Ordonnancement avec communications pour systèmes multiprocesseurs dans divers modèles d'exécution

Frédéric Guinand

Thèse

présentée par

Frédéric GUINAND

pour obtenir le grade de Docteur

de l'Institut National Polytechnique de Grenoble

(arrêté ministériel du 30 mars 1992)

Spécialité : Informatique

Ordonnancement avec communications pour architectures multiprocesseurs dans divers modèles d'exécution

Date de soutenance : 7 Juin 1995

Composition du jury

Président :	Jacques VOIRON
Rapporteurs :	Paul FEAUTRIER
	Dominique DE WERRA
Examinateurs :	Evripidis BAMPIS
	Philippe CHRÉTIENNE
	Denis TRYSTRAM

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul
(Institut de Mathématiques Appliquées de Grenoble)

2

Table des matières

Avant-propos

La lecture de ce document peut ne pas être linéaire. Les trois parties qui le composent sont, dans une large mesure, indépendantes. Afin d'en faciliter la lecture, la majeure partie des définitions sont regroupées dans l'annexe sise en fin de volume.

Par convention, les mots ou expressions écrits en **gras** représentent des passages clés pour le sujet étudié dans le chapitre ou dans le paragraphe. Pour certaines notions le terme anglo-saxon est d'un usage plus pratique ou plus explicite que son équivalent français. Il lui est alors substitué et est typographié en *italique*.

La quasi totalité des graphes représentés sont des graphes de précédence sans cycle. Les arcs sont orientés du haut vers le bas, et les flèches des arcs ne sont, en général, pas représentées.

Partie I

Introduction

En quelques décennies, l'informatique a investi une grande partie des activités humaines, de la vie quotidienne à la vie professionnelle. Depuis l'ordinateur personnel équipant un nombre toujours croissant de ménages, jusqu'aux systèmes complexes dont disposent les départements informatiques de grands groupes industriels, militaires ou civils. Pour un grand nombre d'applications, l'arrivée rapide et en masse des ordinateurs a changé l'échelle de temps sur laquelle étaient basés les projets et réalisations de toute sorte. Mis en appétit par les possibilités apportées par l'informatique, de nombreux secteurs d'activité ont eu, et ont encore, le désir de calculer, de gérer ou d'organiser plus vite et plus gros. Une première réponse à cette demande a été l'amélioration de la capacité d'intégration des processeurs (nombre de transistors pour une surface donnée) et la diminution des temps de cycle dont la conjonction a permis d'augmenter de façon exponentielle la puissance de calcul des processeurs. Cependant, l'intégration et la vitesse d'horloge sont à ce point optimisées que les limitations physiques sont proches, et avec elles celles des performances réalisables par un seul processeur. Un gain, même faible, sur l'une ou l'autre de ces deux caractéristiques n'est obtenu qu'au prix de gros effort de recherche et financiers. L'attrait que représente la coopération entre processeurs est double, d'une part, elle permet d'obtenir des performances hors d'atteinte des machines monoprocesseurs, d'autres part, il s'agit de systèmes évolutifs dont le prix des éléments de base n'est pas nécessairement élevé. D'autre part, toute amélioration des performances des composants de base se répercute sur les machines paralléles. Pour l'ensemble de ces raisons, il semble raisonnable de penser que le parallélisme jouera à l'avenir un rôle de premier plan dans la recherche de performances.

D'abord cantonnés au seul calcul scientifique, les ordinateurs parallèles voient leur champ d'application s'étendre chaque jour davantage. Ces dernières années, plusieurs grands constructeurs ont proposés de telles machines [Rum94, TW91], ce qui autorise à penser que dans un proche avenir leur diffusion va encore s'accélérer. Pour la plupart, ces ordinateurs sont acquis par de grandes entreprises et par des centres de recherche publics et privés. Si l'intérêt pour ce type de machines ne se dément pas, elles restent difficile à utiliser. Une partie de ces grands entreprises ne disposent pas d'une équipe de spécialistes pour les faire fonctionner et pour les programmer. Or, une formation est aujourd'hui encore nécessaire pour un informaticien "traditionnel" désirant exploiter de manière efficace leur puissance. Il paraît donc clair que la diffusion de ce type de matériel nécessite la recherche de solutions pour rendre leur utilisation plus simple. Certains organismes publics et privés œuvrent en ce sens. L'objectif est la mise au point de logiciels pour masquer les difficultés en automatisant une partie des tâches de parallélisation des applications. Une partie de ces logiciels est regroupé sous le terme **d' environnements de programmation parallèle** (EPP) [ERL90, WG90, YG92a].

Parallélisme de données et de contrôle

Ces environnements admettent en entrée un programme décrit dans un langage séquentiel, annoté ou non, ou parallèle. En sortie il délivre un programme dont le code est parallèle. Parmi les différentes manières de générer un code parallèle, le **parallélisme implicite** reporte tous les problèmes au niveau du compilateur, au contraire du **parallélisme explicite** qui demande à l'utilisateur davantage de recul pour la description de l'application. L'in-

vestissement demandé à l'utilisateur consiste en un **découpage** de son application. Cette subdivision permet très souvent de produire un code dont l'exécution est nettement plus efficace qu'elle ne le serait dans le cas d'une approche implicite. A nouveau, deux approches émergent du parallélisme explicite, selon l'objet du découpage. La première, basée sur les **données**, produit du code parallèle à partir du programme de départ et de **directives de compilation**. Cette approche met en œuvre diverses techniques d'analyse comme par exemple l'ordonnancement de nids de boucles [Fea91, Fea92, DR93]. Pour des applications régulières et requérant de fortes puissances de calcul, cette approche peut donner de bons résultats. Dans la seconde approche, reconnue sous le nom de **parallélisme de contrôle**, ce sont les **calculs** que l'on découpe. Cette approche permet de fournir une réponse plus adaptée aux problèmes basés sur des données **irrégulières** ou dont la taille varie durant l'exécution. Du point de vue l'utilisation des ressources, dans l'approche parallélisation automatique, **l'allocation** des processeurs est guidée par la distribution de données, alors qu'elle est déterminée par l'analyse de l'agencement des calculs dans l'approche parallélisme de contrôle.

Démarche de parallélisation

Dans une démarche de type parallélisme de contrôle se dégagent essentiellement trois étapes de traitement. La première consiste, à partir du code source, à **identifier** le parallélisme et à **fractionner** l'application en un ensemble d'éléments qui interagissent : le **graphe de programme** que l'on appelle aussi **graphe de tâches**. La seconde phase de traitement, qui constitue l'étape centrale du processus de parallélisation est **l'allocation** de la puissance de calcul aux divers éléments de l'application décrite par le graphe de tâches. Le dernier traitement génère le code parallèle. Ces trois étapes sont mises en œuvre au moment de la **compilation**, ou éventuellement en tout début d'exécution pour des programmes fortement dépendant des données.
Au moment de l'exécution, le support d'exécution de l'environnement prend le relais. Il est constitué d'éléments de mesures de performances, de prises de traces (dans le but de réexécutions déterministes ou de simulations) et d'un **répartiteur dynamique de la charge** jouant, durant l'exécution, le même rôle que le module responsable de l'allocation mis en œuvre au moment de la compilation.

Construction du graphe de tâches

Le programme source est le point de départ de toute analyse. Il peut être écrit dans un langage séquentiel, avec ou sans directives de compilation, ou dans un langage permettant d'exprimer de façon explicite le parallélisme. Selon le langage utilisé, et la présence d'annotations, l'analyse de la première étape s'en trouve ou non facilité. L'objectif de ce premier traitement est de déterminer les groupes d'instructions qui doivent être exécutées en séquence, ceux qui sont indépendants, et les groupes qui nécessitent des données en provenances d'instructions appartenant à d'autres groupes. Cette analyse a pour but de générer une représentation **modélisée** du programme sous la forme d'un **graphe**, entité plus facile à manipuler que le programme lui-même.
Dans ce modèle, les sommets du graphe sont les groupes d'instructions et sont appelés des

tâches. L'utilisation d'une donnée par une tâche en provenance d'une autre tâche constitue une **relation de précédence** et correspond à un arc dans le graphe. La notion de **granularité** est liée à la taille ou à la structure des tâches, le grain le plus fin correspondant à la définition d'une tâche par instruction. Sa détermination dépend en partie du rapport entre coût de communication et coût de calcul, elle est donc fortement dépendante de la machine cible.
Le graphe produit à l'issue de la première étape constitue la donnée d'entrée de la phase d'allocation qui joue un rôle important dans la génération d'un code parallèle efficace. Réaliser une bonne allocation, c'est assurer une bonne distribution des ressources de calcul en fonction du temps, des caractéristiques des tâches et de leurs dépendances.

Étape d'ordonnancement et de placement

Ce problème est exprimé différemment si le graphe est orienté ou non. Lorsqu'il ne l'est pas, il n'existe aucun ordre d'exécution entre les tâches. Les relations de précédence sont remplacées par des transferts de données. Si un tel graphe est entièrement connu au moment de la compilation, les sommets sont valués par des volumes ou des temps estimés de calculs, et les arêtes sont valuées par des volumes d'échanges de données ou des temps nécessaires pour effectuer ces transferts. Dans un tel cadre, déterminer une allocation c'est résoudre un problème de **placement**, c'est associer à chaque tâche un processeur avec pour objectif la minimisation d'une fonction de coût. Il s'agit souvent de rechercher le meilleur **compromis** entre **l'équilibrage de la charge** et la **minimisation des communications**. Lorsque le graphe est orienté, un ordre partiel existe entre les tâches. Les sommets sont aussi valués par des volumes ou des temps estimés de calculs, et les valeurs affectées aux arcs correspondent à des volumes ou à des durées estimées de communications. A la différence d'un graphe non orienté, les tâches reçoivent des données avant leur exécution et ne peuvent communiquer des résultats qu'à la fin de leur exécution. Un arc n'est pas une liaison permanente entre deux tâches. La détermination d'une allocation, pour un graphe orienté, pose le problème du choix du processeur et d'une date d'exécution. Résoudre le problème de l'association d'un processeur et d'une date d'exécution à chaque tâche c'est déterminer un **ordonnancement** [BESW93, CD72, CC88]. La plupart de ces problèmes mettent en jeu des machines, des graphes et des objectifs à atteindre. Le but des stratégies d'ordonnancement est donc d'allouer à chaque sommet du graphe une date d'exécution (ou plusieurs lorsque les tâches peuvent être divisées dans le temps (hypothèse de préemption)), sur un processeur (ou plusieurs s'il s'agit de tâches multiprocesseurs), de manière à optimiser la valeur de l'objectif.

Modèles d'exécution

Parmi les objectifs des environnements de programmation parallèles se trouve la **portabilité**, c'est-à-dire la capacité d'adaptation à des machines différentes. Pour atteindre ce but, chacun des éléments de l'environnement, et plus particulièrement le module responsable de l'ordonnancement, doit avoir cet objectif. Il faut donc s'efforcer de trouver des méthodes et des stratégies qui puissent s'adapter à divers types de **modèles d'exécution**. De nombreuses tentatives de modélisation de machines parallèles ont vu le jour depuis une dizaine

d'années [PU87, RS87, PY88, CZ89, ACS90, CZ90, CF91, BKT94, MR94]. La multiplication du nombre de ces travaux coïncide avec l'intérêt grandissant apporté à l'étude de l'impact des communications dans l'efficacité des algorithmes développés pour les machines parallèles. Afin d'améliorer l'efficacité des algorithmes parallèles, les travaux dans ce domaine ont suivi plusieurs axes de recherche. Le premier privilégie l'aspect **non uniforme des accès mémoires** dû à la coexistence au sein d'un même ordinateur de plusieurs types de mémoires [ACS90]. Un autre axe de recherche se focalise sur le **mode de fonctionnement asynchrone** dû par exemple aux interruptions systèmes [CZ89, CZ90, MR94]. L'expression des coûts de communication en fonction de l'éloignement des processeurs et des diverses parties de la mémoire qui est considérée comme distribuée est une autre approche discutée dans [PY88, CF91].

Organisation du document

L'étude de certains de ces modèles d'exécution constitue la première partie de ce document. Un chapitre est consacré à la description de ces modèles ainsi qu'à la mise en évidence de points communs entre eux.
Le second chapitre traite de l'étude de l'ordonnancement des arbres dans un modèle n'autorisant pas le recouvrement des communications par des calculs, et dont l'objectif est la minimisation du surcoût défini comme la somme des temps d'inactivité et des temps de communication [ABR90]. Dans ce chapitre, nous prouvons que les problèmes de l'ordonnancement de chaîne et d'arbres binaires sur seulement deux processeurs sont NP-complet. Pour ces problèmes, nous proposons un algorithme qui produit des ordonnancements optimaux pour les arbres binaires complets et pour les structures de types chenille (*caterpillar*). Pour un nombre plus important mais fixé de processeurs, un algorithme pour ordonnancer des arbres complets est décrit et son optimalité est prouvée.
Dans le dernier chapitre de cette partie, nous proposons un modèle d'exécution qui prend en considération davantage de paramètres que les modèles classiques. Parmi ceux-ci, nous tentons de prendre en compte le mode de commutation des messages. A cette occasion, nous montrons qu'il peut être intéressant pour faire de l'ordonnancement de réutiliser des techniques et algorithmes issus du domaine de l'optimisation des communications.

Dans la seconde partie, la même d'étude est faite pour des modèles autorisant le recouvrement des communications par des calculs. Le premier chapitre traite de la résolution des problèmes d'ordonnancement d'arbres pour un système biprocesseurs. Un nouvel algorithme qui produit des ordonnancements optimaux est présenté. La stratégie mise en œuvre pour le cas de tâches unitaires et de communications unitaires (*UECT*) est également adaptée au cas des arbres dont les tâches ou les communications ne sont plus unitaires. Il est montré qu'il est possible de déterminer un ordonnancement optimal pour des arbres dont les communications sont unitaires et dont les tâches ont des durées d'exécution égales à une ou deux unités de temps.
Dans le second chapitre, le même type de stratégie est repris pour la mise au point d'une heuristique d'ordonnancement d'arbres *UECT* sur un nombre $m \geq 2$ fixé de processeurs. Cette heuristique est de même qualité que l'heuristique proposée par Varvarigou et al. [VRKL94],

mais elle se généralise plus facilement pour des granularités voisines.
Dans le troisième chapitre nous étudions l'ordonnancement d'arbres sur un système formé de processeurs uniformes. Ce type de système prend de plus en plus d'importance avec le développement des réseaux de stations, et les études menées dans le cadre d'un environnement hétérogène devraient être en augmentation à court terme. Après un bref tour d'horizon des travaux qui mettent en jeu des processeurs uniformes, nous présentons un algorithme pour des arbres complets sur deux processeurs, et nous prouvons l'optimalité des ordonnancements obtenus.

Enfin, dans la troisième partie, nous décrivons un processus de parallélisation avec lequel cadre la majeure partie des environnements existants aujourd'hui. Ce processus présente quelques inconvénients. Ainsi, la détermination de la granularité a des retombées à plusieurs niveaux pour la production d'un code parallèle efficace. De même, au moment de la compilation le graphe n'est que très rarement complètement déterminé, or les modules ordonnancement et placement statiques qui existent à l'heure actuelle ne travaillent que sur des graphes entièrement valués. Pour quelques une de ces questions, dans le cadre du projet APACHE, nous proposons une solution qui consiste essentiellement à effectuer l'ordonnancement et de la construction de graphes simultanément. Cette discussion constitue le second chapitre de cette partie et représente les spécifications du module d'ordonnancement statique du projet APACHE.

Notations

Afin d'unifier la façon d'intituler un problème d'ordonnancement donné, une notation a été mise au point [GLLK79] et à été enrichie [Vel93b]. Brièvement, rappellons que cette notation est constituée de trois champs $\boldsymbol{\alpha} \mid \boldsymbol{\beta} \mid \boldsymbol{\gamma}$.
Le premier champ, α, représente les **caractéristiques des machines** ainsi que leur nombre. Pour les problèmes qui sont traités dans ce document, les machines sont des processeurs, et ceux-ci sont soit identiques soit uniformes (caractérisés par des vitesses différentes). Lorsque les processeurs sont identiques, la notation commence par P, et par Q lorsqu'ils sont uniformes. Selon que le nombre de processeurs n'est pas limité, est fixé égal à m ou est un des paramètres, la notation du problème commence par $\bar{P}$ (ou parfois P_∞), P_m ou P (si les processeurs sont identiques).
Le second champ, β, représente les **caractéristiques du graphe de précédence**, ainsi que certaines **hypothèses induites par le modèle d'exécution**. Pour la majeure partie des problèmes qui sont mentionnés dans ce document, la structure du graphe appartient à l'ensemble $\{prec, arbre, chaîne\}$, où $prec$ représente un graphe sans cycle orienté, $arbre$ un graphe dont la structure est arborescente, et $chaîne$ un ensemble de chaînes. Lorsque rien n'est spécifié, les tâches sont indépendantes. Dans ce champ sont également spécifiées les volumes ou les temps de calcul des tâches et de communication (valuation des arcs). Enfin, ce champ permet aussi d'exprimer que la duplication (dup) et le préemption des tâches ($pmtn$) sont autorisées.
Le dernier champ, γ, représente le **critère d'optimisation**. Dans ce travail, le critère pris en compte dans la majorité des cas est le **minimum des maximums des dates de fin**

d'exécution des tâches (*minimum of maximum completion time*) noté $\boldsymbol{C_{max}}$ et souvent appelé *makespan*.

Partie II

Modèles d'exécution

Chapitre 1

Introduction

Dans l'informatique comme dans un grand nombre de domaines, la complexité du comportement réel du système à étudier et à utiliser est telle qu'il n'est pas possible de prendre en considération l'ensemble des paramètres qui le caractérisent. La solution envisagée pour pouvoir malgré tout concevoir des méthodes, des techniques et des algorithmes pour ces systèmes est de mettre au point une représentation **modélisée** de ce système. Arrivé à ce point, deux types de modèles sont distingués : les modèles généraux et les modèles dédiés. Les premiers permettent la conception d'algorithmes indépendamment de la machine sur laquelle ils devront être implémentés. Le principal avantage étant que les algorithmes conçus pour ces modèles sont potentiellement réutilisable sur un grand nombre de machines. Les seconds sont plus proches d'une machine cible, le nombre de paramètres pris en compte est généralement plus important que dans le précédent, mais les algorithmes conçus pour ce type de modèles ne sont pas adaptables sans un important effort sur une autre machine.

Si l'on considère la technologie séquentielle, l'ordinateur de Von Neumann constitue un modèle à la fois simple, une unité centrale de calcul connectée à une unité de stockage, la mémoire, et un modèle permettant la conception d'algorithmes relativement efficaces. Lorsqu'il s'agit de modéliser un ordinateur parallèle, ou un système distribué, les difficultés vont croissant. Depuis quelques années un intense effort de recherche a été produit dans ce domaine afin de mettre au point de bons modèles.

Le but essentiel de ces modèles est d'aider la conception d'algorithmes dont l'exécution puisse être efficace sur une machine réelle. Cependant, spécialement dans le cas du parallélisme, l'algorithme n'est pas le seul élément garantissant une exécution efficace. En effet, une fois l'application analysée, elle est **partitionnée** en un ensemble de **tâches**, liées entres elles par des **relations de précédence** (cas orienté) ou **d'échange de données** (cas non orienté). Le graphe résultant de ce partitionnement et de l'analyse des dépendance de données doit ensuite être distribué entre les processeurs de la machine. Cette opération est à la charge d'un outil **d'ordonnancement** (cas orienté), et de **placement**. Ce traitement pouvant intervenir au moment de la compilation (graphe déterminé statiquement) ou au moment de l'exécution. Dans ce dernier cas, des techniques de répartition dynamique de la charge et de régulation de charge sont mis en oeuvre. Nous sommes plus particulièrement intéressé

par les graphes entièrement déterminés au moment de la compilation (analyse statique). En outre, tous les modèles conçus pour les machines parallèles ne permettent pas de faire de l'ordonnancement statique comme nous le verrons plus avant.

D'un point de vue facilité d'utilisation, le modèle **PRAM** a rencontré un vif succès. La plupart des contraintes architecturales étant cachées, les concepteurs d'algorithmes peuvent porter toute leur attention sur la parallélisation de leur application, sans se soucier de la future implémentation sur une machine réelle. En ce sens, ce modèle répond totalement au besoin de création de nouveaux algorithmes parallèles. Cependant, l'implémentation de ces algorithmes sur machines réelles est rarement d'une grande efficacité, ce qui a encouragé de nouvelles recherches dans plusieurs directions.

La première consiste à adapter ces algorithmes en vue d'une implémentation sur une machine réelle. Les méthodes employées visent à permettre une simulation de ces algorithmes sur des machines réelles [Har94]. Les techniques associées tentent de résoudre trois principaux problèmes. Dans une **CRCW** (lecture et écriture concurrentes) les **accès concurrents** doivent être transformés en des **accès séquentiels**. Toujours lors d'accès concurrents, une importante question concerne la **gestion de la mémoire** de telle sorte que les accès concurrents à un même espace (ou module) mémoire se produisent le moins souvent possible. Le troisième de ces problèmes concerne le **routage**. Un algorithme de routage efficace doit permettre le traitement des requêtes de lecture et d'écriture sans entraîner d'important ralentissement.

Le second type de recherche se cristallise autour de la conception de nouveaux modèles d'exécution. Arguant du fait que le modèle PRAM cache la majeure partie des contraintes architecturales, les chercheurs essaient d'identifier les principaux paramètres et critères jouant un rôle clé dans l'efficacité d'un algorithme parallèle. Parmi ceux-ci la prise en compte des coûts de synchronisation et la non uniformité de l'environnement d'exécution ont présidés à la définition du modèle **APRAM** (*Asynchronous PRAM*) [CZ89]. Un première extension de ce modèle **APRAM à vitesse variable** permet de tenir compte de la variation des vitesses des processeurs due aux interruptions systèmes, au fonctionnement multitâches des noyaux d'exécution, ainsi qu'aux entrées/sorties et à la création dynamique de nouveaux processus [CZ90]. D'autres extensions permettant la lecture et l'écriture de blocs de mémoire, de manière pipelinée (**Delay-PRAM**) ou non (**Bloc-PRAM**) ont été décrit dans [MR94].

D'autres paramètres semblent avoir une certaine importance pour la conception d'algorithmes parallèles efficaces. La non uniformité des accès mémoire à été prise en compte à la fois par Papadimitriou et Ullman [PU87] et par Aggarwal, Chandra et Snir [ACS90]. Ils considèrent une mémoire à deux niveaux. Un accès à la mémoire locale est considéré de coût nul, alors qu'un accès à la mémoire globale (distribuée dans [PU87] et partagée dans [ACS90]) nécessite un coût unitaire ou d. La façon de prendre en compte cette non uniformité marque l'importance de la **localité** des accès mémoire et incite la construction d'algorithmes qui privilégient ce type d'accès. La production d'algorithmes efficaces nécessite alors la recherche d'un **compromis** entre les communications, conséquence du parallélisme, et le temps d'exé-

cution, d'autant plus important que l'algorithme est séquentiel [PU87, ACS90]. Le même type de compromis est nécessaire pour la conception d'algorithmes parallèles dans le modèle introduit par Anderson, Beame et Ruzzo [ABR90] pour des machines parallèles à mémoire partagée comportant un faible nombre de processeurs. La principale différence provient des paramètres considérés comme étant de grande importance pour l'efficacité des algorithmes. Papadimitriou et Ullman focalisent leur attention sur le temps d'exécution et sur les communications, Aggarwal, Chandra et Snir considèrent que le nombre d'étapes de calcul (les processeurs sont synchronisés), le temps total et le nombre de processeurs sont les paramètres les plus importants pour leur modèle. De leur côté, Anderson, Beame et Ruzzo proposent le **surcoût** (*overhead*), somme des temps d'inactivité et de communication, comme le paramètre essentiel.

La localité des données est exprimée d'une manière voisine dans le modèle proposé par Bampis, König et Trystram, modèle dans lequel seules les communications entre voisins sont autorisées [BKT94]. Ce modèle, **LXRAM** (*Local XRAM*), est une extension de **XRAM** présenté par Cosnard et Ferreira [CF91], permettant de prendre en compte le réseau d'interconnexion. Toujours concernant les communications, certains chercheurs ont proposés la possibilité de pouvoir éliminer ces communications par **duplication** de tâches [PU87, PY88], ou par **recouvrement** des communications par des calculs [RS87, PY88]. Cette dernière hypothèse rend compte de certaines caractéristiques architecturales d'un ensemble de nouvelles machines dotées de processeurs de calcul appariés à un processeur dédié ayant la charge des communications.

Dans la suite de cette partie, certains de ces modèles sont présentés, eu égard à la pertinence des paramètres pris en compte et à la possibilité de produire des ordonnancements pour de tels modèles. Nous restreignons donc cette étude aux modèles **LPRAM**, ainsi qu'aux modèles présentés par Anderson, Beame et Ruzzo (appelé par la suite modèle **ABR**), Papadimitriou et Ullman (modèle **PU**), Rayward-Smith (**RS**) et, Papadimitriou et Yannakakis (**PY**).
Pour le modèle PU, nous proposons un algorithme qui permet de produire des ordonnancements optimaux pour des arbres k-aires complets. Le compromis nécessaire entre les communications et le nombre de processeurs utilisé est mis en évidence. Pour le modèle ABR, nous étudions l'ordonnancement de graphes à structure arborescente. Nous montrons tout d'abord que la détermination de l'ordonnancement optimal d'un arbres binaires sur seulement deux processeurs est un problème difficile. Nous proposons une heuristique itérative qui permet de produire des ordonnancements optimaux sur deux processeurs pour des arbres complets ainsi que pour des arbres de type **chenille** (*caterpillar*). Dans un second temps, un algorithme pour ordonnancer des arbres complets sur un système formé de m processeurs identiques est présenté et l'optimalité des ordonnancements qu'il produit est prouvée. L'ordonnancement d'arbres dans des modèles autorisant le recouvrement n'est pas étudié dans cette partie, mais dans la suivante.

Chapitre 2

Description de quelques modèles d'exécution

Dans ce chapitre nous décrivons brièvement quelques modèles d'exécution, en partant du modèle **PRAM**. Pour chacun des modèles présentés,

2.1 Le modèle PRAM

2.1.1 Description

Une **n-PRAM** (*Parallel Random Access Memory*) est une machine composée d'un ensemble de n processeurs possédant chacun une mémoire locale et liés entre eux via une mémoire globale [FW78]. La mémoire peut être partagée ou distribuée entre les processeurs. Le mode de fonctionnement est **synchrone**. Ceci implique que les processeurs peuvent exécuter différents programmes (*MIMD*), mais aucun d'entre eux ne peut commencer l'exécution de l'instruction i de son programme tant que tous les autres processeurs n'ont pas achevé l'instruction $i-1$. En une unité de temps, un processeur peut **lire** en mémoire globale, **exécuter** une opération et **écrire** dans la mémoire globale.

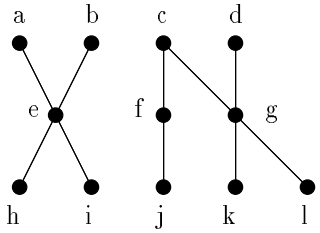
Lorsque la mémoire est partagée, le modèle spécifie de quelle manière sont autorisés les accès concurrents à la mémoire. On distingue trois types d'accès suivant le degré de concurrence. Le modèle le moins puissant qui n'autorise qu'une lecture et qu'une écriture à la fois dans la mémoire partagée : *EREW PRAM*. Le modèle *CREW PRAM* (*Concurrent Read Exclusive Write*) qui ne permet qu'une écriture à la fois, mais plusieurs lectures concurrentes, enfin, le plus puissant, le modèle *CRCW PRAM* (*Concurrent Read Concurrent Write*), qui permet à la fois les lectures et les écritures concurrentes. Dans ce dernier cas, il existe plusieurs stratégies pour résoudre les conflits d'écriture. Soit les processeurs qui veulent écrire dans le même espace mémoire doivent écrire la même chose (*Common*), soit l'un d'eux est choisi, de façon arbitraire (*Arbitrary*) ou parce qu'il possède la plus grande priorité (*Priority*), ou alors les différentes valeurs proposées pour l'écriture dans l'espace mémoire sont combinées (*Combining*).

2.1.2 Ordonnancement dans le modèle PRAM

Le fait d'inclure les accès mémoire dans un cycle comprenant aussi le calcul est équivalent à les considérer de coût nul. Ainsi, lorsque le graphe de précédence est connu de manière statique, toutes les techniques d'ordonnancement de graphes dont les tâches sont de durées unitaires peuvent être appliquées. Il existe un grand nombre de résultats d'ordonnancement célèbres pour ce modèle. Pour les arbres notamment, le résultat le plus connu est dû à Hu [Hu61]. L'algorithme qu'il propose permet de produire, pour m processeurs, des ordonnancements optimaux pour des forêts d'arbres ou d'anti-arbres. Il s'agit d'un algorithme de liste. Les priorités des tâches sont calculées à partir du critère du chemin critique. D'autres résultats existent pour ordonnancer des forêts formées à la fois d'arbres et d'anti-arbres [GJTY83]. On notera toutefois que pour ce modèle extrêmement simple, un grand nombre de problèmes restent ouverts comme notamment le problème à m machines que l'on note : $P_m \mid prec, p_i = 1 \mid C_{max}$ où C_{max} représente le minimum des maximums des dates de fin d'exécution, encore appelé *makespan*. Pour $m = 2$ machines, ce problème a été résolu par l'algorithme de Coffman et Graham [CG72].

2.1.3 Exemple

La figure 2.1 illustre un ordonnancement valide pour le graphe décrit dans ce modèle, lorsque $m = 3$.



P_1	a	d	g	j
P_2	b	e	h	k
P_3	c	f	i	l

FIG. *2.1 - : Exemple d'ordonnancement dans le modèle PRAM.*

2.2 Le modèle Local-memory PRAM

2.2.1 Description

Dans le modèle *PRAM*, les communications sont indissociables des calculs puisqu'elles sont incluses dans un cycle *lecture-calcul-écriture*. L'objectif poursuivi par Aggarwal, Chandra et Snir est de permettre une estimation de la complexité des communications pour les *PRAMs*. Pour cela, ils proposent un modèle de machine *PRAM-CREW* formé d'un ensemble de processeurs disposant chacun d'une **mémoire locale non limitée**. Ils appellent ce modèle **Local-memory PRAM** [ACS90]. Tous les accès mémoire sont pris en compte, mais ils ne le sont pas de la même façon. Deux types d'accès sont identifiés. Les accès à la **mémoire locale**, dont le **coût** est **négligé**, et les accès à la **mémoire globale**, de **coût unitaire**. Toute exécution d'un programme est précédée de la lecture et du chargement des données

(ce sont les noeuds du graphe qui n'ont aucun prédécesseur) depuis la mémoire globale et terminée par l'écriture des résultats dans la mémoire globale. Les processeurs peuvent exécuter des programmes différents sur des données différentes (*MIMD*), mais sont synchronisés. Une exécution est décrite comme un ensemble de **phases de calcul** et de **phases de communication** qui durent chacune une unité de temps. Pendant cette unité de temps, soit les processeurs lisent et/ou écrivent un mot dans la mémoire globale (phase de communication), soit ils exécutent une opération sur au plus deux mots disponibles dans leur mémoire privée (phase de calcul). Cette restriction sur la taille des données revient à considérer des graphes dont le grain de calcul est le plus fin possible. Du point de vue des communications, la possibilité de ne lire et de n'écrire qu'un mot à la fois peut être interprété comme un modèle de machine *1-port* [Rum94]. Les graphes pris en compte sont tous formés de tâches de coût de calcul unitaire. Par contre, les communications considérées peuvent ne pas être unitaires. En effet, les auteurs introduisent un paramètre **l** décrivant le nombre d'unités de temps nécessaires pour effectuer une communication. Dans le cas où les communications sont unitaires, les graphes de programme dont on dispose sont des graphes dits *UECT* (pour *Unit Execution and Communication Time*).

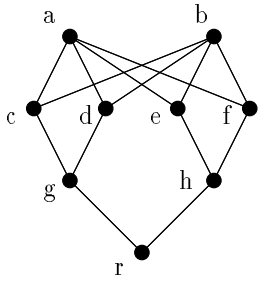
2.2.2 Ordonnancement dans le modèle LPRAM

Un ordonnancement est défini comme une suite d'étapes de calcul et d'étapes de communication. Le synchronisme sur lequel repose le fonctionnement de ce modèle de machine ne permet pas à un processeur de faire un calcul alors qu'un autre processeur fait un accès mémoire. Ainsi, la qualité des ordonnancements dépend de trois paramètres :

- le nombre de processeurs utilisés
- le nombre d'étapes de calcul
- le temps total de calcul

Le temps total de calcul est le résultat du nombre d'étapes de calcul additionné au nombre d'étapes de communication (éventuellement multiplié par l). Dans ce modèle, déterminer un bon ordonnancement est équivalent à chercher un **compromis** entre l'équilibrage de la charge de calcul et la localité des données et des calculs sur les processeurs. Ce compromis provient de ce que la diminution du nombre d'étapes de calcul (résultant d'une augmentation du nombre de processeurs) entraîne dans le cas général une augmentation du nombre d'étapes de communication. Ainsi, déterminer le meilleur ordonnancement revient à résoudre un problème dans lequel le nombre de processeurs est une des inconnues. La difficulté de recherche de ce compromis est aggravée par l'importance de la **localité** des données dans un calcul, c'est-à-dire par la recherche de composantes connexes du graphe pour économiser des communications, en séquentialisant le moins possible l'exécution du graphe.

2.2.3 Exemple



nombre de processeurs	nb d'étapes de com^{tion}	nb d'étapes de calcul
1	3	7
2	4	4
4	5	3

La	Lb	c	d	e	f	g	h	r	Er

La	Lb	c	d	g	Lh	r	Er
La	Lb	e	f	h	Eh		

La	Lb	c	Ld	g	Lh	r	Er
La	Lb	d	Ed				
La	Lb	e	Lf	h	Eh		
La	Lb	f	Ef				

FIG. 2.2 - : Exemple d'ordonnancement dans le modèle *LPRAM*. *Li* représente la lecture de la donnée i et Ej l'écriture de j dans la mémoire globale.

L'exemple de la figure 2.2 permet de mettre en évidence plusieurs choses. La première est que le nombre d'étapes de calcul est borné par le nombre de nœuds du graphe (pour le maximum) et par la hauteur du graphe moins un (pour le minimum). $n \geq$ nombre d'étapes de calcul $\geq h - 1$
Pour les communications, les bornes sont plus difficiles à établir.

2.2.4 Extensions du modèle LPRAM

Le modèle topologie-PRAM

Récemment, une extension du modèle PRAM prenant en considération la **topologie du réseau d'interconnexion** a été proposée dans [CF91]. L'idée de base est que le modèle PRAM peut être considéré comme un modèle idéalisé de machine parallèle à mémoire distribuée dont le réseau d'interconnexion serait totalement connecté. Cette extension (appelé XRAM) est en quelque sorte un modèle générique, en effet, elle est la base d'un ensemble de modèles dépendant de la topologie. Le nom de ce nouveau modèle est obtenu en substituant la première lettre du nom du réseau d'interconnexion au X (par exemple HRAM lorsque le réseau est un hypercube). La notion de localité introduite dans LPRAM est ici aussi prise en considération. Un processeur peut en effet accéder à sa propre mémoire ainsi qu'à celles de ses voisins pour un coût nul. Lorsque ce processeur désire acquérir des données qui n'appartiennent ni à lui ni à ses voisins, un **algorithme de routage** doit être mis en œuvre. Ainsi, en une unité de temps, chaque processeur peut lire une donnée située dans sa propre mémoire ou dans la mémoire locale d'un de ses voisins, exécuter une tâche et ranger le résultat dans sa mémoire ou dans la mémoire d'un de ses voisins. Dans le cas où un processeur a besoin d'une donnée qui n'est pas disponible dans le voisinage, un coût de communication est calculé, fonction de l'éloignement et de l'algorithme de routage.

Le modèle localité-PRAM

Ce modèle constitue une application restrictive du précédent. Les auteurs remarquent que le coût principal dans XRAM est dû au routage. De manière à limiter ce coût, les auteurs proposent un modèle dans lequel **seuls les accès de voisinage sont permis**. Toute autre communication qu'entre voisin n'est pas autorisée, et les communications de voisinage sont considérées de coût nul [BKT94]. Cette hypothèse de localité est exprimée par : *Un processeur P_i peut exécuter une tâche T_j si et seulement si tous les prédécesseurs de cette tâche ont été exécutées sur P_i ou sur un processeur voisin.* L'une des différences essentielle par rapport aux modèles précédents est la prise en compte de la structure algorithmique du programme à ordonnancer. Cette prise en compte permet d'ordonnancer de manière asymptotiquement optimale les grilles et les arbres binaires complets sur toutes les topologies classiques.un anneau infini.

2.3 Le modèle de Papadimitriou et Ullman

2.3.1 Description

Dans ce modèle d'exécution de machines parallèles à mémoire distribuée, on retrouve les deux types d'accès mémoire, l'accès local de coût nul et l'accès à distance qui coûte une unité de temps. Les auteurs essaient de définir une méthodologie d'utilisation des machines parallèles. Pour cela, ils tentent d'identifier des paramètres pour estimer les performances des algorithmes indépendamment de la machine cible [PU87]. Les critères qu'ils retiennent sont le **temps d'exécution (sans tenir compte des attentes dues aux communications)** et les **communications**. Le programme est modélisé par un graphe orienté sans cycle (*DAG*), avec les hypothèses d'exécution suivantes :

- un nœud[1] ne peut être exécuté que si tous ses prédécesseurs l'ont été,
- tous les nœuds ont des temps d'exécution **unitaires**,
- un processeur ne peut exécuter qu'un nœud par unité de temps,
- un nœud peut être **dupliqué**[2]. Les copies de ce nœud sont exécutées sur des processeurs distincts,
- soient deux nœuds u et v liés par une relation de précédence (par exemple u précède v) et tels que u est exécuté sur le processeur P et v sur le processeur P'. Si P est différent de P' alors l'arc $((P,u),(P',v))$ est appelé **arc de communication** (*communication arc*). Si P et P' représentent le même processeur alors le coût de la communication est nul et l'arc liant u et v n'est pas considéré comme étant un arc de communication.

Il s'agit d'un modèle de machine à mémoire distribué, on ne parle donc plus de *lecture* mais **de réception de message**, et une *écriture* devient un **envoi de message**. La grande différence

[1] la notion de *nœud* est ici équivalente à la notion de *tâche* définie en annexe
[2] voir la définition de la *duplication* en annexe

entre ce modèle et LPRAM est qu'un processeur peut gérer **plusieurs communications simultanément**. D'autre part, la duplication permet d'éliminer des communications par exécution d'une même tâche sur plusieurs processeurs différents. A partir de ces hypothèses et définitions, trois mesures sont retenues comme ayant une grande importance dans l'indication de l'efficacité d'un algorithme :

- le **temps de calcul** ne tenant pas compte des communications T_{calcul},
- le **nombre total de communications** générées par l'algorithme (*total message traffic*) noté T_{trafic},
- le **nombre maximum de communications** pour l'ensemble des chemins du graphe (*total elapsed time due to communications*) noté T_{delai}.

Si l'on se ramène à la notion d'arc de communication, T_{trafic} est égal à la somme de tous les arcs de communication du graphe, et T_{delai} au nombre maximum d'arcs de communication dans un chemin du graphe pour l'ensemble des chemins du graphe. Pour la mesure de T_{trafic} et de T_{delai}, lorsque des nœuds sont dupliqués, une seule copie par arc est considérée, c'est-à-dire qu'un arc (u, v) dans le graphe de départ sera comptabilisé comme arc de communication si et seulement s'il n'existe aucun couple $((P, copie(u)), (P', copie(v)))$ (où $copie(x)$ peut être une copie du nœud x ou le nœud lui-même) tel que P et P' soient confondus.

2.3.2 Ordonnancement dans le modèle PU

A partir des trois mesures définies dans le paragraphe précédent, de multiples critères de performance peuvent être construits. Si le but est de construire un algorithme qui n'engorge que peu ou pas le réseau, mais qui soit rapide, la somme $T_{calcul} + T_{trafic}$ est un critère qui peut donner une idée moyennée du trafic généré par l'algorithme. Si seule la date de fin d'exécution nous intéresse alors $T_{calcul} + T_{delai}$ semble être le critère le mieux adapté pour en avoir une idée. Lorsqu'un critère inclut le terme T_{trafic} alors le minimiser est souvent synonyme de rechercher un **compromis** entre le temps de calcul et le nombre de communications. En effet, le temps d'exécution diminue avec l'augmentation du nombre de processeurs et les communications augmentent avec l'augmentation du nombre de processeurs. Cependant, ponctuellement, avec l'utilisation de la duplication, une augmentation du nombre de processeurs n'entraîne pas une augmentation des communications. C'est le cas notamment des arbres qui s'exécutent de la racine vers les feuilles. Si un arbre complet de ce type est exécuté sur un nombre de processeurs égal au nombre de feuilles, tous les chemins peuvent être dupliqués, de telle sorte qu'aucune communication n'apparaissent. Le temps est lui égal à la hauteur de l'arbre. Ainsi, pour un anti-arbre binaire complet de hauteur h, formé de $n = 2^h - 1$ nœuds, la duplication des tâches permet de réduire le terme T_{trafic} de façon d'autant plus importante que le nombre de processeurs est grand. Si le nombre de processeurs est égal au nombre de feuilles de l'arbre, ce terme est réduit de $\frac{n-1}{2}$ à 0. Ainsi, si le critère est la somme $T_{calcul} + T_{trafic}$, la duplication permet de faire passer sa valeur de $log(n) + \frac{n-1}{2}$ à $log(n)$.

2.3.3 Exemple

Nous présentons un exemple simple de graphe afin d'illustrer la duplication et le mécanisme de calcul des différents temps de communications.

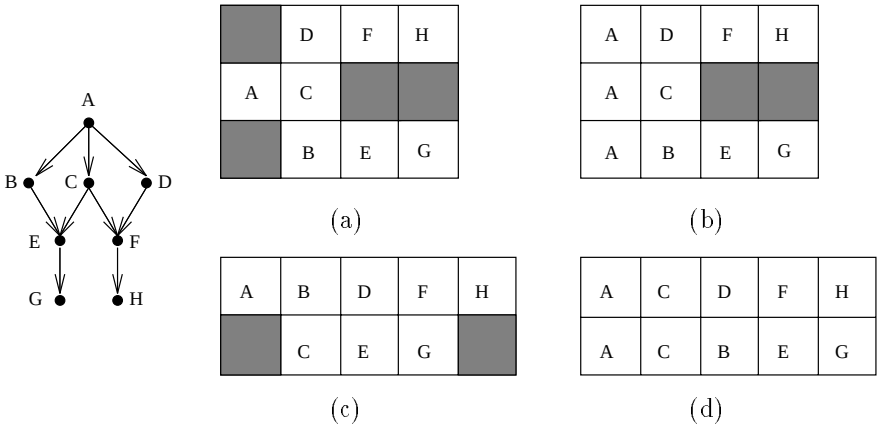


FIG. *2.3 - : Un exemple d'ordonnancement dans le modèle PU.*

On dispose du graphe illustré en figure 2.3. Le critère de performance considéré est la somme $T_{total} = T_{calcul} + T_{trafic}$. Le nombre de processeur n'est pas fixé, toutefois, la largeur du graphe[3] étant égal à trois, il serait vain de considérer plus de trois processeurs.

Le premier ordonnancement sur trois processeurs permet de minimiser le temps de calcul, par contre, le nombre de communications dans le cas où la duplication n'est pas utilisée (cas (a)) est égal à : 4. Les arcs de communication sont (A, B), (A, D), (C, F) et (C, E). Pour ce même nombre de processeurs, lorsque la première tâche est dupliquée (cas (b)), le nombre de communications n'est plus que de 2 ((C, F), (C, E)), ce qui entraîne $T_{total} = 4 + 2 = 6$. Cette valeur constitue la valeur optimale que l'on puisse obtenir pour ce critère sur trois processeurs.

Si l'on diminue le nombre de processeurs, le temps de calcul augmente de 1, par contre, le nombre de communications diminue. Dans le cas où la duplication n'est pas utilisée (cas (c)), le nombre de communications est égal à 3. Si les tâches A et C sont dupliquées toutes les communications sont éliminées (cas (d)) et $T_{total} = 5 + 0 = 5$. On obtient donc la minimisation de la somme pour un nombre de processeurs inférieur à la largeur du graphe et en utilisant la duplication.

Si le critère à optimiser n'est plus la somme $T_{calcul} + T_{trafic}$, mais la somme $T_{calcul} + T_{delai}$, la minimisation est atteinte à la fois avec deux et trois processeurs. En effet, dans le cas (b), le nombre maximum d'arcs de communication pour un chemin du graphe est égal à 1 (l'arc (C, E) dans le chemin (A, C, E, G)) ou l'arc (C, F) dans le chemin (A, C, F, H)). Ce qui donne un temps total égal à 5 (4+1). Il est à noter que le cas (a) réalise également la

[3]définition en annexe

minimisation de la somme, sans utiliser la duplication.
Cet exemple, par sa simplicité, illustre la complexité de la recherche d'un compromis entre minimisation du temps de calcul et minimisation des communications.

2.4 Le modèle Anderson Beame et Ruzzo

2.4.1 Description

L'extension proposée par les auteurs pour le modèle PRAM est liée à la notion de **graphes à flots de données**. Dans ce cadre, les communications sont prises en compte et la notion de **surcoût** (*overhead*) est introduite. Ce surcoût est égal à l'addition de deux termes : le temps d'inactivité des processeurs et le temps nécessaire pour une synchronisation ou pour ordonnancer. Le graphe est souvent considéré comme acyclique (DAG). De plus les tâches sont considérées comme étant **insécables** et **unitaires**.

Dans ce modèle, un algorithme est considéré d'autant plus efficace qu'il minimise la valeur du temps parallèle définie par la somme des temps séquentiel, d'inactivité et de communication, divisée par le nombre de processeurs, fraction notée par :

$$T_{par} = \frac{T_{seq} + T_{inact} + T_{com}}{p}$$

Les auteurs introduisent deux notions supplémentaires leur permettant de manipuler des groupes de tâches afin de minimiser les communications. Un **travail** (*job*) représente un **regroupement** de tâches unitaires, et le **graphe de travaux** (*schedule graph*) est le graphe résultant du regroupement des tâches en travaux. Les nœuds de ce graphe représentent les travaux et les arcs représentent les relations de précédence entre eux. L'exécution d'un travail se décompose en trois phases :

- le chargement, depuis la mémoire partagée, des données nécessaires au calcul des tâches appartenant au travail,
- l'ensemble des calculs des tâches,
- le rangement des résultats produits par les tâches appartenant au travail.

L'ensemble chargement des données et rangement des résultats est considéré de coût unitaire, quel que soit le volume des données en entrée et des résultats en sortie. Ainsi, le regroupement des tâches peut permettre un gain de communications, en effet, selon l'hypothèse "coût constant pour un accès à la mémoire", l'exécution d'un groupe de trois tâches sur un processeur nécessite quatre unités de temps (trois de calcul et une d'accès mémoire (chargement-rangement)) tandis que l'exécution séparée de ces trois mêmes tâches nécessiterait trois fois deux unités de temps soit, six unités de temps.

Un travail pour commencer son calcul doit avoir toutes les données nécessaires au calcul de toutes ses tâches. Ces données sont chargées **en une seule fois**. Et c'est en une seule

fois que les résultats sont rangés dans la mémoire globale partagée à la fin de l'exécution de la dernière tâche du travail. Ce type de fonctionnement revient à considérer que certaines lectures sont anticipées et certaines écritures retardées. Ce modèle diffère donc notablement des modèles présentés précédemment. Par contre, comme dans le modèle PU, un processeur peut lire plusieurs données simultanément, et peut écrire plusieurs résultats simultanément également. Cependant, ces deux opérations cumulées ne requièrent qu'une unité de temps, contre deux unités de temps dans le modèle PU.
Le surcoût dû aux synchronisations est proportionnel au nombre de travaux du graphe. Contrairement au modèle LPRAM, une lecture et une écriture sont traitées de la même façon que les autres instructions. De ce fait, l'ordonnancement ne peut plus être découpé en un ensemble de phases de calcul et de phases de communication.

2.4.2 Ordonnancement dans le modèle ABR

L'objectif des algorithmes d'ordonnancement pour le modèle ainsi défini est de minimiser le temps parallèle. Or, puisque T_{seq} et p sont constants, la minimisation du temps parallèle se ramène à celle du surcoût $T_{inact} + T_{com}$. On dispose pour minimiser cette quantité de la possibilité de regrouper les tâches entres elles. Un regroupement permet de gagner une unité de temps de communication par tâche regroupée. Cependant, le regroupement des tâches en travaux introduit des relations de précédence supplémentaires entre les tâches du graphe de départ. Ainsi, le regroupement est un facteur de réduction du parallélisme potentiel de l'application, et peut donc induire une augmentation de l'inactivité des processeurs. De la même façon, l'équilibrage du nombre de tâches par processeur est un facteur minimisation de l'inactivité des processeurs et entraîne généralement une augmentation des communications. Le but est alors, pour une application donnée, de trouver un **compromis** entre **l'équilibrage de la charge** (minimisation de l'inactivité) et **le regroupement des tâches** unitaires (minimisation des communications).

Concernant les hypothèses d'exécution des travaux, il faut noter qu'un travail ne peut échanger des informations durant son exécution, ce qui entraîne des restrictions sur l'ensemble des regroupements valides (tous les arcs de communication liant deux travaux doivent être orientés dans le même sens). D'autre part, l'ensemble des arcs entrant dans un travail correspondent à des données qui doivent être disponibles dès le début de ce travail, même si certaines de ces données ne sont utilisées que pour l'exécution de la dernière tâche appartenant à ce travail. De la même façon, les envoies de données correspondant aux arcs sortant d'un travail ne sont effectués qu'à la fin de ce travail. Ceci signifie qu'à l'intérieur d'un travail, aucun ordonnancement n'est utile. Pour un graphe donné, ordonnancer consiste donc à regrouper les tâches en un nombre minimum de travaux, en conservant assez de parallélisme pour que le temps d'inactivité ne soit pas trop important.

2.4.3 Exemple

Le graphe à flots de données présenté comporte 7 tâches unitaires. La mesure de l'ordonnancement est équivalente à celle du surcoût lui-même égal à la somme des périodes

d'inactivité et des communications. Dans le premier exemple d'ordonnancement, les travaux sont au nombre de 7 (de T_1 à T_7), un travail correspond ici à une tâche, donc le graphe des travaux est le même que le graphe à flots de données. Le surcoût est égal à 9 (2 périodes d'inactivité plus 7 travaux).

Dans le second ordonnancement, les travaux ne se réduisent pas à une tâche. Il y a trois travaux (de J_1 à J_3). Le premier travail correspond au regroupement des tâches T_1, T_2 et T_4. On remarque que le travail J_1 précède le travail J_3, résultat du regroupement, ce qui entraîne une contrainte de précédence entre les tâches T_4 et T_7. Dans le graphe des travaux, J_1 et J_2 sont indépendants et J_3 dépend des deux précédents. La conséquence immédiate est que le nombre de périodes d'inactivité augmente, on passe de deux unités (dans le premier exemple qui consistait en un regroupement trivial) à quatre. Par contre le nombre de communications diminue sensiblement de 7 à 3. Au total, le surcoût obtenu par cet ordonnancement est égal à 7, et constitue une amélioration.

Enfin, le troisième ordonnancement formé de quatre travaux. Pour ce regroupement, le dernier travail J_3 qui semblait être la cause du nombre important de périodes d'inactivité est scindé en deux travaux J_3 et J_4, ce qui permet d'obtenir un ordonnancement optimal avec un surcoût de 5.

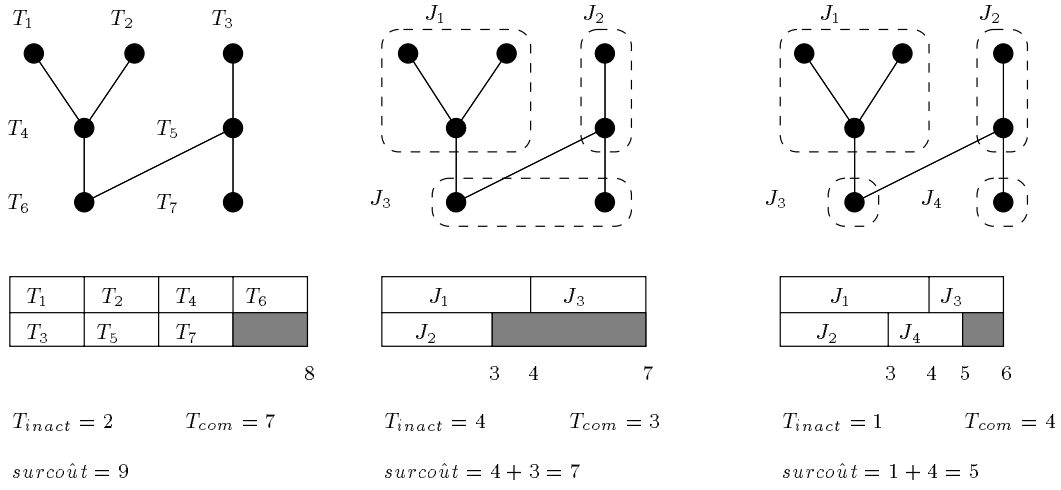


FIG. *2.4 - : Importance du regroupement pour ordonnancer dans le modèle ABR.*

2.5 Le modèle de Papadimitriou et Yannakakis

2.5.1 Description

Comme pour le modèle PU, le travail des auteurs a pour but d'évaluer les performances d'un algorithme modélisé par un graphe orienté sans cycle, quel que soit l'ordinateur parallèle. Leur méthode s'appuie sur l'identification d'un unique paramètre pour modéliser une

machine, **le rapport entre temps d'exécution d'une instruction et temps de communication d'un message** (*message-to-instruction ratio*), noté τ. Dans tous les modèles de machines à mémoire partagée, la valeur de ce paramètre est égale à une constante. Lorsque sont considérées des machines à mémoire distribuée, τ dépend du réseau d'interconnexion, et du nombre de processeurs dont dispose la machine. Dans le modèle PU, les problèmes liés à l'architecture de la machine ont été contournés par l'utilisation de plusieurs mesures de performance. Dans le présent modèle, une seule mesure de performance est retenue, le minimum des maximums des dates de fin d'exécution, encore appelé *makespan* et noté C_{max}. Une machine avec un réseau d'interconnexion particulier est identifiée par la valeur de τ. Si p est le nombre de processeurs, pour un réseau en anneau la valeur de τ est de l'ordre de p, s'il s'agit d'une grille, $\tau = O(\sqrt{p})$, pour un hypercube, $\tau = O(log(p))$.

Enfin, dans ce modèle, le **recouvrement des communications par des calculs** est autorisé. Cette hypothèse revient à considérer que dans la machine, un processeur est dédié au traitement des communications. L'algorithme est une fois encore modélisé par un graphe orienté sans cycle, et les hypothèses d'exécution sont les suivantes :

- un processeur ne peut exécuter qu'une tâche par unité de temps,
- toutes les tâches ont des temps d'exécution unitaires,
- une tâche peut être dupliquée,
- soit une tâche u exécutée à la date t sur le processeur P, soit une tâche v successeur immédiat de u, alors, soit v est exécutée sur P à partir de la date $t + 1$, soit v est exécutée sur un autre processeur à partir de la date $t + \tau + 1$,
- le **recouvrement**[4] des communications par des calculs est autorisé.

Cette dernière hypothèse permet de masquer certaines communications. Le statut des instructions *envoyer-message* et *recevoir-message* change. Dans les modèles précédents, soit ces instructions (*envoyer-message* et *recevoir-message* ou *lire-donnée* et *écrire-donnée*) sont noyées dans chaque calcul atomique (PRAM), soit elles apparaissent clairement (LPRAM, PU, ABR), et leur coût ne peut être ramené à 0 que si la contrainte de localité d'exécution est vérifiée. Avec le recouvrement, les communications existent mais leur coût n'est pris en compte que lorsqu'elles ne sont pas effectuées concurremment avec une instruction de calcul. Le recouvrement permet de faire un envoi et une réception de message en filigrane d'un calcul sur chacun des processeur émetteur et récepteur.

2.5.2 Ordonnancement dans le modèle PY

L'un des objectifs des algorithmes d'ordonnancement dans ce modèle est la minimisation de la date de fin d'exécution. Contrairement au modèle PU, les communications non recouvertes vont apparaître dans le diagramme de Gantt[5]. Cette persistance dans le diagramme

[4] cette notion est définie en annexe
[5] défini en annexe

implique que cette communication sera prise en compte pour tous les chemins passant par le nœud qui est en attente des données de cette communication, ce qui constitue une différence par rapport au modèle PU qui ne prenait en considération que les arcs de communication rencontrés sur un chemin donné.

Les techniques d'ordonnancement ne sont plus tout à fait les mêmes. Le **regroupement** (*clustering*) des tâches prend une importance toute particulière puisqu'il permet à la fois de profiter de la localité des données et de rassembler du calcul pour recouvrir les communications. A la différence du modèle ABR dans lequel les travaux constituent des groupes de tâches fermés aux communications, en ce sens qu'un processeur exécutant un travail ne communique pas avec les autres processeurs, les groupes de tâches de PY sont ouverts aux communications. Par contre, comme dans PU, un processeur peut communiquer plusieurs messages à la fois. Cette hypothèse déjà admise dans le modèle PU caractérise des machines dont les possibilités de communication ne sont pas bornées ou possédant un nombre de ports ou de liens égaux au maximum des degrés sortants et entrants des noeuds du graphe. Il existe déjà un grand nombre de résultats pour ce modèle. Les concepteurs de ce modèle ont mis au point un algorithme qui permet d'obtenir des ordonnancements distants au plus d'un facteur deux d'un ordonnancement optimal (avec utilisation de la duplication).

Pour une valeur de τ égale à 1, une variante de ce modèle a été proposé par Rayward-Smith, modèle qui permet le recouvrement des communications par des calculs, mais qui n'autorise pas la duplication des tâches.
L'ordonnancement des arbres avec les hypothèses de ce modèle d'exécution est traité dans la partie deux.

2.5.3 Exemple

Nous présentons un exemple simple pour illustrer les hypothèses de duplication et de recouvrement des communications par des calculs.

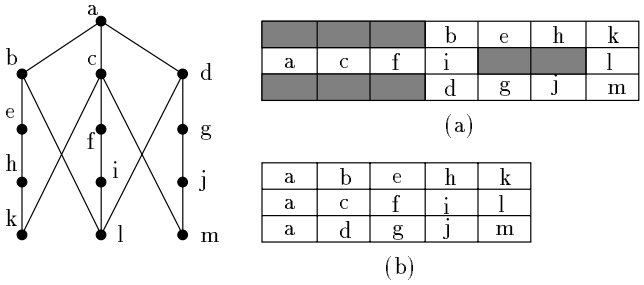


FIG. *2.5 - : Importance du recouvrement et de la duplication dans le modèle PY. Dans cet exemple* $\tau = 2$.

Le nombre de processeurs est considéré comme fixé à trois et $\tau = 2$. Dans le cas (a), la duplication n'est pas employée, ce qui entraîne un décalage dans le déroulement de l'ordonnancement. Par contre, lorsque la duplication est employée (cas (b)), le recouvrement apparaît clairement. Durant l'exécution des tâches e et h, la communication de c vers k est masquée. Il en est de même pour les communications de c vers m, de b vers l et de d vers l.

2.5.4 Le modèle RS

Un modèle similaire à PY a été proposé par Rayward-Smith [RS87]. Les hypothèses d'exécution sont sensiblement les mêmes :

- les tâches sont toutes de durée unitaire,
- les communications sont également de durée unitaire,
- les tâches **ne peuvent pas être dupliquées**,
- le recouvrement des communications par du calcul est autorisé

Le fait de considérer à la fois les temps de calcul et les coûts de communications unitaires a fait appelé ce modèle **UECT**. Ainsi, pour la plupart des problèmes d'ordonnancement pour lesquels il est fait mention d'un graphe *UECT*, il est entendu que le modèle d'exécution sous jacent est le modèle RS.

2.6 Similitudes et différences entre modèles

A l'exception du modèle PRAM, ces modèles possèdent quelques points communs. La première hypothèse d'exécution que l'on retrouve partout est celle que l'on peut appeler **hypothèse de localité**, qui permet à un processeur d'enchainer des calculs sans être obliger d'effectuer des accès mémoire couteux.

On peut ensuite distinguer deux types de comportement lors des accès mémoire. D'un côté les modèles qui demandent une synchronisation de tous les processeurs pour les accès mémoire (LPRAM, PU) et de l'autre les modèles qui considèrent que les communications sont gérées localement par chaque processeur (ABR, PU, PY, RS). On note que le modèle PU est présent dans les deux catégories, cela provient du fait que deux mesures du nombre de communications sont proposées dans ce modèle. L'une est globale T_{trafic}, et sa prise en compte se rapproche des modèles à accès mémoire synchrones et l'autre est locale T_{delai} et ne concerne que le chemin comportant le plus d'arcs de communications.

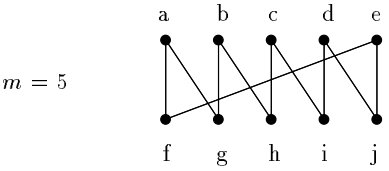
Enfin, une distinction peut être faite entre les modèles pour lesquels le coût des communications est effectif (LPRAM, PU, ABR) et ceux qui permettent le recouvrement total des communications (PY, RS).

2.6.1 Similitudes entre les modèles LPRAM et PU

Pour cette analyse, toutes les phases de communication pour la lecture des entrées et pour l'écriture des résultats sont omises. Sous cette condition, il existe de nombreuses similitudes entre les deux modèles. En effet, si l'on considère la somme des termes T_{calcul} et T_{trafic} de PU et la somme des termes ϕ_{calcul} et ϕ_{com} représentant respectivement le nombre de phases de calculs et le nombre de phases de communications dans le modèle LPRAM, alors, si m est le nombre de processeurs :

$$\phi_{calcul} + \phi_{com} \leq T_{calcul} + T_{trafic} \leq \phi_{calcul} + m\phi_{com}$$

En effet, dans PU, toutes les communications sont comptées, c'est-à-dire qu'une unité de temps est ajoutée par arc de communication recensé. Or, dans LPRAM, plusieurs lecture-écriture peuvent être effectuées dans une même phase de communication (un tel cas est illustré par l'exemple qui suit la description de LPRAM ci-dessus) donc, $\phi_{calcul} + \phi_{com} < T_{calcul} + T_{trafic}$. Mais, si, dans un ordonnancement LPRAM, chaque phase de communication ne comprend qu'une lecture-écriture, c'est-à-dire ne correspond qu'à un seul arc dans le graphe de tâches, alors ces deux sommes sont égales $\phi_{calcul} + \phi_{com} = T_{calcul} + T_{trafic}$. D'autre part, lors d'une même phase de communications, un processeur peut écrire une donnée et en lire une, donc au total, dans une même phase de communications, m lectures et m écritures peuvent être effectuées. Ces m opérations de lecture-écriture correspondent à m arcs dans le graphe de tâches, donc, la valeur de T_{trafic} pour ce genre de situation est égale à $m\phi_{com}$. Et, dans tous les cas, $\phi_{calcul} = T_{calcul}$. La figure 2.6 illustre une telle situation. Cette inégalité signifie que si l'on dispose d'un ordonnancement dans le modèle LPRAM alors il est possible de le transformer en un ordonnancement pour le modèle PU dont on peut borner la qualité. De la même façon la donnée d'un ordonnancement dans PU dont la somme est égale à S nous garantie qu'il existe un ordonnancement dans LPRAM dont le nombre de phases cumulées (calcul plus communication) est inférieur ou égal à S.



a	E(a) L(e)	f
b	E(b) L(a)	g
c	E(c) L(b)	h
d	E(d) L(c)	i
e	E(e) L(d)	j

Ordonnancement dans LPRAM

$\phi_{calcul} = 2$

$\phi_{com} = 1$

a	f
b	g
c	h
d	i
e	j

Ordonnancement dans PU

$T_{calcul} = 2$

$T_{trafic} = 5$

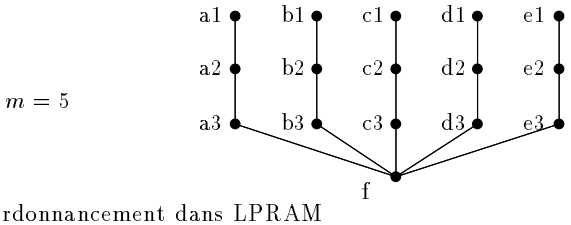
FIG. 2.6 - : Comparaison des valeurs des paramètres du modèle LPRAM et du modèle PU lorsque T_{trafic} est pris en compte.

Dans ce genre de situation, $T_{calcul} + T_{delai}$ donne une borne inférieure, d'où la seconde inégalité :

$$T_{calcul} + T_{delai} \leq \phi_{calcul} + \phi_{com}$$

En effet, lorsqu'un noeud possède plusieurs arcs entrant et qu'au moins deux de ses prédécesseurs n'ont pas été exécutés sur le même processeur, alors l'exécution de ce noeud demande au préalable au moins deux phases de communications pour acquérir les données qui sont nécessaires à son exécution. Or, tout chemin passant par ce noeud ne détecte en ce noeud, au plus, qu'un arc de communication, donc T_{delai} est incrémenté de 1. La figure 2.7 illustre cette situation. De plus, si l'on note A le maximum des degrés entrant pour l'ensemble des

noeuds du graphe, alors $\phi_{calcul} + \phi_{com} \leq T_{calcul} + m \times A \times T_{delai}$. Il faut noter toutefois que le modèle LPRAM présenté par Aggarwal et al. n'est proposé que pour des graphes dont l'arité entrante est égale à deux.



Ordonnancement dans LPRAM

a1	a2	a3	L(b3)	L(c3)	L(d3)	L(e3)	f
b1	b2	b3	E(b3)				
c1	c2	c3	E(c3)				
d1	d2	d3	E(d3)				
e1	e2	e3	E(e3)				

Ordonnancement dans PU

a1	a2	a3	f
b1	b2	b3	
c1	c2	c3	
d1	d2	d3	
e1	e2	e3	

$\phi_{calcul} = 4$

$\phi_{com} = 4$

$T_{calcul} = 4$

$T_{delai} = 1$

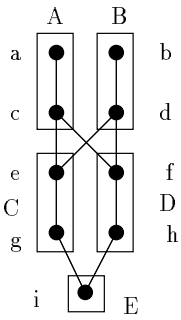
$T_{trafic} = 4$

FIG. 2.7 - : Comparaison des valeurs des paramètres du modèle LPRAM et du modèle PU lorsque T_{delai} est pris en compte.

2.6.2 Similitudes entre ABR et LPRAM

Comme pour le cas précédent, le nombre de phases de communications nécessaires pour acquérir les données d'entrée dans LPRAM sont omises, par contre les phases de communication pour le rangement des résultats sont prises en compte. Il est difficile de comparer les modèles ABR et LPRAM puisque leur mode de fonctionnement n'est pas le même. Nous nous restreignons aux ordonnancements dans ABR qui sont synchrones, c'est-à-dire ceux pour lesquels l'exécution des travaux est synchrone pour l'ensemble des processeurs, même date de début et même date de fin d'exécution. Nous verrons dans le second chapitre que ce type d'ordonnancements peut être obtenu pour des graphes particuliers. Alors, le modèle ABR peut être vu comme une *LPRAM* dont les phases de communications contigues ne seraient comptabilisées que pour une unité de temps. En effet, il est possible lors d'une fin de travail que plusieurs lectures et plusieurs écritures se fassent comme le montre l'exemple de la figure 2.8. Ce modèle compte un coût de communication égal à un quelque soit le nombre de données reçues par un travail, et quel que soit le nombre de résultats produits par ce même travail, alors que le modèle LPRAM compterait autant de phases de communications

qu'il y a de paires lecture d'une donnée-écriture d'un résultat. Donc $\frac{T_{com}}{m} \leq \phi_{com}$.



Ordonnancement dans LPRAM

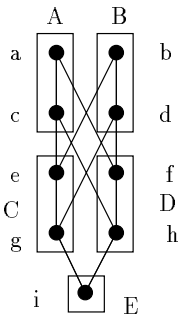
a	c	E(c) L(d)	e	g	L(h)	i	E(i)
b	d	E(d) L(c)	f	h	E(h)		

$\phi_{calcul} = 5$

$\phi_{com} = 3$

Ordonnancement dans ABR

A	C	E
B	D	



surcoût = $T_{inact} + T_{com} = 2 + 5 = 7$

$T_{seq} = 9$ $T_{par} = 8$

Ordonnancement dans LPRAM

a	c	E(a) L(b)	e	E(c) L(d)	g	L(h)	i	E(i)
b	d	E(b) L(a)	f	E(d) L(c)	h	E(h)		

$\phi_{calcul} = 5$

$\phi_{com} = 4$

FIG. *2.8 - : Comparaison d'ordonnacements pour les modèles ABR et LPRAM.*

D'autre part, dans LPRAM, le nombre de phases de calcul ne tient pas compte de l'activité des processeurs et du type d'opérations qu'ils effectuent, donc chaque période d'inactivité additionnée à T_{inact} dans ABR est comptabilisé dans ϕ_{calcul}. Chaque période d'inactivité peut être considéré comme une opération NOP, et compte pour un m^{ieme} d'une phase de calcul. Ainsi, $\frac{T_{seq} + T_{inact}}{m} = \phi_{calcul}$. Finalement, si l'on dispose d'un ordonnancement dans le modèle LPRAM, alors on est sûr qu'il en existe un dans ABR tel que :

$$\frac{T_{seq} + T_{inact} + T_{com}}{m} \leq \phi_{calcul} + \phi_{com}$$

2.6.3 Similitudes entre les modèles PU et PY

Dans le modèle PY ou RS, le recouvrement des communications par des calculs est autorisé. Dans le modèle PU, suivant le critère retenu, soit aucun recouvrement n'est autorisé, c'est le cas si l'on prend en considération T_{trafic}, soit **une partie** des communications est recouverte. De ce point de vue, les mesures de communications retenues dans le modèle PU ne permettent pas de conclure quant à la légitimité de cette hypothèse. En effet, comme l'illustre la figure 2.9, pour certains graphes, le nombre de communications retenues par la mesure de T_{delai} est supérieur au nombre de communications qui apparaitraient si le recouvrement était autorisé comme c'est le cas dans un ordonnancement de ce même graphe dans

le modèle PY (cas(a) de la figure avec $\tau = 1$). Ceci laisse à penser que cette hypothèse n'est pas valide. Par contre, pour d'autres graphes, il existe certains noeuds terminaux d'arcs de communications, en lesquels la communication n'est pas comptée, ce qui revient à considérer qu'une partie des communications est recouverte (cas(b) de la figure). Si l'on examine les deux arcs de communications du cas (b), le noeud terminal de l'arc (c, d) appartient au chemin (a, b, d, e, g), de même que le noeud terminal de l'arc (f, g). Or une seule communication est comptabilisée dans T_{delai}, alors que le statut de ces deux noeuds est le même. Donc, si l'on examine la somme $T_{calcul} + T_{delai}$, la valeur de cette somme, ne donne pas d'indication sur la date de fin d'exécution des tâches dans un modèle d'exécution autorisant le recouvrement des communications par des calculs.

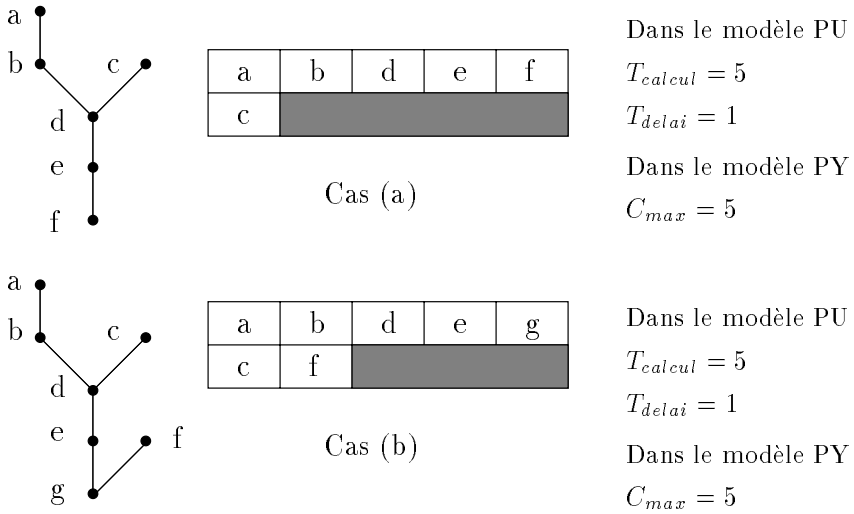


FIG. 2.9 - : Prise en compte du recouvrement dans le modèle PY par rapport à un ordonnancement dans le modèle PU.

2.6.4 Conclusion

Dans la majeure partie des nouveaux modèles d'exécution qui sont proposés aujourd'hui, les communications sont prises en compte. La différence essentielle entre tous ces modèles se situe dans la manière de les prendre en compte. Dans le modèle LPRAM, toutes les phases de communications sont comptées. Une communication est un couple lecture-écriture. Une phase de communication peut donc correspondre à un nombre de communications compris entre un et le nombre de processeurs. Dans le modèle PU, si l'on considère T_{trafic}, toutes les communications sont comptées. Ce nombre correspond au nombre de couples lecture-écriture de LPRAM, qui sur le graphe est identifié par un arc de communication. Tous les autres arcs sont ignorés du fait de l'hypothèse de localité. En revanche, T_{delai} ne rend compte que des communications pour lesquelles les sommets d'un arc de communications appartiennent tous les deux au chemin parcouru. Ce qui correspond au nombre maximum de couples lecture-écriture appartenant à un chemin, pour l'ensemble des chemins. D'une certaine manière, dans le modèle ABR une situation similaire se produit puisque le nombre de communications est égal au nombre de travaux. Mais, au lieu d'une valeur maximale, la valeur considérée, T_{com},

est une valeur moyenne. C'est une valeur proche également de T_{trafic}, bien que pondérée par le nombre de processeurs, mais qui correspond à un modèle général de machines pour lequelle un ensemble de communications faites au même moment est considéré comme un flot de données de coût unitaire. Dans tous ces modèles, les communications ne peuvent pas être masquées. Certains paramètres permettent de considérer qu'il existe des communications qui sont masquées, c'est le cas notamment de T_{delai} pour le modèle PU ou de T_{com} pour le modèle ABR, mais dès qu'une communication a lieu, il n'est pas possible d'avoir l'un ou l'autre de ces paramètres qui soit égal à 0. Ce n'est plus le cas pour les modèles PY et RS qui correspondent à des modèles généraux de machines à mémoire distribuées. Dans ce type de modèle, le recouvrement des communications est garant de l'efficacité des algorithmes. Pour cela, les techniques envisagées doivent exploiter l'hypothèse de localité des données, ce qui est réalisable à l'aide de stratégies de regroupement. D'autre part, le recouvrement est effectif lorsque les communications sont anticipées en envoi de messages et retardées en reception de messages, c'est-à-dire lorsque les communications sont effectuées de façon asynchrones. Ce dernier point correspond à des couples lecture-écriture destructurés, c'est-à-dire pour lesquelles l'écriture est effectuée avant la lecture. C'est en partie ce qui se passe dans le modèle ABR, ce qui expliquera en partie la bonne adéquation de certaines stratégies mises en oeuvre pour le modèle RS (partie 2) et réutilisables pour ce modèle (chapitre suivant de la présente partie).

