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

Université Batna 2 – Mostefa Ben Boulaïd
Faculté de Mathématiques et d'Informatique
Département d'Informatique

Thèse

En vue de l'obtention du diplôme de
Doctorat Informatique

UNE APPROCHE DE DISTRIBUTION ET D'ORDONNANCEMENT FIABLE

Présentée par :

Abbache FARID

Soutenue le: 28/06/2018

Devant le jury composé de :

Président	Dr. Hamoudi KALLA	MCA	Université de Batna 2
Rapporteur	Dr. Sohaila BOUAAM	MCA	Université de Batna 2
Examinateurs	Dr. Lamri SAYAD	MCA	Université de M'sila
	Dr. Adbelhamid DJEFFAL	MCA	Université de Biskra

Remerciements

Je remercie vivement Hamoudi Kalla, mon directeur de thèse, pour m'avoir encadré pendant toutes ces années, cette thèse lui doit beaucoup. La disponibilité, la confiance et la liberté qu'il m'a accordées au cours de ce travail de thèse, m'ont permis d'entreprendre de nombreuses expériences et ont grandement contribué à la richesse de cette thèse.

J'exprime ma sincère reconnaissance aux membres du jury, Djeffal Abdelhamid, Lamri Sayad, et au Présidente, Bouam Souheila. Leurs commentaires ont largement contribués à l'amélioration de ce document.

J'aimerais remercier aussi tous les collègues qui par leurs conseils et leurs encouragements ont contribué à l'aboutissement de cette thèse.

Mes derniers remerciements s'adressent à ma famille. Je remercie tout particulièrement mes frères, mes sœurs surtout mes parents et ma femme, qui m'ont toujours aidé, soutenu et encouragé au cours de mes études.

Abstract: modified bacterial foraging algorithm for allocation and scheduling tasks on processors and links with high reliability is proposed (MBFA). MBFA ensure high reliability without redundancy of processors or links by using an efficient and simple structure. MBFA is improved by adding an efficient model to deal with the weak solution (i.e., allocation with low reliability). Also a simulated annealing technique is integrated into MBFA in order to strengthen the search of best solution. The efficiency and effectiveness of MBFA are tested using random generated problems. MBFA is tested over the published approaches HPSO and GAA using density parameter (*DS*), number of nodes (*N*) and the communication to computation ratio parameter (*CCR*). The results show the superiority of MBFA over the published approaches in all test cases.

Keywords: task allocation; heterogeneous distributed system; reliability; bacterial foraging algorithm; simulated annealing algorithm.

Résumé : L'algorithme modifié de recherche de nourriture bactérienne pour l'ordonnancement et la distribution (allocation) des tâches sur des processeurs et des liens à haute fiabilité est proposé (MBFA). MBFA assure une haute fiabilité sans redondance des processeurs ou des liens en utilisant une structure efficace et simple. MBFA est amélioré en ajoutant un modèle efficace pour faire face à la solution faible (c'est-à-dire, allocation avec une faible fiabilité). Aussi une technique de recuit simulé est intégrée dans MBFA afin de renforcer la recherche de la meilleure solution. L'efficience et l'efficacité de MBFA sont testées en utilisant des problèmes générés au hasard. MBFA est testé sur les approches publiées HPSO et GAA en utilisant le paramètre de densité (DS), le nombre de nœuds (N) et le paramètre du taux de temps communication au temps de calcul (CCR). Les résultats montrent la supériorité de MBFA sur les approches publiées dans tous les cas de test.

Mots-clés: ordonnancement et distribution des tâches; système distribué hétérogène; fiabilité; algorithme bactérien de recherche de la nourriture; algorithme de recuit simulé

Table des Matières

Table des figures **7**

Liste des tableaux **9**

Introduction générale **10**

Chapitre 1: Problèmes d'optimisations et métaheuristiques **12**

1.1 Introduction 12

1.2 Modèles d'optimisation 12

1.2.1 Modèles d'optimisation classiques 13

1.3 Complexité des problèmes 17

1.4 Méthodes d'optimisation 19

1.4.1 Méthodes exactes 20

1.4.2 Algorithmes approximatifs 20

1.4.3 Algorithmes gloutons 21

1.5 Les métaheuristiques 21

1.5.1 Principaux concepts communs pour la métaheuristique 23

1.5.2 Manipulation des contraintes 28

1.5.3 Réglage des paramètres 33

1.6 Métaheuristiques basées sur une seule solution 34

1.6.1 Concepts communs pour une solution unique metaheuristique 34

1.6.2 Solution initiale 37

1.6.3 Recherche locale(RL) : 38

1.6.4 Recuit simulé (simulated annealing (SA)) 39

1.7 Métaheuristiques basées sur une population de solution 44

1.7.1 Concepts communs fondées sur une population de solution 44

1.7.2 Algorithmes génétiques 47

1.7.3 Algorithme bactérien du recherche de la nourriture (bacterial foraging optimisation algorithm(BFOA)) ... 48

1.7.4 L'Algorithme de batte (Bat Algorithm) ... 49

1.7.5 Recherche de coucou (Cuckoo Search (CS)) ... 50

1.8 Conclusion ... 50

Chapitre 2: Les systèmes distribués ... 51

2.1 Introduction ... 51

2.1.1 Définition d'un système distribué ... 51

2.2 L'objectif d'un système distribué ... 53

2.2.1 Point Faible ... 56

2.3 Types de systèmes distribués ... 57

2.3.1 Systèmes de calcul distribués ... 57

2.3.2 Systèmes d'information distribués ... 60

2.3.3 Systèmes distribués pervasifs ... 60

2.4 Architectures des systèmes distribués ... 61

2.4.1 Styles Architecturaux ... 62

2.5 Processus ... 63

2.5.1 Threads ... 64

2.5.2 La communication dans les systèmes distribués ... 66

2.6 Spécification de système distribué utilisés ... 69

2.6.1 Modèle d'architecture : ... 69

2.6.2 Modèle d'application ... 70

2.6.3 Modèle de fiabilité ... 71

2.6.4 Modèle d'allocation des taches sous les contraintes de capacité processeurs est mémoires ... 72

2.7 Un état de l'art ... 73

2.7.1 Introduction ... 73

2.7.2 Approches approximatives (un état de l'art) ... 73

2.8 Conclusion ... 75
Chapitre 3 : Algorithme modifié de la recherche bactérienne ... 77
3.1 Introduction ... 77
3.2 L'algorithme modifié de la recherche bactérienne (modified bacterial foraging algorithm) ... 77
3.2.1 L'algorithme de la recherche bactérienne classique ... 77
3.2.2 Algorithme proposé ... 79
3.2.3 Evaluations expérimentales ... 83
3.2.4 Conclusion de la contribution ... 93
3.3 Conclusion ... 93
Chapitre 4 : Algorithme modifié de recherche discrète du coucou ... 94
4.1 Introduction ... 94
4.2 L'algorithme de recherche de coucou classique ... 94
4.2.1 Représentation de coucou (solution) ... 95
4.3 Algorithme modifié de recherche discrète du coucou (modified discrete cuckoo search (MDCS)) ... 96
4.4 Evaluation de performance ... 98
4.4.1 Conclusion de la contribution ... 102
4.5 Conclusion ... 102
Conclusion générale et perspectives ... 103

Table des figures

Figure 1-1 Le processus classique dans la prise de décision: formuler, modéliser, optimiser et implémenter 13
Figure 1-2 Modèles d'optimisation classique. 14
Figure 1-3 Illustration graphique du modèle PL et de sa résolution. 15
Figure 1-4 Instance TSP avec 52 villes. 17
Figure 1-5 Classes de complexité des problèmes de décision. 18
Figure 1-6 Méthodes d'optimisations classiques. 19
Figure 1-7 Quelques codages classiques: vecteur de valeurs binaires, vecteur de valeurs discrètes, vecteur de valeurs réelles et permutation. 24
Figure 1-8 Mappage entre l'espace des solutions et l'espace des codages. 25
Figure 1-9 Stratégies d'initialisation des paramètres. 34
Figure 1-10 Principaux principes des métaheuristiques à base de solution unique. 35
Figure 1-11 Exemple de voisinage pour un problème de permutation de taille 3. Par exemple, les voisins de la solution (2, 3, 1) sont (3, 2, 1), (2, 1, 3) et (1, 3, 2). 36
Figure 1-12 Optimum local et optimum global dans un espace de recherche. Un problème peut avoir de nombreuses solutions optimales globales. 37
Figure 1-13 Processus de recherche locale utilisant une représentation binaire de solutions, un opérateur de déplacement par retournement (retourner, se déplacer) et la stratégie de sélection du meilleur voisin. La fonction objectif à maximiser est x3 -60x2 + 900x. La solution optimale globale est f (01010) = f (10) = 4000, tandis que le local final optima trouvé est s = (10000), à partir de la solution s0 = (10001). 38
Figure 1-14 Principaux principes de P-métaheuristiques. 44
Figure 2-1 Un système distribué organisé en middleware. 52
Figure 2-2 Un exemple de système informatique en cluster 58
Figure 2-3 Une architecture en couches pour les systèmes de calcul en grille 59
Figure 2-4 Le style (a) basé sur couche (b) a basé sur objet. 63
Figure 2-5 Le style basé sur l'événement 63
Figure 2-6 Un serveur multithread organisé dans un modèle dispatcheur / Travailleur. 65
Figure 2-7 Un message typique tel qu'il apparaît sur le réseau. 66
Figure 2-8 Les étapes de l'écriture d'un client et d'un serveur dans DCE RPC. 67
Figure 2-9 Quatre combinaisons pour des communications faiblement couplées utilisant 68

Figure 2-10 Une architecture générale pour le streaming de données multimédia stockées sur un réseau. 69
Figure 2-11 Un exemple de système distribué hétérogène et un graphe d'interaction de tâches. (a) Système distribué hétérogène, (b) un graphe d'interaction de tâche (GIT), et (c) Temps d'exécution attendus (expected execution times). 71
Figure 4-1 Représentation de la i^{ème} bactérie 79
Figure 4-2 Comparaison de fiabilité sous le paramètre nombre de tâches........ 88
Figure 4-3 Comparaison de fiabilité sous le paramètre de densité 89
Figure 4-4 Comparaison de fiabilité sous paramètre CCR 90
Figure 4-5 Comparaison de fiabilité sous la taille du paramètre de la population........ 91
Figure 4-6 Comparaison de fiabilité sous le paramètre du nombre de processeurs........ 92
Figure 4-7 Comparaison de la performance entre MBFA, HPSO et GAA........ 93
Figure 5-1 Comparaison de la fiabilité de DCS, MDCS and HPSO........ 101

Liste des tableaux

Tableau 1-1 Effet du nombre de villes sur la taille de l'espace de recherche........................... 17
Tableau 2-1 Différentes formes de transparence dans un système distribué (ISO, 1995). 54
Tableau 2-2 Exemples de limitations d'évolutivité 55
Tableau 3-1Désignations des paramètres utilisées.. 84
Tableau 3-2 Valeurs des paramètres utilisées 84
Tableau 3-3 Paramètres et valeurs utilisées pour MBFA, HPSO et GAA 85
Tableau 4-1 Désignation des paramètres du système et du problème...................................... 99
Tableau 4-2 Valeurs des paramètres système et problème... 99
Tableau 4-3 Paramètres des MDCS et l'HPSO 100

