Le placement de taches sur des architectures parallèles

Françoise André, Jean-Louis Pazat

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N° 614

LE PLACEMENT DE TACHES SUR DES ARCHITECTURES PARALLÈLES

Françoise ANDRÉ
Jean-Louis PAZAT

Février 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Françoise ANDRE
IRISA - Campus de Beaulieu
35042 RENNES CEDEX

Jean-Louis PAZAT
Université de Bordeaux I
UER Maths et Info.
351 Cours de la Libération
33405 TALENCE Cédex

Publication Interne n° 338

Janvier 1987 - 36 pages

LE PLACEMENT DE TACHES SUR DES ARCHITECTURES PARALLELES

TASK ASSIGNMENT IN DISTRIBUTED COMPUTER SYSTEMS

Domaines : Architectures Parallèles
Algorithmes de Placement.

Mots-clés : Placement de tâches, architectures parallèles, placement optimal, problème NP complet, méthodes de résolution exactes et heuristiques.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

RESUME

Le problème que soulève l'adaptation d'un programme parallèle à une architecture répartie, ou "problème de placement de tâches", a pris une grande importance suite au développement des langages parallèles et des architectures réparties.

Dans cet article, nous tentons de faire le point sur ce sujet dans le cadre des architectures à liens fixes : nous précisons la notion de placement ainsi que les critères de définition d'un placement optimal. Ce problème étant NP-complet, nous examinons ensuite les méthodes de résolution exactes et approchées les plus courantes. Nous étudions enfin trois algorithmes particuliers dont les limites communes nous permettent de définir les voies de recherches encore ouvertes sur ce difficile problème.

ABSTRACT

A major issue in the design of distributed computer systems is the choice of allocation of the objects (tasks, communication links...) to the devices (processors, buses...).

This paper addresses the problem of static tasks assignment in computer architectures.

We first investigate the problem of achieving optimal assignments. As this problem is found to be NP-complete in the general case, we focuse on currently used heuristic algorithms. We describe and compare three algorithms which represent the main technics.

TABLE DES MATIERES

1. Cadre de l'étude

Nous nous intéressons au problème du placement d'un programme parallèle sur une architecture répartie. Ce problème a pris de l'importance ces dernières années et est devenu très concret, suite aux développements d'une part des langages parallèles (langages de programmation comme ADA, CSP, etc, ou langages de commande comme celui du système UNIX), et d'autre part des architectures réparties (telles que Cm* [Swan 77], Cosmic Cube [Seitz 85] ou iPSC [Asbury 85]).

Avant d'aborder le sujet proprement dit il est nécessaire de préciser la signification que nous donnons dans le cadre de cette étude aux termes "programme parallèle" et "architecture répartie", tant le domaine couvert serait vaste si l'on conservait à ces deux notions leur sens le plus large.

Nous définissons un programme parallèle comme l'expression d'un algorithme sous forme d'un ensemble de tâches communicantes. Une tâche est un programme séquentiel classique auquel on ajoute des primitives de communication (du type rendez-vous, messages ou autres), qui constituent les seules interactions d'une tâche avec les autres.

La classe d'architectures qui nous intéresse plus particulièrement représente toutes les architectures constituées de plusieurs processeurs qui sont chacun composés d'une unité de contrôle, d'unité(s) de traitement, de mémoire et de coupleur(s) d'entrées/sorties (i.e. architectures de la classe MIMD). Par la suite, nous désignerons cette entité de base de l'architecture indifféremment par le nom de processeur ou de site. Les seules interactions entre les processeurs sont réalisées par les coupleurs d'entrée/sortie via des liens de type bus ou des liaisons bipoint. Nous écartons donc de notre étude le cas des architectures "tableaux" (classe SIMD) pour lequel le placement de tâches ne se pose pas puisqu'il n'existe qu'une seule unité de contrôle qui exécute un seul programme. Remarquons que pour ces architectures, il existe un problème de placement des objets dans les mémoires qui est parfois assez proche de notre problème.

L'adaptation la plus simple entre une architecture et un algorithme est évidemment l'identification de l'un à l'autre. Les architectures systoliques [Kung 82] sont conçues selon ce principe. A partir de la spécification du problème, donné par exemple sous la forme d'un système d'équations comme dans la méthode de Quinton [Quinton 84], on se propose de concevoir un réseau de processeurs spécialement adapté. Un processeur est créé pour exécuter une tâche et les interconnexions matérialisent les flots de données. Nous considérons donc ici le problème de placement comme réglé bien qu'un mécanisme d'adaptation se pose dans le cas où l'on veut utiliser des circuits intégrés comportant des défauts de fabrication de connexions ou des cellules défectueuses. Dans [Yalamanchi 85], il est fait état de travaux ayant pour objectifs d'adapter de tels circuits.

On peut classer dans une catégorie voisine les architectures parallèles dont les liens sont dynamiquement modifiables pour s'adapter à la plupart des algorithmes.

Dans [Snyder 82], Snyder décrit le calculateur CHIP (Configurable HIghly Parallel computer) constitué d'un réseau de processeurs interconnectés par l'intermédiaire de circuits d'aiguillage (Switch) programmables. On peut ainsi représenter exactement la structure de certains algorithmes présentant une topologie classique du type anneau logique, arborescence. Le coût et la complexité des circuits d'aiguillage limitent encore l'utilisation de telles architectures.

Nous avons ainsi restreint notre étude aux architectures à liens fixes, pour lesquelles un algorithme de placement du programme est nécessaire. La topologie de l'architecture, c'est-à-dire les liens entre les processeurs est à priori fixée de manière judicieuse en utilisant soit une structure qui s'adapte "au mieux" à n'importe quel algorithme, soit une structure particulièrement adaptée à certains types d'algorithmes.

- Si l'architecture doit s'adapter au mieux à n'importe quel algorithme, elle doit idéalement posséder un maillage complet entre tous ses processeurs. Ceci

étant pour l'instant irréalisable dès que le nombre de processeurs devient important, on utilise des structures de rayon assez faible ; le rayon d'une architecture étant défini comme la distance maximale entre deux processeurs quelconques. Des structures telles que les Hypercubes (du type iPSC) permettent de minimiser les routages tout en limitant le nombre de liens physiques sur chaque processeur.

- Si on restreint le domaine d'utilisation de l'architecture à certains types d'algorithmes ou de langages, il est possible de déterminer une structure adaptée. Les structures arborescentes semblent particulièrement aptes à l'évaluation de langages fonctionnels (FP par exemple [Backus 78]) tandis que les anneaux sont utilisés dans de nombreux algorithmes de service pour résoudre des problèmes tels que la synchronisation ou l'exclusion mutuelle [Raynal 85].

Aussi intéressante que soit la topologie d'une architecture à liens fixes, il se pose toujours un problème : comment adapter un algorithme divisé en tâches à une architecture donnée.

Nous définissons tout d'abord au paragraphe 2, le problème de placement de tâches sur une architecture et introduisons la notion du placement optimal.

Nous étudions au paragraphe 3 les méthodes de résolution possibles et présentons les classes d'algorithmes de résolution.

Enfin, nous examinons trois algorithmes particuliers qui ont été choisis pour leur bonne représentativité.

En conclusion, nous dégageons les limites de ces algorithmes ainsi que les voies de recherche ouvertes.

