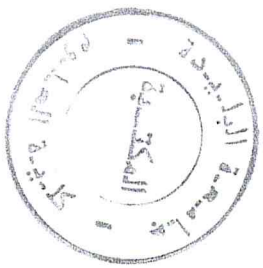
République Algérienne Démocratique et Populaire.
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique.

Université Saad Dahlab, Blida
USDB.

Faculté des sciences.
Département informatique.



Mémoire pour l'obtention
d'un diplôme d'ingénieur d'état en informatique.
Option : **S.I**

Sujet :

Développement d'un environnement de résolution du problème de partitionnement par l'approche des colonies de fourmis

Présenté par : Louzri Abderezak
Ouamane Farouk

Promoteur : M[r] M. KOUDIL

Organisme d'accueil : Université Saad Dahlab, Blida

- 2004/2005-

Dédicaces

À mon exceptionnel regrettés père dont j'aurai souhaité la présence en ce jour mémorable.

A la mémoire de mes chers et regrettés grands-parents dont j'aurai souhaité la présence en ce jour mémorable.

À ma très chère mère, mon inépuisable source d'affection, qu'elle trouve ici l'expression de tout mon amour et tendresse et de toute ma reconnaissance pour tout ce que je lui dois.

À mes très chères frères, sœurs, beaux-fréres et belles-soeurs

À mes cousins et cousines qui m'apportent une grande joie en les voyant. Prions le bon dieu que nous restions unis.

À mon cher ami et binôme FAROUK.

À mes très chères et exceptionnelles amies, Hamza, qui m'ont toujours aidé, et partagé mes joies et mes soucis.

À mes très chers amis, Hamza, Hishame, Yazid, ..., qui n'ont pas hésité à nous rendre service dans les moments les plus délicats.

À tous les étudiants & étudiantes de ma promotion.

À tous mes amis & amies qui ont rendu agréables mes études à BLIDA.

À tous mes amis et amies à l'université de SAAD DAHLEB DE BLIDA.

À mon très cher ami et frère, Nacer à l'institut nationale d'informatique (INI) dont j'aurais souhaité la présence en ce jour de soutenance.

Enfin, à tous ceux qui luttent pour la réussite, et ceux dont l'existence est bénéfique pour autrui, ...

Je dédie ce modeste travail

abderrezak

Dédicaces

A mon cher père pour son aide et sa confiance

A ma très chère mère qu'elle trouve ici l'expression de toute ma gratitude et de toute ma reconnaissance pour ses sacrifices

A ma très chère grand mère

A mes très chers frères

A mes très chères sœurs

A tout mes amis et amies

A mes cousins et cousines

A mon cher binôme abderezak

Je dédie ce modeste travail

Remerciements

Nous tenons à remercier très vivement notre promoteur, Mr Koudil Mouloud, pour leur entière disponibilité, leur simplicité, leur précieuse directive, leur rigueur scientifique et leur encouragement qui nous ont poussé à donner le meilleur de nous même.

Nous remercions par cette occasion, Mr hamza et nacer pour son aide précieuse. Nos vifs remerciements pour notre chers parents pour, leurs soutiens continus.

Nous tenons à souligner notre profonde gratitude à nos amis et amies, grâce à leurs efforts ont rendu possible l'achèvement de ce mémoire.

En fin, Nous remercions tous ceux qui nous ont aidé ou ont l'intention de nous aider.

Résumé

En intelligence artificielle et en théorie de complexité, le problème de PARTITIONNEMENT est un grande importance, ils jouent un rôle historique et théorique très important.

Du point de vue Informatique, la résolution de ce problème en cherchant la solution optimale est prohibitive en temps de calcul. Pour cela, le recours aux méthodes pseudo-aléatoires (**La méthode ascendante, Recuit simulé et les Algorithmes génétiques)** et la méthode de recherche guidée (**Recherche tabou)** est nécessaire.

Notre objectif est la simulation d'un environnement parallèle implantant la méthode d' ANT COLONY pour la résolution de problème (Partitionnement).

Mots Clés : Partitionnement Matériel/Logiciel, CO-DESIGN, ANT COLONY Recherche Tabou, Recuit Simulé, Algorithmes Génétiques, La méthode Ascendante et les architectures séquentiel.

SOMMAIRE

LISTE DES FIGURES

Liste des tableaux

Introduction Générale

Introduction Générale

Depuis l'avènement de la technologie Informatique, l'optimisation occupe une place de plus en plus grande dans le monde scientifique. Un grand nombre de théoriciens informaticiens, pensent qu'il existe un nombre important de problèmes intraitables en pratique. En raison de leur importance, un moyen de résolution devait être trouvé. La solution proposée préconise l'utilisation de l'optimisation, c'est à dire, la recherche de la solution qui se rapproche le plus possible de la solution optimale et qui soit calculable en un temps raisonnable (Polynomial) **[OUM 00]**.

Il existe dans la littérature, plusieurs problèmes appartenant à la classe NP-complet [ALL94], c'est à dire, qui sont difficiles à résoudre en pratique en raison de leur temps de résolution qui évolue exponentiellement. Parmi ces problèmes, on cite, le problème de Satisfiabilité d'une formule booléenne [HAN90], le problème du Partitionnement Matériel/Logiciel [ROU95], le problème d'Ordonnancement[CHE88], le problème d'Affectation Quadratique (QAP) [TAI90], le problème du Voyageur du Commerce (PVC) [CER85],... . Dans notre travail, nous nous intéressons au Partitionnement Matériel/Logiciel.

Le problème de Partitionnement Matériel/Logiciel, est une étape très importante de l'activité Co-Design (Concurrent Design), appelé aussi "La Conception Conjointe". Son rôle est la transformation de la spécification de la solution d'un problème en un système mixte, composé d'une partie matérielle et d'une autre partie logicielle. C'est un problème de complexité importante, qui augmente en fonction du nombre de processeurs et d'entités constituant le système à étudier [MCSE].

Etant donné que le problème de Partitionnement est NP-complets, seule des Heuristiques permettent de le résoudre en un temps raisonnable. Cependant, une heuristique, introduit des mécanismes spécifiques au problème. Ce qui est un inconvénient majeur à sa généralisation à tous les cas de problèmes que nous pouvons trouver dans la recherches [AMR95]. Pour aboutir à une résolution plus générale de complexité raisonnable, le recours aux méthodes d'optimisation à caractère général applicables à une grande variété de problèmes tels que : Le Recuit Simulé [KIR83], Les algorithmes Génétiques [GOL94] et la Recherche Tabou [GLO77] se révèle intéressant. Parmi ces méthodes d'optimisation, figure la méthodes de la recherche Tabou. Elle est présentée par Glover [GLO77], c'est une technique itérative qui évolue étape par étape, d'une solution initiale d'un problème d'optimisation combinatoire à une autre solution qui donne une bonne valeur à la fonction objectif. Elle dispose d'une structure très importante appelée liste Tabou, qui sert à stocker les solutions déjà visitées. Cela pour éviter de tomber sur le problème de cycle (Retour à une solution déjà visité). La méthode de la recherche Tabou est avérée en pratique, une méthode efficace pour la résolution d'un grand nombre de problèmes [TAL98.b]. Cependant, les résultats de résolution des problèmes NP-complets restent ouverts

aux éventuelles améliorations. Pour cela, l'idée d'hybrider cette méthode avec les autres méthodes (ayant des paramètres ou des principes différents et des propriétés complémentaires) dans le but de marier leurs avantages et de réduire leurs inconvénients, soit les Heuristiques spécifiques (leur nature dépend du problème à traiter) ou générales (le Recuit Simulé et les Algorithmes Génétiques) peut apporter des améliorations à la qualité de la meilleure solution. Par conséquent, pour résoudre les problèmes Max-Sat et Parritionnement, nous allons implémenter cette idée d'hybridation et de coopération en utilisant les méthodes générales de recherche, la Recherche Tabou, le Recuit Simulé et les Algorithmes Génétiques.

Pour implémenter cette idée de résolution, nous avons besoin d'une machine parallèle où chaque méthode sera exécutée sur un processeur donné, tout en garantissant l'interaction et l'échange des données entre ces méthodes. Comme nous n'avons pas à notre disposition une machine parallèle réelle, nous avons conçu une machine parallèle virtuelle appelée PARE (**PAR**titioning & **E**nvironment). C'est une plate forme qui met à la disposition de l'utilisateur différents choix dans la spécification des paramètres des méthodes et des problèmes à traiter. Pour le cas du Partitionnement par exemple, nous avons laissé la possibilité à l'utilisateur de choisir la fonction objectif qui sera analysée par un analyseur lexical et Syntaxico-Semantique. Cette machine, permet aussi l'interaction avec l'utilisateur au cours d'exécution (visualisation de l'évolution d'exécution en temps réel, orientation de la recherche par l'utilisateur en modifiant les valeurs des paramètres des méthodes, changement de l'état d'un ou plusieurs processeurs,... .).

Ce mémoire est composé essentiellement de six chapitres: Dans le premier, nous allons définir, le Domaine d'application de codesign, ses différentes étapes, on a définit Le partitionnement matériel/logiciel suivi de quelques approches et quelques méthodes de résolution.
Dans le second chapitre, on a citer Les problèmes d'optimisation tel que : Les méthodes exactes et les méthodes approchées avec des Exemples de méthodes de résolution des problèmes d'optimisation. Le troisième chapitre décrit le principe de la méthode de colonie de fourmis, les caractéristiques de son algorithme, l'algorithme général avec un exemple d'application de la colonie de fourmis sur le problème de VC. Le chapitre quatre, est consacré à la présentation des généralités sur les machines parallèles en décrivant la structure générale des environnements, nous avons présentons l'architecture de la machine parallèle PARE. Le chapitre cinq, concerne La colonie de fourmis pour le problème de partitionnement, la conception et la mise en œuvre de l'environnement. Enfin dans le sixième chapitre nous présentons quelques testes et résultats.

CHAPITRE 1 : La conception conjointe matériel/logiciel (codesign)

1. Introduction :

Le terme ***hardware/software concurrent design*** ou ***HW/SW co-design*** traduit par conception conjointe matériel/logiciel représente un processus de conception très important. Il permet aux concepteurs de transformer les spécifications d'un système en un produit industriel. D'après Wolf [WOL94] le codesign est défini comme suit :

"Le matériel et le logiciel doivent être conçus ensemble pour être surs que l'implémentation non seulement fonctionne, mais répond également aux objectifs de performance, prix et flexibilité ».

Au début, la conception conjointe matériel / logiciel consistait à traiter la partie matériel du système avant la partie logiciel. Mais au cours de ces dernières années, les techniques de développement des circuits intégrés ont suivi une évolution très importante. Parallèlement, l'amélioration des architectures matérielles a pu être constatée, concernant en particulier celles qui contiennent des processeurs et des circuits programmables. De plus, un système est formé par un ensemble de circuits matériels et logiciels (programme exécutable).

Une nouvelle approche de conception qui traite simultanément la partie matérielle et la partie logicielle. Cette méthode doit permettre aux concepteurs de faire le découpage d'un système le plus tard possible, et de manière automatique, afin de trouver le meilleur compromis entre une réalisation logicielle est une réalisation matérielle.

2. Domaine d'application de codesign :

Le codesign peut s'appliquer à tout système mixte contenant du matériel et du logiciel. En fait, il est de plus en plus employé pour la réalisation de systèmes mixtes. Ainsi, plusieurs exemples de codesign matériel/logiciel ont été publiés durant ces dernières années. Parmi les domaines les plus importants figurent les suivants :

2.1. Les systèmes embarqués :

Ce sont des systèmes digitaux qui représentent un sous système dans un système global et qui offrent un service spécifique à ce système. La Figure I.1 donne le schéma d'un système embarqué [STO 95, ADA 96, DEM 97].

Chapitre I

La conception conjointe matériel/logiciel (codesign)

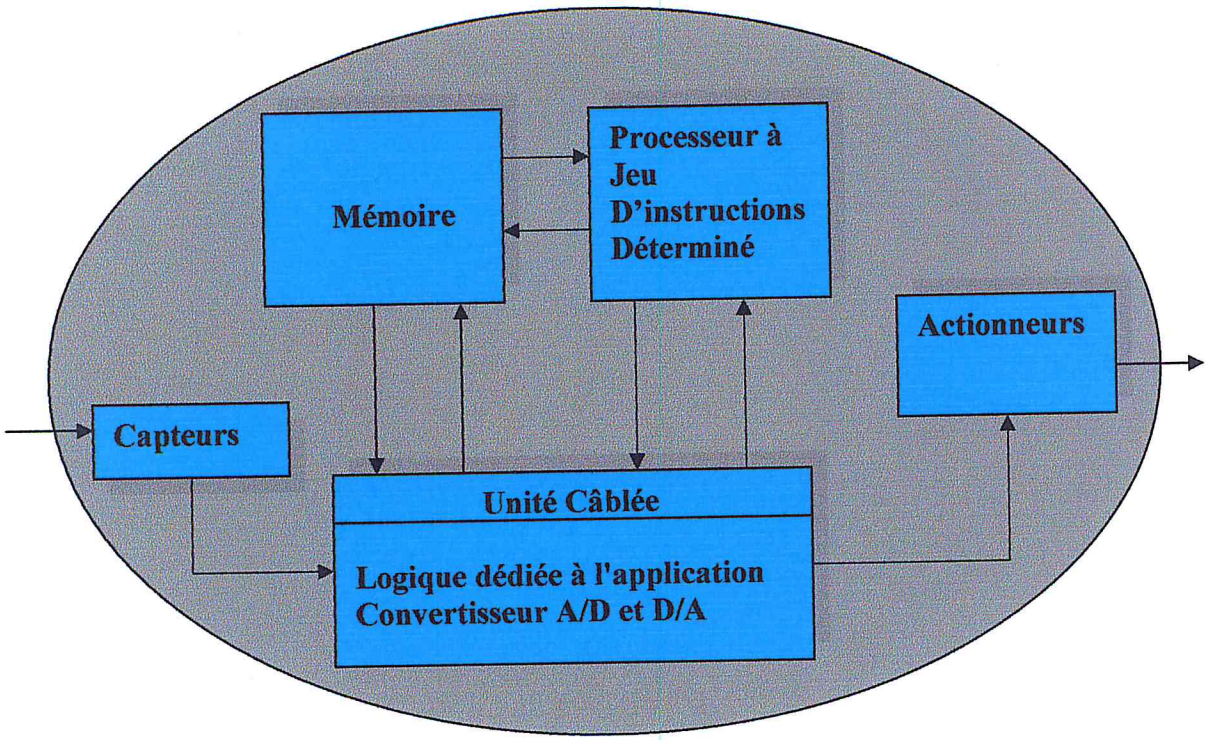


Figure I.1 : Schéma d'un système de contrôle embarqué [DEM 97]

2.2. Les Télécommunications :

C'est l'un des marchés le plus en expansion actuellement, notamment par le développement des télécommunications sans fil et d'internent (nouvelles générations de téléphones mobiles, modems...). C'est l'un des domaines les plus actifs du codesign, car il nécessite la production à faible coût de systèmes à contraintes strictes dans un environnement de grande concurrence, où un temps de mise sur le marché court est absolument crucial. Les produits sont nécessairement fournis en de nombreuses versions différentes mises à jour continuellement. Il est, par conséquent, important que les composants existants puissent être réutilisées aussi souvent que possible, d'où le besoin en produits flexibles conçus à faible coût [STO 95].

2.3. L'accélération d'algorithmes logiciels :

Une portion critique dans un logiciel est une partie de l'application logicielle qui améliorerait la performance globale du système si elle était mise en œuvre en matériel [EDW 97]. La partie matérielle a donc pour but d'améliorer les performances des portions critiques d'un algorithme réalisant l'application désirée. Nous assistons, par exemple, actuellement à l'émergence d'une pléiade de nouveaux systèmes graphiques intégrés aux microordinateurs (cartes de traitement graphique tridimensionnel, coprocesseurs graphiques intégrés...). [KOU02].

