UNIVERSITÉ DE MONTRÉAL

RÉPARTITION AUTOMATIQUE DES TÂCHES PARALLÈLES : APPLICATION DANS LA SIMULATION DE RÉSEAUX ÉLECTRIQUES EN TEMPS RÉEL

TONY WONG
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIAE DOCTOR (Ph.D.)
(GÉNIE ÉLECTRIQUE)
MAI 1999

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

RÉPARTITION AUTOMATIQUE DES TÂCHES PARALLÈLES : APPLICATION DANS LA SIMULATION DE RÉSEAUX ÉLECTRIQUES EN TEMPS RÉEL

présentée par: WONG Tony
en vue de l'obtention du diplôme de: Philosophiae Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président
M. HOULE Jean-Louis, Ph.D., membre et directeur de recherche
M. DESSAINT L.-A., Ph.D., membre et codirecteur de recherche
M. MARCEAU Richard, Ph.D. membre
M. MAHSEREDJIAN Jean, Ph.D. membre

DÉDICACE

À ma compagne de vie qui m'a épaulé durant toutes ces années d'études. Son amour sans bornes m'a aidé à traverser les moments difficiles. Ses encouragements opportuns m'ont donné la confiance pour réussir. Laura, ma tendre épouse, cette thèse vous est dédiée.

REMERCIEMENTS

Je tiens à remercier mon directeur de recherche, le professeur Jean-Louis Houle et mon codirecteur de recherche, le professeur Louis Dessaint pour la confiance et le soutien qu'ils m'ont témoignés durant toute la durée de cette recherche. Les nombreux conseils qu'ils m'ont prodigués et leur gentillesse seront toujours dans mon cœur.

J'aimerais également remercier le président du jury, le professeur Guy Bois pour son travail admirable et son amabilité malgré un horaire fort chargé. De même, pour les membres du jury qui ont accepté de participer à la présentation de ce projet de recherche. Je vous remercie infiniment.

Je remercie toute ma famille et particulièrement mon neveu Frédérick et ma nièce Gloria. Malgré leur jeune âge, ils ont su poser des questions dont certaines réponses ont trouvé leur place dans ce travail.

Enfin, je tiens à exprimer ma reconnaissance envers l'Institut de Recherche d'Hydro-Québec pour son soutien financier et envers les chercheurs du service simulation de l'IREQ qui, par leurs conseils techniques, ont contribué à faciliter la réalisation des travaux de cette recherche.

RÉSUMÉ

Cette thèse présente une méthode heuristique efficace pour la répartition des tâches parallèles dans la simulation en temps réel des réseaux électriques. Dans cette répartition des tâches, les contraintes temporelles sont définies par le temps d'exécution et par les échéanciers rigides des tâches. Tandis que les contraintes spatiales sont définies par la disposition des convertisseurs AN/NA dans certains processeurs et par les interconnexions du réseau des processeurs de l'ordinateur parallèle.

La problématique générale de la répartition des tâches est un problème fort difficile à résoudre. La nature combinatoire de ce problème élimine, à toute fin pratique, la possibilité d'une solution par une méthode d'énumération exhaustive. Les méthodes exactes et approximatives visent à obtenir des solutions optimales mais ne s'appliquent qu'à des cas particuliers du problème. C'est pour ces raisons que nous proposons une méthode heuristique qui est en mesure de donner une répartition adéquate des tâches parallèles pour la simulation en temps réel des réseaux électriques.

Cette méthode heuristique est développée en considérant le problème de répartition des tâches comme un problème de fouille dans un espace d'états. Un cadre formel peut être établi par la définition d'une métrique générale de l'espace d'états. Cette métrique généralisée est la longueur d'un chemin exprimé à l'aide de monoïdes des ensembles. Les propriétés intéressantes des méthodes de fouille heuristique sont alors facilement dégagées. Ce cadre formel est ensuite utilisé pour le développement d'une nouvelle méthode de fouille heuristique à mémoire limitée. L'utilisation constante de la mémoire est un facteur important dans la mise en œuvre pratique du répartiteur de tâches.

Un ensemble d'heuristiques de prétraitement sont utilisées pour traiter les systèmes de tâches contenant des relations de préséance. Ces tâches particulières sont issues de la décomposition explicite des programmes simulant le comportement des composants électriques. Le répartiteur proposé est donc capable d'ordonnancer les tâches avec relations de préséance et d'assigner les tâches sans ordre d'exécution dans les processeurs de l'ordinateur parallèle.

ABSTRACT

This thesis presents an efficient heuristic method for the mapping of parallel tasks in real-time simulation of electrical networks. In this mapping problem, the temporal constraints are defined by the execution time and by the deadlines of the task system. While the spatial constraints are defined by the availability of AD/DA converters and processor network connections of the parallel computer.

The general tasks mapping problem is very hard to solve. Its combinatorial nature inhibits any practical solutions by means of exhaustive enumeration. The existing exact and approximate mapping methods are geared toward obtaining optimal solutions. But they can only be applied to particular and restricted problems. For these reasons, we propose a heuristic method, which enables the mapping of parallel tasks for real-time simulation of electrical networks.

In this research, it is possible to transform the mapping problem into a space-state search problem. A formal framework can then be constructed by defining a generalized metric in the space-state. This generalized metric is the concept of path length as represented by a set monoid. Appealing properties of space-state search techniques can be obtained using this generalized metric. This formal framework was also used in the formulation of an efficient limited-memory heuristic search method. The so-called constant-memory behavior is an important characteristic, which enables the proposed method to solve practical mapping problems.

An effective heuristic preprocessing module is also included in the proposed method. This preprocessing module allows the scheduling of sub-tasks where partial ordering is defined. These precedence-constrained sub-tasks are used to model electrical

components where the sequential execution time is too high to meet the real-time requirements. The coupling of the preprocessing module with the proposed heuristic search method provide the means for the practical solution of different task system.

TABLE DES MATIÈRES

Page

LISTE DES FIGURES

Page

LISTE DES TABLEAUX

Page

LISTE DES SYMBOLES

$<\alpha \mid \beta \mid \rho>$	Dénote un problème de répartition où α indique le nombre de processeurs, β indique le type de tâches et ρ indique le critère d'optimisation utilisé.
$\mathcal{T}$	Système des tâches.
$\mathcal{P}$	Système des processeurs.
$\mathcal{G}$	Ensemble des regroupements de tâches.
$\prec$	Ordre partiel.
$\not\subset$	Non inclusion ensembliste.
$\subseteq$	Inclusion ensembliste.
$\gamma^-(x)$	Ensemble des successeurs de x où x appartient à un système avec relations de préséance.
$\Gamma(y)$	Ensemble des voisins de y où y appartient à un système sans relations de préséance.
$\mathcal{H}, \chi$	Horaire d'exécution et horaire partiel.
$\mathcal{N}, \mathcal{N}^+$	Ensemble des nombres naturels et nombres naturels positifs.
$\mathcal{R}, \mathcal{R}^+$	Ensemble des nombres réels et nombres réels positifs.
$\gamma^+(x)$	Ensemble des prédécesseurs de x où x appartient à un système avec relations de préséance.
$\{\varnothing\}$	Ensemble vide.

$\mid A \mid$	Cardinalité de l'ensemble A.
$\prec_f$	Réduction polynomiale.
P	Un processeur.
R, G, L, C, Y	Matrices des résistances, conductances, inductances, capacités et admittances.
T	Une tâche.
V, I	Matrices de tensions et de courants.

LISTE DES SIGLES ET ABRÉVIATIONS

MH	Mapping Heuristic.
A^*	Méthode de fouille heuristique dans un espace d'états.
A/N-N/A	Convertisseur analogie à numérique et numérique à analogique.
CC, CA	Courant continu, courant alternatif.
DMA	Accès direct de la mémoire.
DPSM	Dynamic Pruning Search Method.
E/S	Entrée/Sortie.
EDD	Temps d'achèvement des tâches le plus court (Earliest Due Date).
EDF	Plus court échéancier d'abord (Earliest Deadline First).
ITS	Iterative Tree Search.
MA^*	Memory-bounded A^*.
MAXCLIQUE	Problème sous la forme décisionnelle qui consiste à séparer un graphe disjonctif en cliques disjointes.
MCVF	Variable la plus contraignante d'abord.
MIMD	Instructions multiple et données multiple.

OL	Ordonnancement par liste de priorité.
OL^+	Ordonnancement par liste modifié pour remplir le temps mort des processeurs.
PARTITION	Problème sous la forme décisionnelle qui consiste à séparer un graphe quelconque en parties disjointes.
ROM, RAM	Mémoire morte et mémoire vive.
R-T-T	Réception-Traitement-Transmission. Mode de communication impliquant la réception, le traitement et la transmission des données.
RTTR	Répartition des Tâches Temps Réel. Problème sous la forme décisionnelle qui consiste à obtenir une répartition des tâches dans plusieurs processeurs.
RTTR-$(\prec, u)$	Problème sous la forme décisionnelle qui consiste à obtenir une répartition des tâches dans plusieurs processeurs tout en tenant compte des relations de préséance et l'utilisation des ressources des tâches.
SIMD	Instructions identiques et données multiple.
SMA^*	Simplified MA^*.
VME	Versa Module Extension.

LISTE DES ANNEXES

Page

Chapitre 1

Introduction

1.1 Problématique

La répartition automatique des tâches est un problème fondamental dans la simulation parallèle et en temps réel des réseaux électriques. La coordination des activités à l'intérieur d'un système distribué est normalement assurée par une politique de répartition des tâches qui permet une utilisation équitable des ressources et améliore le rendement du système [STA88]. De plus, une répartition efficace des tâches peut diminuer le temps d'exécution d'un programme en profitant du parallélisme (implicite et explicite) des tâches à exécuter. Dans le cas d'un système temps réel et distribué, la répartition des tâches joue un rôle encore plus prépondérant. En effet, dans le traitement en temps réel, l'exactitude des résultats d'un calcul ne repose pas uniquement sur la logique du programme mais dépend aussi de l'instant où ces résultats sont produits. En d'autres mots, il ne suffit pas d'obtenir un résultat logiquement correct, il faut également l'obtenir au bon moment [STA88]. Conséquemment, la répartition des tâches dans un système temps réel distribué est plus qu'un désir de réduire le temps d'exécution du programme parallèle. La répartition des tâches ayant des contraintes de type temps réel est essentielle au bon fonctionnement du système tant du point de vue logiciel que matériel.

La répartition automatique des tâches a comme fonction d'assigner les tâches requises pour la simulation des réseaux électriques dans le réseau des processeurs. Cette répartition est basée sur un ensemble de contraintes imposées selon les besoins matériels et logiciels de l'environnement de simulation.

L'obtention d'une politique de répartition des tâches parallèles dans la simulation en temps réel des réseaux électriques n'est pas simple. La raison est qu'il s'agit d'un problème de nature **NP-complet** [GAR75], [GAR79]. Seuls les problèmes de répartition les plus simples possèdent des solutions algorithmiques calculables en un temps polynomial. Par exemple, il existe un algorithme capable d'obtenir une solution optimale en un temps polynomial pour le problème de répartition où le temps de communication entre les tâches est nul et où le graphe des tâches est en structure d'arborescence [HU61]. Cependant, la condition de la solution optimale n'est rencontrée que si le temps d'exécution de toutes les tâches est strictement unitaire. Clairement, ce genre de solution algorithmique n'est pas envisageable pour la simulation en temps réel des réseaux électriques.

Les techniques de répartition des tâches sont intimement liées à leur environnement d'implantation et du mode d'exécution désiré. Une méthode de répartition des tâches peut être très complexe et semi-automatique s'il s'agit d'une planification à long terme pour des simulations hors ligne de grande envergure. Par contre, le temps de réponse du répartiteur doit être très court s'il s'agit des séances de simulation interactive. Dans ce dernier cas, la complexité du répartiteur des tâches est généralement moins grande. Concernant l'environnement d'implantation, un répartiteur des tâches est un module du système d'exploitation de l'ordinateur parallèle. En pratique, les ressources informatiques allouées au répartiteur des tâches sont normalement limitées. Ainsi, l'utilisation de la mémoire vive du répartiteur des tâches doit être une fonction de sa disponibilité offerte par le système d'exploitation.

La conception et la réalisation de la méthode de répartition des tâches doivent tenir compte de ces contraintes pratiques imposées. L'exclusion de ces contraintes pratiques durant la phase de conception risque de produire un répartiteur qui ne convient pas à l'environnement d'implantation envisagé. Le défi est de concevoir une méthode de répartition automatique des tâches qui adhère au rôle qui lui est assigné tout en respectant les contraintes pratiques de l'environnement d'implantation et du mode d'exécution désiré.

1.2 Objectif de recherche

L'objectif principal de cette recherche consiste à apporter une contribution à la solution du problème de répartition automatique des tâches parallèles pour la simulation en temps réel par des méthodes avancées de traitements algorithmiques. Cet objectif implique l'étude et l'analyse des méthodes de répartition et des outils de réalisation existants applicables dans le domaine de la simulation parallèle en temps réel. Puis en formuler une solution novatrice. La poursuite de cet objectif de recherche conduit aux sous-objectifs suivants :

- étude des méthodes générales de répartition automatique des tâches;
- études des méthodes spécialisées pour la répartition des tâches temps réel;
- développement de nouvelles méthodes pour la répartition des tâches temps réel conformes à l'environnement d'implantation et du mode d'exécution désiré;
- vérification des méthodes de répartition développées dans un environnement de simulation en temps réel des réseaux électriques;
- proposition d'une nouvelle méthode adaptée à la simulation parallèle en temps réel des réseaux électriques.

1.3 Méthodologie de recherche

La première étape pour atteindre l'objectif principal et les sous-objectifs de cette recherche consiste à identifier clairement les paramètres et les contraintes matérielles et logicielles du problème de répartition automatique des tâches. Ces paramètres et contraintes nous guideront dans le choix des méthodes à étudier et dans la conception de la nouvelle méthode envisagée. L'identification des paramètres et des contraintes est également utile pour faire ressortir les particularités et les détails cachés du problème.

La répartition des tâches est un sujet de recherche qui concerne plusieurs disciplines scientifiques. Notamment, le domaine de la recherche opérationnelle a grandement contribué à l'élaboration et à la réalisation des premiers travaux sur le problème d'ordonnancement et de la répartition des tâches. Les résultats de ces premiers travaux ont été obtenus grâce à l'utilisation de la théorie des graphes et de la programmation mathématique [COF76]. La branche mathématique portant sur la théorie de décision a été également mise à contribution en montrant que le problème général de la répartition des tâches est indécidable. Cet aboutissement théorique de la mathématique moderne sur la décidabilité nous donne un cadre formel pour l'analyse du caractère **NP-complet** de notre problématique. De plus, si la conjecture **NP** est vraie alors elle devient une motivation supplémentaire pour la recherche d'une alternative à l'approche traditionnelle. C'est ainsi que le domaine de l'intelligence artificielle (I.A.) peut jouer un rôle important dans l'élaboration d'une nouvelle approche de solution à la problématique de la répartition automatique des tâches. En effet, l'utilisation des techniques de l'I.A. est souvent considérée comme une avenue intéressante pour contourner les problèmes non décidables [SCH90].

Donc, pour parvenir aux buts fixés, une combinaison des connaissances des domaines suivants est à envisager :

- la recherche opérationnelle;

- la théorie de décision et la théorie de la complexité;
- l'intelligence artificielle.

Ainsi, l'informatique et particulièrement l'utilisation des graphes pour la modélisation des tâches et des processeurs nous permettront d'étudier le comportement du répartiteur des tâches sous les contraintes imposées. Pour déterminer la nature intrinsèque du problème de répartition automatique des tâches parallèles pour la simulation en temps réel, la théorie des décisions est un outil essentiel. Une fois la nature **NP-complet** prouvée pour notre problématique, l'utilisation des techniques d'I.A. pour la solution du problème est alors pleinement justifiée.

Les réussites récentes dans le domaine de l'I.A. et en particulier la caractérisation formelle des heuristiques [PER88] serviront aux développements d'une méthode heuristique de répartition des tâches. Cependant, le domaine de l'I.A. est vaste, l'application de ses techniques doit être cernée et judicieuse. Leur utilisation doit être adaptée à notre problématique afin d'apporter une contribution réelle et nouvelle.

La méthodologie de recherche préconisée comprend également une revue bibliographique. Dans cette étape, nous nous intéressons surtout à connaître les différentes méthodes de modélisation des tâches et des processeurs présentées dans la littérature. La revue bibliographique nous aidera également à dégager les algorithmes et les méthodes de répartition déjà existants. La catégorisation des différentes méthodes utilisées dans la répartition des tâches a deux bénéfices : *i*) elle permet de situer ce projet de recherche par rapport aux autres; *ii*) elle indique le niveau de contribution apporté par les résultats de cette recherche.

1.4 Structure de la thèse

Afin de simplifier la présentation de cette thèse, un schéma bloc de sa structure est proposé à la figure 1.1.

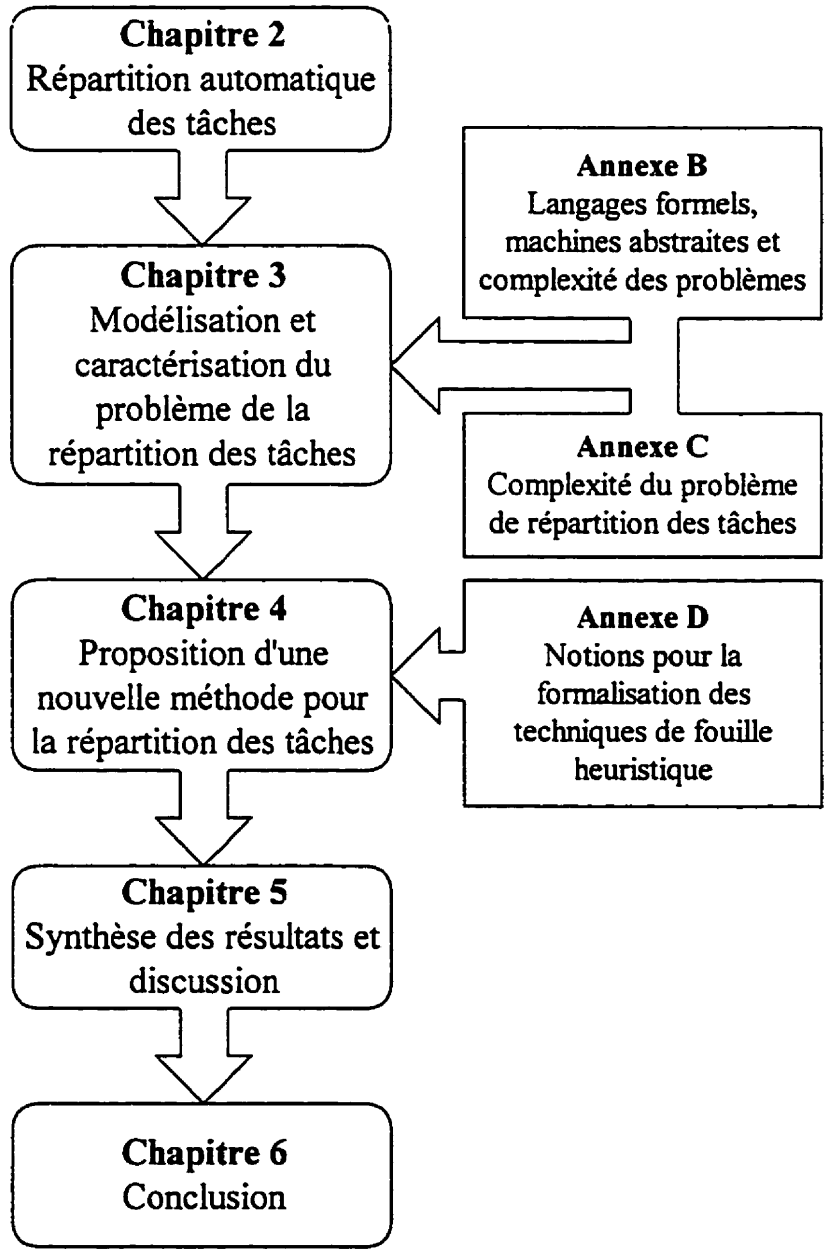


Figure 1.1 Structure de la thèse.

Au chapitre 2, nous présentons la formulation et l'analyse du problème de répartition automatique des tâches. C'est dans ce chapitre que nous explicitons les paramètres et les contraintes caractérisant le problème de répartition des tâches. Aussi, les techniques récentes de répartition et les résultats importants parus dans la littérature sont également exposés. Nous classifions les différentes techniques et approches existantes sous la forme d'un graphe hiérarchique. Cette classification est notre point de départ pour l'élaboration d'une nouvelle stratégie de répartition des tâches qui nous donnera une nouvelle classe de répartiteurs des tâches.

La modélisation du problème de répartition est présentée dans le chapitre 3. Cette modélisation permet de faire ressortir les relations qui existent entre les tâches, les contraintes de communication inter-tâches et la topologie d'interconnexion des processeurs. Nous discutons également dans ce troisième chapitre, les critères de performance appropriés pour mesurer l'efficacité de la classe des méthodes proposées par rapport à une méthode barème. Enfin, une preuve de la nature **NP-complet** de la problématique est également donnée afin de motiver le choix d'une stratégie de conception. Cette démonstration repose sur une nouvelle approche et est beaucoup plus simple que celle utilisée habituellement dans le domaine de l'informatique théorique. Le cadre formel de cette démonstration est présenté dans l'annexe B et la preuve complète est donnée dans l'annexe C.

Le travail impliqué dans la conception d'un système est rarement accompli d'une manière isolée et indépendante des étapes de modélisation. Au contraire, la conception et la modélisation sont des étapes très liées. En effet, le résultat d'une conception repose souvent sur le degré de raffinement du modèle utilisé et le degré de raffinement exigé d'une modélisation dépend de la qualité du résultat souhaité. La relation intime et imbriquée qui existe entre les étapes de la conception et de la modélisation peut être vue comme un processus itératif. Idéalement ce processus doit aboutir à un équilibre entre la

complexité du modèle et la convivialité de la conception. C'est pour cette raison que le chapitre 4 est le chapitre le plus imposant de cette thèse. Dans ce chapitre, une nouvelle méthode de répartition des tâches temps réel est présentée. Cette nouvelle méthode est basée sur une technique de fouille heuristique développée spécifiquement pour la répartition des tâches. La synthèse de cette technique de fouille heuristique a été réalisée à l'aide de graphes d'états munis de monoïdes d'arcs et de valeurs. Les notions formelles concernant les monoïdes et les graphes d'états sont données dans l'annexe D. L'utilisation de ces monoïdes d'ensembles permet une généralisation des résultats obtenus.

La nouvelle méthode de répartition proposée dans le chapitre 4 est évaluée par des simulations. Une évaluation qualitative et quantitative de la méthode proposée a également été obtenue. Ainsi, le chapitre 5 est consacré aux résultats de recherche obtenus et aux discussions sur la validité des méthodes de répartition des tâches proposées selon les exigences énoncées dans le chapitre 3.

Enfin, nous terminerons la présentation de cette thèse par la conclusion de l'approche utilisée dans la conception et la réalisation d'une classe de méthodes pour la répartition des tâches. Nous donnerons dans la conclusion un sommaire de la contribution algorithmique et méthodologique ainsi que la synthèse des résultats obtenus. Nous discuterons également de l'application des résultats de cette recherche et des perspectives intéressantes pour le développement futur.

Chapitre 2

Répartition automatique des tâches parallèles pour la simulation en temps réel

2.1 Répartition des tâches de type temps réel

La simulation numérique en temps réel offre une très grande flexibilité pour l'étude du comportement des réseaux de transport d'énergie. Mais la capacité de calcul d'un système monoprocesseur est parfois limitée. Le développement d'un simulateur basé sur une architecture parallèle de processeurs est donc une solution intéressante. Pour profiter pleinement d'un tel simulateur, un système de répartition automatique des tâches est souhaitable.

Or, le problème général de la répartition des tâches est **NP-complet** [GAR78]. Le problème de répartition qui nous intéresse concerne un système multiprocesseur impliquant des tâches: *i*) périodiques; *ii*) non-préemptibles; *iii*) à échéancier rigide avec contrainte d'exclusion mutuelle dans l'utilisation des ressources. Les méthodes opérantes de façon dynamique ne sont pas en mesure de garantir une solution optimale en un temps polynomial [DER89]. En d'autres termes, seule une fouille exhaustive est en mesure de garantir l'obtention d'une allocation optimale, si elle existe. Nous verrons dans la section 2.1.1 les résultats théoriques connus des différents problèmes de répartition.

Le modèle général du problème de répartition suppose l'existence d'un ensemble de ressources qui doivent satisfaire à un second ensemble, appelé ensemble de consommateurs, selon une certaine politique de service. Reposant sur la nature et les contraintes agissant sur les consommateurs et les ressources, le problème consiste à trouver une politique efficace pour gérer l'accès et l'utilisation des ressources selon une mesure de performance convenable.

Dans le contexte temps réel d'un simulateur numérique, les ressources sont représentées par les processeurs et les modules matériels de l'ordinateur parallèle tandis que les consommateurs sont des tâches effectuant un certain nombre de calculs. Un répartiteur de tâches est l'objet qui administre une politique de gestion gouvernant l'accès des tâches aux ressources de l'ordinateur sous un ensemble de contraintes imposées. La répartition manuelle des tâches n'est possible que dans des cas simples. Pour la simulation de grands réseaux électriques comme ceux d'Hydro-Québec, l'approche manuelle n'est pas réalisable à cause de la nature combinatoire du problème. Une méthode de répartition automatique est donc nécessaire voire essentielle au bon fonctionnement du système de simulation.

Le répartiteur automatique des tâches a comme fonction d'assigner les tâches requises pour la simulation des réseaux électriques dans les processeurs. De prime abord, cette répartition est basée sur les critères suivants: *i*) les processeurs doivent communiquer, dans la plupart des cas, avec les voisins immédiats; *ii*) le regroupement des tâches dans les processeurs est favorisé pour obtenir une meilleure utilisation des processeurs et pour minimiser la quantité d'information à transférer entre les processeurs.

La topologie des tâches requises pour simuler un réseau électrique reflète la topologie du réseau électrique. L'allocation des processeurs est directement liée à l'architecture de l'ordinateur parallèle. De plus, la répartition des tâches est une fonction

de la nature et des caractéristiques des tâches. Le but principal de la répartition automatique des tâches consiste donc à obtenir un horaire d'exécution qui rencontre l'échéancier de toutes les tâches du programme parallèle.

Il existe un grand nombre de travaux effectués dans le domaine de la répartition automatique traitant l'allocation des tâches dans des ordinateurs ayant des topologies d'interconnexion spécifique. Dans le cas d'un système à deux processeurs sans coûts de communication, la méthode d'allocation optimale fait partie de la famille des algorithmes « max-flow min-cut » ayant une complexité temporelle de $O(ne \log n)$ où n est le nombre de tâches et e est le nombre de liens de communication entre les tâches [TAR83]. Il en est de même pour les topologies d'interconnexion linéaire avec coûts de communication [LEE92]. Cependant, la topologie d'interconnexion des ordinateurs parallèles utilisés dans ces algorithmes optimaux est trop simpliste pour être utile dans la simulation en temps réel des réseaux électriques.

Dans le cas des ordinateurs à topologie d'interconnexion arbitraire, la plupart des répartiteurs de tâches pratiques sont des techniques améliorées de la méthode d'ordonnancement par liste [COF73], [ADA74]. Ces répartiteurs utilisent un ensemble de critères de performance et d'heuristiques pour régler la priorité des tâches à allouer dans les processeurs de l'ordinateur parallèle.

La méthode MH (*Mapping Heuristic*) [REW90], est un répartiteur basé sur l'ordonnancement par liste. La valeur de la priorité des tâches est calculée en fonction du niveau d'un sommet dans le graphe des tâches. Les tâches ne sont pas réparties selon la disponibilité des processeurs mais plutôt d'après le temps d'achèvement de la tâche. Cette politique de sélection est connue sous le nom de EFF (Earliest Finishing-time First). Le processeur retenu est celui qui peut achever le plus tôt l'exécution d'une tâche *prête*. C'est-à-dire une tâche dont ses prédécesseurs immédiats ont terminé leur exécution, tout en tenant compte des délais de communication inter-processeur. Ce

répartiteur est l'une des méthodes performantes capables d'allouer un ensemble de tâches dans un ordinateur à topologie arbitraire. Cependant, ce type de répartiteur ne tient pas compte de l'échéancier des tâches et par conséquent, il ne peut répondre aux besoins de la simulation en temps réel.

Dans la littérature, il existe plusieurs critères de performance proposés pour la répartition des tâches de type temps réel. Un grand nombre de fonctions d'objectifs sont formulées pour traiter différents aspects de ces critères de performance. La plupart de ces critères proviennent du domaine de la recherche opérationnelle. Nous pouvons citer, parmi ces fonctions d'objectifs, la minimisation de la somme des temps d'achèvement des tâches, la minimisation de la durée de l'horaire d'exécution des tâches, la minimisation du nombre de processeurs utilisés et la minimisation des retards par rapport à l'échéance des tâches. Il est à remarquer que les dates limites des tâches ne sont pas directement impliquées dans ces fonctions d'objectifs.

Nous pouvons apporter des critiques constructives à plusieurs de ces critères de performance. La somme des temps d'achèvement est un critère qui n'est pas intéressant pour les systèmes temps réels puisqu'il ne considère pas directement les propriétés temporelles des tâches (c'est-à-dire, les dates limites et la périodicité). La minimisation de la durée de l'horaire d'exécution des tâches peut aider à réduire l'utilisation des ressources du système informatique.

Malheureusement ce type de critère de performance ne tient pas compte du fait que chaque tâche possède une date limite bien précise. Il en est de même pour la minimisation du nombre de processeurs utilisés. Les retards par rapport à l'échéance des tâches est un critère plus intéressant que les autres. Ce dernier tient compte explicitement des échéanciers des tâches. En contrepartie, la minimisation des retards n'a d'utilité que si nous pouvons ajouter continuellement les ressources matérielles nécessaires afin d'éliminer les retards.

Pour la répartition des tâches temps réel, nous ne pouvons pas nous contenter d'appliquer simplement les critères de performance. Il nous faut également minimiser le nombre de tâches qui peuvent dépasser leur échéancier. Ainsi, nous pouvons stipuler qu'une méthode optimale de répartition des tâches temps réel est une méthode qui produit à tous coups un horaire d'exécution, soit un horaire dans lequel toutes les tâches satisfont aux contraintes temporelles imposées et dont la fonction d'objectif utilisée est minimisée (ou maximisée). Cette spécification a une portée plus grande que les définitions utilisées dans les méthodes d'optimisation classiques. En effet, un horaire d'exécution n'est utile que si les contraintes temporelles sont satisfaites. L'optimisation de la fonction d'objectif, à elle seule, ne peut garantir l'exactitude des relations temporelles de l'horaire généré.

2.1.1 Résultats connus: cas monoprocesseur

Afin de rendre la lecture de cette sous-section plus facile, nous utiliserons la notation <α | β | χ> pour désigner les différents problèmes de répartition de tâches. Dans cette notation α indique le nombre de processeurs utilisés, β indique le type de tâches impliquées et χ indique le critère d'optimisation utilisé [LAW83].

Soit un système de tâches indépendantes et non-préemptibles où chaque tâche j possède un temps de calcul e_j, un échéancier d_j et un temps de départ $r_t = 0$. Dans ce cas, le retard L de la tâche j par rapport à son échéancier est défini par $L_j = e_j - d_j$ et une solution optimale existe pour le problème <1 | indépendante, non-préemptible | $L_{max} = \max_j \{L_j\}$> par application de la règle de Jackson. Cette règle stipule que toute séquence de tâches placée en ordre non décroissant de d_j dans un processeur est une séquence d'exécution optimale [JAC55]. Cette règle est donc optimale par rapport au critère de retard maximal L_{max}. Aujourd'hui cette technique est connue sous le nom de méthode EDD (Earliest Due Date) [HOR74]. Cependant, lorsque le temps de départ r_t des tâches n'est pas égal à zéro alors la règle de Jackson n'est plus optimale. Par contre,

si les tâches sont préemptibles alors la méthode EDD demeure une méthode optimale pour la minimisation de L_{max} [DER74]. Donc, la règle de Jackson est également optimale pour le problème <1 | indépendante, préemptible, r_i | $L_{max} = \max_j \{L_j\}$>.

En observant de plus près la règle de Jackson, on s'aperçoit que si $L_{max} = 0$ alors toutes les tâches termineront leur exécution à temps puisque dans ce cas $e_j \leq d_j$. Cette observation a été étudiée et des méthodes optimales qui tiennent compte des contraintes temporelles des tâches ont été proposées par Liu et Layland pour le cas <1 | indépendante, préemptible | $L_{max} \leq 0$>. Pour ces problèmes, la séquence d'assignations utilisant la politique du plus court échéancier d'abord ou EDF (Earliest Deadline First) est une séquence d'assignations optimale [LIU73]. De plus, il existe une condition nécessaire et suffisante pour vérifier a priori la faisabilité de la solution. En effet, une solution est envisageable par les méthodes EDF si et seulement si:

$$\sum_j \frac{e_j}{u_j} \leq 1, \qquad (2.1.1.1)$$

où e_j, u_j sont respectivement le temps d'exécution et la période d'exécution de la tâche j.

Une des réalisations basées sur la politique EDF consiste à attribuer aux tâches une priorité statique qui est inversement proportionnelle à leur période. Donc, les tâches de courte périodicité reçoivent une grande priorité et seront exécutées en premier. Dans ce cas, une borne supérieure existe pour le taux d'utilisation du processeur. Cette borne est donnée par [LIU73]:

$$\sum_{j=1}^{n} \frac{e_j}{u_j} \leq n(2^{1/n} - 1), \qquad (2.1.1.2)$$

où n est le nombre de tâches à ordonnancer. Lorsque le nombre de tâches est grand, $n(2^{1/n} - 1) \rightarrow \ln 2$. Cela signifie que le taux maximal d'utilisation du processeur inférieur à 69% est une condition nécessaire pour que toutes les tâches rencontrent leur échéancier.

Il est à noter que toutes ces méthodes considèrent la périodicité et l'échéancier de durée identique. Pour les problèmes où la périodicité et l'échéancier sont différents, les priorités sont assignées selon l'étroitesse de l'échéancier. Plus une tâche possède un échéancier étroit plus sa priorité est haute [LEU82]. La nature optimale de cette méthode n'est établie que dans le sens suivant: Pour un ensemble de tâches T, si une méthode est en mesure d'ordonnancer T en utilisant une technique de priorité fixe quelconque alors une méthode utilisant le schème de priorité basée sur la politique EDF est également capable d'ordonnancer T.

Les différentes méthodes basées sur la politique EDF sont aujourd'hui connus sous le nom collectif d'ordonnancement à taux monotone (rate-monotonic scheduling [GAF91], [SHA94]). La préemption des tâches est le facteur prépondérant dans la nature optimale de ces méthodes [ZHA87]. Des travaux récents ont apporté d'importantes contributions à la robustesse de ces méthodes dans le contexte des ordinateurs monoprocesseurs et multiprocesseurs ainsi que leur implantation dans le langage ADA [LEH89], [ADA92], [HÄR94].

De plus, la méthode à taux monotone a été utilisée en partie dans le projet de la station orbitale Freedom pour la commande en temps réel et dans la gestion en temps réel du nouveau système de contrôle aérien AAS (Advanced Automation System) de l'agence américaine d'aviation FAA (Federal Aviation Agency) [STA94]. Cependant, l'ordonnancement à taux monotone exige la préemption et l'indépendance des tâches. Ce qui exclut les tâches communicantes à échéancier rigide et de périodicité très courte. Ces tâches sont souvent rencontrées dans la simulation en temps réel des phénomènes physiques transitoires. La politique EDF est donc insuffisante pour tenir compte de toutes ces contraintes.

Une modification de la politique EDF capable de traiter des tâches communicantes non préemptibles avec relation de préséance a été proposée par Lawler [LAW73]. Le

problème traité est du type <1 | préséance, non-préemptible | L_{max}>. Le principe de fonctionnement de cette méthode est fort simple. Une liste d'assignations est construite itérativement. À chaque itération, la tâche avec une valeur minimale (selon le critère d'optimisation choisi) est sélectionnée et placée dans la liste à condition que tous ses prédécesseurs sont déjà dans la liste. La méthode de Lawler a une complexité polynomiale $O(n^2)$ où n est le nombre de tâches. Malheureusement cette méthode ne peut tenir compte de l'utilisation exclusive des ressources par les tâches. Les ressources, dans ce contexte, sont des dispositifs autres que les processeurs du système informatique. Ces dispositifs peuvent être des sémaphores de synchronisation, des modules entrées-sorties ou de la mémoire physique.

En effet, dès qu'il existe une utilisation exclusive des ressources par les tâches, les méthodes basées sur les politiques semblables à EDF cesseront d'être efficaces. La raison est que les tâches de plus grandes priorités peuvent être bloquées par les tâches de plus basses priorités en cours d'utilisation d'une ou plusieurs ressources. Pour palier à ce problème, des protocoles de contrôle ont été proposés. Un protocole de contrôle est en fait un estimateur qui tente de déterminer la durée et le taux d'utilisation des ressources par les tâches [RAJ88], [CHE94]. Ces informations sont ensuite intégrées dans la méthode d'ordonnancement afin d'obtenir un horaire d'exécution acceptable. Ainsi, un protocole de contrôle est intimement lié à la méthode d'ordonnancement utilisée et par conséquent, il ne peut être généralisé. Dans un certain sens, on peut considérer les protocoles de contrôle comme des solutions ad hoc visant à résoudre les problèmes particuliers des méthodes d'ordonnancement par attribution de priorités.

En résumé, il existe des méthodes optimales capables de répartir les tâches de type temps réel pour le cas des systèmes monoprocesseurs. Par contre, leur rôle est confiné à des applications où le taux d'utilisation du processeur est faible (69% selon l'équation 2.1.1.2) et où l'utilisation des ressources n'est pas réalisée d'une manière exclusive.

2.1.2 Résultats connus: cas multiprocesseur

Pour les problèmes de répartition des tâches dans un système multiprocesseur, les résultats connus sont plutôt décevants. C'est-à-dire, la plupart de ces problèmes sont de complexité **NP-complet**. Certes, il existe des méthodes optimales pour la répartition des tâches dans des systèmes possédant plus d'un processeur. Or, ces méthodes font intervenir des problèmes qui sont souvent trop élémentaires pour être utiles en pratique. Néanmoins, nous pouvons utiliser ces résultats comme points de référence pour nous guider dans notre recherche.

Les premiers résultats concernent un système de processeurs homogènes dont la matrice de connectivité est pleine. Le nombre de ressources du système est noté par R_k où k est un indice indiquant le type de ressource disponible. Si $R_k = 0$, cela signifie qu'il n'y a pas de ressource de type k dans le système. Une ressource peut donc être utilisée en partie ou en totalité. Par exemple, il est possible d'utiliser seulement une partie de la mémoire physique mais il est toujours nécessaire d'utiliser d'une manière exclusive un port d'entrée-sortie. Le système des tâches est régi par une relation de préséance et chacune des tâches possède un temps d'exécution e_j. Les tâches ne sont pas préemptibles et possèdent un même échéancier.

Le problème de répartition des tâches impliquant deux processeurs, $R_k = 0$, $\forall\ k$ et dont $e_j = 1$, $\forall\ j$ est résoluble d'une manière optimale en un temps polynomial [COF72]. Dans ce cas simpliste une répartition basée sur le principe de "max-flow min-cut" est applicable. Or, si le temps d'exécution des tâches n'était pas uniforme ($\exists\ j$, $e_j \neq 1$), alors le problème serait dans la classe **NP-complet** [GAR75]. En fait même si le temps d'exécution des tâches est confiné dans l'ensemble $e_j = \{1, 2\}$, le problème demeure toujours **NP-complet** [GAR75]. Le tableau 2.1 est un résumé de différents résultats parus dans la littérature impliquant un ensemble de tâches sans préemption.

Tableau 2.1 Résumé des résultats connus pour les systèmes multiprocesseurs.

Nb. proc.	Ressources	Ordre	Temps exéc.	Complexité	Réf.
2	0	arbitraire	uniforme	polynomiale	[COF72]
2	0	indépendant	arbitraire	**NP-complet**	[GAR75]
2	0	arbitraire	1 ou 2 unités	**NP-complet**	[GAR75]
2	1	arborescent	uniforme	**NP-complet**	[GAR75]
3	1	indépendant	uniforme	**NP-complet**	[GAR75]
n	0	arborescent	uniforme	polynomiale	[HU61]
n	0	arbitraire	uniforme	**NP-complet**	[ULL73]

Décidément, presque tous les problèmes dont les tâches ne sont pas préemptibles appartiennent à la classe de complexité **NP-complet**. D'après le tableau 2.1 la répartition des tâches dans un système multiprocesseur est **NP-complet** lorsque le temps d'exécution des tâches n'est pas uniforme ou lorsque les tâches utilisent des ressources. Intuitivement, nous devons recourir à des méthodes heuristiques pour de tels problèmes.

De plus, même si les tâches sont préemptibles, cela ne veut pas dire que le problème est automatiquement résoluble en un temps polynomial. Par exemple, le problème de répartition où le critère d'optimisation est la minimisation de la somme des temps d'exécution pondéré $<n \mid$ préemptible $\mid \sum_j w_j e_j >$ est en fait identique au problème où les tâches ne sont pas préemptibles $<n \mid$ non préemptible $\mid \sum_j w_j e_j >$ [MAC59]. Dans les deux cas, le problème est **NP-complet**. Donc, dans le cas des systèmes multiprocesseurs, la préemption des tâches n'apporte pas nécessaire une solution plus facile.

Nous pouvons penser que l'application des méthodes optimales pour les systèmes monoprocesseurs peut aider à la solution des problèmes multiprocesseurs. Or, le problème $<n \mid$ préséance, non-préemptible $\mid L_{max} >$ présenté dans la section 2.1.1 est

résoluble en un temps polynomial lorsque $n = 1$ mais devient **NP-complet** lorsque $n > 1$ [LAW83].

Même les schèmes utilisant le principe de priorité fixe peuvent causer des problèmes inattendus. Supposons qu'un ensemble de tâches soit réparti d'une manière optimale (dans le sens du critère d'optimisation choisi) dans le réseau des processeurs. Supposons également que cette répartition est obtenue grâce à une attribution de priorités fixes. Alors, pour le même ensemble de tâches et en utilisant la même méthode de répartition, on peut observer une augmentation dans la durée de l'horaire d'exécution si on augmentait le nombre de processeurs utilisés ou si on diminuait le temps d'exécution des tâches. Les tâches qui étaient réparties d'une manière optimale peuvent ne plus rencontrer leurs échéanciers à cause de l'augmentation de la durée de l'horaire d'exécution. Ce résultat surprenant des systèmes multiprocesseurs est appelé l'anomalie de Richard [GRA76]. Cette anomalie signifie que les méthodes par attribution de priorités peuvent donner ponctuellement de très bons résultats mais que ces résultats ne sont plus valides dès que l'on modifie les paramètres du problème à solutionner.

De prime abord, l'anomalie de Richard semble contre intuitive. Après tout on ajoute de nouveaux processeurs ou on diminue la durée d'exécution des tâches pour relâcher les contraintes temporelles et spatiales du problème et non le contraire. Mais pour un système multiprocesseur où l'effet domino est beaucoup plus important, l'anomalie de Richard est une réalité qui se produit fréquemment. On peut imaginer la situation où deux tâches T_i et T_j sont allouées dans le même processeur. En réduisant le temps d'exécution de T_i, la tâche T_j démarre aussi son exécution à un moment plus tôt. En démarrant son exécution à un moment différent, T_j peut très bien causer un blocage à des tâches situées dans un autre processeur. Cette situation de blocage existe si T_j utilise une ressource partagée. Donc, la diminution du temps d'exécution d'une tâche à l'intérieur d'un horaire de durée minimale peut rendre ce dernier non minimale. L'effet

résultant est un horaire inapplicable puisque certaines tâches peuvent ne plus rencontrer leur échéancier.

Il est possible d'éviter l'anomalie de Richard en laissant les tâches tourner dans une boucle vide jusqu'à la fin de leur temps d'exécution original. Mais cette technique est peu efficace en terme d'utilisation des ressources disponibles. Des méthodes ingénieuses ont été proposées pour contourner ce problème [SHE93]. La plupart d'entre elles sont des heuristiques qui consistent à réutiliser la portion du temps disponible à d'autres fins. Cependant ces méthodes ne s'appliquent qu'à des systèmes relativement lents. Elles ne conviennent donc pas à notre problème de répartition où le pas de calcul est dans les microsecondes.

On voit donc la nécessité d'introduire une approche systématique de répartition qui puisse tenir compte à la fois des différentes topologies d'interconnexion matérielle et des exigences des tâches temps réel à échéancier rigide et de courte périodicité. La réalisation d'un répartiteur automatique implique donc l'obtention d'une ou plusieurs procédures pour la solution d'un ensemble de contraintes mixtes tout en évitant les problèmes combinatoires associés.

2.1.3 Répartiteur de tâches

L'allocation des processeurs est directement reliée à l'architecture de l'ordinateur parallèle. De plus la répartition des tâches est une fonction de la nature et des caractéristiques des tâches. Le but principal de la répartition automatique des tâches de type temps réel consiste donc à obtenir un horaire d'exécution qui respecte l'échéancier de toutes les tâches du programme parallèle.

Le répartiteur des tâches est l'un des composants du noyau exécutif du système d'exploitation. Dans notre cas, le système d'exploitation du simulateur est très léger. La

raison est que nous devons minimiser la surcharge imposée par le noyau exécutif afin de donner une plus grande flexibilité dans l'établissement des échéanciers des tâches. La figure 2.1 donne l'emplacement logique du répartiteur dans le système informatique du simulateur.

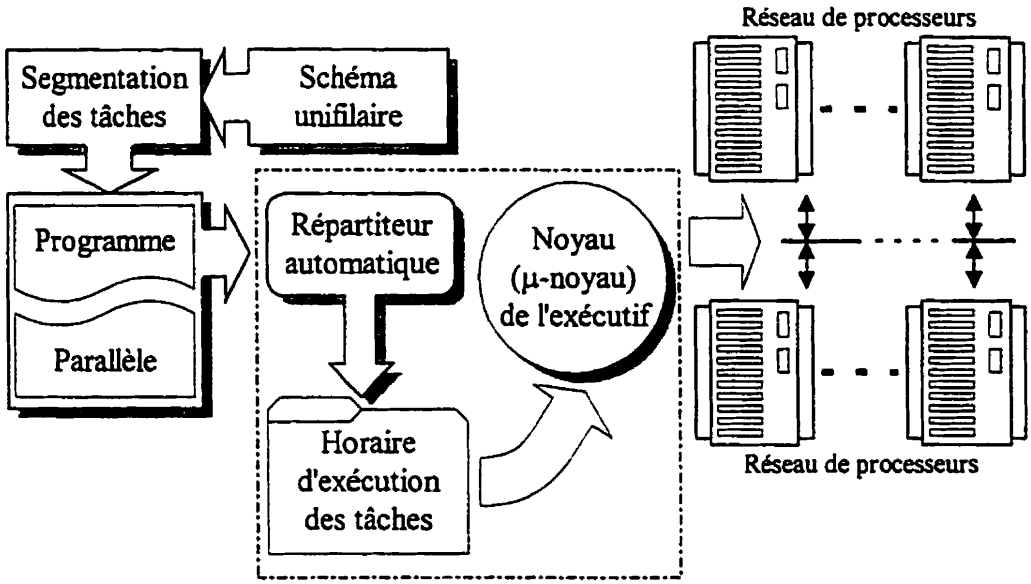


Figure 2.1 Utilisation du répartiteur dans le système informatique du simulateur.

Les tâches du programme parallèle sont générées automatiquement par un module de segmentation des tâches. La section 2.2.1 explique le principe utilisé dans la segmentation des réseaux électriques à simuler. Le programme est ensuite compilé pour produire des modules exécutables distincts. Le répartiteur accepte ces modules et produit un horaire d'exécution.

Le micro-noyau prend en charge les tâches et les distribue dans le réseau des processeurs selon les directives inscrites dans l'horaire d'exécution. Le fonctionnement du système informatique est donc très simple. Mais cette simplicité est nécessaire pour satisfaire les contraintes temporelles du simulateur. Le tableau 2.2 présente les caractéristiques générales de notre problème de répartition des tâches.

Tableau 2.2 Caractéristiques générales du répartiteur automatique des tâches.

(1)	Toutes les tâches du simulateur ont des contraintes de type rigide;
(2)	les tâches peuvent posséder des relations de préséance (ordre partiel);
(3)	le nombre de tâches, leur dépendance, leur périodicité ainsi que leurs besoins en ressources informatiques sont connus a priori;
(4)	le nombre de données à transférer entre les tâches est un paramètre connu a priori;
(5)	les caractéristiques de communication (vitesse, délai de transferts, etc.) des processeurs sont connues d'avance;
(6)	la topologie d'interconnexions des processeurs ne change pas durant l'exécution d'un programme;
(7)	le simulateur est composé de processeurs homogènes ou hétérogènes.

Le répartiteur automatique des tâches a pour fonction d'assigner les tâches requises pour la simulation des réseaux électriques dans les processeurs. Dans cette recherche, la politique de répartition doit tenir compte des caractéristiques générales du problème et des besoins du simulateur parallèle [ROS94]. En plus des caractéristiques générales, certaines exigences sous-jacentes sont imposées au répartiteur de tâches. Ces exigences sont énumérés dans le tableau 2.3.

Alors, l'élaboration du répartiteur des tâches doit être faite en fonction des caractéristiques du problème et les exigences imposées. L'utilité de la méthode de répartition sera donc mesurée par rapport aux caractéristiques et exigences énoncées.

Tableau 2.3 Exigences pour le répartiteur automatique des tâches.

(1)	Les processeurs doivent communiquer, dans la mesure du possible, avec les voisins immédiats;
(2)	le regroupement des tâches dans les processeurs afin de procurer une meilleure utilisation des processeurs et pour minimiser la quantité d'informations à transférer entre les processeurs;
(3)	certaines tâches spéciales doivent être assignées à des processeurs particuliers. Ces tâches spéciales utilisent des ressources dédiées qui n'existent que sur un nombre restreint de processeurs.

2.2 Présentation du problème de répartition des tâches

Cette section présente les caractéristiques des objets manipulés par le répartiteur des tâches. Dans ce travail de recherche, les objets impliqués sont le système des réseaux électriques et le système informatique du simulateur. Donc, l'ensemble des tâches à répartir correspond au système de réseaux électriques à simuler et l'ensemble des processeurs représente le système informatique du simulateur. Nous débutons l'analyse du problème de répartition des tâches par une présentation des réseaux électriques et du système informatique utilisé.

2.2.1 Réseaux électriques

La topologie des tâches requises pour simuler un réseau électrique reflète la topologie du réseau. Ainsi, une tâche est un objet numérique qui effectue un ensemble de calculs. La plupart des calculs sont réalisés dans le domaine discret par des équations récurrentes. Une tâche peut échanger des résultats de calculs avec d'autres tâches. Une des caractéristiques importantes est que les tâches sont périodiques avec des contraintes temps réel de type rigide. Pour les besoins du simulateur, la période d'un pas de calcul

est réglée à $1/f = 50$ µs. À cette fréquence de calcul, nous pouvons simuler des lignes de haute tension en tronçons de longueur minimale de 15km. Cette longueur correspond à la vitesse de propagation de l'onde électrique. On distingue deux types de tâches appelées *ligne* et *poste*. La figure 2.2 montre un exemple d'interconnexions des lignes et des postes.

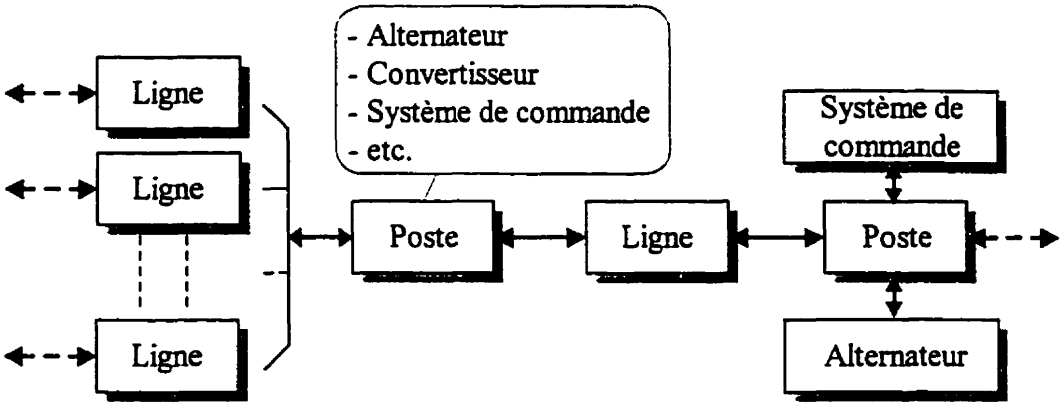


Figure 2.2 Connexions des lignes et des postes d'un réseau électrique.

Nous pouvons remarquer, dans la figure 2.2, que les extrémités des lignes sont terminées par des postes. En effet, une ligne est toujours reliée à deux postes. Les lignes sont des objets numériques qui simulent le comportement d'une ligne de haute tension en fonctionnement. Les postes, quant à eux, peuvent représenter les alternateurs, les convertisseurs CA/CC ou systèmes de commande qui sont reliés au réseau.

Cette modélisation des réseaux électriques permet d'utiliser une technique rationnelle pour réaliser la segmentation des composants du réseau en tâches informatiques. Supposons que les composants à l'intérieur des postes sont modélisés par des circuits équivalents linéaires ou non linéaires. On peut alors considérer les lignes de haute tension comme des éléments d'interface entre les différents postes d'un réseau électrique. Les dimensions de ces lignes par rapport à la longueur d'onde du courant électrique sont telles que l'on doit les considérer comme des lignes de transmission.

Ainsi, la propagation des ondes dans ces lignes de haute tension n'est pas instantanée. Ce délai de propagation des lignes rend possible la réalisation des calculs en parallèle [DUR93], [FAL93].

La relation qui existe entre la tension et le courant en un point *d* d'une ligne de transmission triphasée est donnée par les équations de (2.2.1.1) [STE82, chapitre 5.11], [MAG92, chapitre 2].

$$\begin{aligned} -\frac{\partial \mathbf{V}}{\partial d}(d,t) &= \mathbf{L}\frac{\partial \mathbf{I}}{\partial t}(d,t) + \mathbf{R}\mathbf{I}(d,t), \\ -\frac{\partial \mathbf{I}}{\partial d}(d,t) &= \mathbf{C}\frac{\partial \mathbf{V}}{\partial t}(d,t) + \mathbf{G}\mathbf{V}(d,t). \end{aligned} \qquad (2.2.1.1)$$

Dans ces équations différentielles partielles, le paramètre *d* est la distance mesurée à partir du point de transmission. Les quantités $\mathbf{V}(d, t)$ et $\mathbf{I}(d, t)$ sont des vecteurs de tension et de courant des phases. Les matrices **R**, **L** et **C** représentent les éléments passifs de la ligne et ont des valeurs qui sont reliées aux propriétés physiques de la ligne.

Un modèle simplifié a été proposé pour tenir compte des pertes dans une ligne de haute tension lorsque $\mathbf{R} \ll \omega\mathbf{L}$ [DOM69]. Ce modèle utilise un circuit équivalent contenant des sources de courant et des résistances en parallèle. Pour le cas des lignes triphasées, les phases sont décomposées en trois modes de propagation indépendante. La simulation des lignes de transmission peut être réalisée en utilisant le modèle de la figure 2.3.

Dans ce modèle les pertes de transmission sont représentées par des impédances. La propagation du courant est représentée par des sources de courant situées dans les extrémités de la ligne.

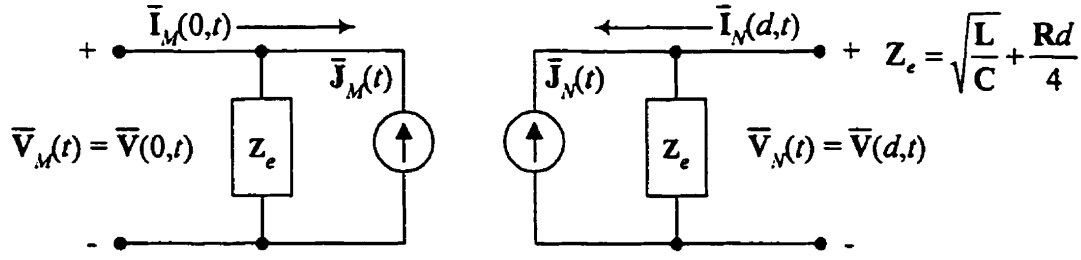


Figure 2.3 Modèle modal d'une ligne de transmission.

Les tensions de phase et les courants de phase sont obtenus en utilisant une transformation appropriée. Pour chaque mode de propagation m, la valeur instantanée des sources de courant du modèle de la figure 2.3 est donnée par [HUY93]:

$$\begin{aligned}
\bar{J}_{mM}(t) &= \frac{1-h}{2}\left(\frac{\bar{V}_{mM}(t-\tau)}{Z_e} + h\bar{I}_{mM}(t-\tau)\right) + \frac{1+h}{2}\left(\frac{\bar{V}_{mN}(t-\tau)}{Z_e} + h\bar{I}_{mN}(t-\tau)\right), \\
\bar{J}_{mN}(t) &= \frac{1-h}{2}\left(\frac{\bar{V}_{mN}(t-\tau)}{Z_e} + h\bar{I}_{mN}(t-\tau)\right) + \frac{1+h}{2}\left(\frac{\bar{V}_{mM}(t-\tau)}{Z_e} + h\bar{I}_{mM}(t-\tau)\right),
\end{aligned} \qquad (2.2.1.2)$$

où $m \in \{1, 2, 3\}$, $h = \left(\sqrt{\frac{L}{C}} - \frac{Rd}{4}\right) \Big/ \left(\sqrt{\frac{L}{C}} + \frac{Rd}{4}\right)$ et τ est le temps de propagation des ondes.

Les tensions modales $\bar{\mathbf{V}}$ de (2.2.1.2), quant à elles, sont déduites à partir des tensions de phases $\mathbf{V}$ des barres en utilisant une matrice de transformation modale $\mathbf{T}_V$. Nous avons donc,

$$\bar{\mathbf{V}} = \mathbf{T}_V^{-1}\mathbf{V}. \qquad (2.2.1.3)$$

Une fois les courants calculés dans le domaine modal, nous pouvons obtenir les courants de phase par une transformation similaire.

$$\mathbf{I} = \mathbf{T}_I\bar{\mathbf{I}}, \quad \mathbf{J} = \mathbf{T}_I\bar{\mathbf{J}}, \qquad (2.2.1.4)$$

où $\mathbf{I}$, $\mathbf{J}$ sont les vecteurs des courants de phase, $\bar{\mathbf{I}}$, $\bar{\mathbf{J}}$ sont les vecteurs de courants dans le domaine modal et $\mathbf{T}_I$ est la matrice de transformation modale pour le courant. Ainsi, le courant des sources au temps t est calculé à partir des valeurs obtenues au temps $t - \tau$. On peut alors effectuer le calcul de I_{mM} et I_{mN} à des intervalles de temps donnés par la valeur

de τ. À cause de ce délai de propagation des lignes, il est possible d'effectuer un certain nombre de calculs en parallèle. En effet, on peut calculer la tension des barres du réseau en même temps que les variables des lignes de transmission. La solution des équations de nœuds peut être réalisée en modélisant les composants à l'intérieur des postes par des éléments RLC (linéaires ou non linéaires) et des sources (commandées ou non).

Par la règle d'intégration trapézoïdale, les éléments inductifs et capacitifs sont équivalents à des circuits constitués d'une résistance en parallèle avec une source de courant tels que représentés dans la figure 2.4 [FAL93]. La valeur instantanée des sources de courant I_L et I_C est calculée à chaque pas d'intégration par :

$$\begin{aligned} I_L(t) &= I_L(t-\Delta t) + \frac{\Delta t}{2L}\left(v_M(t-\Delta t) - v_N(t-\Delta t)\right) + \frac{\Delta t}{2L}\left(v_M(t) - v_N(t)\right), \\ I_C(t) &= I_C(t-\Delta t) - \frac{2C}{\Delta t}\left(v_M(t-\Delta t) - v_N(t-\Delta t)\right) + \frac{2C}{\Delta t}\left(v_M(t) - v_N(t)\right), \end{aligned} \tag{2.2.1.5}$$

où Δt est la durée du pas d'intégration que l'on peut associer à la valeur du délai de propagation τ.

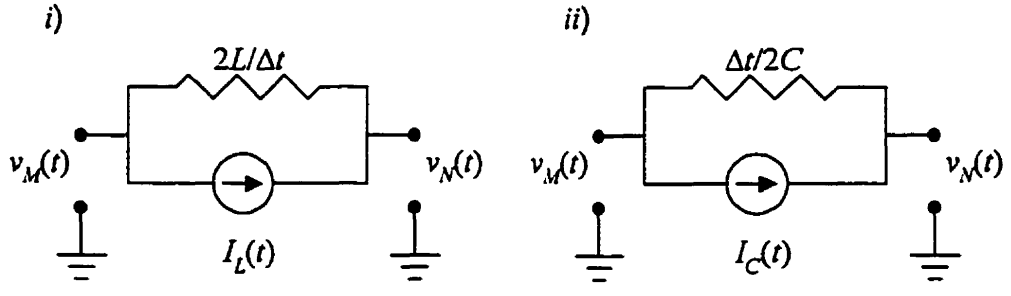


Figure 2.4 Circuits équivalents obtenus par la règle d'intégration trapézoïdale. *i*) Inductance. *ii*) Capacité.

Donc, d'une manière générale, nous pouvons obtenir les tensions de barre par la relation matricielle suivante :

$$\mathbf{V}(t) = \mathbf{Y}^{-1}(t)\,\mathbf{I}(t), \tag{2.2.1.6}$$

où **Y** est la matrice d'admittance contenant toutes les résistances équivalentes des composants et **I** est le vecteur de toutes les sources de courant équivalentes. En ne considérant que les systèmes temps invariants, on peut inverser a priori la matrice d'admittance **Y** avant le début des calculs [QUE96].

Pour des sous-systèmes de plus grande complexité, il est nécessaire d'effectuer les calculs séparément. Par exemple, les modèles numériques des machines synchrones et des turbines demande un nombre de calculs beaucoup plus élevé que les simples branches inductives et capacitives [QUE93]. La puissance de calcul des systèmes uniprocesseurs est nettement insuffisante pour traiter tous ces modèles à l'intérieur d'un pas de calcul. C'est pour cette raison que les modèles des sous-systèmes doivent être calculés par d'autres processeurs. L'ensemble des sous-systèmes est alors représenté par des sources commandées de tension ou de courant.

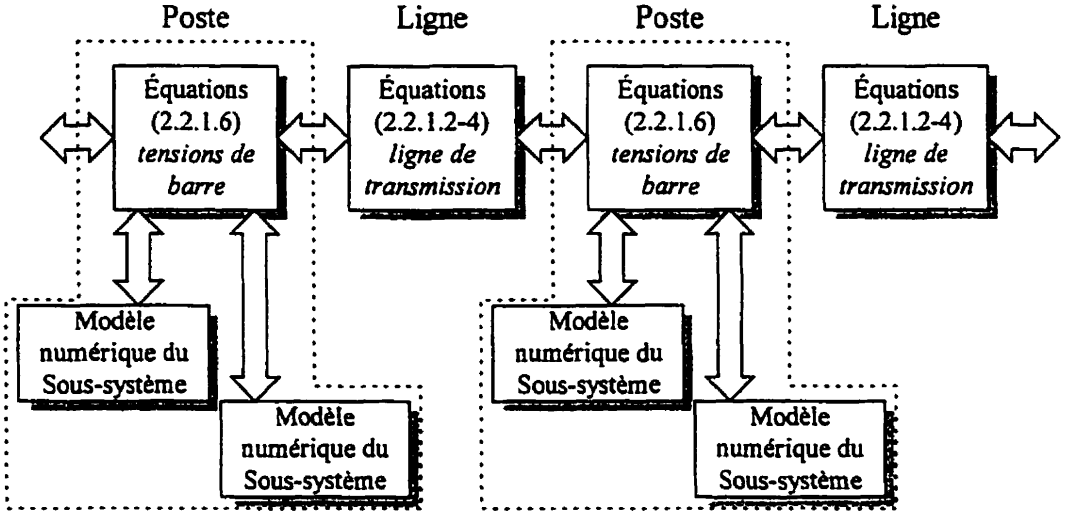


Figure 2.5 Schéma bloc montrant les calculs effectués en parallèle.

Ces sources commandées sont intégrées dans l'équation (2.2.1.6) et elles reçoivent leur consigne à partir des résultats calculés des sous-systèmes. Par conséquent, les postes sont simulés en même temps que les lignes de transmission d'un réseau. L'utilisation d'un système multiprocesseur permet donc le calcul parallèle des modèles numériques

utilisés. Ce concept de calcul parallèle mène naturellement à un ensemble de tâches informatiques communicantes. En effet, l'application de ce concept de parallélisme au schéma bloc de la figure 2.2 résulte en un ensemble d'opérations représentées dans la figure 2.5. Ces opérations de calculs sont effectuées en parallèle.

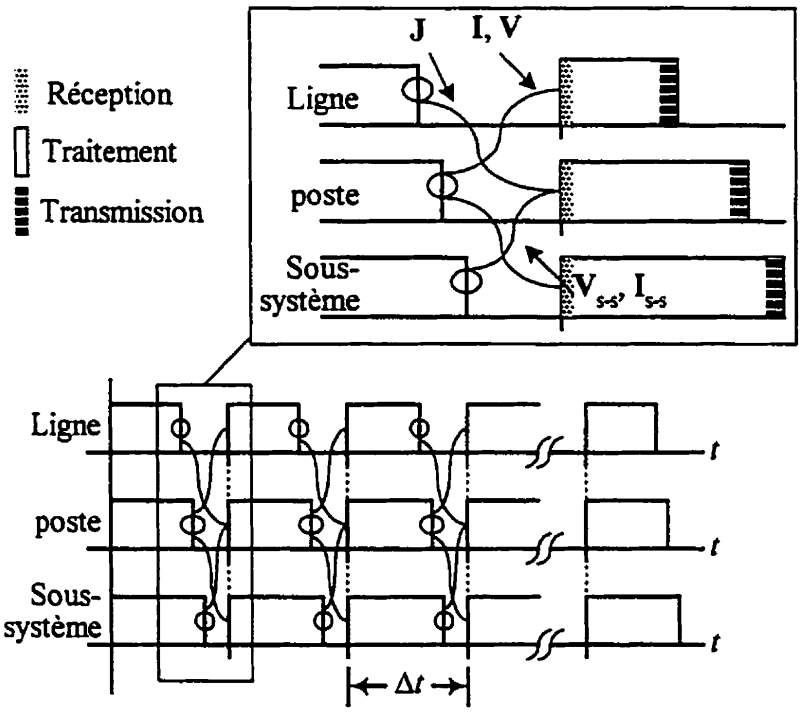


Figure 2.6 Chronogramme pour le transfert des données.

Les transferts des données nécessaires lors des calculs sont indiqués par les flèches bidirectionnelles du schéma bloc de la figure 2.5. À chaque pas de calcul, les équations du circuit équivalent des lignes sont calculées ainsi que les équations des tensions de barre. Parallèlement, les modèles numériques des sous-systèmes des postes sont évalués à l'aide d'algorithmes appropriés. Par conséquent, il est possible de gérer uniformément les transferts de données en utilisant la méthode de communication R-T-T (Reception-Traitement-Transmission) telle que résumée dans la figure 2.6. Cette figure présente l'aspect dynamique des échanges d'information entre les différents composants d'un réseau électrique.

Les tâches informatiques représentant les différents composants du réseau reçoivent leurs signaux au début de chaque pas de calcul. Ainsi, une ligne obtient les tensions et les courants de phase venant des postes (les vecteurs **V** et **I** de la figure 2.6). Un poste, quant à lui, reçoit les courants des sources **J** des lignes de transmission. Enfin, un poste reçoit également la contribution des sous-systèmes en termes de tensions et de courants (les vecteurs $\mathbf{V}_{s\text{-}s}$ et $\mathbf{I}_{s\text{-}s}$ de la figure 2.6) pour le calcul des tensions de barre.

Après la réception des données, chacune des tâches du réseau effectue un ensemble de calculs. Une ligne doit calculer les courants $\bar{\mathbf{J}}$ de ses sources internes. Pour réaliser ces calculs, la ligne transforme les quantités **V** et **I** reçues des postes en quantités modales. Les courants $\bar{\mathbf{J}}$ sont ensuite calculés dans le domaine modale à l'aide de l'équation (2.2.1.2) du modèle simplifié d'une ligne de transmission. Ces courants internes $\bar{\mathbf{J}}$ sont ensuite transformés en courants de phase **J** prêts pour être transférés aux postes qui lui sont reliés. Pour un poste, son travail consiste à former les tensions de barre en utilisant le vecteur de courant **J** venant des lignes, les vecteurs $\mathbf{V}_{s\text{-}s}$ et $\mathbf{I}_{s\text{-}s}$ venant des sous-systèmes ainsi que toutes les branches simples reliées à la barre. Donc, dans son étape de traitement, un poste doit solutionner l'équation nodale $\mathbf{V} = \mathbf{Y}^{-1}\,\mathbf{I}$ pour une barre donnée. L'étape de traitement pour les sous-systèmes des postes consiste à appliquer les algorithmes de calculs appropriés pour simuler leur comportement dynamique pour le pas de calcul en cours. Le traitement des sous-systèmes est également une étape qui exige un grand nombre de calculs puisque la complexité des modèles numériques utilisés est normalement plus grande que celle des autres composants du réseau. Il est donc possible qu'un sous-système soit segmenté en plusieurs parties pour le calcul parallèle [ASH90], [QUE93], [MAH95].

Une fois les calculs terminés, une ligne envoie ses courants internes **J** aux postes reliés à ses bornes. Ces courants seront utilisés par les postes pour le prochain pas de calcul. Il en est de même pour un poste, ce dernier doit transmettre le vecteur des tensions et des courants de la barre aux lignes pour le prochain pas de calcul. Comme

une ligne utilise les tensions $\overline{\mathbf{V}}(t-\tau)$ et $\overline{\mathbf{I}}(t-\tau)$ pour le calcul de ses courants internes, cette façon d'effectuer la communication ne produira pas de pertes de précision dans les calculs. Par contre, pour un sous-système, la transmission des données vers son poste est décalée d'un pas de calcul. L'effet de ce retard sur les tensions de barre ne causera pas de problème si la constante de temps du sous-système est beaucoup plus grande que la durée du pas de calcul [QUE96].

2.2.2 Système informatique du simulateur

Le système informatique du simulateur comprend un ordinateur parallèle ayant une architecture générale MIMD (Multiple Instruction Multiple Data). Durant les travaux de cette recherche, l'ordinateur parallèle est composé de processeurs homogènes TMS320C40 et ils sont organisés en nœuds de calcul. Chaque nœud contient quatre ou huit processeurs.