2. Définition du problème de placement

2.1. Définition générale

Le placement d'un programme composé de 'n' tâches communicantes sur une architecture parallèle disposant de 'p' processeurs est équivalent à la recherche de toutes les applications de T ---> P où T représente l'ensemble des tâches et P l'ensemble des processeurs. Il y a alors p^n cas à étudier (on suppose toujours qu'une tâche ne requiert jamais plus d'un processeur pour son exécution).

L'étude de tous ces cas et même leur énumération ne peut être envisagée dès que n ou p s'écarte de l'unité ; c'est-à-dire dès que le programme ou l'architecture peut être qualifié de massivement parallèle.

Le problème est d'abord de trouver parmi ces p^n cas, au moins un placement possible (c'est-à-dire permettant l'exécution du programme compte tenu des contraintes matérielles imposées par l'architecture). On peut alors se poser le problème du choix d'un meilleur placement parmi tous les placements possibles.

2.2. Réduction du problème

De nombreux auteurs se sont attachés à réduire les dimensions du problème (à défaut de le résoudre) en imposant des contraintes 'à priori' avant toute recherche de solution. Ces contraintes sont de deux types :

- celles qui sont nécessaires à l'existence de solutions : placement imposé de certaines tâches sur des processeurs disposant de ressources matérielles particulières (Entrées/Sorties, mémoire, processeur arithmétique), limitation du nombre de tâches exécutables sur un site (une seule tâche par processeur dans le cas où le système ne permet pas de multiprogrammation),

- celles qui permettent de limiter le nombre des solutions : Regroupement de certaines tâches, limitation de l'utilisation des routages (un routage étant défini comme une méthode de communication entre deux sites par l'intermédiaire d'un ou plusieurs autres sites. Le routage n'est possible que s'il existe un chemin entre ces sites et un algorithme de routage dans le noyau système de chaque site).

Bien que les dimensions du problème puissent généralement être réduites, le problème reste NP complet (cf.§ 3.3. et [Gondran 85]). Il faut également être prudent lorsque l'on s'impose des contraintes pour limiter le nombre de solutions car on risque d'éliminer les meilleures -si ce n'est toutes les solutions.

Dans tout ce qui suit, on se place dans le cas où les contraintes nécessaires permettent d'affirmer l'existence d'au moins une solution. Nous discutons du choix d'une meilleure solution et des moyens d'y parvenir en un temps raisonnable, quitte à imposer des contraintes supplémentaires pour réduire les dimensions du problème. Dans le cas général, il est nécessaire d'autoriser les routages et de supposer l'existence de capacités de traitement, de mémorisation, de communication et d'entrées/sorties "suffisantes".

2.3. Le placement optimal

2.3.1. Les objectifs généraux

Afin de rechercher un "meilleur" placement, il est nécessaire d'en définir les caractéristiques. Nous pouvons définir un placement optimal comme "garantissant les meilleures conditions possibles d'exécution du programme". Ces conditions dépendent du but recherché : l'application peut être du type "temps réel" ou non ; il peut y avoir un compromis à réaliser entre le nombre de processeurs utilisés et la vitesse du traitement ; enfin il existe de nombreux cas particuliers permettant une définition précise du placement optimal dont nous citons quelques exemples ci-dessous au point 3.

1) Cas d'une application "temps réel"

Le placement doit garantir la terminaison de certaines tâches en un temps fixé. Il y a alors avantage à étudier un placement dynamique des tâches. Une telle méthode est généralement complexe à mettre en oeuvre et utilise souvent des algorithmes par enchères ("Bidding Algorithms"). On trouvera dans [Stankovic 84] une telle méthode adaptable à un grand nombre de cas.

Il faut noter que des méthodes d'allocation statique restent utilisables moyennant certaines adaptations [Ma 84].

2) Cas où la durée totale d'exécution doit être minimale

Le but le plus souvent recherché est la diminution du temps total d'exécution du programme. Ce temps dépend des temps d'exécution de chaque tâche sur un processeur et des temps de communication entre les processeurs supportant des tâches. La réduction du parallélisme (placement conjoint de plusieurs tâches sur un même processeur) peut être déterminante ; enfin le temps de communication est parfois le seul facteur critique.

3) Divers compromis et cas particuliers

Il existe de nombreux cas intéressants pour lesquels la définition du placement optimal peut être simplifiée :

- lorsque tous les processeurs sont identiques et reliés de manière équivalente (maillage complet ou bus), il suffit de réunir les tâches les plus communicantes afin de diminuer le temps de communication qui est souvent le facteur critique,

- si la topologie du programme est bien connue, ou si l'on peut s'y ramener par regroupements de tâches, la recherche du placement optimal peut être inutile lorsqu'on arrive à un cas déjà résolu dont on possède une solution en bibliothèque, comme préconisé dans [Berman 84].

- lorsque le nombre de processeurs que l'on veut utiliser n'est pas défini à priori, il est nécessaire d'évaluer le compromis réalisable entre vitesse de traitement et nombre de processeurs utilisés. Ceci peut être réalisé par itération d'un même algorithme de placement sur une architecture de plus en plus étendue. On demandera alors à l'algorithme de placement d'être rapide, à défaut d'être précis. Des algorithmes du type de celui proposé dans [Schwan 85] que nous examinons en 4.1. sont utilisables.

2.3.2. Critique du placement optimal

Dans toutes les méthodes citées, le placement optimal est toujours défini comme minimisant une fonction de coûts. Cette fonction et ces coûts représentent une approximation statique du fonctionnement dynamique du programme et du calculateur.

En effet, la complexité en temps d'un algorithme est très souvent fonction de ses données d'entrée, de même les communications induites par l'exécution d'un programme parallèle peuvent être liées à la distribution des données.

2.4. Bilan

On a pu montrer que la définition d'un placement optimal n'est pas toujours possible ; à cela, il y a essentiellement deux raisons :

- un tel placement n'existe pas toujours (même si on recherche un placement dynamique),

- sa définition exacte et sa recherche nécessitent l'utilisation de modèles trop précis (donc trop complexes) pour la description de l'architecture et du programme.

Vu l'imprécision des modèles utilisés et la complexité des algorithmes de résolution, il semble préférable de rechercher un "bon" placement plutôt qu'un hypothétique placement optimal.

Bien que l'on puisse penser qu'un algorithme de placement dynamique soit plus efficace qu'un algorithme de placement statique, il n'existe que peu d'algorithmes dynamiques actuellement développés du fait de leur complexité et de leur coût. On s'intéresse donc dans la suite uniquement aux algorithmes de placement statique de tâches sur une architecture à liens fixes qui tentent de trouver un placement optimal par minimisation d'une fonction de coûts.

3. Méthodes de recherche du placement optimal

3.1. Les fonctions de coût

Pour des raisons de simplicité, les algorithmes de placement utilisent pour décrire le comportement d'un programme ou d'une architecture de manière statique, des coûts établis sous forme de constantes.

Les coûts descriptifs représentent une caractéristique de fonctionnement d'un élément. Pour le programme, cet élément est une tâche, pour l'architecture c'est un processeur ou un banc de mémoire. Les coûts descriptifs peuvent également représenter les interactions entre ces éléments, c'est-à-dire les communications entre les processeurs ou les accès à une mémoire (pour l'architecture).

