un contexte, le plus général possible, de système distribué. Le système est composé d'un ensemble de nœuds de calcul (stations de travail) reliés par un réseau d'interconnexion (réseau de communication). Le schéma de communication entre les nœuds constitue un graphe complet i.e. un lien virtuel direct existe entre chaque paire de nœuds. Les délais de communication sont inconnus et la vitesse relative des processeurs n'est pas connue avec une précision suffisante, ce qui donne un système asynchrone. Les nœuds communiquent par envoi de messages point-à-point et disposent d'une horloge locale chacun.

1.3.2 Application distribuée, application séquentielle et état d'une application

Une application séquentielle est composée d'un unique processus s'exécutant sur un seul nœud. L'état d'une telle application se compose des parties suivantes :

1. *contenu de la mémoire volatile* : espace des données globales, arborescence des appels fonctionnels et données locales, et code du programme ;

2. *état des registres du processeur*: le contenu des registres du processeur détermine, en particulier, l'instruction en cours d'exécution (état de la progression d'exécution, pointeur de pile, etc);

3. *état du noyau vis-à-vis du processus*: il englobe toutes les informations maintenues par le noyau concernant le processus et les ressources du système qu'il utilise (fichiers, signaux, etc).

Ainsi défini, l'état d'une application séquentielle, à un instant donné t noté $s(t)$, comporte toutes ces informations. Le temps t ne correspond pas forcément au temps réel mais plutôt aux instants de changement d'état ($t \in \mathbb{N}$).

L'évolution de l'application peut s'exprimer par le passage supposé atomique d'un état vers un autre. L'exécution de l'application séquentielle est caractérisée par la séquence: $(s(t_0), s(t_1), \ldots, s(t_{n-1}))$; $s(t_0)$ et $s(t_{n-1})$ représentent respectivement l'état initial et l'état final de cette exécution. $s(t_i) \in S$, où S est l'ensemble des états que peuvent prendre toutes les exécutions possibles de l'application séquentielle.

Une application parallèle ou distribuée est composée de n processus séquentiels communiquant via le réseau d'interconnexion. Le schéma de communication et de disposition des processus sur les nœuds est lié au système distribué sous-jacent.

L'état d'une application distribuée dans ce cas est composé des états locaux des processus la constituant et des messages échangés (empruntant le réseau d'interconnexion, en route vers leurs destinataires). Par conséquent, l'état $s(t)$ d'une application distribuée à un instant physique t est constitué des états locaux de tous les processus de l'application et des messages en transit dans les canaux de communication à cet instant. Cette définition entraine deux problèmes importants:

- la difficulté d'établir un temps physique global, en particulier dans un système distribué asynchrone. Cependant, un temps physique global n'est pas toujours indispensable pour la définition d'état global d'applications distribuées;

- les messages en transit à un moment donné ne peuvent être captés par aucun processus du fait que chaque processus ne peut enregistrer que son état local ainsi que les messages qu'il envoie et qu'il reçoit.

Ces problèmes relèvent de la détermination d'état global cohérent d'une application en vue d'une reprise sur erreur. Les algorithmes de sauvegarde/reprise sont basés sur ce concept. La sous-section suivante est consacrée à l'étude et à la présentation de ces algorithmes.

1.3.3 Les algorithmes de sauvegarde/reprise

Plusieurs classes d'algorithmes de sauvegarde/reprise existent. La différence majeure entre elles réside dans la manière dont l'état global cohérent est déterminé

(avant ou après le recouvrement) et la façon dont les états locaux sont enregistrés (synchrone, asynchrone ou hybride) ainsi que dans le degré de transparence vis-à-vis du programmeur. Suivant le degré de transparence, ces algorithmes peuvent être classés en trois classes principales : les techniques explicites, les techniques semi-transparentes et les approches transparentes (figure 1.3) [DVCL93, KTG97b].

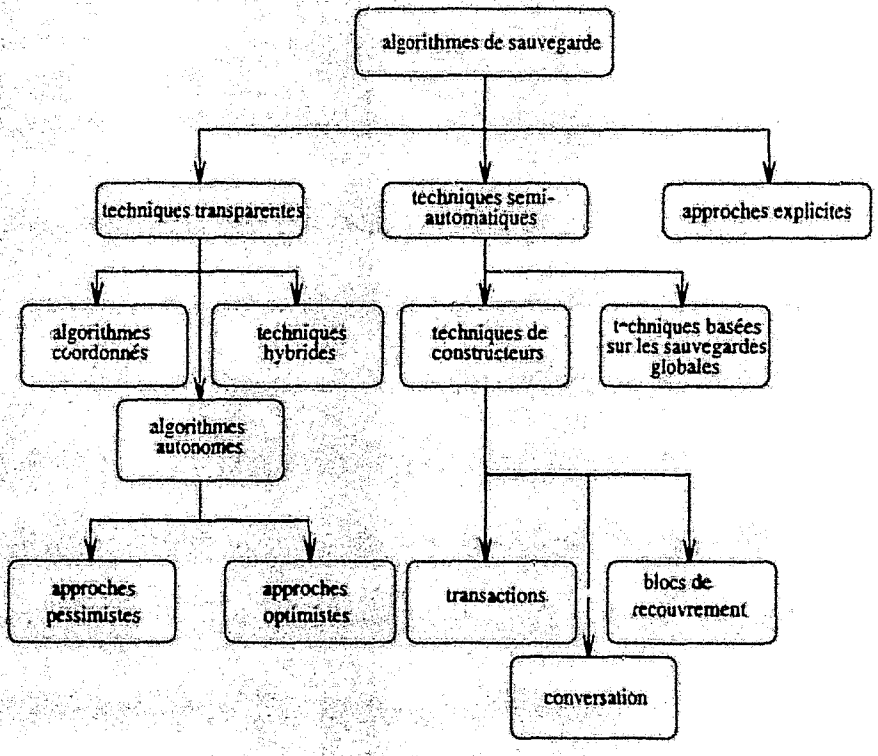


FIG. 1.3 - *Classification des algorithmes de sauvegarde/reprise.*

1.3.3.1 Les approches explicites

Ce sont des techniques dépendantes directement de l'application dans lesquelles, l'implémentation et la gestion des sauvegardes/reprises sont à la charge de l'utilisateur. Les objectifs de ces techniques sont l'exploitation des propriétés particulières de l'application afin de réduire, à la fois, le coût des sauvegardes et des reprises ainsi que la taille des fichiers d'états résultants. Ceci au prix d'une complexité accentuée de la programmation.

Dans [PKD94], un tel algorithme est proposé dans le but de construire des applications de calcul scientifique tolérantes aux fautes. L'approche évite l'utilisation de la mémoire stable (en l'occurence le disque) dans le but de réduire le coût de sauvegarde. Au lieu de cela, le fichier d'état de chaque processus est maintenu en mémoire volatile (supposée à accès plus rapide). De plus, un fichier contenant le

résultat du *ou exclusif sur les bits* de toutes les sauvegardes est maintenu sur un processeur additionnel, dit de *parité*, permettant ainsi de tolérer une faute à la fois. Le contenu des fichiers d'états et la ligne de recouvrement sont définis par l'application. Outre les performances, le problème de l'hétérogénéité est résolu, du fait que l'application parallèle peut être reconstruite à partir des données formatées et non pas à partir des espaces d'adressage des processus qui sont fortement dépendants de l'architecture du nœud et du système d'exploitation. En revanche, la complexité de la programmation est fortement accentuée.

1.3.3.2 Les approches semi-transparentes

Dans cette classe d'algorithmes, le programmeur est directement impliqué dans la définition de la ligne de recouvrement (état global cohérent de l'application). En revanche, le mécanisme de construction de fichier de sauvegarde élémentaire d'un composant de l'application est fourni par des bibliothèques transparentes. Certains auteurs classent les approches de génie logiciel pour la construction d'applications tolérantes aux fautes dans la catégorie des algorithmes de sauvegarde/reprise du fait qu'elles utilisent les abstractions de mémoire stable et de processus fiables [DVCL93].

Dans les approches basées sur la sauvegarde/reprise globale, la sauvegarde est invoquée par l'application qui doit déterminer d'abord un état global cohérent de l'application avant de déclencher l'opération au niveau de chaque processus. La sauvegarde de l'état de chaque processus en revanche est fournie par la bibliothèque. L'intervention de l'utilisateur peut aller jusqu'à la spécification du contenu des fichiers de sauvegarde. Dans les approches basées sur les transactions (actions atomiques), les objets à modifier par la transaction sont dupliqués avant d'entreprendre la moindre opération. La garantie de l'atomicité et la validation de la transaction sont réalisées par la bibliothèque, tandis que la spécification des objets sur lesquels portent les transactions est à la charge du programmeur [Mai93]. Dans les approches de programmation basées sur la *conversation* et les *blocs de recouvrement* [Ran75], avant d'entrer dans une conversation, chaque processus participant procède à la sauvegarde de son état. Le programmeur intervient dans la construction des blocs de recouvrement et la bibliothèque assure la transparence des sauvegardes et des reprises. Les techniques de réplication utilisent également le mécanisme de sauvegarde.

Cette classe d'algorithmes augmente plus ou moins la complexité de la programmation mais pas avec la même ampleur que pour les techniques explicites.

1.3.3.3 Les approches transparentes à l'utilisateur

Cette catégorie d'algorithmes fournit des bibliothèques de sauvegarde/reprise complètes dans lesquelles, l'implémentation et la gestion du mécanisme sont effectuées d'une manière transparente à l'utilisateur. Cette classe comporte les algorithmes *coordonnés*, les algorithmes *autonomes* et les techniques *hybrides*.

1.3.3.3.1 Les algorithmes coordonnés : les algorithmes adoptant cette approche sont généralement inspirés des travaux de Chandy et Lamport [CL85]. Ces algorithmes appelés également *synchrones* ou encore *cohérents* (*consistent checkpointing*) appartiennent à la catégorie d'algorithmes dans lesquels, l'état global cohérent (*la ligne de recouvrement*) est défini au moment de la sauvegarde avant de procéder à l'écriture des fichiers d'état sur le support de stockage 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 [Pla93].

Afin de faire le point sur cette classe d'algorithmes dont l'utilisation est répandue, nous commençons par les méthodes de définition d'état global cohérent.

Définition d'état global cohérent : dans [Lam78], Lamport a défini la relation d'ordre causal *précède (happens-before)* $\rightarrow$ dans un système distribué. Dans cette relation, les événements internes à un même processus ainsi que l'envoi et la réception de messages sont ordonnés. La transitivité et l'antisymétrie permettent de définir un ordre partiel des événements.

Un processus d'une application distribuée évolue en passant d'un état vers un autre. Ces états sont générés par des événements internes (évolution des calculs) ou par des événements correspondants à l'envoi et à la réception des messages.

Coupes et coupes cohérentes : une *"coupe" (cut)* permet de définir l'état global d'un programme parallèle pour un algorithme de sauvegarde/reprise. La figure 1.4 met en évidence l'exécution d'une application parallèle composée de quatre processus s'exécutant sur les processeurs P_0, P_1, P_2 et P_3. Les coupes sont dénotées par les lignes pointillées. C_0, C_1 et C_2 représentent des exemples de coupes.

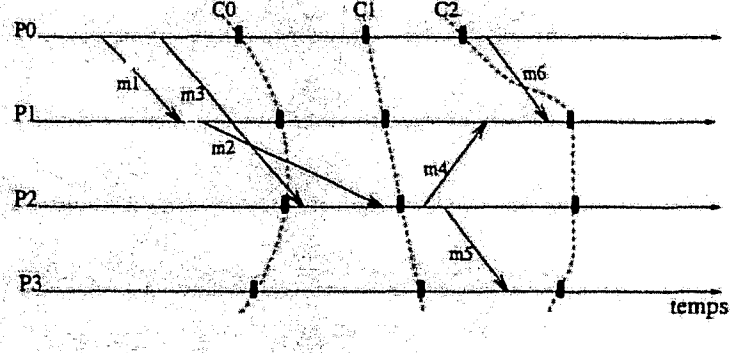


FIG. 1.4 - *Coupes et coupes cohérentes.*

Un message est dit croisant la *coupe* de gauche à droite si son point d'envoi est à gauche de la coupe et son point de réception est à droite de celle-ci (m_2 et m_3 par rapport à C_0). En d'autres termes, l'événement correspondant à son envoi est enregistré dans l'état du processus émetteur et celui correspondant à sa réception

ne l'est pas dans l'état du processus receveur. Le croisement de droite à gauche est la situation inverse (m_6 par rapport à C_2). Une coupe est dite *cohérente* si aucun message de l'application ne la croise de droite à gauche [Pla93]. Les coupes C_0 et C_1 sont cohérentes, tandis que C_2 ne l'est pas.

L'état global cohérent d'une application distribuée est construit à partir des coupes cohérentes [CL85, HMR93]. Le problème des coupes incohérentes provient des messages les croisant de droite à gauche. Observons la figure 1.4, le recouvrement de l'application à partir de l'état global défini par la coupe C_2 remet l'application dans un état où P_0 n'a pas encore émis le message m_6, tandis que P_1 l'a déjà reçu. Cependant, P_0 va réenvoyer le message m_6, ce qui provoquera une incohérence.

Un état global construit à partir de la coupe cohérente C_0 permet un recouvrement cohérent de l'application. Cependant, les messages m_2 et m_3 seront perdus après le recouvrement. En effet, P_0 et P_1 ont enregistré l'émission de m_3 et de m_2 respectivement à P_2, tandis que ce dernier n'a pas enregistré leur réception. Cela ne remet pas en cause la cohérence de l'état global à condition de prévoir un mécanisme qui permet de réenvoyer ces messages à la reprise (journaux, retransmission, etc). Ces messages sont qualifiés de *messages en-transit* par rapport à l'opération de sauvegarde globale.

Il en découle deux propriétés principales que doit vérifier une coupe afin de pouvoir être cohérente [HMR93] :

P1: un message m envoyé par un processus P_i avant la sauvegarde de son état local doit être reçu par le processus destinataire P_j avant sa sauvegarde locale ou doit appartenir à l'ensemble des messages à réenvoyer au moment du recouvrement ;

P2: si un message m est reçu par P_j avant sa sauvegarde locale, m doit être envoyé par P_i avant sa sauvegarde locale également.

Dans ce qui suit, nous présentons quelques algorithmes (coordonnés) proposant des solutions à ce problème. Ces algorithmes diffèrent essentiellement sur les hypothèses faites sur les nœuds et sur la communication.

Algorithmes pour la communication FIFO : dans leur article [CL85], Chandy et Lamport ont traité le problème de définition d'état global cohérent des applications distribuées et ont proposé un algorithme (CL) utilisant des messages de contrôle (*marqueurs*). Les canaux de communication sont supposés fiables et FIFO.

La sauvegarde est effectuée en deux phases. Durant la première, les messages en transit sont calculés et une sauvegarde locale temporaire est réalisée par chaque processus. La validation de ces sauvegardes locales afin de construire une sauvegarde globale cohérente est effectuée durant la seconde phase.

La définition des messages en transit est réalisée comme suit (un coordinateur P_c diffuse le message de sauvegarde m_{CL} sur tous ses canaux sortants) :

- si P_i reçoit le message m_{CL} avant d'avoir effectué sa sauvegarde locale, il diffuse

m_{CL} à tous les processus voisins et procède immédiatement à la construction de sa sauvegarde locale avant d'échanger d'autres messages ;

- si P_i reçoit un message ordinaire m sur le canal c tandis qu'il n'a pas encore reçu le message m_{CL} sur ce canal, il déduit que le message m est un message croisant la coupe. Par conséquent, le message m doit être ajouté au *journal* ;

- lorsque tous les processus ont reçu le message m_{CL} sur tous les canaux entrants, alors, il ne reste plus de messages croisant la coupe et la sauvegarde locale est effectuée. Chaque processus informe le coordinateur P_c en lui envoyant le message de fin de sauvegarde locale m_S.

La validation est conditionnée par l'obtention de l'unanimité des réponses favorables. P_c diffuse alors la décision de validation afin de rendre permanentes toutes les sauvegardes locales. L'algorithme CL possède une complexité en $O(n^2)$ en terme de messages de contrôle échangés si chaque paire de nœuds est reliée par un lien de communication (cas des réseaux de stations ; n étant le nombre de processus participant à la sauvegarde).

Dans [Pla93], un algorithme appelé *Network Sweeping* a été proposé pour les réseaux de communication point-à-point afin de réduire le nombre de messages de contrôle utilisés par l'algorithme CL. L'algorithme est basé sur une période dite de *"sweeping"* (définie par des échanges de messages de contrôle), durant laquelle, un processus peut recevoir des messages en transit. Après le *"sweeping"*, il peut échanger des messages ordinaires sans que ces derniers ne soient journalisés. Sa complexité en terme de messages de contrôle peut atteindre celle de l'algorithme CL si le réseau de connexion constitue un graphe complet.

Algorithmes pour la communication ne préservant pas l'ordre : l'utilisation de marqueurs par les algorithmes précédents dans le cas des canaux FIFO permet de déterminer l'ensemble des messages en transit durant l'opération de sauvegarde globale. Ces messages n'ont aucun effet sur l'ordre de réception des messages ordinaires (canaux FIFO). Lorsque la communication ne respecte pas l'ordre d'émission (canaux non FIFO), l'ordre de réception peut être altéré par les messages de contrôle [HMR93]. Pour remédier à ce problème, des approches, excluant les messages de contrôle ou utilisant des messages de contrôle qui n'affectent pas l'ordre de réception des messages, ont été développées pour ces environnements.

Dans l'algorithme de Lai et Yang [LY87], au lieu d'utiliser des messages de contrôle, les messages ordinaires sont estampillés par une couleur reflétant la période de leur envoi (avant ou après la sauvegarde locale de l'émetteur). Les messages verts sont journalisés au niveau de l'émetteur et du récepteur à la fois. Les sauvegardes locales et les journaux sont envoyés à un coordinateur qui se charge de calculer la sauvegarde globale cohérente. Cette méthode, bien qu'elle n'utilise pas de messages de contrôle, induit une quantité souvent importante de messages stockés et un surcoût dû à la vérification de la couleur de tous les messages reçus.

Pour réduire la quantité importante de messages stockés, Mattern a proposé un algorithme avec moins de messages stockés mais plus lent [Mat88]. Pour cela, des compteurs sont utilisés au lieu de stocker tous les messages verts reçus.

Ahuja [Ahu90] utilise des messages de contrôle de type *ct-passé* et *ct-futur*[7] qui n'altèrent pas l'ordre de réception des messages et permettent d'obtenir des états globaux cohérents. L'algorithme proposé utilise un nombre important de messages de contrôle et nécessite l'estampillage de tous les messages échangés.

Algorithmes pour la communication préservant l'ordre causal : dans ce type d'environnement, l'ordre entre les événements est garanti. Cela permet de réduire la complexité des algorithmes de sauvegarde au prix des protocoles de communication plus complexes (préservant l'ordre causal). Dans [HMR93], deux étapes simples (commune à la plupart des algorithmes coordonnés dans ces environnements) sont spécifiées, pour la réalisation d'une sauvegarde globale cohérente :

- un processus coordinateur diffuse un message m_{local} à tous les autres (y compris lui-même) afin qu'ils procèdent à la réalisation de leur sauvegarde locale ;
- chaque processus P_i effectue sa sauvegarde dès la réception de m_{local}.

Grâce à l'ordre causal, ces deux étapes assurent la cohérence de l'état global car un message m émis par P_i après sa sauvegarde locale ne peut être reçu par son destinataire P_j avant m_{local}. Pour la détermination des messages en transit, des protocoles simplifiés ont été proposés [AV93].

1.3.3.3.2 Les algorithmes autonomes : les techniques autonomes évitent la phase de synchronisation en procédant à la sauvegarde de chaque processus de l'application distribuée indépendemment des autres processus. Afin d'assurer la cohérence des sauvegardes et de prendre en charge les messages en-transit, chaque processus dispose d'un *journal (log)* en mémoire stable ou volatile dans lequel, il enregistre tout ou une partie des messages échangés et leur histoire. Lorsqu'une défaillance surgit, l'algorithme de recouvrement utilise les sauvegardes locales et les journaux afin de calculer un état cohérent de l'application. Par la suite, chaque processus repris est restauré à partir de son état local et tous les messages du journal lui sont réenvoyés dans l'ordre d'origine de leur réception [EZ94, AM98, SY85, ENE92a, SBY88, SW89, WHV+95].

Le problème de l'incohérence d'état de l'application suite à un recouvrement provient des processus *orphelins* i.e. processus survivant dont l'état est incohérent avec les états des processus recouverts [AM98]. Un processus orphelin est créé après le recouvrement d'un processus ayant perdu une partie de son historique de messages. Par exemple, si le processus P_1 retire un message m_1 avant d'envoyer un message m_2

7. Un message *ct-passé* est un message qui ne peut doubler aucun message. De même, un message *ct-futur* est un message qui ne peut être doublé par aucun message.

à P_2 qui le retire à son tour. Si, avant de journaliser m_1, P_1 défaille, le recouvrement de P_1 n'enregistre pas le retrait de m_1 et par conséquent le non-envoi de m_2 à P_2 qui lui, a reçu et a retiré m_2. Par conséquent, l'état global de l'application devient incohérent comme dans le cas des algorithmes coordonnés (figure 1.4).

Les algorithmes autonomes doivent éviter la création de telles situations au moment du recouvrement. Alvisi [AM98] démontre que seulement la journalisation des déterminants des messages est nécessaire *(émetteur, ordre d'émission, destinataire, ordre de retrait)* et caractérise la propriété de ne pas avoir d'orphelins. Chaque processus dépendant d'un message m (retirant un message m' dépendant causalement de m) doit journaliser m. En d'autres termes, pour toutes les exécutions de l'application et $\forall m : (Depend(m) \subseteq Log(m))$ (propriété de sûreté [AM98]). Dans ce cas, m ne peut être perdu suite à la défaillance d'un ensemble de processus et est qualifié de *stable*. Il devient stable également s'il est écrit en mémoire stable. Les méthodes autonomes sont classées en trois sous-classes: pessimistes, optimistes et causales [ENE92a, AM98] (distribuées [Con96]).

Dans les techniques *pessimistes* [PP83], aucun processus P ne peut envoyer un message m avant que tous les messages qu'il a retiré avant m ne soient journalisés (stables) [AM98]. Les techniques pessimistes ne créent jamais de processus orphelins mais bloquent le processus avant chaque retrait de message en contrepartie.

Les techniques *optimistes* relachent cette condition et adoptent une propriété plus faible dans un souci de préservation de performances, en permettant d'envoyer des messages avant la stabilisation de m. Si une défaillance survient et que des orphelins sont créés après le recouvrement, ces derniers sont repris dans un état précédent et ainsi de suite jusqu'à ce qu'ils ne deviennent plus orphelins (phénomène d'*effet domino*). Les algorithmes de cette classe nécessitent la validation des messages à destination du monde extérieur et effectuent la journalisation à la réception [Con96].

Les techniques *distribuées* causales empêchent la création d'orphelins tout en évitant l'inhibition dûe à la stabilisation systématique des messages reçus. Cela est effectué en contraignant un minimum de processus à journaliser m, soit l'ensemble des processus dépendant de m $(Depend(m) = Log(m))$[8]. Cela signifie que m doit être ajouté à $Log(m)$ et $Depend(m)$ simultanément, mais cela nécessite des algorithmes complexes [AM98].

Les algorithmes autonomes souffrent du surcoût induit par la journalisation des messages, de la complexité des protocoles de recouvrement, du problème d'indéterminisme d'exécution et d'effet domino. En contrepartie, ils réduisent la latence durant les interactions avec le monde extérieur. L'*indéterminisme d'exécution* signifie qu'à partir d'un même état initial, plusieurs exécutions différentes de l'application sont possibles. Les actions non-déterministes résultent généralement de l'interaction avec le monde externe au système, tel que l'horloge. L'*effet domino*, en revanche, est le phénomène qui provoque une succession d'opérations de recouvrement arrière,

8. La dépendance est causale et la communication est supposée respecter l'ordre causal.

dans le but de trouver un état global cohérent de l'application distribuée, suite à une opération de recouvrement. L'effet domino peut aller jusqu'à provoquer la reprise de l'exécution de l'application dès le début.

1.3.3.3.3 Les techniques hybrides (adaptatives) : les techniques hybrides tentent de combiner les avantages des deux précédentes catégories d'approches dans le but d'obtenir de meilleures performances. Ces approches sont basées sur la sauvegarde globale sans blocage de l'application. Cela est rendu possible grâce au marquage et à la journalisation de quelques messages applicatifs [DVCL93].

Le principe des techniques hybrides est d'étendre un ensemble de sauvegardes locales S d'un sous-ensemble de processus de l'application à une sauvegarde globale cohérente. Deux types de sauvegardes sont considérés : sauvegardes *programmées (scheduled)*, déterminées par la politique de gestion du mécanisme, et additionnelles *forcées (unscheduled)*, imposées par l'algorithme de sauvegarde afin de maintenir un état global conérent et d'empêcher l'apparition de sauvegardes inutiles. Ce qui permet de s'affranchir ou de minimiser les effets du problème d'effet domino connu par les techniques indépendantes [XN93].

Netzer et Xu [NX95] ont établi les conditions nécessaires et suffisantes pour qu'un ensemble de sauvegardes locales puisse être combiné avec celles d'autres processus afin de construire une sauvegarde globale cohérente. Cette démonstration passe par la définition des chemins zigzagants (z-chemins) qui n'est qu'une formalisation des conditions de cohérence d'états globaux des algorithmes coordonnés.

Un z-chemin est une généralisation des chemins causaux (chaîne de messages commençant après P_p et se terminant avant P_q) dans le sens où il peut exister dans la chaîne formant le z-chemin, un message m_{k+1} émis avant la réception de son prédécesseur m_k dans la chaîne, à condition que cela soit dans le même intervalle de sauvegarde (m_6, m_7 sur la figure 1.5) [XN93, NX95]. Par conséquent, un z cycle contenant une sauvegarde C peut exister $C \rightsquigarrow C$ (le z-chemin part de C et se termine à C) contrairement à l'ordre partiel induit par les chemins causaux.

Manivannan explore la manière dont un ensemble de sauvegardes locales S peut être étendu à une sauvegarde globale et propose un algorithme qui énumère toutes les sauvegardes globales comportant S [MNS97].

Dans [XN93], Xu et Netzer ont proposé un algorithme basé sur ces constats dont l'objectif est de réduire la propagation des reprises (effet domino), tout en évitant la phase de synchronisation induite par les algorithmes coordonnés [XN93]. L'idée est de suivre à la trace (traquer) les chemins zigzagants durant l'exécution de l'application distribuée et de les casser en procédant à une sauvegarde non-programmée. C'est dans ce sens que l'approche est qualifiée d'adaptative. L'approche consiste à détecter uniquement les chemins causaux, ce qui est plutôt plus facile. La détection des z-cycles en utilisant les chemins causaux consiste à vérifier, à chaque réception de message, si celui-ci complète un z-cycle (figure 1.5).

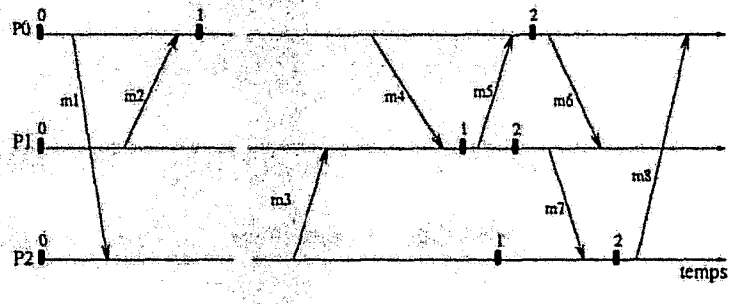


FIG. 1.5 - *Chemin zigzagant formé à la réception tardive de m_3 et non pas de m_2.*

L'algorithme emploie un vecteur de dépendances au niveau de chaque processus. Ce vecteur est ajouté en estampille à chaque message émis.

De leur côté, Baldoni et al. [BHMR95] ont proposé un algorithme amélioré, dans lequel les sauvegardes inutiles sont complètement éliminées. Cet algorithme s'appuie sur la définition d'une relation de précédence entre les intervalles de sauvegarde similaire à celle de Xu et Netzer. Cela est effectué grâce au repérage de tous les z-chemins (causaux et non causaux).

Pour traquer les chemins causaux, une approche similaire à celle de Xu est utilisée. Deux vecteurs sont utilisés à cet effet et estampillent chaque message émis. Pour repérer les chemins zigzagants non causaux, chaque processus maintient un vecteur et une matrice qu'il adjoint en estampille à chaque message émis également.

Les deux algorithmes (en particulier le dernier) sont coûteux en terme de quantité de données estampillées, de vérifications effectuées à chaque réception de message et en sauvegardes additionnelles forcées. Cela est d'autant plus vrai pour les applications sollicitant fortement la communication.

1.4 Systèmes tolérants aux fautes

Les systèmes implémentant la tolérance matérielle aux fautes (la plupart commerciaux) sont généralement conçus pour tolérer une seule faute matérielle à la fois. Pour cela, chaque unité du système est dupliquée de telle sorte qu'il existe toujours un chemin en double accès vers chaque unité. La réplication logicielle des applications est souvent couplée avec la réplication matérielle. Parmi les systèmes adoptant cette approche: NonStop de Tandem [BGH87, ADP94], Integrity S2 de Tandem [Jew91] et Stratus S/32 d'IBM.

Les systèmes basés sur les groupes de processus et la réplication utilisent généralement le modèle *virtuellement synchrone* dont Isis fût le premier [BJ87, Bir93, Bir96], Transis [ADKM92], Highly Available System (HAS) et Advanced Automation System (AAS) d'IBM [Cri91], Delta-4 développé dans le cadre du projet européen ESPRIT [Pow91], Harp implémentant des serveurs de fichiers répliqués [LLSG90],

Rampart [Rei96] dans lequel, les défaillances sont supposées arbitraires (byzantines). Certains systèmes gèrent les partitions multiples également : Horus [RHB94], Relacs [BDM95, BBD96], NavTech [RV95] et Phoenix[MFSW95].

Parmi les systèmes basés sur la sauvegarde/reprise, on trouve: Manetho utilisant la réplication et la sauvergarde/reprise [ENE92b, ENE92a], GatoStar [Sen94], Fail-safe PVM [LFS93], CoCheck [Ste95], MIST [CCG+95] et l'algorithme de Conan [Con96]. D'autres implémentent le modèle des transactions: Argus [LCJS87], Arjuna, Bayou [ea95], etc.

1.5 Conclusion

La sûreté de fonctionnement d'un système se mesure par sa capacité à fonctionner en dépit des fautes. La sûreté de fonctionnement doit être considérée suivant ses trois dimensions: ses attributs, ses entraves et ses moyens. Parmi les moyens de la sûreté de fonctionnement, la tolérance aux fautes est le moyen le plus célèbre et le plus utilisé. Les approches de recouvrement arrière par reprise utilisant les techniques de points de reprise (sauvegardes/reprise) sont très adoptées pour assurer la tolérance aux fautes des applications commerciales à usage général. Les points forts de ces approches sont en particulier la *transparence* et la *souplesse* (le mécanisme de tolérance aux fautes transparent peut être inhibé), le surcoût matériel peu élevé contrairement aux techniques basées sur la redondance, et l'insensibilité aux fautes communes causées par des phénomènes transitoires (une interruption temporaire des canaux de communication dans un réseau peut causer l'arrêt de toutes les répliques).

En revanche, le surcoût temporel lié aux techniques de recouvrement est généralement plus important que celui induit par les techniques de compensation. Ceci est le résultat de l'influence favorable de la *redondance structurelle*[9] [Lap94b]. Par conséquent, ce paramètre peut déterminer l'approche de tolérance aux fautes qu'un système doit adopter. Les systèmes critiques optent souvent pour les techniques de réplication, vu la gravité des défaillances.

Contrairement aux techniques semi-automatiques et explicites, les techniques transparentes réduisent la complexité de programmation au prix de l'augmentation du coût de sauvegardes (temps et espace de stockage). Les algorithmes coordonnés induisent une phase de synchronisation généralement coûteuse et utilisent une quantité importante de messages de contrôle. Les techniques autonomes souffrent des problèmes d'effet domino, d'indéterminisme d'exécution et de la complexité des protocoles de recouvrement. Les techniques hybrides augmentent la latence à cause de la comptabilisation des chemins causaux et zigzagants et échangent d'importantes quantités de données estampillées sur les messages applicatifs. Ces techniques peuvent souffrir du problème d'effet domino également.

9. Il existe une autre approche de redondance dite *temporelle*, qui consiste à répéter l'exécution du service dans le temps

Sur le plan qualitatif, tous les algorithmes de la classe transparente souffrent des problèmes de l'hétérogénéité matérielle et de la redistribution de la charge au moment du recouvrement. Ces problèmes contribuent à la réduction des possibilités de reprise des applications dans le cas d'insuffisance de ressources.

Le chapitre suivant sera consacré à l'étude et à la présentation d'un algorithme de sauvegarde/reprise coordonné pour les applications dites *parallèles adaptatives*. L'algorithme est caractérisé par une moindre complexité, de meilleures performances et des propriétés lui permettant de s'affranchir des problèmes de l'hétérogénéité et de la redistribution de la charge.

Chapitre 2

Sauvegarde/reprise dans les systèmes parallèles adaptatifs

La nature des stations de travail (qualité des composants matériels, aspect multi-utilisateur, comportement humain, etc) augmente considérablement le risque de défaillances des réseaux de stations. Ce risque est encore accentué par les défaillances du réseau de communication lui-même. Si la probabilité de défaillance d'un nœud est de p, alors la probabilité de défaillance d'un réseau de stations composé de n stations[1], est au moins de $1-(1-p)^n$ en considérant les défaillances du réseau de communication. En conséquences, les performances du réseau de stations peuvent être sensiblement affectées, causant la dégradation de la qualité de service et provoquant des pertes importantes en temps de calcul des applications. Une étude statistique, réalisée dans notre laboratoire sur un parc de 42 stations de travail sur une période d'un mois, a révélé un taux de défaillances important. En moyenne, une défaillance est enregistrée toutes les 2.78 *heures*. Ces défaillances sont souvent d'origine logicielle et affectent le nœud entièrement (redémarrage de machines, arrêts de serveurs, etc). Les pertes en temps de calcul peuvent être considérables, en particulier, pour les applications de longue durée de vie (optimisation combinatoire, calcul scientifique, etc).

Le mécanisme de sauvegarde/reprise est une solution logicielle très répandue pour la tolérance aux fautes logicielles (voir chapitre 1). L'approche consiste à effectuer périodiquement des sauvegardes de l'état de l'application sur un support de stockage stable. Dès l'apparition d'une défaillance, l'application est reprise à partir de son dernier état enregistré préalablement en mémoire stable (figure 1.2, chapitre 1). L'application continue ainsi son exécution avec un minimum de pertes. Le choix de ce mécanisme est motivé par trois facteurs :

- sa *transparence :* le programmeur n'a pas besoin de connaître l'existence du mécanisme;

1. Nous entendons par la probabilité de défaillance du réseau la défaillance d'au moins un des nœuds le composant.

- son *efficacité :* il prend en charge totalement les défaillances matérielles et logicielles ;

- son *faible coût :* il ne nécessite pas de matériel spécifique. De plus son surcoût d'exécution est relativement faible comparé à d'autres méthodes, en particulier, s'il est optimisé.

Dans le chapitre précédent, nous avons survolé les différentes approches et techniques de tolérance aux fautes dans les systèmes parallèles et distribués. Dans ce chapitre, une étude de ce problème pour une classe particulière de ces systèmes (*les systèmes parallèles adaptatifs*) est présentée. Nous présentons un nouvel algorithme de moindre complexité ayant de meilleures propriétés que les algorithmes connus. Ces améliorations sont liées, en particulier, aux caractéristiques particulières de la classe d'applications considérée.

Les améliorations obtenues concernent principalement la complexité en nombre de messages de contrôle échangés au moment de la sauvegarde et la taille des fichiers de sauvegarde résultants. De plus, l'algorithme possède des propriétés importantes sur le plan qualitatif concernant en particulier le problème de l'hétérogénéité matérielle et de la redistribution de la charge.

Dans la section suivante, nous présentons l'environnement de programmation parallèle adaptative MARS. La section 2.2 présente l'algorithme de sauvegarde/reprise. Les résultats des expérimentations conduites sur un ensemble d'applications sont alors présentés et analysés.

2.1 MARS : Un environnement de programmation parallèle adaptative

L'algorithme de sauvegarde/reprise est développé dans le cadre du système d'ordonnancement adaptatif de l'environnement multi-utilisateur et hétérogène MARS [Haf98].

MARS est mis en œuvre selon une structure multi-couche de bibliothèques (communication, multi-threading distribué, ordonnancement). La figure 2.1 illustre cette structure avec les différentes bibliothèques utilisées.

Par la suite, nous présentons brièvement chaque couche, ses caractéristiques et son apport. Puis, nous présentons le modèle d'application parallèle adaptative et identifions ses propriétés particulières. Ceci a permis la conception et la réalisation d'un algorithme efficace de sauvegarde/reprise.

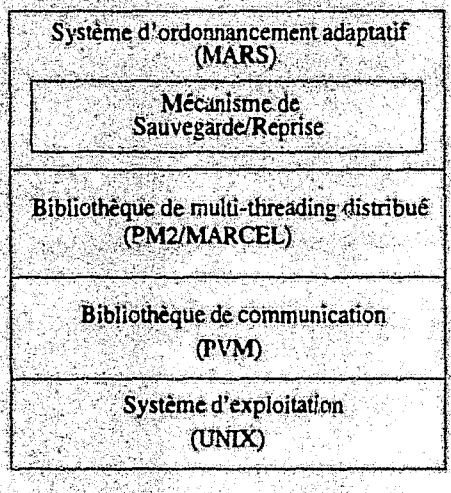


FIG. 2.1 - *Structure multi-couche du système MARS.*

2.1.1 L'environnement matériel

Nous nous plaçons essentiellement dans les réseaux de stations de travail. Un réseau de stations se compose d'un parc important de stations de travail hétérogènes reliées entre elles par un réseau de communication. Chaque nœud est une machine hôte disposant de son propre système d'exploitation et de ses propres unités d'entrées/sorties. Les propriétés suivantes caractérisent ces environnements :

- *disponibilité importante :* globalement, les temps de disponibilité sont considérables. Des études montrent que le pourcentage de temps durant lequel, les stations sont sous-utilisées dans un tel parc est très important. Les concepteurs de Sprite estiment que ce taux varie de 66% à 78% [OCD+88]. Dans Condor, les auteurs reportent 70% à 80% de temps perdu durant le week-end et 50% pendant les jours de travail [LLM88]. V-System estime ce temps à 80% [TLC85]. De son côté, Piranha estime la disponibilité à 82% pendant les jours de travail et 96% durant le week-end [Kam94]. Enfin, MARS reporte environ 85% de temps perdu [Haf98]. Cette disponibilité est estimée suivant des critères de disponibilité propres à chaque système et est affectée par la période de scrutation également. A titre d'exemple, des systèmes tels que Condor ou MARS considèrent que le nœud est non-disponible dès la présence de son propriétaire. En revanche, V-System le considère disponible tant qu'il n'est pas surchargé même en présence du propriétaire ;

- *multi-programmation :* les stations exécutent généralement une version du système Unix, ce qui leur permet d'exécuter plusieurs processus en temps partagé ;

- *nœuds faiblement couplés :* les temps de communication sont généralement non négligeables et le risque de défaillances est important ;

- *environnement multi-utilisateur :* les nœuds, en l'occurence les stations de travail, sont accessibles par différents utilisateurs à distance ou directement sur la console.

Deux problèmes majeurs se posent : la charge des nœuds est irrégulière et dépend du comportement des utilisateurs et la notion de propriétaire doit être prise en compte. Le propriétaire d'une station doit être privilégié sur sa station. Par conséquent, il doit pouvoir disposer de toute la puissance de sa station dès sa présence et ceci, en contraignant les utilisateurs externes à retirer la charge qu'ils avaient placée sur le nœud en question.

2.1.2 La couche de communication

La couche de communication utilisée est l'environnement PVM[2]. PVM est un environnement de programmation parallèle basé sur l'échange de messages [GBD+94]. L'utilisation de cet environnement est répandue pour deux grandes raisons : la première est la simplicité de son installation et de son utilisation. La seconde raison est *le support d'hétérogénéité* qu'il offre à travers l'intégration de différentes plateformes hétérogènes (processeur, système, réseaux).

La notion principale dans PVM est la *machine virtuelle* : ensemble des nœuds de calcul mis à la disposition d'un utilisateur. Sur chaque nœud de la machine virtuelle réside un démon appelé *pvmd*. Celui-ci se charge de la gestion de la machine virtuelle et des échanges de messages entre les processus des applications. Ce mécanisme est complété par une bibliothèque de primitives permettant la gestion de la communication (communication point-à-point, multi-cast, diffusion, etc), la gestion des processus (création, groupe de processus, etc) et la gestion de la machine virtuelle (ajout de nœuds, retrait de nœuds, etc) [GBD+94]. Chaque utilisateur dispose de sa propre machine virtuelle qui prend en charge toutes ses applications. Par conséquent, les machines virtuelles des différents utilisateurs peuvent se chevaucher sur une plateforme multi-utilisateur et multi-programmée.

La gestion des processus dans PVM manque d'efficacité. Le placement d'un processus sur un nœud est effectué sans tenir compte de l'état de l'environnement (charge des nœuds, etc), en utilisant un placement cyclique aveugle. Le chevauchement des différentes machines virtuelles accentue le problème, car il constitue une source de concurrence entre les utilisateurs sur les mêmes ressources. PVM ne prévoit aucun moyen permettant de faire face à ce problème en gérant globalement l'ensemble des utilisateurs.

2. Parallel Virtual Machine.

2.1.3 La couche de multi-threading distribué

L'utilisation du multi-threading a bien démontré son intérêt [Nam96]. D'une part, les performances peuvent être améliorées en désallouant les threads bloqués en attente d'entrées/sorties ou de communication (recouvrement calcul/communication). D'autre part, cela permet un découpage fonctionnel des tâches en confiant chaque fonction à un thread (thread responsable de la communication, de la collecte d'information de charge, du calcul, etc). C'est dans cet esprit que le système MARS a opté pour un support de multi-threading distribué, en l'occurence PM2[3].

Un processus léger (thread) est un flot d'exécution (séquentiel) [Nam96]. Les threads s'exécutent au sein du processus lourd en temps partagé et partagent également son espace de données globales.

Le concept de base dans PM2 est le LRPC[4] qui consiste à exécuter un service par un processus distant se traduisant par la création d'un thread. Un LRPC est l'extension de la notion d'appel procédural à distance dans un environnement distribué. Trois types de LRPCs sont définis : *LRPC synchrone*, *LRPC asynchrone* et *LRPC à attente différée*.

2.1.4 Le système d'ordonnancement adaptatif

Les environnements dits à parallélisme adaptatif sont des systèmes où l'application parallèle est contrainte à ajuster son degré de parallélisme d'une manière imprévisible suivant l'évolution de l'état du système. Ainsi, pour faire face à l'indisponibilité de ressources, l'application parallèle est contrainte à réduire son degré de parallélisme afin de s'adapter à la nouvelle situation. De même, lorsque le système enregistre de nouvelles ressources libres, l'application parallèle est invitée à bénéficier de cette disponibilité en augmentant son degré de parallélisme. Les systèmes MARS [Haf98], CARMI [PL96] et Piranha [Kam94] sont des exemples de tels environnements.

MARS est un système d'ordonnancement adaptatif développé dans notre équipe dans le cadre de la thèse de Hafidi [Haf98]. La figure 2.2 illustre l'architecture du système MARS. Celui-ci se compose d'un *serveur de groupe (Group Server, GS)*, qui gère un pool de nœuds hétérogènes. Sur chaque nœud du groupe réside un *gestionnaire de nœud (Node Manager, NM)*. Ce dernier est chargé d'observer les activités du nœud et de transmettre systématiquement les informations d'état du nœud en question au *serveur de groupe*. La disponibilité du nœud est fortement liée aux indicateurs de charge et au comportement de son propriétaire.

Le système MARS réagit à deux événements principaux : lorsqu'un nœud du groupe devient disponible, le système *déplie* l'application en l'autorisant à lancer des processus sur le nœud en question. En revanche, si un nœud exploité par l'application

3. Parallel Multithreaded Machine.
4. Lightweight Remote Procedure Call.

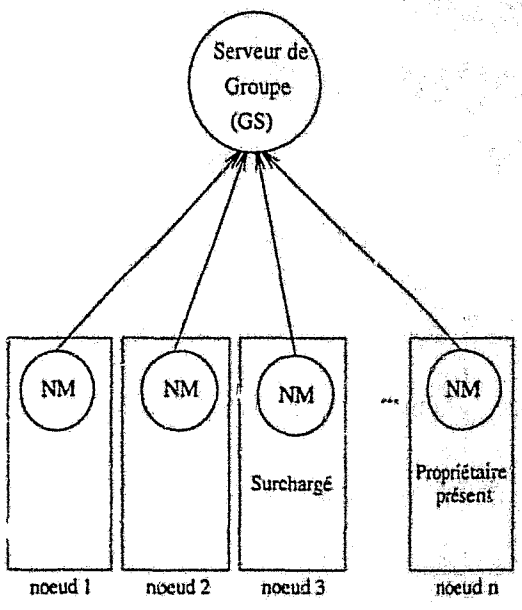


FIG. 2.2 - *Architecture de l'ordonnanceur MARS.*

devient indisponible (surchargé ou réquisitionné par son propriétaire), l'application est invitée à évacuer la charge qu'elle avait sur le nœud.

2.1.5 Application parallèle adaptative

Une application parallèle adaptative est une application dans laquelle le degré de parallélisme varie dynamiquement suivant l'évolution de l'état de l'environnement sous-jacent.

Une application MARS est construite selon un schéma *maître/esclaves*, un esclave par nœud. Le travail est découpé en plusieurs unités. Au niveau du maître, le programmeur spécifie un thread *serveur de travail* responsable de l'allocation des unités de travail aux esclaves et de recevoir les résultats. Le travail d'un esclave consiste à exécuter les étapes suivantes : demande d'une unité de travail, traitement de l'unité et retour de résultats au maître (figure 2.3). Si un nœud supportant un esclave de l'application est réclamé par le système pour une des raisons citées précédemment, l'esclave en question est évacué. Celui-ci retourne ses *résultats partiels et le travail restant* au maître avant de terminer. Cette opération est appelée *repli* de l'application. En revanche, si un nœud devient disponible et que le système décide de l'attribuer à l'application, celle-ci l'exploite en lançant un nouvel esclave sur le nœud fourni. Cette opération est dénommée *dépli* de l'application. Cette méthodologie de développement de l'application parallèle est qualifiée d'adaptative et est illustrée sur la figure 2.3. Les esclaves communiquent avec le maître uniquement aux moments de demande de travail, de retour de résultats ou de résultats partiels. De plus, la communication entre les esclaves n'est pas autorisée. Comme nous allons

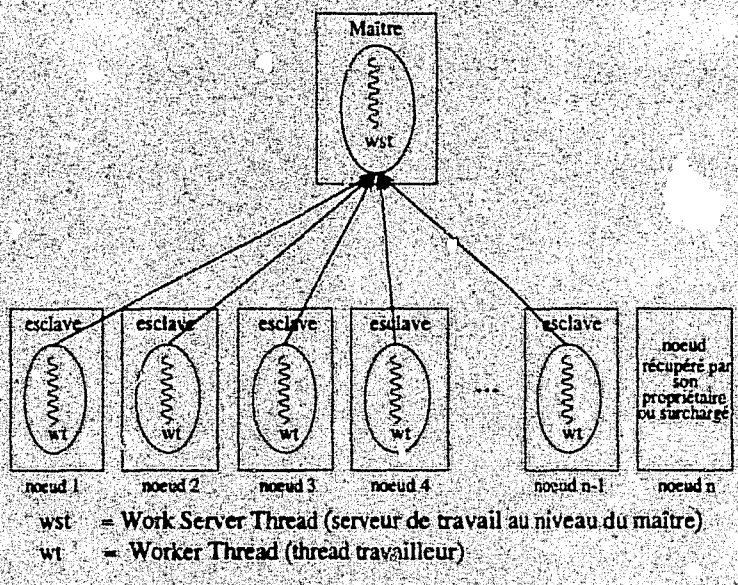


FIG. 2.3 – *Structure d'une application MARS.*

le voir dans la section suivante, cette contrainte permet d'améliorer la complexité de l'algorithme de sauvegarde/reprise et ne réduit pas la classe d'applications prises en charge. En effet, la classe de problèmes traités est grande et importante (voir chapitres 6 et 7).

2.2 Une nouvelle approche de sauvegarde/reprise pour les systèmes parallèles adaptatifs

Dans cette section, nous proposons un algorithme de sauvegarde/reprise dans les environnements adaptatifs. Cet algorithme, conçu sur le modèle des algorithmes coordonnés, est optimisé en considérant les caractéristiques particulières de ce type d'applications.

2.2.1 Sauvegarde d'une application parallèle adaptative

L'algorithme de sauvegarde bénéficie des propriétés des applications MARS pour réduire le nombre de messages de contrôle et la taille du fichier de sauvegarde au moment de l'opération de sauvegarde. L'approche consiste à collecter les résultats partiels de tous les esclaves en utilisant l'outil de *repli* fourni par l'application MARS. Ces unités de travail partiellement traitées sont stockées dans l'espace d'adressage du processus maître. L'état de celui-ci est ensuite sauvegardé. L'algorithme de sau-

vegarde/reprise est décrit ci-dessous (figure 2.4) [KTG97b, KTG97a] :

1. le maître joue le rôle du coordinateur. Lorsqu'une opération de sauvegarde est déclenchée, le coordinateur diffuse un message de contrôle m_{gather} à tous les esclaves afin de collecter les résultats partiels ;
2. à la réception du message m_{gather}, chaque esclave suspend son exécution et lance un thread spécifique qui se charge de remettre les résultats partiels au maître. Le thread de contrôle est synchronisé avec les threads de calcul pour définir un état local cohérent de l'esclave. Cette étape s'achève par l'envoi, par l'esclave, du message $m_{partial}$ au maître. Ce message comporte les résultats partiels de l'unité de travail traitée par l'esclave en question ;
3. après l'envoi de son état local au maître, chaque esclave reprend son exécution sans possibilité de communication avec le maître jusqu'à la réception du message de fin de sauvegarde globale ;
4. après avoir reçu le message $m_{partial}$ de tous les esclaves et avoir rangé les résultats dans son espace de données, le maître procède à sa sauvegarde locale ;
5. à la fin de l'opération, le maître diffuse un message de fin de sauvegarde globale m_{end} à tous les esclaves.

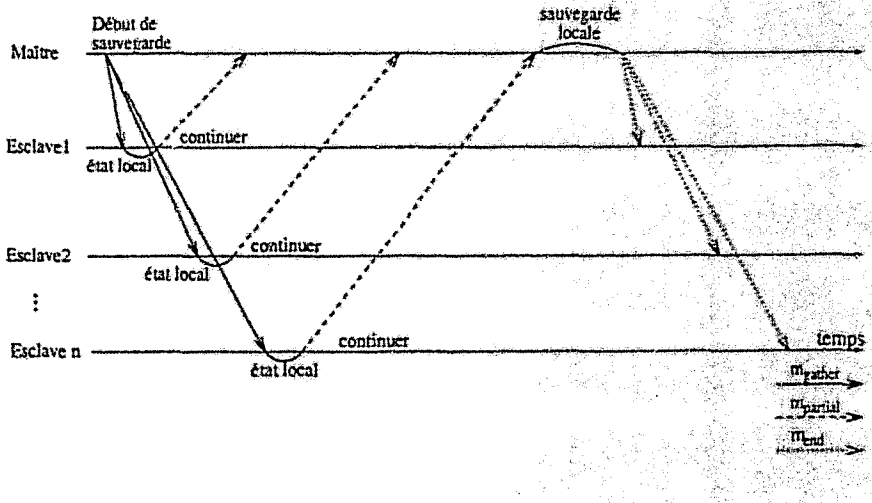


FIG. 2.4 - *L'algorithme de sauvegarde.*

Propriété 1 *L'algorithme de sauvegarde définit des états globaux cohérents de l'application parallèle adaptative.*

Preuve 1 *L'incohérence de l'état global provient des messages croisant la coupe de droite à gauche (voir chapitre 1). m_{ij} est un message croisant la coupe de droite à*

gauche $\Longrightarrow$ m_{ij} est émis par P_i après sa sauvegarde locale et reçu par P_j avant sa sauvegarde locale (définition de message croisant la coupe de droite à gauche).