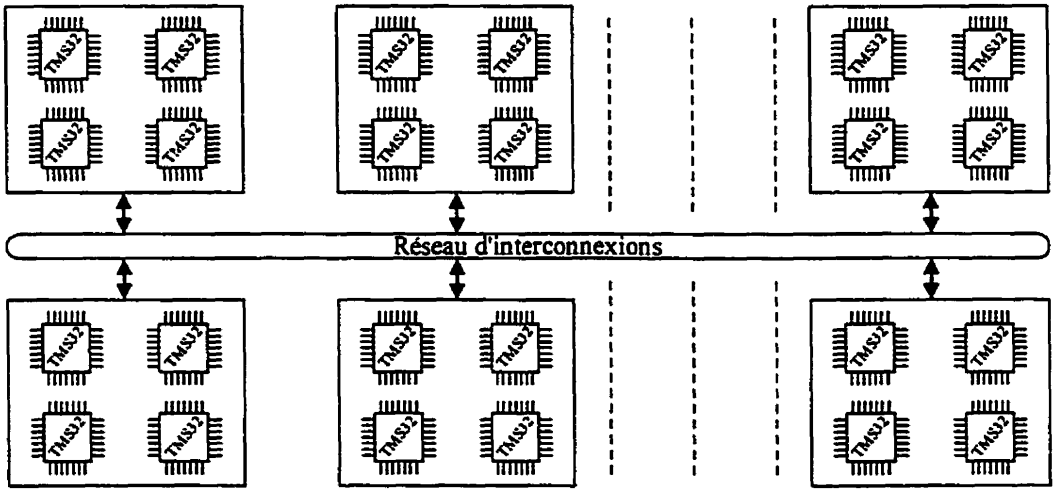


Figure 2.7 Organisation des processeurs de l'ordinateur parallèle.

La figure 2.7 est une représentation du réseau des processeurs. Les processeurs ainsi reliés peuvent transférer des données directement aux voisins immédiats et aux processeurs situés dans n'importe quel nœud du système via le réseau d'interconnexions.

Nous verrons plus loin dans cette sous-section, la composition détaillée du réseau d'interconnexions de l'ordinateur parallèle.

Remarquer que la communication entre les processeurs d'un même nœud peut être réalisée par le passage des messages ou par l'utilisation de la mémoire commune. Cependant, la taille de la mémoire commune est très limitée et est surtout utilisée pour l'entreposage des paramètres des modèles numériques à simuler. La communication entre les processeurs situés dans des nœuds de calculs différents est réalisée exclusivement à l'aide du paradigme de passage des messages. Enfin, le délai de communication inter-nœud est plus long que le délai de communication intra-nœud.

D'un point de vue de réalisation matérielle, les nœuds de l'ordinateur parallèle sont connectés sur un bus de standard industriel nommé VME (Versa Module Extension) établi par la compagnie Motorola. Ce bus est très approprié pour relier des composants hétéroclites pour former un système informatique sur mesure. Les processeurs d'un nœud de calcul, quant à eux, sont reliés aux processeurs des autres nœuds par câblage.

La figure 2.8 est un schéma descriptif représentant les composants matériels du simulateur. Le réseau des processeurs est représenté par la cage au coin supérieur droit. À noter que le bus VME et les interconnexions par câblage des processeurs sont aussi physiquement situés dans cette cage. La construction du schéma unifilaire représentant le réseau à simuler, la segmentation du réseau électrique en tâches informatiques, la compilation des programmes, la répartition des tâches ainsi que la gestion des fichiers sont effectuées à l'aide d'un ordinateur à usage général. Dans le système Hypersim d'Hydro-Québec, il s'agit d'un ordinateur de type poste de travail utilisant le système d'exploitation UNIX [QUE96].

Le simulateur comprend également une série de ports d'entrée-sortie numériques et analogiques. Les sorties numériques servent à enregistrer les données dans le système d'acquisition ou à imprimer sur table traçante. Ces équipements sont normalement utilisés pour effectuer des analyses postopératoires sur les données générées par le simulateur. Les convertisseurs A/N-N/A sont prévus pour effectuer l'interface entre le simulateur et des machines réelles. Ces machines sont commandées dynamiquement par le simulateur au cours de la simulation. Il est également possible de simuler numériquement les machines et les systèmes de commande du réseau électrique.

Le réseau d'interconnexion des nœuds sert à transporter des données transmises par les processeurs. Bien que le bus VME soit le médium principal d'interconnexions du simulateur, il ne sera pas utilisé pour la transmission des messages. La raison est que le bus VME n'est pas un bus déterministe. Lorsque plusieurs messages sont envoyés, ils sont placés sur le bus, le temps d'arrivée de ces messages peut varier grandement. Autrement dit, il est difficile de borner le temps de transfert de ce bus. De plus, la vitesse de transfert du bus VME est de loin inférieure à celle des ports de communication des processeurs utilisés.

Pour les besoins du simulateur, le transfert des messages par le bus VME est inadéquat. Par conséquent, nous utiliserons les ports de communication des processeurs pour effectuer la communication inter-processeur. Le bus VME est plutôt utilisé pour les transferts secondaires. C'est-à-dire, le transfert de programmes exécutables vers les processeurs, le changement des paramètres de simulation et l'enregistrement des données accumulées. À noter que le chargement des programmes et l'enregistrement des données sont effectués hors ligne alors que le changement des paramètres est accompli en ligne pendant le déroulement de la simulation.

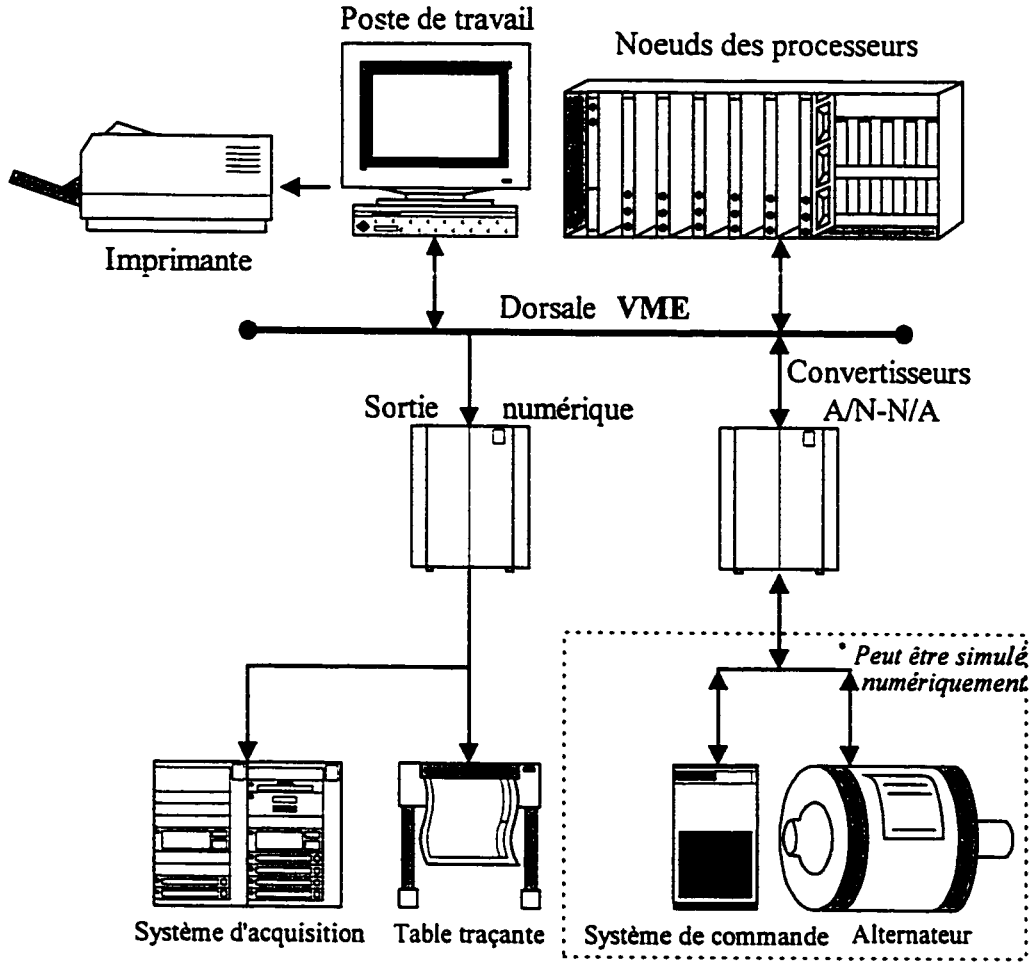


Figure 2.8 Composants matériels du simulateur.

Donc, le réseau d'interconnexion du simulateur est en fait composé de trois composants: 1) le bus VME pour les transferts secondaires; 2) les ports de communication des processeurs pour les transferts de messages; 3) la mémoire commune à un nœud pour les transferts intra-noeuds. L'ensemble de ces méthodes de communication rend le réseau d'interconnexions fort versatile et malléable pour la programmation du simulateur. La figure 2.9 présente les détails du réseau d'interconnexions d'un nœud à quatre processeurs.

Nous pouvons remarquer dans cette figure que chaque processeur possède une banque de mémoire dite *locale* et une banque de mémoire dite *globale*. Contrairement aux étiquettes données, ces banques de mémoire sont particulières à un processeur. Un processeur ne peut adresser directement une position mémoire d'un autre processeur. Ce qui évite la possibilité de corruption accidentelle des données.

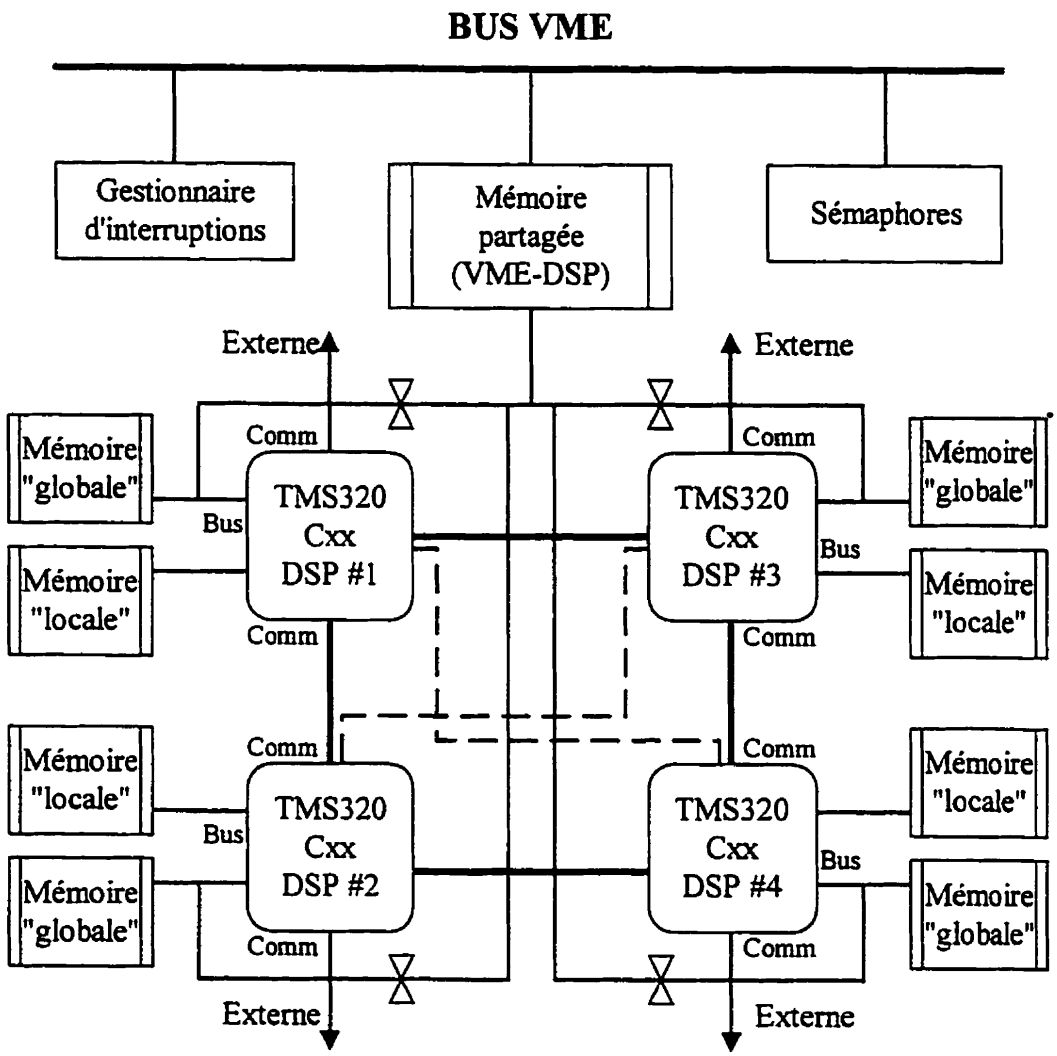


Figure 2.9 Réseau d'interconnexions d'un nœud de calcul.

La mémoire partagée VME-DSP est une mémoire à accès double (dual-port) qui sert à entreposer des variables locales au nœud. Il est possible d'accéder à cette mémoire simultanément via le bus VME et par les DSP d'un nœud. Les adresses de cette mémoire sont communes aux DSP. Un mécanisme d'arbitrage de bus est prévu pour palier aux conflits d'accès de cette mémoire. De plus, l'accès de la mémoire partagée peut être synchronisé d'une manière logicielle par les sémaphores disponibles. Ces sémaphores sont accessibles aussi bien par le bus VME que par les DSP du nœud. Comme le goulot d'étranglement est surtout situé du côté "mémoire partagée-DSP", nous n'utiliserons la mémoire partagée que pour l'entreposage des paramètres de simulation et lors du chargement de programme.

Tel que montré à la figure 2.9, les processeurs d'un nœud sont normalement reliés en anneau par des ports de communication. Cependant, quatre autres ports de communication sont réservés pour réaliser un nœud *pleinement connecté*. Dans une configuration pleinement connectée, chaque processeur d'un nœud est capable de communiquer avec un autre processeur via un lien direct. Enfin, chacun des processeurs possède un ensemble de ports de communication (cet ensemble de ports est noté par l'étiquette "externe" dans la figure 2.9). Ces ports de communication sont dédiés à la connexion inter-nœud. Donc, la communication inter-nœud peut être réalisée en reliant directement les processeurs concernés par ses ports de communication et ce, sans aucune interface logique.

La synchronisation globale des nœuds au cours d'une simulation est réalisée par des interruptions périodiques en provenance du bus VME. Ainsi, chaque nœud de l'ordinateur parallèle possède un gestionnaire d'interruptions. Le maître du bus VME est capable d'interrompre les processeurs individuellement ou d'envoyer les interruptions à tous les processeurs par la méthode de la diffusion publique (broadcasting).

Comme nous l'avons vu, les processeurs choisis pour le simulateur sont des processeurs dédiés au traitement numérique des signaux. Cette décision a surtout été motivée par la grande rapidité de calculs en virgule flottante de ces processeurs. Les processeurs TMS320C40 de Texas Instruments sont capables d'effectuer les multiplications et les divisions en virgule flottante de 40 bits en un seul cycle d'instruction. Pour un cycle d'instruction de 50 ns (horloge système de 40 MHz), le DSP est capable d'environs 20Mflops. La relation entre la fréquence de l'horloge du processeur `Freq_horloge` et la durée d'un cycle d'instruction est `cycle_instruction` = 2 / `Feq_horloge`.

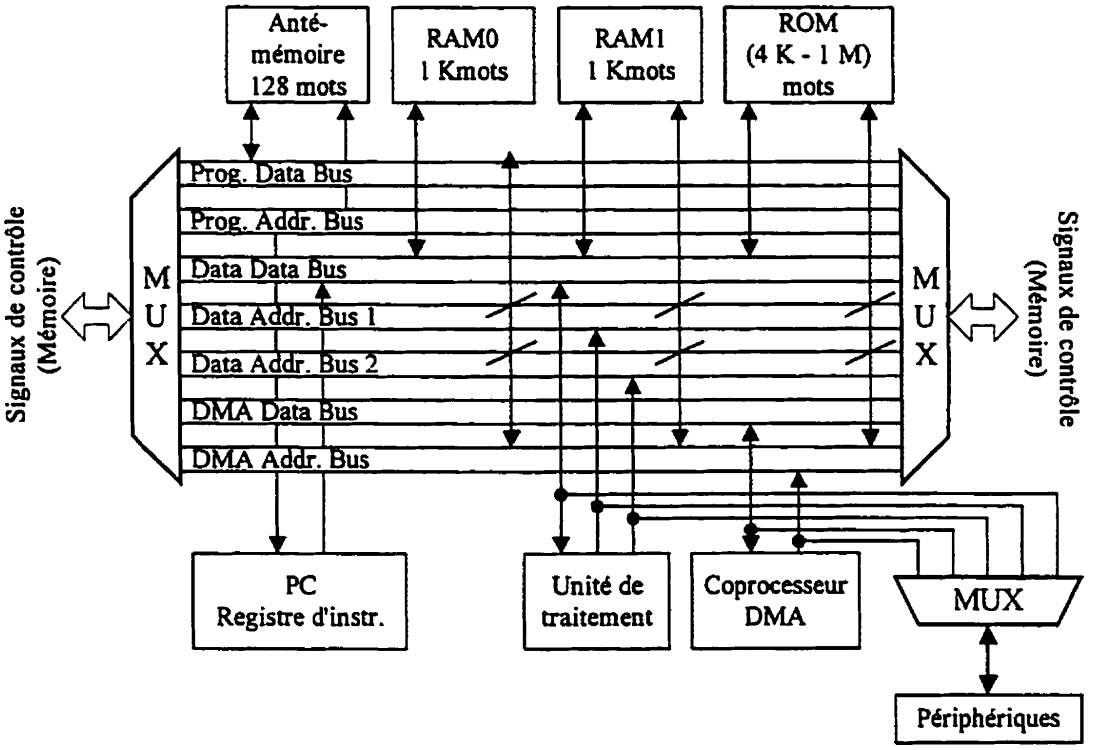


Figure 2.10 Architecture interne du processeur TMS320C40.

L'architecture interne du TMS320C40 est détaillée dans la figure 2.10. L'anté-mémoire d'instructions est réalisée directement sur le circuit du processeur. Cette anté-mémoire d'instructions sert à effectuer le préchargement (prefetch) des instructions du

programme. La capacité de traitement parallèle de ce processeur est représentée par trois bus d'adresses de données appelés respectivement *"Programme"*, *"Data1"* et *"Data2"*. Ces bus internes permettent au processeur de lire une instruction et de lire ou écrire deux opérandes en un seul cycle d'instruction.

Il y a six ports de communication dans le processeur TMS320C40. Ces ports de communication peuvent être contrôlés par le coprocesseur DMA (Direct Memory Access). La vitesse de transfert de chaque port est de 20Mo/s. Si le transfert des données est pré-programmé dans le DMA, alors aucune intervention de la part de l'unité de traitement n'est nécessaire. De plus, le DMA est doté de la capacité d'initialisation automatique, ce qui facilite grandement le transfert répété des données.

La plage de mémoire accessible par le processeur est de 4Gmots de 32 bits. L'adressage est linéaire puisqu'il n'y a aucune segmentation de la mémoire. Les premiers Mmots (32 bits) de la plage de mémoire sont réservés pour la mémoire à lecture seulement («ROM») si le processeur est en mode *micro-ordinateur*. Ces premiers Mmots sont disponibles à l'utilisateur si le processeur est en mode microprocesseur. Dans les secondes Mmots de la plage de mémoire, sont installés deux blocs de mémoire, vive statique (RAM0-RAM1) de 1kmots chacun. Ces blocs de mémoire servent à entreposer des variables locales à accessibilité fréquente. Ces mémoires internes au processeur sont surtout utilisées, en traitement numérique des signaux, pour le stockage des coefficients des filtres numériques.

2.3 Classification des méthodes de répartition des tâches

La répartition peut être *locale* ou *globale*. La répartition locale est spécifique à l'architecture des processeurs utilisés [FOX88]. Par exemple, dans les processeurs avec architecture pipeline, le degré de parallélisme est maximal lorsque le pipeline est plein.

Les méthodes d'ordonnancement local consistent à remplir le pipeline d'instructions le plus souvent possible.

Ce travail de répartition est normalement accompli par les processeurs eux-mêmes ou par le système d'exploitation. Le contrôle à ce niveau est souvent déterminé par le matériel employé. Quant à la répartition globale, il consiste à assigner les différentes tâches d'un programme aux processeurs de l'ordinateur parallèle. Dans notre contexte, un programme de calcul est organisé en un ensemble de tâches réparties dans les processeurs de l'ordinateur. Ce sont donc les méthodes de répartition globale qui nous intéressent. La figure 2.11 présente la taxinomie des méthodes de répartition parus dans la littérature [CAS88], [BUR90], [LEW92], [XU94].

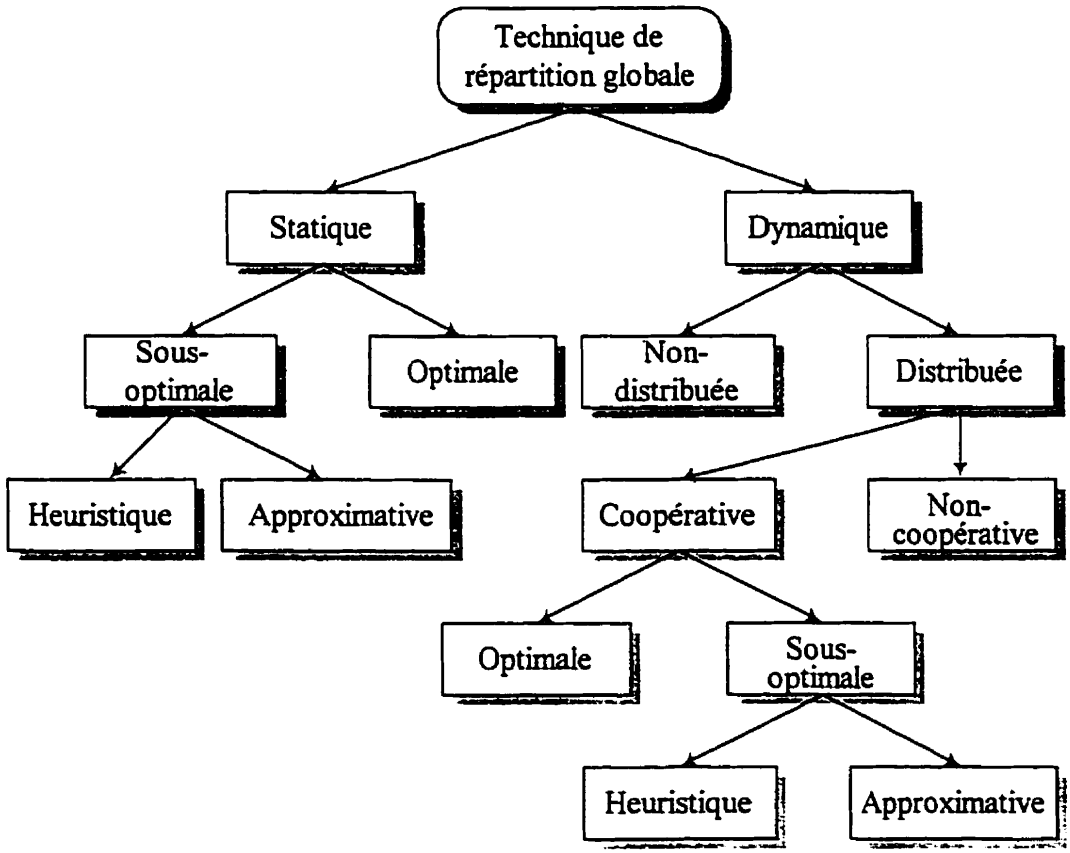


Figure 2.11 Classification des méthodes de répartition des tâches.

Il est possible de distinguer les méthodes de répartition globale en méthodes statiques ou dynamiques. Pour les méthodes statiques, le graphe des tâches ainsi que les contraintes imposées doivent être connus a priori et supposés constants pendant la durée d'exécution du programme parallèle. À l'opposé des méthodes statiques, les méthodes dynamiques répartissent les tâches pendant l'exécution du programme.

La répartition dynamique est effectuée en fonction des ressources disponibles au moment de l'arrivage des tâches. Notons que l'ordonnancement dynamique est souvent réalisé sous forme de balancement de la charge de calculs. L'énumération suivante explique les sous-classes de la figure 2.11. Certaines méthodes de répartition ne sont pas présentes dans cette classification.

Parmi ces méthodes figurent la répartition multiples applications, la répartition avec préemptive et la répartition adaptative. La répartition à multiples applications consiste à ordonner l'ensemble des applications dans un environnement parallèle mais en temps partagé. Une application est préemptible s'il est permis de suspendre l'exécution de certaines tâches de l'application.

Dans le cas du traitement en temps réel à échéancier rigide, la répartition préemptive peut poser des problèmes très difficiles à résoudre. Enfin, la répartition adaptative est un cas particulier des méthodes dynamiques où l'historique du système est un paramètre d'entrée de l'algorithme de répartition. Il est à noter que l'ordonnancement dynamique pose un problème de surcharge (overhead) due aux nombreux changements de contexte nécessaires pour le balancement des tâches pendant l'exécution du programme. De plus, son caractère dynamique impose une limite sérieuse à la complexité de l'algorithme de répartition utilisé.

Répartition statique

Optimale Une répartition optimale n'est que possible pour des problèmes particuliers et simplistes. Pour le cas général, qui a été reconnu comme étant **NP-complet**, aucune solution optimale n'a encore été proposée.

Sous-optimale

Approximatif Une répartition en un temps polynomial est possible en simplifiant les contraintes imposées, le modèle de la programmation parallèle et le modèle de l'ordinateur.

Heuristique Des algorithmes puissants peuvent être trouvés en utilisant l'intuition et l'expérience humaine.

Répartition dynamique

Non-distribuée Dans la répartition dynamique, les décisions de répartition sont prises après l'exécution du programme parallèle. Si le travail impliqué dans le rendu des décisions est assigné à un seul processeur, alors la répartition dynamique est dite *physiquement non-distribuée*.

Distribuée Le travail de la répartition est partagé par le réseau des processeurs de l'ordinateur parallèle. Dans ce cas, l'allocation des tâches peut être non-coopérative ou coopérative.

Non-coopérative Le répartiteur local effectue la répartition sans égard aux répartiteurs voisins.

Coopérative Les répartiteurs locaux collaborent pour obtenir un ordonnancement global basé sur la situation globale de la machine à un moment donné.

Si l'algorithme de répartition nécessite un temps de calcul trop long, il risque d'augmenter le temps d'exécution du programme. C'est pour ces raisons qu'on retrouve souvent dans l'ordonnancement dynamique, des algorithmes très modestes reposant sur quelques heuristiques simplistes.

Pour la simulation en temps réel des réseaux électriques, la répartition statique semble être la méthode la plus prometteuse. Étant donné que la structure du programme est bien définie pour un réseau donné et que cette structure demeure constante durant toute la simulation. Dans ce cas, la complexité des algorithmes peut être arbitrairement élevée sans pénaliser le temps d'exécution du programme. Bien qu'une répartition statique ne puisse profiter de l'évolution de la disponibilité des ressources du système, son efficacité demeure un facteur prépondérant par rapport à celle d'un ordonnancement dynamique.

Chapitre 3

Modélisation et analyse du problème de la répartition des tâches

3.1 Méthodologie

L'approche préconisée pour le présent travail de recherche consiste à répertorier l'ensemble des contraintes qui sont imposées au système de répartition, connaître les caractéristiques des tâches et de celles de l'ordinateur parallèle. Une fois ces différents paramètres obtenus, nous pouvons établir une modélisation des éléments en jeu puis élaborer une stratégie de solution.

Les contraintes imposées au répartiteur dépendent, en grande partie, de la nature des tâches à répartir et des méthodes de simulation utilisées. Les modèles dégagés dans ce chapitre serviront à l'analyse et à la conception du répartiteur des tâches. En modélisant le problème dans son entier, nous pouvons déduire un nombre de résultats intéressants. Ces résultats pourront nous guider dans la réalisation du répartiteur de tâches.

Les étapes suivies dans ce chapitre sont montrées dans la figure 3.1. Nous débutons par la synthèse des informations présentées dans le chapitre 2 en spécifiant les caractéristiques du répartiteur de tâches. À l'aide de cette synthèse, nous aborderons les

étapes de modélisation pour le système des tâches et pour l'ordinateur parallèle du simulateur. Ces modèles serviront à établir le comportement spécifique du répartiteur de tâches.

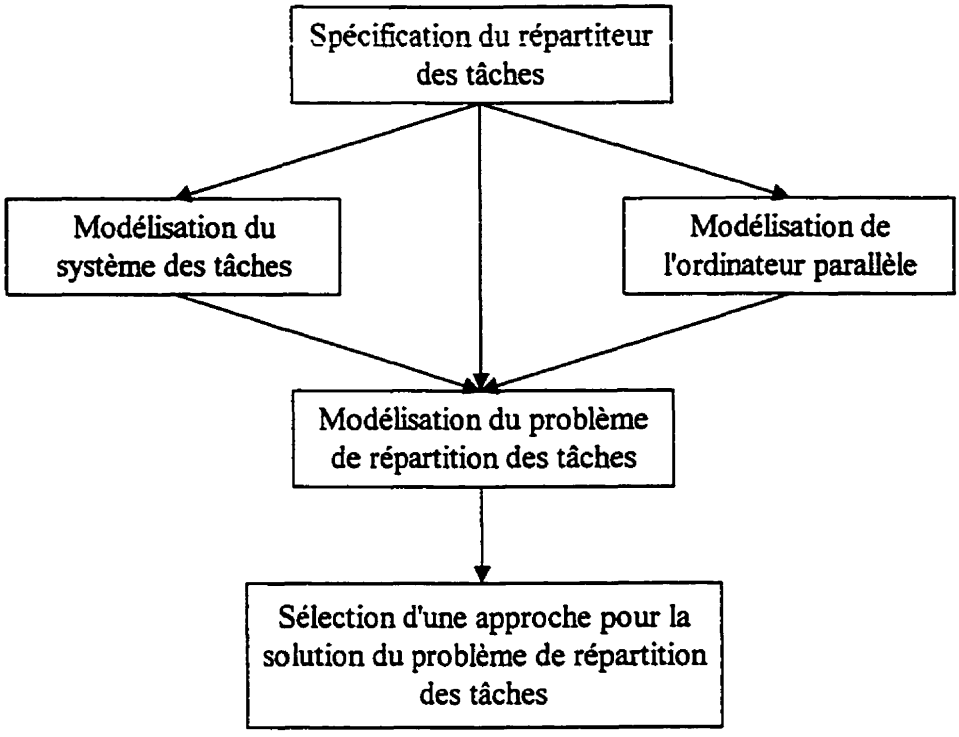


Figure 3.1 Étapes de conception du répartiteur des tâches.

Une analyse approfondie est ensuite effectuée sur le problème de la répartition des tâches. Les résultats de l'analyse seront mis à contribution pour la conception du répartiteur de tâches.

3.2 Spécification du répartiteur des tâches

Les caractéristiques générales des tâches et les exigences du répartiteur des tâches ont été présentées dans la section 2.2 et dans les tableaux 2.2 et 2.3 du chapitre 2. L'étude de ces données nous a permis de les regrouper en plusieurs catégories. La première catégorie concerne les caractéristiques et les contraintes du système des tâches.

Ces paramètres sont reliés à la façon dont les tâches sont générées par le module de segmentation des tâches. Ce module est une application directe de la méthode de simulation utilisée (section 2.2.1, chapitre 2).

La deuxième catégorie s'adresse à la structure du système parallèle constituant le simulateur. Les caractéristiques et les contraintes matérielles de l'ordinateur parallèle sont représentées dans cette catégorie. Enfin, la troisième catégorie traite les exigences et contraintes du répartiteur qui ne sont pas abordées par les deux premières catégories. Le contenu de ces trois catégories de notre problématique est explicité dans les tableaux 3.1 à 3.3.

Tableau 3.1 Spécifications du système des tâches.

(1)	Les tâches sont des objets de calcul reliés entre eux par des résultats de calculs;
(2)	la séquence d'exécution des tâches est périodique;
(3)	la durée d'un pas de calcul est la même pour toutes les tâches;
(4)	le temps d'exécution des tâches est inférieur à la durée d'un pas de calcul;
(5)	certaines tâches peuvent utiliser des convertisseurs AN/NA d'une manière exclusive.

Tableau 3.2 Spécifications du système parallèle.

(1)	Le système parallèle est de type MIMD « Multiple Instruction Multiple Data »;
(2)	le réseau des processeurs peut être homogène ou hétérogène;
(3)	l'architecture du système parallèle à mémoire distribuée;
(4)	les processeurs effectuent les transferts de données via des ports de communication ou via la mémoire commune;
(5)	certains processeurs ont des ports associés à des convertisseurs AN/NA;
(6)	la topologie d'interconnexion des processeurs ne change pas après le démarrage de la simulation.

Tableau 3.3 Implications pour le répartiteur des tâches.

(1)	Les tâches nécessitant des convertisseurs AN/NA doivent être assignées sur des processeurs ayant des ports AN/NA;
(2)	l'horaire d'exécution généré doit favoriser la communication entre processeurs voisins et la réduction des transferts de données inter-processeur;
(3)	la proposition d'une architecture d'interconnexions est souhaitée si la topologie d'interconnexions initiales des processeurs ne convient pas.

On voit par le tableau 3.1 que les tâches sont entièrement caractérisées par leur période d'exécution. Dans la méthode de simulation basée sur le délai de propagation des lignes, une tâche a une période d'exécution égale à la durée de ce délai (section 2.2.1, chapitre 2). L'échéancier des tâches est alors égal ou inférieur à sa période d'exécution. Puisque l'échéancier des tâches est du type rigide, le temps d'exécution des tâches doit aussi être inférieur à la périodicité des tâches. Cette dernière condition est la responsabilité des concepteurs qui réalisent les algorithmes de simulation. Les méthodes numériques utilisées dans la simulation des composants d'un réseau électrique doivent être décomposables en un ensemble de tâches. Le temps de calcul de ces ensembles est limité par l'échéancier des tâches. Ainsi, le temps d'exécution est donné comme une borne supérieure et non une caractérisation exacte du comportement dynamique des tâches.

Un système parallèle à mémoire partagée est par définition un système pleinement connecté : tous les processeurs sont reliés entre eux par la mémoire. Le transfert des données entre processeurs est très rapide. Cependant, il peut exister un goulot d'étranglement au niveau de l'accès des bus de données [TMS96, chapitre 8]. De plus, les systèmes à mémoire partagée ne sont pas extensibles. Le nombre de processeurs est souvent limité afin de réduire la contention dans les accès à la mémoire [BAC97, chapitre 3]. La plupart des réalisations commerciales (excluant les super-ordinateurs destinés pour des applications spéciales) tendent à restreindre le nombre de processeurs à 2^n. Ces réalisations contournent les problèmes de contention par un bus de données à

très grande vitesse. À l'heure actuelle, les machines disponibles sur le marché domestique ont une valeur maximale de $n = 8$ et disposent d'un système dont la vitesse de transfert de données entre la mémoire et les processeurs est près de 1 Gbits/sec [SUN97].

Malgré ces caractéristiques, les machines à mémoire partagée ne conviennent pas à la simulation en temps réel des réseaux électriques. La raison principale est l'intégrité des données en mémoire et la périodicité très courte des tâches. Dans un système à mémoire partagée, un seul espace d'adresse est utilisé par tous les processeurs. Nous devons instaurer une politique de contrôle pour la lecture et l'écriture afin d'assurer l'intégrité des données échangées entre les différents processeurs. L'application de cette politique de contrôle est la responsabilité du système d'exploitation. Or, la périodicité des tâches est de l'ordre de 50μsec (section 2.2.1, chapitre 2). Le temps disponible pour la gestion de l'intégrité des données est tout simplement trop limité.

L'extensibilité et la disponibilité des systèmes à mémoire distribuée sont une considération importante pour la simulation de grands réseaux électriques. Le nombre de processeurs n'est pas limité puisque l'interconnexion des processeurs n'est pas fixe. On peut relier les processeurs par une topologie d'interconnexion particulière pour un problème donné. Cependant, le problème de communication est plus difficile à résoudre. Le transfert de données entre deux processeurs n'est pas toujours direct. À moins d'avoir suffisamment de ports de communication dans tous les processeurs, le transfert de données passera éventuellement par des processeurs intermédiaires. Ce qui complique quelque peu la méthode de communication à utiliser. Cependant, en favorisant la répartition des tâches voisines dans des processeurs voisins, on peut diminuer le délai de communication encouru. De plus, si deux tâches communicantes sont placées dans un même processeur alors le temps de communication est réduit considérablement.

Pour les besoins du simulateur, certains processeurs du système sont associés à des convertisseurs AN/NA. Ces convertisseurs sont reliés à des équipements physiques du simulateur. Une tâche T_i qui utilise un convertisseur AN/NA particulier doit être assignée au processeur P_j qui est relié à ce convertisseur. C'est une contrainte spatiale que le répartiteur doit tenir compte. Cette contrainte spatiale a un effet double sur le répartiteur des tâches. D'un point de vue pratique, elle permet d'effectuer une répartition a priori de certaines tâches dans les processeurs appropriés. Par contre, pour une topologie d'interconnexions donnée, il devient plus difficile d'obtenir une répartition satisfaisante. La contrainte spatiale qui consiste à répartir T_i dans P_j peut effectivement éliminer certaines solutions acceptables de l'espace des solutions. Plus le nombre de contraintes spatiales dans le problème est grand plus l'élagage des solutions est important. À la limite, tout l'espace des solutions peut être éliminé par les contraintes spatiales excessives.

Le problème engendré par les contraintes spatiales excessives est parfois détectable par une «reconnaissance» de la topologie d'interconnexions des processeurs. Les techniques de reconnaissance topologique sont surtout utilisées dans l'ordonnancement des tâches où le système multiprocesseur est séparable en plusieurs sous-ensembles de processeurs. Pour un réseau de processeurs reliés en hypercube de dimension N, il est possible de vérifier en un temps polynomial, l'existence d'un hypercube de dimension $N - i$, pour $i < N - 1$ à partir de n'importe quel nœud de l'hypercube [CHE90], [BAS92]. En utilisant les contraintes spatiales comme nœuds de départ, nous pouvons déterminer la faisabilité d'une répartition par une technique de reconnaissance d'hypercubes. Toutefois ce genre de techniques ne s'applique pas lorsque la topologie d'interconnexions est arbitraire. En effet, le problème général de reconnaissance dans une topologie d'interconnexions arbitraire est un problème **NP-complet** [KRI92]. Autrement dit, la reconnaissance topologique est un problème aussi complexe que la répartition des tâches. Décidément, il y a là une impasse difficile à surmonter.

Malgré ces difficultés causées par les contraintes spatiales excessives, une solution reste envisageable. Au lieu de chercher à éviter le problème avant la répartition des tâches nous pouvons obtenir une meilleure appréciation de la situation après l'opération de la répartition. Si la répartition des tâches est effectuée hors ligne alors nous pouvons changer le réseau des processeurs pour contourner le problème des contraintes spatiales excessives. Pour ce faire, le répartiteur doit pouvoir suggérer les correctifs nécessaires en cas de problème. Cette capacité du répartiteur est un critère important dans la conception du répartiteur des tâches. Aussi, la proposition automatique d'une topologie d'interconnexions alternative par le répartiteur rend son utilisation plus conviviale. Bref, en cas de problème, le répartiteur des tâches ne produira pas seulement un message d'erreur laconique mais il proposera également une solution possible.

3.3 Modélisation du système des tâches

La modélisation des tâches est accomplie par une approche mixte. D'une part, les caractéristiques fonctionnelles des composants du réseau électrique sont explicitées mathématiquement. D'autre part, en utilisant le paradigme de la programmation orientée objet, ces caractéristiques sont considérées comme des attributs d'objets. Ces objets sont caractérisés par des attributs qui contrôlent leur fonctionnement. Certains attributs d'un objet peuvent être partagés par d'autres objets. De plus, ces objets peuvent englober d'autres objets dépendant du niveau d'abstraction utilisé.

Deux aspects fonctionnels des besoins informatiques du simulateur sont à retenir: l'aspect structurel et l'aspect comportemental. L'aspect structurel concerne les caractéristiques statiques des objets alors que l'aspect comportemental concerne le fonctionnement dynamique de ces mêmes objets et du réseau électrique dans son ensemble. En plus des aspects fonctionnels, nous pouvons distinguer deux niveaux d'abstraction dans la description des objets simulant un réseau électrique. Le niveau

élevé d'abstraction donne une vue d'ensemble du réseau électrique sans trop aller dans les détails de la composition des objets. L'accent, dans ce niveau d'abstraction, est mis sur les caractéristiques et la connexité des objets qui simulent le réseau électrique.

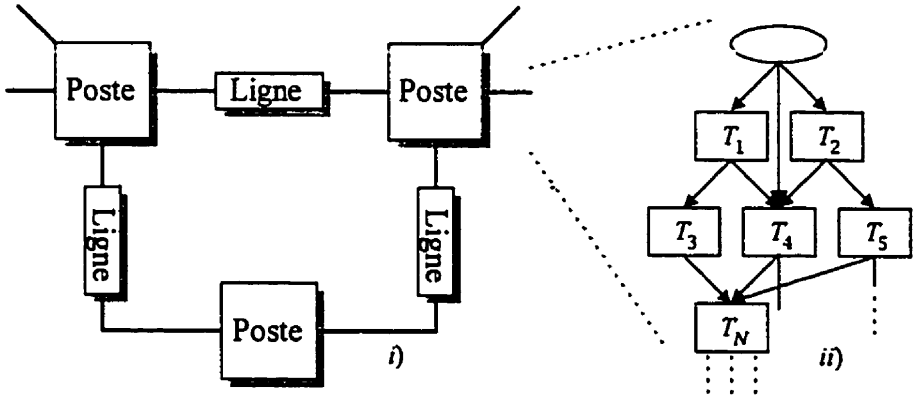


Figure 3.2 La modélisation du système des tâches: *i*) le niveau élevé d'abstraction; *ii*) niveau bas d'abstraction.

Tel que montré dans la figure 3.2, le niveau bas d'abstraction exprime la composition interne et la relation des constituants des objets. C'est à ce niveau d'abstraction que l'on retrouve la séparation du travail en unités élémentaires qui sont les tâches. Autrement dit, dans le niveau élevé d'abstraction, les objets sont des postes et des lignes d'un réseau électrique et dans le niveau bas d'abstraction, les objets sont les tâches qui effectuent les calculs numériques.

3.3.1 Aspect structurel des objets

Dans le niveau élevé d'abstraction, un réseau est composé d'objets qui sont des lignes et des postes. Ces objets sont paramétrisés. La valeur de leurs paramètres peut varier en cours de simulation. Chaque ligne du réseau est reliée à deux postes et chaque poste peut être relié à un nombre quelconque de lignes. À ce niveau d'abstraction du réseau électrique, le parallélisme des calculs est explicite et sa granularité est grosse. Le parallélisme est explicite parce que tous les objets du réseau évoluent en cadence

coordonnée et ce, sans relation de préséance. Le parallélisme est à gros grain parce que chaque objet peut contenir un grand nombre de procédures et routines. Chaque ligne et chaque poste disposent d'un ensemble de ressources utilisables. Ces ressources peuvent représenter un nombre de convertisseurs A/N et N/A, de ports séries et de ports parallèles.

Dans le niveau bas d'abstraction, les lignes et les postes peuvent englober d'autres objets que l'on appelle les tâches. Une ligne ou un poste est donc un macro-objet composé d'un ensemble fini de tâches. Ces tâches peuvent partager en tout ou en partie l'ensemble des paramètres du macro-objet. Le parallélisme des tâches à l'intérieur des macro-objets est implicite parce qu'il peut exister des relations de préséance parmi les tâches. De plus, la granularité de ce parallélisme implicite est plus fine. C'est à ce niveau d'abstraction que l'on distingue précisément l'utilisation des ressources disponibles. Les ressources utilisées seront indiquées dans les tâches.

3.3.2 Aspect comportemental des objets

Dans le niveau élevé d'abstraction, une séance de simulation est constituée d'un très grand nombre de pas de simulation. Il est possible de changer certains paramètres du réseau électrique en cours de simulation. Un cycle de simulation possède trois étapes : réception des signaux, calcul et transmission des résultats. Il y a échange d'information entre une ligne et les postes qui lui sont reliés. De plus, la durée d'un pas de calcul est inférieure à la période d'échantillonnage des lignes.

À chaque pas de simulation, une ligne doit recevoir les résultats du pas de calcul précédent et les signaux des deux postes qui lui sont reliés. Un poste doit recevoir les résultats du pas de calcul précédent et les signaux des lignes qui lui sont reliées. La période d'échantillonnage des lignes est normalement la même que celle des postes. On considère que la durée d'un pas de simulation est inférieure à la période

d'échantillonnage des lignes puisqu'il faut considérer les temps de réception et transmission des données (voir figure 2.6).

Dans le niveau bas d'abstraction, les tâches à l'intérieur des lignes et des postes sont normalement organisées par des relations de préséance. Les contraintes temporelles limites de l'ensemble des tâches d'un macro-objet sont les mêmes que celles du macro-objet. À chaque pas de simulation, les tâches d'un macro-objet doivent prendre les signaux d'entrée, effectuer les calculs avec les paramètres réglés puis envoyer les résultats vers leurs destinations. Les tâches d'un macro-objet peuvent communiquer entre elles mais pas avec les tâches d'autres macro-objets. La communication entre macro-objets est modélisée dans le niveau élevé d'abstraction.

3.3.3 Modèle du système des tâches

Définition 3.3.3.1 Un **système de tâches** est un couple $\mathcal{T} = (\mathcal{G}, \mathcal{L})$, où :

- $\mathcal{G} = \{G_1, G_2, ..., G_N\}$ est un ensemble de regroupements de tâches. Les regroupements de tâches de l'ensemble $\mathcal{G}$ peuvent avoir des caractéristiques en commun;
- $\mathcal{L}$ est une relation binaire entre les éléments de $\mathcal{G}$ exprimant la connexité des groupes de tâches.

■

Chaque regroupement de tâches G_k est un ensemble composé d'un nombre m_k de tâches:

$$G_k \subset \bigcup_{j=1}^{m_k} T_j, \quad m_k = |G_k|, \quad k = 1,2,\ldots N, \tag{3.3.3.1}$$

où T_j, m_k sont respectivement les tâches et la cardinalité des groupes G, et N est le nombre de regroupement de tâches dans $\mathcal{G}$. Ainsi, l'ensemble des regroupements de tâches peut être exprimée par

$$\mathcal{G} = \bigcup_{k=1}^{N} G_k \subset \bigcup_{k=1}^{N} \left\{ \bigcup_{j=1}^{m_k} T_j \right\}. \qquad (3.3.3.2)$$

La relation $\mathcal{L} \subset \mathcal{G} \times \mathcal{G}$ est une relation symétrique mais non réflexive. Autrement dit, $\forall x, y \in \mathcal{G}, (x, y) \in \mathcal{L} \Rightarrow (y, x) \in \mathcal{L}$ et $(x, x) \notin \mathcal{L}$. Donc, il n'y a pas de connexions entre un groupe et lui-même (liens redonnants) tandis que G_i connecté au groupe G_j signifie que G_j est aussi connecté à G_i. Fait à noter que $\mathcal{L}$ n'est pas une relation d'ordre sur $\mathcal{G}$.

Nous pouvons associer à $\mathcal{G}$ une application non symétrique λ: $\mathcal{G} \times \mathcal{G} \rightarrow \mathcal{N}$ qui indique le nombre de données à transférer entre deux regroupements de tâches. Cette application a comme propriété que $\lambda(G_i, G_j) = a > 0 \Leftrightarrow (G_i, G_j) \in \mathcal{L}$. Dans ce cas, a désigne le nombre de données à transférer entre G_i et G_j. Lorsque $\lambda(G_i, G_j) = 0$, il n'existe aucun lien de connexion entre G_i et G_j.

La représentation graphique de l'ensemble des tâches $\mathcal{T}$ est un graphe non orienté et pondéré dans lequel $\mathcal{G}$ est un ensemble de sommets. Les couples $(G_i, G_j) \in \mathcal{L}$ forment l'ensemble des arêtes. Le graphe est non orienté à cause de la symétrie de la relation $\mathcal{L}$ et il est possible de représenter une arête non orientée par deux arcs orientés, mais de sens contraire. L'image de l'application $\lambda(\cdot)$ sur le produit cartésien $\mathcal{G} \times \mathcal{G}$ est alors donnée par des valeurs inscrites sur les arêtes du graphe.

Nous pouvons définir, pour chaque sommet G_i du graphe $\mathcal{T}$, un ensemble $\Gamma_x = \{y \in \mathcal{G} \mid (x, y) \in \mathcal{L}\}$. C'est-à-dire, l'ensemble des sommets y qui sont reliés à x. On peut aussi dire que Γ_x est l'ensemble des voisins de x. La relation $\mathcal{L}$ étant symétrique, il est évident que l'ensemble Γ_x est égal à Γ_x^{-1}. Un chemin est une suite d'arêtes $(G_i, G_{i+1}, \ldots, G_N)$ telle que pour chaque arête (G_i, G_{i+1}), l'extrémité final d'une arête est égale à l'extrémité origine de l'arête suivante. Si $G_i = G_N$, alors on dit qu'il existe un circuit

passant par G_i. La modélisation du réseau électrique implique donc un graphe non orienté avec possibilité de circuits dans le graphe.

Dans le niveau bas d'abstraction, nous décomposons les lignes et les postes en ensembles de tâches. Les caractéristiques et l'utilisation des ressources de ces tâches seront définies. De plus il peut exister, à l'intérieur d'un ensemble de tâches, une organisation hiérarchique basée sur un ordre de préséance. Autrement dit, l'exécution d'une tâche n'est possible que si ses prédécesseurs ont achevé leurs activités.

Définition 3.3.3.2 Un **regroupement de tâches** G de l'ensemble $\mathcal{G}$ est un 5-uplet $G = (\prec, \Im, [a_{ij}], D, [r_i])$, où :

$\Im = \{T_1, T_2, ..., T_{m_k}\}$ est l'ensemble de tâches appartenant au regroupement G;

$\prec$ est un relation d'ordre partiel sur l'ensemble des tâches $\Im$, $T_i \prec T_j$ signifiant que T_i doit être exécutée avant T_j;

$[a_{ij}]$ est une matrice $m_k \times m_k$ exprimant la connexité et le nombre de données en octets à transférer entre les tâches de l'ensemble $\Im$;

D est une constante positive non nulle indiquant la période maximale d'exécution des tâches contenues dans $\Im$;

$[u_{ij}]$ est une matrice $m_k \times n$ qui indique les ressources utilisées par les tâches du regroupement. N est le nombre de ressources disponibles.

■

La relation d'ordre partiel $\prec$ est transitive et non réflexive. En effet, $\forall\ T_i$, T_j et $T_k \in \Im$, nous avons $T_i \not\prec T_i$ et $(T_i \prec T_j) \wedge (T_j \prec T_k) \Rightarrow T_i \prec T_k$. Ainsi, $a_{ij} = h > 0$ si et seulement si $T_i \prec T_j$ et h est le nombre de données à transférer entre T_i et T_j. On peut représenter la matrice $[a_{ij}]$ par une application $f_a : \Im \times \Im \rightarrow \mathcal{N}$ de sorte que $f_a(T_i, T_j) = a_{ij}$ et $f_a(T_i, T_j) = 0 \Rightarrow (T_i \not\prec T_j) \vee (T_j \not\prec T_i)$. Nous avons donc établi une relation fonctionnelle sur l'ensemble des couples $(T_i, T_j) \in \Im \times \Im$. Si $\prec = \varnothing$ pour l'ensemble $\Im$, alors $[a_{ij}] = \mathbf{0}$ puisque toutes les tâches de $\Im$ sont indépendantes et $f_a(\cdot)$ ne sera pas définie. Par contre,

si $\prec \neq \varnothing$ alors on dit que $\gamma^+(T_i) = \{v \in \Im \mid f_a(v, T_i) > 0\}$ est l'ensemble des prédécesseurs immédiats de T_i et $\gamma^-(T_i) = \{v \in \Im \mid f_a(T_i, v) > 0\}$ est l'ensemble des successeurs immédiats de T_i. Le contenu da la matrice $[u_{ij}]$ indique le type de ressources utilisées par chacune des tâches dans $\Im$. Ainsi, l'élément $u_{xy} = b$ signifie que la tâche T_x du regroupement utilise b ressources de type y.

Nous pouvons généraliser la définition 3.3.3.2 pour l'ensemble des regroupements de tâches de $\mathcal{G}$ en dénotant chaque 5-uplet de G par un indice. Ainsi $G_i = (\prec, \Im_i, A_i, D_i, R_i)$ où $\prec$ est la relation d'ordre partiel sur $\Im_i$, A_i est la matrice de connexions des tâches, D_i est la période maximale d'exécution du regroupement et $R_i = [r_i]_i$ est le vecteur des ressources du groupe G_i. L'application $f_a(\cdot)$ peut alors être définie comme une famille de fonctions $\mathcal{F}_i : \Im_i \times \Im_i \rightarrow \mathcal{N}$ pour $1 \leq i \leq N$ où N est la cardinalité de $\mathcal{G}$.

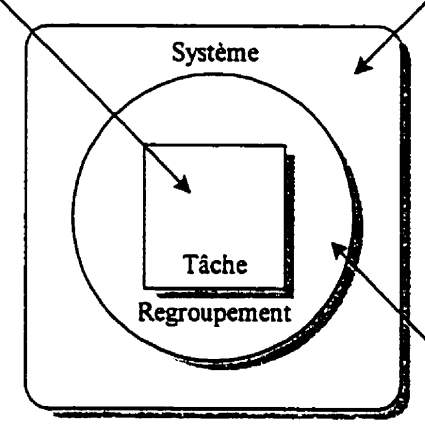
Définition 3.3.3.3 Une **tâche** $T_i \in \Im_j$ du regroupement G_j est caractérisée par un 7-uplet $T_i = (b_i^j, i_i^j, f_i^j, d_i^j, e_i^j, s_i^j, [u_i]_j)$, où :

- b_i^j est le temps de départ de la tâche T_i du groupe G_j;
- i_i^j est le nombre d'instructions à exécuter dans la tâche T_i du groupe G_j;
- f_i^j est le réciproque de la périodicité de la tâche T_i du groupe G_j;
- d_i^j est une constante positive non nulle indiquant la période maximale d'exécution des tâches contenues dans $\Im$;
- e_i^j est le nombre de données en octets à transférer de cette tâche;
- s_i^j est une variable booléenne qui indique la sévérité de l'échéancier. Si $s_i^j = 1$ alors d_i^j est une contrainte de type **rigide**. Dans le cas contraire, d_i^j est une contrainte de type **douce**;
- $[u_i]_j$ est un vecteur de longueur n indiquant l'utilisation des ressources par la tâche T_i du groupe G_j. $u_k = b$ signifie que la tâche T_i utilise b ressource de type k. n est le nombre de ressources disponibles.

■

D'abord, les regroupements de tâches peuvent contenir une ou plusieurs tâches. Donc, $|\Im_j| \geq 1$. Pour un ensemble $\Im_j$ donné, toutes les tâches partagent le même échéancier. Ainsi, pour $T_i \in \Im_j$, $1 \leq i \leq |\Im_j|$, nous pouvons avoir $d_i^j = D_j$ (la période maximale du regroupement G_j). Quand $D_j = \infty$ alors il n'y a pas de date limite pour $\{T_i\}$. Toujours pour un ensemble $\Im$ donné, la valeur $1/f_i^j$ de chacune des tâches doit être bornée par la périodicité maximale D du groupe. Ainsi, $1 \leq 1/f_i^j \leq D_j$ pour $1 \leq i \leq |\Im_j|$. Enfin, le temps de départ des tâches peut être nul, ce qui permet le démarrage simultané des tâches à l'intérieur de l'ensemble des regroupements de $\mathcal{G}$.

Tâche
Une tâche possède un temps de départ, une durée d'exécution, une périodicité, un échéancier (rigide ou doux), le nombre de données à transférer et le type de ressources utilisées.
Une tâche est un bloc de calcul utilisé pour la simulation d'un composant du réseau électrique.



Système de tâches: $(\mathcal{G}, \mathcal{L})$
$\mathcal{G}$ est un ensemble de regroupement de tâches. $\mathcal{L}$ est une relation qui exprime la connexité des regroupements dans $\mathcal{G}$. Le couple $(\mathcal{G}, \mathcal{L})$ *représente l'ensemble des lignes et des postes du réseau électrique à simuler.*
La communication entre les éléments du réseau est représenté par une application symétrique et non réflexive.

Regroupement de tâches
Chaque regroupement comprend un ensemble de tâches $\Im$, une relation d'ordre partiel régissant $\Im$ pour indiquer la préséance des tâches.
Une matrice de connexité est utilisée pour indiquer le nombre de données à transférer entre les tâches. Les ressources disponibles sont représentées par un vecteur.
Un regroupement représente une ligne ou un poste.

Figure 3.3 Résumé de la modélisation du système des tâches.

Un schéma bloc résumant les aspects importants de cette modélisation est présenté dans la figure 3.3. À noter que certains attributs du système des tâches sont dupliqués dans les modèles objets. Cette disponibilité des attributs est réalisée automatiquement par l'application des concepts « d'héritage » et de « polymorphisme » de la programmation orientée objet.

3.3.4 Modèle du système des processeurs

Définition 3.3.4.1 Le modèle du **système des processeurs** est représenté par un 6-uplet $(\mathcal{P}, [B_{ij}], [S_i], [I_i], [L_{ij}], [r_{ik}])$, où:

$\mathcal{P}$ = $\{P_1, ..., P_m\}$ est l'ensemble des processeurs formant l'ordinateur parallèle;

$[B_{ij}]$ est une matrice de connexions $m \times m$ indiquant la topologie d'interconnexion des processeurs;

$[S_i]$ est un vecteur de grandeur m indiquant la vitesse des processeurs $\mathcal{P}$;

$[I_i]$ est un vecteur de grandeur m indiquant le temps de démarrage nécessaire pour envoyer un message par le processeur P_i;

$[L_{ij}]$ est le taux de transmission en octets par seconde du lien de communication reliant les processeurs voisins P_i et P_j;

$[r_{ik}]$ est une matrice $m \times n$ indiquant les ressources disponibles sur le processeur P_i. $r_i^k > 0$ signifie que le processeur i dispose d'au moins une ressource de type k.

■

La topologie d'interconnexion de la machine parallèle peut être exprimée graphiquement par un graphe non orienté simple $H = (S, A)$. Les sommets S représentent les processeurs et les arêtes A représentent les liens de communication. Le temps d'exécution et le délai de communication du système peuvent être modélisés en utilisant la définition du modèle des tâches et la définition du modèle du système des processeurs. Ainsi, le temps d'exécution d'une tâche T_i assignée à un processeur P_j est :

$$E_{ij} = \frac{i_{T_i}}{S_{P_j}}, \tag{3.3.4.1}$$

où i_{T_i} est le nombre d'instructions de la tâche T_i. À noter qu'en pratique, le temps d'exécution E_{ij} est donné en mesurant le comportement asymptotique de la tâche T_i dans le processeur P_j. Dans ce cas, E_{ij} est le temps d'exécution maximal de T_i dans P_j.

Le délai de communication $C(T_1, T_2, P_1, P_2)$, quant à lui, est le temps de communication entre les tâches T_1 et T_2 lorsqu'elles sont exécutées par les processeurs P_1 et P_2 respectivement. Ce délai est donc une fonction des paramètres de performance de l'ordinateur parallèle et du nombre de données à transférer. D'abord, supposons que P_1 et P_2 sont deux processeurs voisins adjacents, alors le délai de communication d'un message envoyé par la tâche T_1 dans P_1 à une autre tâche T_2 dans P_2 par un lien direct est donné par

$$C(T_1, T_2, P_1, P_2) = \frac{a_{T_1T_2}}{L_{P_1P_2}} + I_{P_1}, \tag{3.3.4.2}$$

où $a_{T_1T_2}$ est le nombre de données à transférer entre les tâches T_1 et T_2 (définition 3.3.3.2).

Il est possible que plusieurs messages soient envoyés d'un processeur vers un autre en utilisant le même lien de communication. Dans ce cas, nous devons tenir compte du délai causé par la contention des liens de communication. Si le délai dû à la contention de communication entre les processeur P_1 et P_2 est estimable alors (3.3.4.2) peut être modifiée en

$$C(T_1, T_2, P_1, P_2) = \frac{a_{T_1T_2}}{L_{P_1P_2}} + I_{P_1} + CD_{T_1T_2P_1P_2} \tag{3.3.4.3}$$

où $CD_{T_1T_2P_1P_2}$ est le délai provoqué par la congestion dans le lien de communication reliant les processeurs P_1 et P_2.

En pratique, le coût de démarrage de transmission I_P et le délai de contention CD sont influencés par la méthode de communication utilisée et par la structure du sous-système de communication des processeurs. Pour le cadre de cette recherche, la méthode R-T-T est considérée (section 2.2.1, chapitre 2). Dans cette méthode de communication, tous les regroupements de tâches effectuent en parallèle les étapes suivantes : *i*) réception des messages; *ii*) traitement des données; *iii*) transmission des résultats. Les coûts de transmission et de réception sont bien délimités. Dans les processeurs modernes, le sous-système de communication est normalement contrôlé par un coprocesseur DMA (Direct Memory Access). Le sous-système de communication du processeur TMS320C40 comprend six ports indépendants mais identiques, la figure 3.3 présente les détails fonctionnels d'un de ces ports [TMS92, chapitre 8].

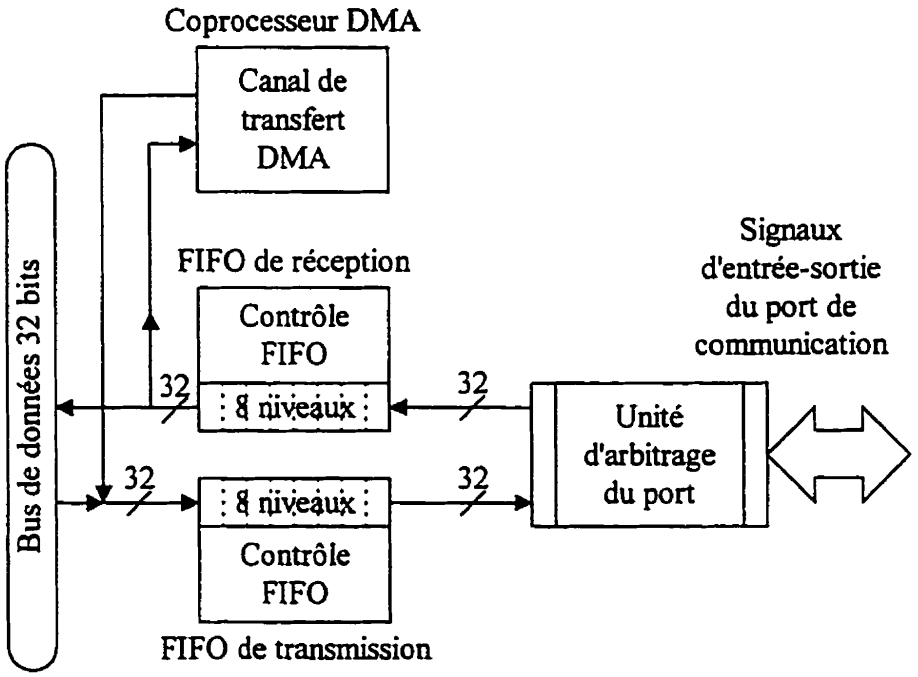


Figure 3.4 Structure fonctionnelle d'un port de communication du TMS320C40.

Chaque port de communication est contrôlé par un canal du coprocesseur DMA. Le transfert des données de la mémoire vers le port de communication est alors effectué

sans intervention du processeur central. Donc, le coût de démarrage l_p est négligeable si la transmission des messages est réalisée par DMA. Chaque port du sous-système de communication est muni d'un tampon de transmission et un tampon de réception de 8 niveaux (32bits chacun). Il n'y aura pas de contention dans la transmission, si la taille totale des messages destinés à un processeur est inférieure à 64 octets (puisque la liaison de deux ports de communication forme un tampon de 16 niveaux). Dans le cas contraire, le contenu de la queue FIFO de réception doit être libéré pour faire place aux nouveaux messages arrivants. Alors, le délai causé par la contention dans un lien de communication est :

$$CD_{T_iT_j,P_iP_j} = \begin{cases} 0 & a_{T_iT_j} \le l_{P_i} + l_{P_j}, \\ \left(a_{T_iT_j} - \left(l_{P_i} + l_{P_j}\right)\right) t_{\text{DMA}} & a_{T_iT_j} > l_{P_i} + l_{P_j}. \end{cases} \qquad (3.3.4.4)$$

Le paramètre l_P représente la longueur en octets de la queue FIFO, Le paramètre t_{DMA} représente le temps de transfert par octet du coprocesseur DMA. Pour un système homogène $l_{P_i} = l_{P_j}$. Pour un système hétérogène le temps de transfert t_{DMA} peut différer d'un processeur à l'autre. Dans ce cas, on utilisera le plus grand des t_{DMA} de P_i et P_j. Pour le TMS320C40, le transfert par DMA d'un mot de 32bits est de 50ns (un cycle d'instruction) [TMS92, chapitre 9].

Pour tenir compte de la situation où le processeur émetteur n'est pas voisin du processeur récepteur. Il est nécessaire de trouver le nombre de sauts parcourus par le message et l'intégrer dans le calcul du coût de communication. Supposons que le message envoyé de P_1 à P_2 utilise le chemin P_1, k_1, k_2, ..., k_z, P_2 où k_1, ..., k_z sont des processeurs intermédiaires du chemin d'envoi. Alors le délai de communication devient

$$C(T_1,T_2,P_1,P_2) = \frac{a_{T_1T_2}}{L_{P_1k_1}} + CD_{T_1T_2P_1k_1} + \frac{a_{T_1T_2}}{L_{P_2k_z}} + CD_{T_1T_2P_2k_z} + \sum_{i=1}^{z-1}\left(\frac{a_{T_1T_2}}{L_{k_ik_{i+1}}} + CD_{T_1T_2k_ik_{i+1}}\right). \qquad (3.3.4.5)$$

Lorsque l'ordinateur parallèle est composé de processeurs homogènes avec les caractéristiques d'E/S partout identiques, le calcul du délai de communication utilisant un chemin de communication indirect peut être simplifié en

$$C(T_1,T_2,P_1,P_2)=\left(\frac{a_{T_1T_2}}{L}\right)(z+1)+CD_{T_1T_2P_1P_2} \qquad (3.3.4.6)$$

où L représente le taux de transmission du réseau des processeurs, z est le nombre de sauts nécessaires pour atteindre le processeur qui est le récepteur du message. $z = 0$ est le cas où les processeurs P_1 et P_2 sont des voisins adjacents. Enfin, un schéma bloc résumant les aspects importants de cette modélisation est présenté dans la figure 3.5.

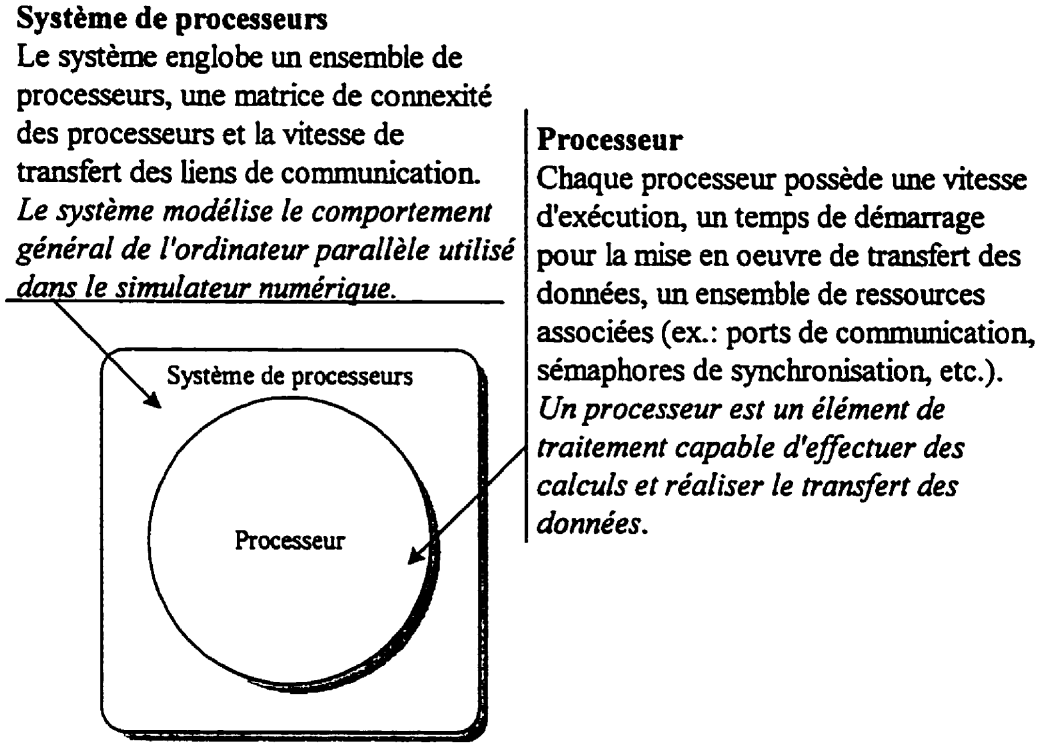


Figure 3.5 Résumé de la modélisation du système des processeurs.

3.3.5 Modélisation du problème de répartition de tâches

Un répartiteur de tâches accepte comme paramètres d'entrée, un modèle du système des tâches et un modèle du système des processeurs. Il produit à la fin de son opération un horaire d'exécution des tâches. Donc, le résultat de la solution du problème de répartition est un horaire dans lequel est inscrite l'affectation du système des tâches dans les processeurs et le temps de départ des tâches. Le temps de départ sert à indiquer l'ordre d'exécution des tâches. Le problème de la répartition des tâches est schématisé dans la figure 3.6.

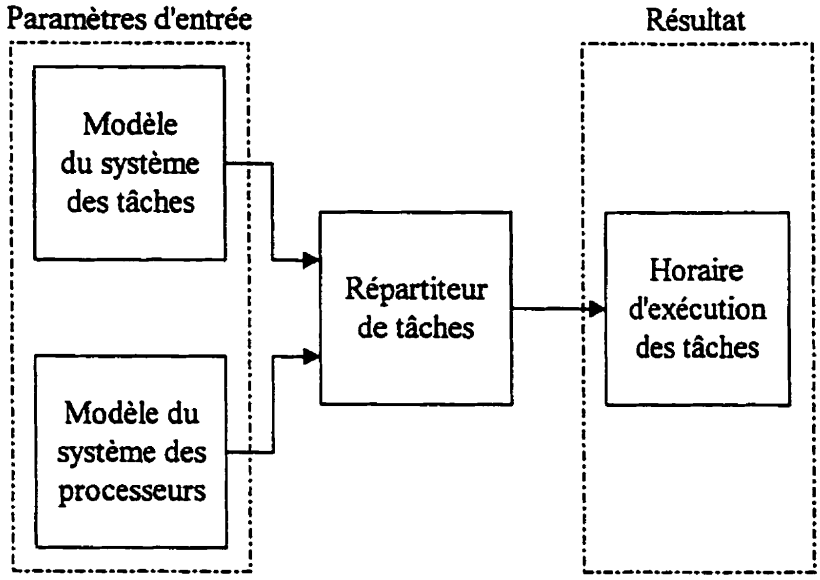


Figure 3.6 Schéma représentant le problème de la répartition des tâches.