Chapitre 3

Ordonnancement d'arbres dans le modèle ABR

3.1 Introduction

Dans le modèle proposé par Anderson, Beame et Ruzzo et décrit dans le chapitre précédent, la qualité d'un ordonnancement est capturé par le **surcoût** (*overhead*) égal à la somme des temps d'inactivité et du nombre de travaux. La constrution d'un bon ordonnancement passe par la recherche du **compromis** entre le parallélisme, générateur de communication, et la séquentialisation des calculs, source de temps d'inactivité. L'objectif est de minimiser la date de fin d'exécution, ce qui revient à minimiser le surcoût. Dans ce chapitre, nous présentons une étude de **l'ordonnancement** d'une classe de graphes que sont les **arbres**. Le problème de la détermination du minimum des maximums des dates de fin d'exécution est montré NP-complet pour des arbres binaires sur deux processeurs. Nous lions ce modèle avec celui décrit par Rayward-Smith [RS87], et nous utilisons un algorithme qui produit des ordonnancements optimaux pour des arbres sur deux processeurs dans ce dernier modèle pour construire de bons ordonnancements dans le modèle qui nous occupe.

3.2 Etudes de complexité

Nous étudions dans cette partie la complexité de deux problèmes d'ordonnancement. Le premier concerne l'ordonnancement des chaînes sur deux processeurs, et le second, le problème de l'ordonnancement des arbres binaires parfaits sur deux processeurs également. Ces problèmes sont notés :

$$P_2 \mid chaînes,\ p = 1 \mid surcoût$$
$$P_2 \mid arbres\ binaires\ parfaits,\ p = 1 \mid surcoût$$

3.2.1 Etude de $P_2 \mid chaînes,\ p = 1 \mid surcoût$

Lemme
S'il existe une division de l'ensemble des chaînes en deux sous-ensembles de chaînes dont le

nombre total de tâches est égal de part et d'autre, alors, il n'existe aucun ordonnancement optimal de cet ensemble de chaînes sur deux processeurs ayant plus de deux travaux.

Preuve
La preuve est évidente, il suffit de considérer un tel ordonnancement, le *surcoût* est égal à la somme du nombre de *travaux* (2) et du nombre de périodes d'inactivité (0), c'est-à-dire 2. Toute autre répartition ne peux donner de meilleur résultat.

Théorème
Le problème consistant à déterminer s'il existe un ordonnacement d'un ensemble de chaînes formées de tâches unitaires sur deux processeurs avec un surcoût égal à 2 est un problème NP-Complet.

Preuve
La preuve est très simple, il s'agit d'une réduction immédiate au problème PARTITION. Commençons par l'énoncé des deux problèmes de décision associés :

INSTANCE de $P_2 \mid \mathit{chaînes},\ p = 1 \mid \mathit{surcoût}$: un ensemble de chaînes dont le nombre total de noeuds est un nombre pair.
QUESTION : Existe-t-il un ordonnancement de ces chaînes sur deux processeurs de telle sorte que le *surcoût* total soit égal à 2?

INSTANCE de PARTITION [GJ79] : un ensemble fini A d'éléments, et une taille $t(a) \in Z^+$ pour chaque $a \in A$.
QUESTION : Existe-t-il un sous-ensemble A' de A tel que :

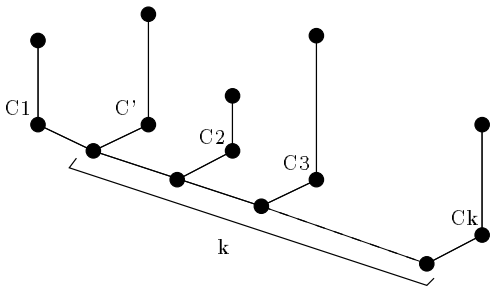
$$\sum_{a \in A'} t(a) = \sum_{a \in A - A'} t(a) ?$$

Le codage de $P_2 \mid \mathit{chaînes},\ p = 1 \mid \mathit{surcoût}$ ne nécessite qu'un nombre polynomial de symboles pour représenter n'importe quel problème de n chaînes de longueurs entières. Le problème est donc dans NP.

Soit une instance de $P_2 \mid \mathit{chaînes},\ p = 1 \mid \mathit{surcoût}$, on peut construire un instance de PARTITION en temps polynomial. En effet, à chaque chaîne est associée une longueur représentée par un nombre entier. Les éléments de l'ensemble A de PARTITION sont les chaînes et la taille d'un de ces éléments correspond au nombres de tâches dont il est formé. Supposons maintenant que la réponse à $P_2 \mid \mathit{chaînes},\ p = 1 \mid \mathit{surcoût}$ soit **oui**, alors il existe un regroupement de l'ensemble des chaînes en deux sous-ensembles égaux, donc il existe un sous ensemble de A tel que la somme des entiers de ce sous-ensemble est égale à la somme des entiers ne lui appartenant pas, mais appartenant à A, ce qui entraine une réponse positive pour PARTITION.
De la même façon, une réponse **oui** à PARTITION implique qu'il existe une partition de A en deux sous-ensemble égaux, et par conséquent qu'il est possible de partitionner l'ensemble des chaînes en deux sous-ensembles dont le nombre total de tâches est égal de part et d'autre, ce qui constitue une réponse positive au problème $P_2 \mid \mathit{chaînes},\ p = 1 \mid \mathit{surcoût}$. □

3.2.2 Etude de $P_m \mid arbres\ binaires,\ p = 1 \mid surcoût$

Nous appelons CC_m (pour *Chenille* formée de *Chaînes* et m processeurs) la famille des arbres binaires dont un élément est représenté en figure 3.1. Pour cette famille d'arbres, la somme des tâches des chaînes C_i est divisible par $m-1$. De plus, le nombre de tâche d'une chaîne quelconque C_i est inférieur ou égal à celui de C', par ailleurs égal à : $C' = \frac{1}{m-1}\sum_{i=1}^{i=k} C_i$.



Pour $m = 3$

C'	k
C"	
C"'	

FIG. *3.1 - : Exemple d'un arbre appartenant à CC_m.*

Un exemple d'ordonnancement est donné pour $m = 3$. Dans ce cas, le nombre de tâches de la chaîne C' est égal à la moitié du nombre total de tâches des chaînes C_i (pour $i = 1$ à $i = k$).

Lemme

Soit A_k un représentant de la famille CC_m, formé de k chaînes C_i, d'une chaîne Ch de longueur k à laquelle sont rattachées toutes les racines des chaînes C_i, et d'une chaîne C' de longueur $\frac{1}{m-1}\sum_{i=1}^{i=k} C_i$ (supérieure à la longueur de n'importe quelle chaîne C_i) reliée à la chaîne en son noeud de plus grande hauteur. Soit l'ordonnancement qui alloue à un processeur deux travaux constitués, pour le premier de la chaîne C', et pour le second de la chaîne Ch, et qui alloue à chacun des $m-1$ processeurs un ensemble de chaînes (chaque ensemble ne formant qu'un seul travail) dont la somme des tâches est égale à la longueur de C'. Alors, cet ordonnancement, s'il existe, est optimal et est l'unique ordonnancement optimal.

Preuve

Lorsque l'on parle d'unique ordonnancement optimal, cela signifie unique structure d'ordonnancement, c'est-à-dire qu'il n'est pas possible de trouver un ordonnancement optimal qui coupe une chaîne C_i en plusieurs, mais il est possible qu'il existe plusieurs répartitions de l'ensemble des chaînes C_i entre les $m-1$ processeurs.

La première remarque que l'on peut faire est que le nombre de travaux est minimum. En effet, il n'est pas possible de diminuer le nombre de travaux pour les $m-1$ processeurs puisqu'ils n'en ont qu'un. De plus, comme il s'agit d'un arbre, à moins d'exécuter la totalité des tâches sur un seul processeur, un des processeurs doit exécuter au moins deux travaux, le dernier devant contenir la racine de l'arbre.

D'autre part, si la taille des travaux est conservée, il n'est pas possible de trouver une autre

répartition puisque C' précède la chaîne Ch dans sa totalité, et que la division d'une quelconque chaîne C_i entrainerait une augmentation du nombre de travaux d'au moins deux, et augmenterait la date de fin d'exécution d'au moins une unité de temps, donc une telle répartition ne pourrait pas être optimale. Si la taille des travaux est changée, et si le premier travail est augmenté, des éléments de Ch doivent être inclus dans ce premier travail, or toutes les chaînes précèdent Ch, ce qui séquentialise l'exécution de tous les travaux et ne peut pas aboutir à un ordonnancement optimal. De même, si la taille du premier travail est raccourcie, des tâches appartenant à des chaînes C_i n'ont pas pu être exécutées durant le premier travail, donc les tâches appartenant à la chaÎne Ch vont être réparties entre plusieurs travaux, ce qui entraine une augmentation de la date de fin d'exécution d'au moins une unité de temps, entrainant une perte d'optimalité.
Ainsi, l'ordonnancent décrit dans le lemme est, s'il existe, le seul ordonnancement optimal pour un arbre A_k appartenant à la famille CC_m. □

Théorème
Si l'on note par $N_{C'}$ le nombre de tâches que comporte la chaîne C', alors, le problème consistant à déterminer s'il existe un ordonnacement dont la date de fin d'exécution des tâches est égale à $N_{C'} + k + 2$ est NP-complet.

Preuve
Nous décrivons la preuve pour $m = 3$. Appelons ce problème $P_3 \mid CC_3 \mid surcoût$ pour simplifier l'énoncé. La preuve consiste en une réduction au problème PARTITION.

INSTANCE de $P_3 \mid CC_3 \mid surcoût$: un arbre de la famille CC_3. La somme des tâches des chaînes C_i ($\sum N_{C_i}$) est divisible par 2, et la somme des tâches de la chaîne C' est égale à $\frac{1}{2}\sum_{i=1}^{i=k} N_{C_i}$.
QUESTION : Existe-t-il un ordonnancement de cet arbre sur trois processeurs de telle sorte que la date de fin d'exécution des tâches soit égale à $\frac{1}{2}\sum_{i=1}^{i=k} N_{C_i} + k + 2$?

INSTANCE de PARTITION [GJ79] : un ensemble fini A d'éléments, et une taille $t(a) \in Z^+$ pour chaque $a \in A$.
QUESTION : Existe-t-il un sous-ensemble A' de A tel que :

$$\sum_{a \in A'} t(a) = \sum_{a \in A - A'} t(a) ?$$

Nous remarquons tout d'abord que le codage de $P_3 \mid CC_3 \mid surcoût$ ne nécessite qu'un nombre polynomial de symboles pour représenter n'importe quel problème basé sur un arbre appartenant à la famille CC_3.

Soit une instance de $P_3 \mid CC_3 \mid surcoût$, on peut construire un instance de PARTITION en temps polynomial. En effet, à chaque chaîne C_i est associé une longueur représentée par un nombre entier. Les éléments de l'ensemble A de PARTITION sont les chaînes C_i. La taille de ces éléments correspond à la longueur de ces chaînes.

Supposons maintenant que la réponse à $P_3 \mid CC_3 \mid surcoût$ soit **oui**, alors il existe un regroupement des chaînes C_i en deux sous-ensembles égaux, donc il existe un sous ensemble de A tel que la somme des entiers de ce sous-ensemble est égale à la somme des entiers ne lui appartenant pas, mais appartenant à A, ce qui est une réponse positive pour PARTITION. De la même façon, une réponse **oui** à PARTITION implique qu'il existe une partition de A en deux sous-ensemble égaux, et par conséquent entraine une réponse positive pour le problème $P_3 \mid CC_3 \mid surcoût$. □

3.2.3 Etude de $P_2 \mid arbres\ binaires,\ p = 1 \mid surcoût$

Pour $m = 2$ le problème reste NP-difficile. L'arbre est légèrement différent du précédent. Une famille d'arbres pour lesquels le problème est NP-difficile est illustrée en figure 3.2.

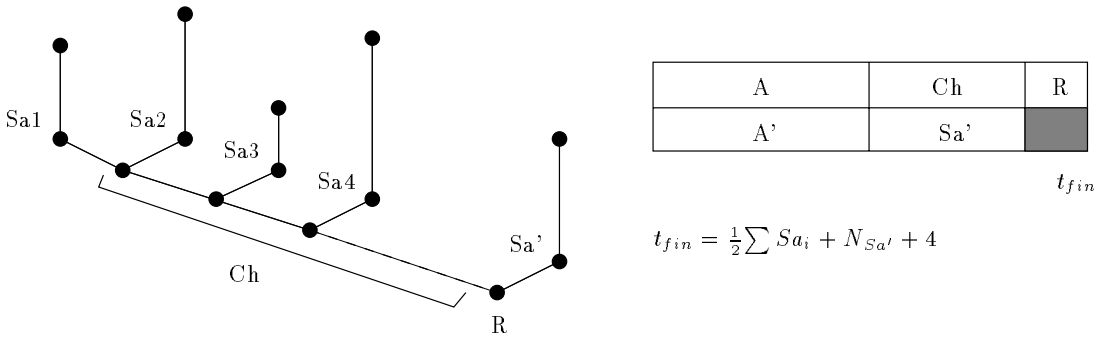


FIG. *3.2 - : Exemple d'un arbre appartenant à CC_2.*

Les représentants de la famille CC_2 ont les caractéristiques suivantes :

- ils sont formés de k sous-arbres, notés Sa_i,
- $N_{Sa_1} + N_{Sa_2} > \left\lfloor \frac{N_{total} - 1}{2} \right\rfloor$. Cette hypothèse interdit l'allocation du sous-arbre formé par Sa_1 et Sa_2 à un seul processeur, c'est-à-dire, interdit, indirectement, qu'un seul des noeuds de la chaîne Ch appartienne à un travail dont la date de début d'exécution est $t = 0$.
- $\sum Sa_i$ est un nombre pair,
- le nombre de noeuds de Sa' est égal au nombre de noeuds de la chaîne Ch.

A partir de toutes ces hypothèses, le problème de la recherche d'un ordonnancement dont la date de fin d'exécution des tâches soit égale à $\frac{1}{2}\sum Sa_i + N_{Sa'} + 4$ se ramène au problème de PARTITION avec les mêmes arguments que précédemment. Le seul argument important étant que pour ce type d'arbres, il n'est pas possible de trouver un ordonnancement optimal différent de celui qui est présenté dans la figure 3.2, ce que nous allons prouver dans le lemme suivant.

Lemme
Pour un représentant de la famille CC_2, si l'ensemble des sous-arbres Sai peut être scindé en deux sous-ensembles tels que la somme des tâches soit égale de part et d'autre, alors, il n'existe pas d'autres ordonnancements optimaux pour ce représentant que ceux qui allouent à un processeur un des deux sous-ensembles de sous-arbres (ce qui constitue un travail), puis le sous-arbre Sa', et au second processeur l'autre sous-ensemble de sous-arbre, suivi de la chaîne Ch. Enfin, un dernier travail constitué de la seule racine.

Preuve
Par convention, on appelera P_1 le processeur qui exécute la racine et la chaîne Ch (ou la fin de cette chaîne). On remarque dans un premier temps qu'il n'est pas possible qu'une des tâches de la chaîne Ch soit exécutée dans le premier travail, sinon, les deux sous-arbres $Sa1$ et $Sa2$ qui précèdent la première tâche de Ch doivent également faire partie de ce travail, ce qui donne pour la date de fin d'exécution une valeur supérieure ou égale à $\left\lfloor \frac{N_{total}-1}{2} \right\rfloor + N_{Ch} + 5$ valeur supérieure strictement à l'optimale.
Si la taille du second travail contient plus de tâches que la chaîne Ch, alors, toutes les tâches appartenant aux sous-arbres Sai alloués à P_2 n'ont pu être toutes exécutées durant le premier travail, donc, il reste certaines de ces tâches dans le second travail exécuté par P_2. Or ces tâches précèdent (directement ou non) des tâches appartenant à Ch, ce qui entraine la création de nouveaux travaux, et rend le nouvel ordonnancement non optimal.
Enfin, si la taille des travaux reste la même, il n'est pas possible qu'un des sous-arbre ait été scindé en plusieurs morçeaux pour la même raison que celle indiquée dans la preuve du lemme précédent.
Ainsi, s'il existe une répartition des sous-arbres en deux sous-ensembles pour lesquels le nombre de tâches de part et d'autre est égal, alors, il n'existe pas d'autres ordonnancements optimaux que ceux qui allouent à un processeur un des deux sous-ensemble de sous-arbres suivi du sous-arbre Sa'. □

3.3 Ordonnancement d'arbres complets sur m processeurs (fixé)

3.3.1 Introduction

Puisqu'il est impossible de construire un algorithme de compléxité polynomial produisant un ordonnancement optimal pour les arbres binaires sur seulement deux processeurs, nous considérons un problème plus simple : l'ordonnancement d'arbres k-aires complets sur m processeurs (m fixé). Nous présentons un algorithme qui produit des ordonnancements optimaux pour ce type d'arbres. L'idée de cet algorithme est d'occuper autant de processeurs que possible tant que le nombre de tâches disponibles est inférieur ou égal au nombre de processeurs, puis lorsque ce nombre de tâches devient supérieur au nombre de processeurs, de répartir la charge totale restante entre tous les processeurs de la manière la plus équitable possible en créant le plus petit nombre de *travaux* possible.

3.3.2 Algorithme

L'algorithme que nous présentons dans ce paragraphe est partagé en deux parties. La première produit un ordonnancement pour les premiers niveaux de l'arbre, et la seconde partie produit un ordonnancement pour les plus hauts niveaux. Le niveau "pivot" est le plus petit niveau contenant un nombre de tâches supérieur au nombre de processeurs. On appelle ce niveau l_{inf}.
La première partie de l'algorithme consiste donc à créer un travail par tâche tant que le niveau traité est inférieur strictement à l_{inf}. Pour la seconde partie, on note k l'arité de l'arbre et h sa hauteur. On note $N_2 = k^{l_{inf}} \times \frac{k^{h-l_{inf}+1}-1}{k-1}$, le nombre de noeud de l'arbre qui, à la fin de la première partie de l'algorithme, n'appartiennent à aucun travail.

Si (N_2 mod $m = 0$) **Alors**
  Si ((N_2 div m) mod $k^{l_{inf}} = 0$) **Alors**
    allouer $\grave{a}$ chaque processeur ((N_2 div m) div $k^{l_{inf}}$) arbres complets
  Sinon
    allouer $\grave{a}$ $k^{l_{inf}}$ mod m processeurs ($k^{l_{inf}}$ div m) + 1
    sous-arbres complets de hauteur $h - l_{inf} - 1$
    allouer aux autres processeurs ($k^{l_{inf}}$ div m)
    sous-arbres complets avec la même hauteur et ajouter des feuilles
    de sorte que tous les travaux aient la mmême longueur
    allouer enfin un nombre égal de feuilles rassembler en un
    travail $\grave{a}$ chaque processeur.
  FinSi
Sinon
  Si ((N_2 div m) + 1) mod $k^{l_{inf}} = 0$ **Alors**
    allouer $\grave{a}$ N_2 mod m processeurs ($k^{l_{inf}}$ div m) + 1
    sous-arbres complets.
    allouer deux travaux $\grave{a}$ chacun des processeurs restants.
  Sinon
    allouer $\grave{a}$ to N_2 mod m processeurs ($k^{l_{inf}}$ div m) + 1
    sous-arbres complets de hauteur $h - l_{inf} - 1$
    allouer aux autres processeurs ($k^{l_{inf}}$ div m)
    sous-arbres complets avec la même hauteur et ajouter des feuilles
    de sorte que tous les travaux aient la mmême longueur
    allouer enfin un nombre égal de feuilles rassembler en un
    travail $\grave{a}$ chaque processeur.
  FinSi
FinSi

3.3.3 Optimalité

Pour prouver l'optimalité des ordonnancements produits par cet algorithme, nous commençons par prouver le lemme suivant qui correspond à la première partie de l'algorithme.

Lemme
L'ordonnancement optimal d'un arbre k-aire complet sur un nombre fixé de processeurs plus grand ou égal au nombre de feuilles de l'arbre est obtenu en considérant chaque tâche comme un travail.

Preuve
Sans perte de généralités, il est possible de considérer un nombre de processeurs égal au nombre de feuilles : $m = k^{h-1}$. D'après la définition du temps parallèle : $T_{par} = \frac{T_{seq}+surcoût}{m}$, la recherche du minimum des maximum des dates de fin d'exécution des travaux est équivalent à la recherche du minimum des surcoûts. Nous calculons le surcoût pour le cas où chaque tâche constitue un travail. L'ordonnancement obtenu est de la forme "marches d'escaliers" (cf. figure 3.3).

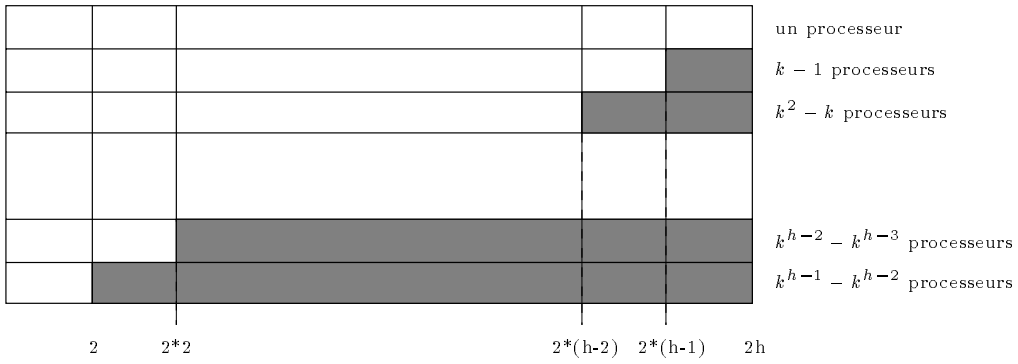


FIG. *3.3 - : forme d'un ordonnancement.*

Considérons un regroupement des tâches qui permette de minimiser le temps parallèle. Nous examinons plus précisement le travail qui contient la racine de l'arbre.
Puisque ce dernier travail contient la racine de l'arbre il est exécuté en séquence avec les autres travaux, c'est-à-dire que lors de son exécution tous les autres processeurs sont inactifs. On peut donc minorer le temps parallèle :

$$T_{par} \geq (i+1) + \left(\left\lceil \frac{\frac{k^h-1}{k-1} - i}{k^{h-1}} \right\rceil + 1 \right)$$

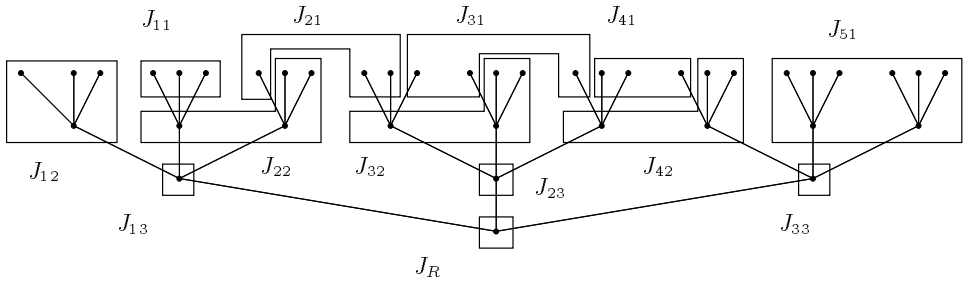
où le premier terme est égal à la taille (i) du dernier travail exécuté plus une unité de temps pour la communication. Le second terme minore le temps d'exécution des travaux qui ont précédés le dernier. On considère que la totalité des tâches appartenant à l'arbre mais n'appartenant pas au dernier travail sont réparties de façon équilibrée entre les processeurs. On remarque alors que le temps parallèle est minimum pour la plus petite valeur possible de la taille du travail final, c'est-à-dire pour $i = 1$.
Ainsi, dans un regroupement permettant d'atteindre le temps parallèle minimum, le dernier travail ne doit pas contenir plus d'une tâche. Si l'on considère maintenant le problème de l'ordonnancement des k sous-arbres complets, dont les racines sont de niveau 2, sur $\frac{m}{k}$ processeurs, le résultat précédent est encore valable. La même propriété est vérifiée à chaque

niveau, depuis le niveau 1 (qui ne contient que la racine de l'arbre) jusqu'au niveau h (niveau des feuilles de l'arbre). Donc, un ordonnancement optimal est obtenu par la création d'un travail par tâche. □

Théorème
Pour un arbre k-aire complet, l'algorithme décrit produit des ordonnancements optimaux

Preuve
L'arbre complet peut être partitionné en deux régions distinctes, dont la partie basse est ordonnancée optimalement par le premier algorithme sur un nombre fixé de processeurs, supérieur au nombre de feuilles de l'arbre. Le problème se ramène donc à la détermination d'un ordonnancement optimal pour une forêt d'arbres k-aires complets de même hauteur. Le nombre d'arbres appartenant à cette forêt est noté α (où $\alpha \geq m$). Pour minimiser la date de fin d'exécution, l'algorithme doit produire des ordonnancements avec une valeur minimum pour le surcoût, somme des périodes d'inactivité et du nombre de travaux. Selon le nombre de tâches de la seconde région, elles peuvent ou non être réparties en un nombre égal de tâches par processeur et dans le meilleur des cas en un nombre égal de sous-arbres. Au total, quatre différentes situations peuvent survenir. Comme pour la description de l'algorithme, on note N_2 le nombre total de tâches, $k^{l_{inf}}$ représente le nombre d'arbres k-aires complets de hauteur $h - l_{inf}$.
Le premier cas correspond à la possibilités de répartir un même nombre de sous-arbres complets par processeur. Dans ce cas, $k^{l_{inf}}$ est divisible par m. La meilleur valeur du surcoût que l'on puisse obtenir est alors m. En effet, la meilleure répartition consiste en un travail sur chaque processeur.
Dans tous les autres cas, pour un arbre complet donné, supposons que l'on construise un ordonnancement avec deux travaux sur chaque processeur. Supposons également que le premier travail exécuté par chaque processeur ne soit formé que de feuilles. Nous allons examiner dans quels cas cet ordonnnacement peut ne pas être optimal, et de quelle façon ce type de situation est gérée par l'algorithme.
Le surcoût est égal à $2m + N_{inact}$. La différence de charge entre les processeurs se limite à une tâche, ce qui borne la valeur de N_{inact} à $m - 1$. Lorsque la charge peut être équilibrée entre les processeurs, ce qui est le cas pour $N_2 \bmod m = 0$, le surcoût est égal à $2m$. Un tel ordonnancement n'est pas optimal si et seulement si il en existe un autre avec un nombre de travaux plus faible. Or, une réduction du nombre de travaux passe par une séquentialisation de certains travaux, soit parce que l'équilibrage de la charge n'est plus réalisé, soit parce que l'augmentation de volume de certains travaux entraîne de nouvelles contraintes de précédence. La disparition d'un travail limité à une tâche entraine deux périodes d'inactivité sur le processeur si la tâche ne lui est pas réallouée, et une période d'inactivité si elle est ajoutée au travail suivant. Dans tous les cas, le nombre de périodes d'inactivité qui en résulte est supérieur ou égal au nombre de travaux disparus.
Lorsque le nombre de tâches ne peut pas être équilibré entre les processeurs, $N_2 \bmod m \neq 0$, le surcoût est égal à $2m + N_{inact}$ et $N_{inact} \neq 0$. Ces périodes d'inactivité proviennent des processeurs dont le premier travail contient une tâche de moins que les autres. La seule manière de réduire le surcoût consiste à réduire le temps d'exécution sur les processeurs ayant une

tâche de plus que les autres. Aucun transfert de charge ne permet d'atteindre ce but, mais la réduction du nombre de travaux peut de temps en temps le permettre. Pour de tels cas, un ordonnancement composé de deux travaux sur tous les processeurs n'est pas optimal. Pour que la réduction du nombre de travaux permette de réaliser cet objectif, il ne faut pas qu'elle entraîne de contrainte de précédence supplémentaire entre les travaux. Ce qui est possible seulement si le nombre de feuilles allouées dans le premier travail à ces processeurs est égal à k fois le nombre de tâches de niveau $h-1$ allouées à ces mêmes processeurs et appartenant au second travail. Ce cas est également traité dans l'algorithme et l'ordonnancement résultant de ce traitement est illustré en figure 3.4. Finallement, dans tous les autres cas, un ordonnancement avec deux travaux sur tous les processeurs permet d'atteindre une date de fin d'exécution optimale. □



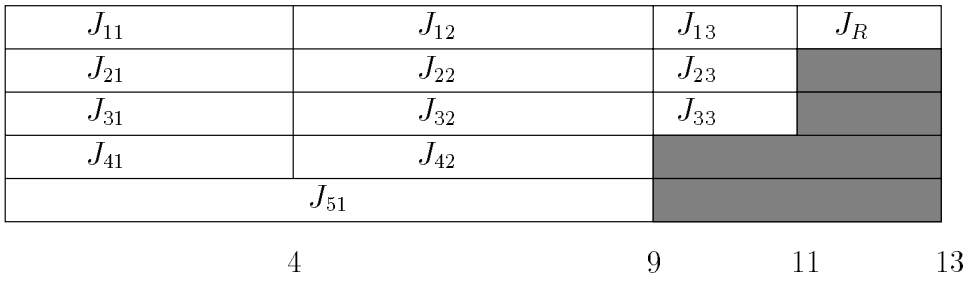


FIG. *3.4 - : Ordonnancement d'un arbre d'arité 3 et de hauteur 4 sur 5 processeurs.*

Exemples

Nous présentons trois exemples pour illustrer la description de l'algorithme.

Dans le premier, le nombre de processeurs est égal à 3, l'arité de l'arbre est égale à 4 et la hauteur égale à 4. Seule la racine est traité par ALGO_1, en effet, le nombre de noeuds du niveau 2 est supérieur à 3. On effectue le calcul d'allocation de sous-arbres complets, ce qui donne 1 sous-arbre complet pour $3-4 \ mod \ 3 = 2$ processeurs. Le dernier processeur reçoit toutes les tâches non encore allouées des niveaux inférieurs à 4, plus quelques feuilles jusqu'à concurrence du poids d'un sous-arbre de hauteur 3 et d'arité 4. Enfin, les tâches sont

réparties équitablement entre les trois processeurs.

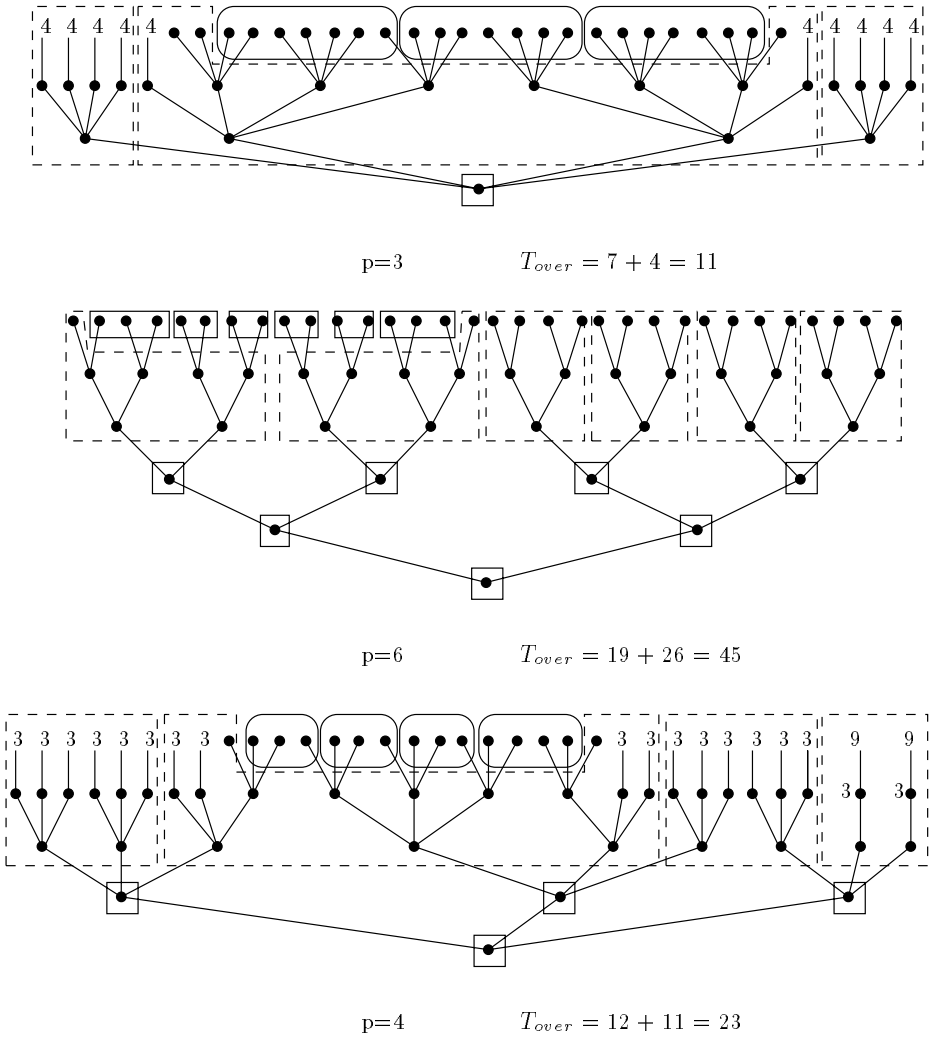


FIG. *3.5 - : Quelques exemples d'application de l'algorithme.*

3.4 Heuristiques

Comme nous l'avons vu précédemment, le problème de la détermination d'un ordonnancement optimal sur deux processeurs pour des arbres binaires est NP-difficile, cette constatation nous pousse donc à étudier des heuristiques. Une heuristique pour ordonnancer des arbres binaires sur deux processeurs est présentée, suivie d'une généralisation aux arbres d'arité quelconque sur un nombre quelconque (mais fixé) de processeurs.

3.4.1 P_2 | *arbres binaires* | *surcoût*

Nous proposons un algorithme pour ordonnancer des arbres binaires sur deux machines. Cet algorithme effectue un ordonnancement en deux temps. Le surcoût est le résultat de la

somme de deux termes : le temps d'inactivité et le temps de communications qui est aussi le nombre de travaux. L'idée est de mettre en oeuvre un algorithme qui produirait dans un premier temps un ordonnancement minimisant l'un des deux termes de la somme. Ensuite, à partir de cette donnée initiale, une mécanique est mise en place pour faire évoluer de manière itérative cet ordonnancement afin d'améliorer la valeur du surcoût. Le terme qui est minimiser est le nombre de périodes d'inactivité. Cette minimisation entraîne la création d'un grand nombre de travaux que la deuxième partie de l'algorithme tente de faire diminuer pour diminuer le surcoût.
Cependant, avant de décrire cet algorithme, nous allons caractériser des ensembles d'ordonnancements dominants pour les arbres. Si l'on note par P_1 le processeur qui exécute le travail contenant la racine de l'arbre.

Résultat préliminaire

Théorème :
L'ensemble des ordonnancements d'arbres tels que tous les travaux sont synchronisés est un ensemble ***dominant****.*

Preuve

Soit un ordonnancement optimal sur deux processeurs qui comporte N_{inact} périodes d'inactivité.
On appelle *Async* l'évènement correspondant au fait que deux travaux commencés en même temps ne finissent pas au même moment, et *Inact* l'évènement qui correspond à la présence d'une période d'inactivité entre l'exécution de deux travaux sur un même processeur.
On se place à la date t telle que pour la première fois depuis $t = 0$ l'un des deux évènements se produise. Supposons que ce soit *Async*, c'est-à-dire que deux travaux commencés au même instant se termine à des dates différentes. Supposons que J_1 se termine à la date t, tandis que J_2 se termine à la date $t' > t$. Si entre t et t' il apparait des périodes d'inactivité et qu'un nouveau travail commence avant t', les tâches de celui-ci sont indépendantes des tâches de J_2, il est donc possible de faire commencer ce travail immédiatement après J_1. Si pendant l'intervalle de temps [t,t'] le processeur exécutant J_1 est inactif, on traite cet évènement comme un événement *Inact* (événement traité ci-dessous). S'il n'existe aucune période d'inactivité pendant l'intervalle de temps [t,t']. La transformation pour synchroniser J_1 et J_2 consiste à rajouter des tâches appartenant au travail succédant immédiatement à J_1 sur le même processeur (J_3), en effet, toutes les tâches de J_3 sont indépendantes des tâches de J_2, il est possible de les exécuter en même temps que les tâches de J_2. Le résultat est illustré sur la figure 3.6.

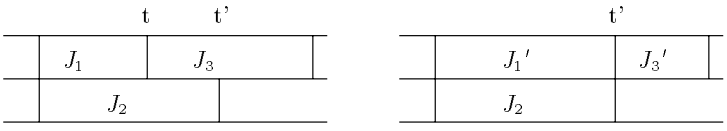


FIG. *3.6 - : Synchronisation de deux travaux.*

Dans le cas d'un événement *Inact*, plusieurs cas peuvent se produire. Si la ou les périodes d'inactivité se situent juste après une synchronisation entre deux travaux exécutés sur les deux processeurs, on déplace, s'il existe, le travail succédant à ces périodes d'inactivité, de façon à avancer l'exécution de ses tâches (voir figure 3.7 Cas (a)).

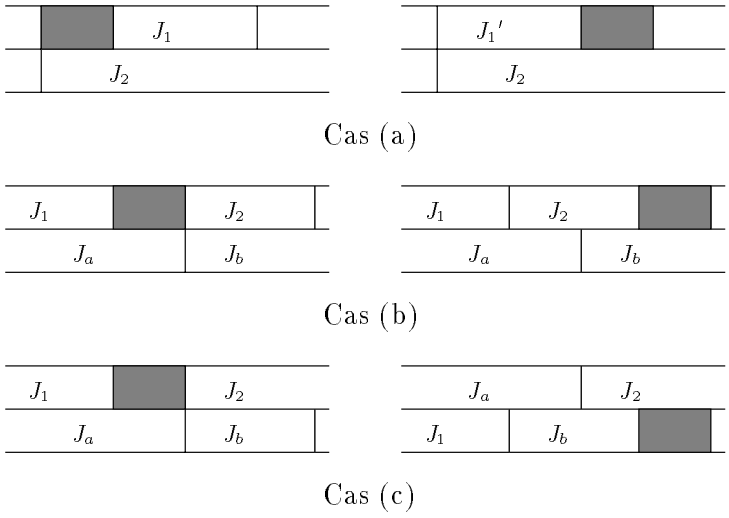


FIG. *3.7 - : Transformations concernant les périodes d'inactivité.*

Si les périodes d'inactivité se situent juste avant un début d'exécution synchrone de deux travaux, plusieurs cas peuvent se produire :

- Le travail J_a précède seulement J_b, on avance l'exécution de J_2 (voir figure 3.7 Cas (b)).
- Le travail J_a précède seulement J_2 (voir figure 3.7 Cas (c)), on inverse les positions de J_1 et J_a, et on avance l'exécution de J_b.
- Le travail de J_a précède à la fois J_b et J_2. Une partie des tâches de J_a, J_{ab}, précède J_b, et une partie, J_{a2}, précède J_2.
  Si $J_{a2} \leq J_1$, N_{inact} tâches appartenant à J_{ab} (J_{ab1}) sont intégrées au travail J_b (voir figure 3.8 Cas (a)). Si, au contraire, $J_{a2} > J_1$, on intervertie J_1 et J_{a2}, et on se retrouve dans la configuration précédente (voir figure 3.8 Cas (b)).

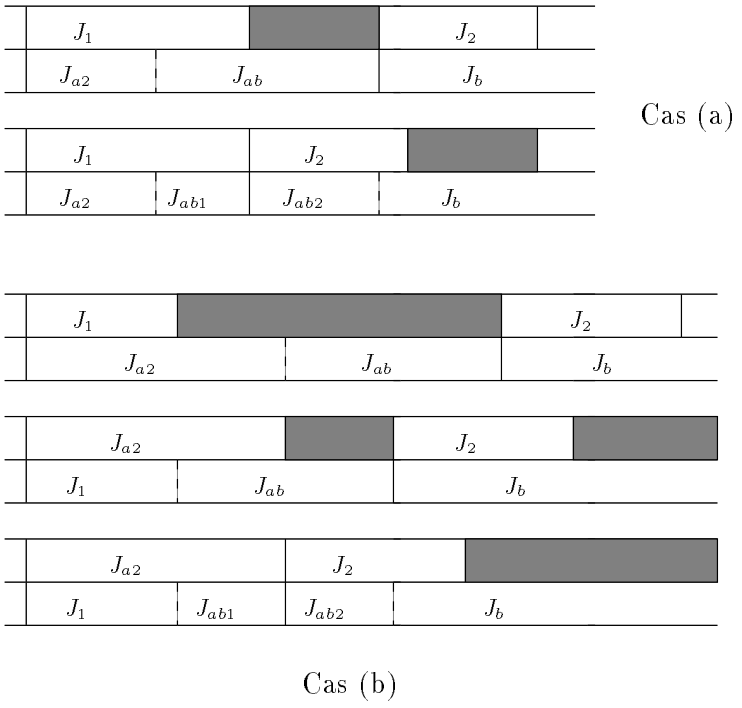


FIG. *3.8 - : Transformations concernant les périodes d'inactivité.*

Ainsi, quelle que soit la situation la structure d'un ordonnancement optimal, il est possible d'éliminer toutes les périodes d'inactivité qui se situent entre l'exécution de deux travaux et de les rassembler à la fin de l'ordonnancement du processeur qui n'exécute pas la racine de l'arbre. □

Heuristique pour $P_2 \mid arbres\ binaires, p = 1 \mid surcoût$

Principe et propriétés

Cette heuristique procède en deux temps. La première étape est une initialisation qui détermine un ordonnancement avec le nombre minimum de périodes d'inactivité. Cela revient à privilégier l'équilibrage de la charge au détriment des communications. Rapporté à la définition du surcoût, il s'agit donc de minimiser le terme T_{inact}, sans se préoccuper du terme T_{com} égal au nombre de travaux. La seconde étape vise à diminuer le nombre de travaux dans le but de diminuer la valeur du surcoût.
La première partie est assurée par un algorithme qui produit des ordonnancements optimaux pour des arbres *UECT* dans le modèle RS[1]. La seconde partie consiste en un regroupement itératif des travaux de sorte que leur volume augmente, diminuant ainsi leur nombre.

Première partie

[1]La description complète de cet algorithme se trouve dans la troisième partie, chapitre 1, sous le nom d'algorithme MAJYC

Le principe de cet algorithme est le suivant. Le premier travail consiste à étiqueter chaque noeud par le nombre de prédécesseurs du sous-arbre dont il est racine (le noeud compris). On appelle cet attribut le **poids** du sous-arbre ou du noeud. Le poids de la racine est égal au nombre total de noeuds de l'arbre N. Ensuite, la charge théorique de l'un des processeurs dans le meilleur des cas est calculée. Ce meilleur des cas correspond à une équirépartition des tâches (la racine mise à part) entre les processeurs, comme si toutes les tâches étaient indépendantes. Cette charge est égale à $(N-1)$ *div* (2). $N-1$ signifie que toutes les tâches sont prises en compte pour cette répartition exceptée la racine durant l'exécution de laquelle aucune autre tâche ne peut être exécutée. On suppose dans toute la suite que la charge est calculée pour le processeur P_2. Par conséquent, la racine est exécutée sur P_1. L'algorithme examine alors les deux prédécesseurs immédiats de la racine (on suppose que l'arité de la racine est égale à deux) et choisi le sous-arbre de plus faible poids. Le sous-arbre est marqué et la valeur de la charge est mise à jour. Il reste un seul noeud non marqué à ce niveau. Si son arité est égale à un, l'algorithme recherche le premier noeud non marqué d'arité deux. En ce premier noeud d'arité deux, une seconde valeur de la charge théorique est calculée. Si cette nouvelle valeur est inférieure à la valeur courante de la charge, on la substitue à cette dernière. Sinon on l'oublie. Si parmi les deux noeuds du niveau courant l'un est de poids inférieur ou égal à la valeur courante de la charge théorique, il est choisi et marqué. Sinon, alors les deux noeuds ont un poids supérieur strictement à la valeur courante de la charge théorique. L'un d'eux est choisi et un nombre de noeuds égal à la valeur de la charge théorique est choisi dans ce sous-arbre. Le processus se termine lorsque la valeur de la charge théorique mise à jour est nulle.

Lemme 1
L'algorithme de choix des sous-arbres permet, sans regroupement, la minimisation du nombre de périodes d'inactivité.

Preuve
Cette propriété est directement issue de la preuve d'optimalité de l'algorithme MAJYC[2]. L'idée de base est que le nombre de périodes d'inactivité est minimum lorsque l'équilibrage de la charge est réalisé. La propriété de cet algorithme est que les dépendances de données ne sont que dans un sens, de P_2 vers P_1, en effet, lorsqu'une tâche est allouée à P_2, l'ensemble de ses prédécesseurs le sont aussi. Ce qui implique que l'exécution des tâches allouées à P_2 peut se faire sans aucune attente. D'autre part, le calcul de la charge pendant l'étape de choix nous garantie que l'allocation d'une tâche supplémentaire à P_2 entrainerait une période d'inactivité sur P_1. Enfin, le calcul et la mise à jour du nombre de tâches allouées à P_2 garantissent la constante activité de P_1. L'ensemble de ces propriétés permet d'affirmer que le choix des sous-arbres assure la minimisation du nombre de périodes d'inactivité. □

Seconde partie

L'étape de recherche est terminée, il faut construire les travaux. Le regroupement des tâches en travaux est effectué itérativement. On effectue un regroupement glouton, c'est-à-

[2]Partie 2 Chapitre 1

dire que l'on commence par les tâches appartenant aux sous-arbres ayant les racines les plus hautes dans l'arbre. Au départ, P_1 dispose d'un certain nombre de tâches qui sont libres d'être exécutées, c'est-à-dire sans contrainte de précédence avec des tâches choisies pour P_2. On note le nombre de tâches de ce travail exécuté sur P_1 Pds_0. De la même façon, on regroupe Pds_0 tâches pour constituer le premier travail de P_2. Les tâches ainsi regroupées dans ce premier travail exécuté sur P_2 libère un certain nombre de tâches sur P_1, tâches qui constituent le second travail exécuté sur P_1. Le processus est répété jusqu'à épuisement des tâches allouées aux deux processeurs. L'exécution des $Pds0$ tâches du premier travail exécuté par P_2 libère obligatoirement des tâches allouées à P_1 sinon, cela signifierait qu'en un noeud non marqué, la charge théorique retenue aurait été supérieure à la valeur calculée en ce noeud. Le résultat de ce premier regroupement est illustré par la figure 3.10. On note que ce regroupement n'augmente pas le nombre de périodes d'inactivité.

Lemme 2
Pour l'ensemble des tâches choisies, le regroupement guidé par les tâches disponibles pour être exécutées sur P_1 est celui qui donne la valeur minimale du surcoût.

Preuve
Le premier travail exécuté par P_1 ne peut pas être de taille plus importante sans changer l'ensemble des tâches allouées à P_2, en effet, ce travail regroupe l'ensemble des tâches allouées à P_1 qui n'ont aucun prédécesseur appartenant à P_2. Cet ensemble est unique pour une allocation donnée. Le premier travail exécuté par P_1 fixe le nombre de tâches du premier travail exécuté par P_2, c'est-à-dire qu'il fixe indirectement le nombre de tâches de P_1 pour lesquelles les prédécesseurs appartenant à P_2 ont terminé leur exécution. A partir du troisième travail exécuté par P_1, l'ensemble des tâches allouées à P_1 appartiennent à une chaîne. De plus, le second travail est constitué de l'ensemble des prédécesseurs de cette chaîne non encore exécutés. Ceci est une conséquence de l'algorithme de choix des sous-arbres pour P_2 qui "coupe" les sous-arbres le long d'une chaîne dont les noeuds ont des poids supérieurs à la valeur courante de la charge qui doit être allouée à P_2. Donc, si le premier travail exécuté par P_2 ne contient pas l'ensemble des prédécesseurs du second travail exécuté par P_1, le second travail de P_1 sera plus petit qu'il ne l'est dans le choix présent. Il en est de même pour la suite. Les sous-arbres allouées à P_2 sont directement liés à des noeuds appartenant à la chaîne exécutée par P_1, les meilleurs choix consistent à exécuter en priorité les tâches appartenant à des sous-arbres dont les racines sont les plus hautes dans l'arbre. □

Ce regroupement permet de minimiser le nombre de périodes d'inactivité sur P_2, mais ne permet pas de minimiser le surcoût. Il existe plusieurs stratégies pour améliorer l'ordonnancement obtenu. Remettre en cause le choix des tâches, mais conserver la même proportion en nombre de tâches, ou conserver les mêmes tâches, mais remettre en cause le regroupement. Nous retenons cette dernière solution car elle semble plus simple à mettre en oeuvre, et surtout de complexité plus faible que la remise en cause du choix des tâches. D'autre part, le choix des tâches effectué par l'algorithme de la première partie permet avec d'autres hypothèses d'exécution d'obtenir un ordonnancement optimal, ce qui autorise à penser que ce choix n'est pas mauvais. La solution envisagée est de diminuer le nombre de tâches allouées

à P_2, de manière à augmenter la taille d'un ou plusieurs travaux de P_1 et donc indirectement celle des travaux de P_2.
On reduit de un le nombre de tâches allouées à P_2. La tâche choisie appartient au sous-arbre dont la racine est de plus haut niveau (parmi l'ensemble des racines des sous-arbres alloués à P_2). Libérer cette tâche garantie l'augmentation de la taille du premier ou du second travail exécuté par P_1.
A chaque étape une nouvelle valeur pour le surcoût est obtenue. On arrête l'itération lorsque le nombre de travaux sur P_2 est égal à 1 ou lorsque le nombre de périodes d'inactivité est supérieur ou égal à la valeur minimale du surcoût obtenue jusqu'à présent. L'itération de ce procédé répond à la recherche d'un compromis entre l'équilibrage de la charge, privilégié par l'ordonnancement initial, et le nombre de travaux, privilégié par le mécanisme d'augmentation de leur volume. On remarque que cette opération est décroissante (non strictement) pour le nombre de travaux, mais qu'elle fait invariablement augmenter le nombre de périodes d'inactivité de deux à chaque étape de l'itération. Pour un graphe donné, l'évolution entre les deux termes du surcoût suit une évolution du type de celle décrite en figure 3.9.

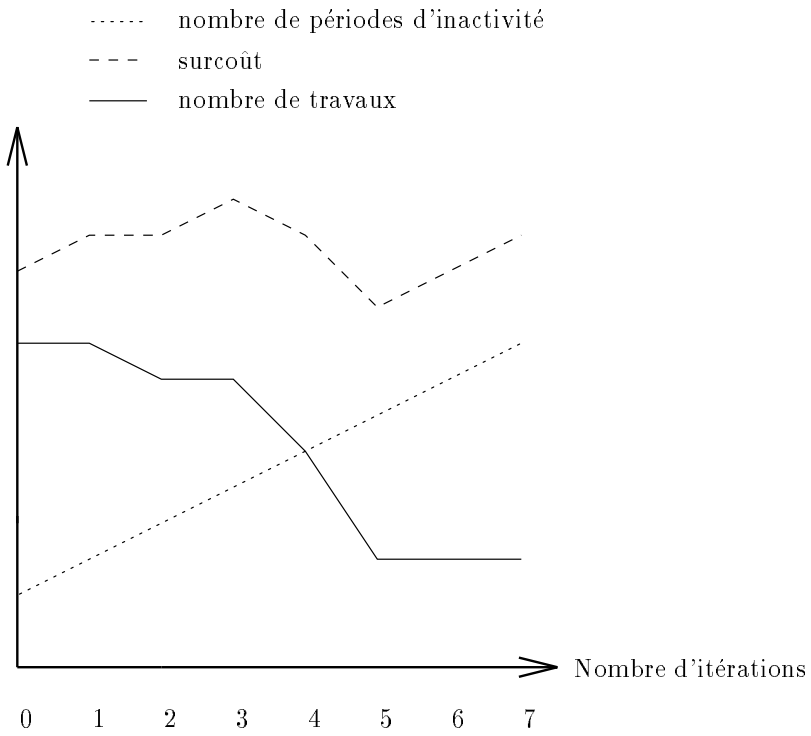


FIG. *3.9 - : Evolution des valeurs de l'inactivité, du nombre de travaux et du surcoût.*

D'après le Lemme 2, ceci signifit que si au départ il existe un regroupement optimal avec l'ensemble des tâches choisies, alors ce mécanisme permet de l'atteindre en un nombre d'itérations linéaire en le nombre de noeuds n. Chaque regroupement nécessitant un temps linéaire lui aussi. Par ailleurs, l'algorithme de choix qui constitue la première étape de cette heuristique est linéaire en le nombre de noeuds, ce qui signifit que cette heuristique est de complexité $O(n^2)$.

Description algorithmique de la réduction et exemples

Quelques notations utilisées dans l'algorithme :

- nous supposons que l'arbre est formé de n noeuds, et que la racine de l'arbre est d'arité 2 (si ce n'est pas le cas, on inclue la chaîne terminale dans le travail contenant le noeud d'arité 2 dont la hauteur est la plus faible),
- on note P_1 le processeur sur lequel est exécutée la racine, et P_2 l'autre processeur,
- on note n_2 le nombre total de noeuds alloués à P_2,
- $N_{travaux}$ le nombre de travaux exécutés sur P_2,
- N_{inact} le nombre de périodes d'inactivité,
- $min_{surcoût}$ représente la valeur minimum du surcoût déterminée jusqu'à présent.

Description algorithmique de la réduction :

TantQue ($N_{travaux} > 1$ et $N_{inact} < min_{surcoût}$ **Faire**
soit E l'ensemble des noeuds dont tous les predecesseurs
sont alloues a P_1. E constitue le premier travail
on cadence les executions des travaux sur P_2 en
fonction des travaux identifies pour P_1
calcul de $N_{travaux}$ et de N_{inact}
$V_{surcoût} = N_{travaux} + N_{inact}$
Si $V_{surcoût} < min_{surcoût}$ **Alors**
$min_{surcoût} = V_{surcoût}$
FinSi
$n_2 = n_2 - 1$
allocation de la racine de plus haut niveau
d'un sous-arbre alloue a P_2
FinTantQue

Exemples

Exemple 1

Dans cet exemple (figure 3.10), le nombre de tâches est égal à 15 (racine comprise). On commence donc par un calcul de la charge maximale : $\frac{15-1}{2} = 7$. Au niveau 2 de l'arbre les deux sous-arbres ont un poids respectifs de 11 et de 3. Le sous-arbre de poids égal à 3 est choisi et marqué. Le nombre de tâches restant à allouer à P_2 est mis à jour et devient égal à 4. Il n'y a maintenant au niveau 2 de l'arbre qu'une seule tâche non marquée (la tâche qui est racine du sous-arbre de poids égal à 11). On remonte le long de la chaîne jusqu'à la première tâche d'arité 2. Cette tâche a un poids égal à 10. Le recalcule de la charge donne 4,

la valeur courante de la charge est donc conservée. Au niveau 4, il y a deux sous-arbres de poids 6 et 3. On choisi l'arbre de poids 3, puis on remet à jour la charge (égale maintenant à 1), que l'on compare avec la valeur calculée au noeud de poids 6. La charge maximale en ce noeud est égale à 2, donc, on conserve 1 comme valeur de la charge. Au niveau 5, on dispose d'un sous-arbre de poids égal à 1, on le marque, et la charge devient nulle, ce qui met fin à l'étape de recherche des tâches.

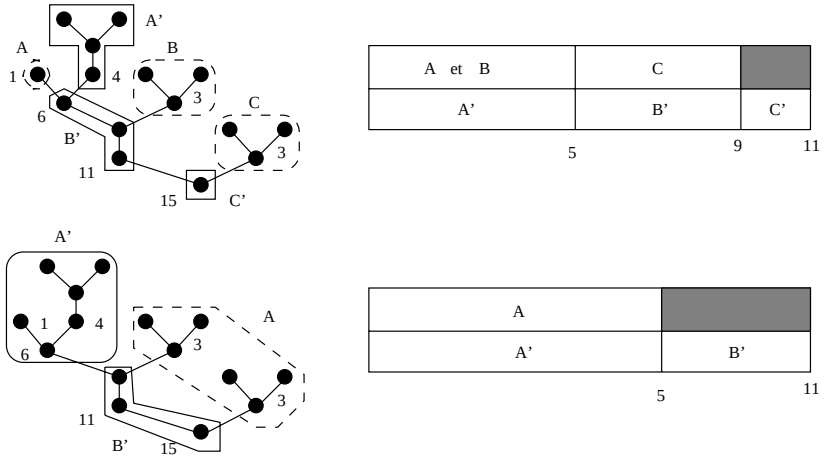


FIG. *3.10 - : Exemple d'ordonnancement fourni par l'algorithme aprés la première étape.*

La seconde étape est de les regrouper. Au départ, P_1 dispose de 4 tâches libres (par tâches libres, on entend des tâches dont tous les prédécesseurs sont aussi alloués à P_1), ce qui permet de regrouper les sous-arbres libellés A et B en un seul travail, puis de considérer le dernier sous-arbre comme un travail. Ce regroupement correspond à un surcoût égal à $2 + 5 = 7$ (périodes d'inactivité plus nombre de travaux). Cette valeur du surcoût est conservée. L'algorithme itère l'étape de regroupement après libération d'une tâche de P_2, en l'occurrence le sous-arbre A formé d'une seule tâche. Le regroupement construit avec ce nouvel ensemble de tâches permet de diminuer le nombre de travaux de deux, c'est-à-dire annule l'augmentation de l'inactivité. Donc, la valeur du surcoût obtenu n'est pas meilleure, et comme le nombre de travaux sur P_2 est égal à 1, l'itération est terminée.

Exemple 2

Dans ce second exemple (figure 3.11), la première étape est menée de la même façon que dans l'exemple précédent. Le premier regroupement correspond à la détermination de quatre

travaux pour P_2, et à un surcoût égal à 11.

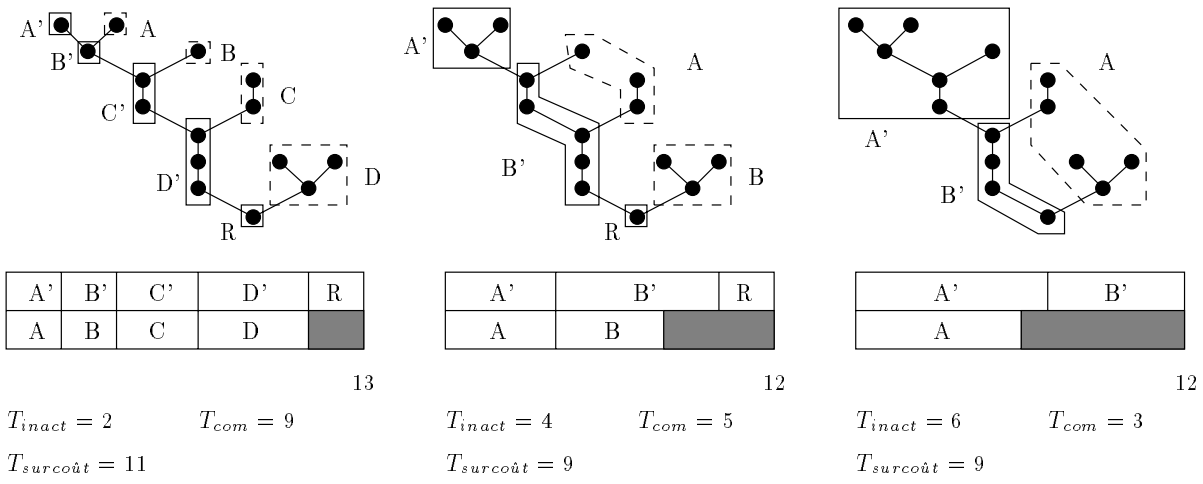


FIG. *3.11 - : Un exemple de regroupement itératif.*

La seule façon de faire évoluer ce surcoût est de diminuer le nombre de travaux, pour cela, on diminue le nombre de tâches à allouer à P_2, puis on recommence la détermination des différents travaux, ce qui permet d'obtenir pour P_2 deux travaux au lieu de quatre précédemment. On parvient donc à faire diminuer le surcoût de 11 à 9. D'autre part, ce nombre de travaux est supérieur à 1 et le nombre de périodes d'inactivité est inférieur strictement à la meilleure valeur trouvée pour le surcoût, donc, une itération supplémentaire est entamée. A ce stade, le nombre de travaux de P_2 est égal à un, il serait donc vain d'essayer de faire diminuer ce nombre de travaux. Une fois encore le surcoût est égal à 9, et ce chiffre correspond à la valeur optimale du surcoût.

Remarque

Pour les arbres à structure **chenille** (*caterpillar*), l'itération sur le regroupement est particulièrement efficace. Les représentants de cette famille sont tels que tous les noeuds sont d'arité égale à deux (exceptées les feuilles de l'arbre), et possèdent un sous-arbre formé d'une seule feuille. Pour un tel arbre, la recherche des tâches en vue d'un regroupement fournit l'ensemble des feuilles moins l'une de celles appartenant au niveau le plus élevé. La valeur du surcoût pour le premier regroupement est égal à $n + 2$, en effet, le nombre de périodes d'inactivité est minimum et égal à 2, et le nombre de travaux est égal au nombre de tâches de l'arbre. Pour le cas particulier des arbres à structure chenille, il est possible de donner une expression du nombre de travaux et du nombre de périodes d'inactivité en fonction de l'étape d'itération notée i (0 pour le regroupement de départ).

$$N_{travaux} = 2[(\frac{n-1}{2} - i)div(2i+1)] + 1 + 2\delta$$

$\delta = 0$ si $(\frac{n-1}{2} - i)mod(2i+1) = 0$ et 1 sinon. Quant au nombre de périodes d'inactivité, il progresse linéairement en fonction de i :

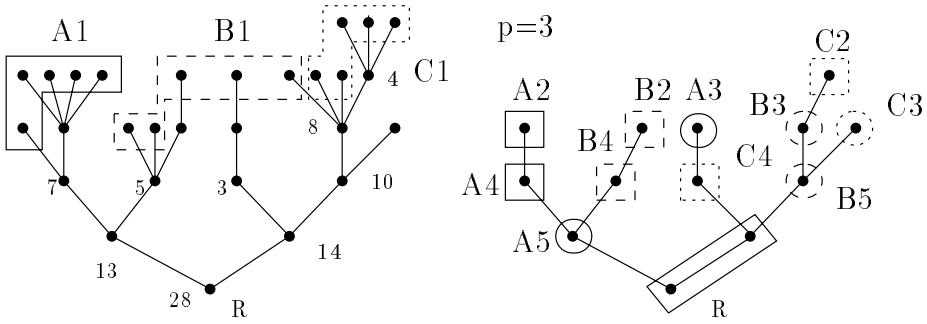
$$N_{inact} = 2(i+1)$$

Il est donc possible pour une valeur de n donnée de calculée au préalable le bon regroupement, en cherchant le i qui minimise la somme $N_{travaux} + N_{inact}$ qui est égale à $T_{com} + T_{inact} = surcoût$.

3.4.2 Ordonnancement d'arbres sur m processeurs (fixé)

La difficulté des problèmes d'ordonnancement d'arbres dans ce modèle nous incite à la mise au point d'heuristiques. Comme cela a été largement exposé dans les paragraphes précédents, l'objectif, la minimisation du surcoût, ne peut être réaliser que par la détermination d'un compromis entre l'équilibrage de la charge et la minimisation du nombre de travaux. Comme base pour une heuristique, il semble naturel de penser à des algorithmes niveau par niveau, du type *HLF*, ou à des algorithmes de regroupement. Cependant, les représentants de la première catégorie sont fortement pénalisés pour des arbres constitués par de longues chaînes et non équilibré. De même, la détermination d'un bon regroupement par les représentants de la seconde catégorie pose des problèmes de famine et de mise en oeuvre. La solution qui est retenue est basée sur les notions combinées de choix niveau par niveau et de regroupement. Il s'agit d'un algorithme qui est adaptable en fonction d'une famille d'arbres cibles. On appelle **profondeur** du regroupement, le nombre d'étapes de recherche de sous-arbres.

Soit N_f le nombre total de feuilles (quel que soit leur niveau dans l'arbre). On simule une équirépartition de N_f sur les m processeurs. A partir de ce nombre, on peut construire un premier ordonnancement, en formant des travaux de tailles égales à $\left\lfloor \frac{N_f}{m} \right\rfloor$, et recommencer le même procédé, en considérant comme ensemble de feuilles, les tâches dont tous les prédécesseurs ont été exécutés (voir figure 3.12). Cette construction correspond à une heuristique purement niveau par niveau. On dit que la profondeur de recherche des sous-arbres est égale à 1.



A1	A2	A3	A4	A5	R
B1	B2	B3	B4	B5	
C1	C2	C3	C4		

6 8 10 12 14 17

FIG. *3.12 - : Application de l'algorithme avec une profondeur 1.*

La seconde possibilité est de ne pas construire un tel ordonnancement, mais de chercher dans l'arbre tous les sous-arbres de tailles inférieures ou égales à $Taille_{max} = \left\lceil \frac{N_f}{m} \right\rceil$. On fait de nouveau la somme de toutes les tâches ainsi répertoriées S, et on divise ce total par m, $Charge_{moyenne} = \left\lfloor \frac{S}{m} \right\rfloor$. Là encore, deux possibilités. La première correspond à la construction d'un ordonnancement par création des travaux de taille $Charge_{moyenne}$. Le processus peut être itéré (voir figure 3.13). Il s'agit alors d'une recherche de profondeur 2.

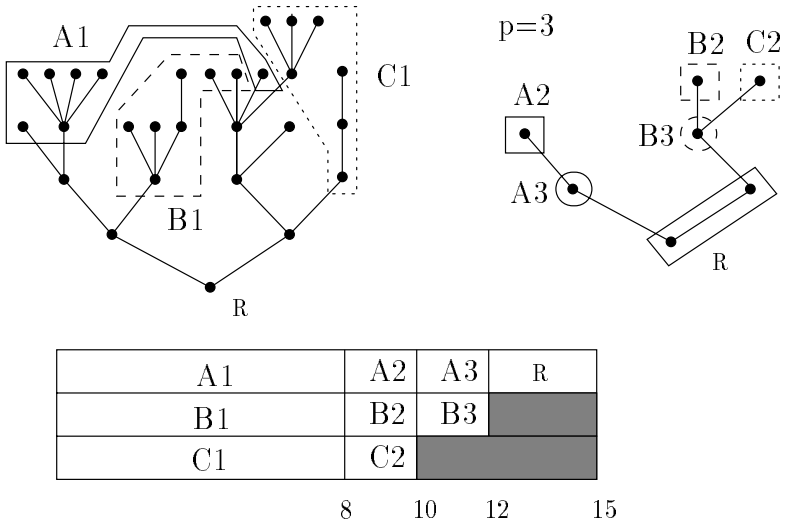


FIG. *3.13 - : Application de l'algorithme avec une profondeur 2.*

L'alternative étant de ne pas construire encore un ordonnancement, mais de repartir à la recherche de sous-arbres de poids inférieurs ou égaux à $Charge_{moyenne}$ (voir figure 3.14). On effectue ainsi une recherche de profondeur 3. Il est facile de constater qu'au fur et à mesure que la profondeur augmente, on se rapproche d'un regroupement à plus gros grain. Le procédé fini par convergé vers une profondeur limite inférieure au nombre de tâches de

l'arbre.

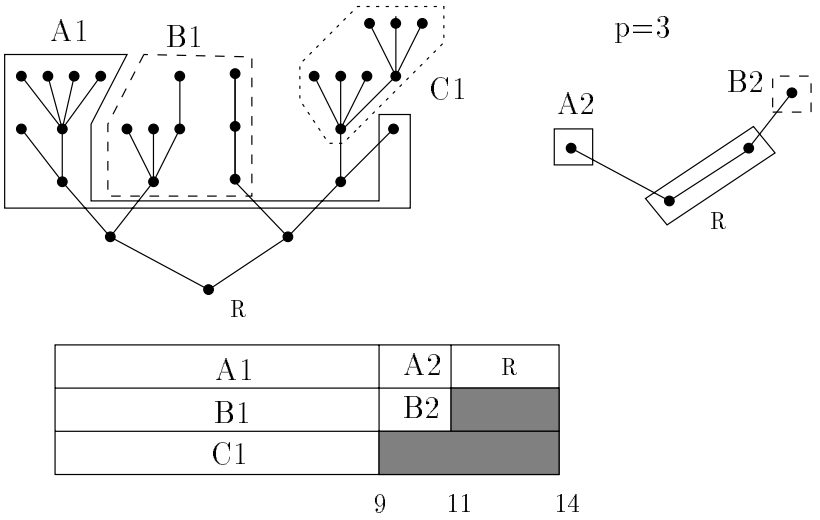


FIG. *3.14 - : Application de l'algorithme avec une profondeur 3.*

Dans l'exemple de la figure 3.12, le regroupement ne concerne que les feuilles, et d'une manière générale, que des tâches indépendantes pour une étape de regroupement donnée. On retrouve ici l'algorithme niveau par niveau.
Dans l'exemple de la figure 3.13, le nombre de feuilles est égal à 16. Donc $Taille_{max} = \left\lceil \frac{16}{3} \right\rceil = 6$. L'ensemble des poids des sous-arbres tels que leur poids respecte la limite $Taille_{max}$ est : $\{1, 5, 5, 3, 1, 1, 1, 4, 1\}$, $S = 22$. A partir de cet ensemble, on effectue un regroupement de façon à obtenir pour chaque travail un total de tâches le plus proche possible de la cahrge moyenne qui est égale à $Charge_{moyenne} = \left\lfloor \frac{22}{3} \right\rfloor = 7$. La stratégie utilisée revient à choisir les plus lourds sous-arbres puis on complète avec les feuilles, pour l'exemple, les travaux sont formés de $5 + 1 + 1$, $5 + 1 + 1$, et $4 + 3$.
Enfin, dans l'exemple de la figure 3.14, on recalcule une valeur pour la taille maximum des sous-arbres recherchés, $Taille_{max} = \left\lceil \frac{22}{3} \right\rceil = 8$. On recherche de tels sous-arbres : $\{7, 5, 3, 8, 1\}$. Les travaux sont les suivants : $C1 = 8$, $A1 = 7 + 1$, et $B1 = 5 + 3$. A la fin de cette première étape, il ne reste que deux feuilles, c'est-à-dire moins de tâches que de processeurs, on alloue donc un travail par processeur, et chaque travail n'est constitué que d'une tâche. Le résultat de cet ordonnancement aurait pu être obtenu par un algorithme de regroupement direct, bien que cela semble difficile.

3.5 Conclusion

Le modèle proposé par Anderson, Beame et Ruzzo est d'un grand intérêt pour les études d'ordonnancements. Par l'introduction de la notion de **surcoût**, ce modèle capture à la fois les problèmes liés aux communications et à l'équilibrage de la charge. La nécessaire recherche d'un **compromis** exprime toutes les difficultés inhérentes à la détermination d'un bon ordonnancement dans de nombreux modèles d'exécution. Dans ce cadre, nous avons prouvé

que le problème de l'ordonnancement des arbres binaires est NP-complet, et nous avons proposé une heuristique pour ce problème particulier, qui produit pour certaines classes d'arbres des ordonnancements optimaux. Le fait que ce problème soit résolu linéairement par divers algorithmes basées sur des stratégies niveau par niveau ou de regroupement dans le modèle RS renforce l'opinion que nous avons de la validité de ce modèle. Pour un nombre plus important de processeurs, nous avons présenté un algorithme qui produit des ordonnancements optimaux pour les arbres k-aires complets. A partir de ce résultat nouveau, nous avons dérivé une heuristique pour l'ordonnancement des arbres quelconques sur un nombre m fixé de processeurs. Cette heuristique peut être utilisée comme un algorithme itératif (du type des algorithmes de recuit simulé ou tabous), en fixant la profondeur de recherche des sous-arbres. Les différentes possibilités de recherches pouvant être représentées par un arbre. On peut noter également que dans la mesure du possible cet heuristique fonctionne de façon synchrone, et pourrait être utilisée pour certains modèles présentés précédemment comme le modèle LPRAM ou le modèle PU (dont la fonction objectif serait basée sur la valeur de T_{trafic}).

Chapitre 4

Autres modèles pour machines à mémoire distribuée

4.1 Introduction

Tous les modèles présentés jusqu'à présent sont des modèles généraux de machines. Les d'algorithmes et les stratégies d'ordonnancement mises en oeuvre dans ce cadre recherchent des possibilités d'applications pour un grand nombre de machines et ce au détriment de certaines caractéristiques architecturales. Cependant, il peut être intéressant pour une machine ou une classe de machines particulière, d'étudier des stratégies d'ordonnancement plus ciblées, prenant en compte davantage de contraintes. C'est dans ce sens que l'étude est menée dans ce chapitre. Parmi les premiers travaux sur le sujet, Chen et Lai ont proposés un algorithme pour équilibrer la charge d'un ensemble de tâches indépendantes ordonnancées sur un hypercube [CL88]. La méthode qu'ils emploient partitionne l'hypercube en sous-cubes dédiés à l'exécution d'un ensemble de tâches. Ce problème se ramène alors à un problème d'ordonnancement de tâches multiprocesseurs. Les travaux de Robertazzi et al. [CR90, BR93] sont plus proches de ce que nous présentons. Ils étudient les problèmes de calcul distribués avec délais de communications modélisés de façon linéaire. La topologie des réseaux d'interconnexion pris en compte est soit un arbre, soit une chaîne soit un ensemble de processeurs partageant un bus commun.

Le travail qui est présenté dans ce chapitre n'est pas achevé, mais trouve sa place dans une étude sur les modèles, et apporte une vue différente de ce qui a été précédemment présenté. Il fait suite à une série de travaux [CR90, BR93] et est basé sur une première version d'un résultat initié par Błażewicz et al.[1] [BD93]. La principale nouveauté par rapport aux résultats d'ordonnancements dans les réseaux point à point existants est l'utilisation de certaines stratégies mises en oeuvre pour l'optimisation des communications dans de tels réseaux.

Le modèle proposé est un modèle générique de machines, dont chaque représentant est défini par la donnée du nombre de processeurs (éventuellement non limité), de la topologie

[1]Ce travail est mené avec l'aide de Christophe Calvin et Denis Trystram du LMC-IMAG, en coopération avec Jacek Błażewicz et Maciej Drozdowski de *l'Université Technique de Poznan* (Pologne).

du réseau d'interconnexion, de l'aspect des noeuds de la machine (noeud simple, ou composé d'un processeur de calcul et d'un processeur dédié aux communications), du nombre de ports de l'élément assurant la gestion des communications. Après une brève description des différents modes de commutations de message, un exemple d'algorithme d'ordonnancement est proposé pour une tâche arbitrairement divisible sur une grille-2D de 5^{2k} processeurs.

4.2 Description de différents modes de commutations de messages

Un mode de commutation est un protocole physique pour le routage des messages. Une description plus détaillée des modes de commutation que nous présentons dans la suite est disponible dans [NM93, Mic94] Parmi les divers modes de commutations, on distingue la **commutation de messages** (*Store-and-Forward*), la **commutation de circuits** (*Circuit-Switching*) et la **commutation de paquets** (*Packet-Switching*). Dans chacun de ces modes, la distance représente le nombre de liens traversés par le message depuis l'émetteur jusqu'au recepteur. Pour tous, le modèle de temps est supposé linéaire. Le temps de communications entre deux voisins est égal à $T_{com} = \alpha + L\tau$, où α correspond à **l'initialisation** du message (*start-up*) (empaquetage, décision de routage), et τ représente le **taux de transmission** (inverse de la bande passante).