Dans notre algorithme, si m est un message croisant la coupe de droite à gauche, deux cas sont possibles : Cas 1) m est un message émis par le maître (P_c) après la réalisation de sa sauvegarde locale et reçu par un esclave P_j avant sa sauvegarde locale ($m = m_{cj}$) ; ou Cas 2) m est un message émis par un esclave P_i après sa sauvegarde locale et reçu par le maître P_c avant sa sauvegarde ($m = m_{ic}$). Seuls ces deux cas peuvent se présenter, puisque la communication directe entre les esclaves n'est pas possible. A noter que la sauvegarde locale d'un esclave est définie par l'envoi des résultats partiels au maître (message $m_{partial}$).

Cas 1 : m_{cj} doit être émis par P_c après l'émission de m_{gather} et reçu par P_j avant l'envoi de $m_{partial}$ et par conséquent, avant la réception de m_{gather} (étape 2 de l'algorithme), or m_{gather} est émis par P_c avant m_{cj} $\Longrightarrow$ m_{cj} double m_{gather} $\Longrightarrow$ contradiction avec l'hypothèse des canaux FIFO ;

Cas 2 : comme pour le cas 1, m_{ic} doit être émis par P_i après l'émission de $m_{partial}$ et reçu par P_c avant la réception de $m_{partial}$ d'au moins un esclave (étape 4 de l'algorithme) $\Longrightarrow$ m_{ic} est reçu par P_c avant la diffusion du message m_{end} $\Longrightarrow$ m_{ic} est émis par P_i après l'émission de $m_{partial}$ et avant la réception de m_{end} (étape 5 de l'algorithme), or d'après l'étape 3 de l'algorithme, P_i ne peut émettre de messages au maître durant cette période. CQFD.

Propriété 2 *L'ensemble des messages en-transit lors d'une opération de sauvegarde globale est vide.*

Preuve 2 *Les messages en-transit lors d'une opération de sauvegarde globale sont les messages à re-émettre suite à un recouvrement.*

m est un message en-transit sur le canal $C(i,j)$ $\Longrightarrow$ m est émis par P_i avant sa sauvegarde locale et reçu par P_j après sa sauvegarde locale.

Supposant que m est un message en-transit par rapport à notre algorithme. Seuls deux cas sont possibles (comme pour le problème de cohérence d'état global), puisque la communication entre les esclaves est interdite :

Cas1 : m est émis par P_c avant sa sauvegarde et reçu par P_j après sa sauvegarde ($m = m_{cj}$) $\Longrightarrow$ m_{cj} est émis par P_c avant l'émission de m_{gather} (puisque P_c cesse d'envoyer des messages ne concernant pas la sauvegarde aux esclaves après la diffusion du message m_{gather} et jusqu'à l'achèvement de l'opération globalement) et reçu par P_j après la réception de m_{gather} (m_{cj} est, en réalité, reçu par P_j après l'émission de $m_{partial}$ par ce dernier) $\Longrightarrow$ m_{gather} double m_{cj} $\Longrightarrow$ contradiction avec l'hypothèse des canaux FIFO ;

Cas2 : m est émis par P_i avant sa sauvegarde et reçu par P_c après sa sauvegarde ($m = m_{ic}$) $\Longrightarrow$ m_{ic} est émis par P_i avant l'émission de $m_{partial}$ et reçu par P_c

après la réception de $m_{partial}$ de P_i (P_c effectue sa sauvegarde après la réception de $m_{partial}$ de tous les esclaves, étape 4 de l'algorithme) $\Longrightarrow$ *$m_{partial}$ double m_{ic}* $\Longrightarrow$ *contradiction avec l'hypothèse des canaux FIFO. CQFD.*

Notons que les unités de travail intermédiaires, qui représentent l'état des esclaves, ne comportent pas les événements associés à la communication. Ceci permet de résoudre implicitement le problème des messages en-transit, fondamental dans les algorithmes de sauvegarde/reprise coordonnés (voir sous-section 1.3.3.3.1), et de l'indéterminisme d'exécution (voir sous-section 1.3.3.3.2).

Propriété 3 *L'algorithme de sauvegarde termine même en présence des défaillances concomitantes des esclaves.*

Preuve 3 *Nous utiliserons dans ce paragraphe la notion de "détection de défaillances" qui sera présentée au chapitre 3. En résumé, la détection est l'action d'avoir connaissance ou de supposer la défaillance d'un composant de l'application par un autre composant.*

Défaillance du maître pendant l'opération de sauvegarde (après la diffusion du message m_{gather} à tous les esclaves). Après avoir reçu le message m_{gather}, chaque esclave P_i procède à la définition de son état local qu'il expédie au maître (étape 2 de l'algorithme). La défaillance du maître sera découverte à ce moment, tous les esclaves seront terminés et l'application recouverte à partir de sa dernière sauvegarde permanente (voir chapitre 3). Toutefois, si un esclave défaille simultanément avec le maître (avant de recevoir m_{gather}), ce message sera simplement perdu.

En revanche, la défaillance d'un ou plusieurs esclaves (sans le maître) pendant la sauvegarde est plus complexe. Suivant la période de la défaillance de P_i, trois situations peuvent se présenter :

Cas1 : avant l'émission du message m_{gather} par P_c : dans ce cas, P_i ne recevera pas le message m_{gather} qui sera perdu. Du côté du maître, deux cas sont possibles :

1. *si le maître arrive à détecter la défaillance avant l'émission de m_{gather}, l'application est reconfigurée (P_i est écarté) et la sauvegarde se poursuit avec les $n-1$ esclaves restants,*
2. *si P_c émet m_{gather} avant la détection de la défaillance, il risque d'attendre indéfiniment la réponse de P_i (message $m_{partial}$). La défaillance est découverte pendant l'attente de P_c et l'application est reconfigurée. A la découverte de la défaillance, un module additionnel est également exécuté. Ce module se charge d'annuler toute attente éventuelle de P_c d'un message en provenance de P_i ;*

Cas2 : après l'émission du message m_{gather} par P_c et avant le retour des résultats partiels (envoi du message $m_{partial}$) par P_i : ce cas est similaire à l'étape 2 du cas 1 car P_c risque d'attendre indéfiniment la réception de $m_{partial}$ de P_i, mais la

détection de la défaillance et le module additionnel permettent de reconfigurer l'application et de poursuivre l'exécution de l'algorithme ;

Cas3 : après l'envoi du message $m_{partial}$ par P_i : dans ce cas, P_c n'aura plus à attendre P_i et la détection éventuelle de la défaillance entrainera la reconfiguration de l'application ultérieurement. CQFD.

Ce raisonnement est appliqué également en cas de repli de l'application (concernant un ou plusieurs esclaves) pendant l'opération de sauvegarde. Cette propriété prouve la terminaison de l'algorithme (problème de terminaison des algorithmes distribués en présence des défaillances [GS95, CT96]), ce qui permet à l'application de progresser.

2.2.2 Reprise d'une application parallèle adaptative

La reprise d'une application consiste à reconstruire l'état du maître à partir du fichier de sauvegarde. Le nouveau maître restauré dispose de l'état intermédiaire du travail dans son espace de données et ne possède pas d'esclaves. Il s'enrôle au système comme une nouvelle application. Les unités de travail partiellement traitées seront re-distribuées dynamiquement aux nouveaux esclaves créés après avoir sollicité le système pour l'attribution de nœuds libres. Ainsi, le nombre d'esclaves et leur localisation sont déterminés en fonction de l'état actuel du système.

2.2.3 Mise en œuvre

Les applications MARS utilisent le mécanisme LRPC fourni par PM2 pour les communications inter-processus [Nam96]. Les communications entre les composantes de l'application et les représentants du système (*serveur de groupe, gestionnaires de nœuds*) utilisent le mécanisme RPC[5].

Une opération de sauvegarde commence par une requête (RPC) envoyée par le maître au *serveur de groupe* (GS) dans le but de différer momentanément les opérations d'ordonnancement (repli, dépli, etc) concernant l'application. Cette étape est nécessaire afin d'éviter de mauvaises interprétations des durées de communication par le mécanisme de *détection de défaillances*, qui peut supposer une défaillance dans ce cas (voir chapitre 3). De plus, le maître doit garantir un état local cohérent en évitant d'enregistrer dans le fichier de sauvegarde les informations concernant l'état actuel du système (repli, dépli, demande d'unités de travail par les esclaves), car ces événements sont considérés comme externes à l'application. La collection des résultats partiels (messages m_{gather} et $m_{partial}$) et l'information des esclaves de la terminaison de l'opération (message m_{end}) sont réalisées en utilisant des LRPCs.

5. Développé par Sun Microsystems.

La sauvegarde de l'état local du maître est réalisée en utilisant une bibliothèque de sauvegarde sous Unix [TL95] (voir section 2.2.4). Avant de procéder à sa sauvegarde locale, le maître crée un processus image qui reporte son état sur la mémoire stable parallèlement avec l'exécution de l'application. Cette opération est similaire à la technique d'optimisation *main-memory*, avec une légère différence qui consiste à créer un processus lourd au lieu d'un thread. Ce qui induit un surcoût légèrement supérieur.

La figure 2.5 illustre les détails de l'implémentation de l'algorithme sous MARS. Le dernier message diffusé aux esclaves pour les informer de la fin de l'opération de sauvegarde globale est dû à des raisons techniques imposées par la couche de communication. Dans le cas où les messages échangés ne sont pas enregistrés dans les états locaux des processus (notre cas) et sans ces contraintes techniques, ce message (m_{end}) n'est pas indispensable.

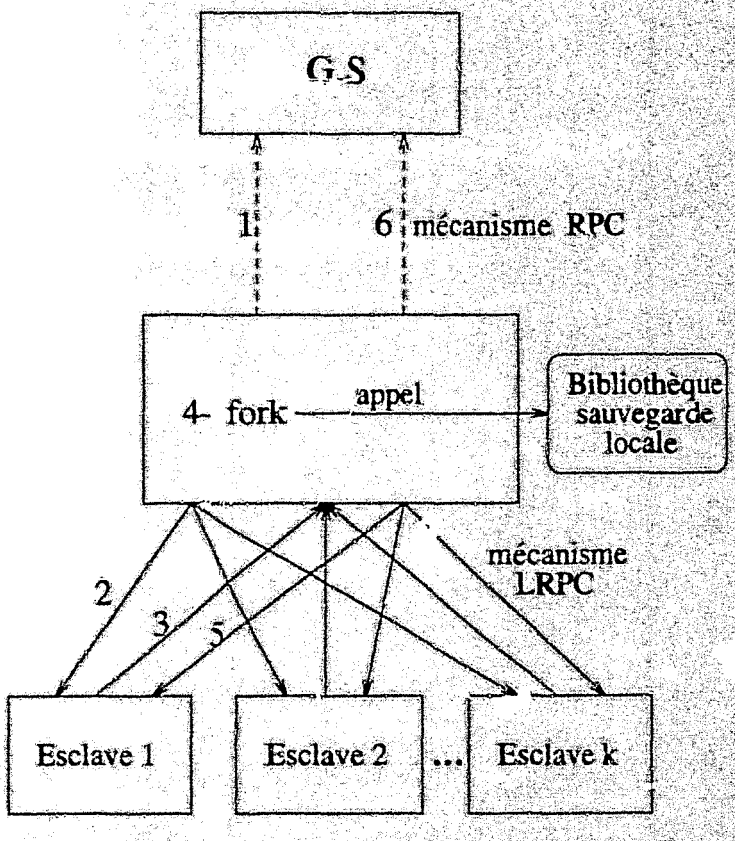


FIG. 2.5 - *Mise en œuvre de l'algorithme de sauvegarde pour MARS. 1)Requête de déconnection du système (RPC). 2)Demande de collecte des résultats partiels m_{gather} (LRPC). 3)Remise des résultats partiels par les esclaves $m_{partial}$ (LRPC). 4)Création du processus image. 5)Message informant les esclaves de la fin de l'opération (LRPC). 6)Ré-enclenchement au système (RPC).*

2.2.4 Sauvegarde/reprise d'un processus séquentiel

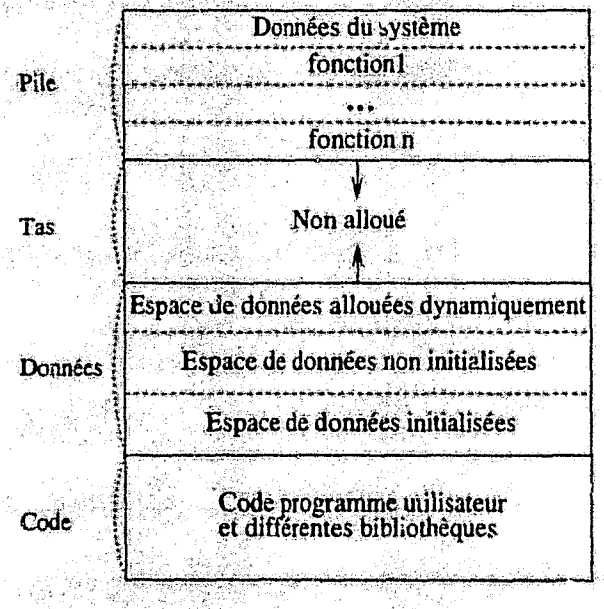


FIG. 2.6 - *Structure d'un processus Unix.*

La figure 2.6 illustre l'organisation typique de l'espace mémoire alloué à un processus Unix (espace d'adressage virtuel). L'espace d'adressage d'un processus Unix peut être décomposé en trois parties : le segment de code, le segment de données et la pile d'exécution. De plus, d'autres informations d'état sont maintenues par le noyau (registres du processeur, état des signaux, état des fichiers ouverts et leurs descripteurs) [TL95, PBkL94].

Lors d'une opération de sauvegarde, le segment de code n'est pas pris en charge, car il est utilisé uniquement en lecture. En revanche, le segment de données (statiques allouées par le compilateur et dynamiques) est sauvegardé à chaque opération de sauvegarde et restitué à la même adresse au moment de la reprise. La pile est la partie de l'espace d'adressage utilisée par le mécanisme d'appel de procédure. Sa taille varie dynamiquement au cours de l'exécution. La pile est sauvegardée en deux parties : le contexte (pointeur de la pile) et les données proprement dites.

Les informations d'état des fichiers ouverts sont captées grâce à l'augmentation des appels système *open*, *dup*, etc. L'état des signaux envoyés au processus (bloqués, pendants) est obtenu par des appels système appropriés. La préservation de l'état de la CPU est fortement dépendante de l'architecture. L'utilisation des signaux pour récupérer l'état du processeur constitue un mécanisme indépendant de la machine pour toutes les plateformes Unix. L'ensemble de ces informations est enregistré dans une table prévue dans l'espace de données du processus.

2.2.5 Complexité et résultats expérimentaux

La plupart des algorithmes de sauvegarde/reprise coordonnés présentés dans la littérature ont une complexité en $O(n^2)$ ou dans le meilleur des cas en $O(n \log n)$[6] en terme des messages de contrôle. De plus, ils considèrent l'espace d'adressage de tous les processus de l'application. Par conséquent, les fichiers de sauvegarde ont souvent des tailles importantes [CL85, Pla93]. Les algorithmes hybrides et indépendants induisent un surcoût important durant l'exécution de l'application et au moment du recouvrement. De plus, ils souffrent de deux problèmes: *l'effet domino* et *l'indéterminisme d'exécution* (voir sous-section 1.3.3.3.2).

Pour notre algorithme, le nombre de messages de contrôle utilisés lors d'une opération de sauvegarde est de: 2 messages RPC pour se déconnecter avant l'opération de sauvegarde et pour se reconnecter à la fin de celle-ci, n messages m_{gather} pour inviter les esclaves à remettre les résultats partiels au maître et le même nombre de messages $m_{partial}$ en réponse à cette requête et finalement n messages pour prévenir les esclaves de la fin de l'opération. En conséquence, le nombre de messages de contrôle ainsi échangés est en $O(n)$.

Notre algorithme de sauvegarde/reprise exploite les propriétés du modèle de programmation adaptative afin de réduire la quantité de données prise en charge dans le fichier de sauvegarde également. En effet, collecter les résultats partiels des esclaves, les stocker dans l'espace de données du processus maître et procéder à la sauvegarde de ce dernier est susceptible de réduire considérablement la taille du fichier de sauvegarde jusqu'à un ordre de n.

En revanche, la collecte de toutes les unités de travail partiellement traitées peut causer la saturation de l'espace de données du maître. Cela est d'autant plus important pour les applications ayant des unités partielles de travail de taille importante. Ce problème peut être résolu en gardant les unités partielles de travail dans l'espace de stockage stable, dont le contenu ne peut être perdu suite à une défaillance, et en procédant à leur sauvegarde d'une manière incrémentale.

Pour évaluer les performances de l'algorithme expérimentalement, nous avons mis en œuvre deux applications:

- une application parallèle pour la recherche de nombres premiers dans un intervalle donné (*prime*);
- une version parallèle de l'algorithme de recherche tabou utilisée pour résoudre des problèmes d'affectation quadratique (*tabu*) [HTG97].

Les détails de conception et de mise en œuvre de ces applications seront présentés dans la troisième partie de ce document consacrée aux applications.

6. n est le nombre de processus de l'application.

Le tableau 2.1 résume les temps d'exécution des deux applications sur un réseau de 13 stations SUN4. La deuxième colonne représente, pour chaque application, le temps d'exécution total exempté du surcoût temporel lié au mécanisme de sauvegarde/reprise. La troisième colonne (*Nombre de sauvegardes*) représente le nombre total d'opérations de sauvegarde réalisées par l'application tout au long de son exécution. La quatrième colonne représente le surcoût temporel total induit par le mécanisme de sauvegarde/reprise. La dernière colonne représente le surcoût du mécanisme par rapport au temps d'exécution total de l'application. La période de sauvegarde est fixée à 5 *minutes* pour ces expérimentations.

Les résultats obtenus montrent qu'avec une période relativement courte, le mécanisme présente un surcoût minimal (inférieur à 1% du temps d'exécution total pour les deux applications).

Application	Sans sauvegarde (sec)	Nombre de sauvegardes	Surcoût de sauvegarde (sec)	%
tabu	18821	63	127	0.67
prime	12434	42	77	0.62

TAB. 2.1 - *Temps d'exécution et surcoût de sauvegarde.*

En réalité, le maître joue un rôle de coordinateur en s'occupant uniquement de l'allocation des unités de travail aux esclaves et de la prise en charge des requêtes du système. Le travail effectif est effectué par les esclaves. Par conséquent, une mesure du surcoût induit par le mécanisme de sauvegarde au niveau de ces derniers constitue un indice plus représentatif de l'efficacité du mécanisme.

Le tableau 2.2 présente les mesures relevées pour les esclaves. Cela permet d'estimer en moyenne le pourcentage de temps durant lequel les calculs sont interrompus par une opération de sauvegarde. En effet, le tableau rapporte pour chaque application dans l'ordre: le nombre moyen d'esclaves participants à une opération de sauvegarde, le surcoût de la sauvegarde la moins longue, le surcoût moyen d'une sauvegarde et le surcoût de la sauvegarde la plus longue. Finalement, des statistiques sont données sur la totalité des mesures comportant l'écart type et le pourcentage moyen du surcoût du mécanisme au niveau des esclaves.

Le nombre moyen d'esclaves par opération de sauvegarde exprime le degré de parallélisme moyen de l'application au moment des sauvegardes. Cela permet d'avoir une idée sur la surcharge éventuelle du maître induite par la réception simultanée des états locaux des esclaves lors de la sauvegarde. Ce nombre est de 11 pour *tabu* et de 12 pour *prime*, ce qui est significatif. Le surcoût de la sauvegarde la moins longue exprime les conditions idéales (charge des nœuds, charge du réseau, etc) durant lesquelles une opération de sauvegarde s'est déroulée[7]. Cela donne 0.46 *sec*

7. La quantité de données retournées par les esclaves n'a pas une grande influence car les résultats

Application	Nbre Moyen esclaves/sauv	Surcoût min d'l sauv (sec)	Surcoût max d'l sauv (sec)	Surcoût moy d'' sauv (sec)	Ecart type	% moyen
tabu	10.96	0.39	1.56	05	0.18	0.32
prime	11.76	0.46	1.77	1.26	0.22	0.40

TAB. 2.2 - *Surcoûts de sauvegarde au niveau des esclaves.*

pour *prime* et 0.39 *sec* pour *tabu*, ce qui est très performant. En revanche, le surcoût de la sauvegarde la plus longue exprime les pires conditions durant lesquelles une opération de sauvegarde a eu lieu. Nous avons enregistré une sauvegarde pour *prime* dans laquelle le surcoût de tous les esclaves était d'environ 1.7 *sec*. Le surcoût moyen d'une opération de sauvegarde au niveau des esclaves exprime le temps moyen passé par l'application dans une opération de sauvegarde. Cette mesure est un peu au dessus de la seconde, ce qui est très raisonnable.

En constatant ces trois mesures, on s'aperçoit qu'elles sont légèrement plus importantes pour *prime* que celles relevées pour *tabu*. Ceci est dû sans doute à la taille des unités partielles de travail plus importante pour *prime*. L'écart type, relativement faible pour les deux applications montre, qu'en général, le surcoût de toutes les opérations de sauvegarde n'est pas loin du surcoût moyen. Enfin, le surcoût moyen du mécanisme montre son efficacité (moins de 0.5% pour les deux applications). Cela est confirmé par le temps d'exécution important des deux applications (plus de 5 *heures* pour *tabu* et environ 3.5 *heures* pour *prime*) et le nombre d'opérations de sauvegarde effectuées (plus de 60 pour *tabu* et plus de 40 pour *prime*).

Outre les performances enregistrées sur le plan temporel, l'algorithme présente une amélioration sur le plan espace de stockage, en réduisant la quantité de données stockées dans le fichier de sauvegarde. Les résultats présentés dans le tableau 2.3 illustrent la taille des fichiers de sauvegarde. En moyenne, environ 1.7 *Mo* pour *tabu* et 3 *Mo* pour *prime*. La taille considérable du fichier de sauvegarde de *prime* résulte des quantités importantes de données que cette application prend en charge. D'ailleurs, ce résultat était attendu lorsque nous avons analysé les temps des opérations de sauvegarde au niveau des esclaves. L'écart type considérable enregistré au niveau de l'application *prime* peut être expliqué par la liste des nombres premiers trouvés qui ne cesse de grandir au fil du temps, et qui est stockée systématiquement dans l'espace de données du maître.

partiels ont des tailles très proches.

Application	Taille moyenne du fichier de sauvegarde (Kbytes)	Ecart type
tabu	1713	76
prime	3354	1320

TAB. 2.3 - *Taille des fichiers de sauvegarde.*

2.2.6 Analyse qualitative

Outre les résultats quantitatifs obtenus, notre approche présente d'autres avantages sur le plan qualitatif:

- **équilibrage de charge**: au moment du recouvrement, l'application se présente avec les unités de travail (non traitées ou partiellement traitées) dans l'espace de données du maître et sans esclave. Ainsi, l'ordonnanceur peut équilibrer la charge en créant les nouveaux esclaves (de l'application restaurée) sur les nœuds les moins chargés;
- **hétérogénéité**: avec les méthodes de sauvegarde/reprise classiques, la sauvegarde d'un processus s'exécutant sur une architecture spécifique impose la reprise du processus sur le même type d'architecture. Cependant, une application reprise, suite à une défaillance, ne peut pas bénéficier de la disponibilité de ressources du système. Notre approche ne considère pas l'espace d'adressage des esclaves, mais plutôt les données qu'ils détiennent. Par conséquent, l'application est reprise sur n'importe quel ensemble de nœuds libres sans considération de l'architecture. L'indépendance de l'architecture matérielle est par conséquent garantie;
- **indépendance de la charge du système au moment du recouvrement**: avec notre approche, une fois qu'une application est reprise, elle ne nécessite pas que la quantité de ressources soit la même qu'au moment de la sauvegarde. En effet, elle peut reprendre avec plus ou moins d'esclaves. Le modèle d'application parallèle adaptative basé sur le paradigme maître/esclaves est à l'origine de ces propriétés;
- **transparence**: le mécanisme de sauvegarde/reprise ne requiert pas d'interventions additionnelles de l'utilisateur du moment où il utilise l'outil de *repli de l'application* de MARS. Par conséquent, l'utilisateur n'est pas tenu à connaître l'existence du mécanisme. Hormis ce fait, une interface de programmation est fournie à l'utilisateur afin qu'il puisse manipuler explicitement le mécanisme de sauvegarde/reprise (voir annexe A);
- **portabilité**: l'algorithme de sauvegarde/reprise est implémenté entièrement au niveau utilisateur en utilisant les outils fournis par les appels système et les différentes bibliothèques du système Unix. L'avantage d'une telle implémentation est que nous n'avons pas besoin de modifier le noyau, ce qui assure la

portabilité du mécanisme sur les différentes plateformes Unix (SunOS, Solaris, OSF et Linux).

2.3 Travaux similaires

De nombreux algorithmes de sauvegarde/reprise classés dans la catégorie des approches coordonnées ont été conçus et mis en œuvre sur l'environnement PVM. Un algorithme de sauvegarde/reprise est développé pour le système MIST [CCG+95]. Sa complexité en terme de messages de contrôle utilisés par l'opération de sauvegarde est en $O(n^2)$. De plus, il prend en charge des quantités de données importantes, car il considère l'espace d'adressage de tous les processus de l'application. Fail-safe PVM [LFS93] et CoCheck [Ste95] utilisent des algorithmes similaires en complexité à celui de MIST. Fail-safe inclut les démons PVM dans l'opération de sauvegarde. Contrairement à ces algorithmes, notre algorithme réduit la complexité en terme de messages de contrôle à $O(n)$ et le fichier de sauvegarde pour une classe spécifique d'applications parallèles: les applications adaptatives basées sur le modèle SPMD. Le tableau 2.4 résume les caractéristiques principales de ces systèmes.

	MIST	**Fail-safe**	**CoCheck**	**MARS**
Complexité en communication	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$
Complexité en espace de stockage	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Intervalle de sauvegarde	statique	statique	statique	dynamique
Paradigme de programmation	processus communicants	processus communicants	processus communicants	maître/esclaves

TAB. 2.4 - *Comparaison d'algorithmes de sauvegarde/reprise coordonnés.*

2.4 Conclusion

D'une part, les applications parallèles issues de nombreux domaines (calcul scientifique, optimisation combinatoire, etc) deviennent de plus en plus complexes et nécessitent par conséquent des temps d'exécution importants. D'autre part, l'évolution des technologies (processeurs, réseaux de communication, etc) ont poussé les réseaux de stations à devenir des plateformes efficaces pour le calcul parallèle. Ces environnements sont généralement partagés entre plusieurs utilisateurs et la fréquence des défaillances est élevée.

La plupart des environnements de programmation parallèle et distribuée (PVM, MPI, LINDA, etc) ne prennent pas en compte l'aspect tolérance aux fautes ni l'ordonnancement adaptatif. De plus, la majorité des algorithmes de sauvegarde/reprise développés ne considèrent pas les propriétés spécifiques de certaines classes d'applications qui peuvent être à la base d'importantes optimisations de ces algorithmes. Notre approche exploite les propriétés des applications adaptatives afin de réduire la complexité de l'algorithme de sauvegarde à la fois en terme de quantité de messages de contrôle échangés au moment de la sauvegarde qui est en $O(n)$ au lieu de $O(n^2)$ pour les algorithmes classiques, et en terme de quantité de données stockées dans le fichier de sauvegarde.

Les performances obtenues à travers les mesures effectuées sur deux applications réelles sont très encourageantes et montrent que le surcoût induit par le mécanisme de sauvegarde est négligeable (moins de 1% du temps d'exécution de l'application avec une période de 5 *mn*).

Nous avons proposé et mis en œuvre un mécanisme de tolérance aux fautes pour des applications classées dans la catégorie d'applications *commerciales à usage général* (voir sous-section 1.1.2), *transparent à l'utilisateur*, *efficace*, *performant* et *bien adapté à l'environnement sous-jacent*, en l'occurence les systèmes d'ordonnancement adaptatif dans les réseaux de stations. De plus, l'algorithme présente des propriétés intéressantes sur deux plans : hétérogénéité et redistribution de la charge.

Comme nous avons pu le constater à travers ce qui a été présenté jusqu'à présent, les défaillances sont caractérisées par plusieurs facteurs : leurs effets sur le système et l'environnement, mais également leur domaine (par valeur ou temporel). Cette dernière caractéristique est en relation étroite avec le modèle de défaillances adopté i.e. comment déclarer une défaillance? La résolution de ce problème facilite la prise en charge automatique des défaillances. Par conséquent, l'intégration du mécanisme de sauvegarde/reprise dans un système réel peut être effectuée efficacement en développant deux autres composantes chargées de la détection et du recouvrement automatique des défaillances.

Un autre aspect des méthodes de sauvegarde/reprise qui détermine leur efficacité relève de la fréquence d'utilisation du mécanisme. En effet, une utilisation périodique avec une période trop courte peut pénaliser l'application et avec une période trop longue peut induire des pertes importantes en cas de défaillance. De plus, la longueur de la période dépend de l'application dans le sens où le surcoût des sauvegardes influence la période. Par conséquent, une période dynamique prenant en compte ces aspects s'impose.

Le chapitre suivant sera consacré à l'étude des problèmes de l'intégration du mécanisme de tolérance aux fautes dans le système MARS (détection des défaillances, recouvrement automatique, etc) et de la gestion de la période de sauvegarde.

Chapitre 3

Gestion de la tolérance aux fautes

Dans le chapitre précédent, nous avons présenté un outil de tolérance aux fautes pour les applications parallèles adaptatives. L'approche utilisée est celle basée sur une méthode logicielle transparente à l'utilisateur : la *sauvegarde/reprise*. Ce chapitre est consacré au traitement d'un autre aspect du problème de la tolérance aux fautes dans les systèmes distribués : la *récupérabilité ("recoverability")* des composants défaillants [Bir96]. Il s'agit de méthodes et de mécanismes permettant d'assurer le fonctionnement d'un système pourvu de fautes (détection des défaillances, recouvrement automatique des composants défaillants, etc). L'objectif de cette composante est en particulier l'utilisation automatique du mécanisme de sauvegarde/reprise. Le recouvrement automatique des composants défaillants et la préservation de la cohérence de l'application diffèrent d'un modèle à l'autre.

Dans le modèle d'applications basé sur les groupes et la diffusion, la cohérence est assurée par des mécanismes tels que l'unicité des vues des membres d'un groupe, l'ordre de retrait de messages, etc. Le *"modèle virtuellement synchrone" (virtually synchronous model)*, introduit par Isis [Bir96], est un modèle utilisant l'ordre causal entre les événements pour garantir la cohérence de l'application. Des mécanismes tels que l'unicité des vues, le transfert d'état et la fusion de partitions sont utilisés pour préserver la cohérence de l'application et pour la prise en charge des défaillances des processus et des liens de communication [Bir96, RHB94, BDM95]. Horus [RHB94] et Relacs [BDM95] sont basés sur le modèle virtuellement synchrone et gèrent les partitions multiples également. La communication entre les processus du groupe est basée sur la diffusion. La stabilisation des messages[1], effectuée pour garantir la cohérence de l'application en cas de défaillance ou de fusion, utilise généralement des protocoles complexes tels que la diffusion régulière de quantités importantes d'informations entre tous les membres du groupe [RHB94]. Le modèle de groupes et de diffusion induit ainsi des surcoûts non négligeables pour effectuer ces opérations même dans le cas des modèles qui ne gèrent pas les partitions multiples tels que Isis. Ce mécanisme est plus adapté aux modèles basés sur la réplication.

1. Un message est dit non-stable s'il n'a pas été vu par tous les membres du groupe.

La section suivante est consacrée à la présentation les méthodes de détection des défaillances ainsi qu'aux modèles de systèmes distribués. La section 3.2 présente notre approche de détection et de recouvrement des défaillances. La reprise automatique permet le recouvrement d'applications adaptatives en impliquant une partie minimale de l'application dans l'opération de reprise. Dans la section 3.3, nous traitons un aspect très important des méthodes de sauvegarde/reprise : la définition de la période de sauvegarde. Pour cela, un modèle mathématique basé sur les techniques de chaînes de Markov est utilisé afin de définir la période de sauvegarde optimale.

3.1 Modèles de systèmes distribués et modes de défaillance

Suivant des considérations temporelles, les modèles de systèmes distribués sont traditionnellement classés en :

- *systèmes asynchrones :* dans lesquels, ils n'existent pas de contraintes temporelles sur la durée de communication, sur la dérive des horloges, ni sur la vitesse relative des processeurs [CT96]. Ce modèle est dit également modèle *Délais Non Bornés* (DNB) [LMP94] ;
- *systèmes synchrones :* dans lesquels, les bornes existent et sont connues a priori (modèle *Délais Bornés et Connus* (DBC) [LMP94]) ;
- *systèmes partiellement synchrones* : dans ce type de systèmes étudiés par Dwork et Lynch, le délai de communication Δ et la vitesse relative des processeurs Φ existent et sont connus, mais seulement après une certaine période de stabilisation appelée *"Global Stabilization Time"* [DLS88]. Ce modèle ressemble en quelque sorte au modèle *Délais Bornés mais Inconnus* (DBI) [LMP94].

Le modèle partiellement synchrone a été introduit dans le but d'améliorer les résultats des systèmes asynchrones dans lesquels de nombreux problèmes sont intraitables (consensus, terminaison déterministe, etc) [FLP85]. Les systèmes synchrones sont trop simplistes et imposent des contraintes fortes. Par conséquent, leur utilisation est très restreinte. Les systèmes asynchrones imposent un minimum de contraintes et leur utilisation est de ce fait répandue.

Le mode de défaillance détermine le comportement du composant lors de sa défaillance et la manière dont celle-ci est aperçue par les autres composants. Il existe deux grandes catégories de modes de défaillance :

- une catégorie simple dite *par arrêt*. Dans cette classe, un composant défaillant ne peut plus initier d'opérations erronées ou envoyer de messages incohérents. Formellement, si le processus p défaille à l'instant t, p n'exécutera plus aucune action au temps $t'(t' \geq t)$. Cette catégorie comporte deux sous-classes : la

sous-classe *figement sur défaillance (fail-stop)* et la sous-classe *silence sur défaillance (crash failure)* [Bir96]. Le mode *figement sur défaillance* suppose que la défaillance d'un composant est instantanément connue par tous les autres composants du système. En revanche, le mode *silence sur défaillance* conclue la défaillance d'un composant à partir de l'absence d'événements tels que l'envoi de messages. Dans la pratique, ce mode conduit souvent à des confusions entre la défaillance du composant proprement dit et celle des canaux de communication qui lui sont reliés ;

- la seconde catégorie comporte les modèles considérant les défaillances arbitraires appelées également *Byzantines* [LSP82, PSL80]. Ce mode est plus général et considère que les défaillances peuvent également se manifester par un comportement arbitraire des composants. Un composant défaillant arbitrairement peut effectuer des opérations erronées ou émettre des messages incohérents, ce qui complique certains protocoles tels que le consensus, la validation de messages, etc.

Nous faisons l'hypothèse d'un système distribué asynchrone dans lequel, les bornes sur la vitesse relative des processeurs et sur les délais de transmission des messages ne sont pas connues. Le système de communication sous-jacent est supposé garantir le séquencement et la livraison des messages faisant apparaître un système de communication possédant des canaux FIFO. Les processus défaillent en s'arrêtant prématurément (mode silence sur défaillance) permettant ainsi aux autres composants du système de découvrir éventuellement la défaillance. Des *détecteurs de défaillances* sont superposés aux processus et leur fournissent des informations sur la défaillance des autres processus du système. La détection des défaillances utilise une approche temporelle basée sur des délais de garde [CT96].

3.2 Détection et recouvrement des défaillances

Les travaux concernant la détection des défaillances sont relativement rares [Bir96]. La majorité des systèmes distribués asynchrones existants utilisent des *délais de garde (timeouts)* afin de détecter les défaillances des composants. L'*inexactitude* de ces méthodes pousse les applications à prendre des mesures parfois erronées comme l'exclusion de processus soupçonnés à tort d'avoir défailli [Bir93, Cri88]. Pour remédier à ce problème, Chandra et Toueg ont introduit le concept de *détecteurs de défaillances non fiables* dont les décisions sont révocables. Des moyens permettant de réhabiliter les membres accusés à tort sont mis en œuvre [CT96]. D'autres chercheurs tels que Vogels mettent l'accent sur les moyens complémentaires permettant d'améliorer la qualité et la précision de la détection (bases de données d'utilisation des réseaux, des ressources, etc) [Vog96].

3.2.1 Détecteurs de défaillances

Un détecteur de défaillances non fiable est un composant externe qui fournit au processus associé des informations correctes ou non sur la défaillance des autres processus du système.

Dans [CT96], ce concept est introduit et utilisé pour la résolution de nombreux problèmes insolvables d'une manière déterministe dans les systèmes asynchrones en présence de défaillances par arrêt (consensus, diffusion atomique, etc). Ces détecteurs de défaillances sont caractérisés par deux propriétés principales: la complétude et l'exactitude.

Les systèmes asynchrones sont sollicités de par leur portabilité, résultat d'[illegible]nce d'hypothèses temporelles sur la communication et la vitesse des processeurs [illegible]ue sur l'état du système. L'inconvénient principal de ce modèle provient de la [illegible]té de distinguer un processus défaillant d'un processus lent, d'où l'impossibilité de résolution de certains problèmes. Plusieurs travaux ont étudié ces problèmes dans des modèles restreints (systèmes partiellement synchrones [DL88]) ; d'autres travaux ont relaxé les problèmes [DLP+86] et une troisième classe de travaux utilise des modèles probabilistes [CD89].

Dans le modèle proposé par Chandra et Toueg, un détecteur local de défaillances est associé à chaque processus. Chaque module local gère un sous-ensemble de processus du système et maintient une liste des processus qu'il soupçonne d'avoir défailli actuellement. Le détecteur de défaillances est non fiable dans le sens où il peut commettre des erreurs en suspectant à tort un processus correct d'avoir défailli. Si le détecteur découvre ultérieurement qu'il a eu tort à propos d'un processus, ce dernier est simplement éliminé de la liste des suspects actuels. Notons que la liste des suspects au niveau d'un processus peut être différente de celles des autres (multiplicité de vues). De plus, les processus corrects ne sont pas affectés par les soupçons erronés.

La plupart des réalisations des détecteurs de défaillances utilisent la *surveillance par délais de garde*. Le délai de garde est souvent augmenté à chaque accusation erronée afin de prévenir la même situation dans le futur [DL88]. Il est clair que dans l'absolu, cette approche ne peut garantir les propriétés attendues. En pratique, cela n'est généralement pas le cas, car augmenter le délai de garde à chaque faux soupçon permet d'atteindre un état de stabilité.

Dans Isis [Bir96], la couche de transport du système de communication est intégrée avec la couche de détection de défaillances dans le but de construire un modèle par figement sur défaillance (fail-stop) [Bir93]. Dès la détection d'une défaillance, une nouvelle vue est d'abord installée par tous les survivants. Les partitions multiples ne sont pas autorisées et un membre suspecté est forcé à quitter le groupe.

Dans notre modèle, les processus appartiennent à deux classes : *processus système* et *processus application*. La classe des processus système se compose des démons de gestion de la communication, de la collecte d'informations de charge, etc. La classe

des processus application, quant à elle, comporte tous les processus utilisateurs.

Les défaillances sont supposées totales dans le cas de la défaillance d'un processus système[2]. Lorsqu'un processus système défaille ou est suspecté de défaillance, le nœud qu'il contrôle est considéré entièrement défaillant, même si la défaillance est causée par l'interruption des liens de communication. En revanche, la défaillance d'un processus application met en cause le processus lui-même uniquement.

La détection de défaillances dans notre modèle utilise deux approches:

1. en utilisant la couche de communication en mode figement sur défaillance[3]. Cela est utile en particulier pour les processus application lorsque l'arrêt n'est pas dû à la défaillance totale du nœud. D'autre part, ce modèle assure une détection rapide des défaillances et permet ainsi de réagir dans les plus brefs délais;

2. en utilisant une approche basée sur les délais de garde dans laquelle, les processus se surveillent mutuellement de telle sorte que la défaillance d'un processus est découverte rapidement sans attendre que le processus soit contacté pour des raisons fonctionnelles. Le modèle ne considère pas les partitions multiples, et si un processus ayant été suspecté à tort rejoint l'application, il sera forcé de quitter le système. Ce choix simple permet de garantir la progression de l'application, car les défaillances de communication peuvent prendre des temps excessifs et les calculs dépendants des tâches traitées par les processus suspects vont se voir bloqués.

Le détecteur de défaillances que nous avons utilisé est illustré sur la figure 3.1. Périodiquement, le détecteur de défaillances associé au processus p diffuse un message de survie aux processus qu'il est chargé de surveiller. Le booléen *pinged* associé à chaque processus surveillé p_i permet de conserver cette information. La réponse de p_i agit sur *pinged* en changeant son état de *vrai* à *faux*. Si au bout d'un délai prédéfini, un processus p_i ne répond pas (*pinged* associé à p_i est toujours *vrai*), il est considéré défaillant. La fonction *failure* a pour conséquence d'informer le processus p de la défaillance de p_i. p doit, par la suite, entreprendre les actions appropriées. Les délais de garde sont suffisamment grands de telle sorte que les problèmes transitoires (charge excessive des nœuds, surcharge du réseau) seront contournés. Par la suite, nous distinguons trois types de défaillances:

- *défaillance du système*: il s'agit de la défaillance du GS. Dans ce cas, toutes les applications doivent être terminées automatiquement, le système relancé et les applications reprises manuellement. Cela suppose, bien entendu, que les utilisateurs soient informés;

2. Cela est dû au fait que l'arrêt d'un démon prive le système du service fourni par celui-ci et rend le nœud par conséquent inutilisable.
3. Mécanisme de notification de PVM.

```
Paramètres :
p : processus auquel, le détecteur de défaillances est associé.
process_set : ensemble de processus que le détecteur est chargé de surveiller.

1- while (1) do
2-     mcast (process_set, ping_message)
3-     ∀p_i ∈ process_set, p_i.pinged = TRUE
4-     sleep (TIMEOUT_DELAY)
5-     for p_i ∈ process_set do
6-         if p_i.pinged then
7-             process_set = process_set − p_i
8-             failure(p_i)
9-         end if
10-    end for
11-    sleep (DETECT_PERIOD)
12- end while
```

FIG. 3.1 - *Détecteur de défaillances.*

- *défaillance du module maître d'une application :* cela implique le recouvrement total de l application ;
- *défaillance d'un esclave :* ce type de défaillances est recouvert par la reprise de l'esclave défaillant sans le reste de l'application.

Afin de permettre le recouvrement partiel des applications, le maître doit tenir compte des allocations i.e. quelle unité de travail est allouée à quel esclave? Par conséquent, la gestion transparente du travail est indispensable.

3.2.2 Gestion transparente du travail : un cadre pour le recouvrement partiel

Afin de permettre une gestion transparente du travail, une interface de programmation est développée. Cette interface basée sur des *graphes de dépendances* permet à l'utilisateur de construire son application parallèle adaptative aisément. La construction de l'application consiste à spécifier les *tâches* de l'application et à décrire les *contraintes de précédence* entre elles.

Une tâche est un fragment de données sur lesquelles, un ensemble d'actions séquentielles doit être exécuté. Au niveau du maître, le programmeur spécifie les tâches de l'application et les confie à la bibliothèque. Celle-ci construit un *espace de travail* et un *graphe de dépendances* qui assure l'ordre d'exécution entre les tâches lors de leur allocation.

L'allocation des tâches aux esclaves est réalisée automatiquement par un module de la bibliothèque au niveau du maître (le *serveur de travail*) sans l'intervention de l'utilisateur. Ce dernier peut intervenir lors de la réception des tâches traitées partiellement et des résultats pour une possible optimisation de l'espace de stockage ou du temps d'exécution ou pour générer de nouvelles tâches dynamiquement.

L'esclave consiste généralement en une boucle commençant par un appel à la bibliothèque interprété par une requête adressée au maître. Le serveur de travail cherche dans l'espace de travail une tâche prête à être exécutée. Lorsqu'il parvient à en trouver une, il la retourne à l'esclave. Ayant obtenu la tâche, l'esclave lance le thread approprié. Lorsqu'il a fini, il appelle une fonction spécifique qui se charge de retourner les résultats au maître. Lorsqu'un ordre de *repli* arrive ou lorsqu'une opération de sauvegarde est déclenchée, une fonction définie par l'utilisateur est appelée automatiquement. Celle-ci permet de remettre la tâche partiellement traitée au maître. Une présentation détaillée du modèle d'exécution sera donnée dans la partie suivante consacrée aux problèmes d'ordonnancement.

3.2.3 Comment est effectué le recouvrement ?

Comme nous l'avons présenté au chapitre 2, MARS se compose d'un *serveur de groupe* (GS) dont le rôle est de contrôler un groupe de nœuds. Sur chaque nœud, réside un *gestionnaire de nœud* (NM) qui informe systématiquement le GS des transitions de l'état du nœud qu'il contrôle (figure 2.2, chapitre 2). Lorsqu'une application s'enrôle au système, elle lance un représentant nommé le *médiateur* sur le nœud du GS. Son rôle est de relancer l'application en cas de défaillance du maître[4].

Les algorithmes de tolérance aux fautes doivent assurer la détection et la gestion efficaces des défaillances et également la cohérence des recouvrements en évitant le recouvrement automatique de certaines défaillances dûes à des fautes de conception, aux intrusions externes ou aux interventions humaines.

3.2.3.1 Recouvrement au niveau système

La détection des défaillances est effectuée suivant la première approche décrite précédemment en prévoyant un *détecteur de défaillances* au niveau du GS afin de surveiller les NMs. En effet, les NMs sont régulièrement contactés. Lorsque l'arrêt d'un NM est détecté, la défaillance du nœud est déclarée et tous les processus s'exécutant sur le nœud sont considérés défaillants. Le GS procède aux recouvrements nécessaires et à la mise à jour des informations d'état du système qu'il maintient. Les structures de données suivantes sont utilisées par les algorithmes :

AppliTable : table d'applications en cours d'exécution dans le système.
NodeTable : table de nœuds (configuration).

4. Ce processus possède les mêmes droits que l'application et survit aux défaillances tant que le système est opérationnel.

```
Paramètres :
NodeID : identificateur du nœud défaillant.

1- delete_opera (NodeID)
2- failed_applis = find (appli : appli ∈ AppliTable ∧ appli.NodeID = NodeID).
3- NewNodeID = find_operat_node(NodeID, -1)
4- for appli ∈ failed_applis do
5-      call (appli.mediator, RECOVER, NewNodeID)
6- end for
7- for appli ∈ failed_applis do
8-      receive (appli.mediator, RECOVER_FEEDBACK, result)
9-      if result.Recovered = FALSE then
10-         if (result.Attemps < MAX_RECOVERY_TRIES) then
11-             NewNodeID = find_operat_node(NodeID, NewNodeID)
12-             call (appli.mediator, RECOVER, NewNodeID) ; goto 8
13-         else
14-             call (appli.mediator, TERMINATE, NULL)
15-             free_appli (appli)
16-         end if
17-     else
18-         if not appli.FailureDiscovered then
19-             Add (NodeTable[NodeID].FailedApplis, appli.ID, appli.Incarn)
20-         end if
21-     end if
22- end for
23- Concurrently => result = Recover (NodeID, RECOVERY_DELAY)
24- if result.Recovered = TRUE then
25-     Add_operat (NodeID)
26- else
27-     Delete_node (NodeID)
28- end if
```

FIG. 3.2 - *Réaction du GS lors de la défaillance d'un NM.*

Défaillance d'un NM : la figure 3.2 résume le comportement du GS lorsque son *détecteur de défaillances* lui reporte la défaillance d'un nœud. Pour chaque application dont le maître s'exécutait sur le nœud défaillant (ligne 2), le GS exécute les actions suivantes :

- il tente de recouvrir l'application entièrement car la défaillance du maître est critique. Cela est réalisé en appelant le médiateur de l'application afin de procéder à la reprise de l'application (ligne 5). Le GS lui fournit un nœud de

même architecture que celui qui a défailli (ligne 3), dans la limite des nœuds disponibles[5] ;

- si la reprise n'a pas pu s'effectuer, d'autres tentatives sont effectuées (lignes 8 à 12). Si après toutes les tentatives, l'échec demeure, une faute permanente est suspectée. Le recouvrement est abandonné et le médiateur, dernière composante de l'application, est terminé (lignes 13 à 16). L'échec de la relance peut être dû à des contraintes techniques liant l'application au nœud (insuffisance d'espace mémoire, etc) ou à la défaillance du nœud de reprise lui même (cas de défaillances simultanées). Dans ce cas, le recouvrement de l'application risque d'être abandonné par oubli. La réitération des tentatives permet d'éviter cette confusion sans être obligé de vérifier si le nœud est réellement défaillant, qui peut prendre un temps excessif ;

- en revanche, si l'application est recouverte, quelques informations déterminant avec certitude la version de l'application (la réincarnation) concernée par cette défaillance sont sauvegardées (lignes 17 à 21). Cela a pour objectif d'éviter des confusions dans le futur entre la défaillance de l'application dûe à la défaillance d'un nœud et celle résultant des fautes applicatives ou humaines et dont le recouvrement automatique est inutile (voir paragraphe suivant).

Parallèlement au recouvrement des applications sur les autres nœuds, une tentative de recouvrement du nœud défaillant est effectuée. Si au bout d'une période pré-déterminée, le nœud n'a pas pu être récupéré, il est tout simplement écarté de la configuration (lignes 23 à 28). Le recouvrement des applications est effectué immédiatement après la détection de la défaillance sans attendre la réparation du nœud, qui peut prendre un temps excessif.

Défaillance du maître : la défaillance du maître peut être vue par le *détecteur de défaillances* du médiateur via la couche de communication (fail-stop) ou après l'expiration du délai de garde suite à une communication dans le cas de la défaillance du nœud. Le médiateur procède à l'information du GS, qui exécute les actions suivantes :

- l'application est recherchée dans la liste d'applications éventuellement recouvertes par le GS dans l'entrée du nœud, en utilisant le numéro de version de l'application. Si elle est retrouvée, cela signifie qu'elle a déjà été recouverte. Par conséquent, elle est tout simplement éliminée de cette liste ;

- en revanche, si l'application n'est pas retrouvée, le GS accuse la découverte de la défaillance du maître avant celle du nœud. Par conséquent, il procède au test du nœud. S'il est effectivement défaillant, il marque la découverte de la défaillance par l'application. Dans le cas contraire, il soupçonne une faute

5. Cela est contraint par la reprise du maître qui doit s'effectuer sur la même architecture que celle sur laquelle, il s'exécutait au moment de la sauvegarde.

de conception ou une intrusion humaine. Par conséquent, l'application est simplement terminée, en l'occurence le médiateur.

Ce protocole sert uniquement à vérifier et à préserver la cohérence des opérations de recouvrement automatique. Le recouvrement effectif est effectué au moment de la détection de la défaillance du NM comme indiqué précédemment. Notons que la terminaison sans défaillance de l'application est exprimée explicitement par un message émis par le maître.

3.2.3.2 Recouvrement au niveau application

Le *serveur de travail* alloue les tâches aux esclaves et garde leur état. La figure 3.3 illustre les transitions d'état des tâches induites par différents événements.