Un horaire d'exécution d'un système de tâches $\mathcal{T} = (\mathcal{G}, \mathcal{L})$ dans un ordinateur parallèle $\mathcal{P}$ de m processeurs est une application R qui assigne chaque tâche de $\mathcal{G}$ à un processeur de l'ordinateur avec un temps de départ pour son exécution périodique. Formellement, $R : \mathcal{G} \rightarrow \mathcal{P} \times [0, \infty)$. Si $R(g) = (P_i, e)$ pour une tâche $g \in \mathcal{G}$, alors nous disons que la tâche g est sélectionnée pour être traitée par le processeur P_i et g débute son exécution au temps e.

À noter qu'il n'existe pas de tâches $u, v \in \mathcal{G}$ et $u \neq v$ tel que $R(u) = R(v)$. Puisque deux tâches différentes ne peuvent s'exécuter dans le même processeur en même temps. S'il existe une relation de préséance parmi les tâches $u \prec v$ alors $R(u) = (P_i, e_1)$ et $R(v) = (P_j, e_2)$ signifie nécessairement que $e_1 < e_2$.

Enfin, un horaire d'exécution est un **horaire faisable** s'il préserve toutes les relations de préséance (si elles existent). Un horaire d'exécution est un **horaire acceptable** s'il préserve toutes les relations de préséance (si elles existent) et satisfait toutes les contraintes temporelles et spatiales imposées.

Nous pouvons utiliser les graphes disjonctifs (split graphs) [GOL80] pour modéliser la répartition des tâches [ALI93]. L'avantage des graphes disjonctifs dans ce contexte est double : *i*) la représentation est intuitive; *ii*) les manipulations sont aisées. Les détails sur l'utilisation des graphes disjonctifs sont donnés dans l'annexe C de cette thèse. Nous allons reprendre les définitions utiles de cette annexe pour aider à la présentation du modèle de la répartition des tâches. Dans les définitions qui suivent, les numéros entre parenthèses sont les numéros de définition correspondants dans l'annexe C.

Un graphe disjonctif est un graphe non orienté. Ce type de graphe est défini par un ensemble S de sommets et un ensemble A d'arêtes, $H = (S, A)$. Pour nos besoins nous ne considérons que les graphes simples. Un graphe simple est celui qui ne possède pas de boucles autour d'un seul sommet ni d'arêtes parallèles partageant les mêmes extrémités.

Définitions 3.3.5.1 (C1.0.1) Un **graphe** est **complet** quand l'ensemble de ses sommets contient exactement une arêtes par pair de sommets.

■

Définitions 3.3.5.2 (C1.0.2) Un **ensemble de sommets** $S' \subseteq S$ d'un graphe $H = (S, A)$ est **indépendant** si ses éléments sont mutuellement disjoints. C'est-à-dire, qu'il n'y a pas d'arêtes reliant les sommets de S'.

■

Définitions 3.3.5.3 (C1.0.3) Un **ensemble de sommets** $S' \subseteq S$ d'un graphe $H = (S, A)$ est **complet** si chaque pair de sommets dans S' est relié par une arête $a \in A$. Ces arêtes sont appelé arêtes complètes.

■

Définitions 3.3.5.4 (C1.0.4) Un **graphe** $H = (S, A)$ est **disjonctif** s'il existe une séparation $S = S_1 \cup S_2$ de ses sommets en un ensemble indépendant S_1 et un ensemble complet S_2.

■

Il est à remarquer que l'ensemble des sommets S_2 forme un sous-graphe complet qu'on appelle communément une clique de H [BER83]. Dans un graphe disjonctif, il n'y a pas de contraintes sur les arêtes qui existent entre les ensembles S_1 et S_2.

Définitions 3.3.5.5 (C1.0.5) Un **graphe disjonctif** est **complet** s'il existe une arête entre chaque sommet dans S_1 et chaque sommet dans S_2. On nomme arête de liaison, une arête reliant un sommet dans S_1 à un sommet dans S_2.

■

Dans un graphe disjonctif complet, il y a exactement $nn_2 - n_2(n_2 - 1)/2$ arêtes où $n = n_1 + n_2$ avec $n_1 = |S_1|$ et $n_2 = |S_2|$. La figure 3.7 montre un graphe disjonctif avec $|S_1| = 3$ et $|S_2| = 4$. Dans cette figure, il existe 12 (n_1n_2) arêtes de liaisons et 6 ($n_2(n_2 - 1)/2$) arêtes complètes.

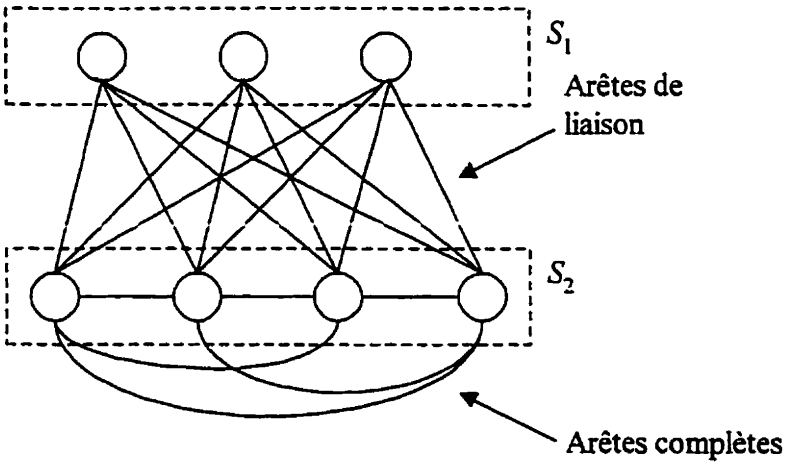


Figure 3.7 Graphe disjonctif.

Le graphe disjonctif représente un problème de la répartition des tâches. Puisque l'on peut représenter l'ensemble des processeurs par les sommets de l'ensemble indépendant S_1 du graphe. L'ensemble complet S_2 du graphe peut représenter l'ensemble des tâches à répartir. La figure 3.8 est une représentation de cette organisation.

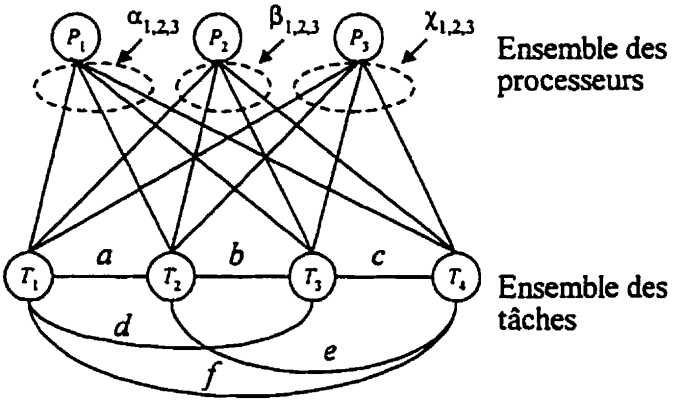


Figure 3.8 Graphe disjonctif représentant le problème de répartition des tâches.

Définitions 3.3.5.6 (C2.0.1) Une **arête complète**, c'est-à-dire une arête reliant deux sommets dans S_2, représente la communication entre deux tâches. Le coût de communication est une valeur $c \in \mathcal{N}$ indiquée sur l'arête.

■

Définitions 3.3.5.7 (C2.0.2) Une **arête de liaison**, c'est-à-dire une arête reliant un sommet dans S_1 à un sommet dans S_2, représente l'exécution d'une tâche (dans S_2) dans un processeur (dans S_1). Le coût de l'exécution est une valeur $c \in \mathcal{N}^+$ indiquée sur l'arête.

■

Les étiquettes $\{a, b, \ldots, f\}$ des arêtes complètes de la figure 3.8 représentent les coûts de communication entre les tâches. Lorsqu'une valeur est nulle, cela signifie qu'il n'y a pas de communication entre les tâches qui sont reliées par l'arête. Les étiquettes $\{\alpha_1, \ldots, \alpha_4, \beta_1, \ldots, \beta_4, \chi_1, \ldots, \chi_4\}$ des arêtes de liaisons de cette même figure représentent les coûts d'exécution des tâches sur les processeurs P_1 à P_3.

Définitions 3.3.5.8 (C2.0.3) Une **clique** d'un graphe $H = (S, V)$ est un sous-ensemble $S' \subseteq S$ dont chaque pair de sommets de S' est relié par une arête $a \in A$.

■

La répartition des tâches dans ce modèle correspond à une séparation du graphe disjonctif en cliques disjointes par une relation $R : S_2 \rightarrow S_1$. Chaque clique du graphe H consiste en un sommet $P \in S_1$ et un ensemble de sommets $S'' \subseteq S_2$. L'ensemble S'' représente les tâches assignées au processeur représenté par le sommet P. Noter que nous intéressons surtout, au problème où le nombre de cliques disjointes obtenues doit être de cardinalité égale à $|S_1|$, c'est-à-dire le nombre de cliques disjointes est égal au nombre de processeurs. Ce cas étant particulièrement approprié pour le problème de répartition des tâches.

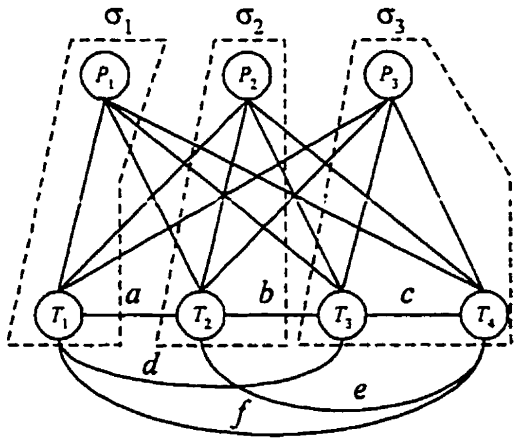


Figure 3.9 Répartition des tâches dans un graphe disjonctif.

La figure 3.9 montre une séparation possible du graphe disjonctif H en cliques σ_i, $1 \leq i \leq |S_1|$. Donc, en prenant $S_1 \equiv \mathcal{P}$ du modèle des processeurs et $S_2 \equiv \mathcal{G}$ du modèle des tâches nous obtenons, par les graphes disjonctifs, un modèle simplifié de notre problème de répartition des tâches. C'est à l'aide de ce modèle que nous allons analyser la nature et la complexité de notre problème.

3.4 Analyse du problème de répartition des tâches

Nous avons établi, dans la section 2.1 du chapitre 2, que le critère de performance le plus important pour le répartiteur des tâches est la satisfaction de toutes les relations de préséance (si elles existent) et de toutes les contraintes temporelles et spatiales des tâches. Dans la définition générale du système des tâches (définition 3.3.3.1), le couple $(\mathcal{G}, \mathcal{L})$ et l'application $\lambda : \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{N}$ décrits l'ensemble des relations qui existent dans le réseau électrique à simuler à un niveau élevé d'abstraction. $\mathcal{G}$ représente les postes et les lignes du réseau, $\mathcal{L}$ est une relation définie sur $\mathcal{G}$ donnant l'interconnexion des éléments. L'application λ sert à indiquer la quantité d'information échangée entre les éléments de $\mathcal{G}$. Le réseau des processeurs, quant à lui, est modélisé par un ensemble de

processeurs $\mathcal{P}$ et divers attributs du simulateur numérique. Les relations de préséance ne se trouvent pas dans ce niveau d'abstraction.

En ne considérant que cette définition du système des tâches, on peut se rendre compte qu'un problème de répartition des tâches S est un problème dans lequel nous devons trouver au moins une application

$$R : \mathcal{G} \rightarrow \mathcal{P} \tag{3.4.1}$$

qui vérifie $\forall(G_i, P_j) \in R$, $\sum_{G_i \in \mathcal{G}, P_j \in \mathcal{P}} E(G_i, P_j) + C(G_i) \leq D$. Le terme $E(G, P)$ est le temps d'exécution de G dans P. $C(G)$ et D sont respectivement le temps de communication exigé et l'échéancier de G. L'obtention d'une telle application pour le problème S n'est pas facile.

La situation se complique davantage si on considère les modèles plus raffinés donnés par les définitions 3.3.3.2 et 3.3.4.1. L'argument que l'on peut soumettre est le suivant : le modèle des regroupements (définition 3.3.3.2) accentue les contraintes du système en introduisant l'éventualité de l'utilisation d'un ensemble de ressources U par les tâches. Le modèle des processeurs (définition 3.3.4.1) admet l'existence d'un ensemble de ressources U' dans le réseau des processeurs. Si $U \subseteq U'$ et $U \notin \{\varnothing\}$ alors la contrainte de S devient $(\sum_{G_i \in \mathcal{G}, P_j \in \mathcal{P}} E(G_i, P_j) + C(G_i) \leq D) \wedge (u_i \subseteq r_j)$. Dans cette contrainte, u_i est l'ensemble des ressources exigées par les tâches de G_i, r_j est l'ensemble des ressources disponibles dans le proecsseur P_j.

En resserrant les contraintes du système, le problème devient plus difficile à résoudre. Par conséquent, l'étude d'un problème simplifié suffit pour bien comprendre la complexité des éléments en jeu. Un argument plus formel est donné à la fin de la section 3.4.1. En effet, nous allons mettre en évidence que le problème S, même dans sa forme simple, est un problème de complexité **NP-complet**.

3.4.1 Complexité du problème

Nous avons mentionné, à la section 2.1.2 du chapitre 2, que la plupart des problèmes impliquant un système multiprocesseur sont de nature **NP-complet**. D'une façon générale, un problème appartient à la classe **NP-complet** si sa résolution en un temps polynomial exige une machine de type non déterministe [MOR98, chapitre 7]. En d'autres termes, un problème **NP-complet** est un problème qui ne peut être résolu en un temps raisonnable par un ordinateur déterministe.

Bien entendu, dans ce contexte, la résolution d'un problème est basée sur des contraintes et critères de décisions ou d'optimisation. Ce sont ces contraintes et critères qui rendent le problème difficile à résoudre. Cependant, une caractéristique d'un problème **NP-complet** est que leur solution peut être vérifiée en un temps polynomial [COR93, chapitre 36]. Autrement dit, si une solution existe pour cette classe de problèmes, nous pouvons vérifier la validité de cette solution. Une explication intuitive de cette caractéristique est comme suit : il est plus facile de vérifier une solution donnée que de la trouver. Un problème appartient à la classe **P** si sa solution peut être obtenue en un temps polynomial. Un problème dont la solution est vérifiable en un temps polynomial est un problème qui appartient à la classe de complexité **NP**. Ainsi, un problème **NP-complet** est aussi un problème **NP** (définition B4.3.3, Annexe B).

La nature **NP-complet** de notre problème de répartition des tâches est le sujet de cette section. Les notions fondamentales de la théorie de complexité sont énoncées dans l'annexe B de cette thèse. Ces notions théoriques sont utiles pour saisir la cohérence de notre démarche. L'annexe B renferme donc les données nécessaires pour la caractérisation de notre problème de répartition des tâches.

La méthode utilisée dans la démonstration **NP-complet** de notre problème est basée sur le travail monumental de Garey et Johnson [GAR78] où des centaines de problèmes **NP-complet** ont été répertoriés. En utilisant une technique de réduction

polynomiale (section B4.3, annexe B), nous pouvons montrer la nature **NP-compet** de notre problème de répartition des tâches. Cette technique transforme un problème connu Ψ appartenant à la classe **NP-complet** en une instance de notre problème initial Φ. Si cette transformation de Ψ en Φ est réalisable en un temps polynomial alors Φ est aussi dans la classe **NP-compet** [COR93, chapitre 36].

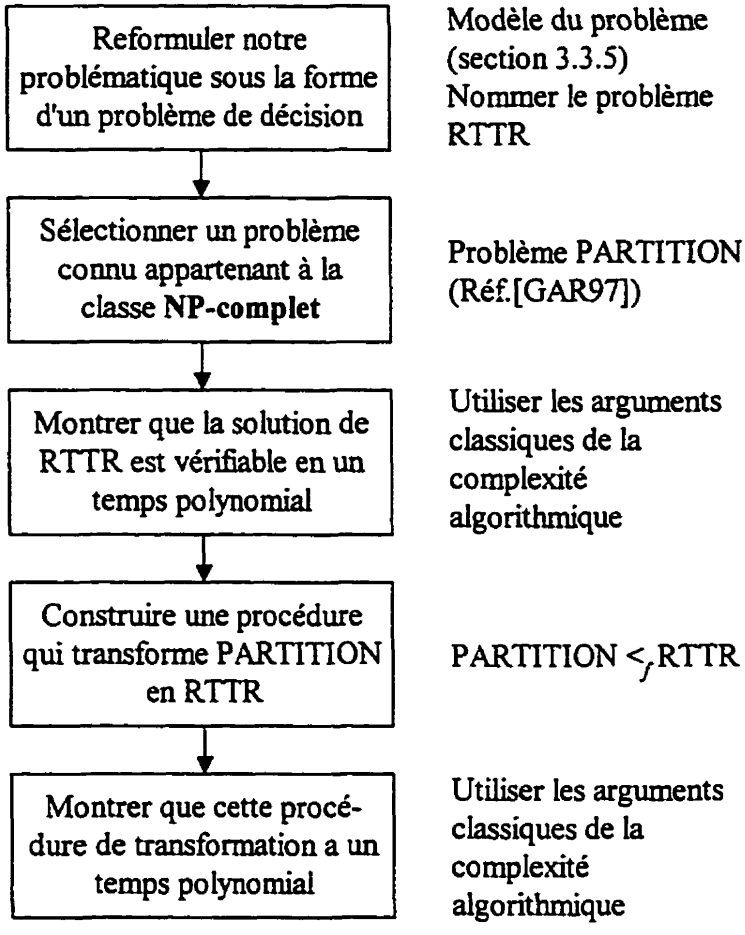


Figure 3.10 Étapes de la démonstration **NP-complet**.

La démonstration complète de la nature **NP-complet** de notre problématique est présentée dans l'annexe C. Cette démonstration utilise le modèle de la répartition des tâches de la section 3.3.5. La procédure de réduction polynomiale est basée sur le problème général impliquant la séparation d'un graphe en m sous-graphe de K sommets.

Ce problème est répertorié dans [GAR78] comme étant **NP-complet**. Notre démonstration consiste à construire une procédure qui transforme ce problème de séparation en une instance de notre modèle de la répartition des tâches. L'exécution de la procédure de transformation peut être réalisée en un temps polynomial. Par conséquent, la réduction polynomiale est possible. Donc, la nature **NP-complet** de notre problématique est établie en démontrant qu'une réduction polynomiale est réalisable.

Nous présentons dans la figure 3.10, les grandes lignes de cette réduction polynomiale et les résultats obtenus . Les détails techniques de la démonstration sont présentés dans l'annexe C de cette thèse. Débutons par une explication des étapes de la démonstration.

La forme décisionnelle de notre problématique est baptisée RTTR (Répartition des Tâches Temps Réel). Cette formulation est tirée directement du modèle du graphe disjonctif de la section 3.3.5.

RTTR = $\{<\mathcal{T}, \mathcal{P}, W, d> : \mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ ensemble des tâches, $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ ensemble des processeurs, $W = [w_{ij}]$ matrice de communication et $d \geq 0$ un entier. Existe-t-elle une répartition des tâches $\mathcal{T}$ dans les processeurs $\mathcal{P}$ de sorte que $C_E + C_C \leq d$ où $C_E = \sum x_{ij}$ est le coût d'exécution de la tâche T_i dans le processeur P_j et $C_C = \sum w_{ij}$ est le coût de communication entre les tâches T_i et T_j si elles sont assignées à des processeurs différents$\}$. (3.4.2)

Pour le problème connu qui sert à la réduction, il est tiré de [GAR79] où sa nature **NP-complet** a été prouvée.

PARTITION = $\{<G', c', l', K, J> : G' = (V', E')$ un graphe simple, c': $V' \rightarrow 1$ une application indiquant le poids des sommets de V', $l' : E' \rightarrow \mathcal{N}$ une application qui donne

la valeur des arêtes, K et J entiers positifs. Existe-t-elle une séparation de V' en m ensembles disjoints $V_1', V_2', \ldots, V_m'$ telle que $\sum_{v \in V_i'} c'(v) \leq K$ pour $1 \leq i \leq m$. De plus, si $E'' \subseteq E'$ l'ensemble des arêtes dont les deux extrémités sont dans deux ensembles V_i' différents alors $\sum_{e \in E''} l'(e) \leq J$ }. (3.4.3)

Ce problème a été sélectionné parce que sa structure est proche de RTTR mais dont la nature est beaucoup plus général. Le problème PARTITION consiste à séparer un graphe quelconque en m ensembles disjoints de sommets. Chaque ensemble doit contenir au plus K sommets. Les arêtes qui relient deux ensembles disjoints ne doivent pas avoir un poids total supérieur à J. PARTITION est un problème de la classe **NP-complet** [GAR79], [EVA79, chapitre 10].

Pour démontrer que RTTR est **NP-complet**, nous devons effectuer une réduction de PARTITION en RTTR. La notation utilisée pour indiquer cette réduction est PARTITION $\leq_f$ RTTR où $\leq_f$ est l'opérateur de réduction et f est la fonction ou procédure de réduction à construire. Pour faciliter cette transformation nous utiliserons un troisième problème qui agira comme problème intermédiaire entre PARTITION et RTTR. Nous utiliserons un problème qui sera similaire à PARTITION et à RTTR. Ce problème est la recherche de cliques maximales dans un graphe disjonctif. Nous l'appellerons MAXCLIQUE. La définition de MAXCLIQUE est donnée dans (3.4.4).

MAXCLIQUE = {$<G, l, B>$: $G = (V_1 \cup V_2, E)$ un graphe disjonctif complet avec V_1 un ensemble de sommets indépendants et V_2 un ensemble de sommets complets, $l : E \rightarrow \mathcal{N}$ une application qui donne la valeur des arêtes, $B \geq 0$ un entier. Existe-t-elle une séparation de V en m cliques disjointes σ_i de sorte que $\sum_{e \in \sigma_i} l(e) \geq B$, pour $1 \leq i \leq m$ où $e \in E$ et $m = |V_1|$ }. (3.4.4)

Pour le problème MAXCLIQUE, le but est de séparer un graphe disjonctif en cliques dont la somme des poids des arêtes est égale ou supérieure à un entier positif. Le nombre de cliques à obtenir est égal au nombre de sommets dans l'ensemble des sommets indépendants du graphe.

À cause de la propriété transitive de l'opérateur $\leq_f$ (théorème B4.3.1, Annexe B), nous pouvons utiliser le problème MAXCLIQUE comme tremplin pour montrer que RTTR $\in$ **NP-complet**, puisque

$$\left.\begin{array}{l}\text{PARTITION} \leq_f \text{MAXCLIQUE} \\ \text{MAXCLIQUE} \leq_f \text{RTTR},\end{array}\right\} \Rightarrow \text{PARTITION} \leq_f \text{RTTR}. \qquad (3.4.5)$$

Nous procédons en deux étapes. D'abord, nous transformons PARTITION en MAXCLIQUE, démontrant ainsi que MAXCLIQUE $\in$ **NP-complet**. Ensuite nous transformons MAXCLIQUE en RTTR pour prouver que RTTR aussi appartient à la classe des problèmes **NP-complet**.

La transformation de PARTITION $\leq_f$ MAXCLIQUE consiste à ajouter un certain nombre d'arêtes de valeur nulle dans le graphe du problème PARTITION pour le changer en un graphe complet. Les arêtes ajoutées ont une valeur nulle pour ne pas changer le système des valeurs du problème original. Nous ajoutons également un certain nombre de sommets au problème PARTITON. Ces sommets ajoutés sont reliés au reste du graphe par des arêtes. Ces arêtes sont arrangés de sorte qu'en choisissant n'importe quel sous-ensemble de K sommets dans le graphe, il existe au moins K arêtes de valeur 1. Si le graphe de PARTITION contient m sommets, cet arrangement est réalisé en ajoutant un nombre de sommets égal à la valeur de K-combinaison dans un ensemble de m éléments. Il est possible de démontrer que ces manipulations transforment le problème PARTITION en une instance de MAXCLIQUE en un temps polynomial (théorème C3.1.2, Annexe C).

Théorème 3.4.1.1 *Le problème MAXCLIQUE ∈ NP-complet.*

Démonstration Selon le théorème C3.1.1, le problème MAXCLIQUE ∈ **NP**. Par le théorème C3.1.2, nous savons que le problème PARTITION est réductible, dans un temps polynomial, en une instance de MAXCLIQUE. Alors, selon le théorème B3.4.1, MAXCLIQUE ∈ **NP-complet.**

□

En utilisant la même démarche, nous construisons une transformation du problème MAXCLIQUE en une instance de RTTR. L'idée de base consiste à faire correspondre l'ensemble des tâches $\mathcal{T}$ à l'ensemble des sommets complets de MAXCLIQUE. De même pour l'ensemble des processeurs $\mathcal{P}$ et l'ensemble des sommets indépendants de MAXCLIQUE. Le coût de communication w_{ij} de RTTR est associé au poids des arêtes complètes. Le coût d'exécution des tâches x_{ij} est associé au poids des arêtes de liaisons du graphe disjonctif de MAXCLIQUE. Nous pouvons démontrer que ces manipulations transforment le problème MAXCLIQUE en une instance de RTTR en un temps polynomial (théorème C3.2.2, Annexe C).

Théorème 3.4.1.2 *Le problème RTTR ∈ NP-complet.*

Démonstration Selon le théorème C3.2.1, le problème RTTR ∈ **NP**. Par le théorème C3.2.2, nous savons que le problème MAXCLIQUE est réductible, dans un temps polynomial, en une instance de RTTR. Alors, selon le théorème B3.4.1, RTTR ∈ **NP-complet.**

□

Enfin, le problème RTTR est une version simplifiée de notre problématique de répartition des tâches.. Les contraintes de préséance et l'utilisation des ressources (contraintes spatiales) ne sont pas pris en compte dans RTTR. Nous appellerons notre

problématique initial RTTR-($\prec$, u) où $\prec$ représente les relations de préséance et u représente le taux d'utilisation des ressources par les tâches. Notre problématique initial est donc une généralisation de RTTR. Nous pouvons obtenir une instance de RTTR à partir de RTTR-($\prec$, u) en appliquant les restrictions suivantes : *i*) l'ensemble des relations de préséance est vide, $\prec = \{\varnothing\}$; *ii*) le taux d'utilisation des ressources par les tâches est nul, $u = 0$. Puisque RTTR appartient à la classe **NP-complet** alors RTTR-($\prec$, u) l'est également.

Lemme 3.4.1.1 Si un problème ζ' est obtenu par application de restrictions à un problème ζ, alors ζ est réductible à ζ' à condition que ces restrictions soient applicables en un temps polynomial.

Démonstration La transformation de ζ en une instance de ζ' consiste à construire une fonction de réduction simulant l'application des restrictions sur ζ. La réalisation des restrictions est effectuée en un temps polynomial. Donc, la réduction est également de complexité polynomiale.

□

Théorème 3.4.1.3 *Le problème RTTR-($\prec$, u) $\in$ **NP-complet**.*

Démonstration Puisque le problème RTTR peut être obtenu par restrictions sur RTTR-($\prec$, u) alors par le lemme 3.4.1.1, RTTR-($\prec$, u) $\leq_f$ RTTR. La condition de complexité polynomiale dans les restrictions est simple car RTTR = RTTR-($\prec$, u) lorsque $\prec = \{\varnothing\}$ et $u = 0$. On peut utiliser une technique similaire à celle du théorème C3.2.1 pour montrer que RTTR-($\prec$, u) $\in$ **NP**. Enfin, par le théorème B3.4.1 RTTR $\in$ **NP-complet** $\Rightarrow$ RTTR-($\prec$, u) $\in$ **NP-complet**.

□

3.4.2 Conséquences du résultat de l'analyse de la complexité

Un problème peut possèder un très grande nombre d'instances. Le fait qu'un problème appartient à la classe de complexité ne signifie pas nécessairement que toutes les instances du problème soient aussi difficile les unes que les autres à résoudre [MOR98, chapitre 6]. Par exemple, la répartition d'une tâche sur un processeur est facile à résoudre. Peu importe le nombre et le type de contraintes, on peut déduire une solution en un temps raisonnable. La solution peut être nulle ou non nulle. Il s'agit là d'une instance triviale du problème.

En fait, RTTR-($\prec$, u) $\in$ **NP-complet** signifie qu'il existe, parmi toutes les instances de RTTR-($\prec$, u), un ensemble d'instances qui ne peuvent pas être solutionnées en un temps polynomial. Ce sont précisément ces instances du problème qui le rendent complexe.

En théorie, si nous pouvons répertorier toutes les solutions obtenues de toutes les instances solutionnées de RTTR-($\prec$, u), le problème devient simple à résoudre. Il suffit de placer toutes ces solutions dans un tableau. La résolution d'une instance du problème est transformée en une fouille dans le tableau de solutions. La solution d'une instance du problème existe si elle est dans le tableau. Autrement, la solution n'existe pas. Dans ce cas idéal, le problème RTTR-($\prec$, u) est un problème décidable et sa complexité est linéaire (complexité de classe **P**). Évidemment, en pratique, on ne peut pas procéder de cette façon puisque certaines instances du problème ont une complexité spatiale ou temporelle super-polynomiale (c'est-à-dire, de complexité quadratique, logarithmique, exponentielle, etc.). Répertorier toutes les instances solutionnées exige une énumération exhaustive, ce qui interdit la faisabilité d'un tel tableau de solutions.

La difficulté majeure est donc située dans cet ensemble d'instances qui présente une complexité super-polynomiale. Il est possible de surmonter cette complexité en utilisant une machine non déterministe [MOR98, chapitre 7]. Le non déterminisme peut

aider à la solution de ce type de problèmes puisqu'il présente un élément de choix dans son fonctionnement (section B2, Annexe B).

Pour le problème de répartition des tâches, une machine non déterministe peut simplement choisir une séquence d'affectations des tâches. Il ne nous reste qu'à vérifier si la séquence obtenue par la machine est effectivement une solution. Le choix d'une séquence d'affections demande au plus une complexité linéaire proportionnelle au nombre de tâches à répartir. Même s'il y a un grand nombre de séquences d'affections possibles, une machine non déterministe n'a pas à effectuer une énumération exhaustive pour trouver une solution. Le temps nécessaire pour vérifier la validité de la solution d'une répartition de tâches est au plus de complexité polynomiale. Si le temps de vérification d'une solution est de complexité polynomiale, la machine non déterministe peut générer une solution beaucoup plus rapidement qu'un algorithme déterministe. Dans ce contexte, une machine non déterministe dotée d'une capacité de sélection peut surmonter la complexité super-polynomiale des problèmes.

Il est à noter que le non déterminisme ne peut transformer un problème dans la classe **NP** en un problème de classe **P**. En effet, le non déterminisme implique une certaine clairvoyance dans le choix des étapes de solution et non dans la caractérisation complète d'un problème. Une machine non déterministe peut résoudre un problème **NP-complet** que si ce problème présente une solution. Elle ne peut pas nous dire si une solution existe pour un problème donné en un temps polynomial. Cette asymétrie entre l'obtention d'une solution et la caractérisation d'un problème est un point important. Parce que, même avec une machine non déterministe, on ne peut décider facilement si un problème possède une solution ou non [VAL85].

3.4.3 Proposition d'une solution

La construction d'une machine non déterministe pour la répartition des tâches n'est pas vraiment faisable. Les ordinateurs numériques modernes sont, par leur conception, parfaitement déterministes. Cependant, il est possible de simuler certains aspects de ce non déterminisme par ordinateur. Il s'agit de l'aspect du choix dans la sélection d'une solution et de l'aspect vérification d'une solution.

Ces aspects d'une machine non déterministe ont été exploités dans des langages de programmation. Par exemple, le langage CHIP (Constraint Handling In Prolog) est une extension du langage Prolog (Programming Logic) utilisé pour le traitement des problèmes sous la forme de satisfaction des contraintes (constraints satisfaction problems). Le langage CHIP est muni d'un moteur d'inférence et d'un ensemble de prédicats qui simulent le comportement du non déterminisme par l'utilisation d'un ensemble d'heuristiques de sélection [HEN89, chapitre 4]. Une de ces heuristiques porte le nom de MCVF (Most Constrained Variable First) et elle consiste à choisir d'abord la variable la plus contraignante du problème pour le développement d'une solution. Si le problème ne peut être satisfait (solutionné), une technique de retour-arrière est appliquée pour recommencer la sélection d'une autre variable. Le mécanisme de sélection-vérification est donc réalisé par des heuristiques (MCVF) et une technique de gestion d'erreur (retour-arrière).

Le langage LISP possède également une extension qui simule le comportement d'une machine non déterministe [SRC97]. Cette extension du langage LISP repose sur le principe utilisé dans Prolog. Le choix d'une solution est réalisé par une fouille en profondeur d'abord et la gestion d'erreur est réalisée par la technique de retour-arrière.

La simulation du non déterminisme n'est pas limitée qu'aux langages de programmation. La célèbre méthode Monte Carlo repose également sur le principe de sélection-vérification [PRE92, chapitre 4]. La réalisation plus explicite de ce principe est

illustrée par les méthodes de fouille heuristique. Notamment, dans la méthode Tabou, dans les algorithmes génétiques et dans la méthode A* [GLO93], [NEU90], [PEA90].

Dans la méthode de fouille Tabou, le choix d'une solution est construite itérativement en utilisant des heuristiques de sélection [GLO93]. Cette méthode maintient en mémoire une liste de choix tabous pour éviter le bouclage dans l'espace de fouille. Cette liste sert aussi à indiquer les choix permis pour les itérations en cours. La taille et la durée de maintenance de la liste tabou sont variables. La taille de la liste représente la grandeur du voisinage de la fouille en cours et la durée de maintenance représente la mémoire à court terme de la méthode. La liste tabou est reconstruite si la solution n'est pas obtenue après un nombre d'itérations prédéterminées. Il s'agit là d'une façon originale de la méthode Tabou pour échapper au piège des minimums locaux [GLO93]. Ainsi, la fouille peut parfois sélectionner un chemin intermédiaire qui n'est pas minimal. Donc, le contrôle sélection-vérification est directement réalisé par la liste tabou de cette méthode.

Les algorithmes génétiques diffèrent de la méthode Tabou à plusieurs points de vue. D'abord, le non déterminisme est encore plus explicite. Le point de départ des algorithmes génétiques est un ensemble de chemins partiels sélectionnés d'une manière aléatoire. Le chemin d'une solution est obtenu en répétant un grand nombre fois les procédures de sélection, reproduction et de mutation [NEU90]. La procédure de sélection consiste à choisir un chemin partiel selon la valeur d'une fonction de sélection. Cette dernière peut être exacte ou heuristique. La procédure de reproduction consiste à joindre deux chemins partiels pour former un nouveau chemin que l'on espère pouvoir nous amener à la solution. La procédure de mutation consiste à changer la composition des chemins d'une manière aléatoire [NEU90]. Ce sont ces étapes qui permettent aux algorithmes génétiques d'éviter les minimums locaux.

Enfin, la méthode A^* est une méthode de fouille dans un espace d'états utilisant une fonction d'évaluation heuristique de type

$$f(n) = r(g(n), h(n)), \qquad (3.4.3.1)$$

où n est un état, r est une relation quelconque, $g(n)$ est le coût du chemin partant de l'état initial jusqu'à l'état n et $h(n)$ est une estimation du coût du chemin minimal partant de n jusqu'à un état but. La simulation du non déterminisme est réalisée par des heuristiques de sélection. Le mécanisme de vérification est réalisé par deux listes OUVERT et FERMÉ. Les états générés mais non encore visités sont placés dans la liste OUVERT. Les états générés et visités sont placés dans la liste FERMÉ. Si la fouille rencontre un cul-de-sac, on peut retirer de la liste FERMÉ un état potentiel pour recommencer la fouille ailleurs dans l'espace d'état. [PEA90]. D'une manière implicite, la valeur de $g(n)$ sert à contourner le problème des minimums locaux.

Toutes ces méthodes de fouille présentent un certain comportement non déterministe. Les règles de sélection n'indiquent pas précisément le chemin à prendre pour obtenir une solution. Elles donnent seulement les caractéristiques désirées des étapes intermédiaires menant vers la résolution du problème. La puissance de ces méthodes est dans la stratégie de contrôle et dans les heuristiques de sélection et de génération des étapes intermédiaires.

Dans la pratique, les méthodes de fouille heuristique ont été appliquées avec succès dans des problèmes de planification de grande envergure. Le système PlanERS1 a été conçu pour la planification des missions d'observation pour le satellite ERS-1 (European Earth Ressource Observation Satellite) de l'agence spatiale européenne [FUC90]. De même pour le système OPTMUM-AIV [AAR94]. Ce dernier sert à planifier l'assemblage des véhicules spatiales pour le compte de l'agence spatiale européenne. Le concept de planification O-Plan, adopté par la compagnie HITACHI pour effectuer la planification à moyen terme et l'ordonnancement des opérations d'assemblage des produits électroniques [CUR91]. Toutes ces systèmes utilisent une ou

plusieurs méthodes de fouilles heuristiques pour accomplir leur travail. Il est donc justifié pour nous de porter une attention particulière à ces méthodes de fouilles heuristiques pour la solution de notre problème.

D'abord les langages de programmation Prolog, CHIP et LISP ne sont pas intéressants pour nous. La raison est d'ordre pratique. Le répartiteur de tâches est un composant du système d'exploitation d'un simulateur numérique (voir figure 2.1, chapitre 2). Il est très inconvénient d'avoir dans le noyau du système d'exploitation un module utilisant un langage déclaratif.

Pour ce qui a trait aux algorithmes génétiques, ils présentent des caractéristiques qui sont indésirables. La configuration initiale de ces algorithmes est obtenue par une collection aléatoire de solutions potentielles. Le non déterminisme de la configuration initiale rend ces algorithmes très difficiles à analyser. La taille de l'ensemble initiale des solutions potentielles n'est pas facile à déterminer mais influence directement la possibilité de solution. En effet, si la taille de l'ensemble initial est petite alors on risque d'élaguer les solutions du problème. Si la taille est grande alors le temps de traitement est grand. Enfin, les étapes sélection, reproduction et mutation rendent la convergence des algorithmes génétiques beaucoup plus lente que les autres méthodes heuristiques.

Les méthodes de fouille Tabou et A^* semblent plus convenables pour notre application. Leurs principes de fonctionnement sont simples et peuvent être programmés facilement en langage C ou C++. Cependant, la méthode Tabou est elle-même une heuristique et son comportement est difficile à caractériser par une analyse formelle. Pour ce qui a trait à la méthode A^*, son efficacité optimale a été démontrée dans le cas d'une fouille heuristique dans un espace d'états [DEC85]. Cette caractéristique optimale de A^* est établie dans le sens suivant : pour un espace d'états donné, aucune autre méthode de fouille heuristique ne peut développer moins d'états que A^* avant d'obtenir une solution optimale. Puisque l'on peut représenter l'espace d'états par un graphe, on

peut caractériser formellement le comportement de la méthode de fouille A^* par une étude des propriétés générales des graphes d'états.

D'un point de vue théorique, cette méthode est intéressante car il est possible de déduire de nombreux résultats en utilisant les connaissances de la théorie des graphes. D'un point de vue pratique, la formulation du problème de répartition des tâches par la représentation d'états est naturelle. Il n'y a pas d'encodage supplémentaire pour représenter le problème comme c'est le cas des systèmes de satisfaction des contraintes CSP (Constraints Satisfaction Problems).

Nous proposons l'utilisation d'une méthode de fouille basée sur la méthode A^* pour la solution du problème de répartition des tâches temps réel. La conception du répartiteur des tâches sera donc réalisée par une étude détaillée des propriétés de l'espace d'états. Le chapitre 4 présente les étapes de conception de cette méthode de répartition des tâches. Les modifications apportées à la méthode de base ainsi que les résultats théoriques obtenus seront également donnés dans le prochain chapitre.

Chapitre 4

Méthode heuristique de répartition des tâches temps réel

Nous avons vu dans les chapitres précédents qu'il est important de satisfaire les contraintes temporelles et spatiales dans les systèmes temps réel à échéancier rigide. Nous devons considérer les besoins en ressources et en communication des tâches exécutant en parallèle. Ces besoins de l'ensemble des tâches sont souvent représentés sous la forme d'un graphe. Toutes les tâches doivent pouvoir utiliser les processeurs et le réseau de communication de sorte que les contraintes temporelles et spatiales peuvent être satisfaites en tout temps.

Ce chapitre présente une nouvelle méthode hors ligne pour la répartition des tâches temps réel à échéancier rigide dont le problème est représenté par le modèle RTTR-($\prec$, u) défini dans le chapitre 3. Cette nouvelle méthode détermine un horaire d'exécution et effectue la répartition des tâches avant le lancement de celles-ci dans le système parallèle. La répartition des tâches est basée sur une méthode de fouille heuristique développée spécialement pour la solution du problème de répartition des tâches en mode hors ligne.

4.1 Méthodes de fouilles heuristiques

La solution des problèmes par la fouille est une technique très utilisée dans le domaine de la recherche opérationnelle et dans le domaine de l'intelligence artificielle [DEO82], [KOR88]. Les algorithmes classiques tels que la fouille en profondeur d'abord, la fouille en largeur d'abord et la fouille à coût uniforme ont été introduits pour

résoudre un nombre de problèmes pratiques [MAR84] [KAR93] [NIL80]. Il est possible de formuler tous ces algorithmes de fouille comme des techniques de fouille dans un espace d'états.

Il a été montré dans le chapitre 3 que le problème de la répartition des tâches RTTR-($\prec$, u) est un problème qui appartient à la classe **NP-complet**. Ce genre de problème peut être solutionné en un temps polynomial par des machines non déterministes. Malheureusement il n'existe pas, à l'heure actuelle, une implantation concrète de ces machines non déterministes. Donc, il peut s'avérer inutile de considérer la résolution de notre problème par une machine non déterministe. Or, les méthodes de fouille heuristique simulent justement certains aspects du comportement des machines non déterministes. Il est vrai que le comportement non déterministe simulé par les méthodes de fouille est très limité Après tout, la fouille est réalisée par des ordinateurs qui sont déterministes. Néanmoins, les méthodes de fouille heuristique présentent un aspect très intéressant, qui est l'incorporation de la notion du choix dans leur principe de fonctionnement. En effet, lors du déroulement d'une fouille heuristique, on doit choisir une transformation du graphe d'états, parmi un ensemble de transformations possibles.

Une machine de Turing non déterministe est caractérisée par l'existence d'une fonction de transition qui place la machine dans un état choisi parmi un ensemble d'états possibles (annexe B, définition B2.0.1). Une machine de Turing non déterministe peut résoudre un problème **NP** parce qu'elle peut *deviner* les bons états nécessaires à la solution du problème, parmi l'ensemble des états possibles, en un temps polynomial [MOR98]. Certes, les méthodes de fouille ne peuvent donner une performance équivalente car elles ne peuvent deviner une solution directement. Cependant, une méthode de fouille heuristique offre la possibilité de choisir une transformation qui convient aux buts recherchés. Ce choix est guidé par des heuristiques où l'on espère pouvoir conduire la fouille à une solution.

4.1.1 Représentation du problème dans la fouille heuristique

La condition nécessaire pour effectuer une fouille est une représentation adéquate du problème à solutionner [POH73]. En effet, si le problème ne peut être représenté convenablement, on ne peut appliquer la fouille heuristique pour le solutionner. D'une manière générale, un problème de fouille P est caractérisé par un 4-uplet $P = (E, s_i, M, B)$ où

- E est l'ensemble des états représentant l'espace d'états du problème;
- $s_i \in E$ est un état initial du problème;
- $M = \{m_1, m_2, \ldots, m_x\}$ est l'ensemble des opérateurs de transformation;
- $B = \{b_1, b_2, \ldots, b_y\}$ est l'ensemble des états but du problème.

(4.1.1.1)

L'ensemble M représente les opérateurs qui transforment le problème d'un état donné en un autre état. L'application successive de ces opérateurs doit nous conduire à un des états but de l'ensemble B. L'ensemble des opérations de transformation est une solution possible du problème. Évidemment si $B \not\subset E$ alors une telle solution ne peut exister. Dans ce cas, la fouille n'est pas applicable au problème. En pratique, il est très difficile de connaître a priori l'inclusion de B dans E. Puisque cette connaissance revient à décider un problème. Dans ce contexte, même une machine non déterministe ne peut faire mieux qu'une machine déterministe (section 3.4.2, chapitre 3). La méthode de fouille heuristique doit donc inclure un mécanisme de vérification pour contrôler les situations où $B \not\subset E$.

Dans notre problème de répartition des tâches temps réel, chaque état dans E est une décision d'assignation d'une tâche i à un processeur j sous forme d'un couple noté (T_i, P_j). L'état initial du problème est une assignation vide. L'ensemble des opérateurs de transformation M représente l'ensemble des règles heuristiques utilisées pour la génération de nouvelles assignations (T_i, P_j). On appellera M l'ensemble des *règles de génération*. Une répartition complète des tâches dans le réseau des processeurs est

réalisée lorsque la fouille a atteint l'un des buts de l'ensemble des états but B. Autrement dit, un horaire complet d'exécution des tâches est obtenu. Donc, le problème de répartition des tâches temps réel peut être représenté par un problème de fouille heuristique caractérisé par un 4-uplet $P_{rt} = (E, s, M, B)$ où

- E est l'ensemble des états représentant toutes les assignations (T_i, P_j) du problème;
- $s \in E$ est une assignation tâche-processeur vide;
- $M = \{m_1, m_2, \ldots, m_x\}$ est l'ensemble des règles heuristiques de génération des assignations candidates;
- $B = \{b_1, b_2, \ldots, b_y\}$ est l'ensemble des assignations finales du problème.

(4.1.1.2)

Pour faciliter la lecture de ce chapitre, nous considérerons désormais un état du graphe d'états comme synonyme d'une assignation dans un horaire d'exécution. Nous construisons un horaire d'exécution des tâches, en partant d'une assignation vide s, par la génération successive de nouvelles assignations (T_i, P_j). Puisque pour chaque état du problème il peut exister un grand nombre d'assignations possibles, les règles de génération M sont utilisées pour limiter ces assignations en un sous-ensemble utile. Les règles de génération doivent générer que les assignations (T_i, P_j) qui satisfont les contraintes temporelles et spatiales imposées. Si un tâche T_i est forcée à être assignée au processeur P_m (contrainte spatiale) alors il est inutile de générer des assignations (T_i, P_j) pour $j \neq m$. De même pour les contraintes temporelles. Si à l'état n, le processeur P_j est déjà utilisé à sa pleine capacité alors les assignations générées par M pour cet état ne doivent pas contenir le processeur P_j (contrainte temporelle).

En dépit de l'application des règles de génération, il n'est pas garanti que les assignations générées soient uniques. C'est pourquoi que les méthodes de fouille sont munies d'une fonction de sélection. Cette fonction de sélection a pour rôle de choisir une seule assignation (T_i, P_j) parmi toutes celles générées pour un état du problème donné. Le processus de génération et de sélection des assignations continuera jusqu'à

l'assignation de la dernière tâche du programme parallèle. Donc, la fouille rencontre un état but dès que la dernière tâche est assignée à un processeur. En retraçant toutes les assignations (T_i, P_j) effectuées depuis l'état initial s jusqu'à l'état final rencontré, on peut produire un horaire complet d'exécution des tâches.

Un horaire d'exécution est un ensemble ordonné de couples $\mathcal{H} = \{((T_i, P_j)_1, \tau_1), ((T_i, P_j)_2, \tau_2), \ldots, ((T_i, P_j)_n, \tau_n)\}$ où

- $(T_i, P_j)_k$ est l'assignation de la tâche T_i dans le processeur P_j pour la k^e entrée de l'horaire d'exécution;
- τ_k est le temps de départ de la tâche contenue dans la k^e entrée de l'horaire d'exécution.

(4.1.1.3)

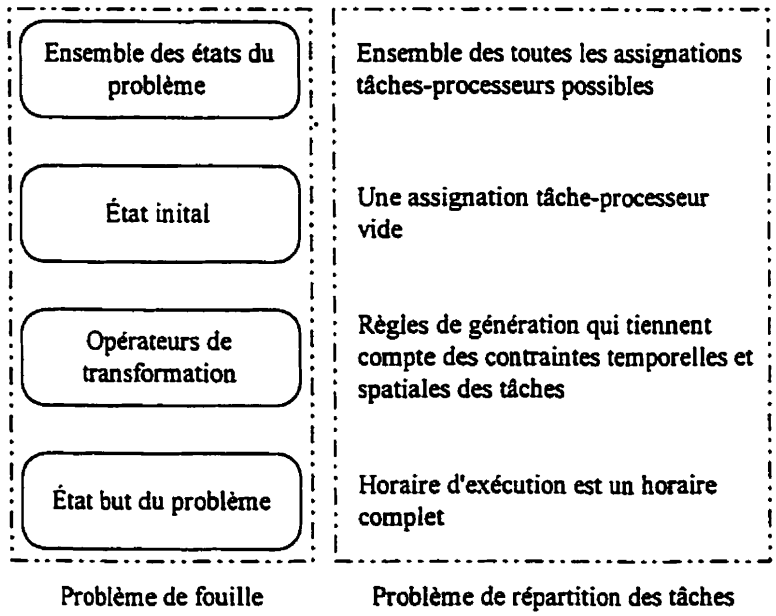


Figure 4.1 Représentation du problème par la fouille heuristique.

Un horaire d'exécution $\mathcal{H}$ est un *horaire acceptable* si toutes les assignations (T_i, P_j) contenues dans l'horaire satisfont les contraintes temporelles et spatiales imposées. De

plus, lorsque $|\mathcal{H}|$ est égale au nombre de tâches dans le système, l'horaire est un horaire acceptable complet ou simplement horaire complet lorsque le contexte est clair. L'objectif de la répartition des tâches par la fouille heuristique est donc de produire un horaire complet en utilisant le modèle du système des tâches et le modèle du système des processeurs développés au chapitre 3. Cet horaire complet est un horaire dans lequel toutes les assignations des tâches satisfont les contraintes temporelles et spatiales en tout temps. Enfin, la figure 4.1 est un schéma résumant notre formulation du problème de répartition des tâches dans le domaine des problèmes de fouille.

4.2 Fouille heuristique pour la répartition des tâches

L'ensemble des actions possibles et disponibles à une méthode de fouille est donné par les opérateurs de transformation. Dans ce sens, un opérateur de transformation est une action qui fait passer la méthode de fouille d'un état x à un autre état y. L'ensemble des opérateurs de transformation et l'état initial forment un espace d'états. Ainsi, un chemin dans l'espace d'états est une séquence d'actions qui conduit la méthode de fouille vers un des états but. Cette formalisation de la fouille heuristique est basée sur les résultats de l'annexe D.

4.2.1 Conditions d'applicabilité

On représente par un graphe d'états l'ensemble des états du problème. Un graphe d'états est un graphe orienté. La section D1 de l'annexe D donne les définitions concernant les graphes d'états. Dans notre problème de la répartition des tâches, l'ensemble des assignations (T_i, P_j) peuvent avoir une cardinalité très grande mais elle est bornée. En effet, il existe m^n assignations possibles pour un système à m processeurs et à n tâches. Évidemment, on ne peut pas représenter toutes ces assignations explicitement dans son graphe d'états. Autrement dit, la méthode de fouille pour la

répartition des tâches doit pouvoir explorer le graphe d'états du problème sans recourir à une énumération explicite des assignations.

Un chemin dans un graphe d'états est une séquence d'actions qui conduit la méthode de fouille vers un état but. Dans la répartition des tâches, ce chemin est une séquence d'assignations qui forment un horaire complet acceptable du système des tâches. Le graphe d'états qui représente ce problème est un graphe fini puisque le nombre maximal d'assignations est m^n. Comme le graphe d'états du problème de répartition des tâches est un graphe d'états fini, il est également *borné sur L* où *L* est la longueur généralisée applicable dans l'espace d'états (théorème D3.0.4). En utilisant la logique du théorème D3.0.6, on peut déduire le théorème suivant.

Théorème 4.2.1 *Il existe une séquence d'assignations des tâches partant de l'horaire vide jusqu'à un horaire complet si un horaire complet existe pour le problème.*

Démonstration Le théorème D3.0.6 indique trois conditions pour qu'une telle séquence d'assignations puisse exister: *i*) la finitude du graphe d'états; *ii*) le graphe d'états est borné sur *L*; *iii*) le graphe d'états est accessible. Nous savons que le nombre d'assignations est borné donc le graphe est fini. D'après le théorème D3.0.4, un graphe d'états fini est borné sur *L*. Donc, cette deuxième condition est remplie. Un graphe d'états est dit accessible si l'ensemble des états but est non vide et coïncide avec l'ensemble des états du problème. Cette dernière condition ne peut être vérifiée que lors de la fouille car nous ne connaissons pas d'avance la composition de l'horaire complet. □

Le graphe d'états du problème de répartition des tâches est très grand. On ne peut que l'expliciter partiellement à l'aide des règles de génération. Si un horaire complet existe, il ne sera découvert que par une application successive des règles de génération. Autrement dit, l'horaire complet sera construit d'une manière progressive. Même dans

cette condition, on peut montrer que l'assignation des tâches de l'horaire complet peut former un horaire à coût minimal.

Théorème 4.2.2 *Un horaire complet est à coût minimal pourvu que ses horaires partiels soient aussi à coût minimal.*

Démonstration Nous savons, par les corollaires découlant du théorème D3.0.6, que le chemin joignant un état n à un état but est minimal si l'état n est situé sur un chemin minimal. De plus, le chemin joignant un état n et son successeur m est également un chemin minimal si n et m sont situés sur un chemin minimal. Transposons les états en assignations de tâches de la forme (T_i, P_j), les chemins en séquences d'assignations et le théorème est démontré.

□

La conséquence directe de ce théorème est la possibilité de construire un horaire complet minimal par une fouille progressive de l'espace d'états du problème. Certes, la condition d'un graphe d'états accessible (théorème 4.2.1) est une condition sine qua non pour la construction d'un horaire complet. Mais comme nous allons le voir, il est possible de détecter cette condition par une réalisation judicieuse de la méthode de fouille.

4.2.2 Méthode de fouille heuristique

D'après les théorèmes 4.2.1 et 4.2.2, nous pouvons réaliser une méthode de répartition heuristique pour la construction progressive d'un horaire complet à coût minimal. Cette méthode heuristique doit pouvoir profiter des connaissances spécifiques à la répartition des tâches temps réel et des connaissances relatives aux propriétés de l'espace d'états du problème.

De plus, si E est l'ensemble des états du problème et B est l'ensemble des états « but » alors la méthode heuristique doit pouvoir détecter la condition

$$E \cap B = \varnothing. \qquad (4.2.2.1)$$

Si la condition (4.2.2.1) est vérifiée alors la méthode de fouille doit terminer immédiatement son travail. En d'autres mots, si $E \cap B = \varnothing$ alors il n'existe pas d'horaire complet. La méthode heuristique doit signaler cette situation et terminer la fouille.

Alors, deux scénarios sont possibles lorsque la méthode de fouille s'arrête. Le premier est la détection de la condition (4.2.2.1). Le second scénario est la rencontre d'un état but avec un horaire complet (minimal ou non). L'objectif ici est de réaliser une méthode de répartition des tâches qui n'admet que ces deux conditions d'arrêt. Une première ébauche de la méthode de fouille heuristique proposée est présentée dans les figures 4.2 à 4.5. La nomenclature du pseudo-code est montrée dans le tableau 4.1.

Tableau 4.1 Nomenclature utilisée pour le pseudo-code.

Étiquette	Signification
CHEMIN(n)	Fonction qui retourne toutes les assignations effectuées jusqu'à l'assignation n.
EXTRACTION(l)	Fonction qui sélectionne une assignation parmi les assignations disponibles dans la liste l.
$f(n), f'(n)$	Valeur et valeur mémorisée de la fonction heuristique pour l'assignation n.
GÉNÉRATION(n)	Fonction qui réalise la génération des assignations (T_i, P_j) selon les règles de génération à partir de l'assignation n.
MISE-À-JOUR( )	Fonction qui réalise la mise à jour des valeurs de f et des pointeurs des assignations obtenues.
OUVERT, FERMÉ	Listes contenant les assignations (T_i , P_j) générées par les règles de génération et par la fonction heuristique de contrôle.
PRED(n)	Fonction qui retourne l'assignation précédente à l'assignation n.
SUCC(n)	Fonction qui retourne une assignation suivante à l'assignation n.

```
    Type-composé Assignation
    {
1       réel f';                          // mémorise la valeur heursitique f
2       pointeur Assignation prédécesseur; // pour retracer l'horaire
                                          // complet
3       tâche t;                          // tâche assignée au processeur p
4       processeur p;                     // processeur choisi pour t
5       réel temps;                       // temps de départ de la tâche t
    }
```

Figure 4.2 Pseudo-code pour la structure de données des assignations.

```
    Function MISE-À-JOUR
    {
1       PRED(p) ← n;
2       f(p) ← f';
    }
```

Figure 4.3 Pseudo-code pour la fonction MISE-À-JOUR.

```
    Function EXTRACTION(LISTE l)
    {
1       retour ( { n | n ∈ G, f'(n) = min_{∀p∈l} f'(p) } );
    }
```

Figure 4.4 Pseudo-code pour la fonction EXTRACTION.

Cette méthode est une variation de la méthode A*. La méthode de fouille heuristique A* est connue pour son efficacité optimale [HAR68], [HAR72], [DEC88]. Nous avons conçu cette variante de A* dans le but d'obtenir certaines propriétés souhaitées pour la répartition des tâches. Ces propriétés sont données à la section 4.2.3.

La méthode de fouille débute par l'extraction d'une première assignation n de la liste OUVERT. Initialement la liste OUVERT est vide. Donc, les règles de génération doivent générer un ensemble d'assignations susceptibles d'être choisies par la fonction d'extraction. Le pseudo-code de la figure 4.4 présente le contenu de la fonction

EXTRACTION. L'extraction s'effectue en prenant une assignation qui donne une valeur minimale pour la fonction d'évaluation heuristique $f(n)$. L'assignation choisie par la fonction d'extraction est retirée de la liste OUVERT et placée dans la liste FERMÉ.

```
    Méthode Répartition
    {
        Assignation n, p;
        réel f';

1       OUVERT ← {s}; FERMÉ ← {∅}; f'(s) ← 0;
2       tant que OUVERT != {∅}
        {
3           n ← EXTRACTION(OUVERT);
4           si n ∈ T alors { HORAIRE(n); arrêt("Succès"); } // solution !
5           OUVERT← OUVERT - {n};
6           FERMÉ ← FERMÉ + {n};
7           répéter ∀ p = GÉNÉRATION(n)
            {
8               f' < f(p);
9               si ((p ∈ OUVERT) && (f' < f(p)) alors MISE-À-JOUR();
10              si ((p ∈ FERMÉ) && (f' < f(p)) alors
                {
11                  MISE-À-JOUR();
12                  OUVERT ← OUVERT + {p};
13                  FERMÉ ← FERMÉ - {p};
                }
14              si ((p ∉ OUVERT) && (p ∉ FERMÉ)) alors
                {
15                  MISE-À-JOUR();
16                  OUVERT ← OUVERT + {p};
                }
            }
        }
17      arrêt("Échec"); // pas de solution !
    }
```

Figure 4.5 Méthode de répartition des tâches (fonction principale).

Ensuite, on génère un ensemble d'assignations à l'aide des règles de génération mais en tenant compte de l'assignation n choisie (ligne 7, figure 4.5). Les éléments de cet ensemble d'assignations sont ensuite classés par la méthode de fouille (lignes 8 à 16). Si l'assignation générée est déjà dans la liste OUVERT et qu'elle possède une valeur heuristique f plus petite que celle déjà calculée (f'), alors la méthode doit effectuer une mise à jour de ces valeurs heuristiques. Si l'assignation générée est déjà dans la liste FERMÉ et qu'elle possède une valeur heuristique f plus petite que celle déjà calculée f',

alors on effectue une mise à jour de ces valeurs heuristiques. On retire l'assignation de FERMÉ pour la placer dans OUVERT. Autrement dit, cette assignation est candidate pour être développée ultérieurement. En fin, si l'assignation générée est une nouvelle assignation (qui n'est pas déjà dans les listes OUVERT et FERMÉ). On doit alors effectuer une mise à jour de ces valeurs heuristiques et placer l'assignation dans la liste OUVERT. Cette assignation est également une candidate pour être développée plus tard.

La fonction de mise à jour (MISE-À-JOUR) est appelée pour modifier la valeur de la fonction heuristique stockée et changer le pointeur de l'assignation précédente. Le stockage de la valeur de f est nécessaire parce que la valeur de f est dynamique. Elle dépend des assignations effectuées précédemment. La mise à jour du pointeur prédécesseur (ligne 1, figure 4.3) est nécessaire pour permettre à la fonction CHEMIN de présenter l'horaire complet une fois la fouille terminée.

Ainsi, la liste OUVERT est utilisée pour entreposer les assignations candidates pour le développement. On choisit toujours une assignation à développer dans OUVERT. La liste FERMÉ sert surtout à éviter le bouclage dans la fouille. En effet, une assignation développée n est toujours placée dans la liste FERMÉ. Elle est remis dans OUVERT si n donne une valeur heuristique plus petite que celle mémorisée (ligne 10 à 13, figure 4.5). On ne reconsidère une assignation déjà développée que si elle présente une meilleure (plus petite) valeur heuristique comparée à celle obtenue lors de son développement précédent.

Nous pouvons comparer le fonctionnement de la méthode **Répartition** à de l'eau qui coule sur une surface. L'eau provient d'une source qui est l'état initial du système. Elle se déplace en suivant un chemin qui a une pente la plus abrupte localement. Lorsqu'elle entre dans un minimum local elle forme une rivière et ce, jusqu'au débordement. Même si le minimum global est situé dans le cratère d'un volcan, il suffit que le niveau d'eau puisse monter suffisamment pour atteindre le bord du cratère et elle

finira par couler jusqu'au plus profond du cratère. Par cette analogie, la liste FERMÉ correspond à la surface sous l'eau de la rivière et la liste OUVERT correspond à la bande de terre le long du rivage. Cette méthode de fouille procède toujours à partir du point le plus bas du rivage.

4.2.3 Propriétés de la méthode Répartition

Dans la méthode de la figure 4.5, il n'existe que deux points de sortie possible. La ligne 4 est une sortie avec un horaire complet. La ligne 17 est une sortie avec aucun horaire. Évidemment, le comportement de la méthode de répartition est fortement influencé par la fonction de contrôle heuristique $f(n)$. Après tout, c'est cette fonction heuristique qui agit comme guide lors de la fouille dans l'espace d'états du problème de répartition. Nous pouvons tirer deux propriétés importantes de la méthode de répartition des tâches sous certaines conditions concernant la fonction heuristique d'évaluation $f(n)$.

Propriété 4.2.3.1 La méthode de fouille s'arrête en sortant à la ligne 4 ou à la ligne 17.

■

Propriété 4.2.3.2 S'il existe un horaire complet pour les tâches, la méthode de fouille le trouvera en sortant à la ligne 4.

■

La propriété 4.2.3.1 indique que le répartiteur des tâches ne pourra jamais boucler indéfiniment. Cette propriété d'arrêt est importante car elle donne l'assurance que la répartition des tâches terminera éventuellement son travail en donnant soit un horaire complet soit un message d'erreur. Quant à la propriété 4.2.3.2, elle indique que le répartiteur des tâches s'arrêtera à coup sûr en produisant un horaire complet si un tel horaire existe. Les conditions nécessaires et suffisantes pour que ces propriétés soient

présentes dans la méthode de répartition des tâches sont données ci-dessous. Pour exprimer succinctement ces conditions nous allons définir quelques termes pour simplifier les énoncées.

Définition 4.2.3.1 Une *itération* de la méthode de répartition des tâches est une boucle complète des lignes 2 à 16 de la figure 4.5. L'exécution de la fonction d'extraction à la ligne 3 à la i^e itération est appelée la i^e *extraction*.

■

Définition 4.2.3.2 On désigne par φ l'ensemble des *numéros d'itérations* de la méthode de répartition des tâches. $\varphi = \{1, 2, \ldots z\}$ où z est le dernier numéro d'itération juste avant la sortie de la méthode.

■

Définition 4.2.3.3 On dénote par $f_j(n)$ la valeur de la fonction d'évaluation heuristique **calculée** à la j^e itération.

■

Définition 4.2.3.4 On dénote par $f_i'(n)$ la valeur de la fonction d'évaluation heuristique **mémorisée** à la i^e itération.

■

Définition 4.2.3.5 On dénote par $\mathcal{W}$ l'ensemble des valeurs possibles donnée par la fonction d'évaluation heuristique $f(n)$.

■

Définition 4.2.2.6 On dénote par OUVERT_i et FERMÉ_i le contenu des listes OUVERT et FERMÉ à la i^e itération de la méthode de répartition des tâches.

■

La condition 4.2.3.1 est une condition nécessaire et suffisante pour que la propriété 4.2.3.1 soit présente. Cette condition assure donc la propriété d'arrêt de la méthode de répartition des tâches. L'ensemble des conditions 4.2.3.1 à 4.2.3.3 sont nécessaires et suffisantes pour que la méthode **Répartition** s'arrête et retourne un horaire complet.

Condition 4.2.3.1 Pour toute assignation n déjà générée, $\forall j \in \varphi$ où j est le dernier numéro d'itération, $\exists i \in \mathcal{N}$ de tel sorte que si l'assignation n est réévaluée par le répartiteur alors pour $j > i \Rightarrow f_j(n) \geq f_i^{'}(n)$.

■

Condition 4.2.3.2 $\forall i \in \varphi, \exists \omega \in \mathcal{W}$ tel que $f(n_i) \leq \omega$. n_i est une assignation extraite à la i^e itération de la répartition des tâches.

■

Condition 4.2.3.3 $\forall i \in \varphi, \exists \omega \in \mathcal{W}, \exists k \in \mathcal{N}$ tel que $f(n_i) \leq \omega \Rightarrow \hat{\eta}(s,n_i) \leq k$. n_i est une assignation extraite à la i^e itération de la répartition des tâches et $\hat{\eta}(s,n_i)$ est le nombre minimal d'arcs dans le graphe d'états reliant l'assignation initiale s (horaire vide) et l'assignation obtenue à la i^e itération.

■

Théorème 4.2.3.1 *La propriété 4.2.3.1 est présente et la méthode de répartition des tâches doit éventuellement terminer son exécution si et seulement si la condition 4.2.3.1 est satisfaite.*

Démonstration La condition 4.2.3.1 est nécessaire. À cause de la façon dont la fonction MISE-À-JOUR effectue son travail, les numéros d'itération $j > i$ implique nécessairement que $f_j(n) = f_i^{'}(n)$. La condition est suffisante car le graphe d'états du

problème de répartition des tâches est un graphe fini. Pour que la méthode ne s'arrête pas il faut que l'une des assignations soit extraite et remise dans liste OUVERT indéfiniment. Pour pouvoir rencontrer cette situation de bouclage, il faut qu'une assignation possède des valeurs f strictement plus petites à chaque réévaluation. La condition 4.2.3.1 interdit cette situation de bouclage et la méthode doit éventuellement terminer son exécution en sortant à la ligne 4 ou à la ligne 17.

□

Lemme 4.2.3.1 *Si le graphe d'états G du problème de répartition des tâches est accessible alors la liste OUVERT est toujours non vide lors d'une extraction quelconque réalisée à la ligne 3 de la figure 4.5.*

Démonstration Si G est accessible, c'est-à-dire G possède au moins un état but. Alors il existe une séquence d'assignations depuis l'horaire initial vide s jusqu'à un horaire complet. Considérons la i^e itération. Si $s \in$ OUVERT$_i$, alors OUVERT est non vide. Considérons le cas où $s \in$ FERMÉ$_i$. Notons par χ la séquence d'assignations $\chi = a_0, a_1, \ldots, a_b$ avec $a_0 = s$ et $a_b \in B$ (ensemble des horaires complets possibles). Un état but n'est jamais développé à la ligne 4 de la figure 4.5. Donc, $a_b \notin$ FERMÉ$_i$. Par conséquent, $\exists\, j \in \mathcal{N}$, $0 \leq j < b$ tel que a_j soit le dernier élément de χ dans FERMÉ$_i$. Si a_j a été placée dans FERMÉ à la x^e itération pour $x \leq i$ alors à la i^e itération tous ses successeurs auront été développé grâce à la boucle des lignes 7 à 16. À la i^e itération tous les successeurs de a_j seront dans OUVERT$_i$ ou dans FERMÉ$_i$. Mais comme a_j est le dernier élément de χ dans FERMÉ$_i$, l'élément suivant a_{j+1} doit être dans OUVERT$_i$. Donc la liste OUVERT n'est jamais vide lorsqu'un horaire complet existe.

□

Théorème 4.2.3.2 *La propriété 4.2.3.2 est présente et la méthode de répartition des tâches s'arrêtera avec un horaire complet si et seulement si les conditions 4.2.3.1, 4.2.3.2 et 4.2.3.3 sont satisfaites.*

Démonstration La condition 4.2.3.1 est nécessaire et suffisante pour l'arrêt de la méthode de répartition des tâches. La condition 4.2.3.2 est nécessaire pour que les valeurs de $f(n)$ soient bornées. La condition 4.2.3.3 est nécessaire pour que le nombre d'assignations de l'horaire initial jusqu'à l'assignation de la $i^{\text{ème}}$ itération soit borné.
Puisque la condition 4.2.3.3 est satisfaite alors $\forall i \in \varphi, \exists\, \omega \in \mathcal{W}, \exists\, k \in \mathcal{N}$ tel que $f(n_i) \leq \omega \Rightarrow \hat{\eta}(s,n_i) \leq k$. Puisque la condition 4.2.3.2 est satisfaite alors $\forall i \in \varphi, \exists\, \omega \in \mathcal{W}$ tel que $f(n_i) \leq \omega$. Donc, $\forall i \in \varphi, \exists\, k \in \mathcal{N}$ tel que $n_i \in E$ où $E = \{n \mid \hat{\eta}(s,n) \leq k\}$. Or, le graphe d'états du problème de répartition des tâches est fini, E est également un ensemble fini (théorème D3.0.2). Alors pour que la méthode ne s'arrête pas, il faut que l'une des assignations de E soit extraite indéfiniment de la liste OUVERT. Ce qui est impossible par la condition 4.2.3.1. Donc la méthode de fouille s'arrêtera éventuellement. D'après le lemme 4.2.3.1, la liste OUVERT n'est jamais vide si un horaire complet existe. Alors, la seule sortie possible de la méthode est une sortie avec un horaire complet.

□

Donc, un répartiteur de tâches réalisé par la méthode de la figure 4.5 possède une fin d'exécution indubitable et un résultat assuré si les trois conditions 4.2.3.1, 4.2.3.2 et 4.2.3.3 sont satisfaites.

4.2.4 Caractéristiques de la fonction heuristique $f(n)$

Pour que la méthode de répartition des tâches termine toujours son exécution, nous devons satisfaire la condition 4.2.3.1. Pour pouvoir produire un horaire complet nous devons satisfaire la condition d'arrêt et les conditions 4.2.3.2 et 4.2.3.3. Toutes ces conditions exigent un nombre de caractéristiques intrinsèques à la fonction de d'évaluation heuristique $f(n)$.

