Ordonnancement hybride pour MPI sur les clusters

Hinde Lilia Bouziane

Stage éffectué au LRI, Université Paris-Sud, Orsay
sous la responsabilité de M Franck Cappello
Co-encadrement: Pierre Lemarinier, Aurélien Bouteiller et Thomas Hérault

3 septembre 2004

Résumé

L'objectif de ce stage est d'étudier une nouvelle approche d'ordonnancement qui soit efficace pour un genre d'applications très utilisées pour les calculs hautes performances dans un environnement parallèle. Le but est d'obtenir un partage équitable des ressources entre utilisateurs tout en accélérant l'exécution de leurs applications et en utilisant au maximum et au mieux les ressources partagées.

Table des matières

Introduction **4**

1 Définitions préliminaires **5**

2 Techniques d'ordonnancement de base **7**
2.1 Le Co-Scheduling 7
2.2 Le Gang-Scheduling 8

3 Travaux relatifs **9**
3.1 Structures existantes du Gang-Scheduling 9
3.2 Meilleure utilisation des ressources avec le Gang-Scheduling 12
3.3 Améliorations de la gestion mémoire 15

4 Étude d'une nouvelle approche d'ordonnancement **18**
4.1 Introduction de l'ordonnancement hybride 18
4.2 Architecture de travail 19
4.3 Ordonnancement Hybride par quantum 21

5 Évaluation des performances **24**
5.1 Protocole expérimental : 24
5.2 Analyse des résultats : 25

Conclusions **30**

Introduction

Les applications scientifiques parallèles à passage de messages pour le calcul hautes performances peuvent s'exécuter sur des machines reliées par réseau très rapide (clusters) ou distribuées sur des sites éloignés de domaines différents (Grilles). Lorsque plusieurs utilisateurs exécutent de telles applications sur des ressources communes, il est impératif d'offrir une utilisation équitable de ces ressources, tout en minimisant l'attente de terminaison de chaque application (temps de réponse). Il faut pour cela traiter les applications équitablement, en minimisant le ralentissement de l'exécution d'une application par rapport à sa durée d'exécution seule, et en exploiter au maximum toutes les ressources. Pour améliorer la qualité de ces services, des stratégies de partage de ressources et d'ordonnancement d'applications sont indispensables.

Généralement, la soumission de plusieurs programmes utilisateurs à exécuter sur des ressources communes se fait de deux manières différentes. La première méthode est une méthode séquentielle où le premier arrivé est le premier servi. La deuxième se fait en espace partagé (partitionnement) où des applications peuvent être exécutées en concurrence sur des machines différentes, tel que chaque application occupe seule un sous ensemble des ressources. Une conséquence de ce type de politique est qu'une application de longue durée et prenant l'ensemble des ressources disponibles bloque l'exécution de toutes les applications, potentiellement de courte durée, soumises ultérieurement. Plusieurs techniques d'ordonnancement ont été implémentées dans des environnements parallèles pour pallier à cet inconvénient.

Le but de se stage est de proposer une nouvelle approche hybride d'ordonnancement d'applications pour des environnements parallèles à partir de différentes approches déjà existantes et largement utilisées. Il s'agit d'améliorer la qualité de service visant à satisfaire équitablement tous les utilisateurs et d'utiliser au mieux les ressources partagées.

Dans un premier temps nous introduisons quelques définitions pour des termes utilisés souvent par la suite. Nous décrivons tout d'abord deux techniques d'ordonnancement à partir desquelles nous nous somme basés pour l'étude d'une nouvelle approche appelée ordonnancement hybride.

Nous exposons ensuite quelques travaux de recherches dans le domaine de l'or-

donnancement. En premier, une description de l'une des techniques d'ordonnancement les plus utilisées dans les environnements parallèles. Ensuite, des idées proposées pour améliorer des algorithmes d'ordonnancement et la gestion de la mémoire qui joue un rôle très important dans l'évaluation des performances. La gestion mémoire étant un facteur important sur lequel nous nous somme basés dans l'étude de l'ordonnancement hybride.

Une description de l'architecture sur laquelle j'ai travaillé pour implémenter l'ordonnancement hybride est détaillée par la suite. Il s'agit d'éléments proposés par l'équipe MPICH-V dans le cadre du projet PCRI Grand-Large afin de développer plusieurs protocoles de tolérance aux pannes pour les applications parallèles à passage de messages MPI. Nous exposons ainsi les principes de l'ordonnancement hybride implémenté à partir de cette architecture avant de présenter les résultats de performance obtenus pour tester et comparer les différentes approches d'ordonnancement étudiées.

Chapitre 1

Définitions préliminaires

Ce chapitre défini quelques termes et principes de base qui seront très utilisés par la suite.

1. **Processeur :** un processeur fait référence à l'ensemble contenant l'unité centrale de calcul (CPU), les registres et la mémoire physique (mémoire vive).

2. **Cluster :** un cluster est un ensemble de stations de travail (ou processeurs) reliées par un réseaux très rapide. Ethernet et Myrinet constituent des exemples de tels réseaux .

3. **Applications parallèles à passage de message MPI (Message Passing Interface) :** il s'agit d'applications qui utilisent un modèle de programmation très utilisé pour le calcul haute performance. Une application MPI s'exécute à travers plusieurs processus sur différents processeurs et qui communiquent entre eux à l'aide d'un réseaux rapide qui relie les processeurs. Dans la suite nous allons nous intéressé à ce genre d'applications sur cluster (fig.1.1).

4. **Partage en temps des ressources (Time-sharing) :** politique de gestion de l'exécution de plusieurs applications dans le temps sur un même ensemble de ressources (CPUs, ressources d'Entrées/Sorties (E/S)...). Le temps est partitionné en plusieurs intervalles que l'on appelle quantum. Durant chaque quantum, une seule application s'exécute sur l'ensemble des ressources qui lui sont nécessaires. Le choix d'une application à exécuter durant un quantum dépend de la politique d'ordonnancement utilisée. La gestion de l'exécution en temps partagé se fait par un ordonnanceur ou Scheduler.

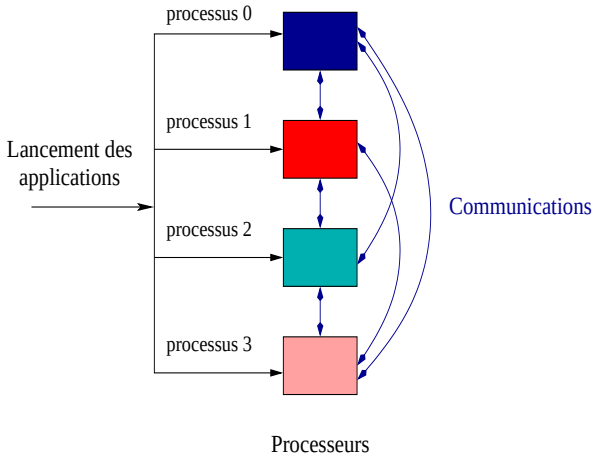


FIG. 1.1 – Exécution d'applications MPI sur 4 processeurs

5. **Mémoire virtuelle et pagination :** la mémoire vive ou physique d'un processeur peut être partitionnée en plusieurs zones que l'on appelle pages. L'espace mémoire d'un programme lors de son exécution peut ne pas être complètement chargé dans la mémoire physique mais par zones, chaque accès à une adresse mémoire dans une zone qui n'est pas en mémoire physique provoque un **défaut de page** et le chargement de cette zone depuis le disque. Plusieurs pages ou zones de différents programmes peuvent être en même temps dans la mémoire physique. Si plusieurs programmes à occupation mémoire importante (comme pour les applications MPI) s'exécutent en temps partagé sur un même processeur, ils partagent aussi la mémoire physique, deux situations sont possibles :
   (a) **In-core ou pages tenant en mémoire :** où l'ensemble des pages de tous les programmes peuvent tenir en mémoire physique. Dans ce cas chaque défaut de page pour un programme provoque le chargement d'une zone de ce programme pour pouvoir continuer l'exécution.