Introduction générale

Dans le monde d'aujourd'hui, la grande préoccupation des grands constructeurs des systèmes informatiques distribués hétérogènes est de produire des systèmes sûrs, tolérant aux pannes et de haute fiabilité. Ce qui a mené à la création d'un domaine dédiée au traitement de la fiabilité et tous ces aspects. Le domaine de fiabilité, disponibilité et le calcul tolérant aux pannes a été développé pour les besoins critiques des applications militaires et spatiales. Les missions dans l'espace lointain de la NASA sont coûteuses, car elles nécessitent divers systèmes de redondance et de récupération pour éviter une défaillance totale. Le stockage informatique moderne utilise des techniques RAID redondantes de disques indépendants pour relier 50 à 100 disques dans un système rapide et fiable. Diverses idées issues de tolérante aux pannes informatique sont maintenant utilisées dans presque tous les systèmes informatiques commerciaux, militaires et spatiaux; dans les industries du transport, de la santé et du divertissement; dans les institutions d'éducation et de gouvernement; dans les systèmes téléphoniques; et dans les centrales à énergie fossile et nucléaire. Les développements rapides de la microélectronique ont conduit à des conceptions très complexes; par exemple, une automobile de luxe peut avoir 30-40 microprocesseurs connectés par un réseau local! De telles conceptions doivent être réalisées en utilisant des techniques tolérantes aux pannes pour fournir une fiabilité, une disponibilité et une sécurité matérielles et logicielles significatives. Les réseaux informatiques ont actuellement un grand intérêt, et leur bon fonctionnement nécessite un haut degré de fiabilité et de disponibilité. Cette fiabilité est obtenue au moyen de multiples chemins de connexion entre des emplacements au sein d'un réseau, de sorte que lorsqu'un chemin échoue, la transmission est réacheminée avec succès. Ainsi, la topologie du réseau fournit une structure complexe de chemins redondants qui, à son tour, fournissent une tolérance aux pannes, et ces principes s'appliquent également à la distribution d'énergie, aux systèmes téléphoniques et d'eau et à d'autres réseaux. Le calcul tolérant aux pannes est un terme générique décrivant des techniques de conception redondantes avec des composants en double ou des calculs répétés permettant un fonctionnement ininterrompu (tolérant) en réponse à une défaillance de composant (défauts). Parfois, les catastrophes du système sont causées en négligeant les principes de la redondance et leurs influences sur l'échec. Malgré l'efficacité des techniques basées sur la redondance, cette technique reste couteuse en terme d'argent (le cout est dupliqué au nombre de copies des éléments matériels où logiciels utilisés), aussi en terme de communication (la géstion d'un élément est plus simple que

plusieurs copies de cet élément). Donc, pour garantir la fiabilité d'un system distribué il faut assurer la fiabilité au niveau d'une ou plusieurs parties de ce système (i.e. gestionnaire de mémoire, gestionnaire de ressources, Allocateur/ordonnanceur des taches,…, etc.). Dans cette thèse, nous utilisons uniquement des techniques logicielles sans redondance au niveau d'allocateur / ordonnanceur des taches pour assurer une haute fiabilité. Ces techniques sont basées sur des méta-heuristiques améliorées et adaptées au problème de distribution et d'ordonnancement des taches sur un système distribué hétérogène.

La thèse est composée de sept éléments : une introduction générale, cinq chapitres et une conclusion générale :

Dans l'introduction générale, nous résumons l'objectif et la structure de la thèse.

Dans le chapitre un, nous donnons les concepts de base et les notions liés aux problèmes d'optimisation qui en général ainsi que les méta-heuristiques où nous ne présentons que les techniques qui nous sont utilisées dans cette thèse.

Dans le chapitre deux, nous présentons les systèmes distribués en général en expliquant les techniques d'ordonnancement et communication. Enfin nous concluons ce chapitre en présentant les modèles utilisés pour spécifier l'architecture, l'application, la fiabilité, et enfin le problème d'allocation dans le système distribué.

Dans le chapitre trois, nous donnons un état de l'art sur la thématique liée au problème d'allocation des taches pour la maximisation de la fiabilité des systèmes distribués.

Dans le chapitre quatre, nous présentons notre premier algorithme amélioré à partir de la méta-heuristique de la recherche bactérienne, où nous avons introduit une nouvelle technique de mappage qui permette une utilisation efficace de l'algorithme classique en ajoutant un nouveau modèle qui améliore les mauvaises solutions, aussi une technique de recuit simulé est intégré afin de renforcer le processus de recherche.

Dans le chapitre cinq, nous présentons le deuxième algorithme qui émule le processus de survivance du coucou, où nous introduisons des nouveaux modèles qui permettent un contrôle efficace de tous les paramètres de l'algorithme avec une mise à jour dynamique. Aussi, nous présentons une nouvelle technique de mappage.

Enfin, une conclusion résume nos contributions et présente quelques pistes de recherche futures.

Chapitre 1: Problèmes d'optimisations et métaheuristiques

1.1 Introduction

Le calcul de solutions optimales est intraitable pour de nombreux problèmes d'optimisation d'importance industrielle et scientifique. En pratique, nous sommes généralement satisfaits des «bonnes» solutions, obtenues par des algorithmes heuristiques ou méta-heuristiques. Les méta-heuristiques représentent une famille de techniques d'optimisation approximatives qui ont gagné beaucoup de popularité au cours des deux dernières décennies. Ils sont parmi les techniques les plus prometteuses et les plus réussies. Les méta-heuristiques fournissent des solutions «acceptables» dans un délai raisonnable pour résoudre des problèmes complexes en science et en ingénierie. Ceci explique la croissance significative d'intérêt dans le domaine méta-heuristique. Contrairement aux algorithmes d'optimisation exacte, les méta-heuristiques ne garantissent pas l'optimalité des solutions obtenues, comme les algorithmes approximatifs, les méta-heuristiques ne définissent pas à quel point les solutions obtenues sont proches des optimales.

1.2 Modèles d'optimisation

En tant que scientifiques, ingénieurs et gestionnaires, nous devons toujours prendre des décisions. La prise de décision est partout. À mesure que le monde devient de plus en plus complexe et compétitif, la prise de décision doit être abordée de manière rationnelle et optimale. La prise de décision consiste en les étapes suivantes [1] (figure. 1.1):

- Formuler le problème: Dans cette première étape, un problème de décision est identifié. Ensuite, une déclaration initiale du problème est faite. Cette formulation peut être imprécise. Les facteurs internes et externes et le (s) objectif (s) du problème sont décrits. De nombreux décideurs peuvent être impliqués dans la formulation du problème.
- Modéliser le problème: Dans cette étape importante, un modèle mathématique abstrait est construit pour le problème. Le modeleur peut s'inspirer de modèles similaires dans la littérature. Cela réduira le problème aux modèles d'optimisation bien étudiés. Habituellement, les modèles que nous résolvons sont des simplifications de la réalité. Ils impliquent des approximations et parfois ils ignorent des processus complexes à représenter dans un modèle mathématique. Une question intéressante peut se poser:

pourquoi résoudre exactement des problèmes d'optimisation réels qui sont flous par nature?

- Optimiser le problème: Une fois le problème modélisé, la procédure de résolution génère une «bonne» solution au problème. La solution peut être optimale ou sous-optimale. Notons que nous trouvons une solution pour un modèle abstrait du problème et non pour le problème de la vie réelle formulé à l'origine. Par conséquent, les performances de la solution obtenues sont indicatives lorsque le modèle est précis. Le concepteur d'algorithmes peut réutiliser des algorithmes de pointe sur des problèmes similaires ou intégrer la connaissance de cette application spécifique dans l'algorithme.
- Implémenter une solution: La solution obtenue est testée pratiquement par le décideur est implémentée si elle est «acceptable». Certaines connaissances pratiques peuvent être introduites dans la solution à implémenter. Si la solution est inacceptable, le modèle et / ou l'algorithme d'optimisation doit être amélioré et le processus de décision est répété.

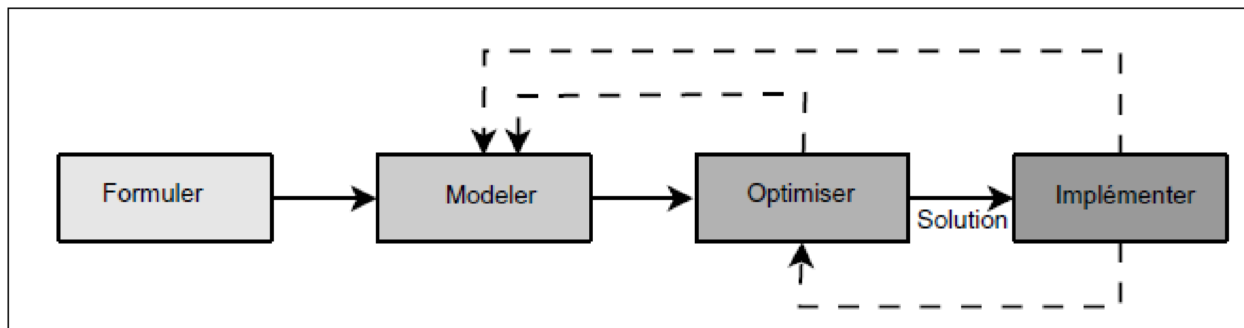


Figure 1-1 Le processus classique dans la prise de décision: formuler, modéliser, optimiser et implémenter

1.2.1 Modèles d'optimisation classiques

Les problèmes d'optimisation sont rencontrés dans de nombreux domaines: la science, l'ingénierie, la gestion et les affaires. Un problème d'optimisation peut être défini par le couple (S, *f*), où S représente l'ensemble des solutions réalisables et f: S → R la fonction objective à optimiser. La fonction objective assigne à chaque solution s ∈ S de l'espace de recherche un nombre réel indiquant sa valeur. La fonction objective *f* permet de définir une relation d'ordre total entre n'importe quelle paire de solutions dans l'espace de recherche.

Définition : *Optimum global* : une solution s* ∈ S est un optimum global si elle a une meilleure fonction objective que toutes les solutions de l'espace de recherche, c'est-à-dire ∀s ∈ S, $f(s^*) \leq f(s)$.

Par conséquent, l'objectif principal dans la résolution d'un problème d'optimisation est de trouver une solution optimale globale s*. De nombreuses solutions optimales globales peuvent exister pour un problème donné. Par conséquent, pour obtenir plus d'alternatives, le problème peut également être défini comme une recherche de toutes les solutions optimales globales.

Différentes familles de modèles d'optimisation sont utilisées dans la pratique pour formuler et résoudre des problèmes d’optimisation (figure 1.2). Les modèles les plus réussis sont basés sur la programmation mathématique et la programmation par contraintes [1].