En effet, pour la condition d'arrêt (condition 4.2.3.1), la fonction $f(n)$ doit être définie de manière à obtenir $f_j(n) \geq f_i(n)$ lorsque $j > i$. Cette caractéristique de $f(n)$ est possible si la fonction heuristique $h(n)$ est décroissante finie. Une fonction est décroissante finie si dans la suite de valeurs prises par $h(n)$, il n'existe pas de sous-suite infinie qui soit strictement décroissante [EPP95]. Le terme $g(n)$ n'intervient pas dans cette caractéristique puisqu'il est calculé en fonction de n et non en fonction du numéro d'itération (définition D2.0.1, Annexe D). Autrement dit, $g_j(n) = g_i(n)$. La caractéristique décroissante finie de $h(n)$ est obtenue dès que $h(n)$ prend ses valeurs dans un ensemble qui est fini [EPP95]. Si l'image de n sous h est un ensemble fini alors la condition d'arrêt de la méthode est satisfaite. Les lemmes 4.2.4.1 et 4.2.4.2 ainsi que le théorème 4.2.4.1 démontrent ce fait.

Lemme 4.2.4.1 *[EPP95] Une fonction heuristique h(n) qui prend ses valeurs dans un ensemble fini est une fonction heuristique décroissante finie.*

Démonstration Par contradiction. Supposons que $h(n)$ prend ses valeurs dans l'ensemble V du monoïde des valeurs d'arcs. $V \subset \mathcal{R}$ est un ensemble infini. Alors, $\forall\ m \in V$, $\exists\ n$ état de G, $\exists\ i \in \varphi$ un numéro d'itération pour lequel $h_i(n) < m$. Il est possible de trouver un numéro d'itération $j \in \varphi$ pour lequel $h_j(n) < h_i(n)$. De même, $\exists\ k \in \varphi$ un numéro d'itération pour lequel $h_k(n) < h_j(n)$ et ainsi de suite. On obtient alors une sous-suite infinie strictement décroissante. Or, ce n'est pas possible si l'ensemble des valeurs de $h(n)$ est un ensemble fini. Forcément, la valeur de $h(n)$ ne sera plus distincte pour certains numéros d'itération si l'ensemble des valeurs de $h(n)$ est un ensemble fini.

□

Lemme 4.2.4.2 *Dans un graphe d'états G fini, borné sur L et sans circuit, f(n) est décroissante finie pourvu que h(n) soit aussi décroissante finie.*

Démonstration Par contradiction. Supposons que $f(n) = r(g(n), h(n))$ n'est pas décroissante finie pour un état n donné. Il existe alors une sous-suite infinie de f qui est strictement décroissante pour cet état. Noter cette sous-suite infinie de valeur par $f_a > f_b > f_c > \ldots >$ où $a < b < c < \ldots <$ sont les numéros d'itération. Les valeurs de $g(n)$ ne dépendent que de n alors nous avons $g_a = g_b = g_c = \ldots =$. La fonction heuristique $h(n)$ est décroissante finie. Donc, au delà d'un certain numéro d'itération i nous aurons $v > u > i \Rightarrow h_v(n) \geq h_u(n)$. C'est-à-dire, $h(n)$ prend un nombre fini de valeur distincte strictement décroissante. Puisque le graphe d'états est fini, borné sur L et sans circuit, la relation r du monoïde de valeurs est croissante à droite ainsi $f_v = r(g_v, h_v(n)) = r(g_u, h_v(n)) \geq r(g_u, h_u(n)) = f_u$. Donc, au delà d'un certain numéro d'itération i, $f_v \geq f_u$ pour $v > u > i$. Ce qui contredit la supposition initiale.

□

Théorème 4.2.4.1 *Si la fonction h(n) prend ses valeurs dans un ensemble fini alors* $f_j(n) \geq f_i'(n)$ *pour* $j > i$.

Démonstration Par le lemme 4.2.4.1, la fonction heuristique $h(n)$ est croissante finie si elle prend ses valeurs dans un ensemble fini. Par le lemme 4.2.4.2, $f(n)$ est également croissante finie. Donc, $f(n)$ n'a pas de sous-suite infinie strictement décroissante. À cause de la façon dont les valeurs de f sont misent à jour (ligne 9, 11 et 15 de la figure 4.5) alors à un certain numéro d'itération i on aura $f_j(n) \geq f_i'(n)$ pour $j > i$.

□

Pour satisfaire à la condition 4.2.3.2, la fonction d'évaluation heuristique doit avoir $f(n_i) \leq \omega$ où n_i est l'assignation extraite de la i^e itération et ω une valeur de l'ensemble image f. Donc, les valeurs de f doivent être bornées. Nous pouvons choisir $\omega = f^*(n)$ si la fonction heuristique $h(n)$ est conçue de sorte que $h(n) \leq r(c(n, n'), h(n'))$ où

n' est un état successeur de n et $c(n, n')$ est le coût exigé pour passer de l'état n à l'état n'. En effet, le théorème suivant montre que ce choix est acceptable.

Théorème 4.2.4.2 *Pour un graphe d'états G fini, borné sur L et sans circuit, la fonction d'évaluation heuristique f(n) est bornée si* $h(n) \leq r(c(n, n'), h(n'))$ *et* $\forall\, b \in B$, $h(b) = \varepsilon$. *L'état n' est le prédécesseur de l'état n, B est l'ensemble des états but et ε est l'élément neutre de la relation r.*

Démonstration $\forall\, n \in G$, $h^*(n)$ est définie puisque G est fini, borné sur L et sans circuit. Donc, $\exists\, b \in B$ tel que $h^*(n) = L^*(n, B) = L^*(n, b)$ (définition D3.1.2, Annexe D). Alors $\forall\, i \in \varphi$ numéro d'itération où l'état n est évalué et $\forall\, i \in \varphi$ numéro d'itération où l'état b est évalué nous avons $h_i(n) \leq r(L^*(n, b), h_j(b))$. Vu que $h(b) = \varepsilon$, nous pouvons écrire $h(n) \leq r(h^*(n), \varepsilon) = h^*(n)$. Puisque $r(g(n), h(n))$ est une estimation de $r(g^*(n), h^*(n))$ (section D2, Annexe D), nous pouvons affirmer que $f(n) \leq f^*(n)$. Donc, la fonction d'évaluation heuristique $f(n)$ est bornée par $f^*(n)$.

□

En terme pratique, le théorème 4.2.4.2 indique que la fonction d'évaluation $f(n)$ doit toujours sous-estimer le coût réel de la solution finale. Le caractère optimiste de la fonction heuristique $h(n)$ a été souligné dans la présentation de l'annexe D. Il est à noter que l'inégalité $h(n) \leq r(c(n, n'), h(n'))$ est équivalente à $h(n) \leq h(n')$ si la relation r du monoïde de valeurs est $(\mathcal{R}^+, +)$. Dans ce cas, l'élément neutre ε du monoïde est 0 et $c(n, n') \geq \varepsilon$. Il est évident que $h(n) \leq h(n') \leq c(n, n') + h(n')$ pour le monoïde $(\mathcal{R}^+, +)$. La fonction d'évaluation heuristique $f(n) = g(n) + h(n)$ est donc une relation du monoïde de valeurs $(\mathcal{R}^+, +)$ dans lequel $g(n) \geq 0$ et $h(n) \geq 0$.

Finalement, pour satisfaire la condition 4.2.3.3, il faut borner les valeur de f et borner le nombre d'assignations dans l'horaire partiel obtenu à la i^e itération. Pour

parvenir à borner le nombre d'assignation dans un horaire partiel, il suffit que l'on ait $f(n_i) \geq L(s, n_i)$. Puisque le graphe d'états du problème est borné sur L (voir l'annexe D, définition D3.0.10), $\forall i \in \varphi, \exists \omega \in \mathcal{W}, \exists k \in \mathcal{N}, L(s, n_i) \leq \omega \Rightarrow \eta(s, n_i) \leq k$ où n_i est l'assignation extrait à la i^e itération de la répartition des tâches et $\eta(s, n_i)$ est le nombre d'assignations dans le graphe d'états reliant l'assignation initiale s (horaire vide) et l'assignation obtenue à la i^e itération. Si $\hat{\eta}(s,n_i)$ est le nombre minimal d'assignations dans l'horaire partiel, alors $L(s, n_i) \leq \omega \Rightarrow \eta(s, n_i) \leq k \Rightarrow \hat{\eta}(s,n_i) \leq k$ puisque $\hat{\eta}(s,n_i) \leq \eta(s,n_i)$. Quand nous avons $f(n_i) \geq L(s, n_i)$ cela signifie que pour $f(n_i) \leq \omega$, nous avons aussi $L(s, n_i) \leq \omega$. Donc, $f(n_i) \leq \omega \Rightarrow \eta(s, n_i) \leq k$. La relation $f(n_i) \geq L(s, n_i)$ satisfait donc les graphes d'états borné sur L.

Tableau 4.2 Caractéristiques de la fonction $f(n)$.

Caractéristique	Signification	Caractéristique	Signification
$f_j(n) \geq f_i'(n), j > i.$	Critère d'arrêt	$h(n) \leq h^*(n) \Rightarrow f(n) \leq f^*(n).$	Critère d'arrêt et f bornée
$f(n_i) \leq \omega.$	f est bornée	$h(n) \leq h(n') \Rightarrow f(n) \leq f(n').$	f bornée
$f(n_i) \leq \omega \Rightarrow \hat{\eta}(s,n_i) \leq k.$	Nombre d'assignations est borné	$f(n) \geq g^*(n).$	Nombre d'assignations est borné

En pratique, la relation $f(n_i) \geq L(s, n_i)$ est équivalente à la relation $f(n) \geq g^*(n)$. Prenons $n = n_i$, et par la définition D3.1.1, $g^*(n) = \hat{L}(s, n_i)$. Alors nous avons $f(n_i) \geq \hat{L}(s, n_i)$. Selon le théorème D3.0.5, pour un graphe d'états fini, borné sur L et sans circuit, il existe une séquence d'assignations tel que $L(s, n_i) = \hat{L}(s, n_i)$. Donc, $f(n_i) \geq L(s, n_i)$ et est équivalente à $f(n_i) \geq g^*(n_i)$. Enfin, nous récapitulons les diverses caractéristiques de la fonction d'évaluation heuristique $f(n)$ dans le tableau 4.2.

4.2.5 Complexité temporelle et spatiale

Les aspects les plus contraignants de la méthode **Répartition** sont le temps de réponse et sa demande en mémoire. Tout d'abord, cette méthode génère toutes les successeurs possibles pour chaque assignation retenue (ligne 7, figure 4.5). Après traitement par la boucle principale (ligne 7 à ligne 16, figure 4.5), les successeurs sont placés dans la liste OUVERT ou dans la liste FERMÉ. Donc, toutes les assignations candidates pour le développement seront éventuellement développées et conservées en mémoire.

On peut toujours construire un graphe d'états où la méthode **Répartition** génère un nombre exponentiel d'assignations candidates. Sans aucune modification de cette méthode, la quantité de mémoire exigée peut devenir trop grande même pour les ordinateurs les plus puissantes dont nous disposons aujourd'hui. Puisque l'on calcule pour chaque assignation développée sa valeur heuristique $f(n)$ (ligne 8, figure 4.5), le temps de réponse de la méthode de répartition des tâches peut aussi avoir une complexité exponentielle. Donc, les aspects avantageux de la méthode **Répartition** sont en fait obtenus sans égard à son comportement général en terme de complexités temporelles et spatiales.

Nous montrons maintenant la condition suffisante pour que la méthode de répartition des tâches génères un nombre exponentiel d'assignations candidates. Ce cas pathologique a été démontré par Martelli dans [MAR77]. Elle utilise un résultat sur les anomalies de l'algorithme de Dijkstra pour le calcul d'un chemin minimal dans un graphe orienté simple, sans circuit et dont les arcs peuvent avoir des valeurs positives ou négatives [JOH73]. Pour les besoins de cette démonstration, nous utilisons l'ensemble des assignations candidates pour représenter la taille du problème. La raison est que la méthode **Répartition** ne développe pas explicitement tout l'espace d'états lors de la fouille. Elle ne considère que la partie intéressante indiquée par la fonction d'évaluation heuristique $f(n)$. On ne peut donc pas utiliser la grandeur du graphe d'états pour

représenter la taille du problème. La définition 4.2.5.1 donne une caractérisation de l'ensemble des assignations candidates.

Définition 4.2.5.1 Soit un graphe d'états $G = (S, A)$ où S est l'ensemble des assignations et A est l'ensemble des arcs pondérés reliant les assignations de S. Une assignation est une candidate pour le développement si elle appartient à l'ensemble $S' = \{s\} \cup \{ n \mid r(g^*(n), h(n)) \leq f^*(s)$ et au moins un parent de n est dans $S'\}$. Dans cet ensemble d'assignations candidates, s est l'assignation initiale (horaire vide) et la relation $r(\cdot,\cdot)$ est celle définie dans le monoïde de valeurs (Annexe D, définitions B3.0.4). $S' \subseteq S$ est l'ensemble des assignations dont la valeur de la fonction d'évaluation heuristique $f(n)$ est inférieure à celle du plus petit chemin entre l'assignation initiale s et l'ensemble des buts.

■

Cette définition de l'ensemble des assignations candidates est correcte puisque la méthode **Répartition** développe toujours les assignations parmi celles qui ont une valeur heuristique inférieure à la $f^*(s)$. Même si certaines d'entre-elles ne seront pas éventuellement développées, elles sont néanmoins conservées en mémoire.

Définition 4.2.5.2 La taille du problème est la cardinalité de l'ensemble des assignations candidates $\aleph = |S'|$.

■

Nous utilisons une analogie entre l'algorithme de Dijkstra et la méthode **Répartition** pour montrer que la méthode de répartition des tâches peut générer un nombre exponentiel d'assignations candidates. Nous pouvons effectuer cette correspondance, parce que l'algorithme de Dijkstra est analogue à celui de la fouille heuristique lorsque le terme heuristique $h(n) = 0$ [MAR77]. Pour ce faire, nous construisons un graphe D pour lequel l'algorithme de Dijkstra s'exécute en $O(2^{\aleph})$

itérations [JOH73]. Par la suite, nous effectuons une transformation d'un graphe D en un graphe d'états G pour démontrer qu'il existe des problèmes où notre méthode de répartition des tâches développe un nombre d'assignations candidates dont la cardinalité est une fonction exponentielle de la quantité $\aleph$.

D'abord construisons un graphe D orienté simple avec $\aleph$ sommets, sans circuit. Les valeurs des arcs peuvent être positives ou négatives. Utilisons les formules de constructions suivantes [JOH73] pour les valeurs des arcs (n_i, n_j) avec $i > j$.

$$
\begin{aligned}
&c_D(n_2, n_1) = -2, \\
&c_D(n_{i+1}, n_1) = c_D(n_i, n_1) - (2^{i-2} + 1), \qquad 2 \le i < \aleph, \\
&c_D(n_i, n_{j+1}) = c_D(n_i, n_j) + 1, \qquad 1 < j+1 < i \le \aleph.
\end{aligned}
\tag{4.2.5.1}
$$

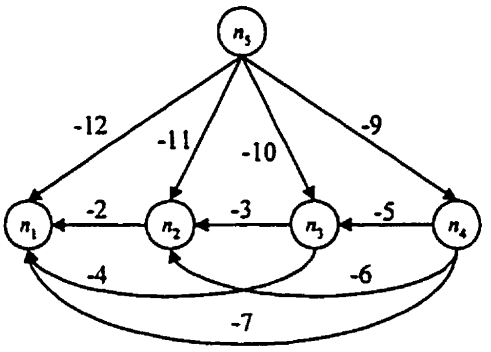


Figure 4.6 Graphe D construit selon les valeurs d'arcs données par les équations (4.2.5.1).

La figure 4.6 montre un exemple de ce graphe avec $\aleph = 5$. Remarquer que ce graphe possède des valeurs d'arcs qui sont exactement $-(2^{i-2} + 1)$ pour $1 < i \le \aleph$. Si le sommet de départ s est $n_\aleph$ alors le chemin minimal entre s et un sommet n_i est

$$
\begin{aligned}
&g(n_\aleph) = 0, \\
&g(n_{i-1}) = g(n_i) - (2^{i-2} + 1), \qquad 1 < i \le \aleph,
\end{aligned}
\tag{4.2.5.2}
$$

$$g(n_i) = \sum_{k=i}^{\aleph-1} c_D(n_{k+1}, n_k), \qquad 1 \le i < \aleph.$$

Par inspection de la figure 4.6 la suite d'arcs n_i, n_{i-1}, ..., n_{i-k+1}, n_k est le chemin minimal des sommets n_i à n_{i-k}. et l'algorithme de Dijkstra s'exécutera en $O(2^{\aleph})$ itérations pour un tel graphe [JOH73].

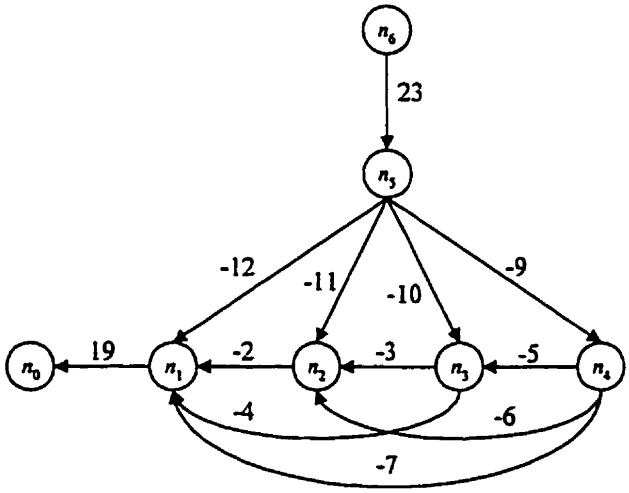


Figure 4.7 Graphe D augmenté par l'addition de deux sommets n_0 et $n_{\aleph+1}$.

Maintenant augmentons le graphe D par l'addition de deux nouveaux sommets n_0 et $n_{\aleph+1}$ avec les valeurs d'arcs suivantes:

$$c_D(n_1, n_0) = -\sum_{i=1}^{\aleph-1} c_D(n_{i+1}, n_i), \tag{4.2.5.3}$$

$$c_D(n_{\aleph+1}, n_{\aleph}) = -\sum_{i=1}^{\aleph-1} c_D(n_{i+1}, n_i) + (\aleph+1). \tag{4.2.5.4}$$

Ces valeurs sont choisies de façon à faciliter la transformation des graphes d'états en un graphe D. La figure 4.7 est le graphe D augmenté. Prenons $n_{\aleph+1}$ comme sommet de départ. L'algorithme de Dijkstra s'exécutera de la même façon mais avec deux itérations supplémentaires. La complexité temporelle de l'algorithme Dijkstra est toujours exponentielle dans le graphe D augmenté.

En prenant n_0 comme un état but et $n_{\aleph+1}$ comme une assignation initiale nous pouvons appliquer la méthode de répartition des tâches à ce graphe D augmenté avec un terme heuristique $h(n)$ nul. La fouille heuristique de la méthode de répartition des tâches se comportera alors exactement comme l'algorithme de Dijsktra [MAR77]. Autrement dit, la méthode **Répartition** développera un nombre exponentiel de sommets avant la fin de son exécution.

Pour la transformation d'un graphe D en un graphe d'états G. La condition suffisante est donnée par les valeurs heuristiques et les valeurs d'arcs suivantes [MAR77]:

$$h(n_0) = h(n_1) = 0, \tag{4.2.5.5}$$

$$h(n_i) = h(n_{i-1}) + (2^{i-2} + 2), \qquad 1 < i \leq \aleph, \tag{4.2.5.6}$$

$$c_G(n_i, n_j) = c_D(n_i, n_j) + h(n_i) - h(n_j), \qquad i > j. \tag{4.2.5.7}$$

Il suffit donc de prendre le graphe augmenté D et d'appliquer les équations (4.2.5.5) à (4.2.5.7) pour obtenir un graphe d'états G. Ainsi, le graphe D augmenté de la figure 4.7 peut être transformé en un graphe d'états G tel que montré dans la figure 4.8.

Dans le graphe d'états obtenu, le coût de l'arc reliant l'état $n_{\aleph+1}$ et l'état $n_{\aleph}$ est exactement égal à la valeur heuristique de l'état $n_{\aleph}$. C'est-à-dire, $c_D(n_{\aleph+1}, n_{\aleph}) = h(n_{\aleph})$. C'est pour cette raison que le graphe d'états G ne contient pas l'état $n_{\aleph+1}$. Dans cette figure, la valeur heuristique $h(n)$ est indiquée dans les carrés.

Il est à remarquer que le graphe d'états G obtenu possède une caractéristique bien spéciale. En effet, pour chaque arc reliant les états n_i et n_j avec $i > j$, nous avons toujours $h(n_i) - h(n_j) > c(n_i, n_j)$. Autrement dit, nous avons construit un graphe d'états dans lequel le terme heuristique $h(n)$ ne donne pas une bonne estimation de $h^*(n)$ peu importe le choix de l'état successeur. Donc, la transformation d'un graphe orienté et sans circuit de

valeurs d'arcs positives ou négatives selon les règles (4.2.5.5) à (4.2.5.7) produit un graphe d'états orienté et sans circuit de valeurs d'arcs positives avec $h(n_i) - h(n_j) > c(n_i, n_j)$ pour $i > j$. Ce graphe d'états est équivalent au graphe D augmenté, cela signifie que la fouille heuristique de la méthode de répartition des tâches doit effectuer $O(2^{\aleph})$ développements avant de terminer son exécution. Puisque la méthode conserve tous les états développés en mémoire, sa complexité spatiale est également $O(2^{\aleph})$.

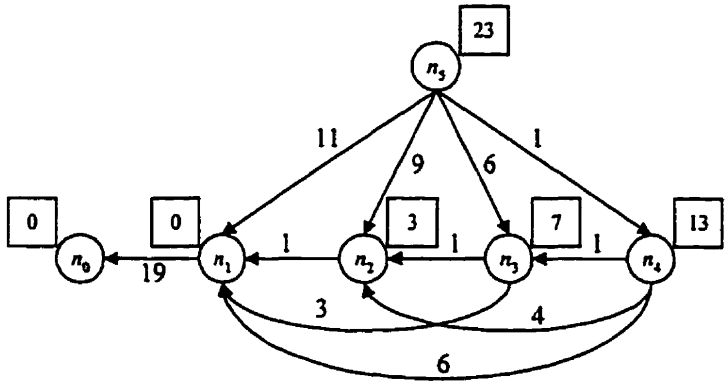


Figure 4.8 Graphe D augmenté est transformé en graphe d'états G.

Le point à souligner est que tout graphe d'états pour lequel $h(n_i) - h(n_j) > c(n_i, n_j)$ est vrai, le nombre d'états développés sera toujours une fonction exponentielle de $\aleph$. En pratique il est très difficile de trouver des heuristiques qui peuvent garantir que $h(n_i) - h(n_j) \leq c(n_i, n_j)$, $\forall n_i, n_j, i > j$. Donc, la croissance exponentielle est une éventualité qui est présente. D'ailleurs, l'imprévisibilité de la croissance exponentielle est une caractéristique des méthodes heuristiques. Parce que la condition $h(n_i) - h(n_j) \leq c(n_i, n_j)$, $\forall n_i, n_j, i > j$ dépend fortement de la fonction heuristique $h(n)$, il arrive fréquemment qu'une méthode heuristique puisse solutionner facilement un problème p mais échoue lamentable pour un problème semblable p' à cause de l'explosion combinatoire.

Pour la mise en œuvre pratique de la méthode de répartition des tâches, nous devons apporter quelques modifications à la méthode Répartition. Nous devons ramener

le temps de réponse de la répartition des tâches en une fonction polynomiale de $\aleph$ et empêcher la croissance exponentielle dans l'utilisation de la mémoire pour le stockage des états lors de la fouille.

4.3 Méthodes de fouille avec limitation de la mémoire

Pour les fouilles heuristiques, la croissance exponentielle de la mémoire utilisée est un problème plus aigu que la complexité temporelle. La raison est que l'espace de la mémoire disponible peut être complètement utilisée en quelques secondes. Par exemple, un poste de travail Sun Ultra-10 possède une puissance de traitement de 100MIPS (méga instructions par seconde) et une capacité maximale de 2Go de mémoire vive. Supposons que la cardinalité de l'ensemble des assignations candidates $\aleph$ est de 40 et que chaque assignation occupe 100 octets de mémoire. Ainsi, $100 \times |2^{\aleph}| \approx 110 \times 10^{12}$ octets sont nécessaires pour conserver toutes les assignations en mémoire. Cette quantité de mémoire dépasse largement la capacité maximale de la machine. Supposons que le temps nécessaire pour développer une assignation est de 100 instructions. Alors, la mémoire vive disponible sera complètement utilisée en moins de 4 minutes. Ainsi, le problème de la complexité spatiale se manifeste bien avant celui de la complexité temporelle.

Pour palier à ce problème de complexité spatiale, nous pouvons imposer une limite à la grandeur des listes OUVERT et FERMÉ utilisées dans la méthode de fouille. Or, le simple fait de limiter la grandeur de ces listes rend la méthode inopérante. Il faut non seulement limiter la grandeur de ces listes mais aussi conserver certaines informations additionnelles qui peuvent continuer à guider la fouille.

Dans la littérature, deux voies s'offrent pour réaliser cet objectif. La première consiste à maintenir, par des modifications mineures, les caractéristiques avantageuses de la méthode **Répartition** sous certaines conditions et d'imposer directement une limite

fixe au nombre d'états conservés en mémoire. L'autre voie consiste à utiliser une méthode de fouille classique réputée pour avoir une complexité spatiale linéaire. Puis ajouter des informations heuristiques pour qu'elle devienne une méthode de fouille heuristique. Ces deux façons d'agir se ressemblent mais produisent des méthodes de fouille qui diffèrent beaucoup dans leur comportement. Cependant, elles ont toutes un point en commun. Toutes ces méthodes de fouille avec limitation de mémoire doivent effectuer explicitement de l'élagage pour réduire le nombre d'états conservés en mémoire.

La méthode SMA* est une méthode de fouille heuristique qui impose directement une limitation sur la quantité de mémoire utilisée pour le stockage des états générés [RUS92]. Cette méthode utilise une seule liste OUVERT de grandeur MEM-MAX. Les états sont entreposés dans cette liste en ordre décroissant de la valeur heuristique *f*. Cette méthode de fouille effectue un élagage explicite lorsque le nombre d'états générés est supérieur à MEM-MAX. L'élagage dans SMA* consiste à enlever un état de OUVERT ayant la plus grande valeur heuristique *f*. Le monoïde de valeurs utilisé est $(\mathcal{R}^+, +)$. Donc, $f(n) = g(n) + h(n)$ et $g(n), h(n) \in \mathcal{R}^+$.

Les figures 4.9 à 4.13 sont le pseudo-code de la méthode SMA*. Elle suit essentiellement les mêmes démarches que celles de la méthode **Répartition**. La différence est dans la limitation explicite de la taille de la liste OUVERT (ligne 14, figure 4.9) et l'implantation d'une fonction d'élagage (ligne 22, figure 4.9). La fonction COMPLET sert à indiquer si un état a été complètement développé. Un état est complètement développé si tous ses successeurs sont générés. Pour pouvoir continuer la fouille malgré la possibilité d'élagage d'un état, la meilleure valeur heuristique *f* est mémorisée dans les ancêtres de tous les états complètement développés (ligne 17, figure 4.9 et fonction RÉTRO-PROPAGATION de la figure 4.13).

```
    Méthode SMA*(entier MEM-MAX)
    {
        Assignation m, n; entier usage;

1       OUVERT ← {s}; usage ← 1;    // s est l'état initial du problème
2       répéter
        {
3           si ((OUVERT = {∅}) || (f(s) = ∞)) alors { arrêt("Échec")};
4           m ← EXTRACTION(OUVERT);
5           si (f(m) = ∞) alors // remettre le parent de m dans OUVERT
6               OUVERT ← OUVERT + {m.prédécesseur};
7           autrement {
8               n ← GÉNÉRATION(m); // génère un successeur de m
9               si n ∈ B alors { HORAIRE(n); arrêt("Succès"); } // solution !
10              si (n != bidon) alors // m est sans issue par manque
11                  si (NBPARENT(n) > MEM-MAX) alors // de mémoire ?
12                      n ← bidon;
13              si (n != bidon) alors
14                  f(n) ← max(f(m), g(n) + h(n));
15              autrement // m est sans issue !
16                  f(n) ← ∞;
17              si COMPLET(m) alors RÉTRO-PROPAGATION(m);
18              si (f(m) != ∞) alors {
19                  si COMPLET(m) alors
20                      OUVERT← OUVERT - {m};
21                  usage ← usage + 1;
22                  si usage > MEM-MAX alors {
23                      ÉLAGAGE(OUVERT);
24                      usage ← usage - 1;
                    }
25                  OUVERT ← OUVERT + {n};
                }
            }
        }
    }
```

Figure 4.9 Pseudo-code de SMA* (fonction principale).

```
   Type-composé Assignation
   {
1      réel f;                          // Valeur de la fonction d'évaluation.
2      réel fm;                         // Meilleure valeur de f pour la partie
                                        // élaguée du graphe d'états.
3      Pointeur Assignation prédécesseur;  // Pour retracer la solution.
4      liste Assignation successeurs;// Pour retracer les successeurs.
5      entier nbsucc;                   // Nombre de successeurs.
6      tâche T;                         // Tâche assignée au processeur P.
7      processeur P;                    // Processeur choisi pour T
8      réel temps;                      // Temps de départ de la tâche T
   }
```

Figure 4.10 Type de données utilisé par SMA*.

```
   Fonction ÉLAGAGE(liste Assignation l)
   {
       // L'élagage consiste à éliminer une assignation de la liste OUVERT.
       // L'assignation à élaguer est celle qui possède la plus grande
       // valeur de la fonction d'évaluation heuristique f.
       Assignation p, q;
1      p ← max {f(q)} où q ∈ l.
       // Enlever cette assignation de son prédécesseur
2      p.prédécesseur.successeurs ← p.prédécesseur.successeurs - {p};
       // Mettre à jour la valeur f du prédécesseur pour la partie élaguée
       // du graphe d'états. Maintenir toujours la valeur minimale.
3      p.prédécesseur.fm=(p.f < p.prédécesseur.fm)?p.f : p.prédécesseur.fm;
       // Effacer p de la mémoire
4      effacer (p);
   }
```

Figure 4.11 Fonction d'élagage réalisée par SMA*.

La fonction d'élagage est exécutée dès que le nombre d'états en mémoire dépasse la taille MEM-MAX. L'élagage consiste à éliminer, de la liste OUVERT, l'état qui possède la plus grande valeur de *f*. Puisque l'on souhaite obtenir une solution à un plus

petit coût possible, l'élimination de l'état avec la plus grande valeur de f est une politique convenable.

```
    Booléen Fonction COMPLET(Assignation n)
    {
        // Vérifier si n est complet. C'est-à-dire, tous les successeurs de n
        // sont générés.
1       si n.nbsucc = longueur(n.successeurs) alors
2           retourne (VRAI);
3       autrement
4           retourne (FAUX);
    }
```

Figure 4.12 Fonction qui vérifie si une assignation a été complètement développée.

```
    Fonction RÉTRO-PROPAGATION(Assignation n)
    {
        // Cette fonction propage la valeur minimale de la fonction
        // d'évaluation heuristique f des successeurs de n vers ses ancêtres.
        Assignation p; réel fm;

        // Mettre à jour la valeur f de n. Si certains successeurs sont
        // élagués alors utilise la valeur élaguée pour effectuer la mise à
        // jour.
1       fm = min(f(p)) où p ∈ n.successeurs;
2       si n.nbsucc = longueur(n.successeurs) alors
3           n.f = fm;
4       autrement
5           n.f = min(fm, n.fm);
        // Mettre à jour le prédécesseur de n
6       si (n.prédécesseur) alors
7           RÉTRO-PROPAGATION(n.prédécesseur);
    }
```

Figure 4.13 Fonction qui effectue la propagation de la valeur f vers les ancêtres.

En effet, lorsqu'un état n possède une valeur $f(n) = \alpha$, le chemin passant par n ne peut avoir un coût inférieur à α [RUS92]. Donc, en élaguant un état avec une valeur f

maximale, nous sommes certains qu'il existe des chemins à moindre coût partant des autres états qui sont encore dans la liste OUVERT. Dans ce sens, l'élagage dans SMA* est plutôt conservateur. La méthode de fouille heuristique SMA* possède la même puissance que la méthode **Répartition** [RUS92] sous une condition qui est nécessaire et suffisante. Cette condition concerne la taille de la mémoire disponible pour la fouille.

Théorème 4.3.1 *[RUS92] La fouille heuristique SMA* est garantie d'obtenir une solution à coût minimal si et seulement si MEM-MAX est plus grand ou égal au nombre d'états générés pour le chemin menant vers la solution à coût minimal.*

□

Autrement dit, en donnant à la méthode SMA* une liste OUVERT de grandeur convenable mais finie, la méthode trouvera toujours une solution minimale si elle existe.

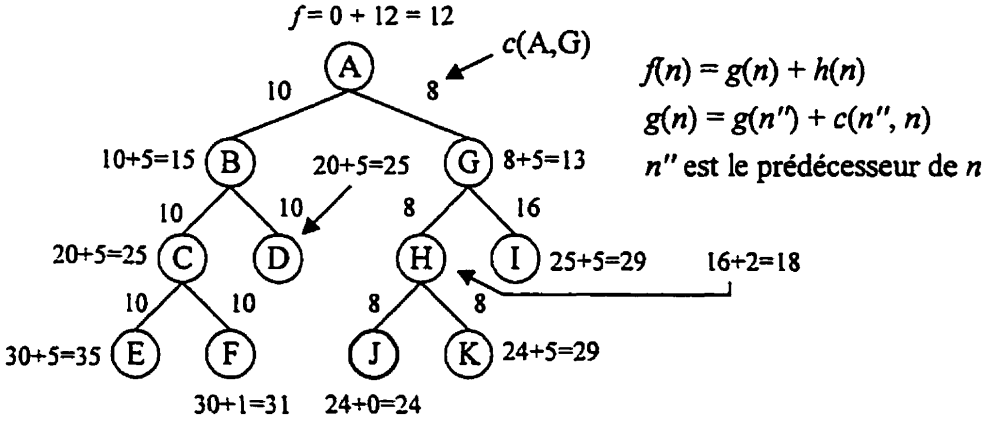


Figure 4.14 Graphe d'états pour la démonstration des méthodes de fouille.

Les figures 4.14 et 4.15 donnent un exemple de fonctionnement de cette fouille heuristique avec limitation de la mémoire. La figure 4.14 est un graphe d'états contenant 11 états. À côté de chaque état est indiqué la valeur de *f*. La valeur des arêtes représente le coût nécessaire pour passer d'un état à un autre. Enfin, l'état J est un état but et sa valeur heuristique *h* est nulle.

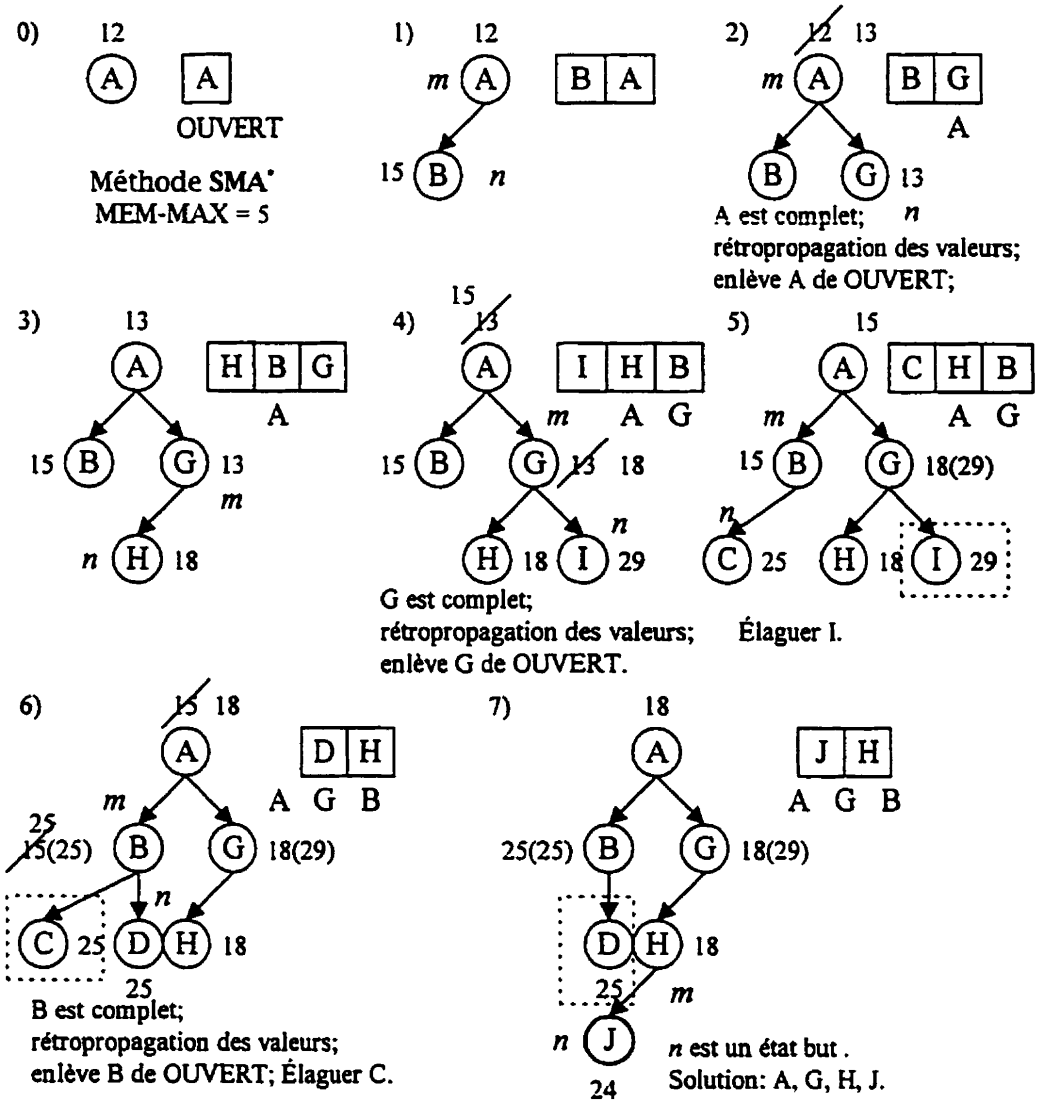


Figure 4.15 Application de la méthode de fouille SMA* (MEM-MAX = 5).

La méthode SMA* utilise un monoïde de valeurs $(\mathcal{R}^+, +)$ et calcule le terme $g(n)$ d'une manière récursive. Ainsi, la valeur de g de l'état n est la somme de la valeur g de son prédécesseur n'' et du coût nécessaire pour passer de l'état n'' à l'état n. Donc, la valeur de $g(n)$ dépend uniquement de n. Le terme g de l'état initial est nul.

La figure 4.15 est une application de la méthode SMA* au graphe d'états de la figure 4.14. La quantité de mémoire est suffisante pour entreposer 5 états. Les états placés dans la liste OUVERT sont en ordre décroissant selon leur valeur heuristique *f*. La fouille s'effectue en développant toujours l'état avec la plus petite valeur de *f*. Dans la figure 4.15, l'état le plus à droite de la liste OUVERT possède la plus petite valeur heuristique.

À l'étape 2, tous les successeurs de l'état A sont développés et la plus petite valeur *f* de ses successeurs est rétro-propagée. L'état A est enlevé de la liste OUVERT mais conservé en mémoire. La rétro-propagation de la valeur minimale de *f* est également effectuée dans les étapes 4 et 6. L'élagage est nécessaire à l'étape 5. L'état à élaguer est celui qui possède la plus grande valeur de la fonction heuristique dans la liste OUVERT.

À l'étape 5, l'état à élaguer est l'état I et il est indiqué par un carré pointillé. L'état but est trouvé à l'étape 7 par le développement de l'état J. La solution obtenue est A, G, H, J. Les états générés par la méthode SMA* sont : A, B, G, H, I, C, D, J. Les états développés sont : A, G, B, H. En utilisant un taille de mémoire MEM-MAX égale à 5, la méthode SMA* appliquée au graphe d'états de la figure 4.14 n'a pas eu à redévelopper les états déjà visités.

Cependant, en pratique, il est souvent impossible de connaître à priori le nombre d'états générés pour l'obtention d'une solution. Si le nombre d'états nécessaires pour l'obtention d'une solution est supérieur à MEM-MAX, alors SMA* terminera son exécution dans un état indéterminé. La méthode de fouille SMA* est incapable de produire des solutions partielles. Les figures 4.16 et 4.17 donnent le résultat d'une fouille par la méthode SMA* dans le graphe d'états de la figure 4.14 mais avec une quantité de mémoire MEM-MAX = 3. La méthode de fouille ne pourra pas trouver une solution puisque le chemin de la solution contient 4 états.

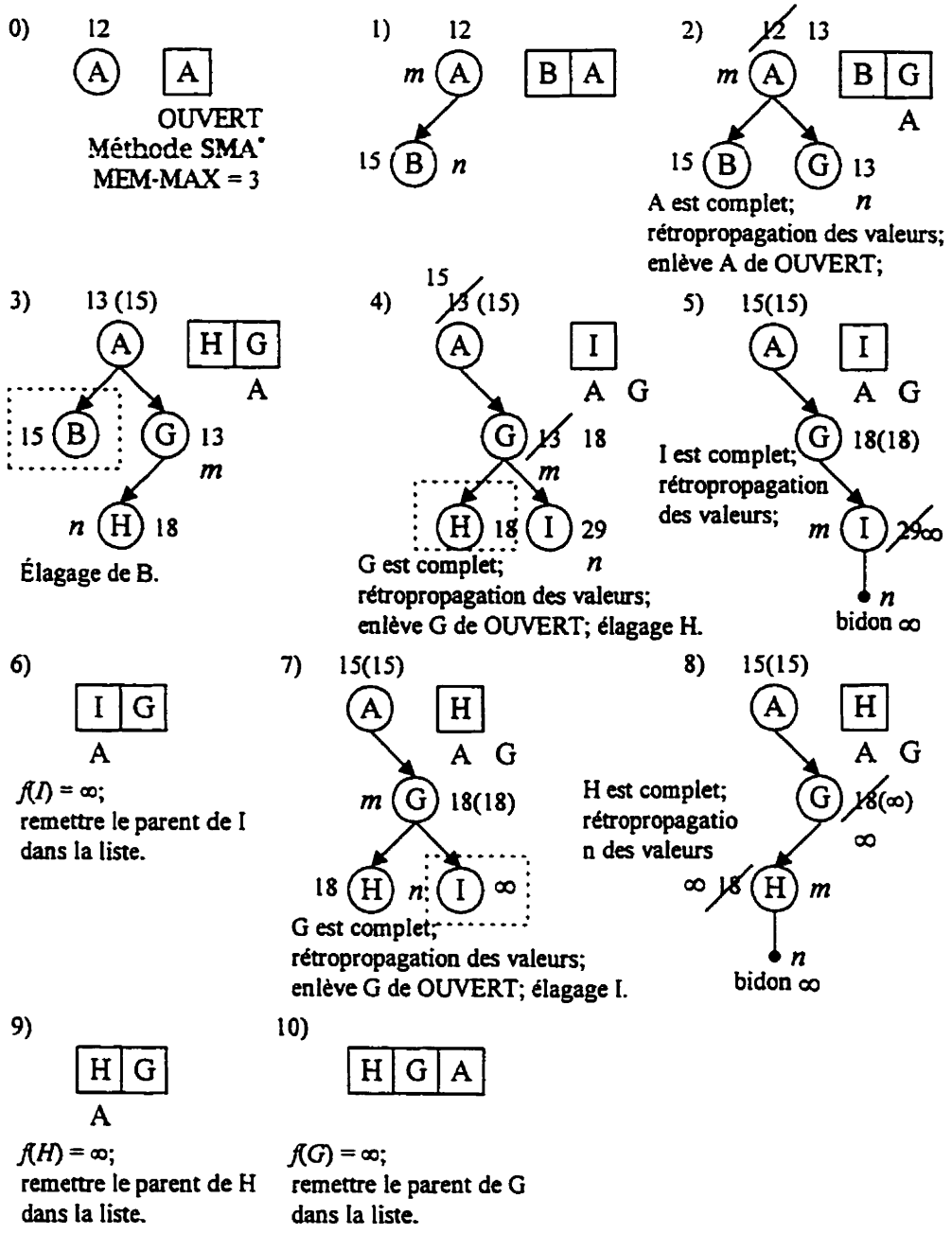


Figure 4.16 Application de la méthode SMA* avec MEM-MAX = 3.

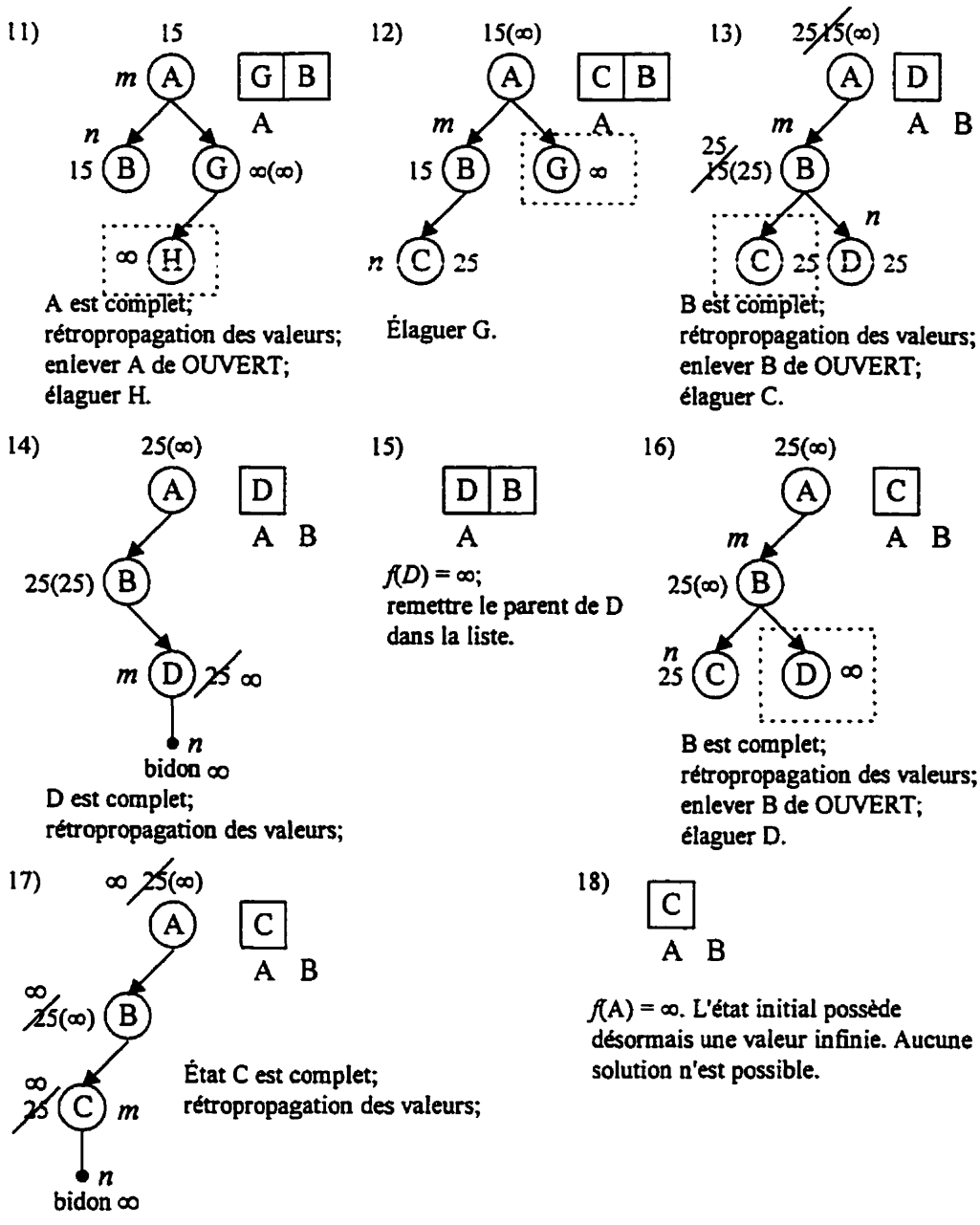


Figure 4.17 Suite de l'application de la méthode SMA* avec MEM-MAX = 3.

La méthode de fouille termine son exécution à l'étape 18. Aucune solution n'a pu être trouvée. Les états générés par la fouille sont : A, B, G, H, I, H, B, C, D, C. On voit l'action de la rétro-propagation des valeurs dans les figures 4.16 et 4.17. Elle permet à la méthode SMA* d'élaguer certaines parties du graphe d'états tout en conservant

suffisamment d'information pour les redévelopper plus tard. Cependant, la rétro-propagation des valeurs empêche l'obtention des chemins partiels lors de la fouille. En effet, lorsque la méthode de fouille rencontre un cul-de-sac (par manque de mémoire ou parce que l'état développé est sans issue), la valeur de la fonction d'évaluation heuristique devient infinie. Si la mémoire allouée pour la fouille est insuffisante alors la valeur infinie sera rétro-propagée dans tous les états retenus en mémoire. Cette situation est rencontrée dans l'étape 17 de la figure 4.17. En conclusion, la rétro-propagation des valeurs peut aider à retenir les chemins intéressants des sous-graphes élagués. Mais elle peut aussi effacer tous les chemins partiels obtenus lors de la fouille.

Une approche différente de celle utilisée dans SMA* est la méthode ITS [GHO94]. Cette dernière profite du fait qu'une fouille en profondeur d'abord a une complexité spatiale linéaire. Si la longueur de la solution minimale est l alors sa complexité spatiale est simplement $O(l)$. Pour contrôler la profondeur de la fouille, et du même coup empêcher une fouille sans issue, la méthode ITS utilise un facteur de contrôle qui indique à la méthode la profondeur maximale à fouiller [KOR85]. Ce facteur de contrôle est justement la fonction d'évaluation heuristique f. Donc, ce sont les valeurs de f qui guideront la fouille.

À l'invocation de la méthode ITS, la profondeur de la fouille est donnée par la valeur de la fonction d'évaluation heuristique f appliquée à l'état initial s du problème. La méthode retient alors la valeur minimale f' obtenue des successeurs de l'état initial s. Une seconde itération est ensuite enclenchée avec comme point de départ l'état initial du problème et à une profondeur donnée par la valeur f'. On procède de la même façon jusqu'à ce qu'on rencontre un état but. Si les valeurs de la fonction d'évaluation heuristique sont monotones et décroissantes alors la fouille finira par trouver le chemin minimal entre l'état initial et l'état but. En tout moment de la fouille, on ne retient que le chemin partiel qui a conduit à la valeur minimale de f. Donc, la complexité spatiale de la fouille est une fonction linéaire de la longueur du chemin minimal parcouru.

Cependant, la fouille en profondeur d'abord utilise trop peu de mémoire car elle ne conserve pas les états déjà rencontrés. La génération multiple des états déjà développés n'est pas rare dans ce genre de fouille. Par conséquent, même si la complexité spatiale de la fouille en profondeur d'abord est linéaire, il n'en demeure pas moins que sa complexité temporelle est souvent exponentielle.

La méthode ITS apporte une solution intéressante au problème de complexité temporelle des fouilles heuristiques en profondeur d'abord. Tout comme la méthode SMA*, la méthode ITS utilise une procédure d'élagage explicite. L'élagage des états sert surtout à compenser la limitation de la mémoire disponible.

Pour ne pas perdre les parties du graphe d'états élaguées, la méthode ITS utilise une technique de rétro-propagation fort différente de celle réalisée dans SMA*. En fait, il n'y a pas vraiment de rétro-propagation des valeurs heuristiques. Il s'agit plutôt d'une mise à jour d'une quantité qui est représentative de la valeur heuristique *f*. Le principe de fonctionnement de l'élagage et de la mise à jour de *f* dans la méthode ITS est comme suit. Lors de la fouille heuristique, un arbre de fouille est maintenu implicitement. Les états sont générés un à un puis placés dans l'arbre de fouille à l'aide de liens reliant les états.

La méthode ITS construit cet arbre de fouille implicite d'une manière particulière. En effet, lorsqu'elle place un état *n* généré dans l'arbre, elle crée en même temps des liens *virtuels* entre l'état *n* et ses successeurs qui ne sont pas encore générés. Il est tout à fait possible de créer ces liens virtuels puisque nous disposons de l'ensemble des opérateurs utilisés dans la génération des états. Au lieu de les invoquer sur l'état *n* pour générer ses successeurs, la méthode ITS effectue un simple test d'applicabilité de ces opérateurs pour connaître le nombre de successeurs de *n*.

La méthode ITS crée un lien virtuel pour chaque opérateur applicable. Ainsi, pour chaque état n considéré, un ensemble de liens virtuels (n, $\mathcal{Q}(n)$) sont placés dans l'arbre de fouille implicite où $\mathcal{Q}$ est l'ensemble des opérateurs applicables. Les états dont les successeurs sont liés par des liens virtuels sont candidats pour l'élagage [GHO94].

Chaque lien (m, n) de l'arbre de fouille contient une variable β. Cette variable sert à maintenir l'estimation du coût du chemin minimal passant par les états m et n. La variable $\beta_{m,n}$ est initialement réglée à $f(m) = g(m) + h(m)$ lorsque l'état m est placé dans l'arbre de fouille.

La valeur $\beta_{m,n}$ est mise à jour à chaque élagage de l'état n. Il est à noter que la méthode ITS utilise aussi un monoïde de valeurs ($\mathcal{R}^+$, +). Lorsque l'état n est choisi pour être élagué, la mise à jour de $\beta_{m,n}$ est effectuée en prenant la plus petite valeur β de tous les liens virtuels de l'état n. La Figure 4.21 est le pseudo-code principal de la méthode ITS. Les structures de données nécessaires sont un peu plus complexes. La Figure 4.18 présente une façon possible pour représenter l'arbre de fouille.

```
    Type-composé Lien
    {
        // idm et idn sont des numéros représentant la position des états
        // dans la liste sommets.
1       entier idm, idn;            // m est le parent de n.
2       entier numop;               // numéro de l'opérateur applicable
                                    // pour générer l'état n.
3       réel β;                     // Valeur de mise à jour de ce lien.
    }

    Type-composé Arbre              // Arbre de fouille
    {
4       Liste Assignation sommets;
5       Liste Lien liens;
    }
```

Figure 4.18 Données utilisées par la méthode ITS.

```
    Fonction AJOUTE(Arbre A, Assignation n, réel g, entier usage)
    {
        // cette fonction ajoute un état dans l'arbre implicite
        entier i, nindice, nbliens;

        // un état est un sommet dans l' arbre implicite
1       nindice ← (A.sommets ← n);
2       usage ← usage + 1;
3       g(n) ← g;
        // s'il n'y a pas d'opérateurs applicables à l'état n ...
4       si (nbliens ← OPERATEUR-APPLICABLE(n)) = 0) alors {
5           A.liens.idm ← nindice; // ajouter un lien bidon
6           A.liens.numop ← bidon;
        }
7       autrement {
            // ajouter des liens virtuels dans l'arbre implicite pour
            // l'état n.
8           pour i = 1...nbliens {
9               A.liens.idm ← nindice;
10              A.liens.numop ← OPERATEUR-APPLICABLE(n);
11              A.liens.β ← g(n) + h(n); // valeur initiale de ce lien virtuel
            }
        }
    }
```

Figure 4.19 Fonction pour la construction de l'arbre de fouille.

Le pseudo-code de la Figure 4.19 montre les étapes impliquées dans la construction de l'arbre de fouille. Les lignes 3 à 11 de cette figure indiquent comment les liens virtuels sont insérés dans l'arbre en même temps qu'un état n. Lorsque ITS ajoute un état dans son arbre de fouille (Fonction AJOUTE, Figure 4.19), la méthode calcule par la fonction OPERATEUR-APPLICABLE le nombre d'opérateurs applicables pour l'état n sans les invoquer. Si $n > 0$, alors on ajoute également les liens virtuels de n dans l'arbre. Par contre, si l'état n n'a pas de successeur, on lui assigne un lien spécial pour empêcher sa régénération (ligne 6, figure 4.19).

```
   Fonction ÉLAGAGE(Arbre A, réel z, entier usage)
   {
      // L'élagage de ITS consiste à éliminer de l'arbre implicite
      // d'un état avec uniquement des liens virtuels dont β est
      // supérieur à la seuil de profondeur heuristique.
      Assignation p, q, n;

1     si ((∃ q | β_q,n > z, ∀ liens (q,n)
          où tous les liens de q sont virtuels) || (q est la première
          assignation trouvée n'ayant que des liens virtuels)) alors
      {
          // mise à jour de la valeur β chez le parent de l'état à
          // élaguer. Eliminer l'état à élaguer.
2         β_p,q ← min_r β_q,r où p est le parent de q dans A;
3         A.sommets ← A.sommet - {q};
4         A.liens ← A.liens - (q, r) où (q, r) sont tous les liens virtuels de q;
5         effacer (q); usage ← usage - 1;
      }
   }
```

Figure 4.20 Fonction de mise à jour des valeurs de β dans ITS.

La fonction d'élagage des états (Figure 4.20)comprend également la routine de mise à jour des valeurs des liens virtuels. Dans la fonction ÉLAGAGE, on élimine le premier état trouvé qui est relié à tous ses successeurs par des liens virtuels ayant une valeur β plus grande que le seuil heuristique z. L'élimination des états ayant des valeurs de liens $\beta > z$ est justifiée. Puisqu'un état q possédant des valeurs $\beta > z$ dans ses liens avec tous ses successeurs signifie que les chemins passant par q sont nécessairement de coût supérieur à z. Et puisque dans une itération donnée z est constante, il s'avère inutile d'explorer les états dont la valeur β est plus grande que z. Enfin, la mise à jour consiste à ajuster la valeur β du lien reliant l'état q et son parent par la valeur minimale des valeur $\beta_{q,r}$ où (q, r) sont les liens virtuels de q.

```
    Méthode ITS(entier MEM-MAX)
    {
        Arbre A; Lien l; réel z; entier usage;
        Assignation m, n;

1       AJOUTE(A, s, 0, usage); // s est l'état initial du problème
2       répéter {
            // seuil de profondeur est la valeur minimale de β des liens
            // virtuels du graphe implicite A.
3           z ← min{β_{p,q}}, ∀(p,q) ∈ A où (p, q) est un lien virtuel du graphe
4           répéter tant que β_{p,q} > z, ∀(p,q) ∈ A {
                // développer un état donc le lien virtuel possède une
                // valeur β ≤ z.
5               l ← Premier lien virutel avec β_{m,n} ≤ z dans A
6               si A.sommets[l.m] ∈ B alors { // Solution !
7                   HORAIRE(A.sommets[l.idm]); arrêt("Succès"); }
                // l'état m est sans issue alors régler β_{m,n} à l' infinie.
8               si l.idn = bidon alors
9                   l.β ← ∞;
10              autrement {
                    // élagage s'il y a lieu
11                  si usage ≥ MAX-MEM alors
12                      ÉLAGAGE(A, z, usage);
                    // Installer le successeur de m dans l'arbre implicite
                    // en générant l'état n avec le numéro de l'opérateur
13                  m = A.sommets[l.idm]; n ← GÉNÉRATION(l.numop);
14                  AJOUTE(A, n, g(m) + c(m, n), usage);
                }
            }
        }
    }
```

Figure 4.21 Méthode de fouille heuristique ITS (fonction principale).

La Figure 4.21 est le code principal de ITS. Cette fouille heuristique début par l'insertion de l'état initial s dans l'arbre de fouille (ligne 1). La profondeur de fouille z est déterminée en utilisant la plus petite des valeurs β des liens virtuels dans l'arbre. La fouille demeure dans la zone délimitée par z tant et aussi long temps que la valeur β des liens virtuels est plus petite que z. Le développement et l'élagage des états sont réalisés

dans les lignes 5 à 12 de Figure 4.21. Un état est développé si et seulement si la valeur β du lien virtuel qui le relie à son parent est plus petite ou égale à z. Évidemment, un état est élagué si le lien virtuel reliant l'état à son parent a une valeur $\beta > z$.

Les propriétés de la méthode ITS sont intéressantes. D'abord, elle ne mémorise que les états du chemin en cours. Donc, sa complexité spatiale est linéaire. Lors du développement d'un état, la méthode ITS sélectionne toujours un lien virtuel (m, n) de valeur $\beta_{m,n} < z$. Ainsi, elle développe toujours les états où le chemin passant est le plus court. Enfin, à cause de la fonction de mise à jour, aucun état ne peut être généré plus d'une fois dans la boucle principale (lignes 4 à 12, Figure 4.21) [GHO94].

Nous avons pu constater une propriété cachée de cette méthode qui peut nous être utile. Puisque la fouille heuristique réalisée par la méthode ITS ne mémorise que les états du chemin en cours, n'importe quel état du graphe d'états peut agir comme point de départ de la fouille. En effet, on peut démarrer la fouille heuristique ITS à un état x sans avoir emmagasiné le chemin de l'état initial à l'état x. Ce fait est une conséquence de la mémoire à court terme de la méthode ITS.

Cependant, si la quantité de mémoire MAX-MEM est insuffisante alors aucune solution partielle ne peut être trouvée, ce qui est contraire à la méthode SMA*. De plus, si le graphe d'états n'est pas accessible, c'est-à-dire, le graphe d'états ne contient pas de but, alors la fouille par la méthode ITS peut boucler indéfiniment sans jamais arrêter.

La figure 4.22 donne un exemple de fonctionnement de cette fouille heuristique avec limitation de la mémoire appliquée au graphe d'états de la figure 4.14 de la page 115. Rappelons-nous que la figure 4.14 est un graphe d'états contenant 11 états. L'arbre implicite construit par la méthode ITS comprend les sommets qui sont les états générés et les liens virtuels de ces états. Dans la figure 4.22, la quantité de mémoire disponible pour la fouille est suffisante pour emmagasiner 5 états.

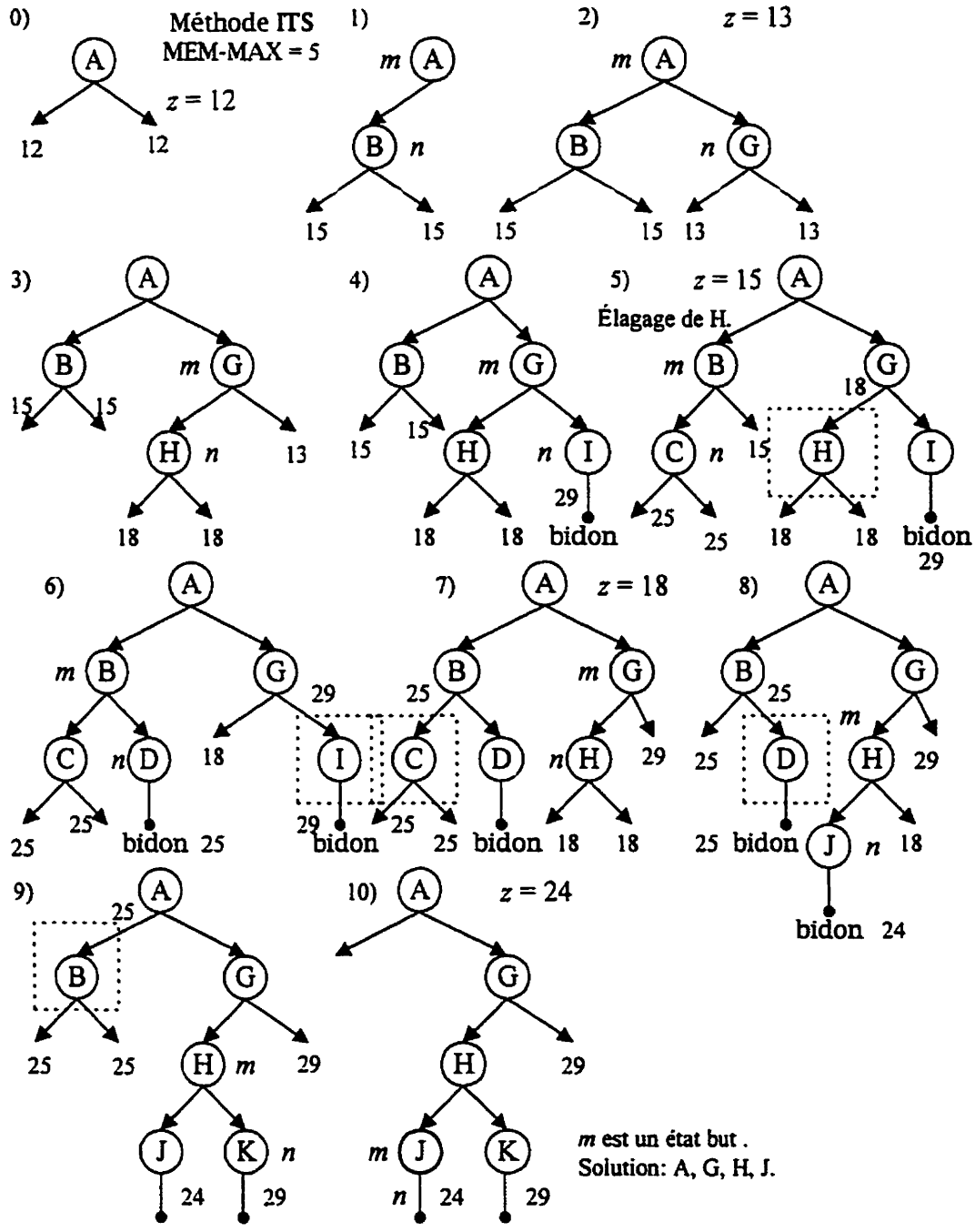


Figure 4.22 Application de la méthode ITS avec MEM-MAX = 5.

Dans l'exemple de la figure 4.22, la profondeur de fouille z a été réajustée à plusieurs reprises lors de la fouille. La méthode ITS développe toujours les états ayant un lien virtuel donc la valeur β est inférieure ou égale à z. Les changements de la valeur z ont eu lieu aux étapes 2, 5, 7 et 10. On peut considérer ces changements de z comme des descentes dans le graphe d'états. Chaque descente conduit la fouille à des profondeurs différentes et de plus en plus près de l'état but. La solution de la fouille a été obtenue à l'étape 10 et les états générés sont : A, B, G, H, I, C, D, H, J, K.

Dans cet exemple, le nombre d'états générés par la méthode ITS est plus élevé que celui de la méthode SMA*. Cependant, nous pouvons observer un avantage indéniable de la méthode ITS. En effet, à l'aide d'une modification mineure, cette méthode est capable de générer une solution partielle lorsque la mémoire disponible est insuffisante pour qu'elle obtienne une solution. Cette situation est montrée dans la figure 4.23 et 4.24. La mémoire disponible est MEM-MAX = 3 et la méthode de fouille ne pourra pas trouver une solution puisque le chemin de la solution contient 4 états.

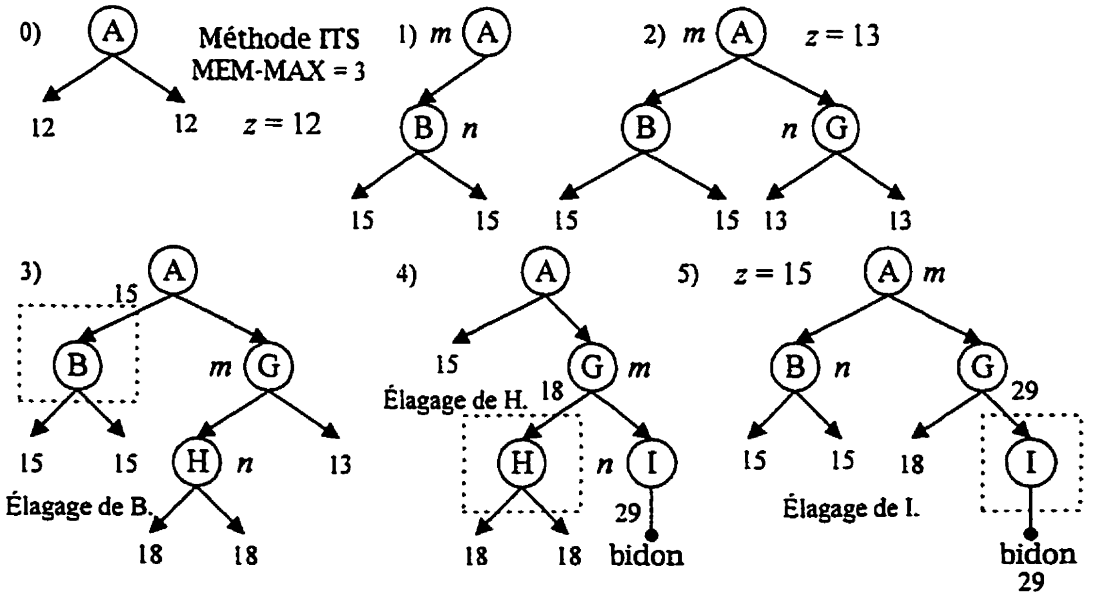


Figure 4.23 Application de la méthode ITS avec MEM-MAX = 3.

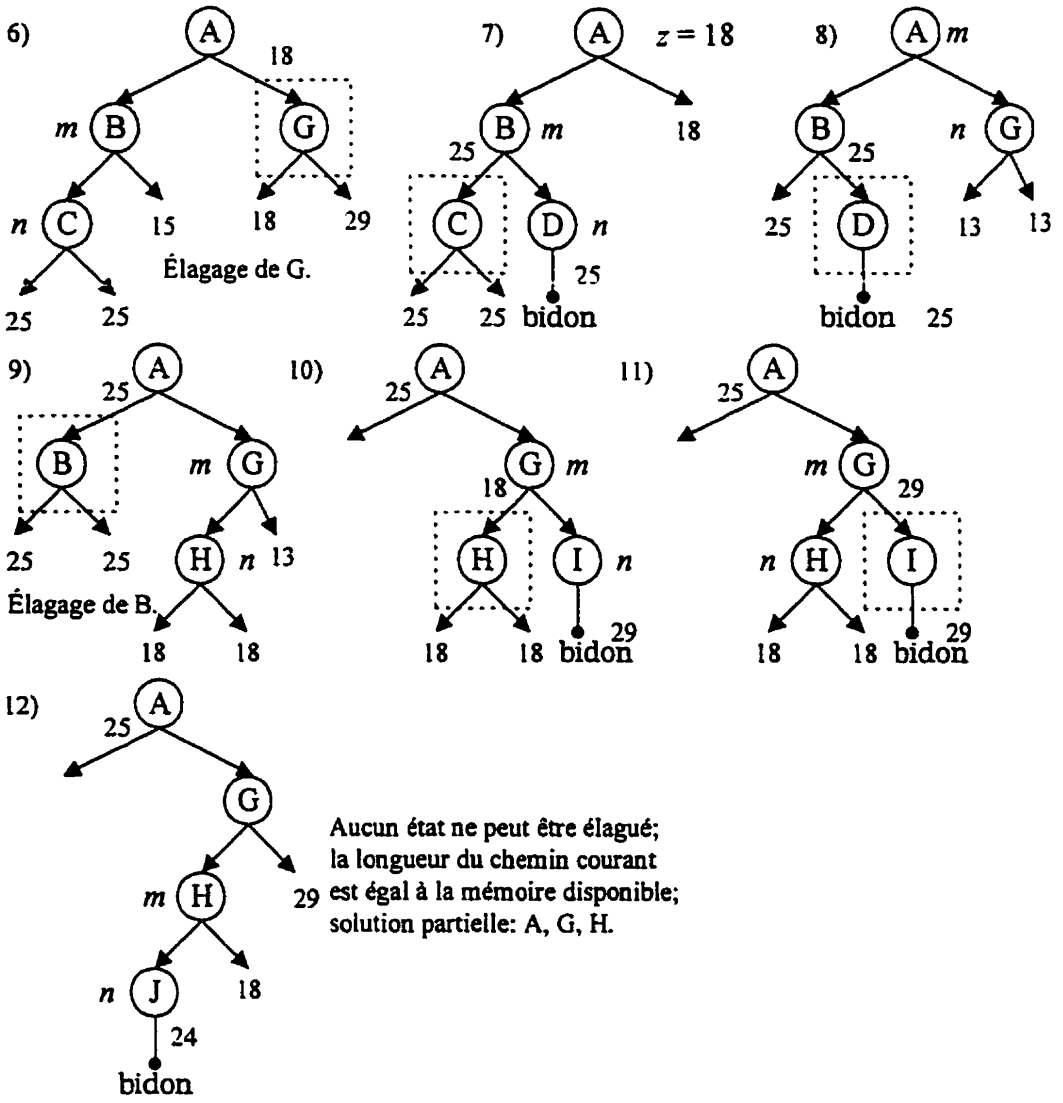


Figure 4.24 Suite de l'application de la méthode ITS avec MEM-MAX = 3.