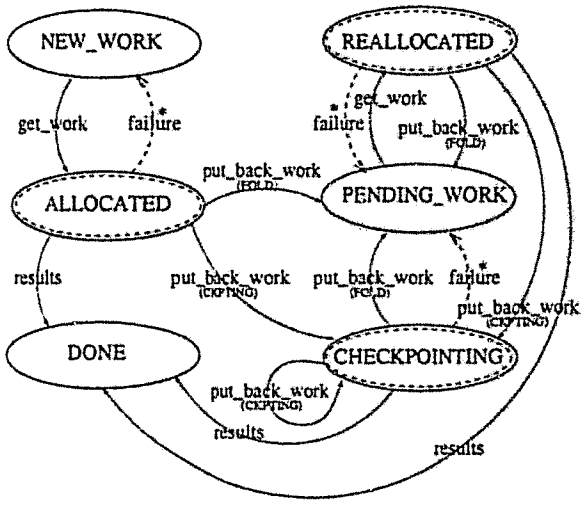


FIG. 3.3 - *Graphe de transition d'état des tâches et recouvrement des esclaves.*

Une tâche affectée à un esclave peut être dans un des trois états : *ALLOCATED*, lorsqu'elle est allouée pour la première fois, *REALLOCATED* pour une tâche ayant été allouée plus d'une fois, ou *CHECKPOINTING* lorsque l'esclave auquel elle est allouée est en état de sauvegarde. Lorsqu'une opération de repli concernant l'esclave est déclenchée, celui-ci retourne la tâche partiellement traitée au maître avant de terminer. Le serveur de travail met l'état de la tâche retournée en *PENDING_WORK*. Le travail effectué par l'esclave évacué pourra être ainsi exploité.

De nouveau, la détection des défaillances des esclaves est assurée par un *détecteur de défaillances* superposé au maître et consacré à la détection des défaillances des esclaves d'un côté et du médiateur de l'autre côté. La détection des défaillances des esclaves est assurée par la couche de communication dans le cas de l'arrêt d'un esclave seul (mode fail-stop), par l'expiration des délais de garde lors d'une communication

avec l'esclave en cas de défaillance du nœud ou encore par le GS lors de la découverte de la défaillance du nœud. Les défaillances sont vues et gérées à ce niveau comme suit :

Défaillance d'un esclave : la défaillance d'un esclave n'est pas évidente à conclure lorsque la défaillance du nœud ne survient pas, car la couche de communication ne peut pas distinguer la terminaison normale et défaillante d'un esclave. Afin de remédier à ce problème, le maître affecte un état à chaque esclave qui reflète son évolution à tout instant. Les états ainsi définis sont classés en deux classes : les états finaux S_F et les états transitoires S_T. L'ensemble des états finaux constitue l'état de terminaison sans défaillance de l'esclave. Si un esclave termine dans un état s : $s \notin S_F$, le maître déduit que l'esclave en question vient de défaillir. Par conséquent, il procède à son recouvrement.

Ce procédé est également utile pour le maître lors d'une opération de sauvegarde. En effet, lors de la déconnexion du système, le message envoyé au médiateur peut croiser des requêtes de *repli/dépli* en provenance du système. L'état des esclaves permet de détecter l'arrivée de ces requêtes. D'un autre côté, pendant l'opération de sauvegarde, le médiateur diffère toutes les requêtes du système. Celles-ci seront réenvoyées après la terminaison de la sauvegarde.

Le recouvrement d'un esclave déclaré défaillant est effectué en cherchant dans l'espace de travail du maître les tâches traitées par l'esclave lors de sa défaillance et en altérant son état comme illustré par la figure 3.3. Ainsi, si la tâche était en état de *CHECKPOINTING* ou de *REALLOCATED*, cela signifie qu'un état partiel de la tâche est disponible. Par conséquent, son état devient *PENDING_WORK* et le travail effectué sur la tâche avant la défaillance pourra être exploité. En revanche, si la tâche était en état *ALLOCATED*, cela signifie que le maître ne dispose d'aucun état partiel de la tâche. Celle-ci passe à l'état *NEW_WORK* et son traitement doit reprendre dès le début.

Défaillance du maître : la défaillance du maître peut être également vue par les *détecteurs de défaillances* associés aux esclaves et par celui associé au médiateur. Les détecteurs de défaillances des esclaves utilisent les informations fournies par la couche de communication ou par l'expiration des délais de garde lors de la communication avec le maître. Le médiateur de son côté est informé par son *détecteur de défaillances* grâce à la couche de communication ou par le GS si la terminaison est causée par la défaillance du nœud. La défaillance du maître provoque les actions suivantes à ce niveau :

- tous les esclaves auront connaissance de la défaillance au bout d'un temps fini grâce à leurs détecteurs de défaillances. Par conséquent, ils procèdent à terminer leur exécution immédiatement. Ainsi, nous assurons qu'il n'y ait plus de processus inutiles (*orphelins*) ou dupliqués après le recouvrement de l'application. Toutefois, si l'application est recouverte avant qu'un esclave P_i de l'ancienne version ne découvre la défaillance du maître, P_i ne pourra pas inter-

agir avec le nouveau maître et finira par découvrir la défaillance, car il possède uniquement l'identificateur de l'ancien maître ;

- lorsque le médiateur prend connaissance de la défaillance du maître grâce à son *détecteur de défaillances*, il transmet cette information au GS et attend ses instructions. Comme nous l'avons décrit précédemment, le GS détermine si la défaillance doit donner lieu à un recouvrement.

Si le GS opte pour le recouvrement de l'application, il informe le médiateur. Celui-ci procède à la création d'un nouveau maître qui se ré-enrôle au système (voir section 2.2.2, chapitre 2).

Défaillance du médiateur : l'arrêt du médiateur est vue par le *détecteur de défaillances* du maître via la couche de communication ou par l'expiration du délai maximum de communication lors de la communication du maître avec le médiateur. Cette défaillance est interprétée comme étant l'arrêt du système (GS). L'application doit par conséquent terminer. Le maître procède alors à la terminaison de ses esclaves avant de s'arrêter. L'application peut effectuer une sauvegarde avant de terminer.

3.2.4 Avantages de l'approche proposée

Hormis le fait que cette nouvelle interface de programmation est indispensable pour le recouvrement partiel de l'application, l'approche présente plusieurs avantages :

- diminution de la complexité de programmation, résultat de l'automatisation de certains aspects de la programmation (communication, synchronisation et allocation de travail) ;
- transparence de la détection et du recouvrement des défaillances ;
- efficacité de l'approche de recouvrement qui implique uniquement les parties défaillantes dans le processus de reprise.

3.3 Gestion de la période de sauvegarde

Déterminer la période de sauvegarde est un problème crucial [DVCL93]. En effet, la période de sauvegarde dépend de plusieurs facteurs liés principalement à l'environnement sous-jacent (nœuds, réseaux, utilisateurs, etc) et à la classe d'application.

Wong et Franklin ont présenté une étude de l'intervalle de sauvegarde pour un algorithme coordonné sur une plateforme massivement parallèle [WF93]. L'estimation de la période optimale est basée sur un modèle analytique utilisant les *chaînes de*

Markov. Le processus[6] peut être dans un des trois états: *Disponible, Sauvegarde* ou *Recouvrement* suivant que le processus est en exécution normale, en train de réaliser une sauvegarde ou en train d'effectuer un recouvrement suite à une défaillance. Les hypothèses suivantes sont considérées:

- les temps d'occupation des états sont Markoviens;
- les défaillances surviennent uniquement lorsque le nœud est en état d'exécution, et constituent un processus *poissonien*;
- les surcoûts et les délais induits par la détection des défaillances ne sont pas pris en compte.

Dans une première version où le système doit attendre la réparation du nœud défaillant avant de procéder au recouvrement, ils montrent que la période optimale est fonction (voir annexe B): du temps moyen d'inter-défaillances d^{-1}, du surcoût moyen de sauvegardes b^{-1}, du nombre de nœuds du système N et du taux d'utilisation sans défaillance du système $U(N)$ (moyenne de l'accélération par nœud). Le temps de recouvrement peut être considérable, empêchant ainsi l'application de progresser. Dans une seconde version où le système est reconfiguré et la charge est redistribuée immédiatement après la défaillance, la période est la même, excepté le fait qu'elle dépende du nombre de nœuds actuellement opérationnels plutôt que du nombre total des nœuds.

Une autre estimation de la période de sauvegarde est développée dans [AS93] pour le système CRAY UNICOS. L'approche utilise un démon qui gère les sauvegardes automatiquement. Ainsi, la période de sauvegarde peut être définie par le système (30 minutes), déterminée par l'utilisateur, ou la sauvegarde périodique automatique est inhibée et la sauvegarde est gérée explicitement par l'utilisateur.

Dans [Vai97], une approche similaire basée sur des temps d'occupation *Markoviens* et une distribution *poissonienne* des défaillances est utilisée pour un modèle d'application séquentielle s'exécutant sur un seul nœud. L'objectif est de déterminer une formule permettant d'optimiser le surcoût de sauvegarde au détriment du coût total de l'opération.

Nous allons reprendre le modèle de Wong et Franklin développé pour le cas d'un nœud unique sans attente de réparation [WF93]. Les hypothèses seront modifiées suivant les caractéristiques de notre environnement, et nous allons étendre ce modèle à notre environnement parallèle adaptatif.

6. Processus et nœud sont confondus car dans une machine parallèle, l'allocation consiste en un processus par nœud et les défaillances affectent le nœud entièrement.

3.3.1 Période de sauvegarde optimale dans le cas d'une application séquentielle

Un processus commence dans l'état *Execution* (figure 3.4). Après une période d'exécution (a^{-1}), il entre dans l'état *Checkpointing*. Il passe b^{-1} unités de temps à effectuer sa sauvegarde. Lorsqu'une défaillance survient, une période s'écoule avant de découvrir la défaillance. Cet état est appelé *Detection* et dure en moyenne e^{-1} unités de temps. Après la découverte de la défaillance, l'application procède au recouvrement en passant à l'état *Recovery*. Après c^{-1} unités de temps passées en recouvrement, elle retourne à l'état *Execution*.

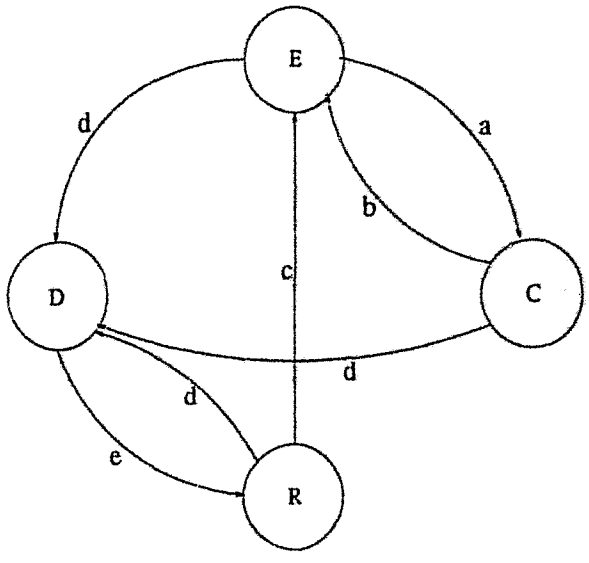


FIG. 3.4 - *Etats de transition d'un nœud unique et taux de transition correspondants: E l'état d'exécution, C celui de sauvegarde, D l'état de détection et R celui du recouvrement.*

Les hypothèses sont différentes de celles de Wong et Franklin, car dans les réseaux de stations, les temps de détection et de réaction sont beaucoup plus importants. En effet, les hypothèses suivantes sont prises:

- les temps d'occupation des états sont Makoviens;
- les défaillances surviennent lorsque le nœud est en état d'exécution, de sauvegarde ou de recouvrement, et constituent un processus *poissonien*;
- les délais induits par la détection des défaillances sont également pris en compte et sont Markoviens.

Les transitions d'états de l'application peuvent être modélisées par une chaîne

Paramètre	signification
a^{-1}	période moyenne de sauvegarde
b^{-1}	surcoût moyen de sauvegarde
c^{-1}	surcoût moyen de recouvrement
d^{-1}	intervalle moyen d'inter-défaillances
e^{-1}	durée moyenne de détection des défaillances

TAB. 3.1 - *Paramètres et transitions d'états du modèle.*

de Markov à temps continu (figure 3.4) et les probabilités de transition sont prises égales à l'inverse des différents intervalles moyens (tableau 3.1).

Notre objectif est de maximiser le temps total d'exécution passé par l'application à effectuer du travail utile. Par conséquent, la fraction de temps passé par l'application dans l'état *Execution* doit être déterminée. Les techniques de chaînes de Markov à temps continu sont utilisées pour calculer l'état stationnaire de la chaîne. La fraction de temps d'exécution ainsi calculée est donnée par:

$$p_E = \frac{ec(b+d)}{dbc + d^2c + adc + ebc + edc + eac + eda + edb + ed^2 + d^2a + d^2b + d^3}. \quad (3.1)$$

En réalité, le coût du recouvrement c^{-1} comporte le temps consacré à la réparation du nœud, à la reprise de l'application r et le temps de recalcul qui correspond au temps écoulé entre la dernière sauvegarde, à partir de laquelle le recouvrement est effectué, et le moment de la défaillance. Ce temps est proportionnel à la période de sauvegarde a^{-1}. Le temps de réparation n'est pas pris en compte dans les calculs car les applications défaillantes sont recouvertes sur d'autres nœuds sans attendre l'éventuelle récupération du nœud défaillant ($c^{-1} = r + a^{-1}$).

La période de sauvegarde optimale correspond au point dans lequel, la fonction p_E atteint son maximum. Ce qui donne:

$$a^{-1}_{optimum} = \sqrt{\frac{1 + dr}{(b+d)d}}. \quad (3.2)$$

Les détails de la résolution et des conditions d'application sont présentés dans la première partie de l'annexe B.

Comme nous pouvons le déduire de la formule 3.2, lorsque la fréquence de défaillances d diminue, la période de sauvegarde $a^{-1}_{optimum}$ augmente. La figure 3.5 trace l'évolution de la période de sauvegarde a^{-1} en fonction de l'intervalle moyen d'inter-défaillances d^{-1} en fixant le reste des paramètres à des valeurs moyennes tirées expérimentalement ($b^{-1} = 2sec$, $r = 3sec$). La période de sauvegarde correspondant à un

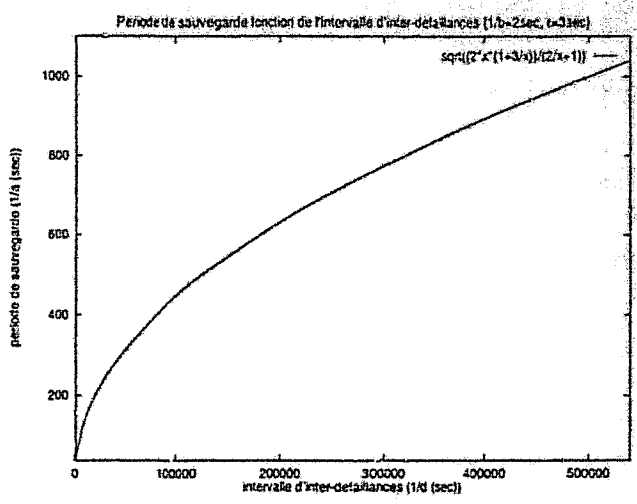


FIG. 3.5 - *Période de sauvegarde en fonction de l'intervalle d'inter-défaillances (1/b=2sec, r=3sec).*

intervalle d'inter-défaillances de $540000sec = 150h$ est d'environ $1039sec = 17.5mn$. Actuellement, l'intervalle de défaillances moyen est d'environ $100h$ [7], la période de sauvegarde correspondante à ce taux est de $848sec = 14.1mn$, ce qui semble judicieux (figure 3.5). Par conséquent, la période de sauvegarde croît avec un intervalle d'inter-défaillances croissant. Ainsi, des opérations de sauvegarde coûteuses et inutiles peuvent être évitées.

La période de sauvegarde dépend également du surcoût de sauvegarde et du coût de recouvrement et croît lorsque ces deux paramètres deviennent de plus en plus importants. Cela est expliqué par le fait que lorsque ces deux paramètres augmentent, ils pénalisent les performances de l'application si les sauvegardes sont effectuées fréquemment. Les figures 3.6 et 3.7 tracent l'évolution de l'intervalle de sauvegarde en fonction du surcoût de sauvegarde et du coût de recouvrement. La période de sauvegarde est étroitement corrélée avec le surcoût de sauvegarde, mais quasiment insensible au coût de recouvrement. De plus, le temps de détection e^{-1} n'intervient pas dans la formule de la période de sauvegarde. Cela est dû au fait que ces deux paramètres sont conditionnés par les défaillances et leur effet sur la période de sauvegarde est limité par la fréquence des défaillances.

La période de sauvegarde optimale définie correspond au cas d'une application à un seul processus. Dans notre modèle, une application comprend n processus, un par nœud. Puisque nous ne procédons au recouvrement total de l'application parallèle que lorsque la défaillance du maître survient, la probabilité de défaillance n'augmente pas lorsque nous étendons le modèle à n nœuds. En revanche, le temps de recalcul proportionnel à a^{-1} dans la formule de recouvrement c peut être affecté par la vitesse d'évolution des calculs avant la défaillance et après le recouvrement. En effet, si l'application évoluait à une vitesse moyenne S_{bf} avant la défaillance et

7. D'après nos statistiques présentées au chapitre 2.

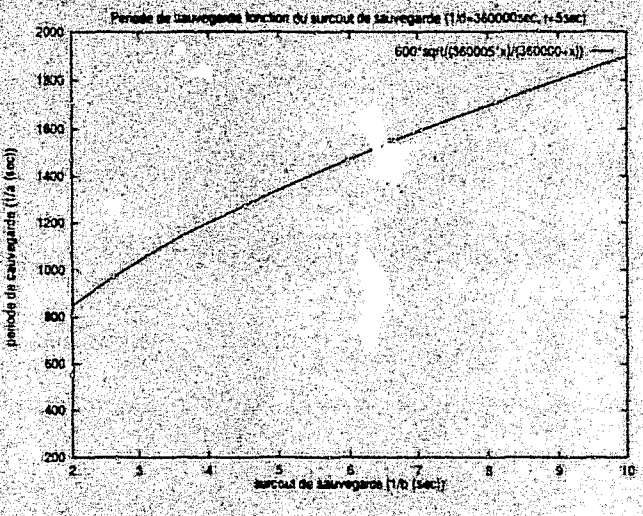


FIG. 3.6 – *Période de sauvegarde en fonction du surcoût de sauvegarde (1/d=360000sec, r=5sec).*

S_{ar} après le recouvrement, le temps de recalcul doit être pondéré par le rapport $\frac{S_{bf}}{S_{ar}}$.

3.3.2 Vers une période adaptative

Comme nous l'avons mentionné, nous nous intéressons aux applications intensives en temps de calcul et de longue durée de vie. Le mécanisme de tolérance aux fautes doit minimiser le plus possible les pertes pouvant être causées par les défaillances et en même temps ne pas contribuer significativement au temps d'exécution de l'application. La meilleure façon pour garantir un tel compromis est de déterminer dynamiquement la période de sauvegarde optimale durant l'exécution de l'application.

Dans la réalisation actuelle, l'estimation de la période de sauvegarde optimale est basée sur des moyennes des mesures relevées pour les sauvegardes précédentes. En effet, si l'exécution de l'application est partitionnée en intervalles délimités par les instants de sauvegarde (figure 3.8), pour l'estimation de l'intervalle de sauvegarde $a^{-1} =]C_i, C_{i+1}]$, nous effectuons les hypothèses suivantes:

- la détection des défaillances est réalisée séparément par le système;
- lors d'une défaillance, nous procédons au recouvrement sans attendre la réparation du nœud défaillant;
- la fréquence des défaillances est prise égale au taux moyen déterminé par nos statistiques $d = 1/360000sec^{-1}$;
- le surcoût moyen de sauvegarde est estimé par la moyenne des surcoûts des sauvegardes précédentes ($b^{-1} = \frac{1}{i+1}\sum_{j=0}^{i} b_j^{-1}$). b_j^{-1} est le surcoût induit par

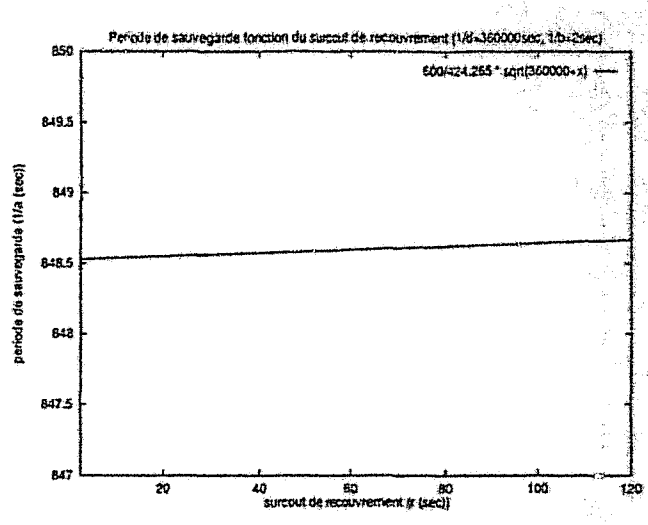


FIG. 3.7 - *Période de sauvegarde en fonction du surcoût de recouvrement (1/d=360000sec, 1/b=2sec).*

la sauvegarde C_j, b_0 est une constante choisie expérimentalement, puisque l'application n'effectue pas de sauvegarde initialement ;

- $r = 1.5b^{-1}$ puisque le coût de recouvrement r dépend étroitement du surcoût de sauvegarde. De plus, dans notre cas, ce coût est augmenté par un temps additionnel lié à la création des nouveaux esclaves après la reprise.

- la vitesse moyenne S_i d'évolution de l'application durant l'intervalle $]C_i, C_{i+1}]$ est estimée par le nombre moyen de nœuds alloués à l'application durant le même intervalle. Ce nombre est pondéré par la fraction de temps durant lequel chaque nœud est utilisé par rapport à la longueur de l'intervalle, et par la puissance relative des processeurs. Le problème qui se pose est que S_{bf} et S_{ar} ne peuvent être connus au moment de la sauvegarde C_i. Pour cela, une moyenne du taux de variation de la vitesse d'évolution de l'application, d'un intervalle de sauvegarde à l'autre du début de son exécution jusqu'à l'instant t_{C_i}, est utilisée. $V(n) = \frac{\sum_{j=i-1}^{j>0} \frac{S_{j-1}}{S_j}}{\lfloor \frac{i}{2} \rfloor}$; j est décrémenté d'un pas de 2 à chaque fois.

Lorsque le terme a^{-1} est remplacé par $V(n)a^{-1}$ dans l'expression du coût de recouvrement c et après le recalcul de p_E et $a_{optimum}$, nous obtenons (annexe B) :

$$a_{optimum}^{-1} = \sqrt{\frac{1 + dr}{dV(n)(b + d)}}. \tag{3.3}$$

La figure 3.9 illustre la variation de la période de sauvegarde en fonction de l'évolution de $V(n)$. Lorsque la vitesse d'exécution de l'application a tendance à augmenter d'un intervalle à l'autre, les sauvegardes doivent être effectuées plus souvent car les calculs évoluent plus vite.

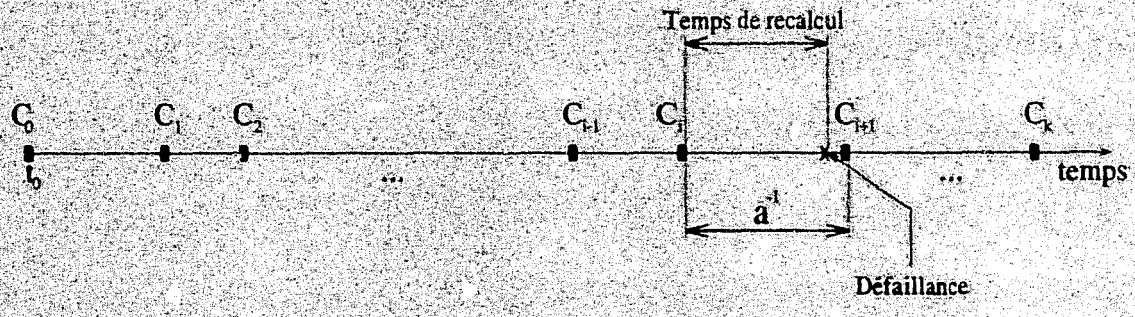


FIG. 3.8 - *Intervalles de sauvegarde et exécution de l'application.*

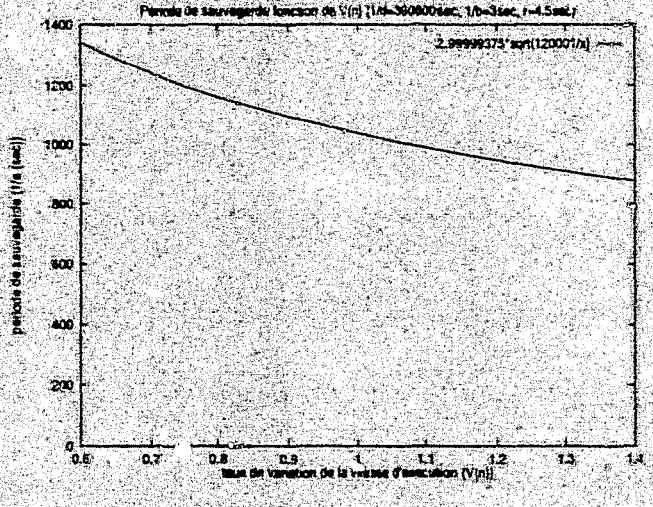


FIG. 3.9 - *Période de sauvegarde en fonction du taux moyen de variation de la vitesse d'exécution de l'application V(n) (1/d=360000sec, 1/b=3sec, r=4.5sec).*

Jusqu'ici, seule la défaillance du nœud supportant le module maître est prise en compte, car c'est uniquement lors de la défaillance de ce nœud qu'un recouvrement total de l'application est effectué. En revanche, la défaillance d'un nœud supportant un esclave entraine le recouvrement de ce dernier uniquement. La défaillance des esclaves peut également influencer la période de sauvegarde mais d'une manière négligeable par rapport à la défaillance du maître. Par conséquent, nous avons choisi d'omettre ce paramètre dans la détermination de la période de sauvegarde.

Hormis la sauvegarde automatique avec une période adaptative, d'autres facilités sont fournies à l'utilisateur afin de pouvoir gérer les sauvegardes explicitement. Ainsi, la période adaptative peut être remplacée par une période statique définie par l'utilisateur ou les sauvegardes automatiques peuvent être inhibées. Dans ce cas, les opérations de sauvegarde sont effectuées sur requête de l'application en invoquant explicitement le mécanisme (voir annexe A).

3.4 Conclusion

L'algorithme de sauvegarde étudié et évalué dans le chapitre 2 est caractérisé par sa simplicité, son efficacité et son moindre coût en terme de temps et d'espace à la différence des algorithmes classiques. De plus, il contribue à la résolution du problème de l'hétérogénéité matérielle et permet la redistribution de la charge au moment de la reprise.

La plupart des algorithmes de sauvegarde/reprise existants ne sont pas utilisés dans les systèmes réels. Par conséquent, l'aspect détection et recouvrement des défaillances n'est pas discuté. Notre approche de tolérance aux fautes utilise des outils simples, pratiques et efficaces pour traiter le problème de défaillances dans un système distribué. Malgré le fait que le modèle semble simple, il évite des surcoûts importants induits par des mécanismes complexes tels que la réplication et la diffusion. Notre réalisation constitue une étude de cas pour un système distribué, tolérant aux fautes, pouvant être intégré dans n'importe quel réseau de stations avec un coût minimal. Notre approche est caractérisée par les avantages suivants:

- un maximum de transparence dans la détection et la gestion des défaillances;
- une politique de recouvrement efficace, permettant la reprise des composants défaillants de l'application uniquement;
- une interface de programmation fournissant une méthodologie de programmation distribuée simple et efficace qui constitue une plateforme pour l'ordonnancement.

La partie suivante sera consacrée au problème d'ordonnancement de ces applications sur ces plateformes hétérogènes.

Deuxième partie

Ordonnancement

Chapitre 4

Modèle de construction et d'ordonnancement d'applications adaptatives

Dans la partie précédente, nous avons traité l'aspect tolérance aux fautes dans l'environnement parallèle adaptatif MARS. Cette partie est dédiée aux problèmes d'ordonnancement et de mise en œuvre d'applications parallèles dans MARS. Ce chapitre est consacré à la présentation d'un modèle d'ordonnancement mono-application et de mise en œuvre d'applications adaptatives. Dans le chapitre suivant, nous abordons le problème d'ordonnancement de plusieurs applications parallèles. Ce chapitre est organisé comme suit: la section 4.1 présente une synthèse des systèmes d'ordonnancement. Dans la section 4.2, nous passons en revue le modèle d'application adopté et l'algorithme d'ordonnancement mis en œuvre. Finalement, nous présentons les résultats de l'évaluation des performances de notre approche.

4.1 Algorithmes d'ordonnancement

Plusieurs classifications d'algorithmes d'ordonnancement dans les systèmes parallèles et distribués ont été proposées dans la littérature [Cas81, WM85, CK88, Tur93, Tal93, Rot94, BF96]. La prolifération des techniques d'ordonnancement a engendré une terminologie multiple et non cohérente. Cette situation accentue la difficulté à établir une taxonomie d'approches d'ordonnancement et complique à la fois leur comparaison et leur évaluation [CK88]. Néanmoins, on distingue deux grandes classes de techniques: temporelles et spatiales (figure 4.1).

4.1.1 Ordonnancement temporel et spatial

L'ordonnancement temporel est connu également sous le nom d'*ordonnancement* ou d'*ordonnancement local*. Il consiste à allouer, dans le temps, le processeur aux processus dans un système centralisé. Dans le cas de processus batch classiques, cela revient à assurer le séquencement des jobs. Si l'ordonnancement est préemptif, la tâche de l'ordonnanceur consiste à allouer les processus aux *"quantums"* de temps

programmés. On parle alors d'un ordonnancement temporel préemptif. Dans le cas des systèmes de type batch, l'ordonnancement est dit non préemptif. La plupart des systèmes dérivés d'Unix sont des exemples d'ordonnanceurs temporels [CD73].

L'ordonnancement spatial est également qualifié d'*ordonnancement global*, de *placement* ou d'*allocation*. Cette catégorie d'ordonnanceurs a pour rôle l'affectation des processus, d'une ou de plusieurs applications parallèles, aux nœuds d'un système parallèle ou distribué. Ce type d'ordonnancement peut également être préemptif ou non, suivant que les processus placés peuvent changer de nœud dynamiquement ou non [BF96]. En résumé, l'ordonnancement spatial décide de l'affectation des processus aux nœuds du système distribué et l'ordonnancement temporel des instants de leur exécution sur les nœuds choisis [CK88]. Les algorithmes d'ordonnancement spatial sont à leur tour divisés en deux grandes catégories : statique et dynamique.

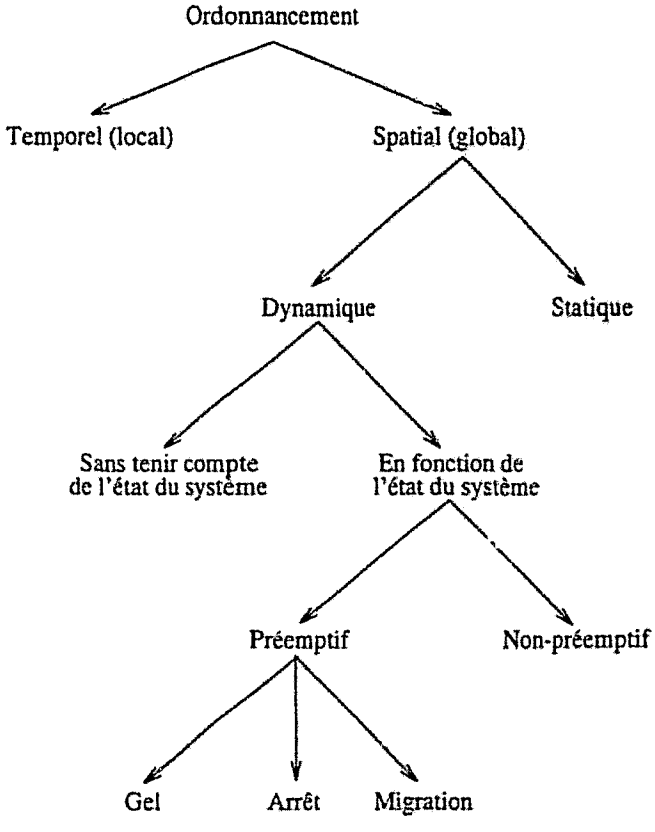


FIG. 4.1 - *Ordonnancement dans les systèmes parallèles et distribués.*

Dans les systèmes d'ordonnancement statique, les décisions d'ordonnancement sont prises avant le lancement des applications et demeurent inchangées jusqu'à leur terminaison. En effet, l'ordonnanceur dispose des caractéristiques de l'architecture cible et d'informations assez précises sur les applications prises en charge. Ce type d'ordonnancement est bien adapté aux environnements dédiés dans lesquels, la

charge n'évolue pas d'une manière imprévisible [ST85, CA82, PF89, Tal93].

En revanche, les systèmes d'ordonnancement dynamique ne disposent pas d'informations sur l'application avant son lancement et ne peuvent prévoir son comportement ni l'évolution de l'état de charge de l'architecture cible. Les décisions d'ordonnancement sont effectuées dynamiquement au moment de la création des processus ou pendant leur exécution, en se basant généralement sur des données dynamiques représentant l'état de charge du système. Par la suite, nous nous intéressons à l'ordonnancement dynamique dans les systèmes distribués.

4.1.2 Ordonnancement dynamique

Les politiques d'ordonnancement visent généralement à optimiser l'utilisation des ressources et à améliorer les performances des applications. La mise en place d'une approche d'ordonnancement doit tenir compte des deux aspects suivants [CK88] :

- *performance :* qui évalue l'apport de l'approche tant à l'utilisateur (satisfaction de ses besoins, temps de réponse, etc) qu'à l'exploitation des ressources ;
- *efficacité :* qui évalue le surcoût induit par l'approche pour l'accès aux ressources.

Les politiques d'ordonnancement dynamique diffèrent selon que les décisions d'ordonnancement utilisent les informations d'état du système ou non.

4.1.2.1 Ordonnancement aveugle

Dans ce type de systèmes, la détermination des nœuds d'exécution est effectuée au moment de la création des processus, et ceci sans tenir compte de l'état du système (charge, utilisateur, etc). PVM [GBD+94] est un exemple de ce type d'environnements. Dans PVM, les processus peuvent être créés pendant l'exécution de l'application et leur placement est effectué d'une manière cyclique sans tenir compte de la charge du nœud cible. Parmi les systèmes adoptant ce type d'ordonnancement, figurent LANDA [Mon96], Isis [Bir96], Linda [CG89], etc.

4.1.2.2 Ordonnancement en fonction de l'état du système

Les systèmes de cette classe effectuent les décisions d'ordonnancement en se basant sur l'état de charge courant des nœuds d'accueil. La décision peut être prise à la création des processus uniquement (ordonnancement non-préemptif) ou pendant leur exécution (ordonnancement préemptif).

Ordonnancement préemptif : les approches préemptives prévoient la migration des processus au cours de leur exécution, suivant l'évolution de l'état du système (GatoStar [Fol92], Sprite [OCD+88], Mach [ABB+86], Condor [LLM88], MIST

[CCG+95]. LoadLeveler [Sup94], Codine [Sof95]). L'ordonnancement temporel peut être également préemptif (voir section précédente). Certains auteurs utilisent également le terme *placement* pour l'ordonnancement spatial non préemptif et *placement et migration* pour l'ordonnancement spatial préemptif [BF96].

La préemption peut être réalisée de diverses manières:

- la technique la plus utilisée est la *migration de processus* qui consiste à arrêter le processus sur la machine d'origine, transférer son espace d'adressage sur la machine d'accueil et reconstruire son état à partir des données transférées [MDP+96, CCK+95]. Cette technique bien qu'efficace, est difficile à réaliser et requiert la compatibilité architecturale des nœuds d'origine et d'accueil;
- les systèmes spécialisés dans la détection et l'exploitation des cycles libres des machines sur les réseaux de stations optent généralement pour l'*arrêt du processus*. Ce choix peut être justifié par la disponibilité de ressources qui n'est pas certaine et par les contraintes imposées par la migration classique. L'arrêt de processus peut être effectué de deux manières différentes: sans récupération des résultats du travail déjà effectué (Far [GMW93], Godzilla [Cra90] et Worm [SH82]), ou avec récupération du travail réalisé (Condor [LLM88, TL95], CARMI [PL95, PL96], GLUnix [GPR+90]). La récupération du travail dans ces systèmes est généralement effectuée par la sauvegarde de l'espace d'adressage du processus, ce qui accentue les contraintes architecturales au niveau de la reprise du processus. Les systèmes effectuant cette récupération au niveau applicatif, tels que MARS, offrent un support d'hétérogénéité implicite;
- d'autres systèmes choisissent de *geler* le processus (DQS [Gre95], PARFORM [CS92], Batch [Dep94], etc). Ce choix peut gêner les utilisateurs locaux du fait qu'une partie des ressources du nœud est toujours utilisée (mémoire, etc).

Ordonnancement non-préemptif . dans les approches non-préemptives, les décisions de placement sont prises à la création des processus et demeurent inchangées (Utopia [ZZWD92], Amoeba [MRT+90], NEST [Ezz86], Spawn [WHH+92], VaxClusters [KLS86]).

Cette classification n'a pas évoqué explicitement la manière dont les informations d'état sont collectées et dont l'état du système est estimé. Rotithor [Rot94] suggère que les deux aspects (estimation de l'état du système et décision d'ordonnancement) soient traités de façon orthogonale.

A- Estimation de l'état du système : cette composante détermine la manière dont les informations d'état sont collectées (nombre de sites impliqués, instants de collecte, etc). L'information disséminée est déterminée sur la base des indicateurs de charge: longueur des différentes files d'attente du processeur [Kun91, ZF87], quantité de mémoire libre [SS84], charge du réseau, etc. La collecte peut être *centralisée* (Sprite [DO87]), *distribuée* (V-System [TLC85]) ou *hybride* (Utopia [ZZWD92]).

L'information échangée peut être *complète*, *partielle* ou *variable* [Sta85, SK90]. La dissémination peut être *volontaire* [SC89], *involontaire* [TL88] ou *composée* [RSZ89, FYN88], et enfin la collecte peut être *périodique* [Sta85], *apériodique* [SK90] ou *combinée* [RSZ89].

B- Décision d'ordonnancement : la décision d'ordonnancement concerne la détermination des nœuds d'accueil des processus et le choix des processus à transférer (éligibilité). La décision peut être *centralisée*, *distribuée* ou *hybride*. Le transfert des processus peut être initié par le nœud contenant la surcharge à transférer (*client*) [ZZWD92], par le nœud d'accueil (*serveur*) [ELZ86] ou par les deux (*symétrique*) [SK90].

En dépit de cette classification, les systèmes d'ordonnancement dynamique peuvent être distingués selon les algorithmes d'ordonnancement qu'ils utilisent: régulation de charge [CA82, NH85, PP83, Zho88, Sve90], enchères [Smi80], etc.

4.1.3 Ordonnancement adaptatif

Certaines propriétés des réseaux de stations et des clusters de processeurs accentuent la complexité de l'exploitation de ressources: multi-utilisateur, hétérogénéité matérielle et logicielle, discontinuité des cycles libres des nœuds, variation dynamique de la charge, utilisation interactive des nœuds, etc. L'ordonnancement d'applications dans ces environnements doit tenir compte de la charge des nœuds et du caractère personnel des stations. Dans la thèse de Hafidi [Haf98], une classification basée sur l'*adaptabilité* de l'application parallèle à la fluctuation de charge et à l'utilisation interactive des machines a été établie. Ainsi, les systèmes d'ordonnancement peuvent être classés en trois classes selon que le nombre et/ou la localisation des processus dépendent on non de l'état du système:

- *systèmes non-adaptatifs :* dans ces systèmes, le nombre et la localisation des processus de l'application sont définis en dehors de toute information d'état de charge et d'utilisation interactive des nœuds;
- *systèmes semi-adaptatifs :* dans les systèmes semi-adaptatifs, les nœuds d'accueil des processus sont déterminés dynamiquement en fonction de l'état de charge du système, tandis que le nombre de processus (degré de parallélisme de l'application) ne dépend pas de l'état du système;
- *systèmes adaptatifs :* dans cette classe d'ordonnanceurs, la localisation et le nombre de processus de l'application sont déterminés suivant l'état de charge et l'utilisation interactive des nœuds d'accueil. L'exploitation de ressources dans les systèmes adaptatifs consiste à détecter dynamiquement les nœuds libres et à les exploiter en y plaçant de nouveaux processus applicatifs. L'indisponibilité des nœuds impose l'arrêt des processus placés sur ces nœuds. Les systèmes adaptatifs les plus connus sont: Piranha [Kam94], CARMI [PL96] et MARS [Haf98].

Les environnements adaptatifs peuvent être distingués selon la manière dont l'arrêt de processus est effectué. Une première catégorie de systèmes prévoit de récupérer le travail effectué par le processus avant sa terminaison et une deuxième se contente d'arrêter le processus. Les systèmes récupérant le travail au niveau applicatif offrent un support implicite pour l'hétérogénéité matérielle, grâce à la possibilité de déplacement des calculs dans des environnements hétérogènes.

En plus de l'hétérogénéité du point de vue architecture, un autre aspect peut distinguer les systèmes d'ordonnancement adaptatifs. Il s'agit de l'hétérogénéité du point de vue puissance relative des nœuds. En effet, les systèmes prenant en compte ce paramètre dans les décisions d'ordonnancement diffèrent de ceux qui traitent tous les nœuds d'une manière équivalente. Les plateformes que nous considérons (réseaux de stations et clusters de processeurs en particulier) se caractérisent par des taux d'hétérogénéité très importants. L'introduction de cet aspect dans les décisions d'ordonnancement est susceptible d'améliorer considérablement l'exploitation des ressources dans ces environnements.

4.2 Construction d'une application adaptative

De nombreux modèles d'applications parallèles utilisant des algorithmes d'ordonnancement basés sur le modèle de graphes de dépendances ont été proposés et évalués. La plupart de ces algorithmes font des hypothèses sur la structure de l'application, sur le temps d'exécution ou ne considèrent pas le problème de l'hétérogénéité [NS88, BL94, Kam94].

Dans Piranha/Linda [Kam94], l'application est composée de tâches (fragments de données) dont l'ordre d'exécution est assuré par un graphe de dépendances. Ils considèrent quelques types de graphes uniquement. Ainsi, la cohérence du graphe est vérifiée avant le lancement de l'application et des optimisations d'allocation peuvent être effectuées sur la base du graphe de dépendances.

Notre objectif est principalement de :

- définir un modèle d'application parallèle permettant la description aisée d'une application adaptative MARS et fournir une interface de programmation à cet effet ;
- développer un ordonnanceur qui se charge d'optimiser l'exécution des applications parallèles adaptatives.

La figure 4.2 présente un modèle pour la construction et l'ordonnancement d'applications parallèles adaptatives. L'application parallèle peut être vue comme une collection de tâches séquentielles dont l'ordre d'exécution est assuré par un graphe de précédence. Les tâches ainsi générées sont passées à un ordonnanceur de l'application qui se charge de leur allocation sur un ensemble de nœuds mis à la disposition de l'application (configuration matérielle).

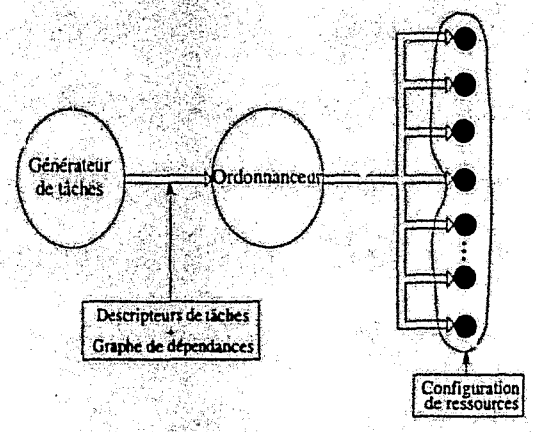


FIG. 4.2 - *Modèle d'ordonnancement dynamique.*

4.2.1 Tâche, application parallèle et environnement adaptatif

MARS est basé sur un modèle dirigé par les données. Une tâche est définie comme étant un fragment de données sur lesquelles, un ensemble d'actions séquentielles doivent être appliquées. L'application parallèle est définie comme étant une succession de tâches dont les contraintes de précédence sont décrites par un graphe de dépendances (*DAG, Directed Acyclic Graph*). Ce modèle vérifie la propriété d'être simple, facilitant ainsi le développement de l'application. En effet, le programmeur doit se soucier uniquement du travail que réalise l'application sur les données du problème. La figure 4.3 illustre la structure du modèle implémenté sur MARS. La gestion de la configuration (ajout et retrait de nœuds) est assurée par un composant externe dont l'étude sera présentée au chapitre 5.

4.2.2 Générateur de tâches

Développer une application suivant ce modèle consiste à partitionner le travail à faire en un ensemble de tâches. L'application est construite en spécifiant le graphe de précédence entre les tâches. En résumé, nous faisons les hypothèses suivantes :

H 4.1 ***Irrégularité des applications*** *: la structure du graphe de dépendances est arbitraire et les durées d'exécution des tâches ne sont pas connues a priori.*

H 4.2 ***Construction dynamique de l'application*** *: le graphe de dépendances peut ne pas être connu entièrement au lancement de l'application.*

Au niveau du maître, le programmeur spécifie les tâches et utilise l'interface de programmation pour les mentionner. La bibliothèque construit ainsi un *espace de travail* et un *graphe de dépendances* qui se chargera d'assurer l'ordre d'exécution des

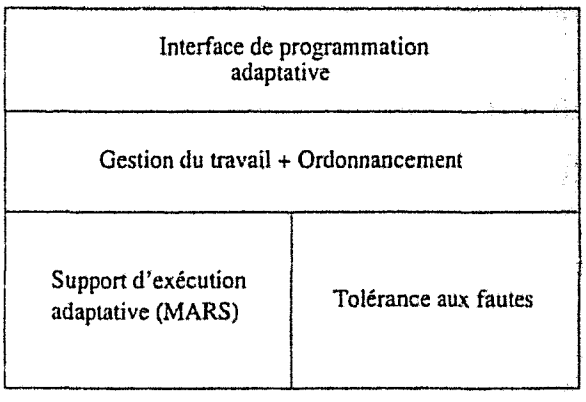


FIG. 4.3 - *Implémentation du modèle.*

tâches au moment de leur allocation (figure 4.4). Ce modèle dirigé par les données présente une solution implicite pour la migration hétérogène.

L'allocation des tâches aux esclaves est effectuée automatiquement par un module de la bibliothèque (*serveur de travail*) en se basant sur les décisions de l'ordonnanceur. De nouveau, l'utilisateur peut intervenir lors de la réception du travail intermédiaire (tâches traitées partiellement) et des résultats pour une éventuelle optimisation de l'espace de stockage ou pour la génération de nouvelles tâches en ligne. La construction dynamique de l'application permet de prendre en charge de nombreuses classes d'applications irrégulières (Branch & Bound, etc).

Le schéma d'exécution de l'esclave consiste généralement à itérer un certain nombre d'actions (figure 4.4) :

1. *demande de travail :* elle est effectuée par un appel à la bibliothèque, interprété par une requête adressée au *serveur de travail.* Ce dernier cherche une tâche prête dans l'espace du travail et l'expédie à l'esclave demandeur avec le nom du thread destiné à son traitement[1] ;

2. *retour de résultats :* à la réception de la tâche, le thread approprié est lancé. Lorsqu'il a fini le traitement, il utilise une fonction de la bibliothèque afin de remettre les résultats au maître ;

3. *retour de travail partiel :* lorsque le système décide de *replier* l'application, l'esclave concerné retourne la tâche partiellement traitée (résultats partiels et travail restant) au maître. L'état partiel de la tâche est construit par une fonction définie par l'utilisateur et son retour est assuré par la bibliothèque.

1. Nous utilisons une table de correspondance entre les threads et les noms de fonctions correspondantes.

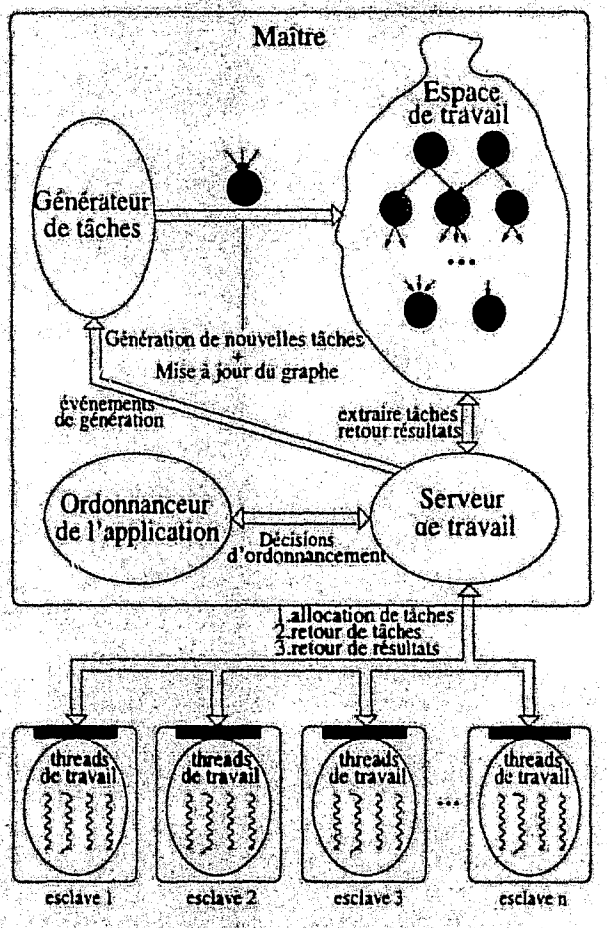


FIG. 4.4 - *Génération et allocation de tâches.*

4.3 Ordonnancement d'applications adaptatives

Notre approche d'ordonnancement est basée sur une extension de l'ordonnancement, basé sur les graphes de dépendances, aux systèmes adaptatifs. Les algorithmes de base de cette classe ont été largement étudiés et évalués [NS88, BL94, Kam94, SL93, IÖ98, MS98, NS93, KA99]. Dans Piranha, le graphe de dépendances est connu avant le lancement de l'application. L'ordonnancement est basé sur des heuristiques spécifiques aux types de graphes utilisés. De plus, le critère d'hétérogénéité n'est pas pris en compte dans les décisions d'ordonnancement.

Le modèle d'application parallèle que nous avons adopté permet à l'ordonnanceur de disposer de certaines caractéristiques concernant l'application parallèle: tâches prêtes à l'exécution, graphe de dépendances, etc. L'optimisation de l'allocation des tâches est alors possible. Les motivations pour l'ordonnancement à ce niveau peuvent être résumées par les deux points suivants :

- l'application parallèle est décrite par la succession d'opérations sur les tâches (graphe de dépendances). L'apport de l'ordonnancement par graphe de dépendances a été montré dans les algorithmes classiques [KA99] ;

- effectuer un ordonnancement à ce niveau dans les environnements adaptatifs est utile dans de nombreuses situations induites par l'hétérogénéité en particulier. Ainsi, les situations, où des tâches sont exécutées sur des nœuds lents pendant que des nœuds plus puissants sont libres, seront évitées.

L'approche d'ordonnancement adoptée consiste à minimiser le temps d'exécution de l'application en essayant de générer le maximum de travail continuellement. En effet, dans certaines situations, des nœuds alloués à l'application sont bloqués en attente de la disponibilité du travail qui doit être généré par d'autres nœuds moins puissants. Deux paramètres sont utilisés : la structure du graphe de dépendances et l'aspect hétérogénéité du point de vue puissance des processeurs.

La figure 4.5 illustre un exemple simple d'un graphe de dépendances d'une application. Dans un environnement hétérogène, il serait plus judicieux d'exécuter la tâche T_2 sur le nœud le plus puissant afin de débloquer le maximum de tâches.