La fonction de coûts utilisée dans un algorithme de placement détermine les caractéristiques du placement optimal que l'on désire obtenir. Elle tient compte de ces caractéristiques par les coûts qu'elle utilise. Son expression doit être suffisamment simple pour être facile à évaluer et éventuellement à borner. Il existe autant de fonctions que d'algorithmes de résolution, chacune possédant des intérêts particuliers. Nous en citerons quelques-unes pour exemple :

- pour une architecture à processeurs identiques et identiquement reliés, la minimisation de la fonction f_o ci-dessous, au moyen des variables de décision Y_{ij} permet la minimisation du temps total de communication.

$$f_o = \sum_{\substack{i,j \\ i \neq j}} C_{ij} \; Y_{ij}$$

avec

C_{ij} : coût de communication entre les tâches i et j
Y_{ij} = 1 si i et j sont sur des processeurs différents
Y_{ij} = 0 si i et j sont sur le même processeur
$i,j \in T$ ensemble des tâches

- pour une architecture à processeurs identiques mais dont les liens peuvent être différents, la fonction f_1 peut être utilisée

$$f_1 = \sum_{\substack{l,q \\ l \neq q}} \sum_{\substack{i,j \\ i \neq j}} C_{ij} \; D_{lq} \; x_{il} \; x_{jq}$$

avec $i,j \in T$; $l,q \in P$ ensemble des processeurs

D_{lq} : coût de communication sur le lien entre les processeurs l et q

C_{ij} : coût de communication entre les tâches i et j

x_{il}, x_{jq} = 1 si la tâche i est sur le processeur l (resp. j placée sur q)
= 0 sinon.

- pour une architecture à processeurs et liens diversifiés, la fonction f_2 permettra de tenir compte des coûts d'exécution sur les processeurs comme préconisé dans [Ma 82] :

$$f_2 = f_1 + \sum_l \sum_i e_i \; k_l \; x_{il}$$

avec les mêmes notations que précédemment et

e_i : coût d'exécution de la tâche i

k_l : coût d'utilisation du processeur l.

- on pourra également tenir compte des coûts d'interférence du placement conjoint de tâches sur un même processeur en utilisant la fonction f_3 (coût du processeur [Lo 84] ou coût de perte de parallélisme [Ward 84]).

$$f_3 = f_2 + \sum_l \sum_{\substack{i,j \\ i \neq j}} x_{il} \; x_{jl} \; B_l$$

avec les notations précédentes et

B_l : coût d'interférence au placement conjoint de 2 tâches sur le processeur l.

Il est possible de définir d'autres fonctions (voir [Price 84]) et d'ajouter des contraintes pour améliorer les modèles. Par exemple, la contrainte suivante permet de limiter le nombre de tâches par processeur.

$$\forall l \in P \sum_{i \in T} x_{il} <= k$$

avec P : ensemble des processeurs

T : ensemble des tâches

k : entier positif ou nul.

3.2. Les différents modèles utilisés

Bien qu'il soit possible de ramener le problème du placement de tâches à des équations du type précédent, les modèles utilisés pour les descriptions de l'architecture et du programme sont différents selon les auteurs.

Le plus souvent, le programme est représenté par un graphe dont les sommets représentent les tâches et les arcs les communications entre tâches. Les sommets sont valués par les coûts d'exécution et les arcs par les coûts de communication. La machine est représentée par un graphe processeurs-liens physiques, valué par les coûts d'utilisation des processeurs pour les sommets et les coûts des communications pour les arcs (figure 1). On utilise souvent une représentation de ces graphes par matrice d'indice :

c(n,n) pour les tâches avec n = nombre de tâches

c_{ii} = coût d'exécution d'une tâche i

c_{ij} = coût de communication entre les tâches i et j $\quad i \neq j$

d(p,p) pour l'architecture avec p = nombre de processeurs

d_{ll} = coût d'utilisation de processeur l

d_{lq} = coût de communication entre les processeurs l et q $\quad l \neq q$

La recherche du placement optimal consiste alors à chercher une inclusion (figure 2).

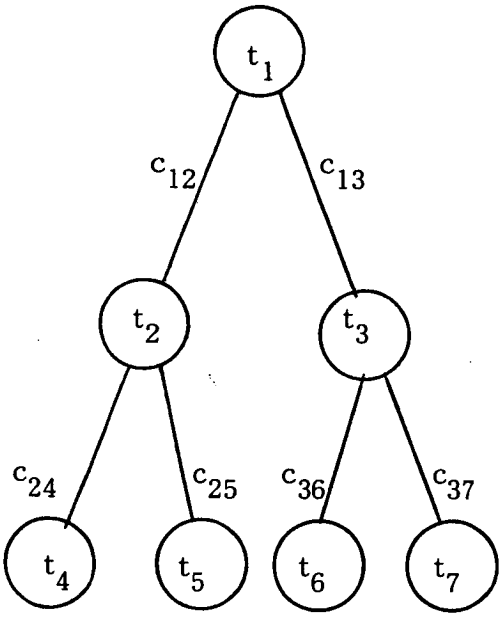
Dans quelques cas [Schwan 85], il est directement fait usage d'équations avec contraintes sans définition explicite de graphes (modèle de proximité).

L'architecture et le programme peuvent aussi être représentés comme dans |Lo 84|, par un graphe biparti. Les sommets "normaux" sont les processeurs. Les sommets "distingués" sont les tâches ; ils sont reliés par des arcs valués représentant les coûts de communications (entre tâches). Il existe un arc entre une tâche et un processeur p_j lorsque la tâche peut être placée sur le processeur p_j. L'arc est alors valué par un coût signifiant le coût d'avoir placé la tâche sur le processeur fictif $\bar{p}_j$ représentant l'ensemble P-p_j (figure 3). La minimisation de la fonction de coûts est réalisée par la recherche d'une partition du graphe de coupe minimale, chaque partition contenant un et un seul processeur ainsi que les tâches qui sont placées sur celui-ci (figure 4). Un tel modèle suppose que tous les processeurs sont reliés de manière équivalente.

3.3. Les classes d'algorithmes de résolution

Le problème de placement de tâches sur une architecture multiprocesseurs étant NP complet, nous sommes amenés à utiliser des algorithmes classiques de résolution de problèmes NP. Ces algorithmes peuvent être exacts (énumérations) ou approchés. On trouvera dans [Gondran 85] plus de détails sur les notions de complexité ainsi que sur les algorithmes de résolution de ces problèmes.

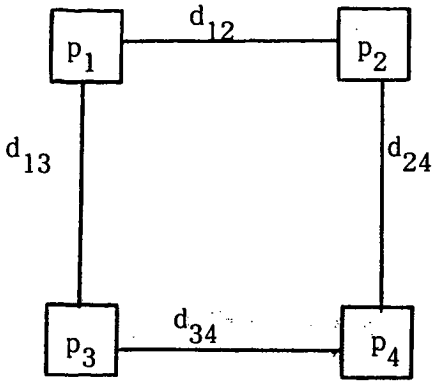
Programme



t_i : tâche

c_{ij} : coût de communication entre t_i et t_j

Architecture



p_i : processeur i

d_{ij} : coût de communication entre p_i et p_j

Figure 1 : Modélisation par deux graphes non orientés.

