MCours.com

2.2.3 Autres métaheuristiques

2.2.3.1 Optimisation par colonie de fourmis

L'optimisation par colonie de fourmis (OCF) [44] est une approche métaheuristique issue de l'étude de comportement des fourmis. L'analogie vient du fait que pendant l'approvisionnement de la colonie, les fourmis laissent des traces de phéromone qui marquent leur passage sur un chemin donné. La phéromone est une hormone très éphémère et reconnaissable entre fourmis d'une même colonie. Quand la nourriture est trouvée, le chemin qui y a mené sera plus imbibé en phéromone que les autres. Progressivement, toutes les fourmis seront au courant de ce chemin et tenteront de le rendre encore plus court.

Dans un algorithme d'OCF, un problème donné est ramené à un graphe. Des fourmis virtuelles parcourent ce graphe et effectuent un traitement probabiliste sur une solution donnée. Le modèle probabiliste utilisé est appelé Modèle Phéromone (Pheromone Model) et consiste en un ensemble de paramètre dont les valeurs sont appelées Valeurs Phéromones (Pheromone Values) [46]. En utilisant ces valeurs, la métaheuristique construit un nouveau segment et évalue la modification par rapport à ce qui a été fait précédemment. Si cette modification est bonne, la trace laissée va être de plus en plus imbibée, attirant de ce fait d'autres fourmis qui vont à leur tour l'étudier. La trace abandonnée va être de moins en moins attirante pour les fourmis et va progressivement disparaître.

Les versions de l'OCF se distinguent par la manière dont les valeurs phéromones sont mises à jour et les principales sont Ant Colony System (ACS) [47] [48] et Max-Min Ant System (MMAS) [140]. Une revue complète des algorithmes OCF [45] ainsi que leurs applications [44] a été faite par Dorigo. Le schéma de base d'un algorithme d'optimisation par colonie de fourmis est illustré à la Figure 2.31. L'algorithme commence par initialiser sa matrice de phéromones avant de débuter dans une première boucle pour un nombre cycle défini par la variable *NbCycles*. Une deuxième boucle est imbriquée à celle-ci pour générer les fourmis de la colonie. Une fourmi est représentée par la variable k et le nombre total de fourmis par *NbFourmis*. Chaque fourmi k construit sa solution s en mettant à jour sa trace

de phéromone locale et évalue ensuite la Fitness F lié à s. Une fois toutes les fourmis créés, l'algorithme identifie la meilleure solution du cycle s^{bc} avec sa valeur F^{bc} et utilise ces valeurs pour une mise à jour globale de la phéromone et de la meilleure solution connue s^{bg} ainsi que sa valeur F^{bg}.

```
Initialiser Phéromone
Pour (Nombre de cycles de 1 à NbCycles)
    Pour (k=1 jusqu'à NbFourmis)
        Construire une solution s
        Mise à jour locale de la Phéromone
        Evaluation de F pour s
    Définir la meilleure solution du cycle courant s^bc et sa valeur F^bc
    Mise à jour globale de la Phéromone avec s^bc
    Mise à jour de la meilleure solution connue s^bg et sa valeur F^bg
```

Figure 2.31 – Optimisation par colonies de fourmis ACS [59]

La nature de l'algorithme d'OCF rend sa parallélisation naturelle, que ce soit dans les données ou dans le domaine. Plusieurs modèles d'OCF parallèles peuvent être retrouvés dans la littérature, nous citons les plus marquants.

Stutzle [139] a abordé la parallélisation de l'OCF avec des exécutions indépendantes. Pour leur part, Michel *et al.* [108], Mendes *et al.* [104] et Middendorf [109] ont proposé des modèles distribués (en îles) avec diverses stratégies d'échanges entre colonies. D'autre auteurs comme Talbi *et al.* [145] et Rahoual *et al.* [129] ont implémenté des modèles maître-esclave où le processeur maître distribue les tâches aux esclaves.

Plus récemment, les bibliothèques standards de programmation parallèle ont été utilisées pour l'implémentation de modèles OCF parallèles. Nous citons Randall *et al.* [130] et Manfrin *et al.* [101] qui ont proposé une parallélisation de l'OCF en ayant recours au standard MPI. Delisle *et al.* [42] ont fait de même en utilisant OpenMP.

