Thèse de Doctorat de l'Université Paris VI

Spécialité Informatique

Présentée par Fabricio Alves Barbosa da Silva
Pour obtenir le grade de
Docteur de l'Université Paris VI

Modélisation et analyse d'une classe d'algorithmes d'ordonnancement pour Machines Parallèles

Soutenue le 1 Decembre 2000,
devant le jury composé de

M. Isaac SCHERSON	Examinateur
M. Jean-Marc GEIB	Rapporteur
M. Denis TRYSTRAM	Rapporteur
M. Alain GREINER	Examinateur
M. Jean-Louis PAZAT	Examinateur
M. Claude GIRAULT	Examinateur

A mes parents
A Rodrigo, Lisiane, Luiza et Beatrice

Avant-Propos

Je souhaite remercier vivement M. Isaac D. Scherson., Professeur à l'Université de Californie à Irvine, qui m'a encadré pendant cette thèse. Ses conseils, encouragements et amitié m'ont été indispensables dans la préparation de ce travail.

J'adresse mes remerciements à M. Alain Greiner, Directeur du département Architecture du Laboratoire d'informatique de Paris VI, pour m'avoir accueilli au sein de son laboratoire.

Je remercie sincèrement M. Denis Trystram, Professeur à l'INPG à Grenoble, et M. Jean-Marc Geib, Professeur au Laboratoire de Informatique Fondamentale de Lille, qui m'ont fait l'honneur d'être rapporteurs dans cette thèse. Je tiens également à remercier M. Jean-Louis Pazat, Maître de conférences de l'IRISA à Rennes, et M. Claude Girault, Professeur à l'Université Paris VI, qui ont accepté de juger mon travail en participant à mon jury de thèse.

Je profite de cette occasion pour adresser mes remerciements à tous les membres du département Architecture du Laboratoire LIP6. Je voudrais aussi remercier les collègues Celio Albuquerque, Marcio Dias et Walfredo Cirne pour leur amitié et leurs encouragements pendant mes séjours aux Etats-Unis. J'adresse une pensée spéciale à Luis Miguel Campos pour son aide pendant la durée de ce travail et son amitié. Je remercie aussi Andrea Costa, pour son soutien et son encouragement constants.

A tous ceux et celles qui m'ont aidé de près ou de loin, à un moment ou à un autre, dans la préparation de cette thèse, je tiens à adresser mes remerciements les plus sincères.

Résumé

L'ordonnancement parallèle est un problème important dont la solution peut mener à améliorer sensiblement l'utilisation des ordinateurs parallèles modernes. Il est défini comme : " Etant donné un ensemble de tâches appartenant à plusieurs applications parallèles dans une machine parallèle, trouver une allocation spatiale et temporelle pour exécuter toutes les tâches efficacement ". Une application parallèle constituée de plusieurs tâches peut apparaître à un instant donné, attendre que les ressources demandées soient disponibles, puis être exécutée. Les temps associés à la phase d'attente ainsi qu'a phase d'exécution sont dépendantes de l'algorithme d'ordonnancement et de la charge de travail.

Dans la majeure partie de cette thèse, nous nous concentrons sur les algorithmes d'ordonnancement basés sur le "Gang scheduling", à savoir, un paradigme où toutes les tâches d'une même application parallèle sont regroupées et ordonnancées de manière concurrente sur des processeurs distincts. Les raisons de considérer l'ordonnancement Gang sont le partage efficace des ressources et la facilité de programmation. L'utilisation du partage de temps parmi les processeurs permet une dégradation graduelle de la performance à mesure que la charge de travail augmente. Les performances des applications parallèles très synchronisées sont fortement améliorées par rapport à un ordonnancement non coordonné.

Cette thèse est divisée en deux parties distinctes : dans la première partie, on présente l'algorithme d'ordonnancement Gang, en identifiant ses avantages et ses faiblesses, puis on effectue une analyse théorique de l'algorithme Gang et des stratégies d'empaquetage. La deuxième partie présente des nouvelles méthodes d'ordonnancement dans une machine parallèle, s'appuyant sur des mesures dynamiques effectuées au moment de l'exécution. Dans cette partie, nous proposons un nouvel algorithme d'ordonnancement parallèle nommé "Concurrent Gang", qui utilise des informations dynamiques obtenues sur les tâches au moment de l'exécution, en vue d'améliorer la performance de l'ordonnanceur parallèle.

Mots-clés

Ordonnancement Parallèle, Ordonnancement Gang, Parallélisme, Coscheduling, Système d'exploitation

Abstract

Parallel job scheduling is an important problem whose solution may lead to better utilization of modern parallel computers. It is defined as : “Given the aggregate of all tasks of multiple jobs in a parallel system, find a spatial and temporal allocation to execute all tasks efficiently”. For the purposes of scheduling, we view a computer as a queueing system. An arriving job may wait for some time, receive the required service, and depart. The time associated with the waiting and service phases is a function of the scheduling algorithm and the workload. Some scheduling algorithms may require that a job wait in a queue until all of its required resources become available (as in variable partitioning), while in others, like time slicing, the arriving job receives service immediately through a processor sharing discipline.

In most of this thesis, we focus on scheduling based on Gang service, namely, a paradigm where all tasks of a job in the service stage are grouped into a Gang and concurrently scheduled in distinct processors. Reasons to consider Gang service are responsiveness, efficient sharing of resources and ease of programming. In Gang service the tasks of a job are supplied with an environment that is very similar to a dedicated machine. It is useful to any model of computation and any programming style. The use of time slicing allows performance to degrade gradually as load increases. Applications with fine-grain interactions benefit of large performance improvements over uncoordinated scheduling.

This thesis is divided into two distinct parts : in the first part, we present the algorithm Gang scheduling, we identify its advantages and weaknesses, and we carry out a theoretical analysis of the Gang scheduling algorithm. The second part presents new methods to improve scheduling in a parallel machine based on runtime measurements at execution time. In this second part we propose a new parallel job scheduling algorithm named Concurrent Gang which uses the runtime information obtained on tasks at execution time in order to improve the performance of the parallel scheduler.

Keywords

Parallel Job Scheduling, Gang Scheduling, Parallel Computation, Coscheduling, Operating Systems

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Dans les monoprocesseurs, l'ordonnancement est l'activité consistant à décider quelle tâche va être exécutée sur l'unité centrale de traitement à un moment donné. Dans les machines parallèles, l'ordonnancement a une autre dimension : il faut décider non seulement quand une tâche sera exécutée, mais également où elle sera exécutée, c'est à dire sur quel processeur. Ainsi les systèmes parallèles nécessitent une allocation bidimensionnelle des ressources aux applications parallèles, à la fois dans le temps et dans l'espace.

L'ordonnancement parallèle est un problème important dont la solution peut mener à améliorer sensiblement l'utilisation des ordinateurs parallèles modernes. Il est défini comme : " Etant donné un ensemble de tâches appartenant à plusieurs applications parallèles dans une machine parallèle, trouver une allocation spatiale et temporelle pour exécuter toutes les tâches efficacement ". Une application parallèle constituée de plusieurs tâches peut apparaître à un instant donné, attendre une période de temps, recevoir le service exigé, et se terminer. Les temps associés à l'attente et aux phases de service sont dépendants de l'algorithme d'ordonnancement et de la charge de travail. Quelques algorithmes d'ordonnancement exigent une attente dans une file d'attente jusqu'à ce que toutes les ressources nécessaires deviennent disponibles (comme dans l'algorithme "variable partitioning" [34]), alors que dans d'autres, comme les algorithmes basés sur le partage dans le temps, l'application parallèle reçoit le service immédiatement.

Dans la majeure partie de cette thèse, nous nous concentrons sur les algorithmes d'ordonnancement basés sur le "Gang scheduling", à savoir, un paradigme où toutes les tâches d'une même application parallèle sont regroupées et concurremment ordonnancés sur des processeurs distincts. Les raisons de considérer l'ordonnancement Gang sont le partage efficace des ressources et la facilité de programmation [35]. Il supporte n'importe quel modèle de programmation. L'utilisation du partage de temps parmi les processeurs permet une dégradation graduelle de la performance à mesure que la charge de travail augmente [34]. Les performances des applications parallèles très synchronisées sont fortement améliorées par rapport à un ordonnancement non coordonné [37].

Pour ces caractéristiques, l'ordonnancement Gang a été implementé par quelques-uns des plus grand fabricants de machines parallèles (IBM [41], SGI [5], Tera MTA [26], Cray [57]) et est utilisé comme stratégie d'ordonnancement par d'importants centres de calcul scientifique, comme le Lawrence Livermore National Laboratory(LLNL) [35], dans son Cray T3E.

Un problème majeur de l'approche Gang scheduling est la performance des programmes parallèles qui font beaucoup d'entrées/sorties, puisqu'une tâche qui effectue une entrée/sortie est bloquée, et le processeur correspondant est inutilisé. En effet, les ordonnanceurs Gang typiques savent seulement ordonnancer des gangs de tâches et non des tâches individuelles. Dans cette thèse nous utilisons l'ordonnancement Gang comme point de départ pour proposer une nouvelle classe de politique d'ordonnancement qui s'appuie sur des mesures dynamiques effectuées au moment de l'exécution pour améliorer l'utilisation des ressources et le débit de traitement dans l'ordonnancement parallèle en général et l'ordonnancement Gang en particulier.

1.1 Serveurs Parallèles et la loi de Moore

Bien que la recherche dans l'ordonnancement parallèle aujourd'hui soit d'avantage orientée vers les processeurs massivement parallèles (MPP) et les serveurs de calcul tels que des serveurs SMP, le parallélisme peut être utilisé même au niveau des stations de travail. L'intégration de processeurs multiples dans une même carte mère, et dans un futur proche, dans une même puce, ramène le calcul parallèle plus près du niveau des ordinateurs de bureau. L'importance croissante du traitement parallèle est mise en valeur par le fait que l'industrie fait face à une période de changement fondamental. Pendant plus de vingt années, les concepteurs d'ordinateurs ont compté sur la loi de Moore pour augmenter la puissance de calcul. La loi de Moore rend compte du doublement de la densité d'intégration (en nombre de transistors par puce) tous les 24 mois et elle est baptisée du nom du chercheur qui a identifié pour la première fois cette tendance, Gordon Moore. Des transistors plus petits ont comme conséquence des puces plus rapides qui se traduisent en ordinateurs plus rapides. Mais il y a un accord général des experts pour prévoir un ralentissement vers 2005, et probablement plus tôt [61].

Une autre façon d'augmenter la performance sans augmentation de la fréquence de base des processeurs est évidemment le traitement parallèle. Les ordinateurs géants d'aujourd'hui contiennent des centaines ou même des milliers de microprocesseurs. Grâce à la baisse du prix des microprocesseurs, le parallélisme s'est déjà introduit sur le marché des stations de travail. Actuellement, il est possible de trouver des serveurs avec huit processeurs intégrés sur la carte mère. Cette tendance est à la base de l'annonce d'IBM en décembre 1999 d'un plan quinquennal pour construire l'ordinateur Blue Gene, qui sera capable d'exécuter 1 million de milliards d'opérations en virgule flottante par second [75]. Le but est la simulation du pepliement des protéines. Chaque processeur du Blue Gene aura une puissance de calcul de 1 gigaflops, soit une puissance comparable à celle des ordinateurs géants d'il y a une décennie. 32 processeurs seront intégrés dans une puce avec 16 méga-octets de mémoire. Soixante quatre de ces puces seront placés sur chaque carte. Ceci correspond à 2 teraflops de puissance de calcul, ce qui est presque autant que les ordinateurs géants les plus puissants d'aujourd'hui, dans un volume plus petit que celui d'un ordinateur de bureau. Huit des ces cartes seront placés dans une tour, et 64 tours seront interconnectées pour créer le Blue Gene.

Une telle utilisation intensive du parallèlisme met en valeur l'importance des ordonnanceurs parallèles pour un usage efficace des ressources disponibles. Cette thèse de donne quelques réponses à des problèmes actuels en ordonnancement parallèle, qui peuvent être aussi bien appliquées aux stations de travail parallèles qu'aux systèmes massivement parallèles.

1.2 Plan de la thèse

Cette thèse est divisée en deux parties distinctes : Dans une première partie, on présente l'algorithme d'ordonnancement Gang, on identifie ses avantages et ses faiblesses, et on effectue une analyse théorique de l'algorithme Gang et des stratégies d'empaquetage. La deuxième partie présente des nouvelles méthodes pour faire l'ordonnancement dans une machine parallèle s'appuyant sur des mesures dynamiques effectuées au moment de l'exécution.

1.2.1 Partie I

La première partie est composée de quatre chapitres. Le chapitre 2 est une introduction générale. Le chapitre 3 pose le problème de l'ordonnancement parallèle, et décrit l'algorithme Gang. Le chapitre 4 présente une analyse de compétitivité des algorithmes d'ordonnancement basés sur Gang. Le 5 chapitre propose une analyse du cas moyen pour le problème de partage de ressources dans l'ordonnancement Gang, en analysant les cas uni et multidimensionnel.

Chapitre 3

Dans ce chapitre nous examinons les travaux existants au sujet de l'ordonnancement parallèle, en soulignant l'importance des stratégies d'ordonnancement dérivées de l'ordonnancement Gang. L'espace des solutions de ce problème a deux dimensions : une temporelle, en raison de l'existence d'applications parallèles multiples qui partagent les ressources dans le temps, et une autre spatiale, étant donné qu'une machine parallèle est composée de multiples processeurs et que plusieurs applications parallèles peuvent s'exécuter simultanément.

Dans le partage dans l'espace, également connu sous le nom de partitionnement, un sous-ensemble de processeurs de la machine est dédié à une application parallèle jusqu'à son accomplissement.

Dans le partage dans le temps, tous les processeurs de la machine sont consacrés seulement à une application parallèle, et des tranches de temps sont allouées à chaque application. L'avantage de cette classe d'algorithmes est que le temps d'attente est réduit pour les applications parallèles nouvellement arrivées : Toutes les applications parallèles seront ordonnancées s'il y a assez de mémoire disponible. Cependant, la perte due à la fragmentation peut être significative, puisque tous les processeurs sont consacrés à l'une seule application à la fois, alors que la plupart des applications n'exigent pas un nombre de processeurs égal au nombre de processeurs de la machine.

En combinant le partage dans le temps et dans l'espace, il est possible de combiner les avantages des deux stratégies, c'est à dire la réduction de temps d'attente avec une utilisation efficace des ressources de la machine. C'est l'approche utilisée, par exemple, dans l'ordonnanceur Gang combiné avec le partitionement. Dans cette stratégie, plusieurs applications peuvent être lancées simultanément, s'il y a les ressources suffisantes disponibles. La

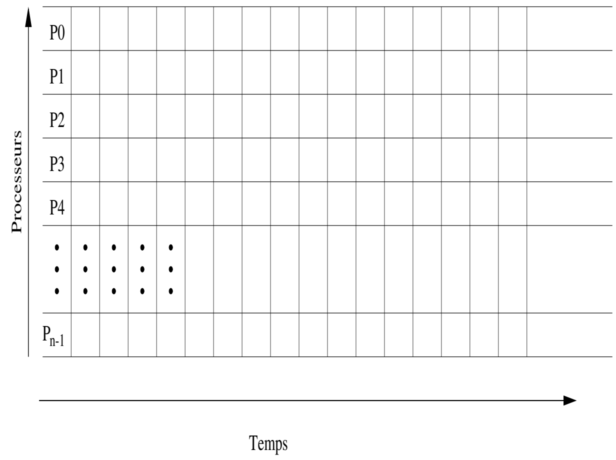

FIG. 1.1: Trace diagram

combinaison du partage dans l'espace et dans le temps par l'ordonnanceur Gang est analysée en détail dans ce chapitre.

L'utilisation des ressources dans le temps et dans l'espace peut être visualisée à l'aide d'un diagramme bi-dimensionnel appellé diagramme de trace. Une dimension représente les processeurs tandis que l'autre dimension représente le temps. Une représentation semblable a été déjà utilisée, par exemple, par Ousterhout dans [72]. Un tel diagramme est illustré dans la figure 1.1.

Chapitre 4

Le chapitre 4 présente une analyse de compétitivité de l'ordonnancement Gang.

Les implications théoriques de l'ordonnancement Gang ont été peu étudiées. Subramanian et Scherson [92, 79] ont fait une analyse de compétitivité de l'ordonnancement Gang en utilisant une métrique nommée “Happiness”. Ils ont montré qu'une variante de l'ordonnancement Gang dénommée *Intruction Balanced Time Slice* est le meilleur algorithme sous cette nouvelle métrique pour une charge de travail composée de programmes parallèles de type VRAM (Vector Random Acces Machine) [7]. Ici nous proposons une analyse indépendante du modèle de programmation, en utilisant comme métrique le temps d'exécution de la charge de travail.

Soit une charge de travail fixe W. Nous définissons $T_{OPT}(W)$ la durée d'exécution de la charge de travail W pour un ordonnancement optimal. Cet ordonnancement optimal est supposé obtenu de façon " statique " grace à la connaissance complète des caractéristiques et durées de toutes les tâches composant l'application. $T_A(W)$ est défini comme le temps d'accomplissement sous un algorithme d'ordonnancement A. L'algorithme A est dit *r-competitive* si pour toutes les charges de travail W, $T_A(W) < r \times T_{OPT}(W)$. Le taux de compétitivité de l'algorithme A est r. Cette technique permettant d'évaluer un algorithme d'ordonnancement dynamique (en ligne) en comparant son exécution à l'algorithme optimal (hors ligne) s'appelle l'analyse de compétitivité. L'analyse de compétitivité permet d'évaluer les algorithmes qui sont limités d'une manière quelconque (par exemple, information limitée, puissance de calcul limitée, nombre de préemptions limité). Cette mesure a été présentée pour la première fois dans l'étude d'un problème de gestion de mémoire. L'algorithme hors-ligne optimal est référencé également en tant qu'adversaire.

Le premier résultat de ce chapitre détermine une borne inférieur pour le taux de compétitivité de l'ordonnancement Gang. En suite nous prouvons que la borne inférieur est en effet la valeur du taux de compétitivité de l'ordonnancement Gang quand celui-ci fait usage de certaines classes d'algorithmes de partage de ressources (dans notre cas les processeurs) qui supportent la migration de tâches.

Deux théorèmes sont prouvés dans le chapitre :

Theorem 1 *La borne inférieur du taux de compétitivité des algorithmes bases sur l'ordonnancement Gang est égal à 4.*

Theorem 2 *Le taux de compétitivité est égal a 4 pour les algorithmes d'ordonnancement Gang avec le partitionnement " first fit decreasing "*

Chapitre 5

Le chapitre 5 fait une analyse des algorithmes de partage de ressources qui peuvent être utilisés dans l'ordonnancement Gang. Une des contributions de ce chapitre est la proposition d'une nouvelle méthode pour l'analyse du cas moyen des algorithmes dynamiques, qu'on a nommé taux de compétitivité dynamique. Le taux de compétitivité dynamique est utilisé pour faire une analyse du cas moyen des algorithmes de partage de ressources unidimensionnels et multidimensionnels.

Dans le problème de partage unidimensionnel, les applications sont représentées par seulement un paramètre : le nombres de tâches qui composent l'application. La machine est caractérisée par le nombre de processeurs. Nous supposons que d'autres ressources, telles que la mémoire, sont illimitées. Nous analysons dans ce cas la variation dynamique du problème d'ordonnancement, où les temps d'arrivée peuvent être différents de zéro. Ce problème est semblable à un problème d'empaquetage dynamique (*on-line bin-packing*).

Dans le cas multidimensionnel les ressources disponibles dans une machine sont représentées par un vecteur m-dimensionnel $R = (R_1, R_2...r_m)$ et les ressources exigées par un travail J sont représentées par un vecteur k-dimensionnel $J = (J_1, J_2...j_k)$.

Les résultats de ce chapitre ont été publiés dans les anales du "1999 International Simposium of Parallel Architectures, Algorithms and Networks". Une version etendue de cet article a été soumis comme article invité dans le "International Journal of Foundations of Computer Science".

1.2.2 Partie II

La deuxième partie de la thèse présente des méthodes pour mesurer les caractéristiques des tâches lors de l'exécution et propose un nouvel algorithme d'ordonnancement parallèle nommé Concurrent Gang utilisant les informations dynamiques obtenues sur les tâches au moment de l'exécution pour améliorer la performance de l'ordonnanceur parallèle.

Chapitre 6

Le chapitre 6 décrit un nouvel algorithme d'ordonnancement basé sur l'ordonnancement Gang , qui utilise des mesures faites au moment de l'exécution pour résoudre le problème du blocage des tâches. Ceci a été fait en utilisant un algorithme de classification dynamique des tâches lors de l'exécution. On utilise la théorie des ensembles flous, pour calculer le degré de similitude de chaque tâche par rapport a un ensemble de classes pré-définies. Des exemples de classes possibles sont : calcul intensif, communication intensive et Entrée/Sortie intensive. Avec cette classification, une priorité locale à un processeur est calculée pour chaque tâche, et cette priorité est utilisée pour décider quelle tâche sera activée en cas de blocage. Le modèle architectural que nous considérons dans ce chapitre est un multi-processeur avec mémoire distribuée composé de quatre composants principaux : 1) Processeur/mémoire, 2) un réseau d'interconnexion point à point, 3) le synchroniseur qui envoi un signal de synchronisation (horloge) à tous les processeurs à intervalles réguliers et 4) un *Front End*. Cette architecture est semblable à celle définie dans le modèle BSP [94].

Les différences principales par rapport à l'ordonnanceur Gang standard sont la définition explicite d'un synchroniseur global et la présence de l'ordonnanceur local qui décide quoi faire si une tâche de l'application exécutée en tant que Gang est bloquée.

L'approche Concurrent Gang est plus avantageuse pour les charges de travail qui font des E/S intensives, comme cela est démontré dans la section des résultats expérimentaux de ce chapitre. Pour des charges de travail qui exigent un ordonnancement coordonné, l'algorithme Concurrent Gang devient équivalent à l'algorithme Gang standard.

L'utilisation des ressources par l'algorithme Concurrent Gang est améliorée parce que, en cas de processeur inactif ou de tâche bloquée, l'algorithme Concurrent Gang essaye toujours de programmer les tâches qui n'exigent pas, à ce moment, l'ordonnancement coordonné avec d'autres tâches de la même application.

Un résultat important de ce chapitre est la comparaison entre le calcul de priorité réalisé par Concurrent Gang et une politique simple de "round-robin" pour choisir quelle tâche doit être activée en cas de blocage. On montre que la définition d'une priorité " intelligente " dépendant du comportement observé de chaque tâche permet d'obtenir de meilleure résultats pour différentes types de charges de travail.

Les résultats présentés dans ce chapitre ont été publiés dans les anales de la conférence "IEEE International Parallel and Distributed Processing Symposium 2000", Cancun, Mexico et aussi dans la conférence "IASTED Conference on Parallel and Distributed Computing Systems 1999", Boston, USA.

Chapitre 7

Dans ce chapitre nous comparons les algorithmes qui se servent des mesures obtenues au moment de l'exécution comme par example Concurrent Gang, avec d'autres algorithmes qui ne se servent pas d'une telle information, comme les ordonnanceurs Gang et les ordonnanceurs locaux inconscients. La conclusion est qu'un ordonnanceur Concurrent Gang sera toujours au moins aussi bon qu'un ordonnanceur Gang utilisant la même stratégie d'empaquetage. Cela est valable aussi pour un ordonnanceur local inconscient, comme cela est démontré dans le théorème suivant :

Theorem 3 *Les ordonnanceurs inconscients, tels que l'ordonnanceur Gang et les ordonnanceurs locaux inconscients, ne peuvent pas être meilleur que l'ordonnanceur Concurrent Gang pour la même stratégie de distribution de tâche, si on utilise le temps d'exécution de la charge de travail comme métrique*

Le chapitre 7 présente aussi une analyse de l'algorithme Concurrent Gang soumis à une charge de travail composée de travaux parallèles irréguliers.

Les programmes parallèles peuvent être classifiés réguliers ou irréguliers. Dans les programmes réguliers le degré de parallélisme demeure constant pendant l'exécution. A l'inverse, dans les programmes irréguliers le nombre de tâches change ou la quantité de calcul par tâche peuvent changer au cours de l'exécution. L'irrégularité d'un programme peut être exprimée en :

- la variation du nombre de tâches pendant l'exécution. Nous nommerons cette variation comme Y-irrégularité.
- la variation de la quantité de calcul exécutée par une tâche pendant l'exécution. Nous définirons cette variation comme X-irrégularité.

Un programme qui présente des irrégularités de X et de Y est un programme complètement irrégulier.

Par rapport à la Y-irregularité, nous proposons une variante de Concurrent Gang liée à un algorithme d'équilibrage de charge qui utilise les informations recueillies au moment de l'exécution, quand le nombre de tâches varie. Par rapport a X-irrégularité, nous montrons que l'ordonnanceur local de Concurrent Gang est capable d'augmenter l'utilisation de la machine de façon efficace en présence des irrégularités de quantité de calcul.

Chapitre 8

Dans ce chapitre, nous analysons et systématisons l'utilisation d'informations recueillies sur les tâches au moment de l'exécution. Notre objectif est d'utiliser des informations tel que le nombre d'appels d'E/S, la durée des appels d'E/S, le nombre de messages reçus, le nombre de messages envoyés, le nombre de barrières et d'autres informations disponibles en fonction de l'architecture de la machine afin d'associer une tâche à une classe prédéfinie en utilisant les ensembles flous et les estimateurs Bayésiens.

Le principal objectif de ce chapitre est de présenter un mécanisme de classification de tâches plus robuste que celui décrit dans le chapitre 6 grâce à l'utilisation de théorie bayésienne de la décision. La théorie bayésienne de la décision est une méthode mathématique qui guide un décideur en choisissant une ligne de conduite face à l'incertitude au concernant les conséquences de ce choix [49]. En particulier nous allons définir dans ce chapitre un classificateur de tâche à l'aide d'un estimateur bayésien adapté à la théorie des ensembles flous.

La classification d'une tâche peut changer avec le temps, puisque nous considérons que les caractéristiques des programmes parallèles peuvent changer pendant l'exécution. Quelques utilisations possibles pour ce mécanisme de classification de tâches sont, par exemple, de décider quoi faire dans le cas d'un processeur inactif, ou définir le temps d'attente d'une réception bloquante dans un mécanisme du type “spin-lock” en fonction de la charge de travail sur un processeur. Une utilisation possible est de fournir un meilleur service aux programmes qui font des E/S intensives dans ordonnancement Gang, en identifiant les tâches qui font beaucoup d'E/S afin de les remettre à plus tard dans les périodes d'inactivité du processeur. Cette approche est différente de celle proposée par Lee et al. [58] puisqu'elle n'interrompt pas les applications parallèles en cours d'exécution.

Des résultats de simulation sont présentés, et concernent des charges de travail mixtes. Un résultat important de ce chapitre est l'amélioration effective du service fourni aux applications parallèles qui font beaucoup d'entrées/sorties grâce à la mesure des temps d'attente lors de l'exécution.

Les résultats concernant ce chapitre ont été publiés dans le volume "Job Scheduling Strategies for Parallel Processing", Lecture Notes on Computer Science, 2000. Une version preliminaire a été publiée et acceptée pour présentation dans le "6th workshop of Job Scheduling Strategies for Parallel Processing", Cancun, Mexico, Mai 2000.

Première partie

Gang Scheduling

Chapitre 2

Introduction

In uniprocessors, scheduling is the activity of deciding which task gets to run on the CPU. In multiprocessors and multicomputers, scheduling has another dimension : not only deciding when a task will run, but also where it will run, i.e. on which Processing Element (PE). Thus parallel systems allow a two-dimensional division of resources among competing jobs, both in time and in space, through parallel job scheduling.

Parallel job scheduling is an important problem whose solution may lead to better utilization of modern multiprocessors parallel computers. It is defined as : "Given the aggregate of all tasks of multiple jobs in a parallel system, find a spatial and temporal allocation to execute all tasks efficiently". For the purposes of scheduling, we view a computer as a queueing system. An arriving job may wait for some time, receive the required service, and depart [40]. The time associated with the waiting and service phases is a function of the scheduling algorithm and the workload. Some scheduling algorithms may require that a job wait in a queue until all of its required resources become available (as in variable partitioning), while in others, like time slicing, the arriving job receives service immediately through a processor sharing discipline.