À l'étape 12 de la figure 4.24, la méthode a développé l'état H et généré l'état J. La mémoire disponible est épuisée. La méthode doit élaguer un état possédant des liens virtuels. Or, aucun état de l'arbre implicite ne peut être éliminé puisque le seul état ayant un lien virtuel est l'état J et il n'est pas encore installé dans l'arbre. Donc la méthode doit s'arrêter. Les états emmagasinés en mémoire au moment de l'arrêt constituent au minimum une solution partielle. Il s'agit là d'un comportement naturel d'une fouille en

profondeur d'abord. La mémoire utilisée par la fouille ITS contient les chemins partiels de profondeur plus petite ou égale à z.

4.4 Synthèse d'une nouvelle méthode

La méthode Répartition présentée dans la section 4.2.2 est une méthode qui garantie l'obtention d'un horaire d'exécution complet. Dans cette méthode de répartition des tâches, une fonction d'évaluation heuristique $f(n) = g(n) + h(n)$ est utilisée pour guider la sélection des assignations tâche-processeur. Le monoïde de valeurs est donc $(\mathcal{R}^+, +)$. Le terme $g(n)$ représente le coût déboursé depuis l'assignation initiale jusqu'à l'assignation n. Le terme heuristique $h(n)$ représente une estimation du coût encore à débourser de l'assignation n jusqu'à la fin de la répartition des tâches. Si un horaire complet existe, alors son obtention est garantie par cette méthode (théorème 4.2.3.2). Cependant elle possède une complexité spatiale qui est exponentielle. Son implantation pratique n'est possible que pour des graphe d'états de très petite taille à cause du nombre d'états conservés dans les listes OUVERT et FERMÉ (Figure 4.5). Pour des applications pratiques, nous devons limiter l'utilisation de la mémoire en imposant une taille fixe à ces listes. La taille de ces listes est alors considérée comme la mémoire disponible pour effectuer la fouille heuristique.

La méthode SMA* [RUS92] réalise une telle méthode de fouille heuristique. Cette méthode de fouille heuristique utilise une seule liste (c'est-à-dire, la liste OUVERT) et sa taille est limitée à une valeur fixe MAX-MEM. Sa performance est comparable à celle de Répartition si la taille de OUVERT est suffisante pour entreposer tous les états générés pour le chemin menant vers un horaire complet. Si non, SMA* produira une erreur et son état final est indéterminé (voir l'exemple des figures 4.16 et 4.17).

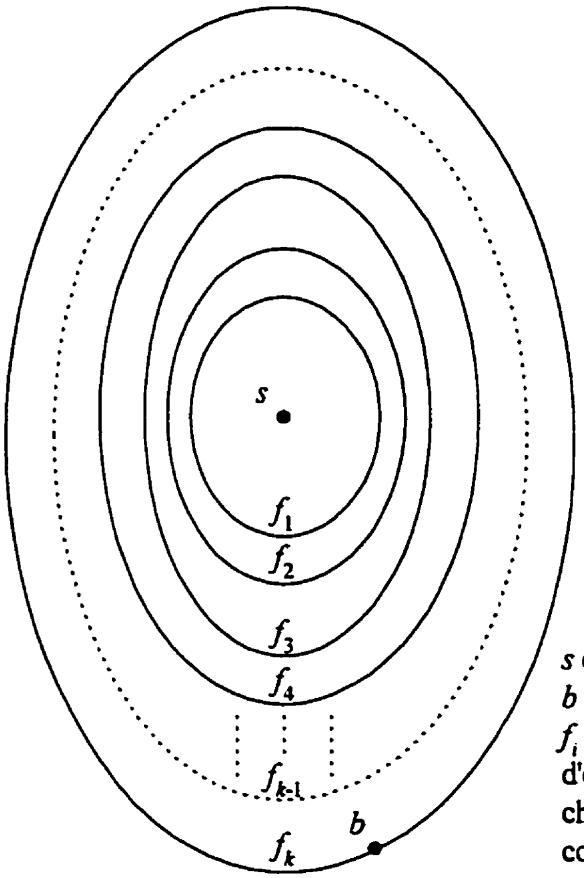
Une approche différente consiste à utiliser une liste de taille fixe pour emmagasiner uniquement les assignations des chemins parcourus. C'est le principe de

fonctionnement de la méthode ITS [GHO94]. Cette méthode est essentiellement une fouille en profondeur d'abord avec comme seuil de profondeur la valeur donnée par l'application de la fonction d'évaluation heuristique $f(n) = g(n) + h(n)$. Elle aussi utilise un monoïde de valeurs ($\mathcal{R}^+$, +) et présente une performance comparable à une fouille en profondeur d'abord classique si la mémoire disponible est suffisante pour entreposer toutes les assignations de l'horaire complet. Cependant, le nombre de régénération des assignations de ITS est plus élevé que la méthode SMA*. La méthode ITS peut donner des horaires partiels à cause de la nature en profondeur d'abord de la fouille (voir l'exemple des figures 4.23 et 4.24). Ces horaires partiels sont les assignations conservées immédiatement avant l'épuisement de la mémoire disponible. Un arbre implicite est construit par la méthode ITS lors de la fouille. À cause de la structure des arborescences, la méthode ITS peut débuter une fouille à partir d'un état quelconque de l'arbre de fouille.

Il a été démontré que dans le chapitre 3, le problème de la répartition des tâches est un problème de la classe **NP-complet**. La répartition des tâches par ordinateur est donc un problème très difficile à réaliser. Il est donc naturel de chercher une solution qui ne sera pas nécessairement optimale. La technique heuristique représentée par la méthode Répartition est excellente mais elle exige une quantité énorme de mémoire. Les techniques heuristiques représentées par les méthodes SMA* et ITS sont des candidates intéressantes. Or, la méthode SMA* nécessite une liste OUVERT de taille plus grande ou égale à $\aleph$ (voir définition 4.2.5.2). La valeur de $\aleph$ n'est pas connue d'avance et on ne peut la connaître qu'une fois la répartition effectuée.

Pour la méthode ITS, elle demande une mémoire proportionnelle à la longueur du chemin menant à un horaire complet. Comme le nombre d'assignations dans un horaire complet est connu d'avance (c'est le nombre de tâches à répartir), on peut facilement borner la taille de la mémoire à utiliser. Or, si la quantité de mémoire est tout juste suffisante pour entreposer les assignations de l'horaire complet, alors la méthode ITS

doit régénérer un grand nombre d'états avant d'atteindre un état but. Cette méthode utilise la mémoire disponible pour entreposer les assignations d'un horaire partiel de longueur inférieure ou égale au profondeur de fouille *z*. Pour pouvoir fouiller d'autres parties du graphe d'états de même profondeur, elle doit élaguer un nombre d'assignations de la mémoire. La possibilité de régénération des assignations déjà rencontrées est donc grande. Par contre, dans le cas où la mémoire disponible est inférieure au nombre d'assignations de l'horaire complet ou si le graphe d'états n'est pas accessible (sans état but), la méthode ITS peut quand même produire des horaires partiels.



s est l'assignation initiale;
b est une assignation finale;
f_i sont les valeurs de la fonction d'évaluation heuristique dans le chemin menant vers un horaire complet.

Figure 4.25 Espace d'états contenant un horaire complet.

Une illustration intuitive du principe de fonctionnement de ces méthodes de fouille montrée dans la figure 4.25. Dans cette figure, s représente l'état initial et b représente un des états but. Un horaire complet comprend k assignations. Par conséquent, il y a k valeurs de la fonction d'évaluation heuristique qui seront retenues lors de la fouille. Pour satisfaire aux conditions d'arrêt avec solution, les valeurs f_i retenues forment une suite finie non décroissantes $f_1 \leq f_2 \leq \ldots \leq f_{k-1} \leq f_k$ (théorème 4.2.4.2). On peut donc tracer un ensemble de frontières, dans l'espace d'états, en fonction de ces valeurs heuristiques. Les ovales de la figure 4.25 représentent justement ces frontières délimitées par les différentes valeurs de f dans un horaire complet.

Dans cette représentation de l'espace d'états, les assignations qui sont situées entre deux frontières f_i et f_j ont une valeur heuristique $f_i < f < f_j$ pour $f_i < f_j$. Ces assignations intermédiaires situées entre deux frontières sont des assignations indésirables. Elles sont indésirables parce qu'elles ne répondent pas aux contraintes temporelles et spatiales imposées ou parce qu'elles mènent à un chemin sans issue. Donc, une fouille efficace doit pouvoir élaguer rapidement ces assignations intermédiaires. Les assignations situées directement sur une frontière donnée ont une valeur heuristique identique. C'est dans ce cas que la performance des heuristiques de génération des assignations devient importante. En effet, on doit pouvoir obtenir un ensemble de frontières dont le contour est le plus aplati possible. Ainsi, la fouille peut concentrer l'effort de recherche dans une bande étroite de l'espace d'états. On voit là l'importance des heuristiques utilisées dans la génération des assignations candidates pour la continuation de la fouille.

La méthode SMA* utilise la rétro-propagation des valeurs heuristiques f dans les ancêtres de l'état élagué pour délimiter l'espace de fouille. Cette rétro-propagation peut conduire la méthode à passer d'une frontière à l'autre rapidement. La fouille effectuée par SMA* est en largeur d'abord. Donc, plus les états indésirables sont élagués tôt dans la fouille, plus le balayage de l'espace d'états est efficace. Les états ancêtres contiennent suffisamment d'information des parties élaguées pour aider à la fouille même si la

méthode doit rebrousser chemin à cause d'un mauvais choix. Ce qui contribue à rétrécir davantage la bande de l'espace d'états délimitée par les heuristiques de génération.

Tableau 4.3 Caractéristiques des méthodes SMA* et ITS.

Caractéristique	Méthode	Commentaire
Nature de la fouille	SMA*	Cette méthode est une fouille en largeur d'abord avec évaluation heuristique $f(n)$ des états à développer.
	ITS	Cette méthode est une fouille en profondeur d'abord. La profondeur de fouille est continuellement ajustée par la fonction d'évaluation heuristique $f(n)$.
Élagage	SMA*	L'élagage est réalisé d'une manière explicite. Il y a rétro-propagation des valeurs de la fonction d'évaluation heuristique dans les ancêtres de l'état élagué.
	ITS	L'élagage est réalisé d'une manière explicite. La valeur de la fonction d'évaluation heuristique est propagée au parent de l'état élagué.
Arrêt de la fouille	SMA*	À cause de la rétro-propagation des valeurs, cette méthode peut terminer son exécution son exécution de deux façons : *i*) arrêt avec un horaire complet; *ii*) arrêt avec $f(s) = \infty$.
	ITS	La nature d'une fouille en profondeur d'abord permet à cette méthode de produire un horaire complet ou un horaire partiel.
Difficultés	SMA*	La mémoire nécessaire pour la fouille est difficile à déterminer. Lorsque la méthode s'arrête à cause d'une manque de mémoire, son état interne est indéterminé.
	ITS	Lorsque les valeurs heuristiques f des états sont éloignées les unes des autres, cette méthode doit développer un grand nombre d'états pour chacune des frontières de l'espace d'états.

Quant à la méthode ITS, elle repose sur la valeur heuristique *f* des états pour déterminer la profondeur *z* de la fouille. Dans le cas idéal, les valeurs de *z* peuvent correspondre à la valeur *f* des frontières et la méthode peut produire très rapidement un horaire complet. Mais normalement, la méthode ITS doit développer tous les états d'une certaine profondeur avant de passer à une profondeur différente. Dans le pire des cas, tous les états générés peuvent avoir une valeur heuristique *f* différente.

Pour la méthode ITS, cela signifie qu'un grand nombre de frontières supplémentaires existent dans l'espace d'états et elle doit les franchir une à une. Ainsi, l'efficacité de la méthode ITS dépend très fortement sur les valeurs de *f*. Plus les états possèdent une valeur *f* divergente, plus la méthode verra de frontières supplémentaires dans l'espace d'états. En pratique, il est très difficile de prévoir complètement le comportement des fonctions heuristiques. La fonction d'évaluation heuristique $f(n)$ peut produire des valeurs très rapprochées dans une région de l'espace d'états mais elles peuvent diverger considérablement ailleurs. On voit par cette discussion que la fouille en profondeur d'abord possède une difficulté qui n'est pas présente dans la méthode SMA*. Enfin, le tableau 4.3 de la page 134 est un résumé de quelques caractéristiques des méthodes SMA* et ITS.

4.4.1 Méthode combinée

Il est donc difficile d'appliquer directement la méthode SMA* ou la méthode ITS au problème de la répartition des tâches. Par contre, il est tout à fait possible de réunir les avantages de ces deux méthodes pour créer une nouvelle approche qui soit plus performante. Il n'est pas nécessaire de s'acharner sur le problème de la détermination de $\aleph$ pour la méthode SMA* et sur la difficulté de la méthode ITS pour des valeurs *f* divergentes. Nous pouvons utiliser une approche combinée de ces deux méthodes. Autrement dit, nous pouvons accorder une quantité de mémoire disponible à la méthode combinée pour le traitement du système des tâches. Si cette dernière obtient un horaire

complet alors la répartition est une réussite. Par contre, si la mémoire réservée ne suffit pas nous aurons quand même un horaire partiel. Dans ce cas, nous pouvons enregistrer l'horaire partiel sur disque et utiliser la dernière assignation trouvée comme état initial pour relancer la répartition des tâches pour des traitements subséquents.

Nous pouvons justifier cette approche combinée par une analogie avec la capacité d'arrêt automatique des postes de travail Unix. Ces ordinateurs Unix sont munis d'un sous-système de gestion d'énergie et peuvent s'arrêter complètement après une période d'inactivité de durée déterminée [SOL98]. Lors de l'arrêt automatique, ces ordinateurs enregistrent sur disque l'état de ses registres et les données qui sont en mémoire. Dès que ces informations sont écrites sur disque, une séquence de terminaison est exécutée pour mettre hors tension le système informatique. Après le redémarrage, ces ordinateurs rechargent en mémoire toutes les informations enregistrées afin de permettre aux utilisateurs de poursuivre leur travail. Les informations enregistrées contiennent uniquement l'état du processeur et l'état de la mémoire au moment de l'arrêt automatique. Autrement dit, seule la configuration des états du système au moment de l'arrêt est conservée. Même si l'ordinateur concerné a effectué un grand nombre d'opérations avant l'arrêt automatique, il n'est pas nécessaire d'enregistrer ces opérations. La configuration des états du système au moment de l'arrêt suffise pour reconstruire l'environnement de travail.

Il en est de même pour la méthode de fouille proposée. Dans cette approche, l'enregistrement des chemins partiels de la fouille correspond à la sauvegarde de la configuration des états des ordinateurs. Chaque chemin partiel enregistré peut être considéré comme une configuration d'un ordinateur en mode d'arrêt automatique. Même si la méthode de fouille a effectué un grand nombre de générations et de développements, la mémorisation du chemin partiel au moment de l'enregistrement suffise pour relancer la fouille. Donc, notre méthode consiste à effectuer une fouille efficace dans l'espace d'états et dès qu'un chemin partiel est obtenu, elle l'enregistre sur

disque. Ces chemins partiels seront examinés si la fouille initiale n'a pu produire un horaire complet. Si tel est le cas, la méthode poursuivra la fouille en utilisant les chemins partiels conservés. Les chemins partiels enregistrés peuvent engendrer d'autres chemins partiels. La méthode relancera la fouille sur ces nouveaux chemins partiels et ainsi de suite jusqu'à l'obtention d'un horaire complet.

Dans le cas où le système des tâches ne possède pas d'horaire complet. Nous aurons quand même les horaires partiels générés lors des fouilles. Ainsi, il est possible de suggérer les modifications à apporter au système de tâches et au réseau des processeurs pour que la répartition soit une réussite. Ces modifications seront formulées à l'aide d'une analyse postopératoire des chemins partiels finaux obtenus par la méthode de répartition des tâches. Cette analyse postopératoire sera présentée au chapitre 5.

4.4.2 Principe de la méthode combinée

Soit $\mathcal{T} = \{T_1, T_2, \ldots, T_x\}$ un ensemble de tâches et $\mathcal{P} = \{P_1, P_2, \ldots, P_y\}$ un ensemble de processeurs avec $y \geq x$. Un horaire d'exécution est une séquence d'assignations $\chi = \{a_1, a_2, \ldots, a_k\}$ où a_i sont les assignations tâche-processeur (T_i, P_j) formant l'horaire χ. $N_\chi = |\chi|$ est le nombre d'assignations dans χ. $N_\mathcal{T} = |\mathcal{T}|$ est le nombre de tâches dans $\mathcal{T}$. Un horaire partiel est un horaire dont $N_\chi < N_\mathcal{T}$. Si l'horaire χ est un horaire complet alors $N_\chi = N_\mathcal{T}$. Le principe de fonctionnement de la méthode combinée est donné dans la figure 4.26.

La méthode de répartition des tâches proposée est une méthode de fouille heuristique combinant les avantages des méthodes SMA* et ITS. Dans l'application de la méthode combinée, le balayage en profondeur d'abord de SMA* est utilisée pour éliminer le problème des valeurs divergentes de la fonction d'évaluation heuristique. Par contre, la technique de mise à jour des valeurs β des liens virtuels de ITS est utilisée pour réduire le coût lors des élagages normaux. La rétro-propagation des valeurs de la

méthode SMA* n'est effectuée que pour les états menant vers des chemins sans issue. La raison est que l'élagage d'un chemin sans issue doit être définitif. La rétro-propagation des valeurs vers les ancêtres d'un état sans issue permet la réalisation de cet élagage définitif.

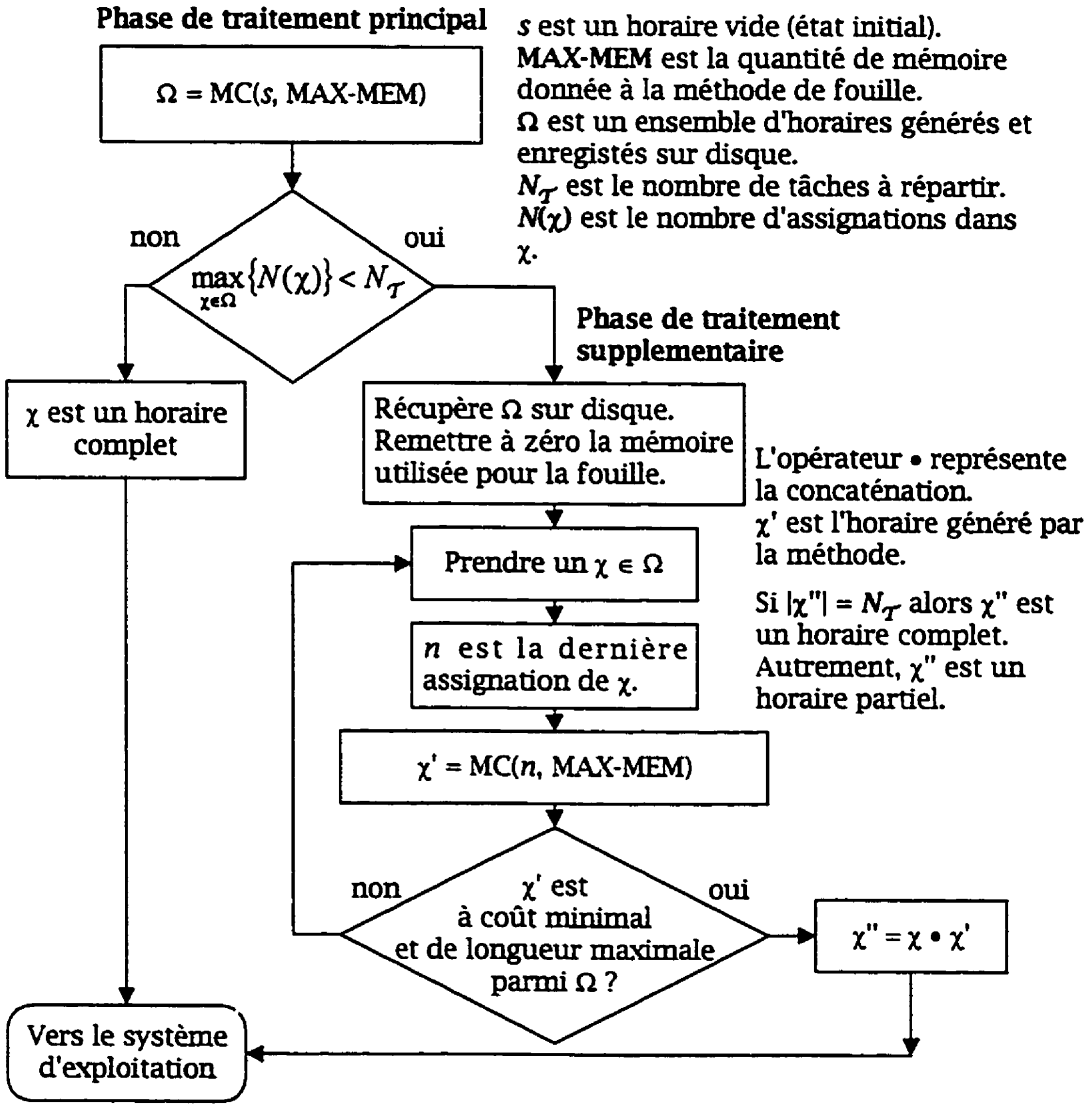


Figure 4.26 Principe général de la méthode combinée.

La méthode proposée tente d'obtenir un horaire d'exécution complet ou partiel avec comme état initial un horaire vide *s* et avec une quantité de mémoire maximale

MAX-MEM. La quantité de mémoire maximale doit être supérieure au nombre de tâches de l'ensemble T. Donc MAX-MEM > N_T. La phase de traitement principal consiste à exécuter la méthode de fouille avec un horaire vide s. À la fin de son travail elle doit générer un horaire χ' . Si $N_{\chi'} = N_T$ alors χ' est un horaire complet et la méthode de répartition des tâches a également terminé son travail. Les assignations de tâches contenues dans χ' seront utilisées par le système d'exploitation pour le chargement et l'exécution des programmes dans l'ordinateur parallèle. Par contre, si l'horaire χ' est tel que $N_{\chi'} < N_T$ alors, la méthode passe dans la phase de traitement supplémentaire.

Dans la phase de traitement supplémentaire, la méthode récupère les horaires partiels sur disque. Soit $\chi' = \{a_1, a_2, \ldots, a_{k'}\}$ un horaire partiel généré lors de la fouille initial. Remarquer que $N_{\chi'} < N_T$ est toujours vrai dans la phase du traitement supplémentaire. L'ensemble de la séquence d'assignations $a_1 \ldots a_{k'}$ est mémorisé sur disque et $a_{k'}$ devient l'état initial s de la phase de traitement supplémentaire, $s = a_{k'}$. Ainsi, l'état $a_{k'}$ et la valeur heuristique $f(a_{k'})$ sont nécessaires pour l'expansion de l'horaire partiel χ'. La méthode de fouille traitera tous les horaire partiels $\chi' \in \Omega$ pour obtenir un horaire complet.

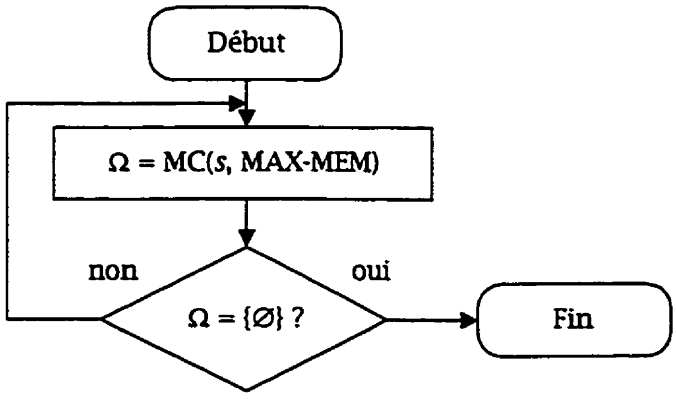


Figure 4.27 Procédure nécessaire pour l'expansion des horaires partiels.

À cause de la limitation de la mémoire disponible, les horaires partiels peuvent produire d'autres horaires partiels. Une procédure de bouclage est nécessaire pour vérifier les résultats intermédiaires de la méthode de fouille. Cette procédure est montrée dans la figure 4.27. La procédure bouclera tant qu'un horaire complet n'est pas obtenu et qu'il existe des horaires partiels encore à explorer.

4.4.3 Formulation de la méthode combinée

Si on examine de plus près le principe de fonctionnement des méthodes SMA* et ITS, on peut remarquer une similitude dans leur façon de gérer l'élagage des états. Il en est de même pour la rétention d'information nécessaire pour retrouver les états élagués.

Dans la méthode SMA*, les états à élagués sont ceux qui ont une valeur d'évaluation heuristique f maximale. De même pour la méthode ITS mais elle est sous la forme d'un seuil de profondeur z. Pour pouvoir régénérer de nouveau les états élagués, la méthode SMA* maintient une valeur f' dans tous les ancêtres d'un état élagué n. La valeur f' représente la partie du chemin minimal passant par n.

Dans la méthode ITS, cette valeur est maintenue dans les liens virtuels du parent dont l'enfant est élagué. À cause de la fouille en profondeur d'abord de la méthode ITS, cette dernière peut ranger les informations de régénération uniquement dans le parent de l'état élagué. Tandis que SMA* doit mettre les mêmes informations dans tous les ancêtres d'un état élagué car elle ressemble à une fouille en largeur d'abord.

Manifestement, il est possible de simplifier les étapes de la figure 4.26 pour tenir compte de ces similitudes entre SMA* et ITS. Plus spécifiquement, nous pouvons mettre en commun l'approche général d'une fouille à caractère *largeur d'abord* de SMA* et le concept de liens virtuels de ITS. La méthode SMA* est en mesure de trouver les horaires partiels en insérant les directives d'enregistrement dans les parties appropriées de la

méthode. Les liens virtuels de ITS permettent une meilleure coordination dans la génération des états et dans l'élagage des chemins sans issues. En effet, le coût de rétro-propagation de SMA* est beaucoup plus coûteux que la simple mise à jour de ITS. Ainsi, une fois un chemin sans issue est découvert, il est inutile de propager cette information dans tous les états ancêtres. Il suffit d'effectuer une mise à jour dans le style de ITS pour que le chemin sans issue demeure élagué.

Il existe deux phases de traitements distincts dans le principe de fonctionnement de la figure 4.26. La phase de traitement principal consiste à obtenir un horaire complet en utilisant la méthode de fouille proposée. Si la limitation de la mémoire l'empêche de produire un horaire complet, la méthode est relancée et elle tentera d'étendre les horaires partiels pour obtenir un horaire complet. Cette séparation entre les deux phases n'est pas vraiment nécessaire. En effet, il est possible de combiner les deux phases de traitement en une seule et de l'appliquer d'une manière répétitive tant et aussi longtemps qu'il existe des horaires partiels dans le problème.

Nous allons donner une nouvelle formulation de la méthode qui peut être réutilisée pour solutionner les horaires partiels générés. Nous avons donné à cette nouvelle formulation l'acronyme DPSM (Dynamic Prunning Scheduling Method). Les figures 4.28 à 4.31 présentent le pseudo-code de cette méthode de répartition des tâches.

```
   Fonction EXTRACTION(Liste Assignation A)
   {
       // Retourner un état de la liste A qui a une valeur heuristique f
       // minimale.

       Assignation a;
1      retourne a = min {a.f}, ∀ a dans A;
   }
```

Figure 4.28 Fonction d'extraction de la méthode DPSM.

```
    Fonction GÉNÉRATION(Assignation A)
    {
        // Retourner un successeur de A qui n'est pas en mémoire et
        // n'est pas relié à A par un lien virtuel dont  β = ∞.

        Assignation a;
1       a = (SUCC(A) && β_{A,a} != ∞ && (a ∉ OUVERT) && (a ∉ FERMÉ));
2       si (a) alors              // Tout comme ITS, on doit aussi ajouter les
3           retourne a;           // liens virtuels initiaux de a dans l'arbre. Ces
4       autrement                 // opérations ne sont pas montrées ici pour
5           retourne bidon;       // simplifier la présentation.
    }
```

Figure 4.29 Fonction de génération d'une assignation.

```
    Fonction SANS-ISSUE(Assignation m, Assignation n)
    {
        // Traitement dans le cas où n est un état sans issue. L'état m est le
        // parent de n.

        // Mettre à jour le lien virtuel entre le parent de m et m.
1       β m.prédécesseur,m = ∞;
2       ENREGISTRE-HORAIRE(s, m, FINAL); // enregistre le chemin partiel
        /*  si prédécesseur de m est dans FERMÉ et qu'il possède un lien
            virutel encore à développer alors le remettre dans OUVERT */
3       si ((m.prédécesseur ∈ FERMÉ) && (∃ β_{p,q} < ∞)) alors {
4           OUVERT ← OUVERT + {m.prédécesseur};
5           FERMÉ ← FERMÉ - {m.prédécesseur};
        }
6       EFFACER(n); EFFACER(m); // élaguer ces états de la mémoire
    }
```

Figure 4.30 Traitement d'une assignation sans issue.

Le pseudo-code de la méthode DPSM est montré dans la figure 4.31. L'extraction des assignations est toujours réalisée à l'aide d'une assignation avec la plus petite valeur de f (voir la fonction d'extraction, figure 4.28). La génération d'une assignation dépend

des opérateurs de génération (fonction SUCC de la figure 4.30) et des conditions intrinsèques à la méthode proposée.

En effet, l'assignation candidate ne doit pas mener vers un chemin sans issue et ne doit pas être déjà en mémoire. C'est pour cette raison que l'on vérifie la valeur β du lien virtuel reliant l'assignation prédécesseur et l'assignation candidate. Une assignation candidate mène vers un chemin sans issue si elle est reliée à son parent par un lien virtuel dont $\beta = \infty$.

```
    Fonction MEM-SANS-ISSUE(Assignation s, Assignation m, Assignation n)
    {
        // Traitement dans le cas où le chemin passant par n a une
        // longueur égale à quantité de mémoire disponible. L'état s est
        // l'état initial de la fouille. L'état m est le parent de n.

1       ENREGISTRE-HORAIRE(s, n, NON-FINAL);
2       β_m,n = ∞;
3       EFFACE(n);
    }
```

Figure 4.31 Traitement d'une assignation sans issue par manque de mémoire.

Lorsqu'une assignation *m* génère un successeur nul, cela signifie que l'assignation *m* mène à un horaire partiel (ligne 9, figure 4.30). Dans ce cas, nous pouvons enregistrer l'horaire partiel sur disque et éliminer l'assignation de la mémoire. Puisque l'ensemble des successeurs de l'assignation *m* est un ensemble vide, on ne peut étendre davantage cet horaire partiel. L'horaire partiel enregistré est un horaire partiel final. Ces horaires partiels finaux sont des données qui seront utilisés dans l'analyse postopératoire de la méthode. Comme il a été indiqué dans la section 3.2 du chapitre 3, le répartiteur de tâches doit pouvoir proposer une nouvelle architecture d'interconnexions du système des processeurs si la topologie d'interconnexions initiales ne convient pas. En analysant, les horaires partiels finaux obtenus, nous pouvons dégager les modifications nécessaires.

Appliquées au système des processeurs, ces modifications permettront une répartition réussite du système des tâches.

```
    Méthode DPSM(Assignation s, entier MEM-MAX)
    {
        Assignation m, n;
        entier usage;

1       OUVERT ← {s}; usage ← 1;
2       FERMÉ ← {∅};
3       répéter {
4           si OUVERT = {∅} alors { HORAIRE(0); arrêt("Échec")};
5           m ← EXTRACTION(OUVERT);
6           si m ∈ T alors { HORAIRE(m); arrêt("Succès"); } // solution !
7           n ← GÉNÉRATION(m);
8           /*  Pour les chemins sans issues                          */
9           si n = BIDON alors { // ajuste la valeur du lien entre m et son
10              SANS-ISSUE(m, n); usage ← usage - 1;  // prédécesseur
11          } autrement {
12              n.f ← max(m.f, g(n)+h(n));      // pathmax
                /*  Pour traiter le cas où la longueur d'un chemin atteint la
                    limite de la mémoire. Inutile de poursuivre la recherche
                    dans cette direction                                */
13              si (LONGUEUR(s, n) = MEM-MAX) alors {
14                  MEM-SANS-ISSUE(s, m, n);
15              } autrement {
16                  si usage > MEM-MAX alors {
17                      ÉLAGAGE(OUVERT); usage ← usage - 1;
                    }
18                  OUVERT ← OUVERT + {n}; // placé en ordre décroissante
19                  usage ← usage + 1;
                }
                /*  Est-ce que tous les successeurs de m sont traités ?
                    Vérifier si un état est complètement développé. Un état
                    est complètement développé si tous ses successeurs
                    sont générés.                                       */
20              si COMPLET(m) alors  COMPLET-SANS-ISSUE(m);
            }
        }
    }
```

Figure 4.32 Pseudo-code de la méthode DPSM.

Une assignation peut aussi conduire à un horaire partiel si le nombre d'assignations dans l'horaire partiel est égal à la quantité de la mémoire réservée pour le stockage des états. Dans ce cas, l'horaire partiel est non final et peut être étendu par une application subséquente de la méthode de fouille. Nous devons également enregistrer sur disque les horaires partiels non finaux. Le traitement des horaires partiels non finaux est donné dans la routine de la figure 4.32.

Le nombre d'horaires partiels générés peut être très élevé. Certains de ces horaires partiels sont inutiles pour l'analyse postopératoire ou pour l'expansion subséquente. Par exemple, pour un système à 40 tâches, un horaire partiel χ_1 comprenant 20 assignations est plus intéressant qu'un horaire partiel χ_2 à 10 assignations. Parce que χ_1 est plus près de l'horaire complet. De même, un horaire partiel non final est plus intéressant qu'un horaire partiel final parce que le premier peut être étendu par une application subséquente de la méthode de fouille. Ainsi, la routine d'enregistrement est une heuristique qui ne conserve que les horaires partiels les plus intéressants.

L'heuristique d'enregistrement des horaires partiels sont données dans la routine de la figure 4.33. Lorsque l'horaire partiel χ est un horaire final, il n'est conservé que s'il contient le plus grand nombre d'assignations. S'il existe un ou plusieurs horaires finaux de même longueur, on ne conserve que celui qui possède la plus petite valeur heuristique *f*. En effet, plus la valeur de *f* est petite plus l'horaire partiel est considérée comme adéquate pour la répartition des tâches.

Cependant, les horaires partiels finaux ne sont pas extensibles, nous devons tenter d'obtenir un horaire partiel final comprenant le plus grand nombre d'assignations possible pour faciliter l'analyse postopératoire. Lorsque l'horaire partiel χ est un horaire non final, il n'est conservé que s'il contient un nombre d'assignations supérieur ou égal aux horaires partiels non finaux déjà enregistrés. Si la longueur de χ est supérieure à celle des horaires non finaux déjà enregistrés alors on peut effacer tous les horaires

stockés. Ainsi, nous aurons toujours un horaire partiel contenant le plus grand nombre d'assignations possible.

```
Fonction ENREGISTRE-HORAIRE(Assignation s, Assignation m,
                            CheminType type)
{
    // Heuristique pour l'enregistrement des horaires partiels.
    // s est l'assignation départ, m est l'assignation courante, type
    // indique si l'horaire partiel est final ou non.
    Entier N1, N2; Réel f;

    N1 = LONGUEUR(s, m);
    cas (type) {
        FINAL:      // Trouver la longueur maximale des horaires
                    // finaux déjà enregistrés.
                    N2 = LONGUEUR-MAX-CHEMIN-FINAL-DISQUE( );
                    si (N1 > N2) alors {
                        // effacer tous les horaires finaux de
                        // longueur ≤ N2
                        EFFACE-CHEMIN-FINAL-DISQUE (N2);
                        ENREGISTRE-SUR-DISQUE(s, m, m.f, type);
                    } autrement si (N1 = N2) alors {
                        // Ne conserve que les horaires finaux de
                        // valeur heuristique minimale.
                        f = VALEUR-MIN-CHEMIN-FINAL(N2);
                        si (m.f ≤ f) alors {
                            EFFACE-CHEMIN-FINAL-DISQUE(N2);
                            ENREGISTRE-SUR-DISQUE(s, m, m.f, type);
                        }
                    }
        NON-FINAL:  // Trouver la longueur maximale de tous
                    // horaires déjà enregistrés.
                    N2 = LONGUEUR-MAX-CHEMIN-DISQUE( );
                    si (N1 ≥ N2) alors {
                        // effacer tous les horaires de longueur ≤ N2
                        si (N1 > N2) alors
                            EFFACE-CHEMIN-DISQUE(N2);
                        ENREGISTRE-SUR-DISQUE(s, m, m.f, type);
                    }
    }
}
```

Figure 4.33 Heuristique pour la sauvegarde des horaires partiels.

Une assignation est complète si tous ses successeurs sont générés. La routine COMPLET-SANS-ISSUE (figure 4.34) vérifie l'état d'une assignation lorsque cette dernière est complète. Si l'assignation s'avère sans issue, alors on peut l'éliminer de la fouille. Une assignation est sans issue lorsque tous ses liens virtuels ont une valeur $\beta = \infty$. On doit également vérifier s'il est nécessaire de remettre le parent de l'assignation complète dans la liste OUVERT. Cette étape est nécessaire pour ne pas découper la partie du graphe d'états qui contient encore des chemins non explorés. Les assignations complètes avec successeurs sont déplacées de la liste OUVERT à la liste FERMÉ. Ce déplacement est nécessaire pour permettre le développement d'une autre assignation de la liste OUVERT.

```
    Fonction COMPLET-SANS-ISSUE(Assignation m)
    {
        // Vérifier si un état mène vers des chemins sans issue. Si oui,
        // éliminer l'état de la mémoire et remettre son parent dans
        // OUVERT s'il y a lieu. Autrement, placer l'état dans FERMÉ.

1       si min {β_m,r} = ∞, ∀ liens virtuels de m alors {
2           RÉTRO-PROPAGATION(m, ∞);
            // si le parent possède encore des successeurs à développer ...
3           si ((m.prédécesseur ∈ FERMÉ) && (∃ βp,q < ∞)) alors {
4               OUVERT ← OUVERT + {m.prédécesseur};
5               FERMÉ ← FERMÉ - {m.prédécesseur};
            }
6           EFFACE(m);
7       } autrement {
8           OUVERT ← OUVERT - {m}; FERMÉ ← FERMÉ + {m};
        }
    }
```

Figure 4.34 Routine de traitement pour les assignations complètement développées.

On voit par les figures 4.31, 4.32 et 4.34 que l'élagage de la méthode DPSM est beaucoup moins conservateur que les méthodes SMA* et ITS. Il est permis d'effectuer un élagage plus agressif puisque les horaires partiels intéressants sont enregistrés sur

disque. On peut donc libérer de la mémoire les assignations considérées comme sans issue par la méthode DPSM.

Enfin, deux possibilités existent à la fin des applications de la méthode de répartition des tâches. La première est l'obtention d'un horaire complet contenant un nombre d'assignations égal à la cardinalité du système des tâches. La seconde possibilité est l'obtention d'un ensemble d'horaires partiels finaux. Ces horaires partiels finaux ont une longueur identique mais inférieure à la cardinalité du système des tâches. Les horaires partiels seront examinés par le module d'analyse postopératoire pour dégager une alternative à la topologie d'interconnexions du système des processeurs.

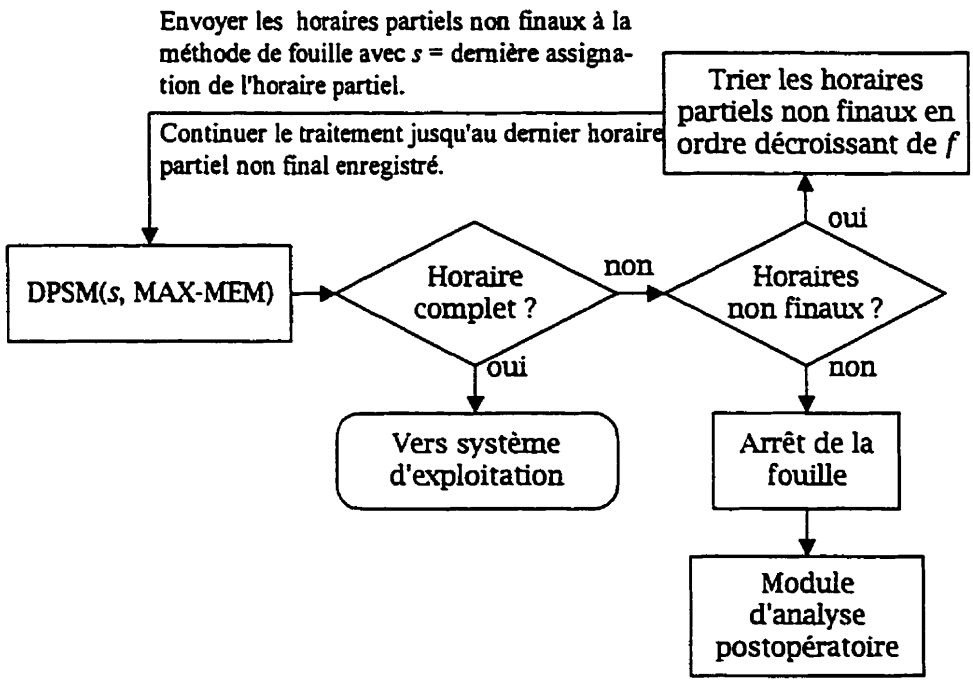


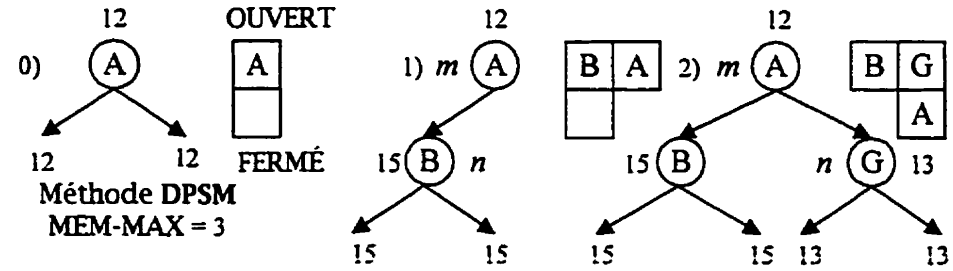
Figure 4. 35 Stratégie de contrôle pour la méthode de répartition des tâches.

La stratégie de répartition des tâches utilisant la méthode de fouille heuristique DPSM est donnée dans la figure 4.35. Dans cette stratégie, les horaires partiels non finaux générés par la fouille sont triés en fonction de la valeur heuristique *f*. Le contrôle est retourné à la méthode de fouille avec comme entrée la dernière assignation de

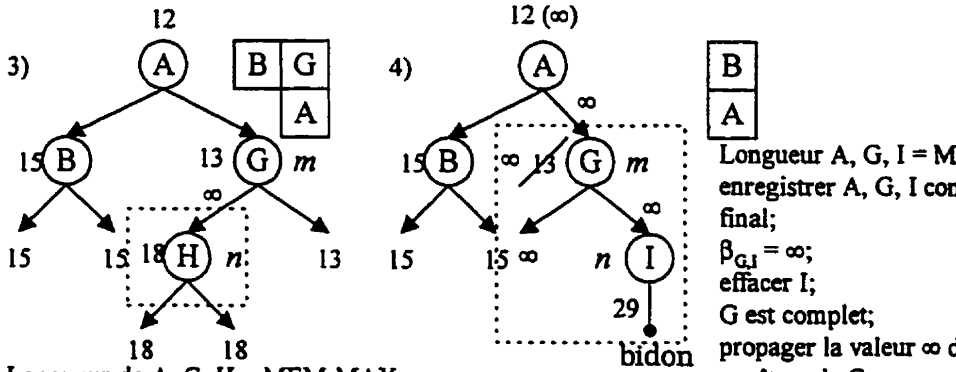
l'horaire partiel ayant la plus petite valeur heuristique *f*. Tous les horaires partiels non finaux sont ainsi traités un à un. Un second ensemble d'horaires partiels non finaux peut être généré à la fin des traitements. Si tel est le cas, la boucle de contrôle est de nouveau relancée. Si l'ensemble des horaires non finaux est vide alors on passe le contrôle au module d'analyse postopératoire. Cela signifie que la méthode est incapable de générer un horaire complet parce que le problème de répartition n'a pas de solution.

Un exemple illustrant le principe de fonctionnement de la méthode DPSM est présenté ci-dessous. La figure 3.36 est une application de la méthode DPSM au graphe d'états de la figure 4.14 à la page 115. La mémoire disponible pour la fouille est MEM-MAX = 3. Dans les étapes 3, 4, 5 et 6 de la figure 3.36, des horaires partiels sont enregistrés sur disque. Tous ces horaires partiels sont jugés non finaux par la méthode de fouille. Ils seront donc réexaminés par une application subséquente de la fouille. L'élagage de cette méthode est beaucoup plus agressif. Dans les étapes 4 et 6, une assignation complète (celle dont tous ses successeurs sont générés) est automatiquement éliminée de la mémoire. C'est pour cette raison qu'elle peut détecter la fin de la fouille seulement après 7 étapes.

Dans cet exemple aucun horaire partiel final n'est généré. Les horaires partiels non finaux générés lors de la fouille sont: *i*) A, G, H avec une valeur heuristique $f(H) = 18$; *ii*) A, B, C avec une valeur heuristique $f(C) = 25$; *iii*) A, B, D, $f(D) = 25$; *iv*) A, G, I avec une valeur heuristique $f(I) = 29$. Tous ces horaires partiels ont trois assignations, ils ont donc la même longueur. De plus, ils sont tous non finaux. Donc, ces quatre horaires partiels seront étendus dans les prochaines applications de DPSM. Ces horaires partiels sont triés en ordre décroissant en fonction de leur valeur heuristique. L'horaire partiel contenant les assignations A, G et H est sélectionné pour effectuer une seconde application de la fouille. La figure 4.37 donne le résultat de cette seconde application.

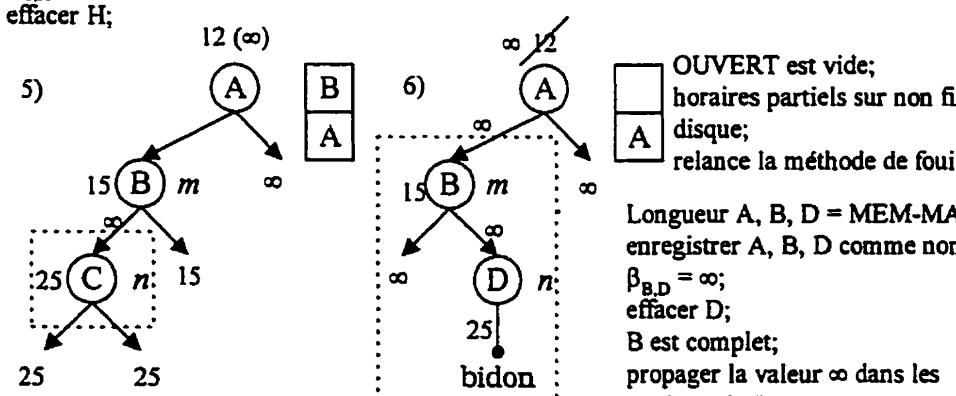


A est complet;
enlever A de OUVERT;
placer A dans FERMÉ.



Longueur de A, G, H = MEM-MAX;
enregistrer A, G, H comme non final;
$\beta_{G,H} = \infty$;
effacer H;

Longueur A, G, I = MEM-MAX;
enregistrer A, G, I comme non final;
$\beta_{G,I} = \infty$;
effacer I;
G est complet;
propager la valeur ∞ dans les ancêtres de G;
effacer G.



Longueur A, B, C = MEM-MAX;
enregistrer A, B, C comme non final;
$\beta_{B,C} = \infty$;
effacer C;

Longueur A, B, D = MEM-MAX;
enregistrer A, B, D comme non final;
$\beta_{B,D} = \infty$;
effacer D;
B est complet;
propager la valeur ∞ dans les ancêtres de B;
effacer B.

Horaires partiels non finaux enregistrés:
A, G, H, f(H) = 18;
A, B, C, f(C) = 25;
A, B, D, f(D) = 25;
A, G, I, f(I) = 29;

Figure 4.36 Application de la méthode DPSM avec MEM-MAX = 3.

L'assignation de départ de cette seconde application est H avec une valeur heuristique f(H) = 18. L'assignation finale est trouvée à l'étape 3. Un horaire complet est alors obtenu par la concaténation des assignations A, G, H et de l'assignation courante J. Il n'est plus nécessaire d'examiner les autres horaires partiels non finaux parce que la première solution trouvée est nécessairement une solution minimale.

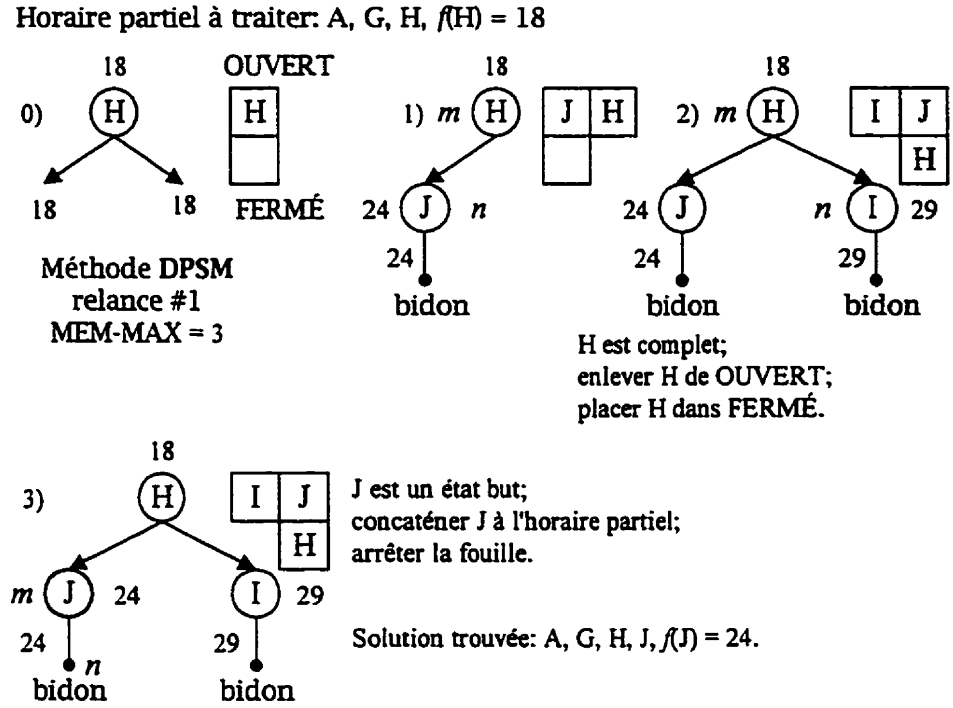


Figure 4.37 Seconde application de la méthode DPSM avec MEM-MAX = 3.

4.4.4 Propriétés de la méthode DPSM

Cette section présente les propriétés importantes de la méthode DPSM. Ces propriétés décrits d'une manière formelle le comportement et les capacités de cette méthode de fouille. Nous débutons la présentation des propriétés par deux définitions et un théorème qui aideront dans la suite des explications.

Définition 4.4.4.1 Une fonction d'évaluation heuristique $f(n)$ est *monotone* si $\forall n$, $f(n) \leq f(S(n))$, où $S(n)$ est un successeur de n.

■

Définition 4.4.4.2 Une fonction d'évaluation heuristique $f(n)$ est *admissible* si $\forall n$, $f(n) \leq f^*(n)$.

■

Théorème 4.4.4.1 *[KOR85] Pour toute fonction d'évaluation heuristique admissible, il est possible de construire une fonction heuristique monotone $f'(n)$ qui est aussi performante que $f(n)$.*

Démonstration Par construction. Si n est un état initial, alors $f'(n) = f(n)$. Autrement $f'(S(n)) = \max(f(S(n)), f'(n))$. Ainsi, $f'(n)$ est monotone car $f'(n) \leq f'(S(n))$. La fonction $f'(n)$ est également admissible car $f'(n)$ est égale à la valeur maximale de f appliquée à tous les prédécesseurs de n. Puisque f est bornée par $f^*(n)$, $f'(n)$ est aussi une borne inférieure sur le coût de l'état n.

□

Propriété 4.4.4.1 La méthode DPSM développe toujours l'état avec une valeur heuristique f minimale.

■

Les états à développer sont puisés de la liste OUVERT. Les états dans cette liste sont maintenus en ordre décroissant. La fonction EXTRACTION (figure 4.28, page 141) s'assure que cette propriété est respectée.

Propriété 4.4.4.2 L'élagage d'un état sans issue ne change pas l'ordre de développement des états tel qu'indiqué par la propriété 4.4.4.1.

■

Un état sans issue *n* est le dernier état d'un chemin qui ne peut mener à un état but. Son élagage ne changera pas la solution finale. De plus, son élimination ne peut changer l'ordre de développement des états puisqu'il ne sera jamais régénéré pour l'application en cours de la fouille. La valeur $\beta = \infty$ donnée au lien virtuel reliant l'état *n* et son parent interdit sa régénération (figure 4.30, page 142).

Propriété 4.4.4.3 L'élagage d'un état *n* dont LONGUEUR(*s*, *n*) ≥ MEM-MAX ne doit pas changer l'ordre de développement des états tel qu'indiqué par la propriété 4.4.4.1.

■

Lorsque LONGUEUR(*s*, *n*) ≥ MEM-MAX cela signifie qu'il n'y a pas assez de mémoire disponible pour les successeurs de *n*. L'élagage de *n* ne change pas l'ordre des états de la liste OUVERT puisqu'il n'est pas installé dans la liste au moment de son élagage.

Propriété 4.4.4.4 L'élagage d'un état complet *n* avec $\beta_{n,q} = \infty$ ou $\forall q$ successeur de *n* ne change pas l'ordre de développement des états tel qu'indiqué par la propriété 4.4.4.1.

■

Un état *n* est complet lorsque tous ses successeurs sont générés. Le fait que tous ses liens virtuels ont une valeur $\beta = \infty$ signifie que ses successeurs sont soit sans issue ou soit qu'il n'y ait pas suffisamment de place pour les entreposer en mémoire. Selon la propriété 4.4.4.2 et la propriété 4.4.4.3, les états successeurs de *n* peuvent être élagués

sans changer l'ordre de développement de la liste OUVERT. L'élimination de n ne peut changer l'ordre de développement des états puisqu'il ne sera jamais régénéré pour l'application en cours de la fouille. La valeur $\beta = \infty$ donnée au lien virtuel reliant l'état n et son parent lors de la rétropropagation des valeurs interdit sa régénération (figure 4.34, page 147).

Propriété 4.4.4.5 La méthode DPSM développe tous les états de même valeur heuristique f avant de développer les états de valeur f supérieure.

■

Sans aucun élagage, la propriété 4.4.4.5 est vérifiée directement par la fonction d'extraction de la figure 4.28 de la page 141. Dans la méthode DPSM, un état n est éliminé de la mémoire si: *i*) n est sans issue; *ii*) la longueur du chemin passant par n excède la mémoire disponible; *iii*) n est complet. Or, les propriétés 4.4.4.2 à 4.4.4.4 nous indiquent que ces élagages ne changent pas l'ordre de développement des états de la liste OUVERT. Donc, la méthode DPSM développe toujours les états de même valeur f avant de développer ceux qui ont une valeur f supérieure.

Propriété 4.4.4.6 L'enregistrement des solutions partielles sur disque ne change pas la nature des solutions obtenues par la méthode de fouille.

■

Soit $\chi = (s, \ldots, n_D, \ldots, n_F)$ une suite ordonnée d'états où s est l'état initial, n_F est le dernier état de la suite. L'ensemble des successeurs d'un état n est noté $\Gamma(n)$. Prenons un ensemble de k sous-suites de χ avec $\chi_i = (n_i, \ldots, n_{iE})$ pour $1 \leq i \leq k$. Pour LONGUEUR(χ_i) > LONGUEUR(χ_j), $\forall j \neq i$ nous conservons la sous-suite χ_i puisqu'elle contient un plus grand nombre d'états. Pour LONGUEUR(χ_i) = LONGUEUR(χ_j) alors nous conservons les sous-suites où $\Gamma(n_{iE}) \neq \{\varnothing\}$. Il s'agit là un résumé de la politique d'enregistrement des horaires partiels de la méthode DPSM. Cette procédure d'enregistrement est exécutée à chaque application de la fouille. Si la suite χ représente la séquence d'assignations d'un

horaire complet alors nécessairement $f(s) \leq \ldots \leq f(n_D) \leq \ldots \leq f(n_F)$ où n_F est un élément de l'ensemble des états but.

Pour la première application, $n_i = s$. Pour les applications subséquente de la fouille, l'état n_i est nécessairement différent des applications précédentes. Les applications subséquentes de la méthode de fouille sont réalisées en utilisant la dernière assignation des horaires partiels non finaux enregistrés. Donc, pour la m^{ξ} application de DPSM, $n_i^{(m)} = n_{iE}^{(m-1)}$ où $n_{iE}^{(m-1)}$ est la dernière assignation des horaires partiels non finaux de l'application précédente.

La fonction d'évaluation heuristique $f(n)$ est monotone alors $f(n_{iE}^{(m-1)}) \leq f(n_i^{(m)})$. Par la propriété 4.4.4.5 de la méthode DPSM, nous avons $f(n_i^{(m)}) \leq f(n'^{(m)}_i)$ où $n'^{(m)}_i \in \Gamma(n_i^{(m)})$ pour $\forall i$ et $\forall m$. Si la fonction d'évaluation heuristique $f(n)$ est admissible alors $f(n'^{(m)}_i) \leq f^*(n'^{(m)}_i)$ pour $\forall i$ et $\forall m$. Donc la concaténation des horaires partiels non finaux est un ensemble de séquences d'assignations ordonnées $\chi_i = (s, \ldots, n_i^{(m)}, \ldots, n_{iE}^{(m)}, \ldots, n_i^{(m+p)}, \ldots, n_{iE}^{(m+p)})$. Si un horaire complet existe alors l'une des séquences de χ_i aura comme dernière assignation un élément de l'ensemble des états but.

Enfin, la figure 4.38 donne une illustration de l'effet des applications successives de la méthode DPSM. Lors de la première application, la méthode produit 2 horaires partiels non finaux. À la deuxième application, 7 horaires partiels finaux et 1 horaire partiel non final sont générés. Les horaires partiels de l'application #1 sont effacés puisqu'ils sont contenus dans ceux de la deuxième application. À la troisième application, deux nouveaux horaires partiels non finaux sont obtenus. Tous les horaires partiels de l'application #2 sont effacés puisque ceux de l'application #3 contiennent un plus grande nombre d'assignations. Un horaire complet est trouvé à la quatrième

application de la méthode de fouille. Ainsi, chaque application de la méthode DPSM est un découpage de l'espace d'états fouillé par l'application précédente.

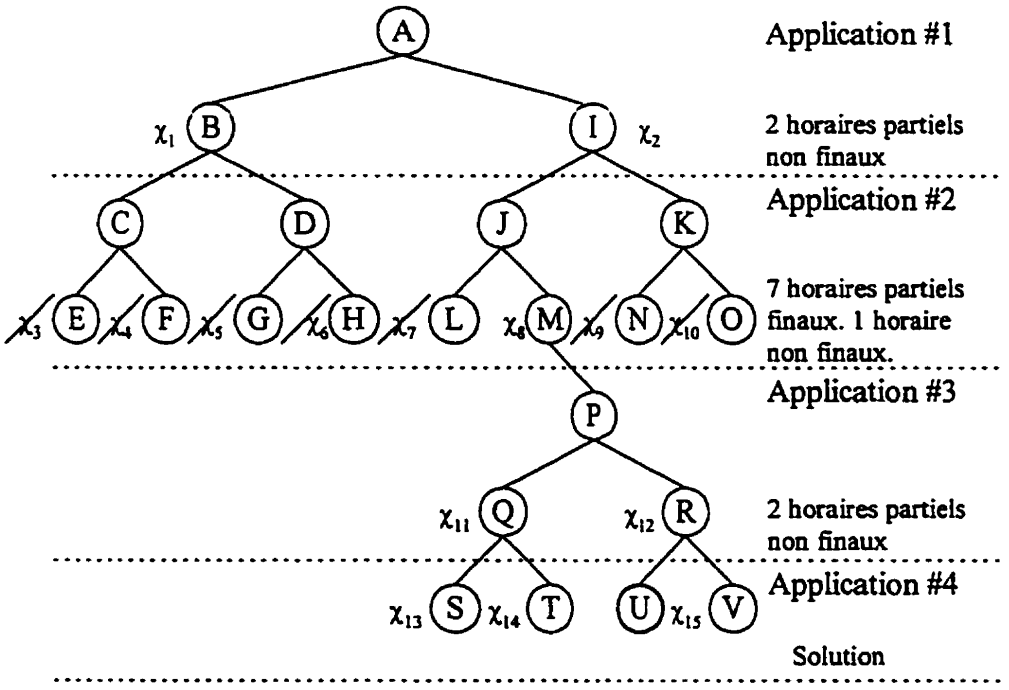


Figure 4.38 Illustration de la propriété 4.4.4.6 de la méthode DPSM.

Théorème 4.4.4.2 *La méthode DPSM obtiendra un horaire complet si le problème de répartition des tâches admet une solution.*

Démonstration Selon la propriété 4.4.4.5, la méthode DPSM développe toujours les états de même valeur heuristique f avant de développer les états de valeur f supérieure. Selon la propriété 4.4.4.6, l'enregistrement des solutions partielles sur disque ne change pas la nature des solutions obtenues par la méthode de fouille. Si le problème possède une solution, la méthode DPSM doit éventuellement développer une assignation finale. À moins qu'il existe une infinité d'assignations de valeur $f(n) < f^*(n)$. Cette condition existe si une assignation possède une infinité de successeurs ou s'il existe un horaire complet à coût fini mais avec un nombre infini d'assignations. Or, ce n'est pas possible

puisque le graphe d'états du problème est fini et borné sur L. Donc, la méthode DPSM doit produit un horaire complet à la fin de son exécution.

□

Chapitre 5

Heuristiques de répartition et performances de la méthode

5.1 Motivation

Ce chapitre présente les heuristiques utilisées dans la répartition des tâches par la méthode DPSM. Ces heuristiques sont nécessaires dans le prétraitement des groupes de tâches avec ordre de préséance, dans la génération des assignations candidates, dans le calcul de la fonction d'évaluation heuristique $f(n)$ et dans l'analyse postopératoire d'une répartition sans solution.

Le prétraitement est nécessaire lorsqu'un composant du réseau électrique est réalisé par un ensemble de tâches possédant un ordre de préséance. Les assignations candidates sont des couples comportant une tâche et un processeur. Elles sont susceptibles d'être choisies par le répartiteur lors de la production d'un horaire d'exécution. Les heuristiques de génération représentent donc les contraintes temporelles et spatiales du problème de la répartition des tâches. Quant à la fonction d'évaluation heuristique $f(n)$, elle est utilisée pour la sélection des assignations candidates. Elle doit représenter une certaine mesure de performance que l'on tentera de minimiser. Enfin, lorsque le répartiteur des tâches ne peut obtenir une solution satisfaisante, les heuristiques de l'analyse postopératoire sont utilisées pour permettre à

la méthode DPSM de proposer une alternative à la connexion du système des processeurs. Cette proposition doit pouvoir mener à terme la répartition des tâches.

C'est également dans ce chapitre que nous présentons les performances de la méthode proposée. Ces résultats sont obtenus en contrôlant différents paramètres du système des tâches et du système des processeurs qui sont soumis à la répartition. Ainsi, les caractéristiques des tâches et des processeurs sont générées d'une manière synthétique. La méthode de répartition des tâches peut alors être étudiée dans un environnement contrôlé.

5.2 Prétraitement des regroupements de tâches

Dans le modèle du système des tâches (section 3.3.3, chapitre 3), le réseau électrique est représenté par un ensemble de regroupements de tâches $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ et une relation $\mathcal{L}$ qui donne la connexité des regroupements de tâches. Ces regroupements de tâches peuvent contenir plus d'une tâche. Lorsque $|G_j| > 0$, alors G_i représente un composant dont les calculs sont décomposés explicitement en tâches parallèles. Dans ce cas, une relation d'ordre partiel $\prec$ doit exister. Pour tenir compte de ces regroupements de tâches, nous utilisons une heuristique de prétraitement qui consiste à ordonnancer ces tâches à l'aide d'une méthode d'ordonnancement par liste.

L'ordonnancement par liste est une heuristique simple et rapide capable ordonnancée des tâches avec un ordre de préséance. Chaque regroupement de tâches possède également une période d'exécution maximale D. Donc, l'objectif du prétraitement est d'ordonnancer les tâches $T_i \in G_j$ en tenant compte de l'ordre de préséance des tâches, du coût de communication entre les tâches et de leur échéancier. Ainsi, chaque regroupement de tâches avec $|G_j| > 1$ est prétraité par cette heuristique d'ordonnancement enfin de minimiser le temps de terminaison des tâches.

L'ordonnancement par liste est une méthode classique. Elle est surtout connue pour sa simplicité [HU61], [ADA74]. La formulation standard de cette heuristique est donnée dans la figure 5.1.

```
    Méthode OL(GROUPE_TÂCHE gt, GRAPHE_PROCESSEUR gp)
    {
        // Heuristique de l'ordonnancement par liste (Méthode classique)
        Liste q;    tâche T;    Processeur P;  Horaire h;
        Temps td;

        // Calculer le niveau de chaque tâche dans le graphe gt.
1       CALCULE-NIVEAU(gt);
        // Placer toutes les tâches sans prédécesseur dans la liste.
2       INIT-LISTE(q);
        // Prendre une tâches de la liste ayant la plus haute priorité
3       T = PRENDRE-TÂCHE(q);
        // Assigner T au premier processeur disponible P dans gp avec un
        // temps de départ 0. Mettre le résultat dans l'horaire h.
4       h = ASSIGNE-TÂCHE(h,T, 0, P, q);
        // Ordonnancer le reste des tâches
5       répéter tant que q != {∅} {
            // Mettre à jour la liste selon l'assignationcourante en
            // plaçant de nouvelles tâches prêtes de gt dans la liste q.
6           q = MISE-À-JOUR(q, T, gt);
            // Prendre une tâches de la liste ayant la plus haute priorité
7           T = SÉLECTIONNE-TÂCHE(q);
            // Choisir un processeur de gp qui peut terminer le plus
            // rapidement l'exécution de T en tenant compte du délai de
            // de communication entre les tâches. Obtenir également le
            // temps de départ de la tâche T sur le processeur P.
8           (P, td) = PRENDRE-PROCESSEUR(T, h, gp);
            // Assigner la tâche T au processeur P trouvé.
9           h = ASSIGNE-TÂCHE(h, T, td, P, q);
        }
    }
```

Figure 5.1 Pseudo-code de l'heuristique de l'ordonnancement des groupes de tâches.

Dans cette heuristique, les tâches à l'intérieur d'un regroupement sont représentées par un graphe de tâches. Chaque tâche est assignée une priorité qui est calculée en

utilisant le *niveau* de la tâche dans le graphe (ligne 1, figure 5.1). Le niveau d'une tâche T dans le graphe des tâches est le chemin le plus long partant de la tâche initiale jusqu'à la tâche considérée. La valeur d'un niveau est la somme des temps d'exécution des tâches situées le long de ce chemin [ADA74].

La fonction INIT-LISTE consiste à placer dans la liste q toutes les tâches qui sont *prêtes* à être ordonnancées. Si $\gamma^+(T)$ désigne l'ensemble des prédécesseurs immédiats de T, alors une tâche T est prête dès que $\gamma^+(T) = \{\varnothing\}$. Pour obtenir cette information, un compteur de prédécesseurs est entretenu dans la structure de données de chacune des tâches. Pour une tâche T donnée, la valeur initiale de ce compteur correspond aux nombres de tâches T_i qui satisfont la relation d'ordre partiel $T_i \prec T$. Cette valeur peut être obtenue facilement en parcourant le graphe des tâches. Les tâches sont considérées prêtes et sont placées dans la liste q lorsque la valeur de leur compteur est égale à zéro.

Les tâches sont placées dans la liste q en ordre décroissant de priorité (niveau). La tâche la plus prioritaire est sélectionnée par les applications de la fonction PRENDRE-TÂCHE (ligne 3, figure 5.1). Le processeur utilisé pour exécuter la première tâche sélectionnée est choisi d'une manière aléatoire. Par la suite, les processeurs seront choisis en fonction d'un critère plus serré. En effet, pour une tâche T donnée, le processeur choisi est celui qui peut terminer l'exécution de T le plus rapidement parmi l'ensemble des processeurs disponibles. Pour obtenir ce processeur, nous devons considérer la connexité du système des processeurs et du délai de communication. Pour la connexité des processeurs, nous avons besoin du graphe des processeurs (ou la matrice de connexions, définition 3.3.4.1 du chapitre 3). Une relation qui exprime le temps de communication des tâches assignées dans différents processeurs est nécessaire pour le calcul du délai de communication. Le délai de communication a été expliqué dans la section 3.3.4 du chapitre 3. La fonction SÉLECTIONNE-PROCESSEUR réalise justement ces calculs. Cette fonction retourne un processeur P qui est

$$P = \min_{\tau \in \gamma^-(T), P \in \mathcal{P}} \{C(\tau, T, P_\tau, P) + t_P + E_{TP}\}. \qquad (5.2.1)$$

Dans l'équation (5.2.1), T est la tâche à ordonnancer, $\gamma^+(T)$ est l'ensemble des prédécesseurs immédiats de T, $\mathcal{P}$ est l'ensemble des processeurs disponibles. Le terme t_P est le temps où le processeur P est libre et E_{TP} est le temps d'exécution de la tâche T dans le processeur P. Enfin, $C(\cdot)$ est le délai de communication entre les tâches $\gamma^+(T)$ et la tâche T (équations 3.3.4.5 et 3.3.4.6 de la section 3.3.4 du chapitre 3) Pour un système à N_T tâches et N_P processeurs, la fonction doit examiner tous les processeurs avant d'effectuer un choix. Pour chaque processeur, la fonction SÉLECTIONNE-PROCESSEUR doit également calculée le délai de communication $C(\cdot)$ pour tous les prédécesseurs immédiats $\gamma^+(T)$ de la tâche T à ordonnancer. Dans le pire des cas, $|\gamma^+(T)| = N_T - 1$. Dans ce cas, la complexité asymptotique de la fonction SÉLECTIONNE-PROCESSEUR est $O(N_P\ N_T\text{-}1)$. Enfin, puisqu'il y a N_T tâches à ordonnancer dans le regroupement, la complexité temporelle de l'heuristique OL est $O(N_P\ N_T^2)$.

En général le nombre de tâches d'un regroupement est petit par rapport au nombre de processeurs de l'ordinateur parallèle. Après tout, un regroupement de tâches représente un seul composant d'un poste du réseau électrique à simuler. Donc, la cardinalité d'un regroupement de tâches G est normalement inférieure à la cardinalité de l'ensemble des processeurs, $|G| <<< |\mathcal{P}|$. Ainsi, il nous est permis d'utiliser un sous-ensemble $\mathcal{P}_T$ de processeurs de $\mathcal{P}$ pour effectuer l'ordonnancement de G. $\mathcal{P}_T \subseteq \mathcal{P}$ et la cardinalité de $|\mathcal{P}_T| \leq |G|$. Dans le cas où $|\mathcal{P}_T| = |G|$, la complexité temporelle de OL devient $O(N_T^3)$. L'heuristique OL utilise une liste de priorité pour entreposer les tâches qui sont prêtes pour être ordonnancées. Donc la complexité spatiale de cette heuristique est $O(N_T)$.

Un exemple de l'application de l'heuristique OL est donné dans la figure 5.2. Cet exemple comprend 9 tâches. Le graphe des tâches de cette figure est un graphe orienté. La valeur inscrite à l'intérieur des sommets représente le temps d'exécution des tâches.

La valeur des arcs donne le nombre de données à transférer entre les tâches. Les arcs servent aussi à indiquer la relation de préséance qui existe entre les tâches. Ainsi, la tâche C doit terminer son travail avant le démarrage de la tâche I (c'est-à-dire, C $\prec$ I). Enfin, l'annotation numérique à côté de chaque sommet donne la valeur du niveau de chacune des tâches. Dans l'exemple de la figure 5.2, la tâche C possède un niveau de 2 et la tâche I un niveau de 1. Le résultat de l'ordonnancement par l'heuristique OL utilisant 3 processeurs est donné dans la figure 5.3. Pour simplifier la discussion, le réseau des processeurs est considéré comme pleinement connecté. Le taux de transfert des données des processeurs est de 1 donnée par unité de temps.

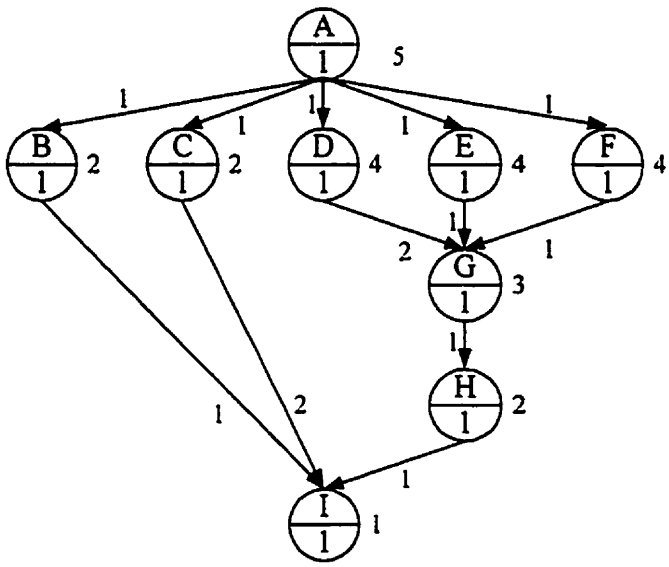


Figure 5.2 Graphe de tâches avec ordre de préséance.

D'après le résultat de la figure 5.3, la durée totale de l'exécution des tâches est de 8 unités de temps. À cause du temps de démarrage de la tâche C dans le processeur P1, la tâche I ne peut terminer son travail qu'au temps $t = 8$. L'horaire d'exécution de la figure 5.3 montre la présence de temps mort au temps $3 \leq t \leq 4$ et au temps $6 \leq t \leq 7$ pour les processeurs P0 et P1. Ce temps mort est créé par le délai de communication. L'inactivité des processeurs dans ce temps mort diminue le taux d'utilisation des processeurs et peut augmenter la durée de l'horaire d'exécution généré.

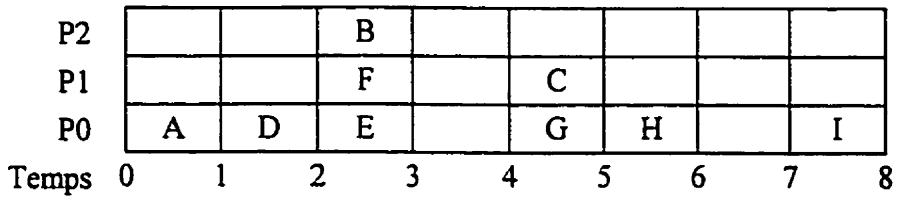


Figure 5.3 Horaire d'exécution obtenu par l'heuristique OL.

Une façon simple pour palier à ce problème consiste à assigner des tâches prêtes dans ce temps mort des processeurs. Pour ce faire, nous modifions la fonction ASSIGNE-TÂCHE. La sélection d'une tâche pour ordonnancer s'effectue toujours par la valeur de leur niveau mais une fouille secondaire est également réalisée. Cette fouille secondaire consiste à trouver dans la liste de priorité, une tâche qui peut être placée dans le temps mort des processeurs. Le pseudo-code de la fonction ASSIGNE-TÂCHE modifiée est donné dans la figure 5.4.

Pour considérer l'effet de cette modification, nous reprenons l'exemple de la figure 5.2 en appliquant l'heuristique OL avec la fonction ASSIGNE-TÂCHE modifiée. Le résultat de l'ordonnancement est expliqué dans la figure 5.5. Lors de l'ordonnancement de la tâche G, la fonction PRENDRE-PROCESSEUR retourne le processeur P0 avec un temps de démarrage $td = 4$. Un temps mort est créé dans l'intervalle $3 \leq t \leq 4$ dans le processeur P0. Alors la fonction ASSIGNE-TÂCHE remplira, dans la mesure du possible, ce temps mort par une tâche de la liste de priorité. La première tâche prête et qui peut être placée dans le temps mort est la tâche B. Par contre, la tâche B peut aussi être placée dans le processeur P2 avec un temps de démarrage *td* plus petit que celui du processeur P0. Donc, la fonction ASSIGNE-TÂCHE doit chercher une autre tâche pour remplir le temps mort de P0. Dans cet exemple, la tâche C de la liste de priorité peut être utilisée pour remplir le temps mort de P0. En résumé, la fonction ASSIGNE-TÂCHE modifiée tentera toujours de placer une tâche prête dans un intervalle de temps mort. Si cette tâche

peut achever son travail à l'intérieur de cet intervalle de temps et si elle ne peut démarrer son exécution à un temps *td* plus petit dans un autre processeur.

```
    Fonction ASSIGNE-TÂCHE(Horaire h, Tâche T, Temps td, Processeur P,
                           Liste Tâche q, GROUPE_TÂCHE gt)
    {
        // Assigne une tâche T à un processeur P et remplir les temps morts
        // du processeur P.
        Temps tm, t1, t2;   Tâche Ta = PRENDRE-TÂCHE(q);
        Booléan trouvé = FAUX;

        // Obtenir le temps mort du processeur P.
1       tm = td - TEMPS-PRÊT(P);
2       si (tm > 0) alors {
3           répéter tant que ((tm != 0) && (Ta != {∅})) {
4               répéter tant que (( trouvé) && (Ta != {∅})) {
                    // Calculer le temps départ de Ta dans P. t1 et t2
                    // sont le temps de démarrage et de fin d'exécution
                    // de Ta dans P.
5                   (t1, t2) = TEMPS-DÉMARRAGE(Ta, P, h);
6                   si (t2 - t1 ≤ tm) alors  trouvé = VRAI; autrement {
7                       trouvé = FAUX; Ta = PRENDRE-TÂCHE(q);
                    }
                }
8               si (trouvé) alors {
                    // Insérer la tâche trouvé dans le temps mort de P.
9                   h = INSÉRER-TÂCHE(h, Ta, t1, P);
                    // Ta est ordonnancée alors mettre à jour q.
10                  q = MISE-À-JOUR(q, Ta, gt);
                    // Voir s'il existe un temps mort créé par l'assignation
                    // de Ta dans P.
11                  tm = tm - (t2 - t1);
                }
            }
        }
        // Ajouter T dans P avec un temps de départ de td.
        h = INSÉRER-TÂCHE(h, T, td, P);
    }
```

Figure 5.4 Pseudo-code de la fonction ASSIGNE-TÂCHE Modifiée.

Avec l'application de la fonction ASSIGNE-TÂCHE modifiée, la durée totale de l'horaire d'exécution est de 7 unités de temps. Le temps mort du processeur P0 est éliminé.

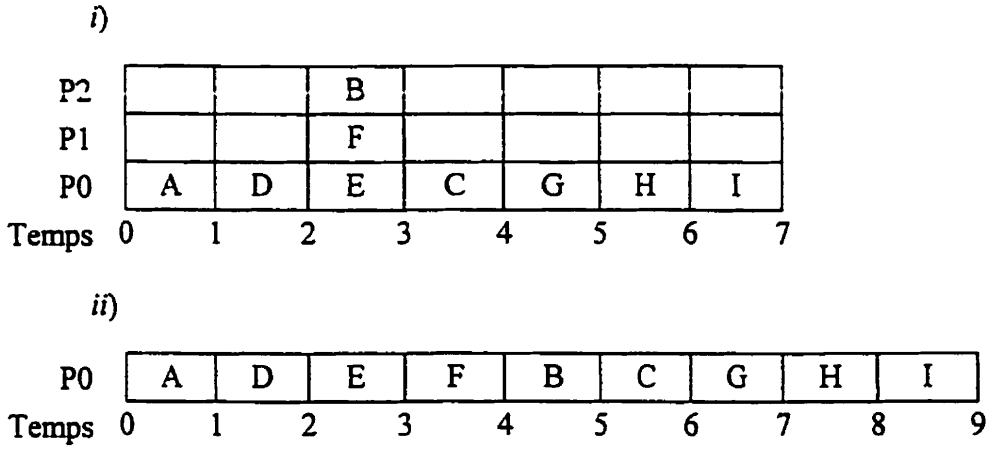


Figure 5.5 Horaire d'exécution obtenu avec l'heuristique OL modifiée: i) $|\mathcal{P}_T| = 3$; ii) $|\mathcal{P}_T| = 1$.

Pour vérifier l'efficacité de cette heuristique, nous avons généré 500 regroupements de tâches. Ces regroupements ont des cardinalités de 20, 60, 100, 160 et 200 tâches. L'ordre de préséance de ces regroupements de tâches est créé d'une manière aléatoire en utilisant une variation de la méthode proposée dans [XU94], [WON95]. Nous pouvons envisager cette méthode de génération des tâches comme le remplissage d'une matrice de $q \times p$ où p est le nombre de processeurs et q est la durée totale de l'horaire optimal. La procédure de génération comporte six étapes: *i*) donner à chacune des tâches du regroupement un temps d'exécution unitaire; *ii*) placer une tâche dans la première ligne et dans la dernière ligne de la matrice; *iii*) placer $q - 3$ tâches dans les $q - 2$ lignes restantes de la matrice; *iv*) relier les tâches de l'étape *iii* par des arcs de valeurs unitaires; *v*) compléter la matrice jusqu'à la cardinalité du regroupement en ajoutant des tâches dans les lignes 2 à $q - 1$ inclusivement de la matrice; vi) relier les tâches de l'étape *iv* par des arcs de valeurs comprises entre 1 et C_C. La constante C_C est la valeur maximale du coût de communication dans le regroupement.

À l'étape *iii* et à l'étape *v* les tâches sont assignées d'une manière aléatoire dans une des p colonnes de la matrice. À l'étape *v*, les tâches d'une ligne l sont reliés à leur prédécesseurs immédiats (les tâches de la ligne $l-1$) par au moins un arc. Pour un regroupement de tâches dont le temps d'exécution est unitaire et dont le coût de communication est borné pour tout le groupe, le chemin minimal entre la tâche de la ligne 1 et la tâche de la ligne q est le chemin réalisé dans les étapes *ii*, *iii* et *iv*. En effet, l'horaire d'exécution réalisé dans les étapes *ii*, *iii* et *iv* possède $q-1$ tâches et $q-2$ arcs. Donc, la durée totale d'exécution des tâches est $(q-1)+(q-2)=2q-3$. Tous les autres chemins contiennent au moins q tâches et ont une durée d'exécution supérieure à $2q-3$. Les paramètres des essais réalisés et la configuration d'interconnexions des processeurs sont donnés dans le tableau 5.1 et la figure 5.6.

Tableau 5.1 Conditions des essais pour la validation de l'heuristique OL^{+}.

Paramètre	**Caractéristique**
Nb. de regroupements	500
Taille des regroupements	$N_T = \{20, 60, 100, 160, 200\}$
Nb. de regroupements par taille	100
Temps d'exécution des tâches	1 unité de temps
Temps de communication entre les tâches	1 à 30 unités de temps
Nb. de processeurs utilisés	Même nombre que N_T

La configuration d'interconnexions des processeurs de la figure 5.6 est tirée du simulateur numérique Hypersim d'IREQ (Institut de recherche d'Hydro-Québec) [IRE95]. Elle utilise 5 ports de communication bidirectionnelle par processeur. Les étiquettes à l'intérieur de la figure indiquent l'état initial des ports de communication. L'étiquette « i » signifie que le port est initialement en mode de réception. Tandis que l'étiquette « o » signifie que le port est initialement en mode de transmission. On peut

augmenter le nombre de processeurs de l'ordinateur en ajoutant des cartes supplémentaires.

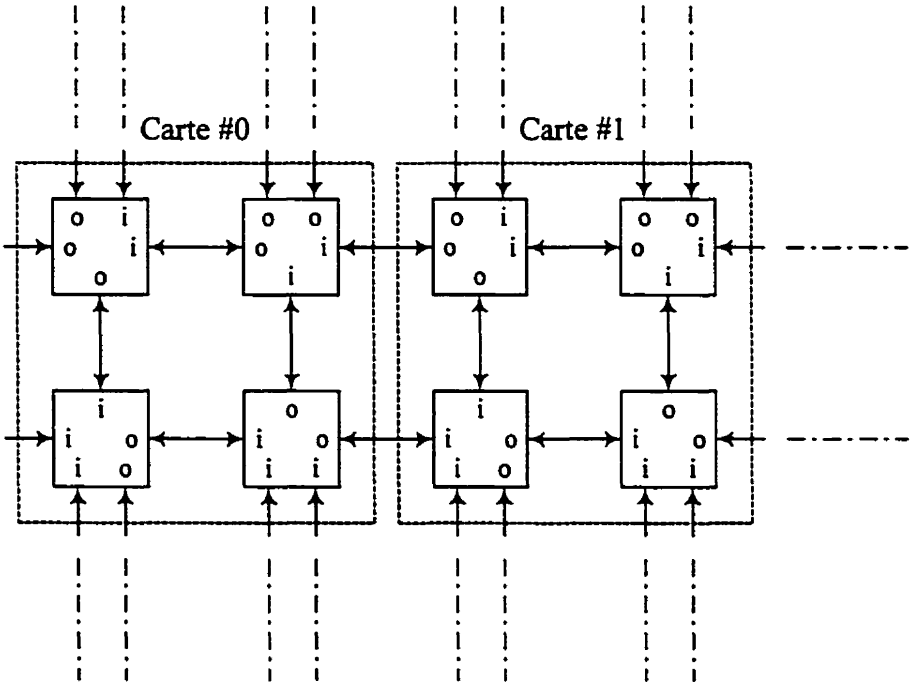


Figure 5.6 Interconnexions des processeurs utilisées pour les essais.

Les tableaux 5.2 à 5.4 présentent les résultats obtenus par simulation. Dans cette présentation, l'heuristique OL est l'heuristique de la figure 5.1 tandis que l'heuristique OL^+ est celle comportant la modification de la figure 5.2. Dans le tableau 5.2, le pourcentage d'amélioration (speedup) relative moyenne $\%A$ de l'heuristique OL^+ est calcué par rapport à l'heuristique OL en fonction du coût de communication. L'amélioration relative moyenne de OL^+ est donnée par [HEN97, Chapitre 1]

$$\%A = \left(\frac{S(OL^+) - S(OL)}{S(OL)} \right) 100\%, \tag{5.2.1}$$

où $S(x)$ est l'amélioration (speedup) donnée par l'ordonnancement monoprocesseur par rapport à un ordonnancement à l'aide de l'heuristique x sur p processeurs. $S(x)$ mesure donc la diminution de la durée totale d'exécution du regroupement des tâches produit

par l'heuristique x. Par exemple, $S(OL^+) = 1.29$ pour le résultat de la figure 5.5. La valeur de $\%A$ est une indication quantitative sur la contribution apportée par l'heuristique OL^+ à la méthode classique de l'ordonnancement par liste.

Tableau 5.2 Amélioration apportée par OL^+ par rapport à l'heuristique OL.

	Nombre de tâches dans les regroupements				
	20	**60**	**100**	**160**	**200**
Coût de communication	$\%A$	$\%A$	$\%A$	$\%A$	$\%A$
$C_C = 1$	7.1	15.18	34.81	44.43	58.79
$C_C = 5$	44.02	38.56	42.68	56.08	62.86
$C_C = 10$	83.63	88.76	73.29	89.87	86.45
$C_C = 15$	93.0	89.54	94.31.	95.54	94.59
$C_C = 20$	77.7	80.37	83.86	102.95	129.69
$C_C = 30$	49.81	57.63	64.91	150.43	163.22

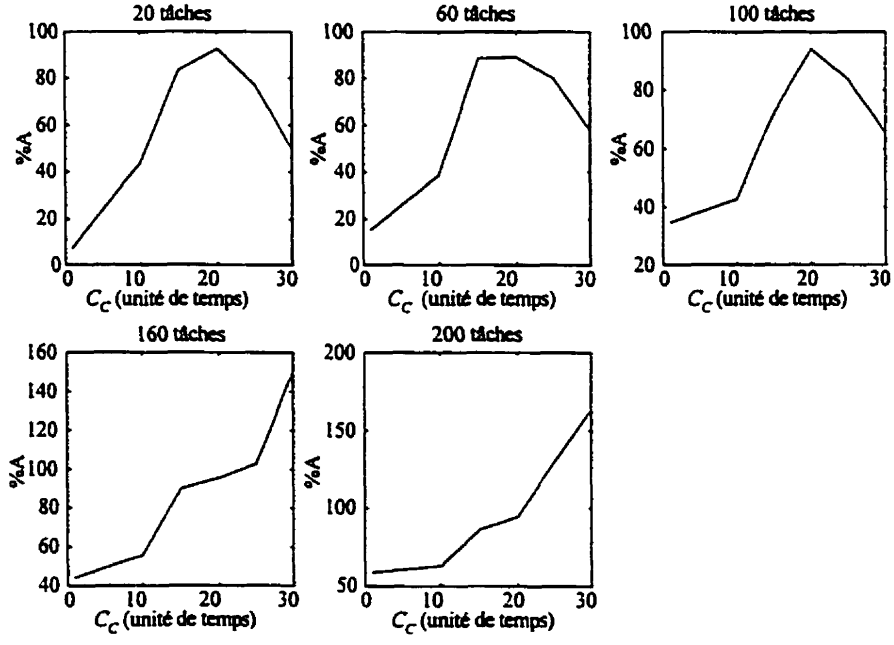


Figure 5.7 Représentation graphique des résultats du tableau 5.2.

D'une manière générale, la modification à la fonction ASSIGNE-TÂCHE permet une diminution appréciable de la longueur de l'horaire d'exécution des tâches. D'après les résultats du tableau 5.2, l'apport du remplissage du temps mort des processeurs est le plus important pour des regroupements de grande cardinalité et un temps de communication élevé. Lorsque le délai de communication est supérieur à 20 fois le temps d'exécution des tâches, l'amélioration apportée par l'heuristique OL^+ dépasse les 100% pour des regroupements de 160 tâches et plus. En effet, plus le délai de communication est grand, plus le temps mort des processeurs est important. Plus le nombre de tâches est grand, plus il y aura de tâches prêtes pour remplir ces temps morts.

Il existe un seuil de maximal dans la contribution de l'heuristique OL^+ pour les regroupements de 100 tâches et moins (figure 5.7, page 169). Ce seuil maximal est atteint lorsque le délai de communication est environ 20 fois le temps d'exécution des tâches. Au delà de $C_c = 20$, l'heuristique OL^+ n'apporte plus d'amélioration croissante. On peut considérer la plage de valeurs $C_C = 1$ à $C_C = 20$ comme une région dans laquelle il existe suffisamment de tâches prêtes pour remplir le temps mort des processeurs. Lorsque $C_C > 20$ fois le temps d'exécution des tâches, les tâches prêtes que l'on peut assigner dans les intervalles de temps morts devient de moins en moins nombreuses. Pour ce qui a trait aux regroupements de plus de 100 tâches, le seuil maximal est situé à des valeurs de C_C qui sont plus grandes. Enfin, peu importe le nombre de tâches dans les regroupements, l'apport de l'heuristique OL^+ doit éventuellement plafonner pour une certaine valeur du coût de communication.

Nous vérifions maintenant le comportement de l'heuristique OL^+ face à des variations du coût de communication. Pour ce faire nous définissons une quantité $\%S$ qui représente l'amélioration (speedup) donnée par l'heuristique OL+ à un certain coût de communication C_C par rapport à l'amélioration donnée par cette même heuristique à un coût de communication $C_C = 1$. La quantité $\%S$ est donc une mesure qui indique la

capacité intrinsèque de l'heuristique OL^+ à traiter des systèmes de tâches à différents coût de communication. Les valeurs de %S sont calculées de manière suivante :

$$\%S = \left(\frac{S_C(OL^+, C_c)}{S_C(OL^+, C_C)\big|_{C_C=1}} \right) 100\%. \qquad (5.2.2)$$

Le terme $S_C(OL^+, C_C)$ de l'équation (5.2.2) est l'amélioration réalisée par l'heuristique OL^+ pour des systèmes de tâches dont le coût de communication est borné par C_C. Les valeurs de $S_C(OL^+, C_C)$ sont obtenues en comparant la durée totale des horaires d'exécution générés pour le cas monoprocesseur et multiprocesseur.

Tableau 5.3 Amélioration moyenne de OL^+ en fonction du coût de communication.

	Nombre de tâches dans les regroupements				
	20	**60**	**100**	**160**	**200**
Coût de communication	**%S**	**%S**	**%S**	**%S**	**%S**
$C_C = 1$	100.0	100.0	100.0	100.0	100.0
$C_C = 5$	50.56	51.64	49.34	51.27	51.47
$C_C = 10$	34.74	36.71	35.87	36.55	34.36
$C_C = 15$	20.5	21.11	21.01	20.89	21.93
$C_C = 20$	11.59	12.03	12.43	12.36	12.38
$C_C = 30$	6.42	5.61	5.77	5.96	5.81

Selon les résultats obtenus, plus le coût de communication des tâches est grand plus la performance de l'heuristique OL^+ est faible. Lorsque le coût de communication est 5 fois plus grand que le temps d'exécution des tâches (C_C = 5), l'amélioration apportée par OL^+ n'est qu'à 50% de celle obtenue pour C_C = 1. Lorsque le coût de communication des tâches est 30 fois plus grand que le temps d'exécution (C_C = 30), l'amélioration de OL^+ dégringole sous les 7%. Ce comportement de l'heuristique OL^+ est similaire pour différentes grandeurs de regroupement. Il existe donc une lacune qui

est propre à ce genre d'heuristique pour traiter des systèmes dont le coût de communication est supérieur au coût d'exécution des tâches.

Enfin nous avons mesuré le temps d'exécution de l'heuristique pour différents regroupements de tâches. Le temps d'exécution est une quantité qui dépend fortement sur les caractéristiques matérielles de l'ordinateur utilisé. C'est pour cette raison que nous présentons le temps d'exécution de l'heuristique comme un rapport plutôt comme une quantité absolue. Nous définissons ce rapport comme suit

$$V_{OL^+} = \left(\frac{t(N_T)}{t(N_T)\big|_{N_T=20}} \right), \tag{5.2.3}$$

où $t(N_T)$ est le temps d'exécution de l'heuristique OL^+ pour un regroupement de N_T tâches. Dans l'équation (5.2.3), la quantité $V_{OL}+$ est une mesure relative calculée en fonction des regroupements de 20 tâches.

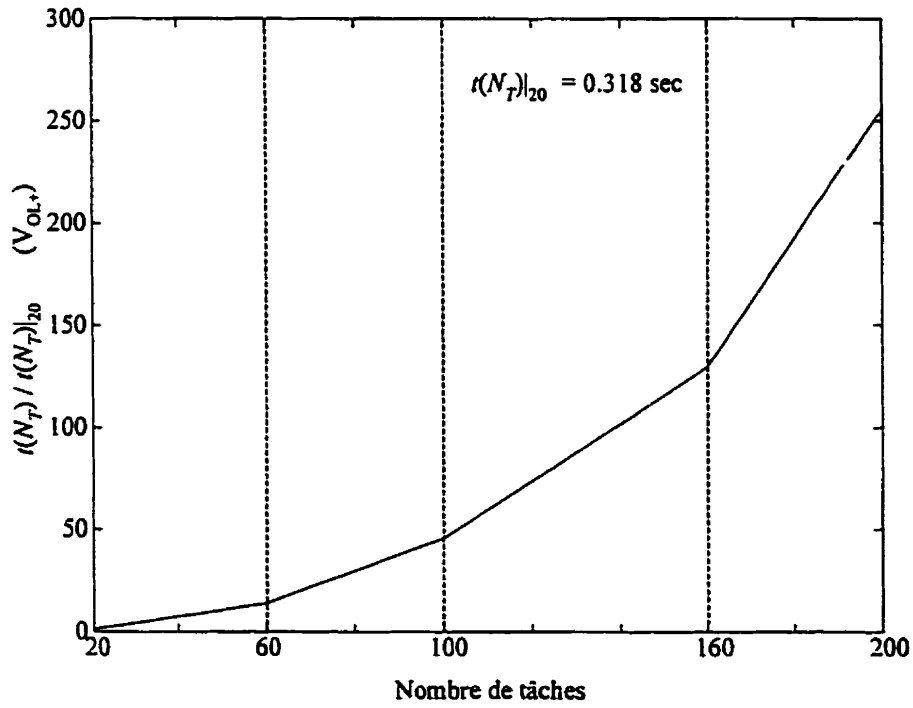


Figure 5.8 Rapport des temps d'exécution de l'heuristique OL^+.

Le temps d'exécution de l'heuristique OL^+ augmente d'une manière exponentielle en fonction du nombre de tâches dans les regroupements. La durée d'exécution est près de 50 fois plus longue pour des regroupements de 100 tâches que celle des regroupements de 20 tâches. Pour des regroupements de 200 tâches, la durée d'exécution de OL^+ est 250 fois plus longue. Décidément, l'heuristique OL^+ n'est pas appropriée pour traiter des regroupements de grande taille.

Dans le but de vérifier la capacité d'ordonnancement de cette heuristique, nous avons relevé la longueur des horaires d'exécution obtenus. Ces résultats sont présentés dans le tableau 5.4. La quantité H est obtenue en prenant la moyenne des longueurs d'horaire de chaque ensemble de regroupements. C'est-à-dire,

$$H = \left(\sum_{i=1}^{N} h_i(\mathrm{OL}^+) / \hat{h}_i \right) \Big/ N . \qquad (5.2.4)$$

Dans le calcul de l'équation (5.2.4), N est le nombre d'essais utilisés pour une taille de regroupement. La valeur de $h_i(\mathrm{OL}^+)$ est la longueur de l'horaire obtenue par OL^+ pour le i^{e} essai. La valeur de $\hat{h}_i$ est la longueur minimale de l'horaire pour le regroupement des tâches du i^{e} essai. Cette dernière est obtenue directement par le générateur des tâches. Pour les résultats du tableau 5.4, le nombre d'essais N est égal à 100 puisqu'il y a 100 regroupements pour les systèmes de tâches de mêmes tailles. Il est à noter que ce type de moyenne arithmétique est aussi utilisé pour les résultats montrés précédemment.

L'heuristique OL^+ est très performant pour des regroupements de petite taille. Pour des regroupements où $N_T = 20$, les horaires obtenus ont une longueur qui est, au plus, 8% supérieurs à celle de l'horaire optimal. De plus, lorsque le temps de communication est égal ou inférieure au temps d'exécution des tâches ($0 \leq C_C \leq 1$), la majorité des horaires obtenus sont des horaires de longueur minimale. Par contre, lorsque la

cardinalité des regroupements est grande, l'heuristique montre des signes de faiblesse. Dans ce cas, l'heuristique génère des horaires qui sont deux fois plus longs que les horaires optimaux (dernière colonne du tableau 5.4 avec $20 \leq C_C \leq 30$).

Tableau 5.4 Longueur moyenne des horaires obtenus par OL^+.

	Nombre de tâches dans les regroupements				
	20	**60**	**100**	**160**	**200**
Coût de communication	*H*	*H*	*H*	*H*	*H*
$C_C = 1$	1.002	1.05	1.104	1.03	1.09
$C_C = 5$	1.051	1.06	1.12	1.18	1.39
$C_C = 10$	1.057	1.07	1.13	1.17	1.55
$C_C = 15$	1.073	1.08	1.15	1.25	1.98
$C_C = 20$	1.076	1.10	1.36	1.47	2.11
$C_C = 30$	1.08	1.24	1.43	1.478	2.23

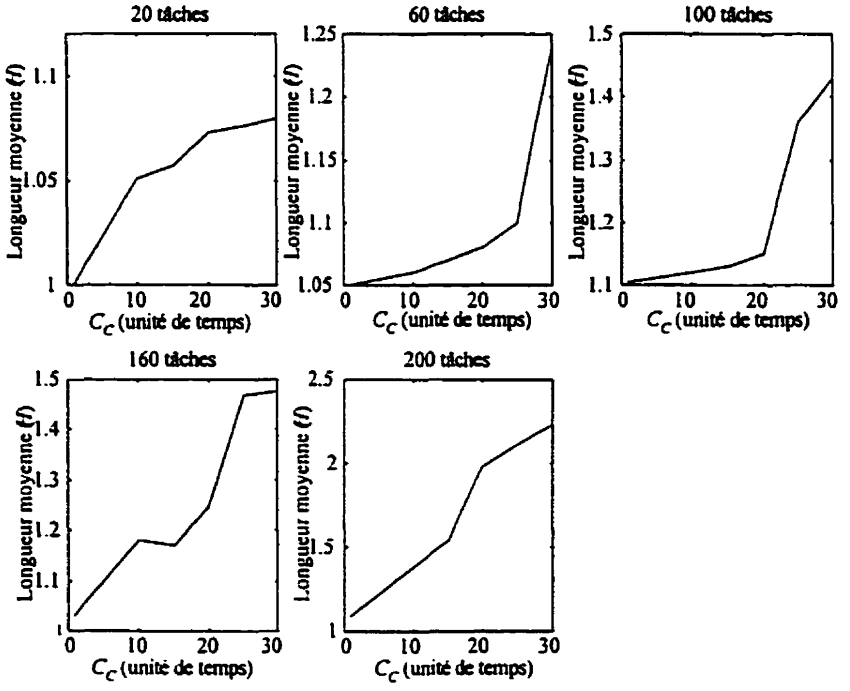


Figure 5.9 Représentation graphique des résultats du tableau 5.4.

5.2.1 Discussion sur les résultats de l'heuristique OL^+

Le prétraitement par l'heuristique OL^+ convient à des regroupements de petite taille. Dans les simulations réalisées, la performance de OL^+ est supérieure à celle de l'heuristique classique OL. D'après les résultats du tableau 5.2 l'heuristique OL^+ apporte une amélioration indéniable à l'ordonnancement par liste. Quant à la capacité intrinsèque, les résultats de OL^+ sont meilleures lorsque la taille des regroupements est inférieure ou égale à $N_T = 20$. À cette taille, l'heuristique est en mesure d'obtenir des horaires d'exécution qui sont, au plus, à 8% des horaires optimaux.

La performance de l'heuristique OL^+ se dégrade rapidement pour des regroupements de grandes tailles. Il en est de même pour des systèmes avec un coût de communication supérieur au coût d'exécution des tâches. En effet, l'amélioration de OL^+ est réduite de moitié dès que le coût de communication est égal à 5 fois le coût d'exécution des tâches.

Ainsi, l'heuristique OL^+ s'applique lorsque la cardinalité des tâches et le délai de communication sont petits. Or, ces caractéristiques sont justement celles des regroupements de tâches des réseaux électriques. Les regroupements de tâches sont conçus pour des composants électriques trop complexes pour être simulés en un pas de calcul. Le comportement de ces composants est alors simulé par un ensemble de calculs parallèles. La quantité de données transférées entre ces calculs est normalement très réduite afin de satisfaire la contrainte temporelle du système.

L'utilisation de l'heuristique OL^+ impose une limite sur le nombre de tâches dans les regroupements et sur la quantité d'informations à transférer entre les tâches. Les algorithmes parallèles utilisés doivent adhérer à ces restrictions pour obtenir la meilleure

performance possible du répartiteur des tâches. Le tableau 5.5 donne les restrictions nécessaires pour les regroupements de tâches.

Tableau 5.5 Limites des paramètres pour les regroupements de tâches.

Paramètre	
Nb. de tâches dans les regroupements	≤ 20
Temps d'exécution / Temps de la communication	≥ 1

5.3 Génération des assignations candidates

Les assignations candidates sont utilisées lors de la fouille heuristique. Les assignations candidates sont générées par des opérateurs de génération. Certaines de ces assignations candidates seront retenues par la méthode de fouille pour la construction d'un horaire d'exécution satisfaisant les contraintes temporelles imposées. Donc, chaque assignation candidate générée doit être une assignation qui remplie correctement les exigences dictées par les contraintes du système. Ainsi, les opérateurs de génération sont des heuristiques qui représentent les différentes contraintes du système des tâches et du systèmes des processeurs.

Les heuristiques utilisées doivent pouvoir générer uniquement les assignations qui satisfont toutes les contraintes temporelles et spatiales. Nous établissons ces heuristiques en utilisant le modèle du système des tâches et le modèle du système des processeurs élaborés dans le chapitre 3 de cette thèse. Dans le modèle de la section 3.3.3 du chapitre 3, le système des tâches est représenté par un couple $\mathcal{T} = (\mathcal{G}, \mathcal{L})$ où $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ est un ensemble de N regroupements de tâches et $\mathcal{L}$ est une relation exprimant la connexité des éléments de $\mathcal{G}$. Nous utilisons une application $\lambda : \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{N}$ pour indiquer le nombre de données à transférer entre deux regroupements de tâches. Chaque regroupement G_i est composé d'un ensemble de m_k tâches $\mathfrak{I} = \{T_1, T_2, \ldots, T_{m_k}\}$. Les attributs de ces tâches sont donnés dans la définition 3.3.3.3 du chapitre 3. Dans cette

section nous traitons les regroupements de tâches dont la cardinalité est l'unitaire, c'est-à-dire, $|G_i| = 1$ ou encore $m_k = 1$ pour l'ensemble des tâches $\Im$ à l'intérieur de G_i. Les regroupements de quantités supérieures étant déjà considérés par l'heuristique d'ordonnancement OL^+ de la section 5.2. Le système des processeurs est modélisé par un 6-uplet ($\mathcal{P}$, $[B_{ij}]$, $[S_i]$, $[I_i]$, $[L_{ij}]$, $[r_{ik}]$) tel que défini dans la section 3.3.4 du chapitre 3. La matrice $[B_{ij}]$ donne la connexité des processeurs de l'ensemble $\mathcal{P}$.

Une tâche $T \in G_i$ et un processeur $P \in \mathcal{P}$ sont considérés comme une assignation candidate si : *i*) la tâche T peut être placée dans P sans excéder l'échéancier de T; *ii*) la tâche T utilise b ressources de type k alors le processeur P doit posséder un nombre de ressources de type k supérieur ou égal à b; *iii*) le processeur P est un voisin des processeurs utilisés pour les tâches voisines de T. Les deux premières conditions sont des conditions absolues. Autrement dit, une assignation candidate (T, P) doit nécessairement satisfaire les conditions *i* et *ii*. La dernière condition est souhaitable pour réduire le temps de communication entre les tâches et permettre une meilleure utilisation du réseau des processeurs.

Pour vérifier la première condition, nous devons connaître les tâches voisines de T. Une tâche T' est voisine de T si elle communique avec T. Un processeur P est choisi s'il peut achever le travail de T à l'intérieur de sa période d'exécution en tenant compte du délai de communication entre la tâche T et ses voisines. Enfin, si la tâche T exige des ressources spécifiques alors le processeur doit pouvoir satisfaire ces demandes de la part de T. La figure 5.10 est le pseudo-code qui effectue la sélection d'un processeur pour une tâche quelconque. Dans cette fonction de sélection, le paramètre $t_d(P)$ est le temps où le processeur P est prêt à recevoir un tâche. Les paramètres $\lambda(T', T)$ et $L(P', P)$ sont respectivement le nombre de données à transférer entre T' et T et le taux de transmission du lien reliant les processeurs P' et P. Les tâches T' sont les tâches voisines de T et sont représentées par l'ensemble $\Gamma(T)$. Les processeurs P' sont les processeurs assignés aux tâches T'. Enfin, les paramètres i_T et S_p sont respectivement le nombre d'instructions de

la tâche T et la vitesse d'exécution du processeur P. En pratique, ces paramètres sont remplacés par le temps d'exécution mesuré de la tâche T dans les processeurs de l'ordinateur parallèle.

```
    Fonction GÉNÉRATION(Assignation n)
    {
        // Sélectionner un processeur pour la tâche T
        Liste Processeur PL, PL'; Tâche T = n.T

        // Obtenir l'ensemble des processeurs libres qui peut achever
        // le travail de T à l'intérieur de son échéancier
1       t_f(P) ← (t_d(P) + Σ_{T',P'∈Γ(T)} λ(T',T)/L_{P',P} + Σ_{T',P'∈Γ(T)} λ(T,T')/L_{P',P} + i_T/S_P), ∀P ∈ 𝒫
2       PL ← {P ∈ 𝒫 | t_d(P) − t_f(P) ≤ D_T};
        // Si la tâche T utilise des ressources, ne conserver que les
        // processeurs disposant de ces mêmes ressources
3       si (u_Tk > 0, ∀ k) alors
4           PL ← {P ∈ PL | u_Tk ≤ r_pk, ∀k};
        // Tenter de conserver les processeurs qui sont voisins aux
        // processeurs des tâches voisines de T.
5       PL' ← {P ∈ PL | B_{P,P'} > 0, ∀P' ∈ Γ(T)};
6       si (|PL'| > 0) alors
7           PL ← PL';
        // Retourner le processeur de la liste PL avec la plus petite
        // valeur de t_f
8       si (|PL| > 0) alors
9           retourne P = min_{P∈PL} t_f(P);
10      autrement
11          retourne bidon;
    }
```

Figure 5.10 Heuristique pour la sélection d'un processeur.

La ligne 2 de la figure 5.10 sert à retenir les processeurs qui peuvent terminer l'exécution de T à l'intérieur de l'échéancier D_T de la tâche. Pour traiter une tâche utilisant des ressources, une seconde sélection est effectuée en parcourant la liste des processeurs retenus. Dans cette seconde sélection (lignes 3 et 4 de la figure 5.10) les

ressources utilisées par la tâche T (u_{Tk}) sont associées à des processeurs disposant les mêmes ressources en nombre et en type (r_{Pk}). Enfin, la fonction GÉNÉRATION tentera de trouver un ensemble de processeurs qui sont voisins des processeurs assignés aux tâches voisines de T. Le processeur choisi par la fonction est celui qui peut achever l'exécution de T le plus vite (ligne 9 de la figure 5.10).

En résumé, les heuristiques de génération des assignations candidates représentent les contraintes imposées au répartiteur des tâches. Elles sont découplées de la méthode de fouille. Par conséquent, elles peuvent être modifiées sans aucun remaniement majeur dans le code source du répartiteur.

5.4 Fonction d'évaluation heuristique $f(n)$

Nous avons vu dans la section 4.4.1 du chapitre 4 que la fonction d'évaluation heuristique de la méthode DPSM est donnée par $f(n) = r(g(n), h(n))$ où $r(\cdot,\cdot)$ est une relation définie sur l'ensemble des valeurs de f. La relation r est associative et admet un élément neutre ε. Pour la méthode DPSM, le monoïde de valeurs $(\mathcal{R}^+, +)$ est utilisé. Dans ce monoïde, la relation r est l'opérateur addition et l'ensemble des valeurs est l'ensemble des nombres réels. L'élément neutre de la relation est le nombre $\varepsilon = 0$.

La fonction d'évaluation heuristique $f(n)$ est composée de deux parties. Le terme $g(n)$ est le coût encouru depuis l'assignation initiale jusqu'à l'assignation n. Le terme $h(n)$ est une sous-estimation du coût encore à débourser de l'assignation n jusqu'à une assignation finale. La méthode de répartition sélectionne les assignations candidates, produites par le générateur d'assignations, qui minimisent la valeur de f.

Dans notre contexte, les coûts impliqués dans le calcul de $g(n)$ et $h(n)$ sont modélisés par le délai communication entre les différents regroupements de tâches. Puisque le problème de répartition est un problème impliquant des tâches de type temps

réel, la minimisation du délai de communication est un critère adéquat. De plus, les tâches à répartir ne possèdent pas de relations de préséance[1], la minimisation du délai de communication permet aussi une meilleure utilisant des processeurs disponibles. En effet, les tâches voisines assignées à des processeurs voisins donnent un délai de communication plus court qu'une répartition dans des processeurs disparates.

5.4.1 Heuristique pour la fonction $g(n)$

D'après le développement théorique sur les caractéristiques de la fonction d'évaluation heuristique présenté dans le chapitre 4, la condition d'arrêt de la fouille heuristique exige que la fonction $g(n)$ soit calculée indépendamment du numéro d'itération (lemme 4.2.4.2). Nous devons donc calculer $g(n)$ en fonction de l'assignation n.

Une façon simple pour la réalisation de ce calcul consiste à rendre la fonction $g(n)$ récursive [PEA90]. Pour chaque assignation développée par la méthode DPSM, la valeur de la fonction $g(n)$ est la sommation de deux quantités. Ces quantités sont le coût de communication propre à la tâche contenue dans l'assignation n et la valeur de la fonction g du prédécesseur de l'assignation n. Nous écrivons d'une manière récursive

$$\begin{aligned} &g(s) = 0, \\ &g(n) = c(n) + g(\mathrm{PRED}(n)), \end{aligned} \qquad (5.4.1.1)$$

où s est l'assignation initiale, $c(n)$ est le coût de communication de la tâche contenue dans l'assignation n et PRED(n) est l'assignation précédente à n dans l'horaire partiel en cours. L'équation (5.4.1.2) donne une façon pour le calcul de $c(n)$.

$$c(n) = \sum_{T',P' \in \Gamma(n.T)} C(T', n.T, P', n.P) \qquad (5.4.1.2)$$

où $C(T', n.T, P', P)$ est le délai de communication entre la tâche T' dans le processeur P' et la tâche T dans le processeur P de l'assignation n. Les tâches $T' \in \Gamma(n.T)$ sont les

[1] Les regroupements avec relations de préséance sont prétraités par l'heuristique OL* de la section 5.2.

tâches voisines de $n.T$. La fonction $C(\cdot)$ a été décrite dans la section 3.3.4 du chapitre 3. À noter que certaines tâches voisines de $n.T$ peuvent ne pas se trouver dans l'horaire partiel courant. Dans ce cas, la fonction $C(\cdot)$ est nulle pour ces tâches voisines non encore réparties.

D'après les équations (5.4.1.1), la fonction $g(n)$ est une fonction croissante. La fonction $g(n)$ prendra sa valeur maximale lorsque la fouille aura développé la dernière assignation de l'horaire.

5.4.2 Heuristique pour la fonction $h(n)$

D'après le développement théorique de la fonction d'évaluation heuristique présenté dans le chapitre 4. La condition d'arrêt de la fouille heuristique et la possibilité d'obtenir un horaire complet exigent un nombre de caractéristiques spécifiques à la fonction $h(n)$. Dans le cas du critère d'arrêt de la fouille, la fonction $h(n)$ doit être conçue de sorte que $h(n) \leq h^*(n)$. Dans le cas de l'obtention d'un horaire complet, la fonction $h(n)$ doit satisfaire la condition $h(n) \geq h(S(n))$ où $S(n)$ est une assignation qui succède à n. Ces deux caractéristiques sont équivalentes à l'admissibilité (définition 4.4.4.1, chapitre 4) et à la monotonie (définition 4.4.4.2, chapitre 4) de la fonction d'évaluation heuristique $f(n)$.

Ces deux caractéristiques de la fonction $h(n)$ sont difficiles à obtenir. En effet, à cause de la nature heuristique de cette fonction, nous ne pouvons pas caractériser son comportement pour toutes les instances de problème soumises au répartiteur. C'est pour cette raison que la méthode DPSM utilise la fonction $\max(f(\text{PRED}(n), g(n)+h(n))$ (théorème 4.4.4.1) pour garantir la monotonie de fonction $f(n)$. Nous établissons la fonction $h(n)$ de la manière suivante. Pour chaque assignation n, la valeur $h(n)$ est le coût de communication des tâches *non réparties* du système des tâches $\mathcal{T}$. De plus, le calcul du coût de communication est réalisé en supposant que toutes les tâches non réparties

seront allouées dans des processeurs voisins sans aucune contention dans la communication. Donc,

$$h(n) = \sum_{T',T'' \in H} \frac{\lambda(T',T'')}{L} + \sum_{T' \in \Gamma(n.T)} \frac{\lambda(n.T,T')}{L} \quad T' \neq T''. \tag{5.4.2.1}$$

Dans le calcul de l'équation (5.4.2.1) le paramètre H représente l'horaire partiel en cours et L est la vitesse de transmission d'un lien du réseau des processeurs. Il est à remarquer que la tâche contenue dans l'assignation n est également considérée pour satisfaire à la définition de $h(n)$. Le coût de communication ainsi calculé est le coût d'un scénario optimiste dans lequel toutes les tâches non réparties seront allouées dans des processeurs voisins. La fonction $h(n)$ est décroissante puisque les tâches assignées ne sont pas incluses dans le calcul. Le nombre de tâches non réparties doit diminuer avec la progression de la répartition.

5.5 Évaluation de la performance du répartiteur

Nous avons effectué une série de simulations afin de vérifier l'efficacité du répartiteur des tâches. Les résultats de ces simulations sont ensuite classés en trois catégories. Les résultats de la première catégorie sont issues d'une étude comparative entre la méthode de fouille A^* et la méthode de fouille DPSM. Cette étude comparative consiste à vérifier le nombre d'assignations développées par la méthode DPSM en fonction de la quantité de mémoire utilisée. Cette comparaison a pour but de valider notre méthode de fouille par rapport à une méthode reconnue comme une méthode optimale. La seconde catégorie détermine d'une manière empirique le nombre d'assignations développées par la méthode DPSM en fonction de la taille des systèmes de tâches. Puisque le temps d'exécution des méthodes de fouille est proportionnel au nombre d'assignations développées, cette étude donne également un aperçu de la complexité temporelle du répartiteur. Enfin, la dernière catégorie concerne les résultats montrant la capacité du répartiteur dans l'obtention d'un horaire acceptable pour

différents systèmes de tâches. Il s'agit de la mise en œuvre complète du répartiteur des tâches. L'utilité du répartiteur est démontrée par les résultats obtenus.

5.5.1 Comparaison des méthodes : nombre de développements

Les conditions de cette comparaison sont données dans le tableau 5.6. Le nombre de tâches pour chacun des essais est limité parce que la méthode A* possède une complexité spatiale qui est exponentielle. Les systèmes de tâches sont générés d'une manière aléatoire sans relations de préséance. La topologie d'interconnexion du réseau des processeurs est celle de la figure 5.6. Le nombre de tâches voisines est réglé à 5 (maximum) afin de réduire le temps de calculs de la fonction d'évaluation heuristique $f(n)$.

Tableau 5.6 Conditions des essais pour la comparaison entre les méthodes A* et DPSM.

Paramètre	Caractéristique
Nb. d'essais	400
Taille des essais	$N_T = \{10, 20, 30, 40\}$
Nb. d'essais par taille	100
Temps d'exécution des tâches	100 unité de temps
Temps de communication	10 unités de temps
Échéancier des tâches	200 unités de temps
Nb. de tâches voisines	Maximum 5 tâches voisines
Nb. de processeurs	Même nombre que N_T
Mémoire allouée à la fouille	$M = M_{DPSM} / M_{A^*} = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$

L'objectif de cette première comparaison est d'obtenir le nombre d'assignations développées par la méthode DPSM en fonction de la mémoire allouée pour la fouille et ce, *sans enregistrement d'horaires partiels sur disque*. C'est pour cette raison que la quantité de mémoire allouée à la méthode DPSM est un paramètre défini en fonction de

la mémoire utilisée par la méthode A*. Ainsi, M = 0.1 signifie que la quantité de mémoire allouée à la méthode DPSM s'élève à 10% de celle utilisée par la méthode A* pour solutionner le même problème. Enfin, les heuristiques utilisées sont les mêmes pour les deux méthodes.

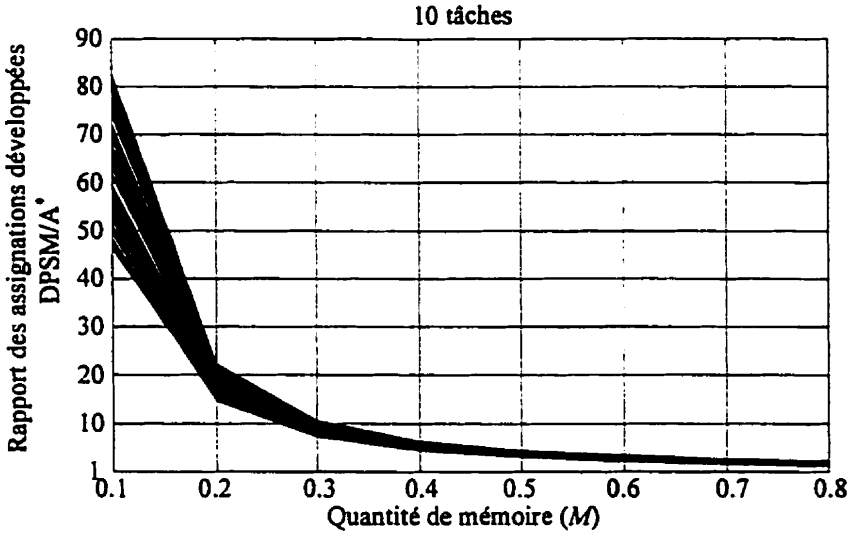


Figure 5.11 Assignations développées en fonction de la mémoire disponible (10 tâches).

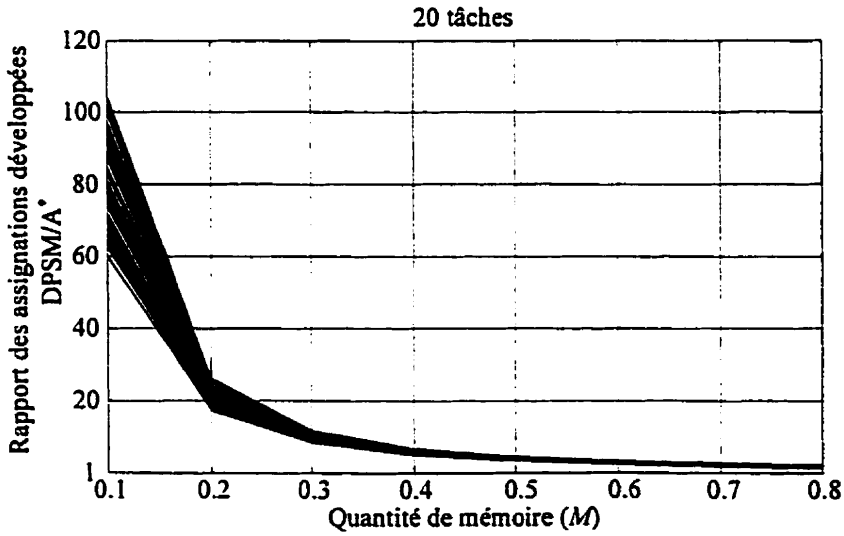


Figure 5.12 Assignations développées en fonction de la mémoire disponible (20 tâches).

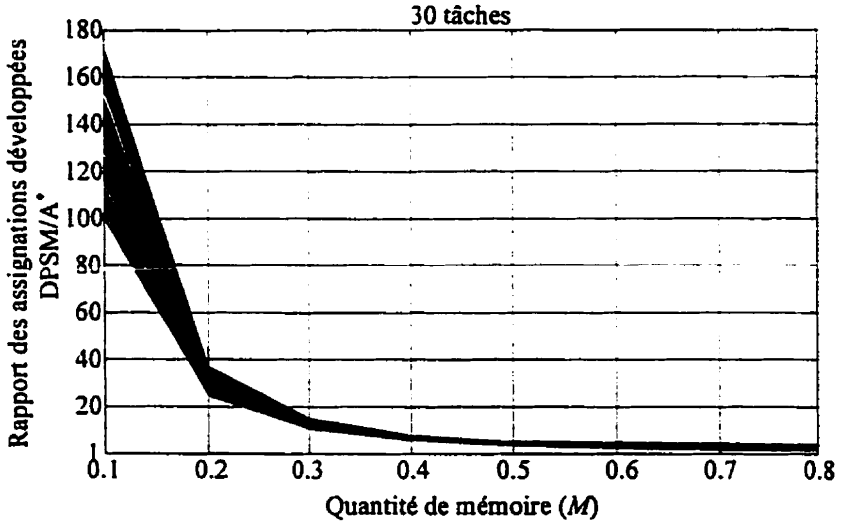


Figure 5.13 Assignations développées en fonction de la mémoire disponible (30 tâches).

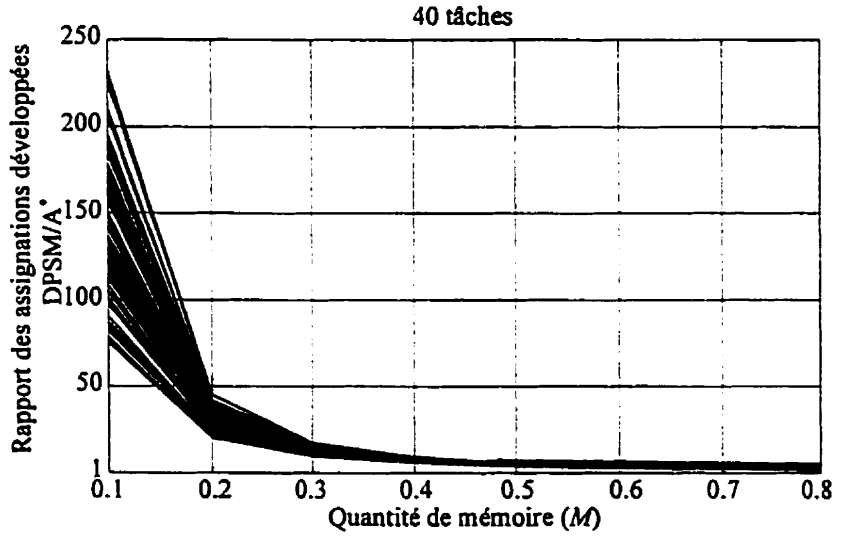


Figure 5.14 Assignations développées en fonction de la mémoire disponible (40 tâches).

Les figures 5.11 à 5.14 montrent le nombre d'assignations développées par la méthode DPSM en fonction de la quantité de mémoire allouée pour la fouille. La quantité de mémoire disponible pour la fouille M est calculée par rapport à la quantité de mémoire utilisée par la méthode A*. La tendance générale de ces quatre graphiques démontre bien l'effet de la limitation mémoire sur le comportement de la fouille. En

effet, le nombre de développements effectués par la méthode DPSM est inversement proportionnel à la quantité de mémoire disponible. Plus la quantité de mémoire allouée est petite, plus la méthode développe un grand nombre d'assignations candidates. Autrement dit, la limitation de la mémoire disponible provoque le développement répété des assignations élaguées. On peut observer ce même comportement de la méthode DPSM pour des systèmes de tâches de taille différente.

Les figures 5.15 à 5.19 montrent un autre ensemble d'essais. Cette fois, la capacité d'enregistrement des horaires partiels de la méthode DPSM est utilisée. L'enregistrement des horaires partiels sur disque permet à la méthode de fouiller des chemins plus longs. En reléguant la fouille à des applications subséquentes de la méthode, on doit pouvoir diminuer le nombre de développements répétés.

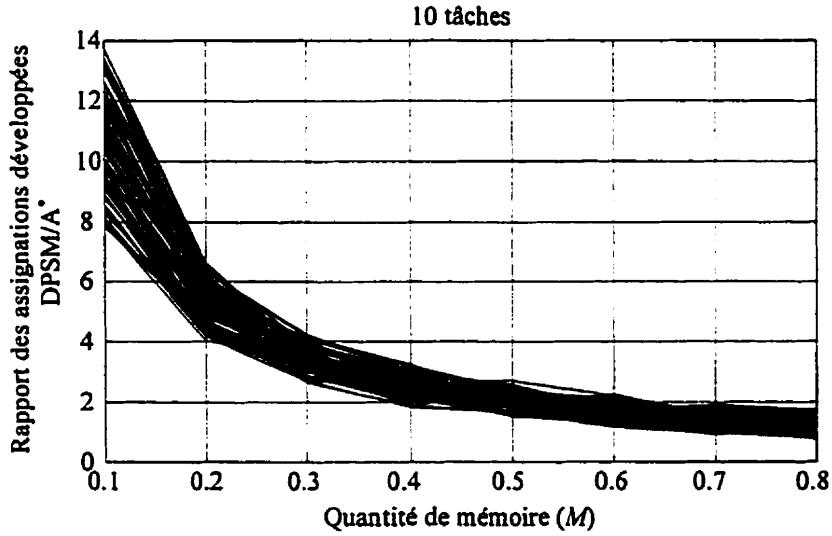


Figure 5.15 Assignations développées en fonction de la mémoire disponible (10 tâches).

Pour des ensembles comprenant 10 tâches, la réduction du nombre de développements est remarquable. À 10% de la mémoire utilisée par A^*, l'enregistrement des horaires partiels a réduit le nombre de développements par un facteur de 5. Sans l'enregistrement des horaires partiels, la méthode DPSM développe 80 fois plus

d'assignations que la méthode A*. Tandis que l'utilisation de l'heuristique d'enregistrement permet à cette méthode de développer seulement 14 fois plus d'assignations. La mémoire disponible pour la fouille étant très restreinte (10% de celle utilisée par A*), ces résultats sont très encourageant.

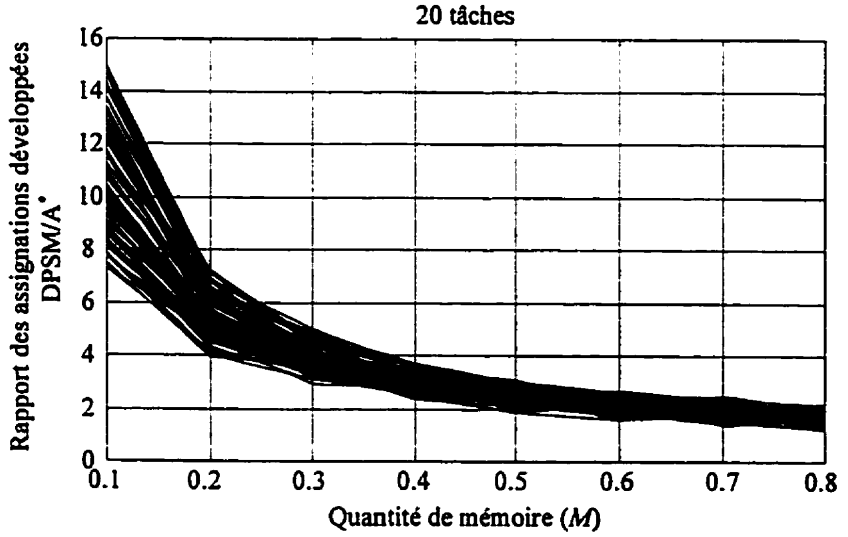


Figure 5.16 Assignations développées en fonction de la mémoire disponible (20 tâches).

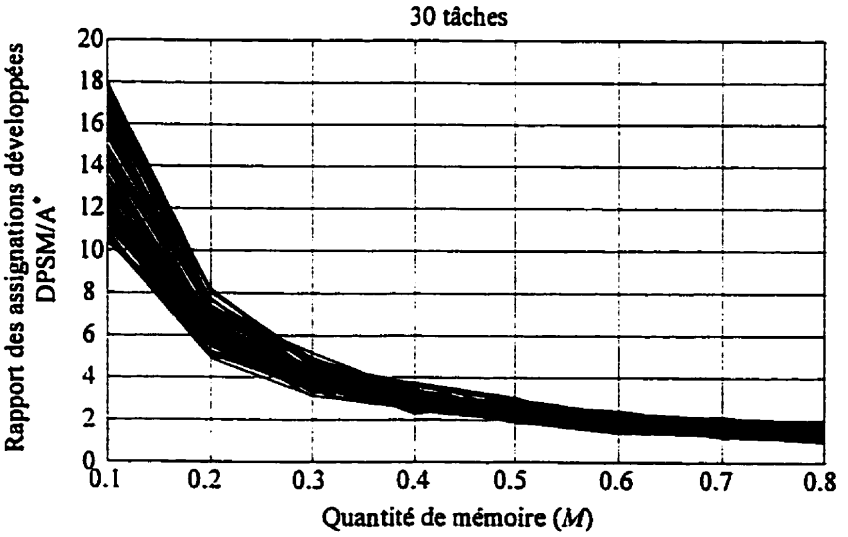


Figure 5.17 Assignations développées en fonction de la mémoire disponible (30 tâches).

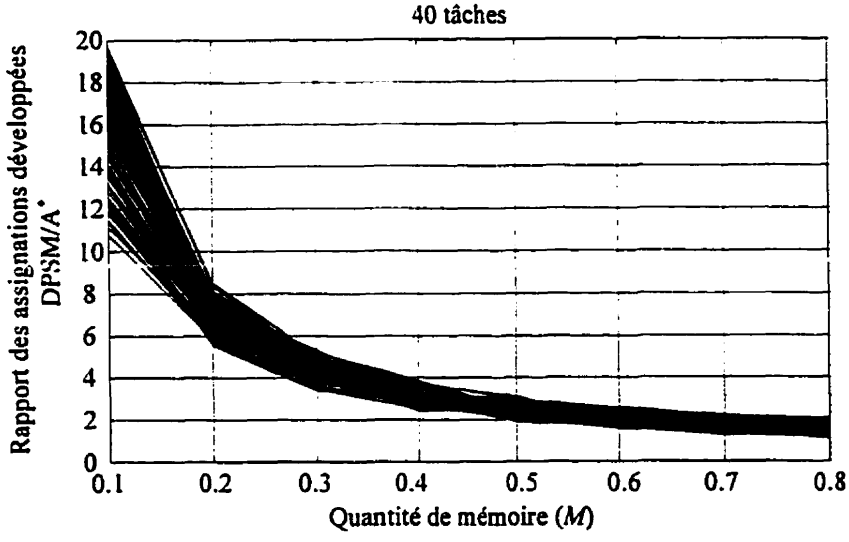


Figure 5.18 Assignations développées en fonction de la mémoire disponible (40 tâches).

L'heuristique d'enregistrement des horaires partiels est d'autant plus efficace que le nombre de tâches à traiter est grand. Pour des systèmes de 30 et 40 tâches, le nombre de développements est réduit par un facteur de 9 et de 11 à 10% de la mémoire utilisée par la méthode A*. On ne peut pas extrapoler cette tendance à des systèmes de tâches de taille supérieure. La quantité de mémoire exigée par la méthode A* est exorbitante puisqu'elle conserve toutes les assignations développées en mémoire. Il a été impossible d'appliquer la méthode A* pour des problèmes de plus grande dimension. Donc, l'avantage de la méthode DPSM est sa capacité de fouille des chemins plus longs et du même coup de traitement des problèmes de grande taille. Ce point fort est montré dans les résultats de la section suivante.

5.5.2 Nombre de développements vs longueur des solutions

L'heuristique d'enregistrement des horaires partiels permet à la méthode DPSM de traiter des problèmes de grande dimension. Pour démontrer cette affirmation, nous avons relevé le nombre d'assignations développées par rapport à la cardinalité des systèmes de tâches. La figure 5.19 présente la moyenne des assignations développées pour des

systèmes de 10 à 600 tâches. Dix essais ont été effectués pour chacun des systèmes de tâches. La quantité de mémoire réservée pour la méthode DPSM est réglée à 50% de la mémoire utilisée par la méthode A* pour la solution d'un problème de 20 tâches. Les conditions de ces essais sont données dans le tableau 5.7.

Tableau 5.7 Conditions des essais sur la longueur des solutions.

Paramètre	**Caractéristique**
Nb. d'essais	100
Taille des essais	$N_T = \{10, 20, 30, 40, 100, 200, 300, 400, 500, 600\}$
Nb. d'essais par taille	10
Temps d'exécution des tâches	100 unité de temps
Temps de communication	10 unités de temps
Échéancier des tâches	200 unités de temps
Nb. De tâches voisines	Maximum 5 tâches voisines
Nb. de processeurs	Même nombre que N_T
Mémoire allouée à la fouille	$M = M_{DPSM} / M_{A^*} = 0.5$ à $N_T = 20$.

Dans ces essais, la méthode de fouille A* est incapable de traiter des systèmes supérieurs à 40 tâches. Cette méthode emmagasine toutes les assignations générées en mémoire afin de réduire le nombre de développements nécessaires pour l'obtention d'une solution. La méthode DPSM, quant à elle, dispose d'une quantité de mémoire limitée pour réaliser la fouille et elle ne conserve en mémoire que les assignations développées. Lorsque le nombre d'assignations d'un horaire partiel dépasse la quantité de mémoire disponible, la méthode DPSM enregistre l'horaire partiel sur disque et concentre sa fouille sur un autre horaire partiel. Après avoir fouillé les horaires partiels d'une certaine longueur, la méthode est appliquée de nouveau sur les horaires partiels enregistrés et ainsi de suite jusqu'à l'obtention d'un horaire complet.

Les résultats de la figure 5.19 montrent cette capacité de la méthode DPSM. Le nombre d'assignations développées est une fonction exponentielle de la taille des problèmes mais la dimension des problèmes traités est bien plus grande. En effet, nous pouvons réaliser, à l'aide de cette méthode, une répartition impliquant des centaines de tâches. Tandis que la fouille par A* est limitée à des systèmes de petites tailles (environs 40 dans nos essais).

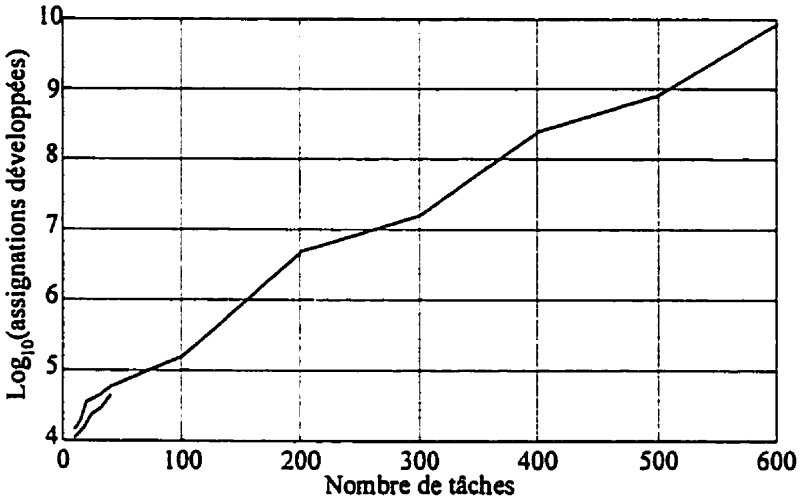


Figure 5.19 Assignations développées en fonction de la taille des problèmes.

La méthode DPSM développe en moyenne 3 millions d'assignations pour des systèmes comprenant 200 tâches. Alors que le nombre d'assignations développées est de l'ordre de 6.3 milliards pour des systèmes de 600 tâches. Même si ces valeurs sont énormes, elles sont néanmoins très inférieures à celles d'une énumération exhaustive.

Nous avons mesuré le temps d'exécution des essais sur un poste de travail SUN de modèle Enterprise 6000 et un ordinateur personnel compatible PC. Le poste de travail comprend 16 processeurs RISC UltraSPARC à 167MHz, 2Go de mémoire vive avec comme système d'exploitation le Solaris 2.5. L'ordinateur personnel PC possède un processeur AMD K2 à 266Mz, 256 Mo de mémoire vive et le système d'exploitation utilisé est le système Windows 95 version 4.00.950B (OSR2). La méthode DPSM a été

codée en langage C++ et est conforme à la norme ANSI. Pour le poste de travail Sun, le code a été compilé, sans aucune option d'optimisation supplémentaire, par le compilateur SPARCworks version 4.2. Quant à la version réalisée pour l'ordinateur personnel, le compilateur utilisé est le Borland C++ version 5.1. Cette compilation a été également réalisée sans optimisation du code généré.

Le programme réalisant la méthode DPSM est un programme entièrement séquentiel. Pour le poste de travail Enterprise 6000, l'exécution du programme est réalisée en mode mono-utilisateur et les appels de système sont exécutés en parallèle par le système d'exploitation. Cet avantage du poste de travail SUN n'est pas présent dans l'ordinateur personnel. Cependant l'ordinateur personnel est essentiellement une machine monotâche. Donc, l'exécution du programme n'est jamais suspendue par le système d'exploitation de l'ordinateur. Ces mesures ont été prises pour vérifier le comportement de la méthode dans différents environnements d'exécution. Enfin, les systèmes de tâches à répartir sont identiques pour les deux ordinateurs utilisés.

Tableau 5.8 Temps d'exécution de la méthode DPSM.

Nombre de tâches	Compatible PC	Poste de travail
10	10.3 secondes	2.2 secondes
20	16 secondes	5.1 secondes
30	31.4 secondes	7.7 secondes
40	57.7 secondes	18 secondes
100	3.1 minutes	43.4 secondes
200	63 minutes	15.7 minutes
300	5.7 heures	1.8 heures
400	7.2 heures	3.6 heures
500	23.9 heures	10.6 heures
600	6.7 jours	2.3 jours

D'après le tableau 5.8, l'exécution de la méthode est plus rapide dans le poste de travail SUN. Cependant, le système à 600 tâches exige quand même deux jours et demi de travail pour l'ordinateur SUN. Les temps d'exécution obtenus montrent bien l'importance du sous-système d'E/S des ordinateurs. En effet, l'enregistrement des horaires partiels effectués par la méthode DPSM exige un système de fichiers qui est performant. L'efficacité du sous-système d'E/S et du système de fichier de la machine Enterprise 6000 est nettement supérieure à celle de l'ordinateur personnel. Même si la cadence du processeur AMD K2 est plus élevée que le processeur UltraSPARC, le temps d'exécution est plus long pour l'ordinateur personnel.

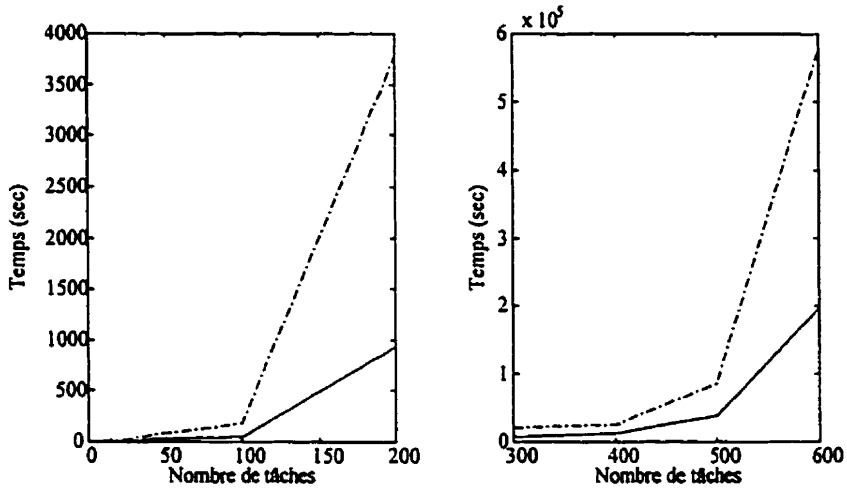


Figure 5.20 Représentation graphique des résultats du tableau 5.8.

Le temps d'exécution de la méthode DPSM peut sembler très long. Plus de deux jours de calculs sont nécessaires pour répartir un réseau de 600 tâches. Or, un réseau électrique de cette envergure peut comprendre plusieurs centaines de barres et de composants. L'étude dynamique et statique en temps réel d'un tel réseau demande un grand effort de préparation. Nous croyons que le temps utilisé pour la répartition des tâches est encore bien en deçà du temps nécessaire pour le travail préparatoire des études de cette dimension.

5.5.3 Performances du répartiteur de tâches

Cette section présente la performance du répartiteur des tâches dans son ensemble. Les systèmes de tâches sont réglés pour simuler différents scénarios pratiques. Le répartiteur des tâches est appliqué en utilisant l'heuristique OL^+ (section 5.2) pour traiter les regroupements de tâches. D'après le tableau 5.5, l'efficacité de l'heuristique est maximale lorsque les regroupements comprennent au plus 20 tâches et lorsque le temps de communication des tâches est plus petit que le temps d'exécution des tâches. Ces limites sont utilisées dans les conditions des essais effectués.

L'échéancier des tâches est fixé à 50 unités de temps tandis que le temps d'exécution des tâches peut varier entre 5 à 40 unités de temps. Le temps de communication des tâches est réglé entre 1 à 10 unités de temps. Ainsi, plus le temps d'exécution et le temps de communication sont longs plus la sévérité des échéanciers est grandes. Le tableau 5.9 donne les conditions utilisées pour ces essais.

Tableau 5. 9 Conditions des essais pour la mesure de la performance du répartiteur.

Paramètre	Caractéristique
Nb. d'essais	500
Taille des essais	$N_T = \{10, 20, 30, 40, 100, 200, 300, 400, 500, 600\}$
Nb. D'essais par taille	50
Nb. De regroupements	0% à 50% de N_T
Nb. de tâches par regroupement	Maximum de 20 tâches
Temps d'exécution des tâches	10 à 40 unité de temps
Temps de communication	1 à 10 unités de temps
Échéanciers des tâches	50 unités de temps
Nb. de tâches voisines	Maximum 5 tâches voisines
Nb. de tâches avec E/S	0% à 50% de N_T
Nb. de processeurs	Même nombre que N_T
Mémoire allouée à la fouille	$M = M_{DPSM} / M_{A^*} = 0.5$ à $N_T = 20$.

La première série de résultats concerne l'effet de la sévérité des échéanciers sur l'efficacité du répartiteur de tâches. Dans ces essais, le nombre de regroupements et le nombre de tâches avec E/S sont maintenus à zéro.

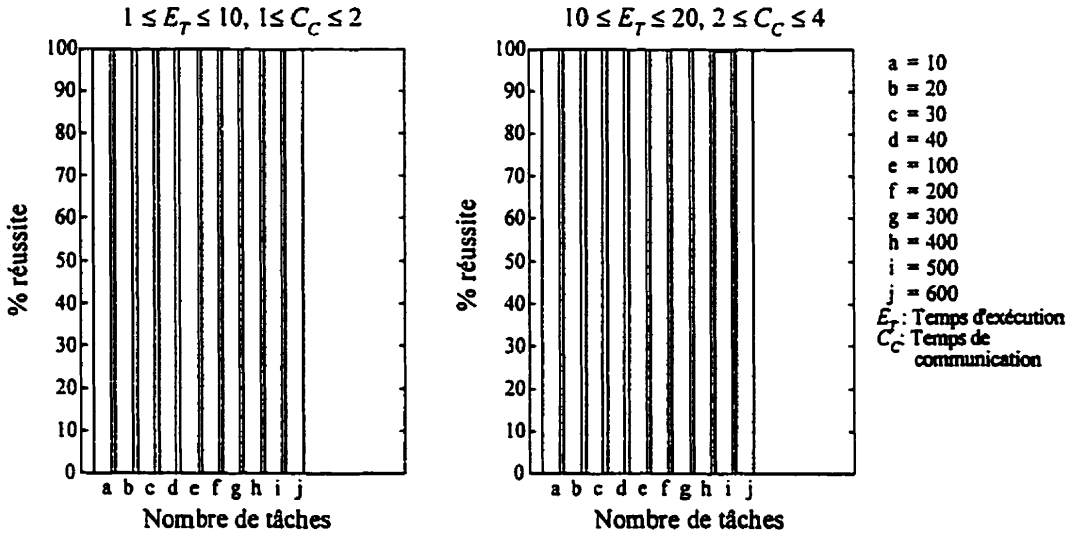


Figure 5.21 Pourcentage de succès du répartiteur de tâches ($1 \leq E_T \leq 20$, $1 \leq C_C \leq 4$).

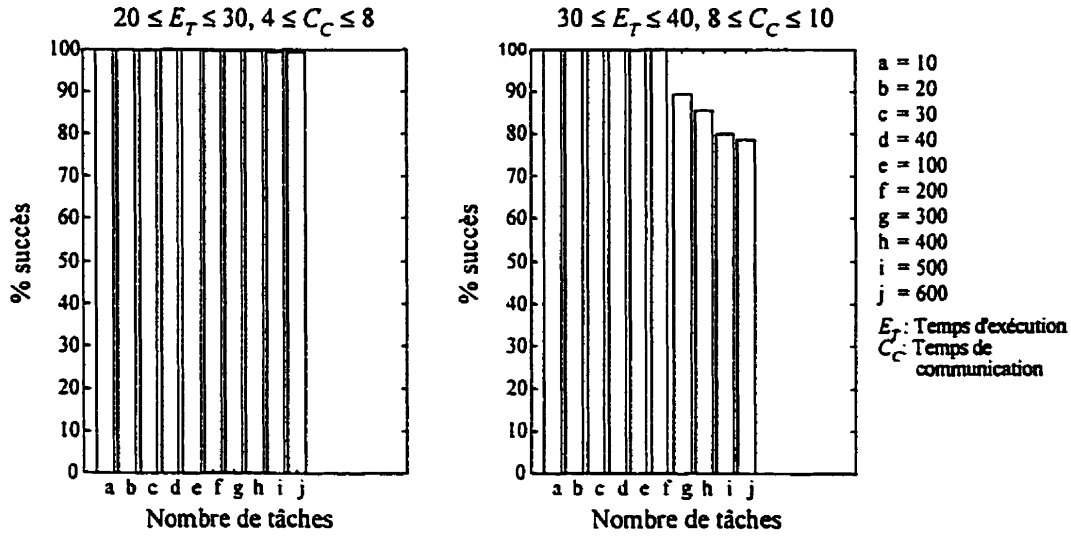


Figure 5.22 Pourcentage de succès du répartiteur de tâches ($20 \leq E_T \leq 40$, $4 \leq C_C \leq 10$).

Dans ces essais, nous mesurons donc le taux de réussite des répartitions en fonction de la sévérité des échéanciers des tâches. Les résultats de ces essais sont présentés dans les graphiques des figures 5.21 et 5.22. Le répartiteur des tâches est insensible lorsque les échéanciers sont peu sévères. Les problèmes de répartition sont résolus à 100% pour des tâches où le temps d'exécution maximal E_T et le temps de communication C_C sont à 40% et à 8% des échéanciers. Lorsque le temps d'exécution maximal E_T et un temps de communication C_C sont réglés à 60% et à 16% des échéanciers, le répartiteur demeure sans faille pour des systèmes de 200 tâches et moins. Une dégradation de performance est perceptible pour les systèmes de 300 tâches et plus mais le taux de succès demeure au-delà de 88.5%.

Pour des systèmes de tâches aux échéanciers très sévères, c'est-à-dire E_T à 80% et C_C à 20% des échéanciers, le répartiteur présente une très bonne performance pour $N_T \leq 40$. Cependant, le pourcentage de succès est réduit à 96% et ce, jusqu'à 78% pour des systèmes de taille supérieure à 100 tâches. En effet, l'étroitesse des échéanciers est très exigeante dans ces derniers essais. Le nombre de solutions possibles est réduit considérablement dans ces problèmes. Donc, il est très difficile de sortir d'une impasse causée par les erreurs des heuristiques lorsque les échéanciers sont très étroits.

Une deuxième série d'essais ont été réalisés pour vérifier l'influence des regroupements de tâches sur la performance du répartiteur. Pour ces essais, le temps d'exécution des tâches est fixé à $1 \leq E_T \leq 30$ unités de temps et le temps de communication est borné entre $1 \leq C_C \leq 8$ unités de temps. Ces valeurs de E_T et C_C ont été choisies pour ne pas perturber la performance normale du répartiteur de tâches. Le nombre de regroupements est ajusté pour totaliser entre 20% à 50% de la taille des ensembles de tâches. Le nombre maximal de tâches dans les regroupements est de 20 tâches.

Les figures 5.23 et 5.24 montrent bien qu'il n'y a aucune dégradation de la performance causée par le nombre de regroupements considérés dans le système des tâches.

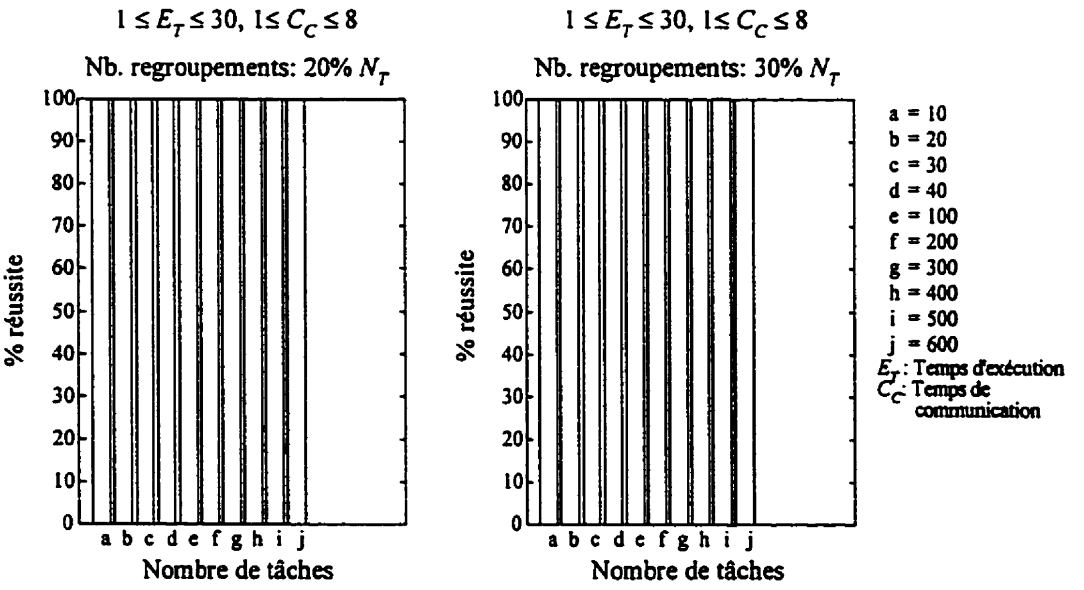


Figure 5.23 Pourcentage de succès en fonction du nombre de regroupements (#1).

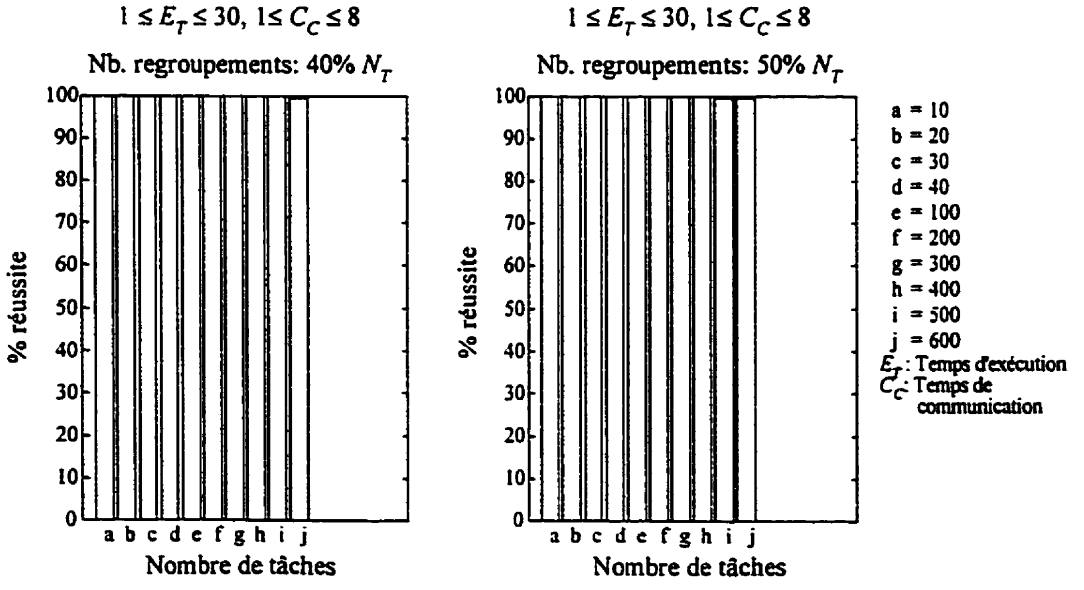


Figure 5.24 Pourcentage de succès en fonction du nombre de regroupements (#2).

Le prétraitement des regroupements de tâches par l'heuristique OL^+ simplifie même le travail de la méthode de fouille. En effet, plus il y a de regroupements de tâches prétraités, plus l'effort de fouille est petit. Le nombre de tâches restantes à répartir est moindre et l'espace de fouille est délimité par le prétraitement des regroupements de tâches.

Le découpage de l'espace de fouille est davantage apparent lorsqu'il y a des tâches utilisant des ports d'E/S. Ces tâches doivent être pré-allouées sur des processeurs particuliers du réseau des processeurs. En conjonction avec le prétraitement des regroupements de tâches, l'espace de fouille est largement élagué. Par conséquent, les chemins intéressants dans le graphe d'états est aussi considérablement réduits.

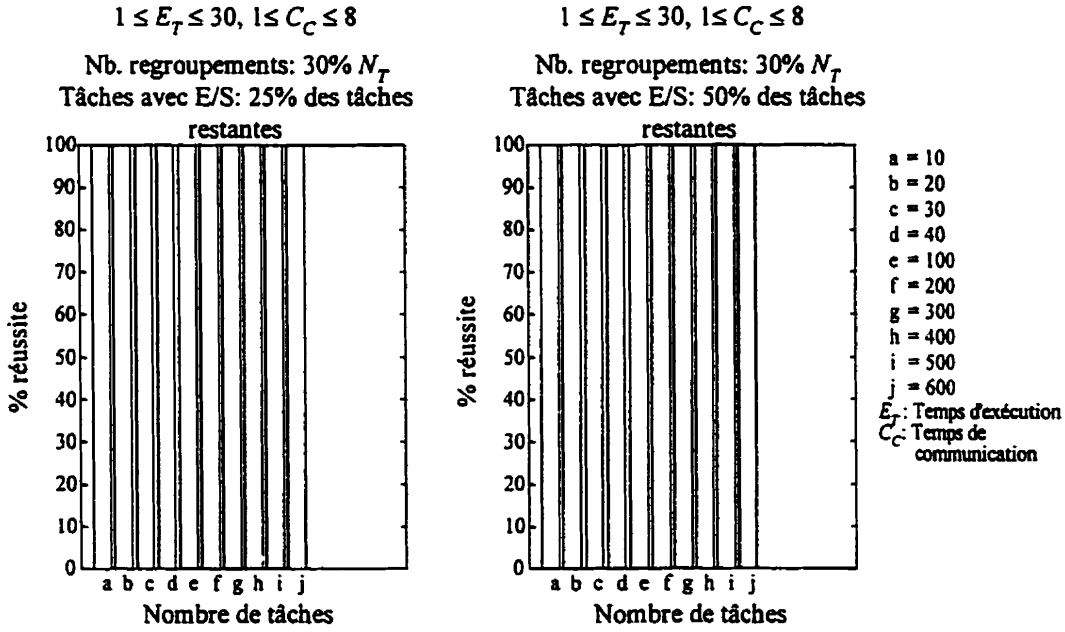


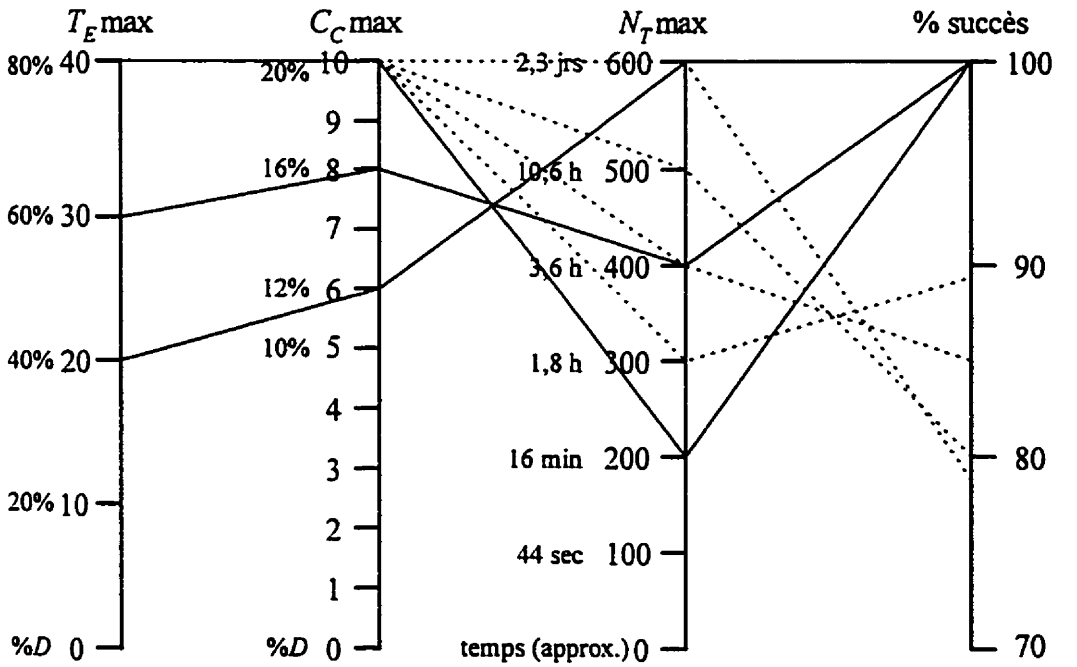
Figure 5.25 Pourcentage de succès en fonction du nombre de tâches utilisant un port d'entrée/sortie.

Il n'est pas surprenant que les résultats de ces essais (figure 5.25) soient si favorables pour le répartiteur des tâches. La préallocation des tâches d'E/S et le

prétraitement des regroupements de tâches favorisent l'obtention d'un horaire complet. Ils imposent des contraintes spatiales non triviales à la méthode de fouille. Si un horaire complet existe alors le répartiteur peut le trouver facilement. Cependant, nous devons être vigilants dans l'interprétation de ces résultats puisque la sévérité des contraintes spatiales produit souvent des problèmes sans solution.

5.5.4 Graphe des performances

À la lumière de ces résultats obtenus, nous sommes en mesure de présenter un bilan des performances du répartiteur des tâches. Ce bilan est présenté sous forme d'un graphique à quatre axes et peut avoir une application directe et pratique pour les utilisateurs du répartiteur de tâches. La figure 5.26 affiche les courbes des paramètres intéressants du répartiteur de tâches.



%D : pourcentage du temps par rapport à l'échéancier des tâches
T_E : temps d'exécution des tâches. C_C : temps de communication des tâches.
N_T : Nombre de tâches.

Figure 5.26 Courbes asymptotes reliant les paramètres du répartiteur de tâches.

Quatre axes sont utilisés dans le graphe des performances de la figure 5.26. Ces axes sont : *i*) temps d'exécution des tâches (T_E); *ii*) temps de communication des tâches (C_C); *iii*) cardinalité des systèmes de tâches (N_T); *iv*) systèmes de tâches répartis en pourcentage. Les valeurs de $\%D$ accompagnant les axes T_E et C_C servent à généraliser l'utilisation de ce graphe des performances. Elles établissent une correspondance entre les valeurs des axes T_E et C_C et la valeur des échéanciers des tâches. Ainsi, $T_E = 20$ représente la portion du temps correspondant à 40% de l'échéancier des tâches. Enfin, le nombre de regroupements présents ainsi que le nombre de tâches utilisant les ports d'entrée/sortie ne sont pas pris en compte dans le graphe des performances. Leurs influences sont minimes pour le répartiteur de tâches (voir les figures 5.23 à 5.25).

La lecture de ce graphe des performances est fort simple. Il suffit de suivre les traits pleins du graphique de gaucher vers la droite. Par exemple, un système de tâches avec un temps d'exécution T_E max à 40% de l'échéancier D avec un temps de communication C_C max à 12% de D, peut avoir un taux de succès de 100% si $N_T \leq 600$ tâches. Il en est de même pour un système de tâches dont le T_E max est à 60% de D et un C_C max de 16%. Le taux de succès pour ces systèmes de tâches est de 100% si $N_T \leq 400$. Enfin, lorsqu'un système de tâches possède un T_E max égal à 80% de l'échéancier D et un C_C max correspondant à 20% de D alors le répartiteur peut obtenir jusqu'à 100% de succès si le nombre de tâches dans le système est inférieur ou égal à 200 tâches.

Quant à l'utilisation de ce graphe des performances, elle est fort pratique. En effet, chaque trait plein reliant les axes est une borne supérieure estimée à partir des résultats de simulation. Par exemple, la courbe reliant $T_E \leq 60\%$, $C_C \leq 16\%$, $N_T \leq 400$ donne un pourcentage de succès maximal de 100%. Toutes les valeurs de T_E, C_C et N_T qui sont inférieures à cette borne estimée donneront également un pourcentage de succès maximal identique. Donc, chacune des valeurs de T_E, C_C et N_T est régit par la borne immédiate supérieure des axes. Si le système à répartir contient des tâches dont le temps d'exécution maximal et le temps de communication sont à 55% et 18% de l'échéancier,

alors le nombre de tâches ne doit pas dépasser 200 pour obtenir un pourcentage de succès maximal de 100%. Évidemment, ces bornes estimées sont très conservatrices. À cause de la nature heuristique du répartiteur de tâches, le graphe des performances ne peut pas être établi d'une manière absolue.

Nous avons donné, dans la figure 5.26, une série de bornes estimées en trait pointillés. Ces bornes estimées sont données en fonction de différentes valeurs de N_T pour $T_E \leq 80\%$ et $C_C \leq 20\%$ de D. Une échelle donnant le temps d'exécution du répartiteur est également disponible dans le graphe. Cette échelle de temps correspond au temps de réponse approximatif du répartiteur pour différents systèmes de tâches. Ainsi, un utilisateur peut connaître rapidement la performance du répartiteur de tâches en consultant les courbes du graphe de performances.

Chapitre 6

Conclusion

6.1 Résumé

L'objectif principal de cette thèse consistait à apporter une contribution à la méthodologie utilisée dans la répartition automatique des tâches parallèles de type temps réel. Plus précisément, ce projet de recherche impliquait une étude du problème de répartition des tâches avec des échéanciers rigides. Ces tâches peuvent avoir ou non des relations de préséance. Les résultats théoriques et pratiques de ce travail visaient essentiellement les applications de simulation en temps réel des réseaux électriques. Les sous-objectifs spécifiques de cette recherche ont été définis à la section 1.2 du chapitre 1. La méthodologie de recherche utilisée pour atteindre ces sous-objectifs a été également définie à la section 1.2 du chapitre 1. La problématique de la répartition des tâches parallèles pour la simulation en temps réel a été présentée au chapitre 2. La modélisation et le développement d'une nouvelle méthode de répartition des tâches ont été présentés aux chapitres 3 et 4. En fin, les résultats obtenus du répartiteur de tâches ont été exposés au chapitre 6.

6.2 Contribution méthodologique

Cette contribution est une méthode pour la répartition des tâches parallèles de type temps réel avec échéanciers rigides. De plus, certains sous-ensembles de tâches utilisent

des ressources autres que les processeurs et peuvent posséder des relations de préséance dans l'exécution de leur travail.

La répartition de ce type de tâches dans un réseau de processeurs est un problème de complexité **NP-complet**. Une preuve de complexité, sous la forme d'un problème de décision, a été présentée. Le répartiteur proposé est donc basé sur une méthodologie heuristique. Cette méthode est formée d'un ensemble d'heuristiques et comprend trois phases de traitements : *i*) la préallocation; *ii*) le prétraitement; *iii*) la répartition des tâches qui ne sont pas préallouées ou prétraitées par une fouille heuristique.

La phase de préallocation consiste à répartir toutes les tâches utilisant des ports d'E/S dans des processeurs disposant des mêmes ressources. Cette préallocation des tâches est nécessaire. Certaines tâches du réseau électrique doivent interagir avec l'environnement externe via les convertisseurs AN/NA.

Le prétraitement des tâches, quant à lui, est utilisé pour ordonnancer des regroupements de tâches parallèles. Ces regroupements de tâches sont générés par la décomposition explicite des programmes simulant le comportement de certains composants complexes du réseau électrique. La cardinalité de ces regroupements de tâches est normalement petite par rapport au nombre total des tâches du réseau électrique. Une heuristique d'ordonnancement OL^+ est employée pour la phase du prétraitement. Cette heuristique est basée sur le principe des méthodes d'ordonnancement par liste de priorités. Dans l'heuristique OL^+, la priorité des tâches à ordonnancer est calculée en utilisant le niveau du graphe des tâches du regroupement. L'heuristique OL^+ effectue également un remplissage du temps mort des processeurs pour obtenir une meilleure utilisant des processeurs.

Les tâches, qui ne sont pas traitées par la préallocation et par le prétraitement, sont réparties par une méthode de fouille heuristique. Donc, la problématique de répartition

des tâches est considérée comme un problème de fouille. La méthode de fouille proposée s'opère dans un espace d'états. Ces états représentent l'ensemble des assignations candidates pour la formation d'un horaire d'exécution complet. Chaque assignation candidate est un couple composé d'une tâche T allouée dans un processeur P. Ainsi, le couple (T, P) est une assignation qui peut être sélectionnée par la méthode de fouille lors de la production d'un horaire d'exécution des tâches. Une analyse formelle des propriétés de la fouille dans l'espace d'états a été effectuée. Cette analyse a été réalisée à l'aide de la notion généralisée de la longueur des chemins dans les graphes d'états.

La méthode de fouille DPSM a été développée pour éviter la croissance exponentielle de la complexité spatiale. Le problème de débordement de la mémoire est normalement associé à la fouille heuristique. La méthode DPSM combine les avantages des méthodes de fouille à mémoire limitée et les idées des techniques itératives de fouille en profondeur d'abord. Dans DPSM, l'utilisation de la mémoire disponible est constante. Un horaire d'exécution des tâches de longueur arbitraire peut être obtenu en appliquant, d'une manière répétée, la méthode DPSM.

Enfin, une version expérimentale de ce répartiteur de tâches a été implantée dans le simulateur HYPERSIM du service simulation de l'Institut de Recherche d'Hydro-Québec. Le système HYPERSIM est un simulateur entièrement numérique conçu pour étudier les phénomènes transitoires dans les réseaux de distribution électrique.

6.3 Travaux futurs

Dans cette thèse, plusieurs points intéressants ont été soulevés. Premièrement, le prétraitement des tâches par l'heuristique OL^+ n'admet que des regroupements de tâches dont la cardinalité est relativement petite (environ 20 tâches/regroupement). Cette heuristique est suffisante pour la plupart des situations normales. Par contre, il serait

intéressant d'augmenter sa capacité pour des applications futures. Or, la complexité temporelle de OL^+ est $O(n^3)$. Par conséquent, la mise en œuvre de cette heuristique doit être modifiée pour des regroupements de plus grandes cardinalités.

Le temps de réponse du répartiteur de tâches est adéquat pour des réseaux électriques de 200 tâches et moins. Pour des réseaux de plus grande dimension, le temps de réponse du répartiteur est acceptable, lorsque comparé à l'effort de préparation nécessaire pour réaliser une étude de grande envergure. Cependant, il y a place à l'amélioration. Une technique qui peut réduire considérablement le temps de réponse du répartiteur est le traitement parallèle. En effet, la méthode DPSM possède un parallélisme qui est naturel. Lors de la construction d'un horaire complet, la méthode DPSM est appliquée, d'une manière répétée, à des horaires partiels obtenus par les applications précédentes de la fouille. Chaque horaire partiel représente une région de l'espace d'état. Ces horaires partiels sont normalement fouillés en séquence. Or, les informations contenues dans les horaires partiels sont suffisantes pour la méthode DPSM d'effectuer des fouilles indépendantes. Nous pouvons accélérer le travail de DPSM en traitant les horaires partiels générés en parallèle.

La version parallèle de la méthode DPSM peut être utilisée dans un réseau d'ordinateurs ou dans un réseau de processeurs disposant une mémoire locale. Dans ce contexte, une application de la méthode DPSM est exécutée sur un nombre de processeurs (ordinateurs) qui fouillent en parallèle les horaires partiels produits par l'application précédente. À la fin d'une application, les résultats sont récoltés. Si des horaires partiels sont produits, ils sont redistribués dans le réseau des processeurs (ordinateurs) et une nouvelle application parallèle de la DPSM est exécutée. Le schème est donc fort simple. Il est de type SIMD (Single Instructions Multiple Data) puisque tous les processeurs exécutent le même programme. Le nombre de données à transférer entre les processeurs (ordinateurs) est peu nombreux. Seulement la dernière assignation des horaires partiels est nécessaire pour la fouille.

Le balancement de la charge des processeurs (ordinateurs) est également facile à réaliser. Puisque les fouilles sont effectuées d'une manière indépendante. Nous pouvons diviser l'ensemble des horaires partiels en sous-ensembles de cardinalité proportionnelle à la puissance de traitement des processeurs (ordinateurs).

Une difficulté immédiate et apparente est la façon avec laquelle on doit réaliser la concaténation des horaires partiels après la récolte des résultats. Après chaque application parallèle de la méthode DPSM, nous devons joindre les horaires partiels obtenus. Si cet assemblage des horaires partiels est réalisé sur disque alors on risque de perturber fortement la performance de la fouille parallèle, puisque les traitements sur disque sont toujours plus lents que les traitements en mémoire. Par contre, l'assemblage en mémoire des horaires partiels exige un transfert de données plus imposant. Enfin, l'application parallèle de la méthode DPSM pour la construction d'un horaire d'exécution mérite d'être étudiée d'une manière plus approfondie.

Références

[ADA74] T. Adams, K. Chandy, J. Dickson, "A comparison of list schedulers for parallel processing systems," *Communications of ACM*, vol. 17, pp. 685-690, Dec. 1974.

[AAR94] M. Aarup et al., "*OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV*," in Knowledge Based Scheduling, San Mateo: Morgan Kaufmann, 1994.

[ADA92] Ada9X Mapping/Revision Team, *Real-time systems Annex*. Cambridge: Intermetrics Inc., 1992.

[ALI93] H. Ali, H. El-Rewini, "Task allocation in distributed systems: A split-graph model," *Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 14, pp. 15-32, 1993.

[ARN92] A. Arnold, I. Guessarian, *Mathématiques pour l'informatique*. Paris: Masson, 1992.

[ASH90] G. M. Asher, M. Summer, "Parallelism and the transputer for real-time high-performance control of AC induction motors," *IEE Proceedings*, vol. 137, no. 4, pp. 179-188, July 1990.

[BAC97] Jean Bacon, *Concurrent Systems*. Essex: Addison-Wesley, 1997

[BAS92] S. Al-Bassam, H. El-Rewini, B. Bose, T. G. Lewis, "Processor Allocation for Hypercubes," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 394-401, 1992.

[BER73] C. Berge, *Graphes et Hypergraphes*. Paris: Bordas, 1973.

[BER83] C. Berge, *Graphes*. Paris: Bordas, 1983.

[BUR90] A. Burns, "Scheduling hard real-time systems: A review," *Software Engineering Journal*, vol. 6, no. 3, pp. 116-128, 1990.

[CAS88] T. L. Casavant, J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141-154, Feb. 1988.

[CHE90] M. Chen, K. G. Shin, "Subcube allocation and task migration in hypercube multiprocessors," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1146-1155, Sept. 1990.

[CHE94] C. M. Chen, S. K. Tripathi, A. Blackmore, "A resource synchronization protocol for multiprocessor real-time systems," *Proceedings 1994 International Conference on Parallel Processing*, vol. 3, pp. 159-162, 1994.

[COF72] E. G. Coffman, R. Graham, "Optimal Scheduling for Two-Processor Systems," *ACTA Informatica*, no. 1, 1972.

[COF73] E. G. Coffman, P. J. Denning, *Operating systems Theory*. Englewood Cliff: NJ: Prentice-Hall, 1973.

[COF76] E. G. Coffman et al. *Computer and Job-Shop Scheduling Theory*. New York: John Wiley and Sons, 1976.

[COR93] T. H. Corman, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*. Cambridge: MIT Press, 1993.

[CUR91] K. W. Currie, A. Tate, "*O-Plan: the Open Planning Architecture*," Artificial Intelligence, vol. 52, no. 1, pp.49-86, 1991.

[DEC85] R. Dechter, J. Peral, "Generalized best-first search strategies and the optimality of A*," *Journal of Association for Computing Machinery*, vol. 32 , no. 3, pp. 505-536, 1985.

[DEC88] R. Dechter, J. Pearl, "The optimality of A*," in *Search in Artificial Intelligence*. L. Kanal, D. Kumar editors, Springer-Verlag, pp. 166-199. 1988.

[DEO82] N. Deo, C. Pang, "Shortest path algorithms: Taxonomy and annotation," *Technical Report CS-80-057*, Computer Science Department, Washington State University, 1982.

[DER74] M. L. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," in *Information Processing 74*, North-Holland Publishing Company, 1974.

[DER89] M. L. Dertouzos, A. Mok, "Multiprocessor on-line scheduling of hard real-time tasks," *IEEE Transactions on Software Engineering.*, vol. 15, no. 12, pp. 1497-1506, Dec. 1989.

[DOM69] H. W. Dommel, "Digital Computer Solution of Electromagnetic Transients in Single and Multiphase Networks," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-88, pp. 388-399, 1969.

[DUR93] R. C. Durie, C. Pottle, "An Extensible Real-Time Digital Transient Network Analyzer," *IEEE Transactions on Power Systems*, vol. 8, no. 1, pp. 84-89, Feb. 1993.

[EPP95] S. S. Epp, *Discrete Mathematics with Applications*, 2nd Edition, Boston: PWD Publishing Company, 1995.

[EVA79] S. Evan, *Graph Algorithms*. Computer Science Press, 1979.

[FAL93] D. M. Falcão, E. Kaszkurewicz, H. L. S. Almeida, "Application of Parallel Processing Techniques to the Simulation of Power System Electromagnetic Transients," *IEEE Transactions on Power Systems*, vol. 8, no. 1, pp. 90-96, Feb. 1993.

[FOX88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon et D. Walker, *Solving problems on concurrent processors*. Englewood Cliff, NJ: Prentice-Hall, 1988.

[FUC90] J. J. Fuchs et al., "PlanERS-1: An expert planning system for generating spacecraft mission plans," *IEE First International Conference on Expert Planning Systems*, Brighton, pp. 70-75, 1990.

[GAF91] J. D. Gafford, "Rate-monotonic scheduling," *IEEE Micro*, Jun 1991.

[GAR75] M. R. Garey, D. S. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM Journal of Computing*, vol. 4, no. 4, pp. 397-411, Dec. 1975.

[GAR78] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman and Company, 1979.

[GHO94] S. Ghosh, A. Mahanti, D. S. Nau, "ITS: An efficient limited-memory heuristic tree search algorithm," *AAAI 94*, pp. 1353-1358, 1994.

[GLO93] F. Glover, E. Taillard, D. De Werra, "*A User's Guide to Tabu Search*" Annal Operational Research, pp. 41, 3-28, 1993.

[GOL80] M. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*. New York: Academic Press, 1980.

[GRA76] R. Graham, "Bounds on the Performance of Scheduling Algorithms," in *Computer and Job Shop Scheduling Theory*. New York: John Wiley and Sons, pp. 165-227, 1976.

[HAR68] P. E. Hart, N. J. Nilsson, B. Raphael, "A formal basis for the heuristic determination of minimal cost path," *IEEE Transactions on Systems Science and Cybernetics*, SSC-4, no. 2, pp. 100-107, 1968.

[HAR72] P. E. Hart, N. J. Nilsson, B. Raphael, "Correction to a formal basis for the heuristic determination of minimum cost paths," *SIGART Newletter*, pp. 28-29, 1972.

[HÄR94] M. G. Härbour, M. H. Klien, J. P.Lehoczky, "Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 1, pp. 13-28, Jan. 1994.

[HEN89] P. V. Hentenryck, *Constraint Sstisfaction in Logic Programming*. Cambridge: MIT Press, 1989.

[HEN97] J. L. Hennessy, D. A. Patterson, *Computer Architecture A Quantitative Approach*. San Mateo: Morgan Kaufmann, 1997.

[HOR74] W. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quarterly*, vol. 21, pp. 177-185, 1974.

[HU61] T. C. Hu, "Parallel Scheduling and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841-848, 1961.

[HUY93] H. Le-Huy, J. C. Soumagne, "Digital Real-Time Simulation of Transmission Lines using Parallel Processors," *IMACS-TC1 4-th International Conference on Computational Aspects of Electromechanical Energy Converters and Drives*, Montréal, pp. 29-32, July 1993.

[IRE95] Service Simulation de Réseaux, "Manuel d'usager Prototype HYPERSIM," *Institut de recherche d'Hydro-Québec*, Québec, Canada, Décembre 1995.

[JAC55] J. R. Jackson, "Scheduling a Production Line to Minimize Maximum Tardiness," *Rapport Technique 43*, Management Science Research Project, University of California, 1955.

[JOH73] D. B. Johnson, "A Note on Dijkstra's Shortest Path Algorithm," *Journal of ACM*, vol. 20 pp. 385-388, 1973.

[KAR93] D. R. Karger, D. Koller, S. J. Phililips, "Finding the hidden path: time bounds for all pairs shortest path," *SIAM Journal on Computing*, vol. 22, no. 6, pp. 1199-1217, 1993.

[KOR85] R. E. Korf, "Iterative Deepening-A*: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97-109, Aug. 1985.

[KRI92] Ramesh Krishnamurti, Eva Ma, "An Approximation Algorithm for Scheduling Tasks on Varying Partition Sizes in Partitionable Multiprocessor Systems," *IEEE Transactions on Computers*, vol. 41, no. 12, pp. 1572-1579, Dec. 1992.

[LAW73] E. L. Lawler, "Optimal Sequencing of a Single Machine Subject to Precedence Constraints," *Management Science*, vol. 19, pp. 544-546, 1973.

[LAW83] E. L. Lawler, "Recents Results in the Theory of Machine Scheduling," in *Mathematical Programming: the State of the Art*, New York: Springer-Verlag, 1983.

[LEE92] C. Lee, D. Lee, M. Kim, "Optimal task assignment in linear array networks," *IEEE Transactions on Computers*, vol. 41, no. 7, pp. 877-880, 1992.

[LEH89] J. P. Lehoczky, L. Shua, Y. Ding, "The rate monotonic scheduling algorithm — exact characterization and average-case behavior," *IEEE 10-th Real-Time Systems Symposium*, Dec 1989.

[LEU82] J. Leung, J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237-250, 1982.

[LEW92] T. G. Lewis, H. El-Rewini, *Introduction to parallel computing*. Englewood Cliff, NJ: Prentice-Hall, 1992.

[LIN97] P. Linz, *An introduction to formal languages and automata*. London: Jones and Barlett Publishers, 1997.

[LIU73] C. L. Liu, J. W. Layland, "Scheduling algorithms for multiprogramming in hard-real time environment," *Journal of ACM*, vol. 20, no. 1, pp. 46-61, Jan 1973.

[MAC59] R. McNaughton, "Scheduling with Deadlines and Loss Functions," *Management Science*, no. 6, pp. 1-12, 1959.

[MAG92] P. C. Magnusson, G. C. Alexander, V. K. Tripathi, *Transmission Lines and Wave Propagation*. Boca Raton: CRC Press, 1992.

[MAH95] Y. Maharsi, V. Q. Do, V. K. Sood, S. Casoria, J. Bélanger, "HVDC Control System Based on Parallel Digital Signal Processors," *IEEE Transaction on Power Systems*, vol. 10, no. 2, pp. 995-1002, May 1995.

[MAR77] A. Martelli, "On the Complexity of Admissible Search Algorithms," *Artificial Intelligence*, vol. 8, no. 1, pp. 1-13, 1977,

[MAR84] R. Marcus, "An application of artificial intelligence to operations research," *Communication of ACM*, vol. 27, no. 10, pp. 1044-1052, 1984.

[MOR98] B. M. Moret, *The Theory of Computation*. Berkeley: Addison-Wesley, 1998.

[NEU90] P. Neuhaus, "Solving the Mapping-Problem: Experiences with a Genetic Algorithm," *Proceedings Parallel Problem Solving from Nature 1990*, Spring LNCS 496, pp. 170-175, 1990.

[NIL71] N. J. Nilsson, *Problem-solving methods in artificial intelligence*. New-York: McGraw-Hill, 1971.

[PEA90] J. Pearl, *Heuristique*. Toulouse: CEPADUES, 1990.

[POH73] I. Pohl, "The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving," *Proceedings of the 3-th International Joint Conference on Artificial Intelligence (IJCAI-73)*, Standford, Carlifornia, pp. 20-23, 1973.

[PRE92] W. H. Press et al., *Numerical Recipes in C The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992.

[QUE93] V. Q. Do, A. O. Barry, "A Real-Time Model of the Synchronous Machine Based in Digital Signal Processors," *IEEE Transactions on Power Systems*, vol. 8, no. 1, pp. 60-66, Feb. 1993.

[QUE96] V. Q. Do, J. C. Soumagne, G. Sybille, G. Cloutier, F. Giguère, "HYPERSIM, A Fully Digital Simulator for Power System Studies," Technical Paper, *Service Technique Simulation*, Institut de recherche d'Hydro-Québec, Canada, 1996.

[RAJ88] R. Rajkumar, L. Sha, J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," *IEEE 9-th Real-Time Systems Symposium*, Dec. 1988.

[REW90] H. El-Rewini et T. G. Lewis, "Scheduling parallel program tasks onto arbitrary Target machine," *Journal of parallel and Distributed Computing*, vol. 9, pp. 138-153, Jun. 1990.

[ROS94] C. Rosu, "Analyse des besoins informatiques d'Hypersim," *Service Simulation de Réseaux*, Institut de recherche d'Hydro-Québec, Québec, Canada, Août 1994.

[RUS92] S. Russell, "Efficient memory-bounded search methods," *ECAI-92 10th European Conference on Artificial Intelligence*, Vienna, Austria, pp. 1-5, 1992.

[SCH90] R. J. Schalkoff, *Artificial Intelligence: An Engineering Approach*. New York: McGraw-Hill, 1990.

[SHA94] L. Sha, R. Rajkumar, S. S. Sathaye, "Generalized rate-monotonic scheduling theory: a framework for developing real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, Jan. 1994.

[SHE93] C. Shen, K. Ramamritham, J. Stankovic, "Ressource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Transactions on Parallel and Distributed Computing*, vol. 4, no. 4, April 1993.

[SOL98] SUN Microsystems, *Solaris 2.6 operating system manual*. 1998.

[SRC97] Screamer Tools, *University of Pensylvania*, Adresse internet: http://www.cis.upenn.edu/~screamer-tools/screamer-intro.html, 1997.

[STA88] J. A. Stankovic, "Misconception about real-time computing," *IEEE Computer*, pp. 10-19, Oct. 1988.

[STA94] J. Stankovic, "Real-Time Computing: A Critical Enabling Technology," *Technical Report UM-CS-1994-058*, University of Massachusetts, 1994.

[STE82] W. D. Stevenson Jr, *Elements of Power System Analysis*. New York: McGraw-Hill, 1982.

[SUN97] SUN Microsystems, *Enterprise 6000 Hardware reference manual*. 1997.

[TAR83] R. Tarjan, *Data Structures and Network Algorithms*. Philadelphia: SIAM, 1983.

[TMS92] Texas Instruments Incorporated, *TMS320C40x User's Guide*. Digital Signal Processing Products, May 1996.

[TRE75] J. P. Tremblay, R. Manohar, *Discrete Mathematical Structures with Applications to Computer Science*. New York: McGraw-Hill, 1975.

[ULL73] J. D. Ullman, "Polynomial Complete Scheduling Problems," *Operating Systems Review*, vol. 7, no. 4, pp. 96-101, 1973.

[VAL85] L. G. Valiant, V.V. Vazirani, "NP is as easy as detecting unique solutions," *Proceedings of 17-th Annual ACM Symposium on Theory of Computing*, pp. 458-463, 1985.

[WON95] T. Wong, "La répartition automatique des tâches dans la simulation en temps réel des réseaux électrique," *Rapport technique EPM/RT-95/14*, Département de génie électrique et de génie informatique, École polytechnique de Montréal, 1995.

[XU94] J. Xu, D. L. Parnas, "On satisfying timing constraints in hard-real-time systems," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp.70-84, Jan. 1994.

[ZHA87] W. Zhao, K. Ramamritham, J. A. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 949-960, Aug. 1987.

ANNEXE A

Glossaire

TERME	DÉFINITION
Assignation	Un couple comprenant une tâche et un processeur.
Assignation candidate	Une assignation satisfaisant les contraintes temporelles et spatiales.
Cardinalité	Nombre d'éléments d'un ensemble donné.
Complexité **NP**	Une classe de problèmes de complexité temporelle non polynomiale.
Complexité **NP-Complet**	Une classe de problèmes de complexité temporelle non polynomiale. Par contre, les solutions de ces problèmes peuvent être vérifiée en un temps polynomial.
Complexité **P**	Une classe de problèmes de complexité temporelle polynomiale.
Diffusion publique	Dans les échanges d'informations, un processeur envoie les informations à tous les processeurs qui lui sont reliés.
État	Dans le contexte de cet thèse, un état est équivalent à une assignation.
Graphe disjonctif	Un graphe non orienté simple séparable en deux ensembles de sommets : un ensemble dont les sommets sont reliés entre eux par une arête et un ensemble dont les sommets sont indépendants les uns des autres.

Horaire acceptable	Un horaire d'exécution qui préserve les relations de préséance des tâches et satisfait toutes les contraintes temporelles et spatiales imposées.
Horaire acceptable complet	Un horaire acceptable contenant toutes les tâches du système des tâches.
Horaire complet	Il s'agit d'un synonyme de horaire acceptable complet.
Horaire faisable	Un horaire d'exécution qui préserve les relations de préséance des tâches.
monoïde	Un objet mathématique doté d'une relation et d'un ensemble d'éléments fini ou infini. Par exemple, $(\mathcal{R}, +)$ représente un monoïde doté de l'ensemble des réels et l'opérateur addition définie sur $\mathcal{R}$.
Pleinement connecté	En rapport avec la connexité d'un réseau de processeurs. Chaque processeur est relié à tous les autres processeurs du réseau.
Regroupement de tâches	Un sous-ensemble de tâches avec relations de préséance. Ces tâches représentent la programmation parallèle d'un composant électrique trop coûteux pour une exécution séquentielle.
Répartition dynamique	Une répartition qui tient compte de l'évolution temporelle et spatiale des tâches et de l'ordinateur parallèle.
Répartition statique	Une répartition des tâches qui considère un environnement constant.

ANNEXE B

Langages formels, machines abstraites et complexité des problèmes

Le but de cette annexe est de présenter la formalisation de la notion intuitive de complexité. Pour cela nous partons d'une vision générale d'une machine abstraite. Une machine abstraite est, d'abord et avant tout, décrite par son état. Chaque instruction étant une modification de cet état. Le programme de la machine abstraite consiste en un ensemble de règles donnant, en fonction d'un état et d'une instruction, le nouvel état de la machine. On associe à la machine un langage, qui est l'ensemble de toutes les suites possibles d'instructions sur cette machine. Le raffinement de cette machine abstraite peut nous conduire à un modèle général de calcul. À partir de ce modèle, il sera alors possible de définir d'une manière précise ce qui est calculable et ce qui ne l'est pas et enfin, ce qui est difficile à calculer; c'est à dire, la complexité des problèmes.

Les résultats à propos des machines de Turing servent de barèmes de comparaison afin d'établir le classement général des problèmes selon leur complexité. Les classes de complexité **P** (polynomiale), **NP** (non déterministe polynomiale), **NP-complet** et **NP-difficile** sont formellement définies. Une méthodologie de classement des problèmes NP-complet est également donnée. Cette méthodologie sera utilisée dans l'annexe C pour démontrer l'appartenance du problème de répartition des tâches temps réel à la classe **NP-complet**.

B 1 Machine de Turing

Une machine de Turing M est composée: *i*) d'une mémoire appelée bande et est comparable à une bande magnétique. *ii*) d'une tête de lecture/écriture qui peut être caractérisée par un nombre fini d'états. La tête de lecture/écriture peut lire un symbole écrit sur une case de la bande, et réagir:

- en écrivant un autre symbole dans la case;
- en changeant d'état;
- en se déplaçant d'une case vers la droite ou vers la gauche.

Un ensemble fini de règles précise les réactions de la tête de lecture. Ces règles sont le *programme* de la machine. Initialement, seul un nombre fini de cases sont marquées avec un symbole autre que le symbole spécial B (blanc). La machine s'arrête quand une situation n'est pas prévue par les règles. L'état global de la machine est constitué de deux parties. Une partie finie qui est l'état de la machine. Une autre partie, comportant un nombre non borné de possibilités, donnée par la suite de symboles écrits sur la bande. Les règles ou le programme de la machine indique comment modifier l'état global de la machine.

Définition B1.0.1 Une machine de Turing consiste en un quintuplet $\mathtt{M} = (\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F})$ où

- $\mathcal{Q}$ est un ensemble fini d'états;
- Σ est un ensemble de symboles appelé alphabet de lecture et écriture comportant un symbole spécial B (blanc);
- $\delta : \mathcal{Q} \times \Sigma \rightarrow (\mathcal{Q} \times \Sigma \times \{\Leftarrow, \Rightarrow\})$ est une fonction appelée fonction de transition où $\Leftarrow$ indique un déplacement vers la gauche et $\Rightarrow$ un déplacement vers la droite;
- $q_0 \in \mathcal{Q}$ est l'état initial de la machine;
- $\mathcal{F} \subseteq \mathcal{Q}$ est l'ensemble des états finaux.

■

B 2 Machine de Turing non déterministe

Comme pour les automates on peut introduire le non déterminisme et définir des machines de Turing non déterministes. Dans un état donné et pour un symbole d'entrée, il existe, pour les machines non déterministes, un choix sur un nombre fini de d'actions. À noter que l'apport du non déterminisme aux machines de Turing n'ajoute aucune puissance supplémentaire dans la décidabilité d'un langage. Par contre, la machine de Turing non déterministe permet la modélisation de certains problèmes beaucoup plus facilement.

Définition B2.0.1 Une machine de Turing non déterministe consiste en un quintuplet $\mathtt{M} = (\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F})$ où

- $\mathcal{Q}$ est un ensemble fini d'états;
- Σ est un ensemble de symboles appelé alphabet de lecture et écriture comportant un symbole spécial B (blanc);
- $\delta : \mathcal{Q} \times \Sigma \to P(\mathcal{Q} \times \Sigma \times \{\Leftarrow, \Rightarrow\})$ est une fonction appelée fonction de transition;
- $q_0 \in \mathcal{Q}$ est l'état initial de la machine;
- $\mathcal{F} \subseteq \mathcal{Q}$ est l'ensemble des états finaux.

■

Nous restreignons aux machines de Turing vues comme accepteurs de langages, c'est à dire, l'on s'intéresse uniquement à l'état dans lequel se trouve la machine quand elle s'arrête. Un mot u est reconnu si l'on peut passer d'une configuration initiale $(q_0 \bullet u)$ à une configuration terminale dont l'état appartient à l'ensemble des états finaux $\mathcal{F}$.

B 3 Relations entre langages et fonctions

Une machine de Turing M peut être vue comme un calculateur. L'on considère alors la fonction partielle f_M qui fait correspondre à un mot u, le résultat du calcul de M sur u.

Définition B3.0.1 Soit M une machine de Turing et $\mathcal{F}$ un sous-ensemble des états finaux. Un mot u est **accepté ou reconnu** par M si M s'arrête pour le mot u dans un état final.

■

Définition B3.0.2 Le langage $\lrcorner_M$ accepté ou reconnu par la machine est l'ensemble des **mots reconnus** par M.

■

Définition B3.0.3 Un langage $\lrcorner$ est **récursivement énumérable** quand il existe une machine de Turing M qui accepte $\lrcorner$. Si, de plus, la machine s'arrête pour tous les mots possible appartenant à $\lrcorner$, alors $\lrcorner$ est récursif.

■

Donc, $\lrcorner$ est récursif si et seulement s'il existe une machine de Turing M qui accepte $\lrcorner$ et qui s'arrête sur tous les mots u dans Σ.

Définition B3.0.4 La fonction partielle f_M qui fait correspondre à un mot u le résultat du calcul de M sur u est la **fonction calculée** par la machine de Turing M. On dit qu'une fonction f est calculable quand il existe une machine de Turing M telle que $f = f_M$.

■

Un mot u n'est donc pas accepté par une machine M quand celle-ci ne s'arrête pas pour u ou encore quand celle-ci s'arrête, pour u dans un état non final. Un langage ⌟ est récursivement énumérable quand il existe une machine de Turing qui s'arrête dans un état final pour tous les mots de ⌟ Pour les mots qui n'appartiennent pas à ⌟, la machine peut donc soit arrêter dans un état non final, soit ne pas s'arrêter. La machine de Turing qui calcule f s'arrêtera pour tout mot u du domaine de la fonction et ne s'arrêtera pas pour les autres.

On dit que la machine M calcule la fonction f_M même si f_M n'est pas une fonction totale. De plus, en informatique, on définit normalement les fonctions calculables sur les entiers et non sur les mots. Cela ne pose aucun problème car il y a bijection entre l'ensemble des mots sur un alphabet fini et l'ensemble des entiers. En fait, il n'y a pas de différence fondamentale entre langage et fonction, il suffit de considérer la fonction caractéristique δ du langage ⌟ qui associe $\delta(u)$ à un mot u, avec $\delta(u) = 0$ si $u \in$ ⌟ et $\delta(u) = 1$ si $u \notin$ ⌟.

Par ailleurs, la thèse de Church [HOP79] dit que toute fonction calculable au sens intuitif du terme est calculable par une machine de Turing [HOP79]. Bien entendu, cette thèse est de nature philosophique et ne peut pas être démontrée. Mais elle est en fait vérifié dans tous les cas concrets. Pour toutes les machines qu'il a été possible de concevoir, on a montré qu'il était possible de construire, des machines de Turing calculant les mêmes fonctions [LIN97]. Cela veut dire que si l'on montre qu'une fonction n'est pas calculable par machine de Turing, alors il ne sera pas possible de construire une machine avec les principes connues actuellement permettant de la calculer. Le théorème de la calculabilité montre l'idée intuitive que tout n'est pas calculable est vérifiée mais d'abord nous présentons quelques résultats de la théorie des nombres qui viendra appuyer le théorème de la calculabilité.

Définition B3.0.5 Un ensemble est **dénombrable infini** si et seulement s'il possède la même cardinalité que l'ensemble Z^+. Un ensemble est dénombrable si et seulement s'il est fini ou dénombrable infini. En théorie des nombres, on dit qu'un ensemble infini A est dénombrable infini si et seulement si nous pouvons établir une fonction $f: Z^+ \rightarrow A$ qui est une fonction bijective (c'est à dire, à la fois injective et surjective).

■

Théorème B3.0.1 *[EPP95] Tout sous-ensemble d'un ensemble dénombrable est aussi dénombrable.*

■

Théorème B3.0.2 *Soit T l'ensemble de toutes les fonctions $f: Z^+ \rightarrow \mathcal{N}$. L'ensemble T n'est pas dénombrable.*

■

Démonstration Soit S, l'ensemble de tous les nombres réels entre 0 et 1. Tous les nombres de S peuvent être représentés dans la forme $0.a_1a_2 \ldots a_n \ldots$, où chaque a_i est un entier de 0 à 9. Cette représentation est unique. Définir une fonction F de S à un sous-ensemble de T telle que $F(0.a_1a_2 \ldots a_n \ldots)$ est une fonction qui fait correspondre chaque entier positif n à a_n. Définir le co-domaine de F égal à l'image de F pour que F soit une fonction surjective. La fonction F est également injective puisque $F(x_1) \neq F(x_2)$, $\forall x_1, x_2 \in \mathcal{N}$, à cause de l'unicité de la représentation utilisée. La fonction F est donc injective de S à un sous-ensemble de T. Mais nous savons d'après le théorème de Cantor que l'ensemble de tous les nombres réels entre 0 et 1 n'est pas dénombrable [EPP95]. Donc S n'est pas dénombrable. Nous avons pu établir une fonction F qui est à la fois une injection et une surjection d'un ensemble non dénombrable à un sous-ensemble de T. Par analogie avec le théorème B3.0.1, l'ensemble T est aussi non dénombrable.

□

Théorème B3.0.3 *[MOR98] L'ensemble de tous les programmes d'un langage donné est dénombrable.*

■

Théorème B3.0.4 *[MOR98] Il existe des fonctions non calculables.*

Nous savons maintenant qu'il existe des fonctions non calculables par machine de Turing. Mais cette distinction est difficile à postuler puisqu'il faut toujours construire une machine de Turing pour infirmer ou affirmer une fonction est calculable ou non.

B 4 Complexité des problèmes

Dans cette section, nous abordons le classement des problèmes classes de complexités. La question à poser ici est: quels sont les problèmes faciles pour lesquels il existe des algorithmes de solution efficaces et quels sont les problèmes difficiles pour lesquels il n'existe pas d'algorithmes efficaces?

Définition B4.0.1 Un **problème abstrait** est donné par un couple $\mathcal{P}=(C, P)$ où

- C est une classe d'objets;
- P est une propriété concernant les objets de C.

■

Un problème abstrait qui est formulé ainsi est appelé un problème de décision.

Définition B4.0.2 La **complexité d'un problème** est l'ordre de grandeur de la complexité dans le cas le pire du meilleur algorithme connu pour le résoudre.

■

Comme pour un algorithme on doit distinguer, pour un problème, sa complexité en temps et sa complexité en espace. Pour nos besoins, nous nous restreindrons à la complexité en temps.

Définition B4.0.3 On dit qu'un algorithme A (représenté par une machine de Turing) **accepte un mot** $u \in \{0, 1\}^*$ si pour l'entrée u l'algorithme produit $A(u) = 1$. Le langage $\lrcorner$ accepté par un algorithme A est l'ensemble des mots $\lrcorner = \{u \in \{0,1\}^* \mid A(u) = 1\}$. De même, un algorithme A rejet un mot u si $A(u) = 0$.

■

Définition B4.0.4 Un langage $\lrcorner$ est accepté par un algorithme A (représenté par une machine de Turing) en un temps polynomial si pour tous mots $u \in \lrcorner$ de cardinalité n, l'algorithme accepte u en un temps $O(n^k)$ ou k est une constante.

■

Noter que même si un langage est accepté par un algorithme, ce dernier ne va pas nécessairement rejeter un mot $u \notin \lrcorner$.

Définition B4.0.5 Un langage $\lrcorner$ est **décidé** par un algorithme A si tous ses mots sont soient acceptés soient rejetés par l'algorithme.

■

Définition B4.0.6 Un langage $\lrcorner$ est décidé par un algorithme A en temps polynomial si pour tous mots $u \in \{0, 1\}^*$ de longueur n, l'algorithme décide u en un temps $O(n^k)$ ou k est une constante

■

Ainsi, pour qu'un algorithme accepte un langage $\lrcorner$, il doit seulement tenir compte des mots de $\lrcorner$. Par contre, pour qu'un algorithme décide un langage, il doit accepter ou rejeter tous les mots dans $\{0, 1\}^*$.

Un algorithme déterministe résout le problème de décision $\mathcal{P}$ quand, avec comme donnée d'entrée un objet c de la classe C, il termine en donnant la réponse OUI si c possède la propriété P et NON dans le cas contraire. La notion de problème de décision peut être étendue à des problèmes d'optimisation. En effet, nous pouvons toujours ramener un problème d'optimisation en un problème de décision en ajoutant une borne (inférieure ou supérieure) aux variables à optimiser.

B 4.1 Complexité polynomiale P

Définition B4.1.1 La classe de complexité **P** est la classe des langages $\lrcorner$ qui peut être décidée par un algorithme en un temps polynomial. $\mathbf{P} = \{\lrcorner \subseteq \{0,1\}^* \mid$ il existe un algorithme A qui décide $\lrcorner$ en un temps polynomial$\}$.

■

En fait, **P** est également la classe des langages qui peut être acceptée en un temps polynomial.

Théorème B4.1.1 *[COR93]* $P = \{\lrcorner \mid \lrcorner$ *est accepté par un algorithme exécutant en temps polynomial}*

■

En d'autres mots, les problèmes de complexité **P** sont des problèmes résolubles par un algorithme (machine de Turing) en un temps polynomial.

B 4.2 Complexité non déterministe polynomiale NP

Pour pouvoir définir la classe de complexité **NP**, nous devons d'abord comprendre ce qu'est la vérification en temps polynomial. La vérification dans ce sens est l'utilisation des algorithmes qui vérifient l'appartenance des langages.

Définition B4.2.1 Un algorithme de vérification A est un algorithme comportant deux arguments d'entrée $A(u, y)$ où

- u est un mot d'entrée;
- y est un mot d'entrée appelé certificat.

■

L'algorithme A vérifie l'entrée u s'il existe un certificat y tel que $A(u, y) = 1$. Le langage vérifié par un algorithme de vérification A est $L = \{u \in \{0, 1\}^* \mid$ il existe $y \in \{0, 1\}^*$ tel que $A(u, y) = 1\}$. Intuitivement, un algorithme A qui vérifie un langage L si pour tout mot $u \in L$, il existe un certificat y que A peut utiliser pour prouver que $u \in L$. En pratique, le certificat y est un indice qui mène vers la solution d'un problème et l'algorithme A doit pouvoir vérifier que u est une solution en utilisant y. Dans certains problèmes, dont la structure est suffisamment complète, on peut ignorer le certificat y et vérifier directement si l'entrée u est une solution. Dans ce cas, l'algorithme A doit pouvoir vérifier que u est une solution d'un problème en un temps polynomial.

Définition B4.2.2 La classe de complexité **NP** est la classe des langages qui peut être vérifié par un algorithme exécutant en temps polynomial. Un langage $L \in$ **NP** si et seulement s'il existe un algorithme $A(u, y)$ exécutant en temps polynomial et une constante c tels que $L = \{u \in \{0, 1\}^* \mid$ il existe un certificat y avec $|y| = O(|u|^c)$ tel que $A(u, y) = 1\}$.

■

Donc, un problème de décision est de classe **NP** s'il est vérifiable par un algorithme exécutant en temps polynomial. Par la définition B4.1.1, si $\lrcorner \in \mathbf{P}$ alors $\lrcorner \in \mathbf{NP}$ puisqu'il existe un algorithme exécutant en temps polynomial pour décider $\lrcorner$. Forcément $\mathbf{P} \subseteq \mathbf{NP}$. Cependant, nous ne savons pas si $\mathbf{P} = \mathbf{NP}$. Intuitivement, la classe **P** comprend les problèmes qui peuvent être résolus rapidement. La classe **NP** comprend les problèmes pour lesquels une solution peut être vérifiée rapidement. De plus, il est souvent plus facile de vérifier une solution bien présentée que de trouver une solution à un problème. On peut appliquer cette analogie aux classes de complexité **P** et **NP** et dire qu'il existe des langages dans **NP** qui ne sont pas dans **P**.

Il existe une argumentation encore plus convaincante qui nous indique que $\mathbf{P} \neq \mathbf{NP}$. Il s'agit de la classe dite **NP-complet** que nous présenterons dans la section suivante.

B 4.3 Complexité non déterministe polynomiale NP-complet

La classe de complexité **NP-complet** possède une propriété très intéressante. Cette propriété dit que si un problème **NP-complet** est résoluble en un temps polynomial alors tous les problèmes dans **NP** sont aussi résoluble en un temps polynomial, c'est à dire, $\mathbf{P} = \mathbf{NP}$. Cependant, aucun algorithme n'a encore été trouvé pour résoudre un problème **NP-complet** en un temps polynomial.

Dans ce sens, les problèmes **NP-complet** sont les problèmes les plus ardus à résoudre dans **NP**. Pour pouvoir dire qu'un problème est **NP-complet**, nous devons d'abord comprendre ce qu'est une réductibilité polynomiale. Nous pouvons par la suite utiliser la réductibilité polynomiale pour prouver qu'il existe d'autres problèmes qui sont dans **NP-complet**. Notamment, nous pouvons montrer que le problème de répartition des tâches temps réel est dans la classe **NP-complet**.

Intuitivement, un problème $\mathcal{P}$ peut être réduit à un autre problème $\mathcal{P}'$ si toute instance de $\mathcal{P}$ peut être facilement remaniée en une instance de $\mathcal{P}'$ où la solution de $\mathcal{P}'$ est également une solution de $\mathcal{P}$. Par exemple, le problème d'équations linéaire de la forme $ax + b = 0$ peut être ramené à la forme quadratique $0x^2 + ax + b = 0$ donc la solution est également une solution de $ax + b = 0$. Donc si un problème $\mathcal{P}$ est réduit à un autre problème $\mathcal{P}'$ alors $\mathcal{P}$ n'est pas plus difficile à résoudre que $\mathcal{P}'$.

Définition B4.3.1 Un problème $\mathcal{P}= (C, P)$ est **réductible** à un autre problème $\mathcal{P}'= (C', P')$ si et seulement s'il existe une fonction $f : C \rightarrow C'$ calculable en temps polynomial, telle que, pour tout objet c de C, c vérifie la propriété P si et seulement si $f(c)$ vérifie P'. La notation $\mathcal{P} \leq_f \mathcal{P}'$ est utilisée pour désigner la réduction d'un problème $\mathcal{P}$ à un problème $\mathcal{P}'$ par le biais de la fonction f.

■

La signification intuitive de $\mathcal{P} \leq_f \mathcal{P}'$ est: résoudre $\mathcal{P}'$ permet de résoudre $\mathcal{P}$. En effet, pour résoudre $\mathcal{P}$, il suffit d'appliquer la fonction f et de résoudre $\mathcal{P}'$. On exige à la fonction f une complexité polynomiale pour éviter que le codage des données par f ne soit pas, trop pénalisant. Cette hypothèse implique que si l'algorithme pour résoudre $\mathcal{P}'$ est efficace, on obtiendra un algorithme efficace pour résoudre $\mathcal{P}$. On dit que la fonction f est une fonction de réduction et l'algorithme qui calcul f, l'algorithme de réduction.

Définition B4.3.2 Lorsque $\mathcal{P} \leq_f \mathcal{P}'$ et $\mathcal{P}' \leq_f \mathcal{P}$ alors $\mathcal{P}$ et $\mathcal{P}'$ sont des problèmes équivalents. De plus si $\mathcal{P} \in \mathbf{P}$ alors $\mathcal{P}' \in \mathbf{P}$ et vice versa.

■

Théorème B4.3.1 L'opération de réduction polynomiale $\leq_f$ est une opération transitive. Si $\mathcal{P} \leq_f \mathcal{P}'$ et $\mathcal{P}' \leq_f \mathcal{P}''$ alors $\mathcal{P} \leq_f \mathcal{P}''$.

Démonstration Selon la définition B4.3.1 la fonction de réduction f est surjective. La relation $\mathcal{P} \leq_f \mathcal{P}'$ signifie qu'il existe une fonction f telle que pour toutes entrées $u \in \mathcal{P} \rightarrow f(u) \in \mathcal{P}'$. Alors,

- si $\forall u \in \mathcal{P} \rightarrow f(u) \in \mathcal{P}'$;
- si $\forall u' \in \mathcal{P}' \rightarrow f'(u') \in \mathcal{P}''$;
- alors $\forall u \in \mathcal{P} \rightarrow f'(f(u)) \in \mathcal{P}''$.

En vertu des propriétés de la composition des fonctions, $f'(f(u))$ est également une fonction surjective. Ce qui démontre bien la transitivité de la relation $\leq_f$.

□

La réductibilité polynomiale permet donc de montrer qu'un problème est au moins aussi difficile qu'un autre problème à un facteur polynomial près. Nous pouvons dès maintenant définir la classe des problèmes **NP-complet.** Ce sont des problèmes les plus ardus à résoudre.

Définition B4.3.3 Un problème est **NP-complet** s'il satisfait les conditions suivantes:

1. $\mathcal{P} \in$ **NP**;
2. $\mathcal{P}' \leq_f \mathcal{P}, \forall \mathcal{P}' \in$ **NP**.

■

Définition B4.3.4 Un problème est **NP-difficile** s'il satisfait la condition (2) mais pas nécessairement la condition (1) de la définition B4.3.3.

■

Théorème B4.3.2 *[MOR98] Si un problème **NP-complet** est solutionné en un temps polynomial alors **P = NP**. Si un problème **NP-complet** n'est pas solutionné en un temps polynomial alors tous les problèmes **NP-complet** ne sont pas solutionnés en un temps polynomial.*

À la lumière des démonstrations présentées, nous pouvons arranger les classes de complexité selon la figure B4—1.

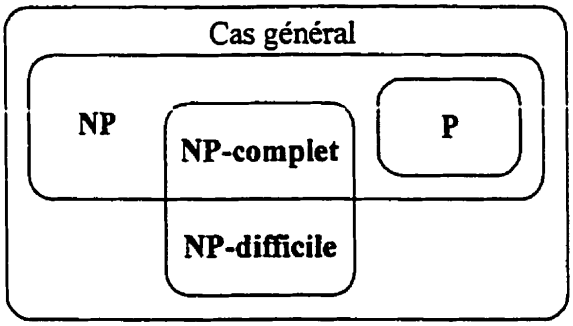


Figure B4—1 Schéma montrant l'appartenance des classes de complexité.

Dans ce schéma, la classe de complexité **P** $\subset$ **NP** et **P** $\neq$ **NP**. La classe **NP-complet** est également dans **NP** selon la définition B4.3.3. mais **NP-complet** $\not\subset$ **P**. Si un problème est réductible à tout problème **NP** mais qu'il n'est pas nécessairement dans la classe **NP** alors il est **NP-difficile**. Notons que tout problème **NP-complet** est également **NP-difficile**.

Nous avons inclus dans ce schéma le cas général qui englobe toutes les classes de complexité connues. La raison est qu'il peut exister des classes de problèmes qui ne sont pas encore découvertes.

B 4.4 Méthodologie de démonstration pour la classe NP-complet

Nous avons défini dans la section précédente qu'un problème $\mathcal{P}$ est de classe **NP-complet** s'il satisfait deux conditions (voir définition B4.3.3). La première condition consiste à pouvoir trouver un algorithme qui vérifie en un temps polynomial la solution de $\mathcal{P}$ ($\mathcal{P} \in$ **NP**). La seconde condition est de montrer que le problème $\mathcal{P}$ est réductible en un temps polynomial à tout problème appartenant à **NP** ($\mathcal{P}' \leq_f \mathcal{P}$, $\forall \mathcal{P}' \in$ **NP**). À première vue, cette dernière condition est très sévère et même impossible de faire en

pratique. Heureusement, le théorème suivant nous montre qu'il suffit de réduire un problème $\mathcal{P}$ à un autre problème $\mathcal{P}' \in$ **NP-complet** pour prouver que $\mathcal{P}$ est **NP-difficile**.

Théorème B4.4.1 *[COR93] Si $\mathcal{P}$ est un problème tel que $\mathcal{P}' \leq_f \mathcal{P}$ pour $\mathcal{P}' \in$ NP-Complet, alors $\mathcal{P}$ est NP-difficile. De plus, si $\mathcal{P} \in$ NP alors $\mathcal{P} \in$ NP-complet.*

■

En d'autres mots, en réduisant un problème connu NP-complet $\mathcal{P}'$ à $\mathcal{P}$, nous réduisons implicitement toutes les problèmes dans **NP** à $\mathcal{P}$. Ainsi une méthodologie peut être établie et qui indiquera comment montrer l'appartenance d'un problème à la classe NP-complet.

Pour montrer qu'un problème $\mathcal{P} \in$ **NP-complet**, il suffit de suivre les étapes suivantes [COR93].

1. montrer que $\mathcal{P} \in$ **NP**;
2. choisir un problème $\mathcal{P}'$ connu comme étant dans la classe **NP-complet**;
3. décrire un algorithme qui calcule une fonction de réduction f qui associe toutes les instances de $\mathcal{P}'$ à une instance de $\mathcal{P}$;
4. montrer que la fonction f satisfait $u \in \mathcal{P}'$ si et seulement si $f(u) \in \mathcal{P}$;
5. montrer que l'algorithme calculant f exécute en un temps polynomial.

Cette méthodologie rend les démonstrations beaucoup plus simples. Il n'est plus nécessaire de montrer que le problème à traiter est réductible à tous les problèmes qui sont dans la classe **NP**. Dans l'annexe C, nous utiliserons ces cinq étapes pour prouver que le problème de répartition des tâches temps réel est un problème de la classe **NP-complet**.

ANNEXE C

Complexité du problème de la répartition des tâches temps réel

Cette annexe donne les détails techniques utilisées dans la démonstration de la complexité **NP-complet** du problème de la répartition des tâches temps réel. Le résultat de la complexité du problème de répartition des tâches temps réel est important puisqu'il nous indique les stratégies possibles pour la résolution de ce problème.

Pour construire la démonstration de la complexité du problème, nous avons besoin d'un modèle de la répartition des tâches. Ce modèle doit être suffisamment simple pour que sa manipulation soit aisée. Cependant, le modèle doit également être suffisamment expressif pour bien représenter le fonctionnement de la répartition des tâches.

Les graphes disjonctifs (split graphs) [GOL80] ont été proposés comme des représentations efficaces pour la modélisation de la répartition des tâches dans les systèmes distribués [ALI93]. L'avantage des graphes disjonctifs dans ce contexte relève de deux points importants: *i*) la représentation est intuitive; *ii*) les manipulations possibles de ces graphes sont tirées de la théorie des graphes, un domaine bien connu. Dans cette annexe, nous appliquerons les graphes disjonctifs comme éléments de base pour la démonstration de la complexité du problème de la répartition des tâches temps réel.

Nous avons vu dans l'annexe B que le classement de la complexité des problèmes est intimement lié à la notion de calculabilité des machines de Turing. De plus, nous

savons qu'un problème $\mathcal{P}$ est **NP-complet** s'il existe un algorithme qui vérifie sa solution en un temps polynomial et s'il est possible d'effectuer une réduction polynomiale de tous les problèmes $\mathcal{P}'$ à $\mathcal{P}$ avec $\mathcal{P}' \in$ **NP** (définition B3.3.3). Des raccourcies existent pour montrer qu'un problème est bien dans la classe **NP-complet**. En effet, pour classer $\mathcal{P}$ dans **NP-complet**, il suffit de montrer qu'un problème $\mathcal{P}'$, reconnu comme étant **NP-complet**, soit réductible en un temps polynomial à $\mathcal{P}$ (théorème B3.4.1). En d'autres mots, nous induisons l'appartenance à la classe **NP-complet** d'un problème à partir d'un autre problème **NP-complet**.

Pour parvenir à ce but, nous utiliserons la modélisation de la répartition des tâches par graphes disjonctifs. Nous introduirons le problème de la séparation en cliques maximales d'un graphe (MAXCLIQUE) comme un problème intermédiaire. Nous montrerons que ce problème intermédiaire est réductible en un temps polynomial au problème général **NP-complet** de la séparation d'un graphe en sous-graphes (PARTITION) [GAR79], [EVA79]. Enfin, nous donnerons une preuve qui montre que le problème de la séparation en cliques maximales d'un graphe est analogue et réductible en un temps polynomial au problème de la répartition des tâches temps réel (RTTR).

C 1 Graphes disjonctifs

Un graphe non orienté est défini par un ensemble S de sommets et un ensemble A d'arêtes, $G = (S, A)$. Pour nos besoins nous ne considérons que les graphes simples. Un graphe simple est celui qui ne possède pas de boucles autour d'un seul sommet ni d'arêtes parallèles partageant les mêmes extrémités.

Définitions C1.0.1 Un graphe est **complet** quand l'ensemble de ses sommets contient exactement une arêtes par pair de sommets.

■

Définitions C1.0.2 Un ensemble de **sommets** $S' \subseteq S$ d'un graphe $G = (S, A)$ est **indépendant** si ses éléments sont mutuellement disjoints. C'est à dire, qu'il n'y a pas d'arêtes reliant les sommets de S'.

■

Définitions C1.0.3 Un ensemble de sommets $S' \subseteq S$ d'un graphe $G = (S, A)$ est complet si chaque pair de sommets dans S' est relié par une arête $a \in A$. Ces arêtes sont appelé **arêtes complètes.**

■

Définitions C1.0.4 Un **graphe** $G = (S, A)$ est **disjonctif** s'il existe une partition $S = S_1 \cup S_2$ de ses sommets en un ensemble indépendant S_1 et un ensemble complet S_2.

■

Remarquez que l'ensemble des sommets S_2 forme un sous-graphe complet qu'on appelle communément une clique de G [BER83]. Dans un graphe disjonctif, il n'y a pas de contraintes sur les arêtes qui existent entre les ensembles S_1 et S_2.

Définitions C1.0.5 Un graphe disjonctif est complet s'il existe une arête entre chaque sommet dans S_1 et chaque sommet dans S_2. On nomme **arête de liaison,** une arête reliant un sommet dans S_1 à un sommet dans S_2.

■

Dans un graphe disjonctif complet, il y a exactement $nn_2 - n_2(n_2 - 1)/2$ arêtes où $n = n_1 + n_2$ avec $n_1 = |S_1|$ et $n_2 = |S_2|$. La figure C1—1 montre un graphe disjonctif avec $|S_1| = 3$ et $|S_2| = 4$. Dans cette figure, il existe 12 arêtes de liaison (n_1n_2 arêtes de liaison) et 6 arêtes complètes ($n_2(n_2 - 1)/2$ arêtes complètes).

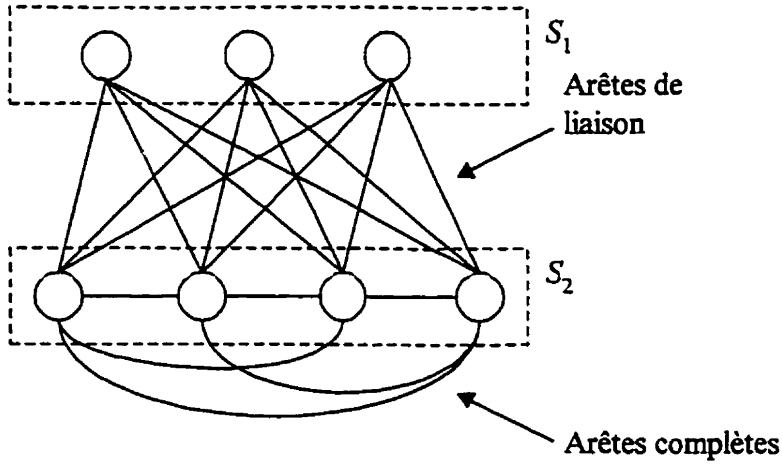


Figure C1—1 Représentation graphique d'un graphe disjonctif.

C 2 Modèle de la répartition des tâches temps réel

Le problème de la répartition des tâches temps réel dans un réseau de processeurs peut être modélisé à l'aide d'un graphe disjonctif. Dans cette modélisation, l'ensemble des processeurs est représenté par les sommets de l'ensemble indépendant S_1 du graphe. L'ensemble complet S_2 du graphe est utilisé pour représenter l'ensemble des tâches à répartir. La figure C2—1 est un représentation de cette organisation.

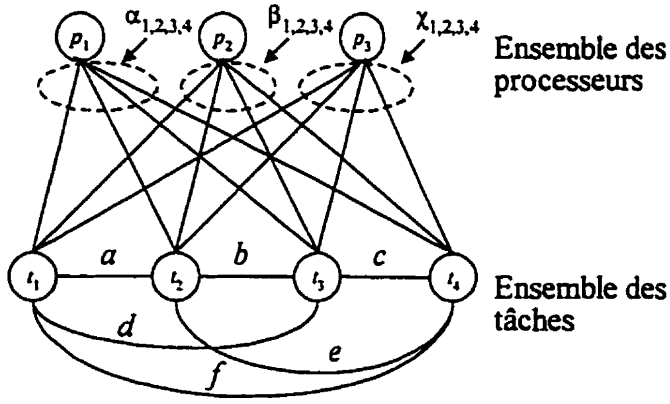


Figure C2—1 Graphe disjonctif représentant les processeurs et les tâches.

En effet, la figure C2—1 est une reprise de la figure C1—1 dans laquelle nous avons ajouté les étiquettes p_i, $1 \leq i \leq |S_1|$ pour les processeurs et les étiquettes t_i, $1 \leq i \leq |S_2|$ pour les tâches.

Définitions C2.0.1 Une **arête complète**, c'est à dire une arête reliant deux sommets dans S_2, représente la communication entre deux tâches. Le coût de communication est une valeur $c \in \mathcal{N}$ indiquée sur l'arête.

■

Définitions C2.0.2 Une **arête de liaison**, c'est à dire une arête reliant un sommet dans S_1 à un sommet dans S_2, représente l'exécution d'une tâche (dans S_2) dans un processeur (dans S_1). Le coût de l'exécution est une valeur $c \in \mathcal{N}^+$ indiquée sur l'arête.

■

Les étiquettes $\{a, b, ..., f\}$ des arêtes complètes de la figure C2—1 représentent les coûts de communication entre les tâches. Lorsqu'une valeur est nulle, cela signifie qu'il n'y a pas de communication entre les deux tâches qui sont reliées par l'arête. Les étiquettes $\{\alpha_1, ..., \alpha_4, \beta_1, ..., \beta_4, \chi_1, ..., \chi_4\}$ des arêtes de liaisons de la figure C2—1 représentent les coûts d'exécution des tâches sur les processeurs p_1 à p_3.

Définitions C2.0.3 Une **clique** d'une graphe $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ dont chaque pair de sommets de S' est relié par une arête $a \in A$.

■

La répartition des tâches dans ce contexte correspond à une séparation du graphe disjonctif en cliques disjoints. Chaque clique du graphe G consiste en un sommet $p \in S_1$ et un ensemble de sommets $S' \in S_2$. L'ensemble S' représente les tâches assignées au processeur représenté par le sommet p. Noter que nous nous intéressons surtout au

problème où le nombre de cliques disjoints est égal à $|S_1|$, c'est-à-dire le nombre de cliques disjoints est égal au nombre de processeurs. Ce cas étant particulièrement approprié pour le problème de répartition des tâches. La figure C2—2 montre une séparation possible du graphe disjonctif G en cliques σ_i, $1 \leq i \leq |S_1|$.

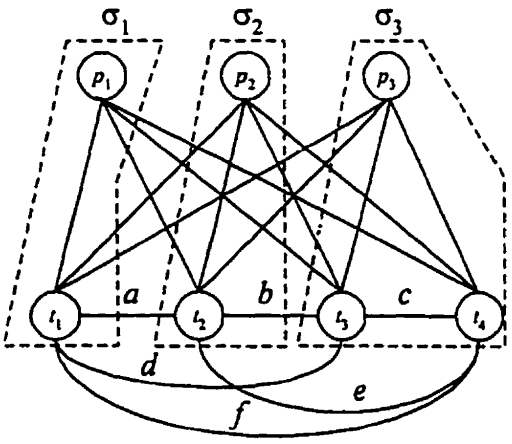


Figure C2—2 Séparation du graphe disjonctif en trois cliques.

Dans la figure C2—2 la tâche t_1 est assignée à au processeur p_1, la tâches t_2 est assignée au processeur p_2 et les tâches t_3, t_4 sont assignées au processeurs p_3. Cette assignation des tâches est fixe. À chaque cycle d'exécution, les mêmes tâches travailleront dans les mêmes processeurs. Remarquer que les arêtes de liaison d'une clique σ_i à une autre clique σ_j n'ont aucune signification puisqu'une tâches ne peut être assignée à deux processeurs en même temps.

C 2.1 Critères d'optimisation

Pour pouvoir porter un jugement sur les résultats obtenus de la séparation du graphe disjonctif en cliques disjoints, nous devons utiliser un critère d'optimisation. Ce critère sert à mesurer l'exactitude et l'efficacité des assignations tâches-processeurs obtenues. La fonction objectif envisagée doit être une fonction des valeurs des arêtes de

liaison et des arêtes complètes des cliques résultantes. Une des mesures la plus commune dans la répartition des tâches temps réel est la minimisation du coût total C_T du programme parallèle. Ce coût total comprend le coût d'exécution C_E et le coût de communication des tâches C_C. L'ensemble des coûts

$$C_T = C_E + C_C , \qquad \text{(C2.1.1)}$$

doit être inférieur à une valeur d qui représente l'échéancier imposé par le système. Ainsi, les cliques obtenues doivent satisfaire la relation

$$C_T \leq d , \qquad \text{(C2.1.2)}$$

pour qu'elles soient jugées acceptables. Remarquer que le coût de communication C_C est une fonction des données à transférer entre deux tâches assignées à des processeurs différents. Comme l'indique la définition C2.0.1, il peut exister dans le graphe disjonctif des arêtes complètes de valeur nulle. Donc, le sous-graphe complet généré par l'ensemble S_2 représente bien le système des tâches.

En utilisant le modèle du graphe disjonctif, le coût d'exécution des tâches assignées à un processeur est la somme des valeurs sur toutes les arêtes de liaisons appartenant à toutes les cliques. Le coût de communication est donné par la somme des valeurs sur toutes les arêtes complètes qui n'appartiennent pas à une clique. Supposons une assignation des tâches correspondant à une séparation d'un graphe disjonctif $G = (S_1 \cup S_2, A)$ en ensemble de cliques $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ où $m = |S_1|$. Chaque clique contient un sommet p_i correspondant à un processeur du système parallèle.

Définitions C2.1.1 Pour chaque assignation des tâches $S = \sigma_1 \cup \sigma_2 \cup \ldots \cup \sigma_m$, avec $p_i \in S_1$ un processeur dans la clique i, nous définissons [ALI93]:

- une arête de liaison (s, p_i) est appelée arête de liaison active si $s \in \sigma_i$. Elle est inactive autrement;
- une arête complète (s_i, s_j) est appelée arête complète active si s_i et s_j appartiennent à la même clique. Elle est inactive autrement.

Le coût d'exécution C_E est alors la somme des valeurs de toutes les arêtes de liaison actives. Le coût de communication C_C est alors la somme des valeurs de toutes les arêtes complètes inactive du graphe disjonctif G.

Dans le but de minimiser le coût total de (C2.1.1), les cliques seront obtenues en considérant les deux termes suivants: *i*) la maximisation des valeurs des arêtes complètes actives $c(s_i, s_j)$, $\forall i \neq j$; *ii*) la minimisation des valeurs des arêtes de liaisons actives $c(s, p_i)$. La maximisation des valeurs des arêtes complètes actives est équivalente à la minimisation du coût de communication puisque l'on considère que deux tâches communiquantes assignées à un même processeur ont un coût de communication nul. La minimisation des arêtes de liaison actives revient à diminuer le coût d'exécution des tâches d'une clique. Pour faire concorder les deux termes d'optimisation, nous appliquons une simple transformation sur le second terme de la fonction objectif. Au lieu de minimiser les valeurs des arêtes de liaison actives $c(s, p_i)$, nous allons plutôt maximiser plutôt $\beta - c(s, p_i)$ où β est une constante positive de valeur très grande. La fonction objectif f est alors

$$\max\{f\} = \max\{c(s_i, s_j) + [\beta - c(s, p_i)]\} \qquad \text{(C2.1.3)}$$

où $c(s_i, s_j)$, $i \neq j$ représente le coût de communication à économiser et $c(s, p_i)$ est le coût d'exécution des tâches s sur le processeur p_i. Ainsi, la maximisation de (C2.1.3) est mathématiquement équivalente à la recherche des cliques avec valeurs maximales sur ses arêtes.

C 3 Démonstration de la nature NP-complet du problème

Nous donnons ici une démonstration formelle montrant que le problème de la répartition des tâches temps réel appartient à la classe **NP-complet**. Pour ce faire, nous utiliserons comme objet de base les graphes disjonctifs et la méthodologie de démonstration présentée dans l'annexe B (section B3.4). Le formalise est celui pratiqué

dans le domaine de la théorie de décision. Ainsi, selon (C2.1.2), une instance du problème de la répartition des tâches temps réel peut être mise sous la forme d'un problème de décision par un 4-uplet et une question.

RTTR = {<T, P, W, d> : $T = \{t_1, t_2, \ldots, t_n\}$ ensemble des tâches, $P = \{p_1, p_2, \ldots, p_m\}$ ensemble des processeurs, $W = [w_{ij}]$ matrice de communication et $d \geq 0$ un entier. Existe-t-il une répartition des tâches T dans les processeurs P de sorte que $C_E + C_C \leq d$ où $C_E = \sum x_{ij}$ est le coût d'exécution de la tâche t_i dans le processeur p_j et $C_C = \sum w_{ij}$ est le coût de communication entre les tâches t_i et t_j si elles sont assignées à des processeurs différents}. (C3.1)

Nous utiliserons le nom RTTR pour désigner la version décisionnelle du problème de répartition. L'entier d est l'échéancier des tâches. Le fait que d soit un entier ne rend pas le problème moins intéressant puisque nous pouvons toujours exprimer les coûts en terme d'unités de temps convenable.

Selon la méthodologie de démonstration préconisée (section B3.4), nous avons besoin d'un problème connu appartenant à la classe **NP-complet**. Le problème général de la séparation d'un graphe en sous-graphes est justement connu comme étant **NP-complet** dans la littérature [GAR79]. L'énoncé de ce problème de séparation d'un graphe sous la forme d'un problème de décision est comme suit:

PARTITION = {<G', c', l', K, J> : $G' = (V', E')$ un graphe simple, c': $V' \rightarrow 1$ une application indiquant le poids des sommets de V', $l' : E' \rightarrow \mathcal{N}$ une application qui donne la valeur des arêtes, K et J entiers positifs. Existe-t-elle une séparation de V' en m ensembles disjoints $V_1', V_2', \ldots, V_m'$ telle que $\sum_{v \in V_i'} c'(v) \leq K$ pour $1 \leq i \leq m$. De plus, si E'' $\subseteq E'$ est l'ensemble des arêtes dont les deux extrémités sont dans deux ensembles V_i' différents alors $\sum_{e \in E''} l'(e) \leq J$ }. (C3.2)

D'après la définition du problème de décision (C3.2), le problème général de séparation d'un graphe en m sous-graphes implique que chaque sous-graphe contient au plus K sommets puisque $c'(v) = 1 \ \forall \ v \in V'$. Si les sommets de deux sous-graphes V_i' et V_j' avec $i \neq j$ sont reliés par des arêtes alors la somme des valeurs de ces arêtes ne doit pas dépasser J. Ce sont les contraintes imposées à ce problème de séparation des sommets.

Pour montrer que RTTR $\in$ **NP-complet**, nous proposons l'utilisation d'un problème de décision intermédiaire pour effectuer la réduction polynomiale à partir de PARTITION. Ce problème intermédiaire est un problème concernant la séparation des cliques de valeurs maximales dans des graphes disjonctifs. Nous le désignons par un triplet et une question.

MAXCLIQUE = $\{<G, l, B> : G = (V_1 \cup V_2, E)$ un graphe disjonctif complet avec V_1 un ensemble de sommets indépendants et V_2 un ensemble de sommets complets, $l : E \rightarrow \mathcal{N}$ une application qui donne la valeur des arêtes, $B \geq 0$ un entier. Existe-t-elle une séparation de V en m cliques disjointes σ_i de sorte que $\sum_{e \in \sigma_i} l(e) \geq B$, pour $1 \leq i \leq m$ où $e \in E$ et $m = |V_1|\}$.

Le problème MAXCLIQUE ressemble beaucoup à notre problématique de départ RTTR. L'idée est de trouver l'appartenance de RTTR par l'entremise de MAXCLIQUE grâce à la propriété transitive de la réduction polynomiale (théorème B3.3.1). En effet, pour établir que RTTR $\in$ **NP-complet**, il suffit de montrer que

PARTITION $\leq_f$ MAXCLIQUE (C3.4a)

et

MAXCLIQUE $\leq_f$ RTTR. (C3.4b)

Les sous-sections suivantes sont une série de théorèmes et de démonstrations qui visent justement à montrer qu'il est possible d'effectuer les réductions polynomiales de (C3.4a) et de (C3.4b).

C 3.1 Problème MAXCLIQUE

Nous utilisons la méthodologie de démonstration présentée dans la section B3.4 de l'annexe B. D'abord montrons que le problème MAXCLIQUE est dans **NP**. Rappelons qu'un problème est dans **NP** s'il existe un algorithme de vérification $A(u, y)$ où u est l'entrée et y est un certificat tel que $A(u, y) = 1$ pour tout u une solution du problème. De plus, cet algorithme doit pouvoir donner une réponse en un temps polynomial. Dans notre cas, la structure de MAXCLIQUE est suffisamment complète qu'il n'est pas nécessaire d'utiliser un certificat pour vérifier ce problème.

Théorème C3.1.1 *Le problème MAXCLIQUE ∈ NP.*

Démonstration La solution de ce problème est un ensemble de cliques disjointes σ_1, σ_2, ..., σ_m. Pour vérifier que $\sum_{e \in \sigma_i} l(e) \geq B$ pour $1 \leq i \leq m$, il suffit de placer les arêtes des cliques obtenues dans m listes. Pour chaque liste nous calculons la somme des valeurs des arêtes et la comparer à la valeur de B. Supposons que le graphe est complet avec n sommets. Il y aura $n(n - 1)/2$ arêtes dans le graphe. En considérant l'opération de sommation comme une opération de base nous avons $n^2 - n$ sommations. La formation des m listes de sommets est une fonction linéaire du nombre de sommets dans le graphe. Donc, on peut vérifier la solution de MAXCLIQUE en un temps polynomial $O(n^2)$.

□

Théorème C3.1.2 *Le problème PARTITION est réductible en une instance de MAXCLIQUE dans un temps polynomial.*

Démonstration Nous donnons une procédure de réduction polynomiale qui permet de réduire le problème PARTITION en MAXCLIQUE. Autrement dit, nous construisons un algorithme qui calcule une fonction de réduction f qui satisfait PARTITION $\leq_f$ MAXCLIQUE. Incidemment, nous savons que le problème PARTITION $\in$ **NP-complet** pour $K \geq 3$ et ce, même si l'application $l'(e) = 1\ \forall e \in E'$ [GAR79]. L'idée de base de cette construction consiste à ajouter un certain nombre d'arêtes de valeur nulle dans le graphe G' du problème PARTITION. Ces arêtes de valeur nulle sont nécessaires pour rendre G' complet et conforme à la définition du problème MAXCLIQUE. Nous ajoutons également un certain nombre de sommets supplémentaires dans G' pour former un nouveau sous-ensemble de sommets. Ces nouveaux sommets seront reliés au graphe G' par des arêtes qui posséderont des valeurs particulières. Les arêtes de ces nouveaux sommets ajoutés dans G' peuvent être interprétées comme des arêtes de liaison du problème MAXCLIQUE. Nous allons assigner des valeurs à ces arêtes de liaison nouvellement ajoutées de telle sorte qu'en prenant K sommets ou moins dans l'ensemble des sommets V' du problème PARTITION, il existe au moins K arêtes de liaison avec la valeur 1. La figure C3—1 montre graphiquement cette procédure.

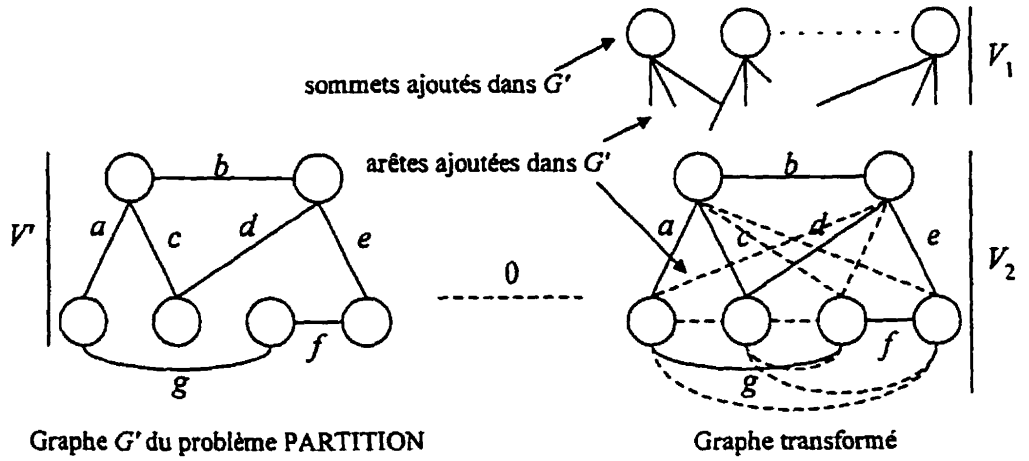


Figure C3—1 Transformation du graphe G' du problème PARTITION.

L'ajout des sommets et des arêtes, montré dans la figure C3—1, transforme le graphe G' en un graphe disjonctif. Évidemment, une solution du problème PARTITION ne doit pas contenir les sommets et les arêtes ajoutés enfin de respecter la structure originale. La description formelle de la construction de l'algorithme de réduction est donnée comme suit:

Étapes de construction

A. Le problème MAXCLIQUE nécessite la séparation des sommets en deux ensembles V_1 et V_2. V_1 est l'ensemble des sommets indépendant et V_2 est l'ensemble des sommets complets. Donner à V_1 un nouvel ensemble de sommets (voir la définition du problème MAXCLIQUE C3.3 et la figure C3—1) dont le nombre est $|V_1| = \binom{m}{K}$. Ce nombre de sommets est pour garantir que l'on peut toujours trouver au moins K arêtes de liaison dans le graphe pour tout $U \subseteq V_2, |U| \leq K$.

B. Assigner $V_2 = V'$. En effet, l'ensemble des sommets indépendants est l'ensemble des sommets V' du problème PARTITION.

C. Assigner à E l'ensemble des arêtes de MAXCLIQUE selon la règle

$$E = E_1 \cup E_2 \cup E_3 \qquad \text{(C3.5)}$$

où $E_1 = E'$, $E_2 = \{(u, v) \mid u, v \in V_2 \text{ et } (u, v) \notin E_1\}$ et $E_3 = \{(u, v) \mid u \in V_1 \text{ et } v \in V_2\}$. Autrement dit, nous ajoutons des arêtes dans le problème PARTITION pour le transformer en un graphe disjonctif complet. Les arêtes dans E_1 sont exactement les arêtes de PARTITION. Les arêtes dans E_2 sont des nouvelles arêtes ajoutées dans V' du problème PARTITION pour rendre l'ensemble V' complet. Ces nouvelles arêtes complètes ont une valeur nulle pour ne pas changer la structure de V'. Les arêtes dans E_3 sont des nouvelles arêtes ajoutées pour relier les sommets dans V_1 et les

sommets dans V_2. Remarquer que E_2 et E_3 sont respectivement les arêtes complètes et les arêtes de liaisons d'un graphe disjonctif ainsi formé.

D. Assigner à l'application $l(\cdot)$ du problème MAXCLIQUE les valeurs suivantes:

$$l(e) = l'(e), \forall\, e \in E_1, \tag{C3.6a}$$

$$l(e) = 0, \forall\, e \in E_2, \tag{C3.6b}$$

$$l(e) = a, a \in \{1, \alpha\}, \forall\, e \in E_3, \tag{C3.6c}$$

où $\alpha \leq -\left(\sum_{e \in E_1} l(e) + m\right)$ un petit entier négatif.

Autrement dit, si l'arête e est dans l'ensemble des arêtes de PARTITION alors utiliser les mêmes valeurs que PARTITION. Pour les arêtes complètes nouvellement ajoutées, on leur assigne une valeur nulle pour rendre le graphe G' du problème PARTITION complet. Pour les arêtes de liaison nouvellement ajoutées, on leur attribue une valeur de 1 ou une petite valeur négative. Nous donnons un arrangement spécial aux valeurs de (C3.6c). En effet, pour $l(e) = l((u, v_i))$ avec $u \in V_1$ et $v_i \in V_2$, $1 \leq i \leq m$, l'arête e est une arête de liaison reliant V_1 et V_2. Comme il y a une combinaison de $\binom{m}{K}$ sommets dans V_1, on aura $|V_1|$ valeurs données par (C3.6c). Arranger ces valeurs dans une matrice carrée $m \times m$ de la manière suivante:

$$M = \begin{bmatrix} 1 & 1 & \dots & 1 & \alpha & \alpha & \dots & \alpha \\ 1 & \dots & 1 & \alpha & \alpha & \alpha & \dots & \alpha \\ 1 & \dots & \alpha & \alpha & \alpha & \alpha & \dots & \alpha \\ \vdots & \dots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \alpha & \alpha & \dots & \alpha & 1 & 1 & \dots & 1 \end{bmatrix}. \tag{C3.7}$$

Le but est de produire une combinaison de valeurs dans la matrice M avec exactement K valeurs 1 et m - K valeur α de sorte que pour n'importe quel sous-ensemble de sommets complets $V \subseteq V_2$, il existe un sommet u dans l'ensemble

indépendant V_1 tel que $l((u, v)) = 1$, $\forall\ v \in V$, $|V| \leq K$. En d'autres mots, les arêtes de liaison sont soient de valeur 1 ou de valeur α et que en tout temps, au moins K arêtes de liaison ont une valeur 1 pour n'importe quel sommet dans l'ensemble indépendant V_1.

E. Assigner à B la constante positive de MAXCLIQUE la valeur de

$$B = \sum_{e \in E'} l'(e) + m - J. \tag{C3.8}$$

Maintenant, nous allons montrer que l'algorithme de réduction produit une solution équivalente pour les deux problèmes. Si nous pouvons répondre par OUI à la question posée par le problème MAXCLIQUE cela signifie qu'il est possible d'obtenir m cliques $\sigma_1, \sigma_2, \ldots, \sigma_m$ de sorte que $\sum_{e \in \sigma_i} l(e) \geq B$, pour $1 \leq i \leq m$. Puisque B est un entier positive et que α est un petit entier négatif, aucune clique ne doit contenir des arêtes de liaison e dont la valeur $l(e) = \alpha$. Ainsi, nous devons avoir m arêtes de liaison avec $l(e) = 1$ dans les cliques et qui sont dans E_3. Alors, on peut écrire que

$$\sum_{e \in \sigma_i \wedge e \in E_3} l(e) \geq B - m. \tag{C3.9}$$

Noter que chaque sous-graphe obtenu du problème de séparation PARTITION est une clique du problème MAXCLIQUE mais avec un sommet en moins puisque l'ensemble des sommets dans V_1 a été ajouté lors de la transformation de G'. De même, chaque sous-graphe obtenu de PARTITION ne contiendra pas les arêtes ajoutés lors de la transformation. C'est-à-dire, les arêtes dans E_2 et E_3 ne sont pas dans les sous-graphes de PARTITION. Soit $\sigma_i = (V(\sigma_i), E(\sigma_i))$ la i^e clique obtenue pour le problème MAXCLIQUE. L'ensemble des sommets des sous-graphes V_i' du problème PARTITION est alors $V_i' = V(\sigma_i) - \{v\}$ où $v \in V_1$. Il en est de même pour l'ensemble des arêtes des sous-graphes de PARTITION, c'est à dire, $E_i' = E(\sigma_i) - (E_2 \cup E_3)$. Donc, les

sous-graphes obtenus de PARTITION est $G_i' = (V_i', E_i')$. Ainsi, nous pouvons réécrire (C3.9) en

$$\sum_{e \in G_i' \wedge e \in E''} l'(e) \geq B - m . \qquad \text{(C3.10)}$$

La somme des valeurs des arêtes qui ne sont pas dans les sous-graphes est égale à

$$\sum_{e \in G_i'} l'(e) = \sum_{e \in E''} l'(e) - \sum_{e \in G_i'} l'(e) . \qquad \text{(C3.11)}$$

On peut remplacer (C3.10) dans (C3.11) pour donner

$$\sum_{e \in G_i'} l'(e) \leq \sum_{e \in E''} l'(e) - (B - m). \qquad \text{(C3.12)}$$

À l'aide de l'équation (C3.8) de l'étape (E) de l'algorithme de réduction, nous avons

$$\sum_{e \in G_i'} l'(e) \leq \sum_{e \in E''} l'(e) - (\sum_{e \in E''} l'(e) + m - J - m) , \qquad \text{(C3.13)}$$

ce qui implique que

$$\sum_{e \in G_i'} l'(e) \leq J . \qquad \text{(C3.14)}$$

Puisque $e \notin G_i'$ sont les arêtes qui ne sont pas inclues dans les sous-graphes de PARTITION. L'équation (C3.14) est équivalente à la formulation originale du problème

$$\sum_{e \in E''} l'(e) \leq J , \qquad \text{(C3.15)}$$

où E'' est l'ensemble des arêtes dont les deux extrémités sont dans deux ensembles V_i' différents (voir l'énoncé du problème C3.2). Nous avons démontrer la deuxième partie du problème PARTITION.

Pour la première partie du problème, nous avons vu qu'aucune clique de MAXCLIQUE ne contiendra une arête de liaison e de sorte que $l(e) = \alpha$ puisque $\alpha < 0$. Donc, seulement les arêtes de liaison avec $l(e) = 1$ seront inclues dans les cliques. Par l'arrangement de l'étape (D) de l'algorithme de réduction, nous avons exactement K arêtes de liaison pour chaque sommet dans V_1. Ainsi, chaque clique de MAXCLIQUE contiendra un sommet dans l'ensemble V_1 et au plus K sommets dans l'ensemble V_2.

Puisque chaque sommet des sous-graphes du problème PARTITON a une valeur de 1, K sommets au plus dans un sous-graphe vaut $\sum_{v \in V'_i} c'(v) \leq K$. Ce qui démontre la première partie du problème PARTITION.

La procédure de réduction transforme le problème de décision PARTITION en une instance du problème MAXCLIQUE. Les étapes de transformation (A) à (E) peuvent être réalisées en un temps polynomial. Comme le problème PARTITION $\in$ **NP-complet** et que MAXCLIQUE $\in$ **NP** selon le théorème B3.1.1 il en résulte que MAXCLIQUE est dans la classe **NP-complet.**

□

C 3.2 Problème RTTR

En utilisant les mêmes démarches que pour le problème MAXCLIQUE, d'abord montrons que le problème RTTR est dans **NP**. Autrement dit, le problème de répartition des tâches temps réel est un problème de la classe **NP**. Il est à noter que l'appartenance de RTTR à **NP** est intuitive puisque RTTR est équivalente à MAXCLIQUE. Par la suite nous donnons un algorithme de réduction polynomiale pour transformer le problème MAXCLIQUE en une instance de RTTR.

Théorème C3.2.1 *Le problème RTTR $\in$ NP.*

Démonstration La solution de ce problème est un ensemble d'assignations de tâches à des processeurs A_1, A_2, ..., A_m. Chaque assignation A_i contient un seul processeur de l'ensemble des processeurs et une tâche ne peut être associée qu'à un seul processeur. Pour vérifier que chaque assignation A_i satisfait $C_E + C_C \leq d$, il suffit de calculer $\sum x_{ij}^k + \sum w_{ij}^k$ pour $1 \leq k \leq m$ où m est le nombre d'assignations et de vérifier que la somme est inférieure ou égale à d, d étant l'échéancier des tâches. x_{ij}^k est le coût d'exécution de la

tâche i dans le processeur j pour la k^e assignation. w_{ij}^k est le coût de communication pour les tâches t_i et t_j assignées dans des processeurs différents pour la k^e assignation. Pour un système avec n tâches et m processeurs, on effectuera n sommations pour le calcul du coût d'exécution C_E et au plus $n(m - 1)$ sommations pour le calcul du coût de communication C_C en supposant que $m \leq n$. La vérification $C_E + C_C \leq d$, demande au plus nm additions. Donc, on peut vérifier la solution de RTTR en un temps polynomial $\theta(nm)$.

□

Théorème C3.2.2 *Le problème MAXCLIQUE est réductible en une instance de RTTR dans un temps polynomial.*

Démonstration Nous allons donner un algorithme de réduction polynomiale qui permet de réduire le problème MAXCLIQUE en une instance de RTTR. Autrement dit, nous construisons un algorithme qui calcule une fonction de réduction f qui satisfait MAXCLIQUE $\leq_f$ RTTR. L'idée de base de cette construction consiste à associer l'ensemble des tâches T et l'ensemble des processeurs P de RTTR à l'ensemble complet V_2 et à l'ensemble indépendant V_1 de MAXCLIQUE. On confère à w_{ij}, le coût de communication de RTTR, l'application $l : E \rightarrow \mathcal{N}$ de MAXCLIQUE où $e \in E = (v_i, v_j)$ et $v_i, v_j \in V_2$. Autrement dit, le coût de communication des tâches est représenté par les arêtes complètes du graphe disjonctif. Enfin, le coût d'exécution des tâches x_{ij} est associé à la fonction $g(e') = \beta - l(e')$ où $e' = (v_i, v_j)$ avec $v_i \in V_1$ et $v_j \in V_2$. La constante β est un entier positif de valeur très grande. Ainsi, le coût d'exécution d'une tâche t_i sur un processeur p_j, $x_{ij} = g(e')$, est en fait la valeur donnée les arêtes de liaison du graphe disjonctif à une constante près. Conséquemment, nous avons transformé le problème MAXCLIQUE en incluant la fonction objectif de C2.1.3. La figure C3—2 montre cette transformation.

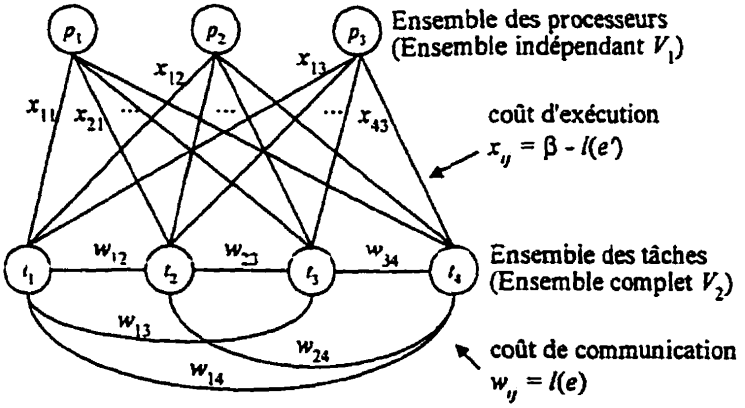


Figure C3—2 Transformation du problème MAXCLIQUE en une instance de RTTR.

Étapes de construction

A. Assigner l'ensemble complet V_2 du graphe disjonctif de MAXCLIQUE, aux tâches T du problème RTTR. $T = V_2$.

B. Assigner l'ensemble indépendant V_1 du graphe disjonctif de MAXCLIQUE aux processeurs P du problème RTTR. $P = V_1$.

C. Assigner l'application $l(e)$, $e = (v_i, v_j)$ avec $v_i, v_j \in V_2$ du graphe disjonctif de MAXCLIQUE aux coûts de communication w_{ij} du problème RTTR. $w_{ij} = l(e)$.

D. Assigner le deuxième terme de la fonction d'objectif (C2.1.3) aux coûts d'exécution x_{ij} du problème RTTR. $x_{ij} = \beta - l(e')$ où $e' = (v_i, v_j)$ avec $v_i \in V_1$ et $v_j \in V_2$.

E. Assigner à d, l'échéancier des tâches du problème RTTR la valeur de

$$d = -B + n\beta + C_C, \qquad \text{(C3.2.1)}$$

où B est un entier positif du problème MAXCLIQUE, n est le nombre de tâches, C_C est le coût de communication donné par $C_C = \sum_{i,j} w_{ij}$ pour $1 \le i,j \le n$.

Maintenant, nous allons montrer que l'algorithme de réduction produit une solution équivalente pour les deux problème. Si nous pouvons répondre par OUI à la question posée au problème RTTR cela signifie qu'il existe une répartition des tâches T dans les processeurs P de sorte que $C_E + C_C \leq d$. Cette répartition des tâches donne une séparation en cliques σ_i pour le problème MAXCLIQUE avec $\sigma_i = \{p_j\} \cup \{t_i \mid t_i$ sont les tâches assignées à $p_j\}$.

Soit la question posée par le problème MAXCLIQUE, $\sum_{e \in \sigma_i} l(e) \geq B$, pour $1 \leq i \leq m$ où e est une arête dans une clique et m est le nombre de cliques. Nous allons montrer que le problème RTTR donne le même résultat que MAXCLIQUE après transformation. Le coût total de la répartition des tâches est $C_E + C_C \leq d$ où d est l'échéancier des tâches. Donc, on peut écrire que

$$\sum_{i,j} x_{ij} + \sum_{i,j} w_{ij} \leq d . \qquad \text{(C3.2.2)}$$

Le terme $\sum_{i,j} x_{ij}$ calcule le coût d'exécution de la tâche t_i dans le processeur p_j et le terme $\sum_{i,j} w_{ij}$ calcule le coût de communication des t_i et t_j lorsqu'elles sont assignées à des processeurs différents. Par l'algorithme de réduction l'inéquation (C3.2.2) devient

$$\sum_{n} (\beta - l(e')) + \sum_{n} l(e) \leq d , \qquad \text{(C3.2.3)}$$

$$n\beta - \sum_{n} l(e') + C_C - \sum_{n} l(e'') \leq d , \qquad \text{(C3.2.4)}$$

où e sont les arêtes complètes dans des cliques différentes, e' sont les arêtes de liaison d'une même clique, e'' sont les arêtes complètes d'une même clique, C_C est le coût de communication du système et n est le nombre de tâches dans le système.

On peut interpréter les différents ensembles d'arêtes de la manière suivantes: *i*) les arêtes e représentent le coût de communication inter-processeur; *ii*) les arêtes e' représentent le coût d'exécution des tâches; *iii*) les arêtes e'' représentent le coût de

communication inter-tâche à l'intérieur d'un même processeur. En remaniant l'inégalité (C3.2.4) et en définissant $\tilde{e}$ comme l'ensemble des arêtes d'une clique, $\tilde{e} = \{(u,v) \mid u,v \in \sigma_i\}$ représentant le coût total d'exécution et de communication à l'intérieur d'une clique, nous obtenons

$$-\sum_n l(e') - \sum_n l(e'') \leq d - C_C - n\beta, \qquad \text{(C3.2.5)}$$

$$\sum_n l(\tilde{e}) \geq -d + C_C - n\beta. \qquad \text{(C3.2.6)}$$

Puisque la valeur de d a été transformée à l'étape (E) de l'algorithme en $d = -B + n\beta + C_C$, nous aurons enfin

$$\sum_n l(\tilde{e}) \geq -d + C_C - n\beta \Leftrightarrow \sum_n l(\tilde{e}) \geq B. \qquad \text{(C3.2.6)}$$

Ce qui démontre bien l'équivalence de MAXCLIQUE et de RTTR. L'algorithme de réduction transforme le problème de décision MAXCLIQUE en une instance du problème RTTR. Les étapes de transformation (A) à (E) peuvent être réalisées en un temps polynomial. Comme le problème MAXCLIQUE $\in$ **NP-complet** selon le théorème C3.1.2 et que RTTR $\in$ **NP** selon le théorème C3.2.1 il en résulte que RTTR est dans la classe **NP-complet.**

□

ANNEXE D

Notions pour la formalisation des techniques de fouille heuristique

Nous présentons les concepts et notions nécessaires pour la formalisation des techniques de fouille heuristique. Nous définissons ce que sont les graphes d'états, les fonctions heuristiques, la notion de longueur d'un chemin dans un graphe et les propriétés des termes heuristiques. Ces notions indispensables sont utilisées dans le chapitre 4 lors de la présentation d'une nouvelle méthode de fouille pour la répartition des tâches temps réel.

D1 Graphes d'états

Dans les techniques de fouille, un problème est représenté par des états et des opérateurs. Les états sont des structures de données caractérisant diverses configurations possibles du problème. Les opérateurs, quant à eux, sont des procédures susceptibles de transformer ces structures de données en d'autres structures de données. L'application successive de ces opérateurs, espère-t-on, peut finir par nous amener à la résolution du problème posé. L'ensemble des états d'un problème peut être représenté par un graphe orienté. Les définitions qui suivent montre les caractéristiques globales des graphes d'états.

Définitions D1.0.1 Un **graphe d'états** est un 1-graphe simple et orienté. C'est-à-dire, un graphe G tel que $\forall n_1, n_2 \in G$, $\exists$ au plus 1 arc de n_1 vers n_2. noté (n_1, n_2). Un graphe d'états est représentatif de l'ensemble des états possibles d'un problème (espace

d'états) et il contient un ensemble de liens parenté entre états. Puisqu'il s'agit d'un graphe orienté, une relation d'ordre partiel existe parmi l'ensemble des états du graphe.

■

Ainsi, chaque sommet du graphe représente un état et chaque arc (p, d) est une exécution d'une opération qui change l'état représenté par p en l'état représenté par d. Chaque sommet n est soit l'état initial du problème soit l'admission de l'état initial du problème comme ancêtre.

Définitions D1.0.2 Les graphes d'états possèdent des **arcs pondérés**. Donc, pour $G = (S, E)$ et un ensemble $V \subset \mathcal{R}$, il existe une application $c : E \rightarrow V$ qui associe l'ensemble des arcs E du graphe d'états aux valeurs de V. $c(a)$ avec $a \in E$ est le coût de a. Si n_1 et n_2 sont les extrémités initiale et finale de a, alors on note le coût de a par $c(n_1, n_2)$.

■

Définitions D1.0.3 Nous appelons **source** du graphe d'état, l'état s s'il est ancêtre de tous les états du graphe. L'état s est l'état initial du problème.

■

La fouille dans un graphe d'états vise à construire un chemin vers un état but. La solution à un problème par la fouille est le chemin de l'état initial s jusqu'à un des états but. Cependant, rien ne nous garanti que ces états but coïncident avec les états du graphe. Si tel est le cas, le problème n'a pas de solution par la fouille.

Définitions D1.0.4 L'ensemble des états but forme un espace buts qui peut être disjoint de l'espace d'états. Dans ce cas, on dit que le graphe d'états est **inaccessible**. Dans le cas contraire, on dira que le graphe d'états est **accessible**.

■

D2 Fonctions heuristiques

Cette section donne les définitions intuitive des termes de la fonction d'évaluation $f(n) = g(n) + h(n)$ utilisée dans les méthodes de fouille heuristique considérées. Ces définitions seront formalisées dans la section D3 à l'aide du concept généralisé de la longueur d'un chemin dans un graphe d'états.

Définitions D2.0.1 On indique par $g^*(n)$ le coût d'un chemin minimal (au sens intuitif du terme) dans un graphe depuis l'état initial s jusqu'à l'état n. Dans toute partie du graphe exploré avant l'apparition de l'état n, nous avons $g(n) \geq g^*(n)$. Donc, $g(n)$ est une surestimation de $g^*(n)$.

■

Définitions D2.0.2 On indique par $h^*(n)$ le coût d'un chemin minimal (au sens intuitif du terme) dans un graphe depuis l'état n jusqu'à un état but le plus proche.

■

Définitions C2.0.3 On indique par $h(n)$ le terme heuristique de la fonction $f(n)$ où $h(n)$ est une fonction qui estime le coût d'un chemin minimal (au sens intuitif du terme) dans un graphe depuis l'état n jusqu'à un état but le plus proche.

■

Si l'on considère $g(n)$ comme une surestimation de $g^*(n)$ et $h(n)$ comme une estimation de $h^*(n)$, il est naturel de considérer $f(n)$ comme une estimation de $g^*(n) + h^*(n)$. En choisissant un état n qui minimise $f(n)$, on favorise le développement d'un état dont on estime qu'il est traversé par un chemin minimal depuis l'état initial s vers les états but.

Intuitivement, on peut espérer que le comportement de la fouille, guidée par une

fonction d'évaluation $f(n)$, sera d'autant plus efficace que si $f(n)$ est une bonne approximation de $g^*(n) + h^*(n)$. Pour s'approcher de $g^*(n) + h^*(n)$, on peut penser qu'il soit pertinent de compenser la surestimation de $g^*(n)$ par $g(n)$ en utilisant une fonction $h(n)$ qui sous-estime $h^*(n)$.

D3 Longueur d'un chemin dans un graphe d'états

Le concept de la longueur d'un chemin dans un graphe d'états peut être généralisé à l'aide d'objets mathématiques convenables. L'idée de cette généralisation est de rendre le concept de la longueur indépendant des métriques utilisées pour le calcul des distances d'un chemin. La longueur des chemins est un facteur important dans l'évaluation de la qualité d'une solution obtenue par la fouille. Son indépendance par rapport aux métriques de distance utilisées lors de la fouille facilite l'analyse de la performance. En effet, les résultats d'analyse seront toujours valables en autant que les métriques de distance soient inclues dans le concept général préconisé.

Ainsi, nous pouvons définir, d'une manière formelle le rôle joué par les différentes fonctions heuristiques lors d'une fouille heuristique. La généralisation de la longueur d'un chemin dans un graphe quelconque à l'aide de monoïdes a été étudiée dans le cadre de la théorie des graphes [TRE75], [ARN92]. Comme nous allons voir, les résultats de cette application des monoïdes ont une portée très grandes dans le cadre des graphes d'états pour la fouille heuristique.

Définitions D3.0.1 Un **monoïde** est un couple (V, r) où V est un ensemble d'objets et r un relation définie sur V. Cette relation doit être associative et doit posséder un élément neutre, noté ε.

■

La définition d'un monoïde est très générale. Cependant un monoïde n'est pas celui d'un groupe de la théorie des nombres puisque sa définition n'exige pas que chaque

élément de V possède un élément symétrique [ARN92]. Il est à noter que r est strictement parlant une loi de composition. Un concept encore plus général que celui d'une relation. On peut tirer une généralisation du concept de longueur à l'aide des monoïdes à cause de la relation associative r qui peut représenter un grand nombre d'opérateurs d'intérêt. Par exemple, le monoïde $(\mathcal{R}, +)$ est utilisé pour représenter la distance euclidienne entre deux points. La définition B3.0.2 donne une instance utile de monoïde.

Définitions D3.0.2 Un *monoïde d'arcs* est un couple $(C, \bullet)$ où C est l'ensemble de toutes les suites finies d'arcs de tous les graphes (incluant la suite vide) et $\bullet$ est l'opérateur de concaténation.

■

La concaténation de deux chemins quelconque est toujours possible mais elle ne donne pas nécessaire un chemin. Il est aisé de vérifier que la concaténation est associative.

Pour représenter les arcs pondérés d'un graphe d'états, nous utilisons les monoïdes de valeurs. Mais pour cela nous avons besoin d'une définition portant sur les relations. Cette définition fait ressortir la notion d'ordre dans un ensemble.

Définitions D3.0.3 Une relation r est *strictement croissante à gauche* sur un ensemble $V \subset \mathcal{R}$ si $\forall x, y, z \in V, \exists\, x' \in V, y > x' \Rightarrow r(y, z) > r(x, z)$. Il en est de même pour une relation *strictement croissante à droite*. La relation r est strictement croissante à droite si $\forall x, y, z \in V, \exists\, x' \in V, y > x' \Rightarrow r(z, y) > r(z, x)$. Lorsque l'on remplace l'opérateur relationnel par $\geq$, on obtient des relations croissantes à gauche (ou à droite).

■

Définitions D3.0.4 Un *monoïde de valeurs* est un couple (V, r) où $V \subset \mathcal{R}$ représentant les valeurs des arcs et r une relation croissante (ou strictement croissante) à

gauche et à droite.

■

Par exemple, un monoïde de valeurs peut être le couple $(\mathcal{N}, +)$. C'est-à-dire, les nombres naturels et l'opération d'addition. On remarque que l'élément neutre dans $\mathcal{N}$ est le nombre 0. Par surcroît, il est possible de définir un monoïde de valeurs par $(]0, +\infty[, \times)$. Dans ce cas l'élément neutre est 1 et la multiplication est strictement croissante.

Définitions D3.0.5 Soit (V, r) un monoïde de valeurs et soit $(C, \bullet)$ un monoïde de concaténation que l'on associé à V. Soit $c : C \rightarrow V$ une fonction définie sur V qui donne la valeur des arcs de C. La *longueur* d'un chemin associée au monoïde (V, r) et à la fonction c est une application L définie sur C et prenant ses valeurs dans V.

■

Autrement dit, pour tout arc $a \in C$, $L(a) = a(u)$. L'image de l'élément neutre sous L est l'élément neutre du monoïde (V, r). Donc L(suite vide) = ε. Enfin, à cause du monoïde de concaténation, $\forall S', S'' \in C$ des suites d'arcs, $L(S' \bullet S'') = r(L(S') \bullet L(S''))$. Nous pouvons vérifier facilement la définition en utilisant le monoïde $(V, r) = (\mathcal{N}, +)$. Dans ce cas, la longueur L a le même sens qu'une longueur dans l'espace euclidien. Dans la définition suivante, on indique ce que signifie un chemin minimal dans un graphe d'états.

Définitions D3.0.6 Soit G un graphe d'états. Soit χ un chemin de G joignant un état i à un état j, tel que $\forall \chi'$ un chemin dans G joignant i à j, $L(\chi) \leq L(\chi')$. On dit que χ est un **chemin minimal** de i à j dans le graphe G et on le note par $\hat{L}(i, j)$.

■

Définitions D3.0.7 Soit $\hat{L}(i,J)$ la longueur d'un chemin minimal entre i et J où i est un état et J un ensemble d'états dans un graphe d'états. C'est-à-dire un chemin χ de i à J tel que $\forall \chi'$ un chemin de i à J, $L(\chi) \leq L(\chi')$. On dit que χ est un **plus court chemin** de i

à J.

■

La définition de la longueur d'un chemin dans un graphe d'états permet de justifier la définition du chemin minimal dans un graphe d'états. On peut aussi montrer qu'un chemin minimal est composé de suites d'arcs de longueur minimale.

Théorème D3.0.1 *Soit G un graphe d'états dans lequel i et j sont des états tels qu'il existe un chemin minimal χ depuis i vers j. Soit n un état dans le chemin χ tel que $\hat{L}(i,n)$ et $\hat{L}(n,j)$ existent. Alors, $\hat{L}(i,j)\ r = (\hat{L}(i,n), \hat{L}(n,j))$.*

Démonstration La partie de χ entre les états i et n est notée χ_1. La partie de χ entre les états n et j est notée χ_2. Puisque $\hat{L}(i,n)$ est définie alors il existe un chemin χ_1' de i à n pour lequel $L(\chi_1') = \hat{L}(i,n)$. De même pour le chemin χ_2, il existe un chemin χ_2' de longueur $L(\chi_2') = \hat{L}(n,j)$. Puisque χ est minimal de i vers j alors $L(\chi) \leq L(\chi_1' \bullet \chi_2') = r(L(\chi_1'), L(\chi_2'))$. Nous savons aussi $L(\chi) = L(\chi_1 \bullet \chi_2) = r(L(\chi_1), L(\chi_2))$ et que $L(\chi_2) \geq L(\chi_2')$. La relation r étant strictement croissante à droite, on a $L(\chi) \geq r(L(\chi_1), L(\chi_2'))$. Puisque r est aussi strictement croissante à gauche, on a $L(\chi) \geq r(L(\chi_1'), L(\chi_2))$. Donc, $L(\chi) = r(L(\chi_1'), (L(\chi_2'))$. Par conséquent, $\hat{L}(i,j) = r(\hat{L}(i,n), \hat{L}(n,j))$.

□

Ce dernier théorème portant sur la propriété des chemins optimaux est importante. Il permet de déterminer pour tout graphe d'états que si un chemin minimal χ existe alors tous les sous-chemins de χ sont aussi des chemins minimaux.

Dans tout graphe d'états fini, on peut constater que le nombre d'arcs formant un chemin est toujours borné.

Définitions D3.0.8 On dénote par $\Gamma(n)$ l'ensemble des **successeurs** d'un état n. Un état m est successeur d'un autre état n s'il existe un arc reliant les deux états tels que n précède m dans le graphe d'états.

■

Définitions D3.0.9 On dénote par $\eta(\chi)$ le **nombre d'arcs** le long du chemin χ. On dénote par $\hat{\eta}(\chi)$ le nombre minimal d'arcs le long du chemin χ. Si le contexte est clair on peut aussi dénoter le chemin par les états de ses extrémités. Donc $\eta(n_1, n_2) = \eta(\chi)$ et $\hat{\eta}(n_1, n_2) = \hat{\eta}(\chi)$.

■

Théorème D3.0.2 *Soit un graphe d'états G fini. Alors $\forall\, k \in \mathcal{N}$ et $\forall$ n appartenant à l'ensemble des chemins minimaux partant de l'état initial s, l'ensemble des états tel que $\hat{\eta}(s,n) \leq k$ est un ensemble fini.*

Démonstration Par définition un état dans un graphe d'états fini possède un ensemble fini de successeurs. Lorsque $k = 1$, le théorème est vérifié par inspection.
Admettons que l'ensemble des états tel que $\{\, n \in G \mid \hat{\eta}(s,n) \leq k \,\}$ est fini pour $k \leq m$-1 et $m \in \mathcal{N}$. Or le graphe est fini et $\forall p$ qui est un état du graphe G, $\Gamma(p)$ est un ensemble fini. Donc il existe un nombre positif $M \in \mathcal{N}$ tel que $\forall p \in \{n \in G \mid \hat{\eta}(s,n) \leq k \}$ implique que $|\Gamma(p)| \leq M$. Comme G est également un 1-graphe, un chemin de m arcs est un chemin de m-1 arcs auquel on a jouté un arc supplémentaire. Ainsi, $|\{n \mid \hat{\eta}(s,n) \leq p\}| \leq (m+1)\, |\{n \mid \hat{\eta}(s,n) \leq p-1\}|$. Donc, l'ensemble $\{n \in G \mid \hat{\eta}(s,n) \leq k \}$ est un ensemble fini pour $k \leq p, p \in \mathcal{N}$

□

On peut généraliser le théorème D3.0.2 pour donner un résultat dont la

signification est beaucoup plus importante.

Théorème D3.0.3 *Soit un graphe d'états G fini. Alors* $\forall k \in \mathcal{N}$, *l'ensemble des chemins partant de l'état initial s* $\{ \chi \mid \eta(\chi) \leq k\}$ *est fini.*

Démonstration Par définition un état dans un graphe d'états fini possède un ensemble fini de successeurs. Lorsque $k = 1$, le théorème est vérifiée par inspection.
Admettons que l'ensemble E_k des chemins partant de l'état initial s, $E_k = \{\chi \mid \eta(\chi) \leq k\}$ est un ensemble fini pour $k \leq m+1$, $m \in \mathcal{N}$. Donc l'ensemble des extrémités finales des chemins dans E_k est également fini. Or le graphe est fini alors $\forall p \in E_k$, $\Gamma(p)$ est aussi un ensemble fini. Donc, il existe un nombre $M \in \mathcal{N}$ tel que $\forall p \in E_k$, $|\Gamma(p)| \leq M$. Comme G est également un 1-graphe, un chemin de m arcs est un chemin de m-1 arcs auquel on a ajouté un arc supplémentaire. Ainsi, $|E_k| \leq (m+1)\ |E_k+1|$. Donc l'ensemble $\{\chi \mid \eta(\chi) \leq k\}$ est borné pour $k \leq p, p \in \mathcal{N}$.

□

Définitions D3.0.10 Un graphe d'états est *borné* sur L (la longueur) dès que $\forall k \in \mathcal{N}$, $\forall q \in V$, $\forall \chi$ chemin partant de s, $\eta(\chi) > k \Rightarrow L(\chi) > q$.

■

C'est-à-dire, si un graphe d'états est borné sur L alors pour toute valeurs d'arc q appartenant à l'ensemble général V, il ne peut exister un chemin partant de l'état initial s et comptant un nombre d'arcs aussi grand que l'on veut mais de longueur L inférieure à q. C'est la finitude de l'ensemble des chemins partant de l'état initial d'une graphe d'états fini.

Théorème D3.0.4 *Tout graphe d'états fini est borné sur L.*

Démonstration Par contradiction. Supposons qu'il existe $q \in V$ et qu'il existe un chemin χ partant de l'état initial s tel que $\eta(\chi) > k$ et $L(\chi) \leq q$, $\forall k \in \mathcal{N}$. Or, le graphe

d'états est fini et selon le théorème B3.0.3, $\eta(\chi) \leq k$ pour tout chemin χ partant de l'état initial s. Donc la supposition est fausse et la définition B3.0.10 tient.

□

Lemme D3.0.1 *Pour tout graphe d'états fini et borné sur L. $\forall q \in V$. l'ensemble E des chemins partant de l'état initial s avec $L(E) \leq q$ est un ensemble fini.*

Démonstration Soit I l'ensemble des chemins χ partant de l'état initial s avec $\eta(\chi) \leq k$. Puisque le graphe est borné sur L, $\eta(\chi) > k \Rightarrow L(\chi) > q$. Si $\eta(\chi) \leq k$ ceci implique que $L(\chi) < q$. L'ensemble E est un sous-ensemble I. Comme I est fini (théorème B3.0.3), E l'est également.

□

Le lemme B3.0.1 indique que le nombre de chemins partant de l'état initial s avec une longueur plus petite ou égale à n'importe quelle valeur d'arc est fini. Nous allons montrer les conditions de l'existence d'un chemin minimal.

Théorème D3.0.5 *Dans tout graphe d'états G fini, borné sur L et sans circuit de longueur strictement inférieure à l'élément neutre du monoïde (V, r), il existe un chemin minimal entre l'état initial s et un état n quelconque. $\hat{L}(s,n)$ existe.*

Démonstration Puisque n est un état de G, il existe un chemin χ' de s à n de longueur $L(\chi') = q$, $q \in V$. Puisque G est fini et borné sur L, le lemme D3.0.1 s'applique. L'ensemble des chemins partant de l'état initial s à n ayant une longueur inférieure ou égale à q est un ensemble fini. On peut donc trouver un chemin minimal $\hat{\chi}$ parmi tous les chemins de cet ensemble fini. Puisque G n'a pas de circuit de longueur strictement inférieur à l'élément neutre $\hat{L}(s,n) = L(\hat{\chi})$. Donc $\hat{L}(n,s)$ existe.

□

Le théorème D3.0.5 montre que $\hat{L}(n,s)$ existe lorsque G est fini, borné sur L et sans circuit. Que peut-on dire des chemins partant de l'état initial jusqu'à un état but lorsque l'espace d'états coïncide en tout ou en partie avec l'espace des buts?

Théorème D3.0.6 *Dans tout graphe d'états G fini, borné sur L et accessible (définition D1.0.4), il existe un chemin minimal entre l'état initial s et l'ensemble des buts B. $\hat{L}(s,B)$ existe. De plus, l'ensemble E des chemins de s à B avec L(E) = $\hat{L}(s,B)$ est un ensemble fini.*

Démonstration Puisque G est accessible, il existe un chemin χ de l'état initial à un état but. Par le lemme B3.0.1 on peut poser $q = L(\chi)$ puisque G est un graphe fini et borné sur L. Toujours selon le lemme B3.0.1, l'ensemble E des chemins partant de l'état initial s tel que $L(E) \leq q$ est fini. Alors, l'ensemble E' des chemins partant de s à l'ensemble des états but B avec $L(E') \leq q$ est un sous-ensemble de E. Donc, $E' \subset E$ et E' est aussi un ensemble fini. On peut donc trouver un chemin minimal $\hat{\chi}$ de s à B tel que $\forall\chi''$ chemin partant de l'état initial s à un des états but de B, $L(\hat{\chi}) \leq L(\chi'')$. Puisque le chemin minimal $\hat{\chi}$ existe, $\hat{L}(s,B)$ existe aussi et l'ensemble des chemins de l'état initial s à l'ensemble des états but B est un ensemble fini puisque $E' \subset E$.

□

Ce dernier théorème nous dit qu'il existe au moins un chemin de l'état initial jusqu'à un état but qui est minimal. Si le graphe d'états est fini, borné sur L et accessible, l'ensemble de ces chemins est également fini.

Deux importants corollaires découlent du théorème D3.0.6. Le premier concerne l'existence d'un chemin minimal d'un état n quelconque à l'ensemble des buts B. Le second concerne l'existence d'un chemin minimal d'un état quelconque et ses successeurs.

Corollaire D3.0.1 *Soit un graphe d'états G fini, accessible et borné sur L et B l'ensemble des états but. Pour tout état n appartenant à n'importe quel chemin de s à B, $\hat{L}(n, B)$ existe et l'ensemble des chemins $E = \{\chi \mid \chi$ de n à B, $L(\chi) = \hat{L}(n, B)\}$ est fini.*

Démonstration Puisque G est accessible, il existe un chemin χ de l'état n à un état but. Par le lemme B3.0.1 on peut poser $q = L(\chi)$. puisque G est un graphe fini et borné sur L. Toujours selon le lemme D3.0.1, l'ensemble E des chemins partant de l'état n tel que $L(E) \leq q$ est fini. Alors, l'ensemble E' des chemins partant de n à l'ensemble des états but B avec $L(E') \leq q$ est un sous-ensemble de E. Donc, $E' \subset E$ et E' est aussi un ensemble fini. On peut donc trouver un chemin minimal $\hat{\chi}$ de n à B tel que $\forall\chi$" chemin partant de l'état n à un des états but de B, $L(\hat{\chi}) \leq L(\chi")$. Puisque le chemin minimal $\hat{\chi}$ existe, $\hat{L}(n, B)$ existe aussi et l'ensemble des chemins de l'état initial n à l'ensemble des états but B est un ensemble fini puisque $E' \subset E$.

□

Corollaire B3.0.2 *Soit un graphe d'états G fini, accessible et borné sur L. Soit m, n $\in$ G tels que n est un descendant de m. Si la relation du monoïde est strictement croissante à droite, alors $\hat{L}(m,n)$ existe.*

Démonstration Il existe un chemin minimal de l'état initial s à l'état m. Forcément il existe un chemin de s à n passant par m. Soit $\chi_{m,n}$ une partie du chemin χ joignant les états m et n. C'est un chemin minimal de m à n. En effet s'il existe un chemin $\chi'_{m,n}$ tel que $L(\chi'_{m,n}) \leq L(\chi_{m,n})$ alors selon la définition de relation strictement croissante à droite $L(\chi_{s,m} \bullet \chi'_{m,n}) < L(\chi_{s,m} \bullet \chi_{m,n}) = L(\chi)$ ce qui est impossible puisque χ est un chemin minimal.

□

Il existe donc un chemin minimal entre un état et ses descendants. Par conséquent, si un graphe d'états est un graphe fini, borné sur L, accessible et sans circuit alors il existe au moins un chemin minimal entre l'état initial s et un des états but. Le nombre de ces chemins minimaux est borné. De plus, si un chemin est minimal, il est également minimal par intervalles. Autrement dit, un chemin minimal dans un graphe d'états remplissant les conditions peut être décomposé en sous-chemins minimaux.

La conséquence pratique de ces constatations est qu'il est possible de concevoir une méthode de fouille qui trouve à coup sûr un chemin à coût minimal si cette solution existe. Bien plus intéressant encore, la méthode de fouille peut procéder en obtenant des sous-chemins minimaux qui la mèneront vers une solution finale. La permission d'utiliser les sous-chemins minimaux pour obtenir une solution à coût minimal est un facteur qui facilite grandement la conception de la méthode de fouille heuristique envisagée.

En guise de conclusion, nous proposons une définition formelle des fonctions heuristiques optimales utilisées dans les méthodes de fouille heuristique. Dans la présentation du chapitre 4, les fonctions heuristiques sont utilisées dans le sens de ces définitions.

D3.1.1 Définitions formelles des fonctions heuristiques optimales

Nous pouvons redéfinir les fonctions heuristiques de la section D2 en terme de longueur L.

Définitions D3.1.1 La fonction $g^*(n)$ est la distance minimale depuis l'état initial s. Cette fonction est définie en tout état n de G et $g^*(n) = \hat{L}(s,n)$. Autrement dit, $g^*(n)$ est une mesure de la longueur d'un chemin minimal de s à n si ce chemin existe.

Pour les graphes sans circuits de longueur strictement inférieur à l'élément neutre $g^*(s) = \varepsilon$ car $\hat{L}(s,s)$ est égale à L(suite d'arcs vide). Comme une suite d'arcs vide est représentée par l'élément neutre de C qui est ε, on a bien $g^*(s) = \varepsilon$. Donc, si le monoïde utilisé est $(\mathcal{N}, +)$, alors $\hat{L}(s,s) = 0$. Par contre, si le monoïde est $(\mathcal{R}, \times)$ alors $\hat{L}(s,s) = 1$.

Définitions D3.1.2 La fonction $h^*(n)$ est la distance minimale de n jusqu'à l'ensemble des états but B. Cette fonction est définie en tout état n de G et $h^*(n) = \hat{L}(n,B)$. Autrement dit, $h^*(n)$ est une mesure de la longueur d'un chemin minimal de n à B si ce chemin existe.

■

Pour les graphes bornés sur L. La fonction h^* est définie pour tout état situé sur un chemin de s à l'ensemble des états but B. Cette constatation est une conséquence directe exprimée par le corollaire D3.0.1.

Définitions D3.1.3 La fonction $f^*(n)$ est la **distance minimale** de la fouille. Cette fonction est définie en tout état n de G pour lequel $\hat{L}(s,n)$ et $\hat{L}(n,B)$ sont définies. Donc, $f^*(n) = r(\hat{L}(s,n), \hat{L}(n,B)) = r(g^*(n), h^*(n))$.

■