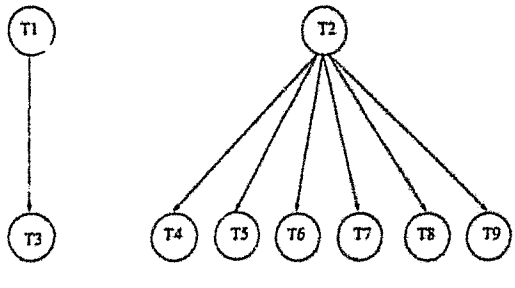


FIG. 4.5 - *Exemple de graphe de dépendances.*

4.3.1 Approche utilisée

En l'absence de toute information sur le coût d'exécution des tâches, nous utilisons une heuristique qui vise à maximiser le nombre de tâches prêtes à tout instant. La terminaison d'une tâche est susceptible de *signaler*[2] toutes les tâches en attente de sa terminaison. Par conséquent, la priorité d'une tâche doit être équivalente au nombre de tâches qui lui sont dépendantes.

$$Prio_t(T_i) = d^+_{G_t}(T_i) \tag{4.1}$$

Où G_t est le graphe de dépendances associé à l'application et comportant toutes les tâches insérées jusqu'à l'instant t ; $d^+(T_i)$ est le degré sortant du nœud T_i (figure 4.6). Une solution plus efficace serait de maximiser la priorité des tâches appartenant aux chemins critiques comme dans la plupart des techniques statiques. D'après

2. Signaler une tâche signifie la rendre prête.

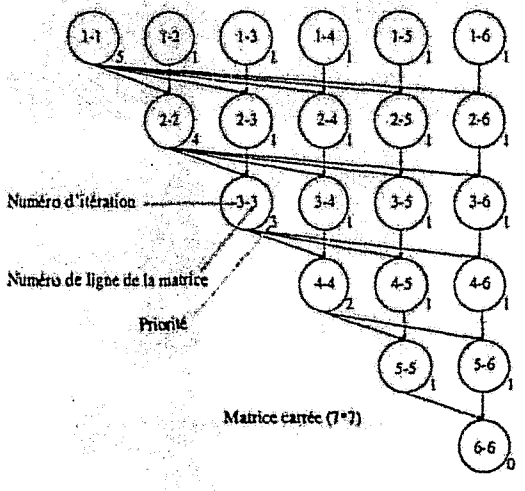


FIG. 4.6 - *Graphe de dépendances de l'application Gauss.*

l'hypothèse 4.2, cela est impossible car le graphe de dépendances n'est pas connu a priori.

D'autre part, les nœuds sont caractérisés par la puissance de calcul $PL_t(N_i)$ qu'ils mettent à la disposition de l'application à l'instant t. Cette puissance de calcul peut être définie par:

$$PL_t(N_i) = \frac{P(N_i)}{L_t(N_i)} \tag{4.2}$$

N_i: représente le nœud i de la configuration.
$P(N_i)$: puissance relative du nœud i.
t: temps, qui correspond aux instants d'invocation de l'ordonnanceur.
$L_t(N_i)$: charge du nœud i à l'instant t.

Par la suite, nous noterons PL_t par PL. Le principe de l'algorithme d'ordonnancement est d'exécuter les tâches de plus haute priorité sur les nœuds libres les plus puissants. L'algorithme d'ordonnancement consiste à optimiser la première allocation au lancement de l'application. Durant son exécution, si un déséquilibre apparaît, un réajustement dynamique de l'allocation doit être effectué. Les structures de données utilisées par l'algorithme sont:

WorkSpace: espace de tâches indexé.
ReadyTasks: liste des tâches prêtes à l'exécution.
AllocatedTasks: liste des tâches en exécution.

4.3.2 Allocation initiale

A son lancement, l'application obtient un ensemble de nœuds sur lesquel le lance des esclaves. L'ordonnanceur commence par trier les nœuds ainsi allou i-

vant l'ordre décroissant de leur PL et les tâches prêtes suivant leur priorité décroissante également. Puis, chaque tâche est associée à un nœud, dans la limite des nœuds et des tâches également, de telle sorte que les tâches de plus haute priorité sont réservées aux nœuds les plus puissants

1. trier l'ensemble des nœuds obtenus initialement C_{t_0} dans l'ordre décroissant de leur PL ;

2. insérer les tâches prêtes dans la file $ReadyTasks$ dans l'ordre décroissant de leur priorité actuelle ($Prio_{t_0}(T_i)$).

Lorsqu'un esclave W_i réclame du travail, le *serveur de travail* transmet la requête à l'ordonnanceur qui détermine la tâche associée à cet esclave et l'alloue. La tâche est alors transférée de la liste des tâches prêtes $ReadyTasks$ à celle des tâches allouées $AllocatedTasks$.

4.3.3 Ajustement dynamique

L'allocation initiale répond aux objectifs de notre approche d'ordonnancement. Durant le cycle de vie de l'application, quelques tâches sont insérées, d'autres terminent ou le nombre d'esclaves change. Certains événements peuvent provoquer un déséquilibre, ce qui impose une réaction de l'ordonnanceur afin de réajuster l'allocation[3]. Les événements peuvent être liés au comportement de l'application. Ils sont au nombre de deux :

E 4.1 *terminaison d'une tâche ;*

E 4.2 *passage d'un ensemble de tâches à l'état prêt (signalement de tâches).*

Il existe d'autres événements générés par l'environnement sous-jacent et qui concernent l'application. Ces événements sont appelés événements systèmes :

E 4.3 *retrait d'un nœud (repli) ;*

E 4.4 *rajout d'un nœud (dépli).*

Du point de vue de l'ordonnanceur, le *repli* et le *signalement de tâche* sont similaires. De la même manière, les événements *terminaison de tâche* et *dépli de l'application* sont similaires car ils créditent l'application d'un nœud libre additionnel. En résumé, trois situations peuvent se présenter à l'ordonnanceur : nouvelle tâche prête, allocation d'une tâche et terminaison d'une tâche.

3. La cohérence de l'application est préservée par le serveur de travail.

4.3.3.1 Nouvelle tâche prête

Une tâche passant à l'état prêt peut être une tâche nouvellement signalée ou une tâche partiellement traitée retournée par un esclave. Lorsqu'une nouvelle tâche est signalée, l'ordonnanceur l'insère dans la liste des tâches prêtes (*insert_ready*).

Lorsqu'un esclave W_r retourne une tâche incomplète T_i au maître suite à un événement quelconque, l'ordonnanceur réagit comme suit :

- il déplace la tâche de la liste des tâches allouées à celles des tâches prêtes (*move_alloc_ready*) ;
- il range les résultats partiels et le travail restant dans l'espace de travail ;
- il appelle ensuite la fonction utilisateur associée si elle est définie.

Transférer une tâche de la liste des *tâches allouées* à celle des *tâches prêtes* et insérer une nouvelle tâche T_i dans la file des *tâches prêtes* implique l'exécution d'autres actions. En effet, une nouvelle tâche prête doit être prise en charge :

- l'ordonnanceur cherche un nœud libre en attente de disponibilité de travail ($\{ N_i \in C_i : PL(N_i) = \max_{N_k \in C_i \wedge state(N_k)=WAITING} PL(N_k) \}$) ;
- si l'ensemble ainsi construit n'est pas vide, un nœud est choisi et est débloqué ;
- en revanche, s'il n'y a pas de nœuds libres, un ajustement de l'allocation basé sur les priorités doit être entrepris. L'ordonnanceur désalloue la tâche la moins prioritaire parmi les tâches allouées, si sa priorité est plus petite que celle de la tâche retournée T_i d'un facteur de C_p[4] ($\{ T_j \in AllocatedTasks : Prio_i(T_j) = \min_{T_k \in AllocatedTasks \wedge Prio_i(T_i) \geq C_p \times Prio_i(T_k)} Prio_i(T_k) \}$).

  Dans ce cas, l'application est en déficit. L'ordonnanceur vérifie s'il doit procéder à la demande de nœuds additionnels auprès du système. Notons que la décision la plus appropriée consiste à choisir le nœud le plus puissant dans la limite de la priorité de T_i. Cette solution est difficile à calculer et peut provoquer une succession de remplacements susceptibles de nuire aux performances du système (forte instabilité).

4.3.3.2 Allocation de tâche

Lorsqu'un esclave W_i s'exécutant sur le nœud N_i réclame du travail, l'ordonnanceur cherche à lui allouer une tâche comme suit :

- si la liste des *tâches prêtes* n'est pas vide, l'ordonnanceur vérifie s'il y a des tâches allouées plus prioritaires que toutes les tâches prêtes et s'exécutant sur

4. C_p est une constante sur la priorité supposée refléter le coût de retrait d'une tâche.

des nœuds moins puissants que N_i. Dans ce cas, cette tâche allouée est retirée dans le but de l'allouer sur N_i à la prochaine étape. La tâche prête de plus haute priorité est donnée par:

$$R = \{T_j \in ReadyTasks : Prio_t(T_j) = \max_{T_k \in ReadyTasks} Prio_t(T_k)\}.$$

T_{rmax} est une des tâches que contient l'ensemble R s'il n'est pas vide.

Le choix de la tâche allouée à retirer est effectué en construisant l'ensemble des tâches allouées plus prioritaires que T_{rmax} d'un facteur de C_p et traitées chacune par un esclave s'exécutant sur un nœud moins puissant que N_i d'un facteur de C_{PL}[5].

$$A = \{T_j \in AllocatedTasks : Prio_t(T_j) = \max_{cnts} Prio_t(T_k)\}.$$

L'expression *cnts* est donnée par: $(T_k \in AllocatedTasks \wedge Prio_t(T_k) \geq C_p \times Prio_t(T_{rmax}) \wedge PL(N_{T_k}) \leq C_{PL} \times PL(N_i))$. N_{T_k} est le nœud d'exécution de l'esclave traitant la tâche T_k.

Si l'ensemble A n'est pas vide, une de ses tâches est retirée dans le but de l'allouer sur N_i à la prochaine étape. Dans le cas contraire, la tâche prête T_{rmax} est allouée;

- en revanche, s'il n'y a pas de *tâches prêtes*, l'ordonnanceur vérifie si un déséquilibre significatif est enregistré. Dans ce cas, un réajustement doit être effectué. De nouveau, le facteur C_{PL} est utilisé pour construire l'ensemble des *tâches allouées* sur des nœuds moins puissants que N_i d'un facteur de C_{PL}.

$$A = \{T_j \in AllocatedTasks \ : \ Prio_t(T_j) = \max_{cnts} Prio_t(T_k)\}.$$

L'expression *cnts* est donnée par: $T_k \in AllocatedTasks \wedge PL(N_{T_k}) \leq C_{PL} \times PL(N_i))$. N_{T_k} est le nœud sur lequel, la tâche T_k est traitée. Si l'ensemble A ainsi construit contient quelques tâches, une d'entre elles (en l'occurence celle traitée sur le nœud le moins puissant) est retirée afin d'être allouée sur N_i à l'étape suivante.

Les deux caractéristiques principales de cette étape de l'algorithme d'ordonnancement sont l'allocation de la tâche prête de plus haute priorité et le réajustement en cas de disponibilité de nœuds puissants. Si un esclave ne peut avoir du travail, il est gardé en attente pour une durée de temps prédéfinie avant de libérer le nœud qui pourra être exploité par d'autres applications.

5. C_{PL} est une constante sur la puissance relative des nœuds qui est supposée refléter le coût de permutation des tâches. C'est en quelque sorte le rapport minimum, entre le PL du nœud destinataire et le PL du nœud source de la tâche à transférer, nécessaire pour améliorer le temps d'exécution de la tâche.

4.3.3.3 Terminaison de tâche

Lorsqu'un esclave W_r a fini de traiter une tâche T_i, l'ordonnanceur exécute les actions suivantes :

- il supprime la tâche de la file des *tâches allouées* ;
- le graphe de dépendances est ensuite mis à jour. Cela permet de signaler éventuellement quelques tâches. Les tâches signalées sont insérées dans la liste des *tâches prêtes* (*insert_ready*) ;
- par la suite, la fonction utilisateur associée est appelée. Cette dernière peut générer d'autres tâches. L'espace de travail peut ainsi être construit durant l'exécution de l'application.

4.3.4 Priorités et dépendances

La priorité d'une tâche croît dynamiquement à chaque fois qu'une tâche dépendante d'elle est insérée ($Prio_t(T_i) = d^+_{G_t}(T_i)$). Durant son exécution, si la priorité d'une tâche augmente cela n'est pas pris en compte immédiatement même si les conditions de réajustement sont désormais vérifiées. En effet, cette nouvelle priorité est prise en compte à partir du prochain retrait éventuel de la tâche. Le temps d'exécution relativement faible des tâches et le surcoût que le réajustement induirait (vérifier la priorité de toutes les tâches à chaque insertion d'une nouvelle tâche) ont influencé notre choix.

En revanche, la dépendance d'une tâche décroît avec la terminaison des tâches dont elle dépend ($Deps_t(T_i) = d^-_{G_t}(T_i)$). Celle-ci devient prête au traitement lorsque cette valeur devient nulle.

4.3.5 Interaction de l'ordonnanceur de l'application avec le système

Mis à part les actions d'enrôlement et de terminaison, de demande initiale de ressources et à des fins de tolérance aux fautes (voir partie I), l'ordonnanceur applicatif interagit avec le système global pour lui communiquer l'état de l'application et ses besoins. Ces interactions sont résumées ci-dessous :

- lorsqu'une tâche additionnelle est insérée dans la liste des tâches prêtes (*insert_ready* et *move_alloc_ready*) et que l'application est en déficit de nœuds ;
- lorsqu'un esclave demande du travail et que la liste des tâches prêtes est vide. Dans ce cas, l'ordonnanceur de l'application demande au système de ne plus considérer l'application dans les nouvelles allocations de ressources jusqu'à nouvel ordre.

Les demandes de ressources ne sont pas adressées à chaque fois qu'une tâche est insérée dans la file des *tâches prêtes*, mais plutôt occasionnellement (si $NbReadyTasks \geq NbAllocatedTasks$).

4.4 Résultats expérimentaux

Cette section est consacrée à l'étude des performances de l'approche d'ordonnancement proposée. Nous présentons des résultats d'expérimentations conduites sur une application de l'élimination de Gauss par blocs (figure 4.6). L'application est exécutée sur un réseau local *Ethernet* $10Mbits/s$ de stations de travail où l'hétérogénéité, du point de vue puissance, est très importante (écart-type de la puissance relative des processeurs du parc d'environ 32).

La figure 4.7 trace le temps d'exécution de l'application en fonction du nombre total (normalisé) de nœuds exploités par l'application N_{base}. N_{base} est donné par :

$$N_{base} = \frac{\sum_{i=0}^{N-1} T_i \times PL_i}{T_p}. \tag{4.3}$$

N : nombre total de nœuds physiques utilisés par l'application.
T_i : temps total durant lequel, le nœud N_i est affecté à l'application.
PL_i : facteur PL du nœud N_i. Puisque la charge n'est pas considérée dans ces expérimentations, ce facteur représente uniquement la puissance relative du nœud.
T_p : temps d'exécution parallèle de l'application.

La figure montre qu'avec un nombre de nœuds relativement faible, les performances de la version itérative sans priorité sont équivalentes à celles de la version basée sur la priorité. Cela peut être expliqué par le surcoût induit par le calcul des priorités sans amélioration des performances. En revanche, la différence devient de plus en plus importante lorsque le nombre de nœuds augmente.

Chaque exécution génère plus de 6000 tâches de très court temps d'exécution (moins de 0.5 sec). Ces conditions ne sont pas favorables à notre environnement. Les préemptions additionnelles en vue de réajustement peuvent causer une dégradation des performances si elles sont excessivement utilisées. Le coût des communications ainsi engendrées ne peut être amorti immédiatement après la préemption mais plutôt à long terme si cette décision permet d'augmenter le nombre de tâches prêtes. La courte durée d'exécution des tâches provoque des changements rapides de l'allocation continuellement, ce qui influe sur la qualité des informations mises à la disposition de l'ordonnanceur.

Afin d'illustrer l'effet de l'heuristique utilisée pour le calcul des priorités, nous avons essayé de mettre en évidence le comportement des deux algorithmes en traçant l'état de l'application dans le temps. La figure 4.8 schématise le nombre de tâches signalées dans le temps (nombre de tâches allouées + nombre de tâches prêtes) pour

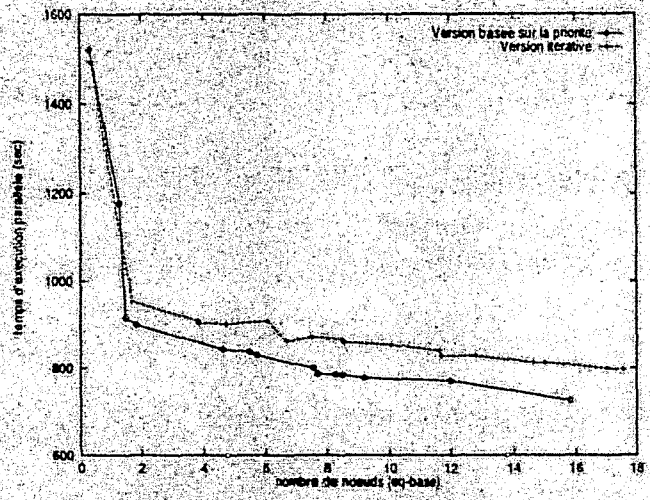
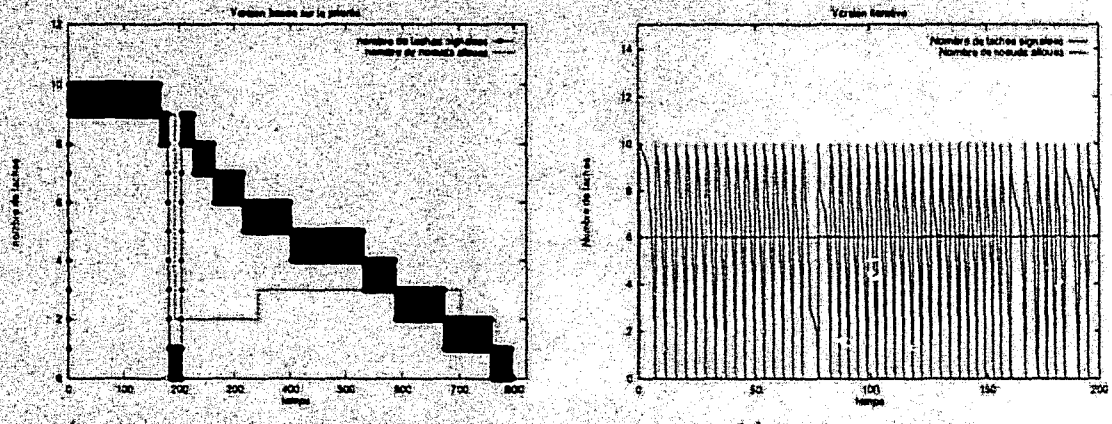


FIG. 4.7 - *Temps d'exécution de l'application en fonction du nombre de nœuds de base pour les deux ordonnanceurs (itératif et celui basé sur la priorité).*

les deux versions. Pour la version itérative, ce nombre varie entre 0 et le nombre maximum de tâches générées pendant l'étape (figure 4.8b). Par conséquent, les performances sont limitées par le nœud le plus lent de la configuration. En revanche, l'approche basée sur la priorité débloque le maximum de tâches à tout moment et permet ainsi de mieux exploiter les nœuds disponibles.



a) Ordonnancement basé sur la priorité b) Version itérative

FIG. 4.8 - *Nombre de tâches signalées dans le temps pour l'élimination de Gauss. 2740 tâches avec un maximum de 10 tâches signalées à un moment donné.*

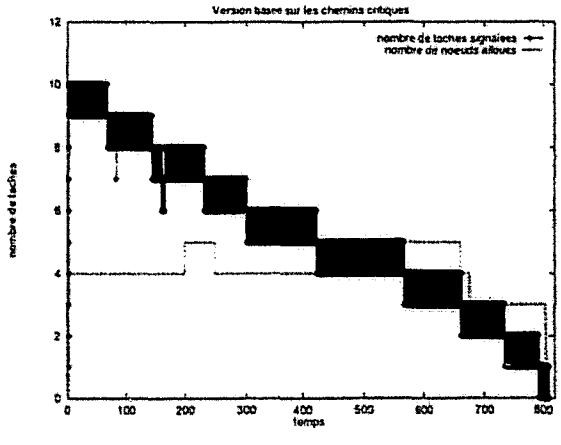
Malgré ces résultats, nous observons des moments où le nombre de tâches chute énormément (au temps 200 sur la figure 4.8a). Ces situations peuvent être le résultat

de scénarios dans lesquels les tâches, susceptibles de débloquer un grand nombre de tâches, dépendent elles-mêmes de tâches de faible priorité. Ceci fait que ces tâches sont débloquées tardivement. Sur la figure 4.6, si les tâches T_{11}, T_{16}, T_{15}, T_{14}, T_{13} et T_{12} sont allouées dans cet ordre sur des nœuds avec un PL décroissant, T_{22} sera vraisemblablement la dernière tâche débloquée à être allouée. A sa terminaison, le nombre de tâches signalées atteint 0 (toutes les tâches de la troisième itération doivent attendre la terminaison de T_{22} qui est retardée par une tâche de faible priorité T_{12}). Pour éviter de telles situations, la priorité de T_{22} doit être reflétée dans celle de T_{12}. Pour ce faire, nous pouvons adopter une approche basée sur les chemins critiques utilisés dans les méthodes statiques [SL93]. L'approche peut être étendue aux environnements dynamiques comme dans les travaux de [IÖ98, MS98]. La remarque principale concernant ces travaux est que les applications considérées sont composées d'un nombre modeste de tâches dont le coût d'exécution est relativement important et le nombre est connu avant l'exécution.

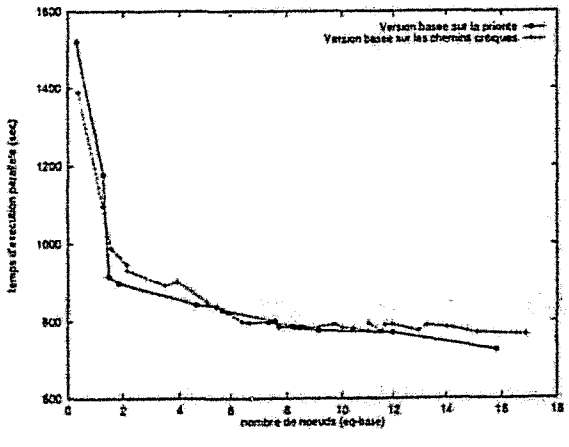
Dans cette approche, les priorités sont calculées récursivement. La priorité d'une tâche est équivalente au nombre de ses successeurs directs augmenté de la somme de leur priorité (formule 4.4).

$$Prio_t(T_i) = d^+_{G_t}(T_i) + \frac{\sum_{T_j \in S_i} Prio_t(T_j)}{C_l}. \tag{4.4}$$

Où $d^+_{G_t}(T_i)$ représente le nombre de successeurs directs de T_i et S_i l'ensemble des tâches dépendantes directement de T_i. C_l est une constante utilisée pour normaliser l'expression et reflète en quelque sorte l'importance de la priorité de T_j dans celle de T_i.



a) Nombre de tâches signalées dans le temps



b) Temps d'exécution de l'application en fonction du nombre de nœuds de base pour les deux versions (priorité simple et chemins critiques)

FIG. 4.9 - *Résultats de l'approche basée sur les chemins critiques.*

La figure 4.9a trace de nouveau l'évolution de l'application avec cette approche. Comme nous pouvons le constater, le problème est résolu et le nombre de tâches signalées ne chute plus (toujours supérieur à 0). La figure 4.9b illustre les performances de l'approche comparées à celles de l'approche basée sur la simple priorité. Les résultats montrent une très modeste amélioration et parfois même une dégradation des performances. Le surcoût induit par le calcul récursif des priorités (plus de 2000 tâches et 300 niveaux) et la difficulté de déterminer la constante de priorité C_p sont vraisemblablement les causes de ces mauvais résultats.

Toutes les expérimentations ont été conduites dans un environnement modérément chargé. Même si nous attendons des résultats similaires pour les situations de charge importantes, il est intéressant d'évaluer ces situations réellement. De plus, puisque la structure du graphe de dépendances affecte les performances, l'investigation de l'effet de ce paramètre est intéressante. La section suivante sera consacrée à l'étude de l'effet de ces paramètres sur les performances de notre approche d'ordonnancement.

4.5 Analyse de performances

L'élaboration d'un modèle analytique, reflétant avec fiabilité les différents paramètres de l'environnement et de l'application, est très complexe pour ces environnements. Nous utilisons des expérimentations sur des applications réelles. En effet, une application réelle, dont les différents paramètres (structure du graphe, coût d'exécution des tâches, charge, etc) peuvent être variés, est utilisée. Ceci nous permet une prédiction réelle des performances.

4.5.1 Caractérisation de l'application distribuée

Afin de pouvoir étudier différentes familles d'applications parallèles, Gelenbe [Gel89] a proposé un modèle pour quantifier la densité des graphes de dépendances. Si A est la matrice d'adjacence du graphe, alors $a(i,j) = 1$ si la tâche T_j est dépendante de T_i (l'arc $a(i,j)$ existe) et 0 sinon. Les $a(i,j)$ sont des variables aléatoires indépendantes définies par [AVAM92] comme suit :

$$\begin{aligned} P(a(i,j)=1) &= \pi & 0 \leq i < j \leq M-1. \\ P(a(i,j)=0) &= 1-\pi & 0 \leq i < j \leq M-1. \\ P(a(i,j)=0) &= 1 & 0 \leq j \leq i \leq M-1. \end{aligned}$$

M est le nombre total de tâches générées par l'application durant toute sa durée de vie. π est la loi de probabilité régissant la densité du graphe. La dernière condition est introduite pour garantir que le graphe soit acyclique.

4.5.2 Caractérisation de la configuration matérielle

La configuration matérielle consiste en un ensemble de stations hétérogènes connectées par un réseau local. Le parc se compose de N nœuds gérés par MARS. Par conséquent, notre application peut s'exécuter avec d'autres applications externes (charge externe dynamique) et en présence d'utilisateurs interactifs. Ces deux paramètres sont pris en compte dans l'attribution des ressources à l'application.

4.5.3 Résultats

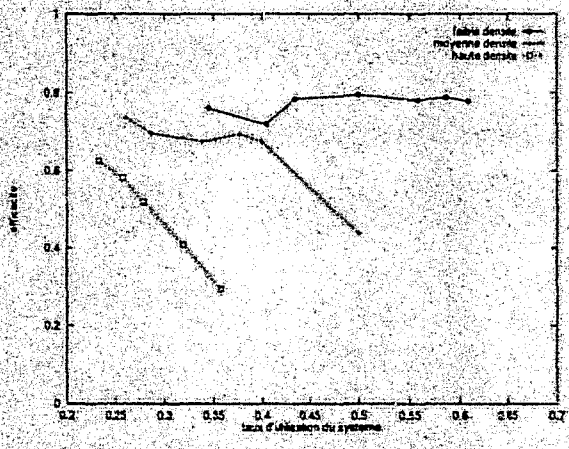
Suivant les valeurs de π, trois classes d'applications peuvent être identifiées:

- applications purement parallèles ($\pi = 0$);
- applications purement séquentielles ($\pi = 1$);
- applications réelles ($0 < \pi < 1$). L'application se compose d'un ensemble de tâches avec des contraintes de dépendances.

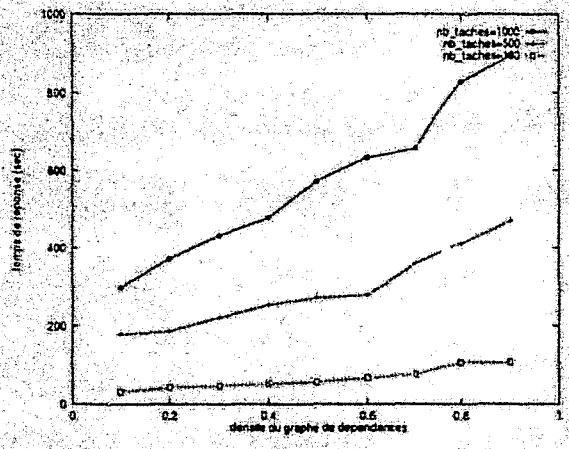
La figure 4.10a trace l'évolution de l'efficacité de l'application en fonction du taux d'utilisation du système pour trois types d'applications, caractérisés par une haute ($\pi = 0.8$), une moyenne ($\pi = 0.5$) et une basse ($\pi = 0.2$) densité du graphe de dépendances. Lorsque le taux d'utilisation est élevé (le système est chargé), l'ordonnancement d'applications peu parallèles provoque des durées d'attente importantes des esclaves. La libération/réacquisition de ressources se multiplie également. Ceci contribue à la baisse de l'efficacité de l'application. Pour le second type d'applications, ce phénomène est également valable mais la dégradation est moins importante, étant donné que le travail est plus abondant. Les applications hautement parallèles sont moins sensibles à la charge du système étant donné que le travail est souvent disponible et que les nœuds surchargés sont retirés à l'application qui utilisera le reste des nœuds.

La figure 4.10b illustre l'effet de la densité du graphe de dépendances π sur le temps de réponse de l'application. Pour chacune des trois applications, la quantité de travail est la même, seul π change. Pour une application de courte durée de vie comportant 100 tâches, l'effet de ce paramètre n'est pas très important étant donné que le temps de réponse de l'application est très court. En revanche, plus la durée de vie est importante, plus l'effet de ce paramètre est déterminant. Entre $\pi = 0.1$ et $\pi = 0.9$, le temps de réponse de l'application de 1000 tâches a triplé.

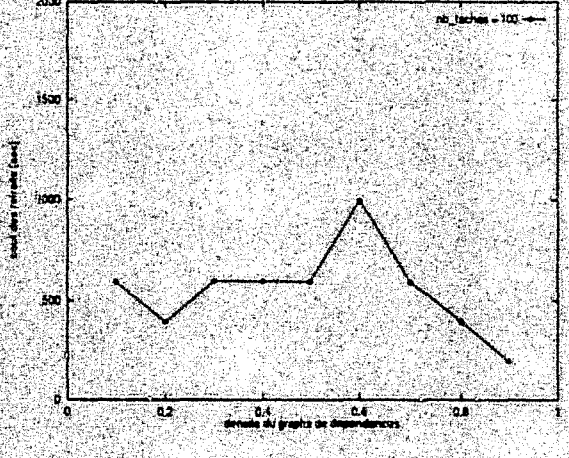
Etant donné que notre algorithme d'ordonnancement effectue des retraits suivant l'importance de la différence de priorité entre les tâches, nous avons essayé de mesurer le coût de ce mécanisme en fonction de la densité du graphe pour une application (figure 4.10c). Plus π est important, moins de travail est généré. Le retrait est par conséquent moins utilisé. Ce qui explique la tendance de ce paramètre à baisser sur la figure 4.10c.



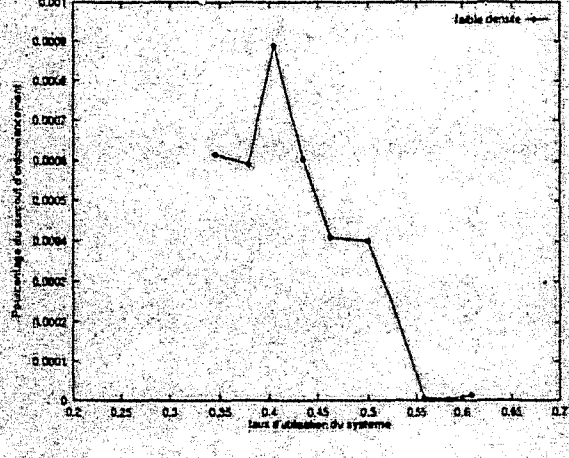
a) Efficacité de l'application en fonction du taux d'utilisation du système.



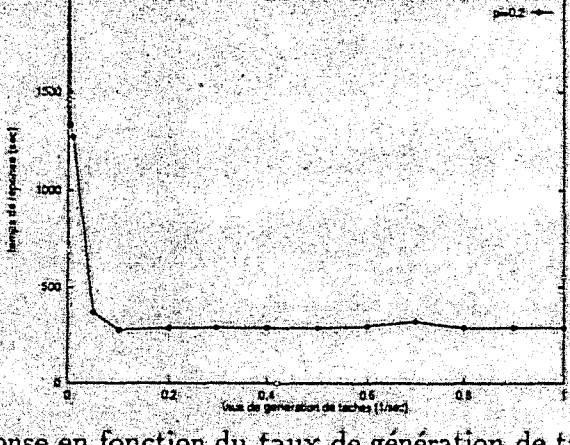
b) Temps de réponse en fonction de la densité du graphe de dépendances (π).



c) Coût des retraits en fonction de π pour une application de 100 tâches.



d) Pourcentage du surcoût d'ordonnancement en fonction de l'utilisation du système.



e) Temps de réponse en fonction du taux de génération de tâches λ ($\pi = 0.2$).

FIG. 4.10 - *Résultats de l'évaluation des différents paramètres.*

La figure 4.10d illustre la contribution du surcoût d'ordonnancement (insertion de tâches, recherche de tâches, mise à jour du graphe, etc) au temps d'exécution de l'application ($\pi \leq 0.3$) dans un environnement de faible charge externe. Lorsque le taux d'utilisation du système augmente, cela signifie que l'application parvient à stabiliser l'ordonnancement (peu de retraits et plus d'efficacité), l'ordonnanceur a tendance à intervenir moins. D'une manière générale, ce surcoût n'est pas très significatif (de l'ordre de 1 pour 1000). Notons que dans toutes ces expérimentations, la totalité des tâches est générée au lancement de l'application.

Finalement, nous avons considéré les arrivées dynamiques des tâches. En effet, les tâches sont générées dans le temps selon un processus de poisson composé [Rue89]. Les instants de génération constituent un processus de poisson de paramètre λ et le nombre de tâches générées est choisi uniformément entre 1 et le nombre de tâches restantes. La figure 4.10e illustre l'évolution du temps de réponse en fonction de λ pour une application fortement parallèle ($\pi = 0.2$). Dès que le nombre de tâches générées est suffisant ($\lambda = 0.1$) pour occuper tous les nœuds, le temps de réponse devient constant.

4.6 Conclusion

Nous avons développé un modèle pour la construction et l'ordonnancement d'applications adaptatives dans des environnements hétérogènes (réseaux de stations et clusters de processeurs). L'instanciation du modèle consiste en une interface de programmation adaptative qui permet de développer les applications parallèles en définissant l'application comme une succession d'opérations sur des données (tâches) sans se soucier de leur allocation. De plus, les tâches peuvent être allouées et réallouées sur des nœuds hétérogènes. Le modèle permet également de prendre en charge les applications irrégulières dans lesquelles les tâches sont générées durant l'exécution. L'approche d'ordonnancement consiste à maximiser le nombre de tâches prêtes à tout instant. Cela est compatible avec les environnements adaptatifs dans lesquels, les nœuds sont détectés et alloués dynamiquement à l'application qui pourra bénéficier immédiatement de leur disponibilité.

L'ordonnancement adaptatif offre plus d'opportunités que les modèles classiques dans les environnements considérés. En effet, l'application peut bénéficier de la disponibilité d'un grand nombre de nœuds tout en respectant la notion de propriété. L'aspect hétérogène est pris en compte dans les décisions de l'ordonnanceur en allouant les tâches susceptibles de générer plus de travail sur les nœuds les plus puissants. L'utilisation des ressources est par conséquent améliorée et les cycles libres peuvent être exploités par d'autres applications dans le système, ce qui constitue un cadre pour l'ordonnancement *multi-application*.

Les résultats d'évaluation de performances obtenus montrent que l'approche d'ordonnancement est intéressante. Le fait d'exploiter les dépendances des tâches avec un faible surcoût, permet d'améliorer les performances des applications moyenne-

ment parallèles (calcul scientifique). D'autre part, l'exploitation de la puissance relative des nœuds permet aux applications hautement parallèles (optimisation combinatoire) et hautement séquentielles d'utiliser toujours les plus puissants nœuds. A travers les résultats obtenus sur l'élimination de *Gauss*, nous avons montré que l'approche améliore les performances même des applications peu adaptées à notre environnement. Les résultats de cette application montrent l'impact de la granularité sur les performances des applications adaptatives. En effet, le modèle adaptatif sur les réseaux de stations requiert un rapport calcul/communication minimum pour atteindre des accélérations significatives.

Chapitre 5

Ordonnancement multi-application

Dans le chapitre précédent, nous avons abordé les problèmes d'ordonnancement dans le cadre d'applications parallèles adaptatives. Ainsi, nous avons proposé et mis en œuvre un support pour l'ordonnancement d'une application parallèle adaptative dans les réseaux de stations. Dans ce chapitre, nous abordons le problème d'ordonnancement de multiples applications sur ces plateformes. Le problème à résoudre peut être formulé comme suit : étant donné un ensemble d'applications, comment affecter les processus de ces applications aux nœuds du système distribué? comment enchaîner leur exécution?

Dans la section suivante, nous présentons une synthèse succinte des approches d'ordonnancement multi-application. La section 5.2 présente l'extension de l'environnement MARS pour l'ordonnancement multi-application. Les sections 5.3 et 5.4 présentent la structure du système et les interfaces entre le système et les applications parallèles. Dans la section 5.5, nous proposons un algorithme d'ordonnancement multi-application et enfin, nous présentons les résultats de l'évaluation de ses performances.

5.1 Techniques d'ordonnancement multi-application

Les techniques d'ordonnancement multi-application ne sont pas très développées dans la littérature. Néanmoins, nous pouvons distinguer les trois catégories suivantes : temporelle, spatiale et combinée.

5.1.1 Ordonnancement temporel

L'ordonnancement temporel d'applications parallèles consiste à planifier l'exécution des processus de ces applications dans le temps. Les méthodes les plus populaires sont celles basées sur le *"gang scheduling"* généralisé au *"co-scheduling"*.

Le *"gang scheduling"* est une technique introduite par Ousterhout dans le système

	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8
S_1	A_1	A_1	A_1	A_1				
S_2	A_2	A_2	A_2	A_2	A_2	A_2	A_2	
S_3	A_3	A_3	A_3	A_3	A_3			

TAB. 5.1 - *Exemple d'une matrice l'allocation d'un algorithme de "gang scheduling".*

Medusa [Ous82, OSS80]. La méthode est basée sur l'ordonnancement temporel avec temps partagé. Le principe repose sur la coordination de l'ordonnancement entre les différents processeurs du système pour une synchronisation plus efficace. Cela consiste à exécuter les processus d'une même application parallèle simultanément durant une période de temps appelée *slot d'ordonnancement*. A la fin du slot, tous les processeurs changent de contexte simultanément afin d'activer un autre slot et ainsi une autre application. Les slots sont activés dans le temps à tour de rôle généralement suivant une politique Round-Robin. Le tableau 5.1 présente un exemple d'une matrice d'ordonnancement utilisée par les algorithmes basés sur le *"gang scheduling"*. Les N_i représentent les nœuds du système, les S_i les slots de temps et les A_i les applications parallèles.

Ces techniques, utilisées principalement dans les systèmes massivement parallèles, souffrent du problème de fragmentation [FR90]. La fragmentation résulte des nœuds non utilisés durant le slot (tableau 5.1) provoqués notamment par les politiques d'allocation de type "première partition qui convient" (*"first-fit"*). Dans [ABM92], des améliorations, pour une implémentation efficace du gang scheduling dans les réseaux de stations, ont été évaluées. Les problèmes de délais de communication importants, de la charge imprévisible, de l'hétérogénéité et de la taille du réseau sont des facteurs qui défavorisent les approches de gang scheduling dans ces environnements.

5.1.2 Ordonnancement spatial

L'ordonnancement multi-application spatial est basé sur le partitionnement des nœuds d'un système parallèle entre les applications. Chaque application dispose ainsi d'une partition exclusive sur laquelle elle évolue jusqu'à sa terminaison (figure 5.1a). Le système commercial LoadLeveler/SP2 d'IBM est un exemple de ce type d'ordonnancement [SCZL96].

Le partitionnement peut être statique ou dynamique. Dans le partitionnement statique, la composition de chaque partition est maintenue inchangée jusqu'à la terminaison de l'application. En revanche, les approches dynamiques permettent de modifier la composition des partitions durant l'évolution des applications.

Dans [SN96], un système basé sur le partitionnement statique est développé

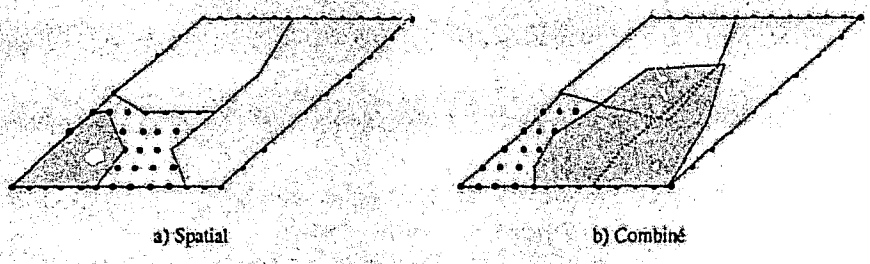


FIG. 5.1 - *Ordonnancement spatial et combiné.*

pour une plateforme hybride combinant des machines parallèles avec des réseaux de stations hétérogènes. La répartition de charge est statique, effectuée au lancement de l'application, sans possibilité d'ajustement dynamique en fonction des besoins de l'application et de l'hétérogénéité des nœuds.

5.1.3 Ordonnancement combiné

Ce type d'ordonnancement est une hybridation des deux catégories d'approches précédentes. En effet, l'ordonnancement combiné consiste à décider à la fois de l'affectation des processus des applications parallèles aux nœuds de la configuration et à planifier leur exécution dans le temps. Les techniques de gang scheduling autorisant le placement de plusieurs applications sur le même slot sont un exemple de cette classe d'ordonnanceurs [ZMJ+99].

Les techniques de gang scheduling ont évolué en utilisant des techniques d'optimisation de plus en plus complexes, dans le but de faire face à de nombreux problèmes et d'améliorer les performances. Parmi les améliorations introduites, le dédoublement de plusieurs applications sur le même slot, l'exécution de l'application sur de multiples slots si les nœuds qu'elle utilise sont libres sur ces slots, le réarrangement d'applications sur les slots (*repacking*) afin de minimiser le nombre de slots, l'exécution d'applications en attente tant que les délais de terminaison des applications les devançant ne sont pas atteints, etc [ZMJ+99, FR90, FR92].

Dans [STHI96], un algorithme nommé TSS est présenté. TSS combine le partitionnement statique d'une grille et le gang scheduling, et autorise le dédoublement d'applications sur le même slot. Le partitionnement utilise la technique de subdivision ou une technique plus élaborée, nommée scrutation adaptative, qui permet d'optimiser le partitionnement. Leur objectif est notamment d'éviter les situations où une application, en tête de la file d'attente ne pouvant tenir sur aucun slot, empêche les autres de profiter des ressources disponibles.

L'environnement MIST utilise le gang scheduling dans les réseaux de stations [ASPM+97]. Le dédoublement de processus sur le même nœud est autorisé. L'allocation d'une application est effectuée en deux étapes. Dans la première étape, la distribution idéale des processus de l'application sur tous les nœuds disponibles est

calculée en considérant l'hétérogénéité. La seconde étape consiste à minimiser le nombre de nœuds utilisés par l'application. Lorsque de nouvelles ressources se libèrent, l'algorithme d'ordonnancement essaye d'étendre l'application aux ressources nouvellement disponibles dans le but de réduire son temps de réponse. Les techniques de gang scheduling ne sont pas très adaptées aux environnements de stations, étant donné la surcoût de synchronisation, l'hétérogénéité des nœuds, la charge imprévisible de la plateforme, etc.

D'autres systèmes utilisent diverses techniques d'ordonnancement multi-application. Dans [GJK93], un ordonnanceur multi-application pour le système d'ordonnancement adaptatif Piranha est présenté. Malgré le fait que Piranha permet d'exploiter les résultats partiels des tâches, les simulations effectuées considèrent que les tâches interrompues doivent être reprises dès le début. L'ordonnancement est basé sur la prédiction des périodes moyennes de disponibilité des nœuds et sur l'estimation du temps d'exécution des tâches des applications parallèles. La préemption est autorisée et l'ordonnancement est périodique La prédiction des périodes de disponibilité des nœuds dans les réseaux de stations, et de la longueur des tâches n'est pas très réaliste et peut nuire aux performances des applications prises en charge.

Dans le système Network Analyser [Mon96], le comportement des stations est étudié et leurs taux d'utilisation sont régulièrement collectés. Deux types de comportement sont distingués : *plat* pour les stations servant essentiellement à effectuer du calcul intensif, et *cyclique* pour les stations dont les utilisateurs ont des habitudes connues. Les durées d'exécution des applications parallèles sont connues a priori. L'approche d'ordonnancement multi-application est basée sur la prédiction du temps de terminaison des applications. Le critère de placement d'une application parallèle est la minimisation de son temps de réponse individuel. Optimiser le temps de réponse individuel des applications peut causer une dégradation de l'utilisation des ressources. De plus, la connaissance a priori du coût d'exécution des applications n'est pas très réaliste dans les réseaux de stations.

L'ordonnancement dans les systèmes à large échelle consiste généralement à décomposer hiérarchiquement le système en groupes de tailles raisonnables. Ce modèle enregistre souvent des performances supérieures et des propriétés plus intéressantes (tolérance aux fautes, extensibilité, etc) que celles des systèmes complètement distribués ou centralisés. Dans [AG91], une approche d'ordonnancement spatial multi-application hiérarchique à deux niveaux est présentée et évaluée. L'approche consiste à partitionner le système en régions symétriques ou groupes. Chaque groupe est régi par un ordonnanceur responsable de l'allocation et de l'optimisation des tâches dans le groupe et du transfert des tâches dans les autres groupes en cas de surcharge des nœuds locaux. La définition des régions dans ce modèle est un problème très complexe.

Dans [Rav96], un modèle d'ordonnancement multi-application et de gestion des ressources dans les systèmes distribués à large échelle est présenté. Un modèle multi-groupe est utilisé pour la gestion des ressources. L'ordonnancement multi-application utilise une approche hiérarchique dans laquelle chaque application dispose de son

propre ordonnanceur. Des gestionnaires de ressources globaux se chargent du partage de ressources entre les applications. La mise en œuvre du modèle a débouché sur deux prototypes. Le premier pour les systèmes parallèles (CM5) où un ordonnancement spatial classique a été utilisé et le second pour traiter le problème de l'informatique mobile. Utopia [ZZWD92] et Condor-flock [ELD+96] utilisent également une structure multi-groupe pour l'équilibrage de charge à large échelle. La formation des groupes est effectuée sur des critères de proximité géographique ou fonctionnelle où des ressources critiques sont rassemblées [ZZWD92]. Le coût élevé des communications et les problèmes de sécurité défavorisent l'exécution des applications dans les groupes distants. Le transfert des applications est par conséquent réduit. Condor-flock traite des applications séquentielles batch de Condor. Utopia n'autorise pas la migration de processus.

Les techniques d'ordonnancement temporel multi-application souffrent de l'inefficacité occasionnée principalement par le problème de fragmentation. Les techniques spatiales, quant à elles, peuvent causer la dégradation de l'utilisation des ressources induite par les temps de synchronisation, de communication et d'entrées/sorties. Les approches d'ordonnancement multi-application combinées constituent une solution prometteuse pour l'exploitation de ressources dans les méta-systèmes. En effet, elles permettent de s'affranchir des problèmes caractérisant des deux classes d'approches précédentes. En revanche, elles peuvent induire des surcoûts causés par des changements de contexte fréquents.

5.2 Modèle d'ordonnancement multi-application

Notre objectif principal est de pouvoir partager les ressources entre plusieurs applications parallèles adaptatives sur les plateformes considérées et de pouvoir répondre à leurs besoins. Un ordonnanceur multi-application, possédant une vue globale des ressources et des applications, a ainsi été développé. Comme le modèle adaptatif prévoie un ordonnanceur pour chaque application, il est plus intéressant de faire coopérer l'ordonnanceur multi-application global avec les ordonnanceurs applicatifs pour une efficacité accrue de l'ordonnancement. Le résultat est un modèle d'ordonnancement multi-application hiérarchique à deux niveaux. L'ordonnanceur doit assurer une gestion efficace et équitable sur les deux plans suivants :

1. *ressources :* contrôler et gérer efficacement les ressources et fournir les outils permettant d'administrer et de connaître l'état de l'ensemble du système ;

2. *applications :* répondre aux besoins des applications parallèles s'exécutant dans le système et résoudre leurs conflits à travers une vue globale du système.

L'*ordonnanceur global* doit donc assurer le partage de ressources entre les différentes applications et la prise en charge de leurs besoins. Les objectifs et les fonctions

de cette composante sont :

- partager les ressources entre les applications parallèles équitablement et prévenir les situations de famine, où un ensemble d'applications s'empare de la totalité des ressources pendant c[illegible]es sont privées pendant de longues périodes ;

- utiliser les ressources du système d'[illegible]ière efficace et ce, en mettant à la disposition des applications en attente de disponibilité de nœuds, les ressources libérées dans les plus brefs délais. En conséquence, le temps de réponse moyen des applications est amélioré et les ressources du système sont mieux utilisées ;

- prendre en compte les contraintes imposées par certaines applications (architecture, système d'exploitation, etc) dans l'allocation de ressources. Les ressources critiques ou spécifiques doivent être allouées en priorité aux applications les plus exigentes.

Afin de pouvoir répondre à ces objectifs, une solution basée sur une interface entre l'ordonnanceur global et les ordonnanceurs d'applications est bien adaptée. En effet, l'interface doit comporter tous les services que peut et doit réaliser l'ordonnanceur global et les ordonnanceurs applicatifs. En d'autres termes, elle comporte un ensemble de protocoles connus à la fois par l'ordonnanceur global et les ordonnanceurs d'applications et qui régissent leurs interactions. La figure 5.2 illustre la structure du modèle d'ordonnancement adopté. Un agent global, dont le rôle est la gestion des ressources du système[1] et le contrôle de l'accès des applications parallèles à ces ressources, constitue la principale composante du modèle. Chaque application, représentée par un ordonnanceur, communique avec l'ordonnanceur global selon l'interface spécifiée afin d'acquérir et de restituer les ressources. Dans la figure 5.2, les entités en ovales représentent des agents persistants (ordonnanceurs global et applicatifs), l'entité désignée par *ressources* représente l'univers des ressources mises à la disposition de l'agent global et par conséquent à la disposition des applications prises en charge.

Dans ce qui suit, nous passons en revue les composantes du système d'ordonnancement global. Ainsi, la structure de l'agent global et la composition des différentes interfaces seront présentées.

5.3 Agent global

L'agent global est l'entité principale du modèle. Son rôle est l'ordonnancement de plusieurs applications parallèles et la gestion des ressources du système. Ainsi, il est responsable du contrôle de la disponibilité de l'utilisation et de l'allocation

1. Les ressources considérées consistent en une configuration de nœuds. Par conséquent, le gestionnaire de ressources sera appelé également *gestionnaire de la configuration.*

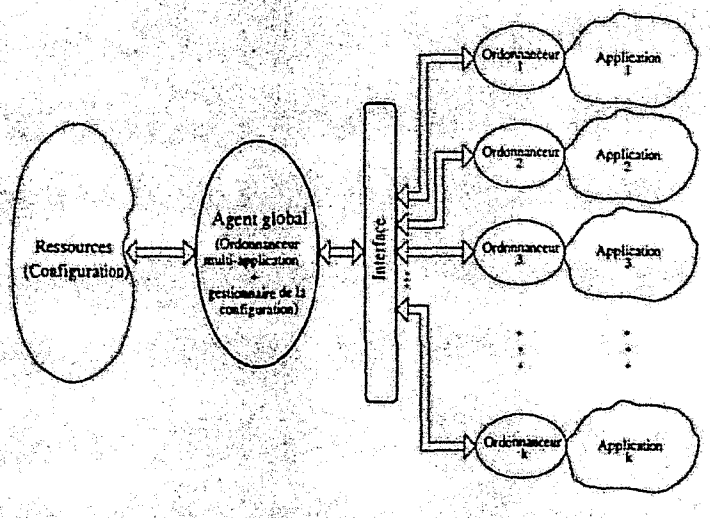


FIG. 5.2 - *Modèle d'ordonnancement multi-application.*

des ressources. Ce double rôle a motivé sa structuration en deux entités, la première est destinée à la gestion et à l'interaction avec l'univers des ressources et la seconde avec le monde des applications. Les deux agents ainsi définis (*ordonnanceur global et gestionnaire de la configuration*) échangent uniquement de l'information nécessaire à leur fonctionnement. Ils sont également complétés par deux modules de détection et de gestion des défaillances des nœuds et des applications. Ce découpage fonctionnel contribue également à la modularité, à la réutilisabilité et par conséquent à l'évolution du système. La figure 5.3 illustre la structure détaillée de l'agent global. L'agent global est centralisé sur un seul nœud. Ses deux composantes "gestionnaire de la configuration" et "ordonnanceur global" sont implémentées par deux processus résidant sur le même nœud.