2.2.3.2 Recuit simulé

Le recuit simulé (RS) est une métaheuristique connue pour être la plus ancienne [24]. Elle a été proposée par Kirkpatrick en 1983 [89] d'après le travail de Metropolis [106] d'où elle puise ses origines statistiques. Le terme recuit est inspiré d'un processus utilisé en métallurgie dans lequel on fait alterner les cycles de chauffage et de refroidissement des métaux pour

minimiser l'énergie des matériaux. Cette métaheuristique utilise une approche de Monte Carlo [24] pour simuler le comportement d'un système qui tend à atteindre un équilibre thermal et ainsi devenir stable. Cette analogie est utilisée pour résoudre les problèmes d'optimisation combinatoire. Il a été prouvé qu'en surveillant de manière précise le taux de refroidissement, l'algorithme est capable de trouver l'optimum global mais paradoxalement, ceci prendrait un temps infini. C'est pour cela que d'autres versions du recuit simulé, le Fast annealing et le Very Fast Simulated Reannealing (VFSR) qui sont exponentiellement plus rapides que la version de base, sont utilisées pour surmonter le problème des délais. Le recuit simulé possède un sérieux avantage par rapport aux autres méthodes par le fait qu'il ne se fait par piéger dans les minima locaux [25].

Le recuit simulé est cependant complexe à paramétrer. Pour ce qui est des problèmes NP-Difficiles, celui qui a été le plus étudié est sûrement celui du voyageur de commerce. Le fonctionnement d'un algorithme de recuit simulé de base est énoncé dans la Figure 2.32. Dans cet algorithme, une solution est notée s et est initialement générée de manière aléatoire grâce à la fonction *Generer_Solution_Aleatoire*. La température T est aussi initialisée à une valeur initiale T_0. L'algorithme explore par la suite le voisinage immédiat de la solution noté N et compare les deux valeurs s et s' selon la fonction objectif f. Si s' est une meilleure solution que s, elle la remplace. Sinon, la nouvelle solution est acceptée selon une fonction de probabilité notée $p(T,s',s)$. Enfin l'algorithme met à jour la valeur de la température de l'itération en cours.

```
s= Generer_Solution_Initiale()
T=T0
Tant que (condition d'arrêt non satisfaite)
        s'=Selection_Aleatoire(N(s))
        si f(s')<f(s)
                s=s'
        sinon
                Accepter s' en tant que nouvelle solution avec une probabilité p(T,s',s)
        Mise_A_jour(T)
```

Figure 2.32 – Algorithme de recuit simulé [89]

Cette métaheuristique est l'une des premières basées sur la recherche locale à être parallélisée. Les différents modèles théoriques de parallélisation ont été largement couvert par

Azencott [17]. D'un autre coté, Aydin *et al.* [16] ont présenté une revue des travaux les plus marquants dans le domaine du Recuit simulé parallèle. Ainsi, nous allons citer les travaux qui sont apparus après cette revue de la littérature. Les auteurs cités précédemment ont identifié des versions classiques de recuit simulé parallèle. Les travaux qui ont suivi ont pris une autre direction en essayant de faire des versions hybrides. A titre d'exemple, Debudaj-Grabysz *et al.* [41] ont fait combiner MPI et OpenMP pour mettre en place une nouvelle version de recuit simulé parallèle avec un modèle hybride de communication. D'autre part, Wang *et al.* [152] ont fait une version hybride en combinant les algorithmes génétiques avec le recuit simulé, cette version est appelée Recuit Simulé Génétique Parallèle (Parallel Genetic Simulated Annealing PGSA).

Dernièrement, Chen *et al.* [32] ont étudié quatre approches différentes de parallélisation du recuit simulé dont deux en y incorporant des algorithmes génétiques. Ils ont conclu que l'hybridation par les algorithmes génétiques est une voie prometteuse pour les versions parallèles du recuit simulé.

2.2.3.3 GRASP

Le GRASP (Greedy Randomized Adaptive Search Procedure) [52] [120] est une métaheuristique qui consiste en une suite de solutions construites par une approche vorace et à leur optimisation par l'exploration de leurs voisinages respectifs.