In most of this thesis, we focus on scheduling based on Gang service, namely, a paradigm where all tasks of a job in the service stage are grouped into a Gang and concurrently scheduled in distinct processors. Reasons to consider Gang service are responsiveness [35], efficient sharing of resources [48] and ease of programming. In Gang service the tasks of a job are supplied with an environment that is very similar to a dedicated machine [48]. It is useful to any model of computation and any programming style. The use of time

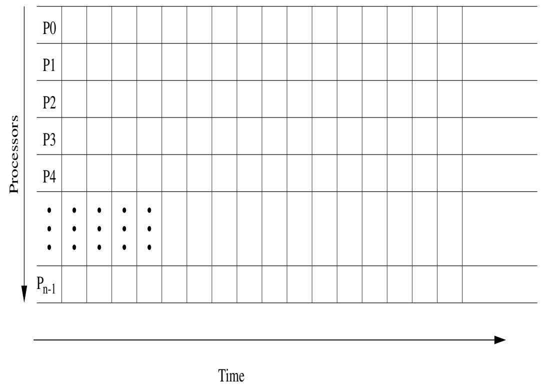

FIG. 2.1: A trace diagram

slicing allows performance to degrade gradually as load increases. Applications with fine-grain interactions benefit of large performance improvements over uncoordinated scheduling [37]. One main problem related with Gang service is the performance of I/O bound and interactive jobs [58] and the fact that typical Gang schedulers only know how to schedule gangs of tasks. In this thesis we use Gang scheduling as a starting point for proposing a new class of scheduling policies that uses runtime measurements to improve utilization and throughput in parallel job scheduling in general and Gang scheduling in particular.

2.1 Dimensions of the Parallel Job Scheduling Problem

Consider the scheduling of a set of parallel jobs. A useful tool to help visualize the time utilization in parallel machines is a bidimensional diagram dubbed *trace diagram*. The trace diagram is also known in the literature as Ousterhout matrix [72]. Referring to figure 2.1, one dimension represents processors while the other dimension represents time. Through the trace diagram it is possible to visualize the time utilization of the set of processors given a scheduling algorithm.

Regarding the trace diagram, the parallel job scheduling becomes hence equivalent to computing the trace diagram for a given workload (a set of jobs). Gang Scheduling in particular can hence be defined with respect to the trace diagram as the concurrent scheduling of the set of tasks of a job in a time slice. The trace diagram is first computed at power up and updated at each workload change (task arrival, completion, etc).

Up to now, the machine was characterized only by its number of processors. In parallel job scheduling, not only the number of processors is important, but also other resources of the machine such as amount of main memory, required disk space, etc. Scheduling under multiple constraints is associated with multidimensional resource sharing analysis, which is stated in the next subsection.

2.1.1 Multidimensional Resource Sharing

In the multi-dimensional case the resources available in a machine are represented by a m-dimensional vector $R = (R_1, R_2, ...R_m)$ and the resources required by a job J are represented by a k-dimensional vector $J = (J_1, J_2, ...J_k), k \leq m$. Observe that the number of dimensions in the trace diagram increases accordingly to the number of different resources considered by the scheduling algorithm. Of particular interest for Gang scheduling algorithms is the amount of memory required for each job, since most of the parallel machines available today do not have support for virtual memory, and the limited amount memory available determine the number of jobs that can share a machine at any given time.

In the case of a distributed memory machine, the machine itself is modeled as a P-dimensional vector $R(t) = (R_1, R_2, ...R_P)$ where R_I represents the amount of memory available at time t in node I. The job is represented as a f-dimensional vector $J(t) = (J_1, J_2, ...J_f)$, where f is the maximum number of tasks of a job and J_I is the amount of memory required for task I at time t.

2.2 Summary of the Thesis and Contributions

In the first part of thesis we begin by making a survey on Gang scheduling. Then a competitive analysis of Gang scheduling is make, which uncover some caveats related to this algorithm. An analysis of resource sharing in Gang scheduling follows. In this analysis we study both the one-dimensional and multidimensional resource sharing problems. We also propose a new methodology for comparing dynamic algorithms named Dynamic Competitive Ratio.

As a consequence of the analysis made in the first part of the thesis, in the second part we propose a new scheduling policy, dubbed Concurrent Gang, that improves utilization and throughput in Gang scheduling. This is true in particular for I/O bound jobs, as we shall see. A comparison between Concurrent Gang, Gang scheduling and oblivious local schedulers is made, and then the runtime measurements scheme first used for Concurrent Gang is systematized for used in different ways and with different scheduling algorithms through the use of Bayesian estimators. Finally we present our conclusions and future directions of research.

Chapitre 3

Previous Work on Gang Scheduling

In this chapter we survey the work in the literature about parallel job scheduling, giving emphasis to Gang scheduling-based strategies. The parallel job scheduling problem consists in how to run a set of parallel jobs in a parallel machine. We define a workload as being composed by a set of parallel jobs. The parallel job itself is composed by a set of tasks. The parallel machine is composed by a set of processing elements (PEs).

The space of solutions of this problem has two dimensions : a temporal one, due to the existence of multiple parallel jobs that shares computing resources, and a spatial dimension, due to the fact that a parallel machine is composed of multiple processors and more than one parallel job may be running in the machine in a given moment of time.

Given a job composed of N tasks, in Gang scheduling these N tasks compose a process working set, as defined originally by Ousterhout [72], and all tasks belonging to this process working set are scheduled simultaneously in different processors, i.e., Gang scheduling algorithms is the class of algorithms that schedule on the basis of whole process working sets. Gang scheduling allows both the time sharing as well as the space sharing of the machine. In the following we first present a classification of the job scheduling problem. Then we discuss the two dimensions of parallel job scheduling and we make a detailed presentation of Gang scheduling, describing some actual implementations of Gang scheduling in parallel systems.

3.1 Classifications Related to the Parallel Job Scheduling Problem

In this section we present some classifications for the parallel job scheduling problem. The classifications/definitions presented in this section will be used in the following chapters of the thesis.

3.1.1 Static vs. Dynamic

A *static* scheduling is the one which all the release times are zero, i.e., all the jobs are available for execution at the start of the schedule. Given a workload $W = \{J_1, J_2, ..., J_n\}$, in this case we have job arrival times $a_i = 0$, for all $1 < i < n$. A *dynamic* scheduling problem allows arbitrary non-negative arrival epochs, i.e. $a_i \geq 0$ where $a_1 < a_2 < ..... < a_n$ for a workload $W = \{J_1, J_2, ..., J_n\}$. By definition $a_1 = 0$.

3.1.2 Preemptive vs. Non-preemptive

In *preemptive* parallel scheduling the execution of any parallel task can be suspended at any time and resumed later from the point of preemption.

A *Non-preemptive* parallel scheduling do not allows preemption. In this case a job runs until completion in the same set of processors that it was originally scheduled.

3.1.3 Periodic vs. Non-periodic

Consider a workload W and a period of time T where there is no changes in W. A *periodic* schedule repeats itself at regular finite intervals in the time dimension for workload W during time T. If the interval tends to infinite the schedule is *non-periodic*.

3.1.4 Clairvoyant vs. Non-Clairvoyant

In clairvoyant parallel scheduling the characteristics of a job (in particular, its execution time, release time and dependence on other jobs) are known a priori, and the scheduler may use this information to assign processors/intervals of time to jobs. In non-clairvoyant parallel scheduling, the scheduler has no knowledge of jobs' characteristics [69], unless the initial processor/memory requirements of a job.

In the parallel job scheduling problem, a clairvoyant scheduler must obtain the characteristic of the job to be schedule by some means. It has been suggested that coarse estimates of job parameters may be obtainable by a static analysis of the code and the input [78], but these techniques often fail outside toy examples. Some papers suggest that the users themselves provides some estimates of job parameters [66, 54]. However, we have to contend with the problem that users will abuse the system by quoting fake values for the parameters. More recent work [90] proposed the prediction of run times from the run times of "similar" applications that have executed in the past. Even with the use of genetic algorithms to determine the "similarity" among applications and templates, the error in the best case still varies between 40 and 59 percent of mean application run times. Beyond that, prediction of running times is only one aspect of clairvoyance. We also have, for instance, the internal characteristics of each job and the corresponding arrival epochs. In function of these observations we will concentrate ourselves in non clairvoyant schedulers, as it seems to us more realistic than its clairvoyant counterpart, although a large part of the research in scheduling theory has been concentrated with clairvoyant scheduling.

3.1.5 Single Level vs. Two-Level

In single level scheduling the act of allocating a processing resource is combined with the act of deciding which task will use this resource. In single level scheduling, the operating system's kernel is the main agent in decisions related to scheduling, with no support for scheduling embedded into the application. An example of a single level scheduler is the scheduling of a SPMD workload by a Gang scheduler.

In two-level scheduling, the resource allocation and the decision about resource utilization are decoupled. The first level deals with resource allocation, and the second with its use. In two level scheduling, the operating system just allocates the computing resources, with the application itself (or the runtime system) being responsible for the actual fine-grain scheduling of tasks on the allocated PEs, in a way that satisfies the synchronization constraints. An example of application level scheduler is loop scheduling in shared memory systems [95, 68]. The problem with two-level scheduling is that it is less convenient for distributed memory architectures, especially if programs are written using the data-parallel programming model [34].

3.2 Resource Sharing on Parallel Job Scheduling

The sharing of a parallel machine, according to how the parallel job scheduling problem was defined, has both a spatial and a temporal dimension : The scheduler is not only responsible for deciding when a task will run, but also where it will run, i.e. in which processing element. That gives another dimension to scheduling in multiprocessors if compared with uniprocessors [34]. The temporal sharing of a machine is also known as time slicing or preemption ; and the space sharing as space slicing or partitioning. These two classifications are orthogonal, and may lead to a taxonomy based on the possible options. Table 3.1 shows the scheduling policies adopted by commercial and research systems, and was based on [34].

3.2.1 Space Slicing

In space slicing, also known as partitioning, a subset of processors of the machine is dedicated one job until completion. More than one job can be scheduled in the machine, provided that there is a sufficient number of processors available. If it is not the case, the arriving job should wait until the requested resources are available. A example such of algorithm is the variable partitioning algorithm [34]. This class of algorithms are specially important, since it is used by many MPP computers [48]. In this algorithm, each job J_i requires a number P_i of processors to execute. The scheduling algorithm is responsible to verify if a partition with P_i processors is available ; if yes, the partition is allocated to the job and it starts running immediately in the reserved partition until completion. If not, the job waits until a number of P_i processors becomes available. This wait time can be significant, delaying new arriving jobs. This delay can be bounded by above when the user is forced to define a maximum processing time for the job. However, this is a information that in most cases is not available at submission time, and killing a job before its completion is not an acceptable condition in most cases. This may lead the user to overestimate the required processing time, which makes the upper bound required not effective.

3.2.2 Time Slicing

In a time slicing only machine, all processors of the machine are dedicated to only one job. The advantage of this class of algorithms is that wait time is reduced for arriving jobs. Eventually all jobs will be scheduled if there is sufficient memory available. However, the waste due to fragmentation can be significant, since all processors of the machines will be dedicated to one job at a time, and in most cases the job will not require a number of processors equal to the number of processors of the machine. A Gang scheduling with no partitioning is an example of a time slicing only algorithm.

3.2.3 Combining Space and Time Slicing

By combining both time and space slicing it is possible to combine the advantages of both strategies, that is, the reduction of job wait time with an effective share of machine resources. This is the approach used, for instance, in Gang scheduling combined with partitioning. In this strategy, more than one job can be Gang scheduled at a time, if there is sufficient resources available. Due to its combined advantages, the combination of space and time slicing through Gang scheduling will be analyzed in more detail for the rest of this chapter.

3.3 Definition of Gang Scheduling

Gang scheduling is a scheduling strategy first proposed by John Ousterhout [72] that combines the following characteristics [34] :

- All tasks of a job are grouped into a Gang. Following Ousterhout original definition, all tasks belonging to a Gang are part of a process working set [72].
- The tasks in each Gang execute simultaneously on distinct PEs, using a one-to-one mapping. There are two variations here. In the first case each task is assigned to a processor and do not move. This version is currently the most popular one. In the second case migratable preemption is possible, where a group of tasks are preempted on a set of processors and resume on another, which may improve processor utilization and reduce fragmentation. An example of such scheme is described on [73].

<table>
<tr><td colspan="3" rowspan="4"></td><td colspan="4">time slicing</td></tr>
<tr><td colspan="3">yes</td><td rowspan="3">no</td></tr>
<tr><td colspan="2">independent PEs</td><td rowspan="2">gang scheduling</td></tr>
<tr><td>global queue</td><td>local queue</td></tr>
<tr><td rowspan="3">space slicing</td><td rowspan="2">yes</td><td>flexible</td><td>Mach</td><td>Paragon/service
Meiko/timeshare
KSR/interactive
transputers
Tera/streams
Chrysalis
TheHive (Beowulf)</td><td>Medusa
Butterfly@LLNL
Cray T3E
Meiko/gang
Paragon/gang
SGI/gang
Tera/PB
MAXI/gang</td><td>IBM SP2, Victor
Meiko/batch
Paragon/slice
KSR/bath
2-level/bottom
TRAC, MICROS
Amoeba
JMS on MPC</td></tr>
<tr><td>structured</td><td></td><td>NX/2 on iPSC/2
nCUBE</td><td>CM-5
Cedar
DHC on SP2
DQT on RWC-1</td><td>Cray T3D
CM-2
PASM
hypercubes</td></tr>
<tr><td colspan="2">no</td><td>IRIX on SGI
NYU Ultra
Dynix
2-level/top
Hydra/C.mmp</td><td>Star OS
Psyche
Elxsi
AP1000</td><td>MasPar MP2
Alliant FX/8
Chagori on K2</td><td>Illiac IV
MPP
GF11
Warp</td></tr>
</table>

FIG. 3.1: Scheduling policies followed in current commercial and research systems (Adapted from [34])

– Time slicing is used, with all the tasks in a Gang being preempted and rescheduled at the same time. Observe that space slicing can also been used along with time slicing through either predefined or dynamic partitioning.

In most cases, all the tasks in the job are considered to be a single Gang. Thus the number of tasks in the job conveys the PE requirements of the job.

In parallel job scheduling in general and gang scheduling in particular, as the number of processors is larger than one, the time utilization as well as the spatial utilization can be better visualized with the help of a bidimensional diagram dubbed *trace diagram*. The trace diagram was first introduced at chapter 2. One dimension represents processors while the other dimension represents time. Through the trace diagram it is possible to visualize the time utilization of the set of processors given a scheduling algorithm. One such diagram is illustrated in figure 3.2.

With the aid of the trace diagram, we should state some important definitions that will be useful in the following chapters of this thesis. Gang scheduling algorithms are preemptive algorithms. We will be particularly interested in gang scheduling algorithms which are *periodic and preemptive*. Related to periodic preemptive algorithms are the definitions of cycle, slice, period and slot. A *Workload change* occurs at the arrival of a new job, the termination of an existing one, or through the variation of the number of eligible tasks of a job to be scheduled. We define *cycle* as the time between workload changes. The *period* is the minimum interval of time where all jobs are scheduled at least once. A cycle/period is composed of *slices* ; a slice corresponds to a time slice in a partition that includes all processors of the machine. A *slot* is the processors' view of a slice. A Slice is composed of N slots, for a machine with N processors. If a processor has no assigned task during its slot in a slice, then we have an idle slot. The number of idle slots in a period divided by the total number of slots in that period defines the *Idling Ratio*.

Gang scheduling has several desirable properties. The most important ones are :

– Gang scheduling supports the abstraction of a dedicated machine for each job [48, 34].
– does not impose any restrictions on the programming model [48].
– Gang scheduling promotes efficient fine-grain interactions among the tasks in a Gang, based on the fact that they are executing simulta-

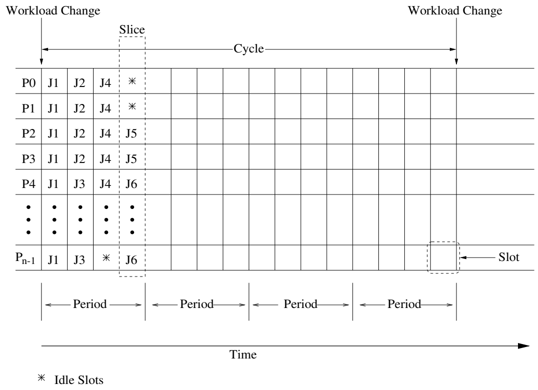

FIG. 3.2: Definition of slice, slot, period and cycle. J1 stands for job 1, J2 for job 2, etc. Job 2 is composed by 4 tasks

neously. Thus it is possible to use busy waiting for synchronization, without fear of waiting for a task that is not running [72, 37].
- hardware communication devices can be accessed directly in user mode without need for protection mechanisms [20].
- a one-to-one mapping also allows tasks to be associated with data structures in local memory [32].
- Gang scheduling provides better response times for short jobs, by virtue of using preemption [35]. Just as in uniprocessor systems, periodic preemption prevents long jobs from monopolizing system resources, and guarantees that every job in the system will execute within a relatively short time.
- Performance is reduced gradually as load increases [35].
- Gang scheduling allows guarantees about the performance to be made [34]. This is so because applications execute in an environment that is essentially the same as a dedicated machine, except for some additional overheads. This characteristic will be exploited in the next chapter, where we make a competitive analysis of Gang scheduling.

Several studies that compared Gang scheduling with other scheduling schemes have concluded that Gang scheduling is a relatively good policy [64, 62, 44, 29, 32]. This is reflected by the fact that Gang scheduling is implemented in a number of commercial platforms, such as Connection Machine CM-5 [22], Intel Paragon [14], Cray T3E [57], Silicon Graphics multiprocessor workstations [5] and other platforms as shown on table 3.1.

However, Gang scheduling has also disadvantages, which are listed below :

- Overhead due to job/machine preemption [20]. It may also have an impact on cache performance. To compensate the overhead of job/machine-wide preemption, typical gang schedulers implement large time slices, in the order of hundreds of milliseconds [20, 45].
- Performance of I/O bound and interactive jobs. In [58] Lee et al. showed that I/O bound jobs suffer under Gang scheduling due to CPU fragmentation. Interactive jobs are also a concern due to the I/O characteristics of these jobs and also because of the long time slices that are normally used on Gang scheduling.
- The preemption in all processors at the same time or in a large subset of them may raise scalability concerns [31]. This specific problem may be solved by using a Distributed Hierarchical Control scheme [36] for scheduling Gangs.

3.4 Implementing Gang Scheduling

There is a number of questions that should be considered when implementing Gangs scheduling on actual machines. The main points are [34] :

- *Multi context switch implementation* Multi context switch implementation is fundamental for Gang scheduling since the preemption of a job as a whole relies on an efficient implementation of the multi context switch mechanism.
- *Saving Job state during global context switches* For some networks, a network preemption may also be necessary together with a job preemption/task in order to avoid the situation where messages are delivered to wrong tasks.
- *Memory and swap considerations* The main limitation for allocation a large number of jobs at the same time in parallel machines is memory. Scientific applications normally require a large amount of memory, which in most cases limits the number of jobs allocated in memory at the same time.
- *Partitioning* One fundamental point is to have a machine partition policy that maximizes the utilization of the machine. Partition can be static or dynamic, that is, there are implementations that impose fixed

partitions and others where the partitions used depend on the size of jobs.

3.4.1 Implementing Multi Context switch

The synchronization of the context-switch operation is typically handled by a central controller. The controller may be explicitly defined as in [83, 85], but this is not mandatory : a floating controller can be used, where any PE that notices a certain condition (e.g. all tasks are blocked) induces the next multicontext-switch [37, 76]. A variant of this is used in IRIX on SGI multiprocessor workstations : the PE that selects the first member of a Gang from the global queue interrupts other PEs that are running low-priority processes so that they will schedule the other Gang members [5].

The controller coordinates the context switching by causing an operating system trap on all the relevant processors. The requirement on this trap is that the variability in the exact time that it occurs on the different PEs be small relative to the scheduling time quantum. Possible implementations are [34] :

- The use special hardware, as in the K2 architecture [91]. K2 is a distributed memory parallel processor interconnected by a bidimensional torus. One interesting characteristic of the K2 is the support for distributed virtual memory. The K2 implements a global interrupt driven synchronization mechanism called *torus synchronization unit*, that is used to perform torus wide context switches.
- The use a software broadcast interrupt [37].
- By using synchronized clocks, that all cause interrupts on their respective PEs at the same time [41, 83, 85]. Once the processors are interrupted, they perform their local context switch. A number of Unix-based implementations are described in the literature, which use signals [20] or change the priorities of the tasks concerned [57].

3.4.2 Saving Job state

Saving the program state on a single PE typically involves no more than saving the CPU register values. In a parallel machine, this has to be done on all the PEs. However, the program might have additional state that is neither in memory nor in the registers, but in transit from one place to another. In many cases, for systems supporting multi-context switch it is necessary to save such communication state together with the computation state [63]. The following discussion uses message-passing terminology, because such architectures are more susceptible to this problem. The problem with messages in transit during a context switch occurs when they arrive at the destination node. There are three main approaches to this problem :

- The first approach is to simply drop such messages, and re-send them the next time that the application is scheduled to run ; it is used in the SHARE scheduler for the SP2 [41]. This approach has the advantage of not requiring any hardware support, so it can be implemented on any machine.
- The second approach is to tag messages with a job ID. When an arriving message does not belong to the currently running job, it is handled anyway. This approach is used on the Meiko CS-2 [4] and the Intel Paragon [74].
- The third approach is to drain the network as part of the context switch operation [20], that is, to do a network preemption along with a job preemption. While this increases the overhead for context switching, it provides each job with exclusive access to a clean network. This facilitates the implementation of efficient user level communication. Security is provided by mapping the communication devices into user space, and using existing hardware protection mechanisms. This approach is used in the the K2 [91], the Connection Machine CM-5 [22], and the RWC-1 [19].

Besides these three solutions, two more solutions are possible when considering different network technologies :

- The first one is when the network support remote write primitives, with each packet being associated with a physical memory address in the remote node, such as in the MPC machine [18]. In this case the packets are written in a non-swappable physical memory address range which is associated with the virtual address space of some predefined process. Therefore there is no risk of a wrong process receiving a message due to coordinated context switch, which is a important advantage when implementing Gang scheduling based strategies.
- The second one is to maintain multiple virtual circuits between PEs, and use a separate one for each job. In this case, the messages can be

left in the network until the job is rescheduled and receives them. This approach is taken in the starT-NG [23] and starT-voyager [1] machines from MIT.

3.4.3 Memory and swap considerations

As large scale parallel applications often require large amounts of memory, it is sometimes not possible to have multiple jobs memory resident. In such cases, switching among jobs implies swapping them to secondary storage [35]. This can be classified as Gang scheduling because either all the tasks execute or none do. However, the additional overhead of swapping implies that this cannot be done on an interactive time scale. The use of swapping and checkpointing together leads to the concept of "migratable preemption" [73]. By using both checkpointing and swapping it is possible to restart the job on a different set of PEs than the one used originally, which can reduce fragmentation [57, 80].

Observe that in some cases is possible to have both fine-grain Gang scheduling among the memory-resident jobs, and coarse grain Gang scheduling by means of swapping. A variant of this approach is used on the Tera Multi-Threaded Architecture, where threads are loaded into separate hardware contexts that are switched on each cycle [26].

3.4.4 Partitioning in Gang Scheduling

As already stated, at a given moment of time more than one job may be sharing the machine, which leads to the concept of partitioning. In Gang scheduling there is two different ways of doing the partition of the machine. The first is to use predefined partitions independent of job size. The second is to use dynamic partitioning, which the slicing of the machine may vary in functions of the sizes of jobs scheduled at a given time. Both strategies are described in following paragraphs.

Gang Scheduling within Predefined Partitions

The simple approach is to first partition the machine into one or more fixed sets of disjoint PEs, and then perform Gang scheduling within each partition independently of the others. Actually, partitioning is not strictly necessary, as it is possible to simply schedule all the PEs as one unit. This approach may lead to severe resource underutilization in massively parallel processors. Therefore it may not be suitable for large systems. A Gang scheduler with predefined partitions was implemented in the connection machine CM-5 [22].

Gang Scheduling with Dynamic Repartitioning

Using fixed partitions runs the risk of significant underutilization of resources due to fragmentation. If all Gangs are not of the same size, it is therefore desirable to change the partitioning at each multi-context-switch. This implies that context switching must be coordinated across groups of PEs, and not only within groups. The problem with this approach is the difficulty of doing the partitioning on the fly. The solution is to look for a suitable partitioning only when the load changes, not at each context switch. When a new application arrives or an old one terminates, applications are matched together so as to utilize as many PEs as possible. Then at each context switch the next set of matching applications is scheduled. Two possibilities are the use of a global synchronizer and subsets of processors that switch context independendly.

Synchronous switching across the whole machine The most common approach to the implementation of dynamic repartitioning is to perform the context switching synchronously across the whole machine. This is done regardless of how the partitioning is supposed to change during the context switching operation. PEs in all the different groups always switch simultaneously, so moving a PE from one group to another during a switch is no problem.

The scheduling algorithms developed by Ousterhout fall into this category [72]. The simplest is the matrix algorithm, which uses a diagram similar to the trace diagram defined previously in this section. The matrix algorithm was implemented in the Medusa operating system on CMΛ [71], in the Meiko CS-2 operating system, in the Gang-scheduler used for the BBN Butterfly at Lawrence Livermore National Lab [42]. In the matrix algorithm, a bidimensional matrix represents the occupation of multiple processors over time. It is used to define when and on which processors a job will run. The time axis is divided in slices, and at the end of each slice a context switching is performed across the whole machine.

Subsets switch independently A global synchronizer is necessary to synchronize the context switching in different groups of PEs if PEs need to move from one group to the other as part of the context switch. But this is required only if one of the groups must grow. There is no need to synchronize if the groups only split into smaller groups. This observation is used in the design of the "Distributed Hierarchical Control" (DHC) scheme [36, 39].

The Distributed Hierarchical Control algorithm partitions the PEs using a buddy system arrangement by successively dividing them into halves. A separate (logical) controller is associated with each partition (see figure 3.3). The size of each partition is a power of 2, and the union of two partitions half the size. This also defines the hierarchy. A controller at level i coordinates activities involving more than half of the 2^i PEs spanned by its subtree. Controllers in low levels of hierarchy provide for local control while those in higher levels are responsible for global coordination. There are also lateral connections among controllers that are used for load balancing.

It is worth noting that the hierarchy describes the logical control structure used by the operating system, and only suggests but does not imply a hardware hierarchy. This hierarchy is used to map tasks to processors as follows : a request to map a new Gang of size S originates from a task executing on some processor. The request ascends the tree of controllers until it reaches the appropriate level for its size, and then moves across to some controller that will balance the overall load.

The scheduling proceeds in waves that propagate down the tree of controllers. The scheduling is carried out in cycles. Each cycle starts with the highest level controller, which includes all PEs, and it will executes all Gang that require more than half of the PEs. After all such Gangs have been scheduled, the PEs are splited into two groups, each group associated with a separated controller. These controllers do not need to synchronize with each other, and context switches in the two groups of PEs are independent. The splitting continues as smaller Gangs are scheduled by lower level controllers, and at the end of the cycle all processors are reunited again on only one partition and the next cycle starts from the top controller.

An important optimization in DHC algorithm is *selective disabling* : if a given controller does not use all available processors, some subordinate controllers are left active and may use the leftover processors to schedule smaller Gangs. Another possible optimization is used when the Gang is smaller than the full group, and consists of splitting the Gang in two and completely utilize one half, leaving a large unused group in the other half that can be used

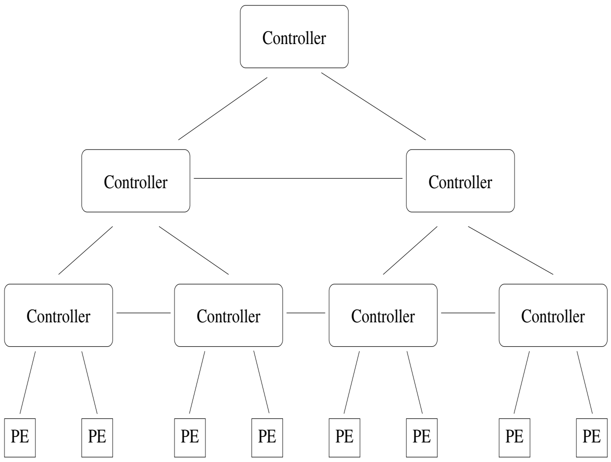

FIG. 3.3: Distributed Hierarchical Control scheme

to schedule another jobs via alternative scheduling. It was shown in [33] that these strategies combined provides better packing, which implies better processor utilization, than other on-line algorithms such as best fit and first fit.

The importance of removing extra synchronization, as is proposed in the DHC algorithm, is that it decouples groups of PEs with different loads. This allows the time slices to be set differently on different groups of PEs, so as to optimize the execution of different Gangs. It also improves the scalability of the system, by removing any components that require full knowledge about the system state. With the Distributed Hierarchical Control scheme, each controller only needs knowledge about the largest Gangs mapped to its group of PEs.

Gang scheduling systems based on the ideas of DHC were implemented for the IBM SP2 [41] and the RWC-1 [19].

Lazy Gang scheduling Taking the idea of independent switching to the extreme leads to the notion of lazy Gang scheduling. In this algorithm, each job has a maximal wait time associated with it, based on its class : interactive and debug jobs have short wait times, while batch jobs may wait a very long time. Each time a job's wait time is exceeded, its priority rises, and the lowest priority jobs in the system are preempted to make space for it. The scheduled job then runs for a certain time, which is proportional to its memory usage. After this period, it itself becomes a candidate for preemption if another high-priority job is waiting. This style of Gang scheduling is used on the Cray T3D at LLNL [35]. A variant based on feedback has also been proposed [80].