5.3.1 Gestionnaire de la configuration

Le gestionnaire de la configuration est chargé de répertorier les ressources, de contrôler leur état et de mettre les nœuds disponibles à la disposition de l'ordonnanceur global. De plus, il doit fournir une interface pour l'administration des ressources. La figure 5.3 présente la structure du gestionnaire de la configuration. Ce dernier comporte essentiellement un système de *collecte d'information de charge* des nœuds, un module de contrôle et de *gestion de la disponibilité* des nœuds et un autre pour la *détection des défaillances.*

L'état de la configuration est conservé dans une table qui comporte toutes les informations concernant les nœuds (caractéristiques techniques, état, etc). L'état de charge des nœuds est contrôlé par l'élément d'information, qui dispose d'un gestionnaire sur chaque nœud (voir chapitre 2). La politique d'information et les critères de disponibilité de nœuds de la première version de MARS sont maintenus [Haf98]. Le module de défaillances se charge de la détection des défaillances des nœuds et

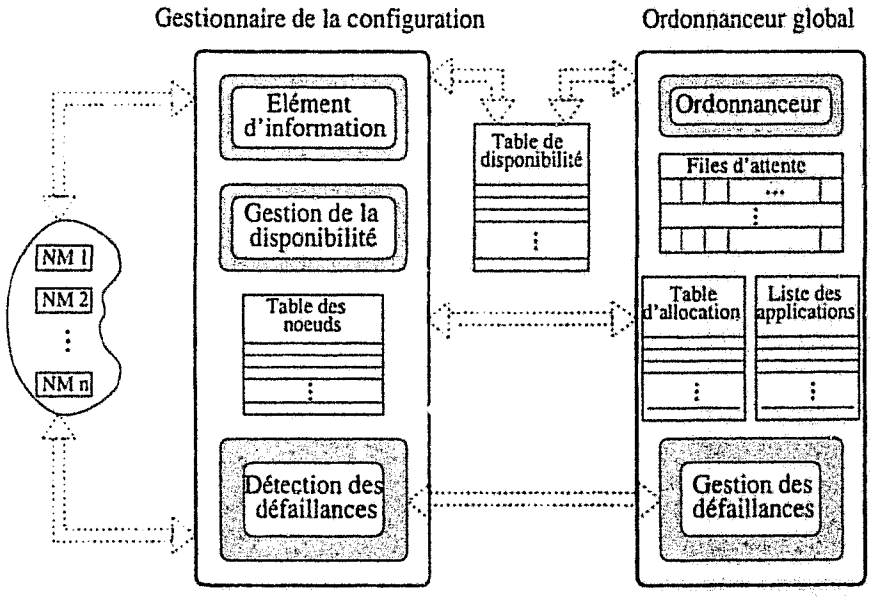


FIG. 5.3 - *Architecture de l'agent global.*

de l'exécution des actions conséquentes comme nous l'avons présenté au chapitre 3 (information de l'ordonnanceur global, récupération du nœud, reconfiguration du système).

5.3.2 Ordonnanceur global

Nous avons vu que le *gestionnaire de la configuration* est responsable de la gestion et du contrôle des nœuds de la configuration. De la même manière, l'*ordonnanceur global* est responsable de la gestion des applications parallèles soumises au système (figure 5.3). Cela consiste à recevoir leurs demandes de ressources et à les satisfaire suivant une politique bien définie et à réagir aux changements dynamiques de l'état du système. L'ordonnanceur global se compose principalement d'un ordonnanceur et d'un module complémentaire responsable de la détection et de la gestion des défaillances des applications (voir chapitre 3).

Le module d'ordonnancement utilise une *liste* contenant toutes les *applications* prises en charge par le système, une *table d'allocation* reflétant l'état de l'allocation et un ensemble de *files d'attente*.

La liste des applications englobe toutes les applications soumises au système actuellement. Pour chaque application, l'ensemble des informations nécessaires à son fonctionnement est préservé (identité, adresse de son ordonnanceur, liste des ressources demandées, etc). La table d'allocation résume l'état des nœuds vis-à-vis des applications. Chaque entrée est réservée à un nœud particulier et contient la liste des processus placés sur le nœud et en particulier les ordonnanceurs d'applications.

Ces informations sont utilisées par l'ordonnanceur afin de vérifier les contraintes d'allocation des applications et par le module de gestion des défaillances.

Les applications prises en charge appartiennent à trois classes suivant le type de ressources demandées. Les trois classes seront présentées dans la section 5.5. Pour chaque classe d'applications, une file d'attente est prévue. L'ordonnanceur global considère toutes les demandes en ressources par une application et essaye de les satisfaire simultanément. Cela permet d'éviter les situations d'interblocage qui conduisent à une mauvaise utilisation des ressources.

La disponibilité des ressources est définie et gérée par le gestionnaire de la configuration (*gestionnaire de la disponibilité*). Les nœuds peu chargés pouvant recevoir une charge additionnelle sont déterminés et mis à la disposition de l'ordonnanceur global à travers la *table de disponibilité*. Cette table est organisée suivant les classes de ressources. En effet, les ressources sont partitionnées en deux classes : WORKSTATIONS comportant les stations de travail et SPECIFIC_RESOURCES pour les autres ressources. Cette classification est motivée par le fait que les stations de travail sont majoritaires dans notre environnement et qu'elles possèdent des propriétés communes permettant à certaines applications parallèles de les utiliser d'une manière équivalente. En plus de l'hétérogénéité du point de vue architecture, l'aspect hétérogénéité du point de vue puissance relative des nœuds est pris en considération dans la table de disponibilité.

5.4 Interfaces de l'agent global

Dans son fonctionnement, l'agent global doit interagir à la fois avec l'administrateur du système et avec les applications prises en charge. Pour cela, deux interfaces bidirectionnelles sont prévues (figure 5.4). La première lie l'ordonnanceur global avec les applications et la seconde le gestionnaire de la configuration avec l'administrateur.

L'ordonnanceur global doit pouvoir informer les applications des changements de l'environnement, de leur demander de réaliser certaines actions et de recevoir leurs requêtes. Les protocoles et les services que comporte cette interface sont définis par les types de service et les structures de requêtes considérés (figure 5.4).

L'administrateur doit intervenir afin de gérer dynamiquement la configuration (ajout, suppression de nœuds, etc) et de positionner les seuils et les facteurs d'utilisation des nœuds. De plus, le système doit avoir le moyen d'informer l'administrateur de l'évolution de l'état de la configuration et des problèmes surgissant dans le temps (reconfiguration du système, etc). Ainsi, une interface avec le système est requise. Les opérations principales autorisées sont l'ajout et la supression dynamiques de nœuds de la configuration. Dans les deux cas, l'ordonnanceur global doit, avec la complicité des applications, pouvoir s'adapter à la situation imposée par l'administrateur. Lorsqu'un nœud est ajouté à la configuration, ses caractéristiques doivent

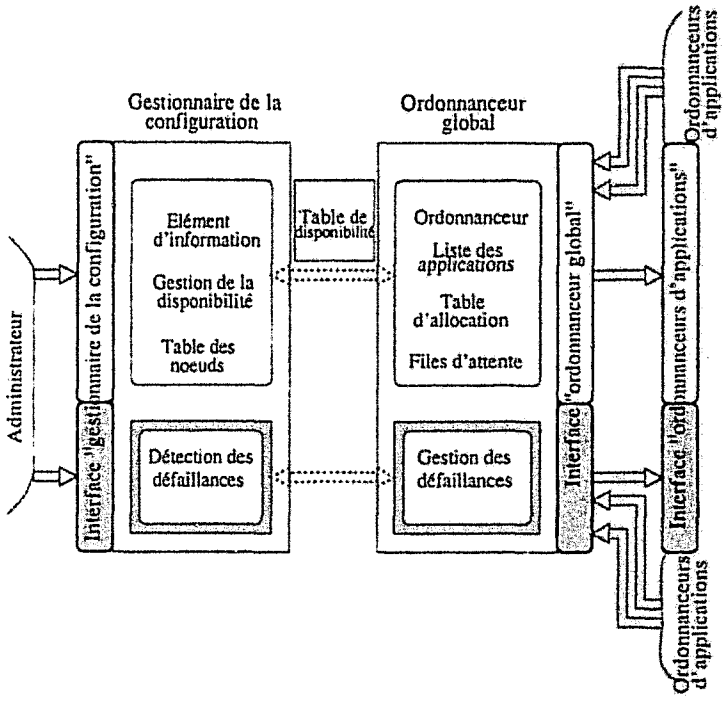


FIG. 5.4 - *Interfaces de l'agent global.*

figurer sur le fichier de configuration (puissance relative, architecture, etc). L'administrateur doit également être en mesure d'arrêter le système. L'ordonnanceur global doit dans ce cas aviser toutes les applications afin qu'elles puissent terminer proprement (procéder à la sauvegarde de leur état, par exemple) avant de mettre un terme aux activités du système. Enfin, le système peut établir un résumé de son état (ressources et applications) et le communiquer au monde extérieur.

5.5 Ordonnancement multi-application

L'ordonnancement multi-application consiste à satisfaire les besoins de plusieurs applications parallèles en compétition pour l'utilisation d'un ensemble de ressources. Les ressources étant classées en deux catégories, les applications parallèles sont distinguées suivant le type de ressources demandées lors de l'émission de leurs requêtes.

5.5.1 Structures des requêtes

Le modèle d'application supporté par notre environnement permet de définir les contraintes d'allocation et d'utilisation des ressources. Trois classes de requêtes sont

ainsi définies :

- *requête WORKSTATION :* cette classe de requêtes indique que l'application souhaite utiliser des stations de travail sans donner d'importance à l'architecture matérielle. Le nombre d'exemplaires demandé est spécifié par une plage (*min-max*) résumant en quelque sorte les bornes du degré de parallélisme en dehors desquelles, l'efficacité de l'application ne peut être améliorée. L'application ne pourra pas évoluer si le nombre de ressources allouées passe en dessous de *min*. La borne *max* pourra être importante afin de pouvoir bénéficier de la disponibilité de ressources lorsque le système devient moins chargé. La requête doit également mentionner la façon dont les ressources seront utilisées i.e. si l'application tolère le dédoublement de ses processus sur le même nœud. La structure de cette requête est : $request(WORKSTATION, min, max, usage\_mode)$;

- *requête WORKSTATION PER TYPE :* cette requête concerne également le parc de stations de travail. La seule différence avec la précédente est que le type (l'architecture) des stations est considéré. La requête se présente comme une agrégation de structures dont chacune mentionne le nombre d'exemplaires demandés du type en question, exprimé par une plage (*min-max*). Les contraintes d'utilisation sont mentionnées pour chaque type séparément.
  $request(WORKSTATION\_PER\_TYPE, \bigcup_{i=1}^{nb\_req\_types}\{(min_i, max_i, usage\_mode_i)\})$;

- *requête SPECIFIC RESOURCE :* cette requête est similaire dans sa structure à la précédente excepté qu'elle concerne les ressources particulières, en dehors des stations de travail. Pour chaque type de ressources particulières, le nombre d'exemplaires est mentionné. Etant donné qu'il s'agit généralement de nœuds de machines parallèles, les ressources de cette classe sont considérées mono-programmables.
  $request(SPECIFIC\_RESOURCE, \bigcup_{i=1}^{nb\_req\_types}\{(min_i, max_i)\})$.

5.5.2 Critères et contraintes d'ordonnancement

A travers nos hypothèses, nous avons vu que les applications considérées sont caractérisées par des durées de vie différentes et souvent importantes (de quelques minutes à plusieurs jours). L'arrivée d'applications peut être importante pendant des périodes données et les besoins peuvent donc dépasser la capacité du système. Par conséquent, l'ordonnanceur global doit tenir compte des facteurs et des paramètres suivants :

C 5.1 ***Charge :*** *lorsqu'un nœud atteint le seuil de surcharge, un ou plusieurs processus doivent être retirés.*

C 5.2 ***Propriétaire :*** *la présence d'un utilisateur sur sa station suggère l'évacuation de toute la charge placée sur le nœud en question dans des délais raisonnables.*

C 5.3 ***Contraintes architecturales*** *: une application demandant un type de ressources particulier doit avoir accès à cette ressource. Par conséquent, ce type d'applications doit être privilégié. La différence entre le nombre de nœuds alloués et le nombre minimum que doit avoir l'application doit être également considérée dans les décisions de l'ordonnanceur, afin de minimiser le nombre d'applications bloquées faute de ressources suffisantes.*

Ces facteurs constituent les contraintes principales imposées à l'ordonnanceur. La politique d'ordonnancement doit également répondre à deux objectifs principaux :

O 5.1 ***Temps de réponse moyen*** *: il s'agit d'un critère essentiel pour toute politique d'ordonnancement. Les applications soumises doivent évoluer dans des temps raisonnables. Le temps de réponse individuel (par application) doit être minimisé. En effet, étant donné que l'hétérogénéité en temps d'exécution des applications considérées est importante, les applications relativement courtes ne doivent pas être pénalisées par les applications de longue durée de vie.*

O 5.2 ***Efficacité et performances*** *: le surcoût induit par l'ordonnancement ne doit pas pénaliser les performances du système lui-même. De plus, les améliorations apportées doivent être d'une qualité suffisante, permettant de compenser l'effort fourni. Le taux d'utilisation des ressources doit être maximisé.*

5.5.3 Algorithme d'ordonnancement

Suivant les hypothèses faites sur les modèles d'applications, sur les propriétés de la plateforme et les contraintes à respecter, nous avons opté pour une politique d'*ordonnancement combinée* (spatiale et temporelle). Ce type d'ordonnancement permet une meilleure utilisation du système, un partage plus équitable des ressources et une meilleure flexibilité. L'ordonnancement local au niveau de chaque nœud est assuré par le système d'exploitation local. L'ordonnanceur est sollicité à chaque fois qu'un des événements suivants se produit :

E 5.1 ***Arrivée d'une nouvelle application*** *: cet événement est généré par l'ordonnanceur de l'application au moment de l'enrôlement. L'application présente une requête qui incite l'ordonnanceur à essayer de la satisfaire dans les plus brefs délais.*

E 5.2 ***Surcharge d'un nœud*** *: la surcharge d'un nœud, détectée par l'élément d'information et communiquée à l'ordonnanceur global, doit provoquer la diminution du degré de parallélisme ou le déplacement de la charge d'une ou de plusieurs applications. Le choix d'applications à désallouer doit faire intervenir certains paramètres liés à l'état de l'application et du système.*

E 5.3 ***Présence d'un propriétaire*** *: la réquisition d'un nœud par un utilisateur implique l'évacuation de tous les processus placés sur le nœud en question.*

E 5.4 ***Disponibilité d'un nœud :*** *la disponibilité d'un nœud peut provoquer la relance d'une ou de plusieurs applications bloquées en attente de disponibilité de ressources, ou profiter aux applications pouvant augmenter leur degré de parallélisme.*

E 5.5 ***Terminaison d'une application :*** *la terminaison d'une application provoque la libération des ressources qu'elle occupait. La libération des ressources n'est pas reportée directement mais à travers l'élément d'information qui détectera la disponibilité de ces nœuds et remontera l'information au gestionnaire de ressources. Cet événement permet uniquement la mise à jour des données de l'ordonnanceur global et du module de gestion des défaillances.*

E 5.6 ***Terminaison d'un processus :*** *de la même façon, la conséquence immédiate de cet événement est la mise à jour de la table d'allocation.*

Dans ce qui suit, nous présentons les algorithmes que nous avons mis en œuvre que l'ordonnanceur global utilise en réaction à ces événements.

5.5.3.1 Nouvelle application

L'arrivée d'une nouvelle application est accompagnée d'une requête exprimant ses besoins en ressources. Cette requête est prise en charge suivant sa classe, son type et l'état du système. Les paramètres suivants sont utilisés :

S : liste d'identificateurs de nœuds à allouer.
S_length : nombre de nœuds à allouer (a priori).
ws_types : nombre total de types (architectures) de stations de travail.
avail_workstations(i) : ensemble de stations de type i disponibles actuellement.

La prise en charge de la requête d'une nouvelle application est effectuée en trois étapes : vérification de la requête, satisfaction de la requête et éventuellement un réajustement si une partie des ressources à allouer est utilisée par d'autres applications.

5.5.3.1.1 Vérification de la requête : cette phase consiste à s'assurer de l'existence des architectures demandées et de la validité des intervalles spécifiés. Ainsi, si la requête est de type WORKSTATION, il faut s'assurer que le *min* n'excède pas le nombre total de processus que l'ensemble des stations peut supporter[2]. Pour les requêtes portant sur de multiples architectures, la procédure est appliquée à chaque type d'architecture. Généralement, les bornes sont ajustées en fonction de la taille de la configuration et une requête n'est rejetée que si sa satisfaction est impossible[3].

2. Le nombre de processus qu'un nœud peut supporter est également appelé nombre de places.
3. Architecture non disponible ou degré de parallélisme minimum excède le nombre total de ressources de ce type de la configuration actuelle.

5.5.3.1.2 Satisfaction de la requête : une fois la cohérence de la requête vérifiée, sa satisfaction est tentée. Cette étape consiste à déterminer l'ensemble S de ressources à allouer à l'application selon la classe de sa requête :

- requête WORKSTATION : si l'application n'autorise pas le dédoublement de ses processus sur le même nœud, chaque station disponible sera comptée une seule fois. La formule 5.1 est utilisée pour calculer *S_length*, le nombre de nœuds ou de places à allouer à l'application (figure 5.5). Ce nombre signifie qu'on doit allouer la moyenne ou le nombre de nœuds disponibles et cette quantité doit être comprise entre *min* et *max*. Dans le cas où la quantité de ressources à allouer n'est pas disponible, un autre algorithme est exécuté (étape suivante).

$$S\_length = Min(Max(min, average, avail), max). \tag{5.1}$$

Où :

min : degré de parallélisme minimum de l'application. C'est la quantité de ressources minimum que l'ordonnanceur doit attribuer à l'application.
max : degré de parallélisme maximum.
average : degré de parallélisme moyen des applications en exécution.
avail : nombre de stations disponibles.
Min et *Max* : fonctions retournant respectivement le minimum et le maximum d'un ensemble de valeurs.

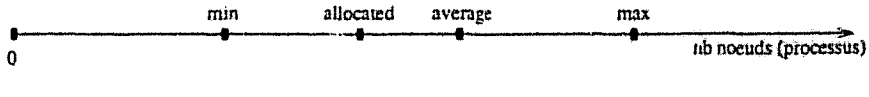


FIG. 5.5 - *Paramètres d'ordonnancement.*

L'ensemble de nœuds S à allouer à l'application est déterminé comme suit :

for $i = 1$ **to** *ws_types* **do**
  while $n \in avail\_workstations(i) \wedge k < S\_length$ **do**
    $S = S \uplus [n]$
    $k = k + 1$
  end while
end for.

En revanche, si l'application autorise le dédoublement de ses processus sur le même nœud, chaque station sera comptée autant de fois qu'elle peut recevoir de processus additionnels (nombre de places disponibles sur la station).

- requête WORKSTATION_PER_TYPE: pour ce type de requêtes, un algorithme similaire est exécuté. L'unique différence avec l'algorithme précédent est qu'il considère une seule architecture à la fois. L'algorithme est exécuté récursivement pour chaque architecture demandée. Ainsi, on détermine la liste S_i de stations de type i à allouer à l'application.

$$S_i\_length = Min(Max(min_i, average_i, avail_i), max_i). \tag{5.2}$$

Où :

min_i : quantité de ressources minimum (stations de type i) que l'ordonnanceur doit attribuer à l'application.
max_i : borne maximum.
$average_i$: moyenne des quantités de ressources (stations de type i) allouées aux applications en exécution.
$avail_i$: nombre de stations de type i disponibles.

L'ensemble de ressources S_i à allouer est déterminé comme suit :

while $n \in avail\_workstations(i) \wedge k < S_i\_length$ **do**
$S_i = S_i \uplus [n]$
$k = k + 1$
end while.

De même, si l'application autorise le placement multiple de processus, chaque station est comptée autant de fois que le nombre de processus qu'elle peut encore supporter.

- requête SPECIFIC_RESOURCE: comme pour le cas précédent, on procède par type d'architecture demandé. Etant donné les caractéristiques de cette classe de ressources (ressources non partageables), le placement de plusieurs processus sur le même nœud n'est pas autorisé. Le calcul de la quantité de ressources et la détermination des nœuds à allouer à l'application sont effectués de la même manière que dans le cas précédent.

5.5.3.1.3 Réajustement (remapping) : cette étape complète l'étape précédente en essayant de satisfaire les demandes des applications qui n'ont pas eu la quantité de ressources nécessaire. Cette étape sert notamment à assurer un partage relativement équitable des ressources.

En effet, si la quantité de ressources S ou S_i attribuée durant l'étape précédente est inférieure à la quantité de ressources requise ($|S| < S\_length$), un retrait de ressources aux applications déjà placées doit être effectué. Cela consiste à trouver

$nb = S\_length - |S|$ nœuds ou places utilisés par les autres applications. Les paramètres suivants sont utilisés :

nb : nombre de nœuds à trouver.
appli : application en question.
$Procs$: fonction retournant l'ensemble des processus d'un nœud ou d'une application.
$Appli$: fonction retournant l'application propriétaire d'un processus.
SA : liste de nœuds (éventuellement de type i) alloués à l'application.
SPD : ensemble de processus à désallouer.
A : liste de tous les nœuds de même classe que la requête de l'application (stations de travail ou ressources spécifiques) et éventuellement de type i.

La table d'allocation est utilisée à cet effet. La libération de nœuds doit vérifier que chaque application ne doit pas se bloquer à cause du passage de la quantité de ressources qu'elle utilise en dessous de la quantité minimum requise *min*. Les contraintes d'allocation de l'application bénéficiaire sont également prises en compte (mono-programmation, etc). Dans le cas où suffisamment de ressources existent, un choix est effectué. Cela se passe en trois étapes :

tape 1 : cette étape consiste à déterminer l'ensemble des nœuds ou des places de l'architecture demandée occupés par d'autres applications (l'ensemble des nœuds allouables à l'application). Cet ensemble est déterminé comme suit :

for $n \in A$ **do**
  for $P \in Procs(n)$ **do**
    if $P \notin Procs(appli) \wedge (Mode(appli) = MULTI\_PROG \vee$
    $Procs(appli) \cap Procs(n) = \Phi \wedge n \notin S)$ **then**
      étape 2. **end if**
  end for
end for.

En effet, il faut vérifier que le processus n'appartient pas à l'application, et que l'application ne possède pas déjà un processus sur le même nœud alors qu'elle n'autorise pas la multi-programmation en utilisant S et la table d'allocation.

tape 2 : l'étape 2 consiste à déterminer l'éligibilité du processus en question à être désalloué. Pour cela, un mécanisme de *"ranking"* est utilisé. L'ensemble de nœuds et de processus à désallouer est construit comme suit :

$R_k = compute\_appli\_rank(Appli(P), class(n), arch(n))$
$appli\_rank = compute\_appli\_rank(appli, class(n).arch(n))$
if $appli\_rank < R_k$ **then**
  $SA = SA \uplus n.$
  $SPD = SPD \uplus P.$

update_rank(Appli(P))
end if.

Une valeur d'éligibilité, basée sur le ranking, est calculée pour l'application propriétaire du processus considéré (P). Cette valeur dépend de l'état actuel de l'application et de ses contraintes. Dans la réalisation actuelle, cette valeur est calculée comme suit :

$$rank(appli, class, arch) = \alpha \times nb\_allocated_{arch} + (1-\alpha) \times (nb\_allocated_{arch} - min_{arch}). \quad (5.3)$$

Où $nb\_allocated_{arch}$: est le nombre de processus que possède actuellement l'application *appli* sur des nœuds d'architecture *arch*.

$0 < \alpha < 1$, $\alpha = 0.5$ dans l'implémentation courante.

Tout retrait de processus à une application est mémorisé et son éligibilité est recalculée pour chaque nouveau processus considéré.

tape 3 : après l'exécution des deux étapes précédentes, la dernière étape consiste à choisir le nombre exact de processus manquants parmi l'ensemble de ceux déterminés précédemment. En effet, si le nombre de places déterminées précédemment excède le nombre voulu, un choix s'impose. Cela est effectué en triant les places disponibles et en choisissant les nb processus correspondant aux applications les plus éligibles. L'ensemble final de ressources à allouer est déterminé comme suit :

if $|SA| > nb$ **then**
Trier SPD (et SA) suivant les valeurs décroissantes de R.
$SA = SA - SA[nb+1..|SA|]$
$SPD = SPD - SPD[nb+1..|SA|]$
end if.

5.5.3.1.4 Allocation : dans les trois étapes précédentes, la liste des nœuds à allouer S et éventuellement la liste des nœuds complémentaire SA et l'ensemble des processus à désallouer SPD sont déterminés. Cette phase complète les étapes précédentes en entreprenant les actions suivantes :

1. si la quantité de ressources $S \cup SA$ (d'au moins une architecture) est inférieure à la quantité de ressources minimum requise ($|S \cup SA| < min$), l'allocation est annulée et l'application est avisée. Par conséquent, elle sera insérée dans la file d'attente correspondante en attendant la disponibilité de ressources ;

2. dans le cas contraire, la liste des places éventuellement trouvées SA est adjointe à la liste initiale S et l'ensemble est communiqué à l'application afin de commencer son exécution. Les processus à retirer sont rassemblés par application et envoyés à l'application correspondante afin de les déloger. L'application est également insérée dans la file d'attente des applications non bloquées afin de pouvoir s'agrandir en cas de sous-charge du système.

La contrainte principale dans cette opération est que l'état des nœuds sur lesquels s'exécutent les processus devrant être désalloués peut changer favorablement ou défavorablement en fonction de la charge externe et de l'évolution des applications entre le moment de la décision et la désallocation.

5.5.3.2 Surcharge d'un nœud

La surcharge d'un nœud est détectée par l'élément d'information et remontée au *gestionnaire de la configuration*. Ce dernier informe l'*ordonnanceur global* afin de procéder à l'allégement de la charge du nœud en question. L'ordonnanceur global choisit ainsi un des processus s'exécutant sur le nœud et l'évacue en espérant que cette désallocation fera passer le nœud dans un état de charge normale.

De nouveau, le choix du processus à évacuer est effectué en considérant l'état actuel des applications concernées (formule 5.3). La répercussion de cette décision sur les applications pénalisées se reportera au moment de la terminaison effective des processus évacués.

Si la charge du nœud demeure au dessus du seuil de surcharge, d'autres désallocations successives sont effectuées. Le gestionnaire de la configuration attend une période prédéfinie avant de reformuler une nouvelle requête de désallocation si un changement significatif de l'état du nœud n'a pas été enregistré.

5.5.3.3 Réquisition d'un nœud

Dès qu'un utilisateur se présente sur sa station, toute la puissance de celle-ci doit lui être restituée. Tous les processus placés sur le nœud en question doivent être évacués. Les processus à désallouer sont rassemblés par application et une requête est adressée à chaque application concernée.

5.5.3.4 Terminaison d'un processus

La terminaison d'un processus d'une application peut être causée par plusieurs événements : terminaison prévisible dûe à la contraction contrôlée (par l'application ou par son ordonnanceur) du degré de parallélisme, terminaison imposée par le système en réponse à la surcharge du nœud, défaillance, etc. A la terminaison d'un

processus, l'algorithme suivant est exécuté :

- les structures de données sont mises à jour (table d'allocation, etc) ;
- si le nombre actuel de processus de l'application (de l'architecture correspondante) est inférieur au nombre minimum qu'elle doit en avoir (synonyme de sa suspension), une tentative d'allocation est effectuée :
  1. l'algorithme de calcul de S ou S_i, suivant la classe de la requête de l'application, est exécuté. La seule différence avec les versions précédentes est que la quantité de ressources manquante est déterminée en tenant compte de l'état actuel de l'application. Ainsi, les différents seuils $(min_i, average_i, max_i)$ sont retranchés du nombre de processus alloués $allocated_i$. Si la requête est de type WORKSTATION, le calcul est effectué par rapport au parc de stations,
  2. de la même manière que pour une nouvelle application, l'algorithme de réajustement est éventuellement exécuté si c'est nécessaire,
  3. si les algorithmes exécutés parviennent à trouver de nouvelles ressources, elles sont allouées à l'application ;
- si la terminaison du processus ne met pas l'application dans une situation critique, aucune allocation n'est immédiatement effectuée. En effet, c'est l'ordonnanceur qui, à travers les files d'attente, effectue les allocations nécessaires.

Notons que la terminaison d'un processus ne fournit pas automatiquement une ressource disponible. Cet événement se répercute éventuellement sur l'état du nœud et l'élément d'information s'en apercevra ultérieurement.

5.5.3.5 Disponibilité d'un nœud

Lorsqu'un nœud n passe à l'état *disponible*, l'ordonnanceur global doit en profiter pour redémarrer le plus grand nombre d'applications bloquées faute de ressources suffisantes. S'il s'agit d'une station, les applications exigeant l'architecture du nœud en question sont favorisées par rapport à celles utilisant les stations de travail d'une manière équivalente.

Une application est choisie et son allocation est tentée (*satisfaction de la requête* 5.5.3.1.2 et éventuellement étapes 1, 2, 3 de *réajustement* 5.5.3.1.3).

Si aucune application bloquée ne peut en profiter, le nœud est alloué aux applications actives pouvant élargir leur degré de parallélisme. De nouveau, les applications demandant une architecture particulière sont privilégiées. Seul l'algorithme de *satisfaction de la requête* 5.5.3.1.2 est exécuté. Un ranking est effectué dans ce cas pour choisir l'application la plus appropriée.

L'exécution de l'algorithme S a deux intérêts: le premier est de s'assurer de la non-existence de ressources non utilisées et le deuxième pour vérifier les contraintes d'allocation (multi-programmation, etc).

Globalement, les requêtes typées sont favorisées. Cela est justifié par le fait que la satisfaction d'une demande portant sur une architecture particulière est plus difficile qu'une requête générale étant donné que le choix est plus large. De plus, les requêtes basées sur l'architecture sont relativement rares. Par ailleurs, l'ordonnanceur essaye de débloquer le maximum d'applications avant d'allouer la ressource aux applications déjà actives. L'objectif de cette politique est de maintenir une certaine équité entre les applications.

5.5.3.6 Terminaison d'une application

La terminaison d'une application correspond à la terminaison de son ordonnanceur. Notons que la table d'allocation est mise à jour à chaque terminaison d'un processus de l'application. La terminaison de l'application permet essentiellement à l'ordonnanceur global de la retirer des files d'attente et ne plus considérer sa requête.

5.5.4 Interaction avec le gestionnaire de la configuration

Une des fonctions principales du gestionnaire de la configuration est de servir l'interface entre les applications et les ressources. Par conséquent, il doit avoir à tout instant l'état des ressources. Cette information est mise à la disposition de l'ordonnanceur global à travers la *table de disponibilité.*

La table de disponibilité est scindée en deux parties: l'une pour les stations de travail et l'autre pour les autres ressources (ressources spécifiques). Cette distinction est justifiée par le fait que les stations de travail sont majoritaires et possèdent des caractéristiques similaires (système d'exploitation Unix, temps partagé, etc). Les applications ont tendance à banaliser ces plateformes en fournissant des versions exécutables sur les différentes architectures.

Pour chaque type d'architecture, les nœuds disponibles (à l'état IDLE) et leur nombre sont mentionnés. Le nombre de places disponibles est également indiqué. Ce nombre est calculé suivant la charge et la puissance relative du nœud. Le nombre total de places disponibles est également mentionné. Ces valeurs concernent uniquement les nœuds *multi-programmés* (stations de travail).

Les événements générés au niveau de l'ordonnanceur global sont issus du gestionnaire de la configuration qui, à travers l'élément d'information, surveille les activités de la configuration. Ainsi, lorsque un *gestionnaire de nœud* informe le gestionnaire de la configuration de la présence d'un propriétaire, de la surcharge, ou de la disponibilité d'un nœud, le gestionnaire de la configuration met à jour la table de disponibilité

et génère l'événement approprié si nécessaire. Ces événements sont :

E 5.7 ***Nœud sous-chargé :*** *lorsqu'un nœud passe à l'état IDLE, le gestionnaire de la configuration est informé. Le nœud est inscrit dans la partie appropriée de la table de disponibilité et le nombre de nœuds et éventuellement de places disponibles, de l'architecture correspondante, sont mis à jour. L'ordonnanceur global est ensuite avisé.*

E 5.8 ***Nœud moyennement chargé :*** *lorsqu'un nœud est légèrement chargé, un processus additionnel peut le mettre dans un état de surcharge. Par conséquent, si le nœud était à l'état IDLE, le gestionnaire de disponibilité le retire de la table de disponibilité afin de prévenir son utilisation et met à jour le nombre de nœuds et le nombre de places disponibles.*

E 5.9 ***Nœud surchargé :*** *la surcharge d'un nœud peut provenir de processus sous le contrôle de l'ordonnanceur global ou d'applications externes. Si le nœud est dans la table de disponibilité, il sera retiré et le nombre de nœuds libres sera mis à jour. L'ordonnanceur global est informé afin de procéder au retrait des processus concernés.*

E 5.10 ***Nœud réquisitionné :*** *lorsqu'un nœud est réclamé par son propriétaire, le gestionnaire de disponibilité vérifie s'il est dans la table de disponibilité, le retire et met à jour le nombre de nœuds disponibles.*

5.6 Evaluation des performances

Dans cette section, nous présentons quelques résultats expérimentaux dans le but d'évaluer les performances de notre approche d'ordonnancement. Les études menées pour évaluer les performances des approches d'ordonnancement multi-application ne sont pas très nombreuses. De plus, elles concernent généralement les systèmes parallèles à mémoire partagée [MEB88, LV90, Bre94]. Nous avons utilisé une application parallèle synthétique tirée du domaine du calcul scientifique. Le nombre de tâches, leur durée et le temps de service de l'application sont les paramètres. Cela permet une évaluation systématique du système. A chaque expérimentation, un ensemble d'applications parallèles adaptatives est généré et exécuté suivant plusieurs politiques d'ordonnancement. Les points clés qui caractérisent les expérimentations sont : le temps de service (coût d'exécution) des applications, la configuration matérielle, la politique d'ordonnancement utilisée, et la distribution des temps d'inter-arrivées.

5.6.1 Caractérisation des applications

Une application parallèle commence son exécution dès la soumission de sa première requête au système, soit à l'instant t^a et se termine lorsque son ordonnanceur

envoie un message de terminaison à l'ordonnanceur global au temps t_i^d. Les paramètres suivants caractérisent l'application :

t_i^a : temps d'arrivée de l'application i.
t_i^d : temps de départ de la même application.
$NbSlices_i$: nombre total de *"slices"* temporels de l'application i. Chaque slice est délimité par la création d'un nouveau processus ou la terminaison d'un processus.
$SliceLength_{ij}$: durée temporelle du slice j de l'application i.
$NbProcs_{ij}$: nombre de processus de l application i durant le slice j.

Les quantités suivantes sont mesurées pour chaque application (n) :

- *temps de réponse (RT_n) ou temps parallèle :* défini comme étant le temps écoulé entre l'enrôlement de l'application et l'instant de sa terminaison.

$$RT_n = t_n^d - t_n^a \tag{5.4}$$

- *temps séquentiel (ST_n) :* ce temps est fourni par l'ordonnanceur de l'application et correspond à l'exécution de la meilleure version séquentielle sur un nœud de base non chargé. Pour les applications adaptatives, il est déduit du coût d'exécution des tâches de l'application (temps CPU) ;

- *degré de parallélisme moyen (APD_n) et taux de variation du degré de parallélisme ($PDVR_n$) :* il s'agit de la moyenne et de l'écart-type du nombre de processus de l'application à chaque instant pondérés par le temps.

$$APD_n = \frac{\sum_{j=1}^{NbSlices_n} NbProcs_{nj} \times SliceLength_{nj}}{RT_n} \tag{5.5}$$

$$PDVR_n = \sqrt{\sum_{j=1}^{NbSlices_n} (\frac{NbProcs_{nj} \times SliceLength_{nj}}{RT_n} - APD_n)^2} \tag{5.6}$$

5.6.2 Caractérisation de la configuration matérielle

La plateforme utilisée pour les expérimentations est composée d'une vingtaine de stations SUN4 reliées par un réseau local Ethernet 10Mbits/s. L'hétérogénéité matérielle est considérable. Le taux d'utilisation du système est estimé suivant les indicateurs de charge et comprend la charge induite par les processus externes.

5.6.3 Expérimentations

Nous avons expérimenté deux politiques d'ordonnancement:

- notre politique d'ordonnancement décrite précédemment (combinée);
- une politique basée sur le partitionnement dynamique (ordonnancement spatial). L'approche que nous avons expérimentée consiste à allouer uniquement les ressources disponibles sans désallocation d'autres applications. Les ajustements des partitions sont effectués à chaque libération de nœud.

L'exécution est effectuée en sessions de durée choisie aléatoirement entre deux bornes. Durant chaque session, un ensemble d'applications adaptatives de différents temps de service est généré. Un modèle basé sur les files d'attente est utilisé. L'arrivée est dynamique et les intervalles d'inter-arrivées sont exponentiellement distribués de taux λ. Notons que les bornes du degré de parallélisme *min* et *max* de toutes les exécutions sont prises respectivement 1 et le nombre de nœuds de la configuration.

La principale mesure de performance est le temps de réponse moyen des applications [LV90, MEB88]. Ceci est évalué par rapport à différents paramètres: taux d'utilisation des ressources de la configuration, taux d'arrivées des applications, degré de parallélisme moyen, variation du degré de parallélisme et le temps de service moyen des applications.

Le temps de réponse moyen d'une session s est défini comme suit:

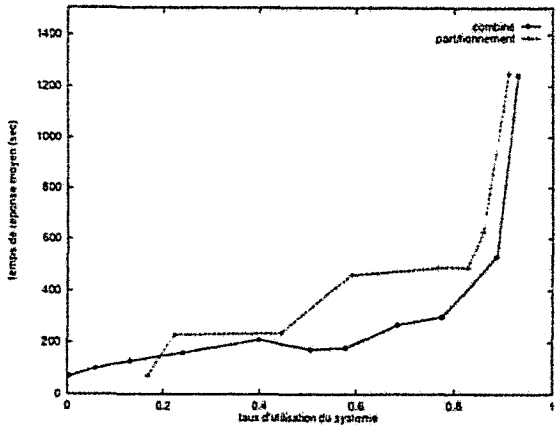
$$MRT_s = \frac{\sum_{i=1}^{NbApplis_s} RT_i}{NbApplis_s} \tag{5.7}$$

5.6.4 Résultats

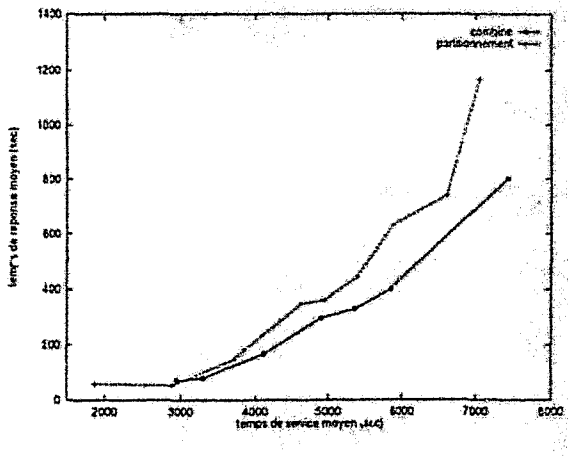
Les résultats des expérimentations conduites sont des moyennes de plusieurs exécutions pour chaque paramètre.

La figure 5.6a trace le temps de réponse moyen (par session) en fonction de l'utilisation du système pour les deux approches d'ordonnancement (spatiale et combinée). Les résultats montrent la supériorité de l'approche combinée qui a tendance à mieux utiliser les ressources grâce à l'ordonnancement temporel qui permet aux processus d'une application d'exploiter les périodes de communication et de synchronisation des autres processus. Ceci est d'autant plus clair lorsque le système est moyennement utilisé (entre 0.4 et 0.8 sur la figure). En revanche, lorsque le système est faiblement chargé (peu d'applications), l'approche spatiale obtient de meilleurs résultats étant donné que les conflits sont quasiment inexistants et que l'approche spatiale provoque moins de changements de contexte que l'approche combinée.

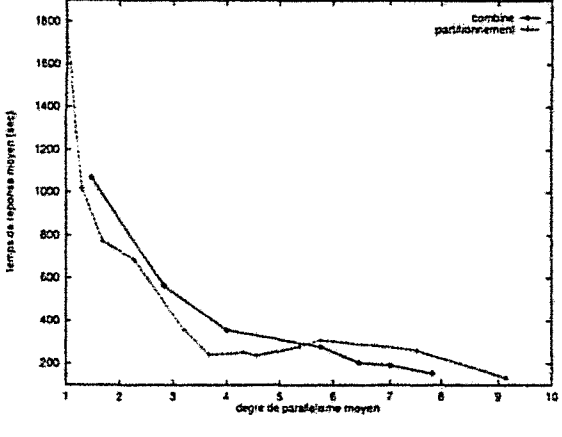
Sur la figure 5.6b, les mêmes remarques que celles concernant la figure précédente peuvent être déduites pour le temps de service moyen. En effet, plus la durée



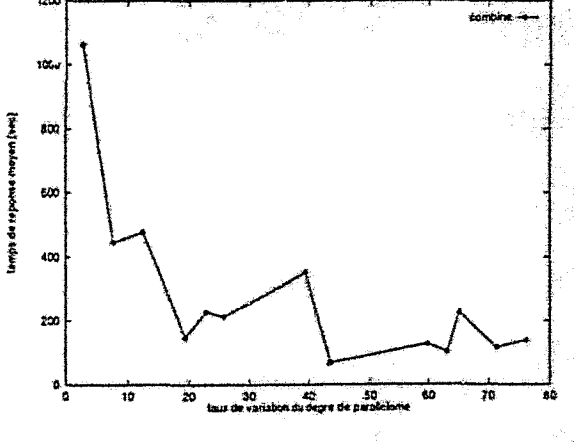
a) Temps de réponse moyen en fonction du taux d'utilisation du système.



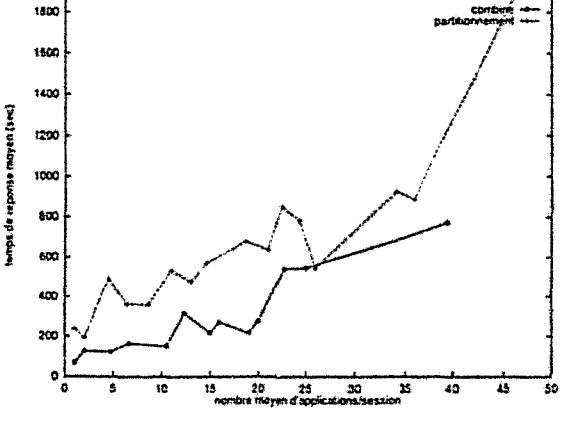
b) Temps de réponse moyen en fonction du temps de service moyen.



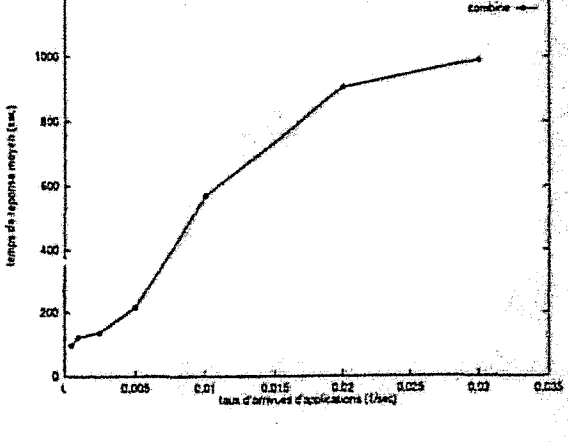
c) Temps de réponse moyen en fonction du degré de parallélisme moyen.



d) Temps de réponse moyen en fonction du taux de variation du degré de parallélisme.



e) Temps de réponse moyen en fonction du nombre d'applications.



f) Temps de réponse moyen en fonction du taux d'arrivées des applications (λ).

FIG. 5.6 - *Temps de réponse moyen en fonction des différents paramètres.*

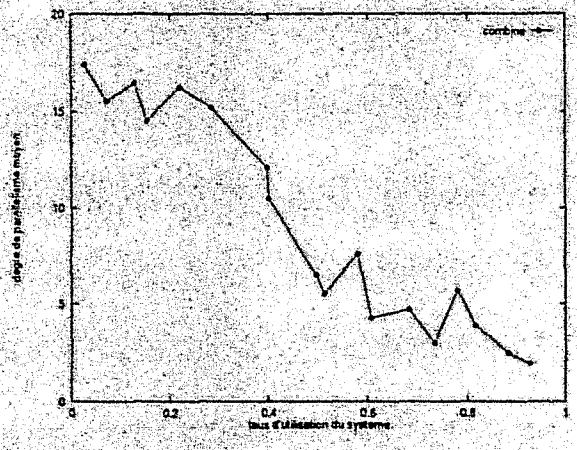


FIG. 5.7 - *Degré de parallélisme moyen en fonction du taux d'utilisation du système.*

moyenne de service est grande, plus le temps de réponse moyen est important. Etant donné que l'approche combinée parvient à mieux exploiter les ressources, elle obtient de meilleures performances sur ce point.

Contrairement à ce que l'on pouvait supposer, le degré de parallélisme moyen reflète la vitesse d'évolution de l'application étant donné que les applications considérées sont adaptatives. En effet, plus le degré de parallélisme moyen est important, plus la vitesse d'évolution des applications est grande. Sur la figure 5.6c, pour des degrés de parallélisme petits (entre 0 et 5 sur la figure), l'ordonnancement spatial est plus performant. Cela peut être expliqué par le fait que dans cette situation, soit le système est très chargé, soit le nombre d'applications est petit et la quantité de travail n'est pas importante. Dans ces cas, l'ordonnancement spatial est naturellement mieux adapté. De plus, vu l'exclusivité des partitions avec l'approche spatiale, le même degré de parallélisme permet à l'application de mieux progresser avec l'ordonnancement spatial qu'avec l'ordonnancement combiné.

La figure 5.7 trace l'évolution du degré de parallélisme moyen en fonction de l'utilisation du système pour l'approche combinée. Plus le taux d'utilisation du système augmente, plus la charge du système augmente (avec plus d'applications). Le partage et l'équité font que les applications obtiennent de plus modestes quantités de ressources. Ce qui provoque la diminution de leur degré de parallélisme.

Le taux de variation du degré de parallélisme traduit principalement l'hétérogénéité en temps de service des applications exécutées et la fluctuation de la charge du système. Sur la figure 5.6d, mis à part la situation du début, le temps de réponse moyen enregistré par l'approche combinée est presque insensible à l'évolution du taux de variation du degré de parallélisme, avec une légère baisse lorsque ce taux devient très important. Cela peut être expliqué par le fait qu'en général, lorsque des applications de différentes tailles sont exécutées, l'équité est toujours respectée. Ceci permet aux applications de plus courts temps de service de terminer rapidement. A noter que la charge externe a une influence sur les résultats.

La figure 5.6e illustre l'évolution du temps de réponse moyen en fonction du nombre d'applications générées par session pour les deux approches. Pour le même nombre d'applications, l'approche combinée réalise un meilleur temps de réponse moyen que l'approche spatiale, en particulier entre les valeurs 4 et 25.

Finalement, le taux d'arrivées des applications λ influence le temps de réponse moyen. Pour la même période de temps ($2400sec$), nous avons généré des applications selon une distribution poissonienne de taux d'arrivées différents λ. La figure 5.6f illustre l'évolution du temps de réponse moyen en fonction de ce paramètre pour l'approche combinée.

5.7 Conclusion

L'ordonnancement multi-application est une solution efficace pour la gestion et le partage de ressources dans les environnements dynamiques. Il permet notamment une optimisation globale suivant des objectifs communs à l'ensemble des applications et/ou au système. De plus, il permet d'assurer la propriété d'équité entre les applications dans l'allocation de ressources. Cette équité ne peut être respectée si la notion d'application parallèle et distribuée n'est pas considérée.

Le fait que les applications adaptatives disposent de leurs propres ordonnanceurs permet une coopération entre les ordonnanceurs des différentes applications avec l'ordonnanceur global. Cela permet une meilleure exploitation des ressources globalement. La notion d'ordonnanceur d'application nous a permis d'étendre le modèle à un modèle hiérarchique dynamique. Les modèles hiérarchiques offrent une modularité importante et une extensibilité aisée.

Les résultats de l'évaluation des performances obtenus montrent l'apport de l'ordonnancement multi-application. Les politiques combinées sont généralement meilleures étant donné la nature de l'environnement (réseaux de stations, charge imprévisible, etc) et les types d'applications. Des résultats similaires à ceux obtenus pour les applications parallèles adaptatives peuvent vraisemblablement être obtenus pour les applications classiques. Il serait également intéressant d'expérimenter une approche basée sur un partitionnement équitable. Cette approche est a priori plus intéressante que celles ne considérant pas l'équité.

Nous avons proposé un modèle d'environnement d'exécution avec une approche d'ordonnancement multi-application transparente, sans hypothèse sur les temps d'exécution des applications et où la tolérance aux fautes est considérée. La structure ouverte du système favorise son extension à d'autres modèles d'applications parallèles dynamiques. Cette possibilité est motivée par le fait que le modèle adaptatif peut s'avérer parfois inefficace pour la mise en œuvre de certaines applications. Ainsi, des applications non-adaptatives peuvent évoluer simultanément avec des applications adaptatives. La structure modulaire du système permet cette évolution d'une manière aisée et non-coûteuse.

L'extension du modèle aux réseaux longue distance peut passer par un modèle multi-groupe. Les délais de communication et les problèmes de sécurité et d'accès défavorisent l'exécution d'applications dans les groupes distants. Cependant, le modèle adaptatif possède des propriétés permettant une structuration hiérarchique de l'application dans le but de s'adapter à ces environnements.

Troisième partie

Applications parallèles adaptatives

Chapitre 6

Méthodologie de programmation parallèle adaptative

Les deux parties précédentes de cette thèse ont été consacrées à l'étude et au développement des aspects système de notre travail : la tolérance aux fautes, l'ordonnancement intra-application (applications adaptatives) et l'ordonnancement multi-application. Cette partie est consacrée à la présentation de la méthodologie de programmation et de la mise en œuvre d'applications adaptatives, ainsi qu'à l'étude de quelques applications développées.

Ce chapitre est consacré à l'étude du modèle de programmation parallèle adaptative et à l'utilisation de l'interface de programmation de l'environnement MARS. Le modèle sera mis en application sur une large classe de problèmes réels dans le chapitre 7 : applications issues du calcul scientifique et de l'optimisation combinatoire.

Il est connu que la complexité de la programmation s'accentue significativement avec le parallélisme. Les applications parallèles classiques sont souvent développées et exécutées sur des plateformes parallèles dédiées. La charge prévisible et l'homogénéité des nœuds (architecture et puissance de calcul) simplifient plus ou moins cette complexité. L'avènement des réseaux de stations et des clusters de processeurs a fait surgir d'autres problèmes accentuant de plus en plus la complexité de la mise en œuvre d'applications parallèles : hétérogénéité, multi-utilisateurs, multi-applications, charge imprévisible, etc.

Le modèle adaptatif induit à son tour des problèmes additionnels. Il contraint l'application parallèle à prévoir les fonctions à entreprendre lors de sa reconfiguration dynamique par le système. L'application adaptative doit gérer les événements de *repli* et de *dépli* [Haf98] et les communications sur le modèle de PM2 (LRPCs, ...). Le chapitre 4 a été consacré à la mise en place d'un outil de programmation permettant de réduire cette complexité, en prenant en charge la gestion et l'ordonnancement des tâches de l'application.