- **Le mode commutation de messages**.
  Lorsqu'un processeur envoi un message de longueur L à un autre processeur localisé à une distance d, le message, dans son intégralité, est envoyé au plus proche processeur sur le chemin et est stocké dans la mémoire de ce processeur. Le processus est répété par tous les processeur appartenant au chemin jusqu'au processeur auquel est destiné le message. Pour un tel mode de commutation, la distance entre processeur est très importante puisque le coût d'une telle communication est égal à :

  $$T_{com} = d(\alpha + L\tau)$$

- **Le mode de commutation de circuits**.
  Avec ce mode de commutation, un envoi de message est précédé par la réservation de tous les liens par lesquels le message doit transiter. Cette réservation est effectuée par une **entête** de message *header*. Le reste du message est envoyé en une fois. Le message n'est pas stocké dans la mémoire d'un processeur intermédiaire, et le coût de la communication ne dépend que très peu du terme dépendant de la distance. Le délai de communication est modélisé par :

  $$T_{com} = \alpha + d\delta + L\tau$$

  Où δ représente le temps de commutation d'un commutateur (*switch*). Généralement, $\delta \ll \alpha$ (le rapport est proche de 0,1% ou 1%). Ce paramètre δ n'est que rarement donné par les constructeurs de machines, et semble (apparamment) difficile à mesurer.

– **Le mode commutation de paquets : *Wormhole, Virtual-Cut-Through and Buffered-Wormhole*.**
Pour ces modes de commutation, le message est découpé en paquets, ce qui constitue la différence majeure avec le modèle précédent (il peut être vu comme une version dynamique du mode de commutation de circuits). En effet, le premier paquet joue le rôle de l'entête du message, mais tous les paquets le suivent immédiatement, et le dernier paquet libère les liens retenus pour le passage du message. Le modèle permettant de calculer le coût d'une communication avec ce mode de commutation est le même que pour le précédent. La différence entre les différents modes provient du comportement des paquets lorsque le premier paquet est bloqué (par exemple, il ne reste aucun lien de libre). Dans ce cas :

  – Dans le mode *Wormhole*, l'avancée des paquets est stoppée. Ils restent tous à la même place.
  – Pour le mode *Virtual-Cut-Through*, les paquets continuent leur avancée jusqu'à ce qu'ils atteignent tous le paquet de tête. Ce modèle considère pour cette raison que la capacité des éléments de stockage intermédiaire (*buffer*) est illimitée.
  – Pour le mode *Buffered-Wormhole*, lorsque le premier paquet est bloqué, un certain nombre de paquets le rejoignent, mais le nombre de ces paquets est limité par la capacité des *buffer*.

Nous ne tenons pas compte de la possibilité d'envoi multiplexé de messages (*pipeline*), en effet, à l'heure actuelle, il n'existe aucun modèle satisfaisant pour estimer le coût de communication lorsqu'un lien est partagé entre plusieurs messages. Il semble que cette méthode est efficace pour un nombre restreint de messages, quel que soit leur longueur, mais que les performances se dégradent pour un grand nombre de petits messages. Ces conclusions sont issues d'expériences [Cal95].

Dans les deux tableaux suivants sont présentés les modes de commutation utilisés par quelques machines parallèles actuelles, ainsi que les valeurs des différents paramètres pour le calcul des coûts de communication.

Nom de machine	**Constructeur**	**Topologie**
iPSC/2	Intel	Hypercube
iPSC/860	Intel	Hypercube
Computing Surface	Meiko	?
T-Node/Mega-Node	Telmat	Reconfigurable
CM-5	Thinking Machine Corp.	Fat-tree
Paragon	Intel	Grille-2D
CS-2	Meiko, Parsys, Telmat	Omega
T3D	CRAY	Tore-3D
SP-1/SP-2	IBM	Omega

Nom de machine	Mode de commutation	α μs	τ μs/byte
iPSC/2	commutation de circuits	136	0.384
iPSC/860	commutation de circuits	350	0.2
Computing Surface	commutation de messages	250	1.000
T-Node/Mega-Node	commutation de messages	4.85	1.125
CM-5	Wormhole	73	0.1
Paragon	Wormhole	100	0.005
CS-2	Wormhole	12	0.02
T3D	Wormhole	?	0.0033
SP-1/SP-2	(Buffered-)Wormhole	?	0.0125

Il est intéressant de constater que la dernière génération d'ordinateurs parallèles sont basés sur des réseaux multi-étages (*fat-tree* et omega), et la distance dans de tels réseaux est généralement assez faible (inférieure à 10). Ainsi, avec un mode de commutation de type *wormhole*, l'importance du terme $d \times \delta$ est très faible.

4.3 Ordonnancement d'une tâche divisible sur une grille 2D

Nous proposons un algorithme pour ordonnancer une tâche arbitrairement divisible sur un nombre non limité de processeurs liées par un réseau d'interconnexion qui est une grille-2D. L'idée de base est d'exploiter certains résultats et travaux provenant du domaine de l'optimisation des communications dans les réseaux point à point. Nous nous intéressons au départ aux stratégies pour faire de la **diffusion**. Le but le l'algorithme présenté est la minimisation de la date de fin d'exécution de la tâche complète (*makespan*). Cette date sera notée C_{max} dans la suite.

4.3.1 Description du modèle de machine cible

Nous considérons une machine parallèle à mémoire distribuée formé d'un ensemble de noeuds liés entre eux par un réseau d'interconnexion dont la topologie est une grille-2D. Les communications dans cette machine se font par échange de messages. Chaque noeud est composé d'un processeur de calcul et d'un processeur dédié aux communications. Tous les noeuds sont supposés identiques. On suppose, pour le cas qui nous intéresse, que le recouvrement des communications par des calculs est possible. Le processeur dédié aux communications est supposé 4-ports (c'est-à-dire qu'il peut simultanément envoyer quatre messages). Le mode de commutation pris en compte est le mode *wormhole*.

Remarques
Pour le type de modèle générique qui est proposé, il est possible de considérer que la totalité d'une communication ne peut être recouverte du fait d'un temps incompressible dû à l'empaquetage du message à l'émission et à l'opération inverse à la reception. Ce surcoût venant s'ajouter aux communications n'est pas obligatoire, mais on le retrouve pour des programmes écrits en (*PVM (Parallel Virtual Machine) ou MPI (Message Passing Interface)*).

Concernant le nombre de ports de l'élément responsable des communications, il faut noter que du point de vue des contraintes architecturales réelles, le principal goulot d'étranglement de la machine se situe entre la mémoire et le processeur de communications.

4.3.2 L'algorithme de Peters et Syska

Puisque avec le mode de commutation *wormhole* le poids du terme dépendant de la distance est faible, il est intéressant de communiquer loin de la source de la diffusion afin de maximiser le nombre de processeurs atteint à chaque étape. Nous utilisons pour la grille-2D l'algorithme de Peters-Syska [PS93], qui produit des résultats optimaux (en nombre d'étapes) sur un tore-2D formé de 5^{2k} processeurs et applicable pour une grille-2D non bornée. Il faut noter que depuis, certains travaux ont généralisés la technique des pavages pour des dimensions de tores différentes [Cal95]. Il faut noter que cet algorithme est optimal pour le mode *wormhole*, mais qu'il produit de moins bons résultats pour des versions *pipeline* du mode de commutation de messages, lorsque les messages sont plus grands [Sys92] [Rum94].

Algorithme de diffusion dans un tore-2D(5^{2k}) :

Cet algorithme est composé d'une série d'étapes composée chacune de deux phases de communications. Durant la première phase, un message est envoyé avec un mouvement de cavalier (du jeu d'échecs), c'est-à-dire qu'il traverse l liens depuis le processeur émetteur dans une direction et $2 \times l$ liens dans l'autre direction (comme l'illustre la figure 4.1). La seconde phase consiste en un mouvement de croix. Pour une grille-2D, comportant 5^{2k} processeurs, le nombre d'étapes est égal à k. Dans la figure 4.1, une seule étape est représentée, contre

deux dans la figure 4.2.

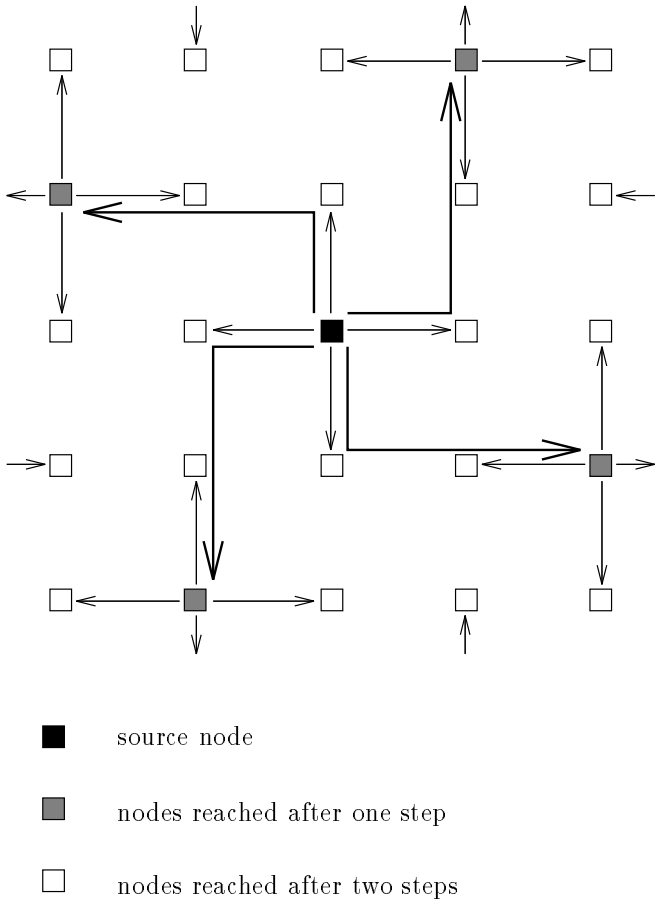


FIG. *4.1 - : Algorithme de Peters et Syska pour un tore-2D avec* 5^2 *processeurs.*

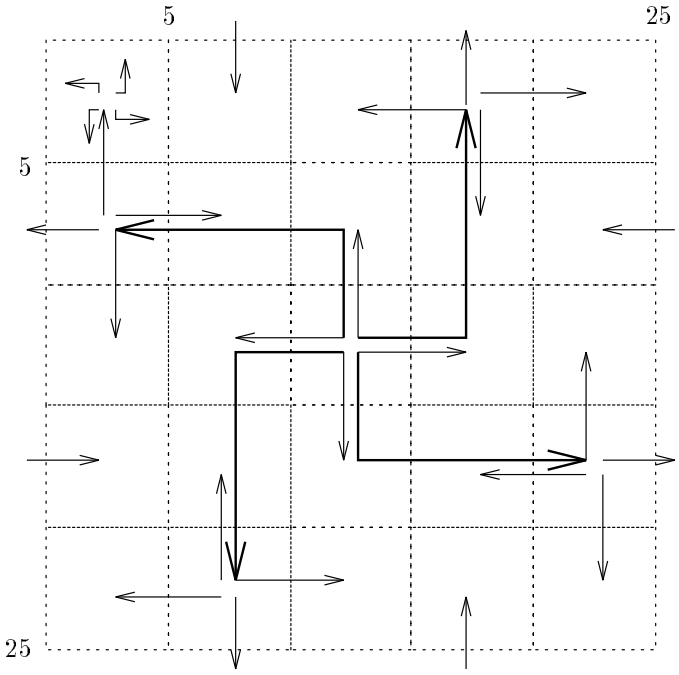


FIG. *4.2 - : Algorithe de Peters et Syska pour un tore-2D avec* 5^4 *processeurs.*

4.3.3 Principe de l'ordonnancement

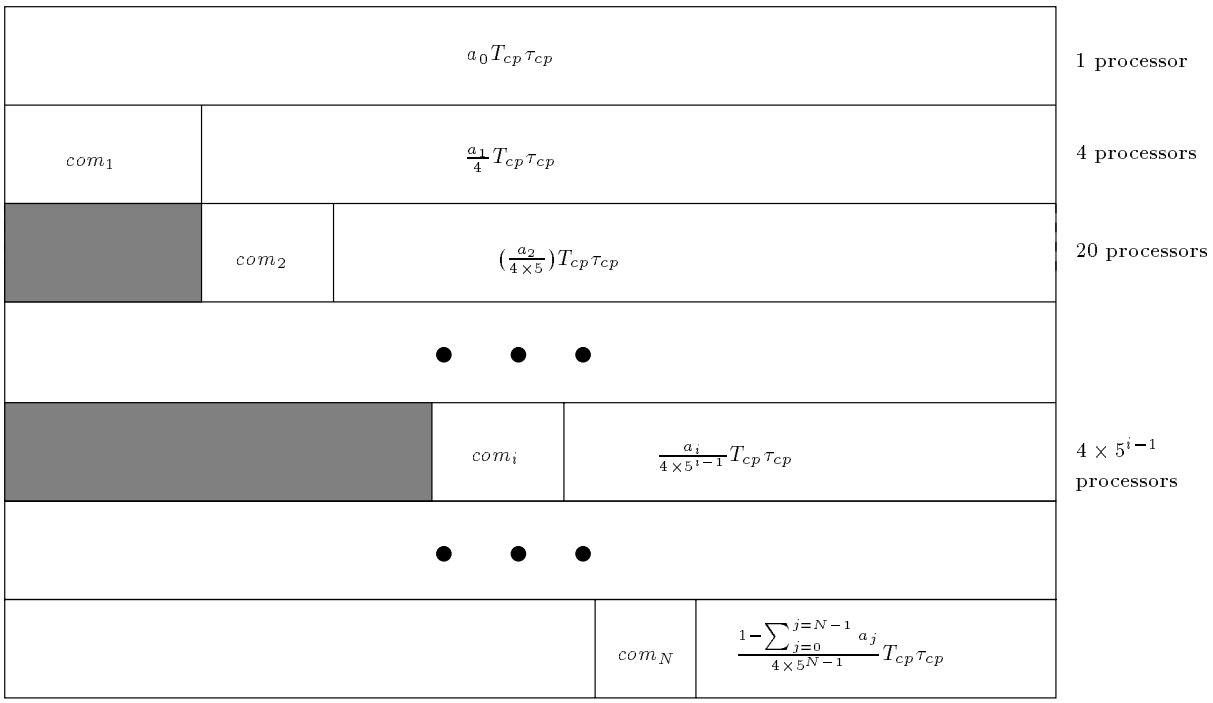
Comme c'est souvent le cas lorsque des communications sont prises en compte, la détermination d'un bon ordonnancement résulte de la recherche d'un compromis entre un grand nombre de processeurs afin de minimiser le temps de calcul, et un faible nombre de processeur afin de limiter le nombre et la durée des communications. Le principe de l'ordonnancement de la tâche est le même que celui proposé par Błażewicz et al. [BD93].

Le premier processeur conserve une partie de la tâche de départ et envoi à quatre autres processeurs le reste de cette tâche (un quart de la tâche restante à chacun d'eux). Chacun de ces processeurs agit de la même façon, il conserve une partie de ce qu'il a reçu et envoi un quart du reste à d'autres processeurs. La question que l'on peut se poser est à chaque étape à quels autres processeurs envoyer une partie de la tâche. Dans la suite, un processeur est dit de **niveau** i si il est atteint à l'étape i de communication. Une première approche consiste à envoyer cette tâche à tous les plus proches voisins. Le nombre de processeurs atteint à l'étape i (qui est aussi le nombre de processeurs appartenant àce niveau) est égal à $4i$ (à partir de la première communication). Donc, au mieux, un total de $2i(i+1)+1$ processeurs travaillant en même temps à l'étape i. De plus, certains processeurs vont recevoir plusieurs fois un message ce qui va déséquilibrer la charge de tous les processeurs. On envisage alors une autre solution, employer la stratégie utilisée pour la diffusion d'un message dans une grille. Pour cela, on utilise l'algorithme de Peeters et Syska précédemment décrit. Avec cette stratégie, le nombre de processeurs du niveau i est égal à $4 \times 5^{i-1}$, et le nombre total de processeurs travaillant en même temps à cette étape est égal à 5^i. La suite des envois et des calculs doit être telle que tous les processeurs terminent leur exécution en C_{max}. La résolution

de ce problème d'ordonnancement revient à résoudre le système d'équations suivant.

4.3.4 Système d'équations

On appelle V le volume de données nécessaire au calcul de la tâche de départ. On note $\tau_{cal}(V_i)$ le temps de calcul de cette tâche pour un volume V_i de donnée. Le premier processeur exécute une partie de cette tâche et envoie le reste à quatre autres processeurs distants. Son temps de calcul est égal à $T^1_{cal} = \tau_{cal}(a_0 V)$ où $a_0 V$ représente la part de la tâche qui est calculée par les processeurs du niveau 0, niveau constitué du seul processeur source. Le reste de la tâche est envoyé à quatre processeurs situés à distance d. Le processus est répété pour chaque ensemble de processeurs d'un niveau qui reçoit des données. C'est-à-dire que l'ordonnancement des processeurs du second niveau est formé par une attente de communication et par une partie calcul (dont une partie recouvre la communication de ces processeurs vers les processeurs du niveau suivant). La coût de l'attente de communication est égale à $T^2_{com} = \alpha + d\delta + \tau_{com}(\frac{1-a_0}{4}V$. La durée du calcul est égale elle à $T^2_{cal} = \tau_{cal}(a_1 V)$. La forme de l'ordonnancement recherché est celui représenté en figure 2.6.



Communications:

com 1:$\alpha + \frac{1-a_0}{4} T_{cm}\tau_{cm}$

com 2: $\alpha + \frac{1-a_0-a_1}{4^2} T_{cm}\tau_{cm}$

....

com i: $\alpha + \frac{1-\sum_{j=0}^{j=i-1} a_j}{4^i} T_{cm}\tau_{cm}$

FIG. *4.3 - : Ordonnancement pour lequel la date de fin d'exécution est la même pour tous les processeurs.*

Lorsque la dimension de la grille n'est pas limitée, la distance est aussi une inconnue. On note d_i la distance séparant un processeur P_j^{i-1} de niveau $i-1$ d'un processeur de niveau i recevant ses données de P_j^{i-1}. Pour la première étape, $d_1 = d$.

- $\left(\alpha + d_1\delta + \frac{1-a_0}{4}V\tau_{com}\right) + \frac{a_1}{4}V\tau_{cal} = a_0 V\tau_{cal}$
- $\left(\alpha + d_2\delta + \frac{(1-a_0-a_1)}{4^2}V\tau_{com}\right) + \frac{a_2}{4\times 5}V\tau_{cal} = \frac{a_1}{4}V\tau_{cal}$
- ...
- $\left(\alpha + d_i\delta + \frac{1-\sum_{j=0}^{j=i-1} a_j}{4^i}V\tau_{com}\right) + \frac{a_i}{4\times 5^{i-1}}V\tau_{cal} = \frac{a_{i-1}}{4\times 5^{i-2}}V\tau_{cal}$
- ...
- $\left(\alpha + d_N\delta + \frac{1-\sum_{j=0}^{j=N-1} a_j}{4^N}V\tau_{com}\right) + \frac{a_N}{4\times 5^{N-1}}V\tau_{cal} = \frac{a_{N-1}}{4\times 5^{N-2}}V\tau_{cal}$
- $\sum_{i=0}^{i=N} a_i = 1$

La distance entre deux processeurs de niveau consécutif en fonction de l'étape de communication dépend de la valeur de N. Pour une valeur de N donnée, la grille est de taille 5^{2N}. La première communication (correspondant au mouvement de cavalier) est à distance $\frac{3}{4}\times 5^N$. La seconde communication correspondant au mouvement de croix est à distance $\frac{5^N}{4}$. Chaque processeur appartenant au niveau deux est le centre d'une grille de dimension 5^{N-1}. Donc le processus recommence et au niveau i, l'expression de la distance est égale à : $d_i = \frac{3}{4}5^{N-\frac{i-1}{2}}$ si i est impair et
$d_i = \frac{1}{4}5^{N-\lfloor\frac{i-1}{2}\rfloor}$ si i est pair.
Pour une valeur donnée de N, il est possible de calculer la valeur de chaque a_i. Cependant, ce système d'équations est difficile à résoudre, nous l'exprimons donc d'une autre manière.

Supposons qu'au départ, la quantité totale de travail soit égale à W. Le temps requis pour l'exécution de cette tâche de façon séquentiel est égal à $W\tau_{cal}$ unités de temps. Le premier processeur conserve pour lui a_0 (une partie de W) pour recouvrir la communication de ce processeur vers les quatre processeurs du niveau 1. La quantité de travail restante est divisée en cinq parties appelées W_1. L'ensemble des processeurs appartenant au niveau 0 et 1 conserve chacun une partie a_1 du travail qu'il a reçu. Cette quantité de travail est celle nécessaire pour recouvrir la communication vers les processeurs du niveau deux. Le processus

est répété pour les niveaux suivants. Cette découpe est illustrée en figure 4.4.

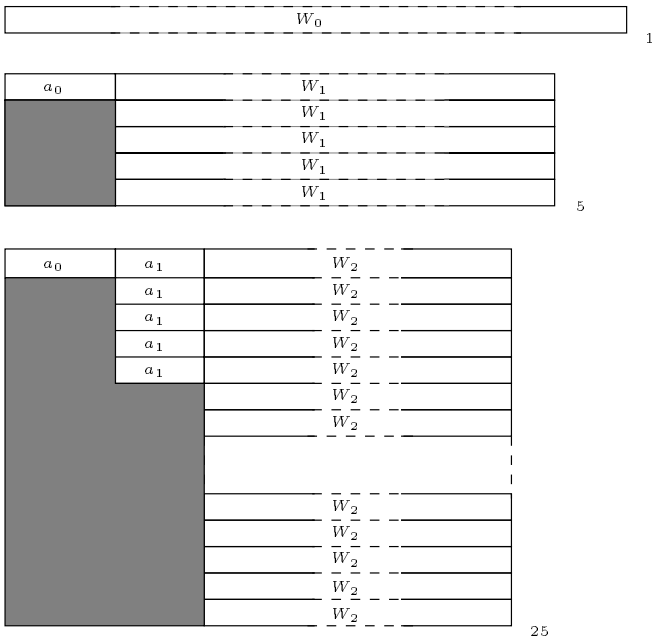


FIG. *4.4 - : Une nouvelle découpe pour l'expression d'un autre système d'équations.*

A l'étape i, le processus donne :

$$W_{i+1} = \frac{W_i - a_i}{5}$$

Le recouvrement des communications par des calculs peut alors s'exprimer par :

$$a_i \tau_{cal} = \alpha + d_{i+1}\delta + W_{i+1}\tau_{com}$$

Le recouvrement de la dernière communication s'exprime alors par[2] :

$$W_N - 1 = (5 + \frac{\tau_{com}}{\tau_{cal}})W_N + \frac{\alpha}{\tau_{cal}}$$

De plus, il arrive un stade auquel le division du travail entre processeurs est inutile car l'exécution de la totalité du travail restant sur un processeur nécessite une durée inférieure au coût d'envoi d'un message de cette longueur. Cette limite est exprimée par :

$$W_N \tau_{cal} \leq \alpha + \frac{W_N - a_N}{5}\tau_{com} + \frac{W_N - a_N}{5}\tau_{cal}$$

La limite lorsque la taille du message tend vers 0 donne une limite inférieure de la taille du dernier travail. Lorsque $W_N - a_N \to 0$ alors $W_N \leq \frac{\alpha}{\tau_{cal}}$. Ce qui donne pour l'expression des W_{N-i} :

$$W_{N-i} = (5 + \frac{\tau_{com}}{\tau_{cal}})^i W_N + \sum_{j=0}^{j=i-1} (5 + \frac{\tau_{com}}{\tau_{cal}})^j \frac{\alpha}{\tau_{cal}}$$

[2] A partir d'ici, nous négligeons le terme en δ, dans un but de clarté. On remarque toutefois que ce terme peut être réinjecté dans les équations à la fin du calcul.

et,

$$a_{N-1-i} = \frac{\alpha}{\tau_{cal}} + \frac{\tau_{com}}{\tau_{cal}} \left((5 + \frac{\tau_{com}}{\tau_{cal}})^i W_N + \sum_{j=0}^{j=i-1} (5 + \frac{\tau_{com}}{\tau_{cal}})^j \frac{\alpha}{\tau_{cal}} \right)$$

On peut donc calculer une valeur limite pour N:

$$W_0 = (5 + \frac{\tau_{com}}{\tau_{cal}})^N \frac{\alpha}{\tau_{cal}} + \frac{\alpha}{\tau_{cal}} \left(\frac{(5 + \frac{\tau_{com}}{\tau_{cal}})^N - 1}{(5 + \frac{\tau_{com}}{\tau_{cal}}) - 1} \right)$$

and then:

$$n \approx \frac{ln\left(\frac{\tau_{cal} W_0}{\alpha}\right)}{ln\left(5 + \frac{\tau_{com}}{\tau_{cal}}\right)} + 1$$

Remarque 1 :
La même méthode peut être appliquée pour des grilles-kD et des hypercubes.

Remarque 2 :
Il est possible de considérer des processeurs uniformes au lieu des processeurs identiques. Cependant, si l'on note m le nombre de processeurs, le nombre d'équations nécessaire pour la résolution du problème est du même ordre que m alors qu'il est de l'ordre de $log(m)$ lorsque les processeurs sont identiques. La méthode de résolution reste la même, mais le τ_{cal} au lieu d'être commun à tous les processeurs devient une valeur propre à chacun d'eux, et est proportionnel à leur vitesse.

Remarque 3 :
Comme pour la remarque 2, le nombre d'équation devient de l'ordre de m (nombre de processeurs) lorsque l'hypothèse 1-port est prise en considération. Dans ce cadre, les communications ne sont plus effectuées en parallèle, mais de façon séquentiel, comme l'illustre la figure 4.5.

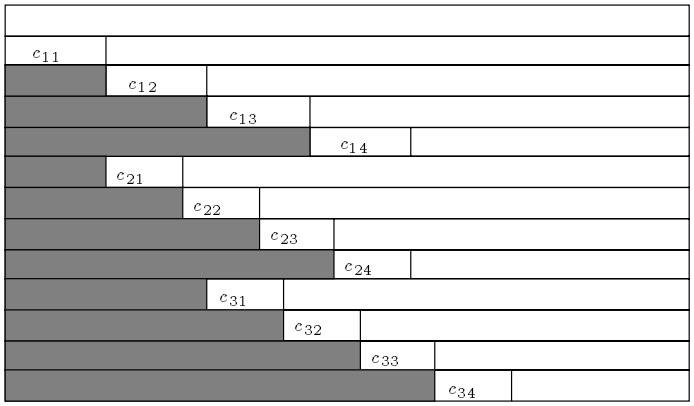


FIG. *4.5 - : Les processeurs dédiés aux communications sont 1-port.*

Remarque 4 :
Lorsque les temps d'empaquetage des paquets ainsi que les opérations inverses ne peuvent pas être recouverts, une partie du coût d'initialisation (α) pour émission est perdu, et la même quantité de temps est perdue à la reception. L'ordonnancement que l'on peut espérer obtenir à la forme indiquée en figure 4.6.

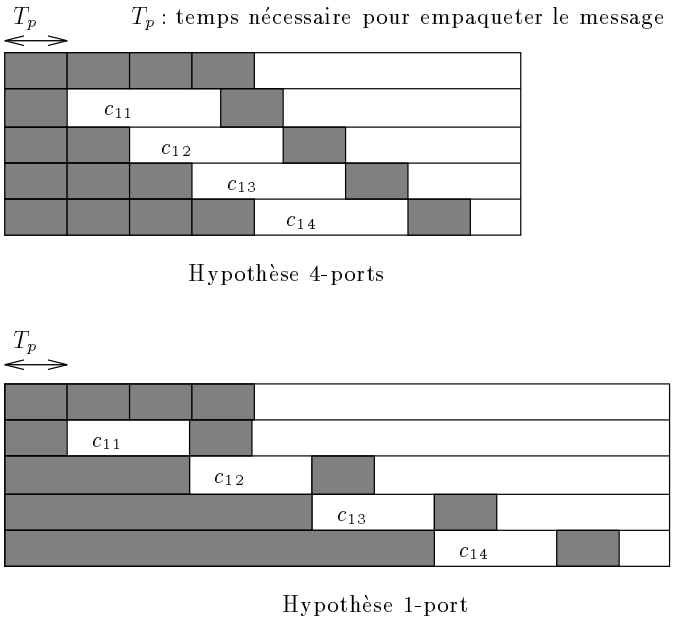


FIG. *4.6 - : L'empaquetage des messages ne peut pas être recouvert.*

Remarque 5 :
Le même type de stratégie peut être mis en oeuvre pour un graphe ayant une structure arborescente, ou un arbre miroir (dans ce dernier cas, en plus d'un algorithme de diffusion, on utilise un algorithme de regroupement des messages du type *all-to-one*).

4.4 Conclusion

Bien que la plupart des machines qui sont produites aujourd'hui ne repose plus sur des réseaux point à point (mais ce n'est pas une généralité comme l'atteste le T3D de CRAY), il est interessant de constater que dès que les communications sont prises en compte ainsi qu'un certains nombre de contraintes architecturales comme c'est le cas dans le domaine de l'optimisation des communications, les stratégies performantes pour ce type de problèmes peuvent être avantageusement exploitées pour la résolution d'autres problèmes. D'autre part, ce modèle générique permet de modéliser une grande partie des machines actuelles disposant d'un réseau d'interconnexion point à point, puisque les modes de commutation de messages s'orientent de plus en plus vers un type *wormhole*. Rien n'empêche ce modèle d'évoluer vers d'autre types de machines, comme les machines basées sur des réseaux multiétages. En effet, il suffit de savoir faire une diffusion efficacement dans ce type de machines pour savoir également savoir ordonnancer un ensemble de tâches indépendantes ou une importante tâche

arbitrairement divisible.

Il est à noter que ce chapitre est l'objet d'un travail en coopération avec l'équipe du Professeur Jacek Błażewicz de l'université de Poznań (Pologne).

Partie III

Ordonnancement d'arbres avec recouvrement

Chapitre 1

Introduction

Parmi les modèles d'exécution autorisant le **recouvrement des communications par des calculs**, ceux décrits par Papadimitriou et Yannakakis [PY88] d'une part, et Rayward-Smith [RS87] d'autre part connaissent un large succès. Il existe pour chacun de ces modèles un grand nombre de résultats.
Les **arbres** sont choisis pour illustrer la mise en œuvre de techniques et la mise au point d'algorithmes. Deux raisons sont à l'origine de ce choix : la **complexité** des problèmes d'ordonnancement dans ces modèles, et l'étendue du spectre des **applications** qui peuvent être représentées par cette famille de graphes particuliers.
En effet, les problèmes d'ordonnancement dans ces modèles sont **difficiles** pour des graphes orientés sans cycle [PY88] lorsque le nombre de processeurs n'est pas limité, et de même complexité pour des arbres lorsque l'objectif est de déterminer le nombre minimum de processeurs permettant d'atteindre la **date de fin d'exécution minimale** [Saa92, Pic93, Vel93b].
De plus, les arbres constituent la représentation privilégiée d'un vaste ensemble d'applications. Par exemple, le processus d'évaluation d'expressions arithmétiques [BOV85, Rev94], les méthodes multifrontales de factorisation symbolique de Cholesky [DGL89, Poz92], le tri bitonique [Bat68], la sommation parallèle, la somme des préfixes [Wyl79, LF80], les traces des exécutions de programmes Prolog ou les algorithmes de recherche sont autant d'applications qui peuvent être représentées sous la forme de structures arborescentes.
Enfin, un certain nombre de difficultés apparaissent plus clairement du fait de la structure particulière des précédences entre les tâches, et permettent d'évaluer l'efficacité des stratégies qui peuvent notamment être employées pour recouvrir les communications.

L'objectif de cette partie est d'étudier le comportement théorique des **stratégies de regroupement** pour l'ordonnancement des arbres sur un système formé de deux processeurs **identiques** dans un premier temps, puis de m processeurs identiques dans un second temps, pour des modèles d'exécution autorisant le recouvrement.
Dans le premier chapitre nous avons présenté un algorithme basé sur une méthode de regroupement de tâches. Cet algorithme produit, pour un système formé de deux processeurs identiques, des ordonnancements optimaux pour des arbres dont les **durées d'exécution** des tâches et les **communications** sont **unitaires** (*UECT*). Nous étudions son comportement pour des **granularités** différentes : arbres à grain fin pour lesquels les temps de

communication sont supérieurs aux temps de calcul, et l'inverse pour les arbres à gros grain. Considérer différentes granularités revêt un intérêt tout particulier pour certaines applications, par exemple, pour la factorisation symbolique de Cholesky multifrontale, le graphe d'élimination des facteurs est un arbre dont les nœuds ne sont pas de poids équivalent (80 % des calculs sont effectués par les tâches appartenant aux trois premiers niveaux) [Duf95].
Dans le second chapitre, une heuristique d'ordonnancement d'arbres sur un nombre limité de processeurs est présentée. Cette heuristique produit, dans un premier temps, un ordonnancement sur un nombre non limité de processeurs puis réduit le nombre de machines utilisées. Une comparaison est proposée avec un algorithme de complexité linéaire présenté par Varvarigou, Roychowdhury, Kailath et Lawler [VRKL94] de type niveau par niveau. Nous donnons en fin de chapitre une idée de calcul de priorité des tâches, basée sur l'arité, la largeur du graphe et le chemin critique, dans le but de produire de meilleurs ordonnancements.
Enfin, dans le dernier chapitre de cette seconde partie, après un tour d'horizon des travaux existant dans le domaine de l'ordonnancement de graphes de tâches sur un système formé de processeurs uniformes (c'est-à-dire avec des vitesses d'exécution différentes) nous étudions l'ordonnancement des arbres sur deux processeurs de ce type. Nous mettons en évidence les principales difficultés inhérentes à cette différence de vitesses. Nous présentons, en outre, un algorithme qui produit des ordonnancements optimaux pour des arbres complets.

Chapitre 2

Ordonnancement d'arbres sur deux processeurs identiques

Cette première partie va nous permettre de mettre au point des techniques pour **l'ordonnancement des arbres avec communications**, sur **deux processeurs**. A partir de ces algorithmes, nous essayerons dans la seconde partie de généraliser les principes de leur mise au point pour la conception d'heuristiques pour m processeurs ($m \geq 3$ et fixé).
Parmi l'ensemble des modèles d'exécution présentés dans la partie I, certains autorisent le **recouvrement des communications par des calculs**. Les problèmes d'ordonnancement traités dans les présents chapitres auront pour cadre de tels modèles. Le premier problème d'ordonnancement d'arbres que nous considérons se note $P_2 \mid arbres,\ p_i = 1,\ c = 1 \mid C_{max}$. Il correspond au modèle *UECT* qui est identifiable avec le modèle décrit par Rayward-Smith [RS87]. Seront examinés tour à tour les problèmes à gros grain puis à grain fin. Le problème noté $P_2 \mid arbres,\ p_i,\ c = 1 \mid C_{max}$ correspond à une hypothèse **gros grain**, c'est-à-dire à des arbres dans lesquels toutes les communications sont inférieures aux temps de calcul (la notion de grain se réfère à la taille d'un calcul). Enfin, le dernier problème pris en compte correspond à une hypothèse **grain fin**, c'est-à-dire à des arbres dans lesquels les communications sont supérieures aux temps de calcul. Il se note $P_2 \mid arbres,\ p_i = 1,\ c \mid C_{max}$. Le modèle de fine granularité peut être identifié avec la description de Papadimitriou et Yannakakis [PY88]. Enfin, remarquons que le fait de considérer des arbres (l'orientation des arcs va des feuilles vers la racine) et non des anti-arbres rend toute duplication de tâche inutile.

2.1 $P_2 \mid arbre,\ p_i = 1,\ c = 1 \mid C_{max}$

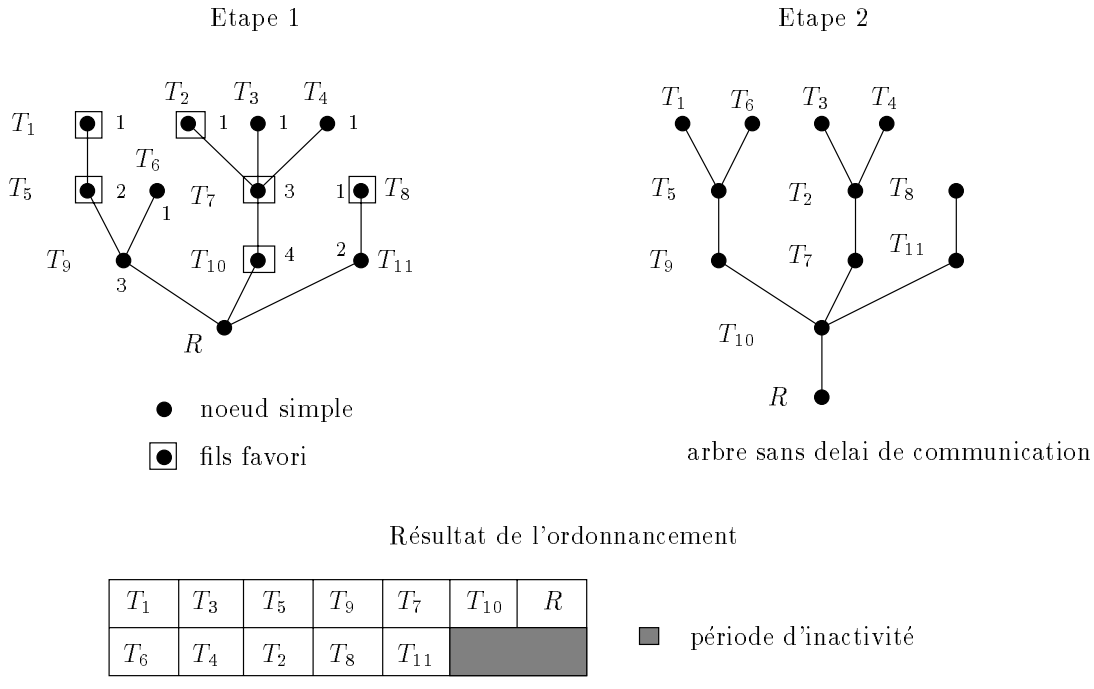
L'ordonnancement d'arbres *UECT* n'a été abordé que récemment. Presque simultanément et de façon indépendante, Bart Veltman, Rachid Saad et Christophe Picouleau ont montré la NP-complétude de l'ordonnancement sur un nombre de processeurs non fixé d'arbres *UECT* [Vel93b, Saa92, Pic93]. Ce même problème pour un nombre m fixé de processeurs peut être résolu par un algorithme de programmation dynamique de complexité $O(n^{2(m-1)})$, proposé par Varvarigou, Kailath et Roychowdhury [VRK93]. L'idée de cet algorithme repose sur la notion **de fils favorisé**, introduite par les auteurs sous le nom de *favorite child property*. A partir de cet algorithme, une heuristique a été dérivée par Lawler, pour m machines, optimale sur 2 processeurs. De la même façon que pour le résultat de NP-complétude à nombre de machines non fixé, trois algorithmes ont été produits simultanément et ont donné le même résultat d'optimalité. Nous décrivons dans la suite rapidement ces différentes méthodes, respectivement développées par Lawler, Veldhorst et Picouleau. Nous poursuivons par la description d'un nouvel algorithme et la preuve de son optimalité.

Auteurs	Complexité	Notation du problème	Stratégies
[Saa92, Vel93b, Pic93]	NP-complet	$P \mid arbre, p = 1, c = 1 \mid Cmax$	-
[VRK93]	$O(n^{2(m-1)})$	$P_m \mid arbre, p = 1, c = 1 \mid Cmax$	prog. dynamique
[Law93, Vel93a]	$O(n)$	$P_2 \mid arbre, p = 1, c = 1 \mid Cmax$	niveau par niveau
[Pic93, GT93]	$O(n)$	$P_2 \mid arbre, p = 1, c = 1 \mid Cmax$	regroupement

2.1.1 L'algorithme de Lawler

L'algorithme de Lawler est basé sur la notion introduite dans [VRK93] et appelée *favorite child property*. Parmi l'ensemble des fils d'une tâche, le fils favorisé est celui qui a la date de fin d'exécution au plus tôt la plus élevée. En cas d'égalité, seul l'un d'eux est choisi pour recouvrir la communication (figure 2.1) (le choix est effectué sur un anti-arbre, donc, sur la figure les fils favorisés sont en fait des parents favorisés). Lorsque le fils favorisé de chaque nœud qui n'est pas une feuille a été déterminé, l'auteur construit un arbre sans délai de communication (*delay-free tree*), qui est à peu de choses près un arbre dans lequel les communications ont été éliminées. A peu de chose près car si un nœud est alloué à un processeur P, et si son fils favorisé est ordonnancé dans l'unité de temps suivante, il devra être alloué au même processeur, alors que dans un arbre sans communication, cela n'a pas d'importance. Ensuite, un algorithme de type chemin critique (par exemple l'algorithme de Hu [Hu61]) est appliqué à ce graphe intermédiaire. Suivent quelques corrections pour prendre en compte les contraintes de précédence et le fait qu'un nœud et son fils favorisé doivent être

alloués au même processeur s'ils sont ordonnancés durant deux unités de temps successives.



T_1	T_3	T_5	T_9	T_7	T_{10}	R
T_6	T_4	T_2	T_8	T_{11}		

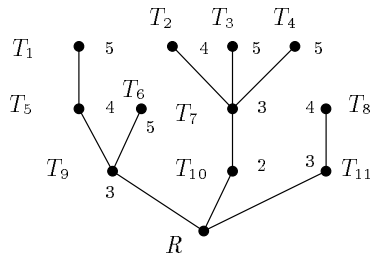
FIG. *2.1 - : Un exemple d'ordonnancement obtenu par application de l'algorithme de Lawler.*

Cet algorithme n'est pas optimal pour un nombre de processeurs supérieur ou égal à trois. Il permet, cependant, d'obtenir de bons résultats. Lawler propose une borne pour le cas au pire de cet algorithme : $\omega_{FilsFavori} \leq \omega_{optimal} + (m-2)$ [Law93]. Nous montrerons dans la seconde partie que cette borne n'est pas atteinte et que la qualité des ordonnancements construits par cet algorithme pour m processeurs sont tels que : $\omega_{FilsFavori} \leq \omega_{optimal} + \left\lceil \frac{(m-1)(m-2)}{2m} \right\rceil$.
Cet algorithme et celui développé par Varvarigou, Roychowdhury et Kailath sont réunis dans [VRKL94].

2.1.2 L'algorithme de Veldhorst

Cet algorithme est constitué de trois étapes. Dans un premier temps il alloue à chaque tâche une étiquette qu'il appelle *scheddist-label* et qui est calculée en partant de la date de fin d'exécution au plus tôt. Ensuite, en partant de la racine à laquelle est allouée l'étiquette 1, le prédécesseur qui a la date de fin d'exécution au plus tôt la plus importante est allouée l'étiquette 2, et 3 à tous ses frères ainsi qu'à son prédécesseur ayant la date de fin d'exécution au plus tôt la plus importante, et cela de façon récursive. Une liste est ensuite créée (les successeurs d'une tâche dans cette liste sont, l'aîné de ses prédécesseurs (nommé par l'auteur *eldest son* (encore une fois il s'agissait d'anti-arbre)) et ses frères). Ce qui donne pour le

même exemple que précédemment (voir figure 2.2) :
$L_{Veldhorst} = (T_4, T_3, T_1, T_6, T_2, T_8, T_5, T_7, T_{11}, T_9, T_{10}, R)$
Enfin, il alloue à chaque tâche une date d'ordonnancement et un processeur en tenant compte des contraintes de précédence et des prédécesseurs (comme précédemment).



T_4	T_1	T_2	T_7	T_{11}	T_{10}	R
T_3	T_6	T_8	T_5	T_9		

FIG. *2.2 - : Un exemple d'ordonnancement obtenu par application de l'algorithme de Veldhorst.*

Remarque :
Cet algorithme et le précédent sont voisins. Ils sont tous deux "niveau par niveau", et ont des étapes en commun. Par exemple, l'étiquetage est équivalent entre le *favorite child* (pour l'algorithme de Lawler) et le *eldest son* (pour celui de Veldhorst). De même, l'arbre sans délai est équivalent à la liste construite par Veldhorst.

2.1.3 L'algorithme de Picouleau

Picouleau présente dans sa thèse [Pic93] un algorithme d'un autre type basé sur des techniques de **regroupement** (*clustering*). Les groupes de tâches considérés ou **travaux** sont des sous-arbres et le critère de choix est le nombre de nœuds appartenant au sous-arbre. Soit un arbre avec N_{sa} sous-arbres différents (étant tous directement liés à la racine), l'auteur note les sous-arbres S_j, et considère que le nombre de nœuds de S_j est supérieur ou égal au nombre de nœuds de S_k (noté respectivement $Poids(S_j)$ et $Poids(S_k)$) si et seulement si $j < k$. L'algorithme s'articule alors de la manière suivante :

- Si $Poids(S_1) \leq \left\lfloor \frac{\sum_{i=1}^{N_{sa}} Poids(S_i)}{2} \right\rfloor$:
  - Détermination de k tel que S_k est le premier sous-arbre pour lequel :
$$\sum_{i=1}^{k} Poids(S_i) > \sum_{i=k+1}^{N_{sa}} Poids(S_i)$$
  - Allocation des $k-1$ premiers sous-arbres et d'une partie du k^{ieme} à un processeur. Le reste des tâches du k^{ieme} sous-arbre et les sous-arbres non encore alloués sont destinés au second processeur.
- Sinon, l'algorithme est appliqué de façon récursive :
  - Construction d'un ordonnancement optimal de S_1 sur deux processeurs

– Création de périodes d'inactivité de manière à autoriser l'addition des sous-arbres restant à allouer.

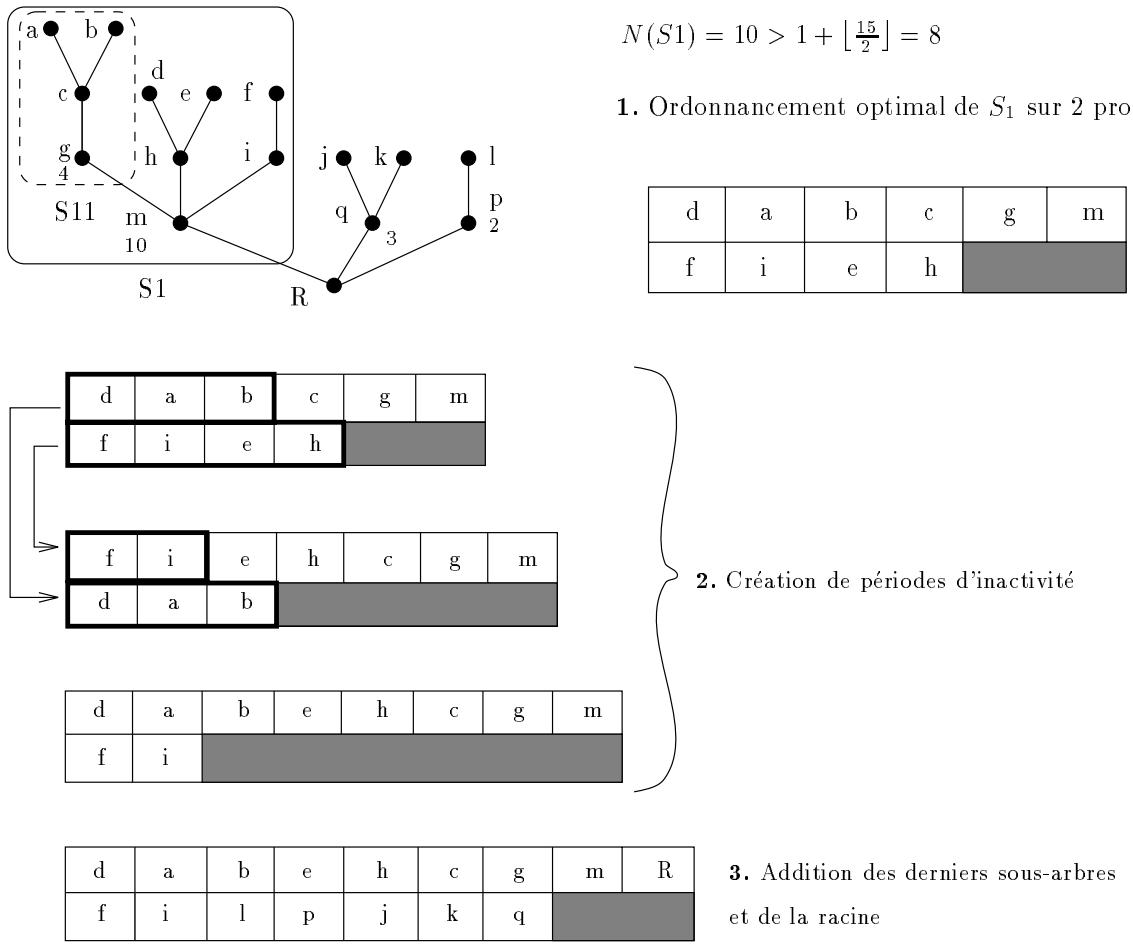


FIG. *2.3 - : Exemple d'ordonnancement obtenu par application de l'algorithme de Picouleau.*

Dans cet exemple, les sous-arbres sont étiquetés de $S1$ à $S3$ (sous-arbre formé des tâches l et p). Lorsque l'algorithme est appelé récursivement sur $S1$, de nouveau les sous-arbres sont nommés de $S11$ à $S13$ (formé des tâches f et i).

2.1.4 Un nouvel algorithme pour $P_2 \mid arbre,\ p_i = 1,\ c = 1 \mid C_{max}$

Notons n le nombre total de tâches de l'arbre, et P_1 et P_2 les deux processeurs. Nous notons P_1 le processeur auquel est allouée la racine.
Les tâches sont notées T_l^i où l correspond au niveau dans l'arbre (au niveau 1 se trouve la racine de l'arbre). Il peut arriver qu'une tâche soit simplement notée T_i dans un souci de simplification des notations et lorsqu'il n'existe aucune ambiguïté. La distance entre la racine et T_l^i est par définition $l - 1$.
Comme pour l'algorithme de Picouleau, le principe de base est une stratégie de regroupement de tâches après un étiquetage de chaque nœud. L'étiquette d'un nœud représente le poids

du sous-arbre dont le nœud est la racine, c'est-à-dire, dans le cas qui nous occupe ($UECT$) le nombre de nœuds du sous-arbre (racine comprise).

Principe de l'algorithme

L'idée principale est d'éliminer les périodes d'inactivité sur le processeur qui exécute la racine (P_1), et d'allouer à P_2 le nombre maximal de tâches de sorte que la propriété de constante activité de P_1 soit respectée. Le nombre maximal de tâches qui puissent être allouées est mis à jour durant l'étape de choix.

L'algorithme est composé de trois phases :

- La première phase consiste à étiqueter l'arbre. L'étiquette de la tâche T représente le nombre de nœuds contenus dans le sous-arbre dont la racine est T (T comprise).
- Les tâches sont distribuées de manière équilibrée entre les deux processeurs. Cette distribution se fait par l'allocation de sous-arbres et par la mise à jour d'un critère de charge noté $Reste$. Ce critère ne concerne que l'un des processeurs. La valeur initiale de $Reste$ est $\left\lfloor \frac{n-2}{2} \right\rfloor$, qui majore le nombre de tâches qui peuvent être allouées au processeur le moins chargé (nous le notons P_2, il n'exécute pas la racine de l'arbre) sans entraîner de périodes d'inactivité sur P_1. L'algorithme est appliqué à partir du niveau deux de l'arbre. Si à ce niveau l'un des sous-arbres est de poids inférieur à $Reste$, il est choisi, tous les nœuds de ce sous-arbre sont marqués, et $Reste$ est mis à jour. Cette mise à jour n'intervient qu'à deux occasions : après l'allocation d'un sous-arbre et lorsqu'à un niveau donné seul un nœud n'est pas marqué. Si à un niveau donné tous les nœuds ont un poids strictement supérieur à $Reste$, alors les tâches sont choisies dans le sous-arbre de plus faible poids. La phase de choix des sous-arbres se termine lorsque $Reste$ est égal à 0.
- Finalement, lorsque les sous-arbres ont été partagés entre les deux processeurs, leur date d'ordonnancement dépend du niveau de la racine. Les premiers sous-arbres exécutés sur P_2 sont ceux qui ont leur racine la plus haute, c'est-à-dire ceux qui ont été choisis en dernier. Pour P_1, on procède de la même façon : exécution des sous-arbres dont les racines sont les plus hautes et dont la totalité des tâches restant à exécuter sont allouées à P_1.

Algorithme MAJYC (Mise A Jour dYnamique de la Charge)

étiquetage de l'arbre
NoeudCourant = racine
NiveauCourant = 2; ListeSousArbresChoisis = $\emptyset$
$Reste = \left\lfloor \frac{n-2}{2} \right\rfloor$

Tant que ($Reste > 0$) **Faire**
examiner tous les noeuds tels que :

$(Niveau(noeud) = NiveauCourant)$ et $(noeud\ non\ marqué)$
les réunir dans une liste L
Si $(Cardinal(L) = 1)$ **Alors**
$Reste = min(Reste, \lfloor \frac{Poids(noeud)-2}{2} \rfloor)$
Sinon
$LP = \{noeuds \mid noeud \in L\ et\ Poids(noeud) \leq Reste\}$
Si (tous les noeuds ont un poids superieur *à* $Reste$) **Alors**
choisir le sous-arbre de plus faible poids
choisir parmi l'ensemble des noeuds appartenant *à* ce
sous-arbre $Reste$ noeuds
$Reste = 0$
Fin Si
Tant que $(Reste > 0)$ et $(LP \neq \emptyset)$ **Faire**
choisir le plus lourd sous-arbre dont la racine est R
marquer l'ensemble de ses noeuds
mettre *à* jour la valeur de $Reste$, $Reste = Reste - Poids(R)$
mettre *à* jour la liste LP en fonction de la nouvelle valeur de $Reste$
Fin Tant que
mettre *à* jour L en fonction des noeuds qui ont *été* marqués
Si $(Cardinal(L) = 1)$ **Alors**
$Reste = min(Reste, \lfloor \frac{Poids(noeud)-2}{2} \rfloor)$
Fin Si
Fin Si
NiveauCourant = NiveauCourant + 1
Fin Tant que

Un exemple complet

Nous considérons un arbre formé de 22 tâches représenté en figure 2.4.
$Reste = \lfloor \frac{22-2}{2} \rfloor = 10$ tâches.
Au niveau 2, trois tâches sont disponibles ayant pour poids respectif 3, 12 et 6. Le sous-arbre de poids 6 est choisi (il s'agit du sous-arbre ayant le maximum de tâches parmi ceux qui en ont moins de $Reste + 1$) et constitue le travail C (voir figure 2.4). $Reste$ est mis à jour $Reste = 10 - 6 = 4$. Au même niveau, il existe une tâche dont le poids est égal à 3 et le sous-arbre correspondant est donc choisi (travail B), ses nœuds sont marqués et $Reste$ est mis à jour $Reste = 1$. Il ne reste ensuite à ce niveau qu'une seule tâche non marquée. On recalcule la valeur de $Reste$. $Reste = min(1, 5) = 1$. On monte dans les niveaux, jusqu'à rencontrer deux sous-arbres de poids supérieur à $Reste$ (5 et 3). Dans le sous-arbre de plus faible poids la dernière tâche est choisie (travail A).
La phase d'allocation est simple, on commence par les sous-arbres dont les racines sont au plus haut niveau, donc par le travail A, puis est ordonnancé le travail B ou le travail C (le

choix est arbitraire). Sur P_1 l'allocation est faite niveau par niveau.

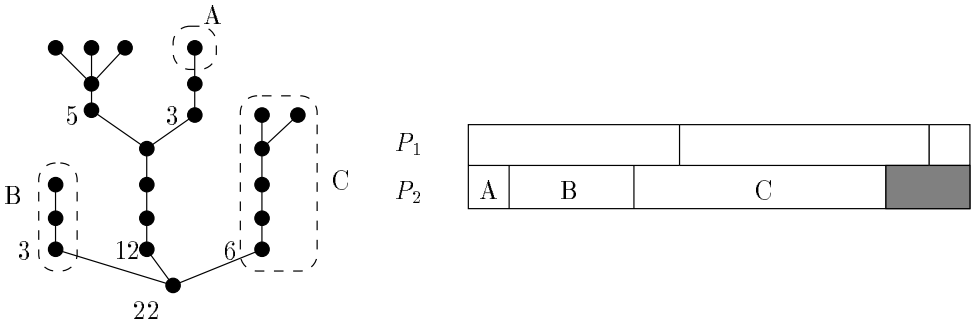


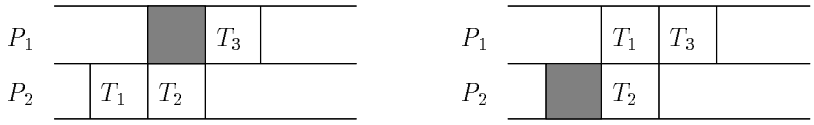
FIG. *2.4 - : Un exemple d'application de ce nouvel algorithme.*

2.1.5 Étude théorique

Nous commençons par quelques remarques préliminaires puis nous donnerons les grandes lignes de la démonstration de l'optimalité qui est détaillée dans [GT93].

Remarques préliminaires

Remarque 1
Soit un arbre dont les temps d'exécution et de communications sont unitaires. Tout ordonnancement optimal d'un tel arbre contenant des périodes d'inactivité sur le processeur qui exécute la racine (P_1) peut être transformé en un nouvel ordonnancement optimal sans période d'inactivité sur P_1 en déplaçant la tâche communicante de P_2 à P_1 (T_1 sur la figure 2.5).



Les périodes d'inactivité sur P_1 sont éliminées par déplacement des tâches communicantes

FIG. *2.5 - : Élimination des périodes d'inactivité sur P_1.*

Remarque 2
L'algorithme présenté produit des ordonnancements dans lesquels il n'existe aucune communication de P_1 vers P_2. En effet, lorsqu'une tâche est allouée à P_2, toutes les tâches du sous-arbre dont elle est la racine sont allouées à ce même processeur.

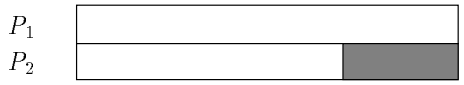
Remarque 3
L'unique ordonnancement optimal pour une chaîne *UECT*, quel que soit le nombre de processeurs est obtenu en allouant la chaîne totale sur un unique processeur. Toute autre allocation,

impliquant une coupure dans la chaîne, entraîne l'apparition d'une communication non recouverte.

Remarque 4
Soit un ordonnancement optimal d'un arbre *UECT* formé de n tâches sur deux processeurs dont l'un est toujours actif. Le nombre de tâches sur le processeur qui n'exécute pas la racine ne peut pas être supérieur à $\left\lfloor \frac{n-2}{2} \right\rfloor$. En effet, durant le calcul de la racine, ce processeur ne peut exécuter aucune tâche, et une tâche est nécessaire pour recouvrir la communication avec le processeur qui exécute la racine.

Optimalité de l'algorithme présenté

L'idée centrale de cet algorithme est de maintenir une activité constante sur l'un des processeurs que l'on a noté P_1. D'après la remarque 1, comme aucune période d'inactivité n'est présente dans l'ordonnancement de P_1, la structure de l'ordonnancement est du type de celle représentée en figure 2.6.
Cependant, il n'est pas possible de conclure à l'optimalité de cet algorithme à partir de cette simple propriété, il est nécessaire d'y ajouter un argument d'équilibrage de charge.



Si P_1 et P_2 sont toujours actifs entre l'exécution de leur première et de leur dernière tâche

FIG. *2.6 - : structure de l'ordonnancement final.*

Proposition 1

L'ordonnancement d'un arbre UECT sur deux processeurs satisfaisant les deux conditions suivantes est optimal:

- *Il n'apparaît aucune période d'inactivité sur* P_1
- *Le nombre de tâches allouées à* P_2 *est égal à* $\left\lfloor \frac{n-2}{2} \right\rfloor$.

Proposition 2

L'algorithme produit des ordonnancements sans périodes d'inactivité sur P_1.

Preuve
Supposons que l'algorithme produise un ordonnancement avec une période d'inactivité sur P_1 et considérons dans un tel cas la première de ces périodes. Soit T_l^i la tâche qui lui succède. Cette situation ne peut se produire que si T_l^i est en attente d'une donnée en provenance de P_2. Supposons en outre qu'il existe au niveau l au moins deux tâches non marquées et notons les T_l^i et T_l^j. Aucune de ces deux tâches n'a été allouée à P_2, leur poids est donc supérieur à la valeur courante de $Reste$, donc : $poids(T_l^i) > Reste$ et $poids(T_l^j) > Reste$.

Examinons maintenant les sous-arbres alloués à P_2 et appartenant au sous-arbre dont la racine est T_l^i ou T_l^j. A la fin de l'exécution de ces sous-arbres, qui sont les premiers exécutés parmi l'ensemble des sous-arbres alloués à P_2, la date est au plus égale à $min(poids(T_l^i) - 1, poids(T_l^j) - 1)$ (sans quoi, au moins l'une des deux tâches eût été allouée à P_2).

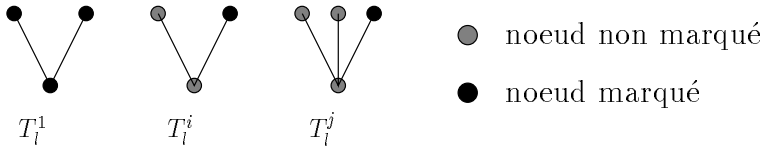


FIG. *2.7 - : Au niveau l, au moins deux tâches ne sont pas marquées.*

Ainsi, la date de fin d'exécution de ces sous-arbres est inférieure au poids du sous-arbre dont aucune tâche n'a été allouée à P_2, par conséquent, il est impossible que la tâche T_l^i se trouvât en attente de données, puisque durant ce laps de temps, P_1 disposait d'au moins $max(poids(T_l^i), poids(T_l^j)) + 1$ tâches à exécuter. Donc cette période d'inactivité n'a pas pu apparaître.

De la même façon, si à ce même niveau seule la tâche T_l^i n'était pas marquée, alors $Reste$ a été mis à jour : $Reste = \left\lfloor \frac{N(T_l^i) - 2}{2} \right\rfloor$. Si T_l^i est en attente de données, et comme il s'agit de la première période d'inactivité, cela signifie que l'activité a été constante jusqu'à ce moment sur P_1. De même, d'après la remarque 2, il n'existe aucune période d'inactivité sur P_2 puisque celui-ci n'est en attente d'aucune donnée. Donc, le nombre de tâches exécutées par P_2 et appartenant au sous-arbre dont T_l^i est la racine (ce sont les premières tâches qui sont exécutées par P_2 puisque ce sont celles qui appartiennent aux sous-arbres dont les racines sont les plus hautes), est strictement supérieure à la valeur de $Reste$ calculée en T_l^i. Ce qui est en contradiction avec le principe de construction de l'algorithme.
Donc, l'ordonnancement produit des algorithmes sans périodes d'inactivité sur P_1. □

Proposition 3

Lorsque le nombre de tâches allouées à P_2 est inférieur à $\left\lfloor \frac{n-2}{2} \right\rfloor$, l'ordonnancement obtenu par l'algorithme est encore optimal.

Preuve
Le nombre de tâches allouées à P_2 est strictement inférieur à $\left\lfloor \frac{n-2}{2} \right\rfloor$ si et seulement si $Reste$ a été mis à jour alors qu'aucun nouveau sous-arbre n'avait été alloué à P_2.

Montrons par induction sur la hauteur h de l'arbre l'optimalité de l'ordonnancement obtenu par l'algorithme.
MAJYC est trivialement optimal pour un arbre de hauteur 1.
Supposons qu'il produise des ordonnancements optimaux pour des arbres dont la hauteur est inférieure ou égale à $h - 1$.

Montrons qu'il produit des ordonnancements optimaux pour des arbres de hauteur h. Soit un tel arbre comportant n nœuds. Si le nombre de tâches allouées à P_2 est égal à $\left\lfloor \frac{n-2}{2} \right\rfloor$ alors l'ordonnancement obtenu est optimal. Sinon, il existe un niveau l dans l'arbre ne contenant qu'une seule tâche non marquée (T_l^A) et racine d'un sous-arbre A. Dans un tel cas, les tâches appartenant à un niveau $l' < l$ sont toutes marquées sauf une, et, entre les niveaux 1 et l, l'ensemble des tâches non marquées forment une chaîne (voir la figure 2.8) de longueur l.

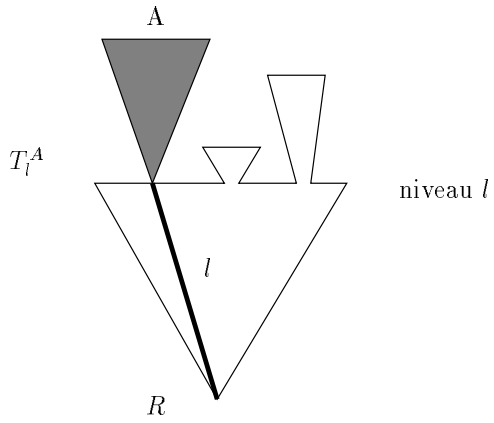


FIG. *2.8 - : Situation de mise à jour de Reste.*

Examinons la date de fin d'exécution obtenue par l'algorithme. Ce temps est égal au nombre de tâches allouées à P_1 puisqu'aucune période d'inactivité n'apparaît sur ce processeur (d'après la Proposition 2). Entre les niveaux 1 et l P_1 n'exécute que la chaîne formée des tâches non marquées, donc : $T_{algo}(R) = l + T_{algo}(T_l^A)$. Or par hypothèse d'induction $T_{algo}(T_l^A) = T_{opt}(T_l^A)$. Donc $T_{algo}(R) = l + T_{opt}(T_l^A)$, or, d'après la Remarque 3, l'ordonnancement optimal de la chaîne ne peut être obtenu que par l'allocation de la totalité de cette chaîne sur un unique processeur, donc $T_{opt}(R) = l + T_{opt}(T_l^A)$.

Ainsi, lorsque le nombre de tâches allouées à P_2 est inférieur à $\left\lfloor \frac{n-2}{2} \right\rfloor$, l'ordonnancement obtenu par l'algorithme est encore optimal, et donc l'ordonnancement produit par MAJYC est optimal. □

2.2 $P_2 \mid arbre,\ p_i,\ c = 1 \mid C_{max}$

2.2.1 Introduction

Il n'existe pas beaucoup de résultats concernant l'ordonnancement d'arbres formés de tâches dont les durées d'exécution sont différentes, bien que ce problème soit assez important en pratique. Les seuls résultats connus concernant des arbres sans communication sont des études pour des ensembles de durées d'exécution relativement restreints. Ainsi, Du et Leung ont prouvé la NP-complétude de tels problèmes pour un nombre non fixé de processeurs ou pour un système formé de deux processeurs mais dont les durées des tâches appartiennent à l'ensemble des puissances d'un entier fixé $k > 1$ [DL88]. Nakajima, Leung et Hakimi présentent un algorithme de liste basé sur un critère "plus gros sous-arbre d'abord" (*LSF* pour *Largest Subtree First*), pour résoudre le problème $P_2 \mid arbre,\ p_i \in \{1,2\} \mid C_{max}$ [NLH81]. La complexité de cet algorithme est $O(nlog(n))$, et l'optimalité est obtenue après réordonnancement des cas que les auteurs nomment **cas dégénérés**, caractérisés par l'exécution de trois tâches de durées deux dans les quatre dernières unités de temps (les arbres considérés sont des anti-arbres). C'est par examen d'un ensemble de cas du même type que Du et Leung construisent, pour un ensemble de durées d'exécution restreint à $\{1,3\}$, un algorithme optimal de complexité $O(n^2log(n))$ [DL89].
Lorsque des communications sont introduites, la plupart des travaux connus mettent en œuvre des algorithmes généraux. Ainsi, pour un nombre non limité de processeurs ($P_\infty \mid prec,\ p_i,\ c_{ij} \mid C_{max}$ avec $max\{c_{ij}\} \leq min\{p_i\}$), un ordonnancement optimal avec duplication de tâches est construit par l'algorithme présenté dans [Chr89]. Dans le cadre d'une factorisation LU, Pozo [Poz92] utilise une méthode multifrontale. L'arbre d'élimination exploité est ordonnancé à l'aide d'un algorithme de regroupement de tâches proche de DSC [YG92b].

Auteurs	Complexité	Notation du problème
[NLH81]	$O(nlog(n))$	$P_2 \mid arbre, p_i \in \{1,2\} \mid Cmax$
[DL89]	$O(n^2log(n))$	$P_2 \mid arbre, p_i \in \{1,3\} \mid Cmax$
[DL89]	Polynomial	$P_2 \mid arbre, p_i \in \{1,k\} \mid Cmax$
	OUVERT	$P_m \mid arbre, p_i \in \{1,k\} \mid Cmax$
[DL88]	NP	$P \mid arbre, p_i \in \{1,k\} \mid Cmax$
[DL88]	NP	$P_2 \mid arbre, p_i \in \{k^l, l \geq 0\} \mid Cmax$
[Chr89]	$O(n^2)$	$P_\infty \mid arbre, p_i, c_{ij} \mid Cmax\ (Max\{c_{ij}\} \leq min\{p_i\})$

Dans une première partie nous examinons le problème $P_2 \mid arbre,\ p_i \in \{1,2\},\ c = 1 \mid C_{max}$. Pour un ensemble de tâches dont les durées d'exécution appartiennent à l'ensemble $\{1,2,...,k\}$ (k est un entier fixé supérieur à 2 et indépendant de la taille de l'arbre), nous donnons, dans la seconde partie, une borne pour le même algorithme de résolution.
Les notations utilisées précédemment sont conservées, n représente toujours le nombre de nœuds d'un arbre, et $Poids(T)$ représente la somme des durées d'exécution des tâches appartenant au sous-arbre dont T est la racine. Le poids d'une tâche T prise isolément est noté $Exec(T)$.

2.2.2 $P_2 \mid arbre,\ p_i \in \{1,2\},\ c = 1 \mid C_{max}$

L'analyse de ce problème commence par une étude de la qualité d'un ordonnancement réalisé par l'algorithme MAJYC : nous montrons que, moyennant une modification mineure, les ordonnancements produits sont proches de l'optimal (à une unité de temps près). Nous étudions ensuite la possibilité de produire des ordonnancements optimaux, par une analyse apparentée à celle décrite dans [NLH81].

Qualité de l'ordonnancement réalisé par l'algorithme MAJYC

L'algorithme MAJYC conçu pour le cas *UECT* peut être appliqué avec une très légère modification pour le premier calcul de *Reste* et pour le choix terminal.
Le premier calcul de *Reste* doit tenir compte de la durée d'exécution de la racine, et donc la première fois, $Reste = \left\lfloor \frac{N_{total} - Exec(racine) - 1}{2} \right\rfloor$, où N_{total} représente la somme des durées d'exécution des nœuds de l'arbre.
L'autre modification concerne le choix terminal, en effet, lorsque les sous-arbres sont choisis par MAJYC, nulle part il n'est fait mention de la durée des tâches composant ceux-ci. Seule la dernière étape prend en compte, implicitement, le caractère unitaire des durées d'exécution. Donc, la première partie effectuant le choix des sous-arbres n'est pas modifiée. Par contre, dès qu'à un niveau donné tous les nœuds ont un poids supérieur (strictement) à la valeur de *Reste*, la stratégie de recherche change. On examine toujours le sous-arbre de plus faible poids, mais la condition d'arrêt de la recherche va être modifiée. La condition d'arrêt dans le cas *UECT* est $Reste = 0$. Pour des arbres dont les tâches ont des durées qui appartiennent à l'ensemble $\{1,2\}$, cette condition devient : $Reste = 0$ ou $Reste = 1$. Si, lors de la recherche dans ce sous-arbre un ensemble de sous-arbres dont le poids est égal à *Reste* est déterminé, l'ordonnancement produit est optimal puisque la valeur finale de *Reste* est nulle. Par contre, si, à la fin de l'examen de ce même sous-arbre la valeur de *Reste* est égale à 1, alors l'ordonnancement produit est à 1 de l'optimal (voir pour illustration la figure 2.9).

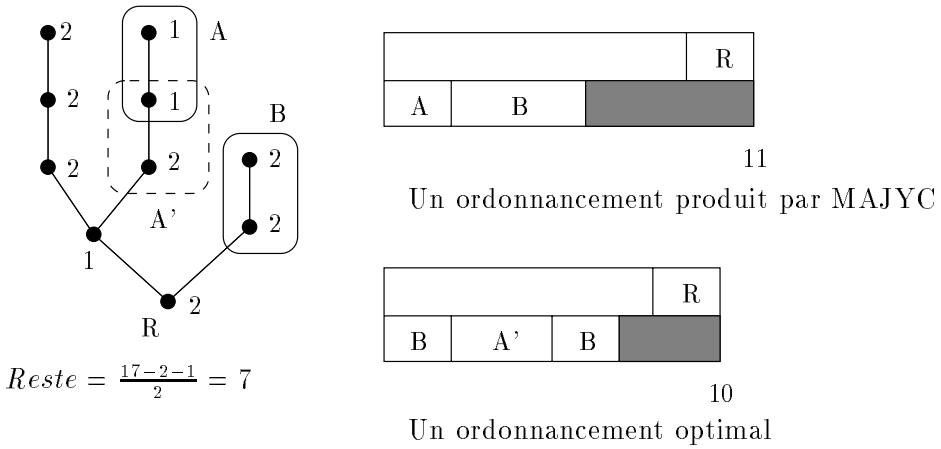


FIG. *2.9 - : Exemple d'arbre pour lequel l'ordonnancement produit par MAJYC n'est pas optimal.*

D'autre part, il n'est pas possible que la valeur de *Reste* soit supérieure ou égale à deux à la fin de l'examen du sous-arbre, sinon une tâche de poids égal à 1 ou 2 aurait été délaissée dans le sous-arbre alors qu'elle était racine d'un sous-arbre de poids inférieur à la valeur courante de *Reste*.

Théorème
Pour le problème $P_2 \mid p \in \{1,2\},\ c = 1,\ intree \mid C_{max}$, *l'algorithme MAJYC produit des ordonnancements tels que :*

$$\omega_{MAJYC} \leq \omega_{opt} + 1$$

Preuve
Elle est directement issue des remarques précédentes. Lors de l'examen du dernier sous-arbre, soit $Reste = 0$, dans ce cas, l'algorithme a trouvé un ensemble de sous-arbres dont la somme des poids est égale à *Reste*, et l'ordonnancement est optimal, soit la valeur terminale de *Reste* est égale à 1 auquel cas, par rapport au nombre de tâches originellement calculé, une tâche supplémentaire aurait peut être pu être allouée à P_2.

Remarque
Il est également possible d'utiliser l'algorithme MAJYC sans modification, à condition qu'il soit précédé par une étape supplémentaire de traitement des noeuds de l'arbre. Toute les tâches qui ne sont pas unitaires peuvent être transformée en chaînes de sous-tâches unitaires avec une étiquette de parenté. L'algorithme est appliqué directement sur cet arbre formé uniquement de tâches et de sous-tâches unitaires. Le résultat fournit par l'algorithme est ensuite légèrement retouché. Si une sous-tâche est allouée sans sa parente elle est alors libérée. Cette situation ne pouvant survenir qu'à une seule occasion, l'ordonnancement obtenu est distant de 1 de l'ordonnancement optimal.

Recherche d'un ordonnancement optimal

Si T est racine d'un sous-arbre, la somme des durées d'exécution des nœuds appartenant à ce sous-arbre sera notée $SDE(T)$. Une telle somme sera parfois appelée **Poids**. La somme des durées d'exécution de l'arbre entier est notée N_{total}.

Dans le cas *UECT*, quel que soit l'arbre, il existe toujours un ordonnancement optimal tel que :
il existe une date T avant laquelle les deux processeurs sont en constante activité, et après laquelle seul le processeur exécutant la racine est actif.

Pour le cas qui nous occupe, ce n'est plus vrai (voir figure 2.10).