2.4. Les systèmes de traitement de signaux digitaux [KOU02] :

Le développement de systèmes utilisant des processeurs de traitement de signaux digitaux constituent une application directe des deux classes précédentes d'architectures et sont un exemple de conception par codesign. Les systèmes conçus dans ce domaine effectuent des opérations de calcul intensif, souvent sous des contraintes de temps critiques. Le matériel est utilisé pour les calculs critiques en temps d'exécution et pour les communications externes, alors que le logiciel se charge des autres processus [STO 95].

3. Pourquoi le codesign ?

La méthode de Co-design s'applique a tout système qui contient a la fois une partie logicielle et une partie matérielle. La Figure I.2 montre que 98% des processeurs vendus en 1998 équipent des systèmes embarqués (enfouis).

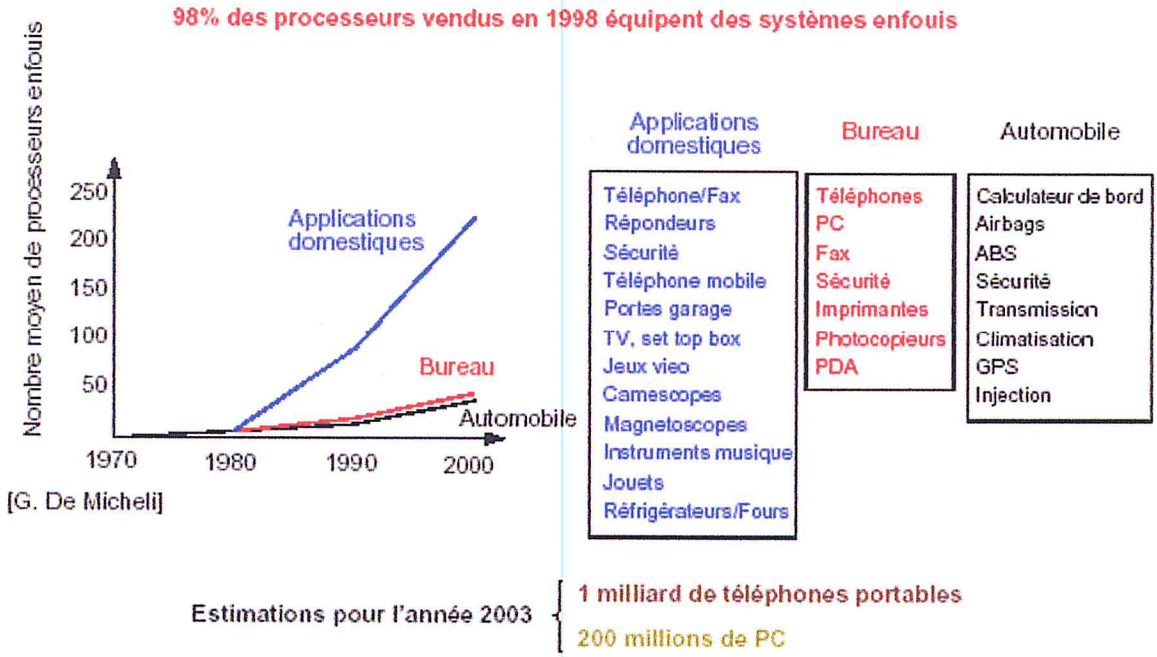


Figure I.2 : domaines d'application de co-design [MIC03]

4. Les différentes étape du co-design :

Le processus du co-design est ses différentes étapes sont illustrés en **Figure I.3**.

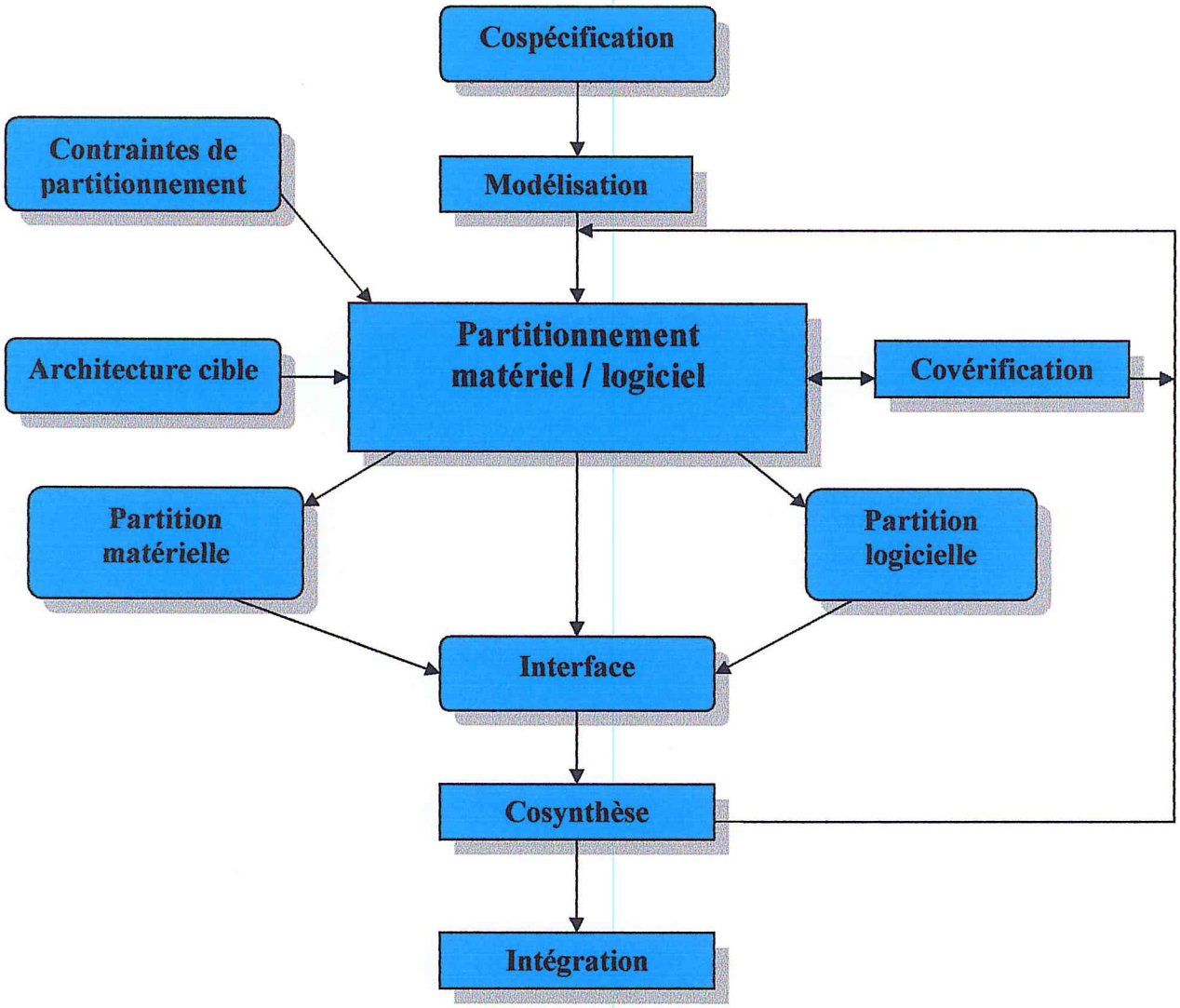


Figure I.3 : Différentes étapes du processus de codesign. [KOU02]

4.1. La co-spécification :

Elle concerne l'étape de spécification du système globalement et de description fonctionnelle. Spécifier le système global revient à donner une représentation abstraite de la solution. L'objectif de l'étape de co-spécification est d'élaborer une description externe la plus complète possible du système à concevoir, et ceci à partir du cahier des charges (expression des besoins).

4.2. Le partitionnement matériel et logiciel :

C'est la décomposition de la solution fonctionnelle d'entrée en une partie logicielle et une partie matérielle compte tenu des performances et des contraintes de temps à satisfaire. Cette étape consiste à déterminer les parties matérielles et logicielles du système en fonction des contraintes imposées et des performances souhaitées.

4.3. La co-simulation :

La co-simulation est une simulation de la description mixte matérielle et logicielle résultant d'un partitionnement. Elle est utilisée pour observer le comportement du système complet, et vérifier un certain nombre de propriétés. Elle permet également d'extraire, avec plus de précision, certaines caractéristiques du système (temps d'exécution, communication, espace occupé…).

4.4. La co-synthèse :

C'est l'étape qui s'occupe de la réalisation du système mixte en transformant les descriptions fonctionnelles en descriptions directement implantables sur les différents composants de l'architecture cible.

5. Le partitionnement matériel/logiciel :

A la fin de l'étape de spécification du système (co-spécification), le processus du co-design active une étape très importante: c'est l'étape de partitionnement matériel et logiciel. Après le choix de l'architecture cible, l'étape du partitionnement donne les parties du système qui doivent être mises en œuvre en logiciel est celles qui doivent être mises en oeuvre en matériel de telle sorte que le produit final (système global) réponde aux contraintes imposés et de performances souhaitées [KUM96].

Comme l'illustre la **Figure I.4**, le partitionnement matériel/logiciel est le problème de déterminer une implémentation pour chaque nœud de telle sorte que le système global soit optimisé [KOU02].

Vahid [VAH 94] propose la définition suivante pour le partitionnement : soit un ensemble d'objets fonctionnels $O = \{o_1, o_2, ..., o_n\}$ qui composent la fonctionnalité d'un système à concevoir. Les partitions matériel/logiciel obtenues après partitionnement sont définies par deux sous ensembles H (matériel) et S (logiciel) tel que :

$$H \subset O,\ S \subset O,\ H \cup S = O,\ \text{et}\ H \cap S = \Phi$$

Une réalisation est donnée à tous les objets fonctionnels du système et un même objet ne peut être réalisé à la fois en matériel et en logiciel. Le concepteur doit donc trouver deux ensembles d'objets répondant aux conditions précédentes. Son expertise est donc très importante, du fait qu'il n'existe pas de règle absolue pour déterminer à priori si une entité doit être mise en œuvre en matériel ou en logiciel. Il est, par exemple, usuel d'utiliser le matériel pour accélérer l'exécution des parties contenant des calculs intensifs, alors que le logiciel est plus adapté aux parties nécessitant plus de flexibilité de mise en œuvre, et sujettes à des modifications fréquentes [KOU02].

De plus, le partitionnement assure la transformation des spécifications de la partie du système générée à l'étape de Co-spécification en une architecture qui se compose d'une partie matérielle et une partie logicielle.

5.2. Les approches de partitionnement :

Le processus de partitionnement est une étape très importante dans le processus de conception de HW/SW. Deux type d'approches on été développés pour résoudre le problème de partitionnement

- **L'approche manuelle :** les premières approches développées sont essentiellement des approches manuelles. Elles sont basées sur un partitionnement effectué par le concepteur. Les arguments des adeptes de cette approche sont que le concepteur, qui possède une grande connaissance de son système, est le plus apte à apporter les optimisations nécessaires [KOU 02].
- **L'approche automatique :** ces dernières années les systèmes mixtes deviennent très complexes. Pour cela, il devient donc plus en plus difficile d'appliquer des techniques de partitionnement manuel, spécialement lorsque le grain est fin. Il est donc nécessaire de recourir à des méthodes automatiques de partitionnement.

5.3. Méthodes de résolutions de partitionnement :

On peut présenter le problème de partitionnement comme un problème NP complet dépendant d'un certain grand de paramètres (prise en compte d'un nombre limité de critères).

Dans ce qui suit, nous décrivons trois des principales approches utilisées pour la résolution du partitionnement en codesign.

5.3.1. Algorithme glouton (Greedy algorithm) :

Cette technique est utilisée dans VULCAN [GUP95] pour lequel toutes les fonctions sont initialement implantées en matériel et sont migrées vers le processeur logiciel en vérifiant les contraintes temporelles. La fonction de coût utilisée est pondérée par la surface du matériel, la taille du programme de code (mémoire) et le taux d'occupation du processeur [MCSE00].

5.3.2. Approche de COSYMA :

Opposée à celle utilisée dans VULCAN. Les fonctions sont au départ toutes implantées en logiciel puis migrées vers le matériel jusqu'au respect des contraintes de performances [MCSE00].

5.3.3. Partitionnement interactif [ISM94] :

Ismail propose une technique de partitionnement interactive (PARTIF) permettant de cibler une architecture hétérogène. Le concepteur peut explorer plusieurs alternatives de partitionnement du système en manipulant une hiérarchie d'automates à états finis. Un ensemble de primitives de transformations d'états (déplacement, regroupement, décomposition,...) est disponible.

Plusieurs critères peuvent intervenir sur le double choix (architecture et allocation) d'un partitionnement tels que par exemple [MCSE00] :

- Les performances statiques (consommation, surface, coût, taille de code du la mémoire etc...). et dynamiques (contraintes de temps, débit, etc....).
- La flexibilité : l'implantation logicielle d'une fonction offre des possibilités d'évolution plus importantes qu'une implantation matérielle...

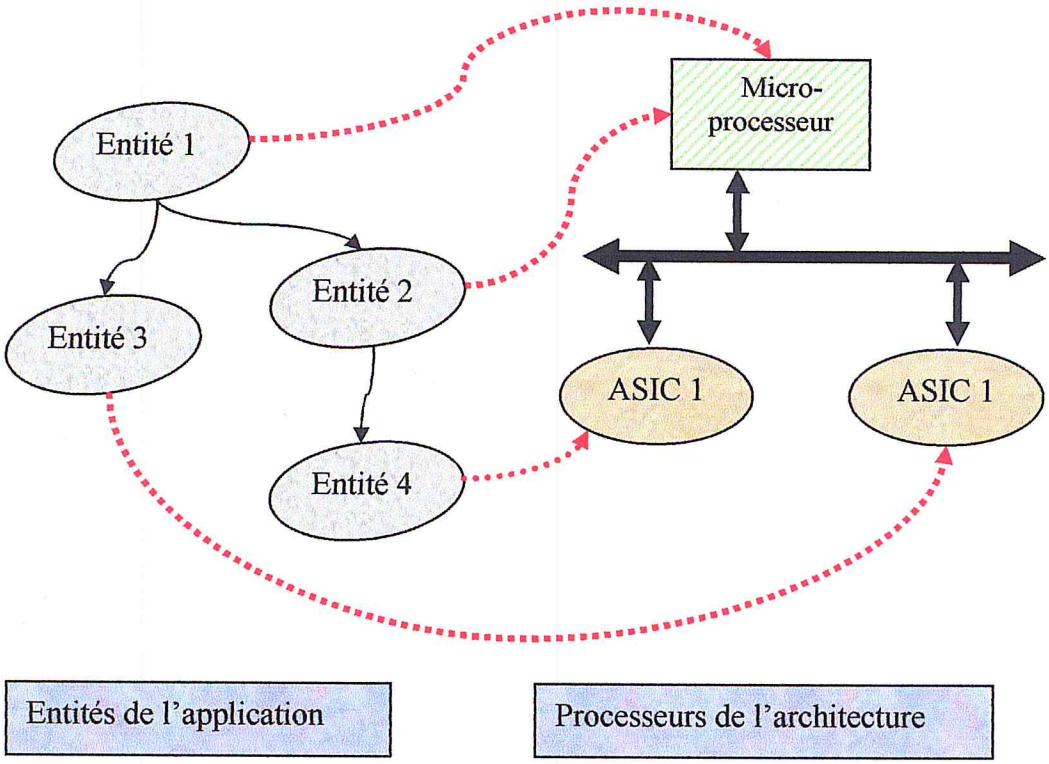


Figure I.4 : Principe du partitionnement. [KOU02]

5.1. Les niveaux de granularité :

Pour les spécifications d'entrée d'un partitionnement, trois niveaux de granularité (appelée aussi finesse de description) du partitionnement sont habituellement utilisés :

- Le niveaux gros-Grain (niveaux objet ou tache).
- Le niveau grain intermédiaire (processus, procédures ou fonctions).
- Le niveau grain fin (instruction, opération arithmétique, circuit élémentaire).