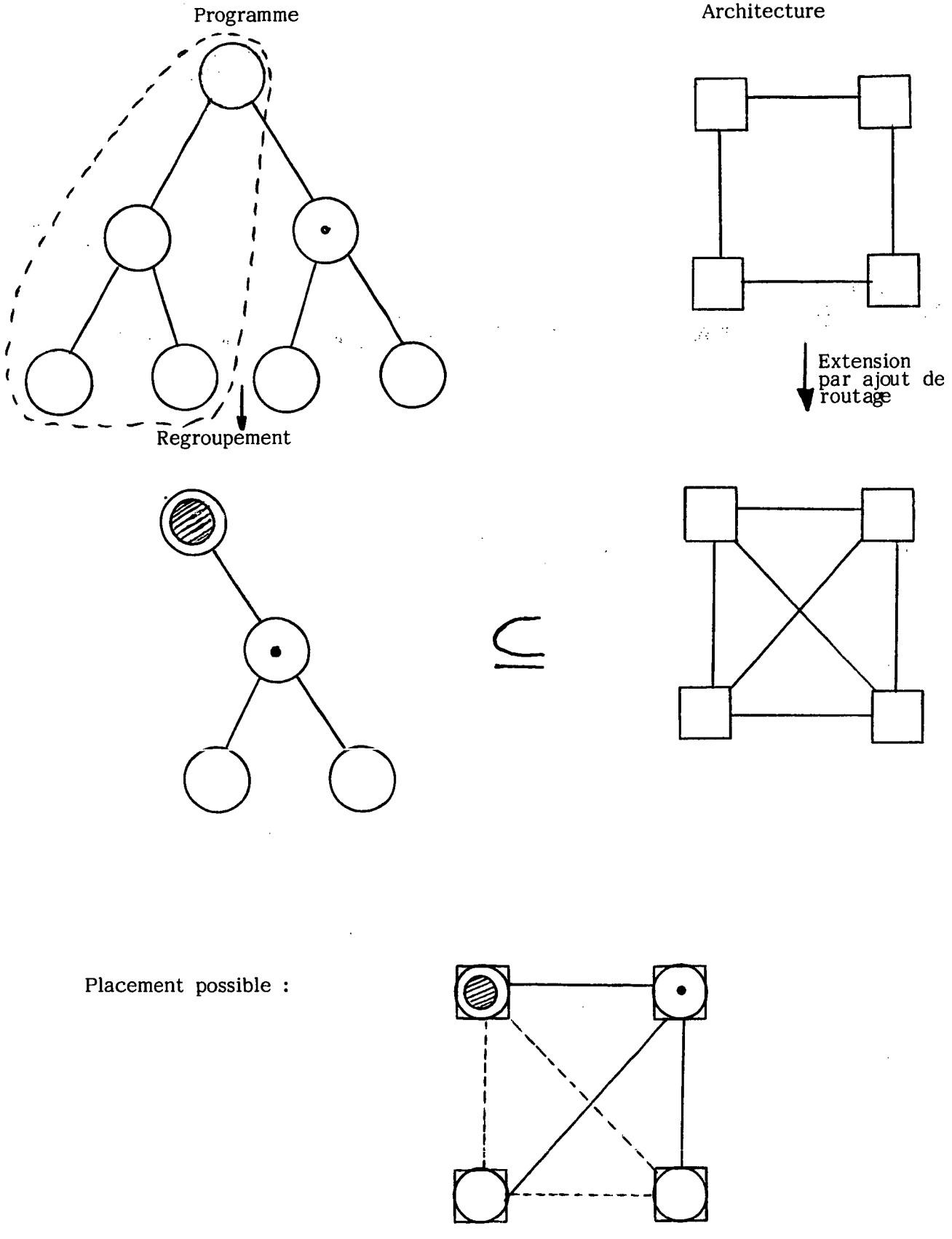
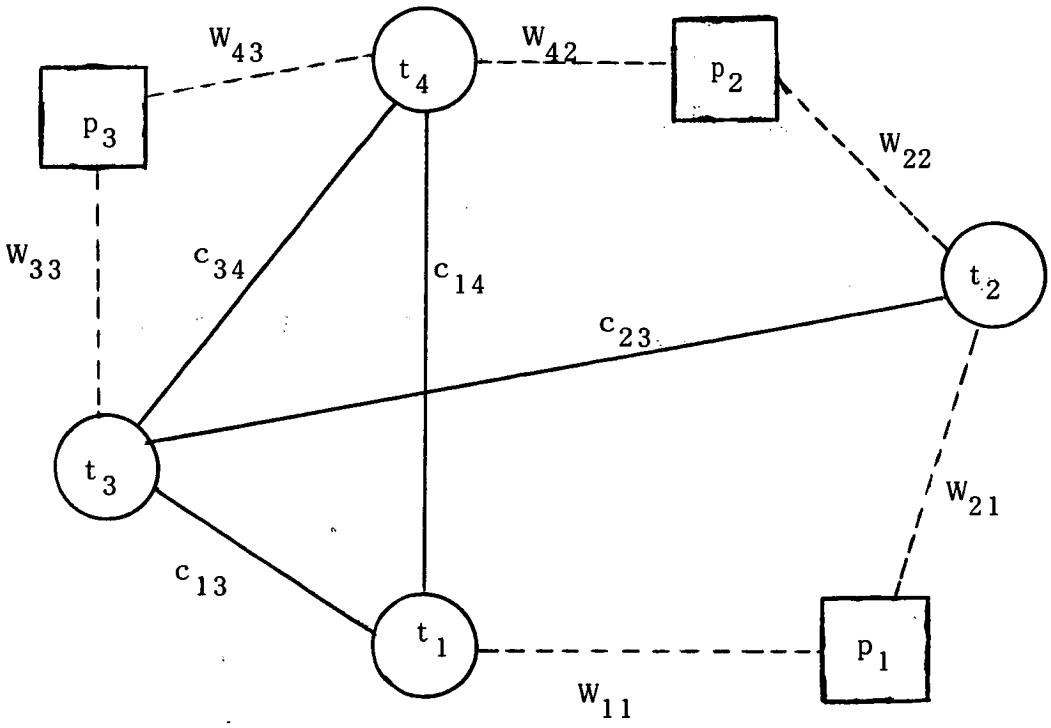


Figure 2 : Placement par recherche d'une inclusion du graphe réduit.