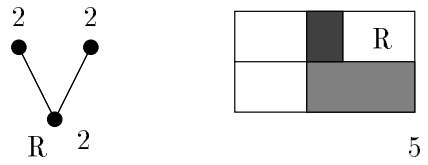


FIG. 2.10 - : Il n'est pas toujours possible de construire un ordonnancement optimal pour lequel l'un des processeurs présente une activité constante.

Un algorithme optimal devrait donc être capable dans certains cas d'allouer une charge de calcul égale pour les deux processeurs (racine exceptée). Considérons un nouvel algorithme basé sur le principe de MAJYC, mais pour lequel *Reste* n'est plus un entier mais un intervalle d'entiers, dont les bornes, notées N_{min} et N_{max} sont calculées de la façon suivante :

$$N_{min} = \left\lfloor \frac{N_{total} - Exec(R) - 1}{2} \right\rfloor$$
$$N_{max} = N_{total} - Exec(R) - N_{min}$$

Appelons ce nouvel algorithme **MAJYC-intervalle**. Il s'applique comme le précédent, et s'arrête lorsque l'une des deux valeurs est nulle. Ainsi, dans le cas présenté en figure 2.10, à la fin de l'allocation, $Reste = [-1, 0]$.

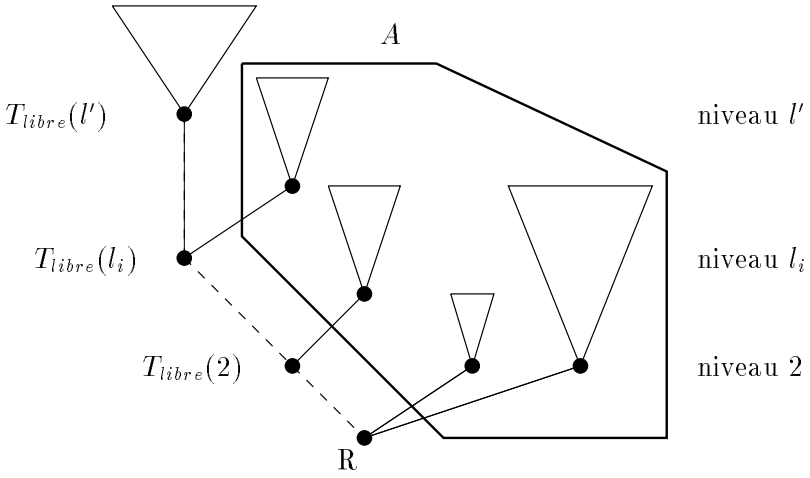
La stratégie de recherche des sous-arbres mise en œuvre dans MAJYC implique que l'ordonnancement d'un arbre soit traité comme un seul ordonnancement. En effet, toutes les étapes de choix des sous-arbres et de remise à jour de la valeur de *Reste* se rapportent à une allocation pour un seul processeur (P_2 en l'occurrence, c'est-à-dire le processeur qui n'exécute pas la racine de l'arbre). Nous allons maintenant considérer que l'ordonnancement d'un arbre est constitué de **sous-ordonnancements** successifs liés entre eux.

Un sous-ordonnancement est une allocation de tâches aux processeurs pour une partie connexe de l'arbre. Le premier sous-ordonnancement comprend la racine, et s'achève lorsqu'une condition est vérifiée. Un nouveau sous-ordonnancement prend le relais. Il débute lui aussi en la racine d'un sous-arbre. Les sous-ordonnancements sont déterminés lors de l'application de l'algorithme. Le premier contient la racine de l'arbre et toutes les tâches qui appartiennent à des sous-arbres dont le niveau des racines est inférieur ou égal à l', premier niveau pour lequel il reste un seul nœud non marqué et tel que la longueur de la chaîne de ce nœud à la racine soit supérieure ou égale à la somme des tâches marquées appartenant à ce sous-ordonnancement. A ce niveau, le sous-ordonnancement est achevé et un nouveau sous-ordonnancement est entamé (voir la figure 2.11).

La stratégie de recherche des sous-arbres, pour un sous-ordonnancement, reste la même que celle mise en œuvre dans MAJYC. Un nouveau sous-ordonnancement est commencé au niveau l' lorsque deux conditions sont réunies. On se place (sans perte de généralité) dans le

cas où l'arité de la racine est supérieure ou égale à deux.

Condition 1
Au niveau l' la borne $N_{min} > 0$ et entre les niveaux 2 et l' il ne subsiste qu'un seul nœud non marqué par niveau, que l'on note $T_{libre}(niveau)$ (le dernier étant $T_{libre}(l')$, comme indiqué en figure 2.11).



$A = \{noeuds\ marqués \mid niveau(noeud) \geq 2\}$
$L =$ longueur de la chaine $Ch\ (T_{libre}(2), T_{libre}(l'))$
$L(l') \geq Poids(A)$

FIG. *2.11 - : Commencement d'un nouveau sous-ordonnancement.*

Condition 2
Si on note $S_{ssam}(l_i)$ la somme des poids des sous-arbres marqués (dont les racines appartiennent toutes aux niveaux compris entre 2 et l_i ($l_i < l'$)), et $L(l_i)$ la longueur de la chaîne liant $T_{libre}(2)$ à $T_{libre}(l_i)$ (c'est-à-dire la somme des temps d'exécution des nœuds qui la composent), alors $L(l_i) \leq S_{ssam}(l_i)$, et $L(l') \geq S_{ssam}(l')$.

Dans un souci de clarté et lorsqu'il n'y a aucune ambiguïté $T_{libre}(l')$ sera noté T_{libre}. Dans la suite la racine de l'arbre est supposée d'arité supérieure ou égale à deux. Le dernier sous-

ordonnancement déterminé par l'algorithme est appelé sous-ordonnancement **terminal**.

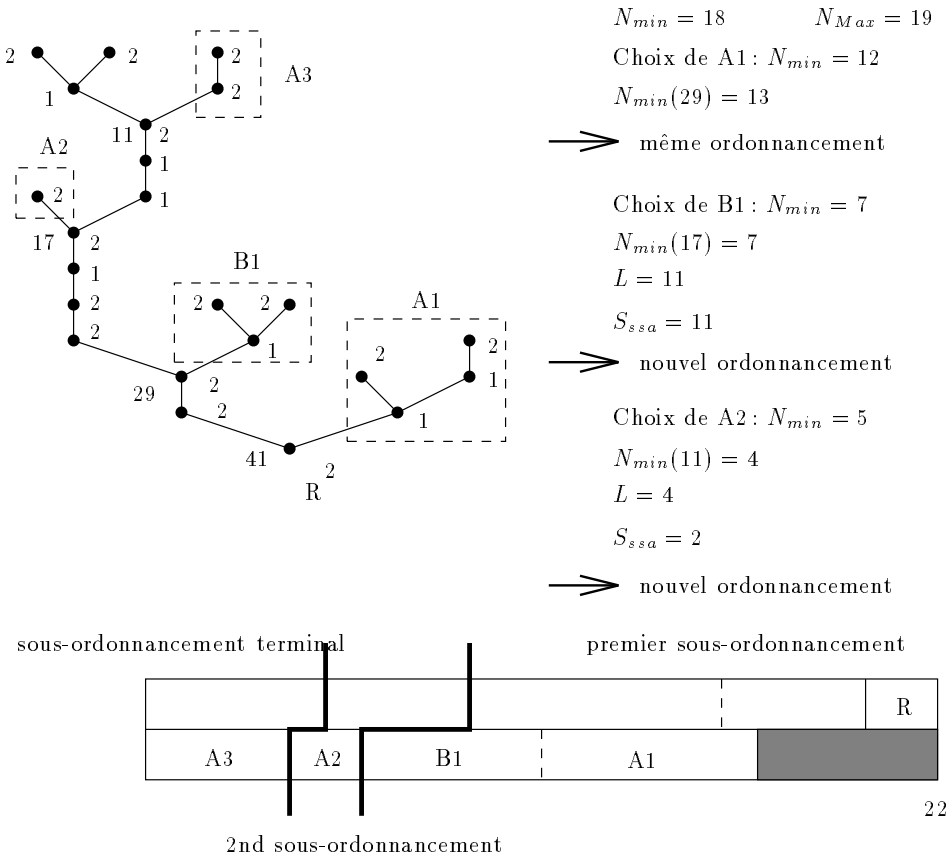


FIG. *2.12 - : Ensemble des sous-ordonnancements d'un arbre.*

Lemme 1

Considérons un ordonnancement formé de sous-ordonnancements successifs. Si le sous-ordonnancement terminal est optimal alors un ordonnancement optimal pour l'arbre entier est obtenu en allouant les chaînes des sous-ordonnancements successifs sur un seul processeur et les sous-arbres marqués sur l'autre.

Preuve

Examinons la forme d'un sous-ordonnancement terminal optimal. Celui-ci est déterminé pour un sous-arbre dont nous notons la racine T_{libre}. Quel que soit l'ordonnancement de ce sous-arbre, si T_{libre} est alloué à P_1, il y a au moins deux unités de temps entre la date de fin d'exécution de la dernière tâche sur P_2 et la date de fin d'exécution de T_{libre} sur P_1 (voir figure 2.13). De plus, par hypothèse, au niveau de T_{libre} un nouveau sous-ordonnancement est commencé, donc, en particulier, la condition 2 est vérifiée, c'est-à-dire que si l' est le niveau de T_{libre} : $L(l') \geq S_{ssam}(l')$, et donc dans tous les cas les ajouts, de la chaîne à la suite de T_{libre} sur P_1, et des sous-arbres marqués sur P_2, garantissent la conservation de l'optimalité, en vertu de la propriété que l'ordonnancement d'une chaîne en présence de communications

est optimal si et seulement si cette chaîne est entièrement allouée à un unique processeur. □

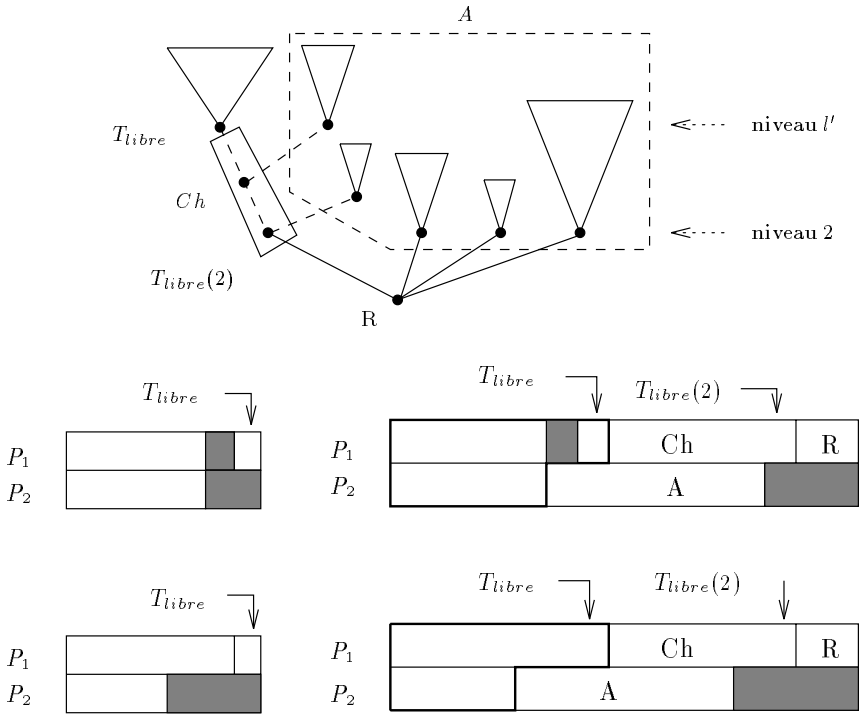


FIG. *2.13 - : Raccordement entre sous-ordonnancements.*

L'optimalité repose donc sur la qualité du sous-ordonnancement terminal. Nous considérons à présent seulement les arbres dont l'ordonnancement n'est formé que d'un sous-ordonnancement terminal. Pour ces arbres, le processus de choix des sous-arbres reste identique. Lors de l'application de l'algorithme, on considère la tâche T_{libre} définie par :

- T_{libre} admet au moins un prédécesseur immédiat alloué à P_2,
- il n'existe aucune autre tâche non marquée au niveau de T_{libre} (noté l') à part elle-même,
- T_{libre} est la tâche de plus haut niveau dans l'arbre admettant les deux précédentes propriétés.

L'ensemble des tâches entre T_{libre} et la racine R (non comprise) est une chaîne que l'on appelle Ch. On note L la longueur de cette chaîne (en terme de somme des temps d'exécution des tâches qui la compose). On note A l'ensemble des tâches appartenant aux sous-arbres marqués dont les racines ont un successeur immédiat appartenant à Ch, et S_{ssam}, la somme des durées d'exécution des tâches appartenant à ces sous-arbres.
Deux cas peuvent survenir, soit $L < S_{ssam} - 1$, soit $L = S_{ssam} - 1$. En effet, si $L > S_{ssam} - 1$, alors en T_{libre} un nouveau sous-ordonnancement aurait été commencé. Dans le premier cas, l'algorithme produit un ordonnancement optimal comme l'illustrent les différents cas de la

figure 2.14. Soit il existe, au niveau $l'+1$, au moins deux tâches dont le poids est supérieur strictement à N_{max}, soit l'étape d'allocation a été achevée au niveau l'.

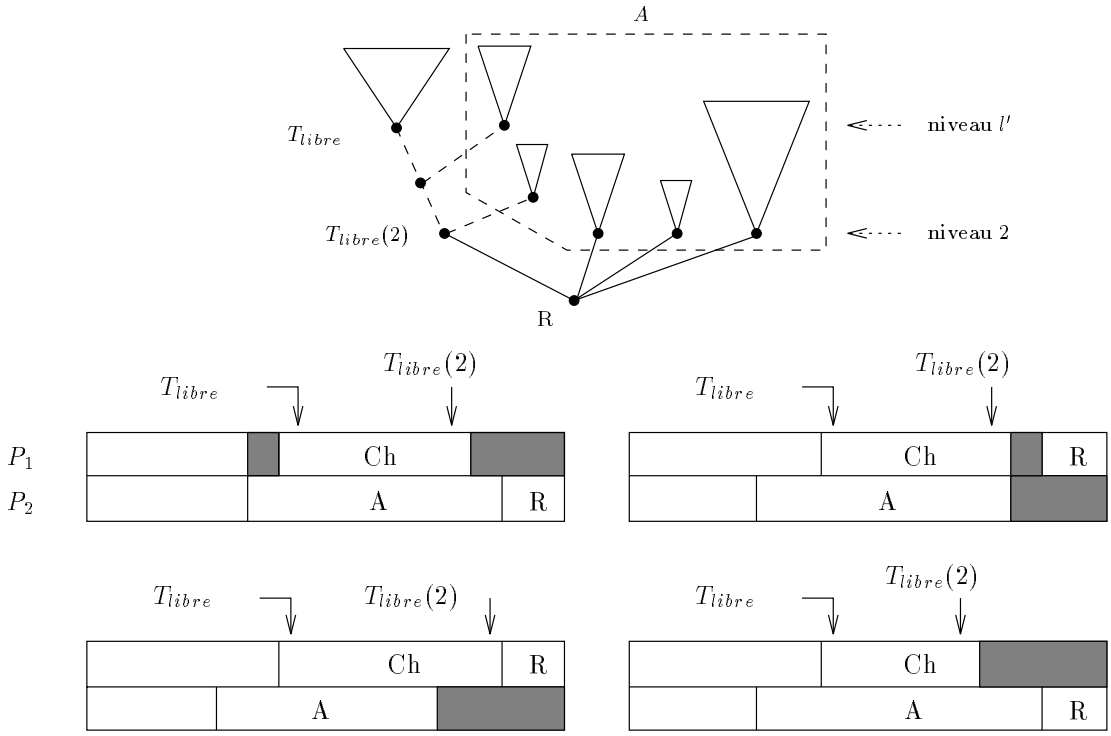


FIG. *2.14 - : Lorsque $L < S_{ssam} - 1$, dans tous les cas, l'ordonnancement produit est optimal.*

Dans le second cas ($L = S_{ssam} - 1$), tout ne se passe pas aussi bien. On conserve les mêmes notations pour T_{libre}, l', Ch et L. Si au niveau $l'+1$ il existe au moins deux tâches dont le poids est supérieur strictement à N_{max}, alors l'ordonnancement produit est optimal car aucune période d'inactivité n'apparaît sur P_1. En revanche, un problème peut survenir lorsque les tâches dont T_{libre} est la racine sont réparties en deux ensembles de même durée d'exécution alloué chacun à un processeur comme le montre la figure 2.15.

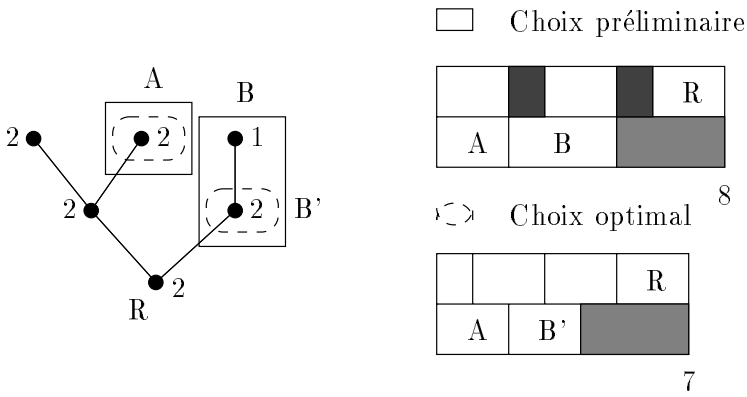


FIG. *2.15 - : $L = S_{ssam} - 1$, et équirépartition des tâches des niveaux supérieurs.*

Un arbre générique suffit pour représenter l'ensemble des cas pour lesquels ce comportement *pathologique* survient (figure 2.16). Si un tel ordonnancement n'est pas optimal il est possible d'en construire un autre tel que les périodes d'inactivité sur P_1 disparaissent et tel que la date de fin d'exécution de la racine soit avancée d'une unité de temps. Nous allons exposer maintenant un ensemble de manipulations sur l'ordonnancement qui permettent de réaliser l'élimination des périodes d'inactivité. On remarque tout d'abord que la première période d'inactivité (celle dont la date est la plus petite) joue un rôle prépondérant. En effet, l'élimination de la seconde n'a pas d'incidence sur la date de fin d'exécution, alors que la disparition de la première permet la désynchronisation des fins d'exécution de Ch et de C' et entraîne donc la disparition de la seconde période.

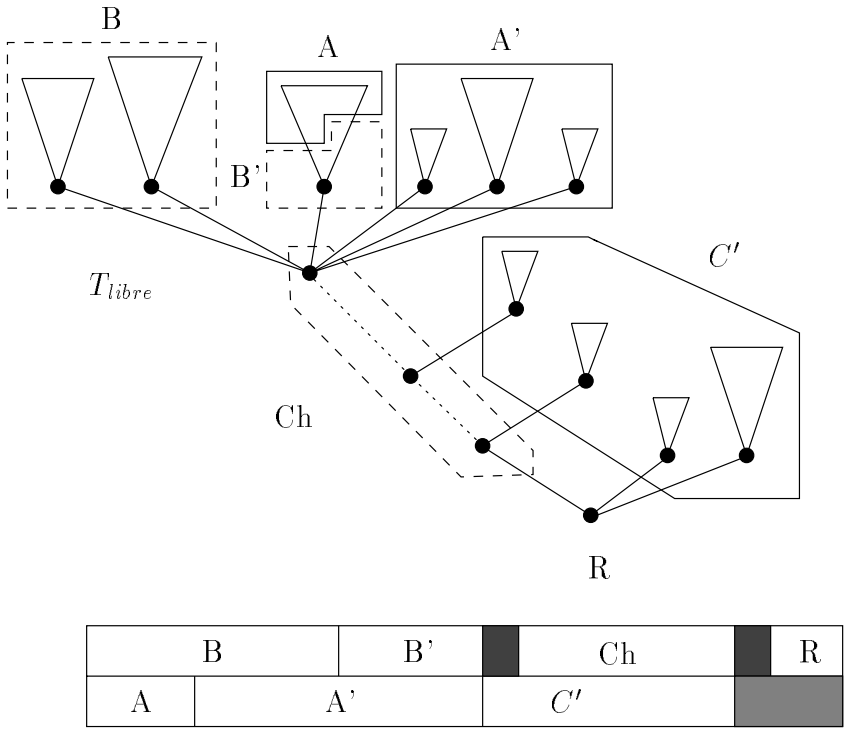


FIG. *2.16 - : Cas pathologique pour lequel l'algorithme ne produit pas l'ordonnancement optimal.*

Les notations de la figure 2.16 sont conservées pour toutes les figures qui suivent.

On remarque tout d'abord que les dates de début d'exécution des tâches de Ch ne peuvent pas être avancée quel que soit l'ordonnancement. En revanche, la date de fin d'exécution de C' peut parfois être avancée d'une unité de temps. Ceci permet d'éliminer la seconde période d'inactivité. Cependant, ceci ne peut être réaliser qu'à la faveur du transfert d'une tâche unitaire de P_2 vers P_1, et cette tâche (appartenant à C') ne peut être autre chose qu'une feuille. En effet, le transfert d'une tâche unitaire qui ne serait pas une feuille entraînerait inévitablement un déséquilibre de la charge entre les processeurs lors de l'exécution de Ch et ne pourrait qu'augmenter la date de fin d'exécution totale. En gardant les notations de la figure 2.16, la figure 2.17 illustre un tel transfert. La date de fin d'exécution de la dernière

tâche exécutée par B ou B' (s'il est non vide) est retardée d'une unité de temps et la feuille de C' s'insère dans cet intervalle. Les dates d'exécution des tâches de C' sont toutes avancées d'une unité de temps. On remarque que pour ce cas, le sous-arbre formé par A et B' peut être vide.

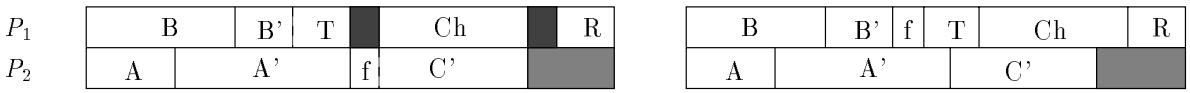


FIG. *2.17 - : Migration d'une feuille unitaire appartenant à C'.*

Si C' ne contient aucune feuille unitaire, seul doit être considéré le sous-arbre dont T_{libre} est racine. Nous nous limitons donc dans la suite à l'étude des transformations de ce sous-arbre. Nous allons montrer comment transformer cet ordonnancement, pour lequel $Exec(A) + Exec(A') = Exec(B) + Exec(B')$, en un autre ordonnancement pour lequel $|Exec(T_1) - Exec(T_2)| = 2$ (où T_1 représente l'ensemble des tâches exécutées par P_1 et appartenant au sous-arbre dont T_{libre} est racine et $|i|$ représente la valeur absolue de i) et la date de fin d'exécution de T_{libre} reste la même. L'ensemble des manipulations présentées suit la localisation de la première tâche unitaire rencontrée. Nous distinguons donc quatre cas, selon que cette tâche appartienne à A, A', B ou B'.

Remarques préliminaires

Lorsqu'une telle transformation est possible, C' est alloué en totalité au processeur dont la date de fin d'exécution est la plus faible et Ch est alloué en totalité à l'autre processeur.
On suppose que l'ensemble des racines des sous-arbres appartenant à A', B et B' ne sont pas des tâches unitaires. Si tel n'est pas le cas, la transformation nécessaire est la migration de l'une de ces racines.
La somme des temps d'exécution des tâches appartenant à A, B, A' et B' vérifient : $Exec(B) \geq Exec(A) + Exec(B')$ et $Exec(A') > Exec(B')$ par construction de l'ordonnancement.
Si A' et B contiennent plusieurs sous-arbres, on suppose que l'ordonnancement de ces sous-arbres sur chaque processeur privilégie l'exécution d'une tâche unitaire le plus tôt possible.
Si l'on note t la première tâche rencontrée dans l'ordonnancement, alors, sans changer l'ordre d'exécution des ensembles de tâches B puis B' sur P_1 et A puis A' sur P_2, aucune autre tâche unitaire n'aurait pu être exécutée avant t.

- **Si $t \in A$**
  Soit E l'ensemble des tâches qui précèdent t dans l'ordonnancement, la solution consiste à échanger l'ensemble de tâches E augmenté de t avec l'ensemble E' des tâches qui sont exécutées en même temps que E et qui appartiennent à B. Une feuille dont l'exécution nécessite deux unités (noté F) de temps et appartenant à A' est exécutée plus tôt afin de recouvrir la communication comme l'illustre la figure 2.18. Le nombre d'unités de temps séparant la date de fin d'exécution de A et la date de début d'exécution de B' reste le même qu'avant la transformation, il n'y a donc aucun problème de

recouvrement pour cette communication.

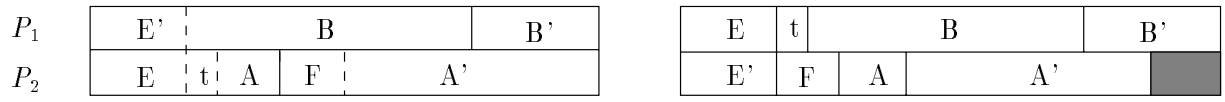


FIG. *2.18 - : La première tâche unitaire appartient à A.*

– **Si $t \in A'$ et $A = \emptyset$**
Si A' contient une feuille qui ne soit pas un prédécesseur de t, alors une transformation du même type que la précédente permet d'éliminer la période d'inactivité avant l'exécution de T_{libre} (scenario représenté en figure 2.19). Un autre scenario peut se produire. Si A' ne contient aucune feuille qui ne soit un prédécesseur de t, alors une transformation est encore faisable si pour l'ensemble E il est possible de déterminer, parmi les tâches exécutées sur P_1, un ensemble E' et une feuille F de durée d'exécution égale à deux tels que: la durée d'exécution de E' soit égale à la durée d'exécution de E plus deux unités de temps et que F ne précède aucune tâche appartenant à E. La transformation est illustrée en figure 2.19.

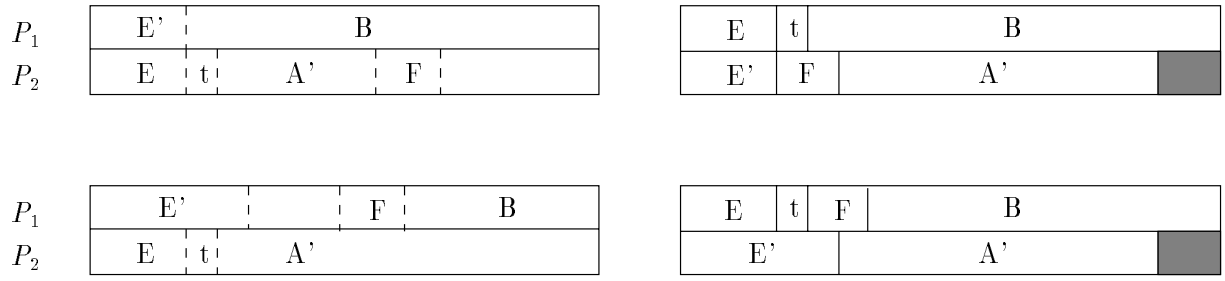


FIG. *2.19 - : La première tâche unitaire appartient à A' et A est vide.*

Si toutes les feuilles de B appartiennent nécessairement à E', et toutes les feuilles de A' précèdent t alors tout ordonnancement optimal du sous-arbre dont T_{libre} est racine a la forme de celui de départ. En effet, à partir de t, P_1 et P_2 exécutent une chaîne. Ainsi, tout transfert de tâches entraîne une communication non recouverte.

– **Si $t \in A'$ et $A \neq \emptyset$**
On effectue une permutation des ensembles de tâches. A est alloué à P_1 et est exécuté avant B. L'exécution de A' est avancée et B' est alloué à P_2. Cependant, la somme des durées d'exécution des tâches appartenant à A n'est, en général, pas égale à celle des tâches appartenant à B'. Donc, avant la permutation, des tâches de A sont ajoutées à B', si $Exec(A) > Exec(B')$, et l'inverse si $Exec(A) < Exec(B')$. Si l'on note nA le nouvel ensemble A et nB' le nouvel ensemble B', $Exec(nA) = Exec(B')$ et $Exec(nB') = Exec(A)$. Si ces égalités ne sont pas possibles, alors soit $Exec(nA) = Exec(B') + 1$,

soit, $Exec(nB') = Exec(A) + 1$. Dans les deux cas, la transformation est terminée (ces transformations sont illustrées en figure 2.20).

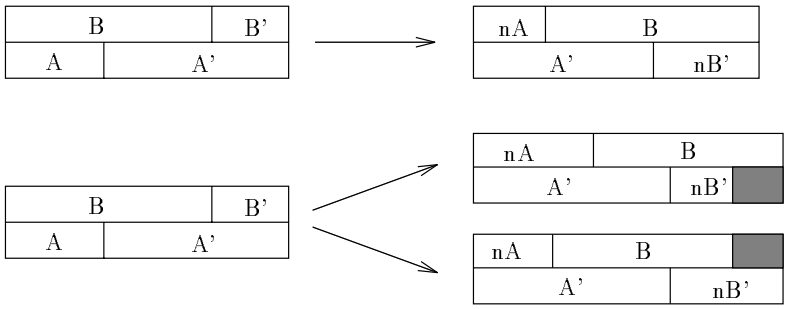


FIG. *2.20 - : Équivalence de A' et B.*

Si $Exec(nA) = Exec(B')$ et $Exec(nB') = Exec(A)$ alors, les ensembles E et E' sont définis comme précédemment. On peut supposer sans perte de généralité que l'ensemble E est de poids strictement supérieur à A. Si ce n'est pas le cas, cela signifie que t est une feuille, la migration de cette feuille de P_2 vers P_1 et son exécution à la date 0 constituent alors une transformation qui répond à notre attente. Le cas où t appartient à nA a été traité précédemment. Les ensembles E augmenté de t et E' sont échangés, et la première tâche de nB' est exécutée à la suite des tâches de E'. La communication de t vers A' est recouverte par T et la communication de E' vers B est recouverte par t (voir la figure 2.21).

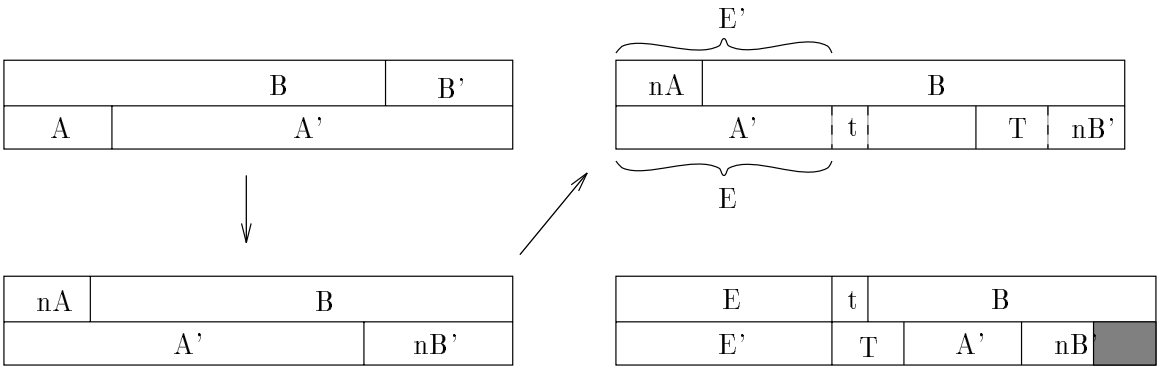


FIG. *2.21 - : La première tâche unitaire appartient à A' et $A \neq \emptyset$.*

- **Si $t \in B$**
  Si $A = \emptyset$, les rôles de B et A' sont rigoureusement équivalents. Si A n'est pas vide, alors, une permutation des ensembles de tâches comme celle présentée en figure 2.20 nous permet de retomber dans les cas ci-dessus traités.

- **Enfin, si $t \in B'$**
  L'ensemble E est constitué des premières tâches exécutées sur P_2 et appartenant à A'.

L'ensemble E' est constitué des premières tâches exécutées appartenant à B' jusqu'à t. Les ensembles vérifient : $Exec(E) = Exec(E') + 2$. E est déplacé sur P_2 et exécuté à partir de la date 0, et E' augmenté de t remplace les tâches de E sur P_2. Comme A n'est pas vide, la communication de E vers A' est recouverte. Le résultat de cette transformation est illustré en figure 2.22.

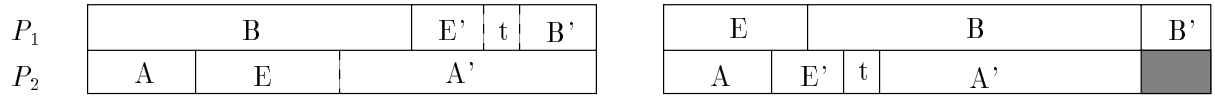


FIG. *2.22 - : La première tâche unitaire appartient à B'.*

En conclusion, les transformations présentées ci-dessus permettent, lorsque cela est possible, de transformer un ordonnancement du sous-arbre dont T_{libre} est racine admettant une période d'inactivité sur P_1 en un nouvel ordonnancement dans lequel cette période d'inactivité est éliminée. Les seuls cas pour lesquels de telles transformations ne sont pas possibles sont :

- lorsqu'il n'y a aucune tâche unitaire dans le sous-arbre, ou,
- lorsque A est vide et au moment de l'exécution de t toutes les feuilles ont été exécutées.

Un exemple pour chacun de ces arbres est représenté en figure 2.23. Dans la suite l'algorithme qui produit ces transformations sera appelé **REARRANGE**.

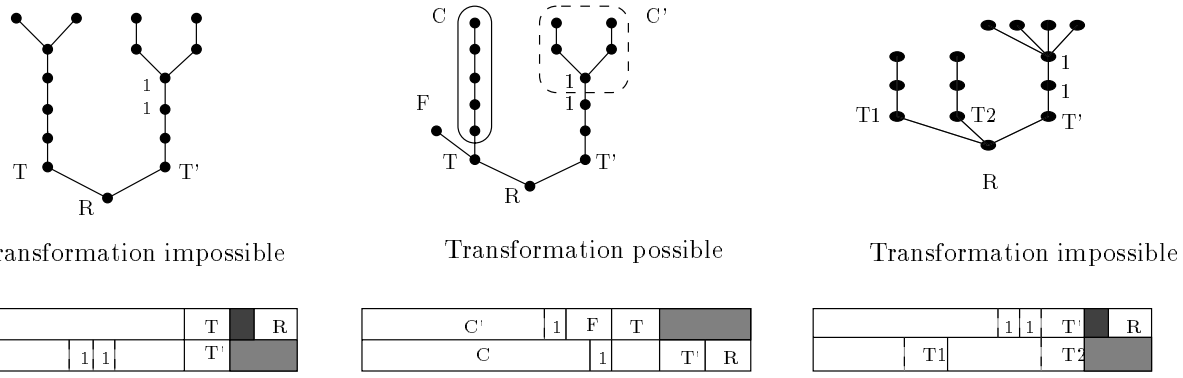


FIG. *2.23 - : Cas pathologiques.*

Exemple :

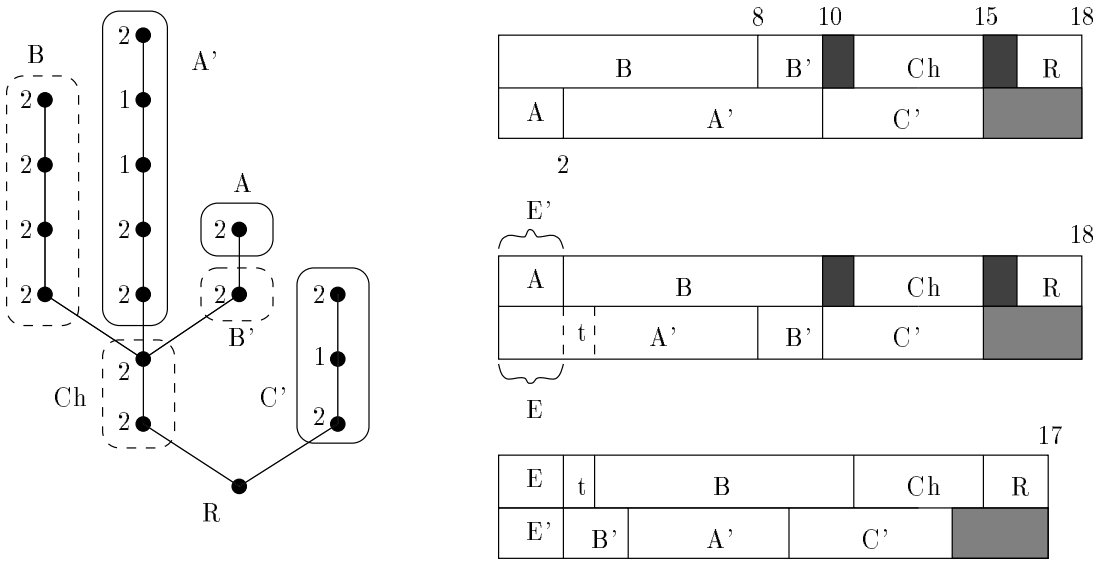


FIG. *2.24 - : $L = S_{ssam} - 1$, et équirépartition des tâches des niveaux supérieurs.*

2.2.3 $P_2 \mid arbre,\ p_i \in \{1, 2, ..., k\},\ c = 1 \mid C_{max}$

Du et Leung conjecturent qu'il est possible d'obtenir des algorithmes optimaux pour les problèmes $P_2 \mid arbre,\ p_i \in \{1, k\} \mid C_{max}$, par une approche énumérative des cas **dégénérés**, méthode que les auteurs utilisent pour la mise au point d'un algorithme optimal pour le problème $P_2 \mid arbre,\ p_i \in \{1, 3\} \mid C_{max}$. Il semble cependant que la complexité de tels algorithmes augmente très rapidement avec k. Nous proposons une heuristique pour le problème plus général où les durées d'exécution appartiennent à l'ensemble $\{1, 2, ..., k\}$, et où les communications sont unitaires (noté $P_2 \mid arbre,\ p_i \in \{1, 2, ..., k\},\ c = 1 \mid C_{max}$). Cette heuristique, basée sur l'algorithme MAJYC, est appelé **MAJYC-k**. Les propriétés citées pour le cas précédent sont encore valables pour ce cas plus général. L'ordonnancement de l'arbre entier est encore une concaténation de sous-ordonnancements, et les deux conditions qui présidaient au commencement d'un nouveau sous-ordonnancement sont les mêmes. L'algorithme MAJYC est précédé par une transformation des tâches non unitaires. L'arbre résultant ne contient que des tâches unitaires (l'idée en avait été présentée dans la remarque 1 du paragraphe précédent). Ainsi, une tâche de durée d'exécution égale à k est transformée en une chaîne de longueur k, dont toutes les tâches ont une marque de parenté (voir figure

2.25).

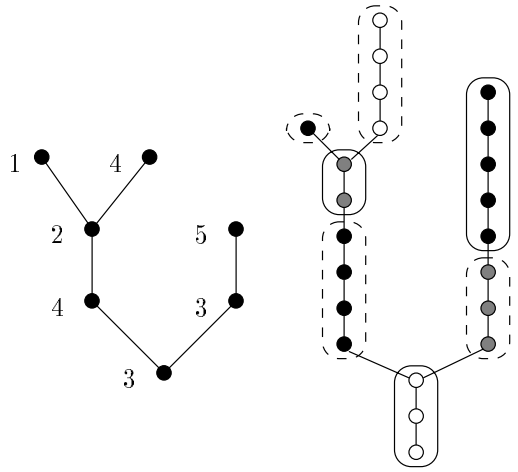


FIG. *2.25 - : Un exemple de transformation des tâches non unitaires.*

La différence entre MAJYC et MAJYC-k se situe au niveau du dernier choix. A la fin de l'ordonnancement, si pour un ensemble de tâches ayant la même marque de parenté il existe un sous-ensemble de tâches choisies non vide ($L_{choisies} \neq \emptyset$) et un sous-ensemble de tâches non choisies non vide également, alors si $Cardinal(L_{choisies}) \geq \left\lceil \frac{k}{2} \right\rceil$, toutes les tâches de même marque de parenté sont choisies également. Cette technique permet de produire des ordonnancements tels que :

$$\omega_{MAJYC-k} \leq \omega_{optimal} + \left\lceil \frac{k}{2} \right\rceil$$

Exemple

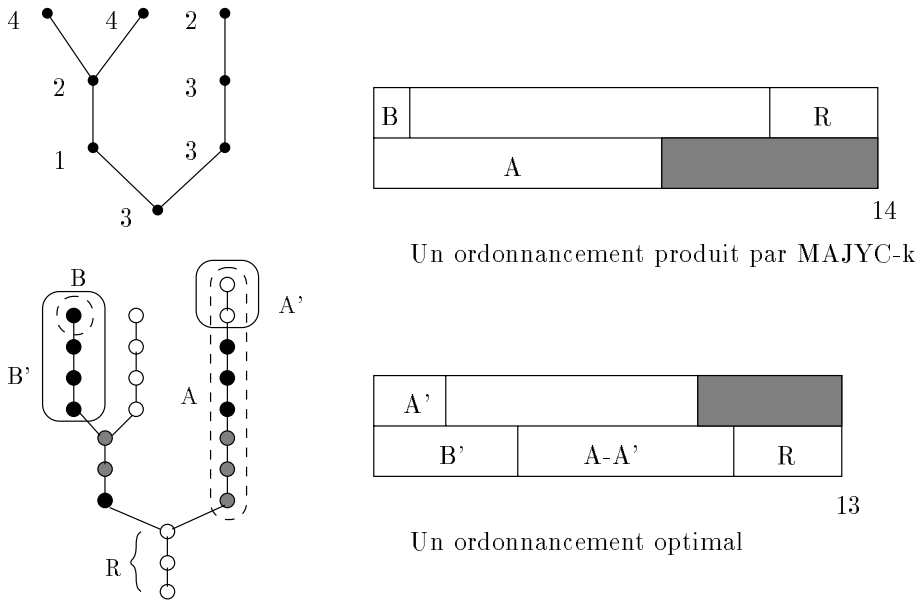


FIG. *2.26 - : Un exemple d'ordonnancement produit par MAJYC-k.*

2.2.4 Conclusion

Pour les problèmes d'ordonnancement d'arbres à gros grain sur deux processeurs en présence de délais de communication, l'algorithme proposé produit des ordonnancements optimaux lorsque les durées des tâches sont restreintes à l'ensemble $\{1, 2\}$, et distants, dans le pire des cas, de $\lceil \frac{k}{2} \rceil$ unités de temps d'un ordonnancement optimal lorsque les durées des tâches sont restreintes à l'ensemble $\{1, 2, ..., k\}$. Il s'agit d'une avancée par rapport au travail de Nakajima et al. [NLH81] puisque les communications sont prises en compte. De plus, leurs travaux ne proposent pas d'heuristiques pour l'ordonnancement d'arbres autres que ceux pour lesquels les tâches ont des durées d'exécution dont les valeurs sont unitaires ou égales à k. On notera d'autre part que l'heuristique présentée dans le dernier paragraphe est assymptotiquement optimale lorsque le nombre de sommets de l'arbre est très grand en rapport de la valeur de k.

2.3 $P_2 \mid arbre,\ p_i = 1,\ c \mid C_{max}$

Alors que le problème précédent est qualifié d'ordonnancement à "gros grain", c'est une granularité fine qui caractérise ce nouveau problème (temps de calcul inférieurs aux temps de communication).
Papadimitriou et Yannakakis prouvent, lorsque la duplication est autorisée et pour un nombre non limité de processeurs, la NP-complétude du problème dans le cas où le graphe est un *DAG* et lorsque τ, le rapport communication/instruction, est de l'ordre de n. Ils proposent un algorithme construisant des ordonnancements proches d'un facteur 2 de l'optimal [PY88]. Pour le même problème, Jung, Spirakis et Kirousis proposent un algorithme basé sur une méthode de programmation dynamique qui produit des ordonnancements optimaux. La complexité de cet algorithme est $O(n^{\tau+1})$ [JKS89]. En s'appuyant sur le principe utilisé dans l'algorithme, les auteurs calculent une borne inférieure de la date de fin d'exécution pour des arbres binaires complets. Pour des anti-arbres, ils mettent en évidence la quantité de calculs dupliqués nécessaire pour atteindre une date de fin d'exécution optimale. Lorsque le graphe est un anti-arbre, Chrétienne présente un algorithme qui permet, en utilisant la duplication et pour un nombre non limité de processeurs, de produire en temps polynomial des ordonnancements optimaux. Lorsque le graphe est un arbre de profondeur au moins 2 (ou un anti-arbre lorsque la duplication n'est plus autorisée), ce même problème devient NP-difficile [Chr89]. Lorsque les tâches sont unitaires et que les durées des communications sont quelconques, Jakoby et Reischuk prouvent que la construction d'un ordonnancement optimal d'un arbre binaire complet est un problème NP-difficile, y compris lorsque le nombre de processeurs n'est pas limité. Ce problème reste de la même difficulté lorsque le graphe est un arbre binaire et que les communications sont uniformes (constante pour un arbre donné, mais dont la valeur dépend de l'arbre). Par contre, lorsque les communications sont égales à une constante quel que soit l'arbre, ils proposent un algorithme qui réalise un ordonnancement optimal pour les arbres k-aires complets [JR92]. Picouleau étudie des problèmes similaires dans lesquels la valeur des communications dépend de la topologie du réseau d'interconnexion. Ainsi, pour une chaîne de longueur infinie, il prouve que l'ordonnancement d'un arbre ou d'un anti-arbre est un problème NP-complet. Pour une topologie décrite par une matrice d'adjacence de taille p, dans laquelle la valeur d'une communication est égale à la distance minimale entre les deux processeurs communicants, déterminer s'il existe un ordonnancement de longueur égale à une valeur donnée est NP-complet [Pic94]. Enfin, l'ordonnancement d'un arbre ou d'un anti-arbre sur une topologie de type étoile est lui aussi

NP-complet. Les principaux résultats sont résumés dans le tableau ci-dessous.

Auteurs	Complexité	Notation du problème	Remarques
[PY88]	NP-complet	$P_\infty \mid dup, prec, p_i = 1, c \mid Cmax$	$c = O(n^2)$
[JKS89]	$O(n^{\tau+1})$	$P_\infty \mid dup, prec, p_i = 1, c \mid Cmax$	
[JKS89]		$P_\infty \mid dup, anti-arbre, p_i = 1, c \mid Cmax$	$Cmax \geq \frac{\tau\ log(n)}{2\ (log(\tau)+log(rec))}$
[JKS89]		$P_\infty \mid arbre, p_i = 1, c \mid Cmax$	$Cmax \geq 2\ \tau\ \frac{log(n)}{log(\tau)}$
[JR92]	NP-complet	$P_\infty \mid arbre, p_i = 1, c_{ij} \mid Cmax$	arbres binaires complets
[JR92]	NP-complet	$P_\infty \mid arbre, p_i = 1, c(n) \mid Cmax$	arbres binaires
[JR92]	polynomial	$P_\infty \mid arbre, p_i = 1, c \mid Cmax$	arbres k-aires complets
[Pic94]	NP-complet	$P \mid prec, p_i = 1, c \in \{1, 2, .., d_{max}\} \mid Cmax$	d_{max}[1]
[Pic94]	NP-complet	P *étoile* $\mid arbre, p_i = 1, c = 2 \mid Cmax$	

Dans la suite, seuls seront considérés les problèmes dont les communications sont constantes. Les tâches sont unitaires, le nombre de nœuds de l'arbre (qui est équivalent à la somme des durées d'exécution des tâches) est noté n . Nous examinons tout d'abord le problème pour $c = 2$, puis nous proposons une heuristique pour c quelconque.

2.3.1 $P_2 \mid arbre,\ p_i = 1,\ c = 2 \mid C_{max}$

Le cas particulier de $c = 2$ est enrichissant parce qu'il permet de mettre en évidence un grand nombre de difficultés inhérentes à une fine granularité, tout en conservant une complexité raisonnable puisqu'il est possible de dériver depuis l'algorithme MAJYC un algorithme optimal pour ce cas particulier. Appelons P_2 le processeur pour lequel les sous-arbres marqués sont choisis, et P_1 le processeur qui exécute la racine de l'arbre.

Qualité de l'ordonnancement réalisé par l'algorithme MAJYC

Dans le cas d'arbres *UECT*, lorsqu'à un niveau donné il existe (au moins) deux nœuds non marqués dont le poids est supérieur à la valeur de *Reste*, alors les tâches restant à allouer peuvent être choisies dans n'importe lequel des sous-arbres ayant l'un de ces nœuds pour racine, et l'ordonnancement produit est optimal. Lorsque les communications sont

strictement supérieures à 1, ce n'est plus vrai (comme le montre la figure 2.27).

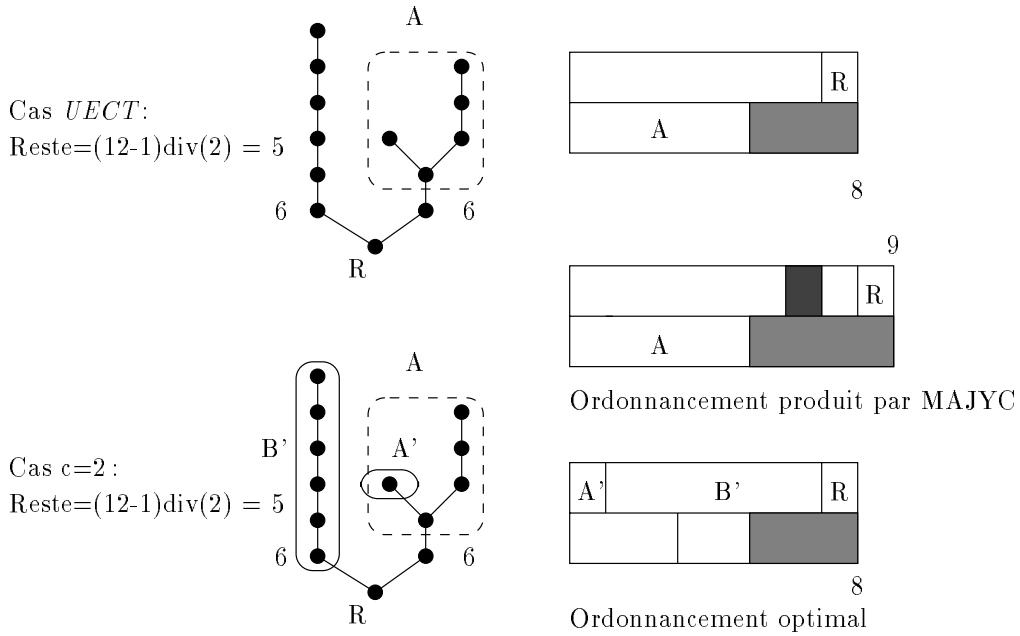


FIG. *2.27 - : Choix des derniers nœuds.*

Dans cette partie, nous allons montrer qu'une version très légèrement modifiée de l'algorithme MAJYC permet d'obtenir des ordonnancements distants d'au plus une unité de temps de l'optimal. La modification concerne le calcul de *Reste*. Le premier calcul de *Reste* doit tenir compte de la durée de calcul nécessaire pour recouvrir complètement une communication. Alors que précédemment une unité de calcul était suffisante il faut maintenant réserver deux tâches pour recouvrir une communication. Donc, $Reste = (n-3)div(2)$.

Appelons **MAJYC-c2** l'algorithme MAJYC pour lequel tous les calculs de *Reste* se font selon l'expression précédente. L'algorithme ne diffère de MAJYC que pour ce calcul. En particulier, les sous-arbres choisis sont toujours marqués, et *Reste* est mis à jour après chaque allocation et lors des situations pour lesquelles un seul nœud non marqué subsiste au niveau courant l (durant l'application de l'algorithme). Pour cette dernière situation, si la valeur de *Reste* calculée est inférieure ou égale à la valeur courante, un ordonnancement optimal du sous-arbre non encore exploré amène trivialement à un ordonnancement optimal de l'arbre entier, en concaténant à l'ordonnancement de P_1 l'ensemble des nœuds non marqués des niveaux inférieurs à l, et en concaténant à l'ordonnancement de P_2 l'ensemble des nœuds marqués appartenant à des sous-arbres dont les racines sont de niveau inférieur à l (ce dernier ensemble étant inférieur ou égal en taille à celui alloué à P_1).

Théorème T0
Les ordonnancements que produit MAJYC-c2 pour le problème $P_2 \mid arbre,\ p_i = 1,\ c = 2 \mid C_{max}$ *sont tels que :*

$$\omega_{MAJYC-c2} \leq \omega_{optimal} + 1$$

Preuve
Durant la phase d'allocation plaçons nous à un niveau donné l. Si $Reste > 0$ et qu'il ne subsiste à ce niveau qu'un seul nœud non marqué, $Reste$ est mis à jour et l'ordonnancement se poursuit au niveau supérieur. Si au niveau l il existe au moins trois nœuds non marqués dont les poids respectifs sont supérieurs ou égaux à $Reste$, alors le nombre total de nœuds est suffisant pour recouvrir la communication entre le dernier sous-arbre choisi et son successeur immédiat. L'ordonnancement produit par MAJYC-c2 est alors optimal. Considérons le cas où ce nombre de nœuds est égal à deux. Soient $Poids(Ssa1)$ et $Poids(Ssa2)$ leur poids respectif. Si $Reste \leq max(Poids(Ssa1)-2, Poids(Ssa2)-2)$, alors le choix naturel de l'algorithme se porte sur le sous-arbre de plus faible poids et il reste au moins deux tâches à exécuter par P_1 pour recouvrir la communication, et comme précédemment, l'ordonnancement produit par MAJYC-c2 est optimal. Le cas $Reste = Poids(Ssa1)-1 = Poids(Ssa2)-1$ constitue le seul cas pour lequel MAJYC-c2 ne produit pas un ordonnancement optimal, mais introduit une période d'inactivité sur P_1 due à la communication, et l'ordonnancement obtenu est distant d'au plus une unité de temps de l'optimal. En effet, l'algorithme choisi $Reste$ tâches dans un des sous-arbres, et la dernière tâche qu'il exécute n'est pas la racine de l'un des sous-arbres.

Recherche d'un ordonnancement optimal

Lorsque $Reste$ est calculé, seul est pris en compte le quotient de la division, $Reste = (n-3)div(2)$. La valeur du reste de la division $(n-3)mod(2)$ donne une indication sur la liberté de choix des nœuds. Lorsqu'il est égal à 0, cette liberté est nulle.
Appelons P_2 le processeur pour lequel les sous-arbres marqués sont choisis, et P_1 le processeur qui exécute la racine de l'arbre.

Théorème T1
Soit S un ordonnancement tel que P_1 soit toujours actif et P_2 exécute $(n-3)div(2)$ tâches. Cet ordonnancement est optimal et contient exactement $2+(n-3)mod(2)$ périodes d'inactivité.

Corollaire C1
Supposons que S soit un ordonnancement tel que P_1 soit toujours actif et tel que P_2 exécute $(n-3)div(2)$ tâches. Le niveau de la dernière tâche exécutée par P_2 est compris entre 2 et $(n-3)mod(2)+2$.

Preuve
Supposons que le niveau de cette dernière tâche (notée T_{der}) soit supérieur ou égal à $(n-3)mod(2)+3$. Dans ce cas, la date de fin d'exécution de T_{der} est minorée par $(n-3)div(2)$ (cas où il n'y a pas d'attente sur P_2). A cette date P_1 a donc exécuté $(n-3)div(2)$ tâches également (P_1 est toujours actif). De plus, P_1 dispose d'au moins 2 tâches (qui n'appartiennent pas à l'ensemble des successeurs de T_{der}) prêtes à être exécutées pour recouvrir la communication. Le niveau de T_{der} est par hypothèse supérieur ou égal à $(n-3)mod(2)+3$. T_{der} a donc au moins $(n-3)mod(2)+1$ successeurs (racine non comprise) qui doivent être exécutées par P_1. A la fin de l'exécution, P_1 a donc exécuté $(n-3)div(2)+c+(n-3)mod(2)+1$ tâches plus la racine et P_2 en a exécuté $(n-3)div(2)$. Au total $n-1-2+1+2+1 = n+1$ tâches

ont été exécutées, alors que l'arbre n'en comportait que n. Cette contradiction implique que le niveau de la dernière tâche exécutée par P_2 est inférieur ou égal à $(n-3)mod(2)+2$.

Modification de l'algorithme en conséquence

Nous ne considérons dans cette partie que des arbres dont les deux prédécesseurs immédiats de la racine sont de même poids égal à $Reste+1$. Pour ce cas particulier $(n-3)mod(2) = 0$. D'après le Corollaire C1, le niveau de la dernière tâche exécutée par P_2 doit appartenir au niveau deux de l'arbre. Or le poids des nœuds de ce niveau est strictement supérieur à $Reste$. La solution consiste à trouver une partition des tâches de l'un des sous-arbres de telle sorte que P_2 exécute la racine d'un des deux sous-arbres. Cela revient à trouver dans le sous-arbre exécuté par P_2 une tâche exécutable par P_1 sans entraîner d'attente sur P_2. Au moment de l'exécution, P_1 commence par exécuter cette tâche, puis exécute son sous-arbre. Pendant ce temps P_2 doit pouvoir exécuter une tâche, puis encore suffisamment pour recouvrir la communication de P_1 vers P_2. Il faut donc que l'un des sous-arbres, issus de la racine de l'arbre entier, contienne un nœud nd (d'arité supérieure ou égale à deux) dont le poids soit supérieur ou égal à 5 : égal au poids de nd lui-même plus le poids des deux tâches exécutées au même instant par les deux processeurs, plus deux tâches pour recouvrir la communication. Mais il faut aussi que les deux tâches qui permettent de recouvrir la communication ne soient pas des successeurs de la tâche exécutée par P_1. Il faut donc que le sous-arbre dans lequel la tâche exécutée par P_1 soit de poids inférieur ou égal à $Poids(nd) - 1 - (1+2)$.
Pour le cas qui nous occupe, il faut donc trouver un nœud d'arité supérieure ou égale à deux et de poids supérieur ou égal à cinq et dont l'un des sous-arbres est de poids inférieur ou égal à $Poids(noeud) - 4$.
Si un tel sous-arbre existe alors il existe un ordonnancement de l'arbre qui alloue $(n-3)div(2)$ tâches à P_2 et tel que P_1 soit toujours actif. Ce qui garantit l'optimalité. D'autre part, tout ordonnancement vérifiant ces deux conditions implique que l'arbre de départ contenait un tel sous-arbre. On remarque que s'il n'existe pas de telle tâche, l'ordonnancement donné par MAJYC-c2 est optimal (pour $c = 2$).
La complexité de cette recherche n'augmente pas celle de l'algorithme.

Exemple

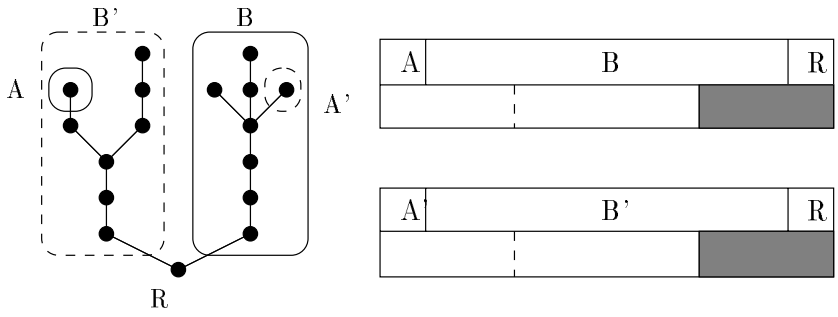


FIG. *2.28 - : Un exemple d'application de MAJYC-c2 avec la procédure de recherche.*

Remarque
La propriété de constante activité sur un processeur est pour ce cas vérifiée. En effet, supposons que nous disposions d'un ordonnancement optimal ayant des périodes d'inactivité dans son ordonnancement. Il ne peut y avoir plus de deux unités de temps successives pendant lesquelles le processeur est inactif. En effet, si trois unités de temps successives étaient inactives alors, soit l'exécution de la tâche suivante peut être avancée d'une unité de temps et l'ordonnancement n'était pas optimal, soit (si l'un de ses prédécesseurs termine son exécution sur le second processeur deux unités de temps avant) l'un de ses prédécesseurs peut être déplacé et le temps total de l'ordonnancement diminué, auquel cas l'ordonnancement n'était pas optimal.
Donc, si la durée d'inactivité est égale à une unité de temps, le prédécesseur qui est exécuté sur l'autre processeur est déplacé. Si cette durée est égale à deux unités de temps, alors le prédécesseur qui est exécuté sur l'autre processeur est déplacé et si cela ne suffit pas, le prédécesseur du prédécesseur également (voir figure 2.29).

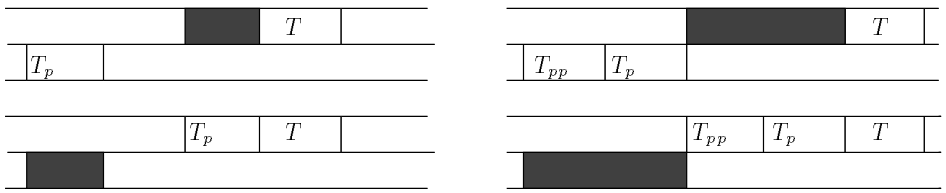


FIG. *2.29 - : Les périodes d'inactivité peuvent toujours être éliminées.*

Conclusion

Nous avons construit un algorithme de complexité linéaire, optimal pour le cas où la communication est constante égale à 2. Il est à noter que l'algorithme proposé par Jung, Kirousis et Spirakis produit des ordonnancements optimaux pour ce même problème, mais avec une complexité de l'ordre de $O(n^3)$.

2.3.2 $P_2 \mid arbre,\ p_i = 1,\ c \mid C_{max}$

Pour une valeur de c plus importante, c'est-à-dire lorsque la granularité est plus fine, le cas au pire n'est plus exactement le même que précédemment. Cependant, le corollaire C1 est valide.
Nous définissons pour ce nouveau problème l'algorithme **MAJYC-c** qui diffère de MAJYC-c2 pour le calcul de $Reste$: $Reste = (n-1-c)div(2)$.
Nous montrons que les ordonnancements construits par cet algorithme sont distants de $(c)div(2)+1$ au plus de l'optimal.

L'écart entre un ordonnancement optimal et un ordonnancement construit par MAJYC-c provient du choix terminal, c'est-à-dire des choix qui interviennent lorsqu'à un certain niveau il existe au moins deux nœuds dont le poids est supérieur strictement à la valeur de $Reste$, et lorsqu'il n'existe à ce niveau aucun nœud ayant un poids inférieur ou égal à cette valeur.

Cependant, contrairement au cas $c = 2$, les ordonnancements construits par cet algorithme sont distants de plus de 1 de l'optimal, comme le montre la figure 2.30.

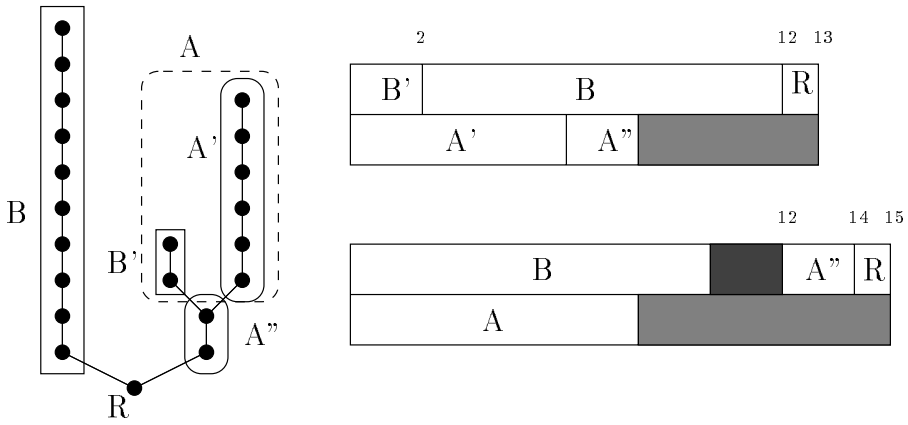


FIG. *2.30 - : Un ordonnancement distant de 2 de l'optimal (c=4).*

Cependant, cet écart ne peut pas être supérieur à $(c)div(2)+1$. En effet, si l'on considère un arbre de n tâches dont l'ensemble des prédécesseurs immédiats sont de poids égaux et au nombre de deux. La charge restant à allouer à P_2 est égale à $\left\lfloor \frac{n-1-c}{2} \right\rfloor$. Donc, $(n-1-c)div(2)+(n-1-c)mod(2)+c+1$ est une borne inférieure de la date de fin d'exécution. Un ordonnancement construit par MAJYC-c choisi dans l'un des deux sous-arbres le nombre de tâches correspondant à la charge calculée. La hauteur maximum de la dernière tâche exécutée sera donc égale à $2+(n-1)div(2)-(n-1-c)div(2)$. Ce qui signifie que le nombre de tâches que P_1 peut exécuter après la date de fin d'exécution des tâches sur P_2 est égal à $(n-1)div(2)-(n-1-c)div(2)$, ce qui est égal à $(c)div(2)$. Donc, au pire, P_1 devra attendre $(c)div(2)+1$ unités de temps pour exécuter la racine.

2.4 Conclusion

Dans ce chapitre, un nouvel algorithme pour l'ordonnancement des arbres *UECT* a été présenté, et son optimalité prouvée pour un système bi-processeur. Bien qu'il ne soit pas le seul algorithme existant qui produise des ordonnancements optimaux pour ce problème, il présente un avantage de pouvoir être facilement adaptable pour ordonnancer des arbres dont la granularité est voisine.

Il a été montré que pour des versions de ce même algorithme très légèrement modifiées, les ordonnancements produits, pour des arbres à gros grain (communications unitaires et durées des tâches comprises entre 1 et k), ne sont distants que de $\left\lceil \frac{k}{2} \right\rceil$ unités de temps de l'optimal. Ce qui signifie que cet algorithme est asymptotiquement optimal pour des arbres comportant un grand nombre de sommets, dès que $n \ll k$. Ce résultat présente un intérêt sachant que ce type de problème est NP-complet sans communication et pour un nombre de processeurs restreint à deux [DL88]. Pour des arbres dont l'exécution des tâches nécessite

une ou deux unités de temps, une analyse fine a permis la mise au point d'un algorithme qui génère des ordonnancements optimaux. Ce résultat est nouveau et généralise une étude de problèmes d'ordonnancement pour de tels arbres [NLH81], par l'ajout de communications.

Lorsque la granularité est fine, c'est-à-dire que le temps requis pour effectuer une communication est supérieur à une unité de temps, MAJYC s'adapte une nouvelle fois de façon interessante, et le même type de comportement que pour les arbres à gros grain se retrouve. L'heuristique présentée pour des arbres dont les temps de communication sont égaux à $c > 1$, ne sont distants que de $\left\lceil \frac{c}{2} + 1 \right\rceil$ unités de temps d'un ordonnancement optimal. De nouveau, cette heuristique est asymptotiquement optimale, dès que $n \ll c$. Enfin, pour le cas particulier où $c = 2$ nous avons proposé un algorithme qui produit des ordonnancements optimaux en un temps linéaire. Cette heuristique, propose une alternative à l'utilisation de l'algorithme proposé par Jakoby et al. [JKS89] qui produit des ordonnancements optimaux mais dont la complexité est forte ($O(n^{c+1})$).

Il apparaît cependant que cet algorithme basée sur une stratégie de regroupement se généralise assez difficilement pour un nombre de processeurs supérieur à deux. Cependant, nous allons montrer dans le chapitre suivant que ce type de stratégies peut être avantageusement mis en oeuvre pour un nombre de processeurs $m \geq 2$ et que certains algorithmes basés sur ces stratégies peuvent manifester une adaptabilité importante pour des granularités voisines.

Chapitre 3

Ordonnancement d'arbres sur m processeurs identiques

Ce chapitre est dédié à l'étude d'heuristiques pour l'ordonnancement des arbres sur un nombre m fixé de processeurs. Nous décomposons comme pour le chapitre précédent ce chapitre en trois principales sections. La première concerne le problème *UECT* correspondant au modèle d'exécution proposé par Rayward-Smith [RS87]. Dans cette partie, une nouvelle heuristique est proposée pour l'ordonnancement des arbres sur m processeurs. Par une analyse du cas au pire, une borne qualifiant la qualité des ordonnancements produits est calculée. Dans une seconde partie, l'ordonnancement des arbres à gros grain est traité (temps d'exécution supérieurs aux temps de calcul). Une borne au pire du comportement de l'heuristique décrite dans la première partie est calculée. Enfin, la dernière section traite de l'ordonnancement des arbres à grain fin, c'est-à-dire correspondant au modèle d'exécution décrit par Papadimitriou et Yannakakis [PY88].

3.1 $P_m \mid arbre,\ p_i = 1,\ c = 1 \mid C_{max}$

Nous avons vu dans le chapitre précédent que le problème de l'ordonnancement des arbres *UECT* est NP-complet lorsque le nombre de processeurs n'est pas fixé [Vel93b, Saa92, Pic93]. Ce même problème est trivial lorsque le nombre de processeurs est non borné, et est résolu de façon exacte par un algorithme de programmation dynamique de complexité égale à $O(n^{2(m-1)})$ [VRKL94]. Une heuristique de complexité linéaire basée sur la propriété du fils favorisé (décrite dans le chapitre précédent) permet d'obtenir des ordonnancements distants de moins de $m-2$ unités de temps de l'ordonnancement optimal [Law93]. Nous proposons dans cette section un nouvel algorithme basé sur une stratégie de regroupement des tâches. Nous exposerons le principe qui permet d'affirmer que la borne proposée dans [VRKL94] peut être améliorée, et par une analyse du cas au pire nous donnons une nouvelle borne qui est atteinte, à la fois par l'algorithme qu'ils proposent et par celui que nous présentons dans la section suivante.

3.1.1 Une heuristique pour $P_m \mid arbre,\ p_i = 1,\ c = 1 \mid C_{max}$

L'idée de cette heuristique est de modifier un ordonnancement optimal produit pour un nombre non limité de processeurs, en réduisant le nombre de processeurs à m.

Principe de l'heuristique

L'ordonnancement optimal initial est construit par un algorithme de regroupement de tâches. La première étape consiste à attribuer à chaque tâche une date d'exécution au plus tôt. Les groupes sont ensuite constitués, en partant de la racine. Lorsqu'un nœud a plusieurs prédécesseurs immédiats dont la date au plus tôt est la même, l'un d'eux est choisi arbitrairement pour appartenir au groupe. Il s'agit d'un regroupement linéaire. Cette étape terminée, tous les nœuds appartiennent à un groupe (éventuellement réduit à une seule tâche) comme illustré sur la figure 3.1. La dernière étape consiste à attribuer à chaque tâche une date d'exécution au plus tard, en fonction de l'appartenance à un groupe de tâches, sachant que la communication entre deux nœuds successifs appartenant à un même groupe est de coût nul.

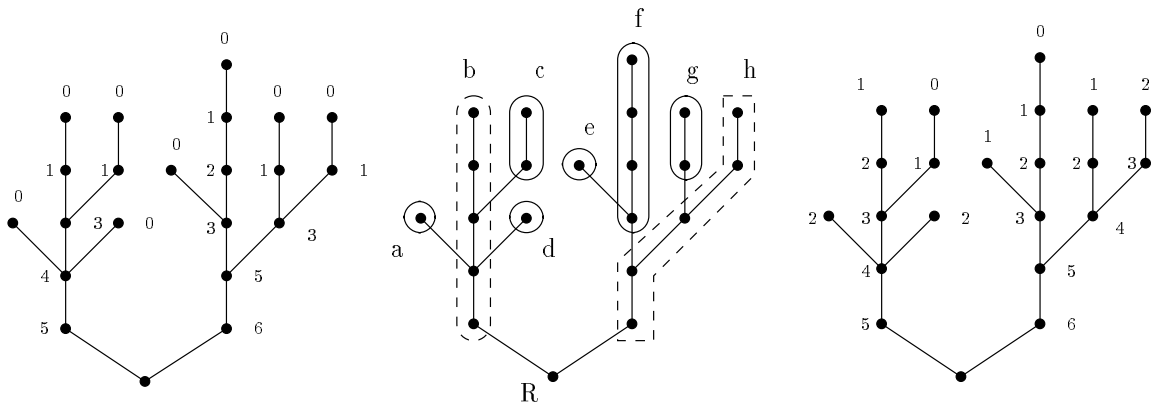


FIG. *3.1 - : Dates au plus tôt, formation des groupes, dates au plus tard.*

La seconde étape réduit le nombre de processeurs utilisés. Le processus de réduction débute à la racine. L'algorithme fait la somme de l'ensemble des processeurs utilisés pour chaque tranche du diagramme de Gantt correspondant à une unité de temps (voir la figure 3.2).

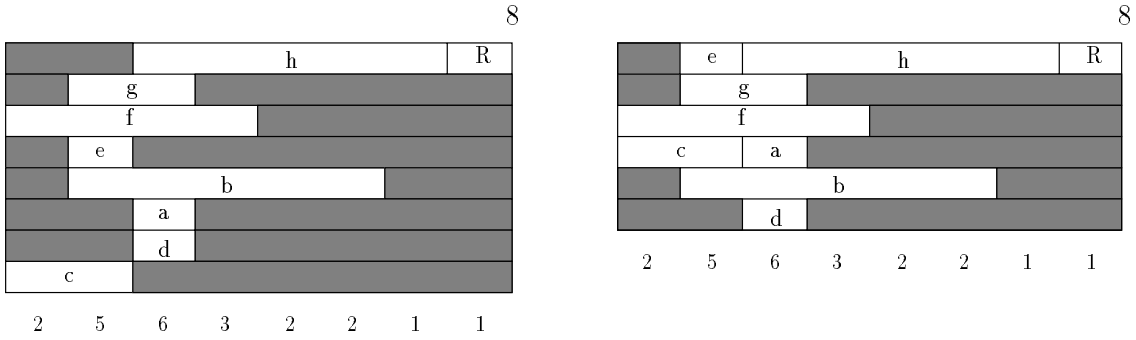


FIG. *3.2 - : Ordonnancement initial de l'arbre (le nombre de processeurs requis est non borné).*

Soit m' le nombre de tâches exécutées durant l'intervalle de temps t'. Si m' est inférieur ou égal à m, le prochain intervalle de temps est examiné. Si $m' > m$, alors il y a trop de tâches ordonnancées dans cet intervalle de temps, l'exécution de certaines d'entres elles doit être avancée[1]. L'algorithme conserve pour cet intervalle de temps les m tâches dont le chemin critique est le plus grand, et avance l'exécution des $m' - m$ autres tâches, ainsi que les toutes les tâches appartenant au sous-arbre dont ces $m' - m$ tâches sont racines. Ce procédé est itéré pour tous les intervalles de temps. Lorsque la totalité des tâches en présence sont des feuilles (chemin critique égal à 1), elles sont réparties en $\left\lceil \frac{m'}{m} \right\rceil$ intervalles de temps.

Description algorithmique de la réduction

On note $N_{IntervalleCourant}$ le nombre de tâches exécutées durant cet intervalle de temps. L'itération commence pour l'intervalle de temps durant lequel la racine est exécutée. Le nombre de processeurs est fixé égal à m.

Tant que ($N_{IntervalleCourant} > 0$) **Faire**
  Si ($N_{IntervalleCourant} > m$) **Alors**
    Pour (tous les noeuds exécutés durant cet intervalle) **Faire**
      conserver les m noeuds ayant les chemins critiques les plus grands
      pour les autres, faire remonter d'une unité de temps l'ensemble
      des noeuds appartenant aux sous-arbres dont ils sont racines
    FinPour
  FinSi
  IntervalleCourant = intervalle de temps précédent
Fin Tant que

[1] avancée, c'est-à-dire que la tâche est repoussée vers la gauche dans le diagramme de Gantt, c'est-à-dire vers la date de début d'exécution de l'arbre.