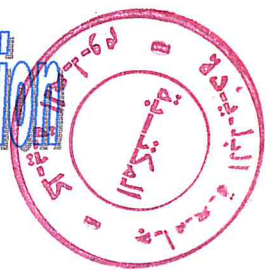
Pour le niveau tâche (coarse-grain partitioning), l'unité d'allocation est la fonction qui est considérée indivisible et dont le comportement n'est pas obligatoirement séquentiel. Pour le niveau intermédiaire, une fonction est décomposée en un ensemble de séquences d'instructions appelées procédures et qui peuvent être allouées sur des processeurs différents. Pour le niveau instruction (fine-grain partitioning), l'unité d'allocation est la plus petite possible puisqu'il s'agit d'une instruction. L'utilisation d'un niveau de granularité fine concerne plutôt des systèmes de faible complexité.

6. Conclusion :

Le développement des circuits intègres et la complexité croissante des systèmes mixtes ont poussé ces dernières années les chercheurs à travailler sur l'étape de partitionnement et essayer de trouver des solutions automatiques. Jusqu'à présent, aucune méthode n'a donné des résultats optimaux pour tous les type d'applications, et il apparaît qu'aucune méthode n'est véritablement meilleure que les autres pour tous les types de systèmes.

Chapitre II

Les problèmes d'optimisation



CHAPITRE 2 : Les problèmes d'optimisation

1. Introduction :

L'optimisation combinatoire occupe une place très importante en recherche opérationnelle et en Informatique. Son importance est justifiée, d'une part, par la grande difficulté du problème d'optimisation [PAP 82], et d'autre part, par les nombreuses applications pratiques qui peuvent être formulées sous la forme d'un problème d'optimisation combinatoire (max sat, partitionnement, voyageur de commerce…) [RIB 94].

La plupart de ces problèmes appartiennent à la classe des problèmes NP-complets. La résolution de tels problèmes est théoriquement facile (balayage de toutes les solutions possibles), mais il n'existe aucun algorithme permettant de trouver à coup sûr la solution optimale en un temps raisonnable.

Etant donnée l'importance de ces problèmes, de nombreuses méthodes de résolution ont été développées en recherche opérationnelle (RO) et en intelligence artificielle (IA).

Ces méthodes peuvent être classées en deux grandes catégories :

- les méthodes exactes qui garantissent de trouver la solution optimale;
- les méthodes approchées qui ne peuvent garantir l'optimalité de la solution.

2. Les méthodes exactes et les méthodes approchées :

2.1. Les méthodes exactes :

Le principe essentiel d'une méthode exacte consiste généralement à énumérer l'ensemble de l'espace de recherche. Pour améliorer l'énumération des solutions, une telle méthode dispose de technique pour détecter le plus tôt possible les échecs, et d'heuristiques spécifiques pour orienter les différents choix.

Les méthodes exactes ont permis de trouver des solutions optimales pour des problèmes de taille raisonnable. Les méthodes exactes rencontrent généralement des difficultés face aux applications de taille importante. Parmi les méthodes exactes existantes, citons le branch and bound.

2.1.1. La méthode de branch and bound :

Cette méthode consiste à trouver la solution optimale en évitant l'énumération systématique de toutes les solutions possibles en élaguant les branches jugées impraticables.

2.2. Les méthodes approchées:

Les méthodes approchées sont conçues pour traiter les problèmes d'optimisation de grande taille.

Depuis une dizaine d'années, des progrès importants ont été réalisés avec l'apparition d'une nouvelle génération de méthodes approchées puissantes et générales, souvent appelées Méta-heuristique [REE 93].

Les méta heuristiques sont représentées essentiellement par:

- les méthodes de voisinage comme : recuit simulé, recherche tabou, ACO;
- les algorithmes évolutifs comme : algorithmes génétiques…

3. Exemples de méthodes de résolution des problèmes d'optimisation:

3.1. La recherche locale :

La recherche locale, appelée aussi la descente ou l'amélioration itérative, représente une classe de méthodes heuristiques très anciennes [FLO 56]. La recherche locale est utilisée en général pour résoudre des problèmes réputés très difficiles tels que le voyageur de commerce.

3.1.1. Définition :

Soit X l'ensemble des configurations admissibles d'un problème, on appelle *voisinage* toute application $N : X \rightarrow 2^X$. On appelle *mécanisme d'exploration* du voisinage toute procédure qui précise comment la recherche passe d'une configuration $s \in X$ à une configuration $s' \in N(s)$. Une configuration s est un *optimum (minimum) local* par rapport au voisinage N si $f(s) \leq f(s')$ pour toute configuration $s' \in N(s)$.

1). Débuter avec une configuration quelconque s de X.

2). Choisir un voisin s' de s tel que $f(s') < f(s)$ et remplacer s par s'

Répéter 2) Jusqu'à ce que pour tout voisin s' de s réponde à $f(s') \geq f(s)$.

Cette procédure fait intervenir à chaque itération le choix d'un voisin qui améliore la configuration courante à l'aide de la fonction objectif.

3.1.2. L'algorithme général de la recherche locale :

L'espace de recherche est constitué de toutes les assignations possibles. U (pour le problème de partitionnement U représente les différentes solutions possibles). La recherche locale la plus simple commence par une configuration initiale aléatoire $U^{(0)} \in U$. Il s'agit alors de générer une trajectoire de recherche comme suit :

V= Meilleur_Voisin ($N(U^{(t)})$)

$$U^{(t+1)} = \begin{cases} V \text{ si } f(V) > f(U^{(t)}) \\ U^{(t)} \text{ si } f(V) \leq f(U^{(t)}) \end{cases}$$

Où :

$N(U^{(t)})$ Le voisinage de la solution à l'itération t.

La fonction meilleur_voisin sélectionne $V \in N(U^{(t)})$ avec la meilleure valeur de la fonction de coût *f*, la recherche locale s'arrêt quand aucune amélioration n'est appliquée a f.

Une amélioration de la recherche locale a été définie ou une nombre spécifique d'itération est exécuté et V est accepter même si elle n'apporte pas d'amélioration a f, l'algorithme est donnes comme suit :

Début:

Répéter

V = Meilleur_Voisin ($N(U^{(t)})$) ;

$U^{(t+1)} = V$;

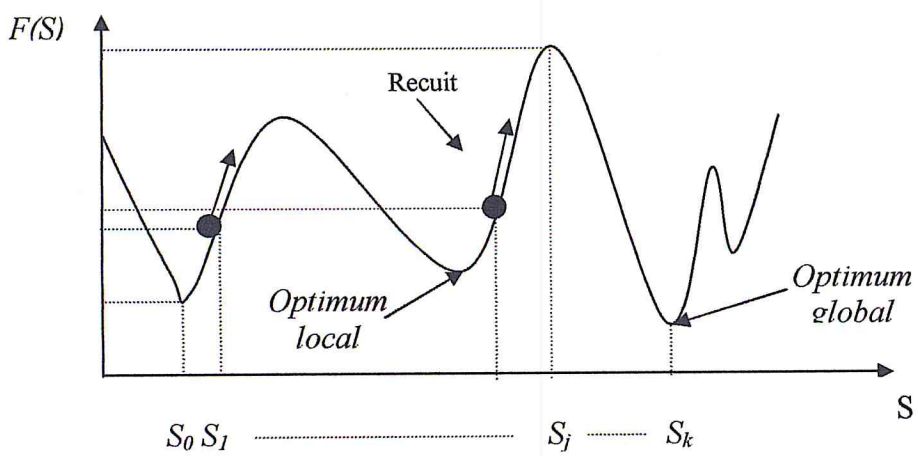
Jusqu'à MaxIter ;

Fin ;

MaxIter : nombre d'itération fixe par l'utilisateur.

3.2. Le recuit simulé (simulated annealing) :

C'est une méthode d'optimisation basée sur l'analogie faite avec le procédé du recuit physique utilisé en métallurgie. Elle consiste à chauffer un matériau à haute température. La phase de refroidissement conduit la matière liquide à retrouver sa forme solide par une diminution progressive de la température. Chaque température est maintenue jusqu'à ce que la matière trouve un équilibre thermodynamique. La Figure II.1 montre le parcours de l'espace de recherche (cas de minimisation) par la méthode de Recuit simulé.



L'algorithme de recuit simulé est donné dans ce qui suit [OUM 00] :

Algorithme RS (T /* Température initiale */)
Début /* Cas de maximisation */
T←T * K /* K : constante de Boltzmann */
/* T : La température */
Générer une solution initiale S_i.
Calculer $F(S_i)$ /* F : La fonction objectif de S_i */
Répéter
Répéter
Générer le voisinage $V(S_i)$.
$S_n \leftarrow S$ /*S est sélectionnée aléatoirement dans V(Si) */
Si $F(S_n) - F(S_i) \geq 0$ **alors** $S_i \leftarrow S_n$
SiNon
Générer Nbr aléatoirement dans]0,1[.
Si Nbr < $e^{(F(Sn)-F(Si))/T)}$ **alors** $S_i \leftarrow S_n$.
Fsinon
Jusqu'à atteindre le nombre d'itérations spécifié.
Générer un réel α aléatoirement dans [0,1].
T← T * α
Jusqu'à atteindre la borne inférieure de la température..
Fin.

3.3. Les algorithmes génétiques :

Les algorithmes génétiques permettent essentiellement de résoudre des problèmes d'optimisation engendrant une suite de solutions approchées et s'améliorant progressivement en simulant l'évolution darwinienne. Dans cette évolution les systèmes biologiques survivent et s'adaptent par des processus naturels tel que:

- La reproduction.
- Le croissement.
- La mutation.

3.3.1. Les opérateurs génétiques :

Sélection:

L'algorithme attribue à chaque individu une chance de participer à la phase reproduction, qui est proportionnelle au rapport de sa qualité d'évaluation par la somme des valeurs de la fonction d'évaluation des individus de la population. Ainsi l'operateur de selection choisit un individu i avec une probabilité :

$$P = F(\text{individu i}) / \sum F(\text{individu j})$$

Avec j variant de 1 à Pop

F(individu i): représente la valeur de la fonction d'evaluation de l'individu i.

Pop :la taille de la population.

Croisement :

Le croissement ou crossover est un opérateur binaire qui s'applique à deux individus parents choisis au hasard dans la population d'individus déjà selectionnés. Il consiste en un échange de gènes entre ces individus parents produisant ainsi deux nouveaux individus.

Mutation :

La mutation est appliquée après le croisement. Elle réalise la modification aléatoire avec une probabilité Pm généralement faible de la valeur d'un ou plusieurs gènes d'un individu.

Remplacement :

On a vu que l'application des opérateurs génétiques (croissement, mutation) permettent d'engendrer une nouvelle population appellée population d'enfants. Les meilleurs individus de cette derniere vont remplacer les plus mauvais de la population des parents.

3.3.2. Principe des algorithmes génétiques :

```
Début
  Générer la population initiale ;
  Tantque (le nombre de generation stable n'est pas atteint) et
          (le nombre de generation total n'est pas atteint) et
          (solution optimale non trouvée) faire
      Pour i :=1 à (taille – Pop) faire
          selectionner deux individus ;
          choisir un  nombre aleatoire entier  Rc dans l'intervalle[0,100] ;
          Si  Rc<taux de croissement  alors appliquer le croissement ;
              choisir entier aleatoire r dans l'intervalle [0,100] ;
              Tantque r<taux de mutation Faire
                  Choisir un gene aleatoire sur 1individu obtenu après
                      croissement et l'inverser ;
                  Choisir un nombre aleatoire entier r dans l'intervalle [0,100] ;
              Fait
              evaluer l'individu produit ;
      Fait
      Remplacer les individus produits en favorisant les meilleurs ;
  Fait
Fin
```

3.4. La recherche tabou (RT) :

La méthode tabou a été développée par Glover [GLO 86]. Cette méthode fait appel à un ensemble de règles et de mécanismes généraux pour guider la recherche de manière intelligente au travers de l'espace des solutions.

3.4.1. Principe :

A l'inverse du recuit simulé qui génère de manière aléatoire une seule solution voisine $s' \in N(s)$ à chaque itération, la recherche tabou examine un échantillonnage de solutions de $N(s)$ et retient la meilleure s' qui améliore la fonction objectif. La recherche tabou ne s'arrête donc pas au premier optimum trouvé.

Cette stratégie peut entraîner des cycles, par exemple un cycle de longueur 2 :

$s \rightarrow s' \rightarrow s \rightarrow s'$... Pour empêcher ce type de cycle, on mémorise les k dernières configurations visitées dans une mémoire à court terme et on interdit tout mouvement qui conduit à l'une de ces configurations. Cette mémoire est appelée la liste tabou.

3.4.2. La stratégie à court terme :

Elle garde la trace des attributs des solutions qui ont changé durant le passé récent, c'est-à-dire les solutions récemment visitées. Elle interdit les solutions déjà visitées en utilisant la liste tabou. Les solutions tabou ne doivent pas rester plus d'un certain temps dans la liste tabou.

3.4.3. La stratégie a long et moyen terme :

Il existe d'autres techniques intéressantes pour améliorer la puissance de la méthode tabou; en particulier, l'intensification et la diversification. Toutes les deux se basent sur l'utilisation d'une mémoire à long terme et se différencient selon la façon d'exploiter les informations de cette mémoire.

Une mémoire a moyen terme intermédiaire est utilisée quelques fois pour intensifier la recherche dans une zone de recherche.

L'intensification : Son principe est de stocker et comparer les caractéristiques d'un nombre sélectionné des meilleures solutions durant une période particulière de recherche.

La diversification : La diversification a un objectif inverse de l'intensification : elle cherche à diriger la recherche vers des zones inexplorées. Sa mise en oeuvre consiste souvent à modifier temporairement la fonction de coût pour favoriser des mouvements n'ayant pas été effectués ou à pénaliser les mouvements ayant été souvent répétés. Une mémoire a long terme est utilisée pour étendre l'espace de recherche en atteignent d'autre régions non encore visitées.

L'intensification et la diversification jouent donc un rôle complémentaire.

3.4.4. L'algorithme général de la méthode de la recherche Tabou est [OUM 00] :

1. *Générer une solution initiale S_0 aléatoirement ou à l'aide* d'un algorithme spécifique. puis l'insérer dans la liste tabou. $S_{max} \leftarrow S_0$.
2. *Générer le voisinage de la solution en cours en appliquant une des stratégies de génération de voisinage.*
3. *Sélectionner une solution S dans ce voisinage en utilisant une stratégie de sélection. ($S \in N(S)$), telle que S ne figure pas dans la liste tabou.*
4. ***si*** *$F(S)>F(S_{max})$* ***alors*** *$S_{max} \leftarrow S$ /* Cas de maximisation/*
5. *Mise à jours de la liste tabou.*
6. ***Si*** *critère d'arrêt vérifié aller à 7*
   Sinon *aller à 2.*
7. *Arrêt.*

4. Conclusion :

Nous avons présenté dans ce chapitre quelques méthodes de résolution telles que : la méthode exacte (branch and bound), la recherche locale (RL), le recuit simulé (RS), les algorithmes génétiques (AG) et la recherche tabou (RT).

Ces méthodes sont adaptables à un très grand nombre de problèmes. Elles se sont révélées efficaces pour un grand nombre de problèmes d'optimisation classiques et d'applications réelles de grande taille.

Dans le prochain chapitre, nous introduisons la méthode utilisée dans notre travail: les colonies de fourmis (ACO).