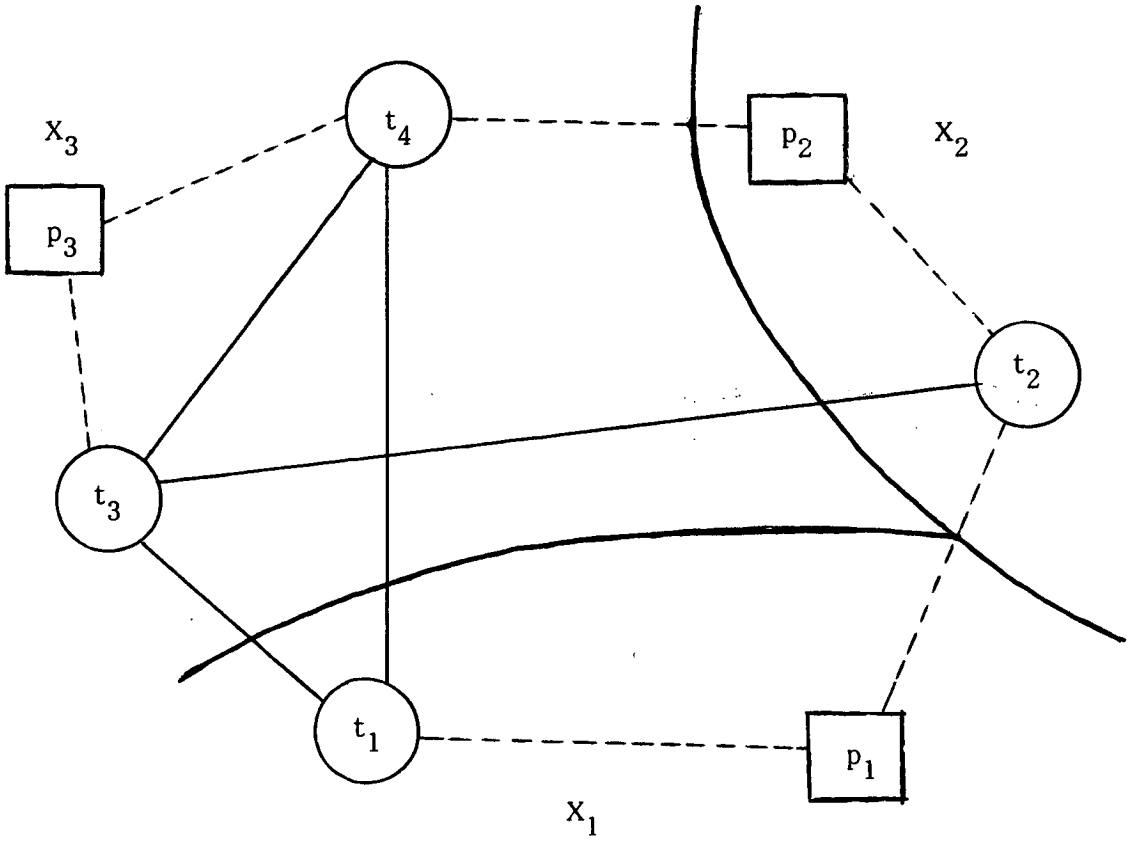
t_i : tâche i

p_j : processeur j

c_{ik} : coût de communication entre t_i et t_k

W_{ij} : coût de placement de la tâche i sur le processeur p_j

Figure 3 : Modélisation par un graphe biparti.



X_i : partition associée au processeur p_i

Figure 4 : Placement par partitionnement d'un graphe biparti.

3.3.1. Les algorithmes exacts

Ce sont les algorithmes qui permettent toujours de trouver une (ou toutes les) solution(s) minimale(s) ou ξ-approchée(s). Une solution est minimale si elle réalise bien le minimum de la fonction de coûts ; elle est ξ-approchée si elle se rapproche à ξ près du minimum, ξ étant fixé. Le problème étant NP complet, ces algorithmes procèdent par énumération.

Lorsque les dimensions du problème deviennent importantes, une énumération explicite s'avère irréalisable. On a alors recours aux procédures classiques par séparation et évaluation ("Branch and Bound") que nous explicitons ci-après. Ces algorithmes réalisent une construction partielle d'un arbre de recherche des solutions.

Chaque sommet de l'arbre correspond à un sous-ensemble de solutions : certaines variables du problème sont fixées, d'autres non. Le sommet de base de l'arbre représente l'ensemble de toutes les solutions réalisables. Les sous-ensembles sont réduits à chaque niveau en descendant dans l'arbre, lors de la séparation d'un sommet.

Il existe diverses méthodes de séparation des sommets [Gondran 85]. Cette séparation est en général binaire (création de 2 successeurs) et réalise une partition du sous-ensemble que représente le sommet courant en deux sous-ensembles disjoints. Ceci correspond à fixer la valeur d'une variable encore libre à "vrai" dans un cas, à "faux" dans l'autre.

Pour éviter le parcours de toutes les branches, le principe des algorithmes par Séparation et Evaluation est d'associer à chaque sommet S parcouru, une évaluation par défaut ev(S) de la fonction de coût f(x) à minimiser. On choisit la fonction ev(S) pour qu'elle définisse un minorant (aussi grand que possible) de f(x) sur S. On a donc $ev(S) \leq f(x)\ \forall x \in S$.

L'intérêt principal de cette évaluation est le suivant : si l'on connaît une solution réalisable $\hat{x}$ du problème et si

$$ev(S) \geq f(\hat{x})$$

alors il n'existe pas de solution meilleure dans le sous-ensemble S et il est inutile de parcourir les branches à partir de S.

Cela suppose pratiquement de connaître une bonne solution réalisable $f(\hat{x})$.

Une brève description des classes d'algorithmes va montrer l'utilisation de cette solution.

Le parcours et la construction de l'arbre (qui sont simultanés) peuvent être faits selon différentes règles, ce qui permet de distinguer différentes classes d'algorithmes : Séparation et Evaluation Séquentielle (SES), Séparation et Evaluation Progressive (SEP), meilleur choix...

Les algorithmes par Séparation et Evaluation Séquentielle ("Branch and Bound depth first") explorent l'arbre en profondeur d'abord, c'est-à-dire en parcourant toute une branche. Dès que l'on a trouvé une solution réalisable, on possède une borne permettant l'élimination d'autres branches, ce qui est intéressant dans le cas des problèmes fortement contraints où il est difficile de trouver simplement une solution réalisable. Par contre, un retour arrière pour explorer une nouvelle sous-branche est nécessaire.

Les algorithmes par Séparation et Evaluation Progressive ("Branch and Bound Breadth first") explorent tous les noeuds "pendants" pour un même niveau de profondeur. Elles sont assez lourdes à mettre en oeuvre mais intéressantes lorsque l'on connaît une bonne solution réalisable avant de débuter le parcours car on élimine rapidement de nombreuses branches.

Une variante des algorithmes par Séparation et Evaluation Progressives, dite du meilleur choix ("Branch and Bound Best first") , explore les noeuds pendants selon l'ordre croissant de leur évaluation.

Pour toutes ces méthodes, il est recommandé de calculer une borne supérieure de la fonction de coûts avant le début de l'algorithme. Cette borne peut être calculée au moyen d'une solution fournie par un algorithme approché. Il peut également être intéressant de renouveler cette opération sur chaque sommet.

Bien que la complexité de ces algorithmes soit identique à celle d'une énumération explicite dans le pire cas, les algorithmes par Séparation et Evaluation sont en moyenne plus efficaces. On trouvera dans [Wah 84] un algorithme de ce type utilisé pour résoudre un problème de placement de fichiers (problème proche du nôtre).

3.3.2. Les algorithmes approchés

Ce sont les algorithmes qui permettent parfois de trouver une solution optimale ou une solution ξ-approchée. Il ne faut pas les confondre avec des algorithmes exacts de recherche de solutions ξ-approchées. Du fait que le problème approché est également NP complet, il n'existe que les algorithmes par énumération qui permettent de trouver une solution ξ-approchée. Un algorithme approché, on devrait dire approximatif, peut fournir une solution éventuellement très éloignée du minimum recherché ([Price 84]), même s'il est "bon" en moyenne. Il n'y a pas ou peu d'espoir de trouver des algorithmes polynomiaux fournissant à coup sur un "bon" placement.