3.5 Variations of Gang Scheduling

Two strategies that are similar to Gang scheduling are coscheduling and family scheduling [34]. These variations will be used by us in following chapters in order to improve the performance of standard Gang scheduling. Coscheduling and family scheduling are detailed below :

3.5.1 Coscheduling

Coscheduling was also originally defined by Ousterhout to describe systems where the operating system attempts to schedule a set of tasks simultaneously on distinct PEs, as in Gang scheduling, but if it is not possible only a subset of the tasks are scheduled [72].

Coscheduling is a more flexible scheme than Gang scheduling, since it allows the scheduling of subsets of tasks from the rest of the Gang. Coscheduling can be highly beneficial if the job's tasks are highly independent, as in embarrassingly parallel jobs and I/O bound jobs. Task belonging to these jobs can make progress even if the whole Gang is not scheduled. However, if tasks synchronize with each other at fine granularity, there no advantage on scheduling only a subset of the Gang [58, 29]. So the performance advantages of coscheduling are related with the characteristics of a job.

However it is possible based on runtime observation of the running tasks to detect which tasks interact at fine fine granularity and then should be scheduled together. Examples of this strategy are found on [38, 31].

3.5.2 Family Scheduling

Family scheduling is a variant of Gang scheduling where the number of tasks is allowed to be larger than the number of PEs. Thus, the operating system is involved in two levels of time slicing : first, there is the coordinated scheduling of the job as a whole across a set of PEs, and then there is the internal scheduling of the job's tasks on these PEs [8]. This can be done using a global queue or using local queues. The difference from two-level scheduling schemes is that the whole job may be preempted (two-level scheduling typically employs non-preemptive partitioning at the job level), and both levels are done by the operating system rather than leaving the second one for the application runtime system.

Family scheduling can be used, for instance, to handle jobs that present irregularities in the number of active tasks during execution. We will develop this approach further in the following chapters.

3.6 Conclusion

In this chapter, a classification of the job scheduling problem was made along with a detailed analysis of Gang scheduling, its variations, advantages and disadvantages. As a consequence of that analysis , we do believe that Gang scheduling is a good scheduling strategy, serving as a starting point for more sophisticated scheduling strategies that are presented in the following chapters.

Chapitre 4

Bounds on Gang Scheduling

4.1 Introduction

We present in this chapter a competitive analysis of a class of gang scheduling algorithms. Gang scheduling are preemptive algorithms where a parallel task is scheduled and preempted in a set of processors in a regular basis. Gang scheduling was first proposed by Ousterhout [72]. Reasons to consider gang scheduling are responsiveness [35], efficient sharing of resources [48] and ease of programming. In gang scheduling the threads of a task are supplied with an environment that is very similar to a dedicated machine [48]. It is useful to any model of computation and any programming style. The use of time slicing allows performance to degrade gradually as load increases. Applications with fine-grain interactions benefit of large performance improvements over uncoordinated scheduling[37].

Consider a set of m parallel tasks $\mathcal{W} = \{T_0, T_1, ..., T_{m-1}\}$ and a set of n identical processors $\mathcal{P} = \{1, ..., n\}$. Associated with each task T_j is a function $t_j(\beta_j) > 0$ defining the task execution time as a function of the number of processors $0 < \beta_j \leq n$ alloted to the task. This problem is known as malleable parallel task scheduling (MPTS). This definition of MPTS differs from [47] as preemption is allowed. The restricted version of MPTS which the processors allotments are know a priori is known as non-malleable parallel task scheduling (NPTS). Generally MPTS can be divided into two subproblems [28] :

- Allotment - Select for any task T_i a number of processors β_i following a predefined criteria

– Scheduling - Apply a scheduling algorithm for the resulting non-malleable instance.

MPTS takes communications into account implicitly by the function representing the parallel execution time with the penalty due to the management of the parallelism[70]. In non-preemptive scheduling, an approximation of guarantee λ for the non-malleable problem on the allotment of the optimal solution provides the same guarantee for the malleable problem. Similarly, the results of this chapter are valid for both MPTS and NPTS problems.

In this chapter we consider that all m tasks of the workload are available at time zero and have finite execution times. We define a workload change as being the completion of a task. All tasks are independent. The objective function (metric) is the overall makespan (also known as schedule length). Given the completion times $C = \{c_o, ..., c_{m-1}\}$ under an algorithm A, the overall makespan is defined as :

$$T_C^A = \max\{c_j : j = 0, ..., m-1\} \tag{4.1}$$

The gang scheduling algorithm analyzed reallocates all remaining parallel tasks at each task completion and has no information about execution time of individual parallel tasks. The dynamic behavior of the algorithm and the limited amount of information available to it are the main reasons that justify the utilization of competitive analysis in this chapter. Competitive analysis is a formal way of evaluating algorithms that are limited in some way (e.g., limited information, computational power, number of preemptions) [17], what is indeed our case. This measure was first introduced in the study of a memory management system [67, 87].

For the definition of the competitive ratio, consider a workload W. We define $t_{OPT}^{over}(W)$ as the overall makespan of workload W under the optimal off-line algorithm. Let $t_A^{over}(W)$ be the overall makespan under an algorithm A. Algorithm A is said to be *r-competitive* if for all workloads W, $t_A^{over}(W) < r.t_{OPT}^{over}(W)$. The competitive ratio of algorithm A is r.

The optimal off-line algorithm is often referred also as an adversary which plays against an arbitrary algorithm and defines an input which forces it to incur a high cost, while at the same time the adversary itself can service the same sequence at low cost. Since gang scheduling are preemptive algorithms, in our case the optimal algorithm (adversary) is also preemptive. The adversary technique is used to prove lower bounds in competitive ratio analysis [46].

The main result of this chapter is to propose a scheduling strategy based on gang scheduling which the competitive ratio is equal to two. This result has two implications. First, the ratio found is independent of both the number of processors and of the number of parallel tasks of the workload. Second, to the best of our knowledge it is the first time that a competitive analysis of a gang scheduling algorithm using the overall makespan as metric is made.

This chapter is organized as follows : section 4.2 describes previous theoretical work on preemptive scheduling. The execution mode of parallel tasks under gang scheduling is detailed in section 4.3. Section 4.4 describes the algorithm analyzed in this chapter. The competitive analysis of gang scheduling is in section 4.5 and section 4.6 contains our final remarks.

4.2 Previous work on Theoretical Results on Preemptive Scheduling

The complexity of off-line algorithms for multiprocessor scheduling has been subject of research for many years [50, 65, 93]. Non-preemptive scheduling in the general case is NP-complete, including the single processor case [15]. However preemptive scheduling, of which Gang scheduling is an example, admits polynomial time solutions [6, 15]. The algorithm proposed in [6] is based on the fact that the number of processors is necessarily finite, and therefore there is only a finite number of ways in which the processors can be divided among different tasks executing simultaneously. The whole set of tasks can be scheduled by using different partitioning schemes for various processor requirements, in a way that the total cumulative time of each partition size satisfies the execution time requirements of tasks with that processor requirement. This leads to a linear programming formulation, where the objective function is to minimize the sum of the times of the different partitioning schemes used. This linear programming problem may be solved using Khachian's algorithm [53] in time bounded from above by a polynomial in the number of variables, the number of constraints, and the sum of logarithms of all the coefficients in the LP problem.

The performance implications of Gang scheduling have been subject of less work. Subramanian and Scherson [92, 79] have made a competitive analysis of Gang scheduling using a new metric dubbed happiness. They proved that a variation of Gang scheduling named instruction balanced time slice is the best possible algorithm under this new metric when considering a workload composed of V-RAM [7] tasks. The ratio found is the maximum inefficiency of a given workload of V-RAM tasks, and by consequence it depends on the characteristics of the workload being considered. The analysis presented in this chapter differs from theirs in the metric and the partitioning strategy used.

4.3 Executing Tasks Under Gang scheduling

In parallel task scheduling in general, and in gang scheduling in particular, as the number of processors is greater than one, the time utilization as well as the spatial utilization can be better visualized with the help of a two dimensional diagram that we call *trace diagram* (this diagram is also known in the literature as Ousterhout matrix [72], as well as Gantt chart). One dimension represents processors while the other dimension represents time. One such diagram is illustrated in figure 4.1.

Gang scheduling algorithms are preemptive algorithms. We will be particularly interested in gang service algorithms which are *periodic and preemptive*. The reason for considering periodicity is because it was proved in [84] that periodic gang scheduling can always achieve equal or better utilization when compared with aperiodic gang scheduling. Related to periodic preemptive algorithms are the definitions of cycle, slice, period and slot. Since in this chapter is considered that all tasks of a workload are available at time zero, a *Workload change* occurs at the the completion of an existing parallel task. We define *cycle* as the time between workload changes. The *period* is the minimum interval of time where all tasks are scheduled at least once. A cycle/period is composed of *slices* ; a slice corresponds to a time slice in a partition that includes all processors of the machine. A *slot* is the processors' view of a slice. A Slice is composed of N slots, for a machine with N processors. If a processor has no assigned task during its slot in a slice, then we have an idle slot.

In Gang scheduling, the task's perspective is similar to that of a dedicated machine during the slices of its execution. Some reduction in I/O bandwidth may be experienced due to interference from other tasks, but CPU and memory resources should be dedicated [48]. We will use this analogy in the following sense : given a task J_1 that requires P_1 processors and takes T_1 units of time to complete if it runs on a dedicated machine with P_1 processors,

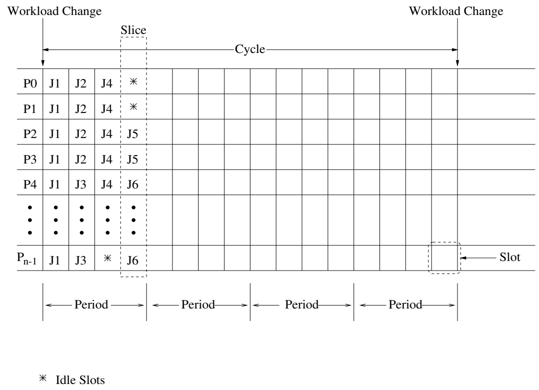

FIG. 4.1: Trace diagram. The figure illustrates a gang scheduling period of 4 slices containing 6 tasks. In this figure the tasks are identified as J1 through J6.

this tasks will complete under Gang scheduling after it is scheduled for a number of slices that add up to T_1. Eventual boundary effects are considered not significant for the competitive analysis.

When a task or a set of tasks are executed on a parallel machine under Gang scheduling, we can divide the utilization of each processors of the machine into two main areas :

- $T_{i,B}^{comp}$ - The time corresponding to number of slots where processor i is allocated for task B.
- S_i^{space} - Corresponds to the time associated with the number of slots where processor i has no task assigned to it due to the following causes :
  - Insufficient number of parallel tasks.
  - Non-optimality of the space sharing strategy used.

We define the *processor-time product* PT_B^C of a task T_B as the number of processors required by T_B (β_B) multiplied by time t_B^C necessary to complete the task T_B on a dedicated machine. Observe that this corresponds to the area $\beta_B \times t_B^C$ in the trace diagram.

$$PT_B^C = \sum_{i=1}^{\beta_B} T_{i,B}^{comp} \tag{4.2}$$

Equation 4.2 is related to one task only. When considering a workload $\mathcal{W}$ with m tasks gang-scheduled on a machine, equation 4.2 becomes :

$$\begin{aligned} PT^C &= \sum_{i=1}^{m} PT_i^C + PT^{space} \\ &= PT^{total} + PT^{space} \end{aligned} \tag{4.3}$$

Where

$$PT^{total} = \sum_{i=1}^{m} PT_i^C \tag{4.4}$$

and

$$PT^{space} = \sum_{i=1}^{n} S_i^{space} \tag{4.5}$$

Where n is the number of processors of the machine, PT^C is the number of processors in the machine multiplied by the overall makespan of workload W, PT^{total} is a sum over equation 4.2 for all tasks that compose the workload W and PT^{space} is the waste due to the packing strategy chosen.

4.3.1 Area Conservation

In this section we make some remarks that will be useful for the rest of this chapter. Given the equation :

$$PT^C = PT^{total} + PT^{space} \tag{4.6}$$

If we divide the equation by n :

$$\frac{PT^C}{n} = \frac{PT^{total}}{n} + \frac{PT^{space}}{n} \tag{4.7}$$

We obtain :

$$t^{over} = t^{total} + t^{space} \tag{4.8}$$

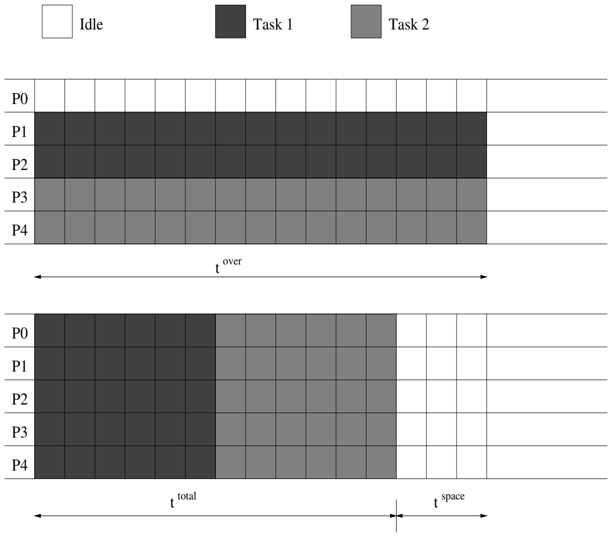

FIG. 4.2: Area Conservation. In the figure two parallel tasks are scheduled. The overall makespan is equal to the sum of t^{total} and t^{space}

Where t^{over} is the overall makespan of workload $\mathcal{W}$. From equation 4.8 we can verify that the area sums PT^{total} and PT^{space} with an overall makespan t^{over} can be converted into two contiguous areas nt^{total} and nt^{space}. The sum of both areas are equal to $nt^{over} = PT^C$. This process is illustrated in figure 4.2.

4.3.2 Partitioning in Gang Scheduling

When considering partitioning the machine among tasks, there are two main possibilities :

- Simple Gang Scheduling : In this case no partitioning is possible and the machine is dedicated to one task at a time, regardless of processor utilization of individual tasks. Observe that in this case, the overall makespan ratio between an optimal schedule and the simple gang scheduler is a function of the workload and/or the number of processors. As an example, consider a workload composed of tasks that are alloted to one processor with a associated duration of 1 time unit. The optimal overall makespan for this workload is $\lceil \frac{m}{n} \rceil$, the the overall makespan for the simple gang scheduler is m, the number of tasks in the workload.

– Concurrent Gang Scheduling : In this case, the machine can be shared by multiple tasks concurrently, by using either static or dynamic partitions. Dynamic partitioning[34] will be considered for the rest of this chapter.

4.4 Algorithm description

The algorithm analyzed in this chapter is a gang scheduler that permits the sharing of the machine among multiple tasks through the use of multiple partitions created dynamically. At time zero, all m tasks are sorted by processor count in a non increasing order. Then all tasks are allocated on the machine using a first fit allocation strategy. This packing strategy is known in the literature as first fit decreasing(FFD). By the end of the allocation, there will be a trace diagram which will indicate the temporal and spatial distribution of tasks.

When a task completes, the same process of sorting and allocation repeats itself. The remaining tasks are again sorted by processor count, and then allocated in the trace diagram. The following steps are executed for each workload change :

1 - Update Eligible task list
2 - Sort tasks by processor count in a non-increasing order
3 - Allocate processors for the parallel task
in the head of the queue
4 - While there is parallel tasks remaining
        Allocate all remaining parallel tasks
        using First Fit as partitioning strategy
5 - Run

4.5 Competitive Analysis

In this section we find a tight bound for the competitive ratio for the gang scheduling algorithm described in the previous section.

Observe that in the following analysis all parallel tasks are available at time zero and migration costs are not considered for both the gang scheduler and the clairvoyant adversary.

The competitive ratio for the algorithm described in section 4.4 is stated in theorem 4.

Theorem 4 *The CR of Gang scheduling using first fit decreasing under overall makespan is 2.*

Proof 1 Lower Bound - *In order to find the lower bound we will use the* adversary *technique. This technique employs an adversary which plays against an arbitrary algorithm and concocts an input which forces it to incur a high cost [46]. The objective of the optimal adversary when creating a workload for competitive analysis is to minimize its overall makespan while at the same time maximizing the overall makespan of the algorithm under analysis. In order to maximize the overall makespan of the gang scheduling algorithm, the adversary creates a workload that will avoid any sharing of the machine among different tasks at the same slice by the Gang scheduler. By doing so the adversary is at the same time minimizing the utilization of the machine and maximizing the completion time of the gang scheduling algorithm, by making the overall makespan a sum of individual completion times of tasks. One workload that has these characteristics is as follows :*

- *The adversary allots n/2+1 processors to all m tasks in order to avoid any task sharing by a Gang scheduler. In this case the overall makespan of the gang scheduler will be the sum of the completion times* $t_i(n/2+1), 0 \leq i \leq m-1$ *of individual tasks.*
- *All tasks are equal, and the tasks are defined in a way that the area* $\beta_i \times t_i(\beta_i)$ *for each task i is minimized when a number of processors* $\beta_i = n/2+1$ *is chosen.*

The reason for considering a workload where all tasks have $n/2+1$ *processors is to avoid any sharing of the machine among diferent tasks at the same slice by a Gang scheduler, even ones implementing a partitioning strategy such as first fit decreasing. As a consequence, the utilization of the gang scheduler is reduced, since there will be at least n/2-1 processors idle at any time. The total makespan will be the sum of the completion time of individual tasks.*

For the optimal scheduler, we claim that in the best case :

$$PT_{opt}^{space} = 0 \tag{4.9}$$

Where $PT_{opt}^{space} = 0$ is the total area in the trace diagram where a processor is idle due to insufficient number of tasks and/or non-optimality of the partitioning strategy. As a consequence we have :

$$PT_{opt}^{C} = \sum_{i=1}^{m} PT_{i}^{C} \tag{4.10}$$

To prove that equation 4.9 is valid we need to produce at least one instance of the optimal scheduling where the amount of computation is perfectly balanced and there is no idle time. First we claim that is possible to define a placement where each processor is allocated to the same number of tasks. This can be done for a combination of n and m where $m \times (n/2+1)$ is a multiple of n. Second, since the processor has full control of the characteristics of the workload, it is capable of define a scheduling without increasing the area $\beta_i \times t_i(\beta_i)$ required for any task i. One example of workload where for maximum utilization is achieved in this case is one composed only by embarrassingly parallel jobs. The scheduling for this workload would be a simple one, with each processor dedicating the same amount of time to each task allocated to it, without the need of coordinated scheduling among processors. Observe that other examples of workload/scheduling with 100% utilization can also be found. The overall makespan of the adversary in the best case will then be equal to the computing area of all tasks divided by the number of processors. Therefore we may consider that in the best case the optimal scheduler has 100% utilization and equation 4.10 holds.

Consider a workload W composed by m tasks, $m > 1$. For an algorithm A we have :

$$PT_{A}^{C} = PT_{A}^{total} + PT_{A}^{space} \tag{4.11}$$

Where PT_{A}^{C} is the area corresponding to the overall makespan of workload W multiplied by the number of processors, as stated in subsection 4.3.1. PT_{A}^{total} is the total computation related with workload W, and PT_{A}^{space} is the idle time due to a non ideal partition of the machine among the different tasks.

With Gang scheduling we have :

$$PT_{Gang}^{C} = \sum_{i=1}^{m} PT_{i}^{C} + PT_{gang}^{space} \tag{4.12}$$

We have already seen in the beginning of this proof that $PT^{total}_{gang} > PT^{space}_{gang}$, as all tasks require n/2+1 processors. However, it is possible to make PT^{space}_{gang} and near as PT^{total}_{gang} as we want. As an example, consider the workload composed only of tasks requiring n/2+1 processors. Clearly in this case is not possible to space share the machine among diferent tasks at a given time as we saw before. As n becomes large, n/2+1 approaches n/2-1, making the difference between PT^{total}_{gang} and PT^{space}_{gang} negligible. Then, for the worst case analysis, we will consider that $PT^{total}_{gang} = PT^{space}_{gang}$. We then have in the worst case for the Gang scheduler :

$$PT^C_{Gang} = 2 \times \sum_{i=1}^{m} PT^C_i \tag{4.13}$$

Dividing equation 4.13 by equation 4.10 we have :

$$\frac{PT^C_{Gang}}{PT^C_{opt}} = \frac{t^{over}_{Gang}}{t^{over}_{opt}} = 2 \times \frac{\sum_{i=1}^{m} PT^C_i}{\sum_{i=1}^{m} PT^C_i} \tag{4.14}$$

$$\frac{t^{over}_{Gang}}{t^{over}_{opt}} = 2 \tag{4.15}$$

Upper Bound - *Our objective is to prove that the workload considered in the lower bound proof is a worst case workload for gang scheduling with first fit decreasing. Similar to equation 4.11 the area corresponding to PT^C_{Gang} can be divided in two different areas :*

$$PT^C_{Gang} = PT^{total}_{Gang} + PT^{space}_{Gang} \tag{4.16}$$

Now let us shift our attention to the schedule obtained by the optimal scheduler. As in the previous case, there are two main areas :

$$PT^C_{opt} = PT^{total}_{opt} + PT^{space}_{opt} \tag{4.17}$$

With $PT^C_{gang} \geq PT^C_{opt}$. We now investigate the variation between the corresponding areas in the two algorithms.

- PT^{total} *- There is no variation in this area for both optimal and Gang scheduling, that is, $PT^{total}_{opt} = PT^{total}_{Gang}$.*
- PT^{space} *- The difference of PT^{space} between a Gang scheduler and an optimal scheduler may be unbounded, and it depends on the packing strategy used in the Gang scheduling algorithm. Observe that ,depending*

on the workload, we may have $PT^{space}_{opt} > 0$. *In particular, the packing strategy FFD used in the algorithm analyzed in this chapter is optimal if the required number of processors for all tasks divide each other, e.g. if they are powers of two [11]. It can be also proved that the number of bins used by first fit decreasing is asymptotically bounded by about 22 percent of the optimal number [3]. Two cases are possible :*

$PT^{total}_{Gang} \geq PT^{space}_{Gang}$. *Here we have the same case of the one described in the lower bound section. In the best case for the optimal scheduler* $PT^{space}_{opt} = 0$. *The worst case for the Gang scheduler will happen when no space sharing is possible at all. Otherwise the packing strategy will reallocate tasks at each workload change, always sharing the machine among tasks at a given time when possible. Then CR = 2 apply.*

$PT^{total}_{Gang} < PT^{space}_{Gang}$. *As we are using a packing strategy that re-maps all tasks at each workload change, and has a behavior optimal for some cases, the only case where* $PT^{space}_{Gang} > PT^{total}_{Gang}$ *is when the Gang schedule reduces itself to variable partitioning (i.e. the period has only one slice) after a time* $t_k \geq 0$ *due to insufficient number of tasks, and the processor utilization is smaller than n/2 for a large amount of time. In this case task durations are different, and the task* T_{max} *with maximum duration is one that have* $|\beta_{max}| < n/2$, *otherwise the assumption* $PT^{total}_{Gang} < PT^{space}_{Gang}$ *does not hold. For both the Gang and the optimal schedulers, the minimum feasible overall makespan will be* t_{max}, *which is the completion time on a dedicated machine of the task* T_{max}. t_{max} *is the maximum completion time over all tasks of the workload. In the following analysis we consider the best case for the optimal scheduler, that is,* t_{max} *as being the overall makespan for the optimal scheduler.*

We can divide the slices corresponding to a scheduling generated by the algorithm described in section 4.4 where $PT^{total}_{Gang} < PT^{space}_{Gang}$ *in two parts : a set of slices with processor utilization* $< n/2$ *and another set with processor utilization* $\geq n/2$. *One conclusion can be derived from this scheduling :*

- *The time associated with the maximum number of slots of utilization* $< n/2$ *is equivalent to* t_{max}.

Otherwise T_{max} *could have been scheduled in one of the extra slices where the processor utilization is* $< n/2$ *considering the partitioning strategy used, since the duration of* T_{max} *is maximum among all tasks.*

As a consequence, in the worst case for the gang scheduler, the number of slices where the processor utilization is $< n/2$ *corresponds a time amount equivalent to* t_{max}. *By other side, the maximum number of slots of processor utilization* $> n/2$ *is also equal to* t_{max}. *Each slice with processor utilization* $> n/2$ *must have at most* $\beta_{max} - 1$ *idle processors, otherwise* T_{max} *could have been scheduled at that slice considering the partitioning strategy employed. Under these conditions, if the number of slices where the processor utilization is* $> n/2$ *corresponds to a period of time greater than* t_{max}, *the assumption* $PT^{total}_{Gang} < PT^{space}_{Gang}$ *does not hold. Concluding, the number of slices with processor utilization* $> n/2$ *must be lower than or equal to* t_{max}. *In the worst case for the gang scheduler, we have for the overall makespan ratio :*

$$\frac{PT^C_{gang}}{PT^C_{opt}} < \frac{2 \times t_{max}}{t_{max}} = 2 \tag{4.18}$$

We finally conclude that CR=2 also apply in this case

4.6 Conclusion and Future Work

Is this chapter is proposed and analyzed a gang scheduling based algorithm which has a competitive ratio of two for a workload composed of m parallel tasks available at time zero. The fact that the algorithm reallocates all parallel tasks at each task completion makes the ratio found independent of both number of processors and number of parallel tasks.

However, the frequent reallocation of parallel tasks also makes an actual implementation of the algorithm very sensitive to migration costs. The analysis made in this chapter considered and "ideal" gang scheduler, where task are reallocated at each workload change and migration costs are not considered. A natural sequence for the work presented on this chapter is to relax these assumptions, and consider limited reallocation of tasks and migration costs.

Chapitre 5

Resource Management in Gang Scheduling

5.1 Introduction

In this chapter we present one and multi-dimensional analysis of the resource sharing problem for gang scheduling on the average case. To do so, a new methodology for average case analysis of scheduling algorithms, dubbed Dynamic Competitive Ratio, is proposed in this chapter.

In the one dimensional resource sharing problem, jobs are represented by only on parameter : the number of tasks that compose the job. The machine is characterized by the number of processors. We suppose that other resources, such as memory, have infinite availability.

In the multi-dimensional case the resources available in a machine are represented by a m-dimensional vector $R = (R_1, R_2, ...R_m)$ and the resources required by a job J are represented by a k-dimensional vector $J = (J_1, J_2, ...J_k)$.

In this chapter we will be concerned with dynamic scheduling, that is, scheduling when arrival times can be different from zero. In dynamic scheduling the trace diagram can be only updated instead of being recomputed at each workload change, depending on the packing strategy used.

This chapter is organized as follows : First, we make a analysis of periodicity in gang scheduling algorithm in section 5.2. The one dimensional resource sharing problem for gang service algorithms is stated and analyzed in section 5.3. The multi dimensional resource sharing problem is the subject of section 5.4.

5.2 Periodicity in Gang Scheduling

To begin our analysis, we prove a theorem that states that periodic schedules achieve better (or at least as good as) spatial utilization than non-periodic ones for a workload composed of SPMD jobs. That stated we may consider only finite trace diagrams (figure 2.1) in the remainder.

Theorem 5 *Given a workload W composed of parallel SPMD jobs, for every temporal schedule S there exists a periodic schedule S_p such that the idling ratio of S_p is at most that of S.*

Proof 2 *First, let's give a definition that will be useful in this proof. We define here* job happiness *in an interval of time as the number of slots allocated to a job divided by the total number of slots in the interval.*

Define the progress of a job at a particular time as the number of slices granted to each of its tasks up to that time. Thus, if a job has V tasks, its progress at slice S may be represented by a progress vector *of V components, where each component is an integer less than or equal to S. Observe that no task may lag behind another task of the same parallel SPMD job by more than a constant C number of slices. We call this behavior as* legal execution rule. *Note that C depends on the characteristics of the program. It can be determined, for instance, by global synchronization statements. In the worst case C slices corresponds to the completion time of the job. Observe that $C < \infty$, since the data partitions in a SPMD program are necessarily finite, so is the program itself. Therefore, no two elements in the progress vector can differ by more than C. Define the differential progress of a job at a particular time as the number of slices by which each task leads the slowest task of the job. Thus a differential progress vector at time t is also a vector of V components, where each component is an integer less than or equal to C. The differential progress vector is obtained by subtracting out the minimum component of the progress vector from each component of the progress vector. The* system's differential progress vector *(SDPV) at time t is the concatenation of all job's differential progress vectors at time t. The key is to note that the SDPV can only assume a finite number of values. Therefore there exists an infinite sequence of times $t_{i_1}, t_{i_2}, \ldots$ such that the SDPVs at these times are identical.*

Consider any time interval $[t_{i_k}, t_{i'_k}]$. One may construct a periodic schedule by cutting out the portion of the trace diagram between t_{i_k} $t_{i'_k}$ and replicating it indefinitely along the time axis.