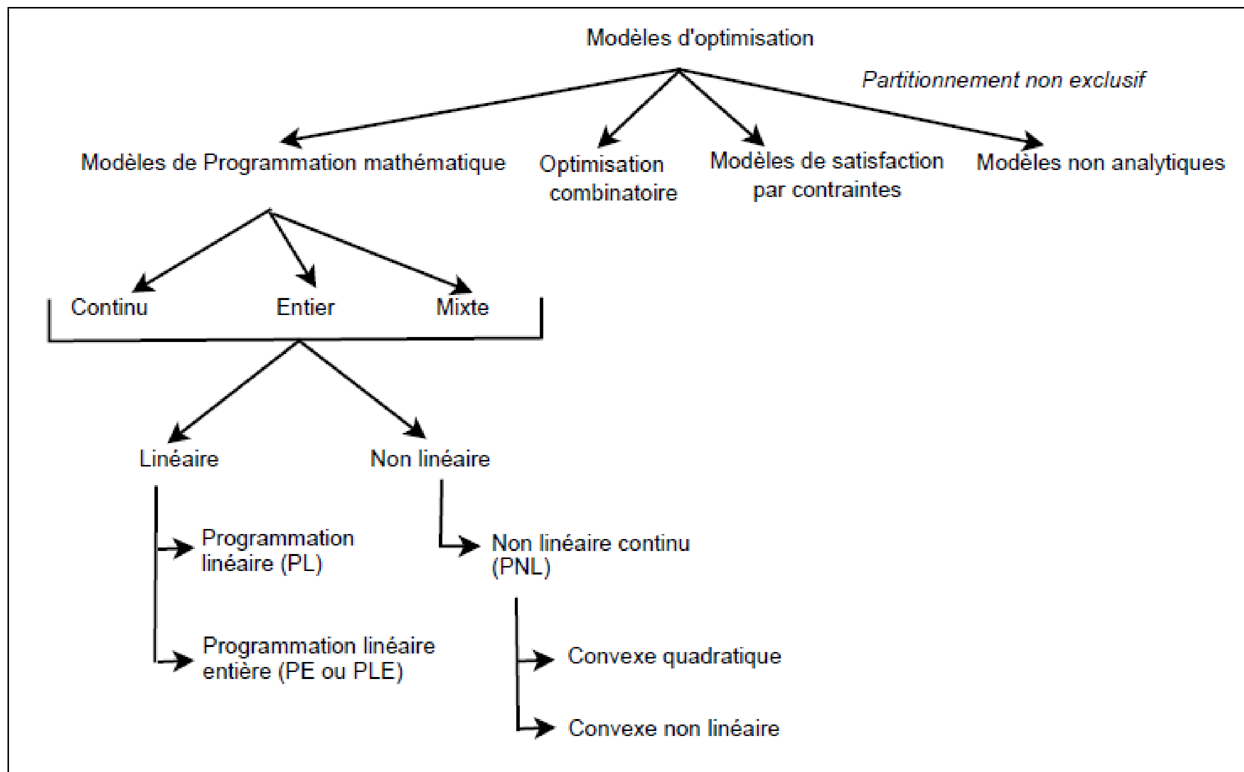


Figure 1-2 Modèles d’optimisation classique.

Un modèle couramment utilisé en programmation mathématique est la programmation linéaire (LP), qui peut être formulée comme suit:

$$\begin{aligned} & \textit{Min } c \bullet x \\ \textit{Sujet à} \quad & A \bullet x \geq b \\ & x \geq 0 \end{aligned}$$

où x est un vecteur de variables de décision continues, et c et b (respectivement A) sont des vecteurs constants (respectivement des matrices) de coefficients. Dans un problème d'optimisation de programmation linéaire, à la fois la fonction objectif $c\ .\ x$ à optimiser et les contraintes $A\ .\ x \geq b$ sont des fonctions linéaires. La programmation linéaire est l'un des modèles les plus satisfaisants de résolution des problèmes d'optimisation. L'efficacité des algorithmes est due au fait que la région réalisable du problème est un ensemble convexe et que la fonction objective est une fonction convexe. Ensuite, la solution optimale globale est nécessairement un nœud du polytope représentant la région réalisable (voir figure. 1.3). De plus, toute solution optimale locale est un optimum global.

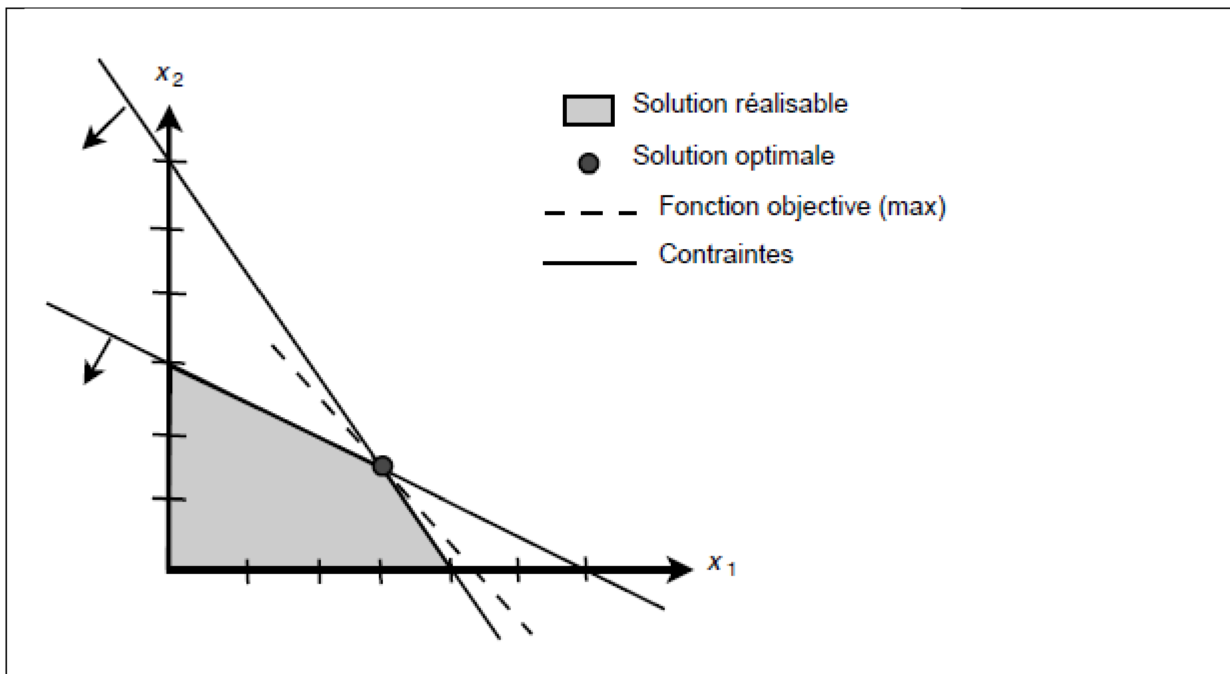


Figure 1-3 Illustration graphique du modèle PL et de sa résolution.

Les modèles de programmation non linéaire (LPN) traitent de problèmes de programmation mathématique où la fonction objectif et / ou les contraintes sont non linéaires [2]. La théorie

de l'optimisation continue en termes d'algorithmes d'optimisation est plus développée que l'optimisation discrète. Cependant, il existe de nombreuses applications de la vie réelle qui doivent être modélisées avec des variables discrètes. Les modèles continus sont inappropriés pour ces problèmes. En effet, dans de nombreux problèmes d'optimisation pratiques, les ressources sont indivisibles (machines, personnes, etc.). Dans un modèle d'optimisation de programme entier (PE), les variables de décision sont discrètes [3]. Lorsque les variables de décision sont à la fois discrètes et continues, nous somme efface des problèmes de programmation d'entiers mixtes (PEM). Par conséquent, les modèles PEM (en anglais MIP) généralisent les modèles PL et PE.

Pour les modèles PE et PEM, des algorithmes énumératifs tels que brancher et lier peuvent être utilisés pour de petites instances. La taille n'est pas le seul indicateur de la complexité du problème, mais aussi sa structure. Les métaheuristiques sont l'un des algorithmes concurrents pour cette classe de problèmes afin d'obtenir de bonnes solutions pour des instances considérées comme trop complexes pour être résolues de manière exacte. Les métaheuristiques peuvent également être utilisées pour générer de bonnes limites inférieures ou supérieures pour des algorithmes exacts et améliorer leur efficacité.

Une catégorie plus générale de problèmes PE est celle des problèmes d'optimisation combinatoire. Cette classe de problèmes est caractérisée par des variables de décision discrètes et un espace de recherche fini. Cependant, la fonction objective et les contraintes peuvent prendre n'importe quelle forme [4].

Exemple 1.1 *Problème de voyageur de commerce*. Le problème d'optimisation combinatoire le plus populaire est peut-être le problème du voyageur de commerce (TSP en anglais). Il peut être formulé de la façon suivante: étant donné n villes et une matrice de distance $d_{n,n}$, où chaque élément d_{ij} représente la distance entre les villes i et j, trouver un tour qui minimise la distance totale en visitant chaque ville une seule fois (cycle hamiltonien) (figure 1.4).

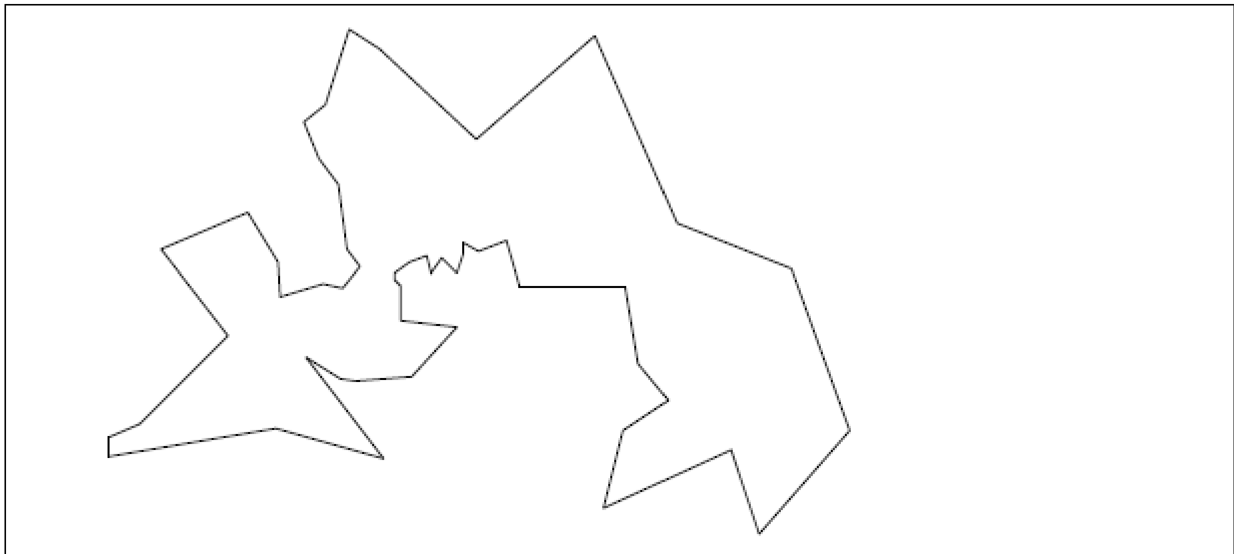


Figure 1-4 Instance TSP avec 52 villes.

La taille de l'espace de recherche est *n!*. Le tableau 1.1 montre l'explosion combinatoire du nombre de solutions concernant le nombre de villes. Malheureusement, l'énumération exhaustive de toutes les solutions possibles n'est pas pratique pour les instances modérées et grandes.