Exemple

On ordonnance l'arbre de la figure 3.1. L'ordonnancement est construit sur huit processeurs, mais le nombre maximum de processeurs utilisés à un instant donné se limite à six (voir figure 3.2). La réduction commence pour le premier intervalle de temps contenant plus de tâches que le nombre de processeurs. Ce premier intervalle contient six tâches. Parmi elles, il y en a trois dont le chemin critique est égal à 1. Ce sont ces trois tâches ainsi que les sous-arbres dont elles sont racines qui sont avancées (étapes 1 et 2). Le nombre de tâches de l'intervalle suivant est égal à huit, il faut donc remonter cinq tâches. Elles sont choisies de la même façon (étapes 2 et 3). Enfin, lorsque toutes les tâches restant sont des feuilles, elles sont réparties en $\lceil m'/m \rceil$ intervalles de temps (étapes 3 et 4).

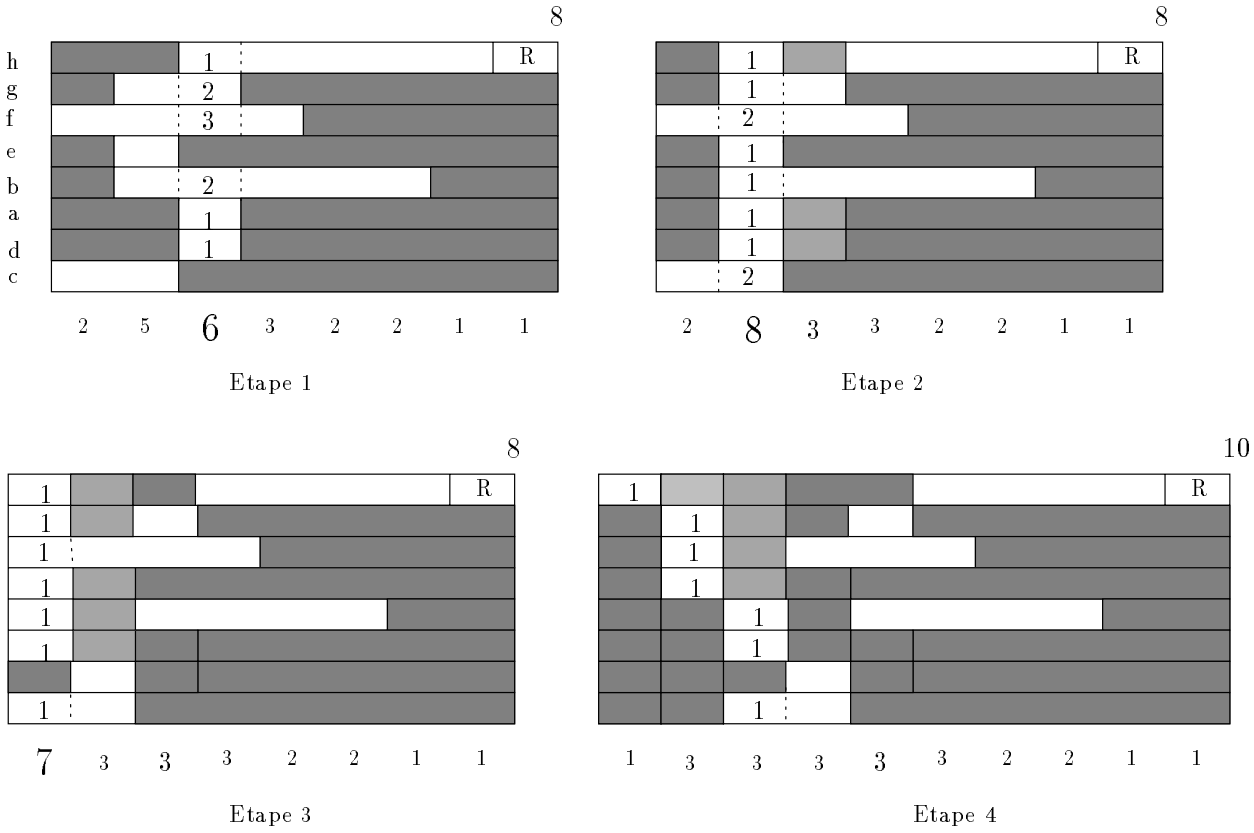


FIG. *3.3 - : Un exemple de réduction pour* $m = 3$.

Remarque

Dans l'ordonnancement initial, les contraintes de précédence ainsi que les attentes de communications sont respectées. Le type de regroupement utilisé par cet algorithme est un regroupement linéaire, ce qui implique que lorsqu'un groupe est coupé (ce qui ne se produit que lorsque l'exécution de la première partie d'un groupe est avancée), les deux parties résultantes peuvent être allouées à deux processeurs différents, sans attente de communication. Cette remarque explique que le nombre de tâches par intervalle de temps est équivalent au nombre de processeurs utilisés.

3.1.2 Analyse au pire et nouvelle borne

L'heuristique que nous venons de présenter est de même complexité que celle présentée dans [VRKL94]. Il y a équivalence entre le choix du fils favori dans l'heuristique à m machines qu'ils présentent et la constitution des groupes de tâches avec priorité en fonction du plus long chemin jusqu'à une feuille que nous présentons. La réduction que nous présentons est une stratégie voisine de l'algorithme de Hu [Hu61]. Le critère de priorité dans l'algorithme de Hu est *HLF* (*Highest Level First*), c'est-à-dire que les tâches qui sont exécutées en premier sont celles dont le chemin jusqu'à la racine est le plus long. Elles sont alors exécutées **au plus tôt**. Dans la réduction que nous proposons, les tâches les plus prioritaires sont celles dont le chemin jusqu'à une feuille est le plus long et sont exécutées **au plus tard**.

Analyse au pire

Nous présentons dans cette section le cas au pire pour ces heuristiques. A partir de ce cas au pire nous calculons une nouvelle borne de la qualité des ordonnancements. Cette borne n'est pas prouvée dans le présent document, mais dans [GRT95]. Nous présentons toutefois deux résultats pour des types d'arbres particuliers.

Lemme 1
Lorsque la largeur maximum de l'arbre est inférieure ou égale à m, l'heuristique présentée produit des ordonnancements optimaux.

Preuve
Considérons un tel arbre et notons n_2 le nombre de noeuds du niveau 2, et n_i le nombre de noeuds du niveau i. Notons également t la date de fin d'exécution de la racine. Parmi l'ensemble des noeuds du niveau 2, l'un des noeuds N appartenant à un plus long chemin est choisi pour être ordonnancé à $t-2$. Parmi l'ensemble des prédécesseurs de ce noeud, un seul pourra être ordonnancé à $t-3$. Nou le notons N_p. D'autre part, les autres noeuds candidats à un ordonnancement à cette date sont les noeuds du niveau 2 qui ne sont pas encore ordonnancés et qui ne sont pas des prédécesseurs de N. Au total, le nombre de noeuds candidats pour être ordonnancés à $t-3$ est inférieur ou égal à $n_2 - n_N + 1$ (si l'on note n_N le nombre de prédécesseurs de N) et est inférieur ou égal à m. En revanche pour la date $t-4$, le nombre de noeuds candidats à un ordonnancement sont les noeuds libérés par N qui ne sont pas encore ordonnancés, soit $n_N - 1$ plus un noeud pour chacun des $n_2 - n_N + 1$ noeuds ordonnancés à la date $t-3$. Au total, au plus $n_2 - n_N + 1 + n_N - 1 = n_2$ noeuds sont candidats pour être ordonnancés à $t-4$. De la même façon, le raisonnement peut être reproduit pour les $t-i$ en tenant compte des noeuds dont les successeurs ont été ordonnancés à $t-(i-2)$, et il est possible de montrer que le total des noeuds candidats à un ordonnancement est inférieur ou égal à m pour tous les niveaux. En d'autres termes, pour ce type d'arbres, le critère de plus long chemin est suffisant pour produire des ordonnancements optimaux. □

Lemme 2
Lorsque la largeur minimum de l'arbre est supérieure ou égale à m (à partir du niveau deux jusqu'au dernier niveau), l'heuristique présentée produit des ordonnancements optimaux.

Preuve
Si l'on examine le diagramme de Gantt produit pour ce type d'arbres, c'est évident. A partir du second intervalle de temps unitaire, tous les processeurs sont actifs jusqu'à l'avant dernier intervalle dans lequel n'est exécuté que le prédécesseur de la racine qui a été choisi pour recouvrir la communication. C'est-à-dire que l'ordonnancement est d'efficacité maximale pour de tels arbres car le nombre de périodes d'inactivité est minimisé. □

Le cas au pire se présente lorsqu'à chaque choix de tâche, la mauvaise est choisie par l'algorithme, c'est-à-dire lorsque le critère du plus long chemin n'est pas adapté. Deux exemples représentent ce pire cas.

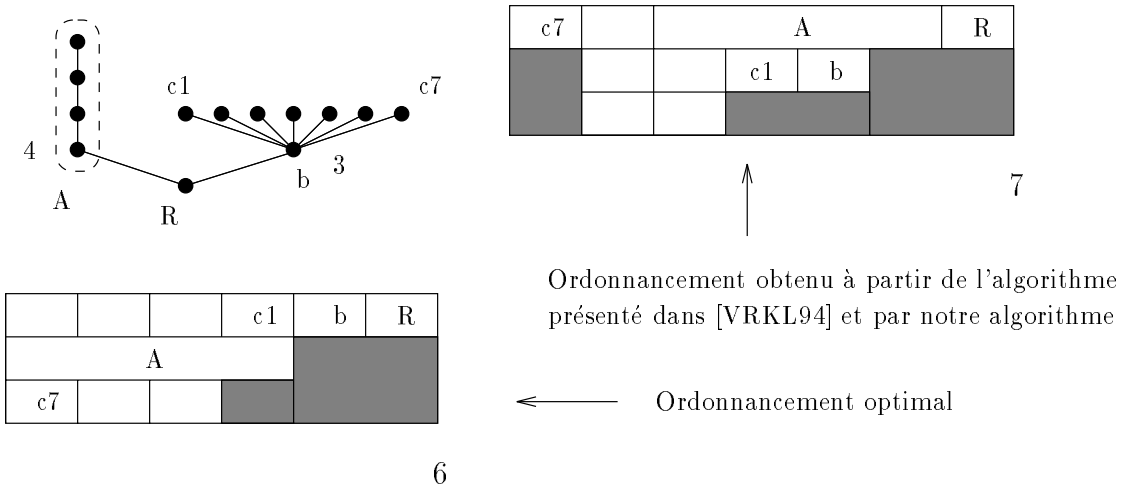


FIG. *3.4 - : Cas au pire pour ces algorithmes: premier exemple.*

Ce second exemple montre le type de graphe dont l'ordonnancement par notre heuristique mène au pire cas. Ces arbres peuvent être construit de façon systématique pour un ordonnancement sur m processeurs :

Soit R_1 un noeud successeur de $m + 1 + \dfrac{(m-1)(m-2)}{2} + 2$ feuilles
Pour $i = 2$ *à* $i = m - 1$ **Faire**
ajouter un noeud R_i dont R_{i-1} est l'un des
prédécesseur immédiat
et dont C_{i-1}, la racine d'une chaîne
de longueur $2i$
FinPour

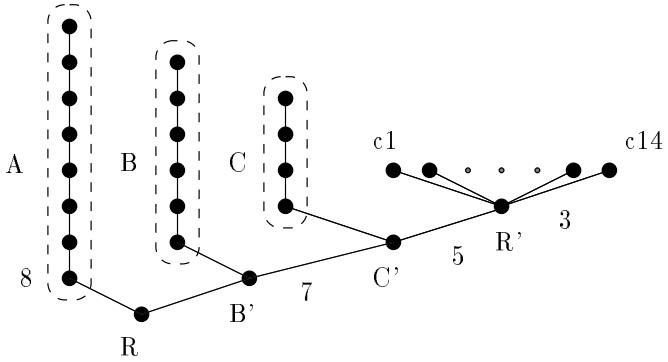


FIG. *3.5 - : Cas au pire pour ces algorithmes. Exemple 2: le graphe.*

La structure des ordonnancements dans le pire cas apparaîssent alors clairement dans le diagramme de Gantt. Il s'agit d'un escalier dont les marches sont deux fois plus longues par rapport à ce qu'elles sont dans un ordonnancement optimal, comme le montre la figure 3.6.

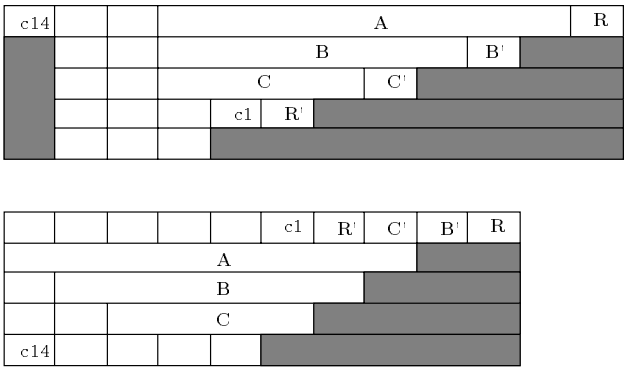


FIG. *3.6 - : Cas au pire pour ces algorithmes. Exemple 2: le diagramme de Gantt.*

La perte en temps est nulle pour les deux premiers processeurs, ce qui explique d'une autre façon l'optimalité de ces deux heuristiques pour $m = 2$. A partir du troisième processeur, une unité de temps de calcul pour un processeur est perdue par le troisième processeur, puis deux unités de temps de calcul rapporté à un processeur sont perdues par le quatrième, puis i unités de temps par le $(i+2)^{ième}$ processeur. Au bout du compte, c'est $\sum_{i=1}^{i=m-2} i$ unités de temps de calcul rapportées à un processeur qui sont perdues. Donc, au total $\frac{(m-1)(m-2)}{2}$ unités de temps ont été perdues. Mais, ces unités de temps perdues doivent être rapportées au nombre de processeurs. Donc, en définitive, la date de fin d'exécution atteinte par les heuristiques est distante de moins de $\left\lceil \frac{(m-1)(m-2)}{2m} \right\rceil$ unités de temps d'un ordonnancement optimal.

Remarques

Il semblerait que l'arité soit pour beaucoup dans le cas au pire, malheureusement, il ne semble pas aisé de parvenir à maîtriser à la fois le chemin critique et l'arité comme critères

concurrents. Toute tentative de mélange trivial de ces deux critères semble voué à l'échec comme le montre la figure 3.7, dans laquelle le noeud qu'il faut privilégier pour recouvrir la communication (B) n'est pas situé sur un plus long chemin et n'a pas une arité supérieure à celle de l'autre noeud candidat (A).

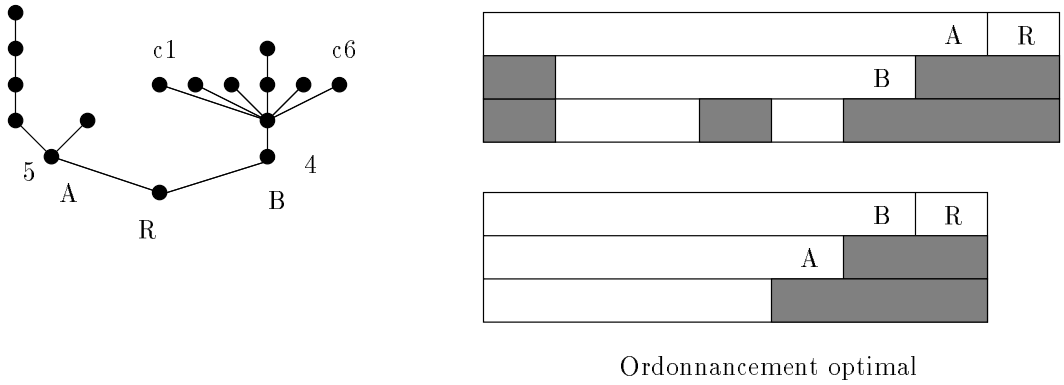


FIG. *3.7 - : Intervention de l'arité.*

3.1.3 Conclusion

L'heuristique que nous venons de présenter vérifie le même cas au pire que l'heuristique basée sur une stratégie niveau par niveau présentée dans [VRKL94]. Cependant, cette heuristique peut être adaptée sans gros effort à des arbres dont la granularité n'est plus tout à fait la même.

3.2 Granularités voisines

Lorsque les arbres considérés ne sont plus *UECT*, les techniques de regroupement de tâches semblent plus robustes au changement de granularité que ne le sont les algorithmes de listes. En particulier, notre algorithme de regroupement est plus souple d'utilisation que la méthode consistant à construire un *delay-free-tree*. En effet, cette dernière oblige tous les prédécesseurs d'un nœud à attendre au moins deux unités de temps pour communiquer si le *favorite child* est un nœud dont la durée d'exécution est égale à deux unités de temps. De même, lorsque les communications sont égales à deux unités de temps ou plus, la réservation d'une tâche pour recouvrir la communication n'est pas suffisante. Dans les deux paragraphes suivants, une adaptation de l'heuristique présentée précédemment va être exposée. La constitution des groupes de tâches ne change pas. Par contre, la réduction est légèrement différente, elle dépend du poids des tâches et du plus long chemin.

3.2.1 $P_m \mid arbre,\ p_i,\ c = 1 \mid C_{max}$

Principe

Lorsque le graphe est à gros grain, la stratégie de regroupement linéaire utilisée précédemment permet encore de produire des ordonnancements optimaux lorsque le nombre de

processeurs n'est pas limité. Cependant, l'exécution d'une tâche ne se limite plus à un intervalle d'une unité de temps, ce qui implique que pour un intervalle de temps donné $[t, t']$, toutes les tâches ne se terminent pas en t'. L'idée est alors de continuer à toutes les considérer, celles qui se terminent et celles qui sont en cours d'exécution. Le critère de choix des tâches dont l'exécution est maintenue en place et celles dont l'exécution est avancée est le plus long chemin jusqu'à une feuille, comme c'était déjà le cas précédemment.

Exemples

Il n'existe aucune différence entre le cas d'arbres à gros grain et d'arbres *UECT* en ce qui concerne le calcul des dates au plus tôt, de la constitution des groupes et du calcul des dates au plus tard.

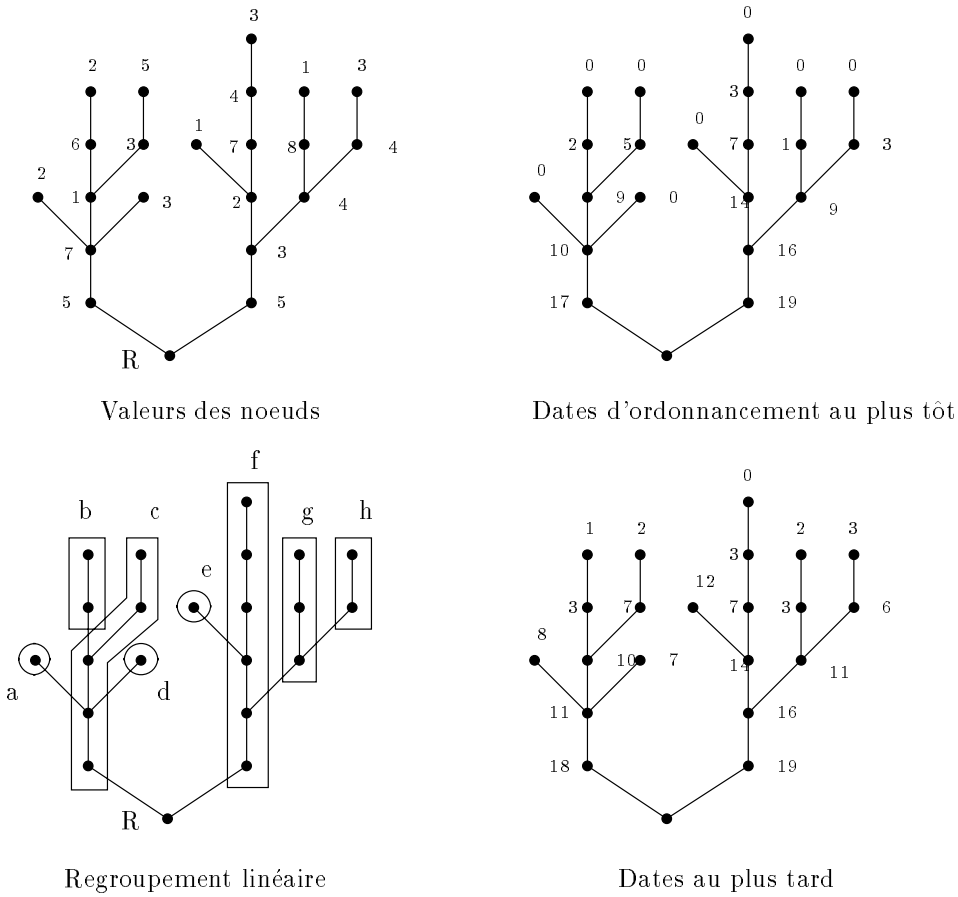


FIG. *3.8 - : Dates au plus tôt, formation des groupes, dates au plus tard.*

De la même façon que pour le cas *UECT*, les groupes de tâches sont constitués et ordonnancés sur un nombre non limité de processeurs. La date de fin d'exécution de cet ordon-

nancement initial est minimale.

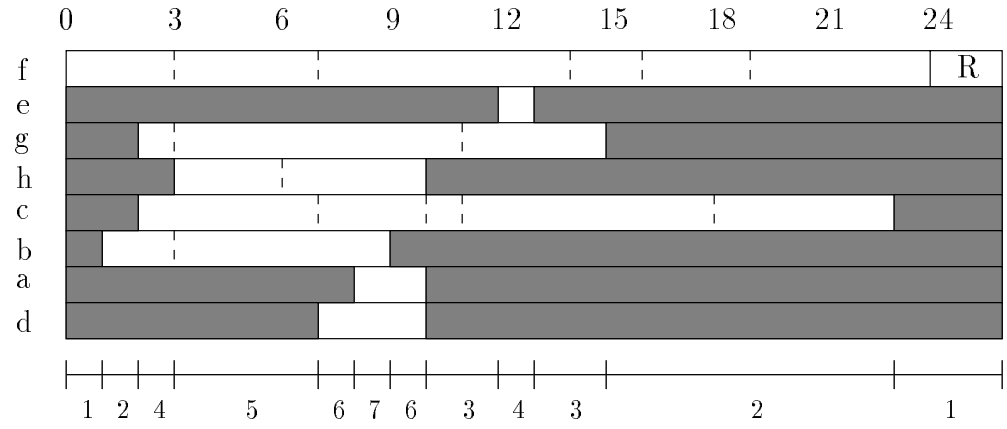


FIG. *3.9 - : Ordonnancement initial de l'arbre sur un nombre non limité de processeurs.*

C'est pour la réduction que l'heuristique diffère légèrement. Pour un intervalle de temps $[t, t+1]$ donné, toutes les tâches qui sont exécutées dans cet intervalle sont examinées, y compris les tâches dont la date de fin d'exécution est plus grande que $t+1$. Si toutes les tâches qui ne sont pas conservées dans cet intervalle terminent leur exécution en $t+1$, alors tout se passe comme dans le cas précédent. Les dates d'exécution de ces tâches sont avancées d'une unité de temps, ainsi que celles de toutes les tâches appartenant aux sous-arbres dont

elles sont racines. Ce cas est illustré par l'exemple de la figure 3.10).

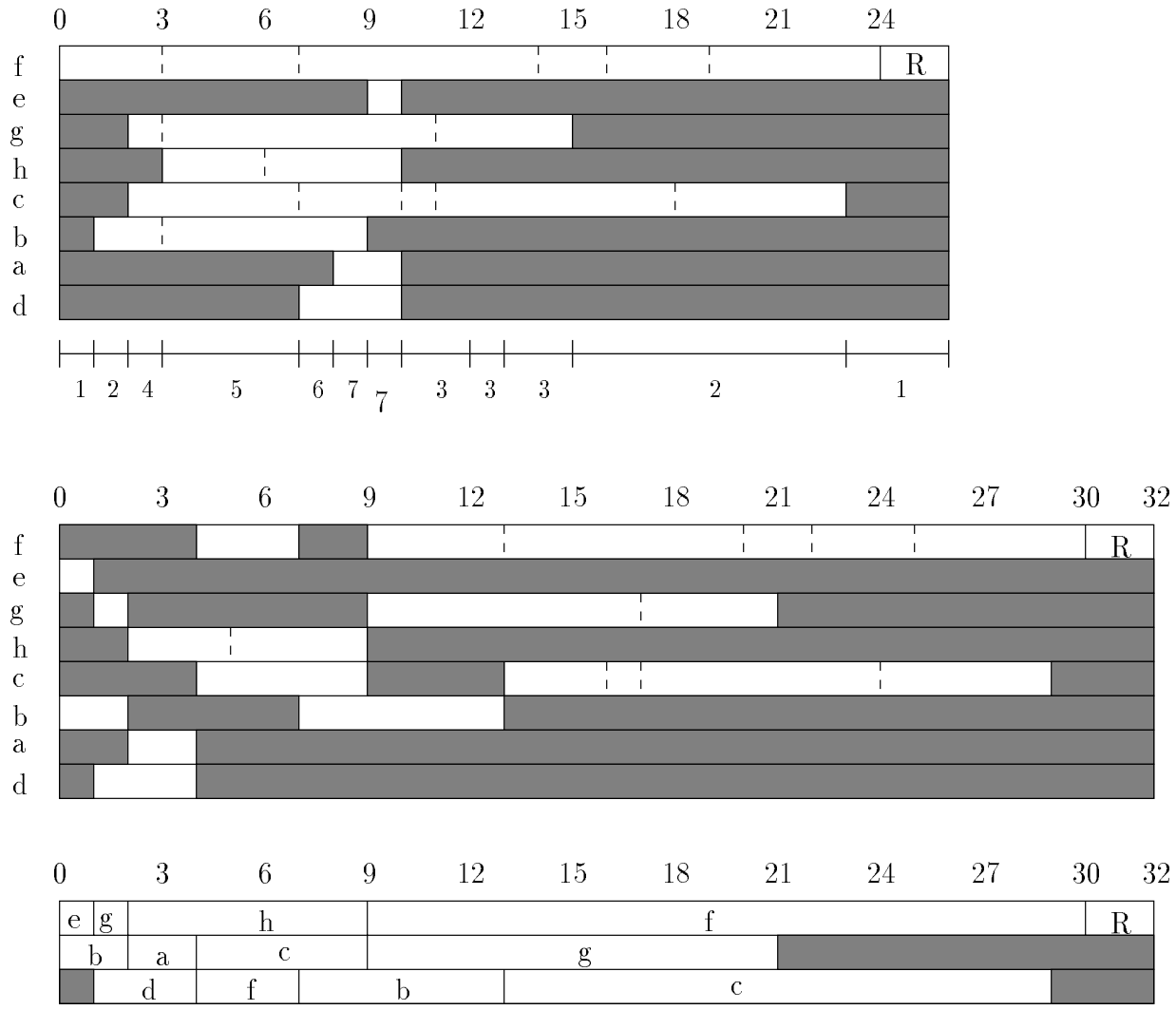


FIG. *3.10 - : Réduction du nombre de processeurs utilisés.*

Par contre, si une tâche T dont la date de fin d'exécution est strictement supérieure à $t+1$ (notons-la t") est choisie pour être avancée, elle va libérer $t" - (t+1)$ unités de temps sur le processeur auquel elle a été allouée. Or il est possible qu'une ou plusieurs tâches examinées dans l'intervalle $[t, t+1]$ aient une date de fin d'exécution supérieure à $t+1$, il peut donc être possible de recaser une ou plusieurs de ces tâches dans l'espace libéré par l'avancement de l'exécution de T. Ainsi, lors de l'examen des tâches exécutées durant un intervalle de temps (unitaire), toutes les tâches dont l'exécution doit être avancée et dont la date de fin d'exécution est supérieure à $t+1$ sont conservées dans une liste, et éventuellement recasées

si c'est possible. Ce cas de figure est représenté en figure 3.11.

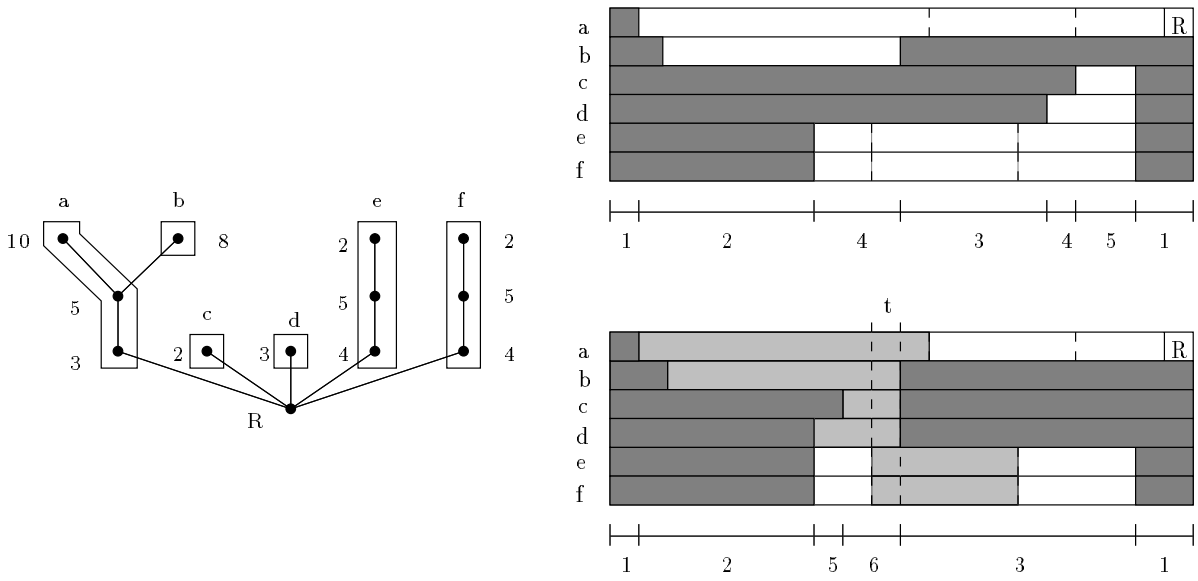


FIG. *3.11 - : Réduction de 8 à 3 du nombre de processeurs utilisés.*

Pour l'arbre de la figure 3.11, un regroupement est calculé ce qui donne un ordonnancement optimal sur six processeurs (le nombre de groupes). On suppose que le nombre de processeurs disponibles est limité à trois. La réduction commence normalement, seuls trois processeurs sont disponibles, ce qui entraîne l'avancée de l'exécution des groupes c et d jusqu'à ce que leur date de fin d'exécution correspondent à celle du groupe b. Arrivé à cette date t, six tâches sont ordonnancées pour être exécutées simultanément (elles apparaîssent en gris clair sur la figure 3.12). L'une d'entre elles appartient au groupe a, l'autre constitue le groupe b, elles sont en concurrence avec deux tâches appartenant aux groupes e et f, ainsi qu'avec les groupes c et d. Les priorités[2] respectives de ces tâches sont $\{10, 8, 7, 7, 2, 3\}$. Il ne peut y avoir que trois tâches s'exécutant de façon concurrente, donc l'exécution de l'une des tâches appartenant au groupe e ou f va être avancée. Or cette tâche avait déjà été en concurrence au préalable avec les groupes c et d, ce qui fait que l'avancée de son exécution laisse des emplacements vides dans l'ordonnancement, qui constitue une perte de temps s'il n'est pas possible de faire revenir une tâche, ce qui correspond au Cas 1 de la figure 3.12. Par contre, lorsque le retour d'une tâche est possible, le nombre d'unités de temps perdues peut être diminué comme le montre le Cas 2 de la même figure. Si le retour arrière est autorisé, la linéarité de l'algorithme semble être remis en cause, notamment parce qu'il faut effectuer

[2]la priorité est donnée ici par le plus long chemin jusqu'au début de l'exécution d'une feuille

un tri parmi les tâches candidates à ce retour.

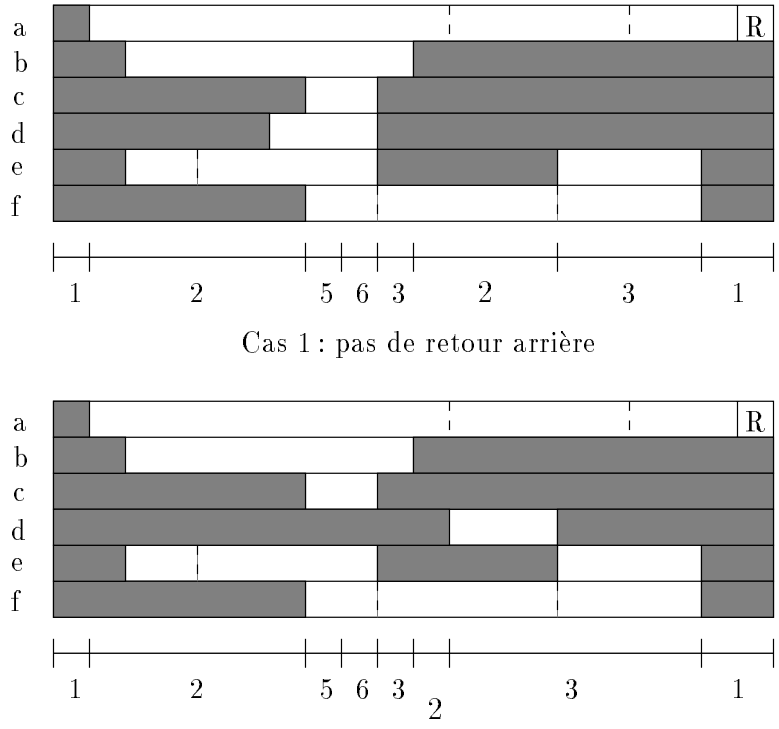


FIG. *3.12 - : Réduction du nombre de processeurs utilisés.*

3.2.2 $P_m \mid arbre,\ p_i = 1,\ c \mid C_{max}$

Principe

Lorsque le graphe est à grain fin, ce ne sont plus les mêmes problèmes que l'on rencontre. On remarque tout d'abord qu'une stratégie de regroupement linéaire ne permet pas d'atteindre l'optimal. Nous mettons donc en oeuvre une heuristique pour le calcul des dates d'ordonnancement au plus tôt de chaque tâches. Considérons une tâche T possédant plusieurs prédécesseurs T_p^i de dates au plus tôt d_i. On suppose que les d_i sont ordonnées par ordre décroissant. On considère les deux dates les plus grandes d_1 et d_2. Si $Min(d_1 + d_2, d_2 + c) = d_1 + d_2$, les groupes, dont T_p^1 et T_p^2 sont issues, fusionnent. Sinon, tous les groupes sont ordonnancés sur un processeur différent. Dans le cas d'une fusion, le processus est itéré pour l'ensemble des autres prédécesseurs. On recommence la même opération avec d_3, Si $Min(d_{1+2} + d_3, d_3 + c) = d_{1+2} + d_3$, les groupes fusionnent. Le processus de fusion s'achève lorsque $Min(d_{1+\ldots+i}, d_i + c) = d_i + c$. A la fin de cette étape de datation au plus tôt couplée avec le regroupement des tâches, on calcule les dates au plus tard et les groupes sont ordonnancés sur un nombre non limité de processeurs.

Lors de la réduction, les problèmes sont d'une autre nature que précédemment. Le problème principal vient du fait que lorsqu'une tâche est coupée du groupe auquel elle appartient,

elle ne doit pas être simplement avancée d'une unité de temps mais de c unités de temps si cette tâche n'est finalement pas exécutée sur le même processeur que son successeur immédiat, ou d'une unité de temps dans le cas contraire. Au départ, on dispose d'un groupe par processeur. Lorsque l'exécution du groupe complet est avancée, alors le groupe n'est déplacé que d'une unité de temps puisque la communication a déjà été prise en compte, par contre, lorsqu'un groupe est coupé en deux, il est alors avancé de c unités de temps. A la fin de la réduction, si un sous-groupe de tâches est ordonnancé sur le même processeur que l'ensemble de ses successeurs immédiats, alors, si des périodes d'inactivités apparaissent elles peuvent être comblées, seules les contraintes de précédence doivent être respectées. Il peut être pertinent de n'autoriser la division que lorsque tous les candidats ont une longueur de chemin critique supérieure de $c-1$ à celle de la tâche appartenant au groupe que l'on souhaite diviser.

Exemples

Le principe que nous venons de décrire est illustré par un cas simple dans la figure 3.13. Pour cet exemple, la communication est fixée égale à 5, et le nombre de processeurs fixé à deux. Les fusions successives amènent à la construction de quatre groupes de tâches.

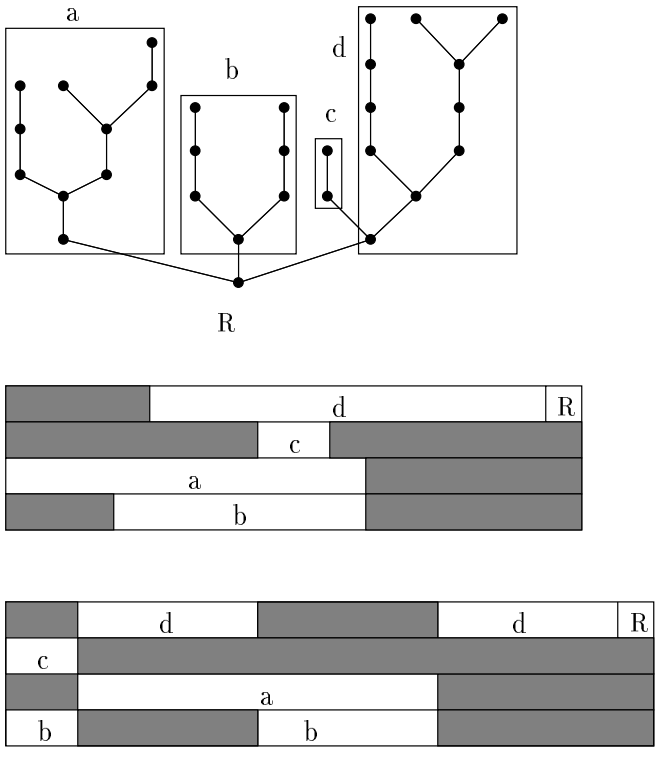


FIG. *3.13 - : Ordonnancement pour arbres à grain fin.*

3.3 Conclusion

Dans cette partie nous avons proposé plusieurs heuristiques basées sur des techniques de regroupement de tâches. Nous avons montré dans un premier temps que pour le cas *UECT*, notre heuristique faisait aussi bien qu'une heuristique par niveau présentée dans [VRKL94]. Cependant, nous montrons par la suite que pour d'autres granularités, moyennant de petites variations dans le calcul des dates au plus tôt ou dans l'étape de réduction, cette heuristique peut également produire des ordonnancements. La caractérisation de la qualité de ces ordonnancements pour des arbres dont les tâches ou les communications ne sont pas unitaires semble difficile. Une validation expérimentale sur des graphes aléatoires est envisagée.

Chapitre 4

Ordonnancement d'arbres sur processeurs uniformes

4.1 Introduction

Dans ce chapitre, nous cessons de considérer des processeurs identiques au profit des processeurs dits **uniformes**[1]. La particularité des systèmes formés par de tels processeurs est que le temps requis pour l'exécution d'une tâche T varie suivant le processeur sur lequel elle est exécutée, c'est-à-dire que la vitesse de deux processeurs uniformes n'est pas obligatoirement la même.

L'hypothèse de processeurs ayant des vitesses différentes est une réalité. Par exemple, toutes les machines connectées à un même réseau n'ont pas toutes la même puissance et *a fortiori* la même vitesse d'exécution pour une tâche donnée. Cependant, dans le cas général, pour un ensemble quelconque de tâches, il n'existe pas de relation entre les vitesses d'exécution des machines, on parle de processeurs sans relation[2]. Nous nous bornerons à étudier un sous-ensemble de ces processeurs qui rassemble ceux pour lesquels il existe une relation entre les vitesses et le temps d'exécution des tâches. Cette hypothèse nous permet de prendre en compte les processeurs uniformes mais également les processeurs sans relation, avec la contrainte supplémentaire que les tâches exécutées soient toutes d'un type bien déterminé, c'est-à-dire pour lesquelles il est possible de déterminer une relation entre les caractéristiques des processeurs et les temps d'exécution des tâches. Nous avons parlé de machines différentes connectées à un réseau, mais, même lorsque les machines sont identiques, lors de la montée en puissance d'un système multiprocesseur (qu'il s'agisse d'un réseau ou de tout autre système), les machines de remplacement ne sont pas toujours exactement identiques aux machines d'origine, de plus en plus souvent, les circuits équivalents sont moins chers, plus petits et fournissent de meilleures performances.

Du point de vue des applications, il n'existe pour l'instant que peu de travaux traitant des problèmes à base de processeurs uniformes. Ce phénomène semble être dû essentiellement

[1] définis en annexe.
[2] défini en annexe.

à deux facteurs. D'une part, les problèmes prenant en considération des machines uniformes sont plus difficiles que les problèmes correspondants utilisant des processeurs identiques. Par exemple, le problème $P_m \mid tree,\ p_i = 1 \mid C_{max}$ est résolu depuis 1961, alors que le problème $Q_2\ \{v_1, v_2\} \mid tree,\ p_i = 1 \mid C_{max}$ (où v_1 et v_2 représentent les vitesses des deux processeurs) n'est toujours pas résolu dès que $v_1 \neq v_2$ et $v_1, v_2 \neq 1$. D'autre part, il n'existe toujours pas à l'heure actuelle de modèle satisfaisant pour les architectures à base de processeurs uniformes. Pourtant, il existe une réelle demande de résultats ou de méthodes de résolution de problèmes liés à ces architectures, ainsi, comment répartir au mieux une importante charge de travail entre différents types de machines, quelle corrélation existe-t-il entre des algorithmes d'algèbre linéaire et une architecture formée de RS6000 d'un SP1, d'un T3D et de quelques stations Silicon Graphics ? Même s'il n'est pas possible d'apporter des réponses à toutes les questions, l'étude de tels problèmes peut permettre à terme une réelle amélioration de l'utilisation des différentes machines présentes sur les réseaux.

Dans la suite, nous limitons l'étude des problèmes avec processeurs uniformes pour des tâches non préemptibles, et pour le seul objectif de la minimisation du maximum des dates de fin d'exécution des tâches.

4.2 Tour d'horizon des travaux d'ordonnancement sur processeurs uniformes

4.2.1 Introduction

Le problème de l'ordonnancement de graphes de tâches, sans préemption, sur un système multiprocesseur formé de **processeurs uniformes** n'a donné lieu qu'à peu de travaux. Le problème le plus étudié dans ce contexte est celui de l'ordonnancement de travaux[3] indépendants [LL74a, LL74b, HS76, GIS77, CS80, FL83, Dob84, HS88]. Parmi les travaux prenant en compte les contraintes de précédence, Baer [Bae74] examine l'ordonnancement des arbres et des graphes orientés sans cycle dont les tâches sont unitaires et sur un système formé de deux processeurs. Enfin, Gabow étudie le comportement des algorithmes HLF et HLA pour des systèmes formés de deux processeurs uniformes [Gab88], et pour des graphes dont les tâches sont unitaires.

Dans les paragraphes suivants, le critère d'optimisation, le minimum des maximums des dates de fin d'exécution est noté C_{max} comme à l'accoutumé. Cependant, pour caractériser la performance d'un algorithme A, on notera la valeur du *makespan* pour cet algorithme C^A_{max}.

4.2.2 Ordonnancement de tâches indépendantes

Une grande partie des résultats pour l'ordonnancement de tâches indépendantes sur processeurs uniformes résultent d'une adaptation d'algorithmes mis au point pour des processeurs identiques. Parmi les premiers, Liu et Liu [LL74b, LL74a] analysent le comportement des algorithmes de liste. Il s'avère que les performances de ces algorithmes adaptés aux processeurs uniformes sont de qualité moyenne :

$$\frac{C^{liste}_{max}}{C^{optimal}_{max}} \leq 1 + \frac{\max_i v_i}{\min_i v_i} - \frac{\max_i v_i}{\sum_i v_i}.$$

Lorsque la fonction d'allocation des tâches aux processeurs privilégie la date de fin d'exécution de la prochaine tâche qui doit être allouée, plutôt que de l'allouer au premier processeur inactif, alors, si m représente le nombre de processeurs du système, Cho et Sahni [CS80] prouvent que la qualité des ordonnancements obtenus est :

$$\frac{C^{fin-exec}_{max}}{C^{optimal}_{max}} \leq 1 + \frac{\sqrt{2m-2}}{2} \text{ et } (m > 2).$$

Toujours pour des algorithmes de liste, une étude du comportement de l'algorithme LPT[4] pour ce même problème est présentée dans [GIS77]. Les auteurs prouvent que la qualité d'un ordonnancement LPT par rapport à un ordonnancement optimal est :

$$\frac{C^{LPT}_{max}}{C^{optimal}_{max}} \leq 2 - \frac{2}{1+m}.$$

[3] définition en annexe.

[4] *Largest Processing Time* décrit en annexe

A titre de comparaison, pour des processeurs identiques, Graham prouve [Gra69] que la qualité d'un ordonnancement LPT est :

$$\frac{C_{max}^{LPT}}{C_{max}^{optimal}} \leq \frac{4}{3} - \frac{1}{3m}.$$

Pour ce même algorithme Dobson propose une borne qui tient compte du rapport entre la taille de la plus grosse tâche et le maximum des dates de fin d'exécution des tâches [Dob84].

Toujours pour le même problème, Friesen et Langston [FL83] utilisent une version modifiée de l'algorithme MULTIFIT et parviennent à générer des ordonnancements dont la qualité est :

$$\frac{C_{max}^{MULTIFIT}}{C_{max}^{optimal}} \leq \frac{7}{5}.$$

Lorsque la fonction à minimiser est le *makespan*, la moyenne pondérée, et non pondérée, des temps de flots[5], Horowitz et Sahni proposent des algorithmes exacts [HS76]. Les problèmes traités étant NP-complets, ils présentent également une famille d'algorithmes approchés qui déterminent des solutions garanties proches de l'optimal :

$$\frac{C_{max}^{A_k}}{C_{max}^{optimal}} \leq 1 + \frac{1}{k}.$$

La complexité de ces algorithmes est exponentielle en le nombre de processeurs $O(n^{2m}k^{m-1})$. Pour une qualité d'ordonnancement comparable, mais une complexité inférieure, Hochbaum et Shmoys proposent une famille d'algorithmes $\{A_\epsilon\}$ basés sur l'étude d'un problème dual [HS88]. Ils ramènent le problème initial au problème du *bin packing*[6] dans lequel la taille des boites dépend directement de la vitesse des processeurs. Cependant, ce problème est lui aussi NP-complet y compris lorsque les processeurs sont identiques (c'est-à-dire lorsque les boites sont de tailles égales). La solution proposée est une résolution d'une version relaxée de ce problème. Les boites étant de tailles fixées (S_i), la recherche d'un ordonnancement optimal reviendrait à déterminer une partition de l'ensemble des tâches de façon à ce que le remplissage d'une boite B_i soit au plus égal à S_i. La relaxation de ce problème autorise un dépassement de la capacité des boites, et l'algorithme recherche alors une partition de l'ensemble des tâches de telle sorte que le remplissage d'une boite B_i soit inférieur ou égal à $(1 + \epsilon)S_i$. Ces algorithmes sont polynomiaux en le nombre de tâches et exponentiels en l'inverse de la valeur de ϵ.

L'ordonnancement de tâches indépendantes sur des systèmes à base de processeurs uniformes a été étudié sous d'autres formes, par exemple, Błażewicz et al. [BDSW90] présentent un algorithme de complexité $O(n(m + log(n))$ pour le problème de l'ordonnancement avec préemption de tâches indépendantes nécessitant l'utilisation simultanée de plusieurs processeurs pour leur exécution.

[5] *mean flow time* et *weighted mean flow time* definis en annexe

[6] ce problème est défini en annexe sous le nom du problème de remplissage de boites

4.2.3 Ordonnancement avec contraintes de précédence

Le nombre de travaux traitant de l'ordonnancement de tâches soumises à des contraintes de précédence sur des systèmes à base de processeurs uniformes est nettement plus restreint. Jean-Louis Baer est l'un des premiers à s'intéresser à ce type de problèmes [Bae74]. Il propose des algorithmes pour ordonnancer des graphes sur deux processeurs uniformes. L'objectif des algorithmes présentés est de minimiser le maximum des dates de fin d'exécution des tâches. Les tâches sont identiques et la préemption n'est pas autorisée. Dans un premier temps l'auteur présente un algorithme qui construit des ordonnancements optimaux, pour des ensembles de tâches indépendantes, sur deux processeurs de vitesses respectives m et n (entières). Il généralise ensuite l'algorithme de Hu [Hu61] (utilisé avec un nombre de processeurs restreint à deux) pour des vitesses égales à 1 et 2. Cet algorithme, nommé TREE-1-2, produit des ordonnancements optimaux. L'idée de base est de répéter une séquence type, formée de, deux tâches sur le processeur ayant une vitesse égale à 2, et de une tâche sur le processeur de vitesse 1 (voir figure 4.1). Dans le cas où les vitesses sont égales à 1 et 3, Baer propose une procédure, nommée SUBTREE-1-3, qui permet la construction d'ordonnancements tels que :

$$C_{max}^{optimal} \leq C_{max}^{SUBTREE-1-3} \leq C_{max}^{optimal} + 1.$$

Il conjecture qu'il doit être possible de construire de telles procédures pour n'importe quel ensemble $\{1, v\}$ de vitesses (avec v entier).

Pour les graphes orientés sans cycle, l'auteur construit un algorithme, nommé SUBGR-1-2, dont la structure est la même que celle de TREE-1-2, mais pour lequel la liste est construite au moyen de l'étiquetage proposé dans l'algorithme de Coffman et Graham [CG72]. Pour un système constitué de deux processeurs dont les vitesses sont égales à 1 et 2, la qualité des ordonnancements obtenus est :

$$\frac{C_{max}^{SUBGR-1-2}}{C_{max}^{optimal}} \leq \frac{6}{5}.$$

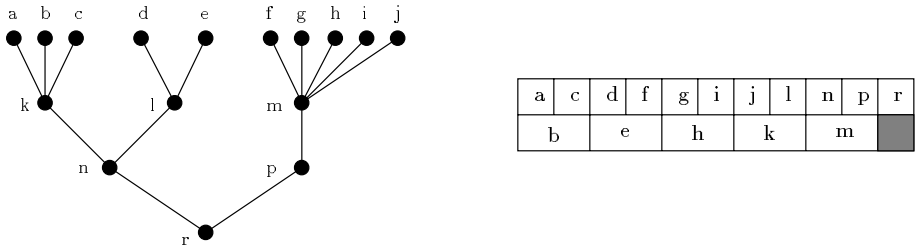


FIG. *4.1 - : Un exemple d'ordonnancement obtenu avec l'algorithme TREE-1-2.*

Comme pour un ensemble de tâches indépendantes [LL74a, LL74b], la qualité des ordonnancements produits par des algorithmes de liste pour des graphes orientés sans cycle est telle que :

$$\frac{C_{max}^{liste}}{C_{max}^{optimal}} \leq 1 + \frac{\max_i v_i}{\min_i v_i} - \frac{\max_i v_i}{\sum_i v_i}.$$

Toujours pour l'ordonnancement de graphes orientés sans cycle formés de tâches identiques sur deux processeurs uniformes (le plus rapide utilise f unités de temps pour exécuter une tâche et le plus lent s unités de temps ($s \geq f$)), Gabow étudie dans [Gab88] le comportement des algorithmes HLF (*Highest Level First* et HLA[7]).

L'auteur montre que dans le cas où la différence $\Delta = s - f$ est égale à 1, l'algorithme HLA permet de construire des ordonnancements optimaux. La complexité de cet algorithme est en $O(2^L(n+a))$ où L représente le nombre de niveaux du graphe, n le nombre de noeuds et a le nombre d'arcs. Il remarque cependant que si un ordonnancement optimal ne contient aucune période d'inactivité, il peut être construit en un temps linéaire par HLA (exactement en $O(n+m)$).

Dans le cas où les vitesses prises en compte sont plus générales, Gabow s'est intéressé plus particulièrement aux algorithmes fournissant des ordonnancements proches de l'optimal. Ainsi, si l'ordonnancement obtenu pour deux processeurs identiques est adapté au cas de deux processeurs uniformes, la qualité de l'ordonnancement obtenu est :

$$\frac{C_{max}^{optimal\_identique}}{C_{max}^{optimal\_uniforme}} \leq 2 - \frac{f}{s}.$$

L'utilisation de HLF pour deux processeurs uniformes permet d'améliorer la borne pour deux cas particuliers :

$$\frac{C_{max}^{HLF}}{C_{max}^{optimal}} \leq \frac{5}{4} \quad \text{lorsque } \frac{f}{s} = \frac{1}{2}.$$

$$\frac{C_{max}^{HLF}}{C_{max}^{optimal}} \leq \frac{6}{5} \quad \text{lorsque } \frac{f}{s} = \frac{2}{3}.$$

[7] *Highest Level first with Abstentions* décrits en annexe

4.2.4 Résumé des résultats

Nous présentons dans le tableau suivant une compilation des résultats d'ordonnancement avec processeurs uniformes pour des problèmes avec ou sans relation de précédence.

Problème	**Résultat**	**Complexité**	**Remarques**	**Références**
$Q_m \mid\mid C_{max}$	optimal	$O(m^n)$		[HS76]
$Q_m \mid\mid C_{max}$	$\omega_{A_\epsilon} \leq (1+\epsilon)\ \omega_{opt}$	$O(n^{2m}\epsilon^{1-m})$		—-
$Q_m \mid\mid C_{max}$	$\omega_{LPT} \leq (2 - \frac{2}{1+m})\ \omega_{opt}$	$O(n\ log(n)$	LPT	[GIS77]
$Q_m \mid\mid C_{max}$	$\omega_{MULTIFIT} \leq \frac{7}{5}\ \omega_{opt}$	$O(n\ log(n)$	MULTIFIT	[FL83]
$Q_m \mid\mid C_{max}$	$\omega_{A_\epsilon} \leq (1+\epsilon)\ \omega_{opt}$	$O(mn^{\frac{10}{\epsilon^2}+3})$		[HS88]
$Q_2 \mid tree, p_j = 1 \mid C_{max}$	optimal	$O(n)$	vitesses = {1,2}	[Bae74]
$Q_2 \mid tree, p_j = 1 \mid C_{max}$	$\omega_{SUBTREE-1-3} \leq \omega_{opt} + 1$	polynomial	vitesses = {1,3}	—-
$Q_2 \mid prec, p_j = 1 \mid C_{max}$	$\frac{\omega_{SUBGR-1-2}}{\omega_{opt}} \leq \frac{6}{5}$	$O(n^2)$	vitesses = {1,2}	—-
$Q_2 \mid prec, p_j = 1 \mid C_{max}$	$\frac{\omega_{P2}}{\omega_{opt}} \leq 2 - \frac{f}{s}$	$O(n+a)$	$f \leq s$	[Gab88]
$Q_2 \mid prec, p_j = 1 \mid C_{max}$	$\frac{\omega_{HLF}}{\omega_{opt}} \leq \frac{5}{4}$	$O(n)$	$\frac{f}{s} = \frac{1}{2}$	—-
$Q_2 \mid prec, p_j = 1 \mid C_{max}$	$\frac{\omega_{HLF}}{\omega_{opt}} \leq \frac{6}{5}$	$O(n)$	$\frac{f}{s} = \frac{2}{3}$	—-

4.3 L'ordonnancement des arbres complets avec communications

Le problème de l'ordonnancement de graphes de tâches prenant en compte les communications a été considéré notamment dans [PY88, Chr89, JKS89, CP91, JR92]. Dans tous ces travaux, le critère de performances retenu a été la **minimisation du maximum des dates de fin d'exécution** des tâches, et les systèmes multiprocesseurs sont tous formés de **processeurs identiques**.
Dans [PY88], le nombre de processeurs considérés est **non borné**. Pour des graphes dont les tâches sont de durées unitaires et dont les communications sont égales à une constante τ, les auteurs décrivent un algorithme qui produit des ordonnancements distants d'un facteur au plus deux de l'optimal. De plus, cet algorithme produit des ordonnancements asymptotiquement optimaux pour des arbres binaires complets.
Cette borne asymptotique est améliorée dans [JKS89] par un facteur deux pour des arbres binaires complets. Enfin, dans [JR92], les auteurs prouvent que le problème qui consiste à ordonnancer des arbres dont les tâches sont de durées unitaires avec communications reste NP-difficile pour des arbres binaires et des communications uniformes et pour des arbres binaires complets et des communications variables.
Pour l'ordonnancement d'arbres formés de tâches unitaires sur un système formé de processeurs uniformes, seul Baer présente dans [Bae74] un résultat d'optimalité pour deux processeurs (de vitesses respectives 1 et 2) sans tenir compte des communications. L'algorithme employé par Baer est une forme adaptée de l'algorithme de Hu [Hu61].
Le problème que nous considérons est celui de l'ordonnancement des arbres k-aires complets sur un système multiprocesseur formé de deux processeurs uniformes dont les vitesses, entières, sont notées v_1 et v_2. Ce problème est noté dans la littérature :

$$Q2\ \{v_1, v_2\} \mid arbres\ complets,\ p_j = 1,\ c = 1 \mid C_{max}.$$

Les principaux résultats sont regroupés dans le tableau ci-après.

Problèmes	**Résultats et Remarques**	**Complexité et NP-complétude**	**Références**
$P_\infty \mid in-tree,\ p_j = 1,\ c(n) \mid C_{max}$	arbres binaires	NP-difficile	[JR92]
$P_\infty \mid in-tree,\ p_j = 1,\ c_{ij} \mid C_{max}$	arbres binaires complets	NP-difficile	[JR92]
$Q2\ \{1,2\} \mid in-tree,\ p_j = 1 \mid C_{max}$	optimal	$O(n)$	[Bae74]

4.4 Notations et définitions

Nous considérons dans cette partie des arbres dont toutes les tâches sont **identiques**. Le système multiprocesseur est formé de deux processeurs **uniformes**. Nous supposons de plus que les communications inter-processeurs nécessitent la même durée, **une unité de temps**, quel que soit le sens de ces communications. Le processeur le plus rapide sera noté $\boldsymbol{P_r}$ et le processeur lent $\boldsymbol{P_l}$. Leur vitesses respectives sont notées $\boldsymbol{v_r}$ et $\boldsymbol{v_l}$. Dans la suite, nous ne manipulerons pas les vitesses, qui correspondent au nombre de tâches exécutées

par unité de temps, mais **l'inverse de ces vitesses**, c'est-à-dire au **nombre d'unités de temps nécessaires** pour chaque processeur **pour exécuter une tâche**. Nous notons ces grandeurs $\boldsymbol{a_r}$ et $\boldsymbol{a_l}$. Par la suite, a_r et a_l seront appelées **anti-vitesses** par commodité de langage. Dans ce contexte, une tâche est dite **unitaire** lorsqu'elle nécessite a unités de temps pour son exécution sur un processeur dont la vitesse est égale à $\frac{1}{a}$. Ceci signifie que dans le cadre d'un système formé de processeurs uniformes, l'attribut des tâches correspond davantage à une **granularité** qu'à un temps d'exécution.

4.5 Quelques résultats préliminaires

Parmi l'ensemble des travaux existants sur des problèmes prenant en considération les processeurs uniformes, l'étude de l'ordonnancement des arbres UET par Jean-Louis Baer a été décrite dans la section précédente (tour d'horizon des travaux prenant en compte les processeurs uniformes). L'auteur énonce deux théorèmes concernant la forme de certains ordonnancements qui constituent un ensemble **dominant**[8].

Théorème A :
Dans le cas où les inverses des vitesses des deux processeurs sont égales à 1 et v (entier quelconque), il existe pour l'ordonnancement d'un arbre une solution optimale telle que les deux processeurs soient occupés jusqu'à une date K_2 et ensuite le plus rapide des processeurs est seul actif jusqu'à une date $K_1 = C$ (où C représente le maximum des dates de fin d'exécution des tâches).

Théorème B :
Si l'ensemble de l'inverse des vitesses est formé par deux entiers différents de 1 alors, dans le cas général, il est possible qu'il n'existe aucune solution optimale ne contenant pas de périodes d'inactivité sur le processeur le plus rapide.

La preuve de ces deux théorèmes est donnée dans [Bae74]. Nous allons étendre ces résultats dans le cas des arbres dont les tâches sont encore unitaires mais pour lesquels les communications sont prises en compte et sont unitaires.

Théorème 1 :
En présence de communications unitaires, pour deux processeurs uniformes avec des anti-vitesses a_r et a_l (avec $a_r \leq a_l$):

- **Cas 1, $\boldsymbol{a_r = 1 = a_l}$** (processeurs identiques)
  Il existe toujours un ordonnancement optimal tel que les deux processeurs soient en activité jusqu'à une date K et qu'ensuite seulement l'un des deux soit actif jusqu'à la date C (maximum des dates de fin d'exécution des tâches).

- **Cas 2, $\boldsymbol{a_r = 1}$ et $\boldsymbol{a_l}$ entier**
  Il existe toujours un ordonnancement optimal tel que P_r soit toujours actif, mais il

[8] définition en annexe

est possible que dans tout ordonnancement optimal une période d'inactivité apparaisse entre l'exécution de deux tâches sur P_l.

– **Cas 3, $a_r, a_l \neq 1$ et entiers**
Il n'est pas toujours possible de trouver un ordonnancement optimal qui garantisse une activité constante pour le processeur le plus rapide.

Preuve
Lorsque les processeurs sont identiques, l'algorithme présenté précédemment (MAJYC) construit des ordonnancements tels que le processeur exécutant la racine, et identifiable à P_r est toujours actif. De plus, il n'existe aucune communication de P_r vers P_l, et les ordonnancements produits par MAJYC sont optimaux, donc la propriété est vérifiée.

Pour le cas 2, nous supposons acquis qu'il existe toujours un ordonnancement optimal pour lequel la racine de l'arbre est exécutée sur le processeur le plus rapide. Parmi ce sous-ensemble d'ordonnancements optimaux, considérons un ordonnancement contenant des périodes d'inactivité sur les deux processeurs. Toutes les périodes d'inactivité qui apparaissent sur le processeur le plus rapide sont dues à des communications. De façon à les éliminer, les tâches communicantes exécutées par P_l doivent être transférées de P_l vers P_r par le mécanisme décrit en figure 4.2.

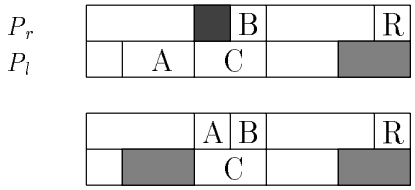


FIG. *4.2 - : Élimination des périodes d'inactivité présentes sur P_r.*

Ainsi, toutes les périodes d'inactivité peuvent être éliminées dans l'ordonnancement de P_r, mais ce n'est pas le cas pour le processeur lent comme on peut le constater sur l'exemple illustré par la figure 4.3.

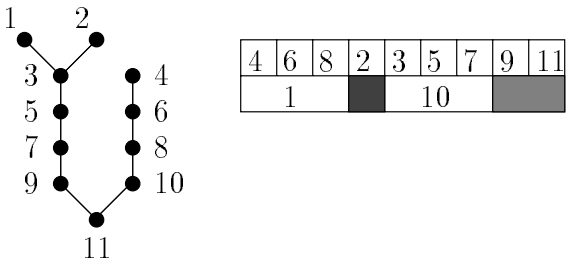


FIG. *4.3 - : Exemple d'une période d'inactivité persistante sur P_l.*

Dans le dernier cas, on ne peut que constater qu'il n'est pas toujours possible d'éliminer les périodes d'inactivité sur P_r (voir figure 4.4 (les vitesses sont égales à 2 et 3)). □

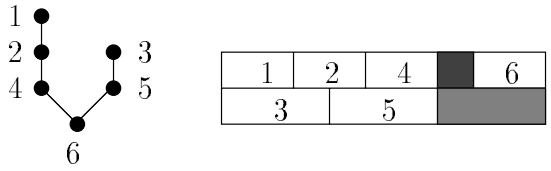


FIG. *4.4 - : Un ordonnancement optimal avec une période d'inactivité sur P_r.*

Le résultat qui suit correspond au calcul d'une borne minimum de la date de fin d'exécution de la racine pour un arbre formé de n tâches.

Théorème 2 :
Soient deux processeurs uniformes, P_r et P_l, dont les inverses des vitesses respectives sont a_r et a_l (entières et supérieures ou égales à 1) telles que $a_r \leq a_l$. Soit un arbre formé de n tâches identiques. Chaque tâche requiert a_r unités de temps pour son exécution sur P_r (et a_l sur P_l). Alors, si l'on note $C_r = ((n-1)a_l)div(a_r + a_l)$ et $C_l = ((n-1)a_r)div(a_r + a_l)$, le minimum des maximums des dates de fin d'exécution de tâches sur ce système biprocesseur vérifie :

Si $C_r + C_l = n - 2$ *:* $C_{max} \geq a_l(C_l + 1) + (a_r + 1)$ *si* $a_r(C_r + 1) > a_l(C_l + 1) + 1$, *et,*

$C_{max} \geq a_r(C_r + 2)$ *si* $a_r(C_r + 1) \leq a_l(C_l + 1) + 1$.

Si $C_r + C_l = n - 1$ *:* $C_{max} \geq a_r(C_r + 1) + 1$.

Remarque 1
Si de tels ordonnancements existent, le nombre total de tâches allouées à chaque processeur, racine mise à part, se déduit de ce théorème.
Si $C_r + C_l = n - 2$ et $a_r(C_r + 1) \leq a_l(C_l + 1) + 1$ alors le nombre de tâches allouées à P_r (resp. P_l) est : $N_r = C_r + 1$ (resp. $N_l = C_l$). Si $C_r + C_l = n - 2$ et $a_r(C_r + 1) > a_l(C_l + 1) + 1$ alors $N_r = C_r$, et $N_l = C_l + 1$. Enfin, Si $C_r + C_l = n - 1$ alors $N_r = C_r + 1$ et $N_l = C_l$.

Remarque 2
Pour un système multiprocesseur dont les inverses des vitesses sont entières et différentes, l'ensemble des ordonnancements pour lesquels la racine est exécutée sur le processeur le plus rapide (P_r), est un ensemble **dominant**. En effet, les inverses des vitesses étant entiers, l'exécution de la racine sur un processeur qui n'est pas le plus rapide nécessite au moins une unité de temps de plus que pour son exécution sur P_r. Cette unité de temps peut être mise à profit pour effectuer une communication de ce processeur vers P_r.

Preuve du théorème 2
Intuitivement, au mieux, l'ensemble des tâches de cet arbre peuvent être équitablement ré-

parties entre les deux processeurs, en terme de temps d'exécution. La racine est cependant exclue de cette répartition puisque aucune autre tâche appartenant à cet arbre ne peut être exécutée en même temps.

Oublions pour un instant les contraintes de précédence. Équilibrer la charge des processeurs est équivalent à leur allouer un nombre de tâches proportionnel à leur puissance respective, c'est-à-dire allouer $C_r = ((n-1)a_l)div(a_r+a_l)$ tâches à P_r et $C_l = ((n-1)a_r)div(a_r+a_l)$ tâches à P_l. Cependant, $C_r + C_l$ ne représente pas toujours la totalité des $(n-1)$ tâches, tout au plus sait-on que $C_r + C_l \geq n-2$.

Si $C_r + C_l = n-1$ alors l'exécution de C_r tâches sur P_r nécessite la même durée que l'exécution de C_l tâches sur P_l. La racine est exécutée sur P_r, ce qui rajoute au *makespan* a_r unités de temps, et P_l communique des données à P_r pour l'exécution de la racine, ce qui nécessite une unité de temps. Au total, $C_{max} \geq a_r C_r + a_r + 1$.
Si $C_r + C_l = n-2$, il reste une tâche à allouer T. Il suffit alors de considérer son exécution sur chacun des deux processeurs, et de l'allouer à celui qui permet la minimisation de la date de fin d'exécution des tâches : C_{max}.

On considère, d'après la Remarque 2, que la racine est exécutée sur P_r. De manière évidente, T est allouée à P_l si son exécution sur P_l permet de minimiser la date de début d'exécution de la racine. C'est le cas si le temps d'exécution de T sur P_l plus une unité de temps pour la communication est strictement inférieur au temps d'exécution de cette même tâche sur P_r, ce qui se traduit par la relation : $a_r(C_r+1) > a_l(C_l+1)+1$. Finalement, si cette relation est vérifiée, T est allouée à P_l et $a_l(C_l+1) > a_r C_r$ donc $C_{max} \geq a_l(C_l+1) + a_r + 1$ puisque la racine est exécutée sur P_r. Enfin, si $a_r(C_r+1) \leq a_l(C_l+1)+1$, T est allouée à P_r et une borne minimum pour le *makespan* est : $C_{max} \geq a_r(C_r+2)$. □

4.6 Application aux arbres complets

Nous considérons pour cette partie des arbres complets, d'arité k et de hauteur h, c'est-à-dire formés de $n = (k^h-1)/(k-1)$ tâches identiques. Le système multiprocesseur est formé de deux processeurs. A partir du Théorème 2 et de la Remarque 1, il est possible de calculer les valeurs N_l et N_r. Nous exposons dans une première partie un algorithme qui permet d'atteindre la borne du *makespan* (calculée à partir du Théorème 2). La condition d'application de cet algorithme est $N_l \geq h-1$. Dans une seconde partie, nous mettons en œuvre une technique qui permet de résoudre le cas où $N_l < h-1$.

4.6.1 Algorithme du cas général

Cet algorithme commence par le calcul de N_r et N_l. Pour l'allocation des N_r tâches à P_r, on utilise un algorithme qui n'alloue que des sous-arbres complets.

Au niveau i de l'arbre, les tâches sont des racines d'arbres complets de hauteur $h-i+1$. L'algorithme alloue des sous-arbres dont le niveau des racines va en augmentant, c'est-à-dire

dont la hauteur va en diminuant. Cette allocation s'arrête lorsque le nombre total de tâches allouées est égal à N_r. Si l'on note N_r^i le nombre de sous-arbres complets de hauteur i et de niveau $h+1-i$ alloués à P_r, alors :

$$N_r^{h-1} = (N_r)\, div \left(\frac{k^{h-1}-1}{k-1}\right),$$

$$N_r^{h+1-i} = \left(N_r - \sum_{j=h-1}^{j=h+2-i} N_r^j \frac{k^j-1}{k-1}\right) div \left(\frac{k^{h+1-i}-1}{k-1}\right).$$

L'allocation se termine au niveau l pour lequel : $N_r - \sum_{j=h-1}^{j=l} N_r^j \frac{k^j-1}{k-1} = 0$.

Les sous-arbres complets ainsi alloués à P_r sont ordonnancés dans l'ordre décroissant de leur poids.

Dans le cas, le plus courant, où $N_l \geq h-1$, le nombre de tâches allouées à P_r est égal à N_r défini dans la Remarque 1. L'ordonnancement obtenu est optimal si et seulement si toutes les communications de P_r vers P_l sont recouvertes. En effet, dans ce cas de figure, aucune période d'inactivité n'apparaît sur P_l, et donc l'exécution de toutes les tâches allouées à P_r (exceptée éventuellement la racine) se fait sans attente. Ce que nous allons prouver par récurrence sur la hauteur des arbres complets.

Lorsque $h = 1$, l'arbre n'est constitué que de la racine et le résultat est évident.
Supposons que l'ordonnancement produit par l'algorithme vérifie cette propriété pour les arbres k-aire complets de hauteur h.
Montrons-le pour des arbres de hauteur $h+1$. Deux cas peuvent survenir :

- si le nombre de sous-arbres complets de hauteur h alloués à P_r est inférieur strictement à $k-1$, alors P_l commence par exécuter les arbres complets de hauteur h qui lui sont entièrement alloués et les communications de P_r vers P_l sont recouvertes de façon évidente.

- Si le nombre de sous-arbres complets de hauteur h alloués à P_r est égal à $k-1$ alors le dernier sous-arbre complet de hauteur h est partagé entre P_r et P_l. Cependant, le nombre de nœuds appartenant à ce sous-arbre et alloués à P_r est strictement inférieur au nombre de nœuds alloués à P_r lors de l'ordonnancement d'un arbre k-aire complet de hauteur h. Donc, par hypothèse de récurrence, il n'apparaît dans l'ordonnancement de P_l aucune période d'inactivité due à une attente de communication. □

Exemple

Nous considérons un arbre d'arité 4 et de hauteur 4. Le nombre total de nœuds de l'arbre est égal à $n = 85$. Les inverses des vitesses des processeurs P_r et P_l sont respectivement

$a_r = 3$ et $a_l = 8$.
D'après le Théorème 2 et la Remarque 1 : $C_r = (84 \times 8)div(3+8) = 61$ et $C_l = (84 \times 3)div(11) = 22$. De plus $C_r + C_l = 61 + 22 = 83 = n - 2$. Il reste donc une tâche à allouer. Or, $Max(a_r C_r, a_l C_l) = 3 \times 61 = a_r C_r$ et $a_r(C_r + 1) = 3 \times 62 = 186 > 175 = 8 \times 23 + 1 = a_l(C_l + 1) + 1$. Donc, la tâche restante est allouée à P_l. En conséquence, $N_r = 61$ et $N_l = 23$. La phase d'allocation commence par les sous-arbres complets de niveau 2, de hauteur 3 et dont le poids est égal à 21 : $N_r^3 = (61)div(21) = 2$. Il reste $61 - 42 = 19$ tâches. $N_r^2 = (19)div(5) = 3$, enfin $N_r^1 = (4)div(1) = 4$. On vérifie que l'on a bien l'égalité $N_r = 61 = N_r^3 \times 21 + N_r^2 \times 5 + N_r^1 \times 1 = 42 + 15 + 4$. Sont ordonnancés d'abord les quatre sous-arbres complets de niveau 4 et de hauteur 1, puis les trois sous-arbres de niveau 3 et de hauteur 2, enfin les deux sous-arbres de niveau 2 et de hauteur 3. L'arbre complet, les sous-arbres complets choisis et le diagramme de Gantt correspondant à cet ordonnancement sont représentés en figure 4.5.

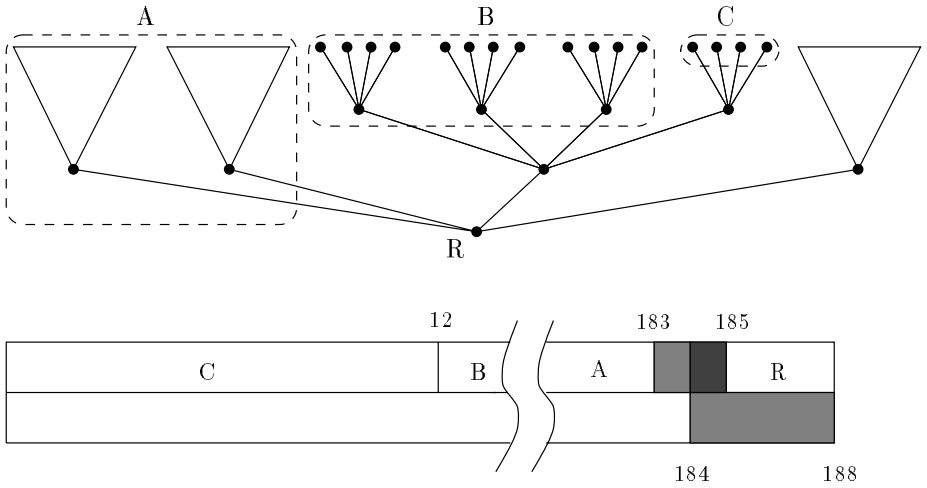


FIG. *4.5 - : Un exemple d'application de l'algorithme lorsque* $N_l \geq h - 1$.