Chapitre III

Les colonies de fourmis

CHAPITRE 3 : Les colonies de fourmis

1. Introduction :

Plusieurs méthodes ont été proposées pour résoudre les problèmes d'optimisation combinatoire à espace d'état discret tel que problème du voyageur de commerce, le routage dans les réseaux, le problème d'affectation quadratique... Cependant, le principal inconvénient de ces méthodes est leur temps calcul de calcul prohibitif. Des heuristiques ont été développées pour résoudre le problème de temps tel que le recuit simulé, la recherche tabou ou les algorithmes génétiques. Une nouvelle méthode a été introduite en 1996 par Marco Dorigo, appelée "colonie de fourmis". Cette technique offre un haut degré de flexibilité (la colonie s'adapte aux brusques de changement d'environnement) et robuste (la colonie continue à fonctionner lorsque certains individus échouent à accomplir leur tâche).

Les éthologistes ont montre que les fourmis étaient capables de sélectionner le plus court chemin pour aller du nid vers la source de nourriture, grâce au dépôt et au suivi des pistes de phéromone (qui est une matière hormonale).

Les applications actuelles des algorithmes d'ACO s'appliquent tombent à deux importantes classes de problèmes d'optimisation combinatoires: statiques et dynamiques.

2. Problèmes statiques :

La topologie et le coût ne changent pas durant la résolution des problèmes. Par exemple, pour le TSP classique, les emplacements des villes et les distances interurbaines ne changent pas pendant l'exécution de l'algorithme.

3. Problèmes dynamiques:

La topologie et les coûts peuvent changer pendant la résolution. Un exemple de tel problème est le routage dans les réseaux de télécommunications ou le problème de partitionnement.

Les algorithmes d'ACO pour résoudre ces dernières deux classes des problèmes sont très semblables d'une perspective à niveau élevé, mais elles diffèrent de manière significative dans des détails de mise en place. Avant d'introduire la méta heuristique colonie de fourmis, nous exposons l'expérience suivante :

Les fourmis se déplacent "en ligne", suivant le chemin des fourmis précédentes. Si on place un obstacle sur cette ligne de manière à ce que le contournement de l'obstacle par un côté soit plus court

que par l'autre, au bout d'un certain temps toutes les fourmis contourneront l'obstacle par le côté le plus court (cet exemple est nommé le pont à deux branches)..

4. Comportement des fourmis réelles [ALE02] :

Les fourmis déposent en marchant des marqueurs chimiques appelés phéromones. Les autres fourmis suivent le chemin tracé par ces phéromones. Plus un chemin est marqué, plus elles ont tendance à le suivre. Lorsque l'on pose l'obstacle, le chemin de phéromone est coupé, et les fourmis choisissent au hasard l'un ou l'autre côté pour contourner l'obstacle.

Plus un chemin est plus court, plus il y aura de fourmis qui franchiront l'obstacle en passant par ce côté. Sur 6 fourmis, 3 fourmis pourront être passées par le côté court alors que les 3 fourmis étant passées par le côté long n'auront pas encore fini de contourner l'obstacle. Une fourmi arrivant après les précédentes, en sens inverse, verra donc plus de phéromone sur le chemin qui passe par le côté court, et prendra donc ce chemin.

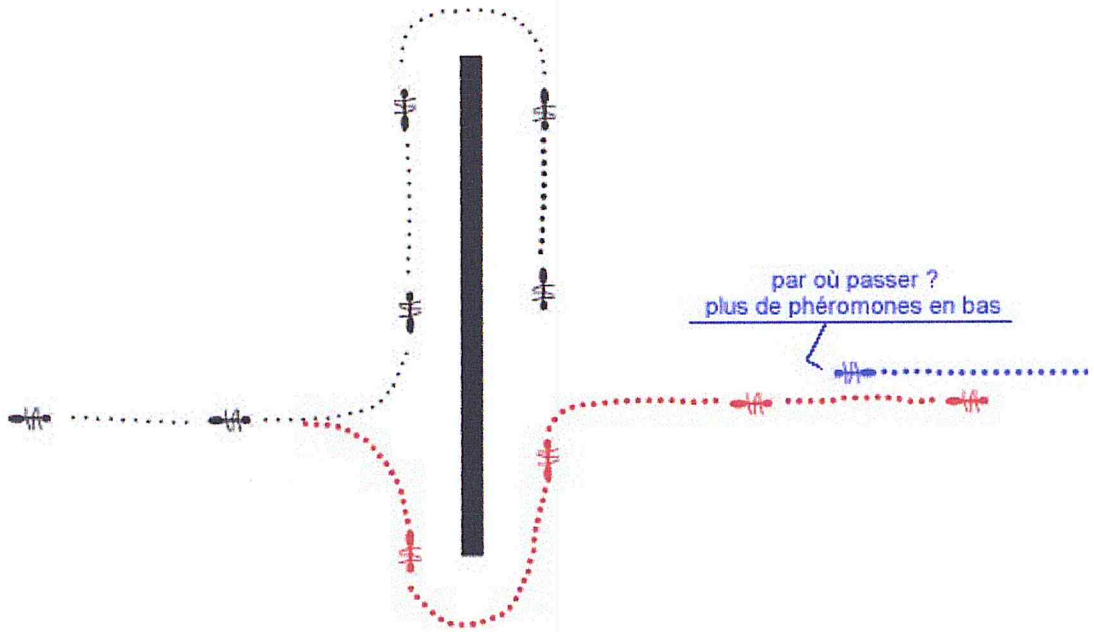


Figure III.1: Comportement des fourmis réelles

Les fourmis rouges ont pris un plus court chemin. Elles sont plus nombreuses à avoir franchi l'obstacle, donc plus de phéromone a été déposée du côté rouge que du côté noir. La fourmi bleue venant de l'autre sens choisira donc le côté rouge [ALE02].

5. La Meta heuristique colonies de fourmis :

Cette Meta heuristique a été introduite pour la première fois dans la thèse de doctorat de Marco dorigo (1992). Elle est inspirée par les études sur le comportement collectif des fourmis (dembourg et A.L 1983). A l'origine, l'optimisation par colonie de fourmis a été conçue pour résoudre le Problème du

Voyageur de Commerce (PVC). L'objectif est de trouver la tournée le plus courte pour un ensemble de villes données. La métaheuristique des colonies de fourmis est appliquée à des problèmes combinatoires à espace d'état discret.

6. Les caractéristiques des problèmes à espace d'état discret [DOR99a] :

- un ensemble fini de composants C= $\{c_1,c_2,\ldots,c_{Nc}\}$ (exp. sommet d'un graphe), tel que Card(C)=Nc.
- un ensemble fini de connexions (transition) L= $\{l_{ci,cj} / (ci,cj) \in \overline{C}\}$, ou $\overline{C}$ =C×C (produit cartésien).
- un coût $j_{ci,cj} \equiv J(l_{ci,cj},t)$ associé a chaque connexion $l_{ci,cj} \in L$, qui peut être parfois paramétré par le temps t.
- un ensemble fini de contraintes : $\Omega \equiv \Omega(C, L, t)$, assignées aux élément de C et L.
- $s=< c_1, c_2,\ldots, c_k,\ldots >$ est une séquence d'éléments de C (ou respectivement de L). la séquence est appelles l'état du problème. Si s est l'ensemble de toute les séquence possible, $\overline{S}$ est l'ensemble de toute les (sous) séquence qui respectent les contraintes $\Omega(C, L, t)$, $\overline{S}$ est dons S. Les élément de $\overline{S}$ définissent les états possibles du problème. La longueur de la séquence s, est le nombre de composants de s, elle est exprimée par $|s|$.
- la structure de voisinage est définie comme suit : on dit que s_2 est un voisin de s_1 si s1 et s_2 appartiennent à s, et on peut atteindre S_2 à partir de s1 en un seul pas logique (si c_1 est le dernier composent de la séquence s_1, il existe $c_2 \in C$, tel que $l_{ci,cj} \in L$ et $s_2 \equiv < s_1, c_2 >$).
- Ψ est une solution si elle appartient a $\overline{S}$ et satisfait toutes les contraintes du problème.
- $J\psi(L, t)$ est le coût associé à chaque solution Ψ. $J\psi(L,t)$ est une fonction de tous les coûts $j_{ci,cj}$ des connexion appartenant à la solution Ψ.

Les informations pendant le processus de recherche sont stockées dans la table de phéromone qui associe à chaque connexion $l_{ci,cj}$ une quantité de phéromone $\tau_{i,j}$. Cette table est une mémoire utilisée pour guider les recherches ultérieures effectuées par d'autres fourmis.

7. La colonie et ses fourmis :

Dans ce qui suit, nous donnons plus de détails sur le comportement des fourmis.

7.1. Les propriétés de la fourmi [DOR99a] :

- Une fourmis cherche une solution réalisable à coût minimal $j_\psi = \min_\psi j_\psi(L, t)$.
- Une fourmi K a une mémoire M^k utilisée pour stocker des informations concernant le chemin emprunté. Cette mémoire est aussi utilisée pour construire les solutions réalisables, évaluer les solutions trouvées et garder la trace du chemin emprunté.
- Une fourmi K peut être assignée à un état initial S_s^k est n ou plusieurs conditions d'arrêt e^k souvent l'état initial et un sommet.

- une fourmi K dans l'état $s_r=<s_{r-1}, i>$ peut passer à l'état $s_{r+1}=<s_r, j>$, tel que
- $N_i^k = \{j/ (j \in N_i) \wedge (<sr, j> \in \overline{S})\}$ est l'ensemble des un voisins de i.
- Les fourmis construisent leur solution d'une manière progressive. Elles démarrent à partir de l'état initial et passent par les voisins possibles. La procédure de construction se termine quand toutes les fourmis vérifient leurs critères d'arrêt.
- Une fourmi K dans le nœud i peut passer au nœud j en utilisant des règle de décision probabilistes.

7.2. Les propriétés de la colonie de fourmis :

Dans la même colonie :

- Chaque fourmi cherche à trouver une solution propre à elle.
- Chaque fourmi utilise ses propres informations et les informations locales des nœuds (composants) visités.
- Une fourmi communique avec les autres indirectement, à travers les informations lues/écrites de variables sauvegardant les valeurs de phéromone.

7.3. Les fourmis artificielles :

Les véritables fourmis sont attirées par leurs pistes de phéromone. Celles-ci peuvent persister de quelques heures à plusieurs mois suivant les conditions. Le but en informatique n'étant pas de reproduire parfaitement le modèle biologique mais de s'en inspirer, les informaticiens ont augmenté le taux d'évaporation de phéromone dans leurs programmes, de manière à ce que les mauvais chemins, moins empruntés, disparaissent sous l'effet de cette évaporation.

7.4. L'inspiration des fourmis réelles :

Les fourmis artificielles sont inspirées des fourmis réelles. Dans ce que suit nous allons donner quelques points de ressemblance [DOS01] :

- Les algorithmes de colonies de fourmis sont composés d'une colonie de fourmis artificielles concurrentes et coopératives.
- Les fourmis artificielles changent quelques informations numériques sauvegardées localement.
- La phéromone est le seul moyen de communication entre ces fourmis.
- Un mécanisme d'évaporation modifie l'information phéromone dans le temps, il permet de diriger la recherche vers une nouvelle direction.
- Les fourmis cherchent à minimiser la distance du chemin reliant leur nid aux sites destinataires, et ne violent par les étapes.
- On applique une politique de décision probabiliste pour se déplacer à travers les étapes adjacentes.

7.5. Différences entre fourmis artificielles et fourmis réelles [DOR 99a]:

Les fourmis artificielles ont quelques propriétés qui ne se trouvent pas dans les fourmis réelles :

- Elles ont un état interne privé, représenté par la mémoire des actions du passé.
- La quantité de phéromone déposée est une fonction de la qualité de la solution trouvée.

- Les fourmis artificielles peuvent être enrichies par les capacités supplémentaires comme la prévision, l'optimisation locale, le retour arrière.

8. Les caractéristiques de l'algorithme d'ACO [DOS 01] :

8.1. Les pistes de phéromone :

Le premier choix le plus important quand on applique l'algorithme ACO est la définition de la signification des pistes de phéromone. L'exemple suivant explique cette notion. Quand on applique l'ACO au problème du PVC, l'interprétation standard de la piste de phéromone, utilisée dans l'application, fait référence au caractère désirable de visiter une ville i directement après avoir visité la ville j.

8.2. Le nombre de fourmis :

Pourquoi utiliser une colonie de fourmis au lieu d'utiliser une seule fourmi ?
Bien qu'une fourmi seule soit capable de produire une solution, mais de qualité moindre, il est plus intéressant d'utiliser toute une colonie. Dans le cas de problèmes d'optimisation combinatoire, l'usage de m fourmis, $m>1$ qui construisent r solutions chacune (c.-à-d., l'algorithme ACO est exécuté r fois) pourrait être équivalent à l'usage d'une seule fourmi qui produit $m*r$ solutions. Les résultats théoriques sur la convergence des algorithmes ACO montrent qu'ils sont performants quand le nombre de fourmis m est supérieur à 1, étant donné qu'une seule fourmi peut ne pas bien explorer l'espace de recherche. Les solutions de bonne qualité sont le résultat de l'interaction collective des fourmis. Elles sont obtenues par la communication indirecte en lisant et écrivant des informations dans les variables sauvegardant les valeurs de phéromone.

8.3. La liste candidate :

La liste candidate est utilisée lorsque les algorithmes ACO sont appliqués aux problèmes de grandes tailles, là où la taille du voisinage est très grande. Cela engendre un voisinage volumineux pour une fourmi donnée, ce qui donne un grand nombre de mouvements possibles à choisir. Les inconvénients pouvant être rencontrés sont :

(i) La construction de la solution est ralentie considérablement,
(ii) La probabilité que beaucoup de fourmis visitent le même état est très faible.

Résolution du Partitionnement H/S avec l'ACO: Dans de telles situations, le problème peut être réduit par l'usage d'une liste de candidats. La liste candidate contiendra un ensemble réduit de voisins d'un état donné. L'usage de ces listes permet aux algorithmes ACO de se concentrer sur les composants les plus intéressants, en réduisant la dimension de l'espace de recherche.

L'idée de la liste candidate vient de l'inspiration d'autres techniques comme la Recherche Tabou, où certaines formes de listes candidates sont utilisées.

8.4. L'évaporation de phéromone :

L'intensité de la phéromone est décrémentée dans le temps. L'utilité est de rendre les arcs visités de moins en moins attirants. En favorisant l'exploration des chemins les moins visités, les fourmis auront tendance à ne pas converger vers un chemin commun. Si les fourmis explorent des chemins différents, alors il y a une plus forte probabilité que l'une d'elles trouve une amélioration de la solution. Par contre, si elles convergent toutes vers le même chemin, l'utilisation de *m* fourmis sera inutile.

8.5. Les actions de démon :

Les actions de démon (processus de veille) peuvent être utilisées pour implémenter des actions centralisées, qui ne peuvent pas être effectuées par les fourmis individuelles [DOS01]. Le démon peut observer le chemin trouvé par chaque fourmi de la colonie, et rajouter de l'extra phéromone sur les composants utilisés par la fourmi qui a construit le meilleur chemin (solution). Ceci en respectant la phéromone ***on-line*** déjà déposée. Cette mise à jour est dite la mise à jour "***off line*** "de phéromone.

9. Importance de l'heuristique :

La possibilité d'utiliser l'information heuristique pour diriger la construction probabiliste de la solution des fourmis est importante parce qu'elle donne la possibilité d'exploiter la connaissance des détails du problème. Cette connaissance peut être à priori disponible (c'est la situation la plus fréquente dans des problèmes statiques) ou rendue disponible durant l'exécution (c'est la situation typique dans des problèmes dynamiques). Dans des problèmes statiques, l'information heuristique η est calculée une fois au temps d'initialisation et est reste la même durant l'exécution de l'algorithme. Un exemple est l'utilisation de la longueur $\eta_{ij} = 1/d_{ij}$ des arcs reliant les villes.

Définissons dans ce qui suit l'information heuristique statique et l'information heuristique dynamique [DOS 01]:

9.1. Heuristique statique :

Dans les problèmes statiques, les informations heuristiques sont calculées à l'initialisation du problème et elles restent inchangées. Leurs avantages sont:

(i) faciles à calculer,
(ii) peuvent être calculées une fois à l'initialisation,
(iii) à chaque itération de l'algorithme ACO, une table peut être pré-calculée avec les valeurs de phéromone.