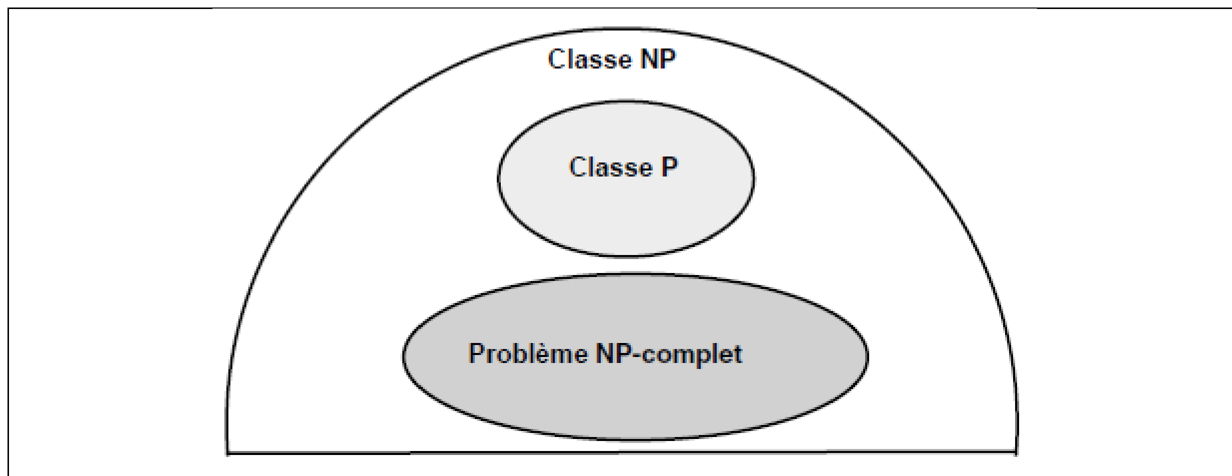
Tableau 1-1 Effet du nombre de villes sur la taille de l'espace de recherche

Nombre de villes n	Taille de l'espace de recherche
5	120
10	3, 628, 800
75	2.5×10^{109}

1.3 Complexité des problèmes

La complexité d'un problème équivaut à la complexité du meilleur algorithme pour résoudre ce problème. Un problème est tractable (ou facile) s'il existe un algorithme de temps polynomial pour le résoudre. Un problème est intraitable (ou difficile) si aucun algorithme polynomial n'existe pour résoudre le problème. La théorie de la complexité des problèmes traite des problèmes de décision. Un problème de décision a toujours une réponse par oui ou par non [1].

Un aspect important de la théorie de calcul de complexité est de catégoriser les problèmes en classes de complexité. La classe de complexité représente l'ensemble de tous les problèmes qui peuvent être résolus en utilisant une quantité donnée de ressources de calcul. Il existe deux classes importantes de problèmes: P et NP (figure 1.5).



erugiF1-5 Classes de complexité des problèmes de décision.

La classe P représente la famille de problèmes où un algorithme de temps polynomial connu existe pour résoudre le problème. Les problèmes appartenant à la classe P sont alors relativement "faciles" à résoudre. La classe de complexité NP représente l'ensemble de tous les problèmes de décision qui peuvent être résolus par un algorithme non déterministe en temps polynomial.

Un problème de décision $A \in$ NP est NP-complet si tous les autres problèmes de la classe NP sont polynomialement réduits au problème A. La figure 1.5 montre la relation entre les problèmes P, NP et NP-complets. Si un algorithme déterministe polynomial existe pour résoudre un problème NP-complet, alors tous les problèmes de classe NP peuvent être résolus en temps polynomial.

Les problèmes NP-difficiles sont des problèmes d'optimisation dont les problèmes de décision associés sont NP-complets. La plupart des problèmes d'optimisation du monde réel sont NP-difficile pour lesquels n'existent pas des algorithmes efficaces pour les résoudre. Ils nécessitent un temps exponentiel (à moins que P = NP). Les métaheuristiques constituent une

alternative importante pour résoudre cette classe de problèmes. Beaucoup de problèmes populaires académiques sont NP- difficiles parmi eux:

- Problèmes de séquencement et d'ordonnancement tels que l'ordonnancement du flux de production.
- Problèmes d'affectation et de localisation tels que le problème d'assignation quadratique (QAP), le problème d'assignation généralisée (GAP).

1.4 Méthodes d'optimisation

Suivant la complexité du problème, un problème peut être résolu par une méthode exacte ou une méthode approximative (figure 1.6). Les méthodes exactes obtiennent des solutions optimales et garantissent leur optimalité. Pour les problèmes NP-complets, les algorithmes exacts sont des algorithmes non polynomiaux (à moins que P = NP). Les méthodes approximatives (ou heuristiques) génèrent des solutions de haute qualité dans un délai raisonnable pour une utilisation pratique, mais il n'y a aucune garantie de trouver une solution optimale globale [1].

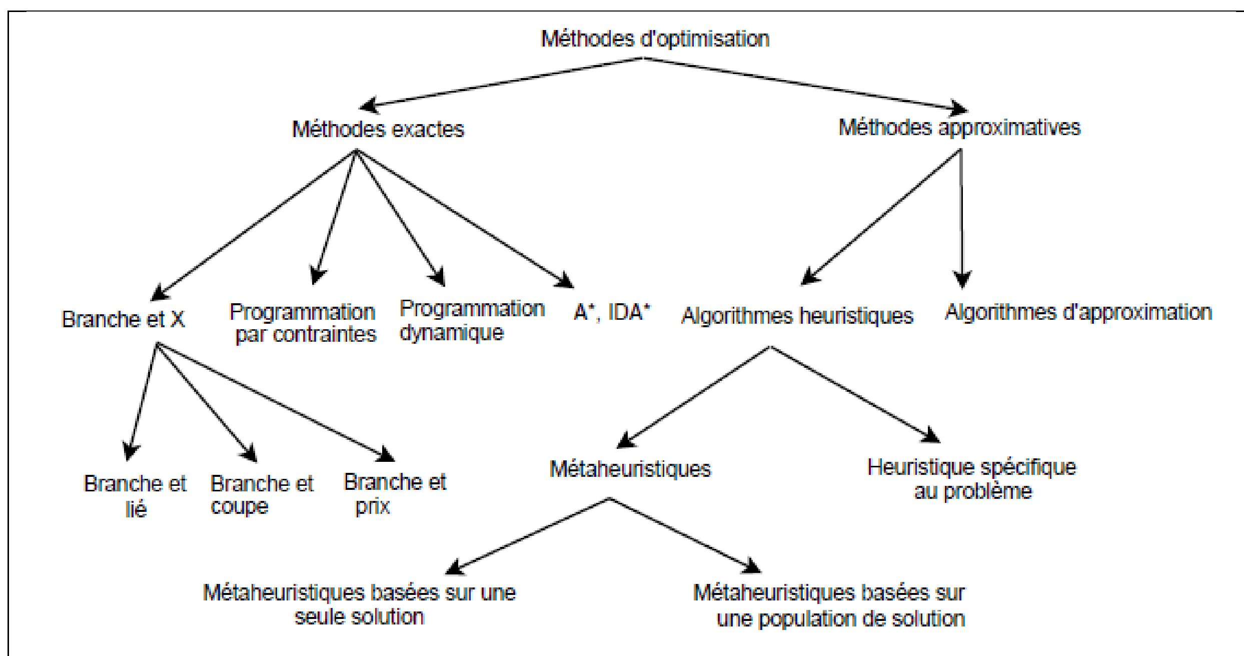


Figure 1-6 Méthodes d'optimisations classiques.

1.4.1 Méthodes exactes

Dans la classe des méthodes exactes on peut trouver les algorithmes classiques suivants: programmation dynamique, famille d'algorithmes de branche et X (branche et liaison, branche et coupe, branche et prix) développés dans la communauté de recherche opérationnelle, programmation par contraintes et famille A* d'algorithmes de recherche (A*, Iterative deepening A* (IDA*)) [5] développés dans la communauté de l'intelligence artificielle [6]. Ces méthodes énumératives peuvent être considérées comme des algorithmes de recherche arborescente. La recherche est effectuée sur l'ensemble de l'espace de recherche intéressant, et le problème est résolu en le subdivisant en problèmes plus simples.

La programmation dynamique est basée sur la division récursive d'un problème en sous-problèmes plus simples. Cette procédure est basée sur le principe de Bellman selon lequel «la sous-politique d'une politique optimale est elle-même optimale» [7]. Cette méthode d'optimisation par étapes est le résultat d'une séquence de décisions partielles. La procédure évite une énumération totale de l'espace de recherche en élaguant des séquences de décision partielles qui ne peuvent conduire à la solution optimale. L'algorithme de brancher et lier et A* sont basés sur une énumération implicite de toutes les solutions du problème d'optimisation considéré. L'espace de recherche est exploré en construisant dynamiquement un arbre dont le nœud racine représente le problème à résoudre et tout son espace de recherche associé. Les nœuds feuilles sont les solutions potentielles et les nœuds internes sont des sous-problèmes de l'espace total de la solution. L'élagage de l'arbre de recherche est basé sur une fonction de délimitation qui élague les sous-arbres qui ne contiennent aucune solution optimale. La programmation par contraintes est un langage construit autour de concepts de recherche arborescente et d'implications logiques.

1.4.2 Algorithmes approximatifs

Dans la classe des méthodes approchées, deux sous-classes d'algorithmes peuvent être distinguées: les algorithmes d'approximation et les algorithmes heuristiques. Contrairement aux heuristiques, qui trouvent habituellement des solutions raisonnablement «bonnes» dans un temps raisonnable, les algorithmes d'approximation fournissent une qualité de solution prouvable et des limites d'exécution prouvables. Les heuristiques trouvent de "bonnes" solutions sur les instances de problèmes de grande taille. Ils permettent d'obtenir des performances acceptables à des coûts acceptables dans une large gamme de problèmes. En

général, les heuristiques n'ont pas de garantie d'approximation sur les solutions obtenues. Ils peuvent être classés en deux familles: les heuristiques spécifiques et les métaheuristiques. Les heuristiques spécifiques sont conçues pour résoudre un problème et / ou une instance spécifique. Les métaheuristiques sont des algorithmes à usage général qui peuvent être appliqués pour résoudre presque tous les problèmes d'optimisation.

1.4.3 Algorithmes gloutons

Dans les algorithmes gloutons ou constructifs, nous partons de zéro (solution vide) et construisons une solution en assignant des valeurs à une variable de décision à la fois, jusqu'à ce qu'une solution complète soit générée. Dans un problème d'optimisation, où une solution peut être définie par la présence / l'absence d'un ensemble fini d'éléments $E = \{e_1, e_2, \ldots, e_n\}$, la fonction objectif peut être définie comme $f: 2^E \rightarrow R$, et l'espace de recherche est défini comme $F \subset 2^E$. Une solution partielle s peut être vue comme un sous-ensemble $\{e_1, e_2, \ldots, e_k\}$ des éléments e_i de l'ensemble de tous les éléments E. L'ensemble définissant la solution initiale est vide. À chaque étape, une heuristique locale est utilisée pour sélectionner le nouvel élément à inclure dans l'ensemble. Une fois qu'un élément e_i est sélectionné pour faire partie de la solution, il n'est jamais remplacé par un autre élément. Il n'y a pas de retour en arrière des décisions déjà prises. Typiquement, les heuristiques gloutonnes sont des algorithmes déterministes. L'algorithme 1.1 montre le modèle d'un algorithme glouton [1].

Algorithme 1.1 Modèle d'un algorithme glouton.
$s = \{\}$; /∗ Solution initiale (nulle) ∗/ **Répéter** e_i = Heuristique- Locale($E \setminus \{e/e \in s\}$) ; /∗ l'élément suivant sélectionné dans l'ensemble E moins les éléments déjà sélectionnés ∗/ **Si** $s \cup e_i \in F$ **Alors** /∗ tester la faisabilité de la solution ∗/ $s = s \cup e_i$; **Jusqu'à** que la solution complète soit trouvée

1.5 Les métaheuristiques

Contrairement aux méthodes exactes, les métaheuristiques permettent d'aborder des instances de problèmes de grande taille en fournissant des solutions satisfaisantes dans un délai raisonnable. Il n'y a aucune garantie de trouver des solutions optimales globales. Les

métaheuristiques ont reçu de plus en plus de popularité au cours de ces dernières années. Leur utilisation dans de nombreuses applications montre leur efficacité à résoudre des problèmes complexes et de grande taille. L'application de métaheuristiques tombe dans un grand nombre de domaines; telle que [1]

- La conception technique, l'optimisation topologique et l'optimisation structurelle en électronique et VLSI, l'aérodynamique, la dynamique des fluides, les télécommunications, l'automobile et le robotique.
- L'apprentissage automatique et l'exploration de données en bioinformatique et en biologie computationnelle, et en finance.
- La modélisation, la simulation et l'identification de systèmes en chimie, la physique et la biologie, le contrôle, le traitement d'image et signal.
- La planification de problèmes de routage, la planification des robots, les problèmes d'ordonnancement et de production, le logistique et le transport, la gestion de la chaîne d'approvisionnement, l'environnement, etc.