En conclusion, les ordonnancements produits par cet algorithme lorsque $N_l \geq h - 1$ sont optimaux. Cependant, il existe des cas pour lesquels cet algorithme produit des ordonnancements qui ne sont pas optimaux, lorsque $N_l < h - 1$, c'est-à-dire lorsque le rapport des vitesses est de l'ordre du nombre de nœuds de l'arbre divisé par la hauteur : $\frac{v_r}{v_l} = O(\frac{n}{h})$. Ce cas particulier apporte des difficultés dont la résolution ouvre des portes pour la mise en œuvre d'algorithmes pour des arbres quelconques. Nous présentons dans le paragraphe suivant une étude précise des cas particuliers pour lesquels l'algorithme présenté précédemment ne produit pas systématiquement un ordonnancement optimal.

Nous présentons dans la suite un exemple d'application de l'algorithme exposé ci-dessus, puis le cas particulier pour lequel $N_l < h - 1$ est traité dans le paragraphe suivant.

4.6.2 Cas particuliers

L'algorithme présenté précédemment n'est valable que lorsque le nombre de tâches allouées à P_l est supérieur ou égal à $h-1$ (où h est la hauteur de l'arbre). Dans le cas contraire, des problèmes peuvent survenir.

Exemple

L'exemple de la figure 4.6 présente l'un de ces cas limites et les problèmes qu'ils entraînent pour l'allocation des tâches aux processeurs. On remarque que dans les deux cas, quatre tâches ont été allouées à P_l, pourtant seul le second ordonnancement permet d'atteindre l'optimal. L'explication est donnée par le calcul du niveau, l_{der}, de la dernière tâche, T_{der}, exécutée par P_l. Cette dernière tâche précède exactement $l_{der}-2$ tâches (si l'on exclue la racine).

Résultats préliminaires

Corollaire 1
Pour un arbre donné, s'il existe un ordonnancement optimal tel que $C_{max} = a_r(C_r+2)$ *alors aucune période d'inactivité ne doit apparaître dans l'ordonnancement de* P_r *et si l'on note* l_{der} *le niveau de la dernière tâche exécutée par* P_l *:* $a_r(C_r+1-(l_{der}-2)) \geq a_l C_l+1$ *c'est-à-dire* $l_{der} \leq C_r + 1 - \left\lceil \frac{a_l C_l + 1}{a_r} \right\rceil + 2$.

Remarque
La quantité $l_{der}-2$ correspond au nombre de tâches allouées à P_r et qui ne peuvent pas être exécutées tant que la communication en provenance de T_{der} n'est pas achevée (soit une unité de temps après la date de fin d'exécution de cette tâche).

Corollaire 2
Pour un arbre donné, s'il existe un ordonnancement optimal tel que $C_{max} = a_l(C_l+1)+a_r+1$ *ou* $C_{max} = a_r(C_r+1)+1$, *alors une période d'inactivité pour attente de communication apparaît sur* P_r *juste avant l'exécution de la racine, seule tâche exécutée après cette attente,*

donc : $l_{der} = 2$.

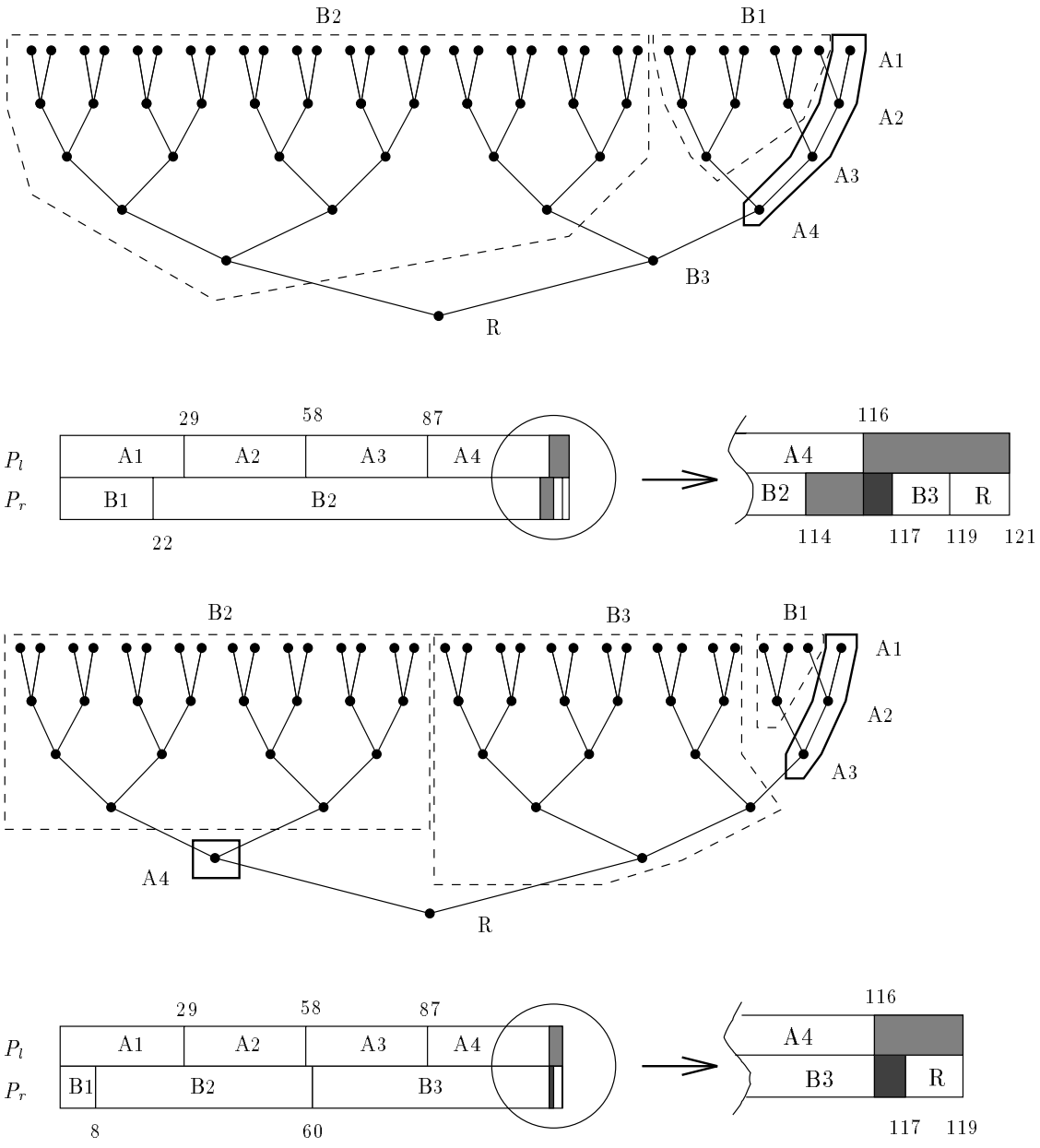


FIG. *4.6 - : $a_r = 2$, $a_l = 29$ et $h = 6$.*

Remarque
La validité du Théorème 2 ne se limite pas aux arbres complets, donc les bornes minimales calculées sont aussi valables pour des arbres quelconques dont les tâches sont identiques. De la même façon, l'application des corollaires ne se limite pas aux arbres complets.

Appliquons cette relation à l'exemple de la figure 4.6 pour lequel $h = 6$.

– $C_r = (62 \times 29)div(31) = 58$ et

- $C_l = (62 \times 2) div (31) = 4$ donc
- $C_r + C_l = n - 1$
- $C_{max} = a_r(C_r + 1) + 1 = 2 \times (58 + 1) + 1 = 119$
- $l_{der} = 2$

Résolution du problème

Les deux corollaires présentés ci-dessus fournissent une base de départ pour l'ordonnancement d'arbres complets lorsque $N_l < h - 1$. T_{der} représente toujours la dernière tâche exécutée par P_l. La première phase consiste à calculer N_r, N_l, C_{max} et l_{der}.

Si $N_l \geq h - (l_{der} - 1)$ alors, bien que P_l empêche l'exécution de certaines tâches allouées à P_r, ce dernier n'attend pas car il lui en reste suffisamment pour recouvrir la communication en provenance de T_{der}. Ceci signifie que l'allocation d'une chaîne de N_l tâches à P_l, dont la première est une feuille, garantie l'optimalité de l'ordonnancement construit.

Si $N_l < h - (l_{der} - 1)$ et si $N_l = 1$ alors l'ordonnancement est obtenu après une simple vérification. Une tâche sera exécutée sur P_l si cela minimise la date de fin d'exécution de l'arbre, ce qui se traduit par :

$$a_l \leq (n - (h - 1))\, a_r - 1.$$

On suppose donc que $N_l > 1$, le choix des tâches allouées à P_l se fait de la façon suivante :

T_{der}, la dernière tâche que P_l doit exécuter, est choisie au niveau l_{der} dans un sous-arbre différent de celui où les $N_l - 1$ autres tâches sont choisies. Ces dernières sont choisies de telle sorte qu'aucune de ces tâches n'ait de prédécesseur alloué à P_r, avec la contrainte supplémentaire que le nombre de feuilles allouées à P_l soit minimum (ce qui est réalisé par l'allocation de sous-arbres complets comme dans le cas précédent pour les tâches allouées à P_r). L'ordonnancement des différentes tâches est alors le suivant :

Les premières tâches ordonnancées sur P_r sont celles qui appartiennent au sous-arbre dont T_{der} est racine, alors que P_l commence par l'exécution des $N_l - 1$ autres tâches. Cependant, pour être sûr que cet ordonnancement permette d'atteindre la borne minimum du *makespan* calculée, il faut s'assurer qu'aucune période d'inactivité n'apparaît dans l'ordonnancement de P_l avant l'exécution de T_{der}. C'est effectivement le cas si l'exécution des $N_l - 1$ autres tâches recouvre l'exécution de toutes les tâches appartenant au sous-arbre complet dont T_{der} est racine, plus la communication, ce qui se traduit par :

$$(N_l - 1)a_l \geq \left(\frac{k^{h - l_{der} + 1} - 1}{k - 1} - 1 \right) a_r + 1.$$

Cette condition est vérifiée dès que $k \geq 3$ ou $N_l \geq 3$ ou $l_{der} \geq 3$. Nous montrons par contradiction pour chacun de ces trois cas.

- Si $N_l \geq 3$ et si la condition n'est pas vérifiée alors

$$(N_l - 1)a_l < \left(\frac{k^{h-l_{der}+1} - 1}{k-1} - 1\right) a_r + 1,$$

  ce qui signifie que l'exécution d'au moins deux tâches par P_l nécessite moins de temps que l'exécution d'un sous-arbre complet de hauteur, au plus, $h-1$ par P_r, or ceci est en contradiction avec le fait que l'exécution d'une chaîne de longueur N_l par P_l entraîne une attente sur P_r c'est-à-dire contredit l'hypothèse selon laquelle $N_l < h - (l_{der} - 1)$. Donc, lorsque $N_l \geq 3$ la condition est vérifiée.

- Si $k \geq 3$ et si l'exécution d'une tâche par P_l ne peut recouvrir l'exécution d'un sous-arbre complet de hauteur au plus $h-1$ par P_r alors l'exécution d'une tâche supplémentaire par P_l (nommément T_{der}, accolée à la première tâche) n'entraîne pas d'attente sur P_r puisqu'il lui reste $k-2$ sous-arbres complets de hauteur $h-1$ à exécuter. Donc, la condition est vérifiée lorsque $k \geq 3$.

- Enfin, si $l_{der} \geq 3$ et si l'exécution des $N_l - 1$ tâches ne permet pas de recouvrir l'exécution d'un sous-arbre de hauteur $h-(l_{der}-1)$ alors, il reste encore au moins un sous-arbre de la même hauteur prêt à être exécuté par P_r, ce qui est en contradiction avec l'hypothèse $N_l < h - (l_{der} - 1)$. Donc dans ce cas aussi, la condition est vérifiée.

Finalement, le seul cas qui puisse poser problème est celui pour lequel, $N_l = 2$, $k = 2$ et $l_{der} = 2$.

Si $C_l + C_r = n - 1$ alors $\dfrac{a_l}{a_r} = 2^{h-1} - 2$, en effet, $\dfrac{a_r(n-1)}{a_r + a_l} = 2$. Donc, T_{der} devra attendre une unité de temps avant de pouvoir commencer son exécution, ce qui va décaler sa date de terminaison de une unité de temps, et donc entraîner une attente de deux unités de temps sur P_r, une unité de temps par manque de calcul disponible plus une unité de temps pour attente de communication. L'ordonnancement dans lequel T_{der} est exécutée sur P_l n'est pas optimal si et seulement si $a_r = 1$, un ordonnancement optimal est alors obtenu en faisant migrer T_{der} de P_l vers P_r. La figure 4.7 illustre une telle situation, avec $a_r = 1$ et $a_l = 14$ ($T_{der} = B$). Dès que $a_r > 1$, l'ordonnancement est optimal puisque l'ajout de T_{der} à P_r ne

peut qu'augmenter la date de fin d'exécution.

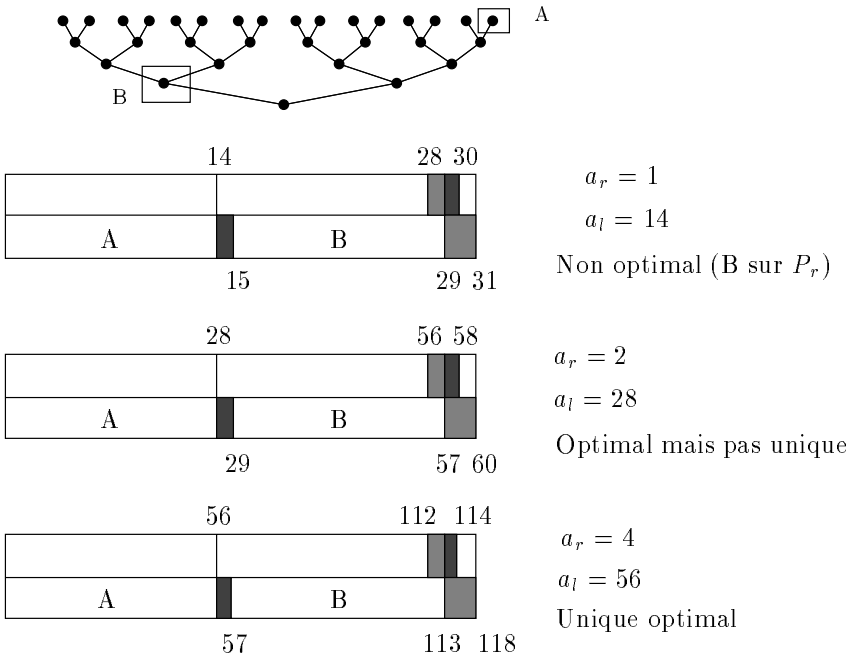


FIG. *4.7 - : $k = 2$, $l_{der} = 2$, $N_l = 2$, $C_r = 28$ et $h = 6$.*

Si $C_l + C_r = n - 2$, alors le cas qui nous intéresse est celui pour lequel $N_l = 2$. Deux cas se présentent. Soit $N_r = C_r + 1$, soit $N_l = C_l + 1$.

Dans le premier cas, $\frac{a_l}{a_r} < 2^{h-1} - 2$. Lorsque $\frac{a_l}{a_r} \leq 2^{h-1} - 3$, $l_{der} = 3$ permet d'atteindre la borne du *makespan* calculée dans le théorème 2 (voir la figure 4.8). En revanche, lorsque $2^{h-1} - 3 < \frac{a_r}{a_l} < 2^{h-1} - 2$, il n'est pas possible d'atteindre la borne du *makespan*, mais le choix de B au niveau 2 garantie une fin d'exécution égale à $a_r(C_r + 1) + 1$, qui constitue la

date de fin d'exécution optimale, comme l'illustre l'exemple de la figure 4.8.

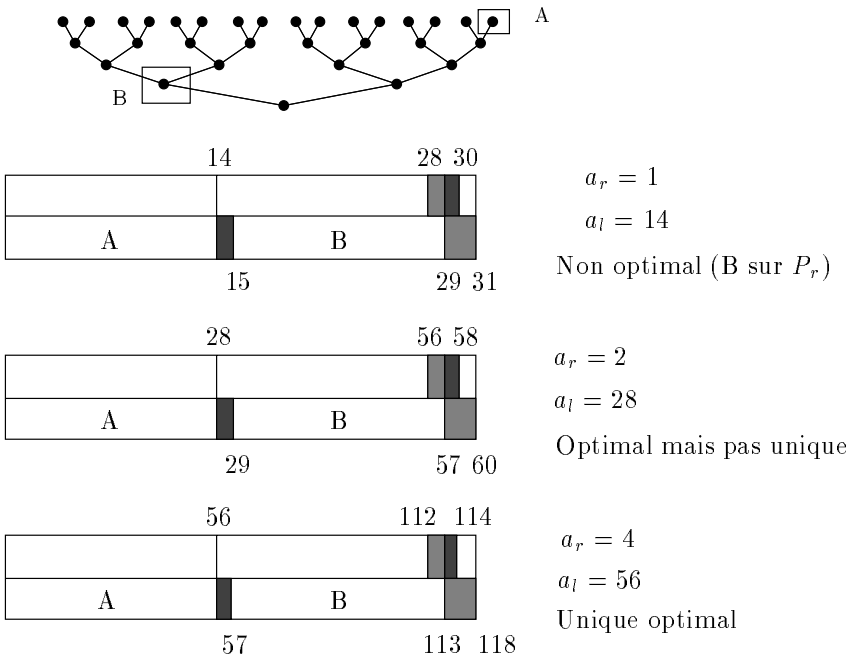


FIG. *4.8 - : Cas où* $\frac{a_l}{a_r} < 2^{h-1} - 2$.

Le second cas est plus simple car il est résolu directement par le calcul de l'affectation de la seconde tâche à P_l. Comme $\frac{a_l}{a_r} > 2^{h-1} - 2$, la tâche B n'attend jamais pour son exécution, ce qui signifit que l'ordonnancement sur P_l est constitué de l'exécution de deux tâches sans attente, et la même chose pour P_r, à part attente éventuelle avant l'exécution de la racine. Mais cette attente est garantie inférieur à a_r, sinon N_l aurait été égal à 1 (d'après le théorème 2).

Finallement, lorsque $N_l < h - 1$, l'ordonnancement d'un arbre k-aire complet est obtenu par la détermination de deux sous-ensembles de tâches allouées à P_l. Le premier est une chaîne de longueur $h - 2$ partant d'une feuille, et le second est une tâche isolée, entièrement précédée par des tâches allouées à P_r. Les seuls cas particuliers interviennent pour les arbres binaires complets pour lesquels le rapport des inverses des vitesses $\frac{a_l}{a_r}$, est tel que $N_l = 2$. Si $\frac{a_l}{a_r} \leq 2^{h-1} - 3$ alors le choix de la tâche isolée au niveau 3 de l'arbre garantie l'obtention d'un ordonnancement optimal tel que $C_{max} = a_r(n-2)$. Si $2^{h-1} - 3 < \frac{a_r}{a_l} < 2^{h-1} - 2$ alors l'optimalité peut être obtenu en choisissant la tâche isolée au niveau 2. Les ordonnancements sont tels que $C_{max} = a_r(n-2) + 1$. Dans le cas où $\frac{a_l}{a_r} > 2^{h-1} - 2$, alors si $N_l = 2$, cette tâche appartient au niveau 2, et le nombre de périodes d'attente sur P_r est inférieur à a_r, c'est-à-dire que $C_{max} \leq a_r(n-2) + a_r$. On retrouve une certaine cohérence dans ces résultats, plus le processeur P_l est lent par rapport à P_r, et plus on s'éloigne de la borne minimum atteignable par des ordonnancements optimaux.

4.6.3 Conclusion

En résumé, nous avons proposé un algorithme qui produit des ordonnancements optimaux pour le problème noté $Q_2\ \{v_1, v_2\} \mid arbres,\ p_i = 1,\ c = 1 \mid C_{max}$, où $\frac{1}{v_1}$ et $\frac{1}{v_2}$ sont des entiers supérieurs ou égaux à 1.
Lorsque $N_l \geq h - 1$, l'algorithme décrit au début de cette section permet de produire des ordonnancements optimaux.
Lorsque $N_l = 1$, un simple test sur les vitesses permet de savoir si l'allocation d'une tâche à P_l permet d'atteindre l'optimal.
Lorsque $N_l \geq h - (l_{der} - 1)$, l'allocation d'une chaîne de tâches à P_l permet d'atteindre l'optimal.
Lorsque $N_l < h-(l_{der}-1)$ alors une nouvelle stratégie doit être mise en place. Cette stratégie dicte une découpe de l'ensemble des tâches allouées à P_l en deux sous-ensembles dont le second n'est formé que d'une unique tâche, T_{der}, dernière tâche exécutée par P_l. Nous avons montré que lorsque $k \geq 3$ ou $N_l \geq 3$ ou $l_{der} \geq 3$, cette découpe permet d'atteindre l'optimal car aucun processeur n'attend. Enfin, pour le cas spécial où, à la fois l'arité, le nombre de tâches allouées à P_l et le niveau maximum de la dernière tâche exécutée par P_l sont égaux à deux, nous savons détecter une attente sur P_r qui entraîne une perte d'optimalité, et nous savons résoudre ce problème en diminuant de un le nombre de tâches allouées à P_l.

4.7 Application aux arbres quelconques

4.7.1 Introduction

Nous étudions dans cette partie l'ordonnancement des arbres quelconques dont les communications sont unitaires, et dont les tâches nécessitent pour leur exécution $\frac{1}{v}$ sur tout processeur de vitesse v. Les bornes du minimum des maximums des dates de fin d'exécution (Théorème 2), et de la valeur du niveau de la dernière tâche exécutée par le processeur le plus lent (Corollaires 1 et 2) sont encore valides. Pour les arbres complets, deux cas sont distingués. Dans le premier cas, le nombre de tâches allouées à P_l est inférieur strictement à $h-1$, alors qu'il est supérieur ou égal à cette quantité dans le second cas. Ces deux cas correspondent à la possibilité de trouver un ensemble connexe[9] de tâches contenant à la fois une feuille et une tâche liée à la racine. Pour les arbres complets, si les vitesses sont d'un ordre de grandeur inférieur au nombre de tâches de l'arbre alors cette condition est vérifiée. Ce n'est pas le cas des arbres quelconques dans lesquels de longues chaînes peuvent apparaître. La stratégie qui semble alors la plus appropriée pour obtenir de bons ordonnancements est celle qui permet dans le cas des arbres complets d'obtenir des ordonnancements optimaux lorsque $N_l < h - (l_{der} - 1)$, c'est-à-dire celle qui définie des ordonnancements formés d'au moins deux groupes de tâches connexes dont l'un au moins contient une tâche de niveau inférieur ou égal à l_{der}.

[9] pour deux nœuds du groupe il existe une chaîne qui les lie et dont tous les sommets appartiennent au groupe.

Après la description de quelques définitions, nous décomposons l'ensemble des structures arborescentes en plusieurs sous-ensembles. Pour chacun de ces sous-ensemble, un algorithme, de faible complexité, qui produit des ordonnancements optimaux est présenté. Finalement, nous mettons en évidence une stratégie de complexité plus importante qui permet de produire des ordonnancements distants de la date de fin d'exécution optimale de une unité de temps dans le pire des cas.

4.7.2 Notations et définitions

Nous conservons une partie des notations utilisées jusqu'à présent et nous introduisons quelques notations supplémentaires :

- Les processeurs sont toujours notés $\boldsymbol{P_l}$ (le plus lent) et $\boldsymbol{P_r}$ (le plus rapide),
- $\boldsymbol{l_{der}}$ est définie dans les corollaires 1 et 2,
- Les inverses des vitesses sont notées $\boldsymbol{a_l}$ et $\boldsymbol{a_r}$ et $a_r \leq a_l$,
- $\boldsymbol{Poids(n_1)}$ représente le nombre total de prédécesseurs du nœud n_1 plus un,
- Nous notons $\boldsymbol{L_{lourds}}$ l'ensemble des nœuds dont le poids est supérieur strictement à la valeur de N_l et dont le niveau est compris entre 2 et l_{der}. $\boldsymbol{L_{legers}}$ est défini de la même façon pour rassembler les nœuds dont le poids est inférieur ou égal à la valeur de N_l,
- $\boldsymbol{MaxNiv_{lourds}}$ représente le nombre maximum de nœuds de même niveau appartenant à L_{lourds}.

Définition 1 : *nous définissons la notion de* ***groupe*** *comme un ensemble connexe de nœuds alloués à* P_l.

Nous notons alors :

- $\boldsymbol{r_{groupe}}$ le nœud de plus bas niveau appartenant au groupe,
- $\boldsymbol{N_{groupe}}$ le nombre de nœuds appartenant au groupe,
- un nœud est qualifié **d'interne** s'il appartient au sous-arbre dont r_{groupe} est racine, mais pas au groupe,
- un nœud est qualifié **d'externe** s'il n'appartient pas au sous-arbre dont r_{groupe} est racine,
- $\boldsymbol{N_{interne.P_l}}$ (resp. $N_{externe.P_r}$) le nombre de tâches allouées à P_l (resp. P_r) appartenant au sous-arbre dont r_{groupe} est la racine mais n'appartenant pas au groupe,
- $\boldsymbol{N_{externe.P_l}}$ (resp. $N_{externe.P_r}$) le nombre de tâches allouées à P_l (resp. P_r) n'appartenant pas au sous-arbre dont r_{groupe} est la racine.

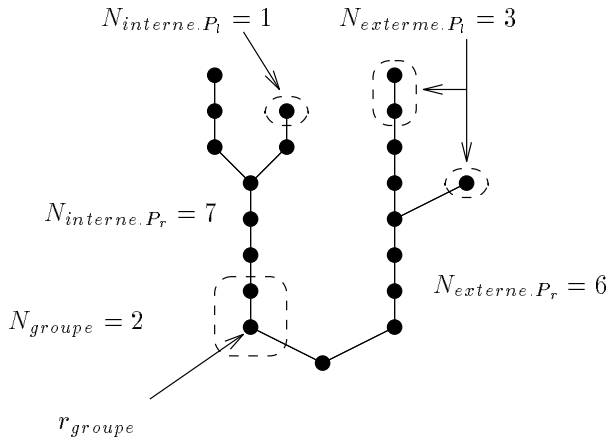


FIG. *4.9 - : Illustration des notations définies ci-dessus.*

Définition 2: *un groupe est dit* ***totalement contraint*** *si aucune tâche lui appartenant ne peut commencer son exécution avant la fin de l'exécution de* ***toutes*** *les tâches internes allouées à P_r. Dans le cas contraire, le groupe est dit* ***partiellement contraint***.

Exemples

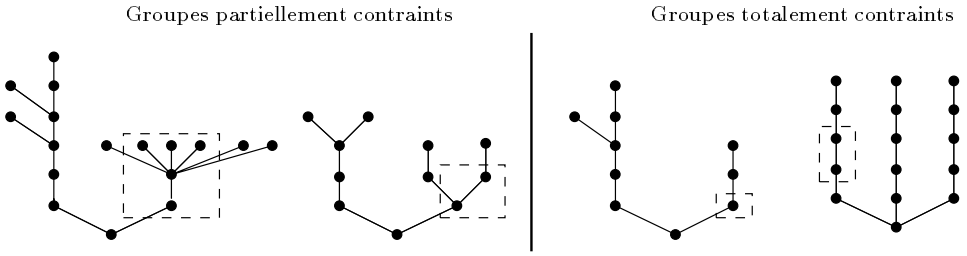


FIG. *4.10 - : Exemples de groupes partiellement et totalement contraints.*

Nous proposons dans les paragraphes suivants plusieurs algorithmes. Le premier est l'algorithme MAJYC que nous avons présenté au Chapitre 1. Nous montrons que cet algorithme produit des ordonnancements optimaux pour des arbres vérifiant certaines propriétés. Le second algorithme que nous présentons est basé sur la stratégie employée pour les arbres complets lorsque $N_l < h - (l_{der} - 2) - 1$. Enfin, le troisième algorithme permet de joindre les deux premiers et constitue une heuristique pour ordonnancer tout arbre ayant des tâches identiques et des communications unitaires.
Sans perte de généralité, nous supposons que la racine de l'arbre est d'arité supérieure ou égale à deux. Si tel n'est pas le cas, un ordonnancement optimal pour l'arbre entier est obtenu, à partir d'un ordonnancement optimal pour le sous-arbre dont la racine est le nœud de plus bas niveau dont l'arité est supérieure ou égale à deux, en ajoutant la chaîne, liant ce nœud à la racine de l'arbre, à P_r.

4.7.3 Algorithme *MAJYC*

Considérons l'algorithme MAJYC proposé dans le Chapitre 1 pour l'ordonnancement des arbres *UECT* sur deux processeurs identiques. Nous allons étudier la qualité des ordonnancements que cet algorithme produit lorsque les deux processeurs sont uniformes.

Lemme 1 :
Soit un arbre formé de n tâches. Soient N_r et N_l calculés à partir du Théorème 2 et soit l_{der} calculé à partir des Corollaires 1 et 2. Si $L_{lourds} = \emptyset$ alors un ordonnancement optimal, pour cet arbre, atteint la borne du makespan calculée et peut-être obtenu par MAJYC.

Preuve :
Dans tous les cas de figure exactement N_l tâches sont allouées à P_l. L'ensemble des tâches allouées à P_l peut être décomposé en deux sous-ensembles de cardinal respectif N_l^1 et N_l^2. Le premier sous-ensemble correspond aux tâches qui appartiennent à un sous-arbre qui n'est pas entièrement alloué à P_l. Le second correspond à l'ensemble des tâches qui appartiennent à des sous-arbres qui sont entièrement alloués à P_l. De même pour les tâches allouées à P_r : N_r^1 et N_r^2 (voir la figure 4.11).

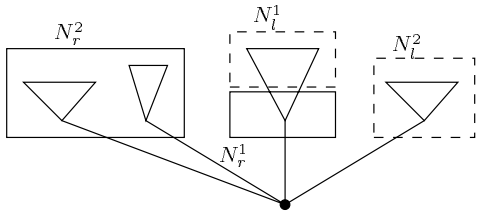


FIG. *4.11 - : Ordonnancement optimal lorsque $L_{lourd} \neq emptyset$.*

Le seul problème qui pourrait survenir serait que P_r attende des données en provenance de N_l^1. Or ce n'est jamais le cas.

- Si $C_{max} = (C_r+2)a_r$ alors $(N_r^1+N_r^2)a_r = C_{max}-a_r$ et $(N_l^1+N_l^2)a_l+1 \leq C_{max}-a_r$ donc, $N_l^1 a_l + 1 + N_l^2 a_l \leq N_r^1 a_r + N_r^2 a_r$, or s'il y a attente de données, $N_l^1 a_l + 1 > N_r^2 a_r$, donc $N_l^2 a_l < N_r^1 a_r$, mais comme les sous-arbres sont de taille au plus N_l, $N_r^1 \leq N_l - N_l^1 = N_l^2$ donc $N_l^2 a_l < N_l^2 a_r$ ce qui est impossible puisque $a_l \geq a_r$.

- Si $C_{max} = (C_l+1)a_l + a_r + 1$ alors une attente, limitée à $a_l N_l + 1 - a_r N_r$ unités de temps, est permise sur P_r. Si une attente supérieure se produit lors de la communication de P_l vers P_r après l'exécution des N_l^1 tâches, $a_r N_r^2 + (a_l N_l + 1 - a_r N_r) < a_l N_l^1$ et donc $N_l^2 a_l < N_r^1 a_r$, et la conclusion du cas précédent s'applique ici aussi.

- Si $C_{max} = a_r(N_r + 1) + 1$ alors une seule unité de temps d'attente peu se produire dans un ordonnancement optimal qui atteint la borne de C_{max} calculée. Par un raisonnement similaire au cas précédent, la conclusion du premier cas est valable une fois encore. □

Corollaire 1 :
Ce résultat est encore valable lorsque $MaxNiv_{lourds} \neq 0$ *et lorsque le cardinal de* L_{legers} *est supérieur ou égal à* N_l*, et à condition que la liste* L_{legers} *soit triée par ordre décroissant de poids.*

4.7.4 Algorithme *ORDO_UNIF*

L'algorithme ORDO_UNIF est une version de l'algorithme MAJYC adapté au cas des arbres pour lesquels le nombre de tâches appartenant à des sous-arbres dont les racines appartiennent à L_{legers} est inférieur strictement à N_l. La structure de l'algorithme reste la même, mais une procédure de recherche et de **saturation** d'un groupe de tâches est ajoutée.

Définition 3 : *Un groupe G est dit* ***saturé*** *si :*

- *Le niveau de* r_{groupe} *est inférieur ou égal à* l_{der}*.*
- *Toutes les tâches allouées à* P_l *et appartenant au sous-arbre dont* r_{groupe} *est la racine appartiennent à G, donc, les* $N_l - N_{groupe}$ *autres tâches allouées à* P_l *appartiennent à des sous-arbres dont les racines ne sont pas des prédécesseurs de* r_{groupe} *.*
- *L'addition d'une tâche à G entraîne sur* P_l *une attente d'au moins une unité de temps.*

Lemme 2 :
Un groupe totalement contraint et saturé vérifie les relations suivantes :
$(Poids(r_{groupe}) - N_{groupe})\, a_r + 1 \leq N_{externe.P_l}\, a_l$ *et*
$(Poids(r_{groupe}) - (N_{groupe} + 1))\, a_r + 1 > (N_{externe.P_l} - 1)\, a_l$.

Principe et description de l'algorithme

Lorsque $L_{legers} = \emptyset$, l'algorithme MAJYC choisi le sous-arbre de plus faible poids et alloue un groupe non contraint de tâches de ce sous-arbre à P_l. Lorsque les processeurs sont identiques, cette stratégie est optimale comme cela a été prouvé dans le premier chapitre. Malheureusement, lorsque les vitesses des processeurs ne sont pas identiques, une telle

stratégie n'est plus optimale comme le montre la figure 4.12.

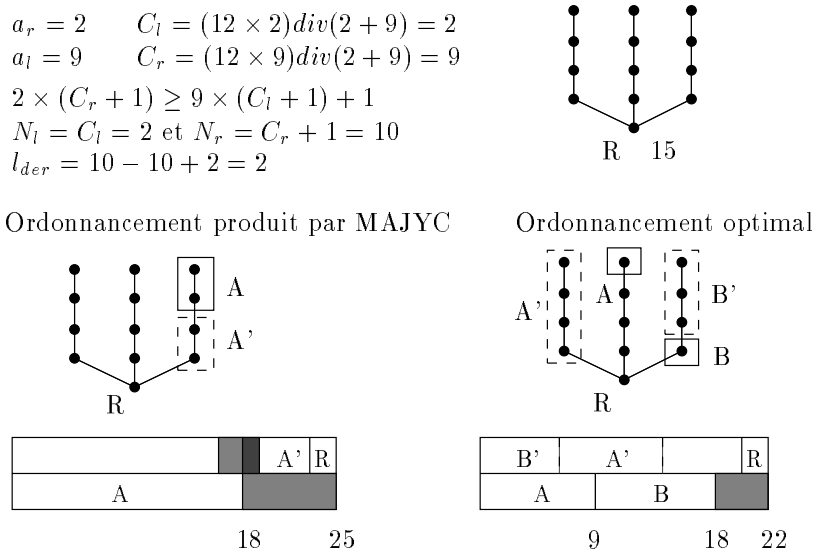


FIG. *4.12 - : Ordonnancement produit par MAJYC lorsque* $L_{legers} = \emptyset$.

La perte d'optimalité provient du fait que les nœuds dont le poids est inférieur ou égal à N_l peuvent tous appartenir à des niveaux supérieurs strictement à l_{der}. Or d'après le Corollaire XX, pour atteindre la borne du *makespan* calculée dans le Théorème 2, la dernière tâche exécutée par P_l doit appartenir à un niveau inférieur ou égal à l_{der}. L'idée est alors de choisir une telle tâche et de l'allouer à P_l. Mais, comme le sous-arbre entier ne peut être alloué à P_l (il est trop lourd), une partie seulement l'est. Cette partie constitue le groupe et il est contraint, c'est-à-dire que l'exécution de l'ensemble de ses tâches nécessite l'exécution d'un ensemble de tâches allouées à P_r dont le nombre est égal à $N_{interne.P_r}$. Il faut alors que P_l exécute des tâches externes à ce sous-arbre pour recouvrir l'exécution des $N_{interne.P_r}$ tâches. Finalement, les différentes situations qui peuvent survenir sont décrites par la valeur de $MaxNiv_{lourds}$ et par le cardinal de la liste L_{legers}. Lorsque le cardinal de L_{legers} est supérieur ou égal à N_l, alors une procédure du type *HSF* (*Heavy Subtree First*) permet d'atteindre l'optimal (Corollaire 1). Dans l'autre cas, deux procédures sont décrites qui permettent de construire, si besoin est, un groupe contraint de tâches. L'algorithme est le suivant :

Etiquetage de l'arbre
Calcul de N_l, N_r et l_{der}
Construction de L_{legers} et de L_{lourds}
Calcul de $MaxNiv_{lourds}$
Appel de ORDO_UNIF(N_l,L_{legers},L_{lourds},$MaxNiv_{lourds}$)

Expression de ORDO_UNIF(N_l,L_{legers},L_{lourds},$MaxNiv_{lourds}$)

Si $N_l = 1$ **Alors**

une feuille de plus bas niveau qui n'est pas allouée est choisie
on vérifie que l'allocation de cette feuille à P_l donne une meilleure
date de fin d'exécution que tout l'arbre alloué à P_r
$N_l = 0$
FinSi
Si $(Cardinal(L_{legers}) \geq N_l)$ et $(N_l > 0)$ **Alors**
Appel de HSF
Sinon
Tous les noeuds appartenant à L_{legers} sont alloués à P_l
$N_{legers} = Cardinal(L_{legers})$
$N_l = N_l - N_{legers}$
Si $MaxNiv_{lourds} \geq 2$ **Alors**
Appel de SATURE1
Sinon
Appel de SATURE2
FinSi
FinSi

Procédure *HSF*

Les noeuds de L_{legers} sont triés par ordre décroissant de poids
TantQue $N_l \neq 0$ **Faire**
choisir le premier de la liste
mettre à jour N_l et L_{legers}
FinTantQue

La complexité de cette procédure est dominée par celle du tri des éléments de L_{legers} qui dans le pire des cas est de complexité $O(n^2)$.

Procédure SATURE1

soit $T_{lourd.leger}$ le noeud de plus faible poids de L_{lourd}
$r_{groupe} = T_{lourd.leger}$
$N_{groupe} = 1$
TantQue $(Poids(r_{groupe} - N_{groupe})a_r + 1 \leq N_{externe.P_l} a_l$ **Faire**
ajouter un noeud au groupe
$N_{groupe} = N_{groupe} + 1$
$N_{externe.P_l} = N_{externe.P_l} - 1$
FinTantQue
enlever le dernier noeud ajouté au groupe
$N_{groupe} = N_{groupe} - 1$

$N_{externe.P_l} = N_{externe.P_l} + 1$
Si $(N_{groupe} \geq 1)$ **Alors**
les $N_{externe.P_l} - N_{legers}$ noeuds sont choisis dans le sous-arbre
dont la racine est le noeud de plus faible poids (excepté $T_{lourd.leger}$)
de L_{lourds} ($T_{lourd.leger2}$)
les noeuds de plus bas niveau appartenant à ce sous-arbre sont choisis en premier
soit $N_{libres.P_r}$, sous-partie de $N_{externe.P_r}$ rassemblant les noeuds
qui ne sont pas précédés du dernier noeud
externe exécuté par P_l
Si $(N_{interne.P_r} + N_{libres.P_r})a_r \leq (N_{externe.P_l} - N_{legers})a_l$ **Alors**
le dernier noeud choisi dans le sous-arbre dont la racine est
$T_{lourd.leger2}$ est alloué à P_r
FinSi
Sinon
les $N_{externe.P_l} - N_{legers}$ noeuds sont choisis dans le sous-arbre
dont la racine est $T_{lourd.leger}$
soit $N_{libres.P_r}$, l'ensemble des noeuds qui ne sont pas précédés
par le dernier noeud externe exécuté par P_l et appartenant
au sous-arbre dont $T_{lourd.leger}$ est racine
Si $(N_{externe.P_r} + N_{libres.P_r})a_r \leq (N_{externe.P_l} - N_{legers})a_l$ **Alors**
le dernier noeud choisi dans le sous-arbre dont la racine est
$T_{lourd.leger}$ est alloué à P_r
FinSi
FinSi

Parmi l'ensemble des nœuds appartenant à L_{lourds}, le nœud le plus léger est choisi pour être la racine du groupe. La boucle **TantQue** correspond à la saturation du groupe dont r_{groupe} est racine. Cette opération consiste à construire le plus gros groupe possible en r_{groupe}. La condition d'arrêt est que le temps d'exécution des nœuds externes ($N_{externe.P_l}a_l$) n'est plus suffisant pour recouvrir l'exécution des nœuds internes allouées à P_r ($Poids(r_{groupe} - N_{groupe})a_r$) plus une unité de temps pour la communication. Le dernier nœud ajouté au groupe est alors enlevé afin que P_l n'attende pas le résultat de l'exécution des $N_{interne.P_r}$ tâches. Le nœud choisi pour être ajouté au groupe doit simplement être un prédécesseur immédiat d'un des nœuds du groupe (pour conserver la connexité du groupe). Pour que la construction du groupe soit validée, il faut en outre que P_r n'attende pas de données en provenance de la dernière tâche externe exécutée par P_l. On distingue deux cas, suivant si N_{groupe} est nul ou non. S'il est nul, cela signifie qu'il n'est pas possible de construire un groupe totalement contraint en r_{groupe}, on abandonne donc la construction du groupe et on choisi la totalité des tâches allouées à P_l dans le sous-arbre de plus faible poids. La notion de nœuds internes n'a plus de signification, par contre on appelle encore nœuds externes les nœuds exécutés par P_r qui n'appartiennent pas au sous-arbre dont $T_{lourd.leger}$ est racine. On vérifie donc que l'ensemble des tâches disponibles pour l'exécution pour P_r est suffisant pour recouvrir l'exécution des tâches externes n'appartenant pas à L_{legers}. Seules les tâches qui sont des successeurs de la dernière tâche externe exécutée par P_l ne sont pas libres pour

l'exécution (ceci sera prouvé dans le Lemme 3). Si la condition n'est pas vérifiée, la dernière tâche externe exécutée par P_l est allouée à P_r.
La complexité de cette procédure est linéaire en fonction du nombre de noeuds de l'arbre.
Procédure SATURE2

on remonte la chaîne des noeuds de poids supérieur *à* N_l
soit $r_{sous-arbre}$ le premier noeud rencontré et
d'arité supérieure ou égale *à* 2
en ce noeud N_l est recalculé (on appelle cette nouvelle valeur N_l')
Si ($N_l' \leq N_l$) **Alors**
  on considère ce sous-arbre comme un arbre *à* part
  entière et on calcule ${l_{der}}'$ et ${MaxNiv_{lourds}}'$
  on construit ${L_{legers}}'$ et ${L_{lourds}}'$
  Appel de ORDO_UNIF(N_l',${L_{legers}}'$,${L_{lourds}}'$,${MaxNiv_{lourds}}'$)
Sinon
  $l_{der} \leq (Poids(r_{sous-arbre}) - 1) + (Niveau(r_{sous-arbre}) + 1) - \left\lceil \frac{N_l a_l + 1}{a_r} \right\rceil$
  construire L_{legers} et L_{lourds}
  calculer $MaxNiv_{lourds}$
  Appel de ORDO_UNIF(N_l,L_{legers},L_{lourds},$MaxNiv_{lourds}$)
FinSi
l'ensemble des noeuds liant la racine *à* $r_{sous-arbre}$ sera
entièrement allouée *à* P_r
et les N_{legers} noeuds entièrement allouée *à* P_l

Lorsque l'ensemble des nœuds appartenant à L_{lourds} forment une chaîne, on considère le premier nœud dont l'arité est supérieure ou égale à deux (il est inutile de diviser cette chaîne qui sera entièrement allouée à P_r). En cette nouvelle racine d'arbre, on recalcule N_l (noté ici N_l') comme si cet arbre était pris isolément et qu'on veuille l'ordonnancer sur notre système formé de deux processeurs uniformes. Si la valeur de N_l' est inférieure ou égale à N_l alors on peut considérer l'ordonnancement de cet arbre indépendamment de l'ordonnancement de l'arbre dont il fait partie, en effet, obtenir un ordonnancement optimal pour l'arbre entier implique que l'ordonnancement obtenu pour ce sous-arbre soit également optimal. On fait donc un appel récursif à la procédure ORDO_UNIF. Lorsque le nombre de tâches restant à allouer à P_l est inférieur à ce que l'on a calculé, on appelle une fois encore récursivement ORDO_UNIF, mais en lui passant des paramètres qui ne correspondent plus à ceux qui auraient été passés si l'arbre avait été considéré isolément.

Quelques exemples

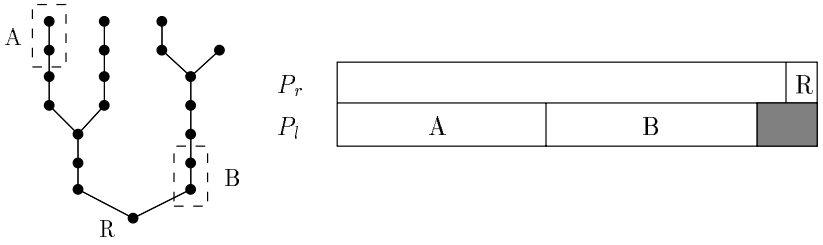


FIG. *4.13 - : Un exemple d'ordonnancement avec groupe saturé.*

4.7.5 Analyse théorique

Lemme 3 :
Soient un arbre de n tâches, un groupe G totalement contraint saturé et $N_{externe.P_l}$ tâches. L'exécution d'au moins $N_{externe.P_l} - 1$ tâches est recouverte par l'exécution des $N_{interne.P_r}$ tâches.

Preuve
Supposons que l'exécution de $N_{externe.P_l} - 1$ tâches ne soit pas recouverte par l'exécution de $N_{interne.P_r}$ tâches alors $(Poids(r_{groupe}) - N_{groupe})\, a_r < (N_{externe.P_l} - 1)\, a_l$ ce qui est en contradiction avec la propriété de saturation de G exposée dans le Lemme 2. □

Lemme 4 :
Soit un arbre de n tâches, si $MaxNiv_{lourds} \geq 3$ et si $L_{legers} = \emptyset$ alors SATURE1 produit un ordonnancement optimal par construction d'un groupe contraint et saturé localisé dans le sous-arbre de plus faible poids.

Preuve
Si $MaxNiv_{lourds} \geq 3$ il existe un niveau inférieur ou égal à l_{der} tel que trois nœuds au moins ont un poids strictement supérieur à N_l. Soit l le niveau le plus élevé tel que $MaxNiv_{lourds} \geq 3$, notons A, B, C les trois nœuds de plus faible poids et de niveau l tels que $Poids(A) \leq Poids(B) \leq Poids(C)$.
Si $(Poids(A)-1)a_r+1 \leq (N_l-1)a_l$ alors un groupe saturé peut être construit et $r_{groupe} = A$. D'après le Lemme 3 l'exécution de $N_{externe.P_l} - 1$ tâches est recouverte par l'exécution des $N_{interne.P_r}$ tâches. Il ne reste donc plus que l'exécution d'une tâche à recouvrir or il reste au moins un sous-arbre de poids supérieur ou égal à $Poids(A)$, donc l'exécution des $N_{externe.P_l}$ tâches est recouverte, P_r n'attend pas et l'ordonnancement construit est optimal.
Si $(Poids(A) - 1)a_r + 1 > (N_l - 1)a_l$ et si $N_l \geq 2$ alors cela signifie que l'exécution, sur P_l, d'un groupe non contraint de N_l tâches dans A est recouverte par l'exécution de $Poids(B) + Poids(C)$ tâches par P_r, ce qui est en contradiction avec l'hypothèse $L_{legers} = \emptyset$.
Si $N_l = 1$, un ordonnancement optimal dans lequel P_l exécute une tâche existe si et seulement si il existe une feuille dont le niveau $l \leq l_{der}$. □

Lemme 5 :
Soit un arbre de n tâches, si $MaxNiv_{lourds} \geq 2$ *et* $L_{legers} \neq \emptyset$ *alors SATURE1 produit un ordonnancement optimal par construction d'un groupe contraint et saturé localisé dans le sous-arbre de plus faible poids dont la racine n'appartient pas à* L_{legers}.

Preuve
Soient A et B les deux nœuds de plus faible poids et de niveau l (niveau le plus élevé tel que $MaxNiv_{lourds} \geq 2$), tels que $Poids(A) \leq Poids(B)$. On essaie de construire un groupe contraint et saturé en A (c'est-à-dire $r_{groupe} = A$).
Si $(Poids(A)-1)a_r + 1 \leq (N_l - 1)a_l$, alors l'exécution de $N_{externe.P_l} - 1$ tâches est recouverte par l'exécution de $N_{interne.P_r}$ tâches. En effet, d'après le Lemme 2, $(N_{interne.P_r} - 1)a_r + 1 > (N_{externe.P_l} - 1)a_l$ donc, $N_{interne.P_r}a_r + 1 > (N_{externe.P_l} - 1)a_l$. Or, $L_{legers} \neq \emptyset$ donc, le nombre de tâches allouées à P_l et incluses dans le sous-arbre dont B est racine est inférieur ou égal à $N_{externe.P_l} - 1$. Ainsi, toutes les communications sont recouvertes, celle de P_r vers P_l par hypothèse de saturation et celle de P_l vers P_r (à part éventuellement la dernière communication pour la racine si $C_r + C_l = n - 1$ ou si $C_{max} = a_l(C_l + 1) + a_r + 1)$ d'après le Lemme 2 et parce qu'au moins une tâche appartient à un sous-arbre dont la racine n'est pas un prédécesseur de B. Dans ce cas, l'ordonnancement produit est optimal.
Si $(Poids(A) - 1)a_r + 1 > (N_l - 1)a_l$ alors par l'algorithme alloue des sous-arbres dont les racines appartiennent toutes à L_{legers}, et l'ordonnancement est optimal d'après le Corollaire 1. □

Lemme 6 :
Soit un arbre de n tâche, si $MaxNiv_{lourds} \geq 2$ *et* $L_{legers} = \emptyset$ *alors SATURE1 produit un ordonnancement distant de au plus* $a_r - 1$ *unités de temps d'un ordonnancement optimal.*

Preuve
Lorsque $MaxNiv_{lourds} \geq 2$, si un groupe de tâches saturé peut-être construit alors P_l n'attend pas de données en provenance de P_r pour l'exécution des tâches appartenant au groupe. Par contre, si la condition sur l'attente de P_r n'est pas satisfaite alors l'une des tâches externes allouée préalablement à P_l est allouée à P_r. D'après le Lemme 2, l'exécution des $N_{externe.P_l} - 1$ tâches est recouverte par l'exécution des $N_{interne.P_r}$ tâches. Donc P_r n'attend plus de données en provenance de P_l. Cependant, il est possible pour certains arbres de construire des ordonnancements dans lesquels P_l aurait conservé ses $N_{externe.P_l}$ tâches comme le montre la figure 4.14. Dans un tel cas, si $C_{max} = a_r(C_r + 2)$, l'ordonnancement est distant

de a_r unités de temps de l'ordonnancement optimal.

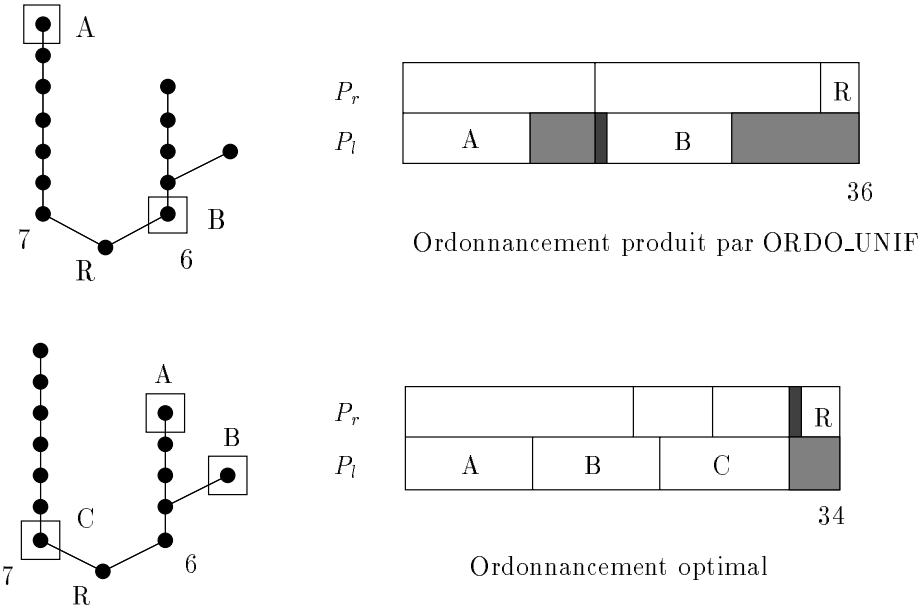


FIG. *4.14 - : Exemple d'ordonnancement produit par ORDO_UNIF qui n'est pas optimal.*

Si un groupe de tâches saturé ne peut pas être construit (illustré par la figure 4.15) alors $N_{externe.P_l} a_l < N_{interne.P_r} a_r + 1$. La totalité des tâches de P_l sont choisies dans le sous-arbre de plus faible poids (celui dans lequel l'algorithme a essayé de construire un groupe contraint saturé). P_r dispose donc d'au moins $N_{interne.P_r} + 1$ tâches prêtes à être exécutées (le poids du second sous-arbre est supérieur ou égal au poids du plus faible). D'autre part, $N_l = N_{externe.P_l} + 1$ donc $(N_l - 1)a_l + 1 < (N_{interne.P_r} + 1)a_r + 1$. Ce qui signifie que la communication de la dernière tâche exécutée par P_l est recouverte. Au total, une fois encore, dans le pire des cas $N_l - 1$ tâches sont allouées à P_l et dans ce cas aucune période d'inactivité due à une communication n'apparaît dans l'ordonnancement. La distance des ordonnancements produits est au plus égale à a_r unités de temps. □

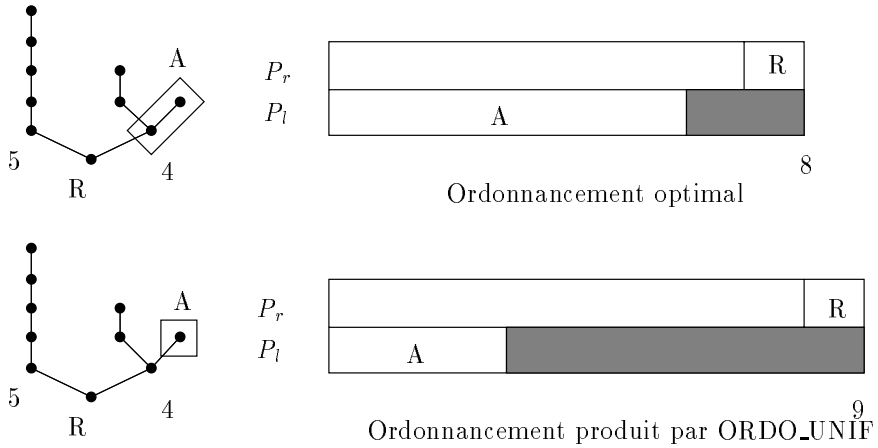


FIG. *4.15 - : Autre exemple pour lequel l'ordonnancement produit n'est pas optimal.*

Corollaire 6 :
Les ordonnancements produits par ORDO_UNIF pour des arbres dont les tâches sont identiques et requièrent respectivement a_r unités de temps sur le processeur rapide et a_l ($a_l \geq a_l$) unités de temps sur le processeur le plus lent sont distants de au plus a_r unités de temps d'un ordonnancement optimal.

Preuve
Le nombre de tâches calculé au départ n'est diminué qu'à trois occasions. Lorsque des tâches sont allouées à P_l, on diminue alors N_l du nombre de tâches allouées. Lorsque $MaxNiv_{lourd} = 1$, $N_l > 0$ et toutes les tâches appartenant à L_{leger} ont été allouées à P_l. Dans ce cas, N_l est recalculé en le premier nœud dont l'arité est supérieure ou égale à deux. Si la nouvelle valeur est inférieure à la valeur courante de N_l, elle est remplacée. Cela signifie que jusqu'au niveau de cette racine intermédiaire, toutes les tâches n'appartenant pas à la chaîne liant ce nœud à la racine de l'arbre ont été alloués à P_l. La seule manière d'augmenter alors le nombre de nœuds alloués à P_l serait de le forcer à exécuter davantage de nœuds dans le sous-arbre ce qui entraînerait inévitablement des attentes pour cause de communications. Enfin, Lorsque $MaxNiv_{lourds} \geq 2$, le Lemme 6 montre que les ordonnancements produits sont distants au plus de a_r unités de temps de l'optimal. De plus, cette procédure n'est pas appelée récursivement, cette erreur ne peut donc être commise qu'une seule fois. □

4.8 Heuristiques à m processeurs

Nous proposons dans cette partie une heuristique pour l'ordonnancement des arbres complets sur un nombre $m > 2$ fixé de processeurs. Cette heuristique est basée sur l'algorithme pour deux processeurs qui produit des ordonnancements optimaux. On suppose maintenant que l'on dispose de m processeurs P_1 ... P_m de vitesses respectives v_1 ... v_m. On suppose que $v_1 \geq v_2 \geq ... \geq v_m$. On suppose de plus que les v_i sont des vitesses rationnelles telles que $v_i \leq 1$.

4.8.1 Algorithme

Cet algorithme est composé de deux étapes. La première concerne l'allocation des tâches aux processeurs et la seconde concerne l'ordonnancement sur chaque processeur des tâches qui lui ont été allouées. L'idée de base de cette heuristique est de se ramener à un problème à deux processeurs que l'on sait traiter de façon efficace. Pour cela, on regroupe les processeurs en deux sous-ensembles E_P et $E_{P'}$, en essayant d'équilibrer la puissance de calcul.

Première phase : elle consiste à regrouper l'ensemble des processeurs en deux ensembles de processeurs tels que la puissance de calcul disponible soit équilibrée (autant que possible puisqu'il s'agit d'un problème difficile). Pour cela, soit v_i la vitesse du processeur P_i, on utilise un algorithme de type *LPT* (*Largest Processing Time first*) répartir les processeurs dans les deux sous-ensembles.

Seconde phase : on considère maintenant les deux super-processeurs P et P' issus du regroupement de la première phase. On calcule pour ce système bi-processeur le nombre de tâches allouées à chacun d'eux (comme cela avait été fait pour le cas $m = 2$) : N_P et $N_{P'}$.

$$N_P = ((n-1)v_P)div(v_P + v_{P'}) \text{ et } N_{P'} = ((n-1)v_{P'})div(v_P + v_{P'}).$$

De plus, si $N_P + N_{P'} = n-2$, un calcul de fin d'exécution est effectué pour allouer la dernière tâche à l'un des deux super-processeurs.

Troisième phase : l'arbre est partagé par le même algorithme utilisé pour le cas $m = 2$. C'est-à-dire que le nombre de tâches allouées au processeur le plus rapide est décomposé en un ensemble d'arbres complets de même arité que l'arbre en entrée, mais de hauteurs décroissantes (de $h-1$ jusqu'à 1).

A la fin de cette étape regroupant les trois phases qui viennent d'être décrite, l'un des super-processeur est chargé de l'exécution d'une forêt d'arbre complets de même arité que l'arbre complet d'origine, et le second est chargé de l'exécution d'un ensemble d'arbres complets dont l'arité est variable et d'une chaîne de longueur au plus $h-1$.
Le procédé décrit ci-dessus est réappliqué itérativement pour chaque sous-ensemble de processeurs. Cet appel récursif ne se termine que lorsque les sous-ensembles E_P et $E_{P'}$ ne sont formés que d'un unique processeur.
A la fin de l'étape d'allocation, à chaque processeur est alloué un certain nombre de tâches, on résout pour chaque couple de processeurs (il s'agit cette fois-ci des P_i) l'ordonnancement des tâches. La technique d'ordonnancement utilisée est simple, il s'agit d'un algorithme de liste. On ordonnance des sous-arbres complets, et non chaque tâche prise de façon isolée. La plus grande priorité est donnée aux racines R_i dont le successeur immédiat est alloué à un autre processeur et telles que le niveau des R_i est le plus élevé.

4.8.2 Exemple

Dans l'exemple que nous développons, l'ensemble des processeurs est formé de quatre processeurs P_1 (de vitesse $v_1 = \frac{3}{5}$), P_2 ($v_2 = \frac{1}{4}$), P_3 ($v_3 = \frac{1}{7}$), P_4 ($v_4 = \frac{2}{15}$). L'arbre consi-

déré est complet, d'arité 3 et de hauteur 5. Le nombre de tâches de l'arbre est donc égal à 121.

Étape d'allocation :

Première phase : rassemblement des processeurs en deux sous-ensembles de processeurs dont l'algorithme essaye d'équilibrer la puissance de calcul. $E_{P'} = \{P_1\}$, $E_P = \{P_2, P_3, P_4\}$. Les puissances cumulées respectives sont égales à $\frac{3}{5}$ pour P' et $\frac{221}{420}$ pour P.

seconde phase : le nombre de tâches allouées à chacun de ces processeurs sont donc : $N_{P'} \geq \frac{120\times(3/5)}{473/420} = 63$ et $N_P \geq \frac{120\times(221/420)}{473/420} = 56$. Or $63 + 56 = 119$, il reste donc une tâche à allouer. On calcule sur quel processeur elle se finirait le plus tôt. $64 \times \frac{5}{3} < 57 \times \frac{420}{221}$. On l'alloue donc à P', donc, $N_{P'} = 64$ et $N_P = 56$. Il y a ici une légère modification de l'algorithme pour $m = 2$, en effet, lorsque les vitesses ne sont plus des inverses d'entiers, mais des rationnels, l'exécution de la racine sur le processeur le plus rapide ne constitue plus un choix toujours judicieux.

Troisième phase : l'arbre complet est réparti pour ces deux processeurs. Les sous-arbres complets de hauteur $h - 1 = 4$ contiennent 40 tâches, ceux de hauteur 3 contiennent 13 tâches, puis 4 tâches pour une hauteur égale à 2, et enfin 1 pour les feuilles. $N_{P'} = 1 \times 40 + 1 \times 13 + 2 \times 4 + 3 \times 1$. Les N_P tâches sont donc constituées d'un arbre complet d'arité 3 et de hauteur 4, d'un arbre complet d'arité 3 et de hauteur 3 et d'un arbre complet d'arité 1 (une chaîne) et de hauteur 3.

L'ensemble $E_{P'}$ n'est formé que d'un processeur, aucune autre étape de division-répartition n'est mise en œuvre pour cet ensemble. Par contre, l'ensemble E_P contient trois processeurs, il faut donc mettre en œuvre une seconde étape de division-répartition.

Allocation pour l'ensemble E_P :

Première phase : $E_{P'} = \{P_2\}$, $E_P = \{P_3, P_4\}$. Les puissances cumulées respectives sont égales à $\frac{1}{4}$ et $\frac{29}{105}$.

seconde phase : $N_{P'} = 26$ et $N_P = 30$.

Troisième phase : la chaîne est allouée à P', puis pour les arbres complets d'arité au moins deux, on applique encore le même algorithme de répartition à tous les sous-arbres complets. $N_P = 2 \times 13 + 1 \times 4$.

Une fois de plus, un ensemble de processeurs est constitué de plus d'un processeur. On répartit entre les deux processeurs qui le constitue les deux sous-arbres complets de hauteur 3 et celui de hauteur 2.

Allocation pour l'ensemble E_P courant :

Première phase : $E_{P'} = \{P_3\}$, $E_P = \{P_4\}$. Les puissances cumulées respectives sont égales à $\frac{1}{7}$ et $\frac{2}{15}$.

seconde phase : $N_{P'} = 16$ et $N_P = 14$.

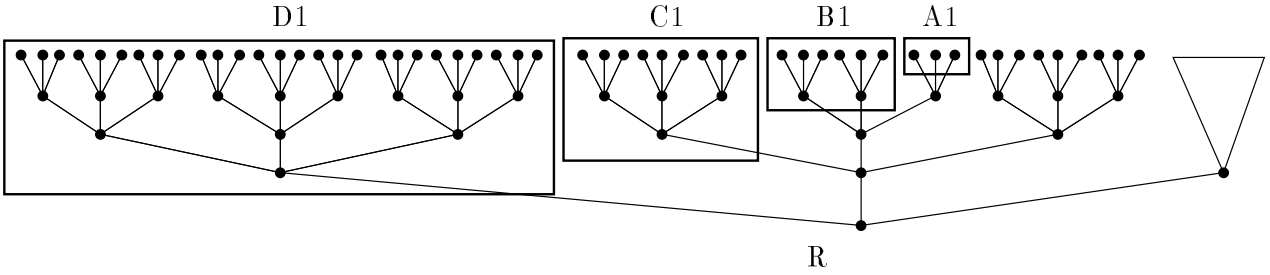
Troisième phase : $N_{P'} = 1 \times 13 + 3 \times 1$. Un sous-arbre complet de hauteur 3 et d'arité 3 est alloué à P' (qui est P_3) ainsi que trois feuilles.

Étape d'ordonnancement :

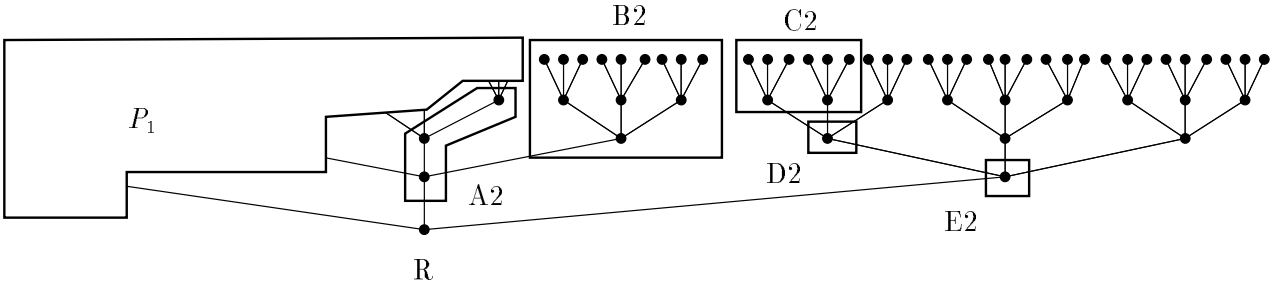
On classe l'ensemble des tâches allouées à un processeur en fonction de la hauteur de la racine du sous-arbre auquel cet tâche appartient. Les sous-arbres dont les racines sont les plus hautes sont les plus prioritaires. Les sous-arbres exécutés par P_1 respecte cette règle. En revanche, lorsque des tâches ne sont pas disponibles bien qu'ayant une plus grande priorité, d'autres tâches sont exécutées à leur place (stratégie gloutonne). Ce cas est illustré par l'exécution sur P_2 de B_2 avant C_2. La stratégie d'ordonnancement est du type *HLF* (*Highest Level First*) appliqué aux racines des sous-arbres.

L'ensemble allocation des tâches et ordonnancement de celles-ci sur les processeurs est

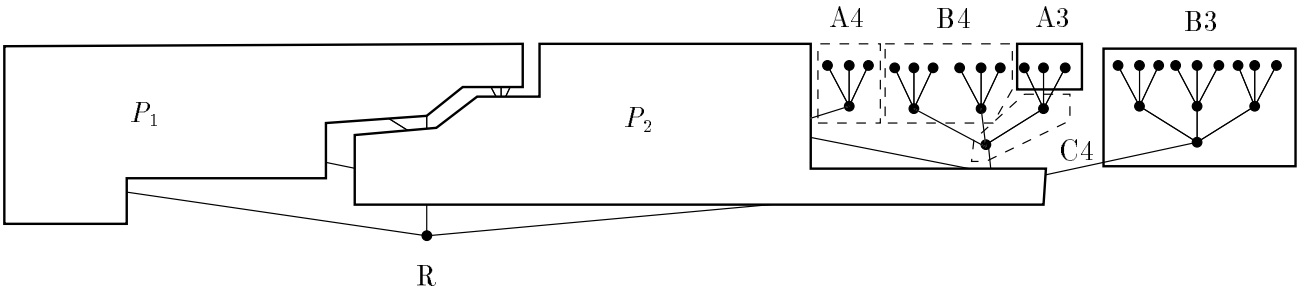
illustré en figure 4.16.



Allocation des tâches à P_1



Allocation des tâches à P_2



Allocation des tâches à P_3 et P_4

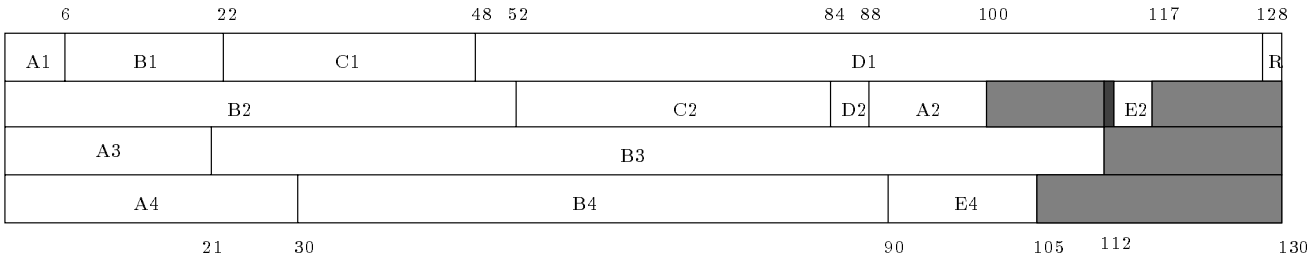


FIG. 4.16 - : Différentes phases de l'ordonnancement d'un arbre complet d'arité trois et de hauteur cinq sur quatre processeurs.

4.9 conclusion

Dans ce chapitre, nous avons étudié le problème de l'ordonnancement d'arbres sur un ensemble de processeurs uniformes. Un algorithme pour ordonnancer des arbres complets sur deux processeurs a été proposé. Il produit des ordonnancements optimaux pour deux processeurs dont les inverses des vitesses sont des entiers supérieurs ou égaux à un. L'algorithme est basé sur une stratégie de regroupement de tâches. Un premier algorithme présenté permet de produire des ordonnancements optimaux dès que la hauteur de l'arbre est inférieure au nombre de tâches que devrait exécuter le moins rapide des deux processeurs. Si cette condition n'est pas vérifiée, il faut mettre en évidence un groupe de tâches dépendant en totalité de tâches exécutées par le processeur rapide. Deux vérifications sont alors nécessaires. La première concerne l'occupation du processeur lent avant l'exécution de ce groupe de tâches. La seconde consiste à s'assurer que le processeur rapide n'attende pas ou suffisamment peu pour ne pas dégrader la qualité de la date de fin d'exécution. Ce type de raisonnement et la stratégie de recherche d'un groupe disponible et d'un groupe entièrement dépendant sont repris pour la mise en oeuvre d'une heuristique pour ordonnacer des arbres quelconques sur deux processeurs. Il est montré que les ordonnancements produits par cet algorithme sont distants de moins de une unité de temps de la date de fin d'exécution optimale. Enfin, dans une dernière partie, nous avons donné la description d'une heuristique pour l'ordonnancement d'arbres complets sur un ensemble de m processeurs (m fixé) dont les inverses des vitesses sont des rationnels. L'étude de la qualité des ordonnancements produits par cet algorithme n'est pas terminée. Une validation expérimentale est également prévue comme pour les heuristiques proposées dans le chapitre concernant l'ordonnancement d'arbres sur m processeurs identiques de granularité différentes de un. Il est intéressant de noter qu'il s'agit à notre connaissance de la seule tentative de conception d'un algorithme pour ordonnancer des graphes de précédence pour un système formé de plus de deux processeurs uniformes.

Partie IV

Ordonnancement dans les Environnements de Programmation Parallèles

Chapitre 1

Introduction

Le développement des **environnements de programmation parallèle** (EPP) est un élément clé pour l'avenir des ordinateurs parallèles. Un nombre croissant de rencontres et congrès réunissant les représentants d'organismes de recherche, de fabricants de machines et d'industriels, atteste que le parallélisme tend à se diffuser. Depuis 1993, des rencontres sont organisées par ORAP[1] et les JIP[2] dont la première édition eut lieu en 1994 à Lyon et qui sera organisée cette année à Bordeaux. Des machines parallèles, localisées partout en France, sont mises à la disposition d'utilisateurs par le biais du réseau RAPID[3]. Au niveau européen enfin, le programme *HPCN*[4] va également dans ce sens.

Dès lors qu'il s'agit de produire des codes industriels, le besoin d'outils se fait plus pressant. L'objectif à atteindre pour ces logiciels est de fournir une plateforme d'aide à la programmation parallèle, afin de permettre une programmation **facile, efficace et portable**. Malheureusement, ces objectifs sont contradictoires.

Le projet **apache**[5] vise en partie ces objectifs [Pla94]. L'accent est mis sur la production d'un code parallèle **efficace et portable**. C'est la raison pour laquelle le **parallélisme explicite** a été choisi. La préférence a été donnée au **parallélisme de contrôle** pour répondre au besoin de parallélisation d'applications **non régulières**, pour lesquelles une approche parallélisme de données ne semble pas adaptée. Au même titre que les problèmes à structures de données régulières, ces problèmes se retrouvent dans de nombreux domaines, y compris dans les milieux industriels[6].

Il existe à l'heure actuelle quelques EPPs qui visent les mêmes objectifs. La démarche de parallélisation qu'ils utilisent est sensiblement la même pour tous. Nous la décrivons dans le chapitre suivant et resituons quelques uns de ces EPP dans cette démarche. Dans la suite

[1] ORganisation Associative du Parallélisme.

[2] Journées Industrielles du Parallélisme.

[3] Réseau de ressources des Applications Parallèles pour l'Industrie et le Développement scientifique.

[4] *High Performance Computing Network.*

[5] Algorithmique Parallèle et pArtage de CHargE.

[6] Séminaire sur les techniques nouvelles de traitement des matrices creuses pour les problèmes industriels, Février 1995, Grenoble

de ce chapitre, quelques faiblesses de ce processus sont discutées. Enfin, dans le dernier chapitre, nous tentons d'apporter quelques éléments de réponse qui constituent une partie des spécifications du module ordonnancement statique dans **athapascan**[7]. Ces spécifications reposent sur les idées forces de la mise en œuvre d'ATHAPASCAN, les **poly-algorithmes** et la **granularité adaptative** dans le but de garantir une bonne efficacité et le portage des applications en ATHAPASCAN sur différents systèmes multiprocesseurs.

[7] ATHAPASCAN est l'environnement de programmation. Il est l'un des axes de recherche du projet APACHE.

Chapitre 2

Description d'une démarche fort utilisée

2.1 Une démarche synthétique

La démarche qui est décrite dans ce chapitre est principalement tournée vers les EPP basés sur une approche de type parallélisme de contrôle qui est le type de parallélisme retenu dans le projet APACHE. Cependant, à des fins d'illustration, quelques techniques utilisées dans les approches de parallélisme de données sont mentionnées. De plus, nous privilégions dans les discussions et les remarques le point de vue de l'ordonnancement.