9.2. Heuristique dynamique :

Dans le cas des problèmes dynamiques, les informations heuristiques sont calculées au cours du traitement. Elles dépendent de la solution partielle construite. Par conséquent, elles doivent être recalculées à chaque pas de la promenade d'une fourmi. Cela engendre un temps de calcul plus élevé.

10. Algorithme général de la méthode de colonie de fourmis:

Cet algorithme se résume comme suit :

- La colonie de fourmi cherche d'une manière collective et asynchrone la solution qui présente une bonne caractéristique en affectant des transitions inter-nœud dans le graphe G.
- Les fourmis construisent leur solution d'une manière progressive. Une fois que la solution est construite ou durant sa construction, les fourmis procèdent à l'évaluation de la solution (partielle).

```
Algorithme colonies_de_fourmis ()
Début
    Tantque (condition_d'arrét_non_satisfaite)
    Faire
            Début
            Génération_activation_des_fourmis () ;
            Evaporation_pheromone () ;
            Action_de_démont () ;/*optionnelle*/
            Fin
    Fait
Fin.

Procédure Génération_activation_des_fourmis () ;
Début
    Tantque (ressources_diponibles)
    Faire
            Début
            Ordonnancer_creation_nouvelle_fourmis () ;
            Activer_nouvelle_fourmis () ;
            Fin
    Fait
Fin.
```

```
Procédure activer_nouvelle fourmi ()
Début
    Initialiser_fourmi () ;
    M : =MAJ_mémoire_fourmi () ;
    Tantque (Etat_courant<>Etat_cible)
    Faire
    Début
        A : =lire_Table_Routage_local_Fourmi () ;
        P : =Calcule_probabilité_transition (A, M, Ω) ;
        Etat _Suivant : =Appliquer_Regle_décision (P, Ω) ;
        Aller_à_l'etat_Suivant (l'etat_suivant) ;
        Si (MAJ_PHeromone_pas_a_pas_direct) Alors
        Début
            Déposer_Phéromone_sur_Arcs_Visités () ;
            MAJ_Table_Routage_Fourmi () ;
        Fin ;
    Fin ;
    Si (MAJ_Phéromone_Retarée_Dérect) Alors
    Début
        Pour Chaque Arc _visité εΨ faire
        Début
            Déposer_Phéromone_Sur_Arc_Visité () ;
            MAJ_Table_routage_fourmi () ;
        Fin
    Fin
Fin.
```

10.1. La procédure d'évaporation :

Elle a pour tache de réduire l'intensité de la phéromone, ce qui évite la convergence prématurée vers un optimum local. C'est une sorte de mémoire.

10.2. La procédure action de démon :

Est optionnelle. Elle est utilisée pour implémenter des actions centralisées qui ne peuvent pas être exécutées par les fourmis.

11. Application de la colonie de fourmis sur le problème de voyageur de commerce:

Le P VC, qui est un problème NP-complet très étudié depuis les années 70, est le premier problème qui fut résolu par Marco Dorigo à l'aide des colonies de fourmis.

11.1. Problème :

Etant donné un ensemble de villes C1,…., Cn séparées entre elles par une distance Jci,cj. Le problème consiste à trouver un parcours de longueur minimale passant une fois et une seule par chacune des villes et revenant à la ville de départ. Ce problème est équivalent à la recherche d'un cycle hamiltonien minimal dans un graphe complet pondéré.

Dans l'optique d'un problème de VC, chaque fourmi représente un agent [DOR99] et lorsqu'elle se déplace de la ville i à la ville j (avance du sommet i vers le sommet j), elle laisse une trace sur le chemin i, j (phéromone).

Un agent choisit la prochaine ville à l'aide d'une probabilité de transition Pr(i,j) telle que :

$$P_{ij} = \frac{(\tau_{ij\,0})^{\alpha} * (\eta_{ij})^{\beta}}{\sum_{l=1}^{N_i} (\tau_{il_0})^{\alpha} * (\eta_{il})^{\beta}}$$

τi,j (t) : la quantité de phéromone déposée sur chaque arête (ci,cj)

$$\tau_{ijt} = \rho * \tau_{ijt-1} + \sum_{k=1}^{m} \Delta\tau_{ijt}$$

Avec :

m : nombre de fourmis.

ρ : l'évaporation de phéromone 0<ρ<1.

$\Delta\tau^{k}_{i,j}$: La quantité de phéromone déposer par la fourmis K sur l'arrêt (ci,cj).

Tel que : $\Delta\tau^{k}_{i,j} = \begin{cases} 1/l_k & \text{si larrêt (i, j) est sur le cycle k de la longueur } l_k \\ 0 & \text{si non.} \end{cases}$

l_k : La longueur du cycle de la fourmi K.

$\eta_{i,j}=1/d_{i,j}$ la visibilité

α, β : sont des paramètres qui permettent de contrôler l'importance relative des deux éléments.

Finalement, une fourmi K possède une mémoire M^k (i) qui donne la $i^{éme}$ ville visitée par la fourmi sur un cycle afin d'obliger celle-ci à former une solution admissible. Si le nombre total de fourmis est m, et le nombre de villes à visiter est n, un cycle est réalisé lorsque chacune des m fourmis complète une tournée de villes.

11.2. Algorithme :

Initialisation :

Pour chaque arête (C_i, C_j) donner une valeur à $\tau_{i,j}$ (0) ;

Affecter M fourmis a M sommets ;

Pour chaque fourmi K sur chaque sommet i poser M^k (1)=i ;

ETAPE A : détermination d'un m-cycle
Tantque le m-cycle n'est pas fini faire
Pour chaque sommet i **faire**
Pour chaque fourmi K sur le sommet i **faire**
Avancer K vers le sommet j M^k choisi avec la Pr(i,j) ;
Mettre j dans M^k.
ETAPE B : calcul des taux de dépôt de phéromone.
Pour chaque fourmi K faire
Calculer la longueur L_k du cycle hamiltonien dans le m-cycle
Pour chaque arête (C_i, C_j)
évaluer $\Delta\tau^k_{i,j}$;
ETAPE C : déterminer le cycle hamiltonien général
Si (nombre de m-cycle<nombre maximal fixé) ou (toutes les fourmis ne choissent pas le même cycle)
alors
Vider pour chaque fourmis la mémoire M^k.
Pour chaque fourmis K sur chaque sommet i M^k (i)=i ;
Incrémenter le nombre de m-cycle de 1 ;
Revenir à l'étape A ;
Sinon
Donner le cycle qui minimise L_k parmi toutes les fourmis ;

12. Conclusion :

Dans ce chapitre, nous avons donné une description de la méta-heuristique des colonies de fourmis, avec un exemple d'application. Dans le chapitre suivant, nous introduisons le modèle d'exécution utilisé pour mettre en œuvre l'algorithme des colonies de fourmis.

Chapitre IV

La Machine PARE (PARtitioning Environment)

Chapitre 4 : La Machine PARE (PARtitioning Environment)

1. Introduction [OUM 01]

La machine PARE s'inspire de la machine PARME (PARtitioning and Max-sat Environment [OUM 00]. PARME comporte un processeur maître qui guide la recherche, collecte les informations et répartit le travail sur une ferme de processeurs esclaves. La communication entre les esclaves passe obligatoirement par le processeur maître qui réceptionne et renvoie les données. La Figure IV.1 représente l'architecture de la machine PARME. Elle contient quatre processeurs (PRT1 pour la méthode tabou N^{0}1, PRT2 pour la méthode tabou N^{0}2, PAG pour l'algorithme génétique PRS pour le recuit simulé et PRS2 pour l'algorithme MMD.

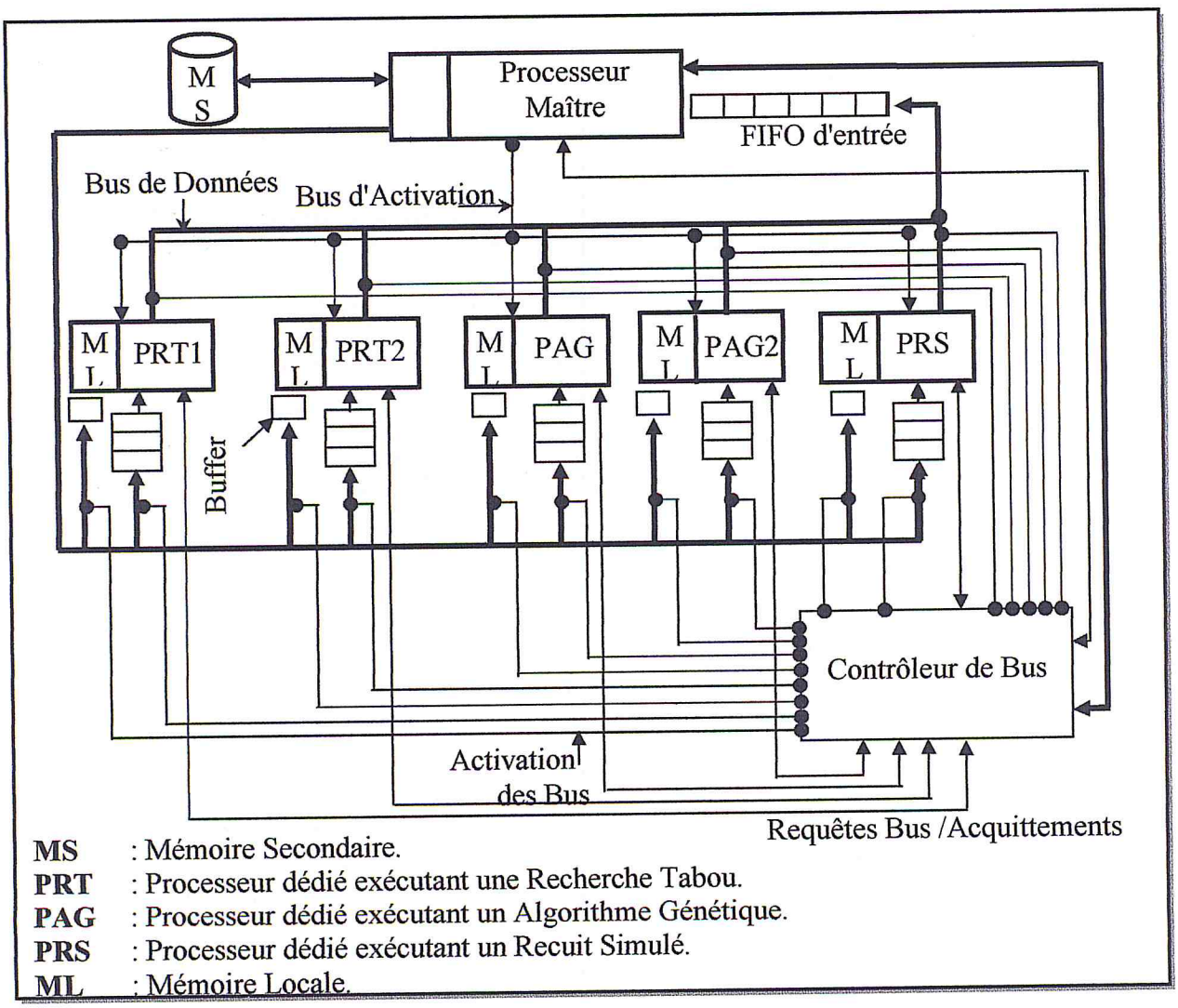


Figure IV.1 : Architecture de la machine parallèle PARME [OUM 00].

La machine PARE étend la machine PARME en mettant en œuvre un processeur esclave responsable de l'exécution des colonies de fourmis. Elle se nomme ainsi car, à la différence de PARME qui résout également le problème du Max-Sat, PARE est exclusivement réservée au problème de partitionnement en codesign.

1.1. Le Processeur Maître [OUM 00]

Le processeur maître est le cœur de la machine. Il s'occupe de la gestion de la ferme des nœuds et du placement des processus de l'application sur les processeurs du traitement des nœuds activés avec leurs différents paramètres. Il active le nombre de nœuds que nécessite l'application en cours et il peut désactiver des nœuds si nécessaire au cours du traitement. Tout échange de données entre les nœuds esclaves passe obligatoirement par le maître. A la réception des données envoyées par les nœuds esclaves, le processeur maître applique soit l'hybridation par l'échange directe des solutions entre les nœuds communiquant, soit l'hybridation par construction de solution. Cette dernière consiste à trouver une nouvelle solution à partir des solutions envoyées par les nœuds esclaves en appliquant, soit le principe d'intensification (prendre les caractéristiques les plus dominantes dans les solutions), soit le principe de diversification (prendre les caractéristiques les moins dominantes dans les solutions) ou la sélection de la meilleure solution ou de la plus mauvaise. La solution générée sera envoyée aux nœuds destinataires. Le processeur maître assure entre autres la communication avec l'extérieur, il peut répondre à la demande de l'utilisateur pour visualiser l'évolution et les statistiques de l'exécution en cours, comme il lui donne aussi la possibilité de modifier les paramètres des méthodes au cours du traitement. Il comporte une mémoire locale où seront stockées les données provenant des nœuds esclaves.

1.2. La ferme des nœuds [OUM 00]

Un nœud exclave de la machine se charge de l'exécution du processus que lui affecte le maître et de la communication avec les nœuds voisins spécifiés au préalable dans une matrice de communication implantée au niveau du maître.

1.3. Le réseau de communication [OUM 00]

La communication Maître/Esclave est assurée par des bus de données indépendants. Chaque nœud dispose de son propre bus connecté directement avec le maître. Cette idée est intéressante lorsque le nombre de nœuds n'est pas grand. Dans ce cas, elle présente l'avantage de ne jamais avoir de conflit de bus ou l'attente de la libération du bus (Bus partagé). Ce type de réseau est moins coûteux en matériel qu'un crossbar, car si PARME contient N nœuds, la réalisation d'un crossbar nécessite N*N liaisons (bus) et N*N commutateurs, par contre notre réseau ne nécessite que 2*N liaisons. En plus de cela, le réseau crossbar nécessite une reconfiguration dynamique en cours d'exécution.

Nous avons introduit quelques améliorations et modification sur l'architecture de PARME que nous venons de décrire afin de permettre l'exécution des colonies de fourmis.

2. Structure générale des environnements:

PARME et PARE ont été conçues à l'aide d'une méthode Orienté Objet en l'occurrence la méthode OOD, " La programmation orienté objets est une méthode de mise en œuvre dans laquelle les programmes sont organisés comme des ensembles d'objets coopérants. Chacun représente une instance d'une certaine classe. Toutes les classes étant des membres d'une hiérarchie de classes unifiées par des relations d'héritage "[BOO92]. Il repose essentiellement sur les modules que résume la Figure IV.2. Précisons que certaines des bibliothèques librairies et classes de PARME ont été réutilisées dans PARE afin d'éviter de perdre du temps à re-développer ce qui existe déjà.