We claim that such a periodic schedule is legal. From the equality of the

SPDVs at t_{i_k} and $t_{i'_k}$ it follows that all tasks belonging to the same job receive the same number of slices during each period. In other words, at the end of each period, all the tasks belonging to the same job have made equal progress. Therefore, no two tasks lag behind another task of the same job by more than a constant number of slices.

Secondly, observe that it is possible to choose a time interval $[t_{i_k}, t_{i'_k}]$ such that the happiness of each job during this interval is at least as much as in the complete trace diagram. This implies that the happiness of each job in the constructed periodic schedule is larger than or equal to the happiness of each job in the original temporal schedule.

Therefore, the idling ratio of the constructed periodic schedule must be less than or equal to the idling ratio of the original temporal schedule. Since the fraction of area in the trace diagram covered by each job increases, the fraction covered by the idle slots must necessarily decrease. This concludes the proof.

A consequence of the previous theorem is stated in the following corollary :

Corollary 1 *Given a Workload W, for the set of all feasible periodic schedules S, the schedule with smaller idling ratio is the one with smaller period.*

Proof 3 *The feasible schedule with smaller period is the one which has the smaller number of slices (resulting in a smaller number of total slots) and which packs all jobs as defined in the Gang Scheduling algorithm. The number of occupied slots is the same for all feasible periodic schedules, since the workload is the same. So the ratio between the number of idle slots, which is the difference between the total number of slots and the number of occupied slots, and the total number of slots is minimized when we have a minimum number of slices, which is the case in the minimum period schedule.*

5.3 Resource Sharing in Parallel Job Scheduling : One Dimensional case

In the one dimensional resource sharing problem, jobs are represented by only on parameter : the number of tasks (which is equal to the number of required processors in Gang Scheduling) that compose the job. The machine is characterized by the number of processors. We suppose that other resources, such as memory, have infinite availability. We will analyze in this section the dynamic variation, where arrival times can be different from zero. Observe that this problem is similar to the one dimensional dynamic (on-line) bin-packing problem as will be shown below.

5.3.1 Packing in Gang Scheduling

Recall that the computation of a schedule (i.e. the computation of the trace diagram) can be reduced to a bin packing problem. In the classical, one dimensional bin-packing problem, a given list of items $L = I_1, I_2, I_3, ...$ is to be partitioned into a minimum number of subsets such that the items on each subset sum to no more than B, which is the capacity of the bins. In the standard physical interpretation, we view the items of a subset as having been packed in a bin of capacity B. This problem is NP-Hard [12], so research has concentrated on algorithms that are merely close to optimal. For a given list L, let OPT(L) be the number of bins used in optimal packing, and define :

$$s(L) = \lceil \frac{\sum |I_j|}{B} \rceil \tag{5.1}$$

Note that for all lists L, $s(L) \leq OPT(L)$. For a given algorithm A, let A(L) denote the number of bins used when L is packed by A and define the waste $w_A(L)$ to be A(L)-S(L).

When applying bin-packing to parallel job scheduling, bins corresponds to slices in the trace diagram, and items represents SPMD jobs.

In this paper we deal with bin-packing algorithms that are dynamic (also dubbed “on-line”). A bin packing algorithm is dynamic if it assigns items to bins in order $(I_1, I_2, ...)$, with item I_i assigned solely on the basis of the sizes of the preceding items and the bins which they are assigned to, without reference to the size or number of remaining items [81, 12]. Two of the most well known strategies for dynamic bin-packing are first fit and best fit. In first fit, the next item to be packed is assigned to the lowest-indexed bin having an unused capacity no less than the size of the item. In best fit the used bins are sorted according to their capacities. The item to be packed is assigned to the bin with the smallest capacity that is sufficient. Best fit can be implemented to run in time $O(N \log N)$, and among online algorithms offers perhaps the best balance between worst and average case packing performance [12]. For instance, the only known on-line algorithm with better expected waste than best fit is the considerably more complicated algorithm of [81] that has expected waste $\Theta(N^{1/2} \log^{1/2} N)$ (compared with $\Theta(N^{1/2} \log^{3/4} N)$ for best fit) which is the best possible for any dynamic algorithm ; this algorithm however has an unbounded asymptotic worst case-ratio.

However, it should be stressed that the problem that we consider in this paper is slightly different from the original dynamic bin packing problem, since each item has a duration associated with it. As items represent SPMD jobs, the duration represent the time it takes to run on a dedicated machine.

5.3.2 Dynamic Competitive Ratio

Competitive ratio(CR) based metrics [17, 69] are used to compare various space sharing strategies. The reason is that the competitive ratio is a formal way of evaluating algorithms that are limited in some way (e.g., limited information, computational power, number of preemptions) [17]. This measure was first introduced in the study of a memory management problem [67, 87]. The Competitive ratio [51, 17] for a scheduling algorithm A is defined as :

$$CR(n) = \sup_{J:|J|=n} \frac{A(J)}{OPT(J)} \tag{5.2}$$

Where $A(J)$ denotes the cost of the schedule produced by algorithm A, and OPT(J) denotes the cost of an optimal scheduler, all under a predefined metric M. One way to interpret the competitive ratio is as the payoff to a game played between an algorithm A and an all-powerful malevolent adversary OPT that specifies the input J [51].

We are interested in the dynamic case, where we have a sequence of jobs $J = \{J_1, J_2, J_3, J_4, ...\}$, with an arrival times $a_i \geq 0$ associated with each job, which is the the case for jobs submitted to parallel supercomputers, as several workload studies show [33, 27]. Observe that consecutive arrival times can vary between seconds to hours, depending on the hour of the day [33]. For instance, let's consider a machine that implements a Gang scheduler using the trace diagram (an example is [71]). Upon arrival of a new job, the front end will look for the first slice with sufficient number of processors in the trace diagram (which is stored in the front end), will allocate the incoming job on that slice, will update the trace diagram, and the new job will start running in the next period. The same sequence of actions is taken for subsequent jobs.

For the dynamic case as defined in the previous paragraph, the definition of equation 5.2 is not convenient. For a dynamic scheduling the number of jobs n can be of order of thousands or tens of thousands of jobs, but they are *spaced in time*, in a way that, at each instant of time, we would have typically tens of jobs at most scheduled in the machine. Beyond that, competitive analysis has been criticized because it often yields ratios that are unrealistic high for "normal" inputs, since it consider the worst case workload, and as a result it can fail to identify the class of online algorithms that work well [51]. These facts led us to propose a new methodology for comparing dynamic algorithms on parallel scheduling based on the competitive ratio.

For the application of CR methodology in dynamic scheduling, let's consider as reference (adversary) algorithm the optimal algorithm OPT for a predefined metric M applied at each new arrival time. The OPT scheduler will be a clairvoyant dynamic adversary, with knowledge about all arrival times and the characteristics of all jobs. We will call this methodology of comparing dynamic algorithms as Dynamic Competitive Ratio (CR_d), and the scheduler defined by applying OPT at arrival times as OPT_d. Formally we have :

$$CR_d(N) = \frac{1}{N} \sum_{\tau=1}^{N} \frac{A(\tau)}{OPT_d(\tau)} \tag{5.3}$$

Where N represents the number of arrival times considered. Observe that CR_d only varies at arrival times. As Workload we have a (possibly infinite) sequence of jobs $J = \{J_1, J_2, J_3, J_4, ...\}$, with an arrival time $a_i \geq 0$ associated with each job. At the time of each arrival, workload changes are taken into consideration in a way that only those jobs that are still running at the arrival time are considered by both A and OPT algorithms.

A very important question is the determination of the workload to be used in conjunction with CR_d to compare different algorithms. When choosing the new coming job at each arrival time, we have two possibilities : either selecting a "worst case" job that would maximize the CR_d or considering synthetic workload models with arrival times and running times being modeled as random variables. Since with the "worst case" option we may create fake workloads that never happen in practice and leads to the sort of criticism we cited before, we believe that the best option is to use one of the synthetic workload models that have been recently proposed in the literature [27, 33]. These models and its parameters have been abstracted through careful analysis of real workload data from production machines. The objective with this approach is to produce an average case analysis of algorithms based on real distributions.

A lower bound for the packing problem under dynamic competitive ratio is derived in the following theorem.

Theorem 6 *For the dynamic packing of SPMD jobs in the trace diagram and for any non-clairvoyant scheduler,* $CR_d(N) \geq 1$, $N > 0$

Proof 4 *Consider N=1. In the case of a workload composed by one embarrassingly parallel job with a degree of parallelism P, if we have a non clairvoyant scheduler that schedules each task in a different processor, as would do* OPT_d*, we have* $CR_d(1) = 1$*.*

Conversely, an optimal clairvoyant scheduler will always be capable of producing a scheduling at least as good as a non-clairvoyant one, since the clairvoyant scheduler has all the information available about the workload at any instant in time. So $CR_d(N) \geq 1$*.*

For bin-packing, the reference or optimal algorithms will be simply the sum of item sizes s(L), since $s(L) \leq OPT(L)$, and it can be easily computed. However, in order to use CR_d to compare the performance of algorithms, we must first define precisely the workload model we will use in the CR_d computation, which is done in the next section.

5.3.3 Workload Model

The workload model that we consider in this paper was proposed in [27]. This is a statistical model of the workload observed on a 322-node partition of the Cornell Theory Center SP2 from June 25, 1996 to September 12, 1996, and it is intended to model rigid job behavior. During this period, 17440 jobs were executed.

The model is based on finding Hyper-Erlang distributions of common order that match the first three moments of the observed distributions. Such distributions are characterized by 4 parameters :

- **p** – the probability of selecting the first branch of the distribution. The second branch is selected with probability 1 - p.

- λ_1 – the constant in the exponential distribution that forms each stage of the first branch.

- λ_2 – the constant in the exponential distribution that forms each stage of the second branch.

- **n** – the number of stages, which is the same in both branches.

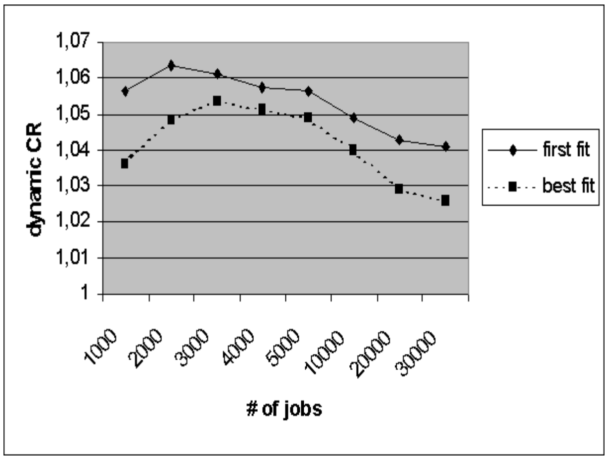

FIG. 5.1: Sequence 1 - no migration

As the characteristics of jobs with different degrees of parallelism differ, the full range of degrees of parallelism is first divided into subranges. This is done based on powers of two. A separate model of the inter arrival times and the service times (runtimes) is found for each range. The defined ranges are 1, 2, 3-4, 5-8, 9-16, 17-32, 33-64, 65-128, 129-256 and 257-322.

Tables with all the parameter values are available in [27].

5.3.4 CR_d applied to First Fit and Best Fit

We conducted experiments measuring the CR_d of first fit and best fit strategies. Our objective is to verify the behavior of these on-line packing strategies in terms of number of slices used when compared against a clairvoyant on-line algorithm. In particular, we are interested in knowing if the non-optimality of either first fit or best fit can lead to an unbounded growth of the number of slices in a period when compared against the optimal on-line algorithm for the workload model chosen. The experiments consisted of computing the CR_d of both best fit and first fit for a sequence of SPMD jobs generated through the model described in the previous section. The machine considered has 128 processing elements, and the size of jobs varied between 1 and 128 tasks, divided in 8 ranges. Then the first and best fit strategies were applied, and the number of slices used by each algorithm for each job arrival was computed. This number was then divided by the number of slices

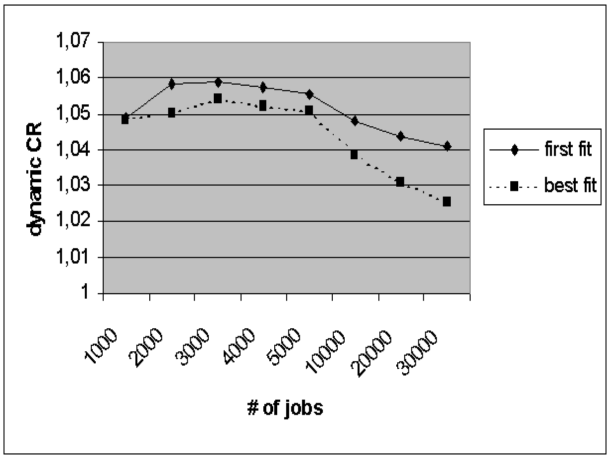

FIG. 5.2: Sequence 2 - no migration

that would be used considering the sum s(L) as defined in equation 5.1. The smaller the number of slices, the smaller the period is, with more slots dedicated to each job over time.Two cases were simulated :

- With job migration - All jobs return to the central queue and are redistributed among all processors at each new arrival.

- Without job migration - In this case an arriving job is allocated accordingly to a given algorithm without changing the placement of other jobs.

Two sequences of jobs were randomly generated using the model described in the previous subsection. Both sequences have 30000 jobs. Results for the first sequence without task migration are shown in figure 5.1. The horizontal axis represents number of job arrivals. We can see that best fit always achieved better results than first fit, but with a small difference between the two strategies. The bigger difference between the CR_d of first fit and the CR_d of best fit was around 2%. Results for the second sequence of jobs are shown on figure 5.2. The results are similar to the one obtained for the first figure, but the larger difference for this sequence was also around 2%. Observe that for a large number of jobs (> 10000) results of both sequences are almost equal for both the best fit and first fit algorithms. There is no unbounded growth of the number of slices when compared against the clairvoyant on-line algorithm in any case.

CR_d calculation for both sequences when task migration is considered are shown in figures 5.3 and 5.4. We can observe that the CR_d for both

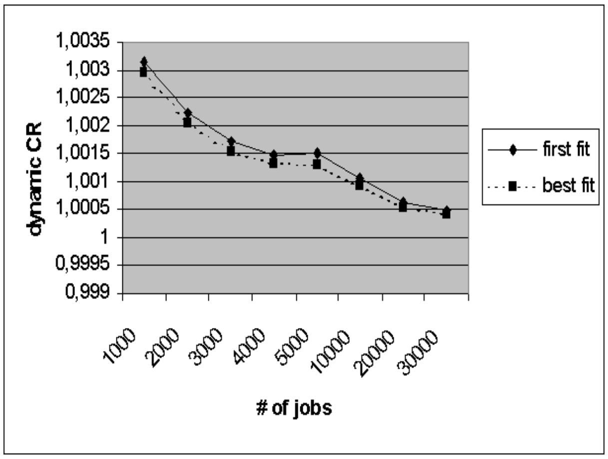

FIG. 5.3: Sequence 1 - with migration

algorithms is at least one order of magnitude smaller than the results with no migration. As a consequence, the difference of the CR_d between the two algorithms become even smaller, with less than 1% in the worst case. Again, CR_d calculations for the two sequences became almost equal for a large number of jobs.

5.4 Resource Sharing in Parallel Job Scheduling : Multi-dimensional case

In the multi-dimensional case the resources available in a machine are represented by a m-dimensional vector $R = (R_1, R_2, ...R_m)$ and the resources required by a job J are represented by a k-dimensional vector $J = (J_1, J_2, ...J_k), k \leq m$. Of particular interest for Gang service algorithms is the amount of memory required for each job, since most of the parallel machines available today do not have support for virtual memory, and the limited amount memory available determine the number of jobs that can share a machine at any given time.

Maximizing memory utilization in order to allow multiple jobs to be scheduled simultaneously is a critical issue for Gang service systems. Many

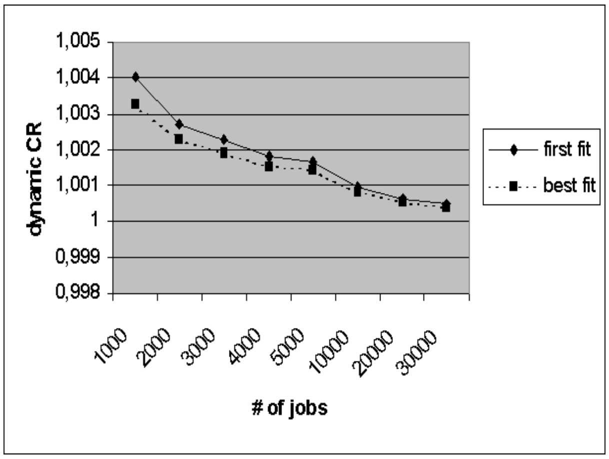

FIG. 5.4: Sequence 2 - with migration

parallel applications demand a large amount of memory to run, and since these machines normally do not have support for virtual memory, eventually all applications submitted to a machine at a given time will not fit into the main memory available, which means that some jobs will have to wait before receiving service.

In the case of a distributed memory machine, the machine itself is modeled as a P-dimensional vector $R(t) = (R_1, R_2, ...R_P)$ where R_I represents the amount of memory available at time t in node I. The job is represented as a F-dimensional vector $J(t) = (J_1, J_2, ...J_F)$, where F is the maximum number of tasks of a job and J_I is the amount of memory required for task I at time t. As we are dealing with SPMD jobs, this formulation can be simplified as we will consider that the amount required by all tasks will be the same, so the requirements of a job are represented by a bidimensional vector $J = (P_J, M_J)$, where P_J is the number of tasks (which is equal to number of processors in Gang service algorithms) of job J and M_J is the maximum memory requirement among all tasks.

Observe that this problem is different from the two-dimensional (geometric) bin-packing problem [11], in which rectangles are to be packed into a fixed width strip so as to minimize the weight of packing, since the memory segments required by a job do not need to be contiguous. It also differs from the vector bin packing problem. In the d-dimensional version of the vector packing problem, the size of an item is a vector $I = (I_1, I_2, ...I_d)$ and the capacity of a bin is a vector $C = (C_1, C_2, ...C_d)$. No bin is allowed to contain items whose vector sum exceeds C in any component. In the multidimensional resource sharing problem stated in this section the dimensions of the size of an item and the capacity of a bin can be different, as we are considering a distributed memory machine and the resources (memory) available in the machine are defined in a per-processor basis, not in a global basis. This leads to different solutions than those proposed in [60], since a direct association between kth component of the resource vector R and the kth component of requirement vector J is not mandatory.

5.4.1 Memory Fit algorithm

Our objective is to maximize the number of jobs that can be allocated in the same period. To do so, the packing strategy must take into account the amount of memory available on each node. That is the objective of the *memory fit algorithm* proposed in this section. In the memory fit algorithm, the front end at each job arrival chooses the slice and the processors that will receive an incoming job depending on the amount of memory available, in order to balance the usage of memory among the nodes of the machine. If more than one solution is possible, the front end chooses a slice using the best fit strategy. If there is no set of processors in the available slices with sufficient memory to receive the new job, the front end can create a new slice to accommodate the new coming job. Of course a new slice can also be created due to insufficient number of processors in the existing slices, as in best fit and first fit.

If a job arrives and there is no sufficient resources available for its execution, it waits in the global queue until the amount of memory required by the job becomes available. However, if another jobs arrives later and there is a sufficient amount of memory to schedule the job, it is scheduled immediately, as in the backfilling strategy.

When choosing a slice for scheduling a new coming job, first the memory fit chooses the J processors in a slice where there is more memory available, where J is the degree of parallelism of a job. Then computes a "measure of balance" in order to know how much that particular allocation reduces imbalance. The algorithm repeats this sequence for every existing slice and also considers the possible creation of a new slice. The solution that minimizes the memory imbalance in the machine is the chosen.

For the measure of memory imbalance, in this paper we have chosen a

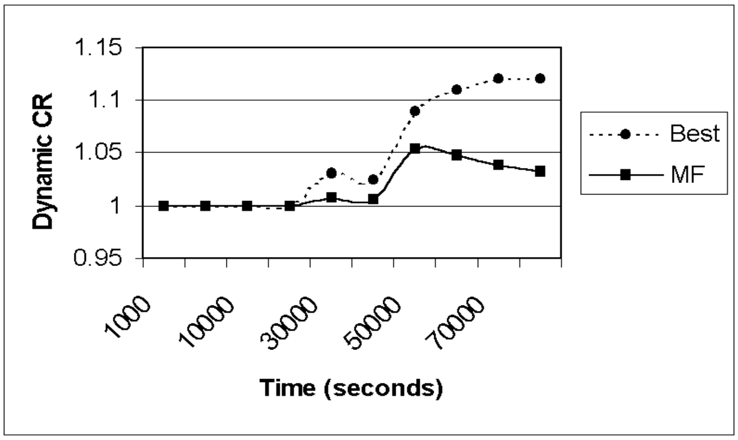

FIG. 5.5: Dynamic competitive ratio applied to memory fit

max/average balance measure [59]. For a given slice, the front end considers the allocation of a job in the set of processors where there is more memory available, and then compute the following measure :

$$B(slice) = \frac{\max_i M_i}{\sum_{i=1}^{P} M_i} \tag{5.4}$$

Where M_i is the amount of memory available in node i. A lower value of B indicates a better balance.

5.4.2 CR_d applied to Memory fit

In this section we apply a sequence of jobs to evaluate the performance under CR_d of the memory fit packing algorithm and we compare with the best fit algorithm of section 5.3, with no migration in both cases. This algorithm was modified to take into account memory requirements. The bin chosen is the one that minimizes processor waste and at the same time has sufficient memory to accommodate the job, regardless of any memory balance. However, we must first define a memory usage for each job in the sequence, since the workload model used did not take into account memory requirements. In this paper, we considered that the memory usage of a job has a direct correlation with this size. This assumption is coherent with workload modeling that was used as a guide line for scheduler design in parallel

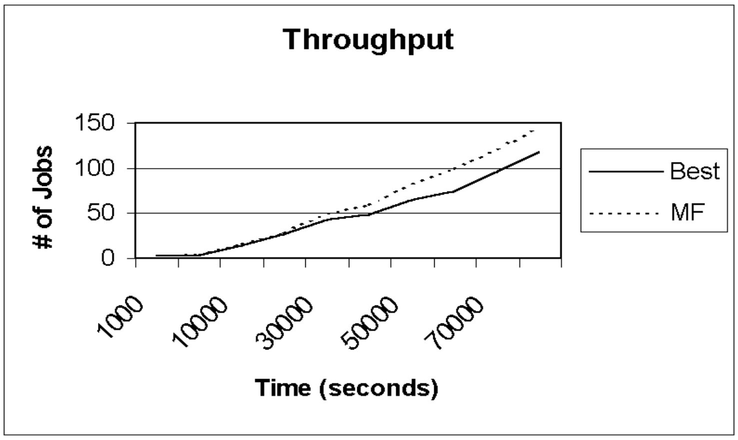

FIG. 5.6: Throughput of both best fit and memory fit

machines, notably in the Tera MTA [26]. In this work was observed that jobs with large amounts of parallelism have large memory requirements and use a lot of resources. Small tasks, on the other hand, use resources in bursts, have small memory requirements and are not very parallel. Based on these observations we defined that for each job, the memory utilization of a job varied between 2 MB and 256 MB per node in function of the size of the job and the ranges defined in subsection 5.3.3. For instance, we considered that 1 task jobs require 2 MB of memory, 2 task jobs require 4 MB per task, 3-4 task jobs require 8 MB per task, and so on. Each node had 512 MB of main memory.

Our objective is to maximize the number of jobs that fit in one period of the machine by maximizing the memory utilization of the machine. The reference algorithm always has a memory utilization of 100%, given that enough jobs have arrived to fill in the memory, due to its clairvoyance. Simulation results are shown in figures 5.5 and 5.6. Figure 5.5 illustrates the evolution of CR_d over time. Figure 5.6 shows the throughput of both algorithms. Observe that the number of arrivals and arrival times submitted to both algorithms are the same, since the same sequence of jobs were submitted to both algorithms. In both cases the horizontal axis represents time in seconds. The evolution of the system was simulated for 10000, 20000, 30000, 40000 and 50000 seconds. These values are chosen in order to verify the evolution of the system during a working day (50000 seconds represents 13.8 hours).

We can observe that the Memory Fit algorithm not only yields better memory utilization than best fit, but it also improves the throughput, as illustrated by figure 5.6. This is a direct consequence of the capability of the memory fit algorithm of allocating more jobs in the same period.

5.5 Conclusion

In this chapter we analyzed questions related to resource sharing in parallel scheduling algorithms. One conclusion derived from that analysis is that multidimensional resource sharing analysis is necessary when defining a packing strategy for this class of algorithms, as the comparison between best fit and memory fit in the limited memory analysis demonstrated.

To provide a sound analysis of Concurrent Gang performance, a novel methodology, dubbed dynamic competitive ratio, based on the traditional concept of competitive ratio is also introduced. Dynamic competitive ratio was used to compare packing algorithms submitted to a workload generated by a statistical model, and to compare packing strategies for job scheduling under multiple constraints. For the unidimensional case we can conclude that there is not a large difference between the performance of best fit and first fit under the workload model considered, and first fit can be used without significant system degradation. For the multidimensional case, when memory is also considered, the better performance of memory fit over best fit under dynamic CR let us conclude that the packing algorithm must try to balance the resource utilization in all dimensions at the same time, instead of given priority to only one dimension of the problem.

Deuxième partie

Concurrent Gang

Chapitre 6

Concurrent Gang

6.1 Introduction

In this chapter we propose a class of scheduling policies, dubbed Concurrent Gang, that is a generalization of Gang-scheduling and allows for the flexible simultaneous scheduling of multiple parallel jobs in a scalable manner. In order to do that, the Concurrent Gang scheduler identifies the characteristics of each task at run time and takes a decision about the best way of scheduling tasks depending on these characteristics. Along with that, a solution to the problem related to Gang schedulers of what to do when a task blocks is proposed.

The architectural model we will consider in this chapter is a distributed memory processor with four main components :1) Processor/memory modules (Processing Element - PE), 2) An interconnection network that provides point to point communication, 3) A synchronizer, that send a synchronization (clock) signal to all PEs at regular intervals of L time units and 4) a front end, where incoming jobs arrive. This architecture is similar to the one defined in the BSP model [94].

This chapter is organized as follows : in section 6.2 we present the task classification policy that is used in the Concurrent Gang algorithm. The Concurrent Gang scheduler is described in section 6.3. In section 6.4 experimental results are presented and analyzed, and section 6.5 contains our final remarks for this chapter.

6.2 Task Classification

We will use information gathered at runtime to allow each PE to classify each one of its allocated tasks into classes. Examples of such classes are : I/O intensive, communication intensive, and computation intensive. Each one of these classes is equivalent to a fuzzy set [96]. A fuzzy set related with a class A is characterized by a membership function $f_A(T)$ which associates each task T to a real number in the interval [0,1], with the value of $f_A(T)$ representing the "degree of membership" of T in A. Thus, the nearer the value of $f_A(T)$ to unity, the higher the degree of membership of T in A, that is, the degree to which a task belongs to class A. For instance, consider the class of I/O intensive tasks, with its respective membership function $f_{IO}(T)$. A value of $f_{IO}(T) = 1$ indicates that the task T belongs to the class I/O intensive with maximum degree 1, while a value of $f_{IO}(T) = 0$ indicates that the task T has executed no I/O statement at all. Observe the deterministic nature of degree of membership associations. It is also worth noting that the actual number of classes used on a system depends on the architecture of the machine.

The information related to a task may be gathered during system calls and context switches. Information that can be used to compute the degree of membership are the type, number and time spent on system calls, number and destination of messages sent by a task, number and origin of received messages, and other system dependent data. These informations can be stored, for instance, by the operating system on the internal data structure related to the task.

When applying fuzzy sets for task classification, the value of $f(T)$ for a class is computed by the PE in a regular basis, at the preemption of the related task. As an example, let's consider the I/O intensive class. The exact way of computing being system dependent, one way of doing the computation is as follows : On each I/O related system call, the operating system will store information related to the call on the internal data structure associated to the task, and at the end of the time slice, the scheduler computes the time spent on I/O calls in the previous slice. Then the scheduler computes the time spent in I/O over the last N times where the task was scheduled (N can be, for instance, 3). This average determines the degree of membership of a particular task to the class I/O intensive. As many jobs proceed in phases, the reason for using an average over the last N times a task was scheduled is detection of phase change. If a task changes from an I/O intensive phase to a computation intensive phase, this change should be detected by the scheduler. In general, the computation of the degree of membership of a task to the class I/O intensive will always depend on of the number and/or duration of the I/O system calls made by the task. The same is valid for the communication intensive class; the number and/or duration of communication statements will define the degree of membership of a task to this class. For the class computing intensive, degree of membership will also be a function of system calls and communication statements, but in another sense : for a smaller the number of system calls and communications there is a increase of the degree of membership of a given task to the class computing intensive.