Dans une approche parallélisme de contrôle, l'allocation des processeurs aux tâches de calcul (ou de traitement de façon plus générale) est guidée par l'agencement de ces tâches entres elles. Les dépendances entres tâches sont décrites et modélisées par un graphe de précédence lorsque la granularité choisie le permet. La première étape de toute démarche basée sur cette approche commence donc par **produire un graphe de précédence** à partir du code source. La seconde étape correspond à l'allocation des processeurs aux tâches de calcul, il s'agit de l'étape **d'ordonnancement et de placement**. Enfin, la dernière étape avant l'exécution est celle de la **génération de code**, avec insertion éventuelle de primitives de communication (pour une synchronisation). Assez rapidement, nous décrivons les outils qui sont mis en oeuvre au moment de l'exécution, afin d'améliorer les décisions prises par les étapes précédentes (mesures et analyses des performances, prises de traces), ou de réguler la charge des processeurs lorsque l'analyse statique s'avère insuffisante ou erronée. L'ensemble des étapes de cette démarche ainsi que les outils qui lui sont rattachés sont représentés en figure 2.1.

2.1.1 Production du graphe

Du point de vue de l'ordonnancement, le modèle le plus couramment utilisé est basé sur un graphe de précédence. Les sommets de ce graphe sont appelés des tâches et sont valués par des temps ou des volumes de calculs estimés, soit par l'utilisateur soit par un module spécialement conçu à cet effet. Les arcs du graphe représentent les relations de précédence

entre les différentes tâches, ils sont valués par des volumes de communications ou des temps correspondants aux transferts. Il faut noter que l'estimation automatique de la durée d'une tâche constitue un problème ardu et fortement dépendant de la machine cible, ce qui est un obstacle à la portabilité. Il peut être préférable de valuer sommets et arcs par une expression de complexité en fonction du volume des données.

La production du graphe nécessite plusieurs étapes. Elles sont traitées différemment selon le code source disponible en entrée de l'analyseur. En effet, si l'application est déjà codée à l'aide d'un langage séquentiel, les seules informations disponibles sont reçues sous la forme du code source. Si l'application est analysée en vue de son implémentation dans un langage exprimant une forme de parallélisme alors davantage d'informations peuvent être disponibles, mais leur type et leur nombre dépendent des spécifications du langage. Dans les deux cas cependant, on peut mettre en évidence les trois mêmes étapes : **détection et l'extraction du parallélisme, détermination de la granularité, et valuation des sommets et des arcs et analyse des dépendances de données**.

Il est important de remarquer que dans la majeure partie des environnements existants, la granularité est fixée, conditionnée par le parallélisme détecté, et il n'est plus possible d'y

revenir une fois cette étape achevée.

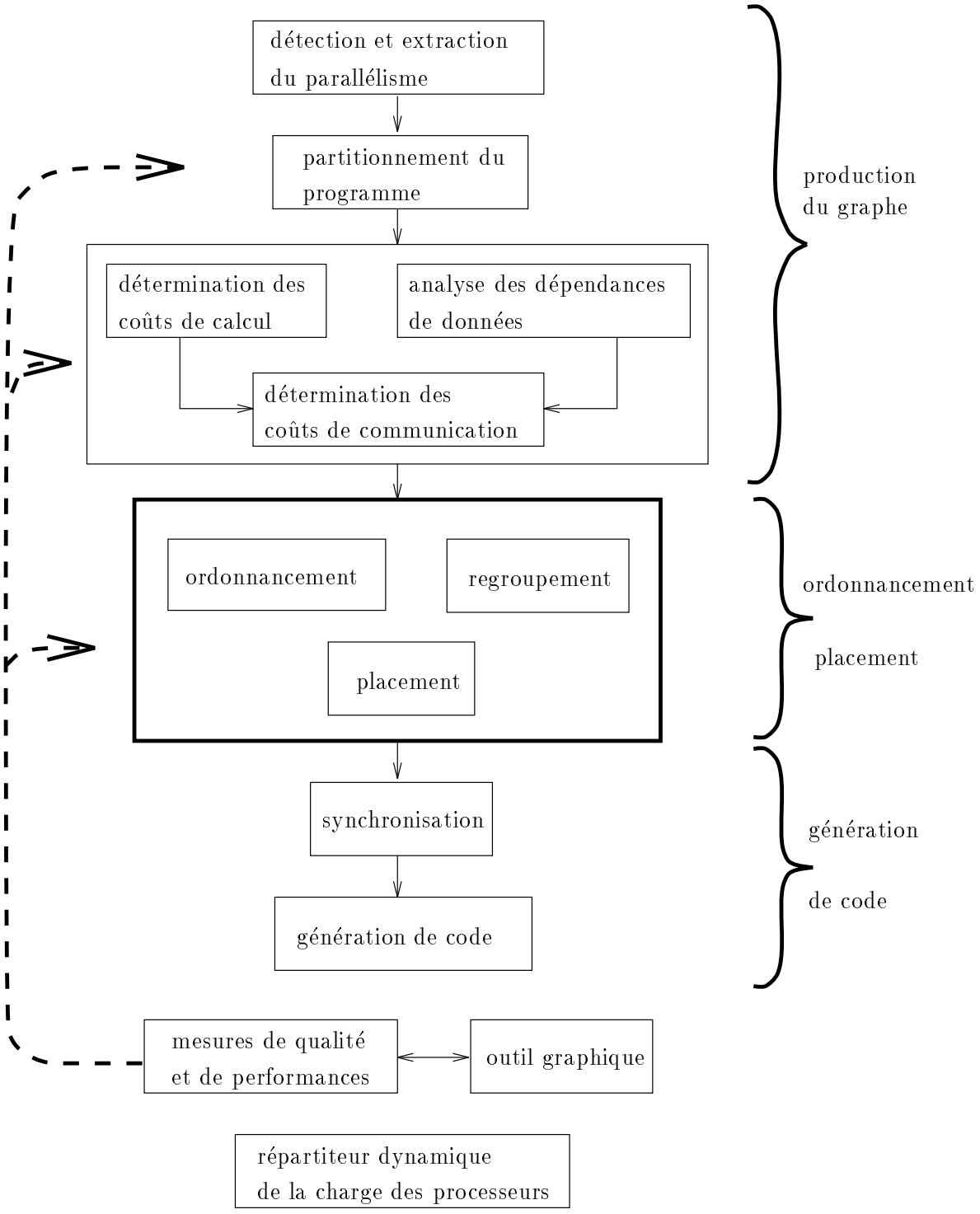


FIG. *2.1 - : Une démarche générale dans une approche parallélisme de contrôle.*

– **Détection et extraction du parallélisme**
Ce module peut prendre place avant ou après la détermination de la granularité. Dans

le cas où la granularité n'est pas fixée, la détection est faite au niveau de l'instruction. Lorsque l'application est exprimée au moyen d'un langage séquentiel, l'analyse peut être effectuée en tenant compte de directives de compilation ajoutées par l'utilisateur. Les directives peuvent être données à deux instants différents, soit avant l'analyse (CRAY/fpp, KAP/CRAY), soit durant le processus de mise en évidence du parallélisme de l'application (FORGE 90, PARASCOPE, PAT) [Che93]. Durant cette phase, un effort particulier est fait sur l'analyse des nids de boucle (boucles imbriquées) suivie d'analyse des dépendances de données. Idéalement, à la fin de cette étape, le maximum de parallélisme potentiel de l'application a été mis en évidence.

Lorsque les langages permettent une expression directe du parallélisme dans le code (HPF, HYPERTOOL) ou via une expression du graphe de dépendance (PYRROS, HeNCE et PVM, SCHEDULE), on obtient à la fin de cette étape un graphe de tâches exprimant le maximum de parallélisme possible. Lorsque la granularité n'est pas encore fixée, la question de l'évaluation des coûts de calcul ou de communication ne se pose pas.

- **Détermination de la granularité**
  La détermination d'une granularité pour une application et sur une architecture données est un problème difficile [KL88]. Cette étape est souvent appelée **étape de partitionnement** (partitionnement du programme). L'élément prend en entrée le programme et les indications issus de l'analyse précédente, ainsi qu'une description de la machine cible. En effet, la détermination de la granularité dépend de la machine ainsi que de contraintes systèmes (temps partagé...). Le but est de déterminer la taille minimale du groupe d'instructions de base : **le grain**. Dans la plupart des cas, le grain utilisé est la procédure, d'autres solutions ont été proposées pour rechercher une meilleure adéquation entre la taille du grain et l'application comme par exemple le *grain packing* [KL88]. On peut imaginer une machine à grain fin pour laquelle la meilleure granularité serait l'instruction (c'est le cas des machines SIMD). Le grain constitue alors l'unité de base en terme de tâche, il est constitué d'un ensemble insécable d'instructions. Ainsi, si l'utilisateur a spécifié que toute procédure ou fonction est défini comme une unité de programme insécable, le grain ne pourra être plus fin que la procédure. En revanche, selon le programme, il se peut qu'un grain soit formé de plusieurs procédures ou fonctions. Cette partie du processus de parallélisation pose de nombreux problèmes d'automatisation et l'utilisateur joue généralement le premier rôle. Dans de nombreux PPE, il fixe une taille minimale pour les grains et peut imposer des contraintes de non agglomération pour certains types d'instructions.

  La détermination de la granularité est à la fois difficile et cruciale. Si la taille de grain choisie ou imposée est trop importante, tout le parallélisme de l'application n'a pas été exploité et les performances peuvent être de qualité médiocres. Inversement, une granularité trop faible peut entrainer une dégradation des performances dùe à une distribution excessive de fractions de programme, entrainant de nombreuses communications. Ce module et le précédent sont fortement liés, une application présentant

un parallélisme potentiel maximal (toutes les instructions indépendantes), permet un choix de granularité variant de l'instruction simple au programme complet (sauf annotation contraire de l'utilisateur), tandis qu'une application présentant un parallélisme nul (toutes les instructions dépendantes les unes des autres de façon séquentielle) induit une granularité réduite à un seul grain, le programme dans sa totalité. Entre ces deux extrêmes, toutes les nuances existent, le rôle de certaines annotations de l'utilisateur pouvant se borner à limiter les choix.

- **Analyse des dépendances de données**
  Une fois la détermination des grains effectuée, il faut identifier les relations qui les lient. En entrée on dispose de l'ensemble des grains (que l'on peut maintenant appeler des tâches) et en sortie on récupère un **graphe de tâches**. Dans cet élément, on détermine les relations qui existent entre les tâches. Il existe une relation de précédence entre deux tâches lorsque l'une d'entre elles envoie des données à l'autre. En revanche, lorsque, par exemple, deux tâches échangent des données de manière réciproque le graphe est non orienté. Chaque sommet du graphe correspond à une tâche et une seule et chaque arc ou arête représente les relations entre ces grains (relations de précédence ou échange de données).

- **Détermination ou estimation des coûts**
  L'objectif de cette étape est la valuation du graphe de tâches reçu en entrée. Cet élément tente d'estimer les coûts de calcul de chaque grain ainsi que les coûts de communication entre eux. Les coûts réels sont fonction de la machine cible, du modèle de communication, via la mémoire partagée ou par échange de messages. Dans ce dernier cas, le mode de commutation peut également jouer un rôle. La détermination des coûts de communication ne peut pas se faire avant l'analyse des dépendances de données.

2.1.2 Ordonnancement et placement

Pour pouvoir réaliser un ordonnancement statique, la plupart des PPE basés sur une approche parallélisme de contrôle suppose qu'un graphe valué est disponible en entrée. Certains (Pyrros en particulier) demandent une description de l'application sous forme d'un graphe orienté et valué, ce qui permet d'appliquer directement les heuristiques issues de la théorie de l'ordonnancement pour les systèmes multiprocesseurs. Le premier travail consiste à effectuer l'ordonnancement ou le placement du graphe hérité de l'élément précédent, en tenant compte de l'architecture cible. A la fin de cette phase, des primitives de synchronisation sont éventuellement ajoutées. La génération de code conclue le processus de parallélisation avant l'exécution. Dans la description qui suit, on appelle **grains** les sommets du graphe reçu de l'élément précédent. On parle de **tâches** pour les sommets du graphe en entrée des modules ordonnancement et placement (c'est-à-dire après le passage eventuel du graphe dans le module regroupement).

- **Ordonnancement - Placement**
  On peut considérer cet élément comme un ensemble de trois modules **regroupement,**

ordonnancement, et placement. Le module **regroupement** peut être considéré comme une fonctionnalité de l'élément. Il peut intervenir dans l'ordonnancement et dans le placement comme une aide.
Ce module peut agir à deux moments différents, soit au moment de la compilation, soit au moment de l'exécution, soit pendant ces deux phases. Dans le premier cas, le module prend en entrée le graphe de programme, une description de l'architecture cible plus eventuellement des annotations présentes dans le code. Dans le second cas, les différents modules utilisent des résultats acquis par l'élément de mesures de performances pour faire (par exemple) du placement dynamique par migration de tâches, dans un soucis d'équilibrage de la charge (*dynamic load balancing*).
Les fonctionnalités et les frontières entre les modules ordonnancement et placement dépendent de la définition qu'on leur attribut. Nous en proposons une qui permet de préciser les rôles de chacun.

- **Regroupement**
  Comme préliminaire à l'étape d'ordonnancement ou à l'étape de placement, on regroupe les grains dans le premier cas et les tâches dans le second cas, au vu des propriétés structurelles du graphe de précédence et de la topologie du réseau. Cet élément prend donc en entrée le graphe formé de grains et retourne un autre graphe qui conserve l'orientation du graphe d'entrée, mais formé de tâches. Une tâche pouvant être formée d'un ou plusieurs grains. Cette étape n'est pas obligatoire, il est possible de déterminer un ordonnancement sans passage dans cet élément. Cependant, on remarque dans certains algorithmes d'ordonnancement (chemin critique) une étape préliminaire qui consiste à faire du regroupement. Pour les algorithmes de placement, l'étape et les stratégies de regroupement sont plus importantes. La raison en est que la qualité d'un placement est souvent mesurée par la valeur d'une fonction de coût dans laquelle intervient un terme dépendant des communications. Or, dans le cadre du placement, les communications ne peuvent pas être recouvertes, mais seulement éliminées par regroupement des tâches communicantes. L'hypothèse de localité qui a été présentée dans la première partie permet en effet de considérer que deux tâches exécutées sur un même processeur communique en un temps nul. Notons enfin que l'utilisateur par l'intermédiaire d'annotations peut influer sur les regroupements de grains ou de tâches.
- **Ordonnancement**
  Nous appelons **ordonnancement** l'allocation d'un ou plusieurs intervalles de temps et d'une ou plusieurs machines pour chaque noeud du graphe [LLKS89]. Le graphe considéré est formé soit d'un ensemble de *tâches* indépendantes (où une tâche est un ensemble de grains, obtenue par un passage dans le module regroupement), soit formé d'un ensemble de tâches liées par des contraintes de précédence. Le graphe est alors orienté. S'il comporte des cycles, les techniques à mettre en oeuvre sont celles de l'ordonnancement cyclique [**?**].

L'objet de ce module qui constitue avec le placement le coeur du processus de parallélisation est d'associer à chacune des tâches des intervalles de temps et une (ou plusieurs) machine. Les entrées nécessaires sont le graphe de précédence valué et le nombre et la vitesse des processeurs de l'architecture cible. Le résultat produit est un *diagramme de Gantt*. Ce diagramme est la représentation d'un triplet.

(date de début d'exécution, n° du processeur utilisé, n° de la tâche)

Le problème majeur que l'on rencontre est l'absence d'horloge globale dans un système distribué asynchrone. Donc, les résultats que peut fournir l'ordonnancement ne sont que des indications. Un ordre total sur chaque processeur peut être défini, mais dès que le contrôle n'est pas centralisé, aucun ordre ne peut exister entre des tâches exécutées sur différents processeurs. Dans la plupart des PPE existants, il n'existe pas de contrôle de l'exécution.

– **Placement**
Nous définissons le *placement* comme l'allocation d'un ou plusieurs processeur(s) à chaque noeud du graphe.

Il y a deux entrées possibles pour ce module. Si le graphe reçu en entrée de l'élément était orienté, le module allocation prend en entrée le résultat de l'ordonnancement, le diagramme de Gantt. A partir de ce diagramme, on construit un graphe non orienté. Dans ce graphe, un sommet est représenté par un des processeurs anonymes du diagramme de Gantt, le coût associé à ce sommet correspond à la somme des temps d'exécution des tâches exécutées par le processeur anonyme. Une arête entre deux sommets de ce graphe représente les échanges de données entre les processeurs, et le coût associé correspond à la somme des données échangées entre ces deux processeurs Le second cas correspond à un graphe d'entrée non orienté, le graphe n'est alors pas passé dans le module ordonnancement. Dans les deux cas, le graphe récupéré en entrée de ce module est non orienté. D'une manière générale, si le graphe d'entrée comporte plus de tâches que le nombre de processeurs disponibles dans l'architecture cible, il convient de faire du regroupement avec le souci constant de satisfaire au mieux aux critères d'équilibrage de la charge et de minimisation des communications. Dans le cas où le nombre de tâches du graphe est égal au nombre de processeurs disponibles (résultat parfois produit par l'ordonnancement), le seul critère à optimiser est celui de la minimisation des communications, en tenant compte pour cela de la topologie du réseau, c'est-à-dire du graphe de processeurs.

2.1.3 Synchronisation et génération de code

Synchronisation :
Pour certain modèles de programmes (type maitre-esclave) et d'architectures (SPMD, SIMD), il est nécéssaire d'effectuer une synchronisation à certains niveaux du programme.

Génération de code :
Dernière étape avant l'exécution, ce module dépend directement de l'architecture cible. Dans le cas où l'environnement intègrerait la possibilité de pouvoir exécuter l'ensemble des tâches sur un ensemble de machines hétérogènes, ce module est en fait composé d'un ensemble de générateurs de code, un pour chaque processeur différent.

2.1.4 Durant l'exécution

A plusieurs reprises dans ce qui précède, il a été question de possibilité de reporter certaines décisions au moment de l'exécution (répartition dynamique de la charge par exemple). Pour espèrer avoir de bons résultats lors de ces prises de décision, il faut la présence d'un module de mesures de qualités et de performances.

Mesures de qualité et de performances
Si jusqu'à cette étape tout pouvait être considéré comme s'exécutant en séquence, il n'en va pas de même pour ce module qui intervient au niveau de l'exécution et dont les résultats peuvent permettre d'agir à deux niveaux. Le premier niveau consiste à faire de l'enregistrement de données et de différer leur analyse et leur exploitation à la fin de l'exécution, en vue d'une prochaine exécution. Le second niveau consiste à utiliser directement les mesures (attente de messages, charge d'un processeur donné), afin de donner davantage d'informations au répartiteur dynamique de charge.

Dans le premier cas, cette technique peut permettre d'affiner les estimations des coûts de calcul et de communication. En revanche, un traitement différé ne permet pas de réagir immédiatement à un événement de contention de canal de communication ou de saturation d'un processeur. Lorsque peuvent survenir des événements qui ne sont pas prédictibles, un traitement immédiat ou une diffusion des informations vers des modules dynamiques semble une meilleure approche. C'est le cas notamment pour des exécutions sur un réseau de stations de travail.

2.1.5 Autres

Il y a dans cet environnement deux modules qui interviennent de manière constante durant tout le processus : un outil graphique et **l'utilisateur**.

- **Outil graphique :**
  A plusieurs niveaux dans ce qui a été décrit, il peut être intéressant de pouvoir visualiser le résultat d'une étape particulière comme par exemple le graphe produit après l'analyse de la dépendance des données, le diagramme de Gantt produit par l'ordonnancement ou bien sur les résultats produits par le module de mesures de performances. Ce module peut inclure des fonctionnalités de dialogue avec l'utilisateur lui permettant de modifier un résultat déterminé automatiquement, ou d'imposer certains choix de placement par exemple.

- **Utilisateur :**
  Cet "élément" que l'on peut considérer en partie comme extérieur au logiciel n'en est pas moins essentiel pour l'amélioration des résultats de l'outil, ou pour son guidage. A la manière de l'outil graphique l'utilisateur intervient de façon diffuse par l'ajout d'annotations dans le programme de départ ou encore par l'imposition de certaines directives pour les étapes de partitionnement, de regroupement ou même d'ordonnancement, ou d'allocation (la modélisation de l'architecture cible est à sa charge). Il joue le rôle principale dans l'interprétation des résultats rendus par le module de mesures de qualités et de performances.

2.1.6 Remarques

- Dans la présentation de cet outil nous avons volontairement omis les outils et fonctionnalités de débogage. Il faut noter que très peu d'EPP propose une telle fonctionnalité.

- L'outil tel qu'il est décrit dans ce qui précède n'est pas très bien adapté à certaines situations particulières pour lesquelles toutes les informations permettant de construire le graphe ne sont pas disponibles. Dans de tels cas de figure, il faut récupérer à l'exécution les informations nécessaires à l'accomplissement de certains modules. Par exemple, pour l'étape d'ordonnancement-placement, si le graphe n'est pas entièrement valué, plusieurs stratégies sont possibles. Il est possible de concevoir une approche de type "à l'exécution" qui reviendrait à s'occuper de l'ordonnancement au moment de l'exécution, ainsi, une approche combinant détermination de la granularité et ordonnancement-placement au moment de l'exécution a été proposée dans le système *Stardust*, le problème majeur étant la dégradation notable des performances due au surcroit de travail demandé aux processeurs. Une autre approche consiste à déterminer la granularité à la compilation et d'effectuer la phase d'ordonnancement-placement en partie ou en totalité à l'exécution. On peut en effet concevoir un outil qui mettrait en oeuvre une heuristique d'ordonnancement-placement à partir de graphes incomplets (dont les arcs et les noeuds ne seraient pas tous valués) au moment de l'étape de compilation, le résultat de cette étape étant amélioré à l'exécution par l'examen des performances (en temps réel) pouvant entrainer des migrations de tâches. Ces différents types d'approches sont discutés dans [Sarkar89].

2.2 Autour de cette approche

Une forte proportion des EPP disposant d'un module ordonnancement-placement se retrouve dans tout ou partie de cette démarche de parallélisation[1].

2.2.1 Réalisations pratiques

Hypertool réalise la presque totalité de la démarche exceptée le détermination de la granularité et des coûts. Il ne contient pas de fonctionnalités de répartition dynamique de la

[1] En annexe "Environnements de programmation", quelques EPP sont décrits plus en détail.

charge [WG90].

Pyrros se retrouve dans la démarche à partir de l'étape d'ordonnancement. Comme Hypertool, aucun traitement dynamique n'est prévu par cet environnement [YG92a].

En revanche, un module de répartition dynamique de la charge est prévu dans **ParaRex**, ainsi qu'une analyse statique du graphe de précédence. On remarque en outre que ce graphe n'est pas déterminé par l'environnement, mais par l'utilisateur à l'aide d'une interface graphique [LS93].

Les programmes EDAM de l'environnement **ADAM** sont constitués de deux parties, une partie pour la description du code des processus, et une partie de description du graphe de communications interprocessus. En outre, l'environnement décrit dans [ACC+93] ne comporte pas de répartiteur dynamique de la charge.

Dans **PPSE**, la première partie du processus est assurée par *Reverse Engineering Tool* couplé avec *Parallax* pour permettre à l'utilisateur de visualiser son application et de pouvoir la décrire comme un graphe. **TaskGrapher** correspond à la partie ordonnancement et placement, alors que *SuperGlue* génère du code parallèle à partir des résultats intermédiaires. Enfin, les mesures de performances sont effectuées à l'aide de *EDA* (*Execution Profile Analyser*). Cet environnement couvre la totalité du processus, à l'exception des fonctionnalités de répartition dynamiques [ERL90].

Les objectifs de **HeNCE** sont légèrement différents. A l'opposé des environnements précédents, l'accent est mis sur la partie dynamique. Un module est chargé de l'allocation des procédures avant et pendant l'exécution du programme, cependant, aucun graphe de précédence n'est produit et analysé, la granularité choisie ne s'y prête pas [BDG+93].

2.2.2 Granularité et coopération statique/dynamique

Du point de vue de l'ordonnancement et du placement, les paramètres importants sont :

- le graphe qui modélise le programme,
- les caractéristiques et l'état de la machine cible,
- et les stratégies employées pour l'ordonnancement et le placement.

Dans le cadre d'une analyse statique, les algorithmes de liste basés sur un critère de chemin critique fournissent de bons résultats en pratique, que les communications soient négligées [ACD74], ou prises en compte [YG93a]. Toujours dans un cadre statique, Gerasoulis et Yang montrent qu'une stratégie de regroupement linéaire (*linear clustering*) permet de produire des ordonnancements asymptotiquement optimaux pour des graphes vérifiant l'hypothèse de forte granularité. La qualité de ces ordonnancements reste la même pour des

graphes ne vérifiant que partiellement les hypothèses de forte granularité [YG93a][2].

Cependant, la plupart de ces expériences ne portent que sur des graphes entièrement déterminés au moment de la compilation. C'est-à-dire des graphes pour lesquels la granularité est fixée. Or, si la granularité est définie comme le compromis entre parallélisation et séquentialisation, c'est-à-dire entre équilibrage de la charge et minimisation des coûts de communications, alors elle dépend fortement de la machine cible. Les caractéristiques architecturales jouent un rôle [Cal95], ainsi que l'état de la machine au moment de l'exécution, surtout si le système est multi-utilisateurs. D'une manière générale, ce problème est contourné par les EPP existants. D'une part parce qu'il s'agit d'un problème difficile [KL88], et d'autre part en raison du bon comportement de certains algorithmes d'ordonnancement qui garantissent de bonnes performances lorsque la granularité varie [YG93b]. Il faut remarquer toutefois qu'un graphe peut vérifier les hypothèses de forte granularité sur une machine donnée, sans qu'elles le soient pour une autre machine. De plus, il ne s'agit que d'une analyse statique, ce qui signifie que selon l'état du système multiprocesseur au moment de l'exécution, la bonne granularité du graphe peut être différente de ce qu'elle serait sur cette même machine dans un autre état. En d'autre termes, pour garantir l'efficacité et la portabilité des applications il est nécessaire de pouvoir considérer une granularité variable.

De la même façon, il n'existe que peu de travaux prenant en considération les graphes irréguliers, dont la forme dépend des données en entrées. C'est le cas notamment des problèmes numériques dont les donnés sont des matrices creuses. Pour ces types de graphes, dont la structure est fortement dépendante des données, une analyse à la compilation n'est pas envisageable. Gerasoulis et al. [GJY94] proposent, pour des problèmes de matrices creuses de construire le graphe en utilisant la factorisation symbolique qui permet de trouver les éléments non nuls de la matrice. Une fois ce graphe déterminé, un ordonnancement statique est produit. Ils illustrent leur réflexion par une étude de certains graphes irréguliers comme le problème à N corps, qui modélise le mouvement d'un ensemble de particules[3]. Ce type de graphe présente un fort parallélisme potentiel puisque les calculs se situent au niveau des feuilles du graphe (les sommets qui n'ont pas de successeurs). Lorsque la découpe est non adaptative, l'espace est subdivisé en un certain nombre de boîtes quelle que soit la densité des particules dans un boîte donnée. Le graphe a toujours la même structure (régulière), et Pyrros permet d'obtenir de bons résultats. En revanche, la découpe adaptative de l'espace donne un graphe irrégulier. La méthode présentée tient à jour des listes de boîtes en fonctions de leur voisinage et de leur taille. A partir de ces informations, un graphe de tâches est construit. Quelques problèmes apparaîssent en cours d'exécution. Les particules changent de boîtes, et le poids de celles-ci varie. En s'appuyant sur le fait que le mouvement des particules est faible au cours d'une étape, et qu'une stratégie de regroupement linéaire s'avère être robuste pour de faibles variations de poids, les auteurs conjecturent qu'un ordonnancement peut être utilisé pour plusieurs étapes d'itérations. La question qui reste en suspens est celle du nombre d'itérations pour lequel les performances peuvent être garanties. Enfin, lorsque l'efficacité de l'ordonnancement de départ n'est plus garantie, un réordonnancement

[2]Cette étude est réalisé pour l'algorithme de Gauss-Jordan.

[3]La méthode employée est la *FMM* (*Fast Multipole Method*) [Gre87]

est effectué à partir de nouvelles données, avec une nouvelle étape de partitionnement de l'espace.

Cette dernière approche essaye d'exploiter les techniques de l'ordonnancement statique dans un environnement d'exécution dynamique. Cette méthode qui consiste à mixer les techniques de l'ordonnancement statique dans un contexte dynamique suit la voie qui fait se rapprocher ordonnancement statique et dynamique. Ce qu'il manque jusqu'à présent est un module qui exploite les données fournies par l'ordonnancement statique, afin de voir plus loin dans l'exécution et de prendre une décision de répartition de charge en conséquence. Parmi les environnements présentés, quelques uns proposent à la fois des fonctionnalités d'ordonnancement statique et dynamique. Mais dans tous les cas, ces deux parties ne coopèrent pas. Une des différences qui existe entre une analyse statique et une analyse dynamique se situe au niveau des données en entrée. Dans le premier cas, le traitement est effectué sur un graphe de précédence. La principale difficulté est la détermination de ce graphe au moment de la compilation (volume des calculs, des communications, structure). Le principal avantage de ce type de traitement est la qualité du résultat obtenu. Dans le second cas, le traitement est effectué sur des tâches nouvellement créées. La principale difficulté est de faire un choix de site d'exécution, sans savoir à l'avance si une tâche est critique ou si son exécution peut attendre sans augmenter la date de fin du programme. L'avantage de ce traitement est une bonne connaissance de l'état de la machine au moment du choix du site, et cette connaissance n'est pas prédictible au moment de la compilation.

Chapitre 3

L'ordonnancement statique dans Athapascan

APACHE est un projet réunissant une vingtaine de personnes, qui vise à concevoir un environnement de programmation parallèle dont l'objectif est la réalisation d'un compromis entre l'efficacité et la portabilité [Pla94]. Ce projet est en passe de devenir un projet INRIA.

3.1 Apache et l'environnement Athapascan

Le projet APACHE s'articule autour de plusieurs axes de recherche centrés autour de l'environnement de programmation ATHAPASCAN [Pla94]. Le style de programmation s'impose de lui même sitôt que les objectifs sont clairement définis. La portabilité impose aux programmes d'être indépendants de la machine cible. La recherche de l'efficacité, en revanche, prend en considération un modèle de machine dans le but de calculer une bonne granularité. Le principe de parallélisation d'un programme est celui de son découpage. Dans de nombreux EPP, le grain est imposé ou la détermination est laissée à la charge de l'utilisateur. Le but poursuivi est de déterminer de façon systématique la bonne granularité selon celle de la machine cible. Un effort particulier est envisagé pour produire de façon statique des résultats qui soient exploitables au moment de l'exécution par le répartiteur dynamique de la charge.

Une application ATHAPASCAN est un programme unique. Le code est chargé sur tous les processeurs, mais un seul d'entres eux exécute le processus de départ (le *main*). La programmation en ATHAPASCAN se fait par l'intermédiaire d'une interface basée sur le langage C et sur le principe des appels de procédures à distance (*RPC*, *Remote Procedure Call*). Dans ce modèle, les communications se font par échange de messages. L'exécution d'un programme utilise un noyau de communication et un noyau de fils d'exécution (*threads*). Le premier permet d'effectuer des communications et de faire évoluer la machine sur laquelle les applications s'exécutent. Le noyau de *threads* permet de créer des processus légers à chacun desquels est attaché un contexte [Chr94]. Le code est annoté par l'utilisateur. Ces annotations peuvent être des indications de complexité d'une phase de calcul, ou d'un volume de communication.

A partir de ces indications, un graphe est construit par l'intermédiaire d'une interface de programmation [Bau95]. Le graphe fourni en entrée du module d'ordonnancement possède une granularité qui dépend des indications de l'utilisateur, mais qu'il est possible de changer et d'adapter par le choix des découpes. Lorsqu'un facteur de découpe est choisi, le module ordonnancement demande une nouvelle phase de détermination de graphe pour le sous-graphe issu de ce nœud. Il y a donc un dialogue entre l'outil de production de graphe et le module ordonnancement-placement. Si l'on revient à la démarche décrite dans le chapitre précédent, cela revient à casser la séquentialisation des étapes de production du graphe et d'ordonnancement-placement par l'ajout d'une interaction depuis ce dernier module vers le précédent (voir la figure 3.1).

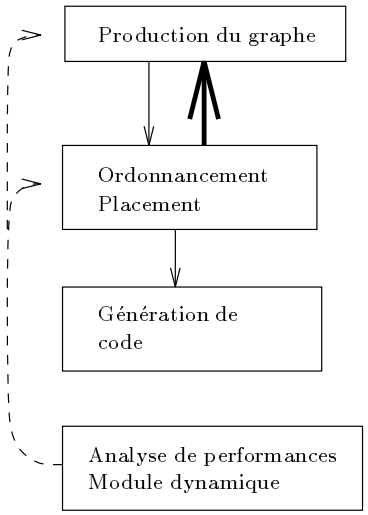


FIG. *3.1 - : La production du graphe et l'ordonnancement-placement statique coopèrent.*

En sortie du module ordonnancement-placement, les dates et sites calculés constituent des données d'entrée pour le générateur de code dans le cas statique ou pour le noyau exécutif dans le cas dynamique.

En cours d'exécution, l'environnement effectue des prises de traces, en vue d'une réexécution déterministe [FdK94], ou pour l'évaluation des performances [MT95, Mai95b], les perturbations introduites sont mesurées [Mai95a]. L'analyse de ces traces, par l'intermédiaire de l'outil de visualisation Scope [Arr94] permet d'ajuster certaines valeurs de nœuds ou d'arcs, de visualiser d'éventuelles contentions dans le réseau, les communications les plus pénalisantes pour l'efficacité de l'exécution de l'application ou toute autre manifestation du même type. Ces résultats peuvent ensuite être exploités pour modifier les calculs de la phase ordonnancement et placement [Bou94].

A côté de l'environnement proprement dit, des stratégies d'ordonnancement sont testées [Azé95], par l'intermédiaire de l'outil ANDES [Kit94] qui constitue une technique pour la modélisation quantitative de graphes. Cet outil a déjà été utilisé avec succès pour l'évaluation des performances d'algorithmes de placement [Bou94, Kit94]. Il permet une exécution réelle

sur une machine cible d'un graphe synthétique, c'est-à-dire que le code effectivement exécuté ne calcule rien, mais correspond à une charge de calcul équivalente à celle de l'application qui est modélisée.

Dans l'environnement ATHAPASCAN, on retrouve à peu près les mêmes modules présentés dans la démarche décrite dans le chapitre précédent. Concernant l'ordonnancement-placement statique et le répartiteur dynamique, les principales différences concernent les interactions de ces modules entre eux et avec les modules voisins. Il y a coopération entre le module ordonnancement-placement statique et la détermination du graphe, et les résultats produits sont acceptés en entrée du répartiteur dynamique qui gère au moment de l'exécution l'allocation des sites et les ordres d'exécution sur chaque processeur. Il utilise pour cela les indications fournies par le module statique si l'état de la machine n'est pas trop éloigné de l'état pris comme hypothèse.

3.2 Ordonnancement statique dans Athapascan

3.2.1 Les programmes et les graphes Athapascan

En ATHAPASCAN, le parallélisme est explicite et est exprimé par le découpage d'un calcul en sous-calculs. La découpe est récursive et peut être arrêtée à chaque étape. L'alternative à la découpe du calcul en sous-calculs est un algorithme séquentiel. Ce mécanisme lié à la notion de **poly-algorithmes** permet d'adapter le grain. D'autre part, dans l'approche choisie, un programme peut être modélisé par un **graphe d'appel de procédures**. Ces graphes sont formés d'une suite d'étapes d'appel de services, correspondant à des nœuds dont l'arité sortante est supérieure ou égale à deux, ou peut être supérieure ou égale à deux (cas des nœuds dont la découpe est adaptative), suivies par des étapes de synchronisations

locales. Un exemple de graphe d'appel de procédures est illustré en figure 3.2.

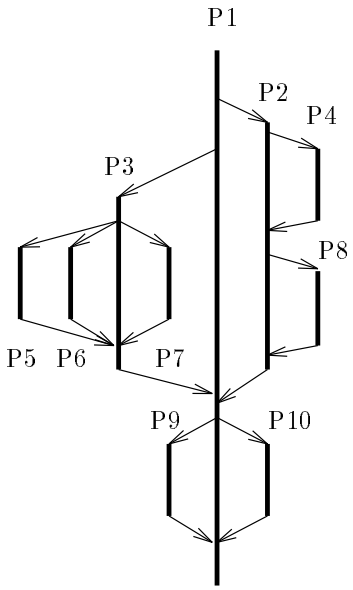


FIG. *3.2 - : Exemple d'un graphe d'appel de procédures.*

Parmi l'ensemble des services requis par l'application, certains permettent une découpe adaptative. Les graphes obtenus sont alors du type de celui qui est illustré en figure 3.3.

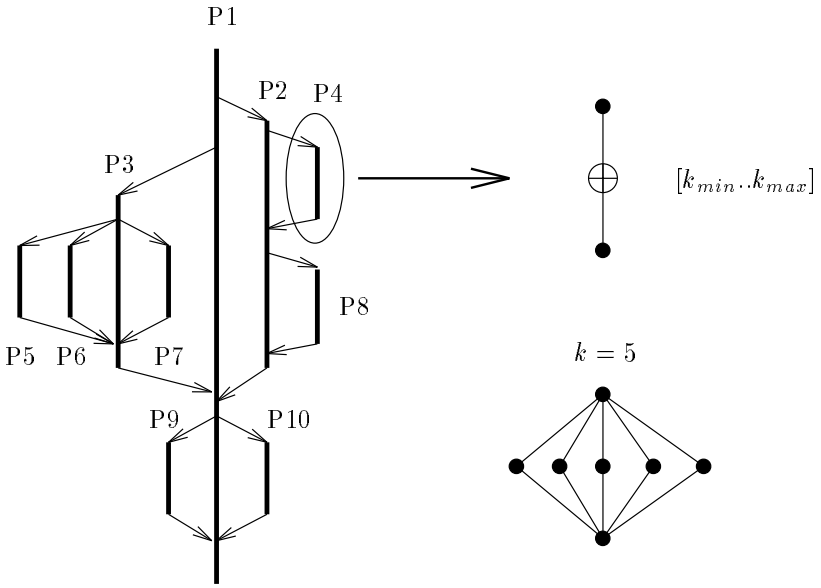


FIG. *3.3 - : Exemple d'un graphe d'appel de procédures avec points de choix.*

3.2.2 La construction et les manipulations du graphe

La construction du graphe se fait par une exécution symbolique. Cette exécution est effectuée à partir de la connaissance de certains paramètres (par exemple la taille des données),

et non de leurs valeurs (évaluation partielle). Une directive permet de masquer les calculs effectifs, et permet donc une interprétation abstraite du programme [Bau95].

Le graphe est constitué de différents types de nœuds.

- Les nœuds SEQ qui correspondent à un ensemble de tâches qui doivent être exécutées en séquentiel,
- les nœuds PAR qui correspondent à un ou plusieurs appels de procédure simultanés (selon l'arité sortante),
- les nœuds SPLIT dont l'arité sortante est variable, ils sont aussi nommé point de choix, car à leur niveau, une décision de découpe peut être prise,
- enfin, les nœuds ALT, qui correspondent aussi à des points de choix, mais dont le facteur de découpe ne peut prendre que quelques valeurs (ce type de nœuds est très voisin du type précédent).

La construction du graphe se fait en coopération avec l'ordonnancement. L'interpréteur abstrait fournit des fonctions pour permettre l'accès aux nœuds et à leurs attributs. Ceux-ci sont au nombre de trois. Le travail séquentiel, que l'on note $T_{seq}(noeud)$ et qui correspond à la somme des travaux qui au niveau d'exploration suivant seraient exécutés de manière séquentielle si une exploration plus poussée n'était pas commandée. Le travail distribué, noté $T_{dist}(noeud)$ et qui correspond à la somme des coûts de calcul qui peuvent être exécutés en parallèle (quel que soit le degré de parallélisme). Enfin, le degré de parallélisme qui représente la largeur du sous-graphe issu du nœud exploré, et qui est noté $dgp(noeud)$. Le travail total d'un nœud est noté $T(noeud)$. Le graphe de départ n'est constitué que d'un méta-nœud. Les opérations sur ces nœuds sont l'expansion en plus de toutes les opérations des nœuds de base. Dans l'exemple de la figure 3.4, le nœud A est un méta-nœud dont l'expansion est un sous-graphe formé des nœuds A_1, A_1F_1 et A_1F_2, et du nœud A_1B.

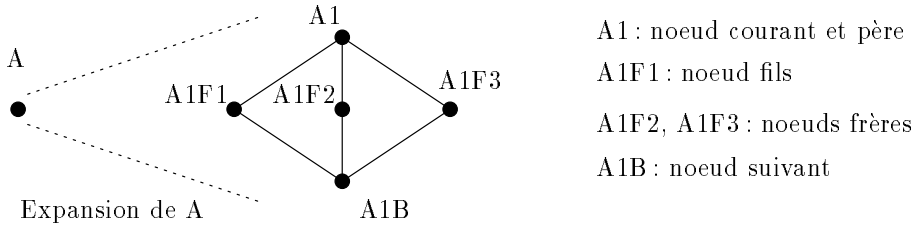


FIG. *3.4 - : Accès aux nœuds et attributs dans un graphe* ATHAPASCAN.

La nécessité d'une exécution symbolique pour la construction du graphe demande la participation du module ordonnancement pour le développement de certains points de choix. De plus, l'expansion mesurée des méta-nœuds permet de calculer un ordonnancement sur un graphe de taille raisonnable, c'est-à-dire que dans le pire des cas seulement, l'ordonnancement sera calculé sur le graphe au grain le plus fin. Le principe du module est le suivant : dans un premier temps, un algorithme de liste est utilisé pour choisir de façon gloutonne un facteur de

découpe dans les nœuds SPLIT qui sont des choix, afin d'aller plus avant dans l'exploration du graphe. S'il s'avère que la priorité donnée à un nœud n'était pas bonne, il est possible de remettre en cause le choix de découpe et celui de l'allocation d'un ensemble de processeurs à ce nœud.

3.2.3 Stratégies d'ordonnancement

Idéalement, les algorithmes utilisés pour ordonnancer des graphes Athapascan devraient présenter des qualités de portabilité, rapidité (faible complexité), adaptabilité aux graphes dont la valuation est incertaine, et devraient, en outre, produire de bons ordonnancements.

L'étude effectuée sur les arbres dans les parties II et III permet d'orienter nos choix vers des stratégies de regroupement. La portabilité est vérifiée (pour les graphes choisis), puisque nous avons vu que l'algorithme MAJYC fonctionne à la fois pour des modèles de machines à mémoire distribuée (modèles PY et RS) ou partagée (modèle ABR), et pour des graphes à grain fin (modèle PY) ou à gros grain (modèles RS et ABR). D'autre part, le regroupement permet de lisser les erreurs d'estimation de coûts de calcul et de communication puisque ces erreurs sont moyennées par la somme des durées des tâches appartenant à un même groupe.

Cependant, il se trouve que les graphes Athapascan sont au départ des méta-graphes pour lesquels le regroupement des tâches est déjà réalisé au niveau ultime puisqu'à l'origine, sans traitement, un graphe Athapascan n'est formé que d'une unique tâche. Deux différentes démarches peuvent être envisagées.

La première consiste à développer entièrement le graphe, par expansion récursive des tâches rencontrées. Le graphe récupéré à l'issue de cette étape d'expansion maximale présente le grain le plus fin voulu par l'utilisateur. Ce graphe peut ensuite être regroupé de plusieurs manières.

La première pourrait être celle qu'utilisent les concepteurs de *Pyrros* avec un algorithme de type *DSC (Dominant Sequence Clustering)*. La seconde se rapproche de la méthode employée par Colin et Chrétienne [Col89] qui consiste à extraire d'un *DAG* un graphe critique dont la forme est une forêt d'anti-arbres. Cette forêt peut ensuite être ordonnancée par l'une des heuristiques présentées dans les parties II ou III selon les caractéristiques de la machine et du graphe, moyennant de plus le respect des contraintes de précédence n'apparaissant pas dans le graphe critique. Il est à noter que l'algorithme qu'ils proposent produit des ordonnancements optimaux lorsque le nombre de processeurs n'est pas limité, et lorsqu'une certaine condition de grosse granularité est vérifiée localement, partout dans le graphe. Cependant, si cette condition n'est pas vérifiée, l'algorithme produit malgré tout des ordonnancements faisables.

Cette première démarche présente l'avantage de produire de bons ordonnancements. Cependant, la complexité des algorithmes mis en œuvre est de l'ordre de $O(n^2)$ voire $O(n^3)$, ce qui limite leur utilisation à des graphes formés d'un faible nombre de nœuds. D'autre part,

la structure du code de départ est perdue du fait de l'éclatement maximal du graphe. Enfin, il est possible que des instructions de condition apparaissent. Ces instructions rendent le traitement statique inopérant dans la majeure partie des cas.

La seconde démarche consiste à descendre dans le graphe par expansions successives des tâches les plus grosses. Les stratégies que l'on peut employer sont de type glouton ou autorisant la remise en question de décisions antérieures (*backtracking*). Ce type d'approche présente plusieurs avantages :

- algorithmes rapides (complexité linéaire en le nombre de nœuds,
- algorithmes adaptables. La profondeur des expansions successives peut être paramètrée, et l'arrêt de l'algorithme peut également être piloté par des seuils (accélération ou efficacité),
- certaines instructions conditionnelles sont noyées dans un groupe de tâches qui n'a pas été développé, et ne posent par conséquent pas de problème,
- enfin, la portée des incertitudes des estimations de coûts est amoindrie par la forte granularité des tâches.

Dans la suite, nous présentons deux stratégies en cours d'implémentation. La première est relativement simple, très rapide et peut être utilisée dans un cadre dynamique. La seconde est plus proche du graphe et tente de tenir compte de la structure interne des tâches afin de tirer le meilleur partie d'une éventuelle redistribution des processeurs.

3.2.4 Première stratégie

Algorithme

Dans les graphes Athapascan, chaque nœud peut être de l'un des quatre types suivants :

- **SEQ** : un nœud de type SEQ est un nœud dont l'expansion au premier niveau est une chaîne formée de deux nœuds. Mais il est possible que l'expansion de l'un de ces nœuds mette en évidence du parallélisme.
- **SPLIT** : est un nœud dont la découpe est variable. Il peut s'agir, par exemple, d'un produit de matrices. Le nombre de sous-tâches est alors le résultat de la découpe en nombre de lignes et colonnes allouées à chaque processeur. On definie cependant une borne maximale de découpe k_{max}.
- **PAR** : est un nœud dont l'arité sortante est fixée (supérieure à un) et dont les tâches issues de son expansion ne sont pas nécessairement les mêmes (comme c'est le cas par exemple d'un nœud SPLIT).
- **ALT** : L'intervalle de découpe $[1..k_{max}]$ du SPLIT devient $k_1, k_2..., k_{max}$.

La stratégie qui suit est entièrement pilotée par le type du nœud. Elle s'applique de façon récursive. Dans l'algorithme suivant, T représente un noeud et $Nprocs$ le nombre de processeurs disponibles pour ordonnancer ce noeud. La quantité totale de travail d'une tâche T_i est notée $W(T_i)$.

OrdoAthap1(T,Nprocs)

Si Nprocs = 1 **Alors**
  T est ordonnancé sur ce processeur
Sinon
  Selon
  type(T) = SEQ (* T = T1;T2 *)
    OrdoAthap1(T1,Nprocs)
    OrdoAthap1(T2,Nprocs)
  type(T) = SPLIT (* T peut être découpé en plusieurs parties *)
    Si $Nprocs > k_{max}$ **Alors**
      allouer à une partie des tâches T_i $(Nprocs)div(k_{max})$ processeurs
      OrdoAthap1(T_i,$(Nprocs)div(k_{max})$)
      allouer aux autres tâches T_j $(Nprocs)div(k_{max})+1$ processeurs
      OrdoAthap1(T_j,$(Nprocs)div(k_{max})+1$)
    Sinon
      partager le graphe en Nprocs parts égales
      appliquer un algorithme du type LPTF (Largest Processing Time First)
    FinSi
  type(T) = ALT (* T peut être découpé en des nombres fixés de parties *)
    partager le graphe en le nombre maximum de parts
    et même comportement que pour le noeud SPLIT
  type(T) = PAR (* tous les successeurs d'un noeuds ne sont pas de poids égaux *)
    expansion de T au niveau 1
    calculer pour chacune des tâches T_i issues de la découpe la proportion
    de processeurs qui doivent leur être alloués
    $NprocsAll(T_i) = (W(T_i) \times Nprocs)div(\sum W(T_i) - Wseq(T))$
  FinSelon
FinSi

Regrouper les tâches ayant $NprocsAll(T_i) = 0$, en nombre
suffisant pour que le groupe ait un processeur

Si il subsiste un groupe de trop faible poids, il est concaténé
au groupe ou à la tâche de plus faible poids

Vérifier que le nombre de processeurs distribué est suffisant
sinon, allouer les processeurs restant aux groupes de plus forts poids

Pour chaque groupe formé par une seule tâche **Faire**

OrdoAthap1(T,NprocsAll)
FinPour

Pour tous les groupes tels que $NprocsAll = 1$ **Faire**
ordonnancer en séquence les noeuds du groupe
FinPour

Pour les groupes ayant plusieurs noeuds et tels que $NprocsAll > 1$ **Faire**
ordonnancer toutes les petites tâches
celles pour lesquelles $NprocsAll = 0$
enfin pour la plus grosse tâche T_G
OrdoAthap1(T_G,NprocsAll)
FinPour

Exemple

Nous partons dans les conditions réelles d'une seule tâche dont nous ne connaissons que le type au niveau 1 de découpage, ainsi que le travail totale contenu dans cette tâche. Du fait que cette tâche est de type SEQ, on sépare le traitement de la première sous-tâche T1 (voir figure 3.5) de celui de la tâche T2 (voir figure 3.6).

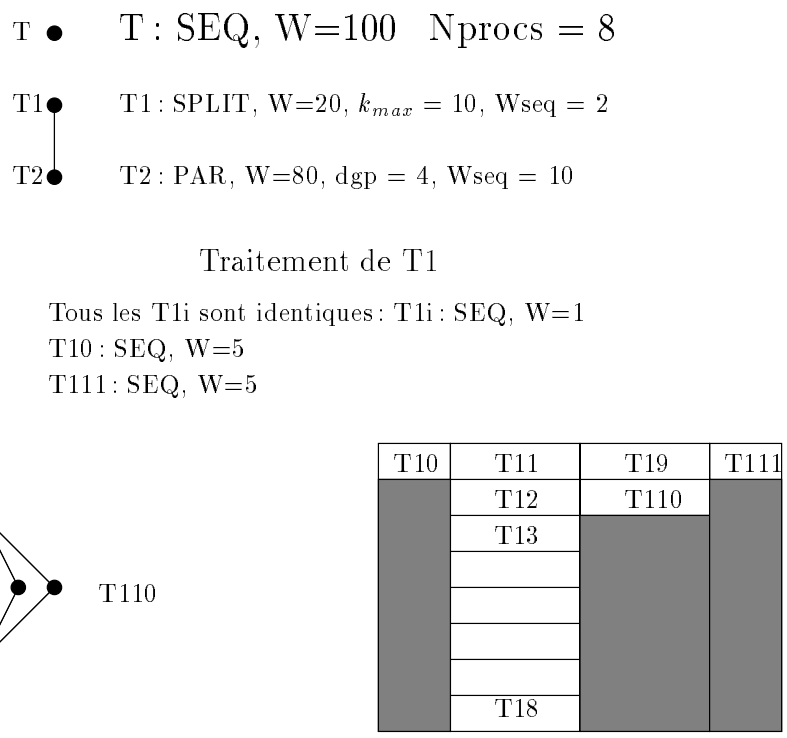
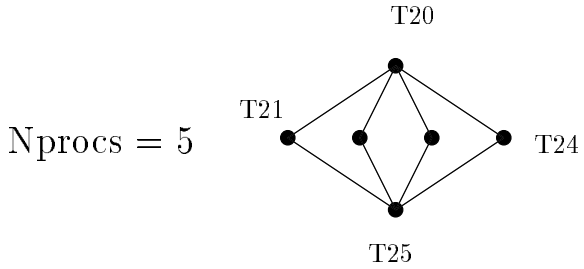


FIG. *3.5 - : Traitement de la première sous-tâche.*

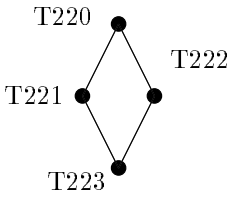
Traitement de T2



Nprocs = 5

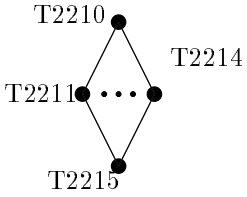
T20 : SEQ, W=5
T21 : SEQ, W=10 — NprocsAll(T21) = 1
T22 : PAR, W =40, Wseq = 4, dgp = 2 — NprocsAll(T22) = 4 + 1
T23 : SEQ, W=15 — NprocsAll(T23) = 1 + 1
T24 : PAR, W = 5, Wseq =2, dgp=3 — NprocsAll(T24) = 0
T25 : SEQ, W =5 — groupe1 : T21+T24

Traitement de T22



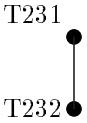
T220 : SEQ, W =2
T221 : PAR, W =30, Wseq = 4, dgp = 4
T222 : SEQ, W =6
T223 : SEQ, W =2
NprocsAll(T221) = 4
NprocsAll(T222) = 1

Traitement de T221



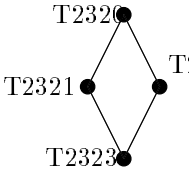
T2210 : SEQ, W =2
T2211 = T2213 : SEQ, W =7
T2212 = T2214 : SEQ, W =6
T2215 : SEQ, W =2
NprocsAll(T221i) = 1

Traitement de T23



T231 : SEQ, W =3
T232 : PAR, W =12,
Wseq = 2, dgp = 2

Traitement de T232



T2320 : SEQ, W =1
T2321 : PAR, W =5,
Wseq = 2, dgp = 2
T2322 : SEQ, W =5
T2323 : SEQ, W =1

FIG. *3.6 - : Traitement de la seconde sous-tâche.*

Le résultat du graphe que l'on a découpé est le graphe illustré en figure 3.7, ainsi que l'ordonnancement obtenu.

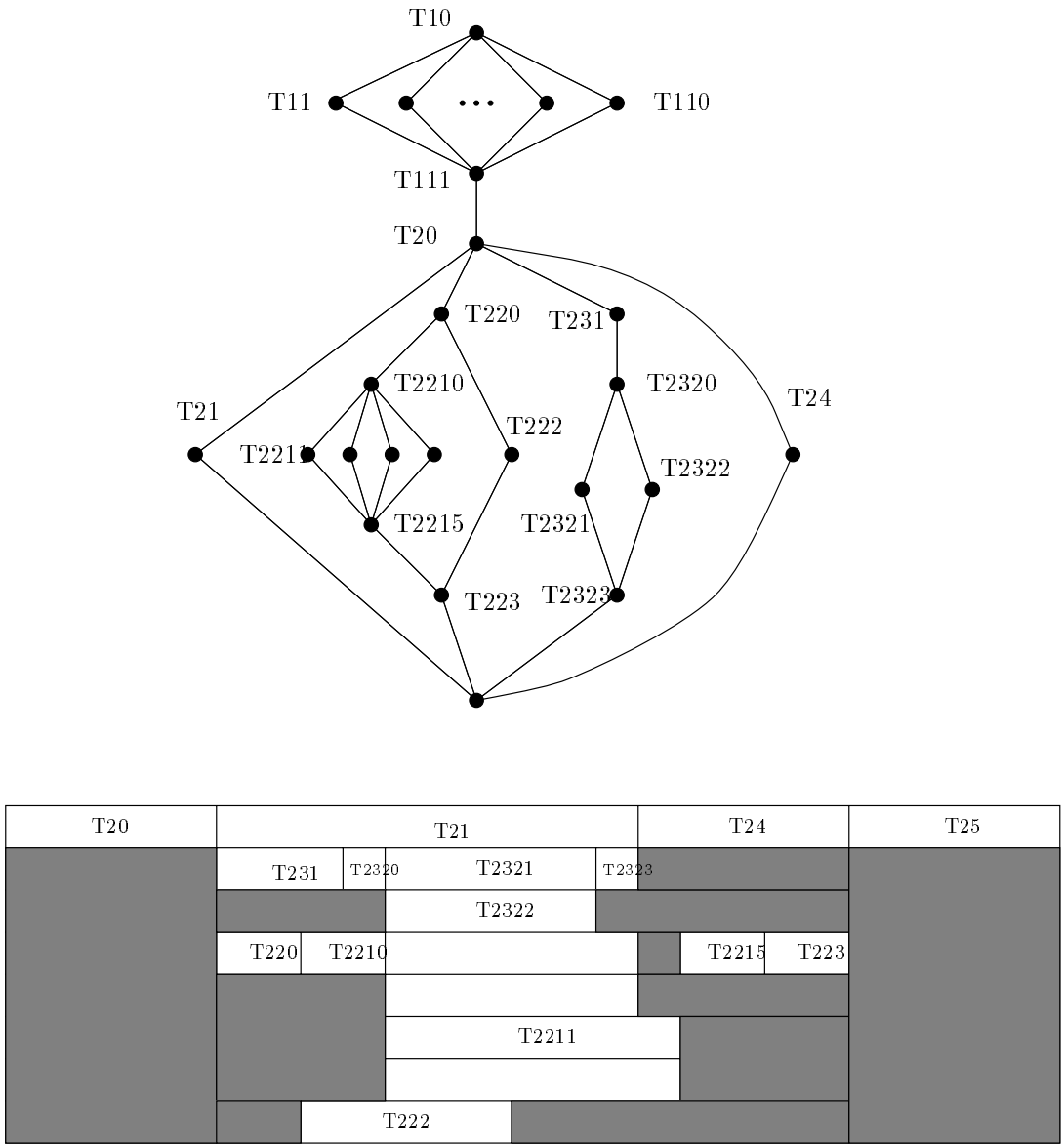


FIG. *3.7 - : Résultat final.*

3.2.5 Seconde stratégie

Le départ de cette seconde stratégie est le même que pour la première. On procède à l'expansion de la tâche de départ.
Comme pour le cas précédent, les noeuds qui posent le plus de problème sont les noeuds PAR, car le parallélisme qu'ils renferment n'est pas visible et il est hasardeux d'essayer d'en faire une estimation avant d'avoir exploré effectivement le graphe.
Le principe de ce second algorithme est en deux temps. Premier temps, on effectue une

découpe mesurée, de telle sorte que le nombre de processeurs nécessaire pour réaliser l'ordonnancement calculé est un multiple MP du nombre de processeurs disponibles.

Si au départ le nombre de tâches est plus important que Nprocs **Alors**
  appliquer un algorithme de type LPTF
Sinon
  on dispose dans un tableau l'ensemble des noeuds
  le premier champ correspond au travail total
  le second champ correspond au travail séquentiel
  le troisième correspond à dgp
  le dernier au plus long chemin dans le sous-graphe
FinSi

ameliore = Vrai
NprocUtilise = nombre de noeuds dans le tableau
NprocUtiliseAvantDecoupe = NprocUtilise

TantQue (ameliore OU NprocUtilise = NprocUtiliseAvantDecoupe) ET ($NprocUtilise \leq MP \times Nprocs$
  choisir le noeud Nd de plus long chemin dans le tableau et procéder à
  la découpe du plus lourd noeud Nl dans le sous-graphe issue du noeud Nd
  NprocUtilise = NprocUtiliseAvantDecoupe + dgp(Nl)-1
  calculer le nouveau plus long chemin
  Si ce nouveau plus long chemin n'améliore pas l'ancien **Alors**
    ameliore = Faux
  Sinon
    ameliore = Vrai
    reporter dans la case du tableau la nouvelle valeur du plus long chemin
  FinSi
FinTantQue

ordonnancer ce graphe sur Nprocs à l'aide des strat*acutee*gies
classiques de l'ordonnancement statique (algorithmes de regroupement par exemple)

3.2.6 Autres stratégies

Les deux stratégies précédentes présentent l'avantage d'être de complexité faible. Cependant, elles présentent d'importantes carences. Notamment, lorqu'une tâche utilise un processeur, il est perdu pour toutes les autres tâches durant la totalité de l'exécution.

Afin de remédier à cet inconvénient, nous suivons actuellement une idée qui consiste à maintenir simultanément un état de développement du graphe et un état de la machine ou du diagramme de Gantt, ce qui est la même chose pour ce qui nous occupe. Ainsi, lorsque le plus long chemin change de sous-graphe, le développement d'une tâche peut réutiliser tout

ou partie des processeurs en fonction de leur disponibilité aux dates d'exécution logiques des nouvelles tâches. Dans un premier temps on peut interdire de bouger l'état de la machine, et dans un second temps il est envisageable de pouvoir autoriser le déplacement de tâches ordonnancées antérieurement, ce qui revient à autoriser une certaine forme de *backtracking*.

3.3 Conclusion et perspectives

L'approche d'ATHAPASCAN est voisine de l'approche choisie dans Pyrros. La principale différence, pour l'ordonnancement statique, est la possibilité en ATHAPASCAN de ne pas développer un graphe jusqu'au grain le plus fin. De plus, les points de choix constituent des réserves de parallélisme et donnent une garantie de robustesse des ordonnancements construits, dans la mesure où le découpage proposé par l'ordonnancement statique peut être remis en cause au moment de l'exécution par le module de répartition dynamique de la charge.

Le prototype est en cours d'élaboration, en collaboration avec Emmanuel Baud pour l'interpréteur abstrait, et avec Jean-Louis Roch, Alexandre Vermeerbergen et François Galilée pour la partie dynamique.

Les premiers résultats obtenus à partir de la première stratégie indique que son comportement est d'autant meilleur que le graphe est régulier et présente dès le départ un fort parallélisme. Il est intéressant d'autre part de constater que l'algorithme ne descend pas à un niveau de découpe très important, ce qui veut dire d'une part que le nombre de tâches exploré est faible, et d'autre part que l'on conserve tout au long de l'ordonnancement des graphes à très gros grain.

Le développement de stratégies pour ce type de graphe n'en est qu'à ses débuts. Les approches considérées jusqu'à présent tentent de façon plus ou moins adroites de se rapprocher de ce qui est connu, c'est-à-dire un graphe entièrement connu et faire sur ce graphe de l'ordonnancement sur un graphe qui est stable du point de vue de la granularité. Il pourrait cependant être intéressant d'examiner de nouvelles stratégies permettant de considérer une allocation (en nombre) variable pour les différent sous-graphe au fur et à mesure de l'exploration.

Partie V

Conclusion et perspectives

Nécessité de la modélisation

Le calcul parallèle engendre un ensemble de problèmes d'algorithmique et d'optimisation que l'on ne peut traiter dans toute leur complexité en un temps raisonnable. La difficulté intrinsèque d'une partie de ces problèmes, l'optimisation combinatoire par exemple, est une des raisons de cette complexité, mais ce n'est pas la seule. La prise en compte des paramètres liés à l'architecture d'un ordinateur parallèle et intervenant lors de l'exécution du programme en est une autre. On a recours dans ce dernier cas à une représentation de la machine, que l'on appelle un modèle, qui ne prend en considération qu'une partie des paramètres.
Cet effort de modélisation porte également sur le programme. Le modèle qui lui est associé rend compte de la dépendance entre les parties du programme. Ces parties constituent les tâches, ce sont des ensembles d'instructions dont la taille dépend du choix de la granularité. Si les dépendances entre deux tâches sont unidirectionnelles (ou inexistantes), pour tous les couples de tâches, alors le graphe est orienté et les relations de précédence sont représentées par des arcs.

L'approche des environnements de programmation parallèle

Si la modélisation de la machine reste dans une large mesure à la charge de l'utilisateur, de plus en plus de plates-formes de développement prennent en charge la détermination du graphe, éventuellement avec l'aide du programmeur. Nous avons mis en évidence une approche commune à une majorité d'environnements de programmation parallèle existants qui privilégient le parallélisme de contrôle exprimé de façon explicite. Cette démarche, description du graphe de précédence, ordonnancement (et/ou placement), défini comme l'allocation à chaque tâche d'une date et d'un site d'exécution, et la génération de code, est actuellement traitée de manière séquentielle.
Malheureusement, les problèmes rencontrés dans la phase de l'ordonnancement et du placement sont fortement liés à la détermination du graphe de précédence et en particulier à celle de sa granularité. Cette constatation, parmi d'autres, a amené les initiateurs du projet Apache à considérer un mode d'expression du parallélisme selon le découpage (éventuellement récursif) d'un calcul en sous-calculs, le but étant de conjuguer les objectifs d'efficacité et de portabilité des programmes Athapascan. Dans ce contexte nous réalisons actuellement un prototype d'ordonnanceur statique travaillant en coopération avec un interpréteur abstrait qui réalise une évaluation partielle pour la détermination et l'exploration d'un graphe de précédence. A l'heure actuelle, cette maquette a pour but de valider l'approche de la description du graphe indépendammenet des performances des stratégies d'ordonnancement. Si le module ordonnancement utilise actuellement des heuristiques basées sur des stratégies gloutonnes, à court terme d'autres stratégies, plus performances, seront testées.

Problématique

Si pour un graphe et un modèle d'exécution[1] donnés, il est parfois possible de déterminer un ordonnancement de bonne qualité, voire optimal, ce n'est plus vrai en général, lorsque

[1] un modèle de machine plus des hypothèses d'exécution telle que la duplication

certaines des hypothèses ne sont plus vérifiées ou lorsque les graphes varient légèrement. C'est dans une optique de recherche d'algorithmes adaptables facilement pour être utilisés dans le cadre d'environnements d'exécution différents et pour des graphes dont le rapport entre les temps de communications et les temps de calcul des nœuds est différent, que nous avons décrit divers modèles d'exécution et entamé une discussion sur les garanties de performances des ordonnancements par passage d'un modèle à un autre. Ce type d'étude permet d'avoir une idée un peu plus précise sur les stratégies qui semblent répondre à ces exigences.

C'est dans cet état d'esprit que nous avons examiné dans le détail une stratégie d'ordonnancement basée sur des techniques de regroupement de tâches. Comme famille de graphes, nous avons choisi les arbres pour leur importance dans le calcul numérique, et comme représentants d'algorithmes classiques de recherche et de tri par exemple. Nous nous sommes attachés essentiellement à deux modèles d'exécution. Le premier modèle que l'on peut considérer comme un représentant des modèles de machines à mémoire distribuée autorise le recouvrement des communications par des calculs [PY88, RS87]. Le second modèle est basé sur la notion de surcoût qui représente la somme des communications et des temps d'inactivité, et peut être considérer à ce titre comme un représentant des modèles de machines à mémoire partagée [ABR90].

Ordonnancement d'arbres par regroupement

Nous avons proposé un algorithme (MAJYC) pour ordonnancer, sur deux processeurs identiques, des arbres dont les tâches et les communications sont unitaires (arbre dit *UECT*) et nous avons prouvé son optimalité dans le premier modèle[2]. Il a été montré de plus, que pour des variations du rapport entre temps de communications et temps d'exécution (changement de la granularité de l'arbre), une adaptation très simple permet de produire des ordonnancements de bonne qualité et asymptotiquement optimaux (pour un grand nombre de nœuds). En outre, moyennant une analyse non triviale, et une modification plus importante, nous avons montré que cet algorithme permet de résoudre un problème ouvert qui constitue une généralisation, par l'adjonction de communications, d'un travail dû à Leung et al. [NLH81, DL88]. Il permet également une amélioration, en terme de complexité, d'un résultat dû à Jung et al. [JKS89]. Toujours dans le même modèle d'exécution, mais cette fois pour un nombre de processeurs plus important (et fixé), nous avons produit une heuristique applicable pour des arbres dont les coûts de communications sont plus grands que les coûts de calcul (arbres à fine granularité) et réciproquement (arbres à gros grain). L'analyse de la borne au pire de cette heuristique [GRT95] pour des arbres *UECT* a permis d'améliorer la borne produite par Varvarigou et al. [VRKL94] d'un facteur deux. Après avoir constater les bonnes performances obtenues par une stratégie de regroupement pour un modèle d'exécution donné et pour différentes granularités, nous avons étudier son comportement lorsque le modèle d'exécution variait. Dans celui que définissent Anderson et al. [ABR90], nous avons montré que le problème de l'ordonnancement des arbres binaires *UECT* est NP-complet sur deux processeurs. Cependant, bien que la qualité des ordonnancements produits par une

[2]Ce problème a été résolu simultanément par plusieurs groupes à partir de techniques différentes [VRKL94, Vel93a, Pic93].

itération de MAJYC n'est pas été étudiée, l'optimalité est atteinte pour les arbres binaires complets et les arbres de type chenille, ce qui laisse supposer un bon comportement pour des arbres moins réguliers. C'est encore dans la même optique que nous avons étudié le comportement de ce type d'algorithmes dans le cadre d'un système formé de processeurs dont les vitesses sont différentes mais linéairement comparables (processeurs dits uniformes). Un système formé de processeurs uniformes constitue un modèle simple de réseau de stations de travail. Dans un tel environnement d'exécution, nous avons montré qu'il est possible de produire des ordonnancements distants d'au plus une unité de temps de l'optimal pour des arbres *UECT* et un système biprocesseurs dont les inverses des vitesses sont des entiers. Nous avons poursuivi l'étude pour un nombre de processeurs supérieur ou égal à deux et pour des arbres complets. Bien que l'analyse au pire n'aie pas été réalisée, au vue des premiers résultats, il semble que la qualité des ordonnancements produits par cette heuristique soit bonne. Le travail mené sur le problème de l'ordonnancement d'un graphe de précédence avec délais de communications sur un système formé de processeurs uniformes est à notre connaissance la première tentative du genre.

Bien que toutes les études ne soient pas terminées, nous pensons pouvoir conclure que les algorithmes basés sur des stratégies de regroupement de tâches sont intrinsèquement plus adaptables aux changements de modèle d'exécution et de granularité, que d'autres algorithmes basés, par exemple, sur des stratégies de listes.

Perspectives

Les perspectives à court terme concernent l'environnement ATHAPASCAN, avec la mise en œuvre d'une ou plusieurs heuristiques qui puissent exploiter efficacement le granularité variable des graphes d'appels de procédure avec points de choix. Dans le cadre de cet environnement, une étude tournée davantage vers le comportement dynamique des exécutions est envisagée : mise en œuvre de courte phase d'ordonnancement "statique" en cours d'exécution, ou d'utilisation de stratégies classiquement appliquées au moment de la compilation. Toujours à court terme, la validation expérimentale, à l'aide du modèle quantitatif de programmes ANDES et du modèle de machines MADRE[3], est envisagée pour les diverses heuristiques proposées dans cette thèse, afin de déterminer la qualité des ordonnancements qu'elles produisent. A plus long terme, il semble intéressant de rechercher des stratégies basées sur des techniques de regroupement afin de satisfaire les objectifs conjugués d'efficacité et de portabilité, et de traiter des graphes de précédence quelconques.

[3]Modèles de base du projet ALPES (ALgorithmique Parallèle et Évaluation de Systèmes).

Partie VI

Annexes

Chapitre 1

Annexe définitions

Le but de cette annexe est de définir les différents termes utilisés dans ce document. Cependant, il est possible que lorsqu'un travail est cité, les termes employés diffèrent de ceux qui seront définis ici, cette différence sera indiquée dans le texte. Les définitions sont composées d'un terme français, suivi le plus souvent de son équivalent anglais (ou considéré comme tel) et de la définition proprement dite.
Les définitions sont classées par thèmes. Les premières définitions concernent les graphes. Le second thème correspond aux processeurs, et à certaines hypothèses d'exécution et le troisième est dédié à la description de quelques algorithmes. Suivent quelques définitions d'ordre plus général.

Graphe

- **graphe non orienté** (*undirected graph*) :
  Un graphe G est un couple (S, A) formé de deux ensembles disjoints tels que tout élément de A est une **paire** d'élément de S. S est appelé l'ensemble des **sommets** et A l'ensemble des **arêtes**.

- **graphe orienté sans cycle** (*directed acyclic graph ou DAG*) :
  Lorsque le graphe est orienté, tout élément de A est un **couple** d'éléments de S, et A l'ensemble des arêtes devient l'ensemble des **arcs** du graphe. Le graphe est dit sans cycle ou acyclique si et seulement si il n'existe pas de **chemin fermé** dans le graphe.

- **chemin** :
  Soit $G = (S, A)$ un graphe orienté, un chemin est une suite alternée de sommets et d'arcs : $s_0\ a_0\ s_1\ ...\ a_{k-1}\ s_k$. Lorsque $k > 0$ et $s_0 = s_k$, le chemin est dit **fermé**.

- **chaîne** :
  Soit $G = (S, A)$ un graphe non orienté, une chaîne est une suite alternée de sommets et d'arêtes : $s_0\ a_0\ s_1\ ...\ a_{k-1}\ s_k$. Lorsque $k > 0$ et $s_0 = s_k$, la chaîne est dite **fermée**.

- **nœud** :
  Les termes nœud et sommet sont considérés comme équivalents.

- **graphe de tâches** (*task graph*) :
  On appelle graphe de tâche la modélisation d'un programme. Dans un tel graphe, les sommets sont identifiés aux tâches de calcul du programme.

- **tâche** (*task*) :
  Une tâche représente une partie de l'application modélisée par un graphe et correspond à un nœud de ce graphe.

- **graphe à flots de données** (*data-flow graph*) :
  Il s'agit d'un graphe dans lequel la précédence est induite par les échanges des données.

- **prédécesseur** (avec distinction immédiat ou non) :
  Dans un graphe orienté, le nœud N est un prédécesseur du nœud N' si et seulement si il existe un chemin allant de N à N'. N est un prédécesseur immédiat de N', si et seulement si le couple (N, N') est un élément de A.

- **successeur** :
  Dans un graphe orienté, le nœud N est un successeur du nœud N' si et seulement si il existe un chemin allant de N' à N. N est un successeur immédiat de N', si et seulement si le couple (N', N) est un élément de A.

- **distance** :
  Pour un graphe donné, la distance entre deux nœuds correspond à la somme minimale des valeurs des arcs et sommets traversés pour aller d'un nœud à l'autre. Sauf indication contraire, leur propres valeurs ne sont pas prises en compte. Dans un contexte de communication dans les réseaux point à point, la distance entre deux processeurs correspond au nombre de liens minimum que doit traverser un message pour aller du processeur source au processeur destination.

- **niveau** (*level*) :
  Le niveau d'un nœud N dans un graphe est égal au nombre maximum de nœuds qui le séparent d'une feuille F, N et F compris. Le niveau de la racine d'un arbre est égal à 1. Dans le cas où les tâches sont unitaires, le niveau d'une tâche est aussi égal à la longueur d'un plus long chemin (en terme de temps d'exécution) vers une feuille.

- **hauteur** :
  La hauteur d'un arbre est le nombre maximum de sommets liant la racine à une feuille, racine et feuille comprises. La hauteur d'un arbre formé d'un sommet est égale à un.

- **largeur d'un graphe** :
  La largeur d'un graphe est égal au nombre maximum de nœuds appartenant à un niveau pour l'ensemble des niveaux du graphe.

- **travail** (*job*) :
  Dans le cadre du modèle ABR, des tâches peuvent être regroupées au sein d'une même entité : le travail. L'exécution du travail ne peut pas être interrompue, c'est-à-dire que seules les première et dernière tâches appartenant au travail peuvent échanger des

données avec des tâches appartenant à d'autres travaux. Si les tâches intermédiaires communiquent avec d'autres travaux, la reception des données est anticipée au début du travail, et l'envoie de données est retardée à la fin de l'exécution de ce travail.

- **granularité** (*granularity*) :
  Il s'agit de la taille des tâches en nombre d'instructions. Dans le modèle d'exécution décrit par Papadimitriou et Yannakakis [PY88] la granularité est entièrement déterminée par la donnée de τ.

- **arbre et anti-arbre** (*intree et outtree*) :
  Nous définissons un arbre par un graphe dont tous les noeuds ont un seul successeur, à l'exception de la racine qui n'en a pas. Les feuilles d'un arbre n'ont aucun prédécesseur. Dans un tel graphe, l'orientation va donc des feuilles vers la racine. Par opposition, les noeuds d'un anti-arbre ont un seul prédécesseur, à l'exception de la racine qui n'en a pas. L'orientation va de la racine vers les feuilles. Celles-ci n'ont aucun successeur.

Processeurs, architectures et modèles d'exécution

- **processeurs identiques, uniformes et sans relation** (*identical, uniform and unrelated processors*) :
  Dans un système parallèle, les processeurs sont dits **identiques** lorsqu'ils sont capables d'exécuter chaque tâche, et que la durée de l'exécution d'une tâche T est la même sur chaque processeur.
  Les processeurs sont dits **uniformes** lorsqu'ils diffèrent les uns des autres par leur vitesse. Mais, ils n'ont pas de préférence pour un type particulier de tâches, ce qui signifie que si un processeur est deux fois plus rapide qu'un autre, il le sera pour toutes les tâches soumises au système.
  Enfin, les processeurs sont dits **sans relation** lorsqu'ils préfèrent un certain type de tâches, ainsi, l'exécution d'une tâche T peut durer deux fois plus longtemps sur un processeur P_1 que sur un processeur P_2, et l'exécution d'une tâche T' peut durer quatre fois plus longtemps sur P_2 que sur P_1.

- **duplication** :
  Parmi l'ensemble des modèles d'exécution, certains autorisent la duplication des tâches, il s'agit de permettre le calcul d'une même tâche par plusieurs processeurs dans le but d'économiser une communication.

- **préemption** :
  L'hypothèse de préemption consiste à pouvoir interrompre l'exécution d'une tâche. Mais, ceci ne permet pas de considérer que la tâche peut être partagée en plusieurs parties exécutées simultanément, hypothèse que nous appellerons **divisibilité**. On peut interpréter la préemption comme une hypothèse de partage de la tâche dans le temps.

- **divisibilité** :
  Hypothèse selon laquelle une tâche peut être parallélisée. On peut interpréter la divisibilité comme une hypothèse de partage de la tâche dans l'espace.

- **recouvrement** des communications par du calcul (*overlap*) :
  Cette hypothèse permet de considérer qu'un processeur peut simultanément envoyer des données, exécuter des tâches et recevoir des données. Pratiquement cela signifie que l'on suppose que le processeur est équipé d'un co-processeur dédié aux communications qui a un accès indépendant à la mémoire.

Algorithmes

- **algorithme LPT** (*Largest Processing Time*) :
  Cet algorithme de liste consiste à construire une liste ordonnée des tâches avec comme critère de priorité leur durée d'exécution. Une tâche est de priorité d'autant plus grande qu'elle requiert un temps d'exécution élevé. L'utilisation de la liste est simple, l'algorithme place la tâche qui est en tête de liste sur le processeur sur lequel elle se terminera le plus tôt (à condition que cette tâche soit prête à être exécutée).

- **algorithme HLF** (*Highest Level First*) :
  Il s'agit d'un algorithme de liste dans lequel le critère de priorité est le niveau des tâches. Le plus connu des algorithmes *HLF* est incontestablement celui présenté par Hu [Hu61].

- **algorithme HLA** (*Highest Level first with Abstention*) :
  Il s'agit une fois encore d'un algorithme de liste, la priorité est encore le niveau, mais cette fois, l'algorithme se réserve le droit de ne pas exécuter la tâche la plus prioritaire, en fonction des situations qu'il gère.

Divers

- **ensemble dominant** :
  un sous-ensemble d'ordonnancements est dit dominant lorsque ce sous-ensemble contient un ordonnancement optimal.

- **ordonnancement** (*scheduling*) :
  supposons que l'on dispose d'un ensemble de machines pour traiter un ensemble de tâches ou de travaux. Un ordonnancement est l'allocation d'un ou plusieurs intervalles de temps (cas préemptif) sur une ou plusieurs machines (cas des tâches multiprocesseurs) à chaque tâche ou travail. Un ordonnancement est **valide** ou **faisable** si pour chaque intervalle de temps, il n'existe pas deux tâches distinctes devant être exécutées sur à la même machine et si toutes les contraintes liées aux machines et aux tâches (ou travaux) ont été respectées (dans la majorité des cas que nous traiterons, il n'y aura aucune contrainte supplémentaire). Un ordonnancement est dit **optimal** si il minimise un critère d'optimalité donné. Dans la plupart des problèmes traités dans ce document, le critère choisi est le maximum des dates de fin d'exécution des tâches.

- **placement** (*mapping*) :
  supposons que l'on dispose d'un ensemble de machines pour traiter un ensemble de

tâches ou de travaux. Un placement est l'allocation d'une ou plusieurs machines à chaque tâche ou travail. Un placement est dit optimal s'il minimise une fonction de coût donnée. Généralement ces fonctions de coûts tiennent compte à la fois de l'équilibrage de la charge et de la minimisation des communications, critères qui sont contradictoires.

- **allocation** :
  il est souvent fait mention d'allocation dans ce travail. Dans la majorité des cas, on parle d'allocation de tâches à un processeur. Cette notion est rigoureusement équivalente à l'allocation des machines décrite dans la définition de l'ordonnancement, sans souci de dates.

- **équilibrage de charge** (*load-balancing*) :
  étant donnée une charge de travail, l'équilibrage de la charge consiste à répartir les tâches entre les machines de sorte que la charge de toutes les machines soit égale. Ce problème, dès que les tâches indépendantes et de durées différentes est NP-complet et est connu sous le nom de problème de *bin packing*.

- **regroupement** (*clustering*) :
  le regroupement de tâches est une stratégie utilisée dans certains problèmes d'ordonnancements. Le but est de regrouper des tâches ensemble de manière à économiser le coût des communications entre elles. Ce coût éliminé correspond à l'hypothèse de la localité des données qui est présentée et discutée dans la première partie.

- **partitionnement** (*partitioning*) :
  soit un programme, et sa représentation modélisée par un graphe dont les sommets sont les instructions. Le partitionnement correspond à la définition de tâches formées d'un ensemble d'instructions (leur taille est guidée par le choix de la granularité), et permet de produire un nouveau graphe dans lequel les sommets sont les tâches.

- **diagramme de Gantt** :
  il représente un ordonnancement. Il se présente sous la forme d'un graphique, en abscisse est indiqué le temps et en ordonnée les processeurs. Dans la représentation utilisée dans ce document, les parties du diagramme qui apparaissent en grisées correspondent à des temps d'inactivité et les parties plus sombres à des temps d'attente de communications.

- **périodes d'inactivité** (*idle time*) :
  il s'agit d'un laps de temps durant lequel un processeur n'effectue aucun calcul. Ces périodes sont appelées indistinctement périodes d'inactivité ou **d'oisiveté**. Dans la représentation des ordonnancements, il est fait une distinction entre une période d'inactivité pour cause d'attente d'une communication (la période d'attente est noircie), ou par manque d'activité (la période est grisée).

- **le problème du remplissage de boites *bin packing problem*** :
  Il s'agit de remplir un certain nombre de boites (*bins*) de manière à ce qu'elles soient remplies à peu près de la même façon. Si l'on rapporte ce problème à un système multiprocesseurs, il s'agit de répartir un ensemble de tâches indépendantes de taille

différentes sur un ensemble de processeurs de façon à ce que le maximum des dates de fin d'exécution des tâches soit minimum (cf *makespan*).

- **tâches multiprocesseurs** (*multiprocessor tasks*) :
  il s'agit de tâches nécéssitant plus d'un processeur ou plus d'une machine pour leur exécution. La majeure partie de ces problèmes proviennent des ateliers flexibles et de la gestion de production.

- **maximum des dates de fin d'exécution** (*makespan* ou *maximum completion time*) :
  dans le cadre d'une horloge globale pour un système multiprocesseur, il s'agit de la plus grande date de fin d'exécution des tâches. C'est un critère d'optimalité très populaire car il représente le temps parallèle d'une exécution déterministe d'un ensemble de tâches sur un ensemble de machines. Dans la littérature, on le retrouve sous la notation C_{max}.

- **diffusion** (*broadcast*) :
  La diffusion est l'opération par laquelle une machine envoie un message à l'ensemble des autres machines d'un réseau, en utilisant les lignes de communication de ce réseau [Rum94].

Chapitre 2

Annexe environnements de programmation

2.1 HyperTool

2.1.1 Objectifs de l'outil

En partant de l'idée que la parallélisation est un problème complexe qui ne peut être entièrement résolu que par un expert humain, Hypertool est présenté par ses concepteurs comme un outil d'*aide* au développement de programmes parallèles. La présence et la participation active de l'utilisateur est clairement requise [WG90].

2.1.2 Contexte d'utilisation

Cet outil a été conçu pour des systèmes à échange de messages et pour une programmation de type SPMD (le même programme exécuté par l'ensemble des processeurs, mais sur des données différentes). Les processeurs de l'architecture cible sont supposés être identiques.

2.1.3 Terminologie

Un segment de programme qui n'est pas partitionné est appelé un *processus*. Lorsque plusieurs processus sont regroupés, on parle de *tâches*. Le nombre d'opérations dans chaque tâche définie la *granularité d'opérations* et la taille des données la *granularité de données*.

2.1.4 Description de l'outil

Hypertool prend en entrée le programme partitionné en un ensemble de procédures. L'outil détecte alors les dépendances de données entre les différents processus (ces dépendances de données sont définies par l'affectation des paramètres) et forme un *macro dataflow graph* (un graphe orienté avec un nœud départ et un nœud fin) dans lequel les nœuds représentent les processus et les arcs les échanges de données. Le temps d'exécution d'un nœud est supposé connu au moment de la compilation (une version dynamique est à l'étude), et le temps de transmission d'un message est estimé en utilisant le temps d'initialisation (*startup*), la

longueur du message et la bande passante du canal de communication (les phénomènes de contention ne sont pas pris en compte). Le graphe est ensuite récupéré par le module d'ordonnancement qui alloue les processus aux tâches. L'allocation des tâches aux processeurs est effectuée par le module de placement. A la fin de ces deux étapes, une synchronisation est effectuée, ce qui se traduit par l'adjonction de primitives de communication. Pendant et après exécution, les modules de mesures de performances et de qualité sont mis en œuvre.

Le fonctionnement général de l'outil est illustré figure 2.1.

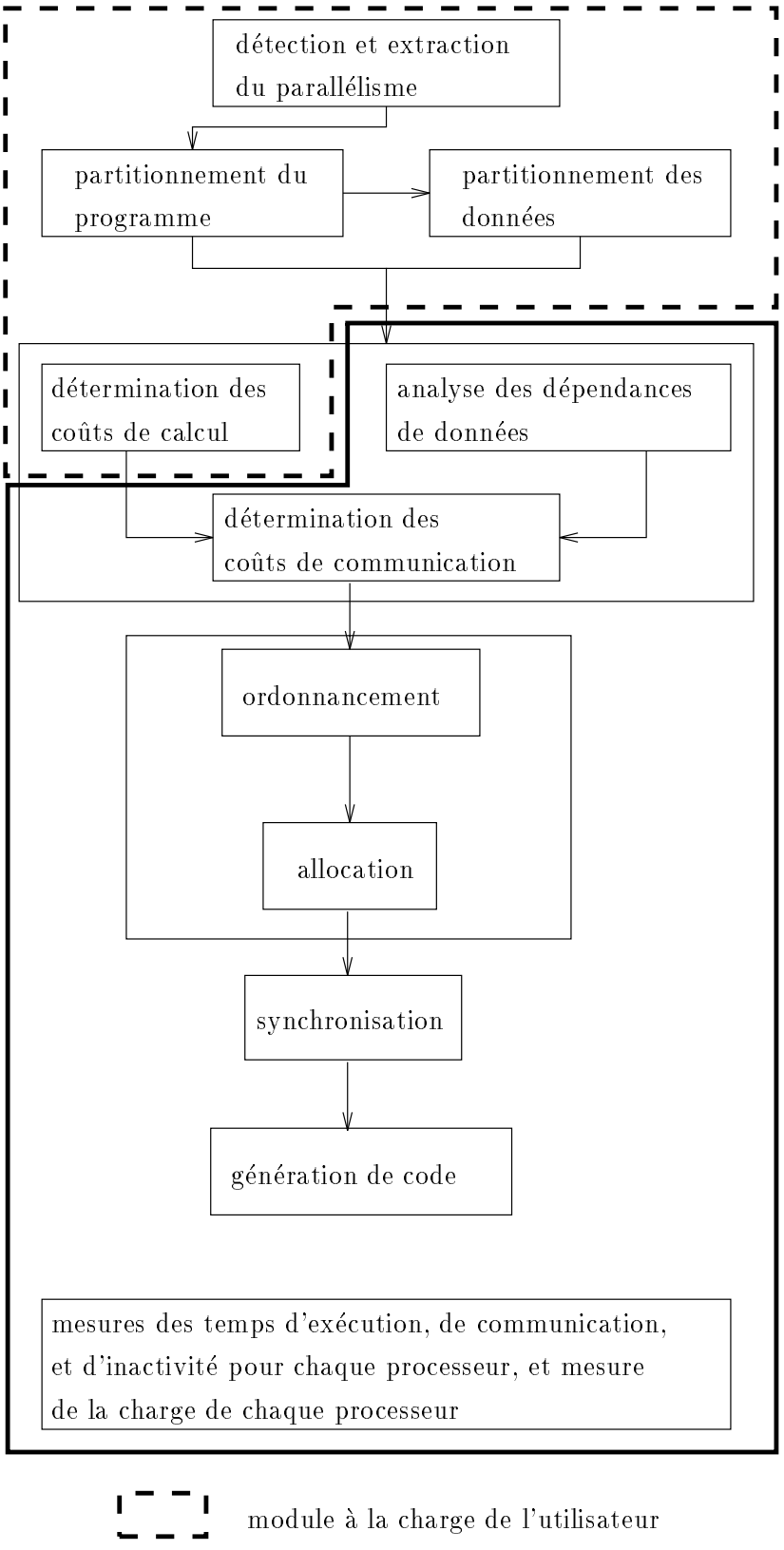


FIG. *2.1 - : Hypertool*

2.1.5 Intervention de l'utilisateur

Le partitionnement du programme au départ est à la charge de l'utilisateur, ce qui signifie que l'utilisateur définie a priori une granularité minimale puisque les processus qu'il a défini sont insécables. Il doit de plus donner une estimation de la durée nécessaire à l'exécution des processus. De même, c'est à lui d'interpréter les résultats rendus par le dernier module et d'agir en conséquence sur la stratégie de partitionnement ainsi que sur la taille de la granularité d'opérations (ce qui parait raisonnable).

2.1.6 L'étape d'ordonnancement et de placement

Le module d'ordonnancement est constitué de deux algorithmes, dans le cadre d'un ordonnancement statique et non préemptif.
Il s'agit d'un algorithme de type *Critical Path*, modifié (les auteurs l'on d'ailleurs baptisé *Modified Critical Path*) afin d'être adapté à un nombre limité de processeurs. Le problème de ce premier algorithme est qu'il ne détermine pas de façon satisfaisante le nombre minimum de processeurs requis pour exécuter un programme donné. Pour améliorer la détermination de ce nombre de processeurs, les auteurs proposent un second algorithme appelé MD (pour *Mobility-Directed*) qui génère des solutions pour le nombre de processeurs requis.
L'allocation utilise comme donnée d'entrée le graphe de tâches déduit du diagramme de Gantt et effectue une allocation de un pour un (une tâche pour un unique processeur et un processeur pour une unique tâche). La fonction de coût qui est à minimiser est la suivante :

$$F = \sum_{i=1}^{E_t} w(e_i) d_i$$

$w(e_i)$ représente la somme des temps de transmission de tous les messages échangés entre les deux tâches reliées par l'arête e_i.
d_i représente la distance entre les deux processeurs auxquels les deux tâches (liées par l'arête e_i) ont été allouées.

2.1.7 Utilisation pratique

- Machines concernées : iPSC/2, iPSC/860
- L'outil tourne sur : SPARC
- Langage utilisé : C
- Interface graphique : non
- Prix : gratuit
- Contact : wu@cs.buffalo.edu (Min-You Wu).

2.1.8 Bilan et conclusion

La démarche de cet outil est intéressante et paradoxale à la fois. Les concepteurs présentent Hypertool comme un outil *d'aide* au développement de programmes parallèles et de façon contradictoire *restreignent* l'accès au différents modules par l'utilisateur. Ainsi, il

ne lui est pas possible d'intervenir dans la politique de regroupement ou d'allocation, par conséquent, les mesures de performances ne permettent d'agir qu'en amont d'Hypertool, au niveau de la granularité et seulement à ce niveau là. Mis à part cet aspect de modularité (qui n'était pas le but de cet outil à l'origine), plusieurs idées sont intéressantes, par exemple, la conservation d'un code séquentiel pour permettre un premier débogage, la portabilité (le changement des modules de synchronisation et de génération de code permet d'adapter l'outil à une autre machine). De même, le choix de n'utiliser qu'un algorithme d'ordonnancement-allocation plutôt que plusieurs plus adaptés à certains types de graphes permet de garantir une compilation dont la durée ne dépend que de la longueur du code. A noter, cet outil semble peu adapté pour une utilisation de machines hétérogènes (comme par exemple réseau de stations). Cet outil est en quelque sorte une boite *clé en main* permettant de faire de la programmation parallèle. Il permet une utilisation simple et rapide d'une machine parallèle.

2.2 PYRROS

2.2.1 Objectifs de l'outil

Il s'agit d'un outil d'ordonnancement statique de graphes de tâches et de génération de code pour des architectures de type MIMD. Cet outil est pourvu d'un langage de graphes de tâches. Le but fixé est clairement une utilisation pratique [YG92a].

2.2.2 Contexte d'utilisation

- Architectures de type MIMD.
- Communications non synchronisées.
- Le recouvrement calcul/communication est autorisé.
- La duplication des tâches n'est pas autorisée.
- Les processeurs de l'architecture cible sont supposés être identiques.

2.2.3 Terminologie

Tâche : unité insécable de calcul. Il peut s'agir d'une instruction, d'une routine, ou d'un programme complet.
Ordonnancement : formé de deux fonctions principales, placer des tâches sur un ensemble de processeurs physiques et déterminer une date de début d'exécution pour chaque tâche.
Regroupement : il s'agit d'une agglomération de tâches en grains plus gros (*clusters*). Un grain (ou *cluster*) est un ensemble de tâches qui seront toutes exécutées sur un même processeur.

2.2.4 Description de l'outil

L'outil prend en entrée un graphe de tâches et produit des ordonnancements pour des machines à mémoire distribuée, du type nCUBE-II. Cet outil est en outre pourvu d'un langage de graphes de tâches qui permet de faire apparaître les définitions de tâches, ainsi

que les poids des tâches et des communications directement dans le code. Il possède aussi un outil graphique qui permet de visualiser les différents résultats intermédiaires et donc de revenir sur le partitionnement originel du programme. La génération de code se fait avec une optimisation des communications (un algorithme de diffusion est disponible) et une amélioration de l'utilisation de la mémoire. Un schéma synoptique de l'outil est présenté en

figure 2.2

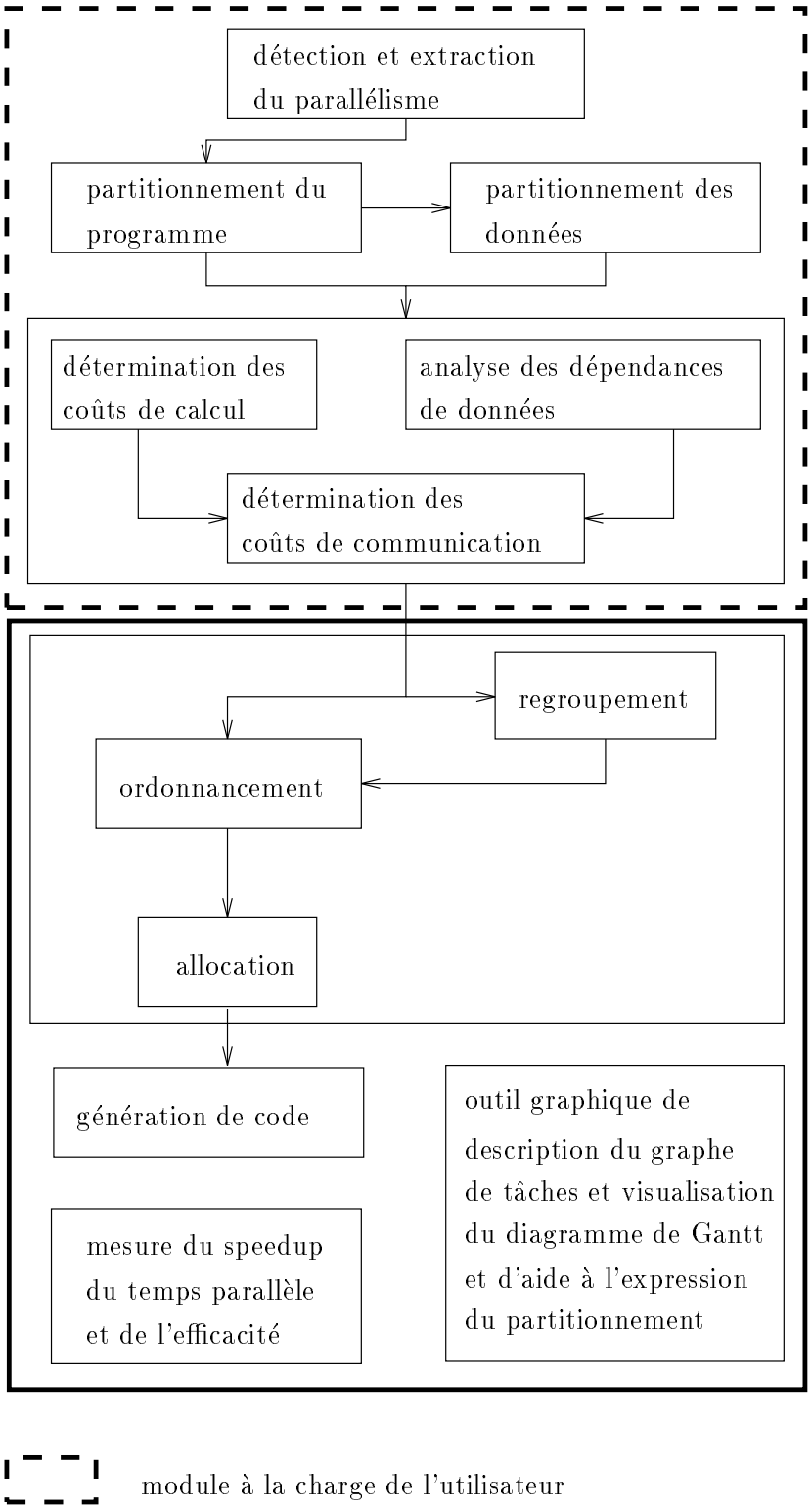


FIG. *2.2 - : Pyrros*

2.2.5 Intervention de l'utilisateur

Toutes les étapes jusqu'à l'obtention du graphe de tâches sont à la charge de l'utilisateur, ainsi, les étapes de détection et d'extraction du parallélisme, de partitionnement et d'analyse des dépendances de données doivent être produites par l'utilisateur. Cependant, les auteurs indiquent que l'outil peut être utilisé comme une aide pour déterminer un bon partitionnement avant l'exécution.

2.2.6 L'étape d'ordonnancement et de placement

L'étape d'ordonnancement est formée de deux étapes, une étape de regroupement de tâches sur un nombre illimité de processeurs qui forment une architecture virtuelle complètement connectée, suivie d'une étape d'ordonnancement des *clusters* sur un nombre p de processeurs correspondant à l'architecture cible. Les algorithmes d'ordonnancement proposés s'appliquent à des graphes de précédence sans cycle, et déterministes. Le déterminisme doit être réalisé soit au moment de la compilation soit au moment de l'exécution, c'est-à-dire, s'il manque des poids pour les tâches, on détermine le temps requis pour leur exécution après une exécution de chaque tâche. L'algorithme proposé pour Pyrros s'inspire d'un algorithme développé par Sarkar [Sar89].
Le module fonctionne de la façon suivante :
Supposons que le graphe de tâches soit complètement déterminé (si tel n'est pas le cas, il le sera à l'exécution), la première étape consiste à regrouper les tâches pour former des *clusters*. Au départ toutes les tâches constituent un *cluster*.
Recherche d'un plus long chemin depuis la source du graphe (il s'agit d'un *macro data flow graph*), jusqu'à la tâche terminale (appelé DS pour *Dominant Sequence*). Tous les arcs du chemin sont marqués *à examiner*.
Tant qu'il reste un arc du chemin qui n'a pas été examiné, mettre l'un des arc à zéro si le temps parallèle (PT) n'augmente pas. Marquer cet arc *examiné*. Déterminer un autre DS.
Cette première étape est suivie d'un regroupement des *clusters* en vue d'obtenir p *super-clusters* (il s'agit du nombre de processeurs de l'architecture réelle). Pour cela, un critère d'équilibrage de charge est pris en compte.
Ces *super-clusters* sont alors alloués de façon bijective aux processeurs de l'architecture réelle, puis l'outil applique une heuristique développée par Bokhari [Bok81].
A la fin de cette étape, le graphe de tâches est remanié afin de tenir compte des distances entre les processeurs, et des temps de communication qui sont devenus nuls. Ensuite, pour chaque processeur une liste de tâches avec priorité est maintenue à jour. Les tâches prêtes (dans le sens où elles ont reçu toutes les données nécessaires à leur exécution) sont ensuite ordonnancées par ordre de disponibilité des tâches et des processeurs sur lesquels elles sont exécutées.

2.2.7 Utilisation pratique

- Machines concernées : nCUBE-II, iPSC/860
- L'outil tourne sur : SUN
- Langage utilisé : C

- Interface graphique : X window en option
- Prix : gratuit
- Contact : gerasoulis@cs.rutgers.edu (Apostolos Gerasoulis).

2.2.8 Bilan et conclusion

Il est clair que les objectifs de cet outil ne sont pas très différents de ceux qui ont motivés les concepteurs d'Hypertool. Ici il s'agit d'un outil volontairement axé sur la partie ordonnancement du graphe de tâches, mais l'objectif est d'être un outil pratique, c'est-à-dire utilisable immédiatement et par tout utilisateur moyennant un minimum de formation. La façon dont le graphe a été obtenu n'est pas du ressort de l'outil. A l'actif de Pyrros on peut noter un bon algorithme d'ordonnancement, qui en temps linéaire permet d'obtenir un ordonnancement pour un grand nombre de graphes avec de bons résultats. Autres points positifs, les auteurs proposent une petite bibliothèque de routines pour les communications, ainsi qu'une boite à outils pour les optimisations d'utilisation de la mémoire.
Par contre on peut lui reprocher de ne proposer que de faibles mesures de performances (par rapport à Hypertool) ainsi que peu de possibilités d'actions à partir des résultats enregistrés à l'exécution. Les remarques qui ont été faites pour Hypertool concernant la modularité sont encore valables pour cet outil. Là encore, il n'est pas adaptable à des réseaux de machines hétérogènes.

2.3 TASKGRAPHER

2.3.1 Objectifs de l'outil

TaskGrapher est un outil qui est présenté comme un *outil pratique d'ordonnancement*. Il fait partie du projet PPSE (*Parallel Pragramming Support Environment*). Les objectifs de PPSE sont de permettre à un utilisateur de pouvoir faire de la programmation parallèle en utilisant une batterie d'outils tels que Reverse Engineering Tool, Parallax, TaskGrapher... Les objectifs qui sont à la base de la création de PPSE sont les suivants :
- admettre en entrée du code source séquentiel,
- utiliser un outil, *Reverse Engineering Tool* permettant de générer des fichiers compatibles avec le format Parallax, de sorte qu'ils puissent être visualisables,
- utiliser *Parallax* pour décrire l'application de manière à faire apparaître le parallélisme (par utilisation de constructions graphiques du type loop...)
- fournir un ordonnancement et un placement de bonne qualité par l'intermédiaire de *TaskGrapher*,
- générer du code pour machines parallèles à l'aide de *SuperGlue*,
- enfin, effectuer des mesures de performances avec *EDA* ("Execution Profile Analyser").
Dans ce qui suit, seul l'outil permettant de faire de l'ordonnancement est détaillé. TaskGrapher est présenté comme un *outil pratique d'ordonnancement*, dans ce sens il se rapproche de Pyrros plus que d'Hypertool. Par contre, au lieu de présenter une seule heuristique pour ordonnancer la plupart des graphes, les auteurs en proposent plusieurs (sept au total) [ERL90].

2.3.2 Description de l'outil

Bien que faisant partie intégrante de PPSE, cet outil peut être utilisé indépendamment de Parallax.
TaskGrapher produit des ordonnancements sous la forme de diagrammes de Gantt. Il produit des graphiques de performances, et une analyse du chemin critique. L'utilisateur peut décrire son programme sous la forme d'un graphe de tâches, choisir une ou plusieurs heuristiques d'ordonnancement ou d'allocation, choisir la topologie de la machine cible (descriptible "à la main"), et observer les estimations de performances à l'aide des outils graphiques. On peut par exemple avoir une estimation en pourcentages du temps total durant lequel chaque processeur a été actif. TaskGrapher possède un simulateur d'exécution, ce qui permet de suivre de façon "dynamique" l'exécution du programme et son comportement.

2.3.3 Intervention de l'utilisateur

L'utilisateur doit fournir le graphe en entrée, un graphe qui doit être totalement valué. Le choix de l'heuristique à appliquer est encore de son ressort. Au vu des résultats rendus par la simulation, il peut réexécuter le même programme mais en appliquant une autre des

sept heuristiques.

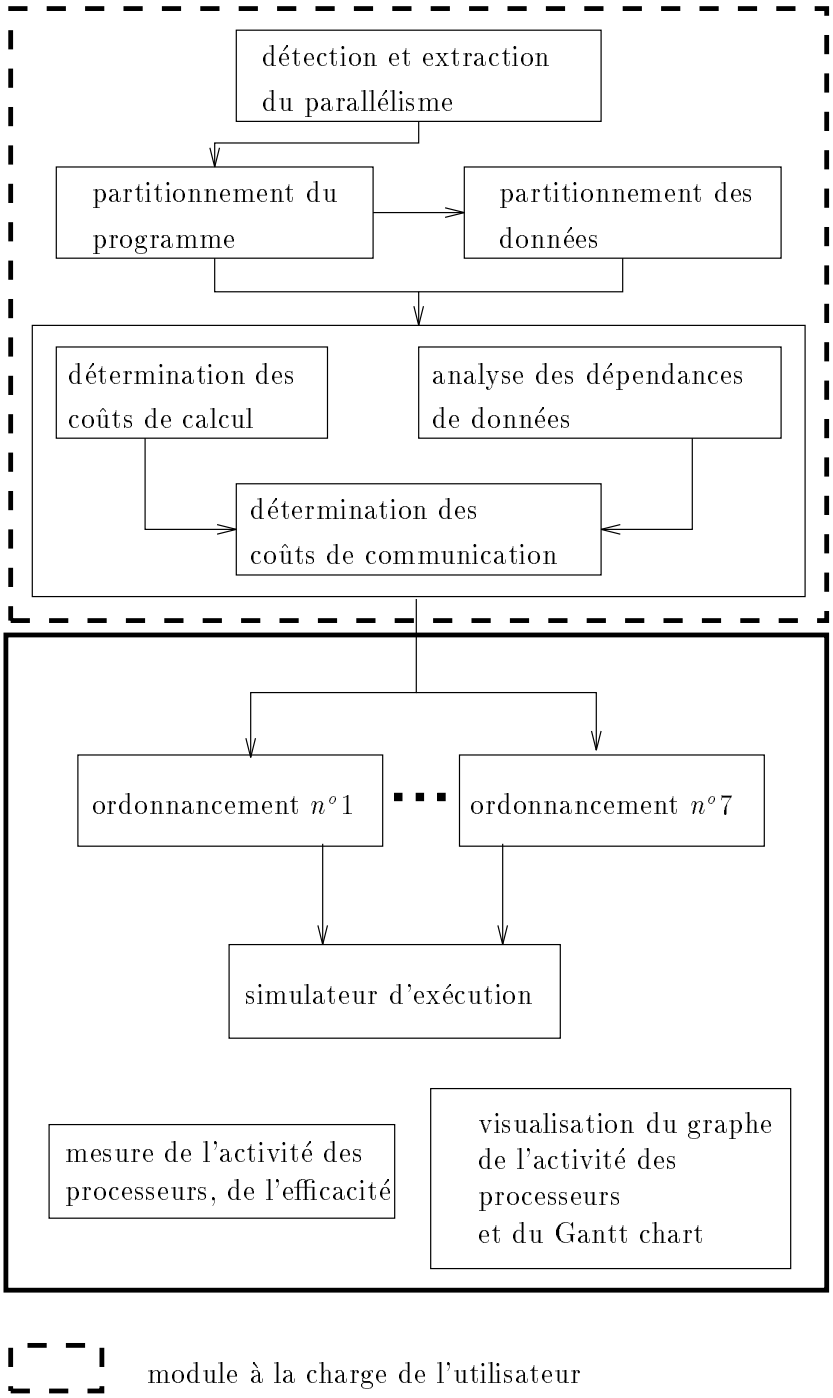


FIG. *2.3 - : Task Grapher*

2.3.4 L'étape d'ordonnancement et de placement

Pas moins de sept heuristiques sont proposées pour ordonnancer le graphe de départ.

- La première est due à Hu [Hu61]. Il s'agit du classique algorithme optimal pour des arbres UET sans communication, sur un nombre m fixé de machines identiques.

- La seconde consiste en une modification de l'algorithme précédent pour tenir compte des communications [Kru87].

- Toujours avec des communications, la troisième heuristique à été développée par Yu [Yu84], en prenant pour base l'algorithme de Hu, et en prenant en considération les communications au moment de l'allocation des tâches. Cette heuristique donne de bons résultats pour des temps de communications importants. A noter que les tâches ont toutes le même temps d'exécution.

- Les trois suivantes sont dues à Kruatrachue [Kru87]. L'une d'entres elles consiste à insérer dans les périodes d'inactivité des processeurs des tâches prêtes à être exécutées. La seconde repose sur l'idée de duplication de tâches de manière à réduire les temps de communications par élimination de certaines communications (notamment les nœuds qui ont plusieurs successeurs). La troisième porte encore sur l'idée de duplication, mais par une duplication totale des prédécesseurs des tâches qui sont déjà allouées. Cet algorithme est en $O(n^4)$.

- Enfin la dernière des heuristique due a à El-Rewini [ER90] qui est une heuristique d'allocation.

2.3.5 Utilisation pratique

- Machines concernées : aucune, simulation
- L'outil tourne sur : Macintosh
- Langage utilisé : Parallax, un graphe défini l'application
- Interface graphique : oui, pour dessiner le graphe de programme et l'architecture
- Prix : gratuit
- Contact : lewis@cs.nps.navy.mil (Ted Lewis).

2.3.6 Bilan et conclusion

Le positionnement de cet outil est différent de celui des deux outils précédents. Il apparaît plus spécifiquement comme un outil d'aide à l'ordonnancement que d'aide à la parallélisation d'une application, la raison en est évidente puisque TaskGrapher fait partie d'un ensemble d'outils permettant par une utilisation précise de chacun d'eux d'atteindre le but de la parallélisation d'une application. L'idée d'utiliser une "boite à outils" pour l'ordonnancement est séduisante, mais elle n'est pas pleinement exploitée. Puisque cet outil semble être fait pour donner une idée à l'utilisateur de la meilleure stratégie à utiliser pour ordonnancer un graphe, plusieurs petites commodités auraient été les bienvenues, comme par exemple :
- Un identificateur de graphes (détection d'une structure arborescente, d'un graphe où toutes

les tâches sont de durées égales, de graphes où les temps de communication sont nuls...)
- Une fonction d'aide à la décision dans le choix d'un ou plusieurs algorithmes en fonction de leur adéquation avec la structure du graphe.
- Une possibilité d'ajout de nouveaux algorithmes, afin de pouvoir mettre à jour cette "base de données" d'algorithmes d'ordonnancement.
Un autre bon point pour cet outil, il est possible de décrire la topologie de la machine, mais là encore, il aurait été intéressant de pouvoir étendre le nombre de paramètres de l'architecture cible (capacités des canaux de communication entre processeurs, capacité de la mémoire de chaque processeur, vitesse de chaque processeur...). Finalement, il s'agit d'un outil qui présente beaucoup de bonnes idées bien que toutes n'aient pas été exploitées à fond. On pourrait cependant le qualifier *d'hybride* en ce sens qu'il peut être à la fois utilisé dans un processus de parallélisation menant à une réelle génération de code, et à la fois comme un outil pour l'ordonnancement. Cet "double casquette" le dessert un peu puisque comme aide à l'ordonnancement les idées qui ont permis son développement n'ont pas été pleinement exploitées et comme élément d'un processus de parallélisation, les heuristiques proposées ne sont pas très généralistes, de complexité importante, et nécessitent l'intervention de l'utilisateur pour en choisir une.

2.4 HeNCE

Objectifs de l'outil

Il s'agit d'un outil d'aide à la programmation. Cet outil est couplé avec PVM, PVM est l'interface de bas niveau (programmation) et HeNCE est présenté comme l'interface de haut niveau. HeNCE a pour but de permettre à l'utilisateur de décrire graphiquement son application de manière à faire apparaître explicitement le parallélisme [BDG+93].

2.4.1 Contexte d'utilisation

- Machines hétérogènes (*Heterogeneous Network Computing Environment*)
- Langage utilisé : PVM

2.4.2 Description de l'outil

Cet outil est composé de plusieurs parties :

– La première partie permet à l'utilisateur de décrire son application parallèle par l'intermédiaire d'un langage graphique. L'utilisateur doit expliciter le parallélisme de son application. Les nœuds du graphe représentent des procédures Fortran ou C. En plus des nœuds simples (et des arcs qui matérialisent les dépendances de données), quatre autres types de nœuds sont disponibles afin de décrire l'application avec le maximum de précision. Le premier correspond à la structure d'une boucle. Le second type de nœud correspond à une dépendance conditionnelle. Dans ce cas là, l'expression est évaluée à l'exécution, et seulement l'une des branches est exécutée. Le troisième donne la possibilité à un nœud d'avoir une arité variable, l'expression permettant de calculer

cette arité étant évaluée à l'exécution. Enfin, le quatrième et dernier type de nœud correspond à une structure de pipeline.

- La seconde partie concerne la configuration du réseau. Cette configuration permet de donner des priorités dans le choix des machines pour l'exécution d'une tâche donnée. Ces priorités sont précisées via une matrice de coûts qui est directement utilisée à l'exécution par HeNCE.

- Avant de pouvoir exécuter le programme, l'outil produit automatiquement les appels PVM pour les communications et les synchronisations. Il produit de même le code pour chaque procédure en fonction de l'architecture cible (qui n'est pas forcément la même pour chaque tâche puisque le réseau est formé de machines hétérogènes). Enfin, il place sur chaque nœud du réseau les procédures qui lui sont allouées.

- L'exécution est réglée par ce que les auteurs appellent *the execute tool*. Durant l'exécution les procédures sont allouées automatiquement en fonction de la matrice des coûts et du graphe produit par l'utilisateur. L'exécution est de type maître-esclave, un des processeurs régit l'ensemble de la distribution du travail. Pendant l'exécution le contexte est sauvegardé afin de permettre une reprise d'exécution en cas de problème.

- Enfin, après l'exécution, une trace est disponible afin de permettre une réexécution parallèle simulée. La visualisation de l'activité des processeurs ainsi que l'ordonnancement des procédures est possible. Les communications et les temps d'attente pour un processeur ou une tâche donnée sont visualisables et reconnaissables par l'utilisation de différentes couleurs. A noter, la visualisation de la trace pendant l'exécution est possible.

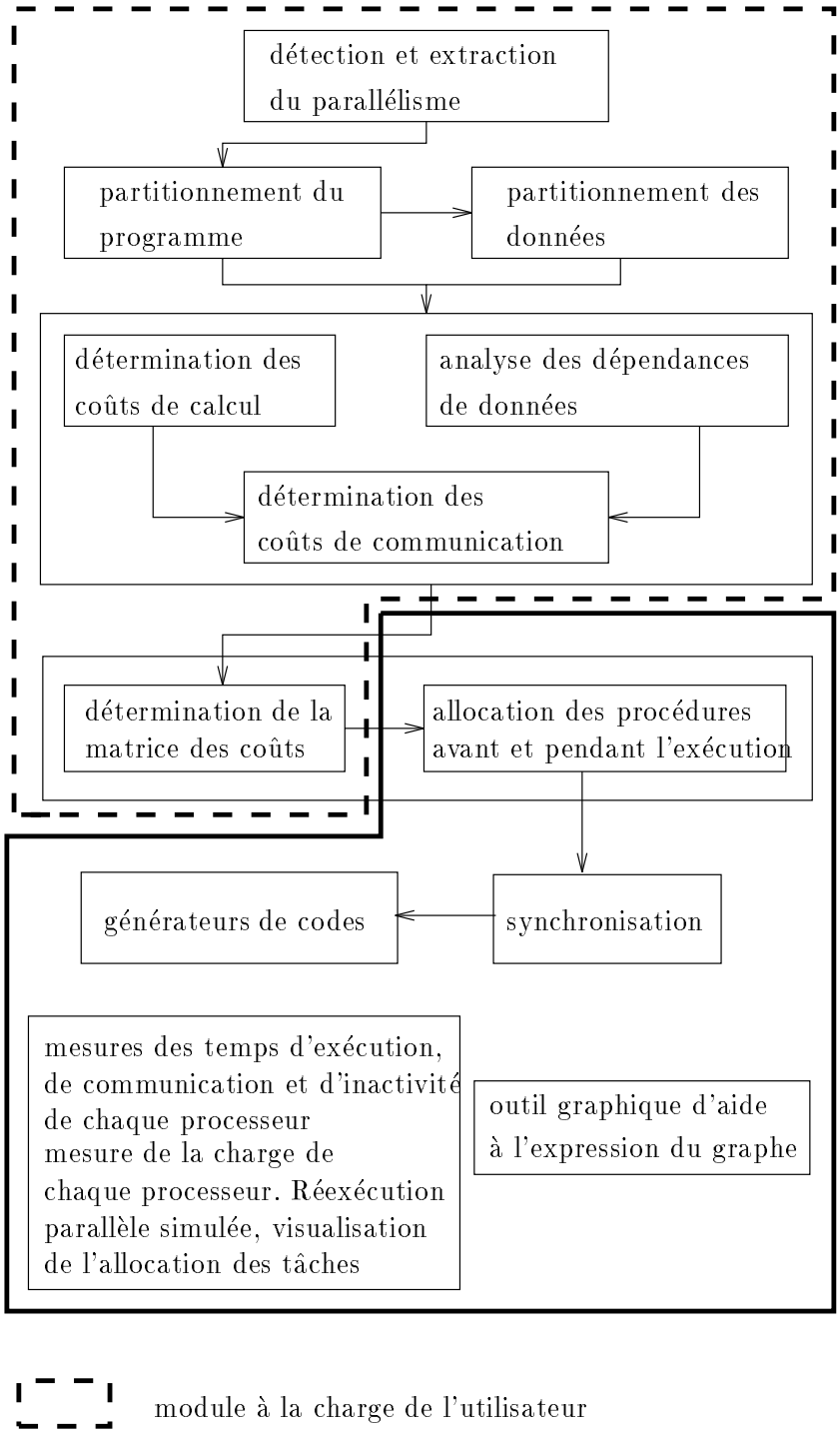


FIG. *2.4 - : HeNCE*

2.4.3 Intervention de l'utilisateur

Il intervient à différents niveaux, dessin du graphe au départ. Il agit aussi par l'affectation d'une priorité à chaque tâche par l'intermédiaire de la matrice des coûts.

2.4.4 L'étape d'ordonnancement et de placement

Cette étape intervient après la définition de l'application sous forme graphique et après la définition de la configuration du réseau. Chaque machine a une charge, pour exécuter une nouvelle routine r, le processeur maître choisie la machine m de telle sorte que le couple (charge de la machine m, coût de l'exécution de r sur m) soit le minimum de tous les m.

2.4.5 Utilisation pratique

- Machines concernées : Toute machine basée sur UNIX
- Langage utilisé : C, Fortran 77
- Interface graphique : X11R4 pour HeNCE
- Prix : gratuit
- Contact : pvm ou hence@msr.epm.ornl.gov

2.4.6 Bilan et conclusion

L'ensemble logiciel HeNCE-PVM constitue un tout cohérent. Concernant la partie allocation, il faut noter plusieurs choses. Premièrement, le coût de chaque procédure est indiqué par l'utilisateur et seulement par lui, ainsi que le choix préférentiel de la machine, cela suppose que l'utilisateur ait une assez bonne connaissance des différentes machines et architectures disponibles, et qu'il soit à même de pouvoir fournir une bonne estimation des coûts de ses procédures. Concernant la partie ordonnancement, il n'y en a pas à proprement parler, l'ordonnancement que l'on peut rencontrer dans HeNCE ressemble plutôt à de la répartition dynamique de charge. L'absence de cette étape s'explique assez facilement par le fait qu'il n'existe pas d'algorithmes efficaces pour ordonnancer sur m machines hétérogènes un ensemble de tâches de durées différentes liées par des contraintes de précédence avec des coûts de communication généraux. Les seuls algorithmes pouvant fournir une réponse pour ce type de problème étant du type recuit simulé ou encore tabou. Dans un premier temps, les seules améliorations qu'il pourrait être possible d'apporter à cet outil serait une partie équilibrage de charge dynamique un peu plus étoffée en amont (module d'estimation de la complexité des procédures par exemple) et en aval (utilisation de plusieurs fonctions de coûts tenant compte des architectures différentes).

2.5 ADAM

Concernant cet outil, certaines questions restent pour l'instant en suspens notamment concernant la valuation du graphe. Quels sont les critères qui permettent cette valuation, l'utilisateur intervient-il? Est-ce automatique? Si oui, comment cela fonctionne-t-il, première exécution? Évaluation de la complexité de chaque instruction? [ACC+93].

2.5.1 Objectifs de l'outil

Cet outil est dédié au développement d'applications pour machines massivement parallèles. Il a été développé dans le but de produire du code parallèle, donc à des fins d'utilisation

pratique.

2.5.2 Contexte d'utilisation

- Machines de type MIMD
- Les processeurs de l'architecture cible sont supposés être tous identiques.

2.5.3 Terminologie

Une *application* est définie comme un *ensemble de processus communicant* par échange de messages.

2.5.4 Description de l'outil

L'outil prend en entrée un programme écrit en langage EDAM (langage spécialement développé pour ADAM). Ce langage est du type CSP. EDAM permet de décrire une application comme un ensemble de processus communicant par échange de messages. Un programme EDAM est composé de deux parties, une partie concernant la description du code des processus, la seconde concernant la description algorithmique du graphe de communication inter-processus. Lorsque le programme est écrit, il est utilisé pour produire un *graphe de communications inter-processus*. Ce graphe est non orienté. Durant la phase de compilation et celle d'initialisation le graphe est *éventuellement* valué, à chaque nœud (processus) et à chaque arc (canal) est affecté un coût de calcul et un coût de communication (un coût de communication entre deux tâches correspond au temps de transmission de l'ensemble des messages échangés entre ces deux tâches).
Le graphe est récupéré par le module de placement. L'outil calcule ensuite le routage nécessaire aux communications, par minimisation d'une fonction de coûts.
Concernant le débogage, l'outil est pourvu d'un simulateur basé sur un noyau multitâche. Il

y a en outre la possibilité d'une réexécution simulée.

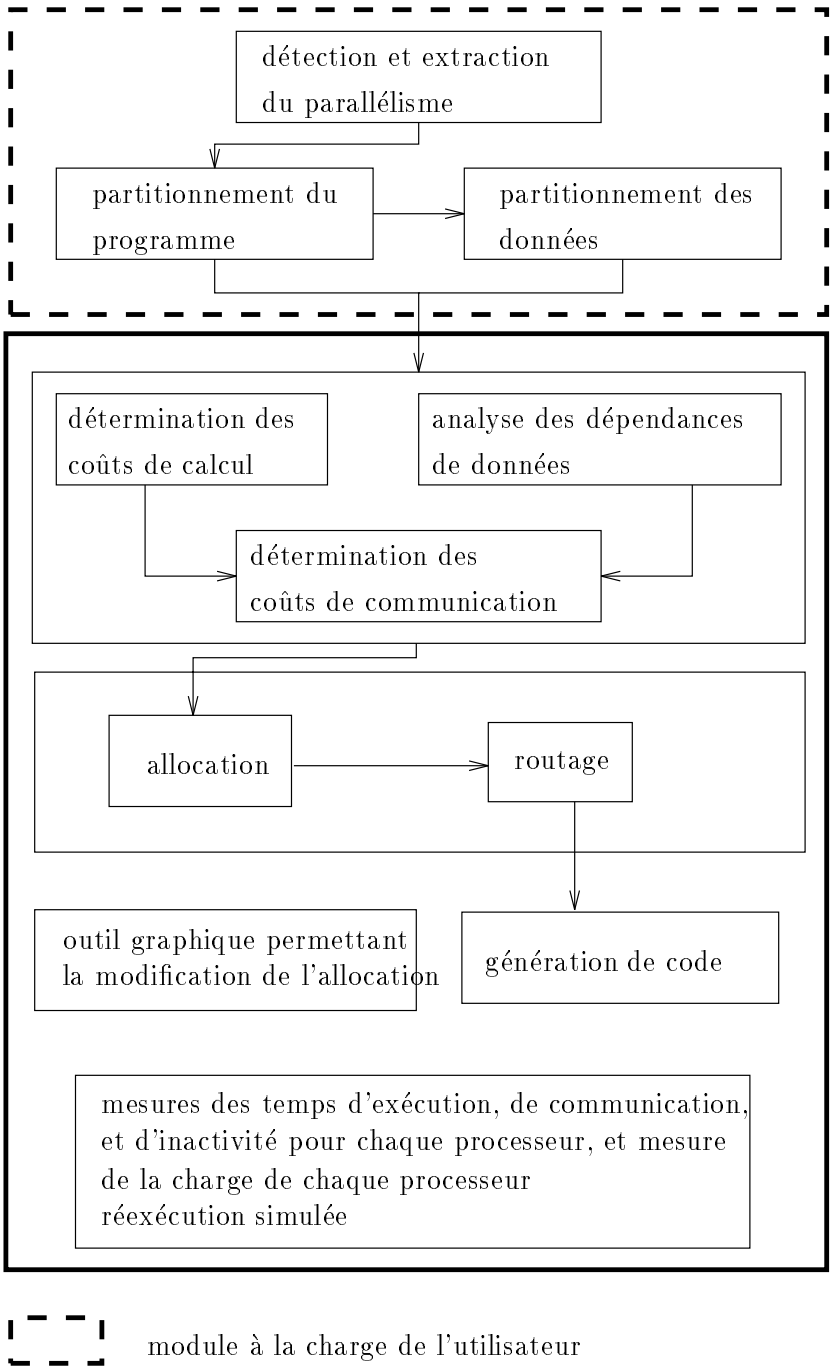


FIG. *2.5 - : Adam*

2.5.5 Intervention de l'utilisateur

L'utilisateur intervient dans l'écriture du programme en EDAM. Il peut intervenir sur les politiques d'allocation, en modifiant de façon manuelle l'allocation déterminée par le module

concerné.

2.5.6 L'étape d'ordonnancement et de placement

Le module concernant le placement des tâches sur les différents processeurs de l'architecture cible accepte en entrée un graphe de communications inter-processus. Ce graphe est non orienté. Le module peut être inactivé, le placement s'effectue alors à la main, ou activé, le placement est automatique, mais peut être modifié manuellement par l'utilisateur via une interface graphique. Les heuristiques de placement sont mises en œuvre à la compilation, il s'agit donc d'un placement statique. la méthode utilisée est constituée de deux parties :
- équilibre de la charge entre les processeurs
- minimisation des communications
Trois heuristiques sont disponibles :
- un algorithme glouton dont le principe est dû à Chen [CG88]
- l'algorithme de Bokhari [Bok81]
- un algorithme basé sur le précédent qui équilibre la charge en passant d'un nombre non limité de processeurs au nombre de processeurs disponibles de l'architecture cible, puis le même algorithme est utilisé pour minimiser les coûts de communication. La fonction de coût utilisée est de la forme :

$$F = \alpha F_1 + \beta F_2 + \gamma F_3$$

Où F_1 représente l'équilibrage de la charge, F_2 représente les communications internes (il s'agit des communications entre des tâches exécutées par le même processeur) et F_3 les communications externes.

2.5.7 Utilisation pratique

- Machines concernées : T-NODE
- Langage utilisé : EDAM
- Interface graphique : disponible pour les différents éléments de l'outil
- Prix : gratuit
- Contact : ...@labri.greco-prog.fr (Alabau, Chaumette, Counilh, Lépine, Roman, Rubi, Vauquelin).

2.5.8 Bilan et conclusion

Dans sa conception, cet outil se rapproche de HeNCE, il est basé sur un couplage avec un langage, EDAM, mais il ne fonctionne pas à partir de langages tels que C ou Fortran, puisqu'il est nécessaire de passer par EDAM pour avoir une description explicite du parallélisme. Une des grosses différences entre HeNCE et ADAM est que ce dernier outil est prévu pour des architectures disposant uniquement de processeurs identiques, ce qui facilite la mise en œuvre d'un module allocation. Concernant cet élément, ADAM est basé sur des algorithmes qui ne peuvent être utilisées que de manière statique. La granularité du graphe de communications

inter-processus est fixée implicitement par la syntaxe du langage EDAM. Cette granularité implique la génération et l'utilisation de graphes non orientés. On peut regretter l'absence d'allocation dynamique (par quelque moyen que ce soit) qui peut permettre dans certains cas de minimiser un mauvais placement de départ (les graphes en entrée du module d'allocation ne sont pas toujours complètement valués). Apparemment, cet outil n'est disponible pour l'instant qu'en version simulée.

2.6 PARALEX

2.6.1 Objectifs de l'outil

Il s'agit d'un environnement qui vise à fournir une aide pour toutes les étapes de la programmation. Un outil automatique pour la distribution, la tolérance aux pannes et l'hétérogénéité pour les applications parallèles et distribuées est proposé. Cet environnement offre en outre la possibilité de réutiliser du code séquentiel.

2.6.2 Contexte d'utilisation

- Réseaux de stations hétérogènes, à bande passante étroite et à latence élevée.

2.6.3 Terminologie

Un programme Paralex est modélisé par un **graphe orienté sans cycle** dans lequel un **nœud** correspond à une fragment de code séquentiel (qui accepte plusieurs paramètres en entrée et délivre un seul résultat).
Les *flots de données* entre ces nœuds sont représentés par des *liens*.
Il existe aussi des *nœuds filtres* qui ont pour but de sélectionner dans le résultat produit par un nœud N, la ou les parties de ce résultat nécessaires à l'exécution des nœuds liés à N.
Une *chaîne* est un ensemble de nœuds liés par une relation de précédence de type séquentiel.

2.6.4 Description de l'outil

Paralex est composé de quatre parties :

- Un éditeur graphique qui permet la représentation du graphe. Dans la description proposée par cet outil, les boucles et la récursivité ne sont pas autorisées pour exprimer un algorithme récursif, il faut déplier les appels successifs (!). La valeur des nœuds est indiquée par l'utilisateur lors de l'édition du graphe.

- Un compilateur qui appelle le compilateur C. Ce compilateur utilise une boite à outils ISIS pour standardiser la représentation des données, afin que les messages échangés soient compréhensibles par toutes les machines. De même, comme plusieurs architectures sont disponibles, pour un même nœud, plusieurs codes sont nécessaires.

- Un chargeur-débogueur.

– Un environnement d'exécution. Cet environnement comprend un contrôleur d'exécution. Dans ce contrôleur est implémenté un module d'équilibrage dynamique de la charge.

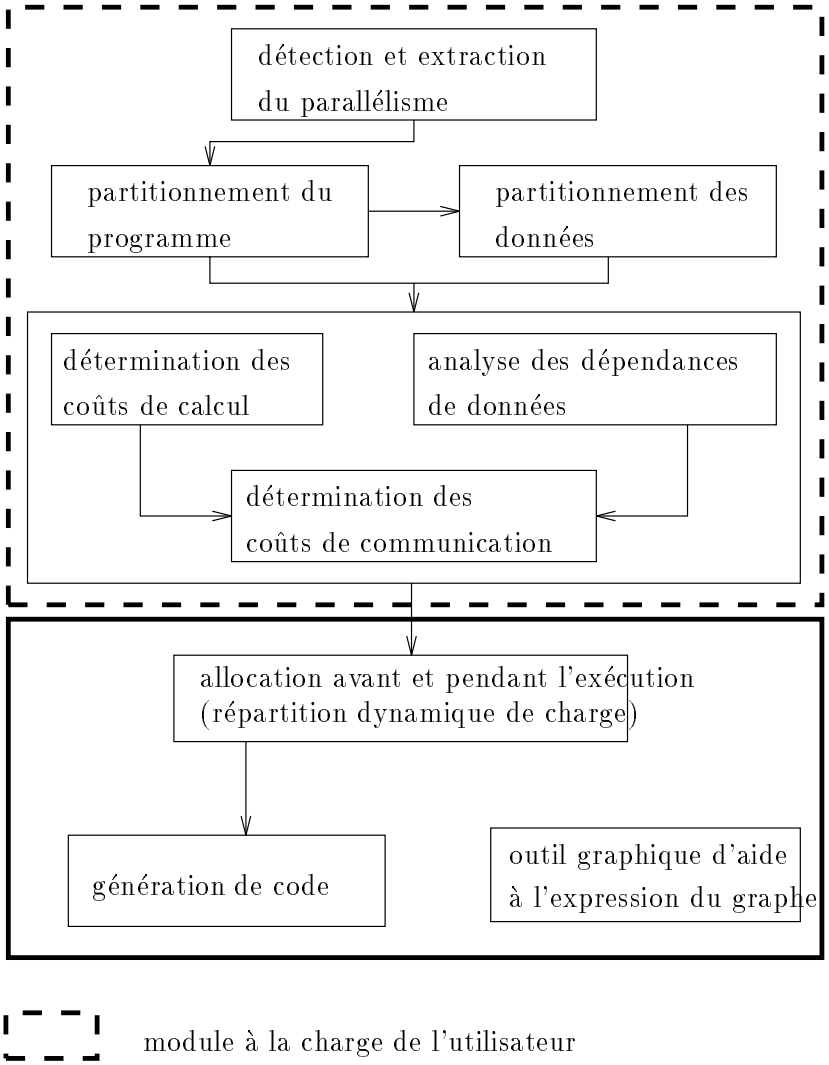


FIG. *2.6 - : Paralex*

2.6.5 Intervention de l'utilisateur

L'utilisateur décrit le graphe de programme à l'aide de l'interface graphique, effectue la valuation de chaque nœud de ce graphe. D'autre part, il décrit dans un fichier *paralex.site* l'ensemble des machines disponibles sur le réseau et pouvant être utilisées.

2.6.6 L'étape d'ordonnancement et de placement

Les graphes pris en compte sont orientés et sans cycle. Les heuristiques utilisées pour l'allocation supposent que l'architecture est formée de machines uniformes (ces machines ont

toutes les mêmes caractéristiques, seule leur vitesse est différente) et formant une topologie complètement connectée. C'est au moment de l'allocation que les arcs du graphe sont valués. Les valeurs des arcs dépendent du nombre d'octets qui vont circuler sur chaque lien. Dans un premier temps sont identifiées toutes les chaînes présentes dans le graphe. L'idée est alors d'effectuer de manière séquentielle les chaînes sur des processeurs en ayant pour objectifs de minimiser la somme des communications sur les liens. Pour cela, l'outil identifie le nombre maximum de chaînes qui peuvent être exécutées en parallèle.
L'algorithme utilisé est un algorithme de regroupement glouton. Lorsque le regroupement est effectué, les "grains" obtenus sont alors alloués aux machines. Cette allocation se fait de façon également gloutonne, l'heuristique utilisée allouant le plus gros grain à la machine ayant la *puissance résiduelle* la plus grande. La puissance résiduelle est définie de la manière suivante :

$$P_{residuelle} = \frac{P_m}{1 + charge_m}$$

P_m la puissance de la machine m est donnée en SPECmark, et $charge_m$ est la somme des longueurs de toutes les chaînes allouées à m jusqu'à présent.

2.6.7 Utilisation pratique

- Machines concernées : stations supportant M680X0, SPARC, PRISM, MIPS et Vax
- Langage utilisé : C
- Interface graphique : oui
- Prix : gratuit
- Contact : Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, Université de Bologne, Dept Maths.

2.6.8 Bilan et conclusion

Cet environnement ressemble beaucoup à ce qui a été fait dans HeNCE. Une des particularités de cet outil est qu'il accorde plus d'importance à la partie allocation que ne le fait HeNCE. Les algorithmes utilisés sont tous des gloutons, bien que cela puisse paraître décevant, il faut noter que sur les architectures hétérogènes, peu de choses existent concernant les stratégies d'ordonnancement et d'allocation (stratégies citées lors de la description de HeNCE), enfin, il manque un élément pour faire de l'analyse de performances. A noter, il s'agit d'une première version et les auteurs la qualifie de version test, ils justifient leur choix d'heuristique simple et de fonctionnalités "spartiates" par ce fait plus une éventuelle économie de temps, à vérifier par des expériences. Paralex : à suivre...

Bibliographie

[ABR90] R.J. Anderson, P. Beame, and W. Ruzzo. Low overhead for parallel schedules for task graphs. In *Proc. of Symposium of Parallel Algorithms and Architecture*, pages 66–75, 1990.

[ACC+93] M. Alabau, S. Chaumette, M. C. Counilh, J. M. Lépine, J. Roman, F. Rubi, and B. Vauquelin. *A Programming Environment Dedicated to a Model of Explicit Parallelism*, pages 193–212. J. Dongarra and B. Tourancheau, 1993.

[ACD74] T.L. Adams, K.M. Chandy, and J.R. Dickson. A comparison of list schedulers for parallel processing systems. *Communication of the ACM*, 17(12):685–690, December 1974.

[ACS90] A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of prams. *Theoretical Computer Science*, 71:3–28, 1990.

[Arr94] Y. Arrouye. Performance Evaluation of Parallel Systems: the Scope Extensible Interactive Environment. Technical Report 15, Laboratoire de Modélisation et de Calcul - IMAG, Grenoble, December 1994.

[Azé95] L. Azéma. *Evaluation des performances de stratégies d'ordonnancement.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble - France, June 1995. Mémoire de DEA.

[Bae74] J. L. Baer. *Optimal Scheduling on Two Processors of Different Speeds*, pages 27–40. North Holland, 1974.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS*, pages 307–314, 1968.

[Bau95] E. Baud. *Un outil de contruction de graphe pour l'environnement* ATHAPASCAN. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble - France, June 1995. Mémoire de DEA.

[BD93] J. Błażewicz and M. Drozdowski. The Performance Limits of a Two-Dimensional Network of Load-Sharing Processors. *Private Communication*, 1993.

[BDG+93] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. *PVM and HeNCE: Tools for Heterogeneous Network Computing*, pages 139–154. J. Dongarra and B. Tourancheau, 1993.

[BDSW90] J. Błażewicz, M. Drozdowski, G. Schmidt, and D. De Werra. Scheduling independent two processor tasks on a uniform duo-processor system. *Discrete Applied Mathematics*, (28):11–20, 1990.

[BESW93] J. Błażewicz, K. Ecker, G. Shmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag, 1993.

[BKT94] E. Bampis, J-C. König, and D. Trystram. Optimal parallel execution of complete binary trees and grids into most popular interconnection networks. In *PARLE'94, Athens, Greece*, 1994.

[Bok81] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30:207–214, March 1981.

[Bou94] P. Bouvry. *Placement de tâches sur ordinateurs parallèles à mémoire distribuée.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble - France, October 1994. in french.

[BOV85] I. Bar-On and U. Vishkin. Optimal parallel generation of a computation tree-form. *ACM Transactions on Programming Languages and Systems*, 7(22):348–357, April 1985.

[BR93] S. Bataineh and T. G. Robertazzi. Ultimate performance limits for networks of load sharing processors. *IEEE Transactions on Aerospace and Electronic Systems*, 1993.

[Cal95] C. Calvin. *Minimisation du sur-coût des communications dans la parallélisation des algorithmes numériques.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble - France, July 1995. in french.

[CC88] J. Carlier and P. Chrétienne. *Problèmes d'ordonnancement.* Masson, collection ERI, 1988.

[CD72] E.G. Coffman and P.J. Denning. *Operating System Theory*. Prentice Hall, 1972.

[CF91] M. Cosnard and A. Ferreira. Designing parallel non numerical algorithms. In *Proc. of Parallel Computing*, pages 3–18, 1991.

[CG72] E.G. Coffman and R.L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[CG88] W. K. Chen and A. F. Gehringer. A Graph-Oriented Mapping Strategy for a Hypercube. In *Proc. of International Conference on Hypercube Concurrent Computers and Applications*, volume 1, pages 200–209, 1988.

[Che93] D. Y. Cheng. A Survey of Parallel Programming Languages and Tools. Technical Report RND-93-005, NASA Ames Research Center, March 1993.

[Chr89] P. Chrétienne. A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *European Journal of Operations Research*, (43):225–230, 1989.

[Chr94] M. Christaller. Athapascan-0a sur PVM 3 définition et mode d'emploi. Technical Report 11, Laboratoire de Modélisation et de Calcul - IMAG, Grenoble, June 1994.

[CL88] G. I. Chen and T. H. Lai. Premptive scheduling of independent jobs on a hypercube. *Information Processing Letters*, pages 201–206, 1988.

[Col89] J.-Y. Colin. *Problèmes d'Ordonnancement avec Délais de Communication : Complexité et Algorithmes*. PhD thesis, University of Paris VI, Paris, November 1989.

[CP91] P. Chrétienne and C. Picouleau. The basic scheduling with interprocessor communication delays. Technical Report 91.6, MASI, Paris, June 1991.

[CR90] Y. C. Cheng and T. G. Robertazzi. Distributed computation for a tree-network with communication delays. *IEEE Transactions on Aerospace and Electronic Systems*, 26(3):511–516, May 1990.

[CS80] Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, (9):91–103, 1980.

[CZ89] R. Cole and O. Zajicek. The apram: Incorporating asynchrony into the pram model. In *Proc. of STOC*, 1989.

[CZ90] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proc. of STOC*, 1990.

[DGL89] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix problems. *ACM Transactions on Mathematical Software*, 14:1–14, 1989.

[DL88] J. Du and J.Y-T. Leung. Scheduling tree-structured tasks with restricted execution times. *Information Processing Letters*, (28):183–188, July 1988.

[DL89] J. Du and J.Y-T. Leung. Scheduling tree-structured tasks on two processors to minimize schedule length. *SIAM Journal on Discrete Mathematics*, 2(2):176–196, May 1989.

[Dob84] G. Dobson. Scheduling independent tasks on uniform processors. *SIAM Journal on Computing*, 13(4):705–716, November 1984.

[DR93] A. Darte and Y. Robert. Mapping Uniform Loop Nests Onto Distributed Memory Architectures. Technical Report 93-03, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, January 1993.

[Duf95] I. Duff. Solution de problèmes creux par méthodes directes. In *Séminaire sur les techniques nouvelles de traitement des matrices creuses pour les problèmes industriels, Février 1995, Grenoble*, 1995.

[ER90] H. El-Rewini. *Task Partitioning and Scheduling on Arbitrary Parallel Processing Systems.* PhD thesis, Department of Computer Science, Oregon State University, 1990.

[ERL90] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, (9):138–153, 1990.

[FdK94] A. Fagot and J. Chassin de Kergommeaux. Optimized record-replay for RPC-based parallel programming. In *Working conference on programming environments for massively parallel distributed systems*, Ascona, Switzerland, April 1994. IFIP, WG10.3, Birkhäuser, Basel.

[Fea91] P. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–51, 1991.

[Fea92] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. One Dimensional Time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[FL83] D. K. Friesen and M. A. Langston. Bounds for multifit scheduling on uniform processors. *SIAM Journal on Computing*, 12(1):155–166, February 1983.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. of STOC*, pages 114–118, 1978.

[Gab88] H. N. Gabow. Scheduling uet systems on two uniform processors and lenght two pipelines. *SIAM Journal on Computing*, 17(4):810–829, August 1988.

[GIS77] T. Gonzalez, O. H. Ibarra, and S. Sahni. Bounds for lpt schedules on uniform processors. *SIAM Journal on Computing*, 6(1):155–166, March 1977.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness.* W.H. Freeman, New York, 1979.

[GJTY83] M.R. Garey, D.S. Johnson, R.E. Tarjan, and M. Yannakakis. Scheduling opposing forests. *SIAM Journal on Algebraic and Discrete Methods*, 4(1):72–93, March 1983.

[GJY94] A. Gerasoulis, J. Jiao, and T. Yang. Scheduling Structured and Unstructured Computation. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, May 1994.

[GLLK79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[Gra69] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[Gre87] L. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* PhD thesis, Yale University, 1987.

[GRT95] F. Guinand, C. Rapine, and D. Trystram. Worst-Case Analysis of Algorithms for Scheduling UECT Trees on m Processors. *Manuscrit*, 1995.

[GT93] F. Guinand and D. Trystram. Optimal scheduling of uect trees on two processors. Technical Report Rapport APACHE $n^o 3$, Laboratoire de Modélisation et de Calcul - IMAG, Grenoble, November 1993.

[Har94] T. J. Harris. A survey of pram simulation techniques. *ACM Computing Surveys*, 26(2):187–206, June 1994.

[HS76] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, 23(2):317–327, April 1976.

[HS88] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors using the dual approximation approach. *SIAM Journal on Computing*, 17(3):538–551, June 1988.

[Hu61] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[JKS89] H. Jung, L. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays. In *Proc. of SPAA*, 1989.

[JR92] A. Jakoby and R. Reischuk. *The Complexity of Scheduling Problems with Communication Delays for Trees*, volume 621 of *Lecture Notes in Computer Science*, pages 165–177. Springer Verlag, 1992.

[Kit94] J. P. Kitajima. *Modèles Quantitatifs d'Algorithmes Parallèles.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble - France, October 1994.

[KL88] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Transactions on Software Engineering*, pages 23–32, 1988.

[Kru87] B. Kruatrachue. *Static Task Scheduling and Grain Packing in Parallel Processing Systems.* PhD thesis, Department of Computer Science, Oregon State University, 1987.

[Law93] E. L. Lawler. Scheduling trees on multiprocessors with unit communication delays. *Workshop on Models and Algorithms for Planning and Scheduling Porblems, Villa Vigoni, Lake Como, Italy, June 14-18*, 1993.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[LL74a] J. W. S. Liu and C. L. Liu. Bounds on scheduling algorithms for heterogeneous computing systems. In *Proc. of IFIPS '74*, pages 349–353, 1974.

[LL74b] J. W. S. Liu and C. L. Liu. *Performance analysis of heterogeneous multiprocessor computing systems*, pages 331–343. North Holland, 1974.

[LLKS89] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and scheduling: algorithms and complexity. Technical Report BS-R8909, Centre for mathematics and computer science, Amsterdam, June 1989.

[LS93] E. Leu and A. Schiper. *ParaRex: a Programming Environment Integrating Execution Replay and Visualization*, pages 155–170. J. Dongarra and B. Tourancheau, 1993.

[Mai95a] Eric Maillet. Compensating for perturbations of parallel applications induced by software tracing. *submitted to Software Practice and Experience*, 1995.

[Mai95b] Eric Maillet. TAPE/PVM an efficient performance monitor for PVM applications – User Guide. Technical report, IMAG institute, Grenoble, 1995. avaiable by anonymous ftp from imag.fr.

[Mic94] P. Michallon. *Schémas de communications globales dans les réseaux de processeurs : application à la grille torique.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble - France, February 1994. in french.

[MR94] C. Martel and A. Raghunathan. Asynchronous pram with memory latency. *Journal of Parallel and Distributed Computing*, (23):10–26, 1994.

[MT95] Eric Maillet and Cécile Tron. On Effciently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *To appear in Journal of Parallel and Distributed Computing*, 1995.

[NLH81] K. Nakajima, J. Y.-T. Leung, and S. L. Hakimi. Optimal two processor scheduling of tree precedence constrained tasks with two execution times. *Performance Evaluation*, 1:320–330, 1981.

[NM93] L. M. Ni and P. K. McKinley. A survey of wormhole routing techinques in direct networks. *Computer*, (2):62–76, 1993.

[Pic93] C. Picouleau. *Etude des Problèmes d'Optimisation dans les Systèmes Distribués.* PhD thesis, Paris VI, Paris - France, 1993. in french.

[Pic94] C. Picouleau. Uet-uct schedules on arbitrary networks. Technical Report LITP-94.09, Laboratoire Informatique Théorique et Programmation, Paris, January 1994.

[Pla94] B. Plateau. Présentation d'apache. Technical Report 1, (LGI,LMC), Grenoble, November 1994.

[Poz92] R. Pozo. Performance modeling of sparse matrix methods for distributed memory architectures. In *Proc. of CONPAR 92 - VAPP V*, 1992.

[PS93] J.G. Peters and M. Syska. Circuit-Switched Broadcasting in Torus Networks. Technical report, Simon Fraser University, 1993.

[PU87] C.H. Papadimitriou and J. Ullman. A communication time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987.

[PY88] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proc. of 20th ACM Symposium on theory of computing*, 1988.

[Rev94] N. Revol. *Complexité de l'évaluation parallèle de circuits arithmétiques.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble - France, August 1994. in french.

[RS87] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[Rum94] Jean De Rumeur. *Communications dans les Réseaux de Processeurs.* Masson, 1994.

[Saa92] R. Saad. Scheduling with communication delays. Technical Report 754, LRI, University of Paris Sud, Paris - France, 1992.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, 1989.

[Sys92] M. Syska. *Communications dans les architectures à mémoire distribuée.* PhD thesis, Université de Nice - Sophia Antipolis - Laboratoire I3S, December 1992.

[TW91] A. Trew and G. Wilson. *Past, Present, Parallel: a survey of available parallel computing systems.* Springer-Verlag, 1991.

[Vel93a] M. Veldhorst. A linear time algorithm to schedule trees with communication delays optimally on two machines. Technical Report RUU-CS-93-04, Department of computer science, Utrecht, January 1993.

[Vel93b] B. Veltman. *Multiprocessor scheduling with communication delays.* PhD thesis, University of Technology of Eindhoven, Department of Operations Research, Amsterdam, May 1993.

[VRK93] T. A. Varvarigou, V. P. Roychowdhury, and T. Kailath. Scheduling in and out forests in the presence of communication delays. *Manuscript*, 1993.

[VRKL94] T. A. Varvarigou, V. P. Roychowdhury, T. Kailath, and E. Lawler. Scheduling in and out forests in the presence of communication delays. *To appear in IEEE Trans. on Parallel and Distributed Syst.*, 1994.

[WG90] M.-Y. Wu and D. D. Gajski. Hypertool: a programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.

[Wyl79] J. C. Wyllie. *The complexity of parallel computation.* PhD thesis, Cornell University, August 1979.

[YG92a] T. Yang and A. Gerasoulis. A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors. In *Proc. of SHPCC '92*, pages 350–357, 1992.

[YG92b] T. Yang and A. Gerasoulis. Comparison of Clusering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *Journal of Parallel and Distributed Computing*, (16):276–291, 1992.

[YG93a] T. Yang and A. Gerasoulis. List Scheduling With and Without Communication Delays. *Parallel Computing Journal*, (19):1321–1344, 1993.

[YG93b] T. Yang and A. Gerasoulis. On the Granularity and Clustering of Directed Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[Yu84] W. H. Yu. *LU Decomposition on a Multiprocessing System with Communication Delay.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1984.