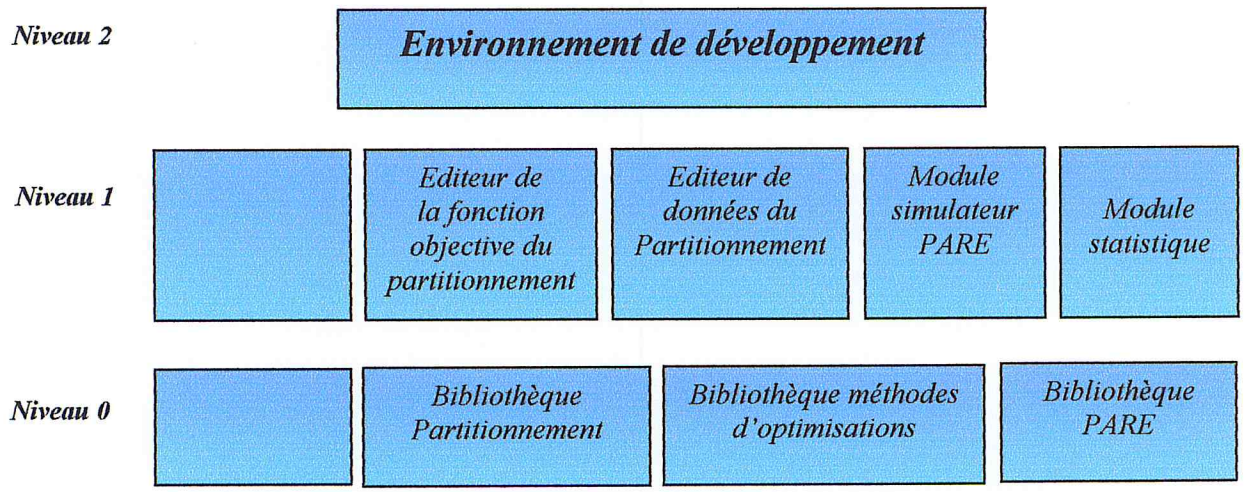


Figure IV.2: La structure générale de l'environnement de Partitionnement [OUM 00]

2.1.1. La bibliothèque Partitionnement

La bibliothèque partitionnement enrichit celle développée dans la PARME [OUM 00]. Elle comporte tous les composants logiciels nécessaires pour l'implémentation du problème de partitionnement. Elle est composée essentiellement de trois classes qui sont:

- Classe Entité.
- Classe Processeur.
- Classe Partitionnement.

Dans ce qui suit, nous allons décrire la structure de ces trois classes :

Classe Entité : Une Entité, est un composant logiciel qui peut être exécuté par un processeur. Elle est implémentée pour représenter le niveau de granularité Coarse-Grain (Niveau objet). Cette classe encapsule toutes les données caractérisant une entité logicielle telles que:

- Le numéro de l'entité.
- Un vecteur contenant l'espace et le temps d'exécution de l'entité sur chaque processeur.

Cette classe dispose de toutes les méthodes nécessaires pour la spécification de ces données, telles que l'introduction et la récupération de ces derniers.

Classe processeur : Cette classe regroupe toutes les caractéristiques d'un processeur, ainsi que les méthodes nécessaires pour la manipulation de ses données. Elle contient :

- Son nom et son numéro.
- Son type (hardware ou Software).
- L'espace global dont il dispose initialement.
- L'espace libre restant après affectation des entités.

Classe Partitionnement : La classe partitionnement implante toutes les données et structures nécessaires pour l'implémentation du problème de partitionnement. Elle est constituée des éléments suivants :

- Une matrice de communication entre les entités du système, où chaque élément représente la quantité de donnée à échanger entre deux entités communicantes. Si deux entités ne communiquent pas, cette quantité est nulle.
- Une matrice des coûts unitaires de communication entre les processeurs. Si deux processeurs ne communiquent pas, leurs coûts de communication est négatif pour interdire la sélection d'une solution contenant ces deux processeurs.
- La liste des processeurs du système.
- Un vecteur d'entités (un élément du vecteur est un objet de type entité) de dimension M (M : Nombre d'entités). Il représente toutes les entités du système et leurs relations avec les processeurs telles que, le temps d'exécution et l'espace occupé par une entité donnée sur un processeur donné. Cette classe dispose de toutes les méthodes d'introduction et de récupération des données ainsi que la méthode d'évaluation des solutions du partitionnement, appelée FonctionPart.

2.1.2. La bibliothèque Méthode d'optimisation :

Cette bibliothèque est composée de 5 sous librairies:

1. La librairie Outils.
2. La librairie Ant Colony.

La librairie Outils

La librairie outils est construite dans le but d'encapsuler tout les objets que partage les trois sous librairies citées ci-après. Ces objets sont les classes suivantes :

La classe Solution

Cette classe est conçue pour représenter une solution. Elle implante un vecteur dynamique de type booléen et son coût qui est de type réel. Nous avons implémenter les différentes méthodes de manipulation, insertion, suppression et modification des éléments du vecteur (valeurs propositionnelles), le calcul de la taille (nombre de variables propositionnelles qu'elle contient), ...

La classe Vecteur de Solutions

La classe vecteur de solution implante un vecteur dynamique d'objets de type solution. Elle est conçue dans le but d'être utilisée dans les méthodes tabou, recuit simule et algorithmes génétiques, essentiellement comme voisinage.

La classe fonction objectif :

Le problème de partitionnement a sa propre fonction objectif. cette classe contient toutes les méthodes nécessaires pour l'introduction et la lecture des données dont elle dispose

La librairie colonie de fourmis :

La méthode colonie de fourmis décrite précédemment et implémenté sous forme d'une classe appelée Classe colonie de fourmis.

2.1.3. La bibliothèque PARE

La librairie PARE est composée de deux sous-librairies, la sous-librairie Maître et la sous-librairie Esclave. Chacune regroupe tous les composants logiciels nécessaires pour l'implémentation de la machine parallèle virtuelle PARE. Cette librairie, fait utilise des threads permettant de simuler les différents processeurs.

La librairie Maître :

Elle est constituée de quatre classes principales :

Classe Proceseur Maitre: englobe toutes les structures necessaires permettant la simulation du processeur Maitre. Elle hérite des proprietés des classes suivantes: la classe FileEntre, la classe MatHybConstSol et la classe MatHybEchangeSol.

Classe File-Entre: comporte un vecteur dynamique où seront déposées les solutions envoyées par les processeurs esclaves. Chaque élement de cette file contient les données suivantes:

- Le numéro du processeur envoyeur.
- La solution courante et son coût.
- L'iteration où est apparue cette solution.
- Le temps d'execution.
- La meilleure solution trouvée jusqu'à présent.

Cette classe comporte toutes les méthodes manipulant la file, telles que, l'insertion, la suppression, la récuperation des eléments de la file, calcul de la taille de la file (Nombre d'eléments de la file).

Classe MatHybConstSol: Cette classe a été introduite pour implanter l'hybridation par construction de solution. Elle encapsule un vecteur dynamique. Chaque élémént du vecteur est une structure de type GroupMetCoop qui comporte:

- Un vecteur de numeros de processeurs
- Le nombre de solutions à utiliser pour générer une nouvelle solution.
- Numero de la méthode d'hybridation choisie.
- Numéro du groupe.

Classe MatHybEchangeSol: Pour implementer l'idée de l'hybridation par échange de solution, nous avons introduit cette classe avec toutes les structures et méthodes nécessaires. Elle contient une table à deux dimensions d'une taille dynamique où chaque ligne contient l'ensemble des processeurs coopérants.

La librairie Esclave:

Elle englobe les structures et classes necessaires pour simuler l'ensemble des processeurs esclaves. Elle comporte essentiellement, la classe Processeur Esclave, la classe Processeur Tabou, la classe Processeur Recuit Simulé, la classe Processu

MMD et la classe processuer Génétique.

Classe Processeur Esclave: La machine PARE a une architecture multi-processeur. Elle dispose d'un ensemble de processeurs ayant des traitements différents (chacun exécute un processeus donné). La classe Processeur Esclave est réalisée dans le but de regrouper toutes les caracteristiques communes à ces processeurs telles que:

- Le numéro du processeur.
- L'état du processeur (activé, bloqué, ou suspendu).
- L'état du bus d'entrée (ouvert ou fermé).
- L'état du bus de sortie.
- Un vecteur dynamique pour représenter la file d'entrée du processeur.
- Le temps total de son execution.

Classe Contrôleur: Cette classe implante le controleur de bus de la machine parallèle. Elle encapsule:

- Un vecteur dynamique d'entiers contenant les numéros des processeurs demandeurs d'accées au bus.
- Un vecteur de priorités de type entiers, il contient les priorités associées aux processeurs pour acceder au bus. Cette classe, encapsule aussi toutes les méthodes necessaires pour le remplissage du vecteur des priorités, pour l'envoie des signaux d'ouverture et fermeture des bus, pour recevoir les demandes des processeurs et les inserer dans le vecteur des processeurs demandeurs d'accés au bus, ainsi que la suppression de ces demandes après leurs acquitement.

2.2. Niveau 1:

Il est composé de:

2.2.1. L'éditeur de la fonction objectif du Partitionnement

C'est le même que celui développé pour PARME [OUM 00]. Il permet d'étudier le problème de Partitionnement, une des données à fixer en entrée est la fonction objectif. Cette fonction dépend de trois paramètres, l'Espace, le Temps d'exécution et le coût de communication.

2.2.2. L'éditeur des données du Partitionnement

C'est le même que celui développé pour PARME [OUM 00]. Au début de l'application, les données à fixer pour le partitionnement sont : l'espace, le temps d'exécution et le coût de communication. Pour cela, l'idéal serait de disposer d'un analyseur de caractéristiques d'espace, de surface et de communication. En l'absence de tel outil, nous proposons deux façons différentes pour introduire ces paramètres. Soit par saisie manuelle, soit par génération automatique.

2.2.3. Module simulateur PARE :

Le module simulateur PARE, simule la machine parallèle PARE. Il comporte les composants principaux suivants : le processeur maître, les processeurs esclaves (généraux et dédiés) et le contrôleur.

Avant de lancer le simulateur PARE, l'utilisateur doit fixer :

- Le traitement à effectuer;
- Les processeurs à utiliser ainsi que leurs paramètres.

2.2.4. Module statistique

Le module statistique offre à l'utilisateur un ensemble d'outils statistiques numériques et graphiques. Ce qui lui permettra de bien étudier et analyser : les résultats obtenus, l'effet des paramètres des méthodes d'optimisation choisies, leurs rapidité de convergence et de pouvoir faire une étude comparative des méthodes

2.3. Niveau 2

Il s'agit de l'interface qui assure la communication entre l'utilisateur et l'outil de simulation réalisé.

3. Conclusion :

Dans ce chapitre, l'environnement de partitionnement est décrit: il s'agit de la machine parallèle PARE, développée autour de la machine PARME [OUM 00], qu'elle étend pour l'exécution de l'algorithme des colonies de fourmis. Les différents modules communs aux deux environnements sont décrits, ainsi que les nouveaux modules insérés pour former l'environnement PARE, ainsi que leur rôle et leur niveau hiérarchique dans le logiciel.

Chapitre V

Les colonies de fourmis pour résoudre le partitionnement

Chapitre 5 : Les colonies de fourmis pour résoudre le partitionnement.

1. Introduction :

L'idée principale de l'ACS (Ant Colony System) [DOR97] est de modéliser le problème comme la recherche du meilleur chemin dans un graphe particulier .le comportement de fourmis est déterminé en définissant :

- L'état de départ;
- Les conditions des terminaisons;
- Les règles de construction;
- Les règles de mise à jour de phéromone;
- Et les actions de démon [DOR99a].

On pourra citer deux stratégies de l'ACS pour le partitionnement :

- la première consiste à démarrer d'une solution initialement vide, et de l'étendre progressivement en lui rajoutant à chaque itération un composant jusqu'à arriver à une solution complète.
- la deuxième consiste à faire démarrer chaque fourmi d'une solution initiale complète générée aléatoirement, et de lui appliquer des flips répétitifs afin de construire une solution finale.

Dans notre travail nous allons mettre en œuvre la deuxième stratégie.

2. La stratégie d'amélioration de la solution :

Dans cette stratégie, le problème de partitionnement est caractérisé par :

- chaque solution S est un vecteur d'entier (numéros des processeurs), dont chaque position du vecteur (numéro d'entité) contient la valeur de processeur auquel elle a été assignée.
- un ensemble C de toutes les solutions possibles. $|C| = Np^{Ne}$
- un ensemble L des connexions entre les éléments de C. Deux solutions S1, S2 de C sont connectées si et seulement si elles différent d'un seul élément.
- le déplacement d'une solution S1 vers une solution S2 se fait par un flip d'un seul composant de S1.
- la fonction objectif d'une solution S (coût de S) est de la valeur de la fonction objectif utilisées dans le partitionnement.
- la taille maximale de liste candidate dans cette stratégie est égale à Ne*(Np-1), (l'ensemble de tous les couples (Ne, Np) qui ne sont pas dans la solution initiale).

3. L'algorithme ACS de base pour la stratégie d'amélioration de la solution :

Début
Initialiser la phéromone ;
Pour (i=1 à MaxTter) **faire**
Pour (chaque fourmi) **faire**
- Générer une solution initiale S0 aléatoirement.
- Construire une solution Sk.
- Appliquer la mise à jour online delayed (retardée) de phéromone.

Fin Pour
- déterminer la meilleure solution trouvée dans cette itération
- appliquer la mise à jour offline de phéromone

Fin pour.
Fin.

4. L'organigramme :

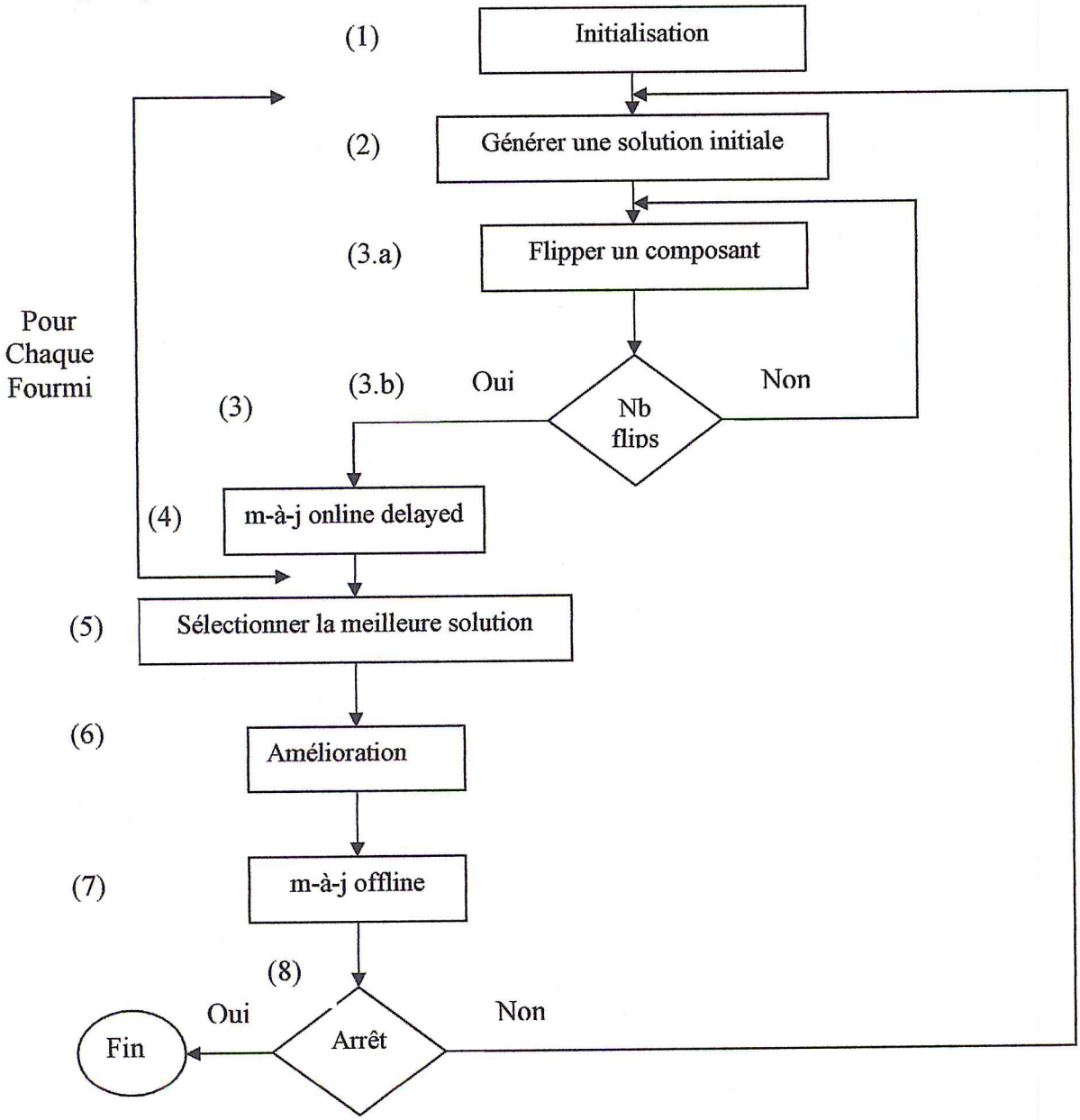


Figure V.1 : Les différentes étapes de la stratégie d'amélioration de la solution.