6.2.1 Fuzzy subsets as points

It helps to see the geometry of fuzzy sets when we apply the theory to a practical problem, in our case parallel job scheduling. In the geometric representation of fuzzy sets [56, 55], the fuzzy power set F(2^X), the set of all fuzzy subsets of X, is represented by a cube. A fuzzy set is represented by a point in the cube. The set of all fuzzy subsets equals the unit hypercube $I^n = [0, 1]$. A fuzzy set is any point in the cube I. So (X,I) defines the fundamental measurable space of finite fuzzy theory.

Vertices of the cube I define non-fuzzy sets. So the ordinary power set 2^X, the set of all 2^n non-fuzzy subsets of X, equals the boolean n-cube B^n :$2^X = B^n$. Fuzzy sets fill in the lattice B^n to produce the solid cube $I^n : F(2^X) = I^n$.

Consider the set of two elements $X = \{x_1, x_2\}$. The non fuzzy power set 2^X contains four sets : $2^X = \{0, x_2, X, x_1\}$. These four sets corresponds respectively to the four bit vectors (0 0),(0 1),(1 1), and (1 0).The 1s and 0s indicates the presence or absence of the i-th element x_i in the subset. More abstractly, we can uniquely define each subset A as one of the two valued membership functions $m_A : X \to \{0, 1\}$

Now consider the fuzzy subsets of X. We can view the fuzzy subset A=(1/3, 2/3) as one of the continuum-many continuous-valued membership function $M_A : X \to [0, 1]$. In this example element x_1 belongs to, or fits in, subset A to degree 1/3. Element x_2 has a membership of 2/3. Analogous to the bit vector representation of finite countable sets, we say that the fit vector (1/3, 2/3) represents A. The element $m_A(x_i)$ equals the i-th fit or or fuzzy unit value. The sets as points concept represents the fuzzy subset A as a point in the I^2, the unit square, as shown in figure 6.1

Viewing a class as a fuzzy sets corresponds to associate them to a point

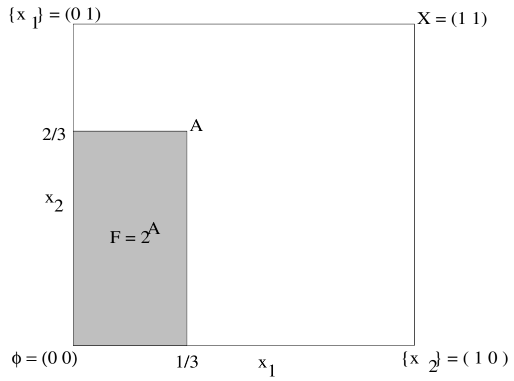

FIG. 6.1: Sets as points. The fuzzy subset A is a point in the unit 2-cube with coordinates of fit values (1/3,2/3). The first element x_1 fits or belongs to A to degree 1/3, the element x_2 to degree 2/3. The cube consists of all possible fuzzy sets of two elements x_1, x_2

in a n-dimensional space, with n being the number of tasks allocated to a processor at time t. That is, given a class A, it can be described at a given time t as A(t) $= (f_A(T_1)$,$f_A(T_2)$,$f_A(T_3)$,...,$f_A(T_n))$ for n tasks.

6.2.2 Example of task classification using number of statements

It is possible to compute the degree of membership of a task with very low overhead and a small amount of stored information by the operating system, based on the number of executed statements related to a task. This can be done, for instance, as follows : at initialization the degree of membership of a task related to each class is equal to 1/2. The measurement related to a time slice is made as follows (in the following, we consider three classes) :

- If the task is blocked due to an I/O call in that time slice, the measurement of the degree of membership of the class I/O bound is equal to 4/5. Otherwise is equal to 1/5.
- If the task executes a number N of communication statements in a time slice, the measurement of the degree of membership of the class communication intensive is equal to 4/5. Otherwise is equal to 1/5.
- If the task is not blocked due to an I/O call and executes less than N communication statements, the measurement of the degree of member-

ship of the class computation intensive for that time slice is equal to 4/5. Otherwise is equal to 1/5.

The measurement for an interval is then summed to the previous total measurement multiplied by 1/5, becoming the new total measurement for a given class. Observe that the total measurement related to each class is a real number between]0,1[.

6.3 Concurrent Gang

In this section we present the Concurrent Gang algorithm by describing the components and the operation of a Concurrent Gang Scheduler.

6.3.1 Definition of Concurrent Gang

Referring to figure 6.2, for the definition of Concurrent Gang we view the parallel machine as composed of a general queue of jobs to be scheduled and a number of servers, each server corresponding to one processor. Each processor may have a set of tasks to execute. Scheduling actions are made at two levels : In the case of a workload change, global spatial allocation decisions are made in a front-end scheduler, who decides in which portion of the trace diagram the new coming job will run. The context switching of local tasks in a processor as defined in the trace diagram is made through local schedulers, independently of the front-end. The global synchronizer is responsible for sending a synchronization signal to all processors every L time units in order to indicate the end of a slice.

A local scheduler in Concurrent Gang is composed of two main parts : the Gang scheduler and the local task scheduler (LTS). The Gang Scheduler schedules the next task indicated in the trace diagram at the arrival of a synchronization signal. The LTS is responsible for scheduling sequential tasks and parallel tasks that do not need global coordination, as described in the next paragraph, and it is similar to a UNIX scheduler. The Gang Scheduler has precedence over the LTS.

We may consider two types of parallel tasks in a Concurrent Gang scheduler : Those that require coordinated scheduling with other tasks in other processors and those that Gang scheduling is not mandatory. Examples of the first type are tasks that compose a job with fine grain synchronization interactions [37] and communication intensive jobs [31]. Second type examples are tasks that compose an I/O bound parallel job, for instance. In [58] Lee et al. proved that response time of I/O bound jobs suffers under Gang scheduling and that may lead to significant CPU fragmentation. On the other hand a traditional UNIX scheduler does good job in scheduling I/O bound tasks since it gives high priority to I/O blocked tasks when the data becomes available from disk. As those tasks typically run for a small amount of time and then blocks again, giving them high priority means running the task that will take the least amount of time before blocking, which is coherent to the theory of uniprocessors scheduling where the best scheduling strategy possible under total completion time is Shortest Job First [69]. In the LTS of Concurrent Gang, such high priority is preserved. Another example of jobs where Gang scheduling is not mandatory are embarrassingly parallel jobs. As the number of iterations among tasks belonging to this class of jobs are small, the basic requirement for scheduling a embarrassingly parallel job is to give those jobs the greater fraction of CPU time possible, even in an uncoordinated manner.

The LTS defines a priority for each task allocated to the corresponding PE. The priority of each task is defined based on the degree of membership of a task to each one of the major classes described in the previous subsection. Formally, the priority of a task T in a PE is defined as :

$$Pr(T) = max(\alpha \times f_{IO}, f_{COMP}) \tag{6.1}$$

Where f_{IO}, f_{COMP} are the degrees of membership of task T to the classes I/O intensive and Computation intensive respectively. The objective of the parameter α is to give higher priority to I/O bound jobs ($\alpha > 1$). In the experiments of this chapter we have defined $\alpha = 2$. The choices made in equation 6.1 intend to give high priority to I/O intensive and computation intensive jobs, since such jobs can benefit the most from uncoordinated scheduling. The multiplication factor α for the class I/O intensive gives higher priority to I/O bound tasks over computation intensive tasks, since those jobs have a higher probability to block when scheduled than computing bound tasks. On the other hand, synchronization intensive and communication intensive jobs have low priority since they require coordinated scheduling to achieve efficient execution and machine utilization [37, 31]. A synchronization intensive or communication intensive phase will reflect negatively over the degree of membership of the class computation intensive, reducing the possibility of a task be scheduled by the local task scheduler(LTS). Among a set of tasks of the same priority, the LTS uses a round robin strategy.

In practice the operation of the Concurrent Gang scheduler in each processor will proceed as follows : The reception of the global synchronization signal will generate an interruption that will make each processing element schedule tasks a Gang as defined in the trace diagram. If a task blocks, control will be passed to the LTS that will schedule one of the other ready tasks allocated in the PE depending on the priority assigned to each one of the tasks until the arrival of the next clock signal. The task chosen is the one with higher priority.

At each workload change the front-end of the Concurrent Gang Scheduler will :

1 - Update Eligible task list
2 - Allocate Tasks of First Job in General Queue.
3 - While not end of Job Queue
    Allocate all tasks of remaining parallel jobs
    using a defined spatial sharing strategy
4 - Run

Between Workload Changes

- If a task blocks or in the case of an idle slot, the local task scheduler (LTS) is activated, and it will decide to schedule a new task based on :

– Availability of the task (task ready)

– Priority of the task defined by the local task scheduler

Considering rigid jobs, that is, jobs which the number of required processors is fixed and does not change during execution, the relevant events which define a workload change are job arrival and job termination.

The local queue positions represent slots in the scheduling trace diagram. The local queue length is the same for all processors and is equal to the number of slices in a period of the schedule. It is worth noting that in the case of a workload change, only the PEs concerned by the modification in the trace diagram are notified.

In the case of creation of a new task by a parallel task, or parallel task completion, it is up to the local scheduler to inform the front-end of the workload change. The front end will then take the appropriate actions depending on the predefined space sharing strategy.

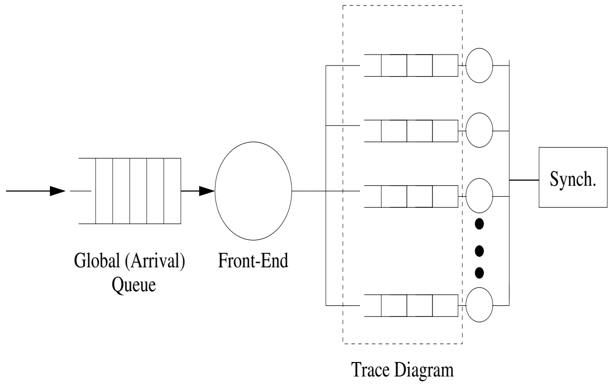

FIG. 6.2: Modeling Concurrent Gang class algorithm

Scalability in Concurrent Gang is improved due to the presence of a synchronizer working as a global clock, which allows the scheduler to be distributed among all processors. The front-end is only activated in the event of a workload change, and decision in the front end is made depending on the chosen space sharing strategy. This differs from typical Gang scheduling implementation where job-wide context switch relies on the front end [20], which limits scalability and efficient utilization of processors when a task blocks.

6.4 Experimental Results

The performance of Concurrent Gang was simulated and compared with the traditional Gang scheduling algorithm, using first fit without thread migration as space sharing strategy. First the simulation methodology is explained and then simulation results are presented and analyzed.

6.4.1 Simulation Methodology

To perform the experiments we used an improved version of a general purpose event-driven simulator, first described in [82], developed by our research group for studying a variety of problems (e.g. dynamic scheduling, load balancing, etc) [43]. The format for describing jobs is a set of parameters used to describe the job characteristics such as computation/communication ratio, I/O duration, etc. Observe that the actual communication type, timing and pattern are left unspecified and it is up to the simulator to convert this user specification into a DAG, using probabilistic distributions, provided by the user, for each of the parameters. Other parameters include the spawning factor for each thread, a thread life span, synchronization pattern, degree of parallelism (maximum number of thread that can be executed at any given time), etc. Even-though probabilistic distributions are used to generate the workload, the generated workload itself behaves in a completely deterministic way. A more complete description of the simulator is in appendix A, as well as the procedure used for its verification.

Once the workload and architecture characteristics are defined, and the module responsible for implementing a particular scheduling heuristics is plugged into the simulator, several experiments can be performed using the same input by changing some of the parameters of the simulation such as the number of processing elements available or the topology of the network, among others. The outputs can be recorded in a variety of formats for later visualization.

The workload model that we consider in this chapter is the same rigid job model used on chapter 5 and proposed in [27]. This is a statistical model of the workload observed on a 322-node partition of the Cornell Theory Center's IBM SP2 from June 25, 1996 to September 12, 1996, and it is intended to model rigid job behavior.

The model is based on finding Hyper-Erlang distributions of common order for inter-arrival and service times that match the first three moments of the observed distributions. As the characteristics of jobs with different degrees of parallelism differ, the full range of degrees of parallelism is first divided into subranges. This is done based on powers of two. A separate model of the inter-arrival times and the service times (execution times) is found for each range. The defined ranges of degrees of parallelism are 1, 2, 3-4, 5-8, 9-16, 17-32, 33-64, 65-128, 129-256 and 257-322 .

Tables with all the parameter values are available in [27].

Four classes of workloads were used in simulations. Our intention is to represent a superset of the major classes we considered in the description of Concurrent Gang. They are :

- Communication Intensive - In this workload all jobs are communication intensive, i.e. the job proceed in phases and for each computation intensive phase there is a communication intensive phase, where the set of tasks related to a job make intensive point to point communications. The semantics used for the point to point communication was non-blocking asynchronous sends and blocking receives. For Gang scheduling, receive statements are executed using the spin only mechanism ; this is due to the fact that typical Gang scheduler do not know what to do if a task blocks. For Concurrent Gang schedulers was used a spin block mechanism, where the task spin for some time waiting for the message before blocking. The reason is that Concurrent Gang schedulers are able to schedule another task if a task in a Gang blocks. The Spin-block mechanism bounds the wait by above in presence of irregularities among tasks. The minimum spin time considered is the baseline time for the network used, that is, the minimum amount of time necessary to keep tasks coordinated if they are already in such state and there is no irregularities among tasks. Point-to-point communication statements appeared in average each 5 ms.

- Computation intensive - In this class all tasks only have local computation instructions, that is, they never block.
- IO intensive - This job type is composed of bursts of local computations followed by bursts of I/O commands, as represented in figure 6.3. This pattern reflects the I/O properties of many parallel programs, where execution behavior can be naturally partitioned into disjoint intervals, each of which consist of a single burst of I/O with a minimal amount of computation followed by a single burst of computation with a minimal amount of I/O [77]. The interval composed of a computation burst followed by an I/O burst are know as phases, and a sequence of consecutive phases that are statistically identical are defined as a working set. The execution behavior of an I/O bound program is therefore comprised as a sequence of I/O working sets. This general model of program behavior is consistent with results from measurement studies [88, 89]. Observe that I/O demands were considered in a first come first served basis.

  The time duration of the I/O burst for the simulations in this chapter were equal to 50 ms in average. The ratio of the I/O working set used in simulations was 1/1, that is, for a burst of 50 ms of I/O there was a burst of 50 ms of computation in average. Observe that I/O requests from different jobs to the same disk are queued and served by arrival order.
- Synchronization intensive - In this workload there is one global syn-

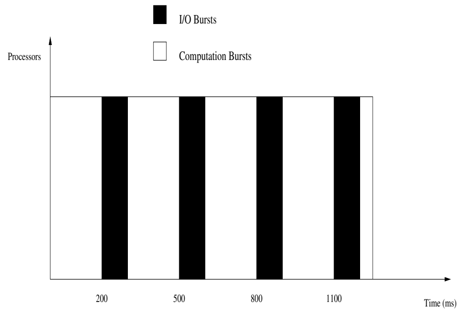

FIG. 6.3: One I/O working set

chronization statement after a variable amount of computation, which is the same for all tasks of a given job. The semantics for the synchronization statement is the same of the communication intensive case, that is, Gang schedulers use spin only, while spin-blocking is used for Concurrent Gang schedulers, since those schedulers can choose another parallel task to run when a parallel task blocks. Global synchronizations occurred in average each 5 ms.

We have simulated a machine with 16 processors using as interconnection network a 2D mesh, as in the Paragon multiprocessor. The simulator's time unit is seconds, and results are obtained for 10000, 20000, 30000 and 40000 seconds. The time slice duration was 200 ms.

All workloads are randomly generated, and then the same set of jobs with their arrival and execution times are presented to Concurrent Gang scheduler with the priority mechanism as defined in section 6.3, a Concurrent Gang scheduler without the priority mechanism (in this case a round robin strategy for defining which task will run in an idle slice is used) and a Gang Scheduler. Space sharing strategy for the Gang scheduler and the Concurrent Gang scheduler is first fit without thread migration. At the end of each simulation, the total idle time and number of completed jobs are returned. It should be noted that the total idle time in the simulations is not composed by idle slots only, but also by the time which a particular task was blocked waiting for I/O, synchronization and communication completion.

6.4.2 Simulation Results

Simulation results for the I/O intensive workload are shown in figure 6.4. The idle time associated with each algorithm is shown in figure 6.5. We can observe a significant improvement over Gang scheduling, both in throughput (jobs completed by unit of time) and total idle time, with Concurrent Gang with priorities having a better performance. These results were expected, since Concurrent Gang provides larger flexibility than Gang scheduling, which is necessary for this kind of job.

We can see a very significant improvement of the modified Gang over the original Gang scheduler, due to the priority mechanism. Essentially, the better performance of Concurrent Gang is due to the fact that it is able to recognize when a task is in an I/O intensive phase, and schedule these tasks as soon as another one blocks, what happens frequently in an I/O bound workload. In Gang scheduling, if an I/O task blocks, the processor remains idle until the end of the time slice. This explains the difference in idle time between Gang scheduling and the two versions of Concurrent Gang. The improvement in throughput observed in consequence of the improved utilization achieved by Concurrent Gang. In [77] , Rosti et al. suggest that that the overlapping of the I/O demands of some jobs with the computational demands of other jobs may offer a potential improvement in performance. The improvement shown in figures 6.4 and 6.5 is due to this overlapping. The detection of I/O intensive tasks and the immediate scheduling of one of these tasks when another task doing I/O blocks results in a more efficient utilization of both disk and CPU resources.

Figure 6.6 shows results of the simulation of a Computation intensive workload, where the jobs have no communication, I/O or synchronization statements at all. As a consequence, tasks in this workload never blocks. The idle time is shown in figure 6.7. Also in this case Concurrent Gang has slightly better utilization than Gang scheduling. The reason is that Concurrent Gang uses the idle slots, that exists due to the non-optimality of the packing strategy used, to schedule computing intensive tasks that the local task scheduler on each PE detects at run-time. Observe that in this case the improvement in utilization is so small that is not enough to cause a significant improvement in throughput. This supports the choice of first fit as a good on-line packing strategy for the workload model considered.

Synchronization intensive workload results are shown in figures 6.8 for the throughput and figure 6.10 for the idle time. The synchronization was always global, i.e. over all tasks of a job. Although Gang scheduling is a better option to schedule those jobs than uncoordinated scheduling, Concurrent Gang had better performance than Gang scheduling for the simulated workload. This is due to the fact that Concurrent Gang with priority can recognize those tasks that are computing intensive, that is, those tasks that belong to jobs having only one task, and reschedule those tasks on idle slots. In the workload model that we used, nearly 40% of the jobs are one task jobs. Since they are one task jobs, there is no synchronization/communication statements, so they do not block and can be considered computing intensive. The better service given to those jobs when idle slots are present explains the better performance of Concurrent Gang when compared with regular Gang for this workload.

Observe also that Concurrent Gang scheduler without the priority mechanism have worse performance than Gang scheduler. The reason is the incapacity of the scheduler without the priority mechanism of differentiate among those jobs that are computing intensive than those that are synchronization intensive. The scheduling of synchronization intensive tasks in idle slots hurts the performance of the job as a whole. This is illustrated in figure 6.9, where we have a trace diagram with four processors and three jobs. Scheduling a synchronization intensive task belonging to job 3 (J_3) in an uncoordinated manner makes this task block at the first barrier call. When the other tasks belonging to job J_3 are scheduled, the blocked task is not, and the local scheduler of Concurrent Gang without priority will schedule a task belonging to another job (job 1) in that slot. As the job is synchronization intensive, the other scheduled tasks belonging to job J_3 will block at the next barrier call after the one that caused the blocking of the first task, since this first task is not scheduled. The other job J_3 tasks will then yield their respective processors, resulting in a degraded performance for job J_3. Since the local task scheduler in this case is oblivious, this process will happen may times, which results in a reduced throughput if compared with Gang scheduling, when all tasks of synchronization intensive jobs are scheduled only in a synchronized way.

In figures 6.11, throughput results for communication intensive (point to point) workload are shown for 16 processors. Idle times are in figures 6.12. Once again we observe a improvement in total idle time and in throughput due to the identification and scheduling of those jobs that are in a computing intensive phase in idle slots. Again, although Gang scheduling is better than uncoordinated scheduling for this kind of job, Concurrent Gang shows to be slightly better than Gang scheduling, due to the better service given to

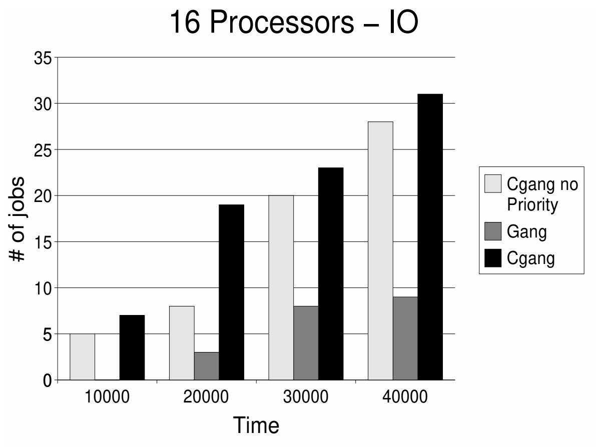

FIG. 6.4: 16 Processors, I/O bound workload : Throughput

computing intensive tasks.

As with the synchronization intensive workload, Concurrent Gang without priorities is worse than Concurrent Gang and Gang scheduling. The reason is the same of the synchronization intensive workload. This happens because, without the priority mechanism, Concurrent Gang schedulers do not have information to decide if a task is a communication intensive or computation intensive. The scheduling of a communication intensive task in a idle slot may put that task into a blocking state, since we use spin block semantics for both Concurrent Gang schedulers. This task will then be blocked during the assigned slice of the job in the trace diagram and then may cause the blocking of other tasks. As in the synchronization intensive workload, this hurts the response time of that job as a whole. Other tasks of the Gang will be executed, but those tasks which communicate with the task that blocked first will eventually block as well.

6.5 Discussion and Conclusion

In this chapter we presented a new parallel scheduling algorithm dubbed Concurrent Gang. The main differences over standard Gang scheduling are the explicit definition of an external global synchronizer and the presence of local task scheduler which decides what to do if a task of the job scheduled as a Gang blocks, as we propose in this chapter a solution to the task blocking

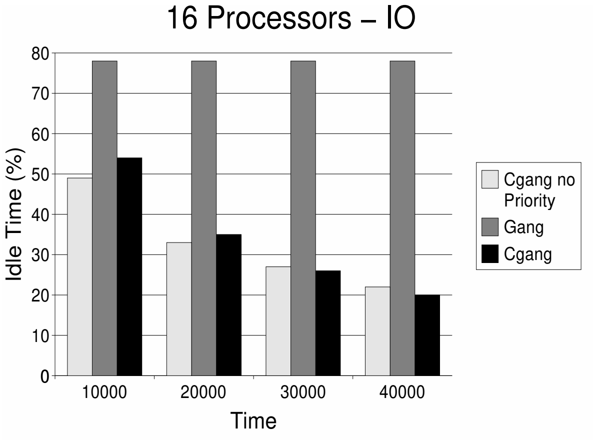

FIG. 6.5: 16 Processors, I/O bound workload : idle time

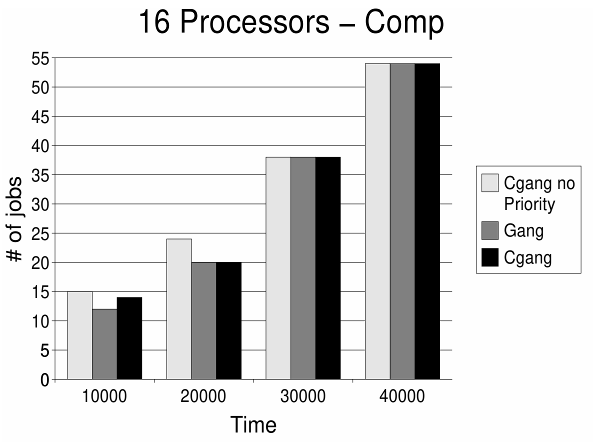

FIG. 6.6: 16 Processors, Computation intensive workload : Throughput

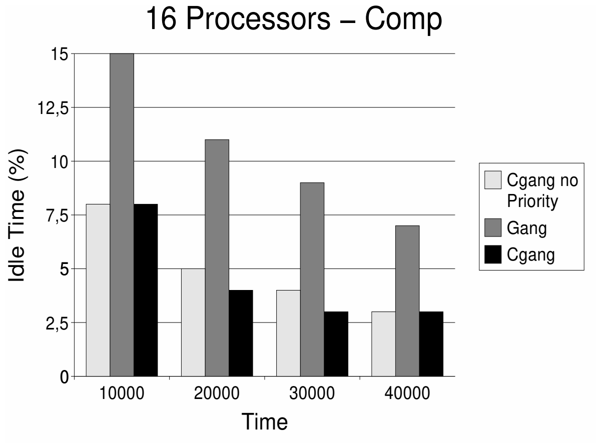

FIG. 6.7: 16 Processors, Computation intensive workload : idle time

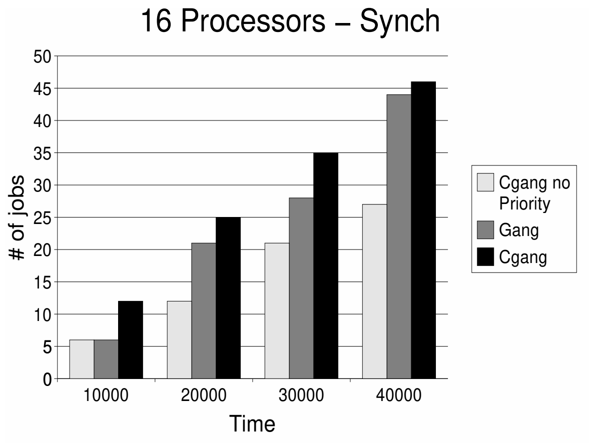

FIG. 6.8: 16 Processors, Synchronization intensive workload : Throughput

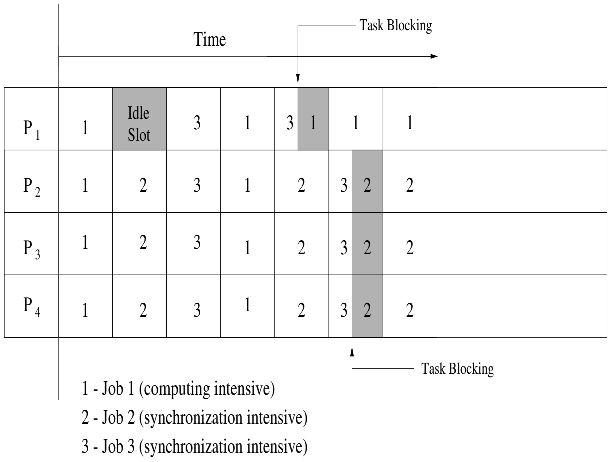

FIG. 6.9: Execution of a synchronization intensive workload by a Concurrent Gang scheduler without priorities. Observe that the scheduling of an isolated that belonging to a synchronization intensive job causes the blocking of the other tasks of the same job on its assigned slice

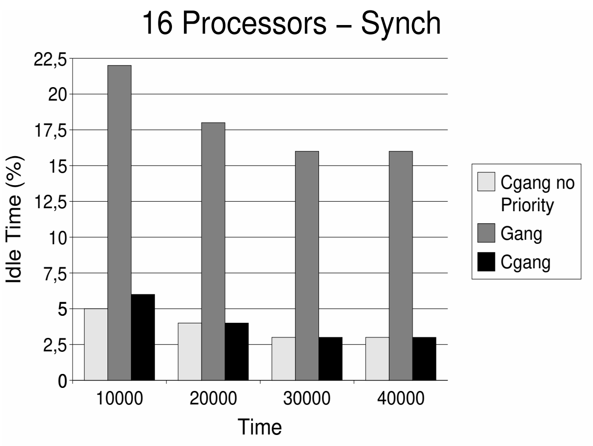

FIG. 6.10: 16 Processors, Synchronization intensive workload : idle time

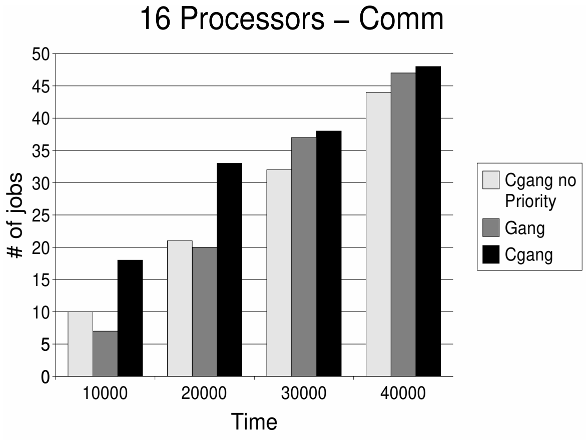

FIG. 6.11: 16 Processors, Communication intensive workload : Throughput

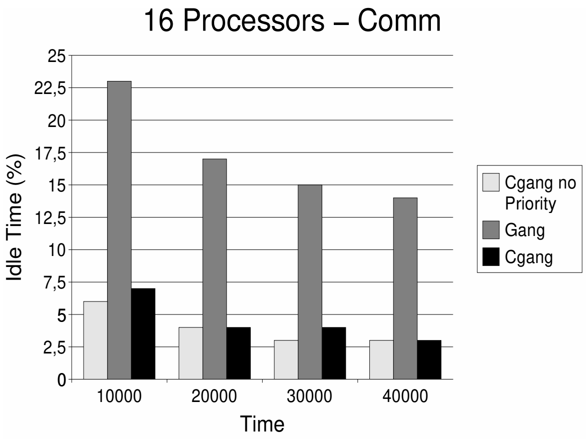

FIG. 6.12: 16 Processors, Communication intensive workload : idle time

problem in Gang scheduling.

The Concurrent Gang approach is more beneficial to workloads that require a more flexible scheduling than is possible with Gang scheduling. An example is I/O bound workloads, as is demonstrated with simulation results. For workloads requiring coordinated scheduled, the Concurrent Gang algorithm becomes equivalent to the standard Gang scheduler, as verified with a communication bound workload. But simulation results showed that even with communications intensive and synchronization intensive workloads Concurrent Gang may yield improved performance over Gang scheduling.

The utilization in Concurrent Gang is improved because, in the event of an idle slot or blocked task, Concurrent Gang always tries to schedule tasks that do not require, at that moment, coordinated scheduling with other tasks of the same job. This is the case, for instance, of I/O intensive tasks and computation intensive tasks. The priority mechanism in Concurrent Gang is used by the scheduler to make a intelligent choice of which task to schedule in a idle slot, thus improving performance over a Concurrent Gang scheduler with no priority as shown in simulations.

Chapitre 7

Concurrent Gang Analysis

7.1 Introduction

This chapter complements the previous one by making a detailed analysis of Concurrent Gang algorithms in two different points : first, we compare Concurrent Gang with Gang scheduling and a oblivious local scheduler. Then we analyze the performance of Concurrent Gang the presence of irregular jobs, and propose an improvement to Concurrent Gang which binds the Concurrent Gang scheduler with a load balancing algorithms to deal with jobs that presents irregularities in the number of eligible tasks during execution.

In the first section of this chapter we compare scheduling algorithm that make use of runtime measurements to gather information about tasks, as Concurrent Gang, against other algorithms that do not make use of such information, such as Gang schedulers and oblivious local schedulers. The main result from this chapter is that oblivious schedulers can not do better than schedulers for which the runtime information is available. We can therefore conclude that a Concurrent Gang scheduler will always be at least as good as a Gang scheduler for the same distribution of tasks among processors. Additionally, we can also conclude that it will always perform better than a local scheduler oblivious to runtime information.

In the second section, we make an analysis of the behaviors of Concurrent Gang in the presence of irregular jobs. In order to make Concurrent Gang even more efficient when a irregular workload is scheduled, the propose in this section a variation of the Concurrent Gang scheduling algorithm that includes a load balancing algorithm. In the case of jobs that have irregularities in the number of eligible tasks during execution, the load balancer is activated to make the execution of that particular job and, as a consequence, of the workload as a whole more efficient.

7.2 Comparison Between Concurrent Gang and oblivious schedulers

The main result of this chapter is presented at theorem 7. It states that oblivious schedulers can not do better than Concurrent Gang scheduler when employing the same task distribution strategy. By that we mean that each task t_i from the set of K tasks present at the machine at time T is allocated to the same processor for all schedulers.

Theorem 7 *Oblivious schedulers, such as Gang scheduler and local scheduler, can do no better than Concurrent Gang for the same task distribution strategy, using schedule length as metric.*

Proof 5 *We should prove that, using the terminology defined in chapter 4, PT^{space} and PT^{idle} of Concurrent Gang is smaller than or equal to a Gang scheduler and a oblivious local scheduler for any workload W on every processor for a given task distribution. In order to do so, we minimize the schedule length on every processor. The following analysis considers one processor at each time. Given S_i, the scheduler length on processor i, the overall schedule length is equal to* $\max(S_1, S_2, ..., S_i, ..., S_P)$

- *Gang scheduler - Since Gang schedulers have no solution to the task blocking problem, when a task blocks the processor becomes idle. In the case of Concurrent Gang, the processor only becomes idle if there is no more tasks to schedule. The total idle time in both schedulers has two components : PT^{space} and PT^{idle}. Let's consider the behavior of both schedulers for each case :*

  $PT_{cg}^{space} \leq PT_{Gang}^{space}$ *- This is true as the Concurrent Gang scheduler will always try to schedule a ready task on a idle slot due to the packing strategy*

  $PT_{cg}^{idle} \leq PT_{Gang}^{idle}$ *- This is due to the fact that the local scheduler of Concurrent Gang detects that a task is blocked (due to I/O for instance), and will try to schedule a ready task to fulfill the inactivity period of a processor.*

– *Oblivious local scheduler - Consider a processor P where the same set of tasks T is allocated for both the local scheduler and the Concurrent Gang scheduler. Our objective is to minimize* P^{idle}*, in order to minimize the schedule length on that particular processor. Let's consider the moment when a given task, T, blocks. Both the Concurrent Gang scheduler and the oblivious local scheduler will try to schedule another task. However, as the oblivious scheduler does not have any information about the internal characteristics of the tasks, it will schedule a task in function of his internal list of priority, which contains no information regarding the internal behavior of the task. The difference of* P^{idle} *between an oblivious local scheduler and Concurrent Gang is mainly due to active waiting by a process, which increases the schedule length on that processor. Two cases are relevant :*

  – Workload composed of tasks with no interactions between themselves - *An example of such workload is one composed by embarrassingly parallel and I/O bound jobs only. The performance in this case of the oblivious local scheduler and the Concurrent Gang Scheduler are quite similar, since both will try to schedule a ready task when the running task blocks, giving priority for I/O bound jobs*

  – Workload with fine grain synchronization/communication jobs - *In this case a Concurrent Gang Scheduler may have a significant advantage over a oblivious local scheduler. The problem of locally scheduling fine grain synchronization jobs becomes worse as the load on the node increases. In [37] it was proved that the time for a single iteration of a fine grain synchronization job using uncoordinated local scheduling is* l^{n-1} *slower than Gang scheduling, where* l *is the load in one processor and* n *is the number of synchronizing tasks. In this case the barrier was implemented using spin only busy wait and the total load was evenly distributed among processors. This wait increases the schedule length on all processors where that particular job is being scheduled. The waste of computing cycles caused by the uncoordinated scheduling varies depending on the implementation of the busy wait mechanism. For spin only, the task spins until the end of time quantum, and the waste of computing cycles can be significant, as mentioned in [37]. This waste can be bounded by using a two phase blocking mecha-*

nism, but even in this case it cannot improve the response time to the level achieved by the Gang scheduler [37]. Finally, a block mechanism can reduce further the waste of computing cycles. But even in this case the local scheduler can do no better than Concurrent Gang due to the increased number of preemptions caused by the non-coordinated scheduling, which hurts the performance of synchronization intensive jobs.

A special case occurs when there are multiple synchronization intensive tasks allocated in the same processor. Similar to [76], let's consider the case where there is only one fine grain synchronization job on the machine, and that a task blocks when reaching a barrier. The number of tasks in this case is much larger than the number of processors. Let the time for thread i on processor j to reach the barrier be t_{ij}. In local scheduling, the time for one barrier to complete is :

$$T_m = \sum_{i=1}^{m} \max_{j=1}^{n}(t_{ij}) \qquad (7.1)$$

Which will be the same for a Concurrent Gang scheduler if no lower bound is defined for the priority. In this case, the Concurrent Gang scheduler will try to schedule another task even if the available tasks have low priorities. It will not schedule a task only when there is no task available, like the oblivious local scheduler. So, in this case, the behavior of both schedulers will be very similar, which implies that equation 7.1 is valid for both.

7.3 Performance of Concurrent Gang in the presence of Irregular Jobs

Parallel programs can be broadly classified into regular and irregular. In regular programs the amount of parallelism remains constant throughout its execution. Examples of regular parallel programs are : FFT algorithms (In [13] Cooley and Tukey present an FFT algorithm that can be easily programmed into data-parallel form), linear algebra algorithms (e.g. matrix-vector product), computational geometry algorithms (e.g. convex hull), graph algorithms (e.g. minimum spanning tree) [7].

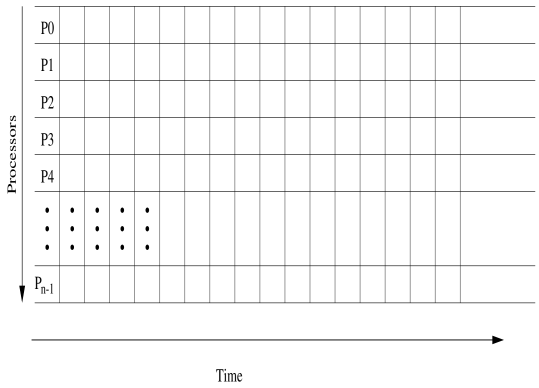

FIG. 7.1: X and Y irregularities

On the other hand, irregular programs change their degree of parallelism (or amount of computation per task) over the course of its execution. Examples include back track searches, theorem proving, ray tracing, monte carlo methods (e.g. to evaluate Feynman path integrals) [92]. In this thesis, we will use the same classification of parallel irregularity as defined in [76]. This classification follows the geometric definitions based on the trace diagram introduced in chapter 4. The irregularity of a program can be expressed in :

- Variation in the amount of computation performed per task during execution. We will define this variation as X irregularity.
- Variation in the number of tasks, that is, the job's degree of parallelism varies during execution. We will term this variation as Y-irregularity.

The concept of X and Y irregularities are illustrated in the trace diagram in figure 7.1. The variation on the number of eligible tasks of a job, that can be viewed as variations in the number of processors the job is actually using, may be visualized through variations in the Y axis. It is possible to view variation of computing power used by a task by following its progression in the X axis.

A program that presents both X and Y irregularities is called a completely irregular program. Most irregular jobs that appear in practice have been shown to display a similar tree structure [52]. By using information gathered at runtime, it is possible to efficiently run irregular jobs using Concurrent Gang.

7.3.1 Y-irregularity and Concurrent Gang

In Concurrent Gang Y irregularity is partly compensated by the local task scheduler, which tries to schedule another task when a slot dedicated to a job is idle due to job irregularity. Upon the creation of new tasks, those tasks will be spawned in idle slots dedicated to the job to which the spawning tasks belongs. Initially is assumed that the number of required processors in Concurrent Gang is the maximum number of tasks during the execution of a job.

For the case where the number of allocated processors for a job J is equal to K, and the number of tasks belonging to this job becomes larger than K , we propose a variation of Concurrent Gang that includes a load balancing phase. Observe that, in most parallel machines, the user requires a given number of processors at submission time to run a job and this number can not change during execution. If there is more tasks than processors allocated, in many sites the scheduler is not allowed to increase the number of processors allocated to that particular job. An idea to solve this problem is to integrate a load balancing strategy to the Concurrent Gang scheduler. Our goal is to keep one basic property of Gang scheduling, i.e. the job J has the illusion of being served by a machine of K nodes, and simultaneously try to balance the number of tasks among those K nodes. Observe that all tasks newly created become eligible to be scheduled by the local task scheduler either on idle slots in the PE they are allocated in or when a task belonging to another job blocks. But in a slot associated with a irregular job, tasks belonging to other jobs will only be scheduled if all tasks associated with job J are blocked. The slot associated with job J in a processor will be shared among all tasks belonging to that job. Note that only the set of processors belonging to job J's slice will participate in the load balancing for this particular job. That means that Concurrent Gang will implement a variation of family scheduling for those jobs where the number of tasks becomes larger than the partition dedicated to that job.

Observe that when the number of tasks becomes larger than the number of processors dedicated to a job, them semantics of the barrier call in case of a synchronization intensive program will change automatically from spin or spin block to blocking, since one should schedule all tasks as soon as possible in order to pass the barrier. This situation can be detected by the runtime measurement system, by identifying the synchronization intensive tasks belonging to the same job in the same processor.

Defining a load balancing strategy In order to do load balancing in a set of processors associated with a job, many strategies are possible, such as receiver-initiated [30], sender initiated [30] and rate of change load balancing [9]. Note that the load balancing in this case will involve only tasks belonging to the irregular job, and it will be done only among those processors associated with that job. The ideal strategy would have the following characteristics :

- Will leave no processor idle when there are a sufficient number of tasks.
- Each task ends up with either $\lceil T/K \rceil$ or $\lfloor T/K \rfloor$ tasks.
- In order to improve scalability, redistribution of tasks should be based only on local knowledge.

Integrating a load balancing strategy with Concurrent Gang Once that one load balancing strategy is chosen, it is necessary to integrate the load balancing algorithm with Concurrent Gang in way that the execution constraints imposed by Concurrent Gang are respected. One possible way of integration is to enable the load balancing at the job level. In Concurrent Gang, each job has a number of processors allocated to in a slice. The load balancing for tasks of a given job can be done among those processors. Eventually a job may have more than one task allocated on one processor if it has a strong Y-irregular behavior. In this case, the scheduler will do a two-level scheduling for that job on its slice through family scheduling. It is clear that additional tasks allocated on the processor may benefit from idle slots or idle time due to task blocking through the local task scheduler/priority mechanism, as all other tasks in the same processor.

7.3.2 X-irregularity and Concurrent Gang

X irregularities in Concurrent Gang are compensated by the local task scheduler that will try to schedule a ready task each time a task block due to, for instance, a barrier.

We may consider the control of the spin time of a task in order to reduce imbalances due to X-irregularity, as we did with Y-irregularities. Runtime measurements can be used to control the spin time of a task. For instance, when there is multiple synchronization intensive tasks from one job in one processor, the spin time should be set to zero in order to minimize the time required to go through the barrier. Another use of runtime measurements to control the spin time will be detailed in the next chapter. In that case, the spin time is set depending on the total workload on a node, and can be used to improve the throughput in the case of frequent imbalances.

7.3.3 Handling completely irregular programs

The irregularities in a completely irregular program is a combination of X and Y irregularities. A completely irregular program will required the combination of strategies used for X-irregular and Y-irregular program in order to achieve efficient execution.

7.4 Conclusion

In this chapter we presented an analysis comparing Concurrent Gang against Local and Gang schedulers. We proved that oblivious schedulers can not do better than Concurrent Gang when using schedule length as metric, for the same distribution of tasks. We also made an analysis of the behavior of Concurrent Gang under an irregular workload, and proposed the integration of a load balancing algorithm with Concurrent Gang to improve the execution efficiency of a Y-irregular workload.

Chapitre 8

Runtime measurements in parallel job scheduling

8.1 Introduction

In this chapter we systematize and make a deeper analysis of the utilization of runtime information in parallel job scheduling, introduced in chapter 6, to improve throughput and utilization on a parallel computer. Our objective is to use information such as number of I/O calls, duration of I/O calls, number of messages arrived, number of messages sent, number of barriers, time spent in spinning while waiting for message/synchronization arrival and other information available depending on the architecture in order to associate a specific task in a given moment of time to one class belonging to a set of predefined classes with the help of fuzzy sets and Bayesian estimators. Observe that the classification of a task may change over time, since we consider, as in [17], that characteristics of jobs may change during execution.

Some possible uses for the task classification information are, for instance, to decide which task to schedule next (as described in chapter 6), to decide what to do in the case of an idle slot in Gang scheduling, or to define spinning time of a task depending on the total workload on a processor. One possible utilization of these concepts is to give better service to I/O bound jobs in Gang scheduling, by using task classification to identify I/O bound tasks either to reschedule them in idle slots (as discussed in chapter 6) or to control the spin time of communication/synchronization tasks to give better service to interactive jobs. This approach is different from the one proposed in Lee et al. [58] since it does not interrupt running jobs.

In section 8.2 we discuss some previous work in parallel/distributed job scheduling that considers the use of runtime information to modify scheduling-related parameters at runtime. Section 8.3 presents the task classification mechanism based on runtime information we use in this chapter. How to use this information to improve throughput and utilization in parallel job scheduling is discussed at section 8.4. Our experimental results are presented and discussed in section 8.5 and section 8.6 contains our final remarks.

8.2 Previous Work

In [2], Arpaci-Dusseau, Culler and Mainwaring use information available at run time (in this case the number of incoming messages) to decide if a task should continue to spin or block in the pairwise cost benefit analysis in the implicit cosheduling algorithm.

In [38], Feitelson and Rudolph used runtime information to identify activity working sets, i.e. the set of activities (tasks) that should be scheduled together, through the monitoring of the utilization pattern of *communication objects* by the activities. Their work can be considered complementary to ours in the sense that our objective here is not to identify activity working sets at runtime but to improve throughput and utilization of parallel machines for different scheduling strategies using such runtime information.

In [58], Lee et al., along with an analysis of I/O implications for Gang scheduled workloads, presented a method for runtime identification of gangedness, through the analysis of messaging statistics. It differs from our work in the sense that our objective is not to explicitly identify gangedness, but to provide a task classification, which may vary over time depending on the application, which can also be used to verify the gangedness of an application in a given moment of time among other possibilities.

8.3 Task Classification using Bayesian Estimators

The objective of this section is to introduce a more robust task classification mechanism than the one described in the chapter 6 using elements of Bayesian decision theory. Bayesian decision theory is a formal mathematical structure which guides a decision maker in choosing a course of action in the face of uncertainty about the consequences of that choice [49]. In particular we will be interested in this section in defining a task classifier using a Bayesian estimator adapted to the fuzzy theory.

A Bayesian model is a statistical description of an estimation problem which has two main components. The first component, the prior model $p(u)$ (this probability function is also known as prior probability distribution) is a probabilistic description of the world or its properties before any sense data is collected. The second component, the sensor model $p(d|u)$, is a description of the noisy or stochastic process that relate the original (unknown) state u to the sampled input image or sensor values d. These two probabilistic models can be combined to obtain a posterior model, $p(u|d)$ (posterior probability distribution), which is the probabilistic description of the current estimate of u given the data d. To compute the posterior model we use Bayes' rule :

$$p(u|d) = \frac{p(d|u)p(u)}{p(d)} \tag{8.1}$$

where

$$p(d) = \sum_u p(d|u)p(u) \tag{8.2}$$

The fuzzy version of equation 8.1 to compute the degree of membership of a task T to a class i as a function of measurement E can be written as [56] :

$$S_E(i) = \frac{S_i(E)f_i(T)}{\sum_1^k S_j(E)f_j(T)} \tag{8.3}$$

Where $S_j(k)$ represents subsethood between two fuzzy sets j and k. In our case $S_i(E)$ is the subsethood between the two fuzzy sets represented by measurement E on task T and class i, that is, the degree of membership of task T relative to class i considering only the data gathered at measurement E. $f_i(T)$ is the degree of membership of task T relative to class i before measurement E. $S_E(i)$ in our case represents the degree of membership of task T relative to class i after the measurement E and becomes $f_i(T)$ in the next interval computation.

8.3.1 Overhead Analysis of Task Classification Computation

It is possible to compute the degree of membership of a task using equation 8.3 with very low overhead and a small amount of stored information by the operating system, based on the number of executed statements related to a task. Here we use a variation of the measurement method proposed at section 6.2.2, adapted to use Bayesian estimators : at initialization the degree of membership of a task related to each class is equal to 1/2, as well as the measurement related to each class. The measurement related to a time slice is made as follows (in the following, we consider three classes) :

- If the task is blocked due to an I/O call in that time slice, the measurement of the degree of membership of the class I/O bound is equal to 1/2. Otherwise is equal to zero.

- If the task executes a number N of communication statements in a time slice, the measurement of the degree of membership of the class communication intensive is equal to 1/2. Otherwise is equal to zero.

- If the task is not blocked due to an I/O call and executes less than N communication statements, the measurement of the degree of membership of the class computation intensive for that time slice is equal to 1/2. Otherwise is equal to zero.

The measurement for an interval is then summed to the previous total measurement multiplied by 1/2, becoming the new total measurement for a given class. Observe that the total measurement related to each class is a real number between]0,1[. Having the total measurement for all classes and the degree of membership of a task to all predefined classes, it is possible to compute equation 8.3. Note that the result of equation 8.3 becomes the new degree of membership of a task T to a class C. Observe that the only overhead associated with measurement is to count the number of communication statements that are associated with a class in a time slice. In the case of an I/O statement, as the task will block anyway, there is no overhead associated with the measure.

8.4 Using task classification in Parallel Job Scheduling

In this section we will describe some possible uses of task classification information and we show how to apply these concepts into another implementation of Gang scheduling, the distributed hierarchical control algorithm

8.4.1 Scheduling Using Runtime measurements

The objective of this section is to systematize the task classification scheme first described in chapter 6 in a way that it may be used by any parallel scheduler.

When using task classification information, the local task scheduler on each PE computes a priority for each task allocated to the PE. This priority defines if a task T is a good candidate for being rescheduled if another task blocks or in case of a idle slot. The priority of each task is defined based on the degree of membership of a task to each one of the major classes described before. As an example of the computation of the priority of a task T in a PE we have [86] :

$$Pr(T) = max(\alpha \times (f_{IO} - f_{COMM}), f_{COMP}) \tag{8.4}$$

Where $f_{IO}, f_{COMP}, f_{COMM}$ are the degree for membership of task T to the classes I/O intensive, Computation intensive and Communication intensive. The objective of the parameter α is to give greater priority to I/O bound jobs ($\alpha > 1$), since was proved by Lee et al. [58] that I/O bound jobs suffer under Gang scheduling. The choices made in equation 8.4 intend to give high priority to I/O intensive jobs (those that are not at the same time communication intensive) and computation intensive jobs, since such jobs can benefit the most from uncoordinated scheduling. The multiplication factor α for the class I/O intensive gives higher priority to I/O bound tasks over computation intensive tasks, since those jobs have a greater probably to block when scheduled than computing bound tasks. By other side, communication and synchronization intensive jobs have low priority since they require coordinated scheduling to achieve efficient execution and machine utilization [37, 31]. A communication intensive phase will reflect negatively over the degree of membership of the class computation intensive (a communication intensive phase will have a negative impact over the measurement related to the class computing intensive, when using the measurement strategy described in the previous section), reducing the possibility of a task be scheduled by the local task scheduler. Among a set of tasks of the same priority, the local task scheduler uses a round robin strategy. The local task scheduler may also define a minimum priority β. If no parallel task has priority larger than β, the local task scheduler considers that all tasks in the PE do intensive communication and or synchronization, thus requiring coordinated scheduling. Observe that there is no starvation of communication intensive jobs, as they will be scheduled in a regular basis by the Gang scheduler itself, regardless of the decisions made by the local task schedulers.

Observe that the parameters α and β define the bounds of the variation of the priority of a task in order to it be considered to rescheduling, as stated in the next proposition.

Proposition 1 *$\alpha \leq Pr(T) \leq \beta$, in order to a task be considered for rescheduling.*

Proof - β is the lower bound by definition. For the upper bound, observe that $f_{IO}^{max} = 1$. So, as $\alpha > 1$, the upper bound is $\alpha \times 1 = \alpha$

Interactive tasks can be regarded as a special type of I/O intensive task, where the task waits for a input from the user at regular intervals of time. These tasks also suffer under Gang scheduling, and should have priority as I/O intensive tasks.

8.4.2 Adjusting Spinning Time depending on the workload

Another parameter that can be adjusted in order to improve throughput of I/O bounds and interactive jobs in Gang scheduling is the spinning time of a task. Our objective is to make changes not only depending on the runtime measurements of the related job, but also considering other jobs where tasks are allocated to the same processor. We consider that a typical workload will be composed of a mix of jobs of different types and it is important to achieve a compromise in order to give a good response for all types of jobs.

The anticipated blocking of a job performing synchronization or communication can benefit those jobs that do not need coordinated scheduling, such as I/O intensive and embarrassingly parallel. So the idea is to determine the spinning time of a task depending on the workload allocated in a processor.

For instance, in a given moment of time if a processor has many I/O intensive jobs allocated to it, this would have a negative impact in spinning time duration. As described in [2], a minimum spin time should be guaranteed in order to insure that processes stay coordinated if already in such a state (baseline spin time). This minimum amount of time ensures the completion of the communication operation when all involved processes are scheduled and there is no load imbalance among tasks of the same job.

Considering Gang scheduling the spinning time of a task may vary between a baseline spin time and a spin only state with no blocking. The main external factor that will have influence in the variation of the spin time is the number of interactive and I/O bound tasks in the workload allocated to one processor. A large number of these tasks would imply a smaller spinning time, in order to use the remaining time until the next global preemption to schedule those tasks, providing better service to I/O bound and interactive tasks. The algorithm we propose to set up the spinning time as a function of the workload on a given PE for a Gang scheduling based algorithm (such as Concurrent Gang) is as follows : If there is one or more tasks in a PE classified as I/O intensive or interactive, a task doing communication will block just after the baseline spin time if the two following conditions are satisfied :

- At least one of the tasks classified as interactive or I/O bound is ready
- There is a minimum amount of time δ between the end of baseline and the next context switch epoch.

If any of the two conditions are not satisfied the task doing communication will spin until receiving the waited response. The δ time depends on the context switch time of the machine. Given γ, the context switch time of the machine, it is clear that $\delta > \gamma$. We can define that $\delta > 2 \times \gamma$, in order to give the job at least the same amount of CPU time that the system will spend in context switch. In our experiments we empirically define it as being 4 times the average amount of time required for a context switch.

If both conditions are satisfied, the tasks will spin for a time corresponding to the baseline spin time, and if no message is received the task blocks and the I/O bound or interactive task can be scheduled. The reason of minimizing the spinning time is the need of I/O and interactive tasks to receive better service in Gang scheduling, and the fact that in Gang scheduling tasks are coordinated due to the scheduling strategy itself; so an application with no load imbalances would need only the time corresponding the baseline to complete the communication.

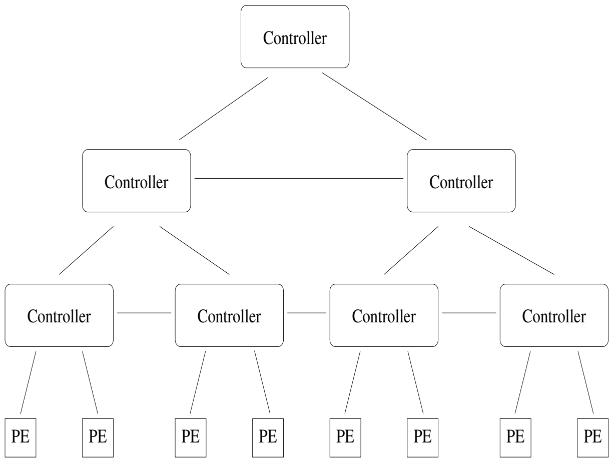

FIG. 8.1: Controller in the Distributed Hierarchical Control scheme

The control of spin time using task classification information is another mechanism available to the scheduler to provide better service to I/O bound and interactive jobs under Gang scheduling based strategies along with the priority computation described in the previous section. Observe that the spin time control depending on the workload is always used in conjunction with the priority mechanism described in section 8.4.

8.4.3 Distributed Hierarchical Control with alternative scheduling

As an example of the use of the task classification mechanism with scheduling algorithms other than Concurrent Gang, in this subsection we integrate scheduling mechanisms based on information gathered at runtime to the distributed hierarchical control algorithm. The distributed hierarchical control (DHC) defines a control structure over the parallel machine and combines time-slicing with a buddy-system partitioning scheme, as described in chapter 3. A schematic representation of such structure for a machine with 8 processors is showed in figure 8.1. In [36] a DHC scheme for supporting Gang scheduling was proposed. Some characteristics of DHC is the utilization of preemption through the Gang scheduling of related tasks, load balancing among controllers, and no migration support. A full description of the distributed hierarchical control was provided in chapter 3.

One optimization of the DHC algorithm of fundamental importance is the use of alternative scheduling. In alternative scheduling, jobs are allowed to run in alternative slots, different from the ones which have been originally assigned to it. In [33], Feitelson proved that a buddy system with alternative scheduling performs better than other classical packing schemes such as best fit and first fit.

Intelligent Alternative Scheduling

The controllers in the DHC algorithm can be used in conjunction with the task classification mechanism to provide an intelligent choice of which job should be scheduled in alternative scheduling when we have a slot composed of idle processors and more than one job to be scheduled in that slot. Since the controller may have access to the task classification information of each one of the processors it controls, the controller can make use of that information to decide what to do with the idle slot. We have three objectives when defining an Intelligent Alternative Scheduling, in order :

- *Improve service given to I/O bound and interactive jobs* The idea is to try to schedule the job with the largest number of I/O bound tasks, since those jobs suffer under Gang scheduling. Since those tasks normally run for a small amount of time and then block again, giving priority to those tasks will allow the scheduler to give better service to these jobs and reschedule other tasks when I/O bound tasks block.
- *Improve service given to Communication/Synchronization intensive jobs* If there are not I/O bound jobs ready to be scheduled, and if the number of processors is enough, we try to scheduler a communication/synchronization intensive job. Since these jobs require coordinated scheduling, an idle slot with enough processors available is a good opportunity for give better service to those jobs, as they normally do not have a high priority for rescheduling in case of task blocking, if compared to I/O and embarrassingly parallel jobs.
- *Improve overall utilization and throughput* Third, we try to schedule individual tasks depending on their priority. If there is no sufficient number of ready tasks available, the controller yield control of the idle processor to the lower level controller.