Dans la conception d'une métaheuristique, deux critères contradictoires doivent être pris en compte: l'exploration de l'espace de recherche (diversification) et l'exploitation des meilleures solutions trouvées (intensification). Les régions prometteuses sont déterminées par les «bonnes» solutions obtenues. En intensification, les régions prometteuses sont explorées de manière plus approfondie dans l'espoir de trouver de meilleures solutions. Dans la diversification, les régions inexplorées doivent être visitées pour être sûr que toutes les régions de l'espace de recherche sont explorées de manière égale et que la recherche ne se limite pas à un nombre réduit de régions. Dans cet espace de conception, les algorithmes de recherche extrêmes en termes d'exploration basé sur la recherche aléatoire (ou d'exploitation basé sur l'amélioration itérative de la recherche locale). En recherche aléatoire, à chaque itération, on génère une solution aléatoire dans l'espace de recherche. Aucune mémoire de recherche n'est utilisée. Dans l'algorithme de recherche locale de base et le plus raide, à chaque itération, on sélectionne la meilleure solution voisine qui améliore la solution actuelle. De nombreux critères de classification peuvent être utilisés pour les métaheuristiques:

1.5.1 Principaux concepts communs pour la métaheuristique

Il y a deux questions de conception communes liées à toutes les métaheuristiques itératives: la représentation des solutions gérées par les algorithmes et la définition de la fonction objective qui guidera la recherche [1].

1.5.1.1 Représentation

La conception de métaheuristiques itératives nécessite un codage (représentation) d'une solution. C'est une question de conception fondamentale dans le développement des métaheuristiques. L'encodage joue un rôle majeur dans l'efficacité de toute métaheuristique. L'encodage doit être adapté et pertinent au problème d'optimisation abordé. De plus, l'efficacité d'une représentation est également liée aux opérateurs de recherche appliqués sur cette représentation (voisinage, recombinaison, etc.). En fait, lors de la définition d'une représentation, il faut garder à l'esprit comment la solution sera évaluée et comment les opérateurs de recherche fonctionneront. De nombreuses représentations alternatives peuvent exister pour un problème donné. Une représentation doit avoir les caractéristiques suivantes:

- La complétude: L'une des principales caractéristiques d'une représentation est sa complétude; c'est-à-dire que toutes les solutions associées au problème doivent être représentées.
- La connexité: La caractéristique de connexité est très importante dans la conception de tout algorithme de recherche. Un chemin de recherche doit exister entre deux solutions quelconques de l'espace de recherche où toute solution de l'espace de recherche, en particulier la solution d'optimum global, peut être atteint.
- L’efficacité: la représentation doit être facile à manipuler par les opérateurs de recherche. Les complexités temporelles et spatiales des opérateurs traitant de la représentation doivent être réduites.

De nombreux codages simples peuvent être appliqués à certaines familles traditionnelles de problèmes d'optimisation (figure 1.7).

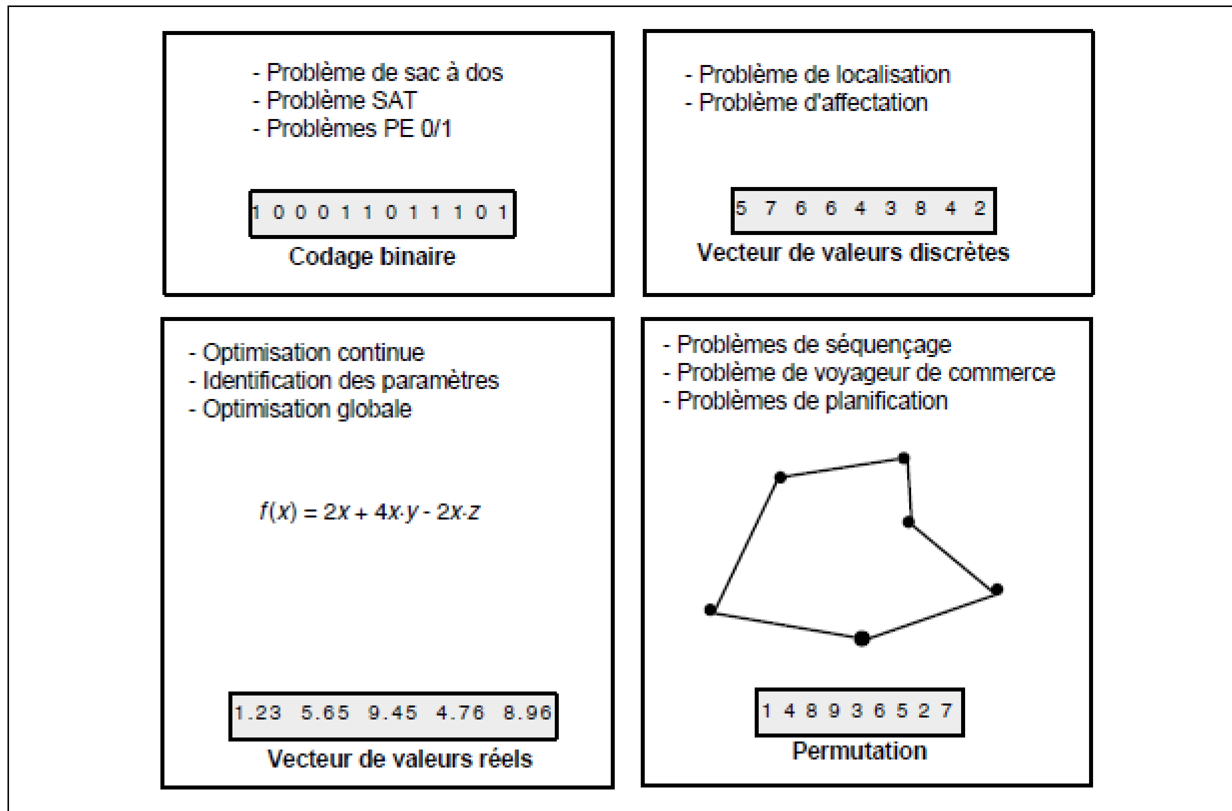


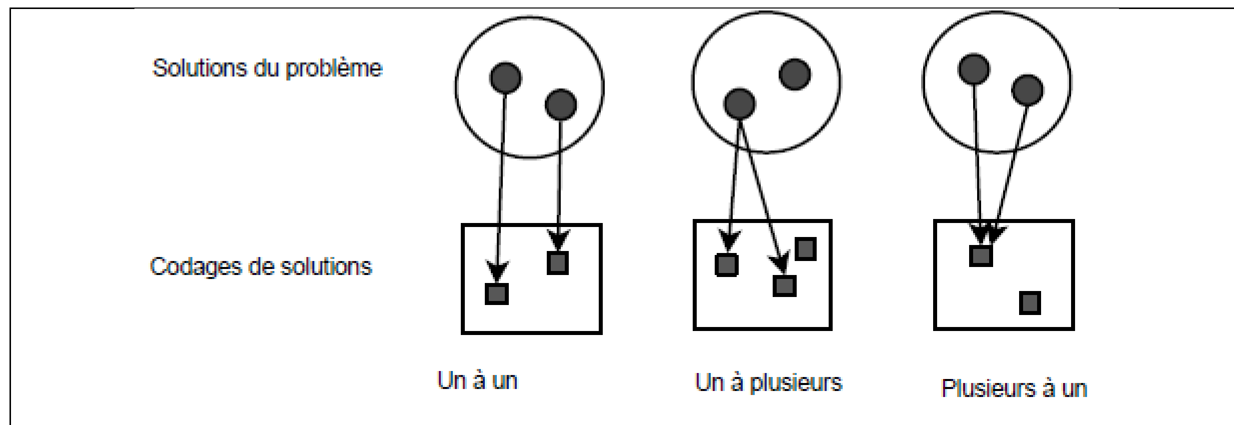
Figure 1-7 Quelques codages classiques: vecteur de valeurs binaires, vecteur de valeurs discrètes, vecteur de valeurs réelles et permutation.

1.5.1.2 Représentation de mappage des solutions

La fonction de mappage de la solution transforme le codage (génotype) en une solution (phénotype). Le mappage entre l'espace de solution et l'espace de codage implique trois possibilités [8] (Figure. 1.8):

- Une-à-Une: C'est la classe traditionnelle de la représentation. Ici, une solution est représentée par un codage unique et chaque codage représente une solution unique. Il n'y a pas de redondance ni de réduction de l'espace de recherche d'origine. Pour certains problèmes d'optimisation, il est difficile de concevoir un tel mappage.
- Une-à-plusieurs: Dans le mappage un-à-plusieurs, une solution peut être représentée par plusieurs codages. La redondance de l'encodage va augmenter la taille de l'espace de recherche et peut avoir un impact sur l'efficacité des métaheuristiques.
- Plusieurs-à-un: Dans cette classe, plusieurs solutions sont représentées par le même encodage. En général, ces codages sont caractérisés par un manque de détails dans le

codage; certaines informations sur la solution ne sont pas explicitement représentées. Cela réduira la taille de l'espace de recherche d'origine. Dans certains cas, cela améliorera l'efficacité des métaheuristiques. Cette classe de représentation est également appelée codage indirect.



erugiF1-8 Mappage entre l'espace des solutions et l'espace des codages.

1.5.1.3 Codages directs ou indirects

Lorsque vous utilisez une représentation indirecte, le codage n'est pas une solution complète pour le problème. Un décodeur est nécessaire pour exprimer la solution donnée par le codage. Selon l'information qui est présente dans le codage indirect, le décodeur dérivera une solution complète. Le décodeur peut être non déterministe. Les codages indirects sont populaires dans les problèmes d'optimisation traitant de nombreuses contraintes telles que les problèmes d'ordonnancements.

1.5.1.4 Fonction objectif

La fonction objective f formule l'objectif à atteindre. Elle associe à chaque solution de l'espace de recherche une valeur réelle qui décrit la qualité de la solution, $f: S \rightarrow R$. Ensuite, elle représente une valeur absolue et permet un ordonnancement complet de toutes les solutions de l'espace de recherche. Certaines fonctions de décodage d peuvent être appliquées, $d: R \rightarrow S$, pour générer une solution évaluable par la fonction f. La fonction objective est un élément important dans la conception d'une métaheuristique. Elle guidera la recherche vers de

«bonnes» solutions de l'espace de recherche. Si la fonction objective est mal définie, elle peut conduire à des solutions non acceptables quelle que soit la métaheuristique utilisée.

1.5.4.1 Fonctions objectives autosuffisantes

Pour certains problèmes d'optimisation, la définition de la fonction objective est simple. Elle spécifie la fonction objective formulée à l'origine.

Exemple 1.2 Fonction objectif de guidage simple : Dans de nombreux problèmes de routage tels que les problèmes de TSP et de routage de véhicules, l'objectif formulé est de minimiser une distance globale donnée. Par exemple, dans le TSP, l'objectif correspond à la distance totale de la tournée hamiltonienne:

$$f(s) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)}$$

Où π représente une permutation codant un tour et n le nombre de villes.

1.5.1.5 Fonctions d'objectif directrice (Guiding function)

Pour d'autres problèmes, la définition de la fonction objective est une tâche difficile et constitue une question cruciale. La fonction objective doit être transformée pour une meilleure convergence de la métaheuristique. La nouvelle fonction objective guidera la recherche de manière plus efficace.

Exemple 1.3 : La fonction objective du problème de satisfiabilité : Formulons une fonction objective pour résoudre les problèmes de satisfiabilité. Les problèmes SAT représentent des problèmes de décision fondamentaux dans l'intelligence artificielle. Le problème k-SAT peut être défini comme suit: étant donné une fonction F du calcul propositionnel sous forme normale conjonctive (CNF). La fonction F est composée de *m* clauses C_i de *k* variables booléennes, où chaque clause C_i est une disjonction. L'objectif du problème est de trouver une affectation des *k* variables booléennes telles que la valeur de la fonction F est vraie. Par conséquent, toutes les clauses doivent être satisfaites.

$$F = (x_1 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3})$$

Une solution au problème peut être représentée par un vecteur de *k* variables binaires. Une fonction objective directe consiste à utiliser la fonction F d'origine:

$$f = \begin{cases} 0 & \text{if is } F \text{ false} \\ 1 & \text{otherwise} \end{cases}$$

Si on considère les deux solutions $s_1 = (1, 0, 1, 1)$ et $s_2 = (1, 1, 1, 1)$, elles auront la même fonction objectif, c'est-à-dire la valeur 0, étant donné que la fonction F est égal à faux. L'inconvénient de cette fonction objective est qu'elle présente une faible différenciation entre les solutions. Une fonction objective plus intéressante pour résoudre le problème sera de compter le nombre de clauses satisfaites. Par conséquent, l'objectif sera de maximiser le nombre de clauses satisfaites. Cette fonction est meilleure en termes de guidage de la recherche vers la solution optimale. Dans ce cas, la solution s_1 (resp s_2) aura une valeur de 5 (respectivement 6). Cette fonction d'objectif conduit au modèle MAX-SAT.

1.5.1.6 Décodage de la représentation

Les questions de conception liées à la définition de la représentation et à la fonction objective peuvent être liées. Dans certains problèmes, la représentation (génotype) est décodée pour générer la meilleure solution possible (phénotype). Dans cette situation, le mappage entre la représentation et la fonction objective n'est pas simple; c'est-à-dire qu'une fonction de décodeur doit être spécifiée pour générer à partir d'une représentation donnée la meilleure solution en fonction de la fonction objective.

1.5.1.7 Optimisation interactive

Dans l'optimisation interactive, l'utilisateur est impliqué en ligne dans la boucle d'une métaheuristique. Il y a deux motivations principales pour concevoir des métaheuristiques interactives:

- *L'intervention de l'utilisateur pour guider le processus de recherche*: dans ce cas, l'utilisateur peut interagir avec l'algorithme de recherche pour converger plus rapidement vers des régions prometteuses. L'objectif ici est d'améliorer le processus de recherche en ligne en introduisant de façon dynamique certaines connaissances de l'utilisateur. Par exemple, l'utilisateur peut proposer des solutions prometteuses à partir d'une représentation graphique de solutions. L'utilisateur peut également proposer la mise à jour des paramètres métaheuristiques. Cette stratégie est largement utilisée dans la prise de décision multicritère dans laquelle une interaction est effectuée entre le décideur et le solveur pour converger vers la meilleure solution de compromis.
- *L'intervention de l'utilisateur pour évaluer une solution*: En effet, dans de nombreux problèmes de conception, la fonction objective nécessite une évaluation subjective en fonction des préférences humaines (par exemple, le goût). Pour certains problèmes, la fonction objective ne peut pas être formulée analytiquement (par exemple, attractivité visuelle). Cette stratégie est largement utilisée dans la création artistique (musique, images, formes, etc.).

1.5.2 Manipulation des contraintes

Traiter des contraintes dans les problèmes d'optimisation est un autre sujet important pour la conception efficace des métaheuristiques. En effet, de nombreux problèmes d'optimisation continus et discrets sont avec des contraintes, et il n'est pas trivial de traiter ces contraintes. Les contraintes peuvent être de toute nature: contraintes linéaires ou non linéaires et d'égalité ou d'inégalité. Les stratégies de gestion des contraintes, qui agissent principalement sur la représentation de solutions ou la fonction objective, sont présentées. Ils peuvent être classés comme des stratégies de rejet, des stratégies pénalisantes, des stratégies de réparation, des stratégies de décodage et de préservation des stratégies. D'autres approches de gestion de contraintes utilisant des composants de recherche non directement liés à la représentation de solutions ou à la fonction objective peuvent également être utilisées, telles que l'optimisation multi-objective et les modèles co-évolutionnaires [1].

1.5.2.1 Stratégies de rejet

Les stratégies de rejet représentent une approche simple, où seules les solutions réalisables sont conservées pendant la recherche, puis les solutions irréalisables sont automatiquement rejetées. Ce genre de stratégies est concevable si la partie des solutions infaisables de l'espace

de recherche est très petit. De plus, les stratégies de rejet n'exploitent pas les solutions irréalisables. En effet, il serait intéressant d'utiliser certaines informations sur des solutions irréalisables pour orienter la recherche vers des solutions optimales globales qui sont en général à la limite entre des solutions faisables et irréalisables. Dans certains problèmes d'optimisation, les régions réalisables de l'espace de recherche peuvent être discontinues. Par conséquent, il existe un chemin entre deux solutions réalisables s'il est composé de solutions irréalisables.

1.5.2.2 Stratégies de pénalisation

Dans les stratégies pénalisantes, des solutions irréalisables sont envisagées lors du processus de recherche. La fonction objective non contrainte est prolongée par une fonction de pénalité qui pénalisera les solutions irréalisables en intégrant les contraintes dans la fonction objective finale. C'est l'approche la plus populaire. De nombreuses alternatives peuvent être utilisées pour définir les pénalités [9].

Par exemple, la fonction objective f peut être pénalisée de manière linéaire:

$$f'(s) = f(s) + \lambda c(s)$$

Où $c(s)$ représente le coût de la violation de contrainte et λ les poids d'agrégation. La recherche permet des séquences de type (s_t, s_{t+1}, s_{t+2}) où s_t et s_{t+2} représentent des solutions réalisables, s_{t+1} est une solution infaisable, et s_{t+2} est meilleure que s_t. Selon la différence entre solutions faisables et infaisables, différentes fonctions de pénalité peuvent être utilisées [10]:

- *Contraintes violées*: une fonction directe consiste à compter le nombre de contraintes violées. Aucune information n'est utilisée sur la proximité de la solution par rapport à la région possible de l'espace de recherche. Etant donné m contraintes, la fonction pénalisée $f_p(x)$ de $f(x)$ est définie comme suit:

$$f_p(x) = f(x) + \sum_{i=1}^{m} w_i \alpha_i$$

où $\alpha_i = 1$ si la contrainte *i* est violée et $\alpha_i = 0$ sinon, et w_i est le coefficient associé à chaque contrainte *i*. Pour un problème avec des contraintes faibles et serrées, cette stratégie est inutile.

- *Montant d'infaisabilité ou de coût de réparation*: Les informations sur la proximité d'une solution par rapport à une région réalisable sont prises en compte. Cela donnera une idée du coût de la réparation de la solution. Par exemple, des approches plus efficaces consistent à inclure une distance à la faisabilité pour chaque contrainte. Considérant les *q* contraintes d'inégalité et les *m-q* contraintes d'égalité, la fonction pénalisée $f_p(x)$ sera formulée comme suit:

$$f_p(x) = f(x) + \sum_{i=1}^{m} w_i d_i^k$$

où d_i est une métrique de distance pour la contrainte *i*, $d_i = \alpha_i g_i(x)$ pour $i = 1, \ldots, q$ et $d_i = | h_i(x) |$ pour $i = q + 1, \ldots, m$. *k* est une constante définie par l'utilisateur (en général $k = 0, 1$), les contraintes $1, \ldots, q$ sont les contraintes d'inégalité et les contraintes $q + 1, \ldots, m$ sont des contraintes d'égalité.

Lors de la résolution d'un problème contraint en utilisant une stratégie pénalisante, un bon compromis pour l'initialisation des coefficients doit être trouvé. En effet, si w_i est trop petit, les solutions finales peuvent être irréalisables. Si le facteur de coefficient w_i est trop élevé, nous pouvons converger vers des solutions réalisables non optimales. La fonction pénalisante utilisée peut être :

- *Statique*: dans les stratégies statiques, un facteur de coefficient constant est défini pour l'ensemble de la recherche. L'inconvénient des stratégies statiques est la détermination des facteurs de coefficient w_i.
- *Dynamique*: Dans les stratégies dynamiques, les coefficients vont changer pendant la recherche. Par exemple, la sévérité de violation des contraintes peut être augmentée avec le temps. Cela signifie que lorsque la recherche progresse, les pénalités seront

plus fortes, alors qu'au début de la recherche, des solutions hautement infaisables sont admissibles [11].

$$f_p(x,t) = f(x) + \sum_{i=1}^{m} w_i(t) d_i^k$$

où $w_i(t)$ est une fonction monotone décroissante avec t.

Il n'est pas simple de définir une bonne fonction de pénalité dynamique. Un bon compromis pour l'initialisation de la fonction $w_i(t)$ doit être trouvé. En effet, si $w_i(t)$ est trop lent, une recherche plus longue est nécessaire pour trouver des solutions réalisables. Autrement, si $w_i(t)$ diminue trop rapidement, nous pouvons converger rapidement vers une solution réalisable non optimale.

- Adaptative: Les fonctions de pénalité présentées précédemment (statique et dynamique) n'exploitent aucune information du processus de recherche. Dans les fonctions de pénalité adaptative, les connaissances sur le processus de recherche sont incluses pour améliorer l'efficacité de la recherche.

La mise à jour des coefficients est faite en utilisant la mémoire de la recherche [33]. La mémoire de recherche peut contenir les solutions les mieux trouvées, les dernières solutions générées, etc. Par exemple, une stratégie adaptative peut consister à diminuer les facteurs de coefficients lorsque de nombreuses solutions réalisables sont générées pendant la recherche, tout en augmentant ces facteurs si de nombreuses solutions infaisables sont générées.

Exemple 1.4 *Pénalisation adaptative*. Considérons le problème de routage de véhicule capacitaire (PRVC). Une stratégie de pénalisation adaptative peut être appliquée pour traiter les contraintes de demande et de durée dans une métaheuristique [12]:

$$f'(s) = f(s) + \alpha Q(s) + \beta D(s)$$

Q(s) mesure la demande excédentaire totale de toutes les routes et *D(s)* mesure la durée excédentaire de toutes les routes. Les paramètres α et β sont auto-ajustables. Initialement, les deux paramètres sont initialisés à 1. Ils sont réduits (ou augmentés) selon un autre paramètre de contrôle μ. Les solutions sont toutes réalisables (ou toutes sont irréalisables), où μ est un paramètre défini par l'utilisateur. Par exemple, La réduction (ou l'augmentation) peut consister à diviser (ou multiplier) la valeur réelle par 2.

1.5.2.3 Stratégies de réparation

Les stratégies de réparation consistent en des algorithmes heuristiques transformant une solution infaisable en une solution réalisable. Une procédure de réparation est appliquée aux solutions infaisables pour générer des solutions réalisables. Par exemple, ces stratégies sont appliquées dans le cas où les opérateurs de recherche utilisés par les algorithmes d'optimisation peuvent générer des solutions irréalisables. Les heuristiques de réparation sont spécifiques au problème d'optimisation en question. La plupart d'entre eux sont des heuristiques gloutonnes. Ensuite, le succès de cette stratégie dépendra de la disponibilité de telles heuristiques efficaces.

1.5.2.4 Stratégies de décodage

Une procédure de décodage peut être vue comme une fonction $g : R \rightarrow S$ qui associe à chaque représentation $r \in R$ une solution réalisable $s \in S$ dans l'espace de recherche. Cette stratégie consiste à utiliser des codages indirects. La topologie de l'espace de recherche est ensuite transformée à l'aide de la fonction de décodage. La fonction de décodage doit avoir les propriétés suivantes [13]:

- Pour chaque solution $r \in R$, correspond une solution réalisable $s \in S$.
- Pour chaque solution possible $s \in S$, il y a une représentation $r \in R$ qui lui correspond.
- La complexité de calcul du décodeur doit être réduite.
- Les solutions réalisables dans *S* doivent avoir le même nombre de solutions correspondantes dans *R*.
- L'espace de représentation doit avoir la propriété de localité dans le sens où la distance entre les solutions dans *R* doit être corrélée positivement avec la distance entre les solutions réalisables dans *S*.

1.5.2.5 Stratégies de conservation

En préservant les stratégies de gestion des contraintes, une représentation spécifique et des opérateurs assureront la génération de solutions réalisables. Ils incorporent des connaissances spécifiques aux problèmes dans les opérateurs de représentation et de recherche pour générer uniquement des solutions réalisables et ensuite préserver la faisabilité des solutions. Cette classe efficace de stratégies est adaptée à des problèmes spécifiques. Il ne peut pas être généralisé pour gérer les contraintes de tous les problèmes d'optimisation. De plus, pour certains problèmes tels que le problème de coloration de graphe, il est même difficile de trouver une solution initiale réalisable ou une population de solutions pour commencer la recherche.

1.5.3 Réglage des paramètres

De nombreux paramètres doivent être réglés pour chaque métaheuristique. Le réglage des paramètres peut permettre une plus grande flexibilité et robustesse, mais nécessite une initialisation minutieuse. Ces paramètres peuvent avoir une grande influence sur l'efficience et l'efficacité de la recherche. Il n'est pas évident de définir a priori quel paramètre doit être utilisé. Les valeurs optimales pour les paramètres dépendent principalement du problème et même de l'instance à traiter et du temps de recherche que l'utilisateur souhaite consacrer à la résolution du problème. Un ensemble de valeurs de paramètres universellement optimal pour une métaheuristique donnée n'existe pas [1].

Il existe deux stratégies différentes pour le réglage des paramètres: l'initialisation des paramètres hors-ligne (ou méta-optimisation) et la stratégie d'ajustement des paramètres en ligne (figure 1.9). Dans l'initialisation des paramètres hors ligne, les valeurs des différents paramètres sont fixées avant l'exécution de la métaheuristique, alors que dans l'approche en ligne, les paramètres sont contrôlés et mis à jour de façon dynamique ou adaptative lors de l'exécution de la métaheuristique.

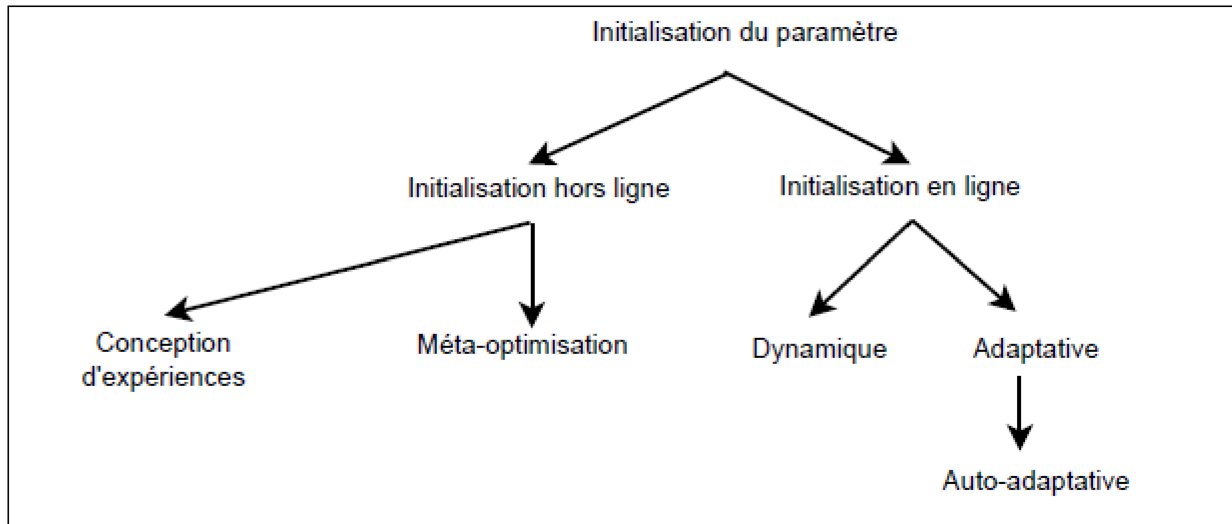


Figure 1-9 Stratégies d'initialisation des paramètres.

1.6 Métaheuristiques basées sur une seule solution

Tout en résolvant les problèmes d'optimisation, les métaheuristiques basées sur une solution unique (S-métaheuristiques) améliorent une solution unique. Ils pourraient être considérés comme des "promenades" à travers les chemins ou les trajectoires de recherche à travers l'espace de recherche du problème en question [14]. Les marches (ou trajectoires) sont effectuées par des procédures itératives qui passent de la solution actuelle à une autre dans l'espace de recherche. Les S-métaheuristiques montrent leur efficacité dans la résolution de divers problèmes d'optimisation dans différents domaines.

1.6.1 Concepts communs pour une solution unique metaheuristique

S-métaheuristiques appliquent itérativement des procédures de génération et de remplacement à partir de la solution unique actuelle (figure 1.10). Dans la phase de génération, un ensemble de solutions candidates est généré à partir des solutions actuelles. Cet ensemble $C(s)$ est généralement obtenu par les transformations locales de la solution. Dans la phase de remplacement, une sélection est effectuée à partir de l'ensemble de solutions candidat $C(s)$ pour remplacer la solution actuelle; c'est-à-dire, une solution $s' \in C(s)$ est choisie pour être la nouvelle solution. Ce processus itère jusqu'à un critère d'arrêt donné. L'algorithme 2.2 illustre le modèle d'un algorithme de recherche locale.

Algorithm 1.2 Modèle de haut niveau de S-métaheuristique.
Entrée: Solution initiale s_0. t = 0; **Répéter** /∗ Générer des solutions candidates (voisinage partiel ou complet) à partir de s_t ∗/ Générer($C(s_t)$) ; /∗ Sélectionnez une solution de $C(s)$ pour remplacer la solution actuelle s_t ∗/ s_{t+1} = Sélectionner ($C(s_t)$) ; t = t + 1 ; **Jusqu'à** que les critères d'arrêts soient satisfis **Sortie**: Meilleure solution trouvée.

Les phases de génération et de remplacement peuvent être sans mémoire. Dans ce cas, les deux procédures sont basées uniquement sur la solution actuelle. Sinon, un historique de la recherche stockée dans une mémoire peut être utilisé dans la génération de la liste de solutions candidates et la sélection de la nouvelle solution. Des exemples populaires de telles S-métaheuristiques sont la recherche locale, le recuit simulé et la recherche de tabous. L'algorithme 2.2 illustre le modèle de haut niveau de S-métaheuristiques.

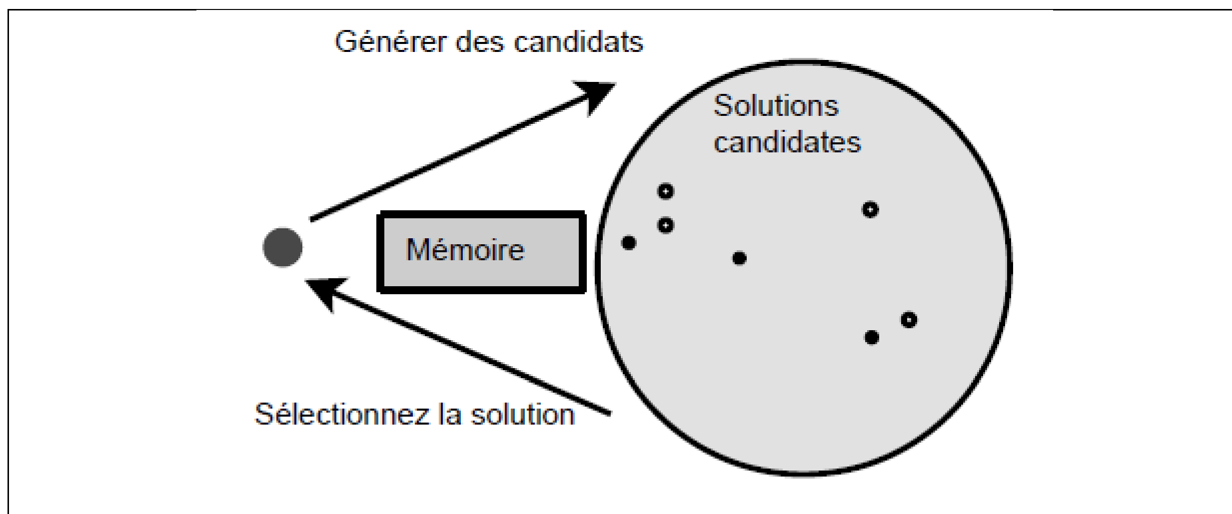


Figure 1-10 Principaux principes des métaheuristiques à base de solution unique.

Les concepts de recherche communs pour toutes les S-métaheuristiques sont la définition de la structure de voisinage et la détermination de la solution initiale.

1.6.1.1 Structure de voisinage

La définition du voisinage est une étape commune requise pour la conception de tout S-métaheuristique. La structure de voisinage joue un rôle crucial dans la performance d'un S-métaheuristique. Si la structure du voisinage n'est pas adaptée au problème, tout S-métaheuristique échouera à résoudre le problème.

Définition 1.1 : *Structure de voisinage* : Une fonction de voisinage N est une application $N: S \rightarrow 2^S$ qui assigne à chaque solution s de S un ensemble de solutions $N(s) \subset S$.

Une solution s' dans le voisinage de s ($s' \in N(S)$) s'appelle un voisin de s. Un voisin est généré par l'application d'un opérateur de déplacement m qui effectue une petite perturbation à la solution s (figure 1.11). La propriété principale qui doit caractériser un voisinage est la localité. La localité est l'effet sur la solution lors de l'exécution du mouvement (perturbation) dans la représentation.

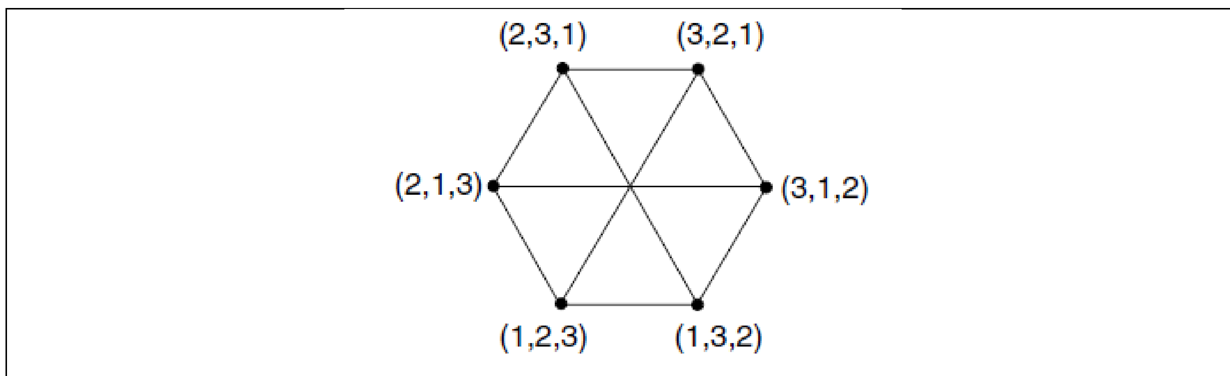


Figure 1-11 Exemple de voisinage pour un problème de permutation de taille 3. Par exemple, les voisins de la solution (2, 3, 1) sont (3, 2, 1), (2, 1, 3) et (1, 3, 2).

Définition1.2 Optimal local : Relativement à une fonction voisine N, un solution $s \in S$ est un optimum local s'il a une meilleure qualité que tous ses voisins; c'est-à-dire $f(s) \leq f(s')$ pour tout $s' \in N(s)$ (figure 1.12).

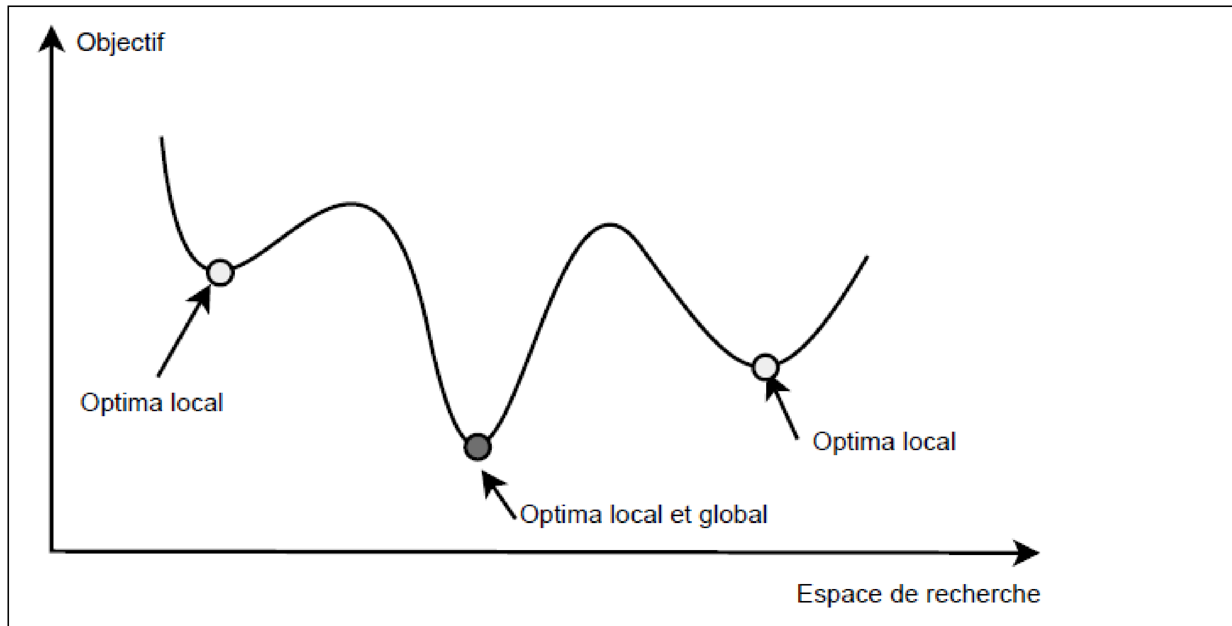


Figure 1-12 Optimum local et optimum global dans un espace de recherche. Un problème peut avoir de nombreuses solutions optimales globales.

1.6.2 Solution initiale

Deux stratégies principales sont utilisées pour générer la solution initiale: une approche aléatoire et une approche gourmande. Il y a toujours un compromis entre l'utilisation de solutions initiales aléatoires et gourmande en termes de qualité des solutions et de temps de calcul. La meilleure réponse à ce compromis dépendra principalement de l'efficacité et de l'efficience des algorithmes aléatoires et gourmands et des propriétés des S-métaheuristiques. Par exemple, plus le voisinage est grand, moins la sensibilité de la solution initiale à la performance des S-métaheuristiques est grande. La génération d'une solution initiale aléatoire est une opération rapide, mais la métaheuristique peut nécessiter un nombre beaucoup plus élevé d'itérations pour converger. Pour accélérer la recherche, une heuristique gloutonne peut être utilisée. En effet, dans la plupart des cas, les algorithmes gloutons ont une complexité polynomiale réduite. L'utilisation d'heuristiques gloutonnes conduit souvent à des optimums locaux de meilleure qualité. Par conséquent, le S-métaheuristique nécessitera, en général, moins d'itérations pour converger vers un optimum local. Quelques approximations d'algorithmes gloutons peuvent également être utilisées pour obtenir une garantie liée à la solution finale. Cependant, cela ne signifie pas que l'utilisation de meilleures solutions comme solutions initiales conduira toujours à de meilleurs optima locaux.

1.6.3 Recherche locale(RL) :

La recherche locale est probablement la méthode métaheuristique la plus ancienne et la plus simple [15]. Cela commence à une solution initiale donnée. À chaque itération, l'heuristique remplace la solution actuelle par un voisin qui améliore la fonction objectif (figure 1.13). La recherche s'arrête lorsque tous les voisins candidats sont plus mauvais que la solution actuelle,

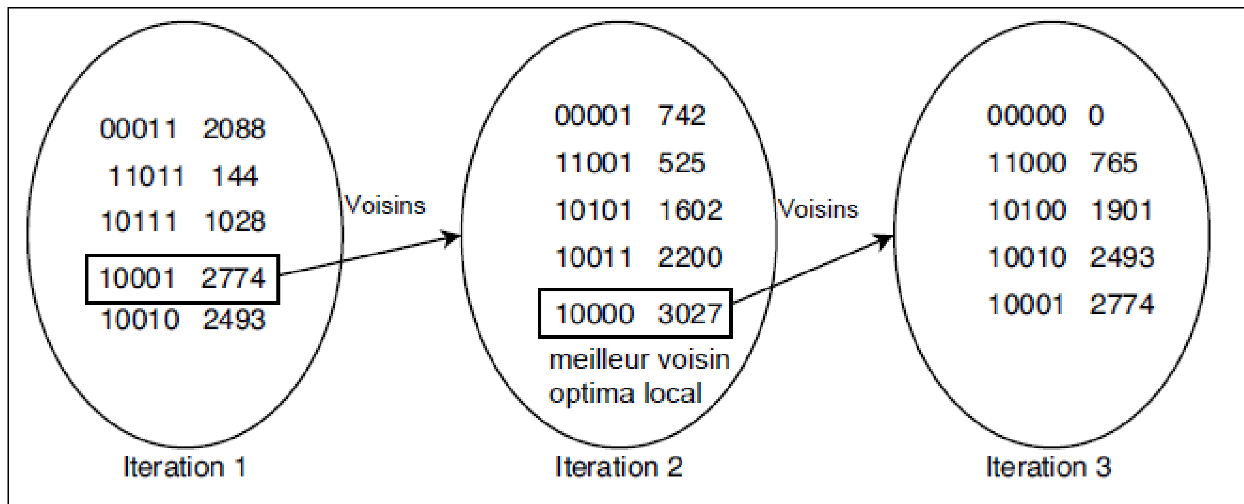


Figure 1-13 Processus de recherche locale utilisant une représentation binaire de solutions, un opérateur de déplacement par retournement (retourner, se déplacer) et la stratégie de sélection du meilleur voisin. La fonction objectif à maximiser est x3 - 60x2 + 900x. La solution optimale globale est f (01010) = f (10) = 4000, tandis que le local final optima trouvé est s = (10000), à partir de la solution s0 = (10001).

ce qui signifie qu'un optimum local est atteint. Pour les grands voisinages, les solutions candidates peuvent constituer un sous-ensemble du voisinage. L'objectif principal de cette stratégie de voisinage restreint est d'accélérer la recherche. Les variantes de RL peuvent être distinguées selon l'ordre de génération des solutions voisines (déterministe / stochastique) et la stratégie de sélection (sélection de la solution voisine). L'algorithme 1.3 illustre le modèle d'un algorithme de recherche locale.

Algorithme 1.3 Modèle d'un algorithme de recherche locale
$s = s_0$; /∗ Générer une solution initiale s0 ∗/
Tanque Critère no Terminer **Faire**
Générer (N(s)) ; /∗ Génération de voisins candidats * / ∗/
Si il n'y a pas de meilleur voisin **Alors** Arrêtez;
s = s′ ; /∗ Choisissez un meilleur voisin s′ ∈ N(s) ∗/
FinTanque
Sortie: Solution finale trouvée (optimum local).

1.6.4 Recuit simulé (simulated annealing (SA))

Le recuit simulé (SA) appliqué à des problèmes d'optimisation émerge du travail de S. Kirkpatrick et al. [16] et V. Cerny [17]. SA a eu un impact majeur sur le domaine de la recherche heuristique pour sa simplicité et son efficacité dans la résolution de problèmes d'optimisation combinatoire. L'algorithme SA simule les changements d'énergie dans un système soumis à un processus de refroidissement jusqu'à ce qu'il converge vers un état d'équilibre (état gelé stable). Ce schéma a été développé en 1953 par Metropolis [18]. SA est un algorithme stochastique qui permet dans certaines conditions la dégradation d'une solution. L'objectif est d'échapper à l'optimum local et donc de retarder la convergence. SA est un algorithme sans mémoire dans le sens où l'algorithme utilise toute information recueillie au cours de la recherche. Depuis une solution initiale, SA procède en plusieurs itérations. À chaque itération, un voisin aléatoire est généré. Les mouvements qui améliorent la fonction de coût sont toujours acceptés. Sinon, le voisin est sélectionné avec une probabilité donnée qui dépend de la température actuelle et de la quantité de dégradation ΔE de la fonction objective. ΔE représente la différence de la valeur objective (énergie) entre la solution actuelle et la solution voisine générée. Au fur et à mesure que l'algorithme progresse, la probabilité que de tels mouvements soient acceptés diminue. Cette probabilité suit, en général, la distribution de Boltzmann:

$$P(\Delta E, T) = \mathrm{e}^{-\frac{f(s')-f(s)}{T}}$$

SA utilise un paramètre de contrôle, appelé température, pour déterminer la probabilité d'accepter des solutions non améliorées. À un niveau de température particulier, de nombreux

essais sont explorés. Une fois l'état d'équilibre atteint, la température diminue progressivement selon un programme de refroidissement tel que peu de solutions non améliorées sont acceptées à la fin de la recherche. L'algorithme 1.4 décrit le modèle de l'algorithme SA.

Algorithme 1.4 Modèle d'algorithme de recuit simulé.

Entrée: Cooling schedule.
$s = s_0$; /∗ Génération de la solution initiale ∗/
$T = T_{max}$; /∗ Température de départ ∗/
Répéter