N° d'ordre : 03/2001-M/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

Université des Sciences et de la Technologie Houari Boumedienne
Institut des Techniciens Supérieurs

Thèse de Magister en Informatique

Présentée par : **Mme BABA-ALI née BENKHIDER SADJIA**

THEME

Une approche Génétique Parallèle pour la résolution du problème de dimensionnement de portes des circuits intégrés VLSI

Soutenue le : 02/06/2001

Devant le Jury :

Mme H. DRIAS	**Professeur, USTHB**	**Présidente**
Mme F. BOUMGHAR	**Maître de Conférences, USTHB**	**Directeur de thèse**
Mme H. BELHADJ-AISSA	**Maître de Conf.erences, USTHB**	**Examinateur**
Mr S. NIAR	**Maître de Conférences, VALENCIENNES**	**Examinateur**
Mr ATTIF	**Maitre de conferences, USTHB**	**Examinateur**
Mr BELOUCHRANI	**Chargé de cours, ENP**	**Examinateur**

Dédicaces

A ma petite famille composée de : Riadh et de nos trois adorables petits anges Assia, Liès et Mehdi

A mes parents dont la patience et les encouragements qu'ils m'ont toujours prodigué n'ont pas d'égale

A mes frères et sœurs qui ont toujours su être là quand j'en avais besoin et à tous les Benkhider.

A mes Beaux-Parents .

Remerciements

Mes remerciements les plus vifs vont à :

- Mme F. Boumghar Maître de Conférences à l'USTHB, pour avoir suivi et dirigé ce travail, pour ses conseils et ses nombreuses re-lectures, malgré ses activités très prenantes.

- Mr A.R. Baba-Ali Chargé de Cours à l'USTHB, pour m'avoir proposé un sujet motivant et complet, pour ses conseils pratiques et son suivi tout au long de ce travail.

- Mme H. Drias Professeur à l'USTHB, pour avoir accepté de présider ce jury et d'examiner ce travail.

- Mmes et Mrs Les Membres du Jury pour avoir accepté d'examiner ce travail.

- Mr S. Niar Maître de Conférences à l'Université de Valenciennes, France, pour le grand intérêt qu'il a prêté à ce travail, pour ses conseils et ses orientations très utiles qu'il me faisait parvenir par Internet, je le remercie vivement.

- Mes collègues du Centre de Calcul de l'USTHB notamment Mr A. Gonzar, Mme Y. Chouaki et Mme S. Rahmouni pour le soutien moral et les encouragements très importants durant mes moments les plus difficiles.

Enfin tous ceux et toutes celles qui ont contribué à la réalisation de ce travail, de près ou de loin : je les remercie du fond du cœur.

Résumé de thèse de magister intitulée :

Une Approche Génétique Parallèle pour la résolution du problème de Dimensionnement de Circuits Intégrés VLSI

S. Benkhider*

Thèse Encadrée par : F. Boumghar
Maître de conférence au département ″Instrumentation et Automatique″
de la Faculté de Génie Electrique & Informatique de l'USTHB

*épouse Baba-Ali : étudiante en magister à l'ITS.

Résumé: Cette thèse décrit la mise en œuvre d'un algorithme génétique pour la résolution du problème de dimensionnement de circuits intégrés VLSI ou *″Gate Sizing″*. Ce problème appartient à la classe des problèmes NP-Complets, étant donnée la taille très importante des circuits VLSI ainsi que leur grande diversité : il n'existe pas d'algorithme exact pour le résoudre. Pour cela, nous nous sommes intéressés à des solutions heuristiques. L'algorithme génétique est une heuristique qui donne des réponses satisfaisantes en un temps raisonnable. Toutefois, dans le cas du problème de dimensionnement, comme la taille d'un circuit (ou individu) est très élevée, le temps d'évaluation de tous les individus de chaque génération est très important. Pour cela, nous avons étudié et proposé dans cette thèse, une approche génétique parallèle de l'algorithme génétique, notamment l'évaluation des individus d'une population en parallèle. Cette parallélisation peut être implémentée sous environnement PVM [1] (Parallel Virtual Machine) qui est un système permettant de ″voir″ une collection de machines connectées en réseau local, comme une seule machine parallèle.

Mots Clés : Circuits intégrés VLSI, Algorithme génétique, optimisation combinatoire, Environnement parallèle.

1 Introduction

Un circuit VLSI est un circuit intégré à très grande échelle. Il caractérise généralement une fonction complète d'un système. Sa complexité est de l'ordre d'un million de portes logiques et plus. Une porte est un composant de base d'un circuit intégré. Actuellement, aucun circuit intégré ne peut être conçu qu'en ayant recours aux logiciels de CAO qui permettent de le simuler et de l'analyser avant de le fabriquer.

Le problème du dimensionnement de circuits intégrés VLSI [2], [3], consiste à rechercher pour un circuit donné une configuration qui permet de réaliser les objectifs du cahier des charges. Pour cela, il s'agit de choisir pour chaque composant du circuit (des portes logiques), ses paramètres pour lesquels il devient suffisamment rapide tout en minimisant sa consommation d'énergie, sa surface, son coût, etc… Le principal paramètre considéré par les méthodes d'optimisation de circuit basées sur le dimensionnement ou sizing, est la dimension des composants élémentaires. Pour cela, chacun des composants sera disponible, dans une bibliothèque de cellules, en plusieurs versions de dimensions différentes mais toutefois équivalentes d'un point de vue fonctionnel. Les composants de grandes dimensions sont plus rapides, mais ont pour inconvénient de coûter plus cher, de consommer plus d'énergie et par conséquent de dissiper plus de chaleur. Tandis que les composants de dimensions faibles sont plus économiques et dissipent moins de chaleur. Par contre ils ont pour inconvénient d'être beaucoup moins rapides. Le dilemme est alors formulé comme suit : Il s'agit de déterminer pour chaque composant la taille optimale, pour obtenir un circuit suffisamment rapide mais qui ait en même temps une surface de silicium minimale, de manière à réduire le coût de fabrication et la dissipation de chaleur.

Ce problème appartient à la classe des problèmes NP-Complets , ce qui signifie qu'il n'existe pas d'algorithmes efficaces pour le résoudre : les seules méthodes exactes existantes sont de type énumératives telle que Branch and Bound,…. Ces dernières ont pour inconvénient d'avoir des complexités exponentielles.

La complexité d'algorithmes est une mesure de la performance ou du temps d'exécution . La performance d'algorithmes est mesurée en fonction de la taille des données par une fonction notée T(n) . Etant donné que des algorithmes complexes peuvent traiter différents cas de figure, la performance est généralement estimée dans le cas le plus défavorable. On s'intéresse plus particulièrement au comportement asymptotique de la fonction T(n) pour avoir une idée des performances de l'algorithme quand la taille des données évolue. Pour cela, la notation 0 (grand O) est utilisée.

On dit que T(n) est en O(f(n)), s'il existe deux constantes c et no telles que :

$$\mathbf{T(n) \leq c\, f(n) \quad \forall\, n \geq no}$$

Cette mesure permet aussi la comparaison d'algorithmes et la détermination approximative des tailles maximales des problèmes qui peuvent être traités par les algorithmes.

On considère généralement qu'un "bon" algorithme est un algorithme polynomial $0(n^k)$ parce que le temps de réponse de l'algorithme évoluera de façon polynomiale avec la taille des données. Par contre, on considère un algorithme exponentiel comme étant un "mauvais" algorithme parce que le temps de calcul va vite devenir prohibitif, dès que la taille du problème augmente au delà d'un seuil.

La distinction entre algorithmes polynomiaux et exponentiels a permis d'établir une classification des problèmes. Si un problème possède un algorithme polynomial, alors on considère qu'il est traitable ou encore qu'il appartient à la classe des

problèmes P qui est considérée comme une classe de problèmes faciles. Si ce n'est pas le cas (i.e. si on ne connaît aucun algorithme polynomial), on considère ce problème comme étant difficile (NP-Complet ou NP-difficile).

Certains problèmes peuvent être résolus par des algorithmes polynomiaux qui sont exécutés sur des machines hypothétiques appelées machines Non_déterministes. Ces machines se caractérisent par un nombre infini de processeurs parallèles qui donnent la possibilité de faire systématiquement des choix optimaux en évaluant chaque possibilité à l'aide d'un processeur.

Dans le cas des problèmes NP-Difficiles, on a généralement recours à des algorithmes approchés appelés heuristiques. Ces heuristiques vont essentiellement déterminer une solution qui approche la solution optimale en un temps raisonnable (ie: avec une complexité polynomiale).

Dans le cas du dimensionnement des circuits intégrés VLSI, il est clair que la taille des paramètres est très importante (plusieurs millions de transistors) et que c'est un problème de choix combinatoire.

Pour cela, nous avons choisi d'apporter une solution à ce problème en ayant recours à un algorithme génétique. Ce dernier est une métaheuristique connue pour sa robustesse et sa simplicité dans la résolution d'un problème NP-complet.

2 Les algorithmes génétiques :

2.1 Introduction :

Les algorithmes génétiques [4], [5] sont des algorithmes d'exploration fondés sur les mécanismes de la sélection naturelle et de la génétique. La sélection naturelle élimine l'une des contraintes majeures à la conception des programmes : la spécification préalable de toutes les caractéristiques d'un problème et des tâches précises qu'un programme doit effectuer pour résoudre ce problème. Bien

qu'utilisant le hasard, ils ne sont pas purement aléatoires. Ils explorent efficacement l'information obtenue précédemment pour spéculer sur la position de nouveaux points à explorer avec espoir d'amélioration des performances.

Les algorithmes génétiques ont été développés par Jacques Holland[16], ses collègues et ses étudiants à l'université du Michigan. Leurs recherches avaient deux objectifs principaux :

- mettre en évidence et expliquer rigoureusement les processus d'adaptation des systèmes naturels,
- concevoir des systèmes artificiels (des logiciels) qui possèdent les propriétés importantes des systèmes naturels.

Cette approche a débouché sur des découvertes importantes à la fois dans les sciences des systèmes naturels et dans celle des systèmes artificiels.

2.2 Principes de base :

Les algorithmes génétiques sont fondamentalement différents des algorithmes classiques d'optimisation selon quatre axes principaux :

- ils utilisent un codage des paramètres et non les paramètres eux mêmes.
- ils travaillent sur une population de points au lieu d'un point unique.
- ils n'utilisent que la valeur de la fonction étudiée, jamais sa dérivée qui nécessite un calcul analytique ou numérique, ni aucune autre connaissance auxiliaire.
- ils utilisent des règles de transition probabilistes et non déterministes.

Toutes ces caractéristiques prises ensemble, contribuent à la robustesse de ces algorithmes.

Comme il a été spécifié au premier point, les algorithmes génétiques utilisent un codage des paramètres. Pour cela, ils manipulent des chromosomes ou chaînes. Ces derniers représentent en réalité chacun une solution possible du problème.
Un chromosome est constitué de gènes, chacun représentant une variable qui caractérise un des paramètres du problème. Il est à noter que dans le codage binaire, un gène est représenté par un bit (0 ou 1).

2.3 Les opérateurs génétiques

Les mécanismes de base d'un algorithme génétique mettent en jeu des copies de chaînes et des échanges de parties de chaînes(ou chromosomes). Au départ, une génération de chaînes est créée en effectuant un tirage aléatoire. A partir de cette génération, les opérateurs suivants :

- la reproduction
- le crossover
- la mutation

sont appliqués pour générer une nouvelle population.

2.3.1 La reproduction

C'est un procédé selon lequel chaque chaîne est copiée en fonction des valeurs de la fonction à optimiser appelée fonction d'adaptation F.
Cela revient à donner aux chaînes dont la valeur de F est plus optimale, une probabilité plus élevée de contribuer à la génération suivante en créant des

descendants. Cet opérateur est une version artificielle de la sélection naturelle où l'adaptation est déterminée par la capacité d'une créature à survivre à tous les obstacles jusqu'à l'âge adulte. Dans notre milieu artificiel, la fonction d'adaptation est l'unique décideur de la vie ou de la mort de chaque chaine-créature.

En pratique, cet opérateur est mis en œuvre en créant une roue de loterie biaisée pour laquelle chaque chaîne de la population présente, occupe une section de la roue proportionnelle à son adaptation.

2.3.2 **Le croisement ou *"crossover"***

Durant cette phase, les éléments nouvellement produits par la reproduction sont d'abord appariés, ensuite un croisement a lieu entre chaque paire de chaînes de la manière suivante :

Soit L la longueur d'une chaine et soit k un entier représentant une position sur la chaîne. Cet entier est choisi aléatoirement entre 1 et (L-1), donc k appartient à l'intervalle [1, L - 1]. Deux nouvelles chaînes sont crées en échangeant tous les caractères compris entre les positions (k+1) et L inclues.

Exemple : considérons les chaînes A1 et A2 de la population initiale :

A1 = 011\0101 (\ est un caractère de séparation pour illustrer le point de croisement!)

A2 = 110\1110

Pour k=3, les chaînes A'1 et A'2 obtenues après croisement sont :

A'1 = 011\1110

A'2 = 110\0101

Les mécanismes de reproduction et de croisement sont donc assez simples mais ce sont leurs actions combinées qui donnent aux algorithmes génétiques leur

puissance, le croisement n'étant rien d'autre que la juxtaposition de choses qui ont bien fonctionné dans le passé.

2.3.3 La mutation

Dans un algorithme génétique simple, la mutation est la modification aléatoire occasionnelle (de faible probabilité) de la valeur d'un caractère de la chaîne. Cela revient simplement à changer un 1 en un 0 ou inversement lorsque le codage est binaire..

Utilisée parcimonieusement avec la reproduction et le crossover, la mutation est une police d'assurance protégeant de la perte prématurée de notions importantes. Par ailleurs, cela permet de créer de nouveaux individus.

Par l'application de ces trois opérateurs, nous obtenons une nouvelle génération dont les individus doivent être évalués. Pour cela, la fonction d'adaptation F déterminera les meilleurs chromosomes à utiliser dans la prochaine génération : ce sont les chromosomes qui ont une fonction d'adaptation la plus élevée qui seront copiés dans la future génération.

En général, la taille d'une population est suffisamment élevée (une centaine) et l'évaluation de tous les chromosomes nécessite un temps appréciable.

Pour cela, nous verrons par la suite comment essayer de réduire ce temps en exécutant en parallèle l'évaluation de plusieurs chromosomes.

3 Description des circuits intégrés VLSI :

3.1 Introduction

Un circuit intégré [6] est un circuit entièrement réalisé sur un substrat semi-conducteur comme le silicium. Les composants élémentaires des circuits intégrés digitaux sont les portes interconnectées par des niveaux de métallisation.

Aujourd'hui, la conception de circuits n'est possible qu'avec des logiciels de C.A.O, vues leurs tailles et leurs complexités. Ces derniers assistent les concepteurs en leur permettant de visualiser, simuler et vérifier leurs circuits avant de les fabriquer. Pour cela, une représentation en mémoire centrale est nécessaire, sous forme de structures de données qui permettent à des programmes d'effectuer les traitements désirés sur ces circuits.

3.2 Représentation en mémoire des circuits VLSI :

La représentation la plus répandue de circuits intégrés[15] est celle qui les décrit sous forme d'un graphe. Dans notre cas, les circuits peuvent être décrits comme un graphe où les nœuds correspondent aux portes. Deux nœuds sont reliés par un arc si les portes correspondantes dans le circuit sont interconnectées. Chaque nœud possède deux poids :

- la surface Si : c'est la dimension d'un composant, notamment une porte logique.
- le délai de propagation de cette porte Di : c'est le temps que met un signal pour traverser cette porte.

Une partie de notre travail a consisté à construire le graphe qui représente un circuit sous forme de structures de listes chaînées dynamiques en mémoire.

3.3 **Position du problème** :

Le problème d'optimisation de performances de circuits intégrés consiste généralement à minimiser leur surface car c'est un critère d'évaluation de leur coût, tout en prenant en compte des contraintes concernant leur délai de propagation [15].

Ce problème peut être vu de manière formelle comme suit : les différentes implémentations possibles d'un circuit définissent un espace de conception (design space) qui est constitué d'un ensemble de points (design point) représentant chacun une implémentation possible du circuit. A chaque point ou implémentation sera associé un ensemble de valeurs représentant les performances.

Dans notre cas, il s'agit de minimiser la surface totale du circuit tout en tenant compte de son délai fixé dans les cahiers des charges.

Plus la dimension ou surface Si d'une porte logique est grande et plus elle est rapide c'est à dire son délai Di est plus court mais son coût est naturellement plus élevée. Il s'agit donc de trouver un compromis à toutes ces données, sachant que la surface totale du circuit dépend de la surface de ses composants et que son délai de propagation total est précisément la somme des délais de toutes les portes situées sur le chemin le plus lent de ce circuit (chemin critique):

$$F = \begin{cases} \text{Min } S \\ \text{avec } D < \text{contrainte} \end{cases}$$

$D = \sum_{j=1}^{k} D_j$ sur le chemin critique, D_j est le délai de propagation d'une porte
k : nb de portes du chemin critique.

$S = \sum_{i=1}^{n} S_i$ pour tout le circuit, Si est la surface d'une porte
n : nb total de portes du circuit.

Il est à noter, toutefois, que le chemin critique d'un circuit est obtenu en ayant recours à un algorithme d'exploration dans un graphe, soit l'algorithme BFS (Breath First Search ou exploration en profondeur), ou bien l'algorithme DFS (Depth First Search ou exploration en largeur) qu'il a fallu implémenter et intégrer dans notre logiciel d'optimisation.

Nous avons formulé le problème de dimensionnment d'un circuit intégré VLSI comme étant un problème de minimisation en utilisant une fonction de pénalisation concernant la contrainte sur le délai, pour obtenir la fonction ″objectif″ suivante:

$$F = \text{Min}\ (S + \lambda * D) \qquad \text{avec } \lambda \text{ pour facteur de pénalisation}$$

Le facteur de pénalisation permet de favoriser la minimisation de la surface par rapport au délai de propagation du circuit ou bien l'inverse. Tout dépend du cahier des charges établi par le concepteur. Si nous voulons que le circuit ait une surface faible, il faut choisir λ petit. Ainsi, notre algorithme génétique favorisera les solutions peu coûteuses en surface. Par contre, si le délai de propagation doit être suffisamment réduit, il faut choisir λ grand pour établir l'équilibre entre la surface et le délai du circuit. En réalité, il s'agit d'une optimisation à multi-critère dans ce problème.

Le choix des dimensions de portes se faisant dans une bibliothèque de cellules, notre problème est donc bien un problème de choix combinatoire.

Ce dernier est appelé problème de dimensionnement (Gate sizing). Chan [2] a d'abord démontré qu'il appartenait à la classe NP-Complet, en démontrant qu'il pouvait être réduit polynomialement au problème de satisfaisabilité. Ensuite Moon [3] a pu démontrer que ce problème appartenait à la classe NP-Hard au sens fort.

3.4 Codage et Application des opérateurs génétiques au problème de dimensionnement des circuits intégrés VLSI :

Comme il a été spécifié, dans le cadre d'un algorithme génétique les paramètres du problème sont codés sous forme de chromosomes.

Dans notre cas, un chromosome ou chaîne représente une configuration d'un circuit possible ou une solution. Chaque gène du chromosome correspond à une porte logique du circuit, alors que la valeur du gène représente la dimension de la porte correspondante. Ceci est illustré par la figure Fig. 1.

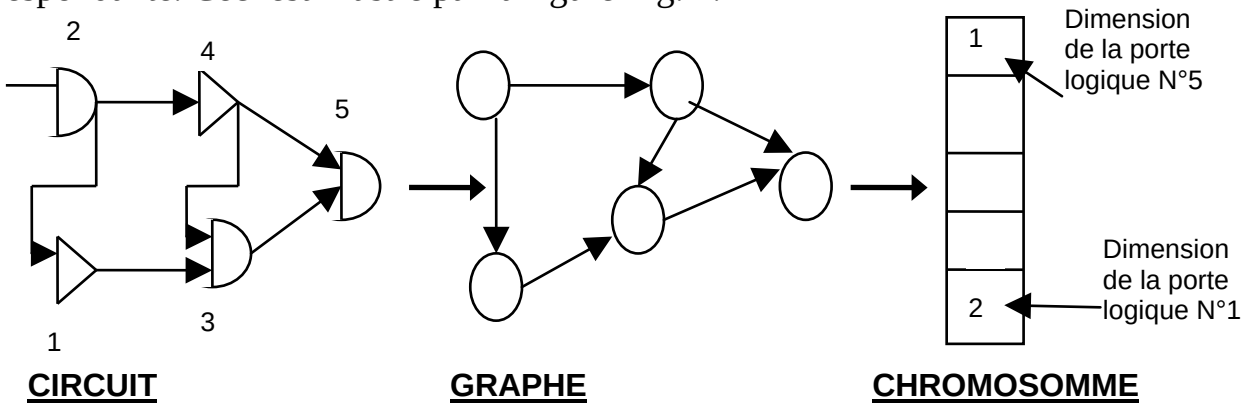


Fig. 1 : Codage du problème de dimensionnement

Nous avons disposé, lors de notre travail, d'une bibliothèque de cellules, dans laquelle les composants du circuit sont à choisir. Chacun de ces composants (des portes logiques) est disponible en au moins deux versions d'un point de vue dimension. Notre codage n'étant pas binaire, nous avons pris pour chaque porte, deux dimensions : l'une minimale (lower_bound), l'autre maximale (upper_bound). Lors de la génération de la population initiale, il y a tirage au sort des types de composant du circuit, entre la valeur maximale et la valeur minimale pour chacun d'entre eux.

Lors de l'application du croisement, il s'agit de mélanger deux parties de deux circuits différents qui ont déjà montré de bonnes performances dans une génération courante de circuits. Ceci nous permet d'obtenir deux nouveaux circuits pour la génération future. La mutation, quant à elle, consiste à modifier la dimension d'une certaine porte dans un circuit donné et avec une très faible probabilité, dans le but de produire de nouveaux circuits. En dernier lieu, il s'agit d'évaluer tous ces nouveaux circuits obtenus. Ce qui est fait en utilisant la fonction objectif F décrite dans la section 3.3. Les figures Fig.2 et Fig.3 illustrent le croisement de deux circuits et la mutation dans un circuit.

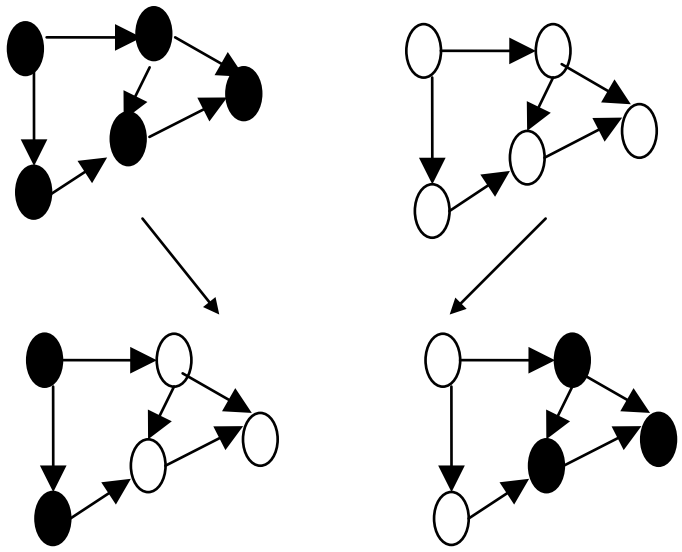


Fig. 2 Illustration du

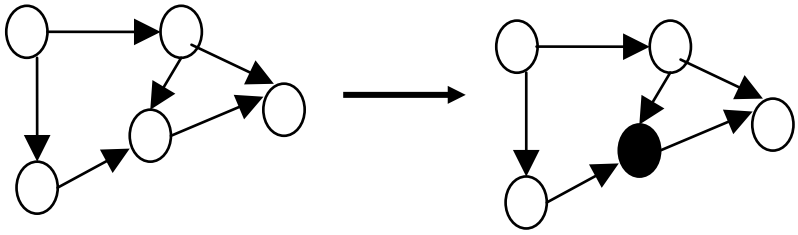


Fig.3 Illustration de la

Notre logiciel d'optimisation lit en entrée la topologie du circuit à analyser puis construit des structures de données dynamiques représentant le circuit sous forme d'un graphe. Par la suite, les opérateurs génétiques sont appliqués pour déterminer une population d'une centaine de circuits, tous équivalents au circuit de départ d'un point de vue fonctionnel mais dont les dimensions des portes sont choisies adéquatement par l'algorithme génétique. Il faudra par la suite évaluer tous ces circuits en calculant la surface totale de chacun d'entre eux ainsi que leur délai de propagation, ce dernier nécessitant la détermination du chemin critique par application d'un algorithme DFS.
Enfin, il faudra choisir les meilleurs circuits à conserver pour créer la génération fututre. Ce travail est alors ré-itéré un certain nombre de fois (de l'ordre d'une centaine) pour pouvoir enfin choisir la meilleure configuration de composants d'un point de vue dimension et qui réalise le meilleur compromis (surface réduite/délai de propagation répondant aux exigences du cahier des charges).

Seulement le temps passé à évaluer une centaine de circuits et ceci pour une centaine de générations de circuits est un temps très appréciable par rapport au temps total d'exécution de notre logiciel d'optimisation. Pour pallier à cet inconvénient, nous avons proposé une approche parallèle de l'algorithme génétique pouvant être implémentée sous environnement de programmation parallèle PVM [1].

4- **Parallélisation de l'algorithme génétique appliqué au problème de dimensionnement :**

4.1 **Introduction :**

Encore récemment, bon nombre d'applications étaient considérées comme impossibles à être mises en œuvre, non pas faute d'une modélisation mathématique exacte ou de méthodes numériques mais faute de calculateurs capables de résoudre ces problèmes, vues les grandes puissances de calculs requises. Ces applications possédaient des représentations mathématiques adaptables aux traitements parallèles mais les machines disponibles n'étaient pas en mesure de mener à bien ce type de traitement. Ce qui a incité des chercheurs à connecter plusieurs processeurs (voir des milliers actuellement) afin d'obtenir une machine plus puissante, capable de traiter en simultaneité de grands flots de données. Ceci n'aurait pu être réalisé si l'on n'avait pas pensé préalablement à développer des logiciels parallèles d'exploitation de ces machines : c'est ainsi que sont nées les architectures à multiprocesseurs et le parallélisme. Ce dernier et plus particulièrement les architectures et algorithmes parallèles sont devenus un domaine incontournable de l'informatique moderne. Leur évolution est très rapide et d'une grande diversité [7]: des supercalculateurs vectoriels aux réseaux de stations de travail, des bases de données réparties à la simulation numérique, des modèles abstraits aux implémentations les plus sophistiquées. Ceci est dû au grand nombre de domaines d'application, qui ne peuvent plus s'en passer, dont : la météorologie, l'imagerie [14], le génie nucléaire, l'aéronautique ainsi que toutes les applications en temps réel d'une façon générale.

Actuellement, nous pouvons dire que le parallélisme a suscité un engouement certain dans la communauté informatique en raison du potentiel de puissance de traitement qu'il représente [10]. Il apparaît aujourd'hui, comme la solution au

problème de rapidité de plus en plus exigé dans les applications scientifiques, industrielles, ... qui ne cessent de demander de plus en plus de performance avec un coût réduit.

Seulement, les machines à multi-processeurs restent encore très coûteuses et difficiles à gérer. Pour cela, les chercheurs dans le monde ont proposé de nouvelles machines parallèles virtuellement : il s'agit de connecter plusieurs processeurs via un réseau local de type LAN pour former une seule machine parallèle. Ce sont des machines parallèles à mémoires distribuées. Il a été développé dans ce but des environnement de programmation parallèles appropriés, tels que PVM (Parallel Virtual Machine), MPI,… Sous ces environnements, une collection de machines séquentielles, vectorielles et parallèles peuvent, en communiquant à travers un réseau local, apparaître comme une super-machine parallèle à mémoire distibuée.. Le terme ″Machine Virtuelle″ désigne cette machine et le terme ″Host″ désigne chaque machine du réseau.

4.2 **Algorithme Evolutionnaire parallèle :**

Nous avons vu, dans ce qui précède, les principes de base d'un algorithme génétique, à savoir qu'il manipule une population de chromosomes, ces derniers représentant chacun, une solution possible du problème traité, ie: un circuit.

Or, de nos jours, la taille d'un circuit (nombre de portes) dépasse le million. La valeur de la fonction ″objectif″ est obtenue en effectuant la somme entre la surface et le délai du circuit multiplié par un facteur de pénalité. Ces calculs doivent être effectué pour chaque chromosome de la génération courante, celle-ci comptant une centaine. Le nombre d'itérations de l'algorithme génétique est également choisi de l'ordre d'une centaine. Il est clair alors, que le temps d'exécution de l'algorithme génétique est consommé essentiellement en évaluations répétitives des individus

des différentes générations , et ce, afin de choisir le meilleur chromosome à la fin de ce processus.

T. BACK et al [11] ont montré que 95% du coût de l'algorithme génétique est consommé durant les évaluations de tous les individus des différentes générations.

Pour pallier à cet inconvénient, nous avons choisi de calculer en parallèle la valeur de la fonction ″objectif″ F en chaque chromosome d'une génération. Pour cela, nous avons choisi d'implémenter notre algorithme génétique parallélisé selon un schéma maitre/esclave [12], illustré par la figure Fig. 4

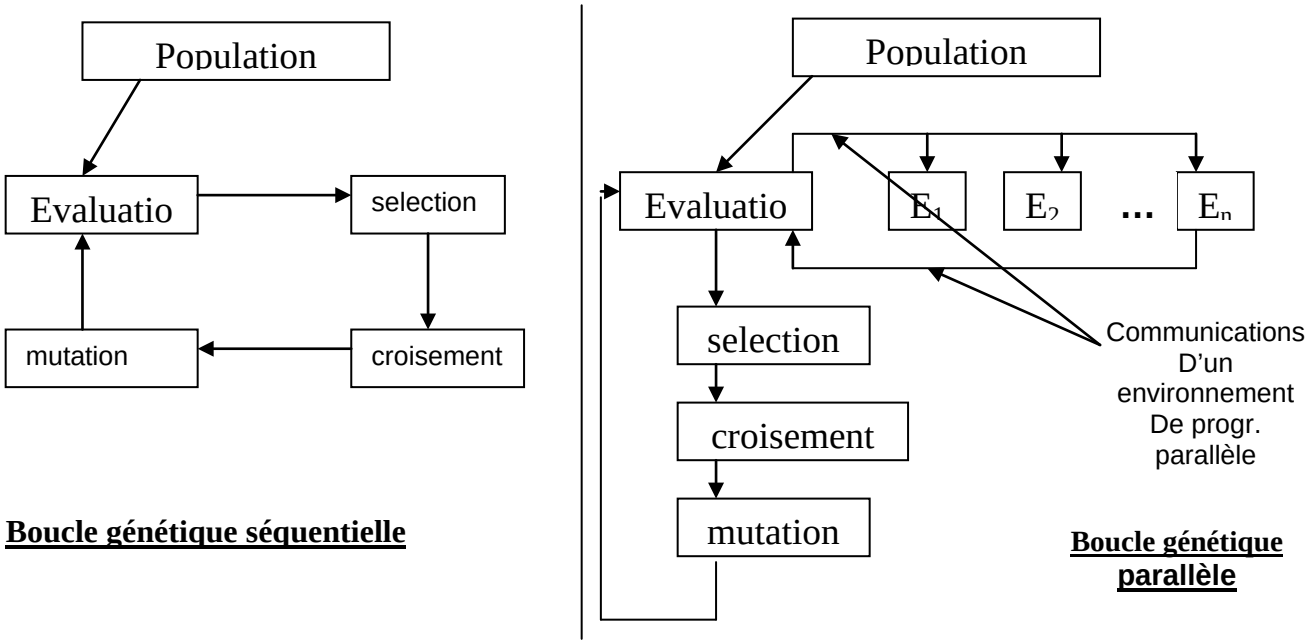


Fig. 4: Illustration de la parallélisation de l'évaluation des individus d'une génération

Le schéma d'implémentation est relativement simple. Nous pouvons le résumer comme suit :

> Sur une machine parallèle virtuelle constituée d'un cluster de P machines connectées via un réseau local, communiquant sous environnement parallèle, l'une de ces machines est déclarée Maître ou *″Main″*, les (P-1)

autres sont déclarées Esclaves ou *"Slaves"*. Le logiciel d'optimisation d'un circuit intégré commence à dérouler ses étapes sur la machine Maître, à savoir la construction dynamique des structures de données qui représentent un circuit en mémoire, puis lance les étapes de l'algorithme génétique. La génération d'une population initiale est effectuée alors, puis il y a envoi des messages vers les autres machines esclaves qui doivent commencer l'évaluation des chromosomes de la 1ère génération.

Les structures de données sont également construites au fur et à mesure sur les processeurs Esclaves pour éviter leur envoi à partir du Maître à travers le réseau et ceci pour les raisons suivantes :

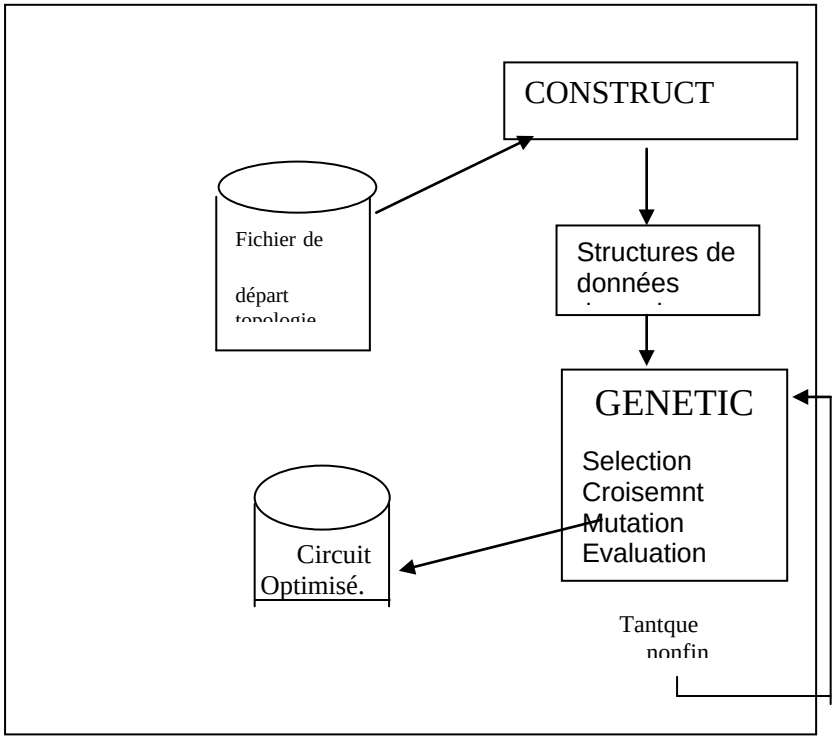
- Eviter de saturer le réseau par un trafic intense des données qui peuvent être utilisées localement sur les autres machines.
- Les données construites dynamiquement (par exemple par Malloc en langage C) contiennent des adresses physiques locales à la mémoire du processeur Main et n'ont donc aucune réalité physique sur les autres machines du réseau.

Dans la mémoire des processeurs Esclaves doit figurer également le code exécutable de la procédure d'évaluation des chromosomes.

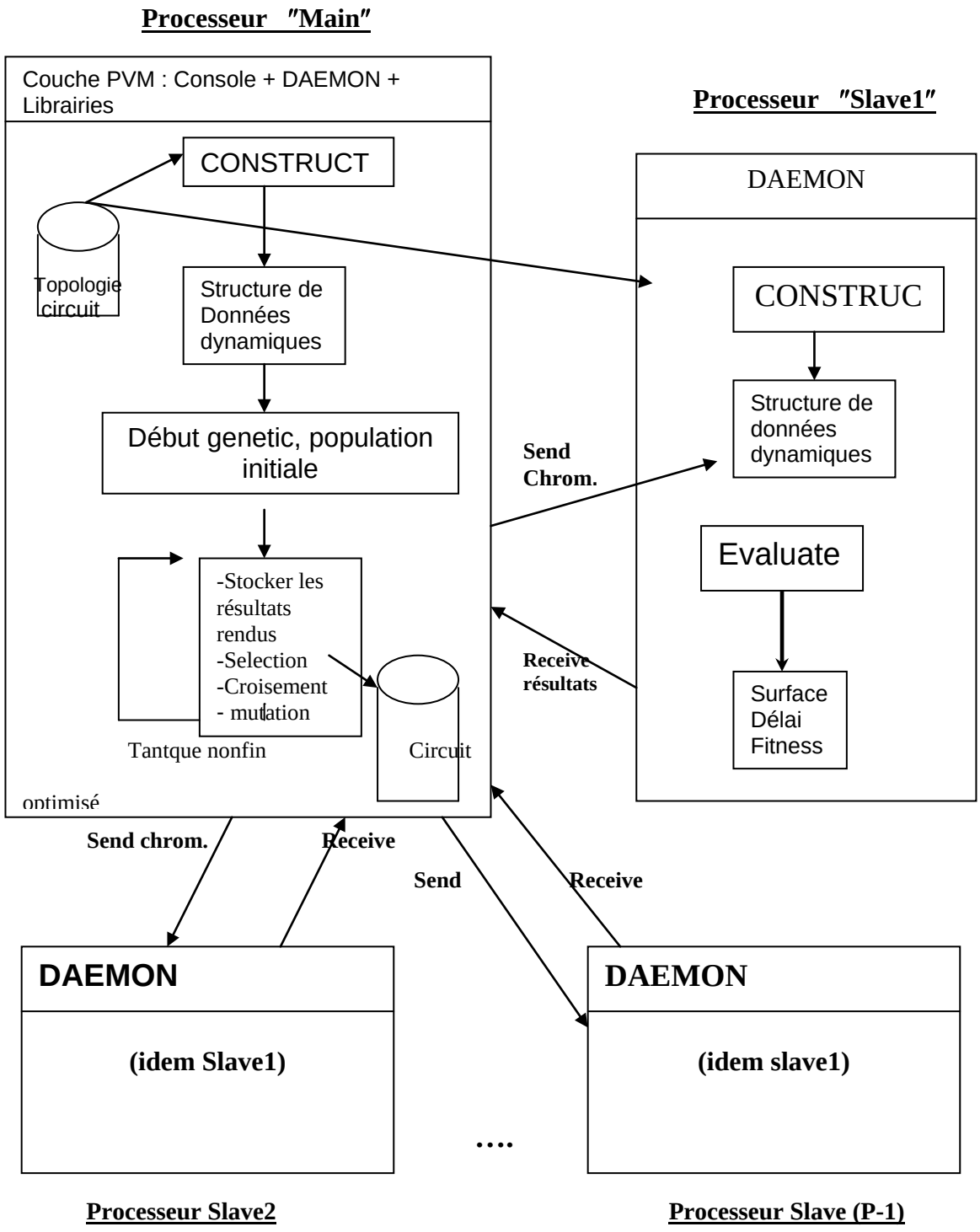
Nécessairement, le nombre de processeurs Esclaves (P-1) est inférieur au nombre N d'individus d'une génération. Pour cela, nous pouvons choisir la taille d'une population N multiple de (P-1) pour pouvoir effectuer des évaluations successives par paquets de (P-1) chromosomes qui sont envoyés vers les processeurs Esclaves pour leur évaluation, puis récupérés par le processeur Main. En réalité, si la machine parallèle virtuelle est constituée d'un réseau de machines hétérogènes, tous les processeurs disponibles n'ont pas la même vitesse d'exécution. Il faudra alors que notre algorithme parallèle prenne en charge cette éventualité afin de minimiser

les temps morts des processeurs Esclaves (*"Idle time"*). D'où, la machine Maître procède par envoie/réception de (chromosome)/(valeur de la fonction objectif) vers/de la machine Esclave "libre" à un moment donné, sans distinction.

La figure Fig.5 illustre un schéma séquentiel d'exécution de notre logiciel d'optimisation des circuits intégrés VLSI en a) et un schéma de proposition d'exécution parallèle pour le même logiciel en b) illustrant les échanges entre le "Main" et les différents "Slaves" de l'architecture parallèle à mémoire distribuée.



a) Schéma d'exécution séquentielle



b) Schéma global des échanges entre Processeur ″Main″ et Processeurs ″Slaves ″ dans le cas d'un environnment parallèle de type PVM

Fig. 5 : Schémas Séquentiel et parallèle d'évaluation : Illustration des échanges entre les machines dans le cas parallèle.

Après avoir vu comment se font les échanges entre les différents processeurs qui constituent une machine virtuelle parallèle, à présent, nous allons énoncer un algorithme génétique parallèle en pseudo-code, qui résume toutes ces étapes précédemment décrites.

DEBUT /*On dispose de (P-1) machines Esclaves, N=taille d'une population*/
Initialize Pop=(X_1,X_2,…, X_N) ;
FOR i=1 to (P - 1) DO
SEND X_i ⟶ Sl_i ;

FOR i=P à N DO
RECEIVE (X,F(X)) ⟵ Sl_γ ; /*Sl_γ représente le 1er processeur esclave libre.*/
SEND X_i ⟶ Sl_i ; /* γ est une valeur arbitraire de l'intervalle [1..P-1]*/
ENDDO ;

WHILE ($i<i_{max}$) DO
RECEIVE (X,F(X)) ⟵ Sl_γ ;
Select ;
Crossover : X'=r(P) ; /*recombiner*/
Mutation : X″=m(X') ;
SEND X″ ⟶ Sl_γ ;
i=i+1 ;
ENDDO

FOR i=1 to (P- 1) DO
RECEIVE (X,F(X)) ⟵Sl_γ ;
Select ;
ENDDO

Return ″The Best″ ;

FIN.

Après création de la population initiale, la 1ère boucle FOR envoie aux (P-1) machines Esclaves des chromosomes à évaluer. La 2ème boucle FOR commence à récupérer les résultats (X,F(X)) à partir du 1er Esclave Sl_γ qui a fini ses calculs, γ étant une valeur arbitraire comprise entre 1 et (P-1). Ce processeur redevenant libre, on lui renvoie un nouveau chromosome à évaluer. Le ″Main″ continue à recevoir les paquets ″résultats″ et prend en charge les autres opérateurs génétiques, jusqu'à un nombre d'itération i_{max}. Finalement, dans la dernière boucle, nous récupérons les résultats à partir des machines Esclaves de la dernière itération pour pouvoir choisir le meilleur chromosome ″the best″ .

Comme précédemment spécifié, cet algorithme prend en compte l'hétérogeinité des machines constituant le réseau en minimisant le temps d'attente des machines ″Slaves″. En effet, il procède par envoi/réception de chromosomes/résultats de n'importe quelle machine ″libre″ du réseau.

D'autre part et comme on peut le remarquer, notre modèle Maître/Esclave est aussi un modèle SPMD *″Single Processus/Multiple Data″*. En effet, le même code est exécuté par les processeurs ″Slaves″ sur un flot de données différentes : ces machines évaluent de la même manière un ensemble d'individus constituant plusieurs générations. Nous pouvons conclure donc que les tâches sont équitablement réparties entre les différentes machines du réseau.

5. Conclusion générale

Les résultats obtenus indiquent qu'à priori la puissance de l'algorithme génétique, trouve pleinement sa confirmation lors de son application à un problème complexe tel que celui du dimensionnement des circuits VLSI.

Le problème du nombre important de paramètres des circuits, la difficulté de modéliser avec précision toutes les caractéristiques des circuits intégrés ont, grâce à la grande souplesse de l'algorithme génétique, pu être pris en compte. Quand aux performances, celles ci sont à améliorées grâce à la propriété de parallèlisation intrinsèque de l'algorithme génétique et d'une implémentation ultérieure lorsque la machine parallèle réelle ou virtuelle sera disponible localement.

Ce travail a été testé avec succès sur des circuits *"Benchmarks"* de type ISCAS'85[13] Le logiciel d'optimisation en séquentiel a été développé sur PC/PentiumII en langage C sous environnment VisualC++ 6.0 et qui a donné des résultats appréciables en un temps peu significatif.

Ce travail a fait l'objet de deux communications internationales [17] [18], la première renfermait une solution génétique au problème du "Gate Sizing", la deuxième une approche parallèle génétique au même problème. Ces deux conférences ont eu lieu en octobre 2000, la première du 22 au 25octobre au MAROC, la seconde du 30 au 02 Novembre en IRAN.

REFERENCES BIBLIOGRAPHIQUES

[1] Al Geist, J. Dongarra et al,
PVM 3 User's guide and référence manual, Engineering Physics and Mathematics Division, Mathematical Section, Oak Ridge National Laboratory, 1994.

[2] PACK P. CHAN,
Algorithms for library-specific sizing of combinational logic,
27 th IEEE Design Automation Conference, pp 353-356, 1990.

[3] M. Moon et al,
A path oriented algorithm for the cell selection problem,
IEEE Transactions on computer Aided Design, pp296-307, March 1995.

[4] D. Goldberg,
Genetic algorithms in search, optimization and machine learning,
Addison-Wesley, Reading, MA, 1989.

[5] Z. Michalewicz
Genetic Algorithms + data structures = Evolutionary programs, Springer-Verlag, 1996.

[6] W. Wolf, Modern VLSI Design, Prentice Hall, 1994.

[7] F.Thomson Leighton
″Introduction aux algorithmes et architectures parallèles″,
Morgan Kaufmann Publishers, 1992.

[8] J. Dongarra.
EuroPVM'95. Parallélisme, réseaux et répartition.
France, Sep. 1995.

[9] Second European PVM Users' Group Meeting.
PVM tutorial. Lyon, France, Sep. 1995.

[10] Thibault DUBOUX,
Régulation dynamique du partitionenement de données sur machines parallèles à mémoire distribuée, these de doctorat en informatique, Ecole Supérieure de Lyon, 1996.

[11] Thomas Back, Thomas Beielstein, Boris Naujoks ,
Evolutionary algorithms for the optimization of simulation models using PVM, in proceeding of EURO PVM' 1995.

[12] P. Lyaet, C. Baranger, The Use of PVM with Workstation Clusters in Application of Genetic Optimisation Techniques, Université de Technologie de Compiègne , Division Modèles Numériques en Mécanique.

[13] F. Berglez and H. Fujiwara,
Aneutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN, ISCAS'85, Juin 85.

[14] F. Boumghar, Parallélisation d'algorithmes de traitement d'images 2D et 3D,
Thèse de doctorat d'état en électronique, Juin 1998.

[15] A.R Baba-ali, Optimisation des performances temporelles des circuits intégrés VLSI, Thèse de doctorat d'état en electronique, Mars 1998.

[16] H. Holland, Adaptation in Natural and artificial systems,, Ann Arbor, The university of Michigan Press, 1975.

[17] **S. Benkhider**, F. Boumghar & A.R Baba-ali,
″An Evolutionary Approach For The Gate Sizing Problem Of Standard Cells VLSI Integrated Circuits″, Conférence Internationale sur les Mathématiques Appliquées et les Sciences de l'Ingénieur CIMASI'2000, MOROCO Oct. 2000.

[18] **S. Benkhider**, F. Boumghar & A.R Baba-ali,
″A Parallel Genetic Approach to The Gate Sizing Problem Of VLSI Integrated Circuits International Conference on Microelectronic ICM'2000, IEEE, IRAN Nov. 2000.

Introduction

Dans le monde de la synthèse de circuits intégrés, nous assistons actuellement à une double évolution. D'une part, ces circuits deviennent d'une grande diversité et leurs tailles en terme de composants ne cesse d'augmenter. D'autre part, la vitesse de fonctionnement est en continuelle augmentation.
En effet, la demande en circuits rapides est sans cesse croissante, dans les domaines de la recherche, de l'industrie, etc… Citons à titre d'exemple les microprocesseurs dont la taille est passée de quelques dizaines de milliers de transistors à quelques dizaines de millions de transistors en un laps de temps très court, ainsi que leurs fréquences d'horloge qui sont passées du MHz au GHz durant cette dernière décennie.

Cette évolution spectaculaire ne cesse de confronter les concepteurs de circuits intégrés à de sérieux problèmes, telles que la consommation d'énergie qui augmente avec le nombre de composants et par conséquent la dissipation de chaleur qui en découle peut provoquer des dysfonctionnements de ces circuits. Le temps de conception est aussi un facteur crucial pour minimiser les coûts.

Ce double impératif impose aux concepteurs de circuits intégrés des efforts considérables. Tel qu'illustré par la figure 1, la conception de circuits intégrés passe par plusieurs phases, chacune d'elles nécessitant des outils de CAO/VLSI très puissants.

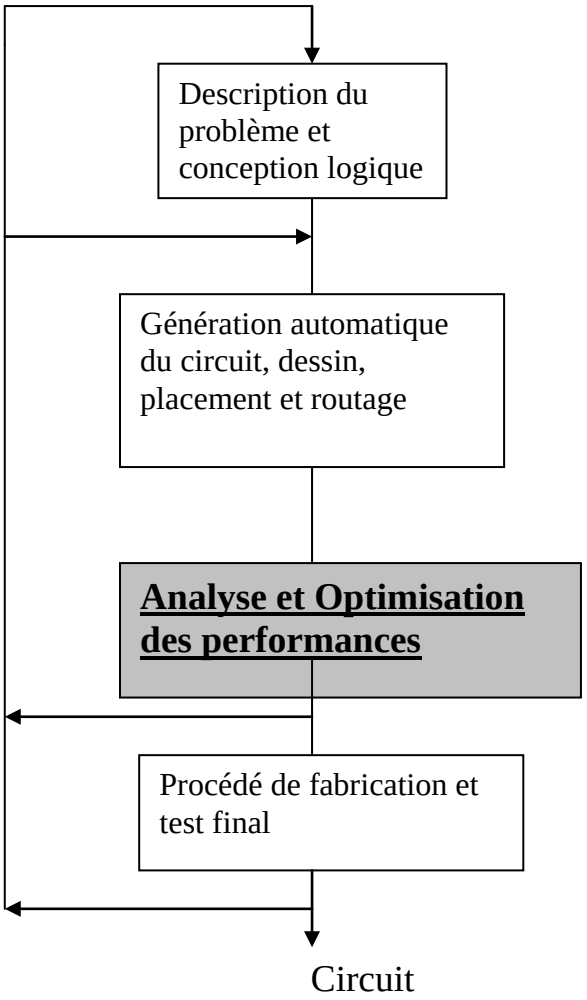


Figure1 : Schéma global du processus de conception de circuits intégrés

Ces outils de CAO ont recours à des algorithmes qui doivent être très performants. En effet, la très grande taille des circuits intégrés, leur complexité et leur diversité, les contraintes posées par les cahiers des charges font que leur modélisation n'est pas toujours aisée. Les circuits intégrés possèdent plusieurs niveaux de représentation au sein desquels plusieurs configurations sont possibles, avec chacune un nombre élevé de paramètres. Par conséquent, l'espace des solutions devient infini pour un circuit donné. Il en résulte que la plupart des problèmes liés à la CAO/VLSI, notamment les algorithmes d'optimisation de performances, sont considérés comme difficiles et sont classés NP-Difficiles (*NP-Hard*). Actuellement, des outils EDA (Electronic Design

Automation) ont été développé pour faciliter la tâche aux concepteurs de circuits intégrés.

Dans le cadre de notre travail, nous nous sommes intéressés particulièrement à l'optimisation des performances temporelles et spatiales d'un circuit intégré. Traditionnellement, cette tâche était confiée aux concepteurs qui, par expérience, optimisaient les circuits de façon empirique, ce qui n'était pas sans conséquences sur la qualité de ces derniers. Actuellement, des outils performants d'optimisation sont développés dans ce but.

Parmi les différents problèmes d'optimisation de circuits intégrés, nous distinguons le problème de dimensionnement de portes logiques ou *"Gate Sizing"*. Un circuit intégré est composé d'un grand nombre de portes logiques. Dans ce problème, chaque porte logique disponible dans la bibliothèque de composants possède plusieurs versions fonctionnellement équivalentes mais de dimensions différentes. Chaque version est caractérisée par une surface et une vitesse différente. Il est important de noter que ces deux paramètres sont antagonistes. En d'autres termes, plus une porte sera rapide et plus sa surface sera importante ; par conséquent le coût du circuit sera grand. De plus, si nous réduisons le coût nous détériorons la vitesse.

Pour cela, il s'agit de déterminer un compromis optimal entre la surface et la vitesse, pour chaque porte logique du circuit. Si la porte est choisie de petite dimension le circuit sera lent, par contre si la porte est choisie de grande dimension le circuit sera rapide mais coûteux et consommera plus d'énergie. Ce problème consiste pour chaque porte du circuit à lui déterminer sa dimension optimale qui réalise le meilleur compromis : rapidité/coût du circuit. Par conséquent, un circuit comportera deux ou plusieurs types de portes logiques fonctionnellement équivalentes mais différentes du point de vue dimension et délai de propagation.

Etant donnée la taille de plus en plus élevée des circuits intégrés, ce problème appartient à la classe des problèmes NP-difficiles ou NP-Hard [21], ce qui

signifie qu'il n'existe pas d'algorithme en mesure de déterminer la solution optimale efficacement. En d'autres termes, il n'existe pas d'algorithme polynômial pour résoudre ce problème. Actuellement, les approches les plus efficaces sont des heuristiques qui, en un temps polynômial vont tâcher de déterminer une solution acceptable à ce problème en un temps raisonnable.

Dans le cadre de cette thèse, nous nous proposons d'expérimenter pour la première fois une approche basée sur une méta-heuristique, qui est l'algorithme génétique. Ce dernier est connu pour sa robustesse et son efficacité dans la résolution de problèmes d'optimisation complexes. Il s'inspire des mécanismes naturels de la sélection et manipulent une population d'individus représentant chacun une solution potentielle au problème traité.
En outre, ils présentent une nature parallélisable. La taille des circuits étant très grande, les performances en temps d'exécution de leur optimisation nécessite la parallélisation de l'approche génétique, ce qui améliore considérablement ses performances déjà respectables, en effectuant de manière concurrente les différentes opérations génétiques.

Le travail présenté dans cette thèse s'inscrit dans le cadre de l'optimisation temporelle (Timing) et spatiale (Sizing) des circuits intégrés VLSI. Il s'articule autours de cinq chapitres :
Le 1er chapitre décrit les fondements généraux de la VLSI. Quelques définitions de concepts y sont présentés afin de faciliter leur compréhension ultérieure. Comme un circuit est représenté en mémoire par un graphe, nous avons jugé utile de rajouter quelques concepts de la théorie des graphes ainsi que de la complexité des algorithmes.
Dans le 2ème chapitre, il est présenté les principes de base d'un algorithme génétique ainsi que quelques différences avec les méthodes de résolution classiques.

Le chapitre III renferme la position du problème de dimensionnement de circuits intégrés ou *"Gate Sizing"* ainsi qu'un état de l'art des travaux effectués dans ce domaine, tandis que le chapitre IV proposera les détails de résolution de ce problème par un algorithme génétique : la formulation mathématique, le codage des paramètres, les opérateurs génétiques,…ainsi que le logiciel d'optimisation développé dans ce contexte. Les tests ayant été menés sur des circuits *"benchmark"*, quelques tableaux de résultats obtenus seront exposés, montrant le conflit surface/rapidité des circuits.
Enfin le chapitre V étudie le parallélisme ainsi que quelques architectures parallèles. Des schémas de parallélisation d'un algorithme génétique sont présentés afin de déduire une approche parallèle génétique pour la résolution du problème de dimensionnement de circuits intégrés VLSI.

Pour compléter et clarifier, nous présentons en annexe A quelques définitions de la VLSI et en annexe B un environnement de programmation parallèle : PVM (Parallel Virtual Machine) qui permet de relier un cluster de machines hétérogènes et de les faire communiquer via un réseau local, afin d'obtenir une seule machine parallèle virtuellement.

Chapitre I

Généralités et Définitions de Concepts de la VLSI.

1- Introduction

Tout ordinateur est conçu à partir de circuits de base, dont le comportement fonctionnel est décrit par l'algèbre binaire. La conception d'un circuit intégré passe par plusieurs niveaux d'abstraction [1]. Chaque passage nécessite plusieurs opérations complexes qui souvent introduisent des erreurs. La détection de ces erreurs est une tâche longue et ardue étant données la taille et la complexité de ces circuits.

Dans le présent chapitre, nous définissons plusieurs concepts importants dans le but de simplifier leur compréhension ultérieure. Ainsi, nous abordons les définitions de la CAO/VLSI et des circuits intégrés.

Ces derniers sont souvent représentés en mémoire par des graphes. Pour cela, nous avons jugé utile de présenter quelques rappels sur la théorie des graphes ainsi que les algorithmes d'exploration des graphes. Par ailleurs, la complexité des circuits VLSI et leurs tailles nécessitent le recours à des heuristiques de CAO pour effectuer les différents traitements en mémoire sur ces circuits, ce qui nous a amené à définir brièvement quelques

concepts de la complexité des algorithmes. Pour clarifier, d'autres notions seront définies en annexe A.

2- <u>Définition d'un circuit VLSI</u> : (Very Large Scale Integration) est un circuit intégré (CI) à très grande échelle qui caractérise généralement une fonction complète d'un système. C'est un dispositif qui intègre actuellement plusieurs millions de transistors. Il est constitué essentiellement par un morceau de silicium monocristallin (puce ou *"chip"*) [2] ayant la forme d'un parallélépipède rectangle d'une hauteur typique de 250 microns. La surface du parallélépipède, portant le circuit proprement dit, peut mesurer de 1 à 100 mm^2, sur laquelle, on crée des composants électroniques. Ceci est effectué en introduisant localement des impuretés ohmiques convenablement dosées, en déposant localement des couches isolantes Si() ou bien conductrices (aluminium, silicium dopé), de manière à y former un circuit électrique correspondant à certaines spécifications.

Le fonctionnement logique du circuit est déduit à partir des fonctions des portes logiques (voir annexe A) de base et les transitions présentes à ses entrées.

3- <u>Définition de la CAO/VLSI et de ses outils</u>

3-1 <u>Définition de la CAO/VLSI</u> :

La CAO (Conception Assistée par Ordinateur) [3] [4] est un ensemble de systèmes, matériels et logiciels, où l'ordinateur intervient pour accomplir très rapidement des tâches de calcul dans le but d'aider l'utilisateur à prendre des décisions logiques tout en le laissant introduire son art, son expérience et son intuition.

La CAO/VLSI utilise plusieurs logiciels de conception qui sont passés par plusieurs phases de développement. Au départ, des logiciels furent développés au fur et à mesure des besoins sans qu'il existe réellement un lien entre eux. Ce qui a eu pour conséquence d'aboutir à un nombre

important d'interfaces différentes développées. Par la suite, les concepteurs réalisèrent des logiciels intégrés organisés autour d'une même base de données, constituant ainsi des ensembles d'outils cohérents.

3-2 Les outils de la CAO/VLSI :

Les outils de la CAO [3] peuvent être divisés en deux grandes classes : les outils de synthèse et les outils d'analyse.

- Les outils de synthèse : ont été crées pour la génération automatique de la circuiterie. Ils regroupent des outils d'aide graphique pour le tracé des masques, des logiciels d'autoroutage pour l'interconnexion des blocs du circuit, des logiciels de placement des blocs du circuit et enfin des logiciels d'aide à la génération du *"layout"* (plan ou masque de fabrication).
- Les outils d'analyse : une fois le circuit dessiné, il faut procéder à la vérification de ses performances, à savoir s'il réalise bien les tâches escomptées et les contraintes fixées par le cahier des charges,... C'est la phase de validation.

La synthèse d'un CI exige le partitionement des phases de conception pour alléger la phase de sa réalisation.

En d'autres termes, le concepteur concentre au début ses efforts sur une certaine tâche, ensuite il décompose cette tâche en sous-tâches qui sera alors décomposée en éléments plus petits jusqu'à atteindre le niveau composant, non partitionnable.

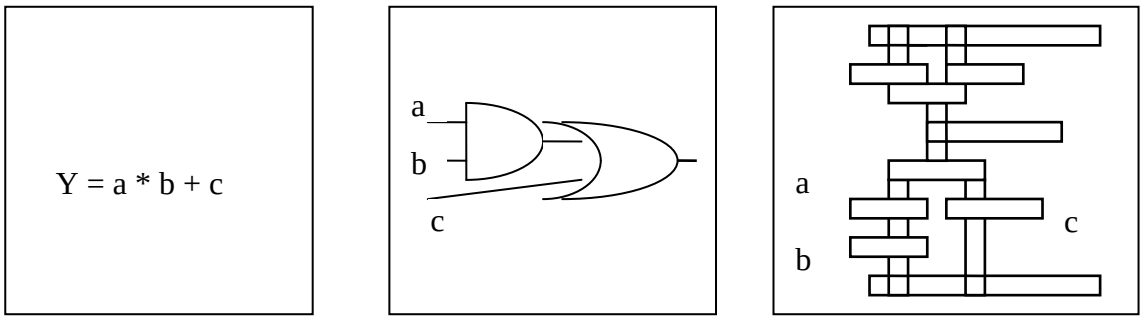
On dit alors qu'un CI passe par plusieurs phases d'abstraction chacune d'elle possédant une hiérarchie de détails selon le choix du concepteur.

En général, la conception d'un CI est exprimée en terme de propriétés : comportementale, structurelle et physique, comme le montre la figure 1.1 . La section suivante présentera les domaines de représentation d'un circuit intégré VLSI.

3-3 Les domaines de représentation d'un circuit intégré :

- Le domaine comportemental : (ou fonctionnel)

Le concepteur ne s'intéresse qu'à ce que fait le circuit (sa fonction). Ce dernier est représenté sous la forme d'équations logiques, et ce en langage évolué. Il est considéré comme une boite noire ayant une interface (des entrées et des sorties) ainsi que des fonctions qui décrivent le comportement de ses sorties en fonction de ses entrées.



a) domaine comportemental b) domaine structurel c) domaine physique ou géometrique

Fig. 1.1 Illustration des trois domaines de représentation d'un circuit intégré

- Le domaine physique : (ou géométrique)

Le circuit est représenté sous forme de ″layout″ (ou schéma) en ignorant le plus possible ce que fait ce circuit. Ses éléments qui sont généralement utilisés en représentation géométrique, sont des niveaux : polygones, cellules, puces, boards et cabinets. La figure 1.2 montre le schéma physique d'une porte de transmission CMOS.

A ce niveau, le concepteur dispose d'éditeurs graphiques évolués qui permettent de gérer la grande masse de rectangles du ″*layout*″ ainsi que leur organisation hiérarchique tout en offrant des options de visualisation très sophistiquées.

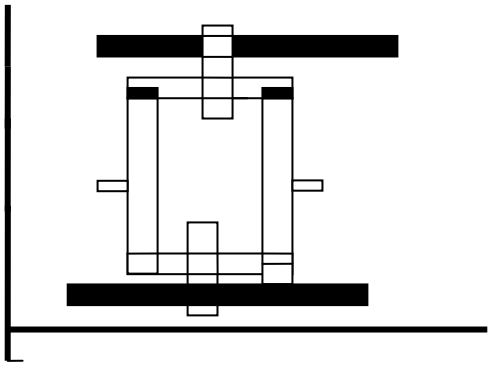


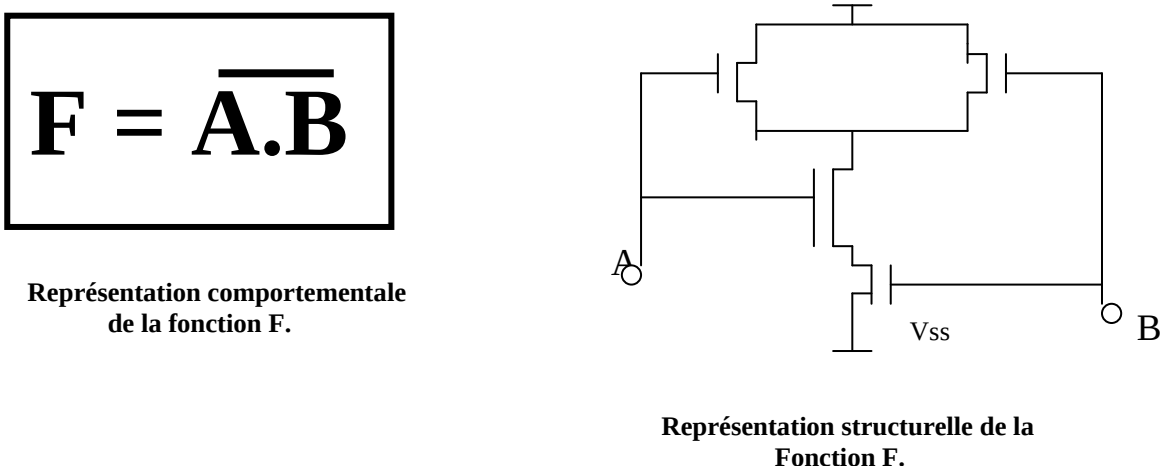
Fig.1.2 : Représentation géométrique de la porte de transmission CMOS.

L'importance du domaine physique, au point de vue performances est fondamentale, puisque celles-ci sont déterminées par les caractéristiques des représentations des CI dans ce domaine. En effet, à partir des composants élémentaires, la surface totale du circuit est déterminée d'une part, et le courant qui passe à travers eux d'autre part. En conséquence, le délai de propagation et la dissipation d'énergie qui dépendent de ce courant, sont déterminés également. De plus, la longueur des interconnexions qui n'est connue que lorsque le circuit est réalisé, donc complètement décrit au niveau du domaine physique, est très importante. C'est cette longueur qui va générer les capacités d'interconnexion indispensables pour la détermination des charges attaquées par les composants. Ces charges ont un effet déterminant sur la vitesse de fonctionnement du circuit.

- Le domaine structurel :

A ce niveau, la structure du circuit est définie en décrivant les interconnexions des éléments le constituant.

Les éléments de base au niveau circuit sont les transistors, les résistances et les capacités. Au niveau logique, les éléments de base sont les portes et les bascules.



Représentation comportementale de la fonction F.

Représentation structurelle de la Fonction F.

Fig. 1.3 Illustration des domaines comportemental et structurel de la fonction F= A.B

L'élément de base du niveau structurel est le diagramme du circuit qui montre les comportements et les interconnexions.

Durant cette phase [5], le concepteur dispose d'outils de description généralement graphiques qui lui permettent de décrire un circuit sous des formes diverses (registres, portes, transistors ...). Une fois le circuit décrit, le concepteur utilise une batterie de simulateurs et d'analyseurs pour valider le circuit et pour l'analyser en termes de performances, de consommation, etc... . A ce stade , le concepteur dispose d'outils de génération automatique de structures régulières qui fournissent le *"layout"* de circuit moyennant l'indication du type de circuit et ses paramètres.

Actuellement le langage de description utilisé pour décrire un circuit intégré est le langage VHDL [6] (Very High Description Langage). C'est un langage de description, de simulation et de synthèse de matériel électronique. Il supporte un mélange de niveaux de description : on peut mélanger dans la réalisation du même circuit électronique, une description comportementale (Behavior) avec une description structurelle (Structural) ou une description logique.

Dans ce qui suit, nous présentons un exemple de description d'un comparateur à un bit en langage VHDL. En (a), la fonction interface est déclarée. En (b), une description du circuit en mode comportementale est présentée. En (c), la description du même circuit est présentée en *"Dataflow"* ie : en équation logique alors qu'en (d), c'est une description structurelle qui est présentée.

```
Entity COMPARE is
  Port (A,B : in BIT ; C : out BIT) ;
End COMPARE ;
```

(a)

```
Architecture BEHAVIOR of COMPARE is
Begin
 Process(A,B)
 Begin
     If (A=B) then C≤'1' ; else C≤'0' ;
     EndIf ;
  End Process ;
End BEHAVIOR ;
```

(b)

```
Architecture DATAFLOW of COMPARE is
Begin
    C≤ not(A xor B) after 10ns ;
End DATAFLOW :
```

(c)

```
Architecture STRUCTURE of COMPARE is
   Component XOR_Gate
      Port(I0,I1 : in BIT ; O : out BIT ;
   End component ;
   Component NOT_Gate
       Port(I0 : in BIT ; O : out BIT) ;
   End component ;
   Signal NET_I : BIT ;
Begin
  U0 : XOR_Gate port map(I0=>A, I1=>B, O=>NET_I) ;
  U1 : NOT_Gate port map(I0=>NET_I, O=>C) ;
End STRUCTURE
```

(d)

La description d'un circuit dans ce langage ne s'effectue qu'en un seul mode au choix du concepteur ie : soit en Comportemental, soit en Dataflow ou bien en structurel.

Etant donnée la taille et la complexité des circuits intégrés actuellement, les concepteurs ne peuvent plus mener à bien toutes les étapes de conception sans l'aide d'outils de la CAO. Pour cela, une représentation adéquate en mémoire de ces circuits est nécessaire.

4- <u>Représentation en mémoire des circuits intégrés</u> :

Actuellement, la conception des circuits intégrés n'est possible qu'avec des logiciels de CAO, du fait de leur taille et de leur complexité. Pour cela, une représentation en mémoire centrale de ces circuits est indispensable, et ce sous forme de structures de données qui permettent aux programmes de la CAO/VLSI d'effectuer aisément les traitements envisagés sur ces circuits. Il existe plusieurs représentations possibles en fonction du domaine de

représentation et du type d'application envisagée. Parmi les représentations les plus répandues, se trouvent :

♦ Le CDFG (Control Data Flow Graph).

♦ Le BDD (Binary Decision Diagramm).

♦ Le réseau booléen (Boolean Network) ; ...

On a souvent recours à une représentation à l'aide de graphes pour symboliser un circuit en mémoire. Pour cela, nous présentons dans ce qui suit, un rappel de quelques définitions de la terminologie de la théorie des graphes.

5- <u>Rappels et terminologie des graphes</u> : [8]

Un graphe G = [X,U] est déterminé par :

1 - un ensemble X dont les éléments sont appelés sommets. Si N=card (X) est le nombre de sommets, on dit que le graphe G est d'ordre N.

2- un ensemble U dont les éléments $u \in U$ sont des couples ordonnés de sommets appelés des arcs. Si u=(i,j) est un arc de G, i est appelée extrémité initiale et j est appelée extrémité terminale. Dans ce cas le graphe est dit orienté.

Dans le cas où l'ordre des couples de sommets n'est pas important, le graphe est dit non-orienté et les arcs sont plutôt des arêtes.

- Un graphe est dit simple si :
  - il est sans boucles.
  - il n'y a jamais plus d'une arête entre deux sommets quelconques.

- Un graphe est dit valué si les sommets et/ou les arêtes possèdent un poids (une valeur).

- Une arête est dite incidente à un sommet si ce sommet est une de ses extrémités.
- Le degré d'un sommet est le nombre d'arêtes incidentes à ce sommet.
- Deux sommets sont dits adjacents s'il existe une arête les reliant. On dit que cette arête les connecte.

Chemin d'un graphe : un chemin P de longueur q est une séquence de q arcs :

$$P = (u_1, u_2, ..., u_q) \text{ avec } u_1=(i_0,i_1),\ u_2=(i_1,i_2), \ldots u_q=(i_{q-1},i_q)$$

Autrement dit, un chemin est une chaîne dont tous les arcs sont orientés dans le même sens. Le sommet i_0 est l'extrémité initiale du chemin P, le sommet i_q est l'extrémité terminale du chemin P.

Pour effectuer des recherches de chemins dans un graphe, il existe plusieurs méthodes d'exploration. Quelques unes sont sommairement décrites dans la section suivante.

6- Les méthodes d'exploration dans un graphe [8]:

6.1- Définition : On appelle exploration d'un graphe tout procédé déterministe qui, durant l'examen exhaustif des sommets d'un graphe permet de choisir à partir du dernier sommet visité le prochain sommet à visiter.

Parmi les méthodes d'exploration les plus utilisées pour l'analyse d'un circuit intégré représenté sous forme d'un graphe, nous citons :

6-2 L'exploration en largeur DFS (Depth First Search) :

C'est une technique puissante d'exploration en largeur. Elle permet d'explorer systématiquement tous les arcs d'un graphe de sorte que chaque arc ne soit exploré qu'une seule fois et chaque sommet visité au moins une fois.

Cette technique appelée ″Backtracking on a graph″ a été développée formellement et utilisée par Hopcroft et Tarjan [12].

A partir d'un sommet v visité, on se déplace vers un sommet w adjacent non encore visité auparavant et ce, le plutôt possible, en quittant v qui peut posséder des arcs non explorés précédemment. Autrement dit, on trace un chemin à travers le graphe passant par un nouveau chemin à chaque fois que cela est possible. Cette méthode est très utilisée car elle simplifie la recherche sur un graphe.

En contrepartie, cette technique présente un inconvénient majeure, à savoir un temps de calcul plutôt long.

Les étapes de l'algorithme DFS :

Cet algorithme utilise une structure de pile.

```
Début
- initialiser la pile ;
- prendre un sommet quelconque pour sommet de départ,
  u = sommet de départ ;
- empiler(u) ;
- Flag(u) = Vrai ;  /*sommet ayant été visité*/
- Tant que Pile(NonVide)
    Faire
    |       u=dépiler() ;
    |       Pour chaque successeur(u) # NIL
    |       Faire
    |       |     Si flag(successeur(u)) = Faux  /sommet non encore exploré/
    |       |     |  Alors   empiler (successeur(u)) ;
    |       |     |          Flag(successeur(u)) = Vrai ;
    |       |       Fsi ;
    |       Fait ;
    Fait
FIN.
```

6-3 La technique HDFS : (Hierarchical DFS)

Cette technique, présentée dans [10], tend à diminuer l'explosion du nombre de chemins de la méthode précédente, de la manière suivante:

Le graphe est décomposé en un ensemble de sous graphes. Chacun d'eux est analysé indépendamment . Les résultats sont propagés au plus haut niveau de la hiérarchie où chaque sous graphe est considéré comme un seul nœud. Il est évident que la complexité de cette technique est de l'ordre de sous graphes analysés, ce qui a pour effet la réduction du temps de calcul par rapport à l'approche DFS.

Seulement, la difficulté rencontrée en essayant de prendre en compte la fonctionnalité du circuit a fait que cette technique ne soit pas souvent utilisée en pratique.

6-4 L'exploration en profondeur : BFS (Breath-First Search) [12]

A partir d'un sommet v, on explore tous les arcs incidents à v, puis on se place au sommet adjacent w. En w, on explore tous les arcs incidents à w. Ce processus est répété jusqu'à ce que tous les sommets du graphe soient explorés.

Cette technique a pour avantage d'être plus rapide que la méthode d'énumération de chemins. En VLSI, elle permet de calculer le délai à travers chaque sous bloc une seule fois, mais elle n'offre aucune information spécifique aux chemins non critiques du bloc.

Les étapes de l'algorithme BFS :

Cet algorithme utilise une structure de file.

Début

Initialiser la file ;

Enfiler(sommet racine) ;

```
Tant que File(nonVide)
Faire u=défiler() ;
    Pour chaque successeur v de u
    Faire  cpt_pred(v) = cpt_pred(v) – 1; /*Décrementer le compteur des prédecesseurs de v*/
        Si cpt_pred(v)=0
        Alors enfiler(v) ;
        Fsi ;
    Fait ;
Fait ;
Fin.
```

7- **Rappel de la définition d'un problème d'optimisation :**

D'une façon générale, un problème d'optimisation est défini comme suit [5, 12] :

Il s'agit de trouver un élément x appartenant à R^n pour :

minimiser f(x) (appelée fonction objectif)
avec les contraintes $g_i(x) >= 0$ **i=1,n**
$h_j(x) = 0$ **j = 1,p**
où f, g_i , h_j **sont des fonctions réelles de x appartenant à** R^n

On distingue deux catégories de problèmes d'optimisation en fonction de la nature de x qui peut être continue ou discrète. On parle d'optimisation combinatoire lorsque la variable x est discrète.

Si on se réfère à cette formulation, dans le cas des circuits combinatoires auxquels nous nous sommes intéressés particulièrement, la fonction objectif f(x) représente la surface totale du circuit. Tandis que les fonctions gi(x), hj(x) représentent des contraintes sur des paramètres tels que le délai de propagation, et la consommation d'énergie fixés par l'utilisateur. Ces contraintes peuvent être

aussi d'ordre technologique, telles que celles des dimensions minimales des transistors autorisées pour une technologie de fabrication donnée.

8- **Rappels sur la complexité d'algorithmes**

La complexité d'algorithmes est une mesure de la performance ou du temps d'exécution . La performance d'algorithmes est mesurée en fonction de la taille des données par une fonction notée T(n) . Etant donné que des algorithmes complexes peuvent traiter différents cas de figure, la performance est généralement estimée dans le cas le plus défavorable. On s'intéresse plus particulièrement au comportement asymptotique de la fonction T(n) pour avoir une idée des performances de l'algorithme quand la taille des données évolue. Pour cela, la notation O est utilisée.

On dit que T(n) est en O(f(n)), s'il existe deux constantes c et no telles que :

$$\mathbf{T(n) \leq c\, f(n) \quad \forall\, n \geq no}$$

Cette mesure permet aussi la comparaison d'algorithmes et la détermination approximative des tailles maximales des problèmes qui peuvent être traités par les algorithmes.

On considère généralement qu'un "bon" algorithme est un algorithme polynômial $0(n^k)$ parce que le temps de réponse de l'algorithme évoluera de façon polynômiale avec la taille des données. Par contre, on considère un algorithme exponentiel comme étant un "mauvais" algorithme sachant que le temps de calcul va vite devenir prohibitif, dès que la taille du problème augmente au delà d'un seuil.

Dans le cas des problèmes NP-Difficiles, on a généralement recours à des algorithmes approchés appelés heuristiques. Ces heuristiques vont essentiellement déterminer une solution qui approche la solution optimale en un temps raisonnable (généralement avec une complexité polynômiale).

Pour cela, nous allons étudier, dans le chapitre suivant, l’algorithme génétique qui est une méta-heuristique connue pour sa robustesse et sa simplicité dans la résolution d’un problème NP-Hard.

Chapitre II

Principe général des algorithmes génétiques

1.Introduction :

Les algorithmes génétiques [13] sont des algorithmes d'exploration fondés sur les mécanismes de la sélection naturelle et de la génétique. Ils utilisent à la fois les principes de la survie des structures les mieux adaptées et les échanges d'informations pseudo-aléatoires pour former un algorithme d'exploration qui possède les caractéristiques de l'exploration humaine.

Bien qu'utilisant le hasard, ils ne sont pas purement aléatoires. Ils explorent efficacement l'information obtenue précédemment pour spéculer sur la position de nouveaux points à explorer avec espoir d'amélioration des performances.

L'objectif des algorithmes génétiques (AG) est de trouver une solution satisfaisante à un problème donné. Cependant, comme les AG sont des heuristiques, rien ne garantit l'optimalité de la solution [9] mais l'expérience a montré qu'ils sont capables de trouver de très bonnes solutions pour un grand nombre de problèmes différents. Ils travaillent en faisant évoluer une population d'individus à travers plusieurs générations. Une valeur dite *"fitness"* est assignée à chaque individu où le calcul de cette valeur dépend du problème traité.

La sélection des individus élimine l'une des contraintes majeures à la conception des programmes : la spécification préalable de toutes les caractéristiques d'un problème et des tâches précises qu'un programme doit effectuer pour résoudre ce problème.

Les algorithmes génétiques ont été développés par J. Holland [14] en 1975, ses collègues et ses étudiants à l'université du Michigan. Leurs recherches avaient deux objectifs principaux :

- mettre en évidence et expliquer rigoureusement les processus d'adaptation des systèmes naturels,
- concevoir des systèmes artificiels (des logiciels) qui possèdent les propriétés des systèmes naturels.

Cette approche a débouché sur des découvertes importantes à la fois dans les sciences des systèmes naturels et dans celle des systèmes artificiels.

2 Les méthodes d'optimisation classiques :

Parmi les méthodes d'optimisation classiques, nous citons : les méthodes fondées sur le calcul, les méthodes énumératives et les méthodes aléatoires.

- Les méthodes numériques différentielles: cherchent à atteindre des extremums locaux en résolvant des systèmes d'équations souvent non linéaires, obtenus en fixant au vecteur nul le gradient de la fonction à étudier. Cela suppose donc que la fonction est continue et dérivable au préalable.
  D'autre part, ces méthodes s'appliquent localement, les extremums qu'elles atteignent sont optimaux dans un voisinage du point de départ.
- Les méthodes énumératives : L'idée générale est simple. Dans un espace de recherche fini ou infini mais discrétisé, l'algorithme teste les valeurs de la fonction à optimiser en chaque point de cet espace, point par point. Cette

méthode est certes précise mais a pour inconvénient majeure l'explosion exponentielle du temps de recherche dès que la taille de l'espace de recherche est grande. Parmi ces méthodes, la méthode Branch and Bound.

- ♦ Les méthodes aléatoires : ont connu un grand succès depuis que les chercheurs ont observé les limitations des méthodes basées sur le calcul et sur l'énumération. Il s'agit d'effectuer des tirages au sort de façon tout à fait aléatoire, de tester la valeur de la fonction à étudier en ces points et de conserver la meilleure . Ces méthodes restent très inefficaces et sont à abandonner.

3- Les meta-heuristiques : ont connu un grand succès et ont apporté une bonne contribution dans la résolution de problèmes NP-difficiles tels que les problèmes d'optimisation combinatoire. Citons la méthode de recherche Tabou ou *"Tabu Search"* et la méthode du recuit simulé ou *"Simulated Anealing"*.

- La méthode ou *"Tabu Search"* **[51]** génère à chaque itération un ensemble de solutions V^* dans le voisinage de la solution courante . La meilleure solution est acceptée même si son coût est supérieure au coût de la solution courante. En effet, la technique Tabou consiste à déterminer à chaque itération une nouvelle solution parmi l'ensemble des solutions voisines de la solution courante et telle que son coût soit le plus faible, ceci permet de choisir le moins mauvais des voisins. L'inconvénient est que si un minimum local est atteint, le passage d'une solution s vers une solution s' tel que coût(s')>coût(s) peut provoquer à l'tération suivante le retour à la solution s puisque s est voisine de s' : le risque est de cycler autour de ce minimum local. Pour cela, une liste T appelée Liste Tabou est introduite dans laquelle les solutions s déjà explorées sont sauvegardées. La meilleure solution s' de V^* est gardée dans T si $s' \notin T$. Cette condition est appelée "la condition tabou". En générant les nouvelles solutions dans ($V^* \backslash T$), l'utilisation de la

"Tabu List" permet de ne pas retomber sur une solution déjà visitée. Parfois, il est nécessaire de reprendre une solution déjà visitée et de continuer la recherche dans une autre direction. Pour cela, lorsque la taille de la liste Tabou est atteinte, une solution nouvelle S_{new} remplacera la plus ancienne solution S_{old} de T. Ce processus prend fin lorsqu'il n'y a plus d'améliorations possibles. La solution obtenue est proche de l'optimale mais rien ne garantit l'optimum global.

- la méthode du recuit simulé **[52]** s'inspire de la technique du recuit qui consiste à chauffer un métal jusqu'à l'état liquide puis le refroidir progressivement en marquant des paliers de température de durée suffisante. L'algorithme du recuit simulé génère aléatoirement une solution s' à partir d'une solution initiale. La nouvelle solution est acceptée ou non selon un choix effectué par la génération d'un nombre aléatoire contrôlé par un paramètre T analoque à la température dans le processus physique du recuit. La température est diminuée en fonction d'un paramètre $\lambda \in [0.85 \;\; 0.95]$.

A chaque température T fixée, un nombre N de transformations est généré. Ce nombre est un paramètre, contrôlé par l'utilisateur en fonction de la qualité de la solution finale souhaitée. L'expérience a montré que l'algorithme du recuit simulé approche asymptotiquement l'optimum global du problème posé.

L'algorithme génétique est aussi une meta-heuristique, fondamentalement différente des algorithmes classiques d'optimisation. Ces différences s'articulent autour de quatre axes principaux [14]:

- ils utilisent un codage des paramètres et non les paramètres eux mêmes.
- ils travaillent sur une population de points au lieu d'un point unique.
- ils n'utilisent que la valeur de la fonction étudiée, jamais sa dérivée qui nécessite un calcul analytique ou numérique, ni aucune autre connaissance auxiliaire. On dit, pour cela, que les AG (algorithmes génétiques) sont aveugles.

- ils utilisent des règles de transition probabilistes et non déterministes.

Toutes ces caractéristiques prises ensemble, contribuent à la robustesse de ces algorithmes . On définit la robustesse d'une méthode [15] par son aptitude à fournir des résultats quelque soit le problème traité par cette méthode et quelque soit les ″benchmarks ″ (jeux d'éssais) utilisés. Or, il a été prouvé théoriquement [13] et expérimentalement que les algorithmes génétiques sont des procédures robustes d'exploration d'espaces complexes.

Comme il a été spécifié au premier point, les algorithmes génétiques utilisent un codage des paramètres. Pour cela, ils manipulent des chromosomes ou chaînes. Ces derniers représentent en réalité chacun une solution possible du problème.

Un chromosome est constitué de gènes, chacun représentant une variable qui caractérise un des paramètres du problème. Il est à noter que dans le codage binaire, un gène est représenté par un bit (0 ou 1).

4- Les principes de base d'un algorithme génétique

Les mécanismes de base d'un algorithme génétique mettent en jeu des copies de chaînes et des échanges de parties de chaînes. Au départ une génération de chaînes est créée en effectuant un tirage aléatoire. A partir de cette génération, les opérateurs suivants :

- reproduction (sélection)
- ″crossover″ ou croisement
- mutation
- évaluation

sont appliqués pour générer une nouvelle population.

5- Les opérateurs génétiques :

5-1 La reproduction (ou sélection)

C'est un procédé selon lequel chaque chaîne est copiée en fonction des valeurs de la fonction f à optimiser, appelée fonction d'adaptation.

Cela revient à donner aux chaînes dont la valeur de f est plus optimale, une probabilité plus élevée de contribuer à la génération suivante en créant des descendants. Cet opérateur est une version artificielle de la sélection naturelle où l'adaptation est déterminée par la capacité d'une créature à survivre à tous les obstacles jusqu'à l'âge adulte. Dans notre milieu artificiel, la fonction d'adaptation est l'unique décideur de la vie ou de la mort de chaque chaine-créature [13].

Il y a différentes manières d'effectuer la sélection [9], la plus importante étant la roue de sélection biaisée. Dans ce cas, il s'agit d'affecter à chaque individu ou chromosome une probabilité d'être sélectionné proportionnellement à la valeur de la fonction d'évaluation en un point.

Donc, chaque chaîne de la population présente, occupe une section de la roue proportionnelle à son adaptation., comme le montre la figure 2.1

Pour qu'un individu soit sélectionné, nous calculons d'abord sa probabilité cumulée :

$$Q_i = \sum_{j=1}^{i} p_j \ .$$

La probabilité p_i de sélection d'un individu i est donnée par :

$$p_i = f_i \Big/ \sum_{i=1}^{n} f_i$$

n : étant le nombre d'individus, f_i est l'adaptation de l'individu i.

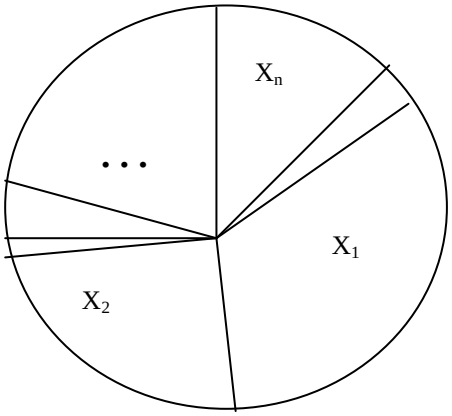


Fig. 2.1 : Simulation d'une roue biaisée

L'individu i est sélectionné si $Q_{i-1} < r <= Q_i$, r étant un réel tiré aléatoirement dans l'interval [0,1].

5-2 Le croisement ou *"crossover "*

Durant cette phase, les éléments nouvellement produits par la reproduction sont d'abord appariés aléatoirement, ensuite un croisement a lieu entre chaque paire de chaînes de la manière suivante : soit L la longueur d'une chaine et soit k un entier représentant une position sur la chaîne. Cet entier est choisi aléatoirement entre 1 et (L-1). Deux nouvelles chaînes sont crées en échangeant tous les caractères compris entre les positions (k+1) et L inclues.

Exemple : considérons les chaînes A1 et A2 de la population initiale :

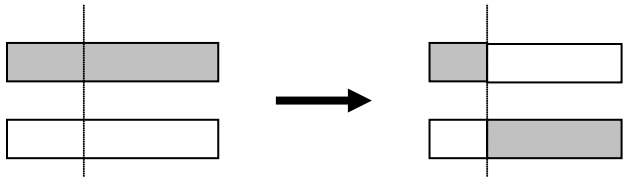
A1 = 011\0101 où \ représente la position du croisement en l'occurrence k.

A2 = 110\1110

Pour k=3, les chaînes A'1 et A'2 obtenues après permutation entre A1 et A2 sont :

A'1 = 011\1110

A'2 = 110\0101

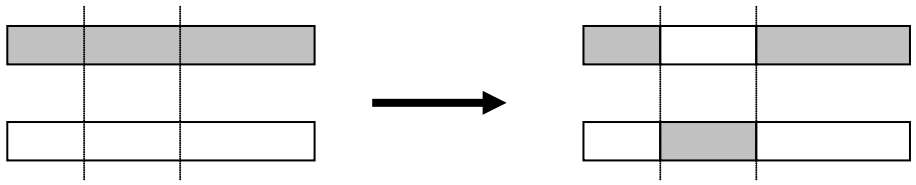


Ce type de croisement est appelé croisement à un point. Il existe des croisements à plusieurs points, notamment à deux points. Les points de croisement k_1 et k_2 sont tirés aléatoirement entre 1 et (L-1), L étant la longueur d'une chaîne.

Le croisement à 2 points consiste à faire échanger aux chaînes des parents les sous chaînes comprises entre les points de croisement.

Exemple : A_1 = 01\1010\100 | A'_1 = 010111100

A_2 = 11\0111\001 | A'_2 = 111010001



Le croisement à deux points permet parfois d'obtenir de nouveaux types d'individus et évite parfois de détruire des types d'individus intéressants.

Le taux de croisement dans une population est contrôlé par la probabilité de croisement p_{cross} . Plus cette probabilité est élevée plus elle permet d'explorer de solutions dans l'espace de recherche et réduit les chances de tomber sur un optimum local. Une probabilité de croisement assez faible permet d'explorer et de combiner des parents ayant des adaptations relativement intéressantes.

Les mécanismes de reproduction et de croisement sont donc simples mais ce sont leurs actions combinées qui donnent aux algorithmes génétiques leur puissance, le croisement n'étant rien d'autre que la juxtaposition de choses qui ont bien fonctionné dans le passé.

5-3 La mutation

Le rôle de la mutation reste très flou dans la génétique aussi bien naturelle qu'artificielle. Cependant, elle est nécessaire parce que, bien que la reproduction et le croisement explorent et recombinent efficacement les notions existantes, ils peuvent parfois perdre de la matière génétique potentiellement utile (des 1 ou des 0 à des endroits précis). Bien que la mutation ne crée généralement pas de

meilleurs solutions au problème traité, elle évite l'établissement de populations uniformes incapables d'évoluer.

Dans un algorithme génétique simple, la mutation est la modification aléatoire occasionnelle (de faible probabilité) de la valeur d'un caractère de la chaîne.
Dans le cas d'un codage binaire, cela revient simplement à inverser un bit. Prise isolément, la mutation constitue une exploration aléatoire de l'espace de recherche mais utilisée correctement avec la reproduction et le crossover, la mutation protège de la perte prématurée de notions importantes [13].

5-5 L'évaluation

Par l'application des trois opérateurs précédents, nous obtenons une nouvelle génération qui doit être évaluée. Pour cela, la fonction d'adaptation intervient en dernier lieu pour déterminer les meilleurs chromosomes à utiliser dans la prochaine génération : ce sont les valeurs de la fonction à optimiser en chaque chromosome qui diront quels sont les meilleurs individus à garder, selon un ou plusieurs critères prédéfinis dans la fonction d'adaptation.
Il est à signaler que la fonction d'adaptation (ou d'évaluation qui est en fait la fonction à optimiser) doit être positive sur tout le domaine de recherche : c'est là la seule contrainte imposée sur la fonction.

6- Structure d'un Algorithme Génétique :

Un AG est communément décrit comme ayant cinq principales composantes :

1- Une représentation chromosomale du problème : c'est un codage des paramètres du problème adéquatement choisi.
2- Une manière (aléatoire ou autre) de générer une population initiale.
3- Une fonction d'évaluation ou d'adaptation, qui joue le rôle de juge (elle juge les membres de la population selon leurs qualités).
4- Les opérateurs génétiques dans le but de produire une nouvelle génération.

5- Des valeurs que l'algorithme génétique utilise pendant son déroulement, telles que la taille de la population, le nombre de générations, les probabilités d'application des opérateurs génétiques,...

Bien que possédant plusieurs variantes, l'algorithme génétique simple se présente sous la forme décrite dans la figure suivante.

BEGIN

- **Etape 0 : définir un codage approprié des paramètres du problème,**

- **Etape 1 : à t=1, créer une population initiale de N individus, aléatoire ou autre , $P(0)=\{X_1,X_2, \ldots, X_N\}$**

Répéter {

- **Etape 2 : Evaluation : calculer la valeur de la fonction objectif pour chaque individu X_i, i=1,…,N.**

- **Etape 3 : Sélection : sélectionner N individus de P(t) et les ranger dans un ensemble S(t). Un même individu de P(t) peut apparaître plusieurs fois dans S(t).**

- **Etape 4 : Crossover : Grouper les individus de S(t) par paires puis, pour chaque paire d'individus :**

- avec la probabilité de croisement p_{cross}, appliquer le croisement à la paire et recopier la nouvelle paire obtenue dans S(t+1) (la paire d'individus est éliminée, elle est remplacée par sa progéniture).

- avec la probabilité $(1 – p_{cross})$, recopier la paire d'individus dans S(t+1)

- **Etape 5 : Pour chaque individu de S(t+1)**

- avec la probabilité de mutation p_{mut}, appliquer la mutation à l'individu, le recopier dans S(t+1)

- avec la probabilité $(1 - p_{mut})$, recopier l'individu dans S(t+1) tel qu'il est.

- **Etape 6 : incrémenter t**

} tant que nb-itérations≤ MAXGEN /*MAXGEN étant le nombre de générations maximum ou bien un autre critère d'arrêt défini d'avance.*/

- **Afficher le meilleur BEST ;**
- **END.**

Un critère d'arrêt peut être choisi de sorte que lorsqu'il n'y a plus d'amélioration possible par rapport à la génération précédente, les itérations sont arrêtées.

7- <u>Conclusion</u> : Les AG sont des algorithmes d'exploration robustes. Lorsque leurs paramètres (taille de la population, nombre d'itérations, les probabilités de tirage,...) sont choisis d'une manière adéquate, ils convergent rapidement vers une solution satisfaisante. De plus, ils sont simples à mettre en œuvre. Leur caractéristique principale est qu'ils soient parallélisables [16]. Cela est dû au fait qu'ils opèrent sur une population d'individus, les opérateurs génétiques pouvant traiter en parallèle chacun d'entre eux. Ce qui a eu pour conséquence le grand intérêt porté à ces méthodes. De plus [9], ils ont fait leurs preuves concernant la résolution de problèmes d'optimisation complexes tels que ceux rencontrés dans la conception et le test automatique des circuits intégrés VLSI. Dans le prochain chapitre sera présenté un de ces problèmes : le *"Gate Sizing"*.

Chapitre III

Position du Problème de dimensionnement des circuits intégrés VLSI

1- Introduction

Dans le cadre de notre travail, nous avons abordé le problème de dimensionnement de circuits intégrés binaire (circuits combinatoires mixtes CMOS-BiCMOS) où le choix se fait entre deux technologies : CMOS et BiCMOS. Le but de notre travail est d'améliorer la performance temporelle d'un circuit en faisant, pour chaque porte, un choix judicieux entre une cellule associée CMOS ou BiCMOS. Le choix devra permettre l'exploitation efficace des qualités respectives de chaque technologie. Ces qualités sont [17]:

- Pour la technologie BiCMOS, la vitesse élevée et la possibilité d'attaquer de grandes charges.
- Pour la technologie CMOS, la densité et la faible consommation d'énergie.

Etant donné un circuit intégré, évaluer ses performances temporelles revient à déterminer son délai de propagation . Ce dernier représente le temps que met le signal à traverser le circuit de ses entrés à ses sorties, estimé sur le chemin le plus long. Afin de minimiser ce temps, il faut choisir des composants élémentaires rapides, telles que les portes logiques de type BiCMOS.

Cependant, ces dernières occupent une surface importante, ce qui n'est pas sans conséquences sur la surface totale du circuit. Il devient alors impératif de choisir judicieusement le type des portes à placer dans le circuit de manière à augmenter sa vitesse sans toutefois pénaliser sa surface totale. Ce problème est appelé couramment dans le monde de la VLSI problème de dimensionnement de portes logiques ou *"Gate Sizing"*.

2- Formulation mathématique du problème de dimensionnement de portes : [5]

Etant donné un circuit intégré représenté par un graphe acyclique orienté G=(V,E), l'ensemble des sommets V est égal à $\{v_1, v_2, ..., v_n\}$, tandis que l'ensemble des arcs E est inclus dans V*V. Chaque porte logique du circuit ou *"gate"* g_i est représentée par un noeud v_i ($vertice_i$) appartenant à G. Chaque connexion du circuit reliant une sortie de la porte g_i à une entrée de la porte g_j est représentée par un arc du graphe (v_i, v_j). Les sommets ayant un degré d'entrée nul sont les entrées primaires du circuit, alors que les sommets ayant un degré de sortie nul sont les sorties primaires. On appelle chemin toute séquence d'arcs adjacents partant des entrées jusqu'aux sorties. On définit une fonction f qui associe chaque sommet v_i au type de la porte g_i associée. Le type d'une porte est représenté par un indice entier, référant au rang de la cellule dans la bibliothèque des cellules.

En résumant, un circuit combinatoire est représenté par un graphe G orienté acyclique et une fonction f. Une bibliothèque de cellules L peut être représentée par une famille d'ensembles finis $L_1, L_2, ...L_k$, où L_i $(1 \leq i \leq k)$ contient une ou plusieurs cellules logiquement équivalentes réalisant la porte de type i, k étant le nombre total de portes disponibles dans la bibliothèque, iluustrée par la figure 3.1.

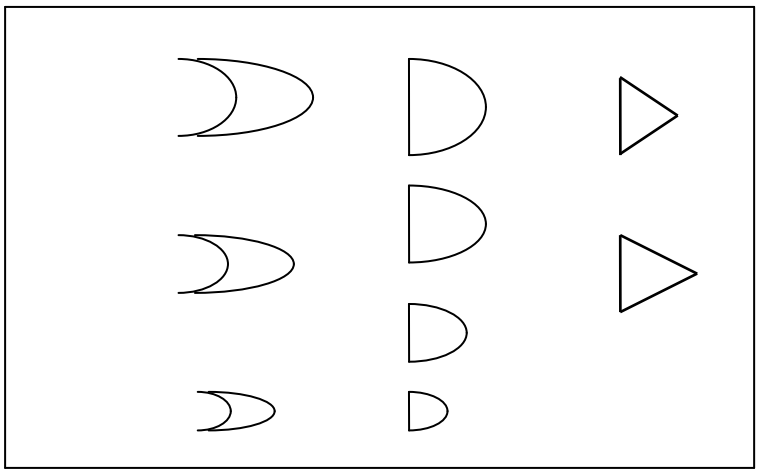


Fig. 3.1 : <u>Illustration d'une bibliothèque de cellules</u>

Chaque cellule [17] sera caractérisée par sa surface a (*area*) et son délai de propagation d (*delay*). On associe L_i, à chaque cellule, pour chaque porte i où :

$$L_i = [\ (a_{i1}\ ,\ d_{i1})\ ,\ (a_{i2}\ ,\ d_{i2})\ ,.... (a_{ip}\ ,\ d_{ip(i)})\ ]$$

qui est un ensemble fini de couples où $p_{(i)}$ représente le nombre de cellules réalisant la porte g_i. En d'autres termes, $p_{(i)}$ est le nombre de types différents de chaque porte. Le symbole a_{ij} représente la surface de la jème cellule réalisant la ième porte. Le symbole d_{ij} représente le délai de la jème cellule réalisant la ième porte.

Etant donnés un circuit décrit par un graphe G et une fonction f, ainsi qu'une bibliothèque de cellules $L=\{L_1,L_2,\ldots\ L_k\}$, une réalisation du circuit à l'aide de la bibliothèque est l'association de chaque sommet v appartenant à G à une cellule de $L_{f(v).}$

Soit $S = (\ S_1,\ S_2,....S_n\ )$ une réalisation où $1 \leq S_i \leq p_{f(v).}$ La surface du circuit à l'aide de la réalisation S est égale à :

$$\mathbf{A(S,G) = \sum_{i=1}^{n} a_{[f(vi),\ si]}}$$

Soit un chemin P constitué de v_{i1}, v_{i2},....v_{il} dans G. Le délai de ce chemin est égal à :

$$d(S,P) = \sum_{j=1}^{l} d_{[f(vij)\,,\,sij]}$$

Le délai du circuit avec la réalisation S est noté:

D(S,G) = max { d(S,P) pour tous les chemins de G }

Le problème de dimensionnement peut être formulé comme suit:
Etant donné un graphe G, une fonction f, une bibliothèque L, et Tmax un délai maximum, il s'agit de :

P

Minimiser A(S,G)

Tel que D(S,G) ≤ T max

Dans ce qui suit, nous allons présenter quelques travaux importants effectués durant cette dernière décennie sur ce problème. Par conséquent, la section suivante présente un état de l'art.

3- Etat de l'art :

Les premiers travaux qui ont été entrepris dans le domaine de dimensionnement au niveau portes[18] [19] ont considéré des hypothèses simplificatrices sur les modèles de délai et sur la structure ou topologie des circuits. Généralement, les délais de propagation des portes étaient considérés comme étant constants et indépendants des charges.
Chan [18] considéra d'abord des cas particuliers de circuits ayant des topologies d'arbres et prouva un résultat théorique important, à savoir que, même dans ce cas de figure, le problème de dimensionnement reste NP-difficile.

Il proposa ensuite un algorithme pseudo-polynômial [voir chap. I] pour les circuits ayant une topologie d'arbre. Puis, il développa un algorithme qui transformait n'importe quel circuit en un circuit équivalent ayant une topologie d'arbre. Cette transformation s'opère par le biais de la duplication des portes ou *"cloning"*. Seulement, cette méthode traite uniquement des circuits de petite taille, car le nombre de portes ajoutées augmente exponentiellement.

Li [20] prouva un autre résultat théorique important qui consiste en l'affirmation suivante : Même lorsque les circuits sont restreints à des chaînes de portes, le problème de dimensionnement est un problème NP-difficile. Li s'intéressa, lui aussi, à une classe particulière de circuits qui sont les circuits ayant une structure de graphe série/parallèle. Il proposa, à cet effet, un algorithme pseudo-polynômial. Par la suite, il prouva que le problème de dimensionnement de portes logiques, quelle que soit la topologie du circuit, est NP-difficile au sens fort [21] (*NP-hard in the strong sense*). Ce qui signifie qu'il n'existe pas d'algorithme pseudo-polynômial pour ce problème dans le cas général. A partir de ce stade, les travaux se sont tournés vers des méthodes approchées du problème et commencèrent, de plus en plus, à considérer le problème de dimensionnement dans sa généralité. Parmi les travaux les plus importants, on peut citer :

- ceux de Moon [19] utilisant une heuristique qui transforme, à l'aide d'une méthode de *"cloning"* un circuit général en circuit ayant une topologie série/parallèle. Ceci permet d'utiliser une méthode proche de celle de Li[21]. La différence qui existe est due au fait que puisque les modèles de délais utilisés considèrent les capacités de charge, une approche partant des sorties et allant vers les entrées est utilisée, contrairement à l'algorithme de Li qui part des entrées vers les sorties. Ensuite la solution obtenue est affinée à l'aide d'une méthode de clonage (duplication des nœuds) plus efficace en termes de temps et d'espace. L'optimisation proprement dite est effectuée

itérativement sur les chemins critiques. Ces travaux se caractérisent par l'utilisation d'un modèle de délai réaliste qui considère les charges.

- ceux de Fang [22] qui sont parmi les premiers travaux qui considèrent la notion de vrais chemins critiques et qui n'optimisent pas uniquement localement sur chaque chemin critique. Bien au contraire, ils essaient de prendre en compte, en même temps, tous les chemins critiques. Ceci est fait grâce à la notion de "*delay contribution*" qui est une mesure qui permet de choisir la porte dont l'optimisation aura le plus d'effet sur la performance du circuit. La "*delay contribution*" des portes est calculée uniquement sur les chemins critiques topologiques. Ensuite, quand une porte est sélectionnée, un test est fait pour savoir si elle appartient à un vrai chemin critique grâce à un algorithme appelé τ-Podem. Les chemins critiques sont calculés en utilisant l'approche "*block oriented*" [23] basée sur l'algorithme de PERT [24] qui permet de déterminer les chemins ayant une flexibilité ou *"slack-time"* nulle.

Toutefois, les travaux de Moon souffrent de l'imprécision des résultats, tandis que ceux de Fang se caractérisent par des temps de calcul élevés. Il devient alors, évident qu'un compromis devra être trouvé de façon à concilier la qualité des circuits optimisés et les temps nécessaires pour les calculs. Il est fort probable qu'à ce niveau se situe l'enjeu des futurs travaux de recherche en ce domaine.

- Nous citons également les travaux de A.R. BABA-ALI [5], en 1998, qui a proposé deux heuristiques polynômiales pour résoudre le problème de dimensionnement, l'une locale, l'autre globale. Dans la première approche[25], la décision de choisir le type de cellules est prise uniquement en fonction de la charge de la porte qui est une information locale à la porte considérée.

Pour cela, un algorithme basé sur l'exploration en profondeur BFS étendu [voir Chap. I], est utilisé des sorties du circuit vers ses entrées.

Cet algorithme permet de traiter les portes du circuit successivement dans leur ordre topologique décroissant, niveau logique après niveau, chaque porte n'étant traitée qu'une seule fois, étant donnée la complexité linéaire de l'algorithme BFS. L'algorithme DFS [voir Chap.I] a également été utilisé pour éliminer les boucles du circuit, permettant ainsi de traiter des circuits séquentiels. Le processus prend fin lorsque les entrées du circuit sont atteintes. Cette approche a donné de bons résultats tout en se caractérisant par une complexité linéaire.

Dans la seconde approche [5], dite globale, les cellules BiCMOS ne seront sélectionnées que si la porte est située sur un chemin critique. Par conséquent, la sélection est effectuée en fonction des caractéristiques de tout le circuit. La méthode qu'il a proposée est itérative. Le chemin critique est déterminé en premier lieu, ensuite ce chemin est parcouru de sa sortie à son entrée. Pour chaque nœud rencontré, la comparaison est effectuée avec la valeur Cx qui est un seuil de charge attaquée, et la sélection de la cellule est faite. Une fois le nœud d'entrée rencontré, on passe au chemin critique suivant. Le processus prend fin lorsqu'il n'y aura plus d'amélioration possible ou bien lorsque le délai de propagation du circuit imposé par le concepteur est atteint.

Ces travaux sont importants vu qu'il arrive à déterminer des solutions satisfaisantes en un temps polynômial.

Actuellement, les appoches d'optimisation de circuit s'orientent vers des heuristiques itératives de traitement des chemins [49, 50].

De plus, de nouvelles méthodes basées sur des modèles de délais statistiques sont apparues récemment [27]. Elles estiment le délai de chaque porte logique par des mesures statistiques plus précises.

Toutefois, les solutions obtenues sont sub-optimales étant donné que le problème du ″Gate sizing″ appartient à la classe des problèmes NP-difficiles. Notre contribution dans ce domaine, consiste à résoudre ce problème par une approche génétique, qui est une méta-heuristique connue pour son efficacité à résoudre des problèmes NP-difficiles. Nous avons choisi cette approche car les algorithmes génétiques sont simples à manœuvrer et à implémenter et surtout pour leur nature parallélisable vu qu'ils sont basés population. De plus, ils ont déjà prouvé leur grande puissance et leur efficacité dans la résolution de problèmes complexes de la VLSI [9].

En conséquence, les détails de cette approche seront proposés dans le prochain chapitre.

Chapitre IV

Résolution du problème de dimensionnement des circuits intégrés VLSI par une approche génétique.

1- Introduction

Lors de l'optimisation de circuits intégrés, deux caractéristiques fondamentales des algorithmes choisis à cet effet, devront être prises en considération : [26]

- la performance des algorithmes
- la précision des résultats

Ces deux caractéristiques sont antagonistes. En effet, plus la qualité des circuits optimisés sera élevée plus les temps de calcul seront prohibitifs. Par contre, plus un algorithme sera rapide, plus la qualité des circuits optimisés sera faible. Par conséquent, le compromis vitesse/précision est un facteur déterminant dans le choix des algorithmes.

Nous avons choisi d'apporter une solution au problème de dimensionnement des circuits intégrés VLSI par un algorithme génétique qui est une heuristique connue pour sa robustesse et pour les solutions satisfaisantes qu'elle fournit en un temps polynômial.

Notre contribution à l'optimisation de circuits intégrés s'effectue selon le diagramme représenté par la figure 4.1

Au départ, la topologie du circuit est fournie en entrée sous forme d'un fichier décrivant ses entrées, ses sorties, ses composants, notamment des portes, ainsi que les interconnexions entre les différentes cellules du circuit. Les spécifications du concepteur et les contraintes du cahier des charges y figurent également. Le logiciel d'optimisation du circuit doit pouvoir fournir toutes les structures de données dynamiques construites au fur et à mesure de la lecture du fichier d'entrée. Il doit renfermer également toutes les spécificités de l'algorithme génétique à savoir le codage de tous les paramètres du problème, les différents opérateurs génétiques ainsi que leur application, l'évaluation des individus,...

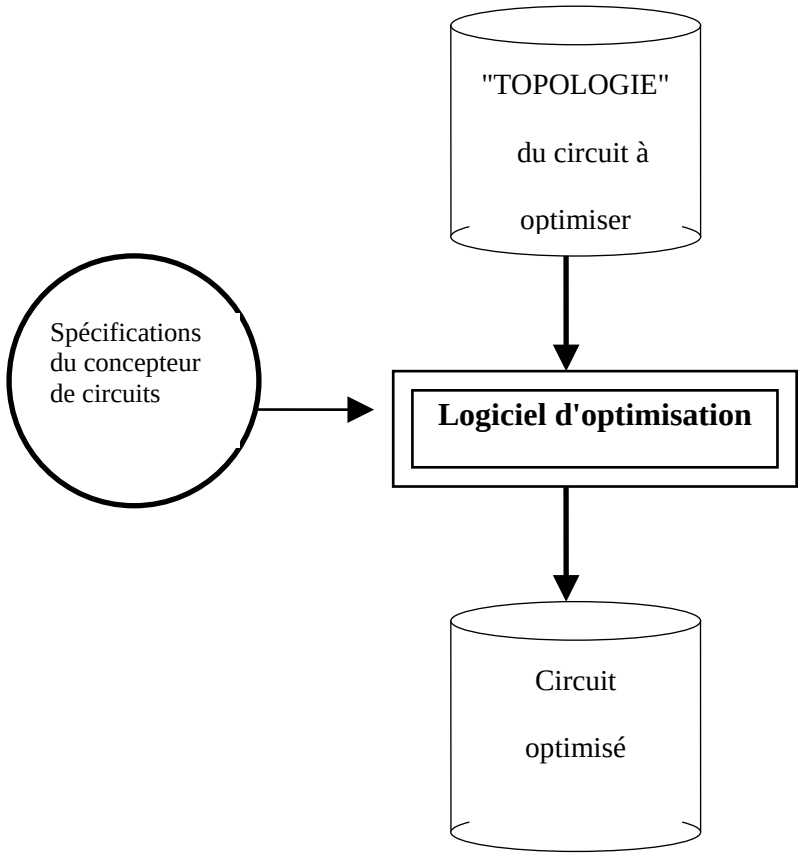


Fig. 4.1 : Diagramme d'optimisation d'un circuit intégré VLSI.

En sortie, il fournit une configuration optimale du circuit concrétisée par un choix judicieux des paramètres de chaque porte logique réalisant le meilleur compromis : surface minimale occupée par le circuit et délai de propagation vérifiant la contrainte temporelle imposée par le cahier des charges.

Dans ce qui suit, nous allons décrire sommairement les étapes d'optimisation prises en charge par ce logiciel, à savoir :

- la construction dynamique du graphe qui représente en mémoire le circuit à optimiser.
- Le codage des paramètres du circuit et les différentes structures de données de l'algorithme génétique.
- Les différents traitements effectuées sur ces structures de données.

2- Structures de données et représentation en mémoire de circuits VLSI :

Un circuit est représenté en mémoire sous forme d'un graphe dont les sommets représentent les portes du circuit et les arcs représentent les nœuds du circuit [voir chapI] .

Il est évident que deux sommets du graphes ne sont reliés par un arc que si les portes correspondantes sont interconnectées dans le circuit. Les arcs du graphe représentent les interconnexions du circuit.

Nous avons disposé d'une bibliothèque de cellules que nous décrirons dans la prochaine section.

Par ailleurs, nous avons testé notre approche sur des circuits *"benchmark"* [28] de format Bdnet. La structure de données proposée pour représenter ces circuits en mémoire est une structure dynamique : ce sont des listes de cellules, de portes, de ports et de nœuds qui composent le circuit.

2.1 Description de la bibliothèque des cellules:

Nous disposons d'une bibliothèque de cellules qui comporte un certain nombre de cellules pouvant intervenir dans la conception d'un circuit intégré, à savoir des portes *AND, NAND, INV, NOR,ORNAND, ANDNOR,...*

La structure de données décrivant chaque cellule de la bibliothèque comporte :

- le nom de la cellule,
- ses paramètres temporels,
- ses paramètres électriques,
- ses paramètres de puissance,
- ses dimensions,
- le nombre de ses ports d'entrées ainsi qu'un pointeur sur leur liste,
- le nombre de ses ports de sortie ainsi qu'un pointeur sur leur liste.

Tous ces paramètres sont fournis pour chaque porte et pour chacun de ses types considérés dans notre cas. Les paramètres temporels d'une porte, qui ont été déterminés expérimentalement, représentent des résistances internes : ils servent au calcul du délai de cette porte.

La figure 4.2 représente un bloc qui décrit une cellule de la bibliothèque.

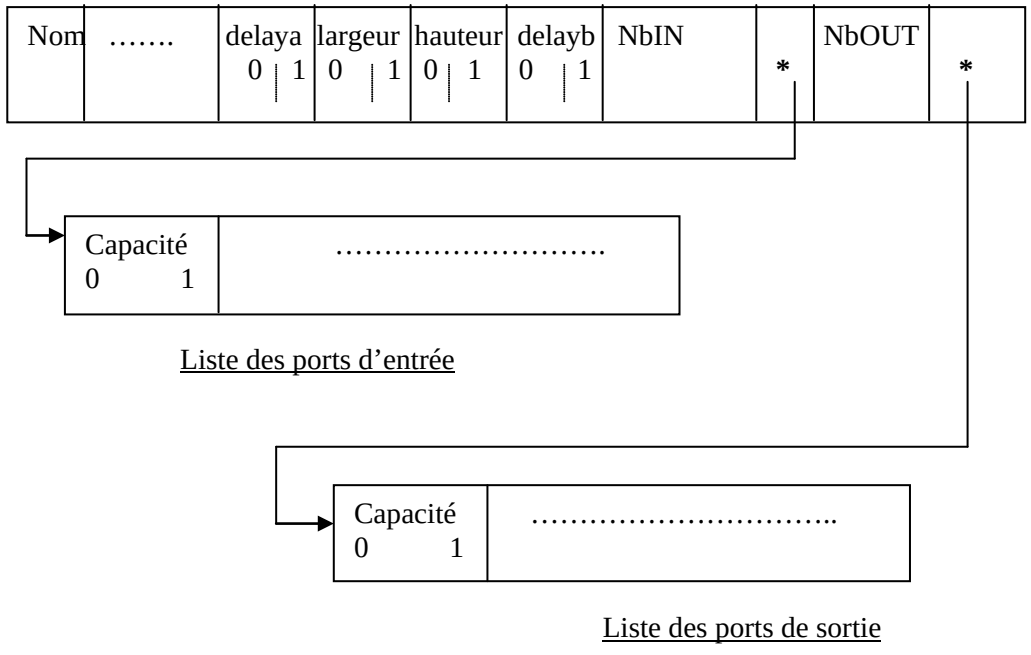


Fig 4.2 : Description d'une cellule de la bibliothèque de cellules

2.2 Représentation en mémoire des composants du circuit :

La structure de données qui décrit la liste de toutes les portes qui composent le circuit comporte :

- le nom de la porte,
- son type (CMOS, BiCMOS,...),
- son adresse dans la bibliothèque des cellules permettant un accès direct à toutes ses coordonnées,
- le délai de la porte car ce dernier dépend de sa position dans le circuit,
- l'adresse de la prochaine porte qui lui est reliée,
- l'adresse des listes de ses ports d'entrée et ses ports de sortie.

La liste des ports d'une porte, quant à elle, comporte la capacité de chacun de ses ports ainsi que ses coordonnées en x et en y. La figure 4.3 illustre les blocs diagramme qui représentent cette structure de données.

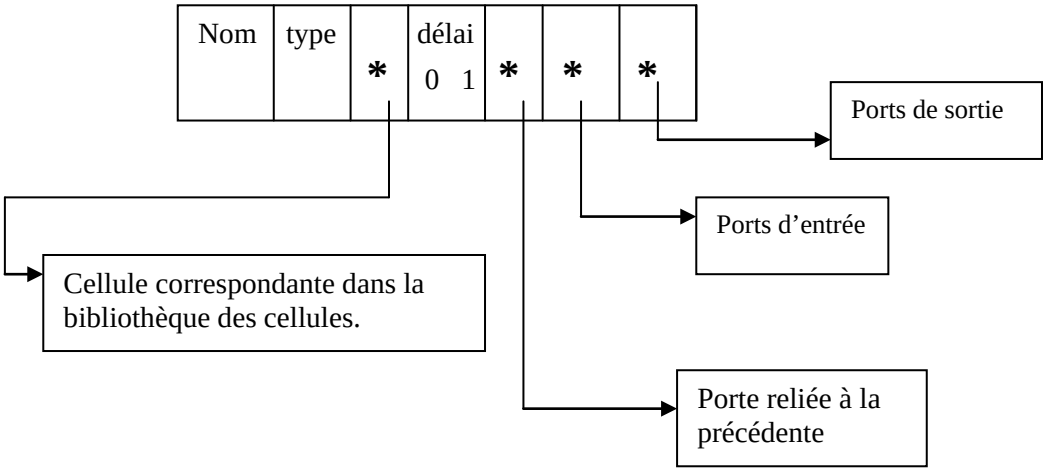


Fig. 4.3 : **Diagramme représentant les structures de données "Porte" et "Ports "**

2.3 Représentation des fils (ou interconnexions) du circuit en mémoire :

La liste des nœuds d'un circuit comporte :

- un identificateur du nœud courant,

- son type : pouvant être de sortie ou d'entrée ou bien quelconque,
- sa capacité qui est fonction des capacités de tous les ports d'entrée des portes qui lui sont connectées.
- Il est spécifié également des adresses vers le prochain nœud, vers le précédent, vers la liste des portes qui le suivent, vers la liste des portes qui le précèdent, des *"flags"* qui précisent si le nœud considéré a déjà été exploré ou non... Toutes ces adresses permettent d'effectuer des traitements sur le circuit dans n'importe quel sens désiré : des entrées vers les sorties ou en sens inverse, en profondeur ou en horizontal.

Toutes ces listes chaînées sont construites dynamiquement au fur et à mesure de l'analyse ligne par ligne du fichier d'entrée contenant la topologie du circuit.
A chaque cellule du fichier d'entrée, il est créé une structure de données de type PORTE. Il est mis à jour, dans celles qui la précèdent, les adresses de liens entre ces cellules et celle nouvellement créée si elles sont connectées dans le graphe du circuit. Les structures de données de type NŒUD auxquels cette porte est connectée sont crées également si elles n'existent pas encore. Les mises à jour des adresses de liens sont effectuées au fur et à mesure que l'analyse en profondeur du fichier d'entrée avance, jusqu'à épuisement de toutes les cellules du circuit.
Une fois toutes les structures de données construites et conservées en mémoire, la phase optimisation par algorithme génétique est lancée.
La figure 4.4 montre les différents liens entre les structures de données qui représentent le graphe du circuit en mémoire.

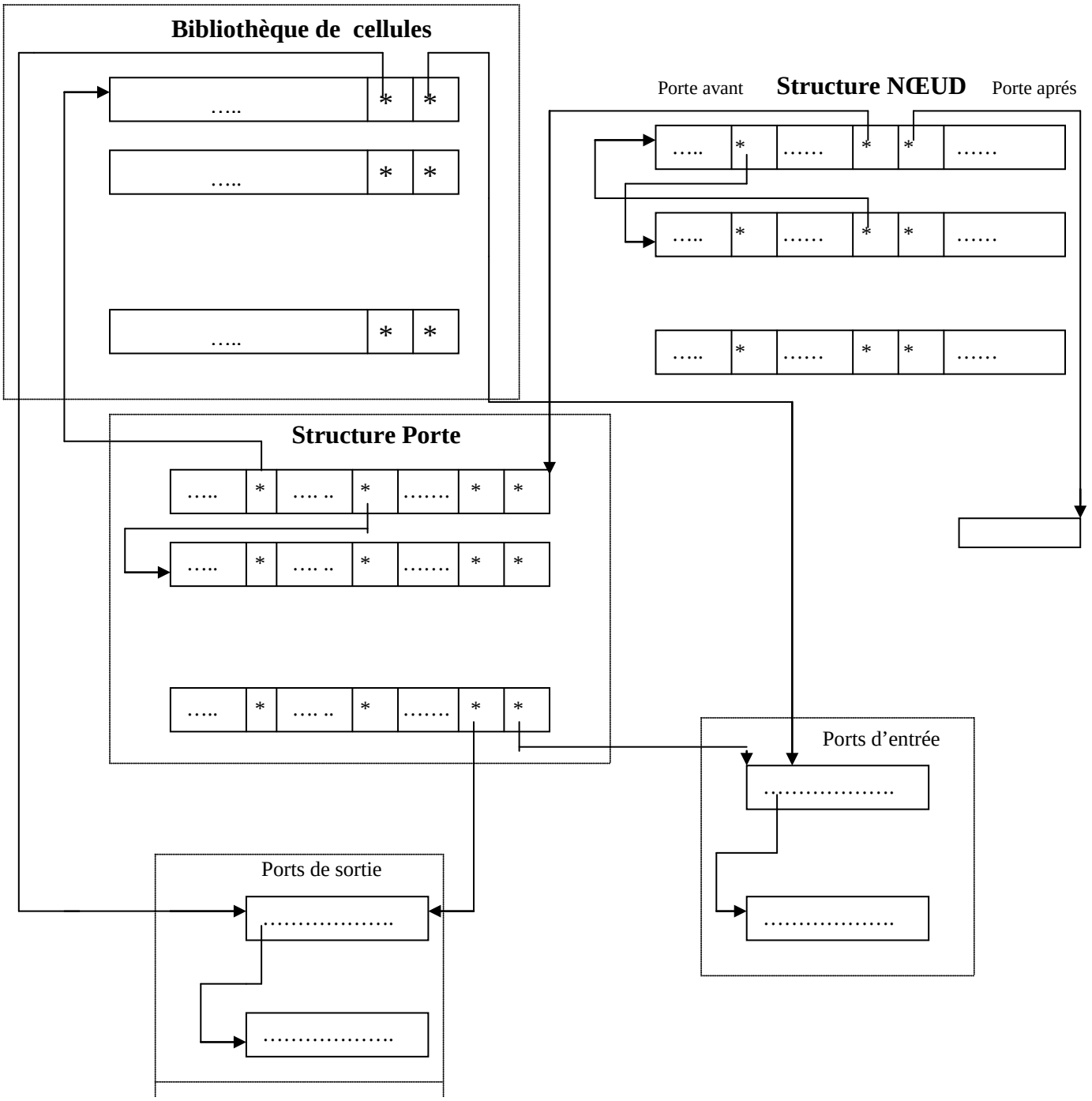


Fig. 4.4 : Illustration des structures de données dynamiques

3- <u>Algorithme génétique, Codage des paramètres</u> :

Dans notre cas, nous voulons optimiser un circuit dont la topologie est fournie en entrée. L'optimisation porte sur la surface totale du circuit et sur son délai de propagation fixé par le cahier des charges. Le circuit doit occuper une surface minimale et fonctionner avec une fréquence déterminée ou imposée par le cahier des charges.

Nous avons choisi de représenter un circuit par un chromosome. Un chromosome est une chaîne de gènes, chacun d'entre eux représente une variable du problème donc une porte du circuit pour nous. En fait, la valeur d'un gène représente le type de la porte choisi : CMOS ou BiCMOS. Par conséquent, la **taille** d'un chromosome est égale au **nombre total de portes** du circuit.

Les portes CMOS sont de petite taille. Pour cela, elles occupent une petite surface mais elles sont lentes présentent donc un délai de propagation plus important. Les portes BiCMOS quant à elles, occupent une plus grande surface mais elles sont beaucoup plus rapides donc présentent un délai de propagation plus petit. D'où, la topologie du circuit étant fournie au départ, il s'agit de choisir, pour chaque cellule, entre les portes CMOS et BiCMOS de manière à réaliser le meilleur compromis entre surface minimale et délai rapide.

Comme spécifié précédemment, un gène dans notre chromosome est une porte qui peut être de type CMOS ou BiCMOS. Le codage choisi est, par conséquent, de type binaire. Cela ne nuit aucunement au principe de généralité, comme nous le verrons dans ce qui suit, car des paramétrages sont utilisés dans le logiciel d'optimisation.

La figure 4.5 montre le codage utilisé : passage du circuit à son graphe qui le représente en mémoire puis, vers le chromosome en codage binaire.

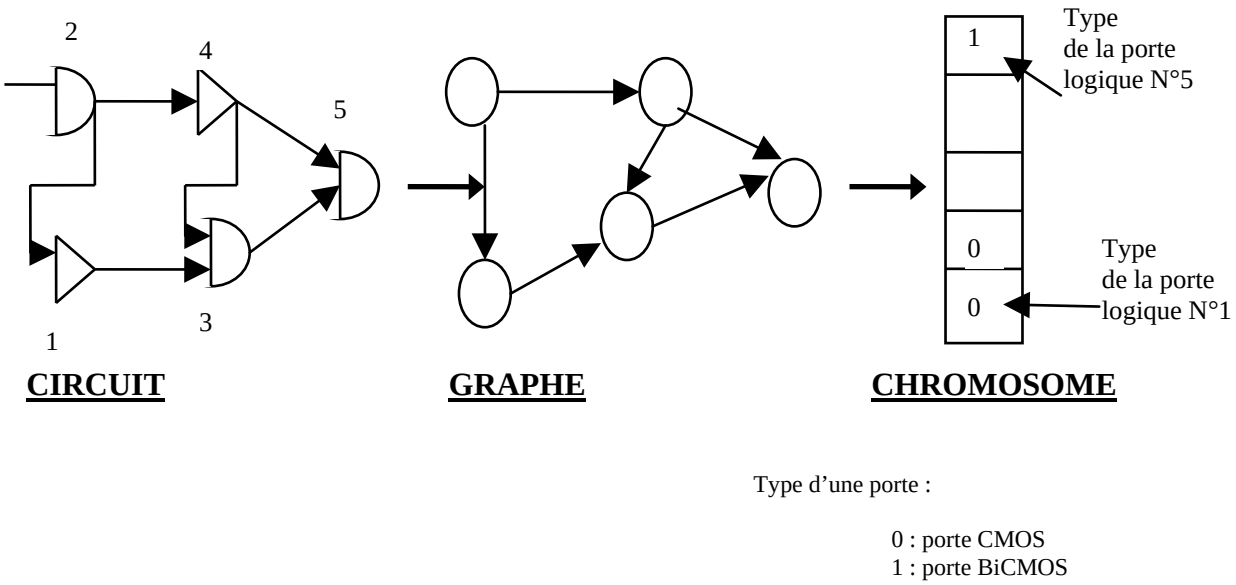


Fig. 4.5: Codage des paramètres du problème de dimensionnement

Un algorithme génétique travaille sur une population d'individus caractérisés par leurs chromosomes représentant dans notre cas, chacun une configuration possible du circuit. L'ensemble de ces chromosomes forme une génération.

La structure de données utilisée pour représenter une génération est un tableau de *"records"* ou enregistrements.

Chaque *"record"* renferme les champs suivants:

- Le chromosome : qui est une chaine de n bits 0 ou 1, où n représente le nombre de variables du problème ie: le nombre de portes du circuit qui peut atteindre des millions.
- La *"fitness"* ou fonction objectif : c'est une valeur réelle qui évalue ce chromosome. En réalité, cette valeur représente les performances de ce circuit (le chromosome) en terme de surface et de délai.
- Un vecteur contenant les valeurs maximales que peut prendre chacune des variables : dans notre cas, codage binaire donc 1.
- Un vecteur contenant les valeurs minimales que peut prendre chacune des variables : 0 pour le codage binaire.

Ces deux vecteurs permettent le paramétrage des différents types d'un même composant, les tirages aléatoires s'effectuant entre les valeurs minimales et les valeurs maximales que peuvent prendre les tailles du composant.

- Des fonctions statistiques sont calculées telles que la *"fitness"* relative et la *"fitness"* cumulative qui sont deux valeurs réelles, permettant d'effectuer la sélection des chromosomes survivants.

La figure 4.6 illustre un bloc du tableau d'enregistrement représentant une population.

Le dernier enregistrement de ce tableau contiendra le meilleur chromosome ainsi que sa valeur ie : la meilleure *"fitness"* de cette génération.

Au départ, un tableau de N individus est créé avec un tirage aléatoire pour chacune des n variables du problème effectué entre sa valeur minimale et sa valeur maximale.

Ensuite, les différents opérateurs génétiques sont appliqués. Le *"crossover"* ou croisement de deux chromosomes représente l'échange de parties de circuits pour en former deux nouveaux. La figure 4.7 illustre le croisement de deux circuits.

Quant à la mutation, occasionnellement, elle consiste à modifier le type d'une porte CMOS en BiCMOS ou inversement. Ceci crée de nouveaux circuits et permet d'enrichir l'espace de recherche. La figure 4.8 illustre la mutation d'un circuit.

Enfin, il y a évaluation de chaque individu. C'est la phase la plus délicate de l'algorithme génétique. En général, une population compte plus d'une centaine d'individus, la taille de chacun d'entre eux pouvant dépasser le million comme précédemment spécifié, il faudra évaluer les performances temporelles et de surface de tous ces circuits.

Chromosome (chaine de bits 0 ou 1)	Fitness Valeur réelle	ValMax. (chaine de bits)	ValMin. (chaine de bits)	Fitness cumul. valeur réelle	Fitness relat. val. réelle

Fig. 4.6 : Bloc enregistrement du tableau de population

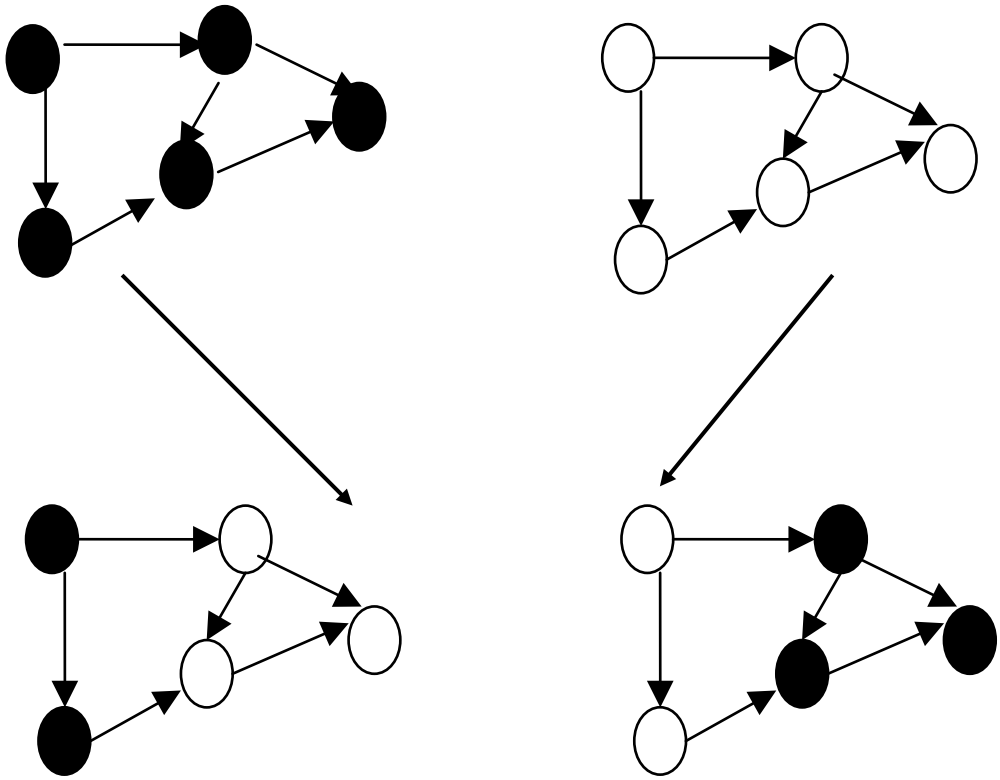


Fig. 4.7 : Illustration du *"crossover"* de deux circuits

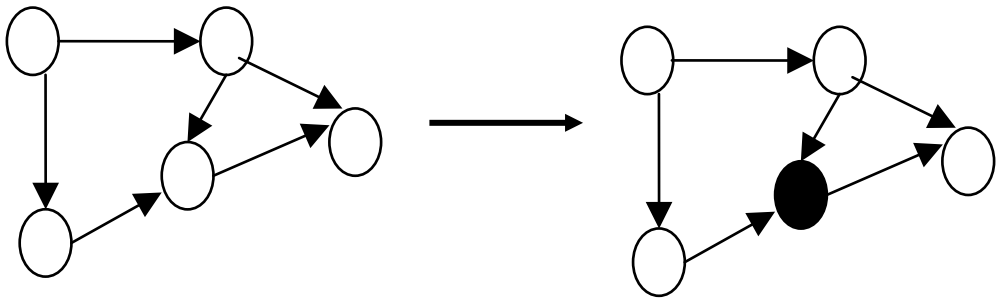


Fig. 4.8 : Illustration de la mutation d'un circuit

Ceci sera effectué par le biais de la *"fitness"*ou fonction objectif. Dans la section suivante, nous allons décrire les étapes de calculs de cette dernière en un chromosome donné.

4- Calcul de la fonction objectif ou *"fitness "*:

4-1 Introduction

L'objectif à atteindre est de minimiser la surface totale du circuit avec une vitesse de fonctionnement supérieure ou égale à celle imposée par le cahier des charges appelée T : contrainte temporelle. D'où, notre fonction objectif doit comporter deux termes :

- l'un minimisant la surface ,
- l'autre le délai de propagation du circuit.

Par conséquent, notre problème d'optimisation peut être formulé comme suit [voir chapitre III] :

$$(1) \quad P = \left| \begin{array}{l} \underset{a_i}{\text{Min}}\ a \\ \text{avec } D \leq T \end{array} \right.$$

4-2 Calcul de la surface d'un circuit :

La surface totale du circuit dépend de la surface de chaque porte ainsi que de la surface occupée par les interconnexions entre ces portes. Seulement, en modifiant la surface des portes, les interconnexions ne changent pas ou très peu donc de manière très insignifiante pour la surface totale du circuit. D'où la surface à minimiser concerne plus précisément la surface des portes.

$$(2) \quad a = \sum_{i=1}^{n} a_i$$ où S_i représente la surface de la porte N° i et n le nombre total des portes du circuit.

4-2-1 Calcul de la surface d'une porte a_i :

Dans la structure de données qui décrit la liste des portes, il est spécifié un champs représentant le type de la porte courante qui est soit CMOS (0) ou BiCMOS (1) dans notre cas, ainsi que l'adresse de cette porte dans la bibliothèque des cellules. A partir de celle-ci, nous pouvons accéder directement à tous les paramètres de cette porte pour chacun de ses types ; à savoir sa largeur, sa hauteur, ses ports d'entrées, ses ports de sortie, ...

La surface occupée par une porte est égale au produit de sa hauteur par sa largeur.

(3) $a_i = haut_i[j] * larg_i[j]$

avec j = type de la porte (j=0 ou 1 dans notre cas.)

4-3 Calcul du délai de propagation total du circuit : Le délai de propagation total du circuit est évalué par la somme des délais de toutes les portes situées sur le chemin le plus lent du circuit[3], appelé chemin critique.

Il existe plusieurs méthodes de détermination du chemin critique. Parmi ces méthodes, celle qui consiste à calculer les temps-mort [10].

Les notions de temps-mort arc et temps-mort nœud datent de l'apparition de l'algorithme PERT [24] mais elles n'ont été adaptées à l'analyse de circuits que bien plus tard.

Le temps-mort arc est une information locale au nœud. Il représente la différence entre le temps d'arrivée critique du signal à la sortie du nœud Tas et la somme entre le délai D du nœud et le temps d'arrivée du signal à l'entrée Tae.

Sa = Tas – (Tae + D).

Le chemin critique est déterminé [10] en suivant les arcs ou les nœuds dont le temps-mort est nul.

Le chemin critique d'un circuit est aussi déterminé par les algorithmes d'exploration des graphes tels que les algorithmes DFS ou BFS [voir chap. I].

Le délai de propagation total du circuit est exprimé par :

(4) $\mathbf{D = \sum_{i=1}^{k} d_i}$ d_i = délai de la porte N°i du chemin critique du circuit
k = nombre de portes situées sur le chemin critique.

4-3-1 Calcul du délai de propagation d_i d'une porte :

Le modèle de délai de propagation d'une porte est un modèle RC [1][5] qui dépend linéairement de la capacité du noeud attaqué par cette porte. A partir des équations précédentes, nous déduisons :

$$\mathbf{d_i = a * C_i + b}$$

où a et b sont les paramètres électriques de la porte, disponibles dans la bibliothèque des cellules pour chacun de ses types. Nous avons choisi ce modèle pour sa simplicité et surtout pour la disponibilité de ces paramètres dont l'accés se fait directement par l'adresse de la porte courante vers la bibliothèque.

D'où : **(6)**

$$\mathbf{d_i = a[j] * C_i + b[j]}$$

j = 0,1 selon le type de la porte courante

C_i = capacité du noeud attaqué.

4-3-2 Calcul de la capacité d'un nœud C_i :

La capacité d'un nœud i donné [1] est égale à la somme des capacités des ports d'entrée de toutes les portes connectées à ce nœud, à laquelle il faudra rajouter la capacité d'interconnexion. Dans la structure de données proposée de la liste des nœuds, il est spécifié un pointeur permettant l'accès direct à la liste des pointeurs de toutes les portes connectées à ce nœud. A partir de l'adresse d'une porte, son pointeur vers la bibliothèque des cellules nous fournit la liste de ses ports d'entrée.

Cette dernière nous fournit par accès direct, dans la structure de données des ports, la capacité des ports d'entrée des portes concernées : il suffit de les sommer :

$$(7) \quad C_i = C_{ni} + \sum_{J=1}^{Nb\_gate} C_j p_{in}$$

où : $C_j p_{in}$ est la capacité du port d'entrée de la jième porte connectée au noeud i.
Nb_gate = Nombre de portes connectées à ce noeud.
C_{ni} = capacité d'interconnexion.

4-3-3 Calcul du chemin critique du circuit par l'algorithme BFS :

Un circuit peut posséder plusieurs chemins critiques, équivalents du point de vue poids, le poids d'un chemin étant défini par la somme des délais des portes situées sur ce chemin. Pour cela, nous avons utilisé l'algorithme BFS (Depth-First-Search) [voir Chap.I] .

Aprés avoir évalué le délai D d'un circuit ainsi que sa surface S et sachant que T correspond à la contrainte temporelle fixée par le cahier des charges sur le délai de propagation du circuit considéré, nous avons modélisé la valeur de la fonction ″objectif″ par la méthode de pénalisation qui consiste à transformer un problème d'optimisation avec contraintes en un problème sans contraintes en intégrant les contraintes dans la fonction objectif. Ceci est effectué grâce à un facteur de pénalisation dont le rôle est d'établir des priorités entre les contraintes concurrentes.

$$F = \begin{cases} \text{Min}[\sum_{i=1}^{n} (a_i) + \lambda * ((\sum_{j=1}^{k} D_j) - T)\,] & \text{si } ((\sum_{j=1}^{k} D_j) - T) > 0 \\ \text{Min}(\sum_{i=1}^{n} a_i) & \text{si } ((\sum_{j=1}^{k} D_j) - T) = 0 \end{cases}$$

où : n est le nombre de portes, k est le nombre de portes du chemin critique,

$a_i \in R$, $d_j \in R$

En d'autres termes et pour simplifier, posons : $a = \mathbf{Min}(\sum_{i=1}^{n} a_i)$

et $\mathbf{D = (\sum_{j=1}^{k} D_j)}$

nous obtenons :

$$\mathbf{F} = \begin{cases} \mathbf{(S + \lambda*(D - T))} & \mathbf{si\ (D\text{-}T) > 0} \\ \mathbf{S} & \mathbf{si\ (D\text{-}T) \leq 0} \end{cases}$$

Où : n est le nombre total de portes logiques du circuit et k est le nombre de portes situées sur le chemin critique.

λ est le facteur de pénalité.

Le facteur de pénalité sert à favoriser la surface par rapport au délai ou bien l'inverse. Tout dépend du cahier des charges : si nous voulons favoriser l'optimisation de la surface (surface la plus minimale possible) par rapport au délai, alors le facteur λ est choisi assez petit. Par contre, si le délai est désigné comme prioritaire lors de l'optimisation, il faut dans ce cas choisir λ assez grand de manière à établir un équilibre entre l'optimisation de la valeur du délai et celle de la surface. Ce facteur sert également à pénaliser les circuits qui ne vérifient pas la contrainte temporelle : lorsque l'écart (D - T) est grand et λ choisi grand, la fonction objectif atteint une grande valeur, le circuit lui correspondant est rejeté par l'algorithme génétique car son but est de minimiser cette valeur.

L'optimisation prendra en compte les deux membres de la fonction ″objectif″ tant que le délai du circuit est plus grand que la contrainte temporelle T, autrement dit, tant que : $(D - T) > 0$

Dés que l'écart (D – T) devient négatif ou nul, ie : le délai obtenu respecte la contrainte temporelle imposée, l'optimisation portera uniquement sur la surface. L'algorithme génétique continue à chercher des configurations de circuits possibles qui favorisent une surface occupée par le circuit la plus petite et qui respectent le délai D<=T.

Il est à noter, par ailleurs, que la complexité de notre fonction objectif ou *"fitness"* est linéaire du fait de la linéarité de la somme pour le calcul de la surface totale du circuit ainsi que la linéarité de l'algorithme BFS pour le calcul de son délai de propagation.

A présent, nous allons présenter un algorithme en pseudo-code qui résume le travail d'optimisation proposé ci-dessus.

Début

- Lire la bibliothèque de cellules ;
- Lire fichier d'entrée contenant topologie du circuit ;
- Initialiser() ; /les paramètres de l'algorithme génétique tels que le tableau de population, les probabilités de tirage, le nombre d'itérations, le nombre d'individus,…/
- Construire() ; /construit le graphe du circuit sous forme de listes chaînées en mémoire/
- Générer population initiale aléatoirement ;
- Evaluer (pop) ; /evaluer la population initiale/
- Tant que (i≤nb_itération)

  Faire selection() ;

  Croisement() ;

  Mutation() ; /changement eventuel d'une porte CMOS en BiCMOS ou inversement./

  Evaluation() ; /calcul de la fitness en chaque chromosome/

  /par calcul du délai et de la surface du circuit correspondant/

  Fait ;

Affiche le meilleur circuit ;

Fin.

5- **Tests et résultats expérimentaux**

Dans ce qui suit, nous allons présenter quelques résultats expérimentaux effectués, comme il a été déjà spécifié, sur des circuits combinatoires *"benchmarks"* de ISCAS'85 [27], décrits dans le tableau suivant :

Nom du circuit	Nombre de portes du Circuit
C17	06
C432	180
C880	263
C499	400
C1908	442
C1355	620
C2670	733

Tableau des circuits benchmark ISCAS'85.

Ces tests ont été menés dans le but de tester notre logiciel d'optimisation des performances temporelles et spatiales des circuits intégrés.

Afin de pouvoir illustrer le compromis vitesse/surface, nous avons effectué plusieurs calculs sur ces circuits.

Au départ, nous avons fixé T=0 la contrainte temporelle ce qui oblige le logiciel d'optimisation à retenir les solutions sans contrainte sur le délai de propagation car : (D - T) = D dans la fonction objectif

Pour chaque circuit, nous avons calculé le délai de base et la surface totale sans effectuer aucune optimisation, ie : pour une configuration du circuit avec toutes ses portes de type CMOS, ainsi que la surface du circuit si on avait opté pour une configuration de celui-ci avec toutes ses portes de type BiCMOS.

Nom du circuit	Surface du circuit CMOS	Surface du circuit BiCMOS	Délai du circuit CMOS
C17	6 000	30 000	1242.5
C432	180 000	892 100	5242
C880	263 000	1 303 850	3709
C499	400 000	1 982 700	8914.25
C1908	462 000	2 192 100	9529.16
C1355	620 000	3 085 500	9873.3
C2670	733 000	3 630 850	3873.4

Nous avons choisi plusieurs valeurs du facteur de pénalisation :

- Elevé (λ=200) pour favoriser l'optimisation du délai de propagation par rapport à la surface totale du circuit.
- Moyen (λ=100) pour établir l'équilibre entre le délai et la surface.
- Faible (λ=50) pour favoriser un peu l'optimisation de la surface par rapport au délai.
- Très faible (λ=10) : cela oblige l'optimisation à porter beaucoup plus sur la surface que sur le délai.

En une première phase nous avons implémenté **un croisement à un point.**
Nous avons mené nos tests avec une population de **200 individus** et un **nombre d'itérations égal à 200**. Les **probabilités de croisement** et de **mutation** ont été fixées à **0.8 et 0.1 respectivement**.

Les résultats expérimentaux obtenus sont résumés dans les tableaux suivants .

λ=200

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimisat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	26 000	846	900	5
C432	452 800	2070	2500	79
C880	698 350	2890	2900	110
C499	1 048 600	7510	7500	164
C1908	1 182 600	7670	7800	187
C1355	1 697 750	7925	8000	271
C2670	2 077 000	2652.8	3000	346

Tableau n°1

λ=100

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimisat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	22 000	897	900	4
C432	448 850	2435	2500	70
C880	646 700	2899	3000	90
C499	1 060 600	7572	7500	167
C1908	1 152 150	7715	7800	182
C1355	1 638 000	8289	8000	256
C2670	1 958 750	3230	3000	310

Tableau n°2

λ=50

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimisat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	18 000	961	900	3
C432	437 100	2511	2500	62
C880	650 650	2950	3000	98
C499	1 005 500	7600	7500	153
C1908	1 110 500	8093	7500	179
C1355	1 635 900	8477	8000	258
C2670	1 942 800	3264	3000	315

Tableau n°3

λ=10

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimisat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	6 000	1242	900	0
C432	431 100	2761	2500	70
C880	635 050	3150	3000	94
C499	1 000 150	7735	7500	156
C1908	1 145 500	8566	7800	171
C1355	1 630 900	9292	8000	245
C2670	1 937 850	3386	3000	303

Tableau n°4

Commentaires :

A travers les résultats obtenus, il est clairement constaté que plus la valeur du facteur de pénalisation λ est faible et plus le délai de propagation est plus élevé mais la surface totale du circuit est plus faible. Inversement, plus la valeur de λ est élevé, plus le nombre de portes BiCMOS rajoutées est plus élevé et donc le délai de propagation est plus faible mais le coût en surface est plus grand.

Ainsi, dans le tableau n°1, nous remarquons que la surface obtenue est grande mais que le délai de propagation respecte la contrainte temporelle T. Le nombre de portes BiCMOS rajoutées est élevé, il se situe entre 40% et 50% du nombre total des portes du circuit, ce qui permet un délai rapide mais une surface élevée.

Dans le tableau n°4 avec un facteur de pénalisation faible, l'inverse est constaté.

D'où nous pouvons conclure que le compromis délai/surface a été pris en charge efficacement par cette approche.

Par ailleurs, le nombre de portes BiCMOS rajoutées dépend également du facteur λ, ce qui est normal car si nous optons pour la miniaturisation le nombre de portes BiCMOS doit être faible puisque ces portes occupent plus de surface que les portes CMOS. Le cas inverse, le nombre de portes BiCMOS rajoutées doit être important pour pouvoir diminuer le délai de propagation du circuit, ce qui est obtenu en augmentant le facteur λ..

Il est à remarquer également que parfois les délais de propagation obtenus ne vérifient pas tout à fait les contraintes temporelles, ceci est dù au fait que l'algorithme génétique demeurera toujours une méta-heuristique et donc une méthode qui approche au mieux la solution sans garantir l'optimalité.

En une seconde phase, nous avons implémenté **un croisement à deux points**.

Les résultats expérimentaux obtenus sont résumés dans les tableaux suivants :

λ=200

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimi ;sat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	26 000	846.50	900	5
C432	440 950	2070	2500	66
C880	690 500	3005.5	3000	108
C499	1 072 500	7404.7	7500	170
C1908	1 158 200	7224	7800	181
C1355	1 677 500	8009	8000	266
C2670	1 970 000	3018.8	3000	313

Tableau n°5

λ = 100

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimisat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	22 000	897	900	4
C432	429 150	2507	2500	63
C880	656 700	2995.2	3000	100
C499	1 037 050	7357.3	7500	161
C1908	1 142 350	7553.5	7800	177
*C1355	1 638 000	8289	8000	256
*C2670	1 958 750	3230	3000	310

Tableau n°6

λ = 50

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimisat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	18 000	961	900	3
C432	417 150	1667.3	2500	60
C880	642 950	2977.66	2900	96
C499	993 450	8000	7500	150
C1908	1 119 000	8036.8	7800	171
C1355	1 614 050	8967.5	8000	250
C2670	1 931 200	3356	3000	303

Tableau N°7

λ=10

Nom du circuit	Surface du circuit après optimisation	Délai du circuit après optimisat.	Valeur de la contr. Tempor. T	Nbre de Portes BiCMOS rajoutées
C17	6000	1242	900	0
C432	433 000	1625	2500	64
C880	634 750	3647	2900	94
C499	969 650	7915	7500	144
C1908	1 119 050	7847	7800	171
C1355	1 609 700	9176	8000	249
C2670	1 946 350	3476	3000	307

Tableau N°8

Commentaires :

En comparant les résultats obtenus au tableau n°1 avec ceux obtenus au tableau n°5, nous constatons une meilleure qualité des résultats obtenus avec un croisement à deux points par rapport au croisement à un point. En effet, la contrainte temporelle imposée pour le circuit est vérifiée avec un coût moindre en nombre de portes BiCMOS rajoutées et donc en surface total du circuit.

Cependant , nous avons remarqué un temps d'exécution plus long.

6- Conclusion : Les résultats obtenus sont satisfaisants car d'une part, dans la plupart des cas ils vérifient la contrainte temporelle, et d'autre part ils sont obtenus après un temps d'exécution linéaire : plus la taille du circuit traité est grande et plus le temps de traitement augmente mais reste raisonnable, de l'ordre de quelques minutes sur une machine de type PeintiumII .

Néanmoins, les algorithmes génétiques sont des méta-heuristiques et non pas des méthodes exactes dans la mesure où ils approchent aux mieux la solution optimale sans toutefois garantir la qualité ou la précision des résultats. Ces derniers peuvent être améliorés dans une seconde phase mais au détriment du temps de leur obtention.

Cependant, la phase la plus coûteuse en temps de calcul est bien la phase d'évaluation de tous les chromosomes d'une génération, la taille de celle-ci étant déjà appréciable sans compter la taille d'un chromosome qui représente, comme déjà spécifié, le nombre de portes logiques qui composent tout le circuit et qui peut atteindre des millions.

Effectuer les évaluations de tous les chromosomes durant toutes les étapes de l'algorithme génétique (200 générations de 200 chromosomes chacune) revient à calculer les délais de propagation ainsi que les surfaces totales de 40 000 circuits, la taille de chacun d'entre eux dépasse un million de portes logiques.

Ceci a motivé notre intérêt porté à l'étude de la parallélisation de notre approche génétique, notamment l'évaluation des chromosomes d'une génération. Le

prochain chapitre présente quelques notions de parallélisme et d'architectures parallèles ainsi que des schémas de parallélisation d'algorithmes génétiques. Il présentera également une approche génétique parallèle pour la résolution du problème de dimensionnement des portes logiques d'un CI.

Chapitre V

Etude de la parallélisation d'un algorithme génétique pour la résolution du problème ″Gate Sizing″ des circuits intégrés VLSI.

1- Introduction

Encore récemment, bon nombre d'applications étaient considérées comme impossibles à être mises en œuvre, non pas faute d'une modélisation mathématique exacte ou de méthodes numériques mais faute de calculateurs capables de résoudre ces problèmes, vues les grandes puissances de calculs requises. Ces applications possédaient des représentations mathématiques adaptables aux traitements parallèles mais les machines disponibles n'étaient pas en mesure de mener à bien ce type de traitement. Ce qui a incité des chercheurs dans le monde à connecter plusieurs processeurs (voir des milliers actuellement) afin d'obtenir une machine plus puissante, capable de traiter en simultaneité de grands flots de données. Cela n'aurait pu être réalisé si l'on n'avait pensé

préalablement à développer des logiciels parallèles d'exploitation de ces machines : c'est ainsi que sont nées les architectures à multiprocesseurs et le parallélisme.

Ce dernier et plus particulièrement les architectures et algorithmes parallèles sont devenus d'une grande importance dans le domaine de l'informatique moderne. Leurs évolutions sont très rapides et très diversifiés [1] : ils sont passés des supercalculateurs vectoriels aux réseaux de stations de travail, des bases de données distribuées à la simulation numérique etc... Nous pouvons dire que c'est le grand nombre de domaines d'application qui a suscité une telle évolution. Citons à titre d'exemple la météorologie, l'imagerie, le génie nucléaire, l'aéronautique ainsi que toutes les applications en temps réel d'une façon générale et qui ne peuvent plus s'en passer.

Actuellement, nous pouvons dire qu'en raison des grandes puissances de traitement qu'il offre, le parallélisme a acquis une importance capitale dans le monde de l'informatique. Nous présentons dans la section suivante un bref historique des machines parallèles conçues dans le monde.

2- **Historique** [32]:

Les années 70 ont donné naissance aux toutes premières machines parallèles : citons l'ILLIAC IV une machine de 64 processeurs élémentaires et qui a été conçue à l'université de l'Illinois aux USA en 1970.

Au milieux des années 70, Cray et Control Data donnèrent le jour à des supercalculateurs afin de pouvoir traiter des applications scientifiques telles que la météorologie, la construction aéronautique, …

Ce n'est qu'au début des années 80 que des machines parallèles ont été construites pour une plus large utilisation. Citons l'ICL Distributed Array Processor (DAP) en Grande Bretagne, la CosmicCube aux USA, de type hypercube. De nos jours, chaque 2 à 3 ans, une nouvelle génération de machines parallèles apparaît, afin de pouvoir augmenter les vitesses de traitement de jour

en jour exigées. Citons l'iPSC et la Paragon d'Intel, les Connection Machines CM2 et CM5 de Thinking Machines et les T3D et T3E de Cray Research.

De plus, actuellement plusieurs stations de travail et de PC reliés via un réseau local de type LAN() peuvent former une machine parallèle virtuellement. Ces machines communiquent par le biais d'une bibliothèque de communications tel que PVM, décrit en annexe.

La grande diversité des machines parallèles impose leur classification. En conséquence, nous décrivons dans ce qui suit, les différentes classifications proposées depuis la fin des années 60.

3- Classification des machines [31] :

Flynn a proposé une classification des machines par séquencement des programmes, selon le flot de données d'entrée sur un groupe d'instructions. Il énuméra alors quatre types d 'architectures essentiels :

- SISD (Single Instruction, Single Data) : ce type regroupe les machines classiques à monoprocesseur et dont la puissance de traitement reste liée à la vitesse d'exécution d'une instruction.
- SIMD (Single Instruction, Multiple Data): Un flot d'instruction est exécuté sur un ensemble de données stocké dans les mémoires des différents processeurs de la machine. Les processeurs élémentaires n'incluent pas d'unité de contrôle dans ce cas car chacun d'entre eux déroule la même instruction sur ses données locales. Il y a une seule unité de contrôle qui sert au décodage des instructions.
- MISD (Multiple Instruction, Single Data) : le même flot de données subit différents traitements sur les différents processeurs de la machine. Exemple : calcul des puissances de X : une variable fournie en entrée, comme l'illustre la figure 5.1 . Actuellement, aucune machine de ce type n'a été construite [4].

- MIMD (Multiple Instructions, Multiple Data) : ce sont les véritables machines massivement parallèles, constituées de plusieurs processeurs exécutant différents codes sur plusieurs flots de données. Dans cette architecture, les différents processeurs intègrent une unité de contrôle. Ils exécutent des flots d'instructions sur des flots de données différents. La machine MIMD fonctionne en mode asynchrone car elle ne comporte pas d'horloge globale.

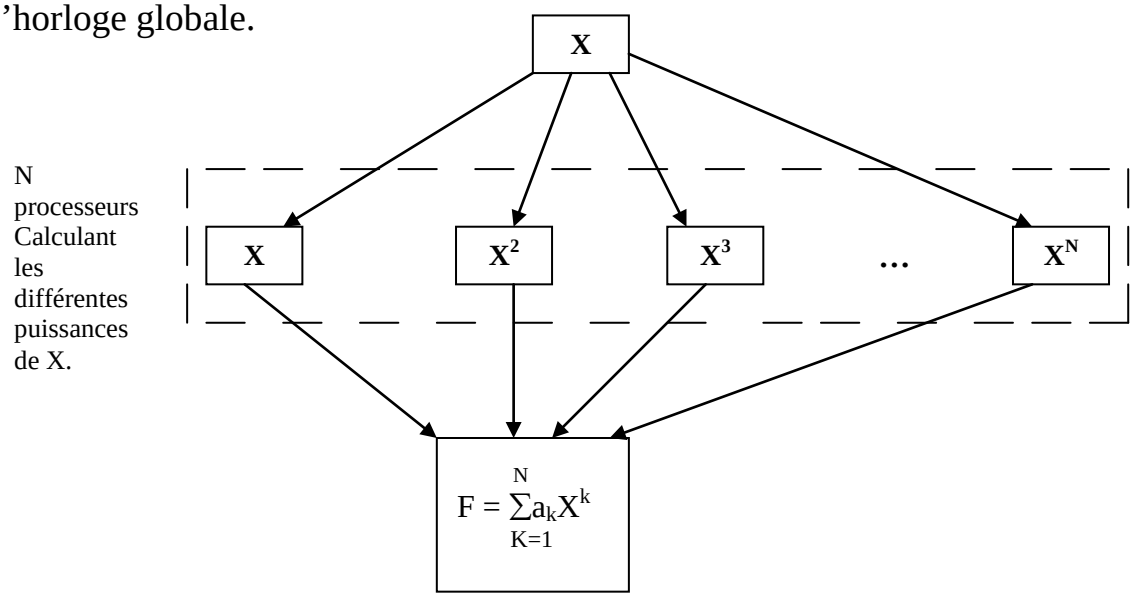


Fig. 5.1 : <u>Illustration d'une architecture MISD</u>.

Cette classification s'avère de nos jours insuffisante car on parle actuellement de réseaux de processeurs, sans compter les types d'échange entre la mémoire et les processeurs (mémoire partagée et/ou distribuée) ainsi que le type des échanges entre ces processeurs.

3-1 <u>Les trois composantes clés d'une architecture parallèle</u> :

Une architecture est caractérisée par [34] :

- le processeur : il peut être de deux types
  - généraux RISC
  - custom : spécifique à une application donnée.
- La mémoire : elle peut être :
  - partagée
  - distribuée

- Le réseau d'interconnexion : il assure les communications entre les différents processeurs de l'architecture. Il peut être :
    - statique : grille 2 ou 3D, hypercubes, anneaux, arbres binaires ,…
    - dynamique : bus, multi-étages, crossbar,…

Actuellement, il existe trois grandes classes de machines parallèles.

3-2 <u>Les machines parallèles à mémoire partagée</u> [33] : se caractérisent par une mémoire unique globale à laquelle tous les processeurs vont accéder. Ces architectures ont été utilisées avec succès pour exploiter le parallélisme, bien qu′elles ne soient pas extensibles et présentent un goulot d'étranglement lors de l'accès à la mémoire, dés que le nombre de processeurs dépasse un certain seuil. Voir figure 5.2.

3-2 <u>Les machines parallèles à mémoire distribuée</u> : ces architectures sont extensibles à plusieurs centaines et même milliers de processeurs. C'est en fait un ensemble de processeurs disposant chacun d'entre eux d'une mémoire locale [voir figure 5.3] et communiquant via un réseau d'interconnexion. Actuellement des réseaux locaux de stations de travail forment une machine parallèle à mémoire distribuée et sur laquelle les applications sont exécutées grâce à une couche logicielle système distribuée comme PVM [voir annexe]. L'extensibilité de ces machines donne le potentiel pour l'exécution d'applications massivement parallèles à un coût raisonnable.

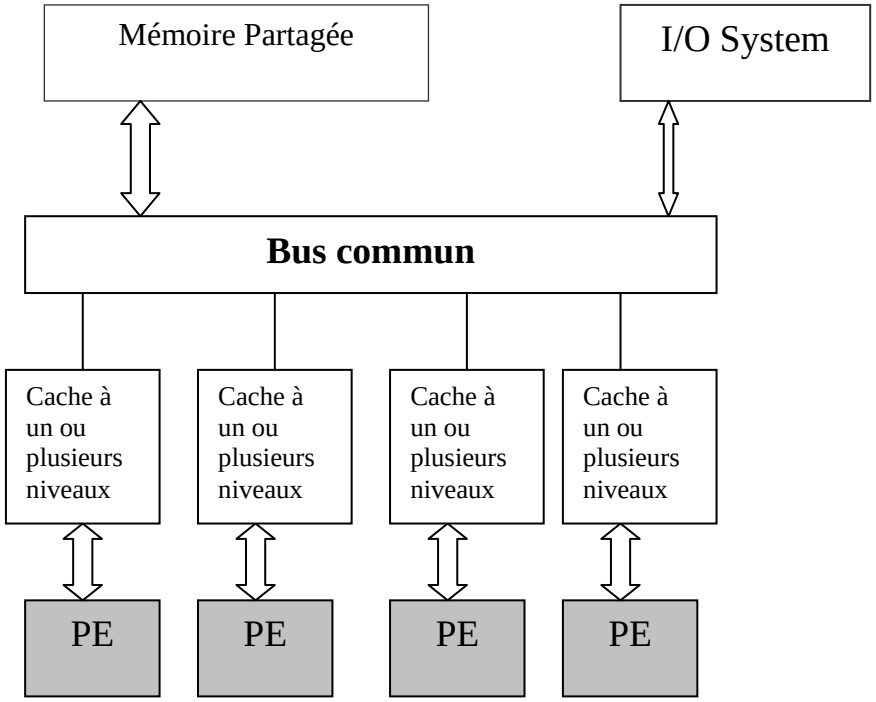


Fig. 5.2 : <u>Architecture MIMD à mémoire et bus partagés.</u>

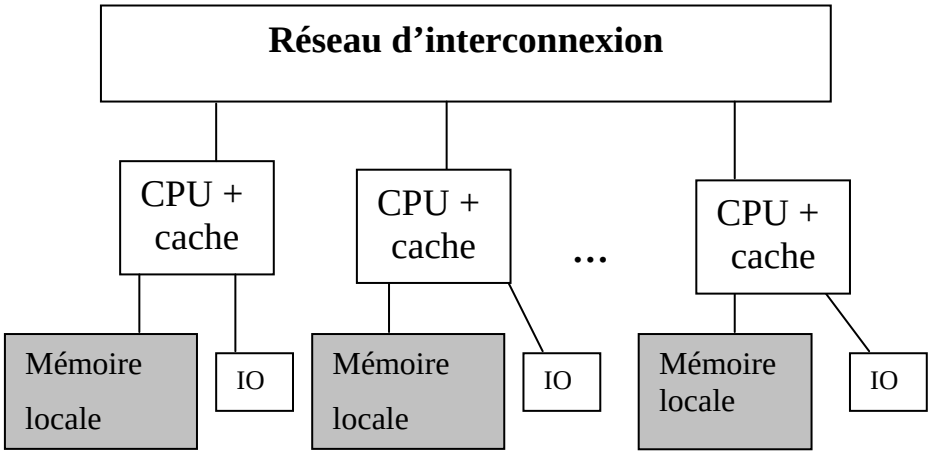


Fig. 5.3 : <u>Machines à mémoire distribuée</u>.

De nos jours, les machines à mémoire distribuée sont de plus en plus utilisées : les processeurs disposant de mémoire locale rapide travaillent par passage de message.

3-4 <u>Les machines à mémoire distribuée-partagée</u> :

Ces dernières années ont vu la naissance d'une nouvelle classe : les machines parallèles à mémoire partagée distribuée (*DSM* : *Distributed Shared Memory*)[42, 43]. Dans les environnements de programmation parallèles sur un réseau de stations de travail, ces dernières ont recours au passage de messages comme moyen de communication (*Messages passing*). Par ailleurs, dans les applications parallèles, le partage de données est un problème inévitable. Or dans les réseaux de stations de travail, ces dernières ne partagent pas d'espace d'adressage physique. Pour pallier à cet inconvénient majeure, il fallait recourir à un espace logique de données partagées entre sites afin de pouvoir échanger des messages et répliquer des données. C'est ainsi que les systèmes à mémoire partagée-distribuée sont nés. Citons à titre d'exemple les systèmes Phosphorus **[42]** voir figure 5.4, WARPphos **[43],** et TreadMarks **[44].**

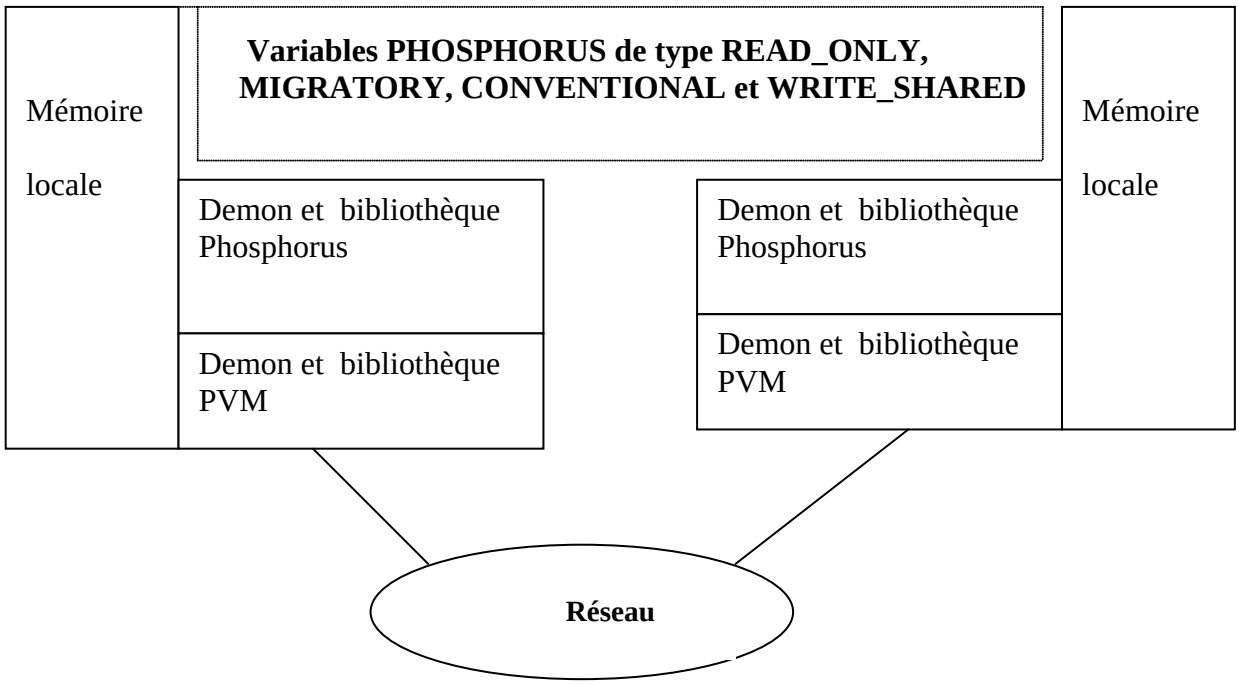


Fig. 5.4 : <u>Phophorus, un système à mémoire distibuée partagée</u>

En réalité, toutes ces architectures présentent chacune des avantages, selon les besoins d'utilisation : [33]

- dans les architectures à mémoire partagée, la difficulté est moindre dans la programmation lorsque les schémas de communication inter-processeurs deviennent complexes ou varient dynamiquement. Ceci simplifie également la conception de compilateurs pour ces machines. Il y a également moins de temps à passer en surveillance des communications par les processeurs.
- Dans les architectures à mémoires distribuées, la communication est explicite et oblige les programmeurs et les compilateurs à prendre en charge cet aspect mais c'est une manière d'étendre la mémoire à moindre coût, surtout lorsque la plupart des accès sont en mémoire locale au niveau de chaque nœud [33].
- Dans les architectures à mémoire partagée-distribuée (Distributed shared memory) la difficulté réside surtout dans l'implémentation de ces systèmes.

Notons pour conclure, qu'on distingue deux types de parallélisme : un parallélisme de données dans lequel le même code est exécuté sur les différents processeurs de l'architecture sur différents flots de données, et un parallélisme fonctionnel où la même donnée est traitée différemment sur les processeurs de la machine parallèle.

Après avoir passé en revue les architectures parallèles, nous allons décrire sommairement une manière d'augmenter leurs performances.

4- **Performances d'une architecture parallèle [35]:**

Le parallélisme a suscité un grand intérêt ces dernières années dans les milieux de l'informatique en raison des puissances de traitement qu'il offre. Seulement, pour la plupart des applications, son exploitation n'est pas toujours efficace étant donnée la mauvaise répartition du traitement sur les ressources disponibles. L'équilibrage de la charge de travail est de nos jours, un domaine de recherche

très actif, afin de pouvoir exploiter au mieux le parallélisme et les architectures parallèles et augmenter ainsi les vitesses de traitement.

Obtenir des performances signifie maximiser l'accélération qui est, pour une application donnée, le rapport entre le temps d'exécution en séquentiel et le temps d'exécution en parallèle. Pour cela, nous avons le choix entre :

- déterminer le nombre de processeurs optimal pour l'exécution d'une application déjà parallélisée.
- trouver pour un nombre de processeurs fixé, la meilleure répartition possible du travail de l'application. Cette deuxième méthode est plus réaliste que la première.

Il n'existe pas encore de mécanisme d'équilibrage dynamique de charge universel. Toutes les applications dont le comportement est connu avant l'exécution ne posent pas de problèmes au moment de l'équilibrage de charges. Néanmoins, cela reste un domaine de recherche pour les applications dont le comportement est estimé dynamiquement à l'exécution, ou bien n'est pas prévisible en temps opportun. En fait, c'est un problème de placement qui consiste à répartir les tâches d'une application sur les processeurs d'une machine parallèle de façon à optimiser une fonction de coût.

Par ailleurs, les performances d'un algorithme sont aussi à évaluer. Pour cela, soit un problème de taille N et soit un algorithme s'exécutant sur une machine à P processeurs pour le résoudre. Deux paramètres évaluent les performances de cet algorithme :

- Le *"Speedup"* $S(P,N) = \frac{T^{Best}(N)}{T(P,N)}$ où T^{Best} = meilleur temps d'exécution obtenu en séquentiel. T(P,N) = temps d'exécution obtenu avec P processeurs
- Efficacité E(P,N) = S(P,N) / P

Idéalement, l'accélération ou *"speedup"* qui est le gain obtenu en parallèle par rapport à l'exécution en séquentiel, devrait atteindre P le nombre de processeurs.

En conséquence l'efficacité serait égale à 1. En réalité ces chiffres ne sont jamais atteints à cause des temps de communications inter-processeurs.

Après avoir présenté quelques architectures parallèles et le parallélisme, les sections suivantes décriront différents schémas de parallélisation d'un algorithme génétique en premier lieu puis les étapes de parallélisation d'un algorithme génétique appliqué à la résolution d'un problème NP-Hard qu'est le *"Gate Sizing"* des circuits intégrés VLSI.

5- Etude de la parallélisation d'un algorithme génétique

5-1 Rappels : Nous avons vu [voir chap. II] les principes de base d'un algorithme génétique, à savoir qu'il manipule une population de chromosomes, ces derniers représentant chacun, une solution possible du problème traité. Il a été spécifié la manière dont une génération est créée au départ, les individus sélectionnés, appariés puis reproduits en effectuant des croisements ou *"crossover"* à des endroits bien spécifiques. En dernier, une mutation occasionnelle (avec une faible probabilité) est effectuée dans le but de créer de nouveaux individus. Après cela, la phase évaluation de chaque nouveau chromosome ainsi obtenu est entamée. Ne seront retenus pour la génération future que les meilleurs chromosomes selon un critère donné.

La nature intrinsèque d'un algorithme génétique présente un avantage majeure : son aptitude à être parallélisé. Dans la section suivante, nous allons présenter différents schémas possibles de parallélisation d'un algorithme génétique.

5-2 <u>Schémas de parallélisation d'un algorithme génétique</u> : [39, 40, 41, 45]

Les algorithmes génétiques sont fondamentalement parallèles, capables de résoudre efficacement des problèmes d'optimisation combinatoire. Le recours au parallélisme permet en effet d'explorer une population de plus grande taille que celle d'un algorithme génétique séquentiel et d'augmenter son efficacité en réduisant les temps de calcul. A ce jour, les méthodes de parallélisation d'un algorithme génétique peuvent être classées comme étant :

- globale
- de migration
- de diffusion.

Ces méthodes reflètent différentes manières d'exploiter au mieux le parallélisme d'AG, la nature de la population ainsi que les mécanismes de recombinaison utilisés et enfin la communication inter-processeurs.

5-2-1 <u>La méthode globale</u> : Cette méthode reflète le modèle Maitre/Esclave [36, 38, 39, 45] souvent utilisé pour paralléliser un AG. A partir de ce modèle, les opérateurs génétiques sont organisés de différentes manières de façon à augmenter la vitesse de traitement. Tout dépend du domaine d'application et du flot de données à traiter.

Basée sur un modèle Maître/Esclave, elle traite une population entière comme une race unique [9,16]. La figure 6.4a illustre une organisation où les éléments d'une population sont traités par paire sur un processeur esclave.

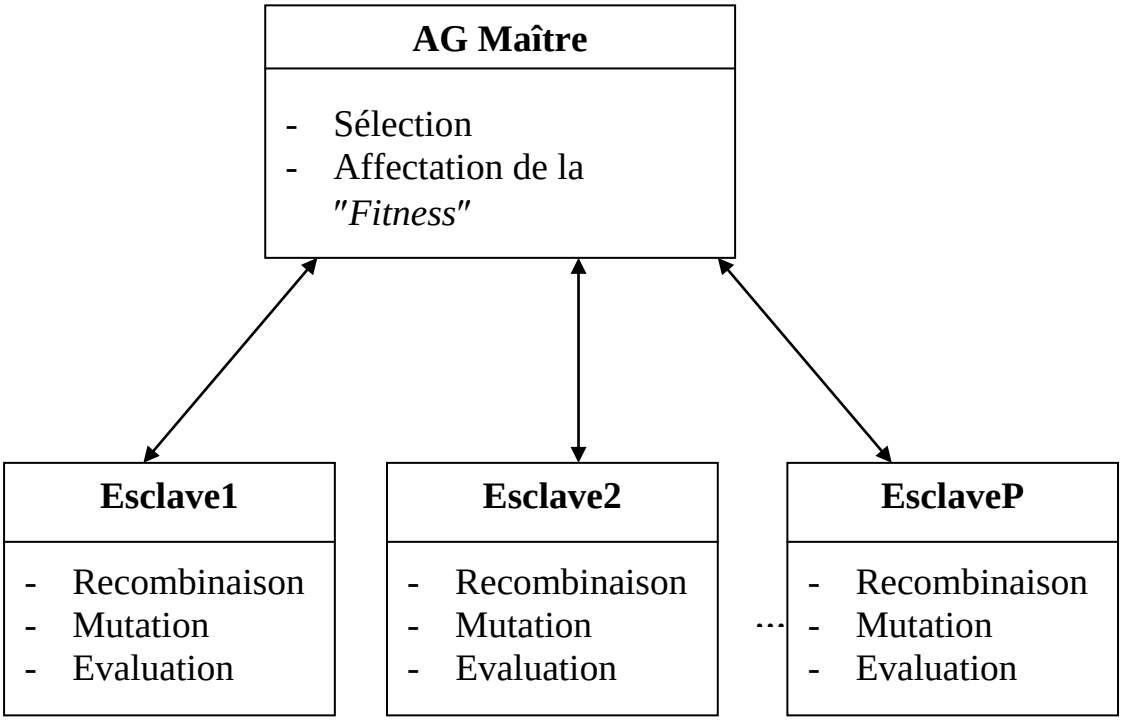


Fig. 6.4a : <u>Méthode de parallélisation Globale</u>

Dans ce cas, un algorithme génétique *"Main"* contrôle toute la population, il effectue les sélections, envoie les chromosomes par paires à chaque esclave. Ces derniers établissent les recombinaisons, les mutations et les évaluations. A la fin, les valeurs de la fonction objectif sont retournées au *"Main"* qui les affectera dans le tableau "Population" aux chromosomes correspondants.

Selon le même schéma, nous pouvons organiser les opérateurs génétiques autrement. Le *"Main"* traite toute la population, individu par individu puis envoie à chaque *"Esclave"* l'évaluation à effectuer, tel que le montre la figure

6.4b. Dans ce cas, un génotype est envoyé par le *"Main"*, un phénotype est renvoyé par l'esclave [17,18].

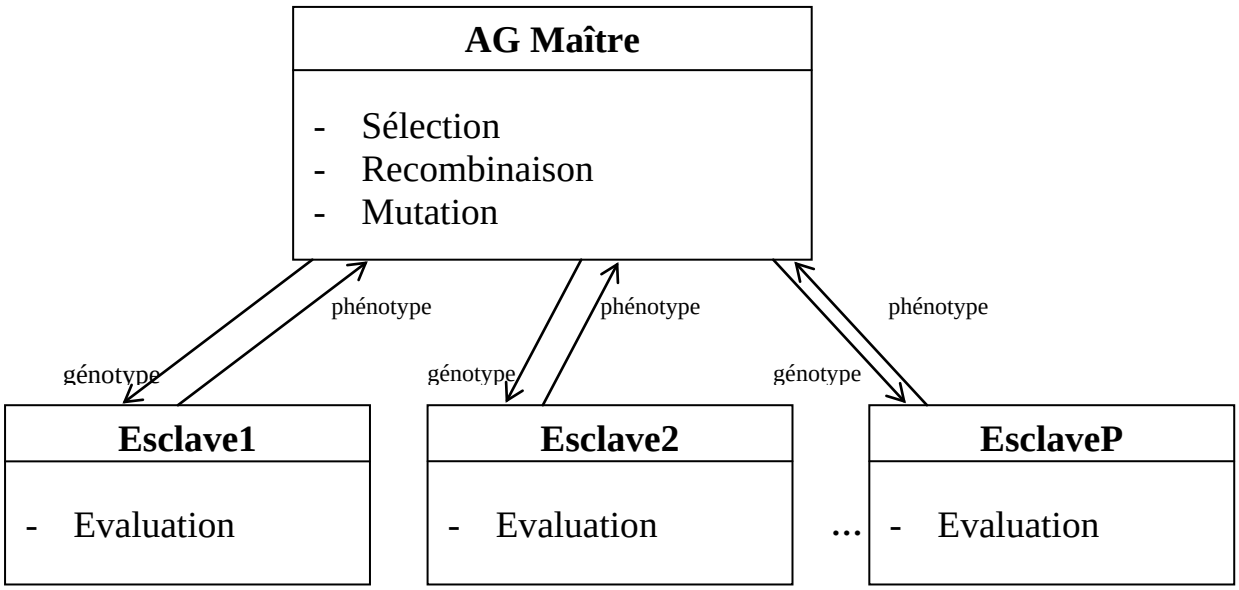
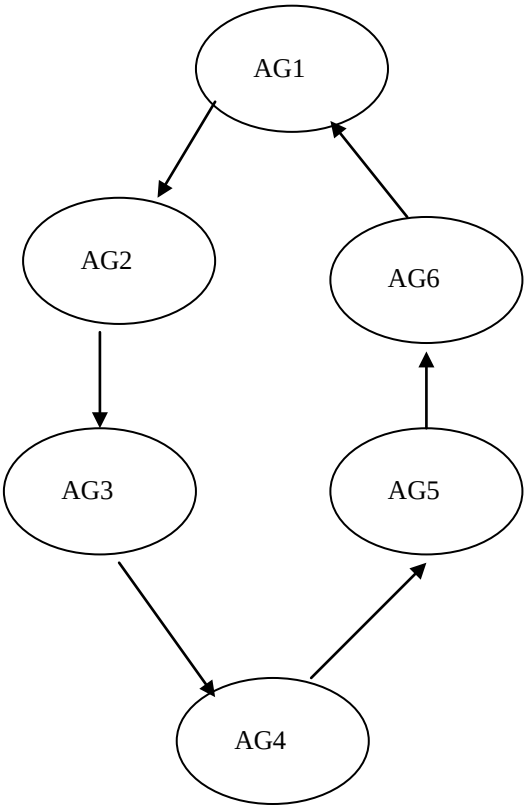
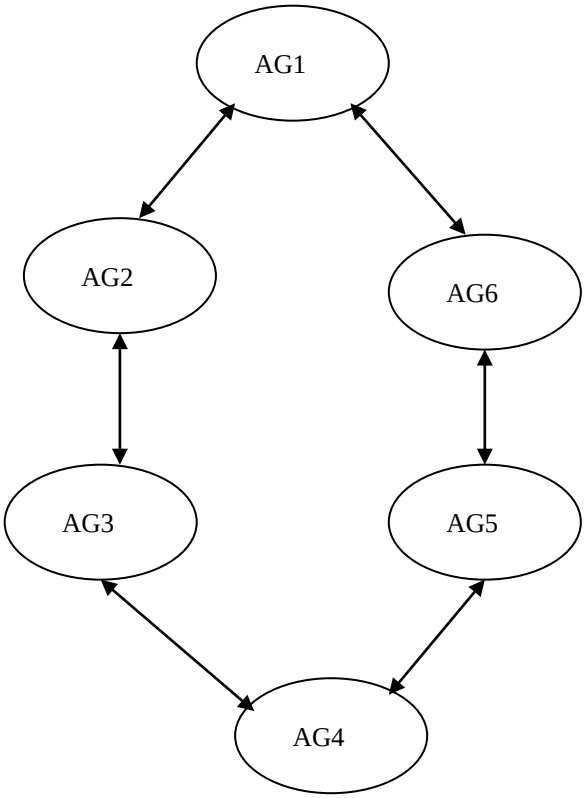


Fig. 6.4b : <u>Méthode Globale, Evaluations effectuées par les Esclaves</u>

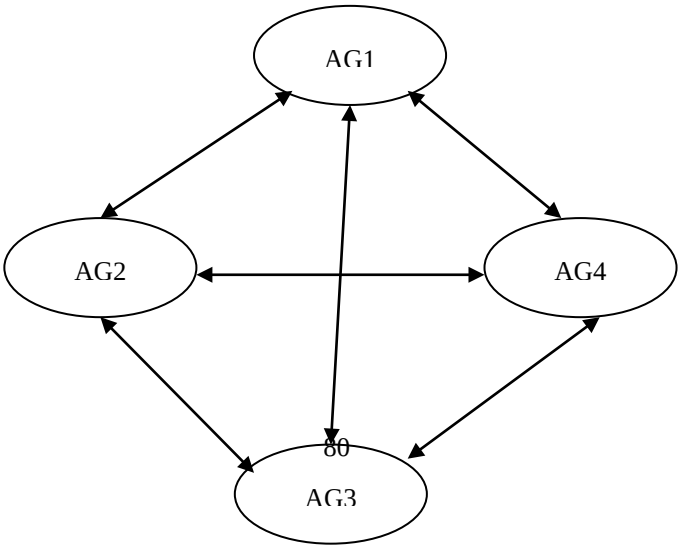
5-2-2 <u>Méthode de migration</u> : Cette méthode divise la population en un certain nombre de sous-populations, chacune d'elles est traitée comme une race à part et sous le contrôle d'un AG conventionnel. Pour encourager la prolifération d'un ″bon″ matériel génétique dans la population totale, des migrations d'individus ont lieu de temps à autre entre sous-populations [11]. Quant aux stratégies de migration, la figure 6.5 montre trois topologies différentes pouvant être utilisées [9], leur choix étant justifiés par le domaine d'application.



Migration en anneau



Migration par voisinage



Dans une telle approche, la description en pseudo-code du processus s'exécutant sur chaque processeur est résumé comme suit [11]:

```
Répéter
  Séquentiel
    Exécution de l'algorithme génétique standard(selection, croisement, mutation)
  Parallèle
    Envoi des meilleurs solutions locales aux processeurs voisins
    Réception des meilleurs solutions de processeurs voisins
    Insertion dans la population
Jusqu'à ce que le critère d'arrêt soit vérifié
```

5-2-3 <u>Diffusion</u> : Comme son nom l'indique, cette méthode diffuse les individus d'une population chacun dans une localisation géographique, comme le montre la figure 6.6. Cela est motivé par les restrictions des réseaux de communication inter-processeurs dans les machines parallèles[9,16].*******

Cette méthode permet aux individus situés dans un voisinage réduit ou très proche de se reproduire entre eux..

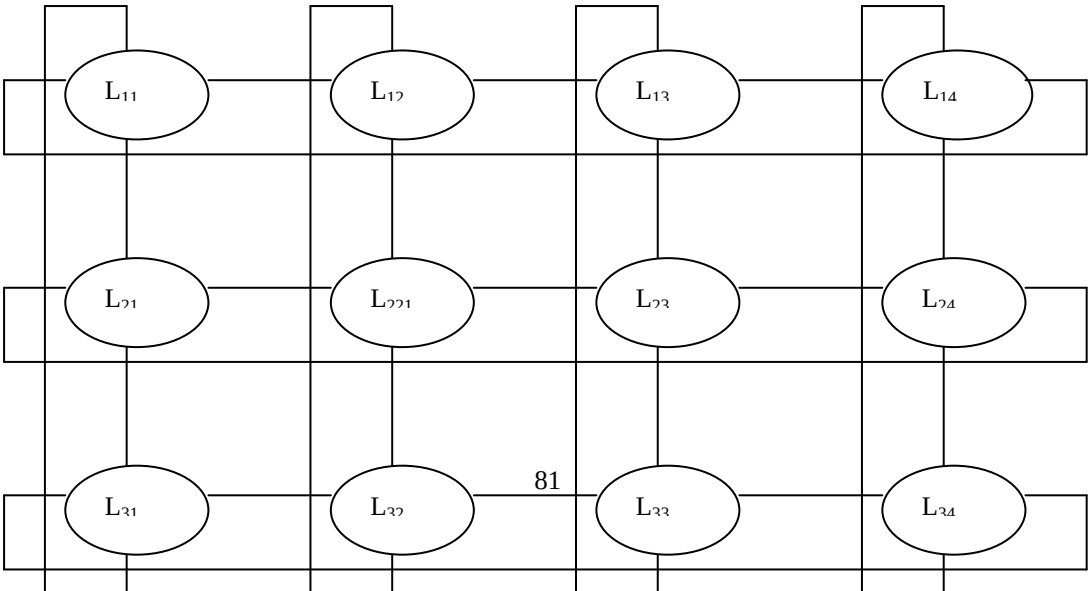


Fig. 6.6 : <u>Méthode de diffusion des individus d'une population génétique</u>

Tous ces schémas de parallélisation d'AG présentent des avantages et des inconvénients [9] : la méthode de migration d'individus s'adapte bien aux implémentations d'algorithmes parallèles sur machines MIMD alors que le modèle de diffusion d'individus conviendrait aux architectures à multiprocesseurs organisés en grilles avec une stratégie de communication inter-processeurs localisés dans un voisinage très proche.

Dans la méthode globale, cela dépend du domaine d'application, du nombre de données à traiter, de la taille des chromosomes, etc. .. Dans le premier cas cité de cette méthode, il serait judicieux de traiter des chromosomes de taille assez réduite afin de ne pas saturer le réseau de communication, sachant que les échanges entre *le Maître* et *les esclaves* s'effectuent par paires de chromosomes.

Dans le 2ème cas, si le calcul de la fonction objectif consomme le plus clair du temps de l'exécution de l'algorithme génétique [16], il serait judicieux de faire évaluer les chromosomes par les processeurs esclaves. Bien que la méthode globale offre un inconvénient majeure : si le processeur Maître tombe en panne, tout le processus s'arrête, elle offre néanmoins un avantage important, à savoir que si le coût de la fonction objectif est très important en temps de calcul par rapport à l'algorithme génétique total, le *"speed-up"* obtenu reste très proche du linéaire [16].

Pour pallier à l'inconvénient précédemment cité, une extension plus robuste de la méthode globale consiste en une implémentation asynchrone concurrente [16, 19, 20]. Utilisant un nombre suffisant de processeurs identiques, chaque AG se déroule sur chacun des processeurs indépendamment sur leurs mémoires locales tout en effectuant des accès à une population stockée en mémoire partagée. Ceci nécessite un accès unique à un individu par un processeur donné à un moment donné. Cette méthode utilise le principe de la mémoire partagée-distribuée.

Malgré la difficulté liée à l'implémentation d'une telle méthode, elle reste néanmoins très tolérante aux pannes de processeurs et de la mémoire partagée.

Les sections suivantes vont présenter un schéma de parallélisation d'un algorithme génétique appliqué au problème de dimensionnement de circuits intégrés.

5-3 Etapes de parallélisation d'un AG appliqué au problème de dimensionnement de circuits intégrés :

Pour évaluer les individus (ou chromosomes) d'une population, dans notre cas les performances d'un circuit intégré en surface et délai de propagation, il est spécifié une fonction dite ″objectif″ ou *″fitness″*, qui évalue un par un les individus d'une population. Dans le cas du *″Gate Sizing″*, nous rappelons [voir chap.IV] que la fonction objectif est donnée par :

$$F = \mathrm{Min}\left[\sum_{i=1}^{n} S_i + \lambda * (D - T) \right]$$

S_i est la surface occupée par chaque composant (porte logique), D est le délai de propagation total du circuit et T est le délai maximum imposé par le concepteur de ce circuit dans le cahier des charges. λ est un facteur de pénalité qui favorise le délai total du circuit par rapport à la surface de ce dernier ou inversement [voir Chap. IV].

Nous remarquons dans ce qui précède, que les étapes de l'algorithme génétique sont relativement simples à effectuer : dans la sélection, il s'agit d'apparier les chromosomes par paires, le croisement ou *"crossover"* consistant à échanger des parties de deux chromosomes entre eux et la mutation qui consiste à inverser un bit (0 en 1 et inversement.) éventuellement.
Par conséquent, le temps passé à exécuter ces étapes est peu significatif car très peu ou aucun calcul n'est effectué en réalité.

Par contre, durant l'évaluation des individus de la population, il faudra pour chacun d'entre eux, calculer la valeur de la fonction objectif F. Pour cela, il faut évaluer la surface totale du circuit représenté par un chromosome donné ainsi que son délai de propagation total [voir chap.IV]. Ceci nécessite des accés en bibliothèque des cellules pour le calcul de la surface totale, ainsi que le déroulement de l'algorithme DFS [voir chap.I] pour la détermination du délai total du circuit. Or actuellement, la taille de ce dernier (en nombre de portes) dépasse le million. La valeur de la fonction objectif est obtenue enfin en effectuant la somme entre la surface et la différence entre le délai du circuit et le délai max imposé, que multiplie le facteur de pénalité. Ces calculs doivent être effectué pour chaque chromosome de la génération courante, celle-ci en compte plus d'une centaine. Le nombre d'itération de l'algorithme génétique est choisi de l'ordre d'une centaine. Il est clair alors, que le temps d'exécution de l'algorithme génétique est consommé essentiellement en évaluations répétitives des individus des différentes générations, dans le but de choisir le meilleur à la fin de ce processus.
Dans le cas de Thomas Back [7] et P. Lyaet [8], ils ont pu montré que 95% du coût de l'algorithme génétique est consommé durant les évaluations de tous les individus des différentes générations.
Pour notre part, le temps consommé en évaluations répétitives dépasse ce chiffre. *****

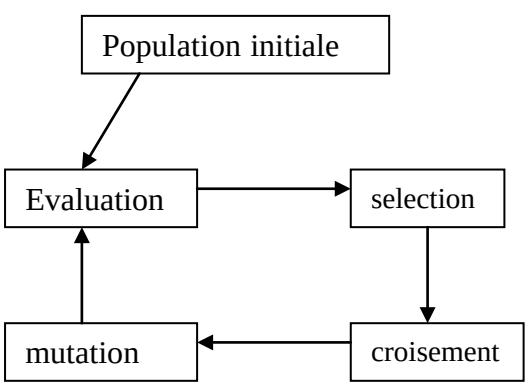
Une solution proposée pour pallier à cet inconvénient est de paralléliser l'évaluation des chromosomes : nous avons donc opté pour calculer en parallèle la valeur de la fonction objectif en chaque individu. Pour cela, nous avons opté pour un schéma parallèle de l'AG selon la méthode globale avec evaluation parallèle de chaque circuit. Cette approche est dite semi-asynchrone., dans la mesure où :

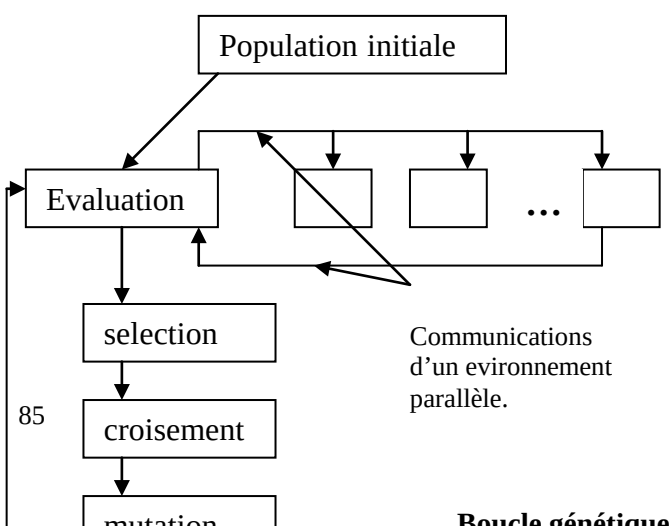
- L'hétérogeinité des processeurs (maître et esclaves) en puissance de traitement est prise en compte, en minimisant leurs temps d'attente.
- L'implémentation sous environnement parallèle PVM serait relativement simple.

La figure 6.7 illustre une boucle génétique séquentielle et une boucle génétique parallèle.

Le schéma d'implémentation est relativement simple. Nous pouvons le résumer comme suit :

Sur une machine parallèle virtuelle constituée d'un cluster de P machines connectées via un réseau local, l'une de ces machines est déclarée Maître ou *"Main"*, les (P-1) autres sont déclarées Esclaves ou *"Slaves"*. Le logiciel d'optimisation d'un circuit intégré [voir chap. IV] commence à dérouler ses étapes sur la machine Maître, à savoir la construction dynamique des structures de données qui représentent un circuit en mémoire, puis lance les étapes de l'algorithme génétique.





Boucle génétique séquentielle

Boucle génétique

La génération d'une population initiale est effectuée alors, puis il y a envoi des messages vers les autres machines esclaves qui doivent commencer l'évaluation des chromosomes.

Les structures de données sont également construites au fur et à mesure sur les processeurs Esclaves pour éviter leur envoi à partir du Maître à travers le réseau et ceci pour les raisons suivantes :

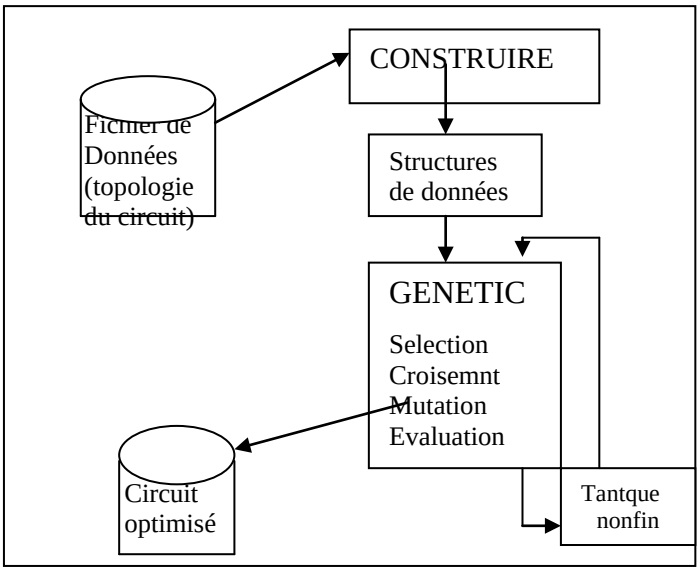
- Eviter de saturer le réseau par un trafic intense des données qui peuvent être utilisées localement sur les autres machines.
- Les données construites dynamiquement contiennent des adresses physiques locales à la mémoire du processeur Main et n'ont donc aucune réalité physique sur les autres machines du réseau.

Dans la mémoire des processeurs Esclaves doit figurer également le code exécutable de la procédure d'évaluation des chromosomes.

Lorsque le nombre de processeurs Esclaves (P-1) est inférieur au nombre N d'individus d'une génération et dans le cas où le réseau des machines est hétérogène, nous proposons une taille N d'une population multiple de (P-1) pour pouvoir effectuer des évaluations successives par paquets de (P-1) chromosomes qui sont envoyés vers les processeurs Esclaves pour leur évaluation, puis récupérés par le processeur Main. En réalité, si la machine parallèle virtuelle est constituée d'un réseau de machines hétérogènes, tous les processeurs

disponibles n'ont pas la même vitesse d'exécution. Il faudra alors que l'algorithme parallèle prenne en charge cette éventualité afin de minimiser les temps morts des processeurs Esclaves (″*Idle time*″). D'où, la machine Maître procède par envoie/réception de (chromosome)/(valeur de la fonction objectif) vers/de la machine Esclave ″libre″ à un moment donné, sans distinction.

La figure 6.8 **** illustre un schéma séquentiel d'exécution de notre logiciel d'optimisation des circuits intégrés VLSI [voir chap. IV, Figure 4.1] en a) et un schéma d'exécution parallèle pour le même logiciel en b) illustrant les échanges entre le ″Main″ et les différents ″Slaves″ de l'architecture parallèle à mémoire distribuée.



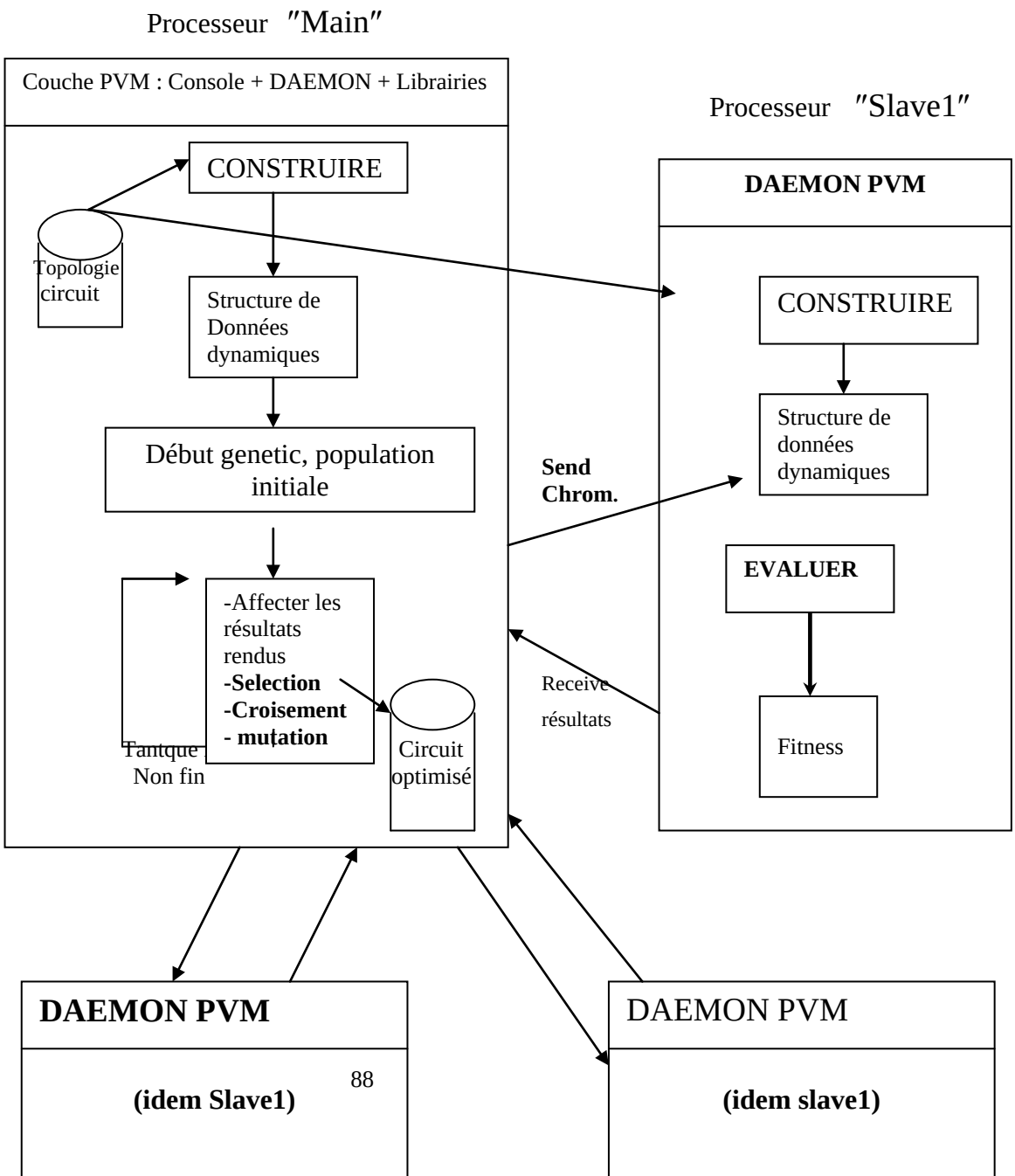
a) Schéma d'exécution séquentielle

En pratique, la machine ″Main″ doit contenir tout le logiciel d'optimisation : lecture des données, construction des structures de données dynamiques représentant le circuit à analyser en mémoire, codage des paramètres, étapes de

l'algorithme génétique … Dés que la population initiale est créée par le processeur ″Main″, celui-ci envoie vers chaque processeur ″Slave″ :

- le fichier contenant la topologie du circuit en entrée à optimiser sous forme de tableaux de données.

Un vecteur chromosome contenant un génotype et un numéro de génotype. Lorsque ce chromosome a été évalué par l'esclave, celui-ci renvoit au Maître un vecteur contenant le numéro du chromosome avec la valeur de la fonction objectif calculée en ce point. Ainsi, nous récupérons la valeur de la fonction objectif avec le numéro du chromosome concerné dans le même paquet, afin de ne pas ″confondre″ les réceptions de résultats qui peuvent arriver vers le ″Main″ dans n'importe quel ordre, avec les chromosomes qui leur correspondent.



Sur les différentes machines Esclaves :

- les structures de données dynamiques sont construites après réception du tableau contenant le fichier du circuit à analyser,
- le code EVALUATION est présent pour pouvoir être executé dés réception d'un chromosome.

5-4 Algorithme d'évaluation parallèle des individus d'une population [7,7bis]:

Après avoir vu comment se font les échanges entre les différents processeurs nous allons énoncer un algorithme génétique parallèle généralisé en pseudo-code, qui résume toutes ces étapes précédemment décrites.

DEBUT /*On dispose de (P-1) machines Esclaves, N=taille d'une population*/
Initialize Pop=(X_1,X_2,…, X_N) ;
FOR i=1 to (P - 1) DO
SEND (N_X,X_i) ⟶ Sl_i ; /* N_X est un numéro affecté au chromosome X */

FOR i=P à N DO
RECEIVE (N_X,F(X))⟵ Sl_γ ;
SEND (N_X,X_i) ⟶ Sl_i ;
ENDDO ;

WHILE ($i<i_{max}$) DO
RECEIVE (N_X,F(X))⟵ Sl_γ ;
Select ;
Crossover : X'=r(X) ; /*recombiner*/
Mutation : X″=m(X') ;
SEND X″ ⟶ Sl_γ ; /*γ est une valeur arbitraire entre 1 et (P-1)*/
i=i+1 ; /* Sl_γ désigne un processeur ″esclave″ libre*/
ENDDO

FOR i=1 to (P- 1) DO
RECEIVE (N_X,F(X))⟵ Sl_γ ;
Select ;
ENDDO
Return ″The Best″ ;

FIN.

Après création de la population initiale, la 1ère boucle envoie aux (P-1) machines Esclaves des chromosomes à évaluer. La 2ème boucle commence à récupérer les résultats (X,F(X)) à partir du 1er Esclave Sl_γ qui a fini ses calculs, γ étant une valeur arbitraire comprise entre 1 et (P-1). Ce processeur redevenant libre, on lui renvoie un nouveau chromosome à évaluer. Après avoir terminé l'évaluation de la première génération, le processeur ″Main″ continue à recevoir les paquets ″résultats″ et prend en charge les autres opérateurs génétiques, jusqu'à un nombre d'itération i_{max}. En dernier, nous récupérons les derniers résultats à partir des machines Esclaves, de la dernière itération pour pouvoir choisir le meilleur chromosome ″the best″ .

L'algorithme qui s 'exécute sur les processeurs Esclaves se résume comme suit :

DEBUT

RECEIVE(NX,X) ⟵ P_{Main} ;

Evaluer(X,F(X)) ;

SEND(NX,F(X)) ⟶ P_{Main} ;

FIN

5-4 <u>Ecriture de l'algorithme génétique parallèle sous environnement PVM</u> :

Sous environnement PVM décrit en annexe, cet algorithme pourrait s'écrire en pseudo-code, sous la forme suivante :

```
#include <pvm3.h>
Main()
{  int mytid, info, tids ;
   int nproc ;  /* nbre de processeurs esclaves */
  int  nb_gate ; /* taille d'un chromosome */
  double VAL;   /* valeur réelle : Fonction objectif*/

  mytid=pvm_mytid() ;

  Initialize Pop=(X1,X2,…, XN) ;
  nproc=(P - 1) ;
  Info=  pvm_spawn(″Evaluer ″, ″Données″, 0, ″″, nproc, tids) ;
```

```
  For  i=1 TO nproc DO
     Pvm_initsend(PvmDataRaw) ;
     Pvm_pkint(NX,1,0) ;
     Pvm_pkint(X,nb_gate,2) ;
      Pvm_send(-1,-1) ;
  ENDDO ;

 FOR i=P à N  DO
      Pvm_recv (-1, -1) ;
      Pvm_upkint(NX, 1, -1) ;
      Pvm_upkdouble(VAL,1,-1) ;

      Pvm_initsend(pvmDataRaw) ;
     Pvm_pkint(NX,1,0) ;
      Pvm_pkint(X,nb_gate,-1) ;
      Pvm_send(-1,-1) ;
 ENDDO ;
WHILE (i<i_max) DO
       Pvm_recv(-1,-1) ;
       Pvm_upkint(NX, 1, -1) ;
       Pvm_upkdouble(VAL, 1, -1) ;
       Select ;
       Crossover ;   /*recombiner*/
       Mutation ;
       /* recommnencer à évaluer par la suite*/
      Pvm_initsend(pvmDataRaw) ;
      Pvm_pkint(NX, 1,-1) ;
      Pvm_pkint(X,nb_gate,-1) ;
       Pvm_send(-1,-1) ;
        i=i+1 ;
ENDDO
FOR i=1 to (P - 1)   DO
      Pvm_recv(-1,-1) ;
      Pvm_upkint(NX, 1, -1) ;
      Pvm_upkdouble(VAL, 1, -1) ;
        Select ;
 ENDDO ;
Pvm_exit() ;

Return ″The Best″ ;

}
```

Cette procédure est la procédure du processeur Maître. Au départ, il y a création de nproc tâches (nproc étant le nombre de processeurs esclaves) par PVM_SPAWN qui spécifie le nom du programme exécutable à exécuter, le deuxième paramètre spécifie les données, le troisième positionné à 0 signifie n'importe quel ″host″ libre du réseau et prêt à exécuter, le quatrième positionné à chaine vide signifie que l'on ne spécifie le nom d'aucun ″host″ (le troisième et le quatrième fonctionnent ensemble), nproc étant le nombre de processeurs esclaves, et enfin le dernier contiendra le nom du tableau des identificateurs des taches ou ″*Task identifiers*″ car il est affecté à chaque tâche s'exécutant sur un processeur.Esclave un identificateur généralement constitué du numéro du processeur et du numéro de la tâche elle même.

Par la suite, il faut initialiser le buffer qui contiendra les données à envoyer, par la routine Pvm_initsend(pvmdataraw). Son paramètre signifie qu'il n'y a pas d'encodage spécial à effectuer avant d'envoyer les données. Dans ce buffer, seront empaquetées les données selon leur type : pvm_pkint( ) pour empaqueter une donnée entière, pvm_pkdouble pour les données réelles en double précision, pvm_pkfloat pour des données en représentation flottante, etc…

Une fois que le bufffer, dont la taille est dynamique, est rempli de données à envoyer, la primitive pvm_send se charge d'acheminer ces données vers un processeur esclave libre. Pour cela, les paramètres (-1,-1) de Pvm_send signifient vers n'importe quel processeur libre.

Dans la même suite d'idées, la réception après traitement des données se fait dans le même ordre, il faut déballer les données empaquetées dans le buffer de PVM. Pour cela, les primitives Pvm_upk* se chargent de déballer en respectant le type des données reçues. Exemple : Pvm_upkint(NX,1,-1) signifie : déballer une valeur entière et la stocker dans NX, Pvm_upkdouble(VAL,1,-1) signifie déballer une valeur réelle en double précision et les ranger dans la variable VAL.

La routine exécutée par les processeurs esclaves se résumerait sous PVM comme suit :

```
#include <pvm3.h>
Main()
{ int tid, m_tid, nb_gate ;

   m_tid=pvm_parent() ;
   tid=pvm_mytid () ;
    pvm_recv(-1, -1) ;
    pvm_upkint(NX,1,-1) ;
    pvm_upkint(X, nb_gate,-1) ;
    evaluation(X,VAL) ;  /* la procedure evaluation évalue X un chromosome et rend un valeur
                            réelle : la fitness dans la variable VAL*/
   pvm_initsend(pvmdataraw) ;
   pvm_pkint(NX,1,-1) ;
   pvm_pkdouble(VAL,1,-1) ;
   pvm_send(m_tid,1) ;
pvm_exit() ;
}
```

Cette procédure trouve l'identificateur du parent dans m_tid par la routine pvm_parent, c'est le numéro de la tâche qui s'exécute sur le processeur Main et qui est nécessaire à connaître pour renvoyer les résultats. Elle déballe les données lui provenant du maitre. Elle procède à l'évaluation d'un chromosome X par évaluer() puis ré-initialise le buffer de PVM pour pouvoir y empaqueter les données à renvoyer au processeur Maitre par pvm_pkint et pvm_pkdouble. Elle renvoie le numéro du chromosome ainsi que la valeur de F la fitness.

6- Conclusion :

Comme précédemment spécifié, cet algorithme prend en compte l'hétérogeinité des machines constituant le réseau en minimisant le temps d'attente des machines ″Slaves″. En effet, il procède par envoi/réception de chromosomes/résultats de n'importe quelle machine ″libre″ du réseau.

De plus et comme on peut le remarquer dans notre cas, notre modèle Maître/Esclave est aussi un modèle SPMD *″Single Processus/Multiple Data″*. En effet, le même code est exécuté par les processeurs ″Slaves″ sur un flot de données différentes mais équivalentes au point de vue taille: ces machines évaluent de la même manière un ensemble d'individus constituant plusieurs générations. Nous pouvons conclure donc que les tâches sont équitablement réparties entre les différentes machines du réseau.

CONCLUSION GENERALE

Etant données la grande diversité ainsi que la grande taille des circuits intégrés VLSI, le problème de dimensionnement de portes est un problème NP-Hard : il n'existe pas d'algorithmes exacts pour le résoudre.

Le travail décrit dans cette thèse consiste à étudier ce problème, à le modéliser et proposer une solution par une méta-heuristique connue pour sa robustesse : l'algorithme génétique. Ce dernier s'inspire des mécanismes de la sélection naturelle et de la génétique. Son application à un problème complexe comme le *"gate sizing"* a prouvé sa puissance et sa grande souplesse. En effet, le nombre très important de paramètres des circuits ainsi que la difficulté de modéliser avec précision toutes leurs caractéristiques ont trouvé une prise en charge relativement simple par cet algorithme. Tel que déjà spécifié dans cette thèse, la littérature a montré le grand intérêt des chercheurs dans le monde porté à ce problème et qui ont essayé de le résoudre durant cette dernière décennie, par application d'heuristiques développées proprement dans ce but. Citons notamment les travaux de P. Chan, Li, Moon, et même AR. Baba-Ali. Actuellement, une approche basée sur l'estimation des délais par des méthodes statistiques a été développée.

Dans cette thèse, une approche génétique a été proposée. Elle a abouti à des résultats intéressants, obtenus au bout d'un temps raisonnable. Le compromis Surface/Délai de propagation apparaît clairement dans les résultats obtenus : plus le délai est court et plus le nombre de portes BiCMOS rajoutées est grand et

donc la surface total du circuit est augmentée, et inversement. Toutefois, il est à signaler que, dans une première approche, nous avons favorisé le temps d'exécution par rapport à la qualité des résultats, car le compromis temps d'exécution/qualité des résultats a toujours été un problème posé. Nous nous sommes surtout intéressés à la rapidité, vu le nombre de paramètres du problème. Ce travail a fait l'objet d'une communication [21] à la 6ème Conférence Internationale sur les Mathématiques Appliquées et les Sciences de l'Ingénieur en octobre2000.

De plus, un algorithme génétique demeurera toujours une méta-heuristique très intéressante du fait de sa nature parallélisable. Cela est dù au fait qu'il manipule une population d'individus, pouvant être traités simultanément.

Dans notre cas, la grande masse d'informations concernant un individu entraîne un temps très appréciable en évaluations de toutes les générations, par rapport au temps total d'exécution de l'algorithme génétique. Dans cette optique, il a été développée dans cette thèse une approche génétique parallèle, dans le but de réduire le temps des évaluations de tous les chromosomes de chaque génération car tel que décrit dans la littérature [7paral], au moins 95% du coût en temps de l'algorithme génétique est consommé par les évaluations. Il a été étudié dans ce cadre, les schémas possibles de parallélisation d'un AG. Nous avons opté pour l'évaluation des chromosomes en parallèle sachant que notre fonction objectif consomme énormément de temps en recherchant les chemins critiques de chaque circuit, de chaque génération par exécution d'un algorithme d'exploration dans un graphe, sans compter le calcul de sa surface totale. Cette approche serait facilement implémentée sous environnement de programmation parallèle PVM ou MPI, dés que la machine parallèle ou virtuellement parallèle sera disponible au laboratoire d'accueil. Ce travail a fait l'objet d'une communication [22] à la Conférence International sur la micro-electronique ICM'2000 en novembre 2000.

Le travail qui a été développé dans cette thèse s'inscrit dans le cadre de l'optimisation temporelle des circuits intégrés VLSI. Il serait intéressant dans une seconde phase de se pencher sur ce problème en essayant d'améliorer la qualité des résultats obtenus, en appliquant des méthodes de recherche à multi-critères ou bien une recherche guidée. D'autre part, il serait intéressant également de prendre en compte les interconnexions entre les composants d'un circuit et d'optimiser (réduire/élargir) leurs dimensions de manière à gagner encore plus en délai, toutefois en perdant un peu plus en surface. Ainsi l'approche proposée serait bichromosomale. De plus, elle regrouperait une optimisation plus complète, étant donné qu'elle prendrait en charge les interconnexions et les portes simultanément (Gate and Wire Sizing). Cependant, nous espérons avoir apporté une humble contribution à la résolution du *"Gate Sizing"* par une approche génétique et qui a été menée pour la première fois dans le monde.

Annexe A

Rappels de Définitions de concepts de la VLSI

Porte logique : est un circuit de base, composé de plusieurs transistors et constitue le fondement matériel des ordinateurs actuels. Plusieurs portes logiques, grâce à des multiples associations, sont capables de réaliser diverses fonctions logiques.

Port : Chaque porte possède des points de connexions appelés ″ports″ et qui indiquent où les fils de connexion sont attachés.

Entrées primaires : c'est l'ensemble des ports identifiés comme entrées et exportés vers le plus haut niveau de la hiérarchie.

Sorties primaires : c'est l'ensemble des ports identifiés comme sorties et exportés vers le plus haut niveau de la hiérarchie.

Bloc logique combinatoire : c'est une portion d'un circuit qui réalise une fonction logique par exemple, une addition entre un certain nombre de bits. Un bloc combinatoire contient uniquement des portes logiques et ne possède pas de mémoire.

Circuit combinatoire : c'est un circuit [2] logique dont les n variables d'entrées x_i et les variables de sortie z_i ne peuvent prendre que deux valeurs distinctes 0 ou 1.

A un état d'entrée x donné du circuit correspond un état de sortie z. Cette correspondance est définie dans la table de vérité du circuit.

Circuit séquentiel synchrone : c'est un circuit combinatoire dont une ou plusieurs de ses sorties sont connectées à certaines de ses entrées.

ANNEXE B

PVM : Un environnement de programmation parallèle.

1-Introduction

Beaucoup de programmes d'application requièrent des temps de calcul fastidieux. La nécessité d'utiliser des machines puissantes et rapides se fait sentir de plus en plus, dans différents domaines, tels que la météo, l'aéronautique, le traitement des images, etc… Pour cela, le recours aux machines massivement parallèles s'avère de jour en jour indispensable. Seulement, ces machines à multiprocesseurs sont excessivement coûteuses. De plus, elles ne sont pas simples à utiliser et à gérer.

Pour ces toutes raisons, l'idée de créer une machine parallèle virtuelle a vu le jour à la fin des années 80 : il s'agissait de faire communiquer plusieurs processeurs situés dans différentes machines via un réseau local de type LAN. Un protocole de communication a été développé dans ce but, appelé PVM (Parallel Virtual Machine).

2-Description et Fondements généraux de PVM :

PVM est une couche logicielle [29] qui permet de faire communiquer plusieurs machines hétérogènes de sorte que virtuellement, ces machines apparaissent comme une seule ressource parallèle. PVM est constitué d'une part d'un ensemble de librairies utilisateurs qui renferment des routines d'initialisation de processus sur les machines du réseau, de communication entre ces processus, de modification portant sur la configuration de la machine parallèle virtuelle, … et d'autre part d'un DEMON *"Daemon"* permettant les échanges entre les machines. Le démon doit être installé sur chaque machine du réseau. C'est un protocole de communication entre les différentes machines hétérogènes du réseau. Ces dernières peuvent être séquentielles, vectorielles ou parallèles.

Le développement de PVM commença en 1989 au Laboratoire National Oak Ridge (ORNL) aux USA mais actuellement c'est un projet encore en développement et regroupant les efforts de plusieurs équipes de recherche telles que Al Geist à l'ORNL, Vaidy Sunderam à l'université de Emory, Robert Manchek à l'université du Tenessee, Adam Beguelin à l'université de Carnegie Mellon, le Pittsburgh Supercomputer Center,…

C'est une recherche qui vise fondamentalement le développement de la science et qui a été lancée par le Département Américain de l'Energie, la Fondation Nationale des Sciences et de l'état du Tenessee.

Sous PVM, une collection de machines hétérogènes : mono-processeurs, vectorielles, parallèles apparaît comme une seule machine à mémoire distribuée, désignée par ″machine parallèle virtuelle″. Le terme ″Host″ désigne chaque composante du réseau, ie : chaque machine reliée au réseau.

PVM offre la possibilité de lancer automatiquement des tâches sur la machine virtuelle, leur permet de communiquer entre elles et de se synchroniser. Une tâche est définie comme une unité de calcul sous PVM d'une manière analogue

à un processus sous UNIX. En communiquant par envoi-réception de messages "*messagse passing*", les différentes tâches d'une même application peuvent coopérer à la résolution d'un problème en parallèle. La taille des messages échangés n'est limitée que par la mémoire disponible. PVM permet même des messages contenant plus d'un type de données à échanger avec les machines ayant différentes représentations de données.

Par conséquent, PVM accepte toute sorte d'hétérogeinité à différents niveaux :

- applications
- machines
- réseaux.

comme le montre la figure B.1.

En d'autre termes, il permet aux tâches d'une application d'exploiter au mieux l'architecture sur laquelle elles évoluent. Il prend en charge toute sorte de conversions de données qui peuvent être requises si deux ″hosts″ utilisent différentes représentations des entiers ou virgule flottante.

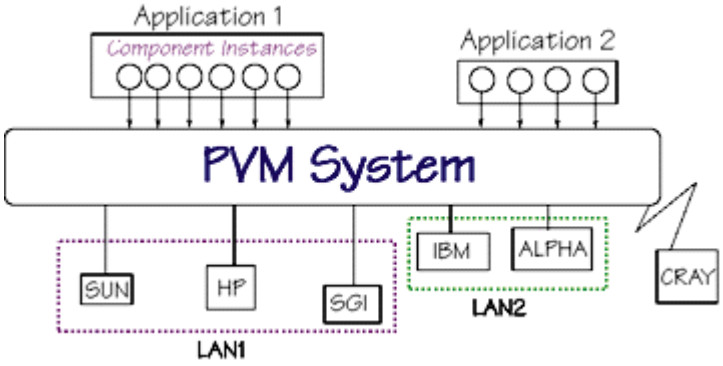
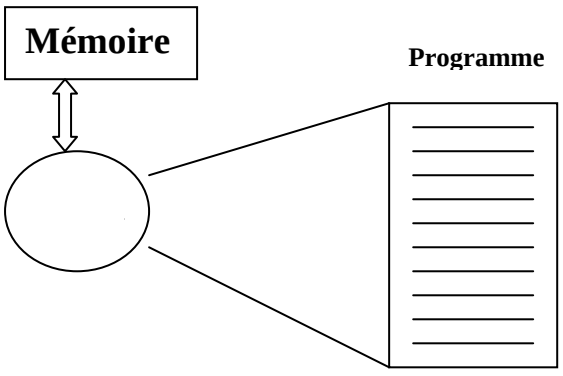


Fig. B.1 : Schéma illustrant une machine parallèle virtuelle

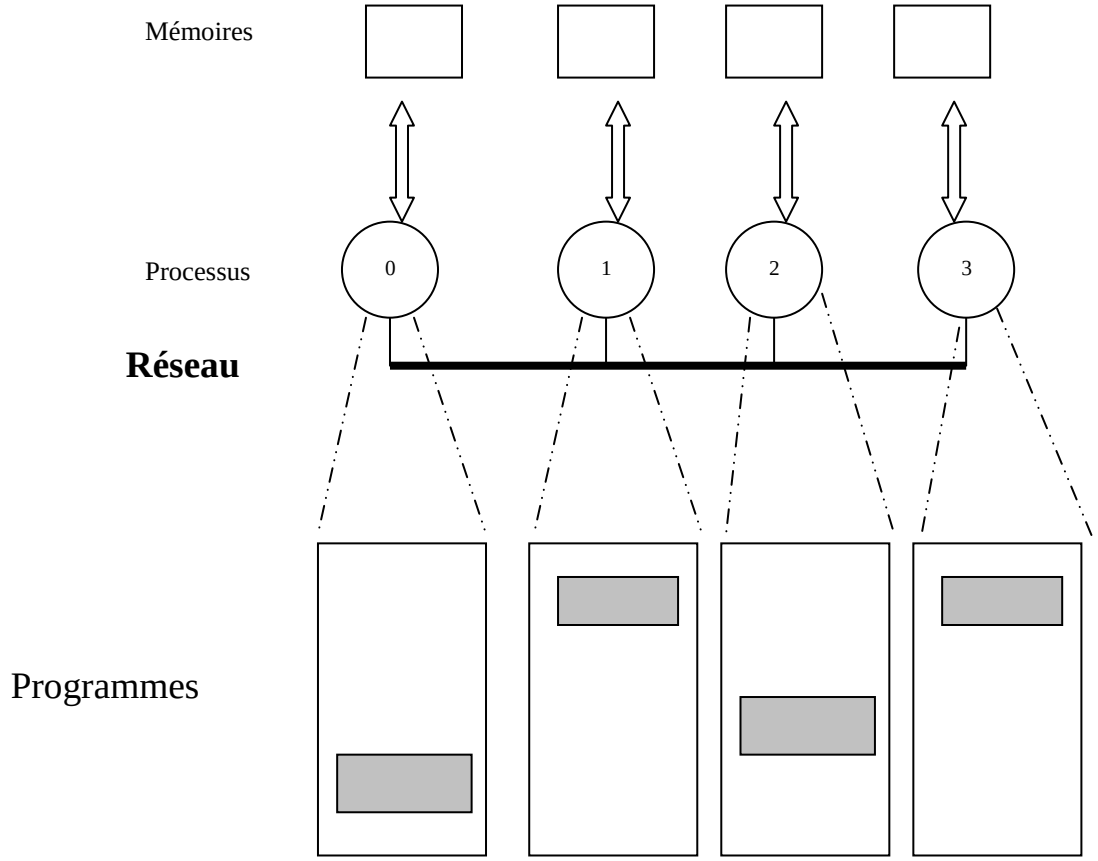
Les tâches d'un programme d'application s'exécutent sur un ensemble de machines sélectionnées par l'utilisateur :

- A tout moment, nous pouvons ajouter/supprimer une machine du réseau : ce qui représente un gain considérable pour la tolérance aux pannes.

- Les programmes d’application peuvent soit ″voir″ l’environnement hardware comme une collection attribuée d’éléments de calculs virtuels ou bien choisir d’exploiter au mieux les capacités de machines spécifiques dans le groupe de ″hosts″ en attribuant certaines tâches aux machines les plus appropriées.
- Il n’y a pas d’affectation ou "*mapping*" imposé par PVM entre un processeur donné et un processus donné. Au contraire, plusieurs tâches peuvent s’exécuter sur un même processeur.
- Plusieurs utilisateurs peuvent configurer des machines virtuelles enchevauchées ("Overlapping virtual machines"). Ainsi, chaque utilisateur peut exécuter des applications PVM simultanément avec d’autres.
- C’est un modèle explicite de passage-de-messages. En effet, une collection de tâches, chacune effectuant une partie d’un traitement d’une application, coopèrent par des envois-réception de messages **explicitement**, comme illustré par la figureB.2.



A) Modèle Séquentiel



b) Méthode par passage-de-message

Après avoir passé en revue une description générale du système PVM, les sections suivantes décriront plus en détail la programmation sur un tel système.

Un programme est constitué d'une série de séquences indépendantes qui peuvent s'exécuter en parallèle. Dans la terminologie PVM, ces séquences sont dénommées tâches. Chaque tâche a deux fonctions :

- réaliser les traitements prévus par la séquence de programme sur la partie des données qui lui sont affectées à ce moment là,
- assurer les communications avec le programme principal (lui-même une tâche) et éventuellement les autres tâches (échange des données initiales et des données traitées, informations de synchronisation ...).

3- Les composants de PVM :

Le système PVM est composé de trois parties :

- Une partie appelée PVMD : est un démon qui réside dans toutes les machines qui constituent la machine virtuelle.
- Une console : elle doit être initialisée sur la machine "*HOTE*" qui permet de lancer l'exécution de l'application sur les différents "*HOSTS*" du réseau.
- Des variables d'environnement.
- Des bibliothèques de routines libpvm3.a, libfpvm3.a, libgpvm3.a contenant les modules des sous programmes PVM.

3-1 Le Démon PVMD3 (DAEMON) :

Un exemple de démon est le programme "*MAIL*" qui s'exécute en arrière plan et qui prend en charge toutes les émissions et les réceptions du courrier électronique d'une machine.

Le démon PVMD3 sert à coordonner les différentes machines composant la machine virtuelle. C'est le démon de la machine "*HOTE*" qui active les démons sur les machines distantes :

Pvmd3 [-options] [hostfile]

Cette commande lance le démon tout en initialisant la configuration de la machine virtuelle. Le fichier "hostfile" contient les noms des machines distantes sur lesquelles se dérouleront les différentes tâches de l'application et sur lesquelles les démons seront activés automatiquement par cette commande. Chaque utilisateur possède son propre fichier "hostfile" dans lequel il configure sa machine virtuelle personnelle.

Les options permettent de modifier, rajouter, etc… certaines spécificités concernant les machines qui composent la machine virtuelle parallèle, prises

une à une, dans le fichier "hostfile". Quelques options utiles sont résumées dans le tableau suivant :

Option	**Signification**
So=pw	PVM demandera un mot de passe sur ce ″host″
Dx=location of pvmd	Dans le cas où PVMd est installé dans un répertoire autre que celui par défaut, spécifier le chemin.
Ep=path-executables	Chemin où se trouvent les exécutables de PVM. Par défaut: $(HOME)/pvm3/bin/$(PVM_ARCH)
Wd=path-user	Chemin du répertoire de travail.
Bx=location-of-debugger	Permet de spécifier la location d'un debogueur d'application. par défaut : $(PVM_ROOT)/lib/debugger

3-2 La console :

La console permet de contrôler la machine virtuelle. Elle se lance en tapant pvm (situé dans le répertoire relatif pvm3/lib). Elle est analogue à celle d'un système multi-tâches : elle admet des commandes à partir de l'entrée standard
Elle permet également de démarrer le démon pvmd local s'il n'est pas présent.
Le prompt **`>pvm`** indique que la console est active.

Quelques commandes importantes de la console

`help`	aide en ligne sur les commandes de la console
`add neptune`	Lance le daemon pvmd sur la machine neptune et donc la rajoute dans la machine virtuelle
Delete saturne	Enlève la machine saturne de la machine virtuelle
conf	Affiche la liste des machines appartenant à la machine virtuelle
pstat	Affiche l'état des tâches qui s'exécutent sur la machine virtuelle.

Mstat	Affiche l'état des hosts
ps	liste les tâches actives de la machine virtuelle
Kill taskid	tue une tâche
reset	tue toutes les tâches
spawn task	Démarre une tâche sur une machine distante.
setenv	Affiche ou positionne les variables d'environnement.

Donc pour lancer une application à partir de la console, les étapes sont les suivantes:

- lancement de pvm
- description de la machine virtuelle (**`add`**)
- lancement de l'application (**`spawn`**) en interactif.

3-3 Les variables d'environnement :

Au départ, PVM a été écrit pour tourner sous UNIX. Actuellement, il est supporté par d'autres sytèmes d'exploitation, et tout récemment, il a été porté sous machines Windows.

Sur chaque machine, il est spécifié deux variables d'environnement :

- PVM_ROOT : cette variable renseigne PVM sur le chemin ou le répertoire où a été installé PVM,
- PVM_ARCH : cette variable renseigne PVM sur la nature de la plateforme sur laquelle il va s'exécuter. Exemple : UNIX, WINDOWS, SUN,DEC,IBM,…

3-4 Les bibliothèques de routines PVM :

PVM offre trois bibliothèques de routines : libpvm3.a contenant des procédures en langage C pour la programmation dans ce langage, libpvmf3.a contenant des sous programmes FORTRAN pour le développement

d’applications dans ce langage et enfin libpvmg3.a contenant des routines pour tâches groupées. L’exécution d’une application sous PVM ne peut se faire qu’après compilation et édition de liens en précisant ces librairies. Les entêtes des programmes doivent comporter : INCLUDE <pvm3.h>.

Une application sous PVM est un ensemble de processus appelés ″tâches″, autonomes, exécutant leur propre code et communiquant via des appels à des sous programmes contenus dans les librairies. Chaque tâche est identifiée par un numéro unique, appelé TID (Task Identifier), affecté par PVM. Ce dernier utilise un entier codé sur 32bits dont un champs de 18 bits pour le numéro de la tâche et un champs de 12bits pour identifier la machine sur laquelle se déroule cette tâche. Cette notion de TID est essentielle dans les communications entre les tâches.

Ces sous programmes peuvent être classés dans les grandes catégories suivantes :

- gestion de l’environnement
- communications point à point
- communications collectives
- gestion des groupes.

Dans la gestion de l’environnement figure la gestion dynamique de la machine parallèle virtuelle par les routines : PVM_ADDHOST et PVM_DELHOST qui permettent de rajouter/supprimer une machine durant le déroulement de l’application parallèle. PVM_MSTAT permet de tester l’état d’une machine : libre ou occupée.. La figure B.3 montre l’affectation des tâches aux différentes machines du réseau.

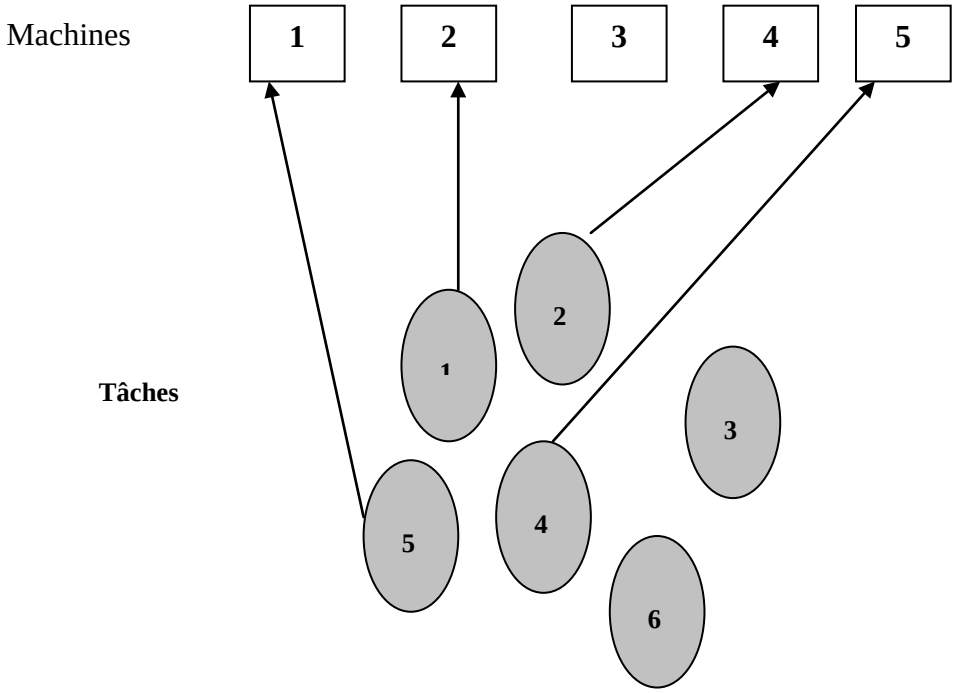


Fig. B.3 : Gestion des machines.

Dans les communications point à point, les tâches communiquent entre elles passant par les démons locaux à chaque machine. Les communications inter-démons utilisent UDP alors que les communications tâche-démon et tâche-tâche utilisent TCP.

Une communication point à point a lieu entre deux processus l'un appelé Emetteur l'autre Récepteur.

La fonction ***pvm_mytid( )*** est la première fonction à appeler par une application ou une tâche PVM car elle permet d'enroller cette application sous environnement PVM. Elle renvoie le TID affecté à cette tâche . Pour gérer les erreurs, l'application peut faire appel à ***pvm_perror( )*** et s'arrêter. Cette fonction imprime un message d'erreur indiquant un problème qui a entravé le bon fonctionnement du programme parallèle. En faisant appel à la routine ***pvm_parent( )*** permet de connaître le TID de la tâche qui a lancé la tâche appelante. Si cette dernière est le processus principal alors un code erreur est renvoyé PvmNoParent et c'est ainsi qu'on peut distinguer le

processus principal des processus ″fils″. La routine pvm_exit( ) doit finir une tâche donnée car elle ″prévient″ PVM que cette tâche s'arrête et qu'elle n'aura plus à utiliser aucune ressource ni routines de PVM.

L'Emetteur et le Récepteur sont identifiés par leur TID. Un message échangé entre eux comporte une enveloppe. Celle-ci est constituée :

- du TID de l'Emetteur
- du TID du Récepteur
- de l'étiquette (TAG) du message
- du type et de la taille des données (entiers, réels, caracères,…).

La figure 5.4 illustre une communication point à point.

Dans l'échange de données, l'envoi de messages se fait en trois étapes :

- attribution d'un identificateur associé à un espace mémoire temporaire (un buffer).

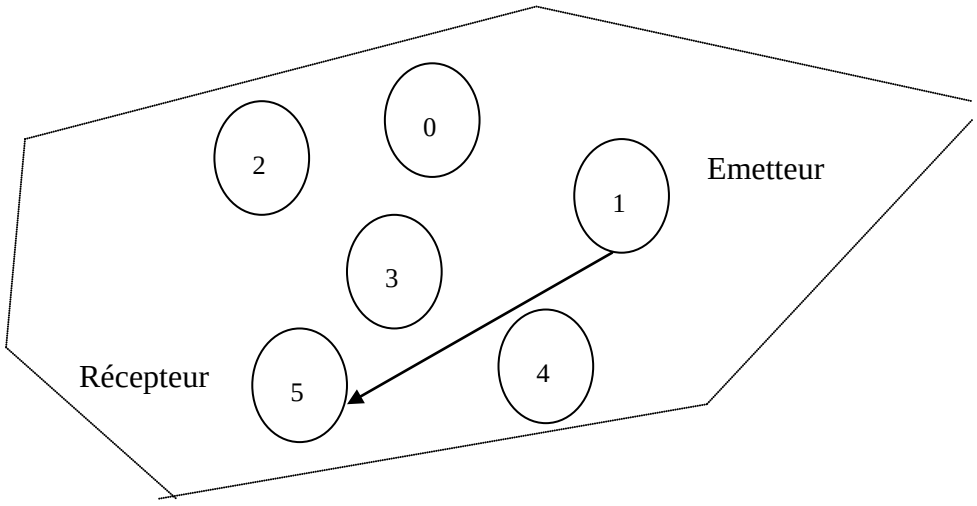


Fig. 5.4 : <u>Communication point à point</u>.

- Compactage des données éventuellement hétérogènes (entières, réelles, car,…) dans cet espace.
- Envoi proprement dit du message contenu dans cet espace.

La réception d'un message se fait en deux étapes :

- Réception du message dans un espace mémoire temporaire (buffer).
- Décompactage des données éventuellement hétérogènes.

Avant d'entamer toute communication il faut commencer par initialiser le ″buffer″ en faisant appel à la routine ***pvm_initsend(encodage, bufid)***

Le tableau suivant résume les valeurs possibles du paramètre encodage et leur signification.

Val. Encodage	Opération
PVMDATADEFAULT	- formatage XDR - Message dupliqué en mémoire - Pour machines hétérogènes
PVMDATARAW	- pas de formatage XDR - Message dupliqué en mémoire - pour machines homogènes
PVMDATAINPLACE	-pas de formatage XDR -message non dupliqué -pour machines homogènes.

Dans les communications point à point, PVM offre trois routines importantes : pvm_spawn, pvm_send et pvm_recv.

- La routine pvm_spawn : permet de lancer dynamiquement l'exécution d'une tâche à distance (sur une machine du réseau).

Int num= pvm_spawn(char task, char **argv, int flag, char* where, int ntask, int *tids)*

Où : task = chaîne de caractères qui contient le nom du fichier exécutable du processus PVM à démarrer. Cet exécutable doit résider sur le ou les hosts sur lequel il sera exécuté.

Exemple : $HOME/pvm3/bin/$PVM_ARCH/filename.

Argv = pointeur sur une table d'arguments de l'exécutable se terminant par NULL. Si l'exécutable n'a pas besoin d'arguments, alors il sera remplacé par NULL.

Flag=entier dont la valeur signifie une certaine option.

Exemple : PVMTaskDefault =0 alors PVM peut choisir n'importe quel host libre pour exécuter la tâche, PVMTaskHost=1 pour spécifier un host particulier, PVMTaskArch=2 pour spécifier le type d'architecture, etc…

Where=chaine de caractères qui spécifie où lancer le processus (nom du host).

Ntask= nombre de copies de l'exécutable à lancer.

Tids=vecteur entier de dimension égale au plus à ntask. Il contient les TIDs des tâches lancées par l'appel à Pvm_spawn.

Si il y a une erreur qui s'est produite au démarrage d'une tâche donnée dans la position correspondante du vecteur on trouvera le code de cette erreur.

Dans la variable de retour numt, on trouvera le nombre de tâches actuel démarré. Si cette valeur est négative alors une erreur système s'est produite

Si cette valeur est positive mais inférieure à ntask alors il y a erreur dans le démarrage de l'une des tâches, il faut consulter le vecteur ntids. D'où il y a toujours possibilité de gérer les erreurs.

-La routine pvm_send : permet d'envoyer un message à partir d'un buffer.

Int info=pvm_send(int tid, int msgtag)

Où

Tid= un entier spécifiant le TID de la tâche réceptrice.

Msgtag=étiquette du message. C'est un entier positif.

Info= entier positif spécifiant un code de retour.

La routine pvm-send envoie un message contenu dans le buffer actif de PVM à la tâche pvm identifiée par Tid. Si l'envoi du message est réussi alors la variable de retour contient un 0.

La routine pvm_send est asynchrone : elle rend la main au système dés que l'envoi du message est en route vers la tâche qui doit le recevoir. On dit que le ″send″ n'est pas bloquant.

La routine pvm_recv : permet de recevoir un message

Bufid=pvm_recv(int tid, int msgtag)

Tid = entier qui spécifie un identificateur de tâche qui doit envoyer un message . Lorsque –1 est mis dans ce paramètre cela signifie que l'on attend un message de n'importe quel host. Le ″receive″ est bloquant dans PVM, ie : une tâche qui fait appel à pvm_receive( ) attend une réponse avant de pouvoir continuer son exécution (elle est peut être en attente de données).
Msgtag= entier étiquette du message

Pour enoyer un message, il faut d'abord initialiser un buffer, empacter les données dans ce buffer pour enfin l'envoyer.
Exemple :

```
Pvm_initsend(PvmDataDefaut)
Pvm_pkint(&i,1,1) ;          permet d'empacter un entier
Pvm_pkstr(name) ;            permet d'empacter une chaine de caractères
Pvm_send(TID,TAG) ;          permet d'envoyer le message à la tâche identifiée
                             par TID avec une étiquette TAG.
```

Pour la réception du message :

```
Pvm_recv(TID,TAG)
Pvm_upkint(&i,1,1)
Pvm_upkstr(name) ;
```

Il faut ″déballer″ les données dans le même ordre.

Notons que l'on peut gérer plusieurs tampons actifs aussi bien pour la réception que pour l'envoi de messages. Ceci est possible grâce aux routine ***pvm_freebuf()*** et ***pvm_kbuf()***. Les routines ***pvm_getsbuf()*** et ***pvm_getrbuf()*** permettent de connaître les identificateurs de buffers actifs courants respectivement pour l'envoi et pour la réception de mesages.

Pour les communications groupées, il y a possibilité de création de groupes de tâches dans PVM. Pour cela, une tâche peut à tout moment rejoindre un groupe par la routine ***pvm_joingroup("nomgroupe")*** ou peut quitter un groupe par ***pvm_lvgroup("nomgroupe")***. Dans le groupe il est affecté à une tâche donnée un TID du groupe gtid. Pour le connaître, une tâche peut appeler la routine :
Gtid=pvm_gettid("nomgroupe", inum).
Ceci permet à l'utilisateur de créer des groupes de tâches qui se synchronisent entre elles par création de barrière de synchronisation.

Enfin pour conclure, nous pouvons dire que PVM est un système complet de communications, nous en avons présenté quelque peu cet aspect de PVM. C'est un environnement de programmation parallèle intégrant toute sorte d'hétérogeinité et permettant l'évolution d'un programme parallèle sur un ensemble de machines reliées via un réseau. Nous ne manquons pas de signaler que cet environnement de programmation parallèle est encore en pleine évolution et sous tests dans le monde.

Références Bibliographiques

[1] W. Wolf, *Modern VLSI design*, Edition Prentice Hall, 1994.

[2] C. Piguet, A. Stauffer, J.Zahnd, *Conception des circuits ASIC numériques CMOS*, Dunod Paris 1990.

[3] A.R. Baba-Ali, *MOSTAFA : Un analyseur temporel pour les circuits VLSI digitaux MOS*, Thèse de magister en cybernetique, soutenue au CDTA Oct. 1990.

[4] N. West, K. Eshraghian, *Principles of CMOS VLSI design, a system perspective*, Addison Wesley, 1985.

[5] A.R. Baba-Ali, *Optimisation des performances temporelles de circuits intégrés VLSI*, Thèse de doctorat d'état en Electronique, soutenue à l'ENP, Mars 1998.

[6] Yu-Chin Hsu, K. F.Tsai, J.T. Liu, E.S. Lin, *VHDL Modeling for Digital Design Synthesis*, Kluwer Academic Publishers, 1996.

[7] D. Gajski & Al, *High level synthesis : Introduction to chip and system design*, Kluwer-academic, 1992.

[8] S. Even, *Graph algorithms*, computer science presse, 1979.

[9] P. Mazumder, E. Rudnick, *Genetic algorithm for VLSI Design, Layout and Test automation 1/e*, Published december 1998 by Prentice Hall PTR, Copyright 1999.

[10] D. Benrabah, *Vérification Intéractive de circuits par l'analyse incrémentale*, thèse de magister en informatique, soutenue à l'INI, juin 1991.

[11] W. Benjamin Wah, C. V. Ramamoorthy, *Theory of algorithms and Computation Complexity with Applications to Software Design*, Handbook of software engineering, pp 64-91, from university of California, Berkeley 1980.

[12] C. Papadimitriou & Al, *Combinatorial Optimisation, algorithms and complexity*, Prentice Hall, 1996.

[13] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, Reading, MA, 1989.

[14] J. Holland, *Adaptation in natural and artificial systems, Technical report*, university of Michigan, Ann Arbor, 1975.

[14bis] L. Davis, *Handbook of genetic algorithms*, Van Nostrand Reinhold, New-York, 1991.

[15] Z. Michalewicz, *Genetic algorithms + data structures = Evolutionary programs*, Springer-Verlag, 1996.

[16] K.S. Tang, K.F. Man, S. Kwong and Q. He, *Genetic Algorithms and their applications*, IEEE Signal processing magazine, pp 22- 37, Nov. 1996.

[17] S. Embaby, A. Bellaouar, M. El-Masry, *BiCMOS digital IC Design*, F. Berglez & Kluwer Academic Publishers, 1993.

[18] P. Pack Chan, *Algorithms for library-specific sizing of combinational logic*, 27th IEEE Design Automation Conference, pp 353-356, 1990.

[19] M. Moon & Al, *A path oriented algorithm for the cell selection problem*, IEEE Transactions on Computer Aided Design, pp 296-307, Mar. 1995.

[20] Wing Ning Li & Al, *On the circuit implementation problem*, IEEE transaction on Computer Aided Design, pp 1047-1051, Aug. 1990.

[21] Wing Ning Li, *Strongly NP-Hard discrete Gate Sizing problem*, IEEE transaction on Computer Aided Design, Aug.1994.

[22] Chen-Ling Fang, *Timing optimisation by gate resizing and critical path identification*, IEEE transaction on Computer Aided Design, 1995.

[23] R.B. Hitchcock & Al, *Timing analysis of computer hardware*, IBM Journal of Research development, pp 100-105, Jan. 1982.

[24] T. Kirkpatrick & Al, *PERT as an aid to logic design*, IBM Journal of Research development, pp 135-141, Mar. 1966.

[25] A.R. Baba-Ali, A. Bellaouar, *A delay optimisation CAD Tool for mixed CMOS/BiCMOS standard cells circuits*, AJSE (Arabian Journal Science and Engineering) Special issue on microelectronics, Oct. 1994.

[26] A.R. Baba-Ali, A. Farah, *An efficient method for signal flow determination in CMOS/VLSI*, in proceeding IEEE EDTC'96 (European Design & Test Conference), Paris 1996.

[27] M. Hashimoto, H. Onodera, *A performance optimisation Method by Gate resizing based on statistical static timing analysis*, IEICE Trans. Fundamentals, Vol. E8 3-A, N° 12, Décembre 2000.

[28] F. Berglez, H. Fujiwara, *A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran*, ISCAS'85, IEEE Jun. 1985.

[29] AL Geist, J. Dongarra & Al, *PVM 3 User's guide and reference manual*, Engineering physics and Mathematics division, Mathematical section, Oak Ridge National Laboratory, 1994.

[30] F. Thomson Leighton, *Introduction aux algorithmes et architectures parallèles*, Morgan Kaufmann Publishers, 1992.

[31] M**.** Flynn, *Very high-speed computing systems*, 1966.

[32] F. Boumghar, *Parallélisation d'algorithmes de traitement d'images 2D et 3D*, Thèse de doctorat en électronique, juin98.

[33] David A. Patterson, John L. Hennessy, *Computer Architecture, A quantitative approach*, Morgan Kaufmann Publishers,inc, 2nd edition,96.

[34] H. Leroy, *Séminaire sur le traitement parallèles des images*, Faculté de Génie Electrique et Informatique, Département Instrumentation, juin2000.

[35] T. Duboux, *Régulation dynamique du partitionnement de données sur machines // à mémoire distribuée*, Thèse de doctorat en informatique, Ecole Supérieure de Lyon, France 1996.

[36] T. Back, T. Beielstein, B. Naujoks, *Evolutionary algorithms for the optimization of simulation models using PVM*, in proceedings of EURO_PVM'95, France 1995.

[37] D. Whiteley, *The GENITOR algorithm and selection : why rank-based allocation of reproductive trials is best.* In J.D. Schaffer, editor, Proceedings of the 3rd International Conference on Genetic Algorithms, pages 116-121. Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[38] P. Lyaet, C. Baranger, *The Use of PVM with Workstation Clusters in Application of Genetic Optimisation Techniques*, Université de Technologie de Compiègne , Division Modèles Numériques en Mécanique.

[39] K.S. Tang, K.F. Man, S.Kwong, Q. He, *Genetic Algorithms and their Applications*, IEEE Signal Processing Magazine, novembre 1996.

[40] B. SidiBoulenouar, S.Niar, R. Andonov, H. Bourzoufi, *Méthodes exactes et méta-heuristiques parallèles pour la résolution du problème du sac-à-dos*, 6ème Conférence Internationale sur les Mathématiques Appliquées et les Sciences de l'Ingénieur CIMASI, Maroc Octobre 2000.

[41] A. Bertoni, M. Donigo, *Implicit parallelisme in genetic algorithm, Artificial Intelligence*, 1993.

[42] R. Cabrera-Dantart, I. Demeure, P. Meunier, *PHOSPHORUS : Adding shared memory to PVM*, Euro PVM Meeting, Octobre94.

[43] I. Demeure, C. Siegelin et al, *WARPphos: Un système matériel et logiciel de mémoire partagée pour un réseau de stations de travail*, *** d'où cela vient?

[44] C. Amza, P. Keleher et al, *TreadMarks: Shared memory computing on Network of workstations*, IEEE Computer, 1996.

[45] A. Chipperfield, P. Flemming, *Parallel Genetic Algorithms*, Parallel and distributed computing handbook, Y.M. Zamoya, Mc GRAW HILL, 1996.

[46] R. Huang, T.C. Fogary, *Adaptive classification and control-rule optimization via a learning algorithm, for controlling a dynamic system*, Proceeding 30th Conference Decision and Control, Brighton, England 1991.

[47] N. Dodd, D. Macfarlane, C. Marland, *Optimization of artificial neural network structure using genetic techniques implemented on multiple transputers*, IOS Press, Transputing 1991.

[48] M. Botshernison, *Survey and comparison of parallelization techniques for Genetic Algorithms*, Berkeley Aout 1994

[49] HR. Lin, TT Hwang, *On determining sensitization criterion in an interactive gate sizing process,* Computer Aided-Design of integrated circuits and Systems, IEEE vol.182, pages 231-238, Fevrier99.

[50] J. Kim, D.H.C Du, *Performance Optimisation by gate sizing and path sensitization, Computer Aided-Design* of integrated circuits and Systems, IEEE vol 175, pages 459-462, Mai98.

[51] F. Glover, *Tabu Search : A tutorial*, Centerfor Applied Artificial Intelligence, University of Colorado, Boulder, Co 80309-0419, February 1990.

[52] S. Kirkpatrick, C. Gelatt, M. Vechi, *Optimization by Simulated Anealing,* IBM Computer science / Engeneering Technology watson Res. Center, YorkTown Heights, NY, 1982.

[53] S. Benkhider, F. Boumghar, A.R. Baba-ali, *An Evolutionary Approach For The Gate Sizing Problem Of Standard Cells VLSI Integrated Circuits*, Proc. 6ème Conférence Internationale sur les Mathématiques Appliquées et les Sciences de l'Ingénieur CIMASI'2000, Maroc Octobre 2000.

[54] S. Benkhider, F. Boumghar, A.R. Baba-ali, *A Parallel Genetic Approach to The Gate Sizing Problem Of VLSI Integrated Circuits* , Proc. Of the International Conference on Microelectronic ICM'2000, IEEE, Iran, Novembre 2000.