Il existe essentiellement 2 types d'algorithmes approchés qui sont polynomiaux ; les algorithmes "gloutons" et les algorithmes "itératifs" :

- Les algorithmes gloutons doivent être initialisés par une solution partielle réalisable qu'ils cherchent à étendre. A chaque étape, on s'interdit de remettre en cause un choix précédemment établi. Il est recommandé de faire d'abord les choix les moins informants (c'est-à-dire qui ont le moins de conséquences) afin qu'un mauvais choix ne compromette pas le résultat final. Chaque choix minimise une fonction partielle des coûts. Nous en verrons un exemple au paragraphe 4.1..

- Les algorithmes itératifs sont initialisés avec une solution complète (éventuellement irréalisable) qu'ils cherchent à améliorer à chaque itération. L'algo-

rithme se termine lorsqu'une itération n'a pas permis d'amélioration. Ces algorithmes sont souvent utilisés pour tenir compte de contraintes (taille mémoire par exemple) qui ont été abandonnées pour la recherche d'une première solution, par un algorithme "glouton" en général.

La plupart des algorithmes de placement comportent plusieurs phases de type différent (glouton, itératif, "Branch and Bound"), afin d'améliorer la qualité du placement obtenu, ceci au prix d'une certaine lourdeur.

Il existe également des algorithmes par séparation et évaluation modifiés soit par limitation du retour arrière, soit par utilisation d'une limite de temps.

3.3.3. Choix d'un algorithme

Il est nécessaire de se fixer clairement l'objectif visé avant de choisir un algorithme : seuls les algorithmes exacts permettent d'obtenir un placement optimal et seuls les algorithmes approchés permettent d'obtenir un placement en un temps polynomial. Aucun compromis ne peut permettre de trouver (à coup sûr) un bon placement en un temps polynomial.

- Si la précision de placement prime, on doit utiliser un algorithme d'énumération. Pour améliorer le temps de calcul, il est possible de rechercher d'abord un (peut-être) bon placement par un algorithme approché [Gabrielian 84]. Ce cas se rencontre lorsque le modèle statique est considéré comme suffisamment précis, ou lorsque le placement a un caractère "définitif" comme pour la construction de circuits spécialisés.

- Si la rapidité de l'algorithme de placement prime, par exemple dans le cas dans un système d'exploitation, on peut se contenter d'un algorithme approché, en n'oubliant pas que les résultats peuvent éventuellement être mauvais.

- Enfin, il existe quelques cas où le problème peut être réduit à un problème polynomial : pour un système biprocesseur, il suffit de trouver une partition d'un graphe en deux classes [Lo 84] ; pour des processeurs identiques et reliés de manière équivalente supportant au plus deux tâches, il existe également un algorithme polynomial. Ces cas sont hélas très restrictifs et il est le plus souvent nécessaire

d'utiliser un des algorithmes évoqués plus haut.

4. Etude de quelques solutions

4.1. Un allocateur de ressources pour Cm*

Le modèle

K. Schwan et C. Gaymon proposent dans [Schwan 85] une méthode de description formelle des architectures parallèles, des tâches et de leurs communications (modèle de proximité). La notion de tâche est précisée dans ce modèle. On considère d'une part les données, qui sont partageables et appelées objets, d'autre part les codes exécutables appelés processus. Les auteurs proposent de résoudre deux problèmes interdépendants : la distribution des objets dans les mémoires et l'allocation des processeurs physiques aux processus. La technique proposée fait appel à une méthode approchée n'utilisant pas de retour arrière.

Cet allocateur est statique (pas de migration d'objets ni de processus en cours d'exécution), il est destiné à être inclus dans le système d'exploitation du multiprocesseur Cm* (StarOs). L'accent est mis sur la rapidité de la méthode et non sur l'optimalité du placement obtenu ; l'argument invoqué étant l'insuffisance du modèle utilisé pour décrire parfaitement l'architecture de Cm*.

La méthode est divisée en 3 phases :

- placement des objets "autour" des processus,
- allocation des processeurs physiques aux processus,
- déplacement éventuel d'objets pour satisfaire les contraintes de taille mémoire.

Le critère d'optimisation est la minimisation des temps totaux de communication.

L'algorithme

La première phase regroupe les objets "autour" des processus qui les utilisent le plus. C'est une procédure de type "glouton" (à chaque itération, un objet est placé). La fonction choisie est une fonction d'efficacité qui doit être maximisée.

$$f(i,k) = x_{ik}\, D_{ik} \left(\sum_{m \in K^*} W_{km} \left(\sum_{j \in I^*} C_{ij}\, x_{jm} \right) \right)$$

x_{ik} = 1 si l'objet i est placé autour du processus k

D_{ik} = fréquence d'accès de l'objet i par le processus k

W_{km} = gain/perte du regroupement des processus k et m

C_{ij} = gain/perte du regroupement des objets i et j

K^* = ensemble des processus

I^* = ensemble des objets.

Le regroupement des objets est effectué par taille décroissante, ce qui permet d'éviter le déplacement des objets de taille importante lors de la troisième phase. La fonction f(i,k) représente le gain du placement de l'objet i autour du processus k. Pour un programme divisé en tâches communicantes sans partage de données explicite, cette phase permet le placement des structures de communications et de synchronisation (boîtes aux lettres).

La seconde phase est totalement dépendante de la configuration de l'architecture utilisée.

Rappelons à ce sujet que l'architecture Cm* regroupe les processeurs par sous-ensemble ou amas ("cluster" dans la terminologie de Cm*). Les communications à l'intérieur d'un même amas sont plus rapides que les communications inter-amas.

La phase 2 de l'algorithme cherche à maximiser la fonction g(k,p) représentant l'efficacité du placement d'un processus k sur un processeur p, pour tout k et tout p. Cette fonction, présentée ci-dessous, est la somme de trois termes reflétant

- le gain dû au regroupement du processus k avec le processus m, déjà placé sur le processeur p,

- le gain dû au placement de k sur p sachant que le processus m est placé

sur un processeur q situé dans le même amas que p,

- ce même gain dans le cas où m est sur un processeur situé dans un autre amas que p.

$$g(k,p) = V_{pp} \sum_{\substack{m \in K^* \\ m \neq k}} Y_{kp}\, Y_{mp}\, W_{km}$$

$$+ \sum_{\substack{q \in r \\ q \neq p \\ p \in r}} V_{pq} \left(Y_{kp} \left(\sum_{\substack{m \in K^* \\ m \neq k}} Y_{mq}\, W_{km} \right)\right)$$

$$+ \sum_{\substack{q \in R^* - r \\ p \in r}} V_{pq} \left(Y_{kp} \left(\sum_{\substack{m \in K^* \\ m \neq k}} Y_{mq}\, W_{km} \right)\right)$$

avec

$k,m \in K^*$; $p,q \in P^*$; $r \in R^*$

K^* : ensemble des processus

P^* : ensemble des processeurs

R^* : ensemble des amas.

Y_{mq} = 1 si le processus m est placé sur le processeur q

0 sinon

V_{pp} : mesure de proximité entre un processeur et sa mémoire

V_{pq} : mesure de proximité entre un processeur p et la mémoire d'un autre processeur q.

Les processus sont ordonnés par ordre décroissant de taille des objets regroupés autour d'eux, les amas par ordre décroissant de leur taille mémoire. La procédure d'allocation est de type "glouton" ; en cas d'égalité de deux placements, on choisit celui qui équilibre les charges mémoires.