5. Explication de l'organigramme :

(1) initialisation de la phéromone : initialiser les valeurs de phéromone sur les composants à une valeur fixée par l'utilisateur.

(2) initialisation des fourmis : pour chaque fourmi, affecter une solution de départ choisie aléatoirement ou selon une stratégie bien définie.

(3) l'amélioration de la solution : dans cette étape chaque fourmi, essaye d'améliorer sa solution en choisissant à chaque étape le composant à flipper dans la solution actuelle, jusqu'à atteindre un nombre de flips fixé au préalable.

(3.a) flipper un composant : choisir dans la liste candidate de la fourmi le prochain composant à flipper dans la solution courante, en appliquant la règle de transition d'état. Une évaporation de phéromone est effectuée sur le composant choisi, pour le rendre moins attirant.

(3.b) solution construite : si le nombre de flips n'est pas atteint, la fourmi réitère le processus depuis l'étape (3.a).

(4) la mise à jour online-delayed : appliquer la mise à jour online-delayed aux composants da la solution finale, selon la qualité de cette dernière.

(5) mise à jour de la meilleure solution : une fois que toutes les fourmis ont achevé la construction de leurs solutions, le démon met à jour la meilleure solution trouvée jusque là en observant toutes les solutions construites par les fourmis.

(6) la recherche locale : on peut appliquer un algorithme de recherche locale dans le but d'améliorer la meilleure solution trouvée par les fourmis.

(7) la mise offline : de l'extra phéromone est rajoutée par le démon aux composant (ou connexion) de la meilleure solution dans le but de les rendre plus attirants pour les générations suivantes.

(8) le critère d'arrêt : relancer les fourmis depuis la phase (2) pour une nouvelle génération tant que le nombre de générations fixé au préalable n'est pas atteint.

Initialisation de la phéromone :

La phéromone est déposée sur chaque composant de l'instance; elle représente l'utilisation précédente de cet élément. L'information phéromone est implémentée en utilisant une table de taille

égale au nombre de sommets du graphe de construction. Toutes les entrées de cette table sont initialisées à une valeur petite (généralement égale à 0,1).

6. Construction de la solution :

Chaque fourmi démarre d'une solution initiale S_0 générée aléatoirement, et construit sa solution S_k itérativement en effectuant des flips. A chaque itération elle choisit une variable X_i à flipper avec une probabilité en utilisant la règle de transition d'état :

$$\text{Si } q \leq q_0 \text{ alors } \quad P_{xi}^{k}(t) = \begin{cases} 1 \text{ si } (i, j) = \text{Argmax}_{(i,j)\in jk} \left\{ \left\{ \tau_{ij}(t) * \left[\eta_{ij}(t)\right]^{\beta} \right\} \right. \\ 0 \quad \text{si non} \end{cases}$$

$$\text{Si } q > q_0 \qquad P_{xi}^{k}(t) = \frac{\tau_{ij}(t) * \left[\eta_{ij}(t)\right]^{\beta}}{\sum_{xl \in jk} \tau_{xl}(t) * \left[\eta_{xl}(t)\right]^{\beta}}$$

Où $X_{ij} = X_{ik}$ a l'instant t ($k \in \{0,1,\ldots,NP-1\}$ avec NP nombre de processeur du système).

$\tau_{ij}(t)$: La valeur de phéromone représentant le composant de la variable (entité) i de valeur (processeur j a l'instant t.

$\eta_{ij}(t)$: La valeur de la fonction heuristique a l'instant t.

jk : l'ensemble des variables non taboue (qui ne son pas encore flipper).

q : une variable aléatoire uniforme distribué sur [0,1].

q_0 : un paramètre de l'algorithme pris entre 0 et 1.

L'augmentation de la valeur de q_0 mène à une concentration de la recherche sur les composants des meilleures solutions. Dans le cas contraire, on favorise l'exploration d'autres zones dans l'espace de recherche.

7. La mise à jour de phéromone :

Elle consiste en une évaporation de la phéromone pour tous les composants, suivie d'un ajout pour les composants de la solution en question.

- L'évaporation est appliquée par la règle suivante : $\tau ij = (1- \rho) * \tau ij$.
- La quantité de phéromone rajoutée aux composants de la solution $S*$ obtenue, est proportionnelle au coût($S*$). Dans la mise à jour online-delayed, l'ajout suit la règle :
- $\tau ij = \tau ij + \rho*$(coût($S*$)/nombre-total-des-clauses).
- Alors que dans la mise à jour offline, est appliquée par la règle :
- $\tau ij = \tau ij + \rho*$(coût (*BestSol*)/coût($S*$)).

8. Exemple :

L'affectation de 3 entité E_0, E_1, E_2, sur deux processeurs P_0, P_1

L'ensemble de solution C contient 16 éléments $|C|=2^3= 8$; Donc on a 8 solutions pour cet exemple:

S_0 est l'affectation des entité E_0, E_1, E_2, aux processeur P_0.

0	1	2
0	0	0

S_1 est l'affectation des entités E_0, E_1, processeurs P_0 et E_2 ou processeur P_1

0	1	2
0	0	1

S_2 est l'affectation de l'entité E_0 ou processeur P_0 et E_1, E_2 aux processeurs P_1

0	1	2
0	1	1

S_3 est l'affectation de l'entité E_0 et l'entité E_2 aux processeurs P_0, et l'entité E_1 aux Processeur P_1.

0	1	2
0	1	0

S_4 est l'affectation de l'entité E_0 aux processeur P_1 et E_1, E_2 aux processeur P_0

0	1	2
1	0	0

S_5 est l'affectation des entités E_0, E_1, aux processeur P_1 et E_2 aux processeur P_0

0	1	2
1	1	0

S_6 est l'affectation de l'entité E_0 et l'entité E_2 aux processeurs P_1, et l'entité E_1 aux Processeur P_0.

0	1	2
1	0	1

S_7 est l'affectation des entité E_0, E_1, E_2, aux processeur P_1.

0	1	2
1	1	1

L'affectation des fourmis :

Modélisation graphique du partitionnement :

Pour pouvoir résoudre le partitionnement avec l'ACO, il faudra d'abord le modéliser sous forme d'un CSP. X = {E0, E1, …, ENbe-1}, D(Ei) ={ P0, P1, …, ENbp-1}, C représente les contraintes du Partitionnement.

Nous avons proposé que les sommets du graphe de construction soient des combinaisons entre chaque entité avec chaque processeur du système. Nous codons chaque couple (N° entité, N° processeur) pour obtenir un entier qui représente l'entité et le processeur désignés. La fonction de codage utilisée est :

Coder : (0..Nbe)* (0..Nbp) ⟶ (0..Nbe*Nbp-1)
(ne,np) ⟶ ***Coder***(ne,np)=ne*Nbp+np

Exemple :

Si nous avons à répartir quatre entités E_0, E_1, E_2 et E_3 sur deux processeurs P_0 et P_1, nous aurons les codes suivants :

Coder (0,0) = 0*2+0 = 0 ;

Coder (0,1) = 0*2+1 = 1 ;

Coder (1,0) = 1*2+0 = 2 ;

Coder (1,1) = 1*2+1 = 3 ;

Coder (2,0) = 2*2+0 = 4 ;

Coder (2,1) = 2*2+1 = 5 ;

Coder (3,0) = 3*2+0 = 6 ;

Coder (3,1) = 3*2+1 = 7 ;

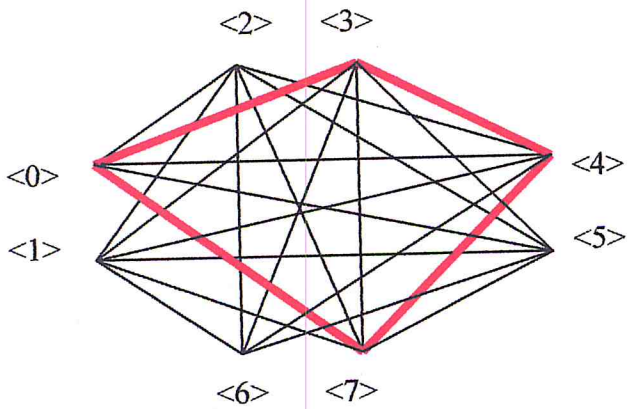


Figure V.2 : Exemple d'un graphe de construction pour le partitionnement.

Le chemin trouvé {0, 3, 4, 7} représente la solution {(0,0), (1,1), (2,0), (3,1)}.

Pour récupérer les couples (entité, processeur) trouvés dans un chemin donné, nous avons conçu une fonction de décodage (***Décoder*** =***Coder*** $^{-1}$) qui fait le travail inverse de la fonction de codage. Elle est définie par : ***Décoder***(c) = (ne, np).

Décoder : (0..Nbe*Nbp-1) ⟶ (0..Nbe)* (0..Nbp)
ne = c **div** Nbp
c ⟶ np = c **mod** Nbp

Dans l'exemple précédent, Décoder(4) = (2, 0) .

Remarque importante :

Dans la codification, nous avons à codifier Nbe*Nbp couples. Donc, l'ensemble de départ contient Nbe*Nbp éléments. Et d'un autre coté, nous savons que les codes doivent être compris entre 0 et Nbe*Nbp-1. Donc on aura au maximum Nbe*Nbp codes possibles. Et puisque chaque couple (e,p)

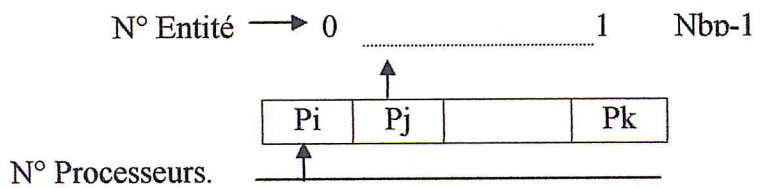
possède un code unique différent de ceux des autres couples, nous aurons a utiliser tous les codes de l'ensemble d'arrivée. Ceci implique que le cardinal de l'ensemble d'arrivée est égal à celui de départ. Et rappelons que la valeur de code maximum est (Nbe*Nbp-1), on peut déduire que *Coder* est une fonction bijective, est *Décoder* est la fonction réciproque de *Coder*. Nous aurons enfin comme conclusion que la codification est unique, ce qu'il fallait démontrer.

9. Modélisation proposée pour résoudre le Partitionnement avec l'ACO:

9.1. Représentation de la solution du partitionnement :

La solution du partitionnement peut être représentée de multiples manières. Dans notre travail, nous avons proposé la représentation suivante :

Soit Nbp, le nombre de processeurs et Nbe, le nombre d'entités.



I, j, k ∈ [0,Nbp-1]

Exemple: Si on dispose de huit entités et de dix processeurs, on peut avoir la solution suivante :

0	1	2	3	4	5	6	7
2	5	5	1	5	3	2	0

- Le processeur N° 2 exécute l'entité N° '0' et l'entité N° 6.
- Le processeur N° 5 exécute l'entité N° '1', '2' et '4'.
- Les processeurs N° 0, 1, 3 exécutent respectivement les entités N° '7', '3', '5'.

Par un vecteur de couples (entité, processeur), de taille égale à Nbe. Chaque élément du vecteur représente une entité et le processeur qui l'exécute codée de la façon suivante :

N° entité * Nbp + N° processeur.

Dans l'exemple précédent on peut avoir le vecteur suivant avec cette représentation :

Nbp=6 ;

0	1	2	3	4	5	6	7
2	11	17	19	29	33	38	42

V [6]= 6*6 + 2.

V [4]= 4*6 + 5.

9.2. Le codage des solutions :

Le codage des solutions est sous format binaire. Nous devons donc calculer le nombre de bits, nb donné par la formule suivante :

$$nb = \begin{cases} [\text{Log}_2(\text{Nbp})] + 1 & \text{si } \text{Log}_2(\text{Nbp}) \text{ n'est pas un entier.} \\ \text{Log}_2(\text{Nbp}) & \text{dans le cas contraire.} \end{cases}$$

[r]: représente la partie entière d'un nombre réel r.

La taille d'une solution codée en Binaire est égale à Nbe * nb. On peut coder la solution de l'exemple précèdent comme suit:

Nbp=10, Nbe=8.

$\text{Log}_2(\text{Nbp}) = \text{Log}_2(10) = 3.32$. N'est pas un entier donc nb = [3.32] +1= 4bits.

On peut donc représenter un processeur par 4 bits. Par conséquent, la solution sera représentée par 32 bits. La solution en Binaire est donc:

	0	1	2	3	4	5	6	7
	0010	0101	0101	0001	0101	0011	0010	0000
Equivalente en décimal	2	5	5	1	5	3	2	0

Si Nbp=8, $\text{Log}_2(8)=3$ c'est un entier donc nb=3 bits. Ainsi, la taille de la solution sera de 24 bits et la solution précédente devient:

	0	1	2	3	4	5	6	7
	010	101	101	001	101	011	010	000
Equivalente en décimal	2	5	5	1	5	3	2	0

9.3. Le décodage :

Les solutions obtenues par les algorithmes d'optimisation utilisés son codées en Binaire. Pour pouvoir évaluer ces solutions en fonction des paramètres liés au partitionnement (l'espace, le temps d'exécution et le coût de communication), nous devons retourner au codage initial par une opération de décodage. Le principe de ce décodage est le suivant :

- Recalculer le nombre de bits 'nb' représentant un processeur dans une solution.
- Découper la solution en tranches de 'nb' bits chacune.
- Décoder chaque tranche du Binaire vers le décimal.

Exemple :

Soit Nbp=10, Nbe=8.

nb =$[\text{Log}_2(10)]$ +1=4bits.

Si on dispose de la solution en Binaire suivante :

0100 0011 0101 0000 0111 0100 1000 1001

On prend chaque tranche de 4 bits et on la décode vers le décimal. Ainsi, la solution décodée est :

0	1	2	3	4	5	6	7
4	3	5	0	7	4	8	9

Problème : Les algorithmes d'optimisations peuvent produire des solutions de la forme :

Solution en binaire :

0100	0011	1100	0000	1111	0100	1000	1001

Après décodage :

4	3	12	0	15	4	8	9

Si on dispose de 10 processeurs, on aura nb = 4bits. Cette solution est incorrecte, car elle contient des numéros des processeurs supérieurs au nombre de processeurs existant. 12>10 et 15>10.

Solution : Pour éviter ce type de solutions, à chaque décodage de 'nb' bits on teste si le nombre obtenu est inférieur au nombre de processeurs existants. S'il est supérieur, on génère un nombre aléatoire inférieur à Nbp, puis on remplace le numéro du processeur erroné par ce nombre dans la solution décodée. Pour garder l'équivalence de la solution décodée avec celle en Binaire on remplace les 'nb' bits décodés par le code en Binaire du nombre généré.

9.4. Contraintes sur les solutions du partitionnement :

Au cours de la recherche, une méthode d'optimisation peut trouver une solution bonne mais incohérente. Cette incohérence est engendrée soit par :

- L'affectation d'une entité à un processeur qui ne dispose pas d'espace libre suffisant pour l'exécuter.
- L'affectation de deux entités communicantes à deux processeurs non communicants.

Pour remédier à ce problème, les solutions incohérentes ne seront jamais sélectionnées. Pour cela, leurs coûts seront multipliés par -1 rendant ainsi les solutions incohérentes négatives.

10. La Mise en œuvre :

Dans cette partie, nous décrivons la réalisation de l'intégration, dans la plate forme PARE de la nouvelle méthode d'optimisation définie en chapitre 3 (les colonies de fourmis). Pour cela, nous avons ajouté un nouveau processeur virtuel **PAntColony** aux différents processeurs déjà existants dans PARME. Ce processeur représente la méthode colonie de fourmis. En premier, nous citons quelques objectifs de la mise en œuvre, puis, nous présentons l'environnement de programmation de notre

logiciel. Ensuite, nous détaillons notre méthodologie dans la réalisation des principaux modules le composant.

10.1. Objectif de la mise en œuvre :

L'environnement de partitionnement permet à l'utilisateur de faire une bonne étude expérimentale sur les différentes approches de résolution des problèmes de partitionnement en codesign. Il permet aussi à l'utilisateur l'expérimentation des différentes méthodes avec différents paramètres, ce qui peut l'aider à trouver le bon paramétrage au problème partitionnement. A l'aide de cette plate forme, on peut comprendre l'évolution et le fonctionnement de l'approche des colonies de fourmis. Pour cela, notre intégration de l'ACO dans le logiciel **PARE** doit être :

- D'une utilisation facile, à travers une interface graphique conviviale.
- Paramétré au maximum, afin d'élargir le champs des expériences et de tests.
- Il doit mettre à la disposition de l'utilisateur un ensemble d'outils statistiques, graphiques et numériques qui permettent une bonne analyse des résultats.
- L'environnement doit être interactif, pour permettre à l'utilisateur la consultation des résultats et la modification des paramètres des processeurs en cours d'exécution.

11. La librairie colonies de fourmis :

La méthode colonies de fourmis décrites dans le Chapitre 3 est implémentée sous forme d'une classe appelée ***Classe AntColony***. De même que pour les autres approches dans PARME, nous avons introduit toutes les méthodes nécessaires pour rendre la classe dynamique et paramétrée. Nous avons donc les méthodes d'insertion et de lecture de la phéromone initiale, des différents paramètres, et du nombre d'itérations. Cette classe peut être complétée, selon les exigences du domaine d'application, en rajoutant d'autres paramètres et méthodes.

12. Conclusion :

Dans ce chapitre nous avons présenté la stratégie d'amélioration de la solution, avec le développement d'un algorithme basé sur la méta heuristique colonie de fourmis pour le problème de partitionnement.

Chapitre VI

Tests et résultats

CHAPITRE 6 : Tests et résultats

1. Introduction :

Nous allons, dans ce chapitre présenter quelques résultats obtenus en appliquant la méthode de la colonie de fourmis à un problème de partitionnement sur deux benchmarks (jeux d'essai).

Tous les tests sont réalisés sur un Intel Pentium IV 2,4GHz avec 256 Mo de RAM. Soulignons que les temps d'exécution concernent uniquement les traitements effectifs de l'algorithme, et excluent les opérations d'affichage et d'entrées/sorties.

2. Présentation des benchmarks utilisés :

Les benchmarks (jeux d'essai) utilisés sont le AG_10_2 et l'Aleatoire_100_5. Le1er benchmark contient 10 entités qui doivent s'exécuter sur une architecture cible comportant deux processeurs. Il est difficile à résoudre du fait qu'il ne comporte que deux solutions réalisables.

Le second benchmark contient 100 entités qui doivent s'exécuter sur 5 processeurs ; nous l'avons généré aléatoirement pour étudier le comportement des heuristiques et de l'environnement sur des benchmarks de très grandes tailles.

La fonction de coût utilisée est une simple combinaison des trois critères (respectivement : Espace, Temps d'exécution et Communication) : **f (x,y,z) = x+y+z.**

2.1. Le premier benchmark (AG_10_2) :

Ce benchmark représente les entités du programme représentant l'algorithme génétique utilisé dans l'environnement Parme. Le but est d'accélérer cet algorithme en le partitionnant sur une architecture parallèle. Le nombre d'entités est de dix et l'architecture choisie est composé de deux processeurs [KOU 02]. Les tables (Tab. 1), (Tab. 2), (Tab. 3), (Tab. 4) et (Tab. 5) résument les caractéristiques des différentes entités composant le benchmark. Il s'agit:

1. des quantités de données échangées entre les différentes entités (matérialisant les volumes de communication transitant sur l'interface entre les processeurs de l'architecture cible) en tableau 1;

2. Des espaces occupés par les différentes entités sur chacun des processeurs de l'architecture cible en tableau 2;

3. Des temps d'exécution des différentes entités sur chacun des processeurs de l'architecture cible en tableau 3;

4. Des caractéristiques concernant l'architecture cible, à savoir les espaces disponibles sur chacun des processeurs (en tableau 4), et les coûts de communication entre les processeurs (en tableau 5).

	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9
E0	0	10	151	130	140	156	191	105	120	45
E1	12	0	150	101	100	150	200	103	130	51
E2	75	70	0	220	220	182	994	0	0	456
E3	0	0	500	0	10	50	385	220	203	352
E4	0	0	455	10	0	501	400	215	215	400
E5	200	156	673	4106	105	0	11	66	60	782
E6	35	35	145	0	0	890	0	0	0	822
E7	200	130	101	200	172	222	356	0	10	39
E8	160	130	120	170	200	222	400	10	0	30
E9	420	420	330	25	27	100	112	100	787	0

Tab. 1 : Matrice des quantités de données échangées entres les entités.

	P0	P1
E0	300	250
E1	300	200
E2	700	585
E3	450	370
E4	450	350
E5	780	666
E6	324	230
E7	330	202
E8	240	190
E9	540	378

Tab. 2 : Matrice des espaces occupés par les entités sur les processeurs.

	P0	P1
E0	609	14200
E1	500	14078
E2	1257	3008
E3	700	21235
E4	721	20230
E5	1325	3515
E6	615	2859
E7	431	12030
E8	222	13039
E9	813	2454

Tab. 3 : Matrice des temps d'exécutions des entités sur les processeurs.

P0	P1
1250	2400

Tab. 4 : Matrice des espaces disponibles sur les différents processeurs.

	P0	P1
P0	1	1
P1	18	1

Tab. 5 : Matrice des coûts de communication entre les processeurs.

2.2. Le deuxième benchmark (Aléatoire_100_5):

Le second benchmark a été généré volontairement de taille très importante pour étudier le comportement de l'environnement et des heuristiques devant des problèmes de partitionnement de grandes tailles. Il comporte 100 entités à répartir sur 5 processeurs.

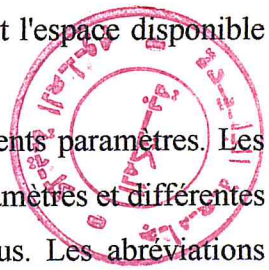
L'utilisation d'un algorithme exact qui examine de manière exhaustive en énumérant toutes les solutions est impossible à mettre en œuvre. En effet, l'espace de solutions comporte 5^{100} répartitions

possibles. Si l'algorithme est exécuté sur une machine à 2.4 GHz et que chaque itération se déroule en un seul cycle d'horloge, le temps nécessaire pour balayer tout l'espace des solutions dépasse $1,04*10^{53}$ années.

Les matrices de données initiales sont trop volumineuses pour être présentées. Dans ce qui suit, les résultats des simulation ayant permis d'obtenir les meilleures solutions sont présentés.

3. Exécution de la colonie de fourmis :

L'abréviation **X** indique qu'aucune solution cohérente n'a été trouvée. Par solution cohérente, nous entendons une solution réalisable. Une solution incohérente, par exemple, consisterait à affecter, à un processeur donné, un ensemble d'entités dont la somme des espaces dépasserait l'espace disponible sur le processeur.



De nombreux tests ont été exécutés avec la Colonie de fourmis avec différents paramètres. Les tables (Tab. 6) et (Tab. 7) résument certains de ces tests avec différents jeux de paramètres et différentes stratégies de la méthode appliqués aux deux benchmarks, et les résultats obtenus. Les abréviations suivantes sont utilisées:

Col : taille de la colonie ; **Nit** : nombre d'itérations ; **Ng** : nombre de générations ; **Can** : taille de la liste candidat ; **H** : l'heuristique (**S** : statique ; **D** : Dynamique ; **D1** : Dynamique individuelle ; **D2** : Dynamique collective ; **N** : Pas d'heuristique) ; **It** : itération où la meilleure solution est apparue ; **Temps** : temps d'exécution pour trouver la meilleure solution.

CF	Col	Nit	Ng	Can	H	Résultats		
						coût	It	Temps
CF1	200	10	50	20	D2	214189.5	1	1,954
CF2	200	10	50	20	D1	214189.5	33	9,994
CF3	100	9	50	18	D	X	X	X
CF4	150	9	50	18	S	X	X	X

Tab.6 : Paramètres et résultats des colonies de fourmis sur le 1er benchmark.

CF	Col	Nit	Ng	Can	H	Résultats		
						coût	It	Temps
CF1	200	50	100	100	D2	3490.5	8	246.98
CF2	100	25	10	50	D2	4051	6	41.92
CF2	250	100	10	200	S	137224.5	3	127.73
CF2	100	100	10	50	D	142155.5	9	64.9

Tab.7 : Paramètres et résultats des colonies de fourmis sur le 2eme benchmark

On peut facilement remarquer dans les deux tableaux précédents (Tab. 6) et (Tab. 7), que l'utilisation de l'heuristique dynamique collective trouve les meilleures solutions par rapport aux autres approches de cette méthode, en terme de qualité et de temps d'exécution (pour le 1er benchmark en un temps : 1,954s). Pour le 2ème Benchmark, elle a réussi à trouver une solution de coût minimal de 3490.5.

Les bons résultats sont obtenus dans la majorité des cas avec $\tau 0=0.1$, .=0.1, $\alpha=0.01$, ß=1, q0=0.8. Donc nous avons fixé les paramètres $\tau 0$, ., α, ß, q0 et nous avons fait varier les valeurs des autres paramètres.

Le 1er benchmark étant difficile, nous avons utilisé une colonie de fourmis assez importante de taille égale à 200 fourmis avec une liste candidate maximale (de la taille du voisinage), l'algorithme s'arrête après 50 générations. L'utilisation de la stratégie amélioration de la solution à permis de trouver la solution optimale.

A travers le second benchmarks, il est possible de remarquer que l'augmentation du nombre de fourmis donne de bons résultats mais ralentit énormément le temps de traitement.

Conclusion Génegale

Conclusion Générale

Dans ce mémoire, nous nous sommes intéressés au problème de partitionnement Matériel/Logiciel. Nous avons commencé par introduire ce problème ainsi que des méthodes de résolution.

Nous avons ensuite présenté une machine parallèle virtuelle conçue pour la résolution de ce type de problème. Elle a été mise en œuvre à travers un outil de simulation pour permettre à l'utilisateur de tester l'heuristique adoptée et comparer différents jeux de paramètres.

L'environnement réalisé permet également d'accomplir des études statistiques portant sur les différents composants, pour l'expérimentation des différents concepts de la méthode à la recherche des meilleurs paramètres s'adaptant au problème donné.

Notre plate forme a été réalisée sous Windows XP à l'aide de l'environnement de développement C++ Builder.

Comme complément de notre travail, de nombreuses extensions peuvent être apportées, parmi lesquelles:

- L'ajout de nouvelles méthodes d'optimisation.
- Doter cet environnement d'assistants permettant l'ajout automatique de nouvelles méthodes de résolution.

Bibliographie

[ADA 96]	ADAMS J.K. AND THOMAS D.E., "Tutorial the design of mixed hardware/software systems", 33rd Design Automation Conference, 1996.
[ALE02]	Alexandre Aupetit," les algorithmes classiques pour résoudre le PVC "http // home.alex.tuxfamily.org/pvc.html Avril 2002
[BOO92]	Grady Booch, "Conception Orientée Objet et Applications", Addison-W publishing, 1992.
[CNRS00]	Intelligence collective des fourmis et nouvelles techniques d'optimisation. INFO 200.
[DEM 97]	DE MICHELI G. AND GUPTA R., "Hardware/Software Co-design", Proceedings of the IEEE, Vol.85, N°3, pp.349-365, 1997.
[DOR 97]	M. Dorigo, et L. M. Gambardella. Ant Colony system A cooperation learning approch to the traveling salesman problem. IEEE Trans. Evol. Comp., 1(1) 53-66, 1997.
[DOR 99a]	M. Dorigo, G. Dicaro. Ant Colony Optimisation A new meta-heuristic. IEEE 99.
[DOR 98]	Marco Dorigo, Giani Di Caro et Luca M.Gambgadella. "Ant Algorithm of Descret Optimization". Tech..Rep.IRIADIA/98-10.uniuversity libre de bruxelles.
[DOS 01]	The Ant Colony Optimization Metaheuristic Algorithms, Applications, and Advances.Marco Dorigo Université Libre de BruxellesIRIDIA.Thomas Stützle, TU Darmstadt, Computer Science, Intellectics Group.
[EDW 97]	EDWARDS M.D., FORREST J. AND WHELAN A.E., "Acceleration of software algorithms using hardware/software co-design techniques", Journal of Systems Architecture, Vol.42, N°9, Feb. 1997.
[GLO 86]	F. GLOVER, Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research* 13533-549, 1986.
[GOL94]	Goldberg,"Algorithmes génétiques exploration, optimisation et apprentissage automatique", Addison-Wesley, 1994.
[GUP95]	Gupta, "Co-Synthesis of Hardware and Software for degital embedded systems", Kluwer Academic Publishers, p 266, 1995
[ISM94]	Ismail, Abib, Jerraya, COSMOS a CoDesign Approche for Communicating Systems, Proceedings of the 3rd International WorkShop on Hardware/Software Codesign, IEEE CS Press, pp 17-24, Sept. 1994, Grenoble, France.
[JIN99]	Jin-Kao Hao, Philippe Galinier, Michel Habib Méthaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes LERIA, U.F.R. Sciences, Université d'Angers, 2 bd Lavoisier, 49045 Angers
[JPM 99]	Jin-Kao Hao, Philippe Galinier, Michel Habib, Méthaheuristiques pour L'optimisation combinatoire et l'affectation sous contraintes. U.F.R. Sciences, Université d'Angers, Ecole des Mines d'Alès, Parc Scientifique Georges Besse, 1999.
[KOU 02]	KOUDIL M., "Une approche orientée objet pour le Codesign", Thèse Doctorat d'Etat, Institut National d'Informatique, Alger, 2002.
[KUM 96]	KUMAR S. AYLOR J.H. JOHNSON B. AND WULF W.A., "The Codesign of Embedded Systems", Kluwer Academic Publishers, 1996.

[MCSE]	"Conception conjointe", www.lester.uni-ubs.fr/8080/~heller/these/thesetoc.html.
[OUM 00]	H. OUMSALEM et K. CHAOUCHE, "Max-Sat et Partitionnement à l'aide de la méthode de Recherche Tabou Hybride implémentée sur une machine parallèle MIMD", Mémoire d'Ingénieur, Institut National de formation en Informatique, INI, 2000.
[PAP 82]	C.H. PAPADIMITRIOU, K. STEIGLITZ, *Combinatorial optimization – algorithms and complexity*. Prentice Hall, 1982.
[REE 93]	C.R. REEVES (Ed.) *Modern heuristic techniques for combinatorial problems*, Blackwell Scientific Publications, Oxford, 1993.
[RIB 94]	C.C. RIBEIRO, N. MACULAN (Eds.), Applications of combinatorial optimization. *Annals of Operations Research* 50, 1994.
[STO 95]	STOY E., "A petri net based unified representation for hw/sw codesign", Masters hesis, Dept. of Computer and Information Science, Linköping University, Sweden, S-581 83, 1995.
[WOL94]	Wolf, "Hardware, Software Co-Design of embedded systems", proceding of the IEEE, Vol 82, No 7, Julay 1994, pp 967-989
[WOO94]	Woodruff, " Simulated Annealing and Tabu Search Lessons from a linear search', Computers Operational Research, Vol 21, N°8, p 823-839 (1994).