Dans notre modèle de programmation, une application parallèle est définie comme étant un ensemble de *tâches* dont les contraintes de précédence sont décrites par un graphe de dépendances [KA99]. Une tâche est un fragment de données sur lesquelles, un traitement séquentiel doit être effectué (voir chapitre 4).

6.1 Environnement d'exécution

Le développement et l'exécution d'une application parallèle adaptative s'effectuent dans le cadre du système d'ordonnancement multi-application présenté tout au long de la partie 2. La figure 6.1 illustre le modèle d'ordonnancement multi-application. Le système dispose d'un ensemble de nœuds de calcul (la configuration) qu'il alloue à un ensemble d'applications parallèles. Chaque application parallèle dispose d'un *générateur de tâches* et d'un *ordonnanceur applicatif* qui se charge de l'allocation des tâches de l'application sur les nœuds acquis et d'interagir avec l'ordonnanceur global pour la demande et la libération de ressources.

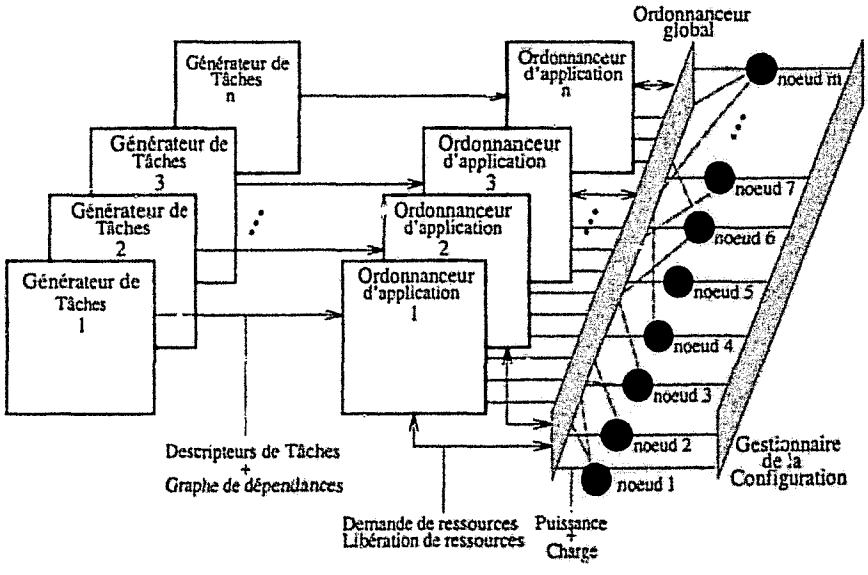


FIG. 6.1 - *Modèle d'ordonnancement multi-application.*

Le modèle adaptatif est un modèle dirigé par les données. Les composants du modèle (générateur de tâches, ordonnanceur applicatif, bibliothèque de programmation adaptative) ont été décrits en détail au chapitre 4. Les interactions avec l'ordonnanceur global ont été également spécifiées et discutées. Dans ce qui suit, nous nous focalisons sur l'étude de la méthodologie de programmation adaptative et sur l'utilisation de l'interface de programmation.

6.2 Environnement de développement

Avant de présenter la méthodologie de programmation adaptative, nous commençons par rappeler la structure d'une application parallèle adaptative et les problèmes associés.

6.2.1 Application parallèle adaptative

Une application adaptative MARS se compose d'un processus *maître* et d'un ensemble de processus *esclaves* (modèle *maître/esclaves*). Les esclaves exécutent un même programme (modèle SPMD) sans communication directe entre eux. Les applications adaptatives sont destinées essentiellement à faire du calcul parallèle dans les réseaux de stations et les clusters de processeurs. Les esclaves sont disposés chacun sur un nœud (figure 2.3, chapitre 2). La tâche du programmeur est, cependant, le développement du code du maître et de l'esclave.

6.2.2 Interface de programmation adaptative

Basée sur le modèle adaptatif, l'interface de programmation développée se charge de la plupart des problèmes liés aux aspects distribué et parallèle du système : allocation de tâches, communication, synchronisation, gestion de ressources, etc. Au niveau du maître, un espace de travail (sac de tâches) dans lequel, sont insérées les tâches générées, est prévu. Le générateur de tâches est invoqué par le programmeur lorsque ce dernier insère une nouvelle tâche ou par le *serveur de travail* lorsque les tâches sont créées dynamiquement. Un graphe de dépendances reflétant les contraintes de précédence entre les tâches est construit également par la bibliothèque. Le *serveur de travail* se charge de l'allocation des tâches aux esclaves et de la récupération des résultats et du travail intermédiaire en respectant les contraintes de précédence établies. Le choix de la tâche à allouer à un esclave, à un moment donné, est effectué par l'*ordonnanceur de l'application*. L'hétérogénéité des nœuds du point de vue de la puissance relative est prise en compte dans le but d'accélérer l'exécution de l'application.

L'esclave comporte essentiellement des threads de calcul. Il interagit avec le *serveur de travail* au niveau du maître pour demander du travail, pour retourner un résultat ou pour remettre un travail intermédiaire.

6.2.2.1 Structure de l'espace de travail

L'espace de travail, tenu par le maître, contient toutes les tâches non terminées de l'application. L'insertion des tâches peut être statique, effectuée au lancement de l'application, ou dynamique, résultant de l'exécution d'autres tâches ou d'autres événements liés à l'application ou à l'environnement.

Les tâches sont identifiées par des entiers (identificateurs) permettant à l'application de les distinguer. A chaque insertion d'une tâche dans l'espace de travail, son identificateur est retourné à l'application. Les identificateurs sont utilisés également pour spécifier les dépendances d'une tâche (l'ensemble des tâches dont dépend la tâche insérée). Ces dépendances permettent à la bibliothèque de construire un graphe de dépendances dynamique (*Directed Acyclic Graph, DAG*) qui assure l'ordre d'exécution des tâches. Notons que la vérification de la propriété DAG du graphe de dépendances est à la charge de l'application. En effet, vérifier cette propriété systématiquement à l'insertion de chaque tâche par la bibliothèque peut induire un surcoût considérable.

La construction incrémentale de l'espace de travail permet d'un côté la prise en charge d'applications irrégulières, dont l'espace de travail ne peut pas être construit entièrement au lancement de l'application. D'un autre côté, le problème d'insuffisance d'espace mémoire peut être résolu en insérant les tâches de l'application en plusieurs étapes. Une primitive est fournie par la bibliothèque afin de récupérer l'espace occupé par une tâche dès sa terminaison.

6.2.2.2 Structure du maître

Dans une application adaptative, le maître (serveur de travail + ordonnanceur applicatif) joue un rôle de coordinateur entre les esclaves et d'interlocuteur avec l'ordonnanceur global. Sur le plan de la méthodologie de développement, l'interface de programmation permet à l'utilisateur de construire l'espace de travail (initialement et dynamiquement) et de reporter les modifications nécessaires pour la cohérence des données. L'algorithme décrit par la figure 6.2 présente la structure générique du maître. Les données de la tâche et les résultats de son exécution sont stockés dans l'espace de travail. Toutefois, si l'utilisateur souhaite modifier ces données ou effectuer certains traitements, les actions nécessaires doivent être spécifiées sous forme de fonctions que la bibliothèque déclenche automatiquement à l'allocation d'une tâche, à la réception d'une tâche partielle ou au retour des résultats finaux. Ces fonctions sont :

- *work_func :* cette fonction comporte les modifications éventuelles des données de la tâche qui doivent être effectuées au moment de son allocation (dépli) ;
- *pendwork_func :* cette fonction permet à l'utilisateur de reporter des modifications éventuelles sur les données de la tâche ou de générer certains événements dans le cas de retour d'une tâche partiellement traitée (repli, sauvegarde, retrait de tâche) ;
- *res_func :* cette fonction est exécutée à la fin de l'exécution de chaque tâche. Elle est primordiale pour les applications adaptatives dont les tâches sont générées dynamiquement. C'est généralement après la terminaison d'une tâche, que les conditions nécessaires pour la création d'autres tâches sont vérifiées (par

exemple, après la terminaison de la dernière tâche d'une série de tâches ou suivant le résultat de l'exécution de la tâche). Cette fonction permet d'évaluer dynamiquement ces critères et de générer éventuellement de nouvelles tâches si les conditions sont vérifiées. Lorsque l'application n'a plus de tâches à insérer, elle peut également l'indiquer afin de libérer les nœuds éventuellement en attente et d'améliorer l'utilisation des ressources par la même occasion. Dès la terminaison de la dernière tâche de l'application, celle-ci peut l'indiquer afin que le système procède à sa terminaison.

La structure *Order_Struct*, avec laquelle ces fonctions sont paramétrées, comporte les données de la tâche (*work*, *pending_work* et *results*) ainsi que d'autres informations d'état de la tâche (identificateur, nouvelle tâche ou partielle).

6.2.2.3 Structure de l'esclave

Un esclave est un processus de calcul contenant un ensemble de threads chargés du traitement des tâches de l'application. Tout au long de son existence, l'esclave demande continuellement du travail au maître, invoque le thread spécialisé dans le traitement du type de la tâche obtenue et retourne les résultats au maître. La figure 6.3 illustre la structure générique de l'esclave. Celui-ci se compose de :

- *la fonction cleanup :* cette fonction est invoquée automatiquement en cas de *repli* ou de sauvegarde de l'application afin de retourner la tâche partiellement traitée au maître ;

- *un ensemble de threads de traitement :* chaque thread est spécialisé dans le traitement d'un type de tâche particulier. Lorsqu'un thread est lancé pour traiter une tâche, il effectue le traitement approprié et retourne le résultat au maître en utilisant une fonction de la bibliothèque.

L'annexe A présente une description complète des primitives de l'interface de programmation parallèle adaptative.

6.3 Etapes d'exécution d'une application adaptative

L'application est lancée par l'exécution du processus maître sur la console d'une station choisie par l'utilisateur. L'application exécute certaines actions comme l'enrôlement et la demande de ressources et réagit à certains événements tels que le repli et le déplì.

```
void work_func(Order_Struct *arg)
{
/* Modifications éventuelles des données avant l'allocation de la tâche */
}
void pendwork_func(Order_Struct *arg)
{
/* Modifications éventuelles des données de la tâche retournée */
}
void res_func(Order_Struct *arg)
{
/* Création et insertion des tâches */
if (conditions de création de nouvelles tâches vérifiées)
     mars_inserttask(work, pending_work, result, func_name, dependencies);
if (plus de tâches à insérer)
     mars_endtasks();
if (dernière tâche de l'application)
     mars_exit();
}
int main(int argc, char **argv)
{ /* Enrôlement au système */
mars_init(MASTER, NULL);

/* Spécification des paramètres de tolérance aux fautes */
mars_setckpt_mode(STATIC_PERIOD);
mars_setckpt_period(300);

/* Positionnement des fonctions de prise en charge dynamique des tâches */
mars_swork_func(work_func);
mars_spendwork_func(pendwork_func);
mars_sres_func(res_func);

/* Création et insertion de la première vague de tâches */
mars_inserttask(work, ...);

/* Demande de ressources auprès du système */
mars_spawn(WORKER_NAME, ...);

/* Attente de terminaison des calculs */
mars_waitexit();
}
```

FIG. 6.2 - *Architecture générale du maître.*

```
void cleanup(any_t arg)
{
/* Préparer les résultats intermédiaires et le travail restant */
mars_putbackwork(partial_task);
}
any_t treat_thread1(any_t work)
{
/* Traitement de la tâche obtenue */
/* Renvoie des résultats au maître */
mars_putbackresults(results);
}
int main(int argc, char **argv)
{
/* Enrôlement au système */
mars_init(WORKER, NULL);

/* Spécification des threads de traitement */
mars_sworkerthread(treat_thread1, params, func_name);
...
/* Positionnement de la fonction cleanup */
mars_sputbackwork_func(cleanup, NULL);

/* Boucle de l'esclave */
for (;;) {
    res = mars_getwork(&work, &worker_thread);
    switch(res) {
        case NEW_WORK:
        case PENDING_WORK:
            mars_startworkerthread(worker_thread, &work);
            break;
        case NO_WORK:
            break;
    }
}
}
```

FIG. 6.3 - *Architecture générale d'un esclave.*

6.3.1 Enrôlement

Afin de pouvoir être identifiée par le système global, l'application (le maître) doit initialement s'enrôler au système. Cette opération permet au module d'ordonnancement de l'application d'envoyer à l'ordonnanceur global quelques informations concernant l'application : utilisateur, nœud de connexion, etc. Après l'enrôlement, le maître peut spécifier éventuellement les paramètres du mécanisme de sauvegarde/reprise (voir partie I). Les fonctions de prise en charge dynamique des tâches définies précédemment doivent également être positionnées. Avant d'envoyer une demande de ressources à l'ordonnanceur global, le premier ensemble de tâches peut être inséré. Cela permet à l'ordonnanceur applicatif d'opérer pendant l'attente de la réponse de l'ordonnanceur global. Dès l'obtention de la première configuration de ressources, les opérations de création d'esclaves et d'allocation de tâches peuvent commencer. C'est à ce moment que l'ordonnanceur de l'application effectue ses premières décisions d'allocation.

De son côté, au début de son exécution, l'esclave s'enrôle au système, positionne la fonction de repli, spécifie les threads de calcul et entame son cycle de travail : demande de travail et déclenchement des threads de traitement. Notons que les esclaves ne peuvent traiter qu'une tâche à la fois. Ce choix est principalement justifié par le type d'applications prises en charge (applications effectuant du calcul intensif avec peu d'opérations d'entrées/sorties).

6.3.2 Demande de ressources

Après les opérations d'initialisation, l'application doit exprimer ses besoins en ressources auprès de l'ordonnanceur global. La demande de ressources est effectuée par l'exécution de la primitive *mars_spawn* et s'accompagne par la spécification du degré de parallélisme *minimum* et *maximum* que peut atteindre l'application adaptative. L'ordonnanceur de l'application complète la requête par d'autres informations, telles que la classe de ressources demandées qui sera des "stations de travail sans considération de l'architecture : WORKSTATION", et l'envoie à l'ordonnanceur global. Ce dernier enregistre la requête qui sera traitée avec celles d'autres applications comme nous l'avons présenté au chapitre 5.

De son côté, l'ordonnanceur de l'application détermine le moment de libération des ressources au système global. De plus, il interagit avec ce dernier et lui communique régulièrement l'état des besoins de l'application.

6.3.3 Repli de l'application

Le repli de l'application est entrepris lorsque le système réclame un ou plusieurs nœuds exploités par l'application. Les raisons d'une telle décision peuvent être induites par la surcharge du nœud, la présence du propriétaire ou pour le partage de

ressources avec d'autres applications. L'opération consiste à retirer les esclaves en question. Ce retrait doit s'accompagner par la récupération des tâches partiellement traitées par le maître.

Au niveau de l'esclave, l'ordre de repli déclenche automatiquement la fonction *cleanup* qui rassemble les données de la tâche en cours de traitement si c'est nécessaire et la retourne au maître. A ce stade, la mission de l'utilisateur dans cette opération s'achève. De son côté, la bibliothèque gère les problèmes additionnels et assure la cohérence de l'opération de repli quelle que soit la situation :

- *repli sans tâche :* cette situation peut être créée par l'arrivée de l'ordre de repli aussitôt après la création de l'esclave et avant de pouvoir obtenir du travail ou lorsque l'esclave est en attente de la disponibilité du travail. Le repli de l'esclave ne déclenche pas dans ce cas la fonction *cleanup* et se réduit simplement à la terminaison de l'esclave ;

- *préservation de cohérence :* le repli est un événement système décidé et ordonné par l'ordonnanceur global. Celui-ci n'a pas connaissance de l'état des calculs et des données internes à l'application. Ainsi, l'arrivée d'un tel événement peut rencontrer les données de la tâche dans un état incohérent (permutation de deux éléments en cours, etc). La première version de MARS avait identifié et remédié à ce problème à travers deux primitives *mars_lock_fold* et *mars_unlock_fold*. Ces primitives permettent d'inhiber le repli, le temps de finaliser ce genre de manipulations (sections critiques).

Permettre à l'utilisateur d'empêcher le repli n'est pas une solution dépourvue de problèmes. En effet, les sections critiques peuvent être assez longues au point de gêner les utilisateurs interactifs. Une solution simple consiste à diminuer la priorité Unix de l'esclave au moment du repli. Cela peut s'avérer insuffisant car le processus consommera toujours du temps CPU et occupera de l'espace mémoire. La complicité de l'utilisateur est vivement encouragée. Celui-ci peut fragmenter ses longues sections critiques en sections plus courtes ou travailler sur des données temporaires et mettre à jour régulièrement les données sur lesquelles portera un éventuel repli.

6.3.4 Dépli de l'application

A chaque allocation d'un nouveau nœud à l'application par l'ordonnanceur global, l'application pourra étendre son degré de parallélisme. On parle ainsi d'un élargissement ou d'un dépli de l'application. Un nouvel esclave est alors créé sur le nœud fourni comme dans le cas de la phase d'initialisation de l'application. L'esclave doit s'enrôler au système (primitive *mars_init*). La bibliothèque communique alors toutes les informations d'identité de l'esclave au maître qui se charge de les faire connaître à l'ordonnanceur global. L'esclave doit ensuite spécifier les threads de traitement qu'il comporte. Cela permet à la bibliothèque de construire une table de threads qui

servira à déterminer automatiquement le thread qui doit être déclenché à la réception d'une tâche[1]. La fonction de repli *cleanup* est ensuite positionnée avant toute demande de travail. L'esclave entre ensuite dans un processus cyclique: demande de travail, déclenchement de thread approprié, qui durera toute sa durée de vie. A la demande d'un travail auprès du maître, plusieurs situations peuvent se présenter:

- *pas de tâche :* si l'espace de travail du maître est momentanément ou définitivement vide, l'appel à la primitive *get_work* ne retourne rien. La suite des opérations est gérée par la bibliothèque suivant les instructions de l'ordonnanceur de l'application. Deux situations peuvent ainsi se présenter:
  - *aucune tâche ne sera générée :* l'application doit se terminer, l'esclave est alors terminé automatiquement,
  - *pas de tâche temporairement:* des tâches pourront être créées ultérieurement, dans ce cas l'ordonnanceur décidera si le nœud doit être restitué au système ou gardé un certain temps en attente de travail. Si l'ordonnanceur opte pour la première alternative, l'esclave est terminé comme dans le cas précédent. Sinon, il est mis en attente et sera débloqué ultérieurement par l'ordonnanceur. Dans ce dernier cas, *get_work* retournera après l'ordre de déblocage afin de permettre à l'esclave de réclamer du travail de nouveau;
- *il y a du travail:* l'esclave obtient une tâche, déclenche le thread de traitement approprié dont le nom de la fonction est retourné par *get_work*. Le thread de traitement effectue les calculs nécessaires avant de retourner les résultats via la primitive *put_back_results*. Si aucun résultat n'est rendu, la primitive informe le serveur de travail afin de marquer la fin de la tâche.

6.3.5 Retrait de tâches

Le retrait d'une tâche permet au maître de récupérer les résultats partiels et d'exploiter ainsi le travail effectué par l'esclave sur la tâche. Dans notre environnement, le retrait d'une tâche peut provenir de multiples causes ou événements qui peuvent être liés au système ou à l'application elle-même. Ces événements sont confondus par l'application qui ne doit pas les distinguer:

- *repli de l'application :* dans ce cas, le nœud doit être restitué au système. L'esclave retourne le travail restant et les résultats partiels avant de terminer;
- *décision de l'ordonnanceur de l'application:* à des fins d'ordonnancement et d'optimisation de l'exécution de l'application, l'ordonnanceur peut être amené à désallouer une tâche, soit au profit d'une autre, soit en vue de la déplacer sur un autre nœud plus puissant dans le but d'accélérer son exécution;

1. Les noms de fonctions doivent correspondre aux noms spécifiés au moment d'insertion des tâches dans l'espace de travail.

- *opération de sauvegarde :* la tâche partielle doit être retournée au maître afin de pouvoir continuer son exécution ultérieurement en cas de défaillance ;
- *décision de l'application :* dans certains cas, l'application parallèle doit effectuer des points de synchronisation afin d'échanger des données ou de vérifier certaines propriétés. Cela peut être effectué de deux manières différentes : par l'esclave ou par le maître. En effet, l'esclave peut décider à partir de certaines informations comprises dans la tâche (un nombre d'itérations, un temps d'exécution, etc) d'arrêter le travail et de remettre la tâche partiellement traitée au maître. Une primitive de la bibliothèque *mars_retreat* est prévue à cet effet. Le maître peut également constituer une barrière de synchronisation en rassemblant explicitement toutes les tâches actives de l'application (*retreat_active_tasks*).

6.3.6 Découpage du travail et construction de l'application adaptative

Développer une application selon le modèle de programmation proposé consiste à découper le travail que doit réaliser l'application en unités élémentaires : *les tâches*. La tâche constitue ainsi l'élément de base de l'application adaptative dont l'exécution est séquentielle (grain). L'efficacité est définie par le facteur $\frac{t_E}{t_C+t_E}$ où t_E et t_C représentent respectivement les coûts d'exécution et de communication d'une tâche en moyenne. Le temps d'exécution total de l'application est, par conséquent, fonction de ces paramètres : $t_P = f(t_E, t_C)$. Dans de nombreuses applications, les temps de communication et de calcul sont corrélés. En effet, plus on augmente le grain plus la quantité de données (travail et résultats) croît. En d'autres termes, le temps de communication d'une tâche dépend également de sa durée d'exécution ($t_C = f(t_E)$). Cette situation conduit à conclure que pour une application donnée et pour une plateforme donnée, il existe une granularité optimale qui donne un temps d'exécution de l'application parallèle optimal. Dans [MTP99], une étude expérimentale est effectuée pour la définition du grain optimal de l'application de Gauss-Jordan par blocs exécutée sur un réseau de stations[2].

Un autre problème, relevant du découpage du travail, concerne l'aspect temporel du processus de génération des tâches. L'espace de travail peut être généré statiquement au lancement de l'application ou dynamiquement suivant l'évolution des calculs. Le découpage statique impose une granularité indépendante de la configuration matérielle sur laquelle l'application sera exécutée. La granularité prédéfinie peut dégrader les performances de l'application si le coût d'exécution des tâches ne correspond pas à la puissance et à l'état de charge des nœuds sur lesquelles, elles sont allouées.

L'ordonnanceur de l'application est en mesure de remédier à ce problème d'une manière transparente à l'application. En effet, en cas de disponibilité de nœuds

2. Sur le réseau du Laboratoire d'Informatique Fondamentale de Lille (LIFL).

puissants, les tâches exécutées sur des nœuds plus lents sont transférées sur les nœuds puissants. Cela permet de résoudre le problème de la granularité d'une manière transparente et de permettre à l'utilisateur de s'affranchir des problèmes complexes liés au système (architecture, état de charge, etc).

En résumé, l'espace de travail peut être connu au lancement de l'application ou au fur et à mesure de son évolution. Le partitionnement peut être statique ou dynamique. Les tâches peuvent être homogènes (de même coût d'exécution) ou hétérogènes (coûts d'exécution différents). Le partitionnement dynamique selon une granularité dépendante de l'état du système est une tâche complexe.

Grâce à l'ordonnanceur de l'application, un partitionnement statique en tâches, dont le temps d'exécution est indépendant des caractéristiques statiques ou dynamiques de la configuration matérielle, suffit. Le partitionnement dynamique peut être aisément utilisé dans le cas où l'espace de travail n'est pas connu a priori ou à des fins d'optimisation de l'espace mémoire.

6.4 Exemple : Multiplication de matrices

Dans cette section, nous allons illustrer la méthodologie de programmation parallèle adaptative sur une application simple : la multiplication de matrices. Le programme consiste à effectuer la multiplication de deux matrices A et B et à ranger le résultat dans une troisième matrice C.

Une manière simple de mettre en œuvre cette application est de découper le travail à faire (calcul de la matrice C) en plusieurs partitions. Chaque partition comporte un ensemble de colonnes. Les partitions peuvent être ainsi calculées en parallèle. L'espace de travail est connu a priori et le découpage est statique.

Au début, les deux matrices sont chargées dans l'espace du maître et vont constituer, avec les descripteurs des tâches, l'espace de travail. Le découpage du travail est statique et une tâche est définie comme étant le calcul d'un ensemble prédéfini de colonnes de la matrice résultat. Les données de la tâche sont : la matrice A et un ensemble de colonnes de la matrice B (figure 6.4). La granularité doit être moyenne, voire fine, afin d'offrir un parallélisme suffisant en cas de disponibilité de ressources. Cependant, elle ne doit pas être excessivement fine afin de ne pas pénaliser les perfomances de l'application. La matrice A est chargée par l'esclave initialement à partir d'un fichier et peut ne pas apparaître au niveau de l'espace du maître.

Les tâches sont ainsi homogènes et leur coût d'exécution est prédéfini. L'étape suivante consiste à définir les informations que doit comprendre une tâche. Comme nous l'avons présenté au chapitre 4, une tâche comporte trois structures définissant le travail initial, le travail inachevé et les résultats finaux. Les trois structures peuvent être confondues. Dans cette application, seules les structures du travail et du travail intermédiaire sont confondues. Afin de pouvoir garder trace de l'état de la tâche, les valeurs des indices j et i mentionnant les colonnes et les lignes de la tâche déjà

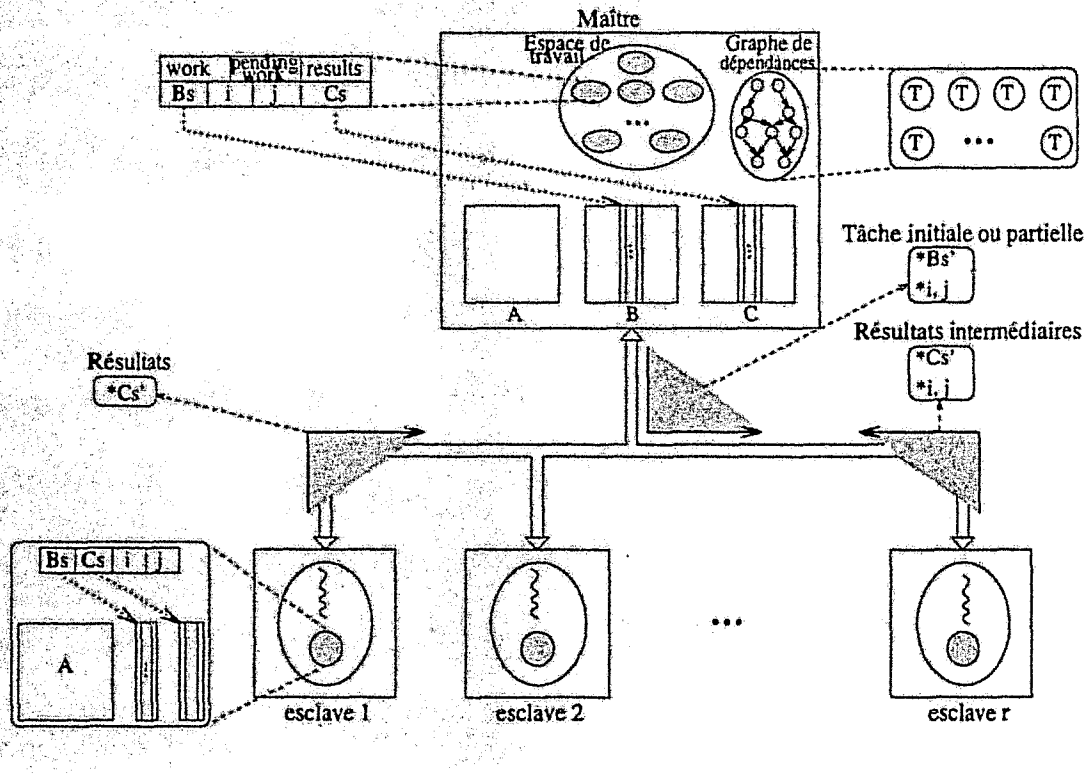


FIG. 6.4 - *Multiplication de matrices parallèle adaptative.*

traitées sont conservées. La stucture des résultats pointe sur le début de la partition de C affectée à la tâche.

L'étape suivante consiste à définir le contexte de la tâche partielle en cas de retrait (repli, sauvegarde, retrait explicite, etc). Ce contexte se compose de l'ensemble de colonnes de Cs calculées ($Cs' \subseteq Cs$) et les indices de colonne et de ligne correspondants (j et i) (figure 6.4). Le résultat final d'une tâche est l'ensemble des colonnes calculées ($Cs' \subseteq Cs$) de i, j à la fin. Les valeurs de i et j conservées au niveau de l'espace de travail permettent également de ranger le résultat à l'endroit approprié. L'utilisateur est ainsi responsable de l'endroit et de la manière dont les données sont empaquetées et désempaquetées.

Les données envoyées à un esclave sont l'ensemble de colonnes Bs ou $Bs' \subseteq Bs$ dans le cas d'une tâche partielle. Les indices i et j permettent de définir la position des colonnes.

Chapitre 7

Applications issues du calcul scientifique et de l'optimisation combinatoire

Dans le chapitre précédent, nous avons passé en revue la méthodologie de programmation parallèle adaptative et l'interface de programmation conçue à cet effet. L'ordonnancement adaptatif d'applications parallèles dans les réseaux de stations concerne essentiellement les applications de longue durée de vie. Parmi les domaines où les applications exigent des puissances de calcul importantes, figurent le calcul scientifique et l'optimisation combinatoire. De nombreux secteurs économiques ont souvent recours aux applications de calcul scientifique et d'optimisation (météorologie, télécommunication, médecine, etc). Dans ce qui suit, nous présentons leur mise en œuvre sur notre environnement parallèle adaptatif MARS. Quelques résultats de performances seront également présentés.

7.1 Applications issues de l'optimisation combinatoire

Certains problèmes d'optimisation combinatoire, les problèmes NP-difficiles, sont réputés être complexes à résoudre et par conséquent très exigents en temps de calcul. Les *métaheuristiques* sont des méthodes efficaces pour la résolution de ces problèmes. Leur intérêt principal est leur capacité à résoudre des problèmes de grande taille dans des temps raisonnables. Dans notre équipe, un intérêt particulier est donné à ces méthodes [Tal98, Haf98, Bac99].

La figure 7.1 énumère les métaheuristiques les plus connues. Lorsque l'espace de recherche n'est pas convexe, les méthodes de recherche locale itérative (*Hill-climbing*) atteignent rapidement leur limite [Bac99]. Les méthodes de *recuit simulé* [KGV83], les algorithmes *génétiques* [Hol75] et la recherche *tabou* [Glo89] ont été utilisés avec succès dans la résolution de plusieurs problèmes d'optimisation. L'hy-

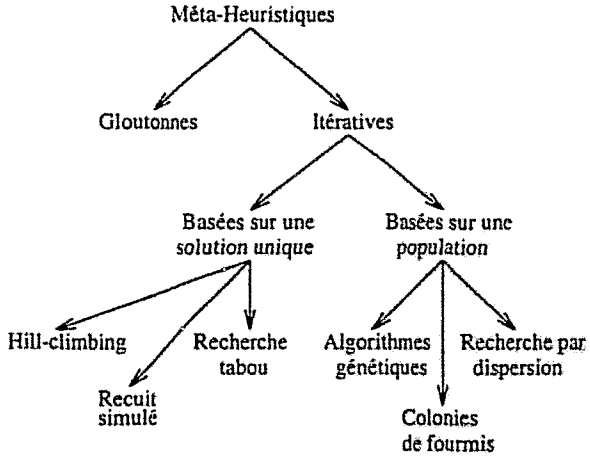


FIG. 7.1 - *Synthèse des métaheuristiques connues.*

bridation des méthodes d'exploration globale et des heuristiques de recherche locale donnent souvent des méthodes plus robustes et plus efficaces [Tal98].

Ici, notre intérêt pour les métaheuristiques porte sur le parallélisme adaptatif. Par conséquent, nous nous focalisons sur les propriétés parallèles adaptatives des applications de ce domaine, que nous avons développées ou portées sur notre système. Nous présentons la méthode tabou et une hybridation du recuit simulé avec une méthode de recherche locale pour le problème de partitionnement de graphes.

7.1.1 Méta-heuristiques et parallélisme adaptatif

Traditionnellement, la parallélisation des méta-heuristiques est effectuée de deux manières différentes [Tal98] : par décomposition du domaine de recherche ou par exécution de multiples algorithmes en parallèle.

La décomposition du domaine consiste à partitionner l'espace de recherche ou le voisinage à évaluer en plusieurs partitions et à exécuter un algorithme de recherche sur chaque partition [Tai93]. Le parallélisme multiple consiste à lancer plusieurs recherches sur le même espace simultanément [CTG93, MO94]. Les approches de cette classe diffèrent suivant la solution initiale associée à chaque algorithme et la manière dont les recherches parallèles coopèrent.

Les méthodes basées sur le partitionnement du domaine sont relativement complexes à mettre en œuvre. De plus, elles impliquent souvent des contraintes de synchronisation fortes et induisent des surcoûts de communication importants. Les algorithmes multiples remédient à ces problèmes, mais peuvent manquer d'efficacité faute de coopération suffisante ou d'ajustement adéquat des paramètres. Dans un environnement de stations de travail où la communication est coûteuse et le parallélisme est opportuniste, la deuxième catégorie d'approches semble la mieux adaptée.

7.1.2 Recherche tabou pour le problème d'affectation quadratique

7.1.2.1 Problème d'affectation quadratique

Un problème d'optimisation combinatoire peut être défini par la spécification d'une paire (X, f), où X représente l'espace de recherche et f la fonction objectif permettant d'évaluer le coût d'une solution, $f : X \dashrightarrow \mathbb{R}$ [TGHK98]. Un voisinage N est une application $N : X \to P(X)$, qui associe à chaque solution $S \in X$ un sous-ensemble $N(S) \subseteq X$ des voisins de S.

Le problème d'affectation quadratique (QAP) est défini par les données suivantes :

- un ensemble d'objets $O = \{O_0, O_1, \ldots, O_{n-1}\}$ et un ensemble de positions $L = \{L_0, L_1, \ldots, L_{n-1}\}$;
- une matrice de flux C, dans laquelle, chaque élément c_{ij} représente le flux entre les objets O_i et O_j, et une matrice de distances D, dans laquelle, chaque élément d_{ij} dénote la distance entre les positions L_i et L_j ;

Le problème consiste à trouver une bijection $M : O \longrightarrow L$, en positionnant les objets, qui minimise la fonction objectif $f = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c_{ij} d_{M(i)M(j)}$. Une solution est représentée par une permutation de n entiers $s = (l_0, l_1, ..., l_{n-1})$ où l_i dénote la position de l'objet O_i. Pour générer une solution voisine, deux éléments de la permutation sont interchangés.

7.1.2.2 Recherche tabou

La méthode *tabou (tabu search)* est une méta-heuristique basée sur l'amélioration itérative locale d'une solution [Glo89]. A chaque itération, la meilleure solution du voisinage est choisie. La recherche tabou est dotée d'une *liste tabou* contenant toutes les solutions récemment visitées afin d'interdire leur choix. Le remplacement de la solution courante est réalisé même si celle-ci est de meilleure qualité que la meilleure solution voisine. Ceci permet d'échapper aux minimas locaux.

D'autres stratégies ont été introduites afin d'améliorer l'efficacité de la méthode : les critères d'aspiration, la diversification, l'intensification, etc. Ces stratégies permettent de diversifier la recherche en explorant des régions de l'espace de recherche peu visitées et d'intensifier les recherches sur une région jugée prometteuse. La figure 7.2 présente l'algorithme de base de la méthode *tabou* dont les paramètres sont :

- la solution initiale S_0 ;
- la taille de la liste tabou T_i ;
- le nombre maximum d'itérations sans amélioration de la fonction coût *max_iter*.

Initialisation
$S = S_0$ /* Solution courante */
$nb\_iter = 0$ /* Itération courante */
$best\_sol = S_0$ /* Meilleure solution trouvée */
$best\_iter = 0$ /* Numéro d'itération de la meilleure solution */
$T_t = \Phi$
Itération : Tant que $(nb\_iter - best\_iter < max\_iter)$ {
$nb\_iter = nb\_iter + 1$;
Générer N^* le voisinage de S : solutions $S' \in N(S)$ non tabou ou satisfaisant les critères d'aspiration.
Choisir une solution S^* minimisant f dans N^*.
Mettre à jour T_t.
Si $f(S^*) \leq f(best\_sol)$ { $best\_sol = S^*$; $best\_iter = nb\_iter$; }
$S = S^*$;
}

FIG. 7.2 - *Recherche tabou de base.*

La mémoire à court terme (liste tabou) se compose des paires d'éléments qui ne peuvent pas être interchangés. La taille de la liste tabou est dynamique et varie de $\frac{n}{2}$ à $\frac{3n}{2}$ [Bac99].

7.1.2.3 Recherche tabou parallèle adaptative

La recherche tabou parallèle implémentée consiste à lancer plusieurs recherches tabou séquentielles sans communication entre elles. En effet, chaque recherche tabou est paramétrée par une solution initiale générée aléatoirement et une taille de la liste tabou différente. Ceci permet d'assurer la diversité de la recherche entre les tabous parallèles. La fonction ou le thread de traitement d'une tâche est un algorithme de recherche tabou séquentielle (application à gros grain).

La figure 7.3 illustre la structure de la recherche tabou parallèle adaptative. Une tâche comprend toutes les informations nécessaires à une recherche tabou séquentielle. Elle est composée initialement de :

- la solution initiale ;
- la taille de la liste tabou.

Durant son exécution, une tâche peut être interrompue et retournée au maître. Le travail restant appelé aussi *solution intermédiaire* comporte :

- la solution courante ;
- l'itération courante ;

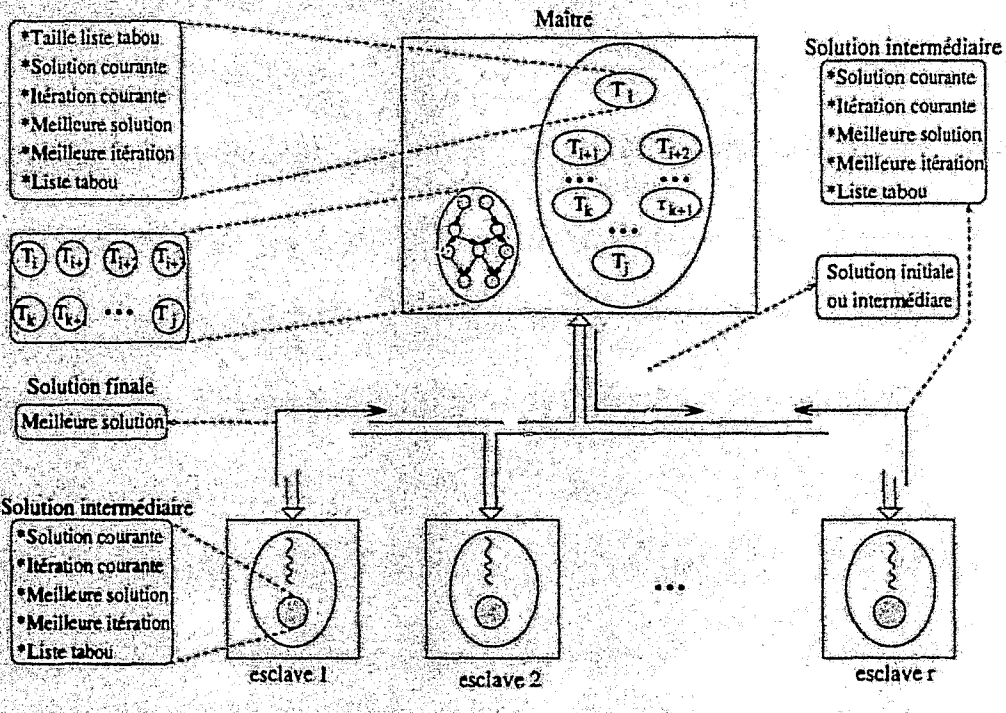


FIG. 7.3 - *Mise en œuvre de la recherche tabou parallèle adaptative.*

- la meilleure solution trouvée;
- le numéro d'itération durant laquelle cette solution a été trouvée;
- le contenu de la liste tabou.

7.1.2.4 Résultats expérimentaux

Dans ce paragraphe, nous présentons quelques résultats de performances de l'application sous l'environnement MARS. La configuration matérielle se compose d'environ 20 stations de travail. Le tableau 7.1 résume les résultats d'exécution de l'application sans d'autres applications MARS, mais avec de la charge externe. Les indicateurs de performances utilisés sont :

$R(A)$: temps de réponse de l'application (temps d'exécution parallèle en secondes).
$T(A)$: temps de service (équivalent au temps séquentiel en secondes).
$Sched$: surcoût de l'ordonnancement au niveau de l'application (en secondes).
N_{base} : nombre (normalisé) de nœuds utilisés par l'application.
$S(A)$: accélération résultante.
$U(A)$: taux d'utilisation du système par l'application.
UFd : nombre de nœuds crédités à l'application (y compris ceux de la première allocation).
Fd : nombre de nœuds débités.

Ret : nombre de retraits de tâches par l'ordonnanceur.
Fl : nombre de défaillances des esclaves.
APD : degré de parallélisme moyen pondéré par le temps.
$Eff(A)$: efficacité résultante.

$R(A)$	$T(A)$	*Sched*	N_{base}	$S(A)$	$U(A)$	UFd	Fd	*Ret*	*Fl*	APD	$Eff(A)$
104.93	575.94	0.0012	7.92	5.49	0.68	7	0	0	0	5.71	0.69
2613.67	17346.14	0.0265	7.65	6.64	0.84	15	1	0	0	5.50	0.87

TAB. 7.1 - *Résultats obtenus pour la recherche tabou.*

Le tableau 7.1 présente les résultats obtenus pour deux exécutions du *tabou* parallèle adaptatif, une exécution étant beaucoup plus longue que l'autre. La remarque principale est que l'efficacité réalisée est très importante malgré la charge externe. De plus, l'efficacité augmente avec le grain (87 % contre 69 %). Le surcoût induit par l'ordonnancement est négligeable. L'excellente efficacité obtenue reflète l'influence de la granularité sur les performances des applications dans ces environnements. Le taux d'utilisation du système par l'application $U(A)$ traduit les temps d'attente des esclaves, mais également l'influence des processus externes qui partagent les nœuds sur lesquels s'exécutent les esclaves de l'application.

7.1.3 Recuit simulé pour le partitionnement de graphes

Le partitionnement de graphes est un problème d'optimisation combinatoire qui possède plusieurs applications : placement VLSI, placement de processus, etc. Considérons un graphe non orienté $G = (V, E)$, où V, de taille N, représente l'ensemble des sommets et E l'ensemble des arêtes. Le problème de partitionnement consiste à trouver une partition de V en deux sous-ensembles A et B de même taille de telle sorte que le nombre d'arêtes croisant les partitions soit minimal (bi-partitionnement).

Le problème consiste donc à trouver les sous-ensembles A et B minimisant la fonction coût f :

$$f = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \delta_{ij}. \qquad \delta_{ij} = \begin{cases} 1 & \text{si } i \in A \wedge j \in B \wedge (i,j) \in E \\ 0 & \textit{sinon} \end{cases}$$

Le problème de partitionnement de graphes (GPP) est un problème NP-difficile et les méthodes exactes ne peuvent être utilisées pour les instances de grande taille. Dans le domaine des méthodes approchées, les algorithmes les plus utilisés sont la recherche locale de Kernighan et Lin (K-L) [KL70] et le recuit simulé.

7.1.3.1 Recuit simulé

Le recuit simulé est une métaheuristique utilisant une analogie avec la mécanique statistique [KGV83]. La méthode procède en générant une configuration initiale aléatoire à une température élevée. Par la suite, de nouvelles configurations voisines sont générées et le système est refroidi suivant un ordonnancement donné. L'acceptation d'une configuration générée dépend de la température courante du système. L'algorithme de base est présenté dans la figure 7.4 et ses paramètres sont :

- la configuration initiale générée ;
- la température initiale T_{max} et la température finale T_{min} ;
- le nombre maximum de configurations acceptées à chaque palier de température N_a.

Initialisation :
Générer la configuration initiale et calculer son coût E_0.
$T = T_{max}$; /* Température courante */
$E = E_0$; /* Coût de la configuration courante */
Itération : {
Générer une nouvelle configuration voisine C aléatoirement.
$E' = f(C)$;
$\Delta E = E' - E$;
$E = E'$;
Accepter C avec la probabilité $P(\Delta E) \rightarrow$ **Si** OK $nb\_accept = nb\_accept + 1$;
Si $(nb\_accept < N_a)$ aller à **Itération**.
}
$T = a \times T$; /* Mise à jour de la température */
Si $(T \geq T_{min})$ aller à **Itération**.

FIG. 7.4 - *Recuit simulé de base.*

Le voisinage pour le problème de partitionnement de graphes consiste à échanger deux sommets entre les deux partitions A et B. Une comparaison entre le recuit simulé et la méthode K-L a montré que le recuit simulé est meilleur pour les graphes aléatoires mais sensiblement moins efficace pour les graphes géométriques [MO94].

7.1.3.2 Méthode hybride et version parallèle

La combinaison du recuit simulé et de la méthode K-L a donné lieu à un algorithme appelé optimisation locale chaînée [MO94]. L'algorithme CLO est illustré

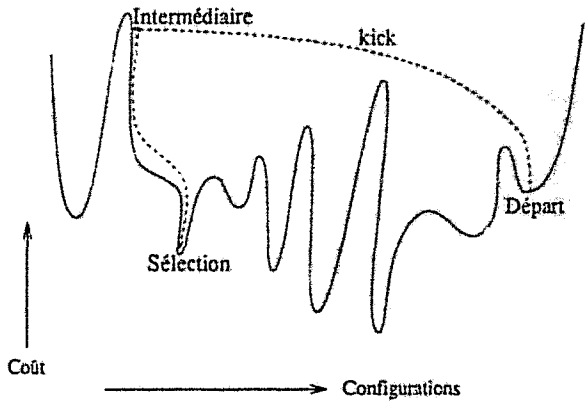


FIG. 7.5 - *Recuit simulé hybridé avec la recherche locale (CLO).*

par la figure 7.5. En supposant que l'algorithme est sur une configuration localement optimale (*Départ* sur la figure 7.5), une perturbation (kick) est appliquée à la configuration afin de diversifier la recherche (*Intermédiaire* sur la figure 7.5). Après la perturbation, la sélection (test d'acceptation) du recuit simulé n'est pas appliquée immédiatement mais après l'application de la recherche locale K-L. Grâce à son efficacité, la méthode K-L permet au recuit simulé d'améliorer ses performances en effectuant la sélection sur des configurations localement optimales. La structure du graphe affecte les performances de la méthode hybride (graphes aléatoires, hautement connexes, graphes géométriques, etc).

La version parallèle consiste à lancer plusieurs algorithmes de recherche hybride en parallèle (figure 7.6). Les algorithmes parallèles effectuent des synchronisations périodiquement afin d'échanger les meilleures configurations trouvées [MO94]. Après un nombre prédéfini d'itérations, chaque algorithme envoie le coût de la meilleure solution locale trouvée au maître. Par la suite, le maître diffuse l'identité de l'algorithme qui détient la meilleure solution globale. Chaque processus peut alors obtenir le meilleur partitionnement global.

Dans la version adaptative, une tâche est paramétrée par :

- la configuration initiale générée aléatoirement ;
- les températures initiale et finale du recuit ;
- le nombre maximum d'itérations à effectuer à chaque palier de température ;
- la période de synchronisation exprimée en terme d'itérations.

En effet, les recherches parallèles effectuent des synchronisations périodiquement, en communiquant la meilleure configuration locale au maître. Les tâches partielles sont ainsi mises à jour en adoptant la meilleure configuration globale comme étant

la nouvelle configuration courante. Les tâches partielles comportent également :

- la configuration courante ;
- le numéro de l'itération courante (et la température courante).

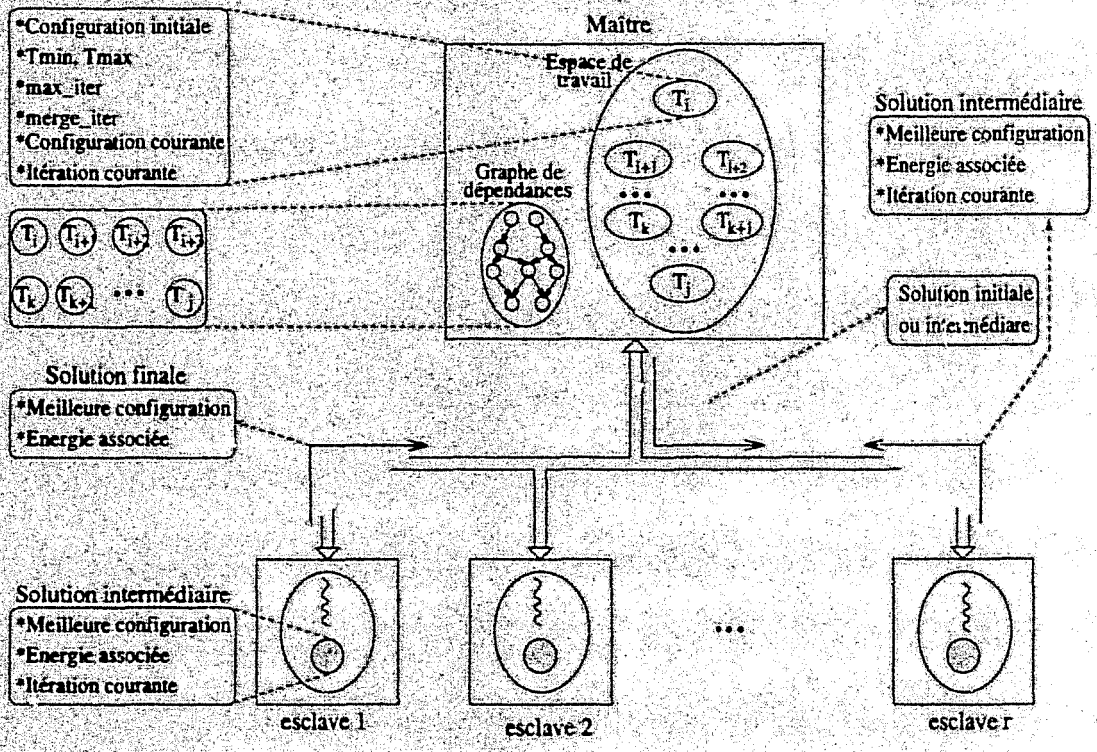


FIG. 7.6 - *La méthode CLO parallèle adaptative.*

La perturbation de la configuration courante (*kick*) est effectuée comme suit :

- trouver aléatoirement deux sommets x et y de telle sorte que x soit dans le sous-ensemble A et y dans B ;
- construire un groupe de sommets de B connectés à x et un autre contenant les sommets de A connectés à y ;
- échanger les deux groupes.

Le maître détient la meilleure configuration globale qu'il met à jour à chaque réception d'une tâche partielle ou finale. Les tâches peuvent être insérées dynamiquement afin de réduire la quantité de mémoire occupée par l'espace de travail. Les tâches sont entièrement indépendantes les unes par rapport aux autres. La figure 7.6 illustre la gestion d'une tranche de tâches insérées $T_i, T_{i+1}, ..., T_j$.

7.1.3.3 Résultats expérimentaux

Le tableau 7.2 résume les résultats de trois exécutions de la méthode CLO. L'efficacité est moins importante que pour la recherche tabou. Ceci peut être expliqué par la nature de l'application et par la charge externe de l'environnement (1 défaillance, 7 retraits, etc). La remarque principale est que, de la même façon que pour la recherche tabou, l'efficacité augmente avec le grain (problème de plus grande taille). Ceci montre de nouveau l'effet de la granularité sur les performances (de 26 à 60 %). Le surcoût d'ordonnancement est toujours négligeable grâce à la nature parallèle de l'application et à son gros grain.

$R(A)$	$T(A)$	$Sched$	N_{base}	$S(A)$	$U(A)$	UFd	Fd	Ret	Fl	APD	$Eff(A)$
68.60	469.3	0.0275	26.10	6.84	0.22	17	0	0	2	12.22	0.26
1623.77	14409.46	0.0059	16.45	8.87	0.55	12	0	5	1	7.13	0.54
18520.5	151035.6	0.0104	13.58	8.16	0.58	12	0	7	1	8.21	0.60

TAB. 7.2 - *Résultats obtenus pour l'algorithme CLO.*

7.2 Applications issues du calcul scientifique

Cette section est consacrée à la présentation et à la mise en œuvre d'algorithmes de calcul matriciel : les algorithmes de résolution de systèmes d'équations linéaires et d'inversion de matrices [Lei95]. Traditionnellement, la parallélisation de ces algorithmes a été réalisée sur des plateformes massivement parallèles.

7.2.1 Résolution de systèmes d'équations linéaires et élimination de Gauss

L'élimination de Gauss est une technique de résolution de systèmes d'équations linéaires et d'inversion de matrices non singulières. Elle est reconnue pour être robuste mais également coûteuse [Lei95].

Considérons le système d'équations linéaires $A\vec{x} = \vec{b}$. Si A est une matrice $n \times n$ non singulière, alors il existe une solution unique $\vec{x}$. L'élimination de Gauss réduit A en une matrice triangulaire, ce qui permet de résoudre aisément le système.

La triangularisation s'effectue en $n-1$ étapes. A l'étape k $(1 \leq k \leq n-1)$, la matrice A est mise à jour comme suit :

$$A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} \times A_{kj}^{(k-1)} / A_{kk}^{(k-1)}, \qquad k+1 \leq i,j \leq n. \qquad (7.1)$$

$$A_{ik}^{(k)} = 0 \qquad k+1 \leq i \leq n \tag{7.2}$$

Le reste des éléments de la matrice A est inchangé pendant la même étape.

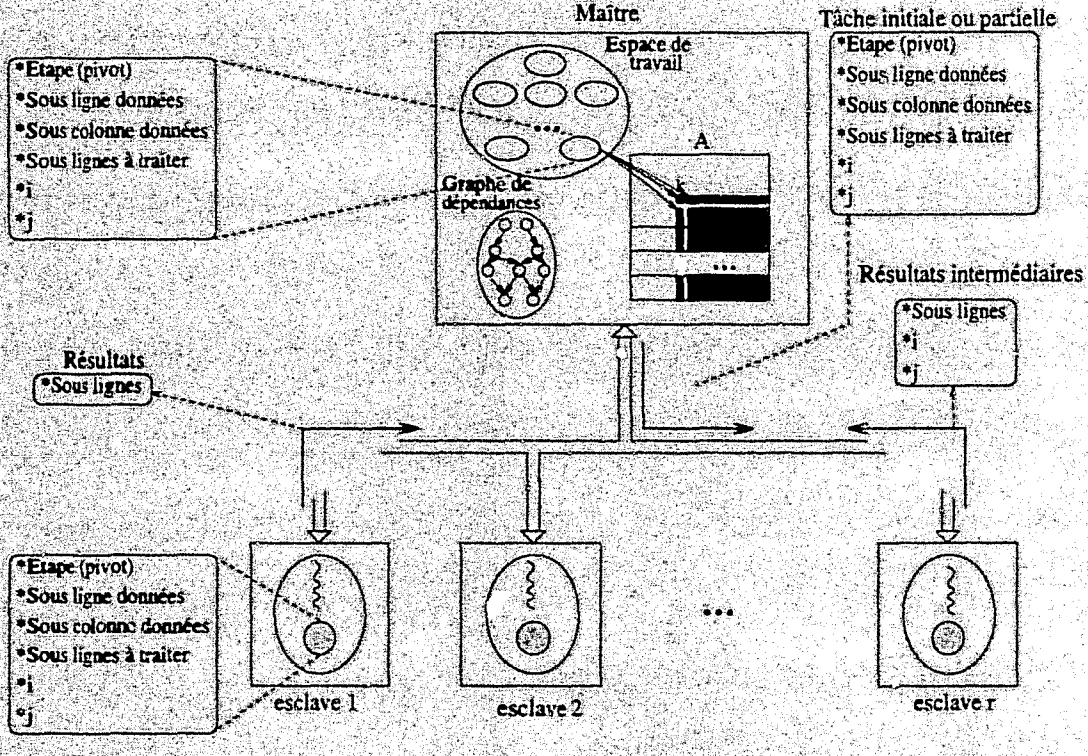


FIG. 7.7 - *Elimination de Gauss parallèle adaptative.*

La figure 7.7 illustre la mise en œuvre parallèle adaptative de l'application et la gestion des tâches. La structure du graphe de dépendances a été présentée au chapitre 4. Notons que chaque tâche se compose d'un ensemble de lignes de la matrice A (bloc).

7.2.1.1 Résultats expérimentaux

Traditionnellement, l'élimination de Gauss souffre de la régression continue du degré de parallélisme à travers les étapes, dégradant ainsi ses performances sur les architectures parallèles. Cette situation est susceptible de se maintenir, voire de se dégrader, dans le cas des réseaux de stations.

Le tableau 7.3 illustre les résultats de l'exécution de cette application. L'efficacité obtenue est médiocre. Ceci est dû à l'effet défavorable du rapport calcul/communication très faible que possède l'application, d'autant plus que, le réseau

est chargé (nombre de replis, de retraits, etc). Concernant le surcoût d'ordonnancement, celui-ci croît considérablement par rapport aux applications d'optimisation, même s'il est toujours négligeable. Le nombre de retraits augmente également dû à la structure plus complexe du graphe de dépendances et à l'hétérogénéité des nœuds.

$R(A)$	$T(A)$	*Sched*	N_{base}	$S(A)$	$U(A)$	UFd	Fd	*Ret*	Fl	APD	$Eff(A)$
457.356	600.7	0.9718	6.76	1.313	0.22	15	1	4	0	6.47	0.19
12243.1	12000.2	16.2387	6.18	0.98	0.26	20	3	7	0	3.72	0.16

TAB. 7.3 - *Résultats obtenus pour l'élimination de Gauss.*

7.2.2 Inversion de matrices par la méthode de Gauss-Jordan

Dans cette section, nous présentons l'implémentation parallèle adaptative de l'application de Gauss-Jordan par blocs. Cette méthode est une technique directe pour l'inversion de matrices [RB87]. Plusieurs versions parallèles sur des plateformes massivement parallèles ont été développées dans la littérature [Tin95]. Dans [MTP99], la méthode est implémentée sur l'environnement MARS de base. Notre objectif est d'exploiter notre interface de programmation afin de s'affranchir des tâches d'ordonnancement qui seront confiées à l'ordonnanceur de l'application.

La version séquentielle de l'algorithme est basée sur l'inversion par blocs. Considérons une matrice A découpée en $q \times q$ blocs de taille fixe b et B la matrice inverse. L'algorithme séquentiel consiste en q étapes. Durant chacune d'entre elles, un ensemble d'opérations (inversion de blocs, multiplication de blocs, etc) est appliqué aux blocs définis. La figure 7.8 présente une description de l'algorithme [MTP99].

La version parallèle adaptative consiste à découper le travail de chaque itération en un ensemble d'opérations. Ainsi, un ensemble de types de tâches a été défini (figure 7.10) :

- *tâche de type 0 :* c'est une inversion de bloc (pivot) ou recopie d'un bloc inversé de A dans le bloc correspondant de B. Par conséquent, une tâche de type 0 dépend uniquement de la dernière tâche exécutée sur le même bloc à l'étape précédente $k-1$ (en l'occurence une tâche de type 3) ;
- *tâche de type 1 :* il s'agit d'une multiplication de deux blocs de A : le bloc pivot, correspondant à la tâche 0 de l'étape k, et le bloc lui-même à l'étape $k-1$ (en l'occurence une tâche de type 3) ;
- *tâche de type 2 :* elle est équivalente à une tâche de type 1 mais concerne la matrice B. Il s'agit donc d'une multiplication du bloc pivot (tâche 0 actuelle) avec le bloc correspondant de B à l'étape précédente (en l'occurence une tâche de type 4 ou 5) ;

```
For (k = 1 to q) {
    A^k_{k,k} = (A^{k-1}_{k,k})^{-1}
    B^k_{k,k} = A^k_{k,k}
    For (j = k+1 to q)
        A^k_{k,j} = A^k_{k,k} A^{k-1}_{k,j}
    For (j = 1 to k-1)
        B^k_{k,j} = A^k_{k,k} B^{k-1}_{k,j}
    For (j = k+1 to q)
        For (i = 1 to q && i ≠ k)
            A^k_{i,j} = A^{k-1}_{i,j} - A^{k-1}_{i,k} A^k_{k,j}
    For (j = 1 to k-1)
        For (i = 1 to q && i ≠ k)
            B^k_{i,j} = B^{k-1}_{i,j} - A^{k-1}_{i,k} B^k_{k,j}
    For (i = 1 to q && i ≠ k)
        B^k_{i,j} = -A^{k-1}_{i,k} A^k_{k,k}
}
```

FIG. 7.8 - *La méthode de Gauss-Jordan par blocs pour l'inversion de matrices.*

- *tâche de type 3* : c'est une triadique impliquant, le bloc de A de même position à l'étape précédente (tâche de type 3 ou 1), le bloc de A à l'étape précédente de même ligne et de colonne k (tâche de type 3 ou 1) et le bloc de A de ligne k et de même colonne à l'itération courante (tâche de type 1) ;

- *tâche de type 4* : c'est une triadique sur B impliquant le même bloc à l'itération précédente (tâche de type 4, 0, 2 ou 5), le bloc de A de même ligne et de colonne k à l'étape précédente (tâche de type 3 ou 1) et le bloc de B de ligne k et de même colonne à l'étape courante (en l'occurence une tâche de type 2) ;

- *tâche de type 5* : il s'agit d'une multiplication de deux blocs, le résultat est dans B. Les deux blocs sont : le bloc pivot et le bloc de A de mêmes coordonnées à l'étape précédente (en l'occurence une tâche de type 3 ou 1).

La figure 7.10 illustre les dépendances entre les tâches pour une itération de l'algorithme de Gauss-Jordan. Durant chaque itération de l'algorithme, le nombre de triadiques est prédominant ($(q-1)^2$ triadiques contre 2 inversions de blocs et $2(q-1)$ multiplications de blocs). Etant donnés le coût d'exécution plus élevé des triadiques par rapport aux autres opérations et la nature parallèle de ces tâches, l'application peut réaliser des performances considérables, en particulier si la granularité est bien adaptée aux triadiques.

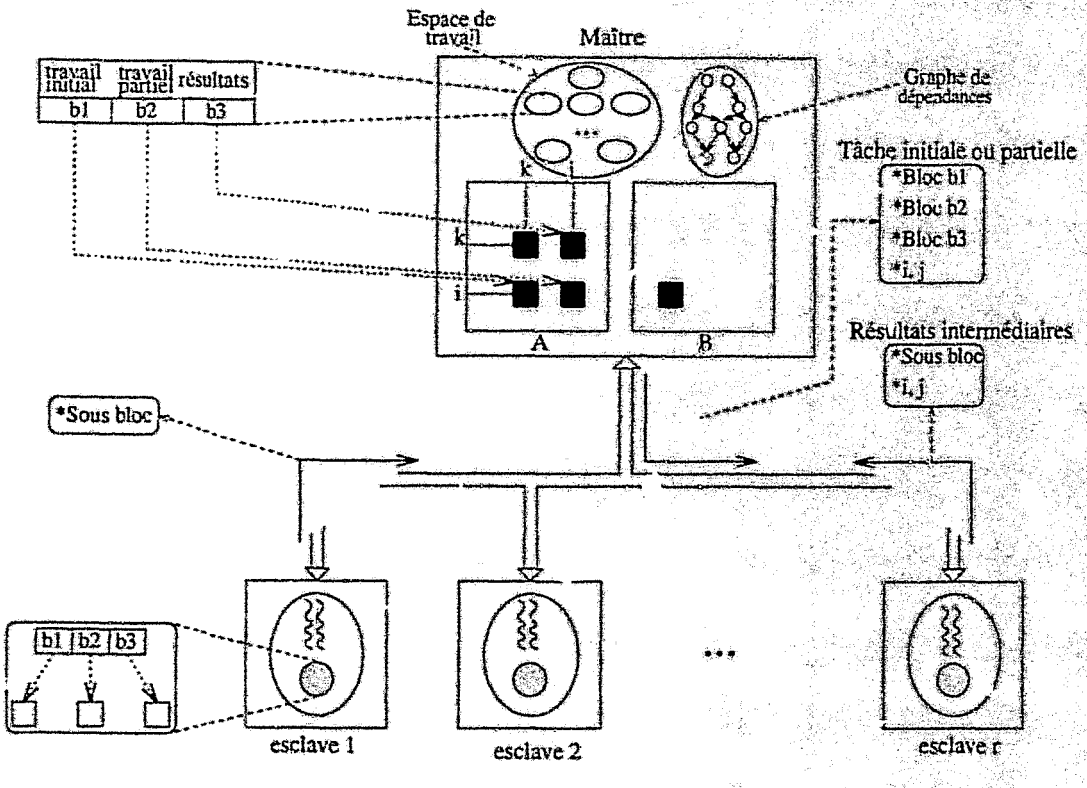


FIG. 7.9 - *Application de Gauss-Jordan parallèle adaptative.*

7.2.2.1 Résultats expérimentaux

Le tableau 7.4 présente les résultats d'exécution de l'application. La remarque principale est que l'efficacité s'est sensiblement améliorée par rapport à l'élimination de Gauss, car l'application possède un rapport calcul/communication plus important que celui de l'élimination de Gauss. L'efficacité est de 40% en dépit de la charge externe. Concernant le surcoût d'ordonnancement, celui-ci est inférieur à celui de l'élimination de Gauss. Ceci est dû à l'effet positif du gros grain de l'application de Gauss-Jordan.

$R(A)$	$T(A)$	$Sched$	N_{base}	$S(A)$	$U(A)$	UFd	Fd	Ret	Fl	APD	$Eff(A)$
1581.88	7016.28	3.1685	11.29	4.44	0.39	40	0	12	0	5.88	0.39
1565.92	6772.36	0.0659	10.84	4.32	0.37	23	0	11	0	5.45	0.40

TAB. 7.4 - *Résultats obtenus pour la méthode Gauss-Jordan.*

La figure 7.11 trace le temps de réponse de l'application en fonction du nombre de nœuds. Le temps de réponse décroît énormément avec l'augmentation du nombre de

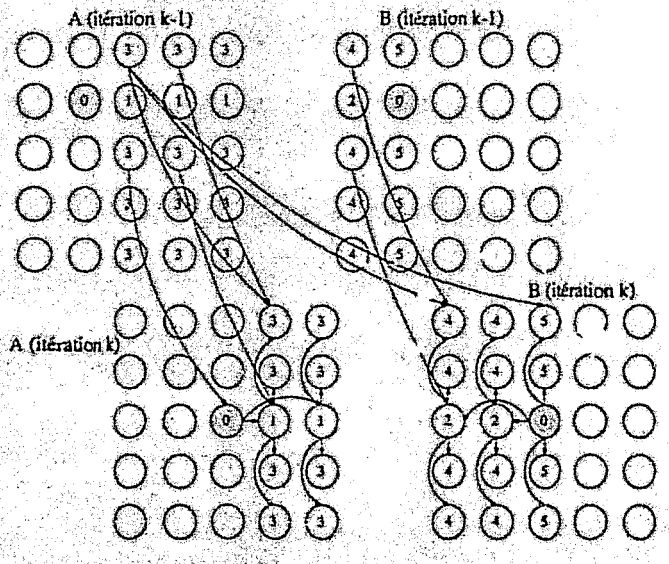


FIG. 7.10 - *Graphe de dépendances de l'application Gauss-Jordan.*

nœuds utilisés, jusqu'à un point de saturation où le nombre de nœuds physiques excède le nombre de tâches d'une itération. Ceci confirme l'adaptation de l'application aux environnements adaptatifs.

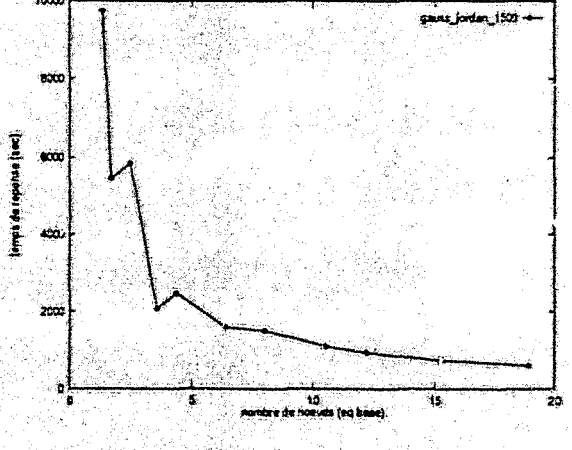


FIG. 7.11 - *Temps de réponse de l'application en fonction du nombre de nœuds utilisés (matrice* 1500×1500*).*

Les résultats de Gauss-Jordan présentés concernent la version itérative. Cependant, existe t-il un parallélisme entre les tâches des différentes itérations? La figure 7.12 illustre les dépendances entre les types de tâches d'une itération. Il est évident qu'il existe un parallélisme potentiel entre les tâches des différentes itérations (figures 7.12 et 7.10).

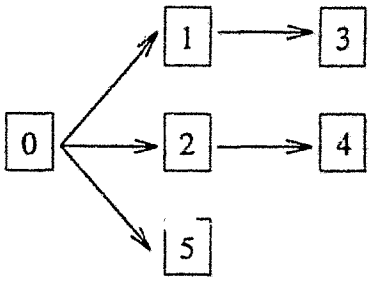


FIG. 7.12 - *Schéma des dépendances des types de tâches de l'application Gauss-Jordan.*

Dans la figure 7.10, à l'itération k, la terminaison d'une tâche de type 0 débloque les tâches de type 1 qui à leur tour autorisent l'exécution des tâches de type 3. Ce qui permet le passage à l'état prêt du pivot et des tâches de type 1 de l'étape $k+1$, puis des tâches de type 3 et ainsi de suite. Cela est effectué pendant que des tâches de l'itération k ne sont pas encore terminées (figures 7.12 et 7.10).

Il est intéressant de mettre en œuvre la version où des tâches de plusieurs itérations sont insérées ensemble comme dans le cas de l'élimination de Gauss. Cette version est susceptible d'obtenir des performances supérieures à celles obtenues par la version itérative.

7.3 Classification

Les techniques de classification (clustering) sont largement utilisées dans de nombreux domaines: imagerie médicale, bases de données et récemment dans le data-mining [JDS8]. La technique IRM (*Imagerie à Résonance Magnétique*) est une méthode d'imagerie médicale de haute qualité. Les algorithmes de segmentation d'images IRM ont été utilisés dans le but de détecter des éventuelles tumeurs du cerveau [BHB+96]. Ces algorithmes sont très coûteux en temps d'exécution et leur parallélisation est de ce fait inévitable. Dans le cadre d'une collaboration avec l'université El-Akhawayn (Maroc), la parallélisation adaptative d'un algorithme de classification basé sur la logique floue nommé *Fuzyy C-Means (FCM)* a été réalisée.

Le problème de classification consiste à identifier un ensemble de c classes dans un ensemble de données X de taille n. Deux problèmes sont à résoudre à cet égard: comment mesurer la similitude entre les paires d'observations? et comment évaluer les partitions? Parmi les méthodes de mesure de similitude, figure la distance entre les observations. FCM est une technique complètement non-supervisée et par conséquent coûteuse. L'intervention humaine peut simplifier et améliorer l'évaluation des classes. Un algorithme semi-supervisé *"Semi Supervised Point Prototype Clutering, ssPPC"* est par conséquent introduit.

7.3.1 L'algorithme FCM

Considérons un ensemble de données $X = \{x_1, x_2, ..., x_n\}$ qui doit être partitionné en c classes. Chaque élément x_i est un vecteur de p mesures réelles décrivant les caractéristiques de l'objet représenté par x_i.

FCM est un algorithme de classification basé sur le calcul des centres des classes en utilisant les données elles-mêmes [JD88]. Les données sont ensuite regroupées en minimisant leur distance par rapport aux centres. Les critères d'appartenance aux classes sont donnés par:

$$minimiser\ J_m(U, V, X) = \sum_{i=1}^{n}\sum_{j=1}^{c} u_{ji}^m \|x_i - v_j\|_A^2. \tag{7.3}$$

tel que: $0 \leq u_{ji} \leq 1, \quad 1 \leq j \leq c \quad 1 \leq i \leq n,$
$\sum_{j=1}^{c} u_{ji} = 1, \quad 1 \leq i \leq n,$ et
$0 < \sum_{i=1}^{n} u_{ji} < n, \quad 1 \leq j \leq c.$

Les paramètres de l'algorithme sont:

T: nombre maximum d'itérations.
m: degré d'incertitude dans la définition des classes.
$E_t = \|U_t - U_{t-1}\|_{err}$: taux de changement de la matrice U entre les itérations $t-1$ et t.
ϵ: seuil de terminaison sur E_t.
U: matrice d'appartenance aux classes $(c \times n)$.
V: c-tuple représentant les centres des classes.
$\|.\|$: norme utilisée, $\|x\|_A^2 = x^T A^{-1} x$.

La distance entre un centre v_i et un point x_k est calculée comme suit:

$$\|x_k - v_i\|_A^2 = (v_i - x_k)^T A^{-1} (v_i - x_k). \tag{7.4}$$

A est une matrice $(n \times n)$ définie positive exprimant la métrique utilisée. Nous utilisons une distance euclidienne (A = Identité).

L'algorithme FCM séquentiel est présenté sur la figure 7.13. Sa complexité est en $O(nc^2p)$.

7.3.2 Algorithme ssPPC et sa version parallèle adaptative

L'algorithme semi-supervisé ssPPC permet d'améliorer les performances des algorithmes de classification non supervisés [BHB+96]. En effet, des données étiquetées sont utilisées pour guider la formation des classes. Les données sont décomposées en deux sous-ensembles: n_d données étiquetées $(\{x_k^d\}, 1 \leq k \leq n_d)$ et n_u données non étiquetées $(\{x_k^u\}, 1 \leq k \leq n_u)$. Les données étiquetées guident la classification des données non étiquetées en c classes. L'algorithme FCM est appliqué aux données

FCM.1 Initialisation:
$v_{i,o} = m + (i - 1/c - 1)(M - m)\ i = 1, ..., c.$
tel que: $m_j = min_k(x_{jk})$ et $M_j = max_k(x_{jk})\ j = 1, 2, ..., p.$
FCM.2 Calculer U:
$u_{ik,0} = [\sum_{j=1}^{c}(\|x_k - v_{i,0}\|_A / \|x_k - v_{j,0}\|_A)^{2/(m-1)}]^{-1}\ \forall i, k.$
For $t = 1$ *to* T {
FCM.3 Calculer V_t:
$v_{i,t} = \sum_{k=1}^{n}(u_{ik,t-1})^m x_k / \sum_{k=1}^{n}(u_{ik,t-1})^m,\ \forall i.$
FCM.4 Calculer U_t:
$u_{ik,t} = [\sum_{j=1}^{c}(\|x_k - v_{i,t}\|_A / \|x_k - v_{j,t}\|_1)^{2/(m-1)}]^{-1},\ \forall i, k.$
FCM.5 Calculer $E_t = \|U_t - U_{t-1}\|_{err}$.
If $E_t < \epsilon$ **then** stop
}

FIG. 7.13 - *La version séquentielle de l'algorithme FCM.*

non étiquetées avec un nombre de classes égal à n_d. Les n_d classes résultantes sont ainsi fusionnées en c classes en affectant chaque classe C_i de centre v_i à la classe de la plus proche donnée étiquetée. L'algorithme ssPPC est décrit dans la figure 7.14.

La version parallèle adaptative consiste à fragmenter l'ensemble de données X en un ensemble de partitions qui seront évaluées en parallèle. La figure 7.15 illustre les dépendances entre les tâches de plusieurs itérations différentes. En effet, pour calculer le vecteur V à une itération, il faut utiliser la matrice U de l'itération précédente et vice-versa.

L'algorithme parallèle adaptatif est décrit ci-dessous. X représente l'ensemble des configurations non étiquetées. Initialement, X est fragmenté en s partitions.

1. Le vecteur des centres V est initialisé avec les configurations étiquetées et les s tâches de type 1 sont construites. Les tâches de type 1 servent à calculer la matrice U de l'étape. Chaque tâche j comporte une copie du vecteur V et un sous-ensemble de X $(x_{(j-1)n/s}, \ldots, x_{jn/s})$ de taille $size = n/s$.

2. A ce stade, l'algorithme FCM doit être appliqué aux configurations non étiquetées (U_j^u, V_j^u), avec n_d classes. Ainsi, chaque esclave calcule une partie de la matrice d'appartenance des membres de la tâche $u_{ik,0}$. Ensuite, il renvoie la partition de la matrice U calculée. Chaque tâche j comporte un sous-ensemble de X défini par $\{x_k, k = (j-1)n/s, ..., jn/s\}$ et une copie du vecteur V. Cette étape complète la phase d'initialisation. $u_{ik,0} = [\sum_{j=1}^{c}(\|x_k - v_{i,0}\|_A / \|x_k - v_{j,0}\|_A)^{2(m-1)}]^{-1}\ \forall i, k.$

3. Après la réception de toutes les parties de la matrice U, le maître génère les tâches de type 2 afin de calculer le vecteur V. Chaque tâche comporte

ssPPC.1 Initialiser les centres des classes avec les données étiquetées.
ssPPC.2 Appliquer l'algorithme FCM (sans FCM.1) à l'ensemble des données non étiquetées ($\{x_k\}, 1 \leq k \leq n_u$), puis exécuter FCM jusqu'à sa terminaison sur (U_f^u, V_f^u).
ssPPC.3 Calculer le voisinage le plus proche parmi les données étiquetées à chaque centre d'une classe finale $v_{j,f}^u$ $(j = 1, ..., n_d)$. Construire la matrice B:
$B = [U_{nn(1)}^d, U_{nn(2)}^d, ..., U_{nn(nd)}^d]$. *Où* $nn(j) = arg\ min_{1 \leq s \leq n_d}\{\|x_s^d - v_{j,f}^u\|\}$, $j = 1, 2, ..., n_d$.
ssPPC.4 Calculer $\hat{U}_f^u = BU_f^u$.

FIG. 7.14 - *L'algorithme ssPPC séquentiel.*

une partition de la matrice U, et sa partie correspondante des données de l'ensemble X. Le résultat de chaque tâche est représenté par α et β:
$\alpha_{i,j} = \sum_{k=1}^{size}(u_{ik,t-1})^m x_k$ $\beta_{i,j} = \sum_{k=1}^{size}(u_{ik,t-1})^m$.

4. En recevant les résultats de toutes les tâches du second type, le maître met à jour le vecteur V comme suit: $v_{i,t} = (\sum_{j=1}^{s} \alpha_{i,j})/(\sum_{j=1}^{s} \beta_{i,j})$ $i = 1, ..., c$.

   Par la suite, il procède à la génération des tâches du type 1 qui se chargeront de calculer la matrice U. Une tâche de ce type comporte une copie du vecteur V et une partition de X.

5. Chaque tâche calcule la partie correspondante de la matrice U et retourne le résultat au maître $u_{ik,t} = [\sum_{j=1}^{size}(\|x_k - v_{i,t}\|_A / \|x_k - v_{j,t}\|_A)^{2(m-1)}]^{-1}$ $\forall i, k$.

6. Après la réception de toutes les parties, le maître évalue l'erreur. Si la précision requise n'est pas atteinte ($E_t \geq \epsilon$) et le nombre d'itérations maximum n'est pas excédé ($t < T$), une autre itération est initiée (étape 3).
   L'erreur au niveau de chaque tâche j est calculée comme suit:
   $error_j = \sum_{k=1}^{size}\sum_{i=1}^{c}(u_{ik,t-1} - u_{ik,t})^2$, pour $j = 1...s$.
   Les erreurs ainsi calculées sont agrégées pour calculer l'erreur global $E_t = (\sum_{j=1}^{s} error_j)^{1/2}$.

7. Lorsque l'algorithme FCM est terminé, le maître calcule les voisins parmi les données étiquetées les plus proches des centres de V et construit la matrice $B = [U_{nn(1)}^d, U_{nn(2)}^d, ..., U_{nn(nd)}^d]$. Ensuite, il génère le dernier type de tâches dont chacune comporte une partition de U et une copie de B. Chaque tâche calcule les membres finaux de chaque classe en fusionnant les n_d clusters en c classes. La tâche k se charge de calculer la sous-matrice $\hat{U}_{f,k}^u = BU_{f,k}^u$ et la retourne au maître, où $U_{f,k}^u = [U_{(k-1)(size)+1}^u, U_{(k-1)(size)+2}^u, ..., U_{k(size)}^u]$.

8. Dès la réception de tous les résultats, le maître reconstruit les classes finales.

La complexité de la version parallèle de FCM est en $O(c^2pn/s)$.

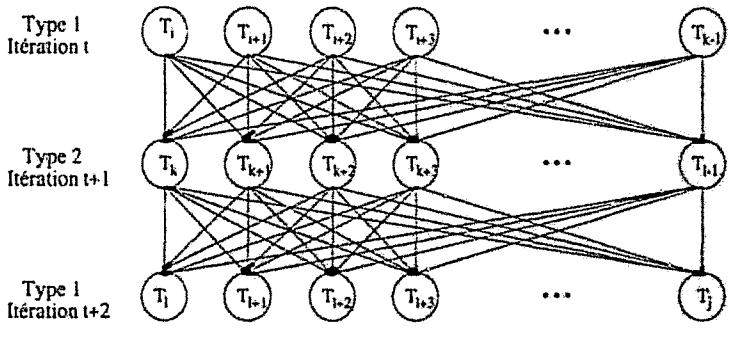


FIG. 7.15 - *Graphe de dépendances de l'algorithme FCM.*

Le retrait d'une tâche consiste à retourner la partie de la matrice U déjà calculée et les indices i et j du reste du travail si la tâche est de type 1. Les résultats partiels d'une tâche de type 2 comportent les partitions α et β calculées.

L'exécution de l'algorithme sur des images de cerveau dure entre 5 et 7 jours sur une configuration composée d'environ une trentaine de stations de travail. L'apport principal de MARS pour cette application concerne l'aspect tolérance aux fautes, en permettant à l'application de progresser.

Appli	$R(A)$	$T(A)$	*Sched*	N_{base}	$S(A)$	$U(A)$	UFd	Fd	Ret	APD	$Eff(A)$
Gauss	970.78	804.84	0.2625	5.59	0.83	0.10	9	0	0	2.09	0.15
Gauss-J	2135.28	6524.4	0.2608	8.34	3.06	0.40	20	2	6	3.76	0.37
Tabu	223.43	1680.27	0.0081	24.84	7.52	0.32	15	0	7	12.67	0.30
CLO	1525.45	7399.26	0.0143	11.06	4.85	0.29	15	6	0	5.25	0.44

TAB. 7.5 - *Résultats obtenus de l'exécution simultanée des quatre applications.*

Le tableau 7.5 présente les résultats d'une exécution simultanée des quatre applications présentées dans ce chapitre, en utilisant la politique d'ordonnancement multi-application combiné. L'efficacité de toutes les applications s'est légèrement dégradée. Ceci est la conséquence de l'exécution simultanée des applications, qui provoque l'intensification des activités des ordonnanceurs applicatifs, d'autant plus que ces applications sollicitent intensivement le processeur. La dégradation de l'efficacité du *tabou* est dûe principalement à sa courte durée d'exécution.

7.4 Conclusion

La plupart des applications que nous avons présentées sont des applications pratiques et traitent de problèmes souvent difficiles à résoudre. Elles se caractérisent

par des durées de vie considérables. Des applications de calcul scientifique aux applications de classification en passant par les applications d'optimisation combinatoire, la durée de vie varie de quelques dizaines de minutes à quelques jours.

L'efficacité obtenue, en particulier, par les applications d'optimisation combinatoire, est très satisfaisante. La nature de l'environnement (multi-utilisateur, charge externe irrégulière, etc) et le mode d'exploitation des ressources (support d'exécution adaptative opportuniste) font que l'efficacité réalisée est un plus. Le surcoût d'ordonnancement intra-application est négligeable, même pour les applications sollicitant le plus l'ordonnanceur (applications du calcul scientifique).

Globalement, les résultats montrent que les applications d'optimisation combinatoire enregistrent une efficacité bien supérieure à celle des applications de calcul scientifique. En effet, le modèle de parallélisme adaptatif favorise les applications avec un rapport calcul/communication élevé. Les résultats expérimentaux montrent également que notre approche d'ordonnancement intra-application est mieux exploitée par les applications du calcul scientifique, qui possèdent des graphes de dépendances plus complexes que ceux des applications d'optimisation combinatoire. Les applications d'optimisation bénéficient également de l'ordonnancement intra-application en exploitant toujours les nœuds les plus puissants de la configuration.

Conclusion et perspectives

Traditionnellement, les applications parallèles étaient exécutées sur des machines massivement parallèles dédiées. La prolifération des réseaux de stations de travail et des clusters de processeurs ainsi que des méta-systèmes intégrant ces plateformes constitue une alternative pour l'exécution de ces applications. Par ailleurs, la complexité accrue des applications parallèles augmente de plus en plus leurs besoins en puissance de calcul (applications de longue durée de vie).

L'exploitation des méta-systèmes est souvent confrontée à de nombreux problèmes liés aux propriétés de ces environnements : risque de défaillance élevé, hétérogénéité matérielle et logicielle à plusieurs niveaux, aspect multi-utilisateur, dynamicité de l'environnement, utilisateurs interactifs, etc. Les systèmes spécialisés dans la détection et l'exploitation des ressources inutilisées dans ces environnements, tels que Worm, Godzilla, Condor, GLUnix, etc, ont montré leur intérêt. Le caractère dynamique des méta-systèmes et l'irrégularité de la disponibilité des nœuds impliquent le déplacement de la charge ou la reconfiguration dynamique des applications. Les environnements dits adaptatifs, qui imposent un degré de parallélisme variable aux applications parallèles, permettent de s'adapter au caractère dynamique de ces environnements. La mise en place des moyens, permettant de résoudre certains problèmes caractéristiques des méta-systèmes et de mieux exploiter leurs ressources, constitue notre principale préoccupation.

Nous avons conçu et réalisé un environnement, pour la gestion de ressources et l'ordonnancement d'applications parallèles sur ces plateformes, qui se caractérise par plusieurs propriétés sur de nombreux aspects :

Tolérance aux fautes

La sauvegarde/reprise est réputée pour être une solution efficace pour la tolérance aux fautes dans les systèmes distribués. Ses principaux avantages résident dans son faible coût, son insensibilité aux fautes transitoires et sa transparence. Les algorithmes et les techniques basés sur cette approche ont connu une prolifération rapide et importante. Cependant, la complexité de ces algorithmes, en terme de messages de contrôle et de surcoût d'exécution, réduit parfois leur utilisation.

Nous avons proposé un algorithme de sauvegarde/reprise qui exploite les caractéristiques des applications parallèles adaptatives afin de réduire cette complexité.

Cela permet d'utiliser le mécanisme sans dégrader les performances de l'application. De plus, grâce aux propriétés particulières du modèle d'applications adaptatives, l'approche présente d'autres avantages tels que l'insensibilité aux problèmes de l'hétérogénéité et l'adaptation à l'environnement et à la disponibilité de ressources au moment du recouvrement. Par ailleurs, l'aspect détection des défaillances et recouvrement automatique d'applications a été développé. Le recouvrement partiel, impliquant uniquement les composants défaillants de l'application adaptative dans le processus du recouvrement, est une propriété importante de notre approche.

Ordonnancement d'applications adaptatives

Nous avons proposé un modèle pour la construction et l'ordonnancement d'applications parallèles adaptatives. L'instanciation du modèle a donné lieu à une interface de programmation facilitant la mise en œuvre d'applications parallèles adaptatives et un module d'ordonnancement visant à optimiser l'exécution de l'application et à mieux exploiter les ressources mises à sa disposition. Les avantages principaux du modèle sont résumés ci-dessous :

- parallélisme adaptatif permettant aux applications parallèles d'exploiter les ressources inutilisées dans les méta-systèmes sans gêner les utilisateurs interactifs.
- dynamicité du modèle, à travers la construction dynamique de l'application et l'ordonnancement préemptif. Ceci est primordial pour les environnements adaptatifs et pour les applications irrégulières en particulier ;
- ordonnancement dynamique efficace, à travers des critères d'ordonnancement simples à évaluer et à utiliser (dépendances des tâches et puissance relative des nœuds). Ces deux critères permettent d'accélérer l'exécution des tâches génératrices de parallélisme. La gestion de l'hétérogénéité permet d'exploiter les nœuds les plus puissants lorsque la disponibilité est abondante, en particulier en phase terminale des applications de longue durée de vie.

Ordonnancement multi-application

L'ordonnancement multi-application permet de préserver et de réaliser certaines propriétés telles que l'équité et la minimisation du temps de réponse moyen des applications. Ceci est très important en particulier pour les applications de longue durée de vie. L'ordonnancement multi-application permet notamment :

- une vue globale du système. Ceci garantie une gestion efficace et une meilleure utilisation des ressources ;
- une possibilité d'exécution simultanée de plusieurs applications parallèles adaptatives et de résoudre leurs conflits à travers une vue unique ;

- une possibilité d'extension du système à d'autres modèles d'applications parallèles, sous réserve qu'ils soient dynamiques (mécanismes de déplacement des calculs, etc). En effet, toutes les mesures ont été prises pour faciliter l'intégration de ces modèles à travers notamment le classement des ressources en différentes catégories et la prise en compte des besoins des applications exigeant des architectures particulières.

Le partage équitable de ressources entre les applications dans les environnements adaptatifs est primordial. En effet, les applications adaptatives ont tendance, en général, à s'étaler pour exploiter la totalité des ressources disponibles. Le système développé vérifie d'autres propriétés telles que la portabilité, la simplicité et l'intégration de la tolérance aux fautes.

Notons que le système est opérationnel depuis quelques mois. L'ensemble (exécutif et interface de programmation) fait environ 15000 lignes de code et est disponible sur le serveur web du LIFL.

Applications

La validation du modèle sur le plan applicatif est importante. En effet, de nombreuses applications ont été développées ou portées sur l'environnement MARS. Ces applications sont utilisées dans des domaines d'actualité (médecine, télécomunication, bases de données, etc) et se caractérisent par de longues durées de vie. Les applications développées concernent en particulier les domaines du calcul scientifique et de l'optimisation combinatoire. La remarque principale est que ces applications réalisent des performances considérables grâce aux propriétés de notre système (tolérance aux fautes à moindre coût, facilité de développement et possibilité de cohabitation à travers l'ordonnancement multi-application, etc). A travers les résultats de performances obtenus, nous remarquons :

- la tolérance aux fautes est primoridale. En effet, les applications s'exécutant pendant des heures (optimisation combinatoire) voire des jours (traitement d'images) ont une espérance de terminaison très réduite sans mécanisme de tolérance aux fautes ;

- les applications à moyen et à gros grain avec un rapport calcul/communication important sont les mieux adaptées aux environnements adaptatifs, étant donné la dynamicité de l'environnement et les délais de communication.

Collaborations

Dans le cadre de ce projet, plusieurs collaborations, entre l'équipe OPAC et d'autres équipes dans d'autres universités à l'échelle nationale et internationale, ont été entreprises. L'objectif de ces collaborations était notamment la mise en œuvre d'applications sur notre système. Les résultats étaient satisfaisants et ont donné lieu

à quelques publications [LBK+99]. Parmi nos partenaires, on trouve: l'université Al-Akhawayn (Maroc). Institut de Physique Nucléaire d'Orsay, l'université centrale du Venezuela, l'équipe MAP du LIFL, etc.

Perspectives

L'exploitation de ressources dans les méta-systèmes est confrontée à de nombreux problèmes additionnels liés notamment à la taille et à l'aspect hétérogène des méta-systèmes. Par conséquent, l'environnement proposé doit être extensible (l'ajout de nœuds ne doit pas dégrader les performances du système). La solution généralement adoptée est le partitionnement des ressources en plusieurs groupes. Plusieurs critères de partitionnement peuvent être utilisés: voisinage géographique, domaine d'administration, etc. Les applications parallèles pourront ainsi exploiter la disponibilité de ressources dans des réseaux géographiquement étendus ("métacomputing"). Malgré cette structuration, l'exécution distante d'applications parallèles demeure coûteuse et pourvue de problèmes. Les applications adaptatives possèdent des caractéristiques leur permettant éventuellement une exploitation de resources efficace dans ces environnements. Le partitionnement de l'application en plusieurs espaces de travail dont la communication entre eux est minimale semble une solution prometteuse (figure 7.16). L'exploration de cette piste doit traiter les aspects suivants:

- hiérarchisation automatique de l'application: il s'agit de fournir les moyens permettant de décentraliser le contrôle de l'exécution et les protocoles d'interaction entre le maître principal et les sous-maîtres (figure 7.16);
- détermination de l'espace de travail de chaque sous-maître de telle sorte que les interactions entre les différents agents soient minimales;
- extension des algorithmes d'ordonnancement et de tolérance aux fautes présentés précédemment à cette nouvelle topologie de l'application et l'étude des problèmes induits (fragmentation du graphe de dépendances, gestion répartie des dépendances, etc).

A court et moyen terme, nous envisageons de:

- effectuer une évaluation des performances de l'approche d'ordonnancement intra-application sur d'autres applications (multiplication de matrices, etc). Il serait intéressant d'effectuer également une analyse plus rigoureuse des performances de l'approche dans le but d'évaluer la part du surcoût séquentiel et du surcoût lié à la communication et à la synchronisation;
- évaluer les performances de l'approche d'ordonnancement multi-application basée sur le partitionnement dynamique équitable;

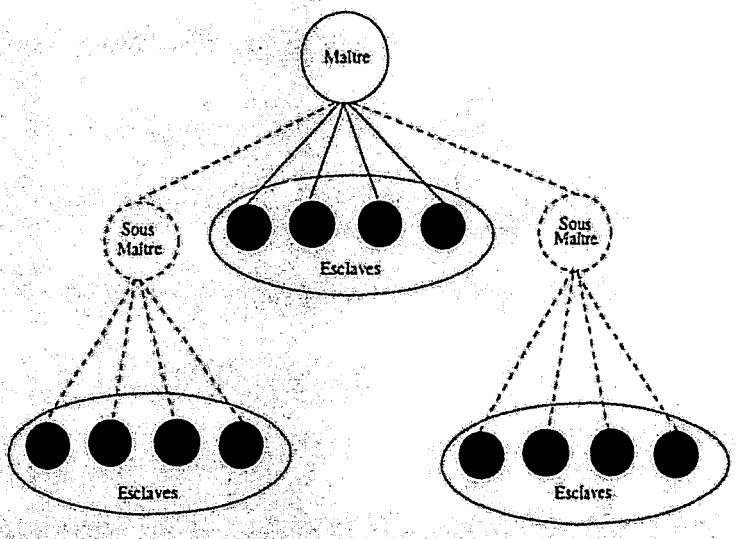


FIG. 7.16 - *Structure hiérarchique de l'application parallèle adaptative.*

- compléter l'ordonnanceur global pour prendre en charge les applications parallèles non-adaptatives, et d'évaluer ses performances en présence de ce type d'applications.

Sur le plan applicatif, notre équipe porte un intérêt particulier aux domaines des télécommunications et de bio-informatique, à travers notamment les applications d'optimisation combinatoire et de classification. Dans le domaine de la bio-informatique, des applications, utilisant des algorithmes génétiques et des algorithmes de classification dans le but d'établir les causes et les attributs associatifs qui contribuent au développement de certaines maladies multi-factorielles, sont en développement. Dans le domaine des télécommunications, des applications pour le design des réseaux et pour le problème d'affectation de fréquences dans les réseaux GSM, utilisant des algorithmes génétiques multi-critères, sont également développées. Etant donné les besoins en performances de cette classe d'applications, nous envisageons la mise en œuvre d'algorithmes parallèles adaptatifs dans le but d'améliorer leurs performances et pouvoir traiter des problèmes de grande taille.

Bibliographie

[ABB+86] M. Accetta, R. Baron, W. Bolosky, R. Rashid, A. Tevanian, and M. Young. Mach : A new kernel foundation for Unix development. In *Summer Usenix*, pages 93–112, Jul 1986.

[ABM92] M.J. Atallah, C.L. Black, and D.C. Marinescu. Models and algorithms for coscheduling compute-intensive tasks on a network of workstations. *Journal of Parallel and Distributed Computing*, 16:319–327, 1992.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *22th International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, MA, Jul 1992. IEEE.

[ADP94] J. Arlat, Y. Deswarte, and D. Powell. Systèmes commerciaux. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*, pages 73–79. Masson, Paris, France, 1994.

[AG91] I. Ahmad and A. Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Trans. Software Eng.*, 17(10):987–1004, Oct 1991.

[Ahu90] M. Ahuja. Global snapshots for asynchronous distributed systems with non-FIFO channels. Technical Report CS92-268, Univ. of California, San Diego, Nov 1990.

[AL81] T. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice.* Prentice Hall, 1981.

[AM98] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, Feb 1998.

[Anc93] E. Anceaume. *Algorithmique de Fiabilisation d'un Système Réparti.* PhD thesis, Paris XI-Orsay, 1993.

[AS93] N. Attig and V. Sander. Automatic checkpointing of NQS batch jobs on CRAY UNICOS system. *Ersheint in proceedings Cray User Group Meeting (Spring), Montreux*, 1993.

[ASPM+97] K. Al-Saqabi, R. M. Proutty, D. McNamee, S. W. Otto, and J. Walpole. Dynamic load distribution in MIST. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, Jun 1997.

[AV93] S. Alagar and S. Venkatesan. An optimal algorithm for distributed snapshots with causal message ordering. Technical report, Univ. of Texas, Dallas, 1993.

[AVAM92] V.A.F. Almeida, I.M.M. Vasconcelos, J.N.C. Arabe, and D.A. Menascé. Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems. In *Supercomputing'92*, pages 683–691, Minneapolis, MN, Nov 1992.

[Avi78] A. Avizienis. Fault-tolerance, the survial attribute of digital systems. In *IEEE, vol 66, number 10*, pages 1109–1125, 1978.

[Avi85] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12), Dec 1985.

[Bac99] V. Bachelet. *Métaheuristiques Parallèles Hybrides : Application au Problème d'Affectation Quadratique.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, 1999.

[BBD96] Özalp Babaoğlu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. Technical Report TR UBLCS-96-3, Department of Computer Science, University of Bologna, Italy, Feb 1996.

[BDM95] Özalp Babaoğlu, R. Davoli, and A. Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. Technical Report TR UBLCS-95-18, Department of Computer Science, University of Bologna, Italy, Nov 1995.

[BF96] G. Bernard and B. Folliot. Caractéristiques générales du placement dynamique: Synthèse et problématique. *Tutorial invité, Ecole d'été MASI-IMAG-INT-PRISM*, Jul 1996.

[BGH87] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in Tandem computer systems. *The Evolution of Fault-Tolerant Systems*, pages 55–76, 1987.

[BHB+96] A. Bensaid, L. Hall, J. Bezdek, L. Clarke, M. Silbiger, J. Arrington, and R. Murthagh. Validity-Guided (Re)clustering with applications to image segmentation. *IEEE Trans. on Fuzzy Systems*, 4(2):112–123, 1996.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[BHMR95] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. Consistent checkpointing in message passing distributed systems. *Research Report N 2564, Unité de recherche INRIA Rennes*, Jun 1995.

[Bir93] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, Dec 1993.

[Bir96] K. P. Birman. *Building secure and reliable network applications.* Manning Publications Co., 1996.

[BJ87] K.P. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. In *11th Symposium on Operating Systems Principles*, pages 123–138, Austin, Nov 1987. ACM Press.

[BL94] R.D. Blumofe and C.H. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual IEEE Conference on Foundations of Computer Science (FOCS'94)*, Santa Fe, New Mexico, Nov 1994.

[Bre94] T.B. Brecht. *Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors.* PhD thesis, University of Toronto, 1994.

[BSW79] P.A. Bernstein, D.W. Shipman, and W.S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3):203–216, May 1979.

[CA78] L. Chen and A. Avizienis. N-version programming: A fault tolerance approach to reliability of software operation. In *8th Int. Symp. on Fault Tolerant Computing*, pages 3–9, Toulouse, France, 1978. IEEE Computer Society Press.

[CA82] T.C.K. Chou and J.A. Abraham. Load balancing in distributed systems. *IEEE Trans. Software Eng.*, SE-8(4):401–412, Jul 1982.

[Cas81] L.M. Casey. Decentralized scheduling. *Australian Comput. J.*, 13:58–63, May 1981.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *15th Int. Symp. on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[CC94] A. Costes and B. Courtois. Composants matériels. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*, pages 47–54. Masson, Paris, France, 1994.

[CCG+95] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. *3rd Annual PVM user's Group Meeting, Pittsburgh, PA*, May 1995.

[CCK+95] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, USA, Feb 1995.

[CD73] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[CD89] B. Chor and C. Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–558, Apr 1989.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, Feb 1988.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, Feb 1985.

[CL94] P. Caspi and J.C. Laprie. Tolérance aux fautes logicielles. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*, pages 85–92. Masson, Paris, France, 1994.

[Con96] D. Conan. *Tolérance aux Fautes par Recouvrement Arrière dans les Systèmes Informatiques Répartis*. PhD thesis, Université Paris 6, 1996.

[Cra90] R. Crandall. Tales of Godzilla: Adventures in distributed computing. Technical Report http://www.liv.ac.uk/HPC/farHomepage.html, NeXTon Campus, 1990.

[Cri88] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Technical Report RJ 5964, IBM Research Laboratory, Mar 1988.

[Cri89] F. Cristian. Probabilistic clock synchronization. In *9th International Symposium on Distributed Computing Systems*, pages 288–296, Newport Beach, CA, Jun 1989.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):57–78, Feb 1991.

[Cro78] Y. Crouzet. *Conception de circuits à large échelle d'intégration totalement autotestables*. PhD thesis, INP Toulouse, 1978.

[CS92] C. Cap and V. Strumpen. The PARFORM: A high performance platform for parallel computing in a distributed workstation environment. Technical Report ifi-92.06, Department of Computer Science, University of Zurich, Jun 1992.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar 1996.

[CTG93] T. D. Crainic, M. Toulouse, and M. Gendreau. Towards a taxonomy of parallel tabu search algorithms. Technical Report CRT-933, Centre de Recherche sur les Transports, Université de Montréal, Montréal, Canada, Sep 1993.

[Dep94] Dept. Pharmaceutical Chemistry, University of California. *The Batch Reference Guide, 3rd Edition, Batch Version 4.0*, Mar 1994.

[DL88] C. Dwork and N. Lynch. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr 1988.

[DLP+86] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, Jul 1986.

[DO87] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, Germany, Sep 1987.

[dTA94] Observatoire Français des Techniques Avancées. *Informatique Tolérante aux Fautes*. Masson, Paris, France, 1994.

[DVCL93] G. Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of checkpointing and rollback techniques. Technical Report 03.1.8 and 03.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, Belgium, Jun 1993.

[ea95] D.B. Terry et al. Managing update conflicts in a weakly connected replicated storage system. In *15th Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, CO, Dec 1995. ACM Press.

[ELD+96] D.H.J. Epema, M. Livny, R. V. Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[ELZ86] D.L. Eager, E.D. Lazowka, and J. Zahorjan. A comparaison of receiver-initiated and sender-initiated adaptive load sharing. *North-Holland Performance Evaluation*, 6(1):53–68, Mar 1986.

[ENE92a] W. Zwaenepoel E. N. Elnozahy. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. In *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, pages 526–531, May 1992.

[ENE92b] W. Zwaenepoel E. N. Elnozahy. Replicated distributed processes in Manetho. In *22th International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 18–27, Jul 1992.

[EZ94] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *24th International Symposium on Fault-Tolerant Computing*, pages 298–307, Jun 1994.

[Ezz86] A.K. Ezzat. Load balancing in NEST: A network of workstations. In *ACM/IEEE Fall Joint Computer Science*, pages 1138–1149, Dallas, Texas, 1986.

[FB89] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, Jan 1989.

[FLP85] M. J. Fisher, N. A. Lynch, and A. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[Fol92] B. Folliot. *Méthodes et Outils de Charge pour la Conception et la Mise en œuvre d'Applications dans les Systèmes Réparties Hétérogènes*. PhD thesis, Université Paris 6, Institut Blaise Pascal, 1992.

[FPR95] E. Fromentin, N. Plouzeau, and M. Raynal. An introduction to the analysis and debug of distributed computations. In *1st IEEE Int. Conf. Algorithms and Architectures for Parallel Processing*, pages 545–554, Brisbane, Australia, Apr 1995.

[FR90] D. G. Feitelson and Larry Rudolph. Wasted resources in gang scheduling. In *The 5th Jerusalem Conference on Information Technology*, pages 127–136, Jerusalem, Israel, Oct 1990.

[FR92] D.G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, Dec 1992.

[FYN88] D. Ferguson, Y. Yemini, and C. Nikolaou. Micro-economic algorithms for load balancing in distributed computer systems. In *International Conference on Distributed Computing Systems*, pages 491–499. IEEE, 1988.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mauchek, and V. Sundernam. PVM: A user's guide and tutorial networked parallel computing. *The MIT Press Cambridge*, 1994.

[Gel89] E. Gelenbe. *Multiprocessor Performance*. 1989.

[GJK93] D. Gelernter, M. Jourdenais, and D. Kaminsky. Piranha scheduling: Strategies and their implementation. Technical Report 983, Department of Computer Science, Yale University, Sep 1993.

[Glo89] F. Glover. Tabu search - part I. *ORSA Journal of Computing*, 1(3):190–206, 1989.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Trans. on Computer Systems*, 9(3):242–271, Aug 1991.

[GMW93] C.J. Gittings, J.S. Morgan, and J. Wetherall. Far: A tool for exploiting spare workstation capacity. Technical Report draft paper, www.liv.ac.uk/HPC/farHomepage.html, Computer Science Department, University of Liverpool, 1993.

[GPR+90] D.P. Ghormley, D. Petrou, S.H. Rodrigues, A.M. Vahdat, and T.E. Anderson. Glunix: A global layer Unix for a network of workstations. Technical Report CA 94720, University of California at Berkeley, Aug 1990.

[Gre95] T.P. Green. *DQS 3.0.2 Readme/Installation Manual*. Supercomputer Computations Research Institute, Florida State University, Tallassee, Jul 1995.

[GS95] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: Bridging the gap. In *International Workshop on Theory and Practice in Distributed Systems*. Springer-Verlag, LNCS, Vol. 938, 1995.

[GS97] R. Guerraoui and A. Schippper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, Apr 1997.

[Haf98] Z. Hafidi. *MARS: Un environnement de programmation parallèle adaptative dans les réseaux de machines hétérogènes multi-utilisateurs*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, 1998.

[Ham50] Hamming. Error detecting and error correcting codes. *Bell Systems Technical Journal*, 1950.

[HMR93] J.M. Helary, A. Mostefaoui, and M. Raynal. Déterminer un état global dans un système réparti. Technical Report 769, Institut de Recherche en Informatique et Systèmes Aléatoires, Oct 1993.

[Hol75] J. H. Holland. *Adaptation in natural and artificial systems*. Michigan Press University, Ann Arbor, MI, USA, 1975.

[HTG97] Z. Hafidi, E. G. Talbi, and J-M. Geib. Parallel adaptive stochastic heuristics. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications PDPTA'97*, Las-Vegas, USA, June 1997.

[HTI97] M.C. Hsueb, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *IEEE Computer*. 30(4):75–82, Apr 1997.

[HY86] T. Hadzilacos and M. Yannakakis. Deleting completed transactions. In *5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 43–47, Cambridge, MA, Mar 1986.

[IÖ98] M. Iverson and F. Özgüner. Dynamic competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In *Seventh Heterogeneous Computing Workshop (HCW'98)*, pages 70–78, Orlando, Florida, USA, Mar 1998.

[Jac91] J. Jacob. The basic integrity theorem. In *International Symposium on Security and Privacy*, pages 89–97, Okland, CA, USA, 1991. IEEE Computer Society Press.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[Jew91] D. Jewett. Integrity S2: A fault-tolerant Unix platform. In *21st Int. Symposium on Fault Tolerant Computing*, pages 512–519, Montréal, Québec, 1991. IEEE Computer Society Press.

[KA99] Y.K. Kwok and I. Ahmad. Parallel program scheduling techniques. In R. Buyya, editor, *High Performance Cluster Computing, Volume I: Architectures and Systems*, pages 553–578. Prentice-Hall, 1999.

[Kam94] D. L. Kaminsky. *Adaptive Parallelism with Piranha*. PhD thesis, Yale University, 1994.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[KL70] B. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49:291, 1970.

[KLS86] N. Kronenberg, H. Levy, and W. Strecker. Vaxclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[KS94] A.D. Kshemkalyani and M. Singhal. Efficient detection and resolution of generalized distributed deadlocks. *IEEE Transactions on Software Engineering*, 20(1):43–54, Jan 1994.

[KTG97a] D. Kebbal, E. G. Talbi, and J. M. Geib. A new approach for checkpointing parallel applications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1643–1651, Las Vegas, Nevada, USA, Jun 1997.

[KTG97b] D. Kebbal, E. G. Talbi, and J. M. Geib. Un algorithme efficace pour la définition de points de reprise des applications parallèles. In *9ème Rencontres Francophones du Parallélisme*, EPF Lausanne, Suisse, May 1997.

[Kun91] T. Kunz. The influence of different workload descriptions on a heuristic load balancing. *IEEE Transactions on Software Engineering*, 17(7):725–730, Jul 1991.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jul 1978.

[Lap94a] J. C. Laprie. Aspects économiques. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*, pages 93–96. Masson, Paris, France, 1994.

[Lap94b] J. C. Laprie. Concepts de base de la tolérance aux fautes. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*, pages 29–46. Masson, Paris, France, 1994.

[LBK+99] H. Lamehamedi, A. Bensaid, D. Kebbal, E-G. Talbi, and A Benllaachia. Adaptive programming: Application to a semi supervised point prototype clustering algorithm. In *International Conference on Parallel and Distributed Processing Techniques and Applications, Languages and Environments*, volume 6, pages 2753–2759, Las Vegas, Nevada, USA, Jun 1999.

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *11th ACM Symposium on Operating Systems Principles*, pages 111–122, Austin, Nov 1987. ACM Press.

[Lei95] F.T. Leighton. *Introduction aux algorithmes et architectures parallèles*. International Thomson Publishing France et Morgan and Kaufmann Publishers, Inc., 1995.

[LFS93] J. Leon, A. L. Fisher, and P. Steenkiste. Fail-Safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University Pittsburgh, Feb 1993.

[LLM88] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor: A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, San Jose, California, 1988.

[LLSG90] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semanctics of distributed services. In *10th ACM Symposium on Principles of Distributed Computing*, pages 43–58, Quebec, Aug 1990. ACM Press.

[LMP94] G. Le Lann, P. Minet, and D. Powell. Systèmes réparties. In Observatoire Français des Techniques Avancées, editor, *Informatique Tolérante aux Fautes*, pages 55–71. Masson, Paris, France, 1994.

[LMS85] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, Jan 1985.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, Jul 1982.

[LV90] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed scheduling policies. In *1990 ACM Sigmetrics Conf, Performance Evaluation Review 18, 226-36*, May 1990.

[LY87] T.H. Lai and T.H. Yang. On distributed snapshots. *Inf. Proc. Letters*, 25:153–158, 1987.

[Mai93] J. Maier. Pact: A fault tolerant parallel programming environment. In *1st International Workshop 'Software for Multiprocessors and Supercomputers: Theory, Practice, Experience' SMS TPE 93*, St Petersburg, Russia, Feb 1993.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In *Int. Workshop on Parallel and Distributed Systems*, pages 215–226, North-Holland, 1988.

[MDP+96] D.S. Milojicic, F. Douglis, Y. Pandaveine, R. Wheeler, and S. Zhou. Process migration. Technical report, The Open Group Research Institute, Cambridge Center, Cambridge, MA, Dec 1996.

[MEB88] S. Majumdar, D. Eager, and R. Brunt. Scheduling in multiprogrammed parallel systems. In *ACM SIGMETRICS 1988 Conf. on Measurment and Modeling of Computer Systems*, pages 104–113, Santa Fe, NM, May 1988.

[MFSW95] C.P. Malloth, P. Felher, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large-scale networks. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, TX, Oct 1995. IEEE Computer Society Press.

[MNS97] D. Manivannan, R.H.B. Netzer, and M. Singhal. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):623–627, Jun 1997.

[MO94] O. Martin and S.W. Otto. Combining simulated annealing with local search heuristics. In G. Laporte and I. Osman, editors, *Metaheuristics in Combinatorial Optimization*. 1994.

[Mon96] T. Monteil. *Etude de Nouvelles Approches pour les Communications, l'Observation et le Placement de Tâches dans l'Environnement de Programmation Parallèle LANDA*. PhD thesis, Laboratoire d'Analyse et d'Architecture des Systèmes, Toulouse, 1996.

[MRT+90] S. Mullender, G. Van Rossum, A. Tanenbaum, R. Van Renesse, and H. Van Staveren. Amoeba: A distributed operating system for the 1990's. *IEEE Computer*, 23(5):44,53, May 1990.

[MS92] S. Mishra and R. D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR92-19, Department of Computer Science, The university of Arizona, Aug 1992.

[MS98] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop (HCW'98)*, pages 57–69, Orlando, Florida, USA, Mar 1998.

[MTP99] N. Melab, E-G. Talbi, and S. Petiton. A parallel adaptive version of the block-based Gauss-Jordan algorithm. In Lecture Notes in Computer Science Jose Rolim, editor, *IEEE IPPS/SPDP'99 (Int. Parallel Processing Symposium / Symposium on Parallel and Distributed Processing)*, San Juan, Puerto Rico, USA, Apr 1999.

[Nam96] R. Namyst. *PM2: Un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, France, 1996.

[Nel90] V.P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, Jul 1990.

[Neu56] Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies, Princeton*, 1956.

[NH85] L.M. Ni and K. Hwang. Optimal load balancing in a multiple processor system with many job classes. *IEEE Trans. Software Eng.*, SE-11(5):491–496, May 1985.

[NS88] D. M. Nicol and J. H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37(9):1073–1087, Sep 1988.

[NS93] H. Nishikawa and P. Steenkiste. General architecture for load balancing in distributed-memory environments. In *13th IEEE International Conference on Distributed Computing*, May 1993.

[NX95] 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, Feb 1995.

[OCD+88] J.K. Ousterhout, A.R. Cherrenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, pages 23–36, Feb 1988.

[OSS80] J. Ousterhout, D. Scelza, and P. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, Feb 1980.

[Ous82] J. Ousterhout. Scheduling techniques for concurrent systems. In *3rd Int. Conf. Distributed Computing Systems*, pages 22–30, Oct 1982.

[PBkL94] J.S. Plank, M. Beck, G. kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. Technical Report TN37996, CS-94-242, Department of Computer Science, University of Tennessee, Knoxville, Aug 1994.

[PBS+88] D. Powell, G. Bonn, D. Seaton, P. Verissimo, and F. Waeselynck. The Delta-4 approach to dependability in open distributed computing systems. In *18th IEEE Symposium on Fault Tolerant Computing*, Los Alamitos, USA, Jun 1988.

[PF89] G. Pujolle and S. Flida. *Modèles de Systèmes et de Réseaux, Tome2: Files d'attente*. Eyrolles, 1989.

[PKD94] J. S. Plank, Y. Kim, and J. J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. Technical Report UT-CS-94-268, Department of Computer Science, University of Tennessee, Dec 1994.

[PL95] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. In *Workshop on Job Scheduling for Parallel Processing, IPPS'95*, 1995.

[PL96] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[Pla93] J.S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, 1993.

[Pow88] D. Powell. Failure mode assumptions and assumption coverage. In *22th IEEE Symposium on Fault Tolerant Computing*, Boston, USA, Jun 1988.

[Pow91] D. Powell. Delta4: A generic architecture for dependable distributed computing. Technical Report RJ 5964, ESPRIT, Berlin, Germany, 1991.

[PP83] M.L. Powell and D.L. Presotto. Publishing, a reliable broadcast communication mechanism. In *9th ACM Symposium on Operating Systems Principles*, Bretton Woods, USA, Oct 1983.

[PS88] F. Panzieri and S.K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Trans. on Software Engineering*, SE-14(1):30–37, Jan 1988.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the Association for Computing Machinery*, 27(2):228–234, Apr 1980.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, Jun 1975.

[Rav96] P. G. Raverdy. *Gestion de ressources et répartition de charge dans les systèmes hétérogènes à grande échelle: Application aux environnements mobiles et parallèles*. PhD thesis, Université Paris 6, 1996.

[RB87] C. Reinsch and F. Bauer. Inversion of positive matrices by the Gauss-Jordan method. In J. Wilkinson and C. Reinsch, editors, *Handbook of Automatic Computation*, volume 2. Springer Verlag, 1987.

[Rei96] M.K. Reiter. Distributed trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–75, Apr 1996.

[RHB94] R. V. Renesse, T. M. Hickey, and K. P. Birman. Design and performance of Horus: A lightweight group communication system. Technical Report TR 94-1442, Department of Computer Science, Cornell University, Aug 1994.

[Rot94] H. G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEEE Proc. Comput. Digit. Tech*, 141(1), Jan 1994.

[RSB90] P. Ramanathan, K.G. Shin, and R.W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, Oct 1990.

[RSZ89] K. Ramamritham, J. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transact. Comput.*, 38(8):1110–1123, Aug 1989.

[Rue89] A. Ruegg. *Processus stochastiques.* Presses polytechniques romandes, 1989.

[RV95] L. Rodrigues and P. Verissimo. Causal separators for large-scale multicast communication. In *15th International Conference on Distributed Computing Systems*, pages 83–91, May 1995.

[SBY88] R.E. Strom, D.F. Bacon, and S. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *18th Int. Symposium on Fault-Tolerant Computing*, pages 44–49, 1988.

[SC89] K.G. Shin and Y. Chang. Load sharing in distributed real time systems with state change broadcasts. *IEEE Transact. Comput.*, 38(8):1124–1142, Aug 1989.

[Sch90] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM, Computing Surveys*, 22(4):299–319, Dec 1990.

[SCZL96] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The easy - LoadLeveler API project. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162. Springer Verlag, 1996.

[Sen94] P. Sens. *Conception et mise en œuvre d'une plate-forme logicielle de tolérance aux fautes pour le support d'applications réparties.* PhD thesis, Université Paris 6, 1994.

[SH82] J.F. Shoch and J.A. Hupp. The Worm programs - early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, 1982.

[Sha48] Shannon. A mathematical theory of communications. *Bell Systems Technical Journal*, 1948.

[Sie91] D.P. Siewiorek. Architecture of falut-tolerant computers: An historical perspective. In *IEEE, vol 79, number 12*, Dec 1991.

[SK90] N. G. Shivaratri and P. Krueger. Two adaptive location policies for global sheduling algorithms. In *10th International Conference on Distributed Computing Systems*, pages 502–509, Paris, France, May 1990.

[Ske82] D. Skeen. Non-Blocking commit protocols. In *ACM SIGMOD Conference on Management of Data*, pages 133–147, Orlando, Florida, Jun 1982.

[SL93] G. C. Sih and E. A. Lee. A compile time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4:175–187, Feb 1993.

[SM77] R.M. Shapiro and R.E. Millstein. Reliability and fault recovery in distributed processing. *In Oceans'77 Conference Record, Vol II*, 1977.

[Smi80] R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comput.*, C-29(12):1104–1113, Dec 1980.

[SN96] V. A. Saletore and T. F. Neff. Metasystem: Integrating a parallel computer and a heterogeneous workstation cluster. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 493–498, Sunnyvale, California, Aug 1996.

[Sof95] Genias Software. Codine: Computing in distributed networked environments. Technical report, Genias Software, Sep 1995.

[SS84] J.A. Stankovic and I.S. Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. In *International Conference on Distributed Computing Systems*, pages 49–59, 1984.

[ST85] C.C. Shen and W.H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, Mar 1985.

[Sta85] J.A. Stankovic. An application of bayesian decision theory to decentralized control of job scheduling. *IEEE Transact. Comput.*, C-34(2):117–130, Feb 1985.

[Ste95] G. Stellner. Ressource management and checkpointing for PVM. In *2nd European User's Group Meeting*, pages 131–136, Sep 1995.

[STHI96] K. Suzaki, H. Tanuma, S. Hirano, and Y. Ichisugi. Time sharing systems that use a partitioning algorithm on mesh-connected parallel computers. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 268–275, Sunnyvale, California, Aug 1996.

[Sup94] J. Suplick. An analysis of load balancing technology. Technical Report CXSOFT, Richardson, Texas, Jan 1994.

[Sve90] A. Svenson. History, an intelligent load sharing filter. In *10th International Conference on Distributed Computing Systems*, pages 546–553, Paris, May 1990.

[SW89] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *18th Symposium on Principles of Distributed Computing*, pages 223–238. ACM SIGACT/SIGOPS, Aug 1989.

[SY85] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug 1985.

[Tai93] E. Taillard. Parallel iterative search methods for vehicle routing problem. *Networks*, 23:661–673, 1993.

[Tal93] E-G. Talbi. *Allocation de processus sur les architectures parallèles à mémoire distribuée*. PhD thesis, Institut National Polytechnique de Grenoble, 1993.

[Tal98] E-G. Talbi. A taxonomy of hybrid meta-heuristics. Technical Report TR-183, Laboratoire d'Informatique Fondamentale de Lille, May 1998.

[TGHK98] E-G. Talbi, J-M. Geib, Z. Hafidi, and D. Kebbal. A fault-tolerant parallel heuristic for assignment problems. In José Rolim, editor, *BioSP3 Workshop on Biologically Inspired Solutions to Parallel Processing Systems, in IEEE IPPS/SPDP'98 (Int. Parallel Processing Symposium / Symposium on Parallel and Distributed Processing*, volume 1388 of *LNCS*, pages 306–314, Orlando, USA, Apr 1998. Springer-Verlag.

[Tin95] O. Tingleff. Systems of linear equations solved by block Gauss-Jordan method using a transputer cube. Technical Report IMM-REP-1995-08, Institute of Mathematical Modelling, Technical University of Danmark, 1995.

[TL88] M.M. Theimer and K.A. Lantz. Finding idle machines in workstation-based distributed system. In *8th International Conference on Distributed Computing Systems*, pages 112–122, San Jose, CA, USA, Jun 1988. IEEE.

[TL95] T. Tannenbaum and M. Litzkow. Checkpointing and migration of Unix processes in the Condor distributed processing system. *Dr. Dobbs Journal*, pages 40–48, Feb 1995.

[TLC85] M.M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemptable remote execution facilities for the V-system. In *10th Symposium on Operating Systems Priciples*, pages 2–12, Dec 1985.

[Tur93] L.H. Turcotte. A survey of software environments for exploiting networked computing. Technical report, Mississipi State University, Jun 1993.

[Vai97] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8):942–947, Aug 1997.

[Vog96] W. Vogels. The private investigator. Technical report, Department of Computer Science, Cornell University, Apr 1996.

[WF93] K. Wong and M. Franklin. Distributed computing systems and checkpointing. In *Second International Symposium on High Performance Distributed Computing, Spokane, Washington*, Jul 1993.

[WHH+92] C.A. Waldspurger, T. Hogg, B.A. Hubermann, J.O. Kephart, and W.S. Stornetta. Spawn: A distributed computational economy. *IEEE Trans. Software Eng.*, 18(2):103–117, Feb 1992.

[WHV+95] Y.M. Wang, Y. Huang, K.P. Vo, P.Y. Chung, and C. Kintala. Checkpointing and its applications. In *25th IEEE Symposium on Fault Tolerant Computing*, Jun 1995.

[WM85] Y.T. Wang and R.J.T. Morris. Load sharing in distributed systems. *IEEE Trans. Comput.*, C-34(3):204–217, Mar 1985.

[XN93] J. Xu and R.H.B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *5th IEEE Symposium on Parallel and Distributed Processing*, pages 754–761, Dallas, TX, Dec 1993.

[ZF87] S. Zhou and D. Ferrari. An experimental study of load balancing performance. Technical Report UCB/CSD 87/336, University of California, Berkeley, CA, Jan 1987.

[Zho88] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, Sep 1988.

[ZMJ+99] B.B. Zhou, P. Mackerras, C.W. Johnson, D. Walsh, and R.P. Brent. An efficient resource allocation scheme for gang scheduling. In *IEEE International Workshop on Cluster Computing*, pages 187–194, Melbourne, Australia, Dec 1999.

[ZZWD92] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical Report 257, Computer Systems Research institute, University of Toronto, Canada, Apr 1992.

Annexe A

Interface de programmation

Cette annexe présente l'API (Application Programmer Interface) de MARS. Une présentation détaillée comportant un manuel de programmation est disponible avec le code source du système sur le serveur web du lifl "**www.lifl.fr/~kebbal**". L'API est découpée en deux parties: fonctions exécutées par le maître et fonctions dédiées aux esclaves.

Pour l'utilisation de ces primitives, le fichier d'entête **mars.h** doit être inclu (`#include <mars.h>`).

A.1 Primitives exécutées par le maître

A.1.1 Enrôlement

```
vcid mars_init(module_status master_or_worker, NULL);
void mars_exit(void);
void mars_waitexit(void);
enum module_status {
        MASTER = 0, WORKER = 1,
};
typedef enum module_status module_status;
```

La primitive `mars_init` permet à l'application de s'identifier auprès de l'ordonnanceur global. Le premier paramètre doit être la constante MASTER pour le maître. Cette fonction doit être la première fonction de l'API appelée avant de solliciter tout autre service.

La routine `mars_exit` est appelée par le maître pour informer l'ordonnanceur global de la terminaison propre de l'application. La primitive `mars_waitexit` permet de bloquer le maître en attendant la terminaison des calculs.

A.1.2 Demande de ressources

```
int mars_spawn(char *WorkerModule, char **args, int **Tids, int Min, int Max);
```

Cette primitive permet à l'application de spécifier la quantité de ressources qu'elle souhaite acquérir, afin que l'ordonnanceur global puisse considérer sa requête. Les ressources sont des stations de travail en général. La fonction retourne le nombre de ressources allouées et les esclaves sont créés par la bibliothèque. Les paramètres de la fonction sont :

- **WorkerModule** : nom du fichier exécutable. Ce fichier doit figurer dans tous les répertoires associés aux architectures considérées.
- **args** : vecteur de chaînes de caratères comportant les arguments de l'esclave.
- **Min, Max** : bornes du degré de parallélisme. La constante ANY peut être utilisée pour indiquer un nombre quelconque.

La fonction doit être appelée une seule fois. Dans le cas où la satisfaction de la requête est impossible (Min > nombre total de nœuds de la configuration par exemple), la fonction retourne *MarsOutOfRes*.

A.1.3 Gestion de l'espace de travail

Construction de l'application :

```
int mars_inserttask(any_t work, any_t pending_work, any_t result,
 char *worker_thread, int *dependencies, int nb_dep);
int mars_freetask(int task_key);
void mars_endtasks(void);
typedef void *any_t;
```

La primitive `mars_inserttask` permet d'insérer une tâche dans l'espace de travail. Les paramètres requis sont : un pointeur sur la structure contenant le travail initial, un second sur le travail intermédiaire et un troisième sur la structure contenant les résultats. Le nom de la fonction spécialisée dans le traitement du type de la tâche doit être également spécifié. Un vecteur d'entiers, contenant les identificateurs des tâches dont dépend la tâche insérée, est également requis.

La routine `mars_freetask` permet de libérer l'espace occupé par la tâche. La primitive `mars_endtasks` indique la fin de la génération de tâches par l'application.

Gestion dynamique :

```
void mars_swork_func(mars_order_func function);
```

```
void mars_spendwork_func(mars_order_func function);
void mars_sres_func(mars_order_func function);

typedef void (*mars_order_func) (Order_Struct *arg);
struct Order_Struct {
        int flag;
        int key;
        any_t work;
        any_t pending_work;
        any_t result;
};
typedef struct Order_Struct Order_Struct
```

Ces primitives sont utilisées pour positionner les fonctions utilisées éventuellement pour la gestion dynamique de l'espace de travail.

Les primitives `mars_swork_func`, `mars_spendwork_func` et `mars_sres_func` permettent de spécifier les fonctions qui doivent être appelées respectivement à chaque allocation de tâche, à chaque remise de tâche partielle et à chaque retour de résultat. La fonction appelée à chaque terminaison de tâche permet d'effectuer certaines opérations telles que la génération dynamique de tâches, l'indication de la fin de génération de tâches ou de la fin de l'application. La structure de données `Order_Struct` comporte le contexte de la tâche qui est composé de pointeurs sur les structures comportant le travail initial, le travail intermédiaire et les résultats ainsi que de l'identificateur de la tâche et d'un flag indiquant s'il s'agit d'un travail initial ou intermédiaire.

Empaquetage/désempaquetage de données :

```
void mars_spackwork_func(mars_pk_func function);
void mars_sunpackres_func(mars_upk_func function);
void mars_sunpackpwork_func(mars_upk_func function);
typedef void (*mars_pk_func) (Order_Struct *arg);
typedef void (*mars_upk_func) (Order_Struct *arg);
```

Ces routines spécifient les fonctions permettant d'empaqueter et de désempaqueter les données et les résultats de la tâche au niveau du maître. Les primitives `mars_spackwork_func`, `mars_sunpackres_func` et `mars_sunpackpwork_func` positionnent les fonctions permettant respectivement :

- l'empaquetage des données de la tâche au moment de son allocation ;
- le désempaquetage des résultats de l'exécution de la tâche ;
- le désempaquetage des données et des résultats de la tâche partielle.

Retrait de tâches :

```
void mars_retreatActiveTasks(void);
```

Cette routine permet au maître d'effectuer une sorte de barrière de synchronisation, en rappelant toutes les tâches en exécution. Les tâches retirées sont retournées comme étant des tâches partiellement traitées.

A.1.4 Tolérance aux fautes

```
void mars_setckpt_mode(CKPT_MODE flag);
int mars_setckpt_period(unsigned int period);
int mars_ckpt(char *ckptfile);
$prompt> MatserModule -restore  ckptfile
enum CKPT_MODE {
        STAT_PERIOD = 0,
        DYN_PERIOD  = 1,
        MAN_CKPT    = 2,
};
typedef enum CKPT_MODE CKPT_MODE;
```

La primitive `mars_setckpt_mode` permet de spécifier le mode de sauvegarde. `flag` peut être STAT_PERIOD : période statique, DYN_PERIOD : période dynamique définie par le système, MAN_CKPT : l'application se charge de gérer explicitement le mécanisme de sauvegarde. Cette fonction peut être appelée plusieurs fois.

La primitive `mars_setckpt_period` permet de spécifier la période de sauvegarde souhaitée en secondes. Son appel met automatiquement le mode de sauvegarde périodique avec période statique.

La fonction `mars_ckpt` permet d'effectuer une opération de sauvegarde de l'application au moment de son appel. Le résultat sera mis dans le fichier `ckptfile`.

Le lancement du maître sur une ligne de commande sans les arguments habituels et avec l'option *–restore* provoque la reprise de l'application à partir du fichier de sauvegarde `ckptfile`.

A.1.5 Divers

```
void mars_perror(char *msg);
void mars_stats(int flag, char *appli_res_file, char *tasks_res_file);
enum {
        MARS_APPLI_STATS  = 1,
        MARS_TASK_STATS   = 3,
```

```
        MARS_GLOBAL_STATS = 5
};
```

La routine `mars_perror` affiche le dernier message relatif à une erreur retournée par un appel à une fonction de l'interface. La chaîne de caractères passée en paramètre `msg` sera concaténée au message imprimé.

La routine `mars_stats` permet de reporter des statistiques sur l'exécution de l'application. Les statistiques seront placées dans les fichiers `appli_res_file` et `tasks_res_file` selon la valeur de `flag`. L'option `MARS_APPLI_STATS` permet de collecter des statistiques sur l'exécution de l'application, `MARS_TASK_STATS` sur les tâches et `MARS_GLOBAL_STATS` permet de communiquer certaines de ces statistiques à l'ordonnanceur global. Les résultats sont rassemblés dans un répertoire destiné aux statistiques. La combinaison de ces options est possible en utilisant le "ou sur les bits".

A.2 Primitives exécutées par les esclaves

A.2.1 Enrôlement

```
void mars_init(module_status master_or_worker, NULL);
void mars_waitexit(void);
```

La primitive `mars_init` permet à l'esclave de s'identifier auprès du système. Le premier paramètre doit avoir la valeur WORKER. Cette fonction doit être la première fonction de l'API appelée avant de solliciter tout autre service.

La primitive `mars_waitexit` permet de bloquer l'esclave en attendant d'être terminé par le système.

A.2.2 Interaction avec le serveur de travail

Demande de travail :

```
mars_worker_thread mars_getwork(any_t work, int *res);
typedef any_t (*mars_worker_thread) (any_t arg);
```

Cette routine effectue un appel au serveur de travail au niveau du maître afin d'obtenir du travail. Un pointeur sur une structure `work`, dans laquelle sont rangées les données de la tâche obtenue, doit être fourni. Le paramètre `res` peut être: `NEW_WORK` pour une nouvelle tâche, `PENDING_WORK` pour une tâche partielle ou `NO_WORK` s'il n'y a pas de travail. La valeur de retour de la fonction est un pointeur sur la fonction à exécuter par le thread créé.

Restour de résultat :

```
void mars_putbackresults(any_t result);
```

Après avoir effectué les traitements nécessaires sur la tâche obtenue, cette fonction est utilisée pour retourner les résultats au maître. Un pointeur sur une structure comportant les résultats à expédier doit être passé à la fonction.

Retour de travail partiel :

```
void mars_putbackwork(any_t pending_work);
```

De la même manière, cette primitive permet de retourner la tâche partielle au maître. Cette fonction est utilisée généralement dans la fonction de repli spécifiée par l'utilisateur.

Empaquetage/désempaquetage de données :

```
void mars_sunpackwork_func(mars_upkw_func function);
void mars_spackres_func(mars_pkw_func function);
void mars_spackpwork_func(mars_pkw_func function);
typedef void (*mars_pkw_func) (any_t arg);
typedef void (*mars_upkw_func) (any_t arg);
```

Comme dans le cas du maître, les primitives `mars_sunpackwork_func`, `mars_spackres_func` et `mars_spackpwork_func` permettent de spécifier les fonctions destinées à:

- désempaqueter les données de la tâche obtenue;
- empaqueter les résultats du traitement de la tâche;
- empaqueter les données de la tâche partielle à retourner.

Le paramètre de la fonction d'empaquetage ou de désempaquetage est un pointeur sur une structure contenant les données à empaqueter ou à désempaqueter.

A.2.3 Gestion des threads de travail

```
void mars_sworkerthread(mars_worker_thread wt, any_t arg, char *wt_name);
pthread_t mars_startworkerthread(mars_worker_thread wt, any_t arg);
typedef any_t (*mars_worker_thread) (any_t arg);
```

La routine `mars_sworkerthread` permet de construire la table de threads de travail supportés par les esclaves. Chaque entrée de la table contient dans l'ordre: un pointeur sur la fonction implémentant le thread, ses paramètres éventuels et le

nom de la fonction qui doit être comparé avec celui obtenu avec la tâche lors de l'appel à la routine `mars_getwork`.

La routine `mars_startworkerthread` permet de lancer le thread dont la fonction et les paramètres sont passés en paramètre.

A.2.4 Repli

Spécification de la fonction de repli :

```
void mars_sputbackwork_func(mars_putback_func putback, any_t arg);
typedef void (*mars_putback_func) (any_t arg);
```

Cette fonction permet de rassembler les résultats et le travail restant de la tâche partielle en cas de repli, de sauvegarde ou de retrait explicite de la tâche. La primitive `mars_putbackwork` doit être ensuite appelée afin de remettre la tâche au serveur de travail.

Sections critiques :

```
void mars_lock_fold(void);
void mars_unlock_fold(void);
```

Ces deux primitives permettent de réaliser des sections critiques qui empêcheront le repli, le temps de finaliser certains traitements critiques, dans un souci de préservation de cohérence.

Annexe B

Calcul de la période de sauvegarde optimale

B.1 Chaînes de Markov

Les chaînes de Markov sont un formalisme permettant de modéliser mathématiquement de nombreux problèmes rencontrés dans la pratique.

Une chaîne de Markov est décrite par une suite de variables aléatoires $\{X_n, n = 0, 1, 2, ...\}$ [Rue89]. L'espace des états S est discret (fini ou dénombrable) et prend ses valeurs souvent dans $\mathbb{N}$. Si on connait pour chaque paire d'états (i, j) et pour chaque instant t_n, la probabilité $p_{ij}(n)$ de transition du processus vers l'état j à l'instant t_{n+1} sachant qu'il était à l'état i à l'instant t_n :

$$p_{ij}(n) = P(X_{n+1} = j | X_n = i).$$

$p_{ij}(n)$ est alors appelé *probabilité de transition à une étape* [Rue89].

B.1.1 Chaînes de Markov à temps discret

Si les probabilités de transition décrites précédemment existent pour tous les états de S, alors $p_{ij}(n) = P(X_{n+1} = j | X_n = i), (\forall i, j \in S)$. Ceci signifie que les probabilités de transition ne dépendent pas du comportement du processus antérieur à l'instant n et que l'état futur du processus ne dépend que de l'état présent (propriété de Markov).

Par la suite, on utilisera les chaînes de Markov homogènes dans le temps, pour lesquelles, les probabilités de transition sont indépendantes du temps $(p_{ij}(n) = p_{ij})$.

$\lim_{t\to\infty} p_i(t) = p_i$ existent et définissent une distribution de probabilité indépendante de la distribution initiale [Rue89]. Ceci implique que $\lim_{t\to\infty} p_i'(t) = 0$. Le système d'équations de Kolmogorov se transforme en équations de balance:

$$\sum_{k\neq n} p_k \lambda_{kn} = p_n \sum_{j\neq n} \lambda_{nj} \; ; \; (n \in \mathbb{N}). \tag{B.1}$$

B.2 Détermination de la période de sauvegarde optimale

La matrice des taux de transition de la chaîne de Markov continue, correspondant à notre système (figure 3.4, chapitre 3), est donnée ci-dessous:

$$P = \begin{array}{c|cccc} & E & C & D & R \\ \hline E & 1-(a+d) & a & d & 0 \\ C & b & 1-(b+d) & d & 0 \\ D & 0 & 0 & 1-e & e \\ R & c & 0 & d & 1-(c+d) \end{array}$$

Résultat 1 *La chaîne de Markov définie ci-dessus est à états finis $\{E, C, D, R\}$ et comprend une unique classe récurrente (irréductible). Par conséquent, elle possède une unique distribution stationnaire.*

En appliquant la formule B.1, nous obtenons les équations de balance suivantes:

$$\begin{aligned}
p_1\lambda_{10} + p_2\lambda_{20} + p_3\lambda_{30} &= p_0(\lambda_{01} + \lambda_{02} + \lambda_{03}) && \text{(B.2)}\\
p_0\lambda_{01} + p_2\lambda_{21} + p_3\lambda_{31} &= p_1(\lambda_{10} + \lambda_{12} + \lambda_{13}) && \text{(B.3)}\\
p_0\lambda_{02} + p_1\lambda_{12} + p_3\lambda_{32} &= p_2(\lambda_{20} + \lambda_{21} + \lambda_{23}) && \text{(B.4)}\\
p_0\lambda_{03} + p_1\lambda_{13} + p_2\lambda_{23} &= p_3(\lambda_{30} + \lambda_{31} + \lambda_{32}) && \text{(B.5)}\\
p_0 + p_1 + p_2 + p_3 &= 1 && \text{(B.6)}
\end{aligned}$$

En remplaçant les λ_{ij} par les taux définis dans la matrice P et les états par ceux de notre processus ($E \to 0$, $C \to 1$, $D \to 2$, $R \to 3$), nous obtenons:

$$\begin{aligned}
p_C \times b + p_D \times 0 + p_R \times c &= p_E \times (a + d + 0) && \text{(B.7)}\\
p_E \times a + p_D \times 0 + p_R \times 0 &= p_C \times (b + d + 0) && \text{(B.8)}\\
p_E \times d + p_C \times d + p_R \times d &= p_D \times (0 + 0 + e) && \text{(B.9)}\\
p_E \times 0 + p_C \times 0 + p_D \times e &= p_R \times (c + 0 + d) && \text{(B.10)}\\
p_E + p_C + p_D + p_R &= 1 && \text{(B.11)}
\end{aligned}$$

B.1.1.1 Probabilité de transition à n étapes

Si $p_{ij}^{(n)}$ est la probabilité de transition de l'état i à j en n étapes, alors $p_{ij}^{(n)} = P(X_{n+k} = j | X_k = i)$, $(n \geq 1, k \geq 1)$.

Si $P^{(n)}$ est la matrice des probabilités de transition à n étapes, alors $P^{(n)} = (P_{ij}^{(n)})$ [Rue89].

B.1.1.2 Probabilités d'état

La probabilité d'état $\pi_k(n) = P(X_n = k)$, $(n = 0, 1, ...$ et $k = 1, 2, ...)$ peut être utilisée pour définir la distribution de X_n. La distribution de X_n peut être écrite sous forme de vecteur-ligne $\pi(n) = (\pi_1(n), \pi_2(n), ...)$, tel que $\sum_{k \in S} \pi_k(n) = 1$.

Pour calculer $\pi(n)$, il faut connaître soit l'état initial (valeur prise par X_0), soit sa distribution initiale $\pi(0)$. En notation matricielle, $\pi(n) = \pi(0)P^n$ [Rue89].

B.1.1.3 Régime transitoire et régime permanent

Les probabilités d'état $\pi_k(n) = P(X_n = k)$ sont exprimées en fonction du nombre n de transitions, il s'agit du régime transitoire. Il est clair que la distribution de X_n varie généralement en fonction du temps et qu'elle dépend de la distribution initiale $\pi(0)$.

D'autre part, si $\pi(n)$ converge vers une distribution limite $\pi = \lim_{n \to \infty} \pi(n)$, cette dernière définit le régime permanent du processus. Dans ce cas, π doit être une distribution de probabilité. Le régime permanent n'est pas influencé par la distribution initiale $\pi(0)$. En pratique, le nombre de transitions nécessaires pour atteindre le régime permanent est fini.

B.1.1.4 Distributions stationnaires

Une chaîne de Markov est dite stationnaire si la distribution $\pi(n)$ de la variable aléatoire X_n est indépendante du temps $(n = 0, 1, 2, \ldots)$. En d'autres termes, si $\pi(0)$ est une distribution stationnaire du processus [Rue89].

B.1.1.5 Propriétés

Les propriétés suivantes sont vérifiées pour une chaîne de Markov finie [Rue89]:

- une chaîne de Markov finie possède toujours une distribution stationnaire;
- une chaîne de Markov finie admet une unique distribution stationnaire si et seulement si elle comprend une seule classe récurrente. Une classe est dite

récurrente s'il est impossible de la quitter. Si l'unique classe récurrente couvre l'ensemble des états S, la chaîne est dite *irréductible*;

- si π est la distribution limite d'une chaîne de Markov, alors π est l'unique distribution stationnaire de cette chaîne.

B.1.2 Chaînes de Markov à temps continu

De la même manière que pour les chaînes de Markov à temps discret, un processus stochastique $\{X(t); t \geq 0\}$ à états dans $\mathbb{N}$ est appelé chaîne de Markov à temps continu, si la distribution conditionnelle du processus à l'instant t, connaissant son état à l'instant t_n $(t_n < t)$, n'est pas modifiée par la connaissance de ses états à des instants antérieurs à t_n [Rue89]. $P(X(t) = j|X(t_n) = i_n, X(t_{n-1}) = i_{n-1}, \ldots, X(t_1) = i_1) = P(X(t) = j|X(t_n) = i_n)$.

Nous utilisons les chaînes de Markov homogènes dans le temps, pour lesquelles les probabilités de transition $P(X(t) = j|X(t_n) = i)$ ne dépendent pas de la position de l'intervalle $[t_n, t]$ relative à l'axe du temps, mais uniquement à sa durée $P(X(t) = j|X(t_n) = i) = p_{ij}(t - t_n)$.

Les fonctions $p_{ij}(t)$ vérifient les propriétés suivantes ([Rue89]) :

- $p_{ij}(t) \geq 0, t > 0$;
- $\sum_{j \in S} p_{ij}(t) = 1, t > 0$;
- $p_{ij}(s+t) = \sum_{k \in S} p_{ik}(s) p_{kj}(t), i, j \in S$ (équation de Chapman-Kolmogorov).

Les transitions ne se font pas d'une manière instantanée $(\lim_{t \to 0} p_{ij}(t) = 0; \forall i \neq j)$.

$P = (\lambda_{ij})_{i,j \in S}$ est appelé matrice des taux de transition de la chaîne de Markov à temps continu.

B.1.2.1 Régime transitoire

En utilisant le système d'équations de Chapman-Kolmogorov, le régime transitoire régissant le processus de Markov est décrit par la relation $p'_n(t) = \sum_{k \neq n} p_k(t) \lambda_{kn} - p_n(t) \sum_{j \neq n} \lambda_{nj}$, $(n \in \mathbb{N})$ [Rue89]. $p'_n(t)$ est le taux de variation de la probabilité attribuée à l'état n et est égal à la différence du flux d'entrée et du flux de sortie de cet état.

B.1.2.2 Régime stationnaire

Les résultats relatifs à l'existence de la distribution limite des chaînes de Markov à temps discret peuvent être démontrés pour le cas continu. La plupart des problèmes pratiques font appel à des processus stochastiques pour lesquels, les limites

La résolution de ce système permet de déterminer la valeur de p_E :

$$p_E = \frac{ec(b+d)}{dbc + d^2c + adc + ebc + edc + eac + eda + edb + ed^2 + d^2a + d^2b + d^3}. \tag{B.12}$$

En réalité, le coût du recouvrement c^{-1} comporte le temps consacré à la réparation de nœud, le temps de restauration du processus r et le temps de recalcul i.e. le temps entre la dernière sauvegarde à partir de laquelle le recouvrement est effectué et le moment de la défaillance. Cet intervalle est proportionnel à la période de sauvegarde a^{-1}. Le temps de réparation n'est pas pris en compte dans les calculs car les applications défaillantes sont recouvertes sans attendre l'éventuelle récupération du nœud (le processus est recouvert sur un autre nœud). Par conséquent, c^{-1} peut être remplacé par $r + a^{-1}$ ce qui donne :

$$p_E = \frac{e(b+d)}{(r+1/a)(\frac{1}{(r+1/a)}(db + d^2 + ad + eb + ed + ea) + eda + edb + ed^2 + d^2a + d^2b + d^3)}. \tag{B.13}$$

Pour calculer la période de sauvegarde optimale, nous devons trouver le maximum du temps d'exécution p_E. En dérivant p_E et en mettant la dérivée à 0, nous obtenons $a_{optimum}$ correspondant au maximum de la fonction :

$$a_{optimum} = \frac{\sqrt{(1+dr)(b+d)d}}{1+dr}.$$

La période optimale est donnée par :

$$a^{-1}_{optimum} = \frac{1+dr}{\sqrt{(1+dr)(b+d)d}} = \sqrt{\frac{1+dr}{(b+d)d}}. \tag{B.14}$$

B.3 Période de sauvegarde optimale en considérant la vitesse d'évolution de l'application

Si nous remplaçons a^{-1} par $V(n)a^{-1}$ ($V(n)$ étant la moyenne du taux de variation de la vitesse d'évolution de l'application, voir sous-section 3.3.2) dans la formule de c_i et après calcul de la dérivée, nous obtenons :

$$p_E = \frac{e(b+d)}{(r+V(n)/a)(\frac{1}{r+V(n)/a}(db + d^2 + ad + eb + ed + ea) + eda + edb + ed^2 + d^2a + d^2b + d^3)}. \tag{B.15}$$

Ce qui donne:

$$a_{optimum} = \frac{\sqrt{(1+dr)(b+d)dV(n)}}{1+dr}.$$

La formule de la période optimale est donnée par:

$$a_{optimum}^{-1} = \frac{1+dr}{\sqrt{(1+dr)(b+d)dV(n)}} = \sqrt{\frac{1+dr}{dV(n)(b+d)}}. \qquad \text{(B.16)}$$

Annexe C

Table des symboles

Cette annexe présente la liste des symboles, avec une brève explication, utilisés dans les différents chapitres de la thèse.

C.1 Symboles utilisés dans le chapitre 4

Ordonnancement d'applications adaptatives :

N_i : représente le nœud i de la configuration.
T_i : représente la tâche i de l'application.
$P(N_i)$: puissance relative du nœud i.
t : temps, qui correspond aux instants d'invocation de l'ordonnanceur.
$L_t(N_i)$: charge du nœud i à l'instant t.
$PL_t(N_i)$: puissance de calcul qu'offre le nœud N_i. Il s'agit du rapport entre $P(N_i)$ et $L_t(N_i)$, t représente le temps.
$ReadyTasks$: liste des tâches prêtes à l'exécution.
$AllocatedTasks$: liste des tâches en exécution.
C_t : configuration de nœuds exploités par l'application au temps t.
$Prio_t(T_i)$: priorité de la tâche T_i au temps t.
$Deps_t(T_i)$: nombre de tâches desquelles dépend la tâche T_i au temps t.
$state(N_i)$: fonction retournant l'état d'un nœud de la configuration.
C_p : constante sur la priorité supposée refléter le coût de retrait d'une tâche.
C_{PL} : constante sur la puissance relative des nœuds supposée refléter le coût de permutation des tâches.

Résultats expérimentaux :

N_{base} : nombre total normalisé de nœuds exploités par l'application.
N : nombre total de nœuds physiques utilisés par l'application.

T_i : temps total durant lequel, le nœud N_i est affecté à l'application.
PL_i : facteur PL du nœud N_i.
T_p : temps d'exécution parallèle de l'application.
S_i : ensemble des tâches dépendantes directement de la tâche T_i (successeurs directs).
$d^+_{G_t}(T_i)$: nombre de tâches desquelles dépend directement la tâche T_i par rapport au graphe de dépendances courant G_t.
C_l : constante utilisée pour normaliser l'expression de la priorité.

Analyse de performances :

$a(i,j)$: arc reliant la tâche T_i à T_j (T_j dépend directement de T_i).
M : nombre total des tâches générées par l'application.
π : densité du graphe de dépendances.
λ : taux d'arrivées (de génération) de tâches pour une distribution de Poisson.

C.2 Symboles utilisés dans le chapitre 5

Ordonnancement multi-application :

request : requête émise par une application. *request* peut être de trois types :
WORKSTATION : stations de travail sans considération de l'architecture ;
WORKSTATION_PER_TYPE : stations de travail avec considération de l'architecture ;
SPECIFIC_RESOURCE : ressources spécifiques.
min_i : quantité minimum de nœuds de type (architecture) i demandée.
max_i : quantité maximum de nœuds de type (architecture) i demandée.
$usage\_mode_i$: mode d'utilisation des nœuds du type i (MULTI_PROG : si l'application autorise le dédoublement de ses processus sur le même nœud, MONO_PROG : sinon).
nb_req_types : nombre de types demandés dans la requête.
S_i : liste d'identificateurs de nœuds de type i à allouer.
S_i_length : nombre de nœuds de type i à allouer (a priori).
ws_types : nombre total de types (architectures) de stations de travail.
avail_workstations(i) : ensemble de stations de type i disponibles actuellement.
nb : nombre de nœuds à trouver durant l'étape de réajustement.
appli : application en question.
Procs : fonction retournant l'ensemble des processus d'un nœud ou d'une application.
Appli : fonction retournant l'application propriétaire d'un processus.
SA : liste de nœuds (éventuellement de type i) alloués à l'application.
SPD : ensemble de processus à désallouer.
A : liste de tous les nœuds de même classe que la requête de l'application (stations de travail ou ressources spécifiques) et éventuellement de type i.
P : processus d'une application.

R_k : rang d'une application.

Evaluation des performances :

t_i^a : temps d'arrivée de l'application i.
t_i^d : temps de départ de la même application.
$NbSlices_i$: nombre total de *"slices"* temporels de l'application i. Chaque slice est délimité par la création d'un nouveau processus ou la terminaison d'un processus.
$SliceLength_{ij}$: durée temporelle du slice j de l'application i.
$NbProcs_{ij}$: nombre de processus de l'application i durant le slice j.
RT_n : temps de réponse de l'application n (temps d'exécution parallèle).
ST_n : temps d'exécution séquentielle de l'application n.
APD_n : degré de parallélisme moyen de l'application n.
$PDVR_n$: taux de variation du degré de parallélisme de l'application n.
$NbApplis_s$: nombre d'applications générées durant la session s.
MRT_s : temps de réponse moyen de la session s.
λ : taux d'arrivées des applications.

C.3 Symboles utilisés dans le chapitre 7

Applications issues de l'optimisation combinatoire :

X : espace de recherche.
f : fonction objectif permettant d'évaluer le coût d'une solution.
$N(S)$: voisinage de la solution $S \in X$.
O : ensemble de n objets à positionner.
L : ensemble de n positions ou localisations.
C : matrice de flux entre les objets.
D : matrice de distances entre les positions.
M : bijection entre les objets et les positions.

Recherche tabou :

S_0 : solution initiale.
T_i : taille de la liste tabou.
max_iter : nombre maximum d'itérations sans amélioration de la fonction coût.
S : solution courante.
nb_iter : itération courante.
best_sol : meilleure solution trouvée.

best_iter : numéro d'itération de la meilleure solution.

Recuit simulé pour le partitionnement de graphes :

$G = (V, E)$: graphe à partitionner (non-orienté).
V : ensemble de N sommets.
E : ensemble des arêtes.
A et B : les deux partitions recherchées.
f : fonction coût.
T_{max} : température initiale.
T_{min} : température finale.
N_a : nombre maximum de configurations acceptées à chaque palier de température.
T : température courante.
C : configuration courante.
E : coût de la configuration courante.
E_0 : coût de la configuration initiale.

Résolution de systèmes d'équations linéaires et élimination de Gauss :

A : matrice $n \times n$ non singulière, à trianguläriser.
$\vec{b}$: vecteur représentant le membre droit de l'équation.
$\vec{x}$: vecteur de solutions du système.
k : numéro de l'étape courante.

Inversion de matrices par la méthode de Gauss-Jordan :

A : matrice découpée en $q \times q$ blocs de taille fixe b.
B : matrice inverse.
k : numéro de l'étape courante.

Classification :

$X = \{x_1, x_2, ..., x_n\}$: ensemble de données à classifier.
c : nombre de classes.
x_i : vecteur de p mesures réelles décrivant les caractéristiques d'un élément de X.
T : nombre maximum d'itérations.
m : degré d'incertitude dans la définition des classes.
$E_t = \|U_t - U_{t-1}\|_{err}$: taux de changement de la matrice U entre les itérations $t-1$ et t.
ϵ : seuil de terminaison sur E_t.

U : matrice d'appartenance aux classes $(c \times n)$.
V : c-tuple représentant les centres des classes.
$\|.\|$: norme utilisée, $\|x\|_A^2 = x^T A^{-1} x$.
A : matrice $(n \times n)$ définie positive exprimant la métrique utilisée.
n_d : taille des données étiquetées.
n_u : taille des données non étiquetées.
$(\{x_k^d\}, 1 \leq k \leq n_d)$: ensemble des données étiquetées.
$(\{x_k^u\}, 1 \leq k \leq n_u)$: ensemble des données non étiquetées.
s : nombre de partitions.

Résultats expérimentaux :

$R(A)$: temps de réponse de l'application (temps d'exécution parallèle en secondes).
$T(A)$: temps de service (équivalent au temps séquentiel en secondes).
$Sched$: surcoût de l'ordonnancement au niveau de l'application (en secondes).
N_{base} : nombre (normalisé) de nœuds utilisés par l'application.
$S(A)$: accélération résultante.
$U(A)$: taux d'utilisation du système par l'application.
UFd : nombre de nœuds crédités à l'application (y compris ceux de la première allocation).
Fd : nombre de nœuds débités.
Ret : nombre de retraits de tâches par l'ordonnanceur.
Fl : nombre de défaillances des esclaves.
APD : degré de parallélisme moyen pondéré par le temps.
$Eff(A)$: efficacité résultante.