La troisième phase (déplacement des objets) vérifie la faisabilité de l'allocation proposée et la modifie si nécessaire. Les contraintes de faisabilité sont de deux ordres : nécessité de la présence de matériel spécialisé pour un processus et capacité mémoire. Ces contraintes ne pouvant être exprimées avant l'attachement physique, les auteurs ont décidé de déplacer les objets entre les mémoires lors de

cette troisième phase. La recherche d'une nouvelle place est réalisée en réutilisant la phase 1 avec ajout des contraintes.

Evaluation

Cet allocateur est rapide (pas de retour arrière ni de phase itérative pour la recherche initiale) et est donc bien adapté à un système d'exploitation. Il faut noter que la structure de Cm* est très particulière, les contraintes topologiques étant quasi inexistantes (toute mémoire est accessible de tout processeur) ; une procédure d'allocation approchée peut donc donner d'assez bons résultats.

4.2. Un algorithme général parfois optimal

Le modèle

Virginia M. Lo étudie deux méthodes de placement dans [Lo 84] : soit la minimisation des coûts d'exécution et de communication, soit la minimisation de ces mêmes coûts ainsi que d'un coût d'interférence dû au placement conjoint de deux tâches sur un même processeur.

Pour ces deux méthodes sont utilisés le même type de modèle et d'algorithme. La représentation du problème est effectuée grâce à un graphe biparti ; un noeud ordinaire représentant une tâche, un noeud distingué, un processeur (figure 3). Un arc entre tâches est valué par c_{ij} (coût de communication entre les tâches i et j) ; un arc entre tâche i et processeur q, par un coût Wiq, représentant le coût de placement sur $\bar{q}$ (cf. § 3.2. et figure 3). Il n'est pas tenu compte des coûts de communication entre les processeurs qui sont supposés identiques (maillage complet ou bus).

Algorithme 1

Le placement sur n processeurs est effectué par une partition du graphe par une n-coupe (n-way cut), chaque partie contenant un et un seul processeur. Le coût d'une telle partition est défini comme la somme des coûts des arcs entre les parties. Afin que ce coût soit exactement égal à la somme des coûts d'exécution des

tâches sur les processeurs et des coûts des communications, la valeur des Wiq a été choisie ainsi :

$$W_{iq} = \frac{1}{n-1} \sum_{\substack{p \neq q \\ p \in P}} e_{ip} - \frac{n-2}{n-1} e_{iq}$$

avec $i \in T$ ensemble des tâches

$p,q \in P$ ensemble des processeurs

$e_{ip}(e_{iq})$ coût d'exécution de la tâche i sur le processeur p (resp.q).

Ce problème est NP complet pour n quelconque ([Gondran 85]) ; pour n=2, une solution peut être trouvée en un temps polynomial grâce à un algorithme de flot maximum/coupe minimum (maxflot/mincut). L'idée de Lo est donc de rechercher les n-coupes en recherchant successivement les coupes (2-coupes).

Dans une première phase, on réalise pour chaque processeur p_i une coupe minimale : l'une contenant p_i, l'autre contenant tous les autres processeurs considérés comme un seul processeur $\overline{p_i}$; ceci au moyen d'un algorithme de flot maximum/coupe minimum. Les tâches ainsi placées sur p_i sont celles qu'une n-coupe de coût minimum aurait également placé sur p_i. A la fin de cette phase, on élimine les tâches placées. On recalcule les différents coûts pour les tâches non placées, communicant avec les tâches placées. Cette phase est alors réitérée jusqu'au placement de toutes les tâches ou jusqu'à ce qu'une itération n'ait produit aucun placement. Si le placement réalisé est complet, Lo a montré qu'il est optimal. Lorsque le placement est incomplet, une recherche que nous ne détaillons pas ici est effectuée pour placer les tâches restantes.

Bilan

Le principal intérêt de cet algorithme est sa faculté de trouver parfois un placement optimal et surtout de le savoir. Néanmoins, un tel algorithme ne peut être adapté à n'importe quelle architecture du fait du modèle (on ne tient pas compte des coûts de communications entre les processeurs).

Algorithme 2

Une modification du modèle est proposée par Lo pour éviter les réductions abusives de parallélisme (la minimisation des coûts de communications tendant à placer toutes les tâches sur un seul processeur). Lo introduit des interférences I_{ij} dûes au placement conjoint de deux tâches i et j et modifie ainsi les coûts précédents :

C_{ij} devient $C_{ij}-I_{ij}$

W_{iq} devient $W_{iq} + \frac{1}{2(n-1)} \sum_{j \in T} I_{ij}$

Bilan

Ces deux algorithmes ont permis de placer des programmes parallèles d'une manière assez efficace puisque plus de 90% des programmes placés par ces algorithmes s'exécutent en moins de 1,5 fois leur temps d'exécution optimal (sur une architecture donnée).

4.3. Un algorithme par séparation et évaluation séquentielle

Le modèle

Le modèle utilisé est un modèle de graphe des tâches. La fonction f donnée ci-dessous, proposée dans [Gabrielian 84], doit être minimisée.

Ceci correspond à minimiser les coûts de communication entre les tâches sans tenir compte d'aucune contrainte topologique (ce qui est adapté à une architecture de type bus), ni d'aucun équilibrage de charges (sauf respect des capacités maximum).

$$f = \sum_{k} \sum_{\substack{l \\ k<l}} \sum_{i} \sum_{\substack{j \\ i \neq j}} t_{kl} \; x_{ki} \; x_{lj}$$

avec

$k,l \in T$ ensemble des tâches

$i,j \in P$ ensemble des processeurs

x_{ki} = 1 si la tâche k est située sur le processeur i

0 sinon

t_{kl} représentant le coût de communication entre les tâches k et l

Une contrainte de charge des processeurs est ajoutée :

$$\forall i, \sum_k C_k \, x_{ki} \leq d_i$$

avec

C_k : charge requise par l'exécution de la tâche k

d_i : capacité maximale du processeur i.

Certaines transformations sont applicables sur cette fonction pour en réduire la complexité, cependant une énumération de tous les choix possibles n'est pas envisageable, d'où l'utilisation d'heuristiques.

L'algorithme

Les algorithmes par SES présentent généralement deux inconvénients majeurs :

- Lorsque la borne supérieure utilisée pour l'élimination des noeuds est trop grande, de nombreuses branches sont explorées inutilement au début de l'algorithme.

- La séparation d'un sommet est réalisée par une méthode heuristique qui doit être aussi bien adaptée que possible au problème. Or, le choix d'une heuristique reste souvent lié à l'expérience.

A partir de ces considérations, A. Gabrielian et D.B. Tyler ont proposé un algorithme par SES 'amélioré' dans [Gabrielian 84].

Le calcul d'une borne supérieure de bonne qualité est réalisé par l'utilisation de plusieurs heuristiques sans 'retour-arrière' sur l'arbre de recherche de solutions (une seule branche est alors parcourue). La meilleure solution est retenue pour le calcul de la borne supérieure.

Le choix d'une bonne heuristique pour l'algorithme SES est réalisé d'une manière très simple : l'heuristique choisie est celle qui a donné la meilleure solution (Ce choix pouvant par ailleurs être parfaitement automatisé).