Chaque itération de cette métaheuristique consiste en une phase de construction de solution et en une exploration de voisinage. La construction se fait élément par élément, le choix du prochain élément se faisant par une approche vorace. GRASP essaie de tirer à la fois avantage de l'approche vorace et de l'approche aléatoire. GRASP garde une trace de la meilleure solution et la retourne à la fin de l'algorithme. Cette heuristique est appelée dynamique, en opposition aux autres heuristiques statiques qui assignent un score aux éléments seulement avant la construction. Par exemple, pour le problème du voyageur de commerce qui est basé sur les coûts des arcs dans le graphe, plus l'arc est court, plus son score est haut. La deuxième phase de l'algorithme est une recherche locale qui peut être basique ou encore faire appel à des techniques plus avancées telles que le recuit simulé vu à la Section

2.2.3.2. Le GRASP a l'avantage d'être facile à implémenter avec très peu de paramètres et d'avoir plusieurs applications possibles. La Figure 2.33 résume le fonctionnement du GRASP. L'algorithme de GRASP initialise la meilleure solution s^* à l'infini. Il effectue par la suite une construction vorace sur la solution en cours et trouve un minimum local dans le voisinage noté N. Si cette solution est meilleure que la solution en cours selon la fonction f elle est gardée.

```
s* = ∞
Tant que (Condition d'arrêt non satisfaite)
        Construire_Solution_Vorace(s)
        Trouver le minimum local s' dans N(s)
        si f(s')<f(s*)
                s*=s'
Retourner meilleure solution de s*
```

Figure 2.33 – Algorithme GRASP [52]

Deux stratégies de parallélisation de GRASP ont été identifiées par Alvim [40]. La première consiste à décomposer l'espace de recherche en régions et à appliquer GRASP à chacune d'elle en parallèle. La deuxième consiste à partitionner les itérations et à les distribuer parmi les processeurs. La majorité des algorithmes qui utilisent ces deux stratégies tombent dans la catégorie Marche Multiple Thread Indépendant (Multiple Walk Independent Thread). Parmi les travaux effectués en adoptant cette stratégie, on cite Padalos *et al.* [117], Martins *et al.* [102]. L'apparition plus tard de versions hybrides de GRASP parallèles avec "Path Relinking" a fait naître une nouvelle catégorie appelée Marche Multiple Thread Coopératif (Multiple Walk Cooperative Thread). Aiex *et al.* [5] [6] ont utilisé cette technique pour la résolution du problème d'ordonnancement job shop et du "Three Index Assignement Problem" avec le standard MPI. Les versions hybrides gagnent de plus en plus en popularité comme le montre les travaux les plus récents de Ribeiro *et al.* [131].

2.2.3.4 Recherche avec Tabous

Parmi les autres métaheuristiques dites "classiques", la recherche avec tabous (RT) a été proposée en 1986 par Fred Glover [65]. Le principe de cette méthode est le suivant : à solution donnée, on explore les solutions qui lui sont proches, c'est-à-dire celles qu'on peut atteindre par de simples modifications de la solution initiale. Cet ensemble de solutions est appelé voisinage. Une particularité est toutefois de garder en mémoire, sous forme d'une liste

taboue, les espaces déjà explorés pour éviter d'y retourner.

L'utilisation de cette métaheuristique a été un tournant important dans la résolution de problèmes dans les sciences appliquées, la gestion et l'ingénierie. La recherche avec tabous possède des liens avec les algorithmes dits "évolutionnaires" (dont les algorithmes génétiques) [67]. La mémoire adaptative de l'approche taboue a aussi attiré les chercheurs puisqu'elle permet dans la pratique une amélioration des réseaux de neurones [124]. Les applications actuelles de la recherche avec tabous sont dans la planification de ressources, les télécommunications, l'analyse financière, l'ordonnancement, la planification d'espace, la distribution d'énergie, l'ingénierie moléculaire etc. Plusieurs publications et études ont porté sur la recherche avec tabous et ont contribué à étendre la version de base et permettre de trouver des solutions dont la qualité est supérieure à celle obtenue auparavant [64]. La recherche avec tabous se distingue des autres métaheuristiques par son exploitation de la mémoire. Cependant, cela a aussi contribué à la découverte de stratégies potentielles de gestion de mémoire. Les nouvelles données et principes qui ont émergé de la recherche avec tabous ont donné la base de création à des systèmes dont les capacités sont supérieures à leurs antécédents en terme de temps de réponse, de qualité de solution etc. De plus, il reste plusieurs variations qui n'ont pas encore été explorées qui pourraient faire avancer le domaine encore plus [66]. Par exemple, la concentration sur les bonnes régions, l'identification des régions prometteuses, les modèles de recherches non monotones et l'intégration et l'extension des solutions. La Figure 2.34 illustre un algorithme de recherche avec tabous. L'algorithme commence par générer aléatoirement une solution initiale s et la désigne comme meilleure solution connue s^*. Une initialisation se fait au niveau de la liste taboue et du compteur k. L'algorithme détermine ensuite la meilleure solution (s_{k+1}) dans le voisinage N de s. Si la solution trouvée est meilleure que s^* alors s_{k+1} remplace s^* et la liste taboue est mise à jour. L'algorithme réitère ce processus jusqu'à ce que la condition d'arrêt soit satisfaite.

Crainic *et al.* [34] ont proposé une taxonomie de parallélisation de la recherche avec tabous. Ce travail est considéré comme le travail le plus complet sur le domaine et se base sur une classification à trois dimensions. La première, appelée contrôle de cardinalité, définit si la recherche est contrôlée par un seul processus ou encore par plusieurs qui peuvent collaborer

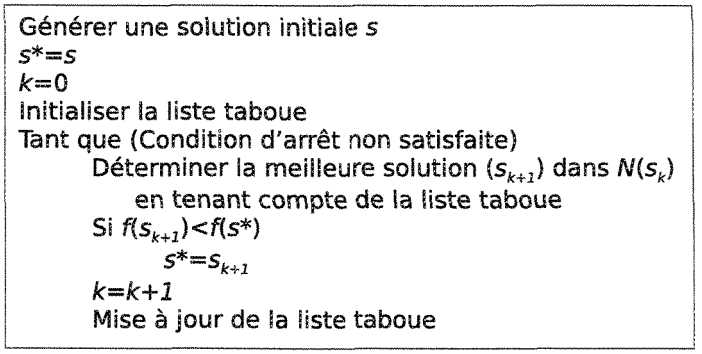

Figure 2.34 – Algorithme de recherche avec tabous [143]

ou non. Dans le deuxième cas, chaque processus est responsable de sa propre recherche et d'établir les communications avec les autres processus. La recherche globale se termine en même temps que les recherches individuelles. La deuxième dimension se réfère au type et à la flexibilité de la recherche. Elle est traitée en quatre niveaux qui correspondent chacun à des schémas de contrôles. La troisième dimension est la différenciation de la recherche. Une revue de littérature complète peut être retrouvée dans Gendreau [64]. L'auteur décrit notamment les versions hybrides de recherche avec tabou comme étant les plus prometteuses du domaine. C'est un constat qui s'est confirmé dans les récents travaux comme ceux de Mack *et al.* [99] ou encore Homberger *et al.* [85].

2.2.4 Autres métaheuristiques parallèles

Les métaheuristiques vues préalablement, mêmes si elles sont les plus populaires, ne forment pas une liste exhaustive des métaheuristiques qui existent. D'autres travaux orientés vers la parallélisation des métaheuristiques ont été conduits.

On peut citer les algorithmes d'estimation de distribution (Estimation of Distribution Algorithms EDA) [112] et [111] qui font partie de la famille des algorithmes évolutionnaires. Ces algorithmes agissent sur une population d'individus et estiment la probabilité de distribution de chaque variable dans ces individus. Cette estimation est par la suite utilisée pour générer un nouvel ensemble d'individus qui tend à être plus proche de l'optimum du problème. Cet algorithme a la particularité d'être exigeant en coûts de calcul, principalement pour la fonction d'estimation. C'est justement cette particularité qui a été le point de départ de l'approche

parallèle de ces algorithmes. Les recherches dans le domaine des algorithmes d'évaluation de distribution sont plutôt récentes. On cite notamment les travaux d'Ahm *et al.* [4] qui ont introduit un framework pour développer des algorithmes d'estimations parallèles et ceux de Mendiburu *et al.* [105] qui ont étendu des algorithmes déjà existants en utilisant de la programmation distribuée.

On cite aussi comme métaheuristique parallèle, la recherche par dispersion parallèle (Parallel Scatter search) qui fait également partie de la famille des algorithmes évolutionnaires. La version séquentielle de cet algorithme, introduite par Laguna *et al.* [91], fait combiner par un processus intelligent, un nombre de solutions jusqu'à atteindre un optimum. La principale différence avec les algorithmes génétiques, est que la recherche de l'optimum local est guidée, dans le sens qu'un échantillon référence (RefSet) est sélectionné parmi l'ensemble de la population. Cet échantillon est intensifié et mis à jour à chaque itération. Une fois que les solutions de cet échantillon sont combinées, une recherche locale est lancée pour tenter d'améliorer les résultats obtenus. L'échantillon est mis à jour avec autant les bonnes que les mauvaises solutions. La version parallèle de la recherche par dispersion est une évolution naturelle de l'algorithme comme moyen efficace de contrer le temps de calcul. Le parallélisme peut être introduit de plusieurs manières, soit en lançant plusieurs algorithmes de recherche locale sur plusieurs processeurs, soit en divisant une même procédure de recherche locale sur plusieurs processeurs. La principale application de cet algorithme a été pour la résolution du problème de la médiane p (The p-median problem) notamment grâce aux travaux de Garcia *et al.* [61].

Une métaheuristique plus récente est celle de la recherche à voisinage variable parallèle (Parallel Variable Neighborhood Search) de Hansen *et al.* [79] qui repose sur le principe de changement systématique de voisinage tant dans la descente vers les optimums locaux, que dans la sortie des vallées qui les contiennent. La parallélisation de cet algorithme sert soit à augmenter la taille du problème à résoudre, soit à limiter le temps de calcul de la recherche locale ou encore à étendre l'espace de recherche. Plusieurs versions parallèles de la recherche à voisinage variable ont vu le jour [110]. La première, appelée version synchrone, permet de

réduire le temps d'exécution de la recherche locale. Il s'agit simplement de distribuer la partie recherche locale entre les processeurs. La deuxième, appelée version répliquée, consiste à lancer en parallèle plusieurs instances de l'algorithme séquentiel, et ceci dans le but d'explorer le plus d'espace de recherche possible.

Pour mieux décrire l'ensemble des métaheuristiques parallèles, on a recours à des classifications. De plus, le nombre croissant de versions dérivées implique la nécessité de regroupement en classes. C'est aussi un moyen efficace de distinguer les métaheuristiques standards et les hybrides. Dans la section suivante, deux classifications sont présentées.

2.2.5 Classification des métaheuristiques parallèles

Les recherches les plus élaborées en ce qui concerne la classification des métaheuristiques parallèles sont celles effectuées par Crainic et Toulouse [33] et Cung *et al.* [36]. Ces deux classifications n'abordent pas la question des métaheuristiques parallèles de la même manière. C'est pour cela que ces deux classifications sont présentées séparément.

2.2.5.1 Classification de Crainic et Toulouse

La classification de Crainic et Toulouse considère les sources de parallélisation dans les métaheuristiques. Selon le niveau dans lequel le traitement parallèle se produit, trois types de parallélisation sont identifiés.

Le premier type (aussi appelé parallélisme de bas niveau) est une stratégie de parallélisation au sein d'une itération de la métaheuristique. Elle a pour but de réduire le temps d'exécution de la version séquentielle de la même métaheuristique. Elle ne cherche ni à améliorer la qualité des solutions, ni à mieux explorer l'espace de recherche.

Le deuxième type est un parallélisme qui concerne les variables de décisions. Cette stratégie de parallélisation est retrouvée dans la littérature sous la forme de métaheuristiques en mode maître-esclave. Le noeud maître distribue les variables parmi les autres noeuds disponibles et ceux-ci procèdent à l'exploration de manière concurrente et indépendante. Le noeud maître récupère ensuite les données partielles des esclaves et les recombine.

Enfin, le troisième type de parallélisation est un stratégie d'exploration multiple à

l'aide de plusieurs processus concurrents. Il est possible que les processus n'utilisent pas la même heuristique. Si les processus communiquent entre eux, la stratégie est appelée coopérative. Par contre, si aucune information n'est échangée entre les processus, la stratégie est appelée indépendante.

2.2.5.2 Classification de Cung et al.

La classification de Cung *et al.* se veut indépendante de l'architecture. Elle ramène les métaheuristiques basées sur la recherche locale à un graphe. Dans ce graphe, les noeuds sont des solutions et les arcs relient les solutions voisines. Les métaheuristiques étant itératives, les itérations sont traduites par des évaluations successives du voisinage de la solution. Ces évaluations sont suivies d'un déplacement vers une direction ou une autre du graphe en évitant les minima locaux. L'ensemble des solutions visitées tout au long de ce processus est appelé marche. Selon comment la marche est conduite, deux grandes classes sont identifiées : marche simple et marche multiple.

La marche simple concerne les métaheuristiques qui explorent le voisinage des solutions avec une trajectoire unique. Leur parallélisation se fait sur la recherche du meilleur voisin, sur la fonction objectif ou encore sur la décomposition du domaine.

La marche multiple concerne les métaheuristiques qui explorent plusieurs trajectoires en leur attribuant chacun un processus. Chaque processus qui entame une marche est appelé thread. Si les threads s'échangent leurs informations respectives, alors une sous-classe appelée marche multiple coopérative est introduite. Si les threads sont isolés alors c'est une marche multiple indépendante.

En plus de disposer de plusieurs classifications, les métaheuristiques parallèles font l'objet d'analyses de résultats poussées. Par opposition aux méthodes exactes où seul importe le temps d'exécution, les métaheuristiques parallèles doivent à la fois satisfaire les critères de rapidité et d'éloignement par rapport à l'optimum. C'est pour cela que les métaheuristiques parallèles disposent également de mesures de performances qui renseignent sur les gains par rapport à leurs homologues séquentielles. Un bref survol de ces mesures est fait dans la section suivante.

2.2.6 Les mesures de performance des métaheuristiques parallèles

Les mesures de performances vues à la Section 2.1.5 sont aussi valables pour mesurer celles des métaheuristiques parallèles. Cependant, comme les métaheuristiques ne sont pas des algorithmes exacts, il convient de nuancer certaines mesures, voir même de les réajuster pour qu'elles conviennent mieux à comparer les métaheuristiques entre elles. Alba *et al.* [9] précisent que pour les algorithmes non déterministes, comme les métaheuristiques, c'est le temps moyen des deux versions séquentielles et parallèles qui doit être pris en compte. Il propose ainsi différentes définitions de l'accélération. Une accélération forte qui compare l'algorithme parallèle au résultat du meilleur algorithme séquentiel connu. C'est ce qui correspond le plus à la vraie définition de l'accélération mais vu la difficulté de trouver à chaque fois le meilleur algorithme existant, cette norme n'est pas beaucoup utilisée. Une accélération dite faible compare l'algorithme parallèle avec la version séquentielle développée par le même chercheur. Il peut alors présenter ses progrès soit en terme de qualité, soit en accélération pure. Barr et Hickman [19] ont présenté une taxonomie différente qui consiste en accélération, accélération relative et accélération absolue. L'accélération relative traite du rapport entre la version parallèle exécutée sur un seul processeur et celle exécutée sur l'ensemble des processeurs. Enfin, l'accélération absolue, qui est le rapport de la version séquentielle la plus rapide sur n'importe quelle machine et le temps d'exécution de la version parallèle.

2.2.7 Conclusion

Presque toutes les métaheuristiques classiques ont donné naissance à des versions parallèles plus ou moins fidèles à leur origine. Ce constat traduit une nécessité de vitesse et de performance que les ordinateurs parallèles tentent de fournir à des problèmes de plus en plus complexes et à des exigences de temps de plus en plus restrictives. Aucune métaheuristique ne peut être qualifiée de "meilleure" par rapport à une autre, mais la préférence va selon la nature du problème à résoudre.

2.3 Les objectifs de la recherche

L'objectif principal de ce mémoire est de concevoir une métaheuristique parallèle pour solutionner un problème d'optimisation combinatoire. Le problème choisi est un problème d'ordonnancement sur machine unique avec temps de réglages dépendants de la séquence. Ce problème sera détaillé à la Section 3.1.

De manière générale, un problème d'ordonnancement consiste à planifier l'exécution d'un certain nombre de tâches en cherchant à optimiser un ou plusieurs objectifs particuliers. Le problème d'ordonnancement traité dans ce mémoire appartient à la classe des problèmes NP-Difficiles. De ce fait, la recherche de solutions au moyen de métaheuristiques est tout à fait justifiée et nous avons opté pour les algorithmes génétiques pour leur grande adaptabilité et leur côté évolutionnaire. Compte tenu du besoin de calculs importants, nous visons à concevoir une métaheuristique parallèle exploitant certaines des structures et des modèles de parallélisation présentés à la Section 2.1.

Les objectifs spécifiques de ce mémoire sont les suivants :

1. Mettre au point un algorithme génétique séquentiel pour la résolution du problème d'ordonnancement et de comparer son efficacité sur des problèmes tests de la littérature.

2. Concevoir une version parallèle de cette métaheuristique dans un modèle à multiples populations et avec une topologie en anneau. Ainsi, il sera possible de mesurer l'impact de la version parallèle tant sur la qualité de solutions que sur le temps d'exécution.

MCours.com