   (b) **Out-of-core ou pages ne tenant pas en mémoire :** où l'ensemble des pages de tous les programmes ne peuvent pas tenir en mémoire physique (surcharge). On a alors des pages dans la mémoire physique et d'autres sur le disque. Dans ce cas un défaut de page pour un programme peut provoquer la sauvegarde de d'une page mémoire du même programme ou d'un autre programme avant de charger la zone demandé. Le choix de la page à remplacer dépend de la politique de remplacement du système d'exploitation, par exemple la politique LRU (Least Recently Used) qui consiste a remplacer la page la plus anciennement référencée (accédée) par le programme.

Chapitre 2

Techniques d'ordonnancement de base

Plusieurs travaux de recherche proposent des techniques d'ordonnancement et de partage de ressources. Voici les deux principales méthodes à partir desquelles nous avons développé une nouvelle stratégie hybride originale.

2.1 Le Co-Scheduling

les systèmes d'exploitation actuels sont capables d'exécuter en même temps des programmes exigeant des tailles mémoire importantes. Cela est possible grâce au système de pagination décrit dans le chapitre précédent. Ces programmes peuvent s'exécuter en temps partagé géré par un ordonnanceur du système d'exploitation. Le Co-Scheduling [1] consiste à lancer en même temps plusieurs applications parallèles sur le même ensemble de processeurs. L'ordonnanceur du système local à chaque processeur gère le partage en temps des ressources (CPU, ressource d'Entrées/Sorties...) utilisées par les différents processus assignés à ce processeur. Cette approche ne demande pas de changements spécifiques dans une architecture ou un système d'exploitation. Elle permet le recouvrement des E/S ou communications d'un processus par le calcul d'un autre, réduisant l'inactivité du CPU. Cependant, l'indépendance des ordonnanceurs peut conduire à des situations où les processus d'une même application ne sont pas ordonnancés au même moment. Cela peut introduire des délais d'attente dans le cas de communications synchrones entre processus. Un autre inconvénient du Co-Scheduling est le nombre élevé de commutations de contextes. Si la taille de la mémoire physique est largement plus petite que celle nécessaire pour l'ensemble des processus, un nombre important de défauts de pages est engendré, ce qui introduit des surcouts importants affectant l'équité, le temps de réponse et le temps total d'exécution des applications.

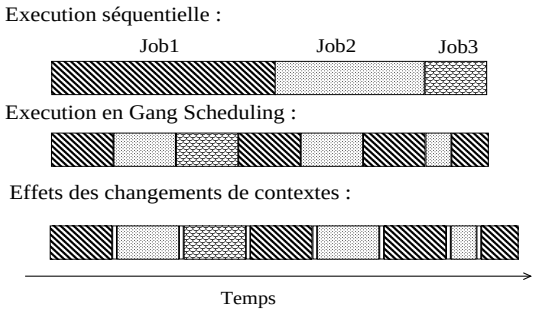


FIG. 2.1 – Ordonnancement équitable et amélioration du temps de réponse avec le Gang-Scheduling

2.2 Le Gang-Scheduling

un gang est l'ensemble des processus d'une même application. Pour implémenter le Gang-Scheduling [2, 3, 4, 5, 6], un ordonnanceur global est ajouté à l'architecture. Il partitionne le temps en intervalle (quantum) et gère l'ordonnancement des gangs suivant une politique qui lui est propre. Durant un quantum, seuls les processus d'une même application s'exécutent sur les processeurs auxquels ils sont assignés. A la fin de ce quantum, l'ordonnanceur global arrête momentanément ce gang et (re)lance le gang de la prochaine application. Cette phase de réordonnancement introduit des délais dans le temps total d'exécution. Cependant la valeur d'un quantum est définie pour être largement plus grande que la durée d'un changement d'application entre deux quantum (Fig.2.1). Cela n'implique pas de délais importants sur le temps de réponse et n'affecte pas l'équité entre applications. L'avantage du Gang-Scheduling est : 1) le traitement équitable des applications, 2) pas de retardement de l'exécution d'une application à cause d'une autre beaucoup plus lente (Fig.2.1) et 3) pour se synchroniser, un processus n'attend pas que les autres soient ordonnancés, puisque seul ces processus occupent les ressources. Cependant, un CPU reste inutilisé durant les communications ou les entrées/sorties d'un processus, alors que ce temps pourrait être utilisé pour le calcul d'une autre application, et obtenir une meilleure exploitation du processeur.

Chapitre 3

Travaux relatifs

Plusieurs structures et algorithmes d'ordonnancement ont été proposés pour les environnement parallèles. L'accent est mis sur le surcout de la commutation de contexte quand on passe d'une application à l'autre et l'utilisation au mieux des ressources.

Dans ce chapitre nous décrivons une structure d'ordonnancement très utilisée dans les environnements parallèles. Nous présentons aussi un travail améliorant l'utilisation des ressources à partir d'une structure de Gang-Scheduling et pour finir, un autre travail sur la gestion de la mémoire virtuelle lors des commutations de contexte.

3.1 Structures existantes du Gang-Scheduling

La première implémentation du Gang-Scheduler sur cluster a été réalisée en 1996 [2, 3], appelée Score-D (Score Daemon), plus connue sur le nom de Score. Il s'agit d'un processus parallèle (ensemble de processus d'un même programme parallèle) implémenté sur UNIX, et dont la structure est décrite dans la figure 3.1.

Un des processus démons de Score est un serveur dédié et le point d'entrée des programmes utilisateurs. Un processus appelé "Front End Process" (FEP) s'exécute sur le poste utilisateur. Il établie, quand un programme utilisateur est lancé, une connexion avec le serveur Score, ensuite il lui envoie des informations qui permettront aux autres processus Score de lancer les processus fils (à l'aide d'instruction **fork** ou **exec** sous UNIX) du processus utilisateur (application). On obtient ainsi, un processus parallèle. Une fois lancé, chaque démon Score se connecte au FEP afin de lui envoyer les données en sortie du processus fils qu'il a créé.

Sur des réseaux rapides tel que Myrinet, utilisé dans ce travail de recherche et d'autres cartes spécialisées, lorsqu'une application 'A' est arrêtée pour passer à une autre 'B' (phase de commutation), il est nécessaire de sauver toutes les informations

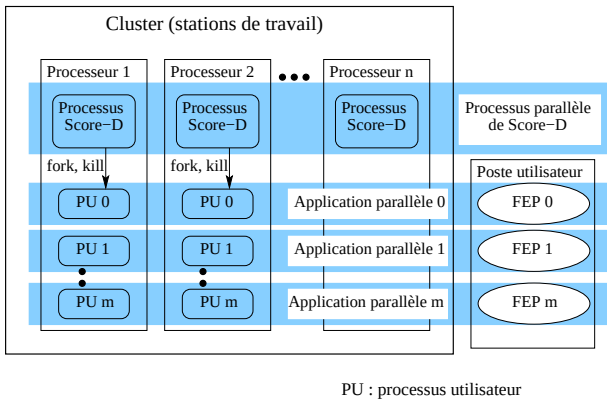


FIG. 3.1 – Structure de Score-D

garantissant une reprise correcte de 'A' à son prochain ré-ordonnancement. L'envoi d'un signal d'arrêt (SIGSTOP sous Linux) à tous les processus de 'A' ne suffit pas. Si deux processus communiquent, que le premier a envoyé un message au deuxième, et que ce dernier ne l'a pas encore reçu quand le signal l'arrête, non seulement le message est perdu pour 'A', mais un processus de 'B' peut le recevoir, impliquant une erreur. On doit ainsi sauver les messages en transit à l'arrêt des processus avant de relancer 'B'. Cette procédure est dite sauvegarde du contexte réseau.

Avant de présenter comment Score ordonnance les applications, il est nécessaire de comprendre le protocole de communication utilisé dans cette recherche sur un réseau Myrinet. Lorsqu'un message est envoyé sur un canal, si la réception s'effectue avec succès, un acquittement Ack est envoyé à l'émetteur. Sinon, dans le cas où le buffer de réception est rempli (échec), c'est un Nack qui est renvoyé pour que l'émetteur réémette le message. Ce contrôle est fait par une librairie de communication appelée PM [7]. Le protocole de communication de PM diffère de celui utilisé dans TCP/IP sur ethernet où les messages en transit sont conservés par le noyau, et dans le cas où un message serait perdu des demandes de réémission auront lieu par la suite de manière transparente au programme utilisateur. PM a été conçue pour les clusters Myrinet et les cartes réseaux rapides afin d'améliorer les latences et les débits par rapport à TCP/IP qui est coûteux en performances, et surtout dans notre cas, pour coopérer avec Score.

La figure 3.2 montre les étapes de la phase de ré-ordonnancement de Score appelée 'Network Preemption Procedure' :

1. Le serveur Score envoie un message FREEZE à tous ses processus pour qu'ils arrêtent les processus utilisateurs en cours d'exécution.

2. Chaque processus Score après réception de FREEZE, envoie un signal d'arrêt SIGSTOP au processus utilisateur qui lui est associé. Il reçoit un signal SIGCHILD confirmant cet arrêt.

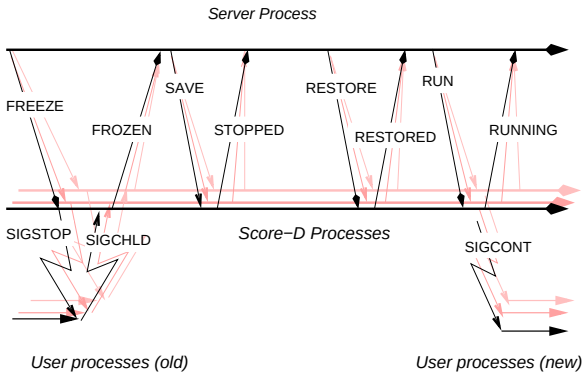


FIG. 3.2 – Étapes de la procédure "Network Preemption"

3. Après cette réception, le processus Score attend qu'il n'y ait plus de messages envoyés par son processus utilisateur sur le réseau : à la réception de Ack ou Nack pour un message envoyé, il est sûr que ce dernier n'est plus en transit. Le processus Score attend alors, la réception des Ack ou Nack pour tous les messages utilisateur émis, puis envoie un message FROZEN au serveur pour lui indiquer la fin de cette étape.
4. Lorsque le serveur Score reçoit tous les messages FROZEN de tous ses processus, et dans ce cas il n'y a plus de messages sur le réseau, il leurs envoie un message SAVE pour sauver le contexte réseau.
5. Chaque processus Score, à la réception de ce dernier, sauvegarde le contexte du canal du processus utilisateur (le contenu des buffers du canal), puis acquitte le serveur avec le message STOPPED.
6. Le serveur, à la réception de tous les acquittements (contexte réseau sauvegardé) envoie un message 'RESTORE' à tous ses processus afin de restaurer le contexte réseau des processus de la prochaine application à relancer.
7. Chaque processus Score restaure le contexte du canal du processus utilisateur qui lui est associé puis acquitte le serveur.
8. Le serveur, lorsqu'il reçoit tous les acquittement, envoie un message RUN pour que tous les processus Score relancent les processus utilisateur.
9. A la réception de ce message, chaque processus Score envoie un signal SIGCONT qui relance le processus utilisateur spécifié, et envoie un acquittement RUNNING au serveur.
10. Le serveur reçoit tous les messages RUNNING. Il sait donc que l'application est relancée et attend un quantum spécifié avant de passer à la prochaine étape d'ordonnancement.

Une approximation de deux tiers du temps de la phase de commutation est passé dans la sauvegarde et la restauration des contextes réseaux. Cela est due principalement aux copies mémoires des buffers. Ainsi, une application à communications intensive et/ou avec de longs messages a un surcout plus important. Dans une telle situation, plus le quantum est grand moins le surcout est élevé. Il a été estimé qu'en doublant le quantum, le surcout est divisé par deux. Les surcouts des commutations de contexte dans cette structure d'ordonnancement sont jugés élevés mais acceptables.

Les résultats des expérimentations ont montré aussi, que le fait d'augmenter le nombre de processeurs ralenti la phase de commutation. Cela est due à la synchronisation des processus durant les étapes 'FREEZE', 'SAVE' et 'RESTORE'. Plus le nombre de processeurs est élevés, plus ils prennent du temps à se synchroniser. Cela représente un premier inconvénient de Score.

Un autre inconvénient de Score est la limitation du nombre de canaux par la librairie PM et la complexité de la gestion du partage de ces canaux : Si plusieurs processus partagent le même canal, chaque passage d'un processus à l'autre provoque la sauvegarde et la restauration de l'unique contexte de ce canal et plus les commutations de contextes sont fréquentes plus les surcouts sont importants. Une amélioration de la structure des canaux a été étudiée [8] et implémentée pour éviter partiellement ce problème, en associant plusieurs contextes par canal. Cela a permis d'améliorer le temps de commutation de contexte dans Score.

Cette architecture a été adaptée pour les différents clusters existants tel que Ethernet et d'autres recherches ont proposé des techniques implémentables sur Score pour réduire les surcouts du ré-ordonnancement, comme nous allons le voir dans les sections suivantes.

3.2 Meilleure utilisation des ressources avec le Gang-Scheduling

Les Gang-Schedulers traditionnels gèrent l'ordonnancement des applications en Round-Robin : dans un tableau ou une liste circulaire à sens unique avec des éléments représentant ces applications, et à la fin de chaque quantum une application est suspendue pour ré-ordonnancer la seule application qui suit la première dans le tableau ou la liste. Cette façon de faire ne prend pas en compte les priorités ni les caractéristiques de chaque application et assigne en même temps tous les processus d'une même application sur autant de processeurs requis. Cela permet d'éviter le blocage des processus en attendant la fin de la communication des autres processus : les processus attendus s'exécutent et progressent, et il n'y a pas d'autres processus à exécuter sur un processeur lorsqu'un processus attend. Cependant, si une application à communications ou E/S intensives est ordonnancée durant un

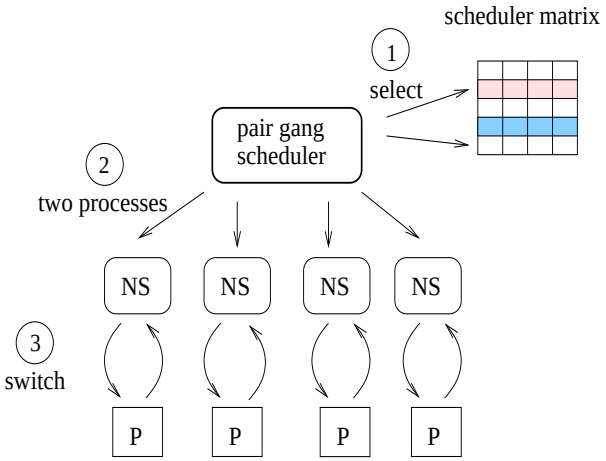


FIG. 3.3 – Architecture pour l'ordonnancement par paires. NS : Ordonnanceur local à un noeud (Node Scheduler) ; P : Processeur

quantum, des processeurs peuvent rester inutilisés durant les communications et les E/S. Cela implique une mauvaise utilisation des ressources.

Une alternative pour le Gang-Scheduling est d'utiliser l'ordonnancement locale (du système d'exploitation) indépendamment sur chaque processeur. L'ordonnanceur local utilise soit un Round-Robin, soit un algorithme de priorité pour ordonnancer des tâches ou processus. Dans le cas où plusieurs processus sont assignés à chaque processeur, un processus 'A' peut bloquer quand il attend un autre processus 'B' pour communiquer (communication bloquante) : 'B' n'est pas en cours d'exécution ou bien, il y a un autre processus que 'A' à exécuter sur le processeur et donc, le système d'exploitation ordonnance cet autre processus et met en attente 'A'. Cela peut introduire des retardements réduisant les performances du système.

Plusieurs approches ont été proposées pour trouver un compromis entre les situations précédentes. Une de ces approches [6], présente l'idée d'ordonnancer les applications par paires, une avec E/S intensive, l'autre avec calcul intensif. Ce sont deux types d'application qui utilisent des ressources différentes et qui produisent ensemble un minimum d'interférence. Un Gang-Scheduler centralisé similaire à celui de Score a été utilisé pour implémenter le "Paired Gang-Scheduler", comme indiqué dans la figure 3.3. Afin de classifier une application comme étant à E/S ou calcul intensif, le temps d'utilisation du CPU est mesuré à la fin de chaque quantum (moyenne de l'utilisation du CPU de l'ensemble des processus d'une même application), le temps restant correspond au temps des E/S. A la fin d'un quantum, un algorithme de prédiction est appliqué pour déduire la prochaine utilisation du CPU d'un job en fonction de ses précédentes mesures. Cela permet au Gang-Scheduler de coupler les applications à relancer en même temps.

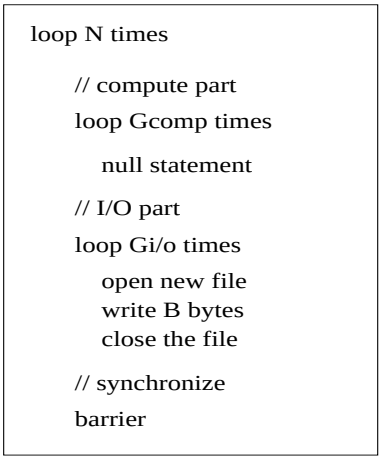


FIG. 3.4 – Modèle des programmes testés pour l'ordonnancement par paires

Une simulation pour évaluer les performances du 'Paired Gang-Scheduling' a été réalisée sur des programmes générés à partir du modèle illustré dans la figure 3.4. N étant le nombre d'itération, $Gi/o = 0$ défini un programme à calcul intensif, $Gcomp = 0$ un programme à E/S intensive, B le nombre de bits à écrire dans un fichier donnée. La barrière de synchronisation (tout le monde attend tout le monde pour pouvoir continuer) permet d'avoir une phase de communications. Ces paramètres sont ajustés selon le but d'une expérience. Un exemple particulier d'ajustement effectué après avoir calculé le temps nécessaire pour la barrière de synchronisation : $Gi/o = 0$, $Gcomp = 25000$ permettant d'avoir une utilisation de 45% du CPU, le reste pour la synchronisation, cela pour tous les jobs à lancer. Deux applications ou plus peuvent être ré-ordonnancer en même temps tant que l'utilisation du CPU prédite pour le prochain quantum n'excède pas une certaine limite inférieure à 100%. Les résultats ont montré que le 'Paired Gang-Scheduling' améliore le temps total d'exécution d'au moins la moitié par rapport au Gang-Scheduling traditionnel et avec un surcout non élevé d'interférence. Or, le fait de ne pas limiter le nombre de jobs à relancer simultanément n'est pas réaliste, car les applications réelles peuvent surcharger la mémoire physique ralentissant considérablement l'exécution de l'ensemble des applications.

D'autres travaux de recherches [9, 10] ont proposé des solutions permettant d'associer des priorités aux processus par l'ordonnanceur local de chaque processeur en fonction de plusieurs données collectées lors de leur exécutions tel que le nombre d'E/S, de messages envoyés ou reçus, de barrières de synchronisation ou encore de la durée de ces événements.

3.3 Améliorations de la gestion mémoire

Un des plus récents travaux de recherche [11] s'est plutôt concentré sur le développement de différentes stratégies d'allocation et de remplacement de pages mémoire. Le but est de réduire le surcout dû aux interférences de plusieurs applications assignées sur un même ensemble de processeur et qui ont une occupation mémoire importante.

Les systèmes d'exploitation actuels utilisent le plus souvent la politique du LRU (Least Recently Used) ou une de ses approximations : lorsqu'une page est demandée et qu'il n'y a pas assez d'espace mémoire disponible, la ou les pages les plus anciennement référencées sont éjectées vers une mémoire secondaire (disque) et remplacées par celle demandée. Cette politique est jugée non adaptée à un environnement parallèle pour les raisons suivantes :

(a) Les applications parallèles demandent des quantum assez grand (de l'ordre de la minute). Une tache à la fin de son quantum, aura des pages récemment référencées qui peuvent occuper longtemps et inutilement de l'espace mémoire.

(b) Lorsque cette même tache est ré-ordonnancée, il se peut qu'une de ses pages, par exemple 'P1', soit encore en mémoire. Or, si une autre page 'P2' est demandée avant que 'P1' ne soit référencée, 'P1' sera éjectée pour être rechargée par la suite.

Dans le Gang-Scheduling, l'ordonnancement est contrôlé par un niveau utilisateur appelé « external user-level scheduler ». Il est possible alors, de savoir quelles pages vont être à court terme utilisées et celles qui ne seront pas aussitôt référencées. Le but des stratégies développées dans cet article, étant d'utiliser cette possibilité pour charger ou éjecter des pages de la mémoire physique en bloc, de minimiser les défauts de page et donc les E/S quand une application parallèle est ordonnancée. Les stratégies proposées sont les suivantes :

1. **Selective Paging Out :** c'est un algorithme de prévention qui fait en sorte de ne pas éjecter, lorsqu'une application est ordonnancée, des pages aussitôt rechargées (situation (b)). Une page mémoire d'un processus est associée à l'identifiant de ce processus. Dans une phase de commutation de contexte, la plus ancienne page chargée en mémoire du processus qui vient juste d'être mis en suspend est sélectionnée pour être remplacée par une page demandée du processus ré-ordonnancé. Cette procédure étant récursive pour un même identifiant de processus, jusqu'à libérer assez d'espace mémoire pour la page demandée. Si la libération de l'espace mémoire occupé par le processus sortant n'est pas suffisante, la procédure s'applique à un autre processus mis en suspend dans la phase de ré-ordonnancement précédente.

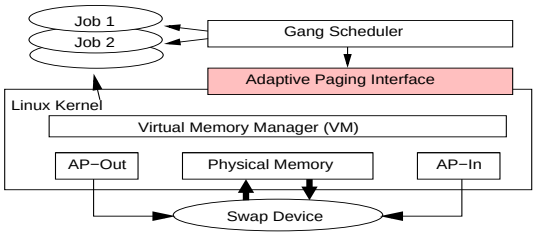


FIG. 3.5 – Architecture pour tester les fonctions de pagination implémentées

2. **Aggressive Paging Out :** le but de cette stratégie est de minimiser le nombre d'E/S effectuée dans une phase de ré-ordonnancement. Plus précisément, lorsqu'il y a un défaut de page pour une application ré-ordonnacée, le noyau éjecte récursivement une page vers le disque jusqu'à obtenir assez d'espace pour celle demandée. Cet algorithme utilise la taille de l'espace de travail (espace total occupé par un processus) du processus entrant ou ré-ordonnancé. Cette taille est calculée à l'aide des accès mémoire d'un processus durant son exécution. Elle est utilisée alors, pour estimer l'espace mémoire à libérer lorsqu'un processus est ré-ordonnancée. Cela dans le but d'effectuer efficacement les E/S en bloc de pages, et non par page, et de minimiser le temps passé dans le swap (remplacements). Cette technique est combinée avec la première pour assurer l'éjection de page appartenant à des processus sortants.

3. **Adaptive Paging In :** cet algorithme consiste à enregistrer, à chaque commutation de contexte, l'identifiant et l'adresse des pages éjectées d'un processus. Lorsque ce dernier est ré-ordonnancé, un défaut de page provoque le chargement de toutes les pages enregistrées. Cela permet de reconstituer le contexte mémoire de la dernière exécution, et de réduire la latence due aux défauts de page.

4. **Background Writing of Dirty Pages :** Il s'agit d'essayer de réduire la durée de la commutation de contexte. L'idée est de mettre à jour les pages modifiées d'un processus en exécution dans la mémoire secondaire. Cela permet, lorsqu'une tache est ré-ordonnancée, de charger des pages en remplaçant d'autres sans avoir à effectuer des écriture vers la mémoire secondaire. Afin d'éviter l'écriture répétée de la même page, la mise à jours en mémoire secondaire est effectuée par une tâche de fond lancée peu avant la phase de commutation de contexte et arrêtée au début de cette dernière.

Les fonctions implémentant ces mécanismes constituent une API introduite dans le noyau UNIX. Elles sont utilisées dans l'architecture décrite dans la fi-

gure 3.5, et peuvent être combinées. A chaque commutation de contexte, le Gang-Scheduler envoi un signal SIGSTOP pour arrêter une application, et SIGCONT pour relancer une autre.

Des expérimentations ont été réalisées sur des applications MPI et ont montré que ces mécanismes réduisent considérablement le temps de commutation de contexte (jusqu'à 90% pour certaines applications). Ce qui permet d'avoir des quantum plus courts et un meilleur temps de réponse pour les différentes applications. La combinaison d'un algorithme avec un autre, permet de pallier les inconvénients du premier, par exemple : 'Aggressive Paging Out' peut causer plusieurs écritures en mémoire secondaire durant la commutation de contexte. 'Background Writing of Dirty Pages' permet de réduire le nombre de ces écritures. Cette dernière technique activée dans les derniers 10% du quantum, améliore de 15% le temps de swap.

Chapitre 4

Étude d'une nouvelle approche d'ordonnancement

4.1 Introduction de l'ordonnancement hybride

Dans un travail de recherche récent [1], il a été prouvé que le Co-Scheduling est plus performant que le Gang-Scheduling. En effet pour la majorité des applications, les communications ne sont pas parfaitement recouvertes par le calcul. Ce résultat est vrai dans le cas où la taille mémoire occupée par les processus de toutes les applications sur un même processeur ne dépasse pas la taille de la mémoire physique (cadre in-core). Dans ce cas, on permet au système d'exploitation de traiter équitablement les processus. Cela est due au fait que l'on évite les multiples défauts de pages causés par la surcharge mémoire (cadre out-of-core), qui dégradent de façon exponentielle les performances en temps d'exécution.

Pour tirer avantage de ce résultat et du Gang-Scheduling, nous avons proposé une nouvelle approche hybride. Il s'agit, en fonction de la taille mémoire nécessaire par application, de 'Co-Scheduler' pour un quantum un nombre d'application tel que la mémoire sur chaque processeur ne soit pas surchargée (cadre in-core). A la fin de chaque quantum ces applications sont arrêtées pour (ré)ordonnancer de la même façon les prochaines applications (fig.4.1).

Plusieurs solutions sont possibles pour implémenter cette nouvelle approches dans un environnement parallèle. Afin d'avoir le contrôle sur l'évolution de l'exécution des applications et avoir une vue complète sur le système, il est préférable que la solution soit une solution utilisateur. Et pour garantir aussi la portabilité de notre solution, aucune modification dans le système d'exploitation (noyau) ne doit être demandée.

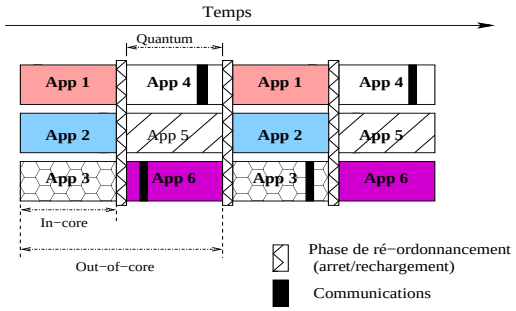
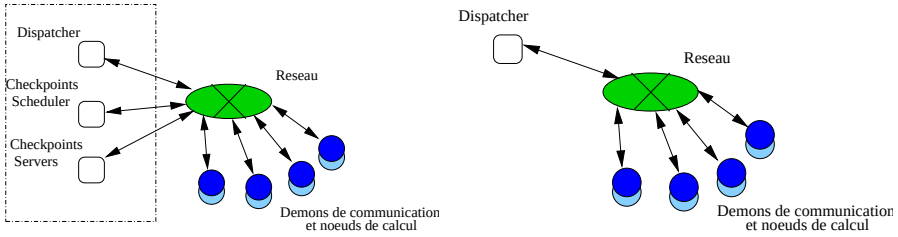


FIG. 4.1 – Exécution de 6 applications 3 par 3 avec l'ordonnancement hybride



(a) MPICH-CL (b) MPICH-DUMMY

FIG. 4.2 – Architecture de base

4.2 Architecture de travail

Dans le cadre du projet MPICH-V [12, 13, 14], le projet PCRI Grand-Large a développé plusieurs protocoles de tolérance aux pannes pour MPI (Message Passing Interface). MPI [15] est une interface de communication de haut niveau permettant d'avoir des communications performantes sur différentes architectures distribuées tel que les Clusters ou les Grilles. Il n'y a pas dans MPI de mécanismes intrinsèques de tolérances aux pannes.

Cette architecture parallèle comprend tous les éléments qui permettent d'arrêter et de reprendre correctement l'exécution des applications grâce à l'algorithme de Chandy-Lamport et au mécanisme de checkpoint que nous allons voir par la suite. J'ai donc étudié un des protocoles implémentés afin de pouvoir y introduire l'ordonnancement hybride pour les applications MPI.

Dans MPICH-V, la reprise après panne d'un processus peut être faite depuis un état précédent de l'exécution de ce processus grâce au mécanisme de prise d'image (checkpoint). Il s'agit de la sauvegarde de l'état (contenu mémoire, résultats intermédiaires...) d'un processus à un instant donné de son exécution. La sauvegarde de l'état d'un processus peut se faire sur un disque local pour une reprise sur le même processeur, ou distant si on souhaite changer de processeur.

Avant d'implémenter l'ordonnancement hybride, j'ai approfondi l'étude de l'un des protocoles tolérants aux pannes. La figure 4.2(a) décrit l'architecture qui implémente ce protocole (MPICH-Vcl). Il est basé sur la prise d'état global (où tous les noeuds sont coordonnés pour la prise d'une image) et l'algorithme de Chandy-Lamport [16]. Je me servirai de ce mécanisme, dans le cas de l'ordonnancement d'applications, pour permettre l'arrêt d'une application et la reprise correcte plus tard. L'architecture de MPICH-V se compose des éléments suivant :

1) **Le Dispatcher :** son rôle est de lancer les serveurs (checkpoint Schedulers et le serveur de checkpoint) et tous les processus MPI d'un même programme sur différents noeuds à travers des démons et de détecter les déconnexions après pannes afin de relancer les processus. Il détecte aussi la fin de l'exécution du programme.
2) **Le Checkpoint Scheduler (SC) :** il envoie les requêtes de prise d'image à tous les noeuds où résident les processus MPI de manière périodique ou à la demande de l'utilisateur. Une image de checkpoint globale est réussie si tous les processus ont acquitté la fin du checkpoint local. Le checkpoint Scheduler assure aussi le relancement de tous les noeuds à partir de la même image de checkpoint globale pour que tous les processus soient coordonnés lorsqu'ils sont relancés.
3) **Le serveur de checkpoint :** un ou plusieurs noeuds pour la sauvegarde distante des images de checkpoint. Lorsqu'un processus est relancé depuis une image de checkpoint, celle-ci est chargée depuis ce serveur.
4) **Les démons :** un démon sur un noeud de calcul gère les communications avec tous les autres noeuds. Un driver joue le rôle d'interface entre ce démon et le processus MPI. Lorsqu'un démon reçoit un premier message d'ordre de checkpoint (drapeau) depuis le checkpoint Scheduler ou un autre démon, il transmet une requête au driver qui se charge de lancer la procédure de checkpoint du processus MPI. Il diffuse au même moment un drapeau vers les autres noeuds pour leur indiquer le début de son checkpoint. Comme les canaux sont First In First Out, les messages sont reçus dans l'ordre de leurs émissions. La réception d'un drapeau indique que tous les messages utilisateurs reçus avant et non transmis au processus MPI sont des messages à sauvegarder. Ces messages ont été émis avant le checkpoint de l'émetteur et ne seront pas retransmis après une reprise de l'exécution de ce dernier. La procédure d'envoi de tous les drapeaux pour avoir une image cohérente du système (i.e à partir de laquelle la reprise sur panne produit une exécution sans perte d'informations) et qui est utilisée dans les architectures distribuées pour ne plus avoir de messages en transit est basée sur l'algorithme de Chandy-Lamport [16].

Un protocole non tolérant aux pannes a été dérivé de cette architecture en supprimant le mécanisme de checkpoint et l'algorithme de Chandy-Lamport

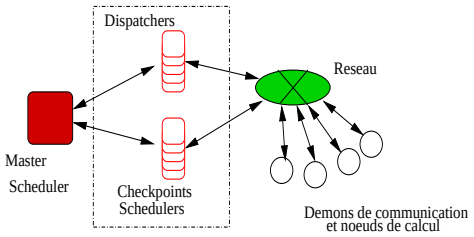


FIG. 4.3 – Architecture pour l'ordonnancement hybride

(Fig.4.2(b)). J'ai utilisé ce protocole pour le Co-Scheduling en lançant l'exécution en simultané de plusieurs applications sur le même ensemble de machines, où chaque application est reliée à son propre dispatcher. Cela permet d'avoir une comparaison équitable avec les autres approches, car seul le mécanisme de checkpoint les différencie.

4.3 Ordonnancement Hybride par quantum

La plupart des implémentations du Gang-Scheduling utilisent des signaux (SIGSTOP/SIGCONT sous UNIX) pour arrêter ou reprendre l'exécution d'un processus. Si un processus P_i est arrêté, ses pages restent en mémoire jusqu'à ce qu'un autre processus ordonnancé sur le même processeur provoque l'éjection éventuelle d'une ou plusieurs pages de P_i. Lorsque P_i est réordonnancé, ses pages sont rechargées en mémoire depuis le disque au fur et à mesure qu'elles sont reréférencées par P_i. Cela induit des surcouts considérables au cours de l'exécution.

Dans notre approche nous n'utilisons pas les signaux, mais le checkpoint global sur disque et l'algorithme de Chandy-Lamport vus dans la section précédente. Une image de checkpoint permet de libérer la mémoire occupée par un processus lors de son arrêt et recharger toute la mémoire d'un processus à son ré-ordonnancement en une seule fois. Le mécanisme de checkpoint introduit donc une gestion explicite de la mémoire sans aucune modification du noyau UNIX. Pour implémenter l'ordonnancement hybride, j'ai apporté des modifications à l'architecture décrite précédemment. J'ai supprimé les serveurs de checkpoint car les images de checkpoint sont sauvées sur les disques locaux aux processeurs. J'ai aussi apporté des modifications de fonctions pour adapter le problème à l'ordonnancement hybride et j'ai développé un ordonnanceur global appelé le Master Scheduler (MSC, Fig 4.3). Le Gang-Scheduling implémenté est dérivé de l'ordonnancement hybride. Il consiste à lancer une seule application par quantum. La programmation a été effectuée en C. Le MSC récupère en entrée le nombre total d'applications à exécuter (N), le nombre d'application à exécuter en simultané ($K \leq N$) et un même quantum pour chacune. Nous faisons en sorte que l'espace mémoire occupé par les K

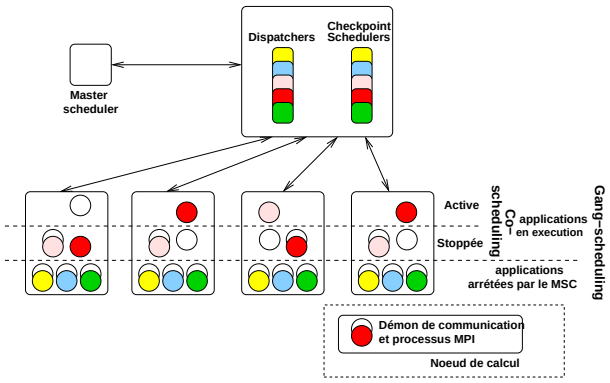


FIG. 4.4 – Ordonnancement des applications pour l'ordonnancement hybrid avec K = 2.

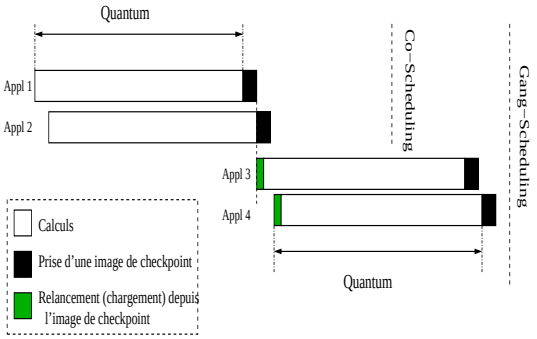


FIG. 4.5 – Exemple d'exécution avec l'ordonnancement hybride, $N = 5$, $K = 2$.

applications n'excède pas l'espace mémoire physique. Cela évite des interférences entre applications provoquant des défauts de pages. La figure 4.4 représente un exemple d'état d'exécution en ordonnancement hybride, avec $N = 5$ et $K = 2$. Pour éviter des écritures simultanées sur le disque lors d'un checkpoint par K applications, ce qui ralentit cette procédure, nous introduisons un décalage minimal entre chaque lancement des K applications sans changer pour autant le quantum d'exécution de chacune (Fig.4.5).

Avant d'ordonnancer toute application, le MSC lance tous les dispatchers et établie une connexion avec tous les checkpoint schedulers mis en place par ces dispatchers. Le MSC envoie un ordre de lancement de K applications à K dispatchers puis attend la fin du premier quantum pour procéder à l'étape d'ordonnancement qui se déroule ainsi :

1) Le MSC envoie un ordre de lancement d'une phase de checkpoint aux SCs des applications en cours d'exécution et dont le quantum est expiré.

2) Un SC à la réception de cet ordre, lance la vague de checkpoint décrite auparavant. Il envoie à tous les démons des processus de l'application associée à ce

SC une notification de fin de la prise de l'image globale.

3) Lorsqu'un démon reçoit cette notification, il arrête l'exécution du processus MPI et sa propre exécution.

4) Un SC envoie une notification au MSC pour qu'il ordonne le (re)lancement de la prochaine application.

Une phase de ré-ordonnancement s'effectue aussi à la fin de l'exécution d'une application indiquée au MSC par le dispatcher concerné. De plus chaque ré-ordonnancement suit ce protocole en 4 étapes.

Chapitre 5

Évaluation des performances

Plusieurs expérimentations ont été effectuées pour évaluer et comparer les performances des différentes approches d'ordonnancement, Co-scheduling, Gang-Scheduling et ordonnancement hybride, sur l'architecture parallèle décrite dans le chapitre précédent. Afin d'avoir une comparaison équitable des trois techniques, le Co-Scheduling et le Gang-Scheduling ont été implémentés toutes deux dans l'architecture MPICH-V. Le Gang-Scheduling est un cas particulier de l'ordonnancement hybride et est donc implémenté sur la même architecture. Les tests ont été effectués dans le même framework.

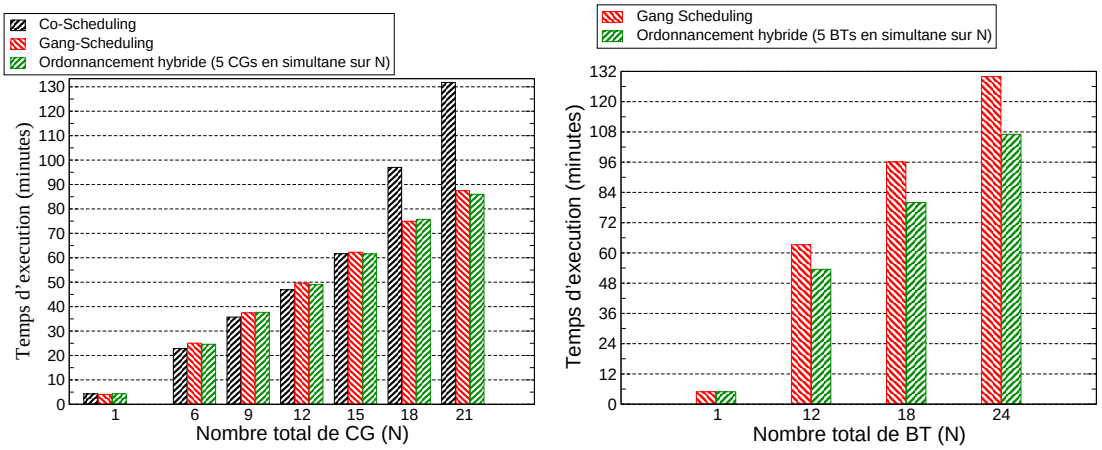
5.1 Protocole expérimental :

Afin de réaliser ces mesures de performances, j'ai utilisé des machines du cluster du LRI équipées de 1 Giga octets de mémoire principale (dont 900 Méga octets espace utilisateur), de disques IDE, de processeurs Athlon XP 2800+ à 2GHz, et Athon XP 1800+ à 1.5GHz et reliées par un réseau Ethernet à 100Mbit/sec. Le système d'exploitation utilisé étant Linux 2.4.20 et la taille du swap a été mise a 1Go pour pouvoir exécuter plusieurs applications à occupation mémoire importante en Co-Scheduling (cadre out-of-core).

Les mesures ont été effectuées sur des NAS Parallel Benchmark [17] écrits par le centre de recherche NASA NAS pour tester les calculs parallèles haute performance :

1. BT est une application effectuant beaucoup de calculs et peu de communications avec une occupation mémoire de 175 Moctets (classe C sur 25 processeurs) ou 135 Moctets (classe B sur 9 processeurs) par processus.
2. CG par contre effectue plus de communications que de calculs avec une occupation mémoire de 157 Moctets (classe C sur 8 processeurs) par processus.

Les performances de chaque approche d'ordonnancement sont évaluées par le



(a) Performances pour CG classe C.

(b) Performances pour BT classe B.

FIG. 5.1 – Comparaison du temps d'exécution de N applications avec les différentes politiques d'ordonnancement

temps de l'exécution de N applications homogènes soumises au même moment pour être exécutées.

5.2 Analyse des résultats :

Plusieurs mesures de performance ont été effectuées pour étudier et comparer le comportement des différentes approches d'ordonnancement. La figure 5.1(a) représente un exemple de ce que nous avions obtenu comme résultats sur un programme CG classe C sur 8 machines avec $K = 5$.

Nombre d'apps	Nombre moyen de défaut de pages pour tous les nœud d'une application, par minutes									Nbr défaut de page par minutes, toutes appls	écart type
	app 0	app 1	app 2	app 3	app 4	app 5	app 6	app 7	app 8		
7	***8.5***	***239.5***	***264.25***	951	1145.25	***1245.5***	1074			704	**474.9**
9	484.58	405.94	564.78	524.4	510.66	577.26	506.94	481.84	509.4	507.4	**47.1**

FIG. 5.2 – Statistiques des défauts de page pour des BT classe C sur 25 processeurs en simultané

Le temps d'exécution total pour N CG concurrents avec le Co-Scheduling évolue de manière exponentielle. Cela est due aux multiples défauts de pages engendrés lorsque N devient important. Malgré la surcharge de la mémoire avec

$N = 9$, 12 et 15, le Co-Scheduling reste un peu plus performant que le Gang-Scheduling et l'ordonnancement hybride. La raison est que CG est caractérisé par sa localité spatiale où une page en mémoire reste longtemps utilisée avant l'accès à une autre page. Cela réduit par nature du modèle d'accès à la mémoire de CG, le nombre de défauts de page. A partir de $N = 18$ les défauts de pages sont plus importants car les applications interfèrent de plus en plus. Pour une autre application (NAS BT) avec un modèle d'accès mémoire différent, l'ordonnancement hybride est plus performant que le Co-Scheduling dès que la mémoire est surchargée (fig.5.1(b)).

Nous remarquons que les performances en temps total d'exécution pour CG avec l'ordonnancement hybride sont proches de celles avec le Gang-Scheduling. Cela est due aux communications intensives de CG, ce qui fait que les recouvrements par des calculs sont faibles. Ce n'est pas le cas avec BT qui effectue moins de communications. D'après la figure 5.1(b) le temps de réponse global est meilleur avec l'ordonnancement hybride.

Les traces d'exécution ont montré que durant un quantum, les K applications effectuent le même nombre d'étapes par application. Cela montre que le système d'exploitation traite équitablement les applications. L'écart type du temps de réponse pour l'ensemble des N applications pour la figure 5.1(b) est de 12.55% pour le Gang-Scheduling comparé à 8.38% pour l'ordonnancement hybride. Ce qui fait qu'on obtient non seulement un meilleur temps d'exécution total mais aussi un traitement plus équitable.

L'évolution linéaire de la courbe pour le Gang-Scheduling ou l'approche hybride s'explique par l'absence de défauts de pages grâce au mécanisme de checkpoint par rapport à ceux qui peuvent être engendrés avec l'utilisation de signaux Unix.

Pour mieux comprendre l'effet des défauts de page sur le traitement des processus par le système d'exploitation lorsque la mémoire physique est surchargée (en Co-Scheduling), nous avons prélevé le nombre de défauts de page par minute par application (fig.5.2). L'ordonnanceur de Linux privilégie une ou plusieurs applications sur l'ensemble des applications de façon à diminuer le nombre de défauts de pages. Les applications non privilégiées génèrent plus de défauts de pages. Par conséquents les applications privilégiées se terminent bien avant les autres.

Afin de déterminer la meilleure technique d'arrêt/rechargement pour arrêter et reprendre l'exécution des applications, j'ai comparé trois techniques présentées dans la figure 5.3 :

A) **Arrêt/rechargement séquentielle :** celle utilisée dans nos expérimentations et qui consiste à attendre la fin d'un checkpoint pour relancer une autre application.

B) **Arrêt/rechargement en parallèle :** relancer une application au même moment où on ordonne la prise d'image de checkpoint.

C) **Préchargement de l'application à relancer :** relancer une application

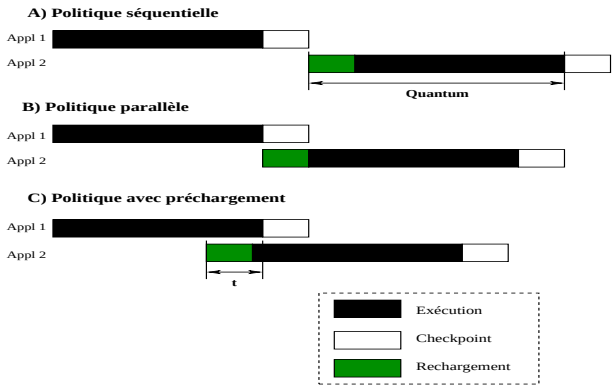


FIG. 5.3 – Politiques d'arrêt/rechargement

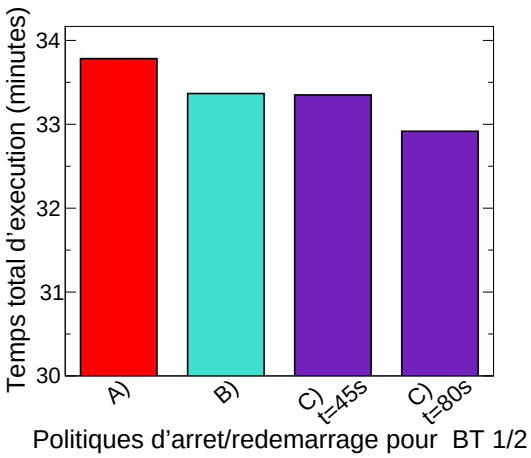
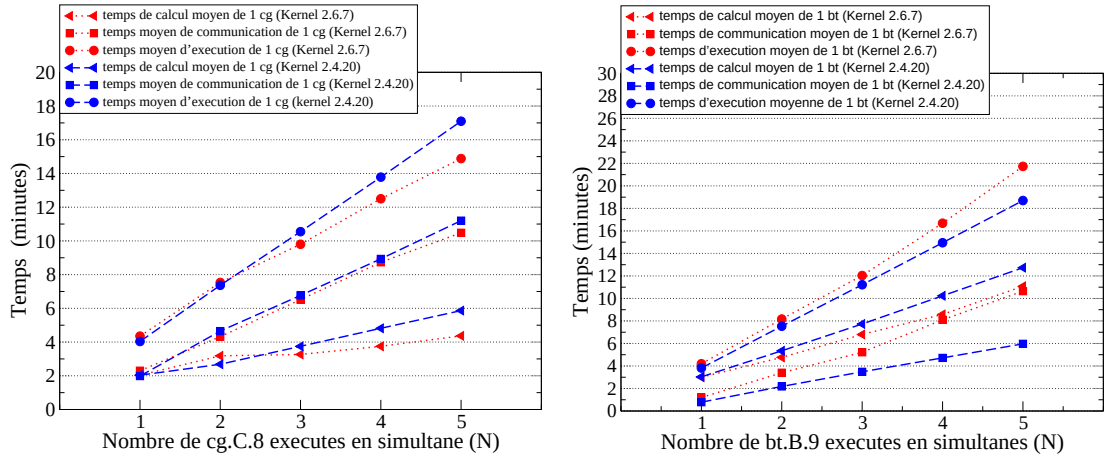


FIG. 5.4 – Comparaison des politiques d'arrêt/rechargement à l'aide de BT classe C sur 25 nœuds dans un cadre in-core (N=2 et K=1)

avant (d'un temps aléatoire t) d'ordonner le checkpoint à une application en cours d'exécution.

La question est de savoir quelle politique est la plus adaptée en essayant de charger au plus tôt la prochaine application et d'accélérer l'exécution. Dans l'ordonnancement hybride on veille à ce qu'on soit tout le temps dans un cadre in-core même lors des prises d'images de checkpoint. Une seule mesure d'exécution dans un cadre in-core est alors suffisante pour la comparaison. la figure 5.4 représente le temps total de l'exécution de 2 BTs en Gang-Scheduling (cas particulier de l'ordonnancement hybride) et qui permet de voir quelle est l'exécution la plus rapide. On constate que la politique C est un peu meilleure et on remarque un écart type de seulement 3% entre les trois politiques. Cependant, les politiques B et C nécessitent l'équivalent d'une application de mémoire libre pour rester dans un



(a) CG classe C sur 8 processeurs.

(b) BT classe B sur 9 processeurs.

FIG. 5.5 – Décomposition temps moyen de calcul et de communications pour une application lors de l'exécution de N applications en Co-scheduling

cadre in-core étant donné qu'on recharge une application avant d'en arrêter une autre. La politique A étant moins coûteuse en mémoire et aussi efficace en temps que les autres, c'est la politique choisie dans toutes les autres expérimentations.

Nombre d'appls (N)	Kernel 2.4.20		Kernel 2.6.7	
	CG	BT	CG	BT
1	0	0	0	0
2	0.13	0.04	0.33	3.35
3	0.13	0.25	0.68	2.04
4	0.30	0.23	1.07	2.41
5	0.35	0.13	1.06	0.85

FIG. 5.6 – Écart type du temps d'exécution de N applications exécutées en Co-scheduling in-core (%)

L'ordonnancement hybride dépend fortement de l'ordonnanceur du système d'exploitation car durant un quantum, on exécute des applications en Co-Scheduling. Donc, une autre étude sur le Co-Scheduling dans un cadre in-core a été effectuée pour étudier et comparer les recouvrements communications-calculs par les ordonnanceurs de différents noyaux Linux. Le but étant aussi de s'assurer que les recouvrements sont toujours aussi bons avec différents noyaux et que les communications ne sont pas trop retardées par les recouvrements. Les mesures consistent à exécuter N applications en simultané dans un cadre in-core, à prélever le temps moyen de communication et de calcul pour une application et de vérifier

que l'ensemble des applications sont traitées équitablement. Cela pour deux noyaux (Kernel) Linux : Kernel 2.4.20 et Kernel 2.6.7. Les résultats sont illustrés dans la figure 5.5 et la figure 5.6.

On constate que les écarts type dans la table (fig.5.6) sont très faibles, ce qui signifie que les deux noyaux se comportent de manière très similaire. Le traitement des applications pour (BT ou CG) par le noyau 2.4.20 est légèrement plus équitable comparé au noyau 2.6.7. Cela explique l'évolution linéaire des courbes pour le noyau 2.4.20. On remarque aussi pour ce dernier que le temps d'exécution d'une application et plus petit que *(N*temps exécution d'une application)* et qu'il s'améliore pour CG plus rapidement que pour BT. Ceci est dû à l'aspect communications intensives de CG et à un meilleur recouvrement communications-calculs.

Contrairement à CG, les performances avec le noyau 2.6.7 ne sont pas meilleures qu'avec le noyau 2.4.20. Le fait que les calculs sont plus rapides ralenti les communications qui sont retardées par les recouvrements par des calculs. Ces recouvrements dépendent fortement de la politique d'ordonnancement des systèmes d'exploitation.

Conclusions et perspectives

Dans le cadre de ce stage, j'ai intégré l'équipe architecture parallèle au sein du projet MPICH-V. J'ai étudié différents protocoles de tolérance aux pannes implémentés pour le calcul MPI. Je me suis intéressée au problème de l'ordonnancement sur une telle architecture. J'ai effectué des recherches sur plusieurs techniques d'ordonnancement et tiré profit des résultat pour développer une nouvelle approche adaptée à notre architecture. J'ai implémenté cette nouvelle approche ainsi que deux techniques classiques pour comparer leurs performances. Ce travail a fait l'objet d'un article accepté au « 5th IEEE/ACM International Workshop on Grid Computing ».

Ce travail représente une étape préliminaire à l'intégration d'un système d'ordonnancement performant pour les Grilles. Disposant d'un ordonnanceur au niveau des clusters, il s'agit désormais de concevoir au niveau de l'ensemble des clusters d'une Grille un ordonnanceur tirant partie des spécificités de chaque site.

Nous souhaitons aussi étudier différentes approches de tolérances aux pannes pour les Grilles. Les Grilles rassemblant des clusters aux domaines administratifs distincts, elles se composent de deux niveaux de hiérarchie : au niveau d'un cluster et au niveau de l'ensemble de la Grille. Le protocole de tolérance aux pannes devra lui-même tenir compte de cette hiérarchie. Pour réaliser celle-ci, une possibilité consiste à intégrer des mécanismes de tolérances aux pannes à différents niveaux de la couche logicielle : au niveau application avec des instructions utilisateurs et/ou au niveau transport pour les actions de reprise automatiques.

Bibliographie

[1] P. Strazdins and J. Uhlmann, "Local scheduling out-performs gang scheduling on a beowulf cluster," Department of Computer Science, Australian National University, Tech. Rep., January 2004.

[2] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda, "Implementation of gang-scheduling on workstation cluster," in *Proceeding of IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Springer-Verlag, April 1996, pp. 126–139.

[3] A. Hori, H. Tezuka, and Y. Ishikawa, "Overhead analysis of preemptive gang scheduling," *Lecture Notes in Computer Science*, vol. 1459, pp. 217–230, April 1998.

[4] J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette, "An infrastructure for efficient parallel job execution in terascale computing environments," in *Proceedings of the IEEE/ACM SC98 Conference*, November 1998, p. 50.

[5] D. G. Feitelson and M. A. Jette, "Improved utilization and responsiveness with gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., vol. 1291. Springer Verlag, 1997, pp. 238–261.

[6] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, June 2003, pp. 581–592.

[7] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato, "PM : An operating system coordinated high performance communication library," in *Proceedings of the international conference and Exhibition on high-Performance Computing and Networking*. Springer-Verlag, April 1997, pp. 708–717.

[8] A. Hori, H. Tezuka, and Y. Ishikawa, "Highly efficient gang-scheduling implementation," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, November 1998, pp. 1–14.

[9] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "Implications of I/O for gang scheduled workloads," *Proceedings of the Job Scheduling Strategies for Parallel Processing*, pp. 215–237, 1997.

[10] F. A. B. da Silva and I. D. Scherson, "Improving parallel job scheduling using runtime measurements," *Proceedings of the 6th Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 18–38, May 2000.

[11] K. D. Ryu, N. Pachapurkar, and L. L. Fong, "Adaptive memory paging for efficient gang scheduling of parallel applications," in *18th International Parallel and Distributed Processing Symposium IPDPS'04*, April 2004.

[12] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov, "MPICH-V : Toward a scalable fault tolerant MPI for volatile nodes," in *High Performance Networking and Computing (SC2002)*. Baltimore USA : IEEE/ACM, November 2002.

[13] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *IEEE International Conference on Cluster Computing (Cluster 2003)*. IEEE CS Press, december 2003.

[14] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in *to appear in IEEE International Conference on Cluster Computing (Cluster 2004)*, 2004.

[15] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI : The Complete Reference*. The MIT Press, 1996.

[16] K. M. Chandy and L.Lamport, "Distributed snapshots : Determining global states of distributed systems," in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75.

[17] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Report NAS-95-020, 1995.