Le fait d'appliquer plusieurs heuristiques pour la recherche d'une bonne solution a déjà été proposée dans [Gondran 85]. Cela est considéré comme assez peu coûteux du fait de la disproportion existant entre la taille de l'arbre de recherche complet (p^n) et la complexité des heuristiques utilisées (O(np)).

Les heuristiques sont la combinaison de la stratégie du choix du prochain objet à placer (fonction de sa taille et des coûts de communication) et de celle du choix de la prochaine ressource à étudier pour le placement (fonction de la taille et de la "proximité").

Bilan

L'originalité de cet algorithme est l'automatisation du choix de la stratégie. Comme les auteurs le remarquent, si l'algorithme fonctionne généralement bien, il existe des cas 'pathologiques' pouvant nécessiter un temps de calcul important. De plus, pour un problème très contraint, la recherche de bonnes solutions peut être longue. On retrouve les critiques habituelles des algorithmes par séparation et évaluation.

4.4. Comparaison récapitulative

Les trois algorithmes étudiés représentent trois points de vue très différents que nous résumons dans le tableau ci-dessous.

Algorithme		[1] [Schwan 85]	[2] [Lo 84]	[3] [Gabrielian 84]
Classe		approché	approché parfois optimal	énumération implicite
Topologie de l'architecture		Cm*	bus ou maillage complet	bus ou maillage complet
Prise en compte de contraintes	mémoire	oui	non	oui
	existence matériel spécialisé	oui	oui	oui
Intérêts particuliers		intégré à un système d'exploitation	parfois optimal	initialisation par algorithmes approchés
Technique de base utilisée		glouton	flot maximum/ coupe minimale	séparation et évaluation séquentielles (Branch and Bound Depht First)

Figure 5 : Comparaison des 3 algorithmes.

5. Conclusion

Ces algorithmes tiennent très peu ([Schwan 85]) ou pas du tout ([Lo 84], [Gabrielian 84]) compte de la topologie de l'architecture. Toutes les architectures cibles sont en effet basées sur des bus et le graphe de l'architecture est un maillage complet.

Le problème de topologie est le problème le plus nouveau que pose l'utilisation des multiprocesseurs de type Transputer [Walker 85], iPSC, etc... et très peu d'algorithmes réalisent une minimisation des coûts de communication et d'exécution en tenant fidèlement compte de l'architecture.

Lorsque l'algorithme est destiné à une seule architecture, il est souvent possible d'opérer des simplifications. Dans [Bokhari 81], le placement optimal est défini comme celui minimisant le nombre de routages, sans tenir compte de la longueur de ceux-ci.

Dans tous les cas, l'algorithme de placement de tâches constitue un élément essentiel pour l'utilisation d'architectures réparties ; l'efficacité du calculateur du point de vue de l'utilisateur étant très liée au placement. Les éléments apportés dans cet article doivent aider à la définition d'algorithmes adaptés aux nouveaux besoins.

Bibliographie

[Asbury 85] R. Asbury, S.G. Frison, T. Roth
"Concurrent Computers Ideal for Inherently Parallel Problems",
Computer Design, 1er Septembre 1985, 99-107.

[Backus 78] J. Backus
*"Can Programming be liberated from the Von Neumann style ?
A Functional Style and its Algebra of Programs"*.
ACM Vol. 21, N°8, Août 1978, 613-641.

[Berman 84] F. Berman, L. Snyder
"On Mapping Parallel Algorithms into Parallel Architectures",
Proc. of the Int. Conf. on Parallel Processing, Bellaire, MI,
21-24 Août 1984, 307-309.

[Bokhari 81] S.H. Bokhari
"On the Mapping Problem".
IEEE Trans. on Computers
Vol. C.30, N°3, Mars 1981, 207-214.

[Gabrielian 84] A. Gabrielian, D.B. Tyler,
"Optimal Object Allocation in Distributed Computer Systems".
Proc. of the 4' Int. Conf. on Distributed Computing Systems,
San Francisco, California, 14-18 Mai 1984, 88-95.

[Gondran 85] M. Gondran, M. Minoux,
"Graphes et Algorithmes".
Collection de la Direction des Etudes et Recherches d'EDF,
EYROLLES, Paris 1985 (2è édition).

[Kung 82] H.T. Kung
"Why Systolic Architectures".
Computer, vol.15, n°1, Janv. 1982, 37-46.

[Lo 84] V.M. Lo,
"Heuristic Algorithms for Task Assignment in Distributed Systems",
Proc. of the Int. Conf. on Distributed Computing Systems,
San Francisco, California, 14-18 Mai 1984, 30-39.

[Ma 82] P.Y. Ma, E.Y.S., M. Tsuchiya,
"A Task Allocation Model for Distributed Systems",
IEEE Trans. on Computers,
Vol. c31, N°1, Janvier 1982, 41-47.

[Ma 84] P.Y. Ma,
"A Model to Solve Timing Critical Application Problems in Distributed Computer Systems",
Computer, Vol. 17, n°1, Mai 1985, 145-159.

[Price 84] C.C. Price,
"Software Allocation Models for Distributed Computing Systems",
Proc. of the 4' Int. Conf. on Distributed Computing Systems,
San Francisco, California, 14-18 Mai 1984, 40-48.

[Quinton 84] P. Quinton,
"Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations",
Proc. of the 11' Int. Symposium on Computer Architecture,
An Arbor, Michigan, 5-7 Juin 1984, 208-214.

[Raynal 85] M. Raynal,
"Algorithmes distribués et protocoles",
EYROLLES, 1985.

[Schwan 85] K. Schwan, C. Gaymon,
"Automatic Resource Allocation for the Cm Multiprocessor"*,
Proc. of the 5' Int. Conf. on Distributed Computing Systems,
Denver, Colorado, 14-17 Mai 1985, 310-320.

[Seitz 85] C.L. Seitz,
"The Cosmic Cube",
Com. on the ACM, Vol.28, n°1, janvier 1985 - 22-33.

[Snyder 82] L. Snyder,
"Introduction to the Configurable HIghly Parallel Computer",
Computer, Vol.15, n°1, Janvier 1982, 47-56.

[Stankovic 84] J.A. Stankovic, I.S. Sidhu,
"An Adaptative Bidding Algorithm for Processes, Clusters, and Distributed Groups",
Proc. of the 4' Int. Conf. on Distributed Computing Systems,
San Francisco, California, 14-18 Mai 1984, 49-58.

[Swan 77] R.J. Swan, S.H. Fuller, D.P. Siewiorek,
"Cm A Modular Multiprocessor"*,
Proc. AFIPS 1977, NOC Vol. 46, 645-655.

[Wah 84] B.W. Wah,
"File Placement on Distributed Computer Systems",
Computer, Vol. 17, n°1, Janvier 1984, 23-32.

[Walker 85] P. Walker,
"The Transputer",
Byte, Mai 1985, 219-235.

[Ward 84] M.O. Ward, D.J. Romero,
"Assigning Parallel Executable Intercommunicating Subtasks to Processors",
Proc. of the Int. Conf. on Parallel Processing, Bellaire,
MI, 21-24 Août 1984, 392-394.

[Yalamanchi 85] S. Yalamanchi, J.K. Aggarwal,
"Reconfiguration Strategies for Parallel Architectures",
Computer, Vol. 18, n°12, Décembre 1985, 44-61.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique