Chapitre 2

Le parallélisme et les métaheuristiques

2.1 Le parallélisme

2.1.1 Introduction

Dans un domaine comme l'informatique, les besoins de performance sont toujours croissants. Les programmes informatiques doivent gérer de plus en plus de données et ce, de plus en plus rapidement. Il est toutefois possible de rendre un programme plus performant en améliorant l'algorithme ou la machine sur laquelle il s'exécute. Une autre approche consiste à combiner les deux options et ainsi adapter un programme à une architecture matérielle plus performante. C'est pour répondre à une demande de plus en plus pressante qu'est apparu le parallélisme.

Traditionnellement, la puissance des ordinateurs a été augmentée en apportant des améliorations dans le matériel et plus particulièrement les circuits électroniques. Grâce notamment à la technologie VLSI (Very Large Scale Integration) qui a permis des avancées spectaculaires dans le domaine des microprocesseurs. À titre d'exemple, les fréquences d'horloge sont passées de 40Mhz (fin des années 80), 2Ghz (2002) [73] jusqu'à 3.6Ghz (2006) [21]. Même si les limites sont loin d'être atteintes, il est constaté que chaque nouvelle amélioration nécessite de plus en plus d'efforts scientifiques et d'investissements financiers. Les concepteurs de processeurs sont ainsi constamment à la recherche d'alternatives pour donner aux processeurs encore plus de puissance. Le parallélisme offre une alternative à cette escalade en mettant côte à côte les processeurs d'une même génération et en tirer un meilleur profit. Ceci se traduit par

un accès multiple aux unités de stockage, des performances extensibles (scalable performance) et des coûts réduits.

D'un point de vue historique, le calcul haute performance en général et le calcul parallèle en particulier, relevaient du domaine spécialisé de l'industrie informatique servant les besoins de la science, de l'ingénierie et de la défense. Des pionniers du parallélisme comme Dennis *et al.* [43] ou Schwartz [136] ont exposé les principes fondamentaux au cours des années soixante-dix. Dennis *et al.* ont publié la première description d'un processeur qui exécute des programmes parallèles présentés sous forme de flux de données tandis que Scwartz a décrit et analysé un nouveau modèle d'ordinateur baptisé "Ultracomputer" dans lequel les processeurs sont reliés par un réseau d'interconnexion. Cependant, ce n'est qu'au milieu des années quatre-vingt que le domaine a connu une importante expansion avec la mise en commercialisation des premières machines parallèles à usage général [121]. Suite à cela, la programmation parallèle est apparue par opposition à la programmation séquentielle "traditionnelle", permettant de contrôler quelles instructions peuvent être exécutées en parallèle sur quels processeurs. Le calcul parallèle, en réduisant le temps d'exécution des programmes, a permis de mettre à portée des calculs qui étaient auparavant impossibles à faire dans des domaines comme la météorologie, la simulation nucléaire, la physique quantique et le génie génétique.

Selon Quinn [125], le calcul parallèle est "le recours à un ordinateur parallèle dans le but de réduire le temps nécessaire à la résolution d'un problème séquentiel". Ainsi, le concept de parallélisme est étroitement lié au matériel utilisé. Selon le même auteur, un ordinateur parallèle est "un ordinateur disposant de plusieurs processeurs et supportant la programmation parallèle". Il fait par conséquent un lien supplémentaire au niveau de la programmation parallèle qu'il définit comme étant "programmer en un langage qui permet une indication explicite sur quelles portions du programme seront exécutées de manière concurrente par les différents processeurs". Le parallélisme possède donc des implications matérielles, algorithmiques et de programmation. Ces différents aspects font l'objet de la section qui suit. En premier lieu, l'environnement matériel dans lequel le domaine du parallélisme évolue est présenté et est suivi par un aperçu de la conception de programmes parallèles selon un modèle particulier. Par la suite, on traite

des standards de la programmation parallèle et des mesures de performance utilisées pour évaluer les résultats. Enfin, des facteurs de performance d'algorithmes parallèles sont discutés.

2.1.2 Les différentes architectures parallèles

L'essence même du parallélisme repose sur le fait de disposer de plusieurs processeurs. Cependant, le fait d'augmenter le nombre de processeurs, de modifier leurs dispositions et de modifier l'accès à la mémoire, fait émerger autant d'architectures nouvelles. Durant plus de trois décennies, du début des années soixante jusqu'au milieu des années quatre-vingt-dix, une grande variété d'architectures parallèles ont été explorées [125]. Un des moyens pour classifier ces architectures est d'avoir recours à la taxonomie de Flynn [53] illustrée à la Figure 2.1. Cette taxonomie, basée sur l'identification et la séparation des flux d'instructions et des données, se compose de quatre principales classifications : SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data), MIMD (Multiple Instruction Multiple Data). Certains auteurs tels Duncan [49] ont proposé leur propre taxonomie mais elle n'a pas eu le même succès que celle de Flynn.

D'abord, la catégorie SISD regroupe les machines à architecture classique telle que la machine de Von Neumann [151] ainsi que la plupart des machines uni-processeurs. Dans cette catégorie, un seul processeur exécute un flux d'instructions sur un flux de données.

En deuxième lieu, la catégorie SIMD regroupe les machines disposant de plusieurs unités de calcul s'occupant chacune de leur propre flux de données. La plupart des ordinateurs SIMD opèrent de manière synchrone grâce à une horloge globale unique. Ce genre d'architecture traite les problèmes où les mêmes instructions sont appliquées sur des données régulières de grande taille.

La troisième catégorie MISD est décrite comme étant une classe de machines ne possédant pas d'exemples concrets. Elle peut être perçue en tant que plusieurs ordinateurs opérant sur un seul flux de données. Les processeurs vectoriels en "pipeline", généralement classés en tant que SISD, sont aussi considérés comme étant de type MISD [12].

Enfin, la dernière catégorie MIMD inclut les machines possédant plusieurs processeurs

opérant sur des flux d'instructions et de données distincts. Les processeurs dans cette catégorie peuvent communiquer à travers une mémoire globale partagée ou par passage de messages.

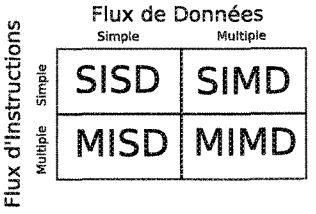

Figure 2.1 – Taxonomie de Flynn [125]

Bien que la taxonomie de Flynn soit un des meilleurs moyens connus pour catégoriser une machine parallèle [125], elle n'est pas assez précise pour décrire les machines actuelles. Alba [10] propose une extension au modèle de Flynn (Figure 2.2) où la classe MIMD est subdivisée selon l'organisation de la mémoire.

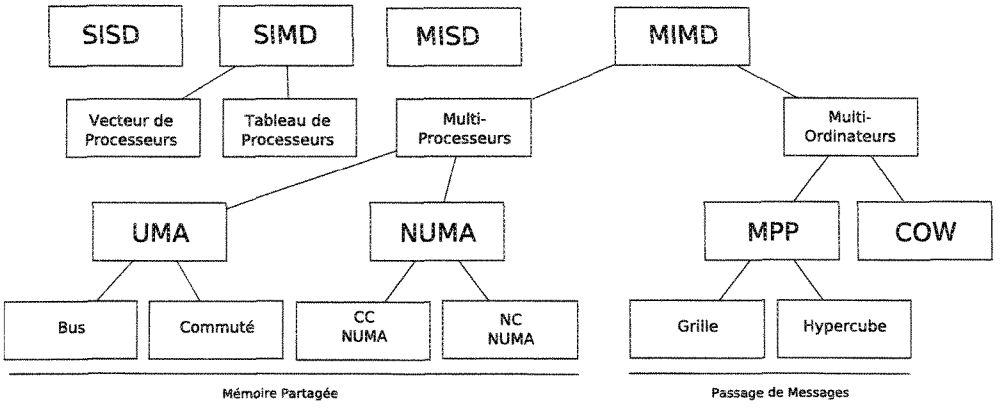

Figure 2.2 – Taxonomie de Flynn étendue [10]

D'autres taxonomies ont aussi été publiées. Les principales étant celles de Snyder [137] et Duncan [49]. La taxonomie de Flynn couvre théoriquement toutes les possibilités d'architectures parallèles. Cependant, la plupart des machines actuelles sont de type MIMD et dans cette même classe, plusieurs catégories apparaissent et font l'objet de la prochaine section.

2.1.2.1 Architectures Multi-processeurs

Lorsque, dans un ordinateur, plusieurs processeurs accèdent à une mémoire partagée, l'architecture ainsi formée est appelée multi-processeurs. On distingue deux catégories : les multi-processeurs centralisés et les multi-processeurs distribués.

Les multi-processeurs centralisés sont une extension d'un ordinateur classique puisque

tous les processeurs ont accès à une même mémoire principale et un bus relie les périphériques entrée/sortie à la mémoire partagée et relie les processeurs les uns aux autres. Chaque processeur a également accès à une mémoire cache qui réduit le temps d'attente de donnée provenant de la mémoire principale. La mémoire cache est une mémoire à accès rapide mais de petite taille, stockant les informations les plus utilisées par le processeur. Deux autres appellations sont aussi utilisées pour désigner les multi-processeurs centralisés : Uniform Memory Access (UMA) Multiprocessor et Symetric Multiprocessor (SMP). L'appellation UMA vient du fait que le temps d'accès à chaque adresse mémoire est constant parmi les processeurs et l'appellation SMP est utilisée pour indiquer l'unicité de la mémoire. Les multi-processeurs ont certains inconvénients comme le problème de cohérence de la mémoire cache. Ce problème se pose par exemple dans le cas où un processeur *P1* effectue une lecture à une adresse X de la mémoire principale à la suite d'une écriture par un processeur *P2* à la même adresse X. Si la valeur retournée par la lecture est l'ancienne valeur alors on peut dire qu'il y a problème de cohérence de la mémoire cache. Plusieurs approches ont été proposées au cours des dernières années afin de résoudre ce problème, notamment au niveau matériel [149]. Le schéma d'un multi-processeur centralisé est illustré à la Figure 2.3.

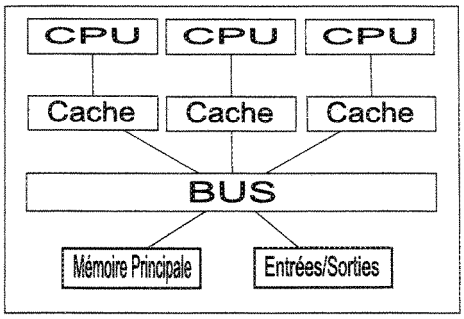

Figure 2.3 – Multi-processeurs centralisés

Le fait que, dans l'architecture précédente, la mémoire soit partagée limite le nombre de processeurs branchés en série. Ceux-ci ne dépassent pas généralement quelques douzaines [125]. Pour remédier à cela, chaque processeur est équipé de sa propre mémoire locale et les entrées/sorties sont distribuées. C'est ce qu'on appelle les multi-processeurs distribués, architecture aussi connue sous le nom de "Non Uniform Memory Access" (NUMA) (Figure

2.4). L'appellation "Non Uniform" vient du fait que le temps d'accès à la mémoire de chaque processeur n'est pas constant, par opposition à l'architecture UMA vue préalablement.

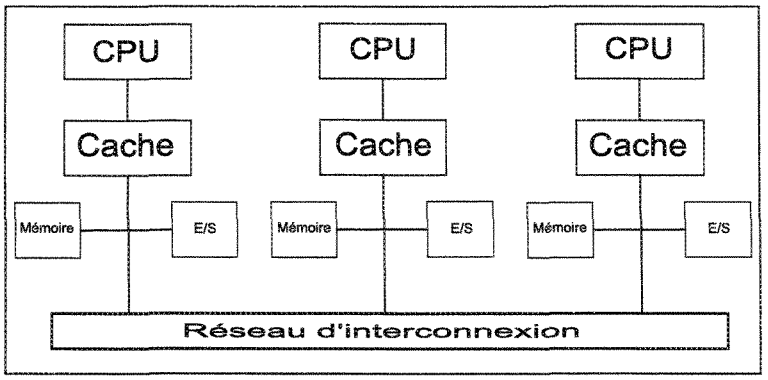

Figure 2.4 – Multi-processeurs distribués

2.1.2.2 Architectures Multi-ordinateurs

Un multi-processeurs dont les espaces d'adressage sont séparés constitue une nouvelle architecture appelée multi-ordinateurs. Puisqu'aucun espace n'est partagé par les processeurs, on évite les problèmes de cohérence de cache moyennant une gestion des communications pouvant être complexe et coûteuse. Deux grandes catégories constituent les multi-ordinateurs : multi-ordinateurs symétriques et multi-ordinateurs asymétriques.

Un multi-ordinateur asymétrique (Figure 2.5) est constitué d'un ordinateur principal et d'une série d'autres ordinateurs secondaires reliés en réseau. Ces ordinateurs utilisent des processeurs dédiés au calcul parallèle. Cette architecture a tendance à être dépendante de la machine principale et de gaspiller ses ressources lorsqu'aucune partie parallèle n'est utilisée [125]. L'appellation asymétrique est conséquente à l'unique accès de l'utilisateur via l'ordinateur principal.

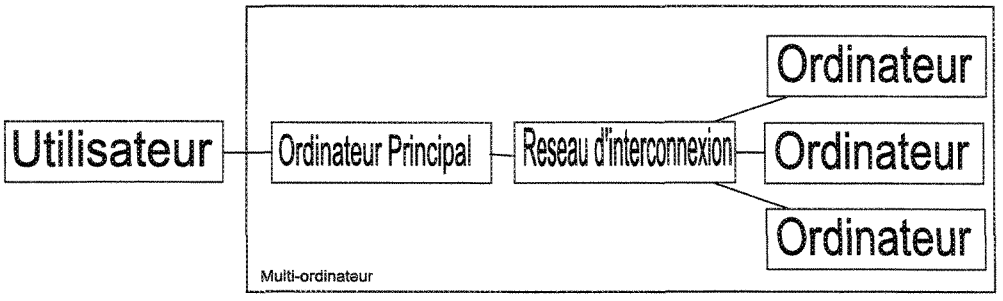

Figure 2.5 – Multi-Ordinateur asymétrique

Dans un multi-ordinateur symétrique (Figure 2.6), tous les ordinateurs exécutent la même portion du programme. Cependant, chaque noeud pouvant servir de point d'entrée/sortie, il n'est pas facile de donner l'illusion de travailler sur un seul système. Le principal souci de programmation devient la répartition équitable de la charge de travail entre tous les processeurs. L'appellation symétrique vient justement de la répartition des points d'entrée/sortie et du partage du calcul entre tous les noeuds.

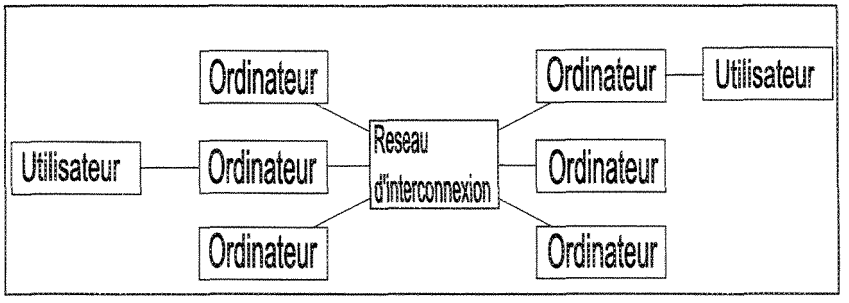

Figure 2.6 – Multi-Ordinateurs symétrique

Parmi les machines parallèles commercialisées, on cite la série Origin de SGI et, plus particulièrement, le modèle Origin 2000 [92] qui est un multiprocesseur ce type ccNUMA pouvant contenir jusqu'à 512 noeuds. Chaque noeud est composé d'un ou de deux processeurs R10000. Pour sa part, le SP2 de IBM [3] est un multi-ordinateur de type SMP qui permet de connecter un éventail de 2 à 512 noeuds. Les noeuds sont connectés par un réseau haute performance utilisant des paquets commutés et dédié à la communication inter-processeurs. Chaque noeud contient sa propre copie du système d'exploitation AIX (IBM).

Cet aperçu des architectures parallèles a permis de présenter un domaine où la diversité du matériel est importante. Chaque architecture citée précédemment possède ses avantages et ses inconvénients. Les multi-processeurs ont l'avantage de supporter de multiples utilisateurs, de ne pas perdre en efficacité lorsqu'ils traitent du code parallèle conditionnel et de disposer des mémoires caches pour réduire la charge sur le bus [125]. Par contre, les multi-processeurs sont limités par le nombre de processeurs qu'ils peuvent contenir et leur prix tend à augmenter de manière exponentielle à chaque extension [10]. Les multi-ordinateurs ont également l'avantage d'être faciles à installer et à étendre, d'être plus flexibles et d'avoir un meilleur rapport prix/performance. Cette architecture, plus précisément les multi-ordinateurs

symétriques est la plus adaptée à la programmation parallèle. Par contre, la rapidité du réseau qui relie les ordinateurs devient un facteur important de performance. Une architecture adéquate ne peut donner de bonnes performances sans des algorithmes conçus pour l'exécution parallèle. Dans la section qui suit, la notion de conception d'algorithmes parallèles va être étudiée.

2.1.3 Conception d'algorithmes parallèles

La démarche à suivre pour concevoir un algorithme parallèle diffère de celle d'un algorithme séquentiel. De nombreux travaux ont été faits dans le domaine de la conception d'algorithmes parallèles. Des auteurs comme Jàjà [87] et Grama *et al.* [73] ont énoncé des approches de conception d'algorithmes parallèles. Foster [56] propose, pour sa part, une méthodologie précise appelée "modèle tâche canal" qui est illustrée à la Figure 2.7. Ce modèle comporte quatre étapes principales : la décomposition, la communication, l'agglomération et l'assignation.

La décomposition a pour but d'identifier le plus grand nombre de sources de parallélisme possible. Cette phase fait en sorte que le problème soit décortiqué en un maximum de tâches élémentaires.

La phase de communication est abordée une fois les tâches élémentaires identifiées. Elle a pour but de les relier par des canaux de communication. C'est une étape purement liée à l'environnement parallèle parce qu'un programme séquentiel n'a pas besoin de faire communiquer ses différents composants.

La troisième étape est celle de l'agglomération. Cette phase impose des choix à faire en fonction de l'architecture parallèle adoptée. Elle consiste à rassembler les tâches primitives en tâches plus grandes, l'objectif étant de minimiser la communication entre les tâches. Un autre but de l'agglomération est d'augmenter le plus possible l'extensibilité du programme et sa portabilité.

Finalement, l'étape d'assignation consiste à attribuer les tâches aux processeurs. L'objectif de cette phase est double : répartir de façon équitable les tâches et minimiser les

communications inter-processeurs.

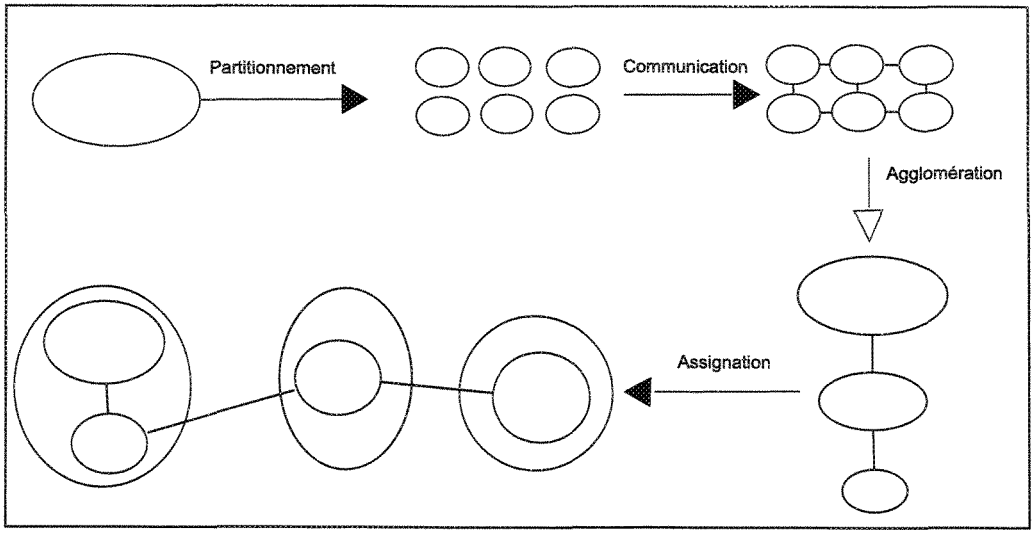

Figure 2.7 – Schéma récapitulatif du modèle tâche canal [56]

Après avoir conçu un algorithme destiné à l'exécution parallèle, l'étape suivante consiste à transposer cet algorithme en programme parallèle. Dans la prochaine section, la programmation de ces algorithmes dans des environnements parallèles sera présentée.

2.1.4 La programmation parallèle

De nombreux langages de programmation et bibliothèques ont été développés pour la programmation parallèle. Ces langages diffèrent les uns des autres par la vision de l'espace d'adressage accordée au programmeur, le degré de synchronisation et la multiplicité des programmes [73]. Les langages de programmation parallèles se basent sur des approches de conception aussi appelés "paradigmes de programmation" dont certains seront expliqués dans les sous-sections qui suivent. En premier lieu, il sera question du modèle à passage de messages et du standard MPI. Ensuite, on présentera le modèle à mémoire partagée avec le standard OpenMP. Enfin, la dernière sous-section fera un survol d'autres approches et de standards de programmation.

2.1.4.1 Modèle à passage de messages et le standard MPI

Le standard MPI (Message Passing Interface) [55] est un modèle de programmation par bibliothèques permettant l'implémentation d'un programme sur une architecture parallèle selon le modèle à passage de messages (Figure 2.8). Le modèle à passage de messages est très

similaire au modèle tâche canal discuté précédemment. Ce modèle suppose que l'environnement matériel est un ensemble de processeurs possédant chacun une mémoire locale.

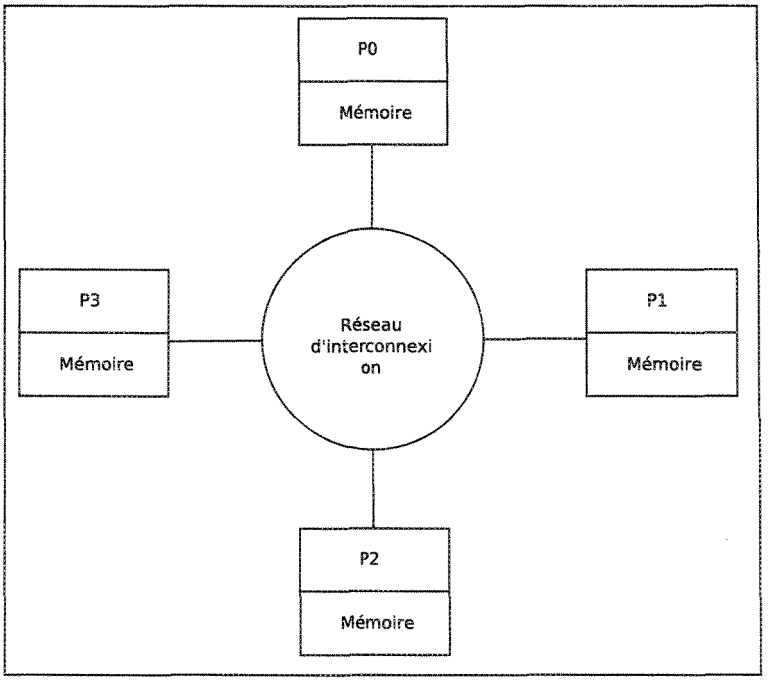

Figure 2.8 – Modèle à passage de messages [125]

Il est à préciser que le modèle à passage de messages est indépendant de l'architecture matérielle et peut être implémenté et utilisé sur une architecture à mémoire partagée. Le modèle à passage de messages repose aussi sur le principe de la parallélisation explicite [73]. En effet, le programmeur doit analyser l'algorithme ou le code séquentiel et identifier les portions qui peuvent être parallélisées. Puisque chaque processeur n'a accès qu'à sa propre mémoire, les seules interactions possibles se font par envois et réceptions de messages. Les opérations permettant l'envoi et la réception de messages sont respectivement "send" et "receive" et peuvent être soumises selon différents modes (bloquant, non bloquant, avec tampon etc.). La Figure 2.9 illustre un exemple d'envoi de messages avec ce modèle.

MPI est essentiellement apparu pour régler un problème de multitude d'interfaces à passage de messages [10]. En effet, les ordinateurs parallèles de première génération avaient des bibliothèques fournies par les constructeurs qui bien souvent n'étaient compatibles qu'avec les machines en question ou celles d'un même constructeur. Même si la différence entre les

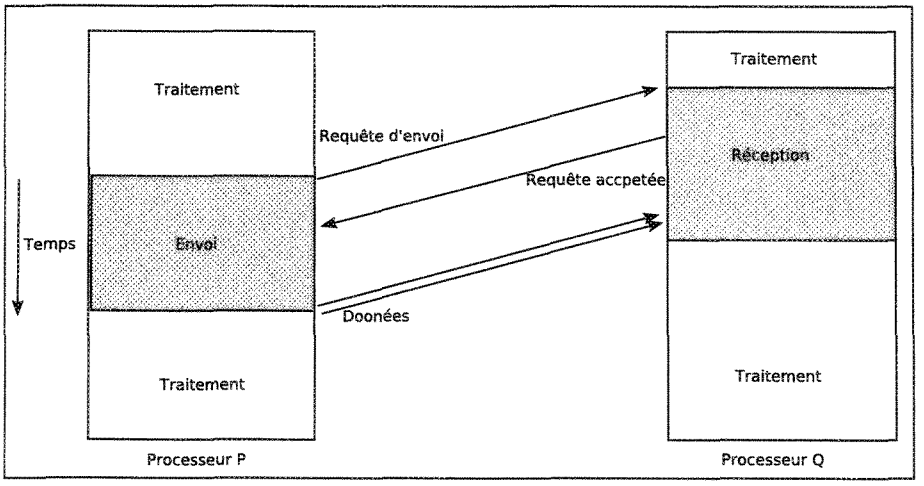

Figure 2.9 – Abstraction d'envoi de messages niveau utilisateur [150]

standards était essentiellement syntaxique, des nuances au point de vue sémantique impliquaient souvent un effort de conversion supplémentaire pour passer d'une machine à une autre. La popularité du modèle à passage de messages a créé le besoin d'unification et de portabilité et MPI a vu le jour principalement dans cette optique [23]. MPI contient plus de 125 routines des plus simples aux plus complexes.

La Figure 2.10 présente un exemple simple de programme MPI permettant d'identifier les processeurs impliqués. Dans ce programme, l'initialisation de MPI se fait par la commande MPI_Init et son arrêt par MPI_Finalize. Toutes les directives parallèles doivent être incluses entre ces deux instructions. Dans l'exemple, on fait appel à MPI_Comm_rank pour connaître le rang du processeur, MPI_Barrier pour synchroniser les processeurs et MPI_Comm_Size pour connaître le nombre total de processeurs. La deuxième partie la Figure 2.10 illustre le résultat en sortie de l'exécution sur 4 processeurs.

2.1.4.2 Modèle à mémoire partagée et le standard OpenMP

OpenMP [116] est un standard de programmation parallèle utilisant le modèle à mémoire partagée illustré à la Figure 2.11. Ce modèle est une abstraction d'un multiprocesseur centralisé. La couche matérielle est supposée être une collection de processeurs ayant chacun accès à la même mémoire. Par conséquent, les processeurs peuvent interagir par le biais de variables partagées.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
 int id ; // rang du processeur
 int p ; // nombre de processeurs

 MPI_Init (&argc , &argv) //lancement de MPI
 MPI_Comm_rank (MPI_COMM_WORLD, &id) ; //affectation du rang du processeur dans id
 printf (" \n Processeur %d prêt pour MPI ",id) ; //affichage des rangs

 MPI_Barrier (MPI_COMM_WORLD) // barrière de synchronisation
 if ( !id) // condition pour le processeur 0
 {
   MPI_Comm_size (MPI_COMM_WORLD, &p) ; //affectation du nombre de processeurs dans p
   printf (" \n Il y a au total %d processeurs sollicités ",p) ; //affichage du nombre de processeurs
 }
 MPI_Finalize () ; // arrêt de MPI
 return 0 ;
}
```

```
Processeur 0 prêt pour MPI
Processeur 1 prêt pour MPI
Processeur 2 prêt pour MPI
Processeur 3 prêt pour MPI
Il y a au total 4 processeurs sollicités
```

Figure 2.10 – Exemple de code MPI

Il se distingue par son utilisation du modèle fork/join (Figure 2.12) où un thread principal se subdivise et se rejoint lorsque nécessaire. Concrètement, le thread maître exécute toutes les portions séquentielles du code et dès qu'il y a besoin d'une exécution en parallèle, un "fork" est enclenché ce qui engendre des thread esclaves qui démarrent à partir de ce point. Au moment où l'exécution parallèle achève, ces threads sont tués ou suspendus et le contrôle revient au thread maître, c'est ce qu'on appelle un "join".

OpenMp exécute un programme de manière séquentielle jusqu'à ce qu'il rencontre une directive de compilateur suite à laquelle il crée les threads parallèles. OpenMp est utilisé comme extension aux langages C, C++ et Fortran.

La Figure 2.13 illustre un exemple de programme OpenMP qui effectue le même traitement que l'exemple vu en MPI, à savoir identifier les processeurs présents. Dans cet exemple, le bloc d'exécution en parallèle est délimité par la directive #pragma omp parallel. L'identifiant du processeur est obtenu avec la fonction get_thread_num. La synchronisation est effectuée avec la directive #pragma_omp_barrier. Enfin, le nombre total de processeurs, qui correspond au nombre de threads, est obtenu grâce à la fonction omp_get_num_threads. Le résultat de l'exé-

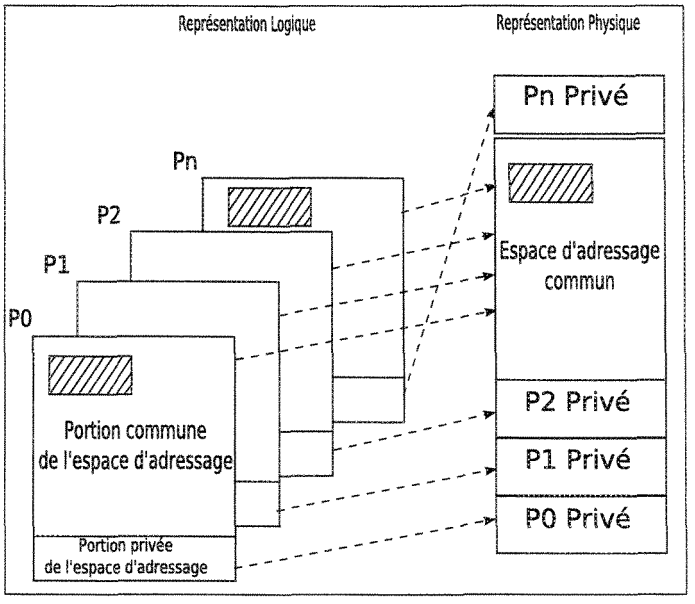

Figure 2.11 – Modèle à mémoire partagée [35]

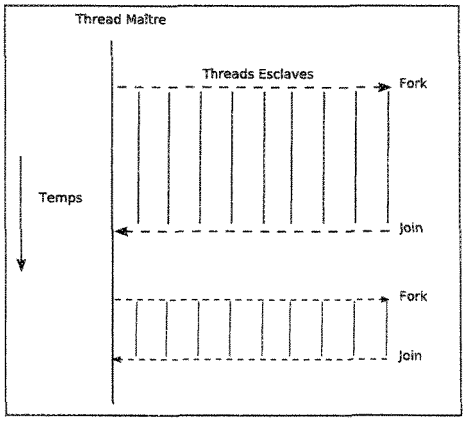

Figure 2.12 – Modèle Fork/Join [125]

cution de cet exemple sur 4 processeurs est illustré dams la deuxième partie de la Figure 2.13.

MPI et OpenMP matérialisent deux approches différentes de programmation parallèle. De par leurs abstractions matérielles différentes, chaque modèle convient à un type d'architecture. En effet, le modèle à passage de messages convient mieux aux architectures de type multi-ordinateur (processeurs ayant chacun leur mémoire et reliés par un réseau) et le modèle à mémoire partagée convient mieux aux architectures de type multiprocesseurs (processeurs accédant à la même mémoire via un bus). De plus, dans le modèle à passage de messages, les processus parallèles restent actifs tout au long de l'exécution du programme

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
 int id; // rang du thread
 int nt; // nombre de threads
 #pragma omp parallel private(id) // début du bloc parallèle
 {
   id = omp_get_thread_num(); // affectation du rang du thread dans la variable id

   printf("\n Thread %d pret pour OpenMP", id); // affichage des rangs

   #pragma omp barrier // barrière de synchronisation

   if (!id) // condition pour le thread 0
   {
     nt = omp_get_num_threads(); //affectation du nombre de threads dans nt
     printf("Il y a au total %d threads sollicités",nt); // affichage du nombre de threads
   }
 }
return 0;
}
```

```
Thread 0 prêt pour OpenMP
Thread 1 prêt pour OpenMP
Thread 2 prêt pour OpenMP
Thread 3 prêt pour OpenMP
Il y a au total 4 threads sollicités
```

Figure 2.13 – Exemple de code OpenMP

alors que dans le modèle à mémoire partagée, un thread est actif au début du programme et ce nombre peut changer dynamiquement tout au long de l'exécution. C'est un avantage considérable par rapport au modèle précédent puisqu'il permet la parallélisation incrémentale, qui est la transformation d'un programme séquentiel en un programme parallèle, un bloc de code à la fois. Un certain nombre d'autres standards de programmation parallèle existent et seront présentés dans la section suivante.

2.1.4.3 Autres standards de programmation parallèle

"Parallel Virtual Machine" (PVM) [141], comme son nom l'indique, fournit à l'utilisateur une abstraction de machine parallèle qui s'occupe de gérer les communications et autres conversions de données parmi les processeurs physiques. À l'instar de MPI, PVM est basé sur le modèle à passage de messages. L'utilisateur manipule un ensemble de tâches qui coopèrent entre elles en ayant la possibilité d'ajouter et de supprimer des ordinateurs à la machine virtuelle. La principale force de PVM est de permettre d'implémenter le modèle à passage de messages sur un réseau hétérogène. Différentes architectures et différents systèmes

d'exploitation peuvent faire partie de la machine virtuelle et tout ceci dans la plus grande transparence. Ceci est possible grâce à l'exécution d'un processus en arrière plan appelé "démon" sur chacune des machines du réseau. Le démon PVM s'occupe de l'harmonisation de la communication entre les différentes machines. De plus, PVM permet à l'utilisateur de décider explicitement quelle tâche va être attribuée à quel ordinateur de la machine parallèle.

D'autres standards de programmation parallèle sont apparus sous la forme d'extension de langages. D'ailleurs, certains auteurs comme Foster [57] considèrent MPI et OpenMP non pas comme des extensions de langages mais comme des approches à base de bibliothèques. Au lieu de définir au complet un nouveau langage, ces extensions ont l'avantage de garder la syntaxe du langage hôte et ne nécessitent pas une formation particulière de la part de l'utilisateur. Pour le langage C++, on peut citer Compositional C++ qui est communément appelé CC++ [31].

Enfin, certains standards sont apparus sous la forme de langages de données parallèles dont les plus notoires ont été traités par Foster [57].

Dans les sections précédentes, l'accent a été mis sur l'importance de l'approche de conception et de programmation pour faire des programmes parallèles performants. Une fois ces phases terminées, ce sont les tests qui déterminent si telle ou telle approche est performante. La section qui suit explore justement les moyens de vérifier la qualité de la conception et de la programmation avec des outils et des formules quantitatives.

2.1.5 Les mesures de performance

Des mesures de performance de programmes parallèles sont mises à disposition pour justifier le choix d'une architecture, prédire le comportement d'un algorithme ou encore calculer le temps économisé par rapport à une autre version. La motivation derrière l'apparition de ces mesures de performance est la nécessité d'avoir des outils dans un domaine aussi complexe que le calcul parallèle. Scherr [135], considéré comme le pionnier de l'évaluation de performance parallèle, a insisté sur le fait que la réaction de la machine par rapport aux exigences de l'utilisateur a besoin d'être estimée à l'avance. À partir de ses recherches, beaucoup de métriques

ont vu le jour et certaines seront exposées dans la partie qui suit. Une fois appliquées, ces mesures sont autant de données statistiques qui alimentent des tables et graphiques pouvant servir de base à des théories et autres démonstrations.

2.1.5.1 Accélération

La première et probablement la plus importante mesure de performance d'un algorithme parallèle est l'accélération [9]. Elle est le rapport entre le temps d'exécution du meilleur algorithme connu sur 1 processeur et celui de la version parallèle. Sa formule générale est :

$$Accélération = \frac{Temps\ d'exécution\ séquentiel}{Temps\ d'exécution\ parallèle}. \tag{2.1}$$

Une des interprétations les plus connues de cette définition est la limite énoncée par Amdahl [13] plus connue sous le nom de "loi d'Amdahl" qui est la suivante : si s est la portion du code qui doit être exécutée séquentiellement, l'accélération est limitée par le haut par $\frac{1}{s+(1-s)/n}$ où n est le nombre de processeurs.

La loi d'Amdahl est connue comme étant la plus fondamentale et plus controversée des résultats dans le domaine de l'évaluation des performances parallèles [100]. Elle est considérée fondamentale par sa simplicité et sa généralité. En effet, elle définit une borne supérieure pour la performance d'un algorithme parallèle se basant sur un seul paramètre logiciel (la fraction séquentielle) et un seul paramètre matériel (le nombre de processeurs). La controverse vient du fait que cette borne impose des limites sévères aux performances et aux bénéfices d'avoir recours au parallélisme.

On distingue trois types d'accélérations : sous-linéaire (inférieure au nombre de processeurs), linéaire (égale au nombre de processeurs) et super-linéaire (supérieure au nombre de processeurs). Il faut toutefois noter que les accélérations sous-linéaires sont les plus répandues. Belding [20] et Lin [95] ont rapporté des accélérations super-linéaires mais la notion est toujours controversée. Des auteurs comme Faber *et al.* [51] stipulent qu'une telle accélération ne peut pas exister parce que les coûts de communications ne permettent pas de l'atteindre. D'un autre côté, des auteurs comme Parkinson [118] défendent l'accélération super-linéaire avec des

résultats expérimentaux.

2.1.5.2 Efficacité

Une autre métrique populaire est l'efficacité (formule 2.2). Elle donne une indication sur le taux d'utilisation des processeurs sollicités. Sa valeur est comprise entre 0 et 1 et peut être exprimée en pourcentage. Plus la valeur de l'efficacité est proche de 1, meilleures sont les performances. Une efficacité égale à 1 correspond à une accélération linéaire.

$$Efficacité = \frac{Accélération}{p} = \frac{Temps\ d'exécution\ Séquentiel}{Temps\ d'exécution\ Parallèle \times p} \tag{2.2}$$

Il existe d'autres variantes de cette métrique. Par exemple, "l'efficacité incrémentale" [9](Formule 2.3) qui donne l'amélioration du temps à l'ajout d'un processeur supplémentaire. Elle est aussi utilisée lorsque le temps d'exécution sur un processeur n'est pas connu. La formule de cette métrique a été généralisée (Formule 2.4) pour mesurer l'amélioration obtenue en augmentant le nombre de processeurs de n à m.

$$ie_m = \frac{(m-1) \cdot E[T_{m-1}]}{m \cdot E[T_m]}. \tag{2.3}$$

$$gie_{n,m} = \frac{n \cdot E[T_n]}{m \cdot E[T_m]}. \tag{2.4}$$

2.1.5.3 Autres mesures

Parmi les autres métriques utilisées pour mesurer les performances des algorithmes parallèles, on cite la "scaled speedup" (Accélération extensible) [9] (Formule 2.5) qui permet de mesurer l'utilisation de la mémoire disponible.

$$ss_m = \frac{Temps\ de\ résolution\ estimé\ pour\ un\ problème\ de\ taille\ n\ sur\ 1\ processeur}{Temps\ réel\ de\ résolution\ pour\ un\ problème\ de\ taille\ n\ sur\ m\ processeurs} \tag{2.5}$$

On cite aussi la "scaleup" (scalabilité) [9] (Formule 2.6) qui permet de mesurer l'aptitude du programme à augmenter sa performance lorsque le nombre de processeurs augmente.

$$su_{m,n} = \frac{Temps\ de\ résolution\ de\ k\ problèmes\ sur\ m\ processeurs}{Temps\ de\ résolution\ de\ nk\ problèmes\ sur\ nm\ processeurs} \tag{2.6}$$

Enfin, Karp et Flatt [88] proposent une autre métrique appelée la fraction séquentielle déterminée expérimentalement (Formule 2.7) et qui permet d'identifier les obstacles à de bonnes performances.

$$f_m = \frac{1/s_m - 1/m}{1 - 1/m} \tag{2.7}$$

Ces obstacles justement peuvent être de sources diverses et vont faire l'objet de la section suivante.

2.1.6 Facteurs influant sur la performance des algorithmes parallèles

La mesure de performances parallèles est une métrique complexe. Ceci est principalement dû au fait que les facteurs de performances parallèles sont dynamiques et distribués [100]. Le facteur communication est parmi les plus influents sur la performance de l'algorithme. Dans beaucoup de programmes parallèles, les tâches exécutées par les différents processeurs ont besoin d'accéder à des données communes (ex : multiplication matrice vecteur). Ceci crée un besoin de communication et freine la performance de l'algorithme. Ces communications sont d'autant plus importantes dans le cas où les processeurs auraient besoin de données générées par d'autres processeurs. Ces communications peuvent être minimisées en terme de volume de données et de fréquence d'échanges. Toutefois, le compromis à faire n'est pas toujours évident puisque les contraintes d'architecture entrent en compte. La taille des données échangées entre les processeurs peut également être limitée par des contraintes d'environnement. Par exemple, cette taille est fixée par défaut à 512Ko pour Mpich2 sur Infiniband [86].

Certains auteurs comme Grama *et al.* [73] expliquent les sources de ralentissement des algorithmes parallèles par les temps d'attente des processeurs. Une façon de combler ces temps d'attente est d'assigner des calculs indépendants aux processeurs inactifs.

Culler *et al.* [35] résument ces facteurs par la formule LogP (Latency, overhead, gap, Processor) qui identifie respectivement latence, surcharge, creux et processeurs. La latence est le délai dans le temps qui survient pendant la transmission d'un message d'un processeur à un autre. C'est la taille des données envoyées qui influence ce temps. Pour sa part, la surcharge est le temps requis pour l'initialisation d'un envoi ou d'une réception d'un message (hors données transmises) car, pendant ce temps, le processeur ne peut effectuer aucune autre opération. Ce paramètre est généralement fonction de l'architecture utilisée et peut être réduit en limitant la fréquence des échanges. Le creux est défini comme étant l'intervalle de temps minimum entre deux envois consécutifs ou deux réceptions consécutives sur un même processeur et représente une caractéristique du processeur (généralement un multiple du nombre de cycle par itération). Enfin, le nombre de processeurs impliqués est en lui-même un facteur de performance. Selon la taille du problème et la taille des données échangées, le nombre de processeurs va être plus ou bénéfique aux performances de l'algorithme.

Malony [100] stipule que la performance parallèle est difficile à mesurer, à identifier et à comprendre. Cependant, avec une approche scientifique basée sur l'hypothèse et l'expérimentation, on arrive à effectuer des améliorations. Cette approche est résumée dans la Figure 2.14.

2.1.7 Conclusion

Les notions qui ont été présentées dans ce chapitre parcourent de façon globale le parallélisme, la conception d'algorithmes, l'implémentation des programmes parallèles et les mesures de performances des programmes parallèles. Le parallélisme est un domaine très prometteur et a permis beaucoup d'avancées dans des secteurs exigeants en matière de calcul. Grâce à cette expansion et à la popularisation des machines parallèles, plusieurs domaines de l'informatique subissent des évolutions dans cette direction. Dans le domaine de la recherche opérationnelle, les métaheuristiques sont un type particulier d'algorithmes très intéressants à examiner du point de vue de la parallélisation. Dans la section suivante, nous allons définir plus spécifiquement leur fonctionnement.

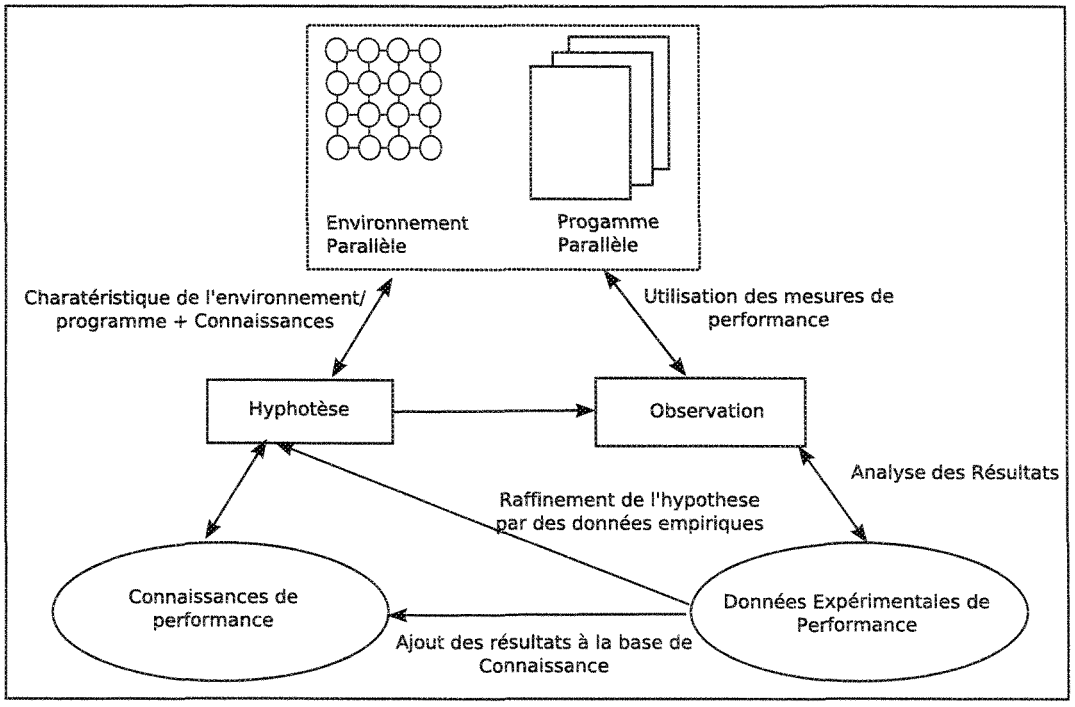

Figure 2.14 – Schéma général d'évaluation de performance parallèle [100]

2.2 Les métaheuristiques séquentielles et parallèles

2.2.1 Introduction

Trouver une solution à un problème d'optimisation combinatoire revient très souvent à concevoir un algorithme qui traite les données en entrée, effectue un traitement et produit une solution. L'algorithmique a cependant démontré que, pour certaines classes de problèmes, il n'existe pas d'algorithme capable de trouver la solution optimale. Une alternative consiste à utiliser une approche de solution de type métaheuristique (du grec meta : "au-delà", heuriskein : "trouver"). Dès lors, à défaut de trouver la solution optimale au problème, on effectue une démarche qui consiste à approximer cette solution. Les métaheuristiques sont utilisées dans les problèmes d'optimisation qui font partie de la classe de problèmes dits NP-Difficiles [62]. Dans cette section, les métaheuristiques les plus connues seront présentées ainsi que les versions parallèles issues de celles-ci.

2.2.2 Les algorithmes génétiques

2.2.2.1 Version séquentielle de l'algorithme génétique

Les algorithmes génétiques (AG) ont été introduits par les travaux de Holland [84] et Goldberg [69] et sont inspirés de la théorie de l'évolution et des sciences de la génétique. Ils appartiennent à la famille des algorithmes évolutionnaires [18]. Ils sont caractérisés par l'utilisation d'une population de structures multiples pour effectuer une recherche simultanée dans plusieurs zones de l'espace du problème. L'algorithme est guidé par un processus itératif qui, de génération en génération, conduit à une population plus adaptée. Le déroulement de ce processus est discuté plus en détails dans les sections qui suivent.

La métaheuristique débute son processus à partir d'un ensemble de solutions appelé Population où chacune de ces solutions est un Individu. La population de départ est généralement formée d'individus générés aléatoirement pour assurer la plus grande diversité possible. Un individu est, pour sa part, constitué d'une séquence de Gènes. Comme dans le processus de sélection naturelle, l'algorithme tente d'éliminer les mauvais traits des individus et de les faire évoluer afin d'en obtenir de meilleurs. Pour ce faire, l'algorithme comporte cinq phases principales : l'évaluation de la Fitness, la sélection, la reproduction, la mutation et le remplacement. Ces cinq étapes combinées font émerger de nouvelles générations à partir de la population courante. Ainsi, l'algorithme boucle sur ces quatre étapes et termine selon diverses conditions d'arrêt telles qu'une limite de temps, un nombre maximum de générations, un manque de diversité dans la population ou encore la stagnation de la fonction objectif depuis un certain nombre de générations. En résumé, la Figure 2.15 illustre un schéma d'un algorithme génétique de base. Dans cet exemple, une population P subit les opérateurs génétiques de l'algorithme au fil des générations notées T. On note par p un individu de cette population sur lequel sera appliqué l'opérateur de mutation.

L'évaluation de la Fitness est généralement représentée par une fonction f et est l'étape dans laquelle on mesure la qualité de chaque individu. Pour pouvoir décider de la qualité d'un individu et ainsi le comparer aux autres, il faut établir une mesure commune

```
Entrée  : Instance du problème
Sortie  : Une solution

T=0  ; Initialisation Population(T)  ; Evaluation(P(T))
Tant que (Condition d'arrêt non satisfaite)
        T=T+1
        Sélection(P(T))
        Reproduction(P(T))
        Mutation(p)
        Remplacement(P(T))
```

Figure 2.15 – Algorithme génétique de base [84]

d'évaluation. Par exemple, pour le problème du voyageur de commerce, on utilisera la distance totale comme fonction à minimiser tandis que dans le cas d'un problème d'ordonnancement, la minimisation du retard total pourrait être la fonction objectif.

La sélection consiste à choisir les individus les plus aptes à se reproduire. Il existe plusieurs mécanismes de sélection parmi lesquels : la sélection par tournoi, la sélection par roulette, la sélection par rang et la sélection déterministe. La sélection par tournoi consiste à choisir un nombre d'individus au hasard (taille du tournoi) et d'en sélectionner le meilleur selon la fonction objectif. Pour ce qui est de la roulette, on dispose au départ de l'ensemble des individus avec leurs "Fitness" respectives. Une probabilité est alors associée à chaque individu en fonction de sa "Fitness" et fera en sorte que les meilleurs individus ont plus de chances d'être choisis. Pour sa part, la sélection par rang trie les individus selon leurs "Fitness" et une nouvelle fonction f' leur est attribuée selon leur rang. Enfin, la sélection déterministe choisit directement les meilleurs individus d'une population. La sélection permet d'identifier un ensemble de couples qui pourront passer à l'étape de croisement.

Le croisement consiste à appliquer des procédures sur les individus sélectionnés pour donner naissance à un ou plusieurs (généralement deux) individus appelés "enfants", dont le code génétique est une combinaison de celui des parents. Les procédures d'échanges sont guidées par des opérateurs de croisements. Ces opérateurs sont un facteur très important pour la qualité des individus engendrés. Plusieurs études ont été conduites sur les opérateurs de croisement et démontrent que le choix doit se faire en fonction de la nature du problème [123] [83]. Dans les paragraphes qui suivent, les opérateurs ont été classés selon les caractéristiques qu'ils tentent de préserver. On peut ainsi distinguer trois catégories d'opérateurs [107] : ceux qui préservent

la position absolue, ceux qui préservent l'ordre et ceux qui préservent l'adjacence (les arcs). À noter que les opérateurs cités s'appliquent lorsque la codification des gènes est telle que chaque gène est unique comme c'est le cas dans le problème du voyageur de commerce. On s'intéresse à cette codification et à ces opérateurs en particulier parce qu'ils s'appliquent au problème traité dans le chapitre suivant.

Parmi les opérateurs qui préservent la position absolue, on note :

- PMX : Partially Mapped Crossover [68]

  PMX est un opérateur de croisement à deux points de coupure qui définit un segment de même longueur dans chacun des parents *P1* et *P2*. Les segments sont indiqués avec une trame foncée dans Figure 2.16 partie (a). Ces segments sont copiés chez les enfants opposés *E1* et E2. Par exemple, *E1* a hérité du segment de *P2* (b). Avec les gènes de ces segments, on établit une liste de correspondance. Cette liste va servir à placer les gènes redondants et elle est formée de la manière suivante : pour chaque position du segment on note x le gène qui s'y trouve et y celui de l'autre enfant dans la même position. Tant que y est retrouvé ailleurs dans le segment de départ, on note y' son correspondant dans l'autre enfant et on remplace y par y'. Par exemple, le gène correspondant à "1" de *E1* est "6" mais ce gène existe aussi dans *E1* et son correspondant est "3". Ainsi dans la liste de correspondance, on note que "1" a pour correspondant "3". La liste complètement formée se trouve à la marge de la partie (b) du schéma. On procède ensuite au placement des gènes hors segment en les copiant des parents respectifs. Par exemple, copier "1" de *P1* dans *E1* provoque un conflit puisque que ce gène existe déjà. On utilise alors son correspondant dans la liste qui est "3". En procédant de manière itérative, on arrive à former les enfants *E1* et *E2* illustrés à la partie (c) de la Figure 2.16.

- CX : Cycle Crossover [115]

  CX est un opérateur qui satisfait la condition suivante : chaque gène d'un enfant provient de l'un des parents à la même position. Les enfants sont donc formés en copiant un gène d'un parent et en éliminant l'autre à la même position puisqu'il va

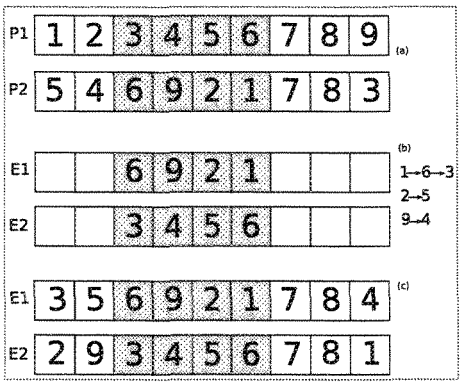

Figure 2.16 – Opérateur de croisement PMX [68]

appartenir au deuxième enfant. Une fois que les positions occupées sont copiées par élimination, on a complété un cycle. Les places restantes des deux enfants sont complétées par les parents opposés. Dans l'exemple présenté à la Figure 2.17 partie (a), la première position de *E1* est attribuée à "1" provenant de *P1*. Le gène de *P2* situé à la même position "4" est recherché dans *P1* et se retrouve se trouve à la quatrième position. Le gène "4" est copié dans *E1* et son correspondant "8" est recherché dans *P1* et retrouvé à la huitième position. "8" est copié dans *E1* à cette même position et son correspondant "3" est recherché dans *P1* et retrouvé à la troisième position. "3" est copié dans *E1* et son correspondant "2" est recherché dans *P1* et retrouvé à la deuxième position. "2" est copié et son correspondant "1" est retrouvé dans *P1* à la première position or cette position est déjà occupée dans *E1* donc le cycle est terminé. Le cycle de *E1* est "1,4,8,3,2" et de manière analogue, le cycle "4,1,2,3,8" est formé pour E2. En dernier lieu, il faut combler les positions vacantes à partir des parents opposés. Par exemple, les gènes "7,6,9" de *E1* proviennent de *P2*. On obtient ainsi les enfants illustrés dans la partie (b) de la Figure 2.17.

- Échange de séquence (Subtour Exchange) [155]

  Cet opérateur sélectionne des sous-ensembles non identiques contenant des gènes communs aux deux parents et les échange. Comme illustré à la Figure 2.18, les sous-ensembles "4567" et "5647" respectivement de *P1* et de *P2* sont sélectionnés dans la partie (a). Dans la partie (b), ces sous-ensembles sont permutés pour

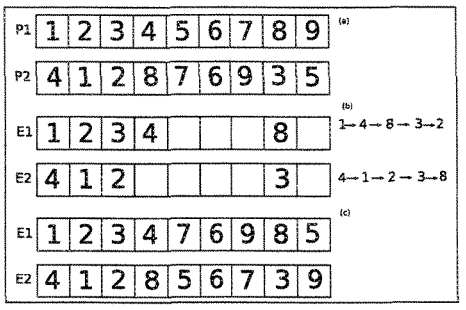

Figure 2.17 – Opérateur de croisement CX [115]

former les enfants *E1* et E2. Comme les deux sous-ensembles sont de même taille et contiennent les mêmes gènes, il ne peut pas y avoir de conflit après l'échange.

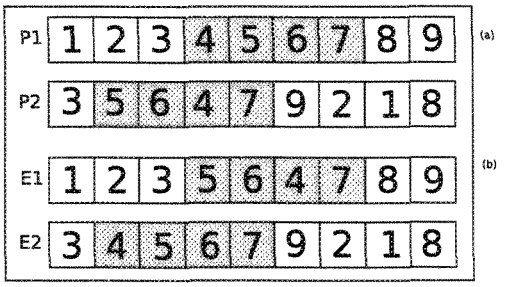

Figure 2.18 – Opérateur échange de séquence [155]

- Croisement uniforme de permutation [39]

  Le croisement uniforme de permutation génère un masque binaire aléatoire. C'est une séquence binaire de même longueur que l'individu telle que les gènes marqués "0" sont conservés dans un des parents et ceux marqués "1" dans l'autre. Les gènes non marqués du premier parent sont permutés selon l'ordre dans lequel ils apparaissent dans le deuxième parent (et réciproquement pour le parent 2). L'exemple de la Figure 2.19 présente un masque à neuf éléments dans la partie (a). Ce masque appliqué aux parents *P1* et *P2* donne la sélection représentée dans la partie (b). En dernier lieu, on procède au réarrangement des gènes sélectionnés. Par exemple, les gènes "3-4-6-8" sont marqués dans *P1* et apparaissent dans *P2* dans l'ordre suivant : "8-6-4-3". C'est donc dans cet ordre qu'ils seront réarrangés dans *E1* comme le montre la partie (c) de la Figure 2.19.

Parmi les opérateurs qui préservent l'ordre, on note :

- OX : Order Crossover [38] [115]

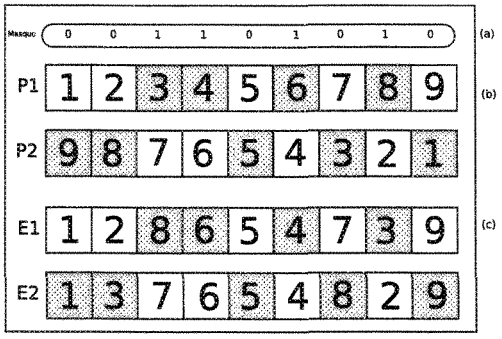

Figure 2.19 – Opérateur uniforme de permutation [39]

OX est un opérateur avec deux points de coupure qui copie le segment formé par ces deux points de coupure dans les deux enfants. Par la suite, il recopie, à partir du deuxième point de coupure ce qui reste des gènes dans l'ordre du parent opposé en évitant les doublons. La Figure 2.20 présente un exemple du déroulement du croisement avec l'opérateur OX. Dans la partie (a), un segment est formé par les points de coupure dans les deux parents et ces segments sont copiés tels quels dans les enfants *E1* et *E2* comme le montre la partie (b). Enfin, on procède à la copie des gènes situés hors du segment copié. Pour cela, on se place à partir du deuxième point de coupure et on choisit les gènes non redondants provenant du parent opposé. Par exemple, dans la partie (c) de la Figure 2.20, on essaie de placer le gène "6" de *P2* après le deuxième point de coupure dans *E1* mais ce gène existe déjà à l'intérieur du segment. Il est donc ignoré et on passe au suivant. Le gène "2" ne présente pas de conflit, il est donc copié et ainsi de suite jusqu'à former les deux enfants *E1* et *E2* tel qu'illustré à la Figure 2.20 partie (c).

- LOX : Low Order Crossover [115]

  LOX est un opérateur semblable à OX à la différence que la distribution hors segment se fait à partir du début du parent opposé et non pas à partir du deuxième point de coupure. Comme illustré à la Figure 2.21 partie (a), un segment est formé par deux points de coupure chez les deux parents *P1* et *P2* et ces segments sont copiés respectivement dans *E1* et *E2* comme montré à la Figure 2.21 partie (b). Enfin, dans la partie (c), les gènes hors segment sont copiés de manière similaire à

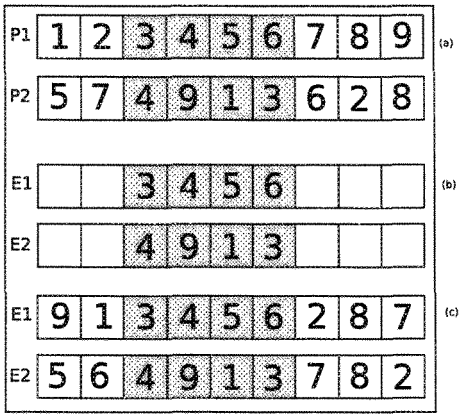

Figure 2.20 – Opérateur de croisement OX [38] [115]

l'opérateur OX, à la différence que le point de départ est la première position du parent opposé. Par exemple, on commence au deuxième point de coupure de *E1* et on copie le premier gène de *P2*. Le gène "5" pose conflit et est ignoré pour passer au suivant. Le deuxième gène, "7", ne pose pas de conflit et il est alors copié. On procède ainsi jusqu'à former les enfants *E1* et *E2* tel qu'illustré à la partie (c) de la Figure 2.21.

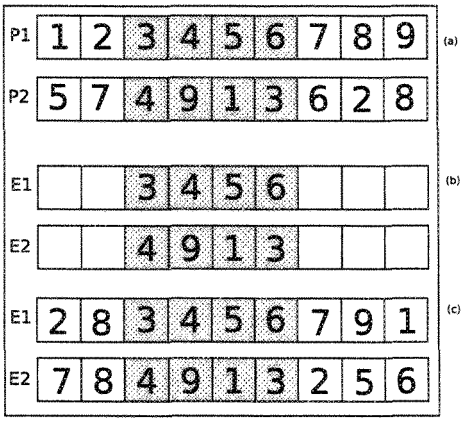

Figure 2.21 – Opérateur de croisement LOX [115]

– OBX : Order Based Crossover [142]

OBX est un opérateur qui s'intéresse à l'ordre relatif des gènes dans les parents. Un sous-ensemble de gènes est sélectionné dans le premier parent. L'ordre des gènes de ce sous-ensemble va être imposé au premier enfant. Les positions auxquelles ces gènes vont être affectés sont retrouvées à partir du deuxième parent. La Figure 2.22 illustre le fonctionnement de l'opérateur OBX. Dans la partie (a), un segment

est formé en sélectionnant un sous-ensemble chez les deux parents *P1* et *P2*. Les gènes de ces segments sont copiés respectivement dans *E1* et *E2* dans les positions auxquelles ils apparaissent dans le parent opposé. Par exemple, la partie (b) de la Figure 2.22 montre la transformation subie par les gènes sélectionnés dans la partie (a). En effet, les gènes sélectionnés de *P1* sont "3-4-5-6" et apparaissent dans *P2* aux positions 1,3,6 et 7. Ce sont donc dans ces positions qu'on va les retrouver dans E1. La même démarche est suivie pour E2. Enfin, les positions vacantes sont remplies à partir des parents opposés. Comme le montre la partie (c) de la Figure 2.22, les positions vacantes de *E1* proviennent de *P2* et inversement pour E2.

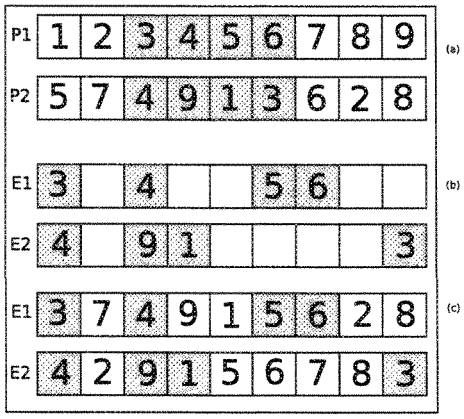

Figure 2.22 – Opérateur de croisement OBX

- PBX : Position Based Crossover [142]

Dans cet opérateur, les enfants sont formés à partir d'un sous-ensemble de positions issues d'un parent et les gènes correspondants à ces positions de l'autre parent. Les autres positions sont remplies avec les gènes restants dans le même ordre relatif que le deuxième parent. Le fonctionnement de l'opérateur PBX est illustré à la Figure 2.23. Dans la partie (a), des positions sont choisies au hasard dans les deux parents *P1* et *P2* et ces positions seront copiées respectivement dans les enfants *E1* et *E2* comme illustré dans la partie (b) de la Figure. Enfin, les positions vacantes sont remplies selon l'ordre relatif du parent inverse. Dans E1, il reste à placer "2,4,6,7,9" et ils apparaissent dans *P2* dans l'ordre suivant "7-4-9-6-2". C'est donc dans cet ordre qu'ils sont placés dans E1. Le même procédé est suivi pour *E2* et les enfants

sont formés comme le montre la partie (c) de la Figure 2.23.

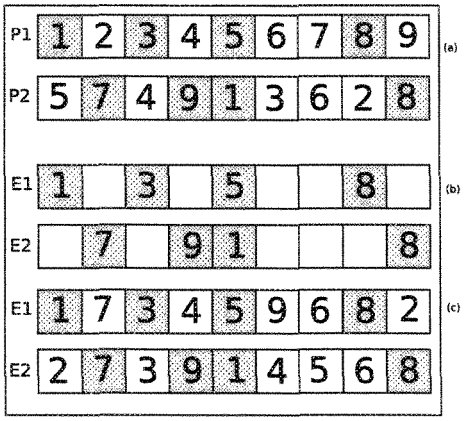

Figure 2.23 – Opérateur de croisement PBX

Parmi les opérateurs qui préservent l'adjacence, on note :

- Croisement par recombinaison d'adjacences (Edge Recombinaison Crossover) [154]

  Cet opérateur tente de faire hériter des adjacences existant dans les deux parents. Il utilise une structure de données spéciale appelée "carte d'adjacence" (edge map). Cette carte évolue à chaque fois qu'un gène est sélectionné pour renseigner sur les adjacences à chaque étape. La Figure 2.24 montre le fonctionnement de l'opérateur par recombinaison d'adjacences. La partie (a) montre la liste des adjacences qu'on peut extraire de la représentation des parents *P1* et *P2*. L'enfant E1, représenté dans la partie (b), est formé en choisissant une première position de l'un des parents (celle de *P1* a été choisie dans l'exemple). Le gène "1" est éliminé de la liste d'adjacence et on choisit un de ses adjacents qui possède le minimum de voisins. Dans l'exemple, c'est le "7" qui est choisi et on réitère le procédé jusqu'à former l'enfant *E1* en entier.

- Croisement heuristique (Heuristic Crossover) [75]

  Cet opérateur est considéré comme une classe à part entière. Il fonctionne en sélectionnant un gène au hasard et quatre de ses adjacences (deux de chaque parent). Il compare ainsi les quatre adjacences et sélectionne la plus courte (si la matrice des distances est connue dans le probleme) ou la moins coûteuse (en recalculant la fonction objectif). L'opérateur boucle sur cette étape jusqu'à ce que l'enfant soit entièrement formé. Dans l'exemple de la Figure 2.25, le gène "3" est choisi de manière

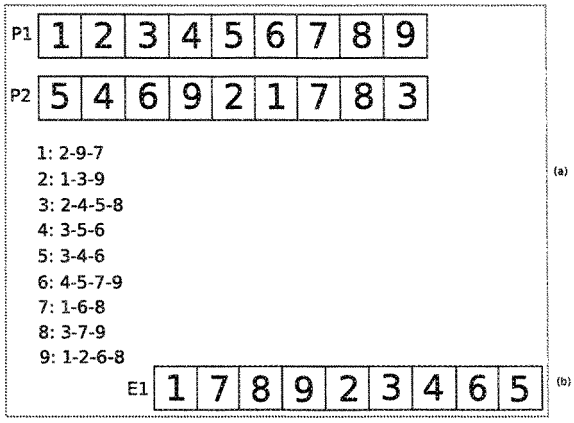

Figure 2.24 – Opérateur de croisement par recombinaison d'adjacences

aléatoire et tous ses voisins dans les deux parents *P1* et *P2* sont comparés. Le choix du gène suivant repose sur les coûts de passage ou encore sur la fonction objectif.

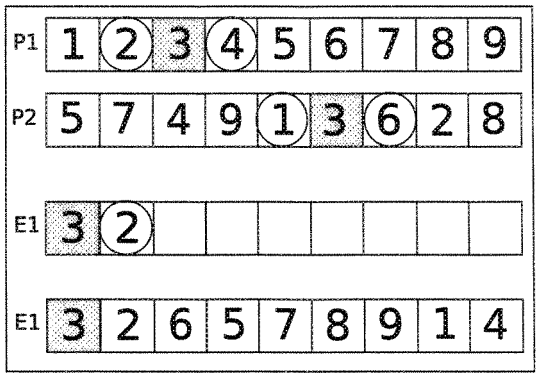

Figure 2.25 – Opérateur de croisement heuristique

Le troisième opérateur génétique d'un algorithme génétique est la mutation. C'est un processus où un changement mineur de code génétique est appliqué à un individu pour introduire de la diversité et ainsi éviter de tomber dans des optimums locaux. Le paramètre associé à cet opérateur est la probabilité de mutation qui exprime en pourcentage, les chances pour que la mutation soit déclenchée. La mutation possède également des opérateurs dont on peut citer les suivants :

- Mutation par inversion : Deux positions sont sélectionnées au hasard et tous les gènes situés entre ces positions sont inversés. La Figure 2.26 montre un exemple de mutation par inversion. L'individu i avant mutation est représenté dans la partie (a) avec le segment formé par les deux positions sélectionnées. L'inversion est effectuée

à l'étape (b) afin de produire l'individu i'.

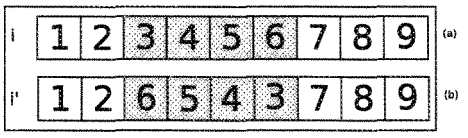

Figure 2.26 – Mutation par inversion

- Mutation par insertion : Deux positions sont sélectionnées au hasard et le gène appartenant à l'une est inséré à l'autre position. Dans la Figure 2.27partie (a), les gènes "3" et "6" de l'individu i sont sélectionnés. La deuxième étape, illustrée par la partie (b), montre l'insertion de "6" avant "3" et le décalage de tous les autres gènes.

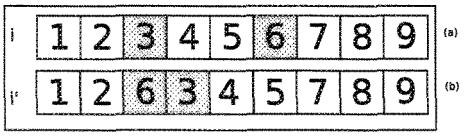

Figure 2.27 – Mutation par insertion

- Mutation par déplacement : Une séquence est sélectionnée au hasard et déplacée vers une position elle-même tirée au hasard. Un exemple de mutation par déplacement est illustré à la Figure 2.28. Dans la partie (a), la séquence "3-4-5-6" est sélectionnée et déplacée à la dernière position pour former l'individu i' représenté dans la partie (b).

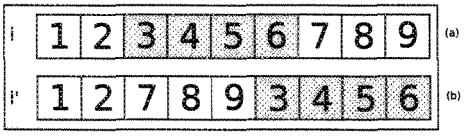

Figure 2.28 – Mutation par déplacement

- Mutation par permutation : Deux positions sont sélectionnées au hasard et les gènes qui s'y trouvent sont permutés. Comme illustré à la Figure 2.29, les éléments "3" et "6" sont sélectionnés dans i (partie (a)) et permutés dans i' (partie (b)).

Certains auteurs tels que Rubin et Ragatz [134] ont implémenté une version particulière de la mutation contenant de la recherche locale. La recherche locale se base sur le concept de voisinage. Ce mécanisme remplace la solution actuelle par une meilleure dans son voisinage immédiat si celle-ci existe. Appliquée à la mutation, la recherche locale va permuter les gènes voisins de l'individu en tentant de l'améliorer.

Dans le même esprit, on peut citer la mutation heuristique (Heuristic Mutation) qui

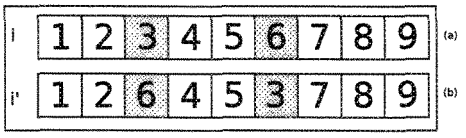

Figure 2.29 – Mutation par permutation

est schématisée à la Figure 2.30. Cette mutation consiste à sélectionner un nombre aléatoire de positions, faire toutes les permutations possibles des positions sélectionnées, évaluer les solutions et en sélectionner la meilleure. Ces deux dernières versions de la mutation sont très

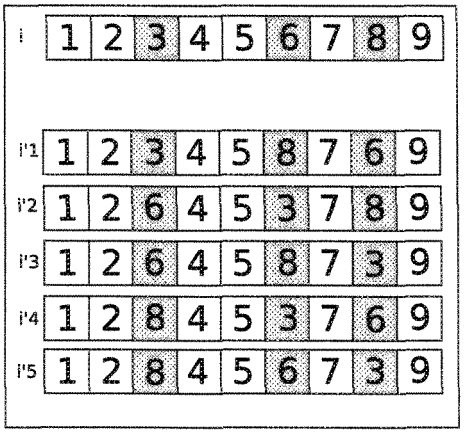

Figure 2.30 – Mutation heuristique

coûteuses en terme de temps d'exécution et ne sont pas fidèles au concept génétique de la mutation puisqu'elle est supposée être aléatoire et minime.

Enfin, le dernier opérateur génétique décrit dans l'algorithme est le remplacement. Ce mécanisme permet la recombinaison entre les nouveaux individus créés et ceux qui se trouvent présentement dans la population. Plusieurs stratégies de remplacement existent dans la littérature [63] parmi lesquelles on cite : l'élitisme, le remplacement complet, le remplacement partiel et le remplacement aléatoire.

L'élitisme consiste à recombiner les parents et les enfants en éliminant ceux qui donnent les moins bons résultats selon la fonction objectif. Le remplacement complet fait en sorte que la nouvelle population soit uniquement constituée d'enfants. Ceci peut se faire à condition que le nombre d'enfants soit égal à la taille de population. Le remplacement partiel quant à lui, reconstruit la nouvelle population avec une proportion d'individus issue des parents et une autre issue des enfants. Enfin, le remplacement aléatoire sélectionne au hasard les

individus à conserver sans pour autant éliminer les meilleurs individus.

Les algorithmes génétiques ont été appliqués avec succès pour la résolution de plusieurs problèmes d'optimisation combinatoire. Une revue de ces problèmes a été faite par Michalewicz [107].

2.2.2.2 Version parallèle de l'algorithme génétique

Les algorithmes génétiques s'adaptent très bien par leur structure à l'exécution en parallèle. Avec la popularisation des machines parallèles, des versions parallèles de l'algorithme génétique ont vu le jour et ont fait l'objet de plusieurs études [29]. Luque *et al.* [98] ont retracé les importants travaux qui ont marqué le domaine des algorithmes génétiques parallèles. Ceux-ci sont résumés dans le Tableau 2.1.

Algorithme	Référence	Description
ASPARAGOS	[72]	AG hautement parallèle. Applique le "hill-climbing" si il n'y a pas d'amélioration
DGA	[148]	AG à population distribuée
GENITOR II	[153]	Modèle en îlot
ECO - GA	[37]	AG hautement parallèle
PGA	[113]	AG en îlot avec migration du meilleur individu et "hill climbing" local
SGA-Cube	[50]	Modèle en îlot implémenté sur nCUBE 2
EnGENEer	[133]	Parallélisation globale
PARAGENESIS	[138]	Modèle en îlot implémenté pour le CM-200
GAME	[138]	Ensemble d'outils orientés objet pour programmation générale
PEGAsuS	[132]	Modèle permettant une programmation de haut niveau sur des machines de type MIMD
DGENESIS	[103]	Modèle en îlot avec migration
GAMAS	[122]	Modèle en îlot avec 4 noeud
iiGA	[95]	AG en îlot avec injection d'individus, hétérogène et asynchrone
PGAPack	[94]	Parallélisation globale de la fonction objectif
CoPDEB	[2]	Modèle en îlot avec un opérateur différent par noeud
GALOPPS	[71]	Modèle en îlot
MARS	[144]	Environnement parallèle avec tolérance
RPL2	[126]	Modèle en îlot très flexible, permettant de définir de nouveaux modèles
GDGA	[82]	Modèle en îlot ayant une topologie en hypercube
DREAM	[15]	Framework pour algorithmes évolutionnaires distribués
MALLBA	[7]	Framework pour algorithmes parallèles
Hy4	[8]	Modèle en îlot hétérogène ayant une topologie en hypercube
ParadisEO	[26]	Framework pour algorithmes parallèles

Tableau 2.1 – Travaux en algorithmes génétiques parallèles [98]

Parmi les travaux les plus récents sur les algorithmes génétiques parallèles, on note ceux de Guo *et al.* [77] qui ont utilisé cette métaheuristique pour la résolution du problème de "conduction de chaleur inverse" (Inverse Heat Conduction) avec des résultats très encourageants.

On cite également Yeh *et al.* [156] qui ont proposé deux algorithmes génétiques parallèles pour la gestion de configuration de produit (Product Configuration Management).

Après avoir présenté sommairement quelques travaux dans le domaine des algorithmes génétiques parallèles, on va aborder plus en détails les caractéristiques de ces algorithmes, à savoir les modèles et les topologies.

Il existe principalement deux modèles d'algorithmes génétiques parallèles : le modèle maître-esclave et le modèle en îlot. Dans le premier modèle, un processeur est dédié aux étapes de sélection, de croisement, de mutation et de remplacement et le reste des processeurs s'occupe de calculer la fitness des individus. Ce modèle possède plusieurs avantages dont la facilité d'implémentation et d'exploration du même espace de recherche que la version séquentielle et ce, plus rapidement. Parmi les principaux travaux sur ce modèle, on peut citer Bethke [22] et Grefenstette [74] qui en ont vanté les mérites. D'autres recherches ont identifié plus tard des problèmes d'extensibilité [54], [80] et [1]. En effet, plus le nombre de processeurs utilisé est grand, plus l'efficacité de l'algorithme diminue à cause de la surcharge de communication.

Ensuite, le modèle en îlots ou encore modèle distribué, est le plus populaire parmi les algorithmes génétiques parallèles. Ce modèle a été initialement introduit par les travaux de Grosso [76]. L'objectif de cette recherche était d'étudier l'interaction entre plusieurs sous-composantes parallèles d'entités évolutives. Grosso a constaté que les solutions trouvées par les îlots isolés étaient de loin moins bonnes que celles atteintes avec une seule grande population. Cependant, lorsque les migrations sont fréquentes, la qualité de solutions est équivalente dans les deux versions. Ceci amène une caractéristique très importante du modèle en îlot : la migration. En effet, pour faciliter le processus de sélection et pour donner de meilleurs résultats, des individus sélectionnés parmi une sous-population peuvent être déplacés vers une autre. Le choix des individus candidats à la migration peut être paramétrable mais, très souvent, il s'agit de faire écouler une période de temps pendant laquelle aucune migration n'est permise pour ensuite choisir un individu (le meilleur, le pire ou aléatoire) et le substituer avec un autre (le meilleur, le pire ou aléatoire) d'une population voisine. La migration est régie par deux paramètres : l'intervalle de migration et le taux de migration. L'intervalle de migration

est la durée après laquelle on permet les échanges entre les sous-populations. Plus il est faible, plus les migrations sont fréquentes. Dans les travaux de Petty *et al.* [119], une copie du meilleur individu est envoyée à toutes les sous-populations après chaque génération dans le but d'assurer un bon mélange des individus. Les conclusions de cet essai sont qu'un tel niveau élevé de communication donne des résultas de même qualité qu'un algorithme génétique séquentiel avec une population sans restriction de croisement. Le deuxième paramètre à considérer dans la migration est le taux. C'est le pourcentage de la population qui sera copiée vers la sous-population de destination. Les recherches de Grosso [76] ont montré qu'il existait un taux de migration critique au-dessous duquel les performances de l'algorithme sont freinées à cause de l'isolation des sous-populations, et au delà duquel les solutions trouvées sont de même qualité que la version séquentielle.

La deuxième caractéristique des algorithmes génétiques parallèles qui va être abordée est la topologie. Il s'agit de la manière par laquelle les îlots sont reliés et ceci a une grande influence sur la propagation des bons individus. En effet, dans le cas extrême où aucune communication entre les îlots n'est permise, les sous-populations n'ont aucune influence les unes sur les autres. Dans le cas contraire où un maximum de connexions est établi, on converge trop vite vers un minimum local et la diversité est trop faible pour permettre à une bonne solution d'émerger.

Plusieurs topologies ont été identifiées et testées [28]. L'efficacité d'une topologie revient à analyser le graphe qu'elle forme. Si l'algorithme n'est exécuté que sur deux sous-populations, la topologie n'a aucune influence vu que seuls les voisins directs importent. Cependant, si plus que deux sous-populations sont sollicitées, le facteur connectivité du graphe intervient. En effet, Cantù-Paz [30] a mis en évidence une relation directe entre la connectivité du graphe formé par ces sous-populations et la qualité des solutions de l'algorithme.

Les algorithmes génétiques sont le thème principal de ce mémoire. Néanmoins, d'autres métaheuristiques séquentielles et parallèles ont été développées et vont être citées dans la section qui suit.