8.5 Experimental Results

In this section we present some simulation results that compares the performance of a Gang scheduler that uses the algorithms described in sections 8.4 and 8.4.2 with another Gang scheduler without such mechanisms, both of them using the same packing strategy (first fit). Our objective is to measure the benefits of using runtime measurements and task classification information by comparing a given scheduler that makes use of runtime information with another one that does not consider it. First we describe our simulation methodology, and then we present and comment the results obtained in our simulations. Simulations using the DHC algorithm are presented in subsection 8.5.3.

8.5.1 Simulation Methodology

To perform the actual experiments we used the simulator already described at chapter 6. This is a general purpose event driven simulator being developed by our research group for studying a variety of problems (e.g., dynamic scheduling, load balancing, etc).

We have modeled in our simulations a network of workstations connected by a network characterized by LogP [25, 24] parameters. The LogP parameters corresponds to those of a Myrinet network, and they were the similar to the ones used in [2], with Latency being equal to 10 μs, and overhead to 8.75 μs. We defined the baseline spin time as being equal to a request-response message pair, which in the LogP model is equal to 2L+4o. Therefore, the baseline time is equal to 55 μs. The number of processors considered were 8 and 16. I/O requests of a job were directed to the local disk of each workstation, and consecutive requests were executed on a first come first serve basis. Quantum size is fixed as being equal to 200 ms and context switch time equal to 200 μs.

The values of the α and β parameters used for simulations were $\alpha = 2$ and $\beta = 0.3$. As stated in proposition 1 the priority should vary inside the bounds defined by α and β in order to a task be considered to reschedule.

For defining job inter arrival, time, job size and job duration we used a statistical model proposed in [27], which was already described at chapters 5 and 6. For the simulations for a 16 processors machine we used 5 ranges of degrees of parallelism, and for a 8 processors machine 4 ranges, as the defined ranges are 1, 2, 3-4, 5-8, 9-16, 17-32, 33-64, 65-128, 129-256 and 257-322. The time unit of the parameters found in [27] was seconds, and the duration of all simulations was defined as being equal to 50000 seconds. A number of jobs are submitted during this period in function of the inter arrival time, but not necessarily all submitted jobs are completed by the end of simulation. A long time was chosen in order to minimize the influence of start-up effects.

In order to avoid the saturation of the machine, we limited the number of tasks that can be allocated to a node at a given moment of time to 10. If a job arrives and there is no set of processors available with less than 10 tasks allocated to them, the task waits until the required number of processors become available.

We use a mix of four types of synthetic applications in our experiments. These classes are similar to the ones defined in chapter 6 :

- *I/O* - This job type is composed of bursts of local computations followed by bursts of I/O commands, as represented in figure 8.2. This pattern reflects the I/O properties of many parallel programs, where execution behavior can be naturally partitioned into disjoint intervals, each of which consist of a single burst of I/O with a minimal amount of computation followed by a single burst of computation with a minimal amount of I/O [77]. The interval composed of a computation burst followed by an I/O burst are know as phases, and a sequence of consecutive phases that are statistically identical are defined as a working set. The execution behavior of an I/O bound program is therefore comprised as a sequence of I/O working sets. This general model of program behavior is consistent with results from measurement studies [88, 89]. The time duration of the I/O burst was equal to 100 ms in average. The ratio of the I/O working set used in simulations was 1/1, that is, for a burst of 100 ms of I/O there was a burst of 100 ms of computation in average. Observe that I/O requests from different jobs to the same disk are queued and served by arrival order.
- *Embarrassingly parallel* - In this kind of application constituent processes work independently with a small amount or no communication at all among them. Embarrassingly parallel applications require fair scheduling of the constituent processes, with no need for explicit coordinated scheduling.
- *Msg* - In this type of synthetic application we model message passing jobs, where messages are exchanged between two processes chosen at random. Each process sends or receives a message every 10 ms in av-

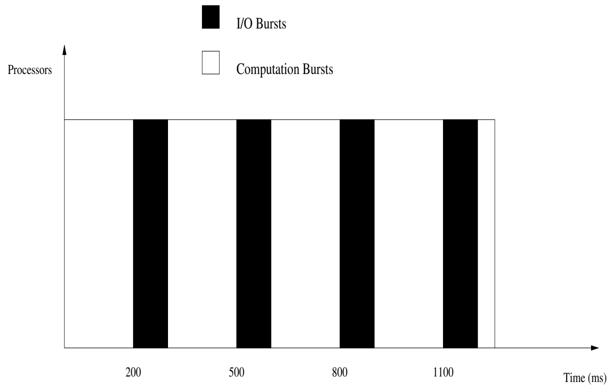

FIG. 8.2: I/O bound job with one I/O working set

erage. The communication semantics used here were the same of the PVM system [21], that is, asynchronous sends and blocking receives. For the modified version of Gang scheduler, the one that incorporates spin control and priority computation, the spinning time of the receive call will be defined by the spin control mechanism described in section 8.4.2. The pure Gang scheduler only implements the spin only mechanism, since the original Gang schedulers do not know what to do if a task blocks.

- *BSP* - This type of application models Bulk Synchronous Parallel (BSP) style jobs [94], where there is a sequence of supersteps, each superstep being composed of a mix of computation/communication statements, with all processes being synchronized between two supersteps. In this type of applications, there is a synchronization call every 50 ms (in average) and all communication/computation generated previous to the barrier call is completed before the job proceeds in the next computation/communication superstep. Again, there is a spin time associated with the barrier and communication calls.

In all simulations, the same sequence of jobs were submitted to both a Gang scheduler with the priority computation and spin time control mechanisms described in section 8.4 and 8.4.2 and another Gang scheduler without such mechanisms. A different sequence is generated for each experiment. The packing strategy was first fit without thread migration. Each workload was composed of a mix of the 4 types of jobs previously defined :

- *IO*- This workload was composed of I/O bound jobs only. As I/O bound jobs suffer under Gang scheduling, this workload was simulated in order to evaluate the performance impact of the modified Gang scheduler if compared against a traditional Gang scheduler.

- *IO/Msg* - This workload was composed of a mix of IO and Msg jobs. At each job arrival, the job type was chosen according with a uniform distribution, with a probability of 0.5 to both jobs
- *IO/BSP* - As in the previous workload, both job types had the same probability of being chosen at each job arrival.
- *IO/Msg/Embarrassingly* - Since the priority mechanisms intends to give better service to I/O bound and Compute intensive bounds, we included the Embarrassingly parallel type in the IO/Msg workload, to verify is there is any improvements in throughput due to the inclusion of computing intensive jobs.
- *IO/BSP/Embarrassingly* - Same case for the IO/BSP workload. As in previous cases, at each job arrival all three job types have equal probability to be chosen.
- *Emb/Msg and Emb/BSP* - These workloads were added to evaluate the impact of the priority mechanism over workloads that do not include I/O bound jobs. They are composed of Embarrassingly parallel jobs with Msg and BSP job types respectively. In this case the spin control is not activated since it is conceived to provide better service to I/O bound and interactive tasks only, as these are the type of jobs that have poor performance under Gang scheduling.

A second set of experiments were performed using the workloads IO/BSP and IO/Msg to compare the performance of a Gang scheduler with both the priority computation and spin control mechanisms with another Gang scheduler having only the priority control mechanism in order to evaluate the impact of the spin control in the results presented.

8.5.2 Simulation Results

Simulations results for the IO workload are shown in figure 8.3. In the utilization column, the machine utilization (computed as a function of the total idle time of the machine on each simulation) of the modified Gang scheduler was divided by the machine utilization of the non-modified version of the Gang scheduler. In the throughput column, the throughput of the modified

Gang scheduler (The number of jobs completed until the end of the simulation, 50000 seconds) is divided by the throughput in the original Gang. We can see a very significant improvement of the modified Gang over the original Gang scheduler, due to the priority mechanism. To explain the reason of such improvement, tables 8.1 and 8.2 show the actual results of simulations for 8 and 16 processors machines under the I/O bound workload. In [77] , Rosti et al. suggest that that the overlapping of the I/O demands of some jobs with the computational demands of other jobs may offer a potential improvement in performance. As in chapter 6, the improvement shown in figure 8.3 is due to this overlapping. The detection of I/O intensive tasks and the immediate scheduling of one of these tasks when another task doing I/O blocks results in a more efficient utilization of both disk and CPU resources. As we consider an I/O working set composed by a burst of 100 ms of computation followed by another burst of 100 ms of I/O, the scheduler implementing the priority mechanism always tries to overlap the I/O phase of a job with the computation phase of another, which explains the results obtained. In the ideal case, the scheduling strategy will be able to interleave the execution of applications such that the ratio of the per-phase computation and I/O requirements is maintained very close to 1, thus achieving a total overlapping of computation and I/O. For this workload, since the utilization of the machine is doubled by using runtime information, we can conclude that the overlap of I/O phase is almost 100%, since the duration of the I/O phase is in average equal to the duration of the computation phase and the utilization obtained for the Gang scheduler without runtime information is due only to the computation phase. The differences between throughput and utilization are due to contention on local disks and not completed tasks. Another interesting point is that, in both machines, about half of the completed jobs were 1 task jobs, since a large amount of jobs generated by the workload model were 1 task jobs.

Comparing the results of figure 8.3 with those of chapter 6, we conclude that the results related to the I/O bound workload in this chapter are expected. The compared performance is reduced since the duration of both the I/O burst and the computation burst doubled when compared against simulation parameters of chapter 6. The difference in absolute values for number of completed jobs is related with the increased simulation time and different workload characteristics.

For the IO/Msg workload, results are shown in figure 8.4. Again, the modified Gang achieved better results for both throughput and utilization. Since

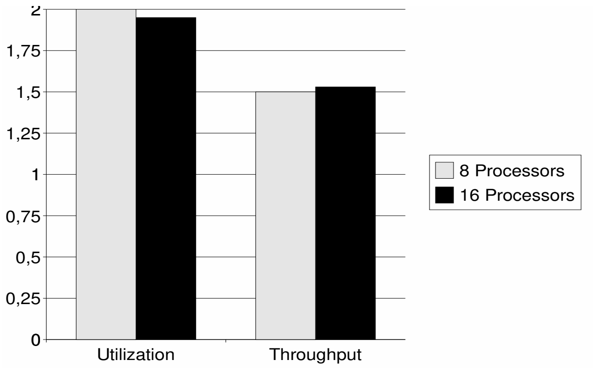

FIG. 8.3: I/O bound workload with one I/O working set

TAB. 8.1: Experimental results - I/O intensive workload - 8 Processors

8 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	60	84
Without Runtime Information	40	42

Gang schedulers have good performance for communication bound jobs, the improvement due the utilization of runtime measurements and task classification is smaller if compared against the results obtained for the IO workload, as the machine utilization of the Gang scheduler without runtime information is better in this case if compared against the results related to the previous workload. Tables 8.3 and 8.4 show the absolute machine utilization for the experiments using the IO/Msg workload. As the machine utilization for the regular Gang scheduler is around 60%, an improvement in utilization as observed with the IO workload is no longer possible.

Results for the IO/Msg/Emb workload are shown in figure 8.5. The greater flexibility of the modified Gang algorithm to deal with I/O intensive and embarrassingly parallel jobs results in an increase in throughput and

TAB. 8.2: Experimental results - I/O intensive workload - 16 Processors

16 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	55	84
Without Runtime Information	36	43

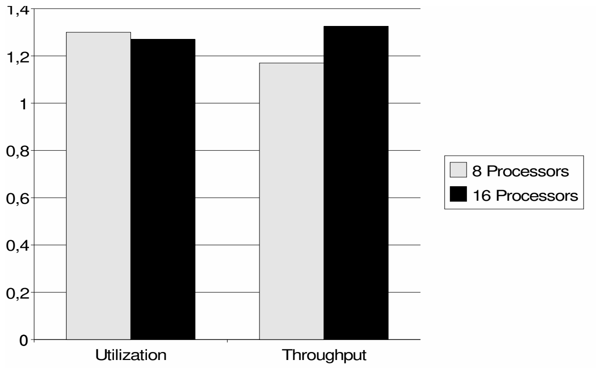

FIG. 8.4: IO/Msg workload

TAB. 8.3: Experimental results - IO/Msg workload - 8 Processors

8 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	50	82
Without Runtime Information	43	63

utilization. It is worth noting, however, that the influence of idle time due to I/O bound jobs is reduced, with the regular Gang scheduler having even better machine utilization if compared against results for the IO/Msg workload, as shown in tables 8.5 and 8.6.

When we substitute the Msg workload for the BSP workload in the previous experiments, results are similar in both relative and absolute values. The reason is that both types of jobs are communication/synchronization intensive, taking advantage of the Gang scheduling strategy. Results for IO/BSP and IO/BSP/Emb workloads are shown in figures 8.6 and 8.7 respectively. As in previous cases, there is improvement over the Gang scheduler without the priority computation and spin control mechanisms in both utilization and throughput. Again, the combination of the priority and spin control mech-

TAB. 8.4: Experimental results - IO/Msg workload - 16 Processors

16 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	53	79
Without Runtime Information	40	62

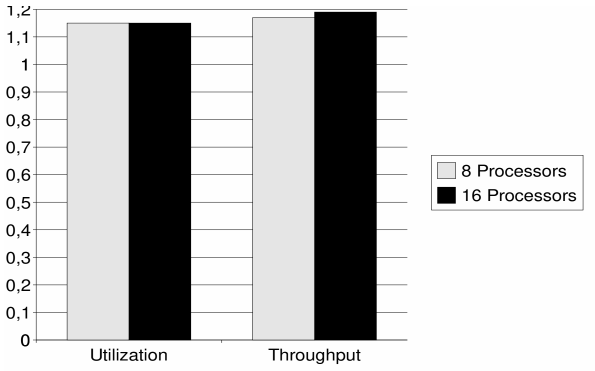

FIG. 8.5: IO/Msg/Emb workload

TAB. 8.5: Experimental results - I0/Msg/Emb workload - 8 Processors

8 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	47	83
Without Runtime Information	40	72

anisms explains the better results obtained by the scheduler using runtime measurements for both workloads.

To evaluate the impact of the spin control mechanism in the total performance of the modified Gang scheduler, we compared the performance between a modified Gang with both the priority and spin control mechanisms and other version of the modified Gang where only the priority computation was active. Results for workloads IO/Bsp and IO/Msg are shown in figures 8.8 and 8.9 respectively. In figures 8.8 and 8.9 the performance of the scheduler with spin control and priority mechanism is divided by the performance of the Gang scheduler with the priority computation only. The gain in throughput is due to the better service provided to I/O bound jobs, while in utilization Gang scheduling with only the priority mechanism has slightly

TAB. 8.6: Experimental results - IO/Msg/Emb workload - 16 Processors

16 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	61	81
Without Runtime Information	51	70

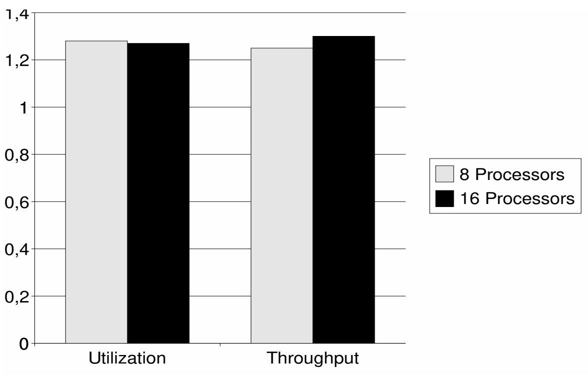

FIG. 8.6: IO/BSP workload

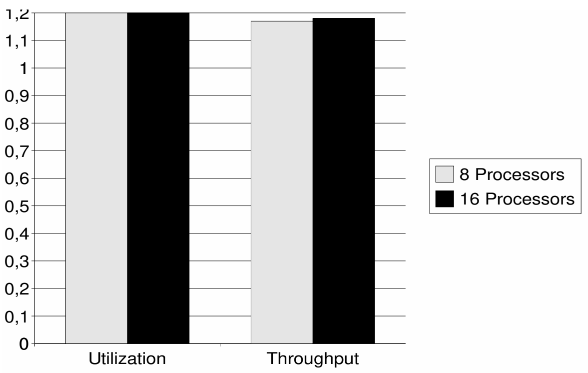

FIG. 8.7: IO/BSP/Emb workload

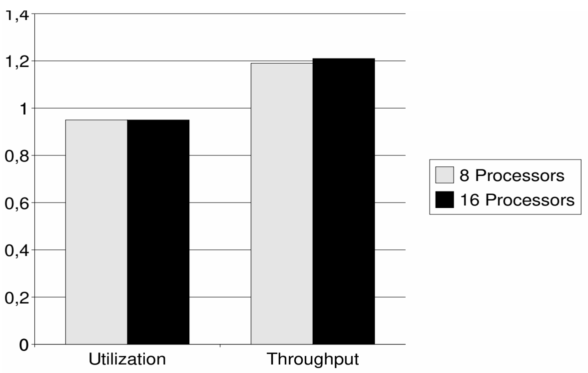

FIG. 8.8: Evaluation of the spin control mechanism - IO/BSP workload

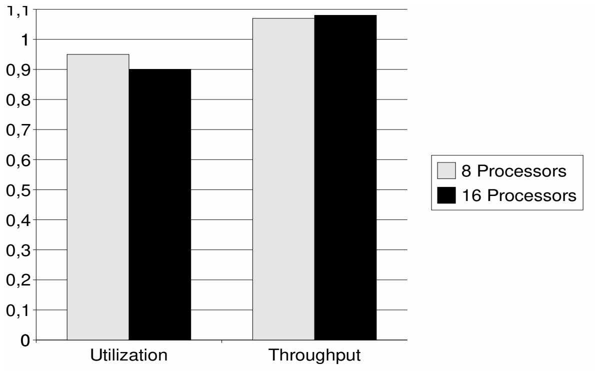

FIG. 8.9: Evaluation of the spin control mechanism - IO/Msg Workload

better performance. This can be explained by the fact the I/O bound jobs run for some time and then block again, while BSP and Msg jobs keep spinning and runs again after receiving the message. As said before, the objective of the spin control mechanism is to achieve a compromise in order to have a better performance for I/O intensive tasks, because these tasks suffer under Gang scheduling. In Gang scheduling with spin control and priority, this compromise is achieved by given a better a service to I/O bound jobs, having as consequence a reduction in the spin time of synchronization/communication intensive tasks.

To evaluate the performance impact for workloads with no I/O intensive jobs, we have simulated two workloads composed of embarrassingly parallel jobs with Msg and BSP jobs respectively. Comparative results are displayed in figures 8.10 and 8.11. Since Gang scheduling has a good performance for both synchronization and communication intensive jobs, the improvement is reduced if compared against the previous workloads. Observe that the performances of both the regular Gang scheduler and the Gang scheduler using runtime information are quite similar. The main improvement in these cases is in utilization and its due mainly to the scheduling of tasks belonging to embarrassingly parallel jobs on idle slots in the Ousterhout matrix [72], that is, those time slices where a processor do not has a parallel task to schedule.

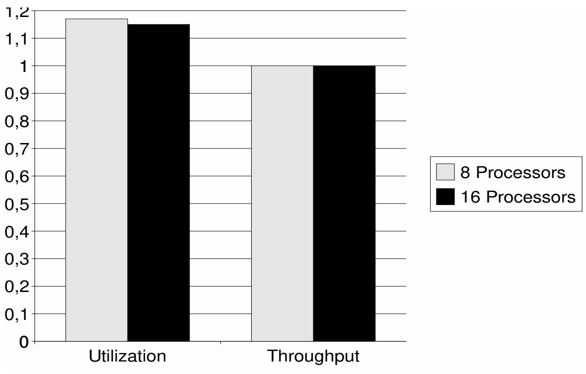

FIG. 8.10: Emb/Msg workload

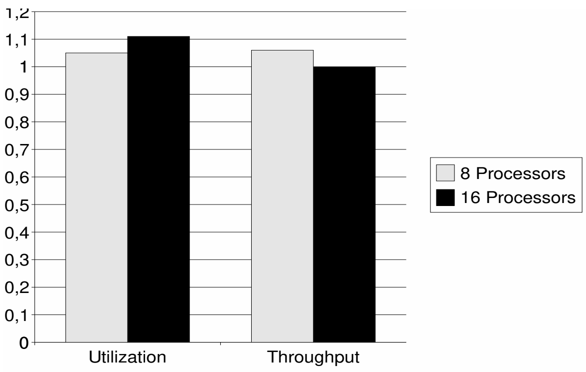

FIG. 8.11: Emb/BSP workload

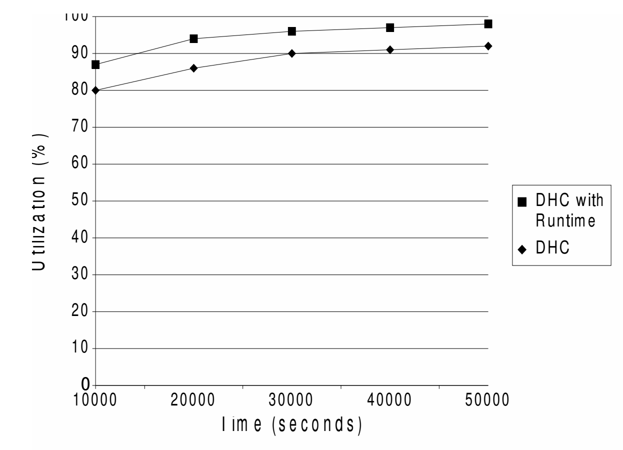

FIG. 8.12: Results for the computing intensive (embarrassingly parallel) workload

8.5.3 Simulations using the DHC Algorithm

In this subsection we compare two algorithms based on the DHC algorithm : One using runtime information, and other oblivious to that information. Results showing the variation of machine utilization over time of the two schedulers for an embarrassingly parallel workload and an I/O bound workload (using one I/O working set [77])are shown in figures 8.12 and 8.13 respectively. The machine in this case has 16 processors and the same simulation parameters described in the previous subsection. The improvement related to the embarrassingly parallel workload is small, since DHC with alternative scheduling does already a very good job in this case, with more the 90% of machine utilization for simulation time superior to 30000 seconds. The improvement observed is due to idle slots not used by DHC with alternative scheduling. When using runtime measurements, the controller of an idle processor is able to recognize tasks belonging to embarrassingly parallel jobs due to the task classification algorithm and then schedule those tasks in idle processors. However, when simulating I/O bound workload the improvement is significant, since these workloads suffer under gang scheduling, and DHC is based on gang scheduling.

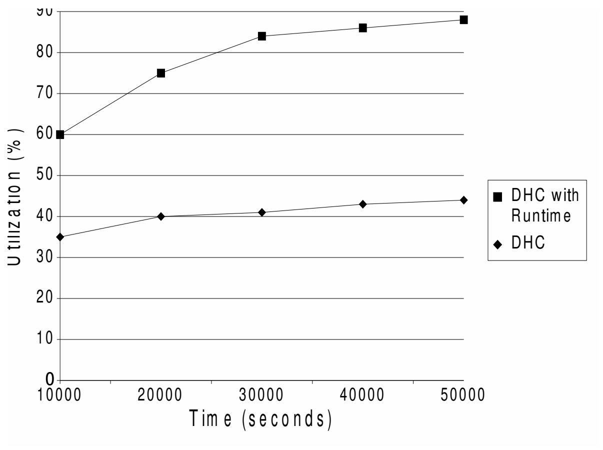

FIG. 8.13: Results for the I/O bound workload

8.6 Conclusion

In this chapter we present some possible uses of runtime measurements for improving throughput and utilization in parallel job scheduling. We believe that incorporating such information in parallel schedulers is a step in the right direction, since with more information available about running jobs in a given moment of time a scheduler will be able to do a intelligent choice about many events in parallel task scheduling, such as what task should have higher priority depending on the base scheduling algorithm used, how to change operating systems parameters in order to improve machine utilization, etc. The increase in throughput and utilization is confirmed by the experimental results we obtained.

However, there a number of possibilities not explored in this chapter that are subject of our current and future research. For instance, questions that we are investigating are the use of runtime information and task classification to improve parallel/distributed scheduling without explicit coordination, the utilization of task classification to identify gangedness of an application, and other ways of using task classification information to improve parallel job scheduling.

Chapitre 9

Conclusion

In the first part of this thesis, we presented the Gang scheduling algorithm and made several performance analysis related to it. In chapter 4 we make a competitive analysis of Gang scheduling using workload completion time as metric. The main result of this analysis is that the competitive ratio of Gang scheduling under completion time is no smaller than 4, being equal to 4 when the Gang scheduler implements a packing strategy based on first fit decreasing with support for job migration.

To provide a sound analysis of the performance of packing algorithms under Gang Scheduling, a novel methodology based on the traditional concept of competitive ratio was introduced in chapter 5. Dubbed dynamic competitive ratio, the new method is used to compare dynamic bin packing algorithms used in this paper. These packing algorithms apply to the Concurrent Gang scheduling of a workload generated by a statistical model. Moreover, dynamic competitive ratio is the figure of merit used to evaluate and compare packing strategies for job scheduling under multiple constraints. It was shown that for the unidimensional case there is a small difference between the performance of best fit and first fit ; first fit can hence be used without significant system degradation. For the multidimensional case, when memory is also considered, we concluded that the packing algorithm must try to balance the resource utilization in all dimensions simultaneously, instead of given priority to only one dimension of the problem.

In the second part of this thesis we propose a new scheduling policy based on Gang scheduling that takes advantage of idle times due to blocked tasks and idle slots. The Concurrent Gang improvement over Gang scheduling is more beneficial to workloads that require a more flexible scheduling than is possible with Gang scheduling. An example is I/O bound workloads, as was demonstrated with simulation results. For workloads requiring coordinated scheduled, the Concurrent Gang algorithm becomes equivalent to the standard Gang scheduler. Concurrent Gang schedule jobs belonging to this workload on the basis of whole process working sets, as in Gang scheduling.

In particular, the significant improvement of Concurrent Gang over Gang scheduling for I/O bound workloads is due manly to the overlapping of computation and I/O bursts from different jobs. The method of defining the degree of membership of a task related to different classes allows the scheduler to decide which task is more suitable to be scheduled at a given time.

In chapter 8 we presented other possible uses of runtime measurements for improving throughput and utilization in parallel job scheduling, as well a a more robust task classifier through the use of Bayesian estimators. Examples of how to use this information in other schedulers is also given. Again, we believe that incorporating such information in parallel schedulers is a step in the right direction, since with more information available about running jobs in a given moment of time a scheduler will be able to do an intelligent choice about different events in parallel job scheduling

9.1 List of Current Publications Related to this Thesis

The publications related to this thesis up to now are :

- Fabricio Silva and Isaac D. Scherson *Efficient Parallel Job Scheduling Using Gang Service* To appear in the International Journal of Foundations of Computer Science.
- Luis Miguel Campos, Mary M. Eshaghian, Isaac Scherson, Fabricio Silva *An Information Power Grid Resource Management Tool* To appear in the Ibero American Journal of Research "Computing and Systems"
- Fabricio Silva and Isaac D. Scherson *Improving Parallel Job Scheduling Using Runtime Measurements* Proceedings of the 6th Workshop on Job Scheduling Strategies for Parallel Processing , Cancun, Mexico, May 2000. An extended version of this paper is to appear in an special issue of Lecture Notes on Computer Science.

- Fabricio Silva and Isaac D. Scherson *Improving Throughput and Utilization in Parallel Machines Through Concurrent Gang* Proceedings of the IEEE International Parallel and Distributed Processing Symposium 2000, Cancun, Mexico, May 2000.
- Fabricio Silva and Isaac D. Scherson *Towards Flexibility and Scalability in Parallel Job Scheduling* 11th IASTED International Conference on Parallel and Distributed Computing and Systems, Boston, USA, November 1999.
- Fabricio Silva and Isaac D. Scherson *Concurrent Gang : Towards a Flexible and Scalable Gang Scheduler* 11th Symposium on Computer Architecture and High Performance Computing, Natal, Brazil, September 1999.
- Fabricio Silva and Isaac D. Scherson *Bounds on Gang Scheduling Algorithms* 2nd International Conference on Parallel Computing Systems, Ensenada, Mexico, August 1999.
- Fabricio Silva and Isaac D. Scherson *Improvements in Parallel Job Scheduling Using Gang Service* Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks - Freemantle, Australia - June 1999.
- Fabricio Silva, Luis Miguel Campos e Isaac D. Scherson *A Lower Bound for Dynamic Scheduling of Data Parallel Programs*, Proceedings of the 4th International Euro- Par Conference - Southampton, UK - September 1998.
- Fabricio Silva, Luis Miguel Campos e Isaac D. Scherson *Improvements in Gang Scheduling for Parallel Supercomputers*, Proceedings of the 8th International Parallel Computing Workshop - Singapore - September 1998.

Troisième partie

Conclusions et Discussion Finale

Chapitre 10

Conclusions

Dans cette conclusion, nous résumons d'abord nos résultats/contributions au domaine de l'ordonnancement parallèle chapitre par chapitre, avant d'introduire quelques pistes pour de nouvelles directions de recherche.

10.1 Chapitre 4

Le premier résultat obtenu dans ce chapitre est une borne inférieure pour le taux de compétitivité de l'ordonnancement Gang, comme indiqué dans le premier théorème du chapitre. Le taux de compétitivité est égal à 4 quand l'ordonnanceur Gang fait usage d'une stratégie de partitionnement du type "first fit decreasing" avec le support nécessaire pour la migration des tâches entre les processeurs. C'est la première fois, à notre connaissance, qu'une analyse de compétitivité indépendante du modèle de programmation pour l'ordonnancement Gang est proposée. Néanmoins Il faut rappeler que l'analyse de compétitivité est une analyse du pire cas. Par exemple, il est possible de visualiser quelques cas simples où le temps d'exécution de la charge de travail obtenu par l'ordonnanceur Gang est optimal, si des coûts liés à la préemption ne sont pas considérés.

10.2 Chapitre 5

Dans ce chapitre nous avons analysé des questions liées au partage de ressources pour les algorithmes d'ordonnancement Gang. Une conclusion de cette analyse est que le partage de ressources multidimensionnel est nécessaire pour définir une stratégie d'empaquetage pour les algorithmes d'ordonnancement Gang, comme le montre la comparaison entre les algorithmes "best fit" et "memory-fit". Une autre contribution de ce chapitre est la proposition du taux de compétitivité dynamique comme nouvelle méthode pour comparer des algorithmes dynamiques. Le taux de compétitivité dynamique a été utilisé pour comparer des algorithmes d'empaquetage soumis à une charge de travail générée par un modèle statistique, et pour comparer des stratégies d'empaquetage pour l'ordonnancement parallèle sous contraintes multiples. Pour le cas unidimensionnel nous pouvons conclure qu'il n'y a pas une grande différence entre les performances de "best fit" et "first fit" sous le modèle de charge de travail considéré, et "first fit" peut être utilisé sans dégradation significative de la performance. Pour le cas multidimensionnel, quand la mémoire est également considérée, la meilleure performance de la stratégie "memory fit" par rapport à la stratégie "best fit" démontre que l'algorithme d'empaquetage doit essayer d'équilibrer l'utilisation des ressources dans toutes les dimensions considérées, au lieu d'accorder la priorité à seulement une dimension du problème.

10.3 Chapitre 6

Dans ce chapitre nous avons présenté un nouvel algorithme d'ordonnancement nommé "Concurrent Gang". Les différences principales par rapport à l'ordonnancement Gang standard sont la définition explicite d'un synchroniseur global externe et la présence de l'ordonnanceur de tâche local qui décide quoi faire si une tâche de l'application exécutée en tant que Gang est bloquée

L'approche Concurrent Gang est plus avantageuse pour les charges de travail qui font beaucoup d'entrées/sorties, comme cela est démontré par nos résultats de simulation. Ces résultats ont aussi montré que même avec des charges de travail ayant beaucoup de communications, l'algorithme Concurrent Gang peut être meilleur que l'ordonnanceur Gang.

10.4 Chapitre 7

Dans ce chapitre a été présenté une analyse comparant l'algorithme Concurrent Gang aux ordonnanceurs local inconscient et Gang. Nous avons montré que ces ordonnanceurs ne peuvent pas être meilleurs que l'ordonnanceur Concurrent Gang si on utilise le temps d'exécution de la charge de travail comme métrique, pour la même distribution des tâches. Nous avons également fait une analyse du comportement de l'algorithme Concurrent Gang sous une charge de travail irrégulière, et nous avons proposé l'intégration d'un algorithme d'équilibrage de charge avec Concurrent Gang pour améliorer l'efficacité d'exécution d'une charge de travail Y-irrégulière.

10.5 Chapitre 8

Dans ce chapitre nous présentons quelques utilisations possibles des mesures faites au moment de l'exécution pour améliorer l'ordonnancement parallèle. Les contributions de ce chapitre sont la proposition d'un mécanisme de classification de tâches utilisant la théorie de la décision Bayesienne et la proposition d'un algorithme de définition du temps d'attente d'une réception bloquante en fonction de la charge de travail du processeur. Nous avons aussi démontré que la classification proposée peut être incorporée par n'importe quel ordonnanceur parallèle, comme par exemple l'algorithme DHC.

Nous croyons que l'utilisation des mesures faites au moment de l'exécution dans les ordonnanceurs parallèles est un pas dans la bonne direction, puisque avec plus d'information disponible sur les tâches en cours l'ordonnanceur est capable de faire un choix plus intelligent sur l'ordonnancement. L'augmentation du débit et de l'utilisation des ressources est confirmée par les résultats expérimentaux présentés dans ce chapitre.

10.6 Nouvelles Directions de Recherche

Dans cette section, nous suggérons quelques directions de recherche dans le domaine de l'utilisation des mesures faites au moment de l'exécution pour l'ordonnancement parallèle.

- On peut chercher à améliorer l'algorithme "Implicit Coscheduling" [16, 2] en ajoutant des informations associées à d'autres tâches dans le calcul du temps d'attente dans une communication blocante.
- Il serait intéressant d'analyser et d'améliorer le comportement de l'algorithme Concurrent Gang sous des situations de pénurie de ressources. Dans ce cas, non seulement les caractéristiques de chaque tâche mais

également la disponibilité des ressources devraient être considérées par l'ordonnanceur.

- Il faudrait enfin améliorer et rendre plus efficace l'intégration entre l'ordonnanceur Concurrent Gang et le système d'équilibrage de charge.

Annexe A

Simulator Verification

This appendix describes the simulation environment[1] used in this thesis, and the methodology used for its verification. The simulator and its documentation is available for download at www.ics.uci.edu/∼schark/.

At the core of the environment is an event-driven simulation engine [10]. Algorithm-specific modules aimed at solving a user's particular problem can be added to the environment at compile time. In addition, several frequently used probability distributions are provided as external modules.

The following sections present the simulator's features in terms of the fundamental concepts, the principles of operation and the internal architecture.

A.1 Fundamental Concepts

The simulator implements a discrete event simulation algorithm. In this type of simulation, the simulator machine concentrates on processing events, and the system does not change its internal states between two consecutive events. A state change may occur only due to the processing of an incoming event. The advantage of this type of simulation when compared with other models, such as discrete time simulation algorithms, is that it is inherently more efficient.

According to the terminology defined by Zeigler [97], the simulator can be classified as a Multicomponent DEVS (Discrete Event System Specification).

[1]Simulator and simulation environment will be used intermixed throughout this appendix

Examples of different components, or objects, implemented in this simulator are processors, network, tasks and jobs. With this formalism, events occurring in one component may result in state changes and/or rescheduling of events in other components. This formalism is quite popular, since the simulation strategies realized in many commercial simulation languages and systems fall into the category of multicomponent DEVS [97].

The Multicomponent DEVS implementation strategy employed by this simulator is the event-scheduling model. Event-scheduling implementation models work with prescheduling of all events. Because of its simplicity, event scheduling simulation is the preferred strategy when implementing customized simulation systems in procedural programming languages. An example is given below [97] :

`Component d`

$S_d^{control} = ev_1, ev_2, ev_3, ..., ev_n$

$\sigma((S_d^{control}))$

`case` $S_d^{control}$

ev_1 : `call` $event - routine_1$

ev_2 : `call` $event - routine_2$

...

ev_n : `call` $event - routine_n$

Algorithm 1 - Event Scheduling Model Implementation (adapted from [97])

In Algorithm 1, the set of event types $S_d^{control} = \{ev_1, ev_2, ev_3, ..., ev_n\}$ divides the state transition function σ_d of component d into n functions $\sigma_d^{ev_i}$, each describing the activity of component d depending on the event type and the state of the component.

A.2 Block Diagram

The input to the simulator, both the architectural and workload descriptions, are given in the form of ASCII files. For the detailed descriptions of both the architectural and workload types of input please refer to [43].

The mechanism by which external modules, implementing a particular algorithm, communicate with the simulation engine is explained in section A.3.

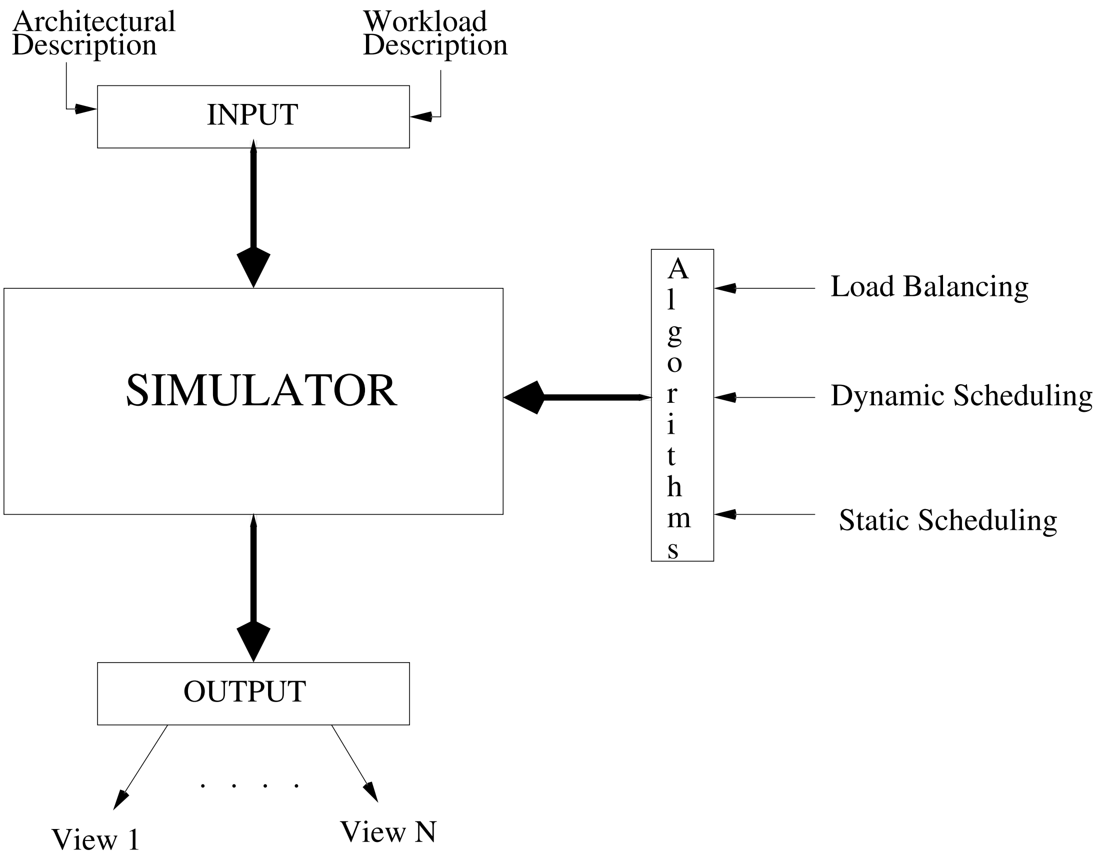

FIG. A.1: The Logical Structure of the Simulation Environment

The output of the simulator is implemented by a module independent of the simulation engine itself. This provides several different "views" that can be customized according to a user's preferences. Note that multiple views can be viewed simultaneously.

The algoritm module exchanges information with the simulation engine through a well defined interface (see A.3). This interface allows for one-way communication only. Information flows from the simulation engine to the algorithm module, or to be more precise, to an instance of a particular user's view. All possible instances must *listen* to a particular event. They differ in two ways :

- The way they implement the actions associated with the triggering of the event
- How they display the information associated with the event

This way, the information associated with each occurrence of the event can be displayed in real-time.

Additionally, all information being tracked by a particular simulation is made available through the output module at the conclusion of the simulation in the form of ASCII files. These files can then be processed to display relevant statistical information.

By separating the algorithm module from the simulation engine, we allow for programmers to develop their own algorithms providing the functionality they desire, instead of us trying to develop a monolithic module using a *one fits all* approach.

A.3 Event-Driven Simulation

An individual algorithm implementation communicates with the simulation engine via *events*. Every module implementing a particular algorithm must *listen* to one or more of the six events exported by the simulator. Once an event is generated, the engine passes it to all registered algorithms (registration is done when an algorithm is initialized). Some of the events are generated regardless of which actions the algorithm may take, for instance the event `JobArrival` is dependent only on the workload, in particular the choice of probability distribution for the inter-arrival time. Others, however, are a direct result of the interaction between the algorithm and the engine. For instance the event `PEIdle` is obviously a direct result of the actions taken by the algorithm when acting upon a previous generated event. A typical example would be a dynamic scheduler algorithm that in response to the event `EndOfTimeSlice` assigns a particular task to the PE in question, which in turn will lead to the triggering of the event `PEIdle` when the task running on that PE performs an I/O operation. No algorithm can, however, schedule an event directly, that is, to force the engine to generate an event at a specific time.

The simulation engine provides support for six events. Their synopsis are :

- PEIdle
  Description :
  Invoked when a PE has become idle for some reason

- EndOfTimeSlice
  Description :
  Invoked when a time-slice has expired on a PE (time slice set by the scheduler through the variable simulator.timeSlice).

- TaskArrival
  Description :
  Invoked when a new task arrives, only if the network is asynchronous

- JobArrival
  Description :
  Invoked when a new job is submitted to the system

- TaskStateChange
  Description :
  Invoked when a task has changed state

- GlobalClock
  Description :
  Called at regular intervals of time (period set by the scheduler through the variable simulator.globalClock)

A.4 Simulator Verification

Verification is the procedure followed to establish that the simulator is guaranteed to faithfully generate the model's output given its initial state and its input. This relation between a simulator and its model is known as *simulation relation* in the literature [97]. A simulator may be capable of executing a whole class of models. That is the case of the simulator described in this appendix. Each main component of the simulator, such as interconnection network, processor and task, may be associated with one or more different models.

There are two general approaches to verification [97] :

- Formal proofs of correctness
- Extensive testing

Formal proofs employ mathematical and logical formalisms to rigorously establish the required relation between the simulator and the model. Unfortunately, such proofs are difficult or impossible to to carry out for large, complex systems [97], and such is the case with our simulator. Moreover, they may also be prone to error since in many cases humans have to understand the symbols and carry out their manipulation.

In the absence of definitive proofs, extensive testing must be done to ensure that all types of conditions that could arise in the operation of the simulator have been covered by test cases. The testing methodology employed during the implementation of the simulator was the following : each component which represents a separate model (for instance, the interconnection network) was tested in isolation, with initial conditions and interactions with other components predefined to cover a reasonable number of testing conditions. Then, after testing all components in isolation, simple examples were applied to the system as a whole in order to ensure that for the coupled system the simulation relation still holds.

Naturally, as one could expect, we can not cover the entire domain for every variable of every component that is visible (i.e. accessible) from the outside world when testing the coupled system.

A possible solution is then to classify the input domain into groups and perform extensive testing for every combination of the input groups, selecting individual values from each group that lead to results that can be verified in practice (i.e. by hand). This procedure is especially important for the testing of boundary values.

Given the previous considerations, two simple methodologies were used for the verification of the simulator as a coupled system :

- Execute testing examples whose results can be verified by tracing every event generated and comparing the evolution of state trajectories and outputs with what had been expected by "running" the simulator by hand.
- Execute testing examples that test boundary conditions of internal variables and/or states.

The first methodology was employed by testing simple scheduling/load balancing algorithms which the behavior can be verified by hand, and then comparing it with the output of the simulator. One example is a variable partitioning scheduling algorithm which the number of jobs, arrival times and execution times are set in advance and workload is composed of rigid regular jobs.

The second methodology was employed in order to guarantee the correct behavior of the simulator under extreme or incorrect input parameters, in particular parameters related with probability distributions.

Bibliographie

[1] B.S. Ang, D. Chiou, L.Rudolph, and Arvind. The StarT-Voyager Parallel System. In *Proceedings PACT 98*, 1998.

[2] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of ACM SIGMETRICS'98*, pages 233–243, 1998.

[3] S. Baase. *Computer Algorithms*. Addison-Wesley Publishing Company, 1988.

[4] E. Barton, J. Cownie, and M. McLaren. Message Passing on the Meiko CS-2. *Parallel Computing*, 20(4) :497–507, 1994.

[5] J. M. Barton and N. Bitar. A Scalable Multi-Discipline, Multiple Processor Scheduling Framework for IRIX. *Job Scheduling Strategies for Parallel Processing*, LNCS 949, 1995.

[6] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, 35(5) :389–393, 1986.

[7] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, Cambridge,Massachusetts, 1990.

[8] R. M. Bryant, H-Y Chang, and B. S. Rosenburg. Operating System Support for Parallel Programming on RP3. *IBM Journal of Research and Development*, 35(5/6) :617–634, Sep/Nov 1991.

[9] L.M. Campos and I.D. Scherson. Rate of Change Load Balancing in Distributed and Parallel Systems. In *Proceedings of the 1999 International Parallel Processing Symposium*, 1999.

[10] Luis Miguel Campos. Resource management techniques for multiprogrammed distributed systems. In *PhD Thesis*. University of California, Irvine, 1999.

[11] E.G. Coffman, M.R. Garey, and D.S. Johnson. Bin Packing with Divisible Item sizes. *Journal of Complexity*, 3 :406–428, 1987.

[12] E.G. Coffman, D.S. Johnson, P.W. Shor, and R.R. Weber. Markov Chains, Computer Proofs, and Average Case Analysis of Best Fit Bin Packing. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 412–421, 1993.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Algorithms*. MIT Press, Cambridge, MA, 1990.

[14] Intel Supercomputer Systems Division. *Paragon User's Guide*. Order number 312489-003, 1994.

[15] J. Du and J. Y-H Leung. Complexity of Scheduling Parallel Task Sistems. *SIAM Journal of Discrete Mathematics*, 2(4) :473–387, 1989.

[16] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of ACM SIGMETRICS'96*, pages 25–36, 1996.

[17] J. Edmonds, D.D. Chinn, T. Brecht, and X. Deng. Non-Clairvoyant Multiprocessor Scheduling of Jobs with Changing Execution Characteristics (extended abstract). In *Proceedings of the 1997 ACM Symposium of Theory of Computing*, pages 120–129, 1997.

[18] A. Greiner et al. La Machine MPC. *Le Calculateur Parallele*, 1998.

[19] A. Hori et al. Time Space Sharing Scheduling and Architectural Support. *Job Scheduling Strategies for Parallel Processing*, LNCS 949 :92–105, 1995.

[20] A. Hori et al. Implementation of Gang Scheduling on Workstation Cluster. *Job Scheduling Strategies for Parallel Processing*, LNCS 1162 :126–139, 1996.

[21] Al Geist et al. *PVM : Parallel Virtual Machine - A User's guide and tutorial for networked parallel computing*. The MIT Press, 1994.

[22] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33 :145–158, 1996.

[23] D. Chiou et al. StarT-NG : Delivering Seamless Parallel Computing. In *Proceedings EUROPAR'95*, 1995.

[24] D. Culler et al. LogP : Towards a Realistic Model of Parallel Computation. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles an Practice of Parallel Programming*, pages 1–12, 1993.

[25] D. Culler et al. A Practical Model of Parallel Computation. *Communication of the ACM*, 93(11) :78–85, 1996.

[26] G. Averson et al. Scheduling on the Tera MTA. *Job Scheduling Strategies for Parallel Processing*, LNCS 949, 1995.

[27] J. Jann et al. Modeling of Workloads in MPP. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291 :95–116, 1997.

[28] J. Turek et al. Approximate Algorithms for Scheduling Parallelizable Tasks. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, 1992.

[29] M. Crovella et al. Multiprogramming on Multiprocessors. In *Proceedings of the 3th IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, 1991.

[30] P. Kok et al. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel and Distributed Technology*, Fall 1996 :25–35, 1996.

[31] Patrick G. Solbalvarro et al. Dynamic Coscheduling on Workstation Clusters. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459 :231–256, 1998.

[32] R Chandra et al. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 12–24, 1994.

[33] D. Feitelson. Packing Schemes for Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1162 :89–110, 1996.

[34] D. Feitelson. Job Scheduling in Multiprogrammed Parallel Systems. Technical report, IBM T. J. Watson Research Center, 1997. RC 19970 - Second Revision.

[35] D. Feitelson and M. A.Jette. Improved Utilization and Responsiveness with Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291 :238–261, 1997.

[36] D. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, pages 65–77, May 1990.

[37] D. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16 :306–318, 1992.

[38] D. Feitelson and L. Rudolph. Coscheduling Based on Runtime Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2) :135–160, 1995.

[39] D. Feitelson and L. Rudolph. Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control. *Journal of Parallel and Distributed Computing*, 35 :18–34, 1996.

[40] D. Feitelson and L. Rudolph. Metrics and Bechmarking for Parallel Job Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459 :1–24, 1998.

[41] H. Franke, P. Pattnaik, and L. Rudolph. Gang Scheduling For Highly Efficient Distributed Multiprocessor Systems. In *Proceedings of Frontiers'96*, 1996.

[42] B. Gorda and R. Wolski. Time Sharing Massively Parallel Machines. In *International Conference on Parallel Processing*, pages 214–217, Volume II, 1995.

[43] Schark Research Group. *http ://www.ics.uci.edu/ schark/simulator*. Simulator Home Page, 2000.

[44] A. Gupta, T. Tucker, and S. Urushibara. The Impact of Operating Systems Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of ACM SIGMETRICS'91*, pages 120–132, 1991.

[45] A. Hori, H. Tezuka, and Y. Ishikawa. Overhead Analysis of Preemptive Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459 :217–230, 1998.

[46] S. Irani and A. R. Karlin. On Online Computation. *Approximation Algorithms for NP-Hard Problems - Ed. Dorit Hochbaum*, 1996.

[47] K. Jansen and L. Porkolab. Linear Time Approximation Schemes for Scheduling Malleable Parallel Tasks. In *Proceedings of 10th annual ACM symposium on discrete algorithms*, pages 490–498, 1999.

[48] M. A. Jette. Performance Characteristics of Gang Scheduling In Multiprogrammed Environments. In *Proceedings of SC'97*, 1997.

[49] J.J.Martin. *Bayesian Decison Problems and Markov Chains*. John Wiley and Sons Inc., New York, N.Y., 1967.

[50] D. S. Johnson. The NP-Completeness Column : an Ongoing Guide. *J. Algorit.*, 4(2) :189–203, 1983.

[51] B. Kalyanasundaram and K. Pruhs. Speed is as Powerful as Clairvoyance. In *Proceedings of the 36th Symposium on Computer Science*, pages 214–221, 1995.

[52] R. M. Karp. Parallel Combinatorial Computing. *Very Large Scale Computation in the 21th Century*, pages 221–238, 1991.

[53] L. G. Khachian. A polynomial algorithm for linear programming (in Russian). *Doklady Akad. Nauk USSR*, 244 :1093–1096, 1979.

[54] K.L.Park and L.W. Dowdy. Dynamic Partitioning of Multiprocessors Systems. *International Journal on Parallel Programming*, 18(2) :91–120, 1989.

[55] B. Kosko. Fuzziness vs. Probability. *International Jounal of General Systems*, 17(2-3), 1990.

[56] B. Kosko. *Neural Networks and Fuzzy Systems : A Dynamical Systems Approach for Machine Intelligence.* Prentice Hall, Inc., 1992.

[57] R.N. Lagerstrom and S. K. Gipp. PSheD : political scheduling on the Cray T3E. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291 :117–139, 1997.

[58] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291 :215–237, 1997.

[59] W. Leinberger, G. Karypis, and V. Kumar. Job Scheduling in the Presence of Multiple Resources Requirements. In *Proceedings of Supercomputing' 99*, 1999.

[60] W. Leinberger, G. Karypis, and V. Kumar. Multi-Capacity Bin Packing Algorithms with Applications to Job Scheduling under Multiple Constraints. In *Proceedings of the 1999 International Conference On Parallel Processing*, 1999.

[61] E. J. Lerner. The End of the Road for Moore's Law ? *IBM Think Research*, 1(4), 1999.

[62] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of ACM SIGMETRICS'90*, pages 226–236, 1990.

[63] K. Li, J. F. Naughton, and J. S. Plank. An Efficient Checkpointing Method for Multicomputers with Wormhole Routing. *International Journal of Parallel Programming*, 20(3) :159–180, 1991.

[64] S-P Lo and V. D. Gligor. A comparative Analysis of Multiprocessor Scheduling Algorithms. In *Proceedings of the 7th International Conference On Distributed Computing Systems*, pages 356–363, 1987.

[65] Jr. M. J. Gonzalez. Deterministic Processor Scheduling. *ACM Computing Surveys*, 9(3) :173–204, 1977.

[66] S. Majumdar, D.L. Eager., and R.B. Bunt. Characterization of Programs for Scheduling in Multiprogramming Parallel Systems. *Performance Evaluation*, 13(2), 1991.

[67] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive Algorithms for On Line problems. In *Proceedings of the Twentieth Annual Symposium on the theory of Computing*, pages 322–333, 1988.

[68] E. P. Markatos and T. J. Leblanc. Using Processor Affinity in Loop Scheduling on Shared Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4) :379–400, 1994.

[69] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1) :17–47, 1994.

[70] G. Mounie, C. Rapine, and D. Trystram. Efficient Approximation algorithms for Scheduling Malleable Tasks. In *Proceedings of SPAA'99*, pages 23–31, 1999.

[71] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa : An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2) :92–105, 1980.

[72] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Comp. Systems*, pages 22–30, 1982.

[73] E. W. Parsons and K. C. Sevcik. Implementing Multiprocessor Scheduling Disciplines. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291 :166–192, 1997.

[74] P. Pierce and G. Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Proceedings of the Scalable High Performance Computing Conference*, pages 184–190, 1994.

[75] R. Pool. Assembling Life's Building Blocks. *IBM Think Research*, 1(4), 1999.

[76] V. Ramakrishnan and I. D. Scherson. Executing Communication-Intensive Irregular Programs Efficiently. In *Proceedings of Irregular 2000*, 2000.

[77] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante. The Impact of I/O on Program Behavior and Parallel Scheduling. In *Proceedings of ACM SIGMETRICS'98*, pages 56–64, 1998.

[78] V. Sakar. Determining Average Program Execution Times and Their Variance. In *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, 1989.

[79] Isaac D. Scherson, Raghu Subramanian, Veronica L. M. Reis, and Luis Miguel Campos. Scheduling computationally intensive data parallel programs. In Bertil Folliot, editor, *Placement Dynamique et Répartition de Charge : application aux systèmes parallèles et répartis*, pages 39–61. Centre National De La Recherche Scientifique, July 1996.

[80] S. K. Setia. Trace-Driven Analysis of Migration Based Gang Scheduling Policies for Parallel Computers. In *Proceedings of International Conference on Parallel Processing*, 1997.

[81] P.W. Shor. How to do better than best fit : Tight bounds for average case on-line bin packing. In *Proceedings of the 32th IEEE Annual Symposium on Foundations of Computer Science*, pages 752–759, 1990.

[82] F.A.B. Silva, L.M. Campos, and I.D. Scherson. A Lower Bound for Dynamic Scheduling of Data Parallel Programs. In *Proceedings EUROPAR'98*, 1998.

[83] F.A.B. Silva, L.M. Campos, and I.D. Scherson. Improvements in Gang Scheduling for Parallel Supercomputers. In *Proceedings 8th International Parallel Computing Workshop*, 1998.

[84] F.A.B. Silva and I. D. Scherson. Efficient Parallel Job Scheduling Using Gang Service. *To appear in the Internation Journal of Foundations of Computer Science*, June 2001.

[85] F.A.B. Silva and I.D. Scherson. Improvements in Parallel Job Scheduling Using Gang Service. In *Proceedings 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, 1999.

[86] F.A.B. Silva and I.D. Scherson. Improving Throughput and Utilization on Parallel Machines Through Concurrent Gang. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium 2000*, 2000.

[87] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2) :202 – 208, 1985.

[88] E. Smirni, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O Requirements of scientific aplications : an evolutionary view. In *Proceedings of the IEEE international Symposium of High Performance Distributed Computing*, pages 49–59, 1996.

[89] E. Smirni and D. A. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation*, 33 :27–44, 1998.

[90] W. Smith, I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459 :122–142, 1998.

[91] P. Steiner. Extending Multiprogramming to a DMPP. *Future Generation Computing Systems*, 8(1) :93–109, 1992.

[92] Raghu Subramanian. A framework for parallel job scheduling. In *PhD Thesis*. University of California, Irvine, July 1995.

[93] J. D. Ullman. NP-Complete Scheduling Problems. *J. Comput. System Science*, 10(3) :384–393, 1975.

[94] L. G. Valiant. A bridging model for parallel computations. *Communications of the ACM*, 33(8) :103 – 111, 1990.

[95] Y. Yan, C. Jin, and X. Zang. Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(1) :70–81, 1997.

[96] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8 :338–353, 1965.

[97] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation, 2nd Edition.* Academic Press, New York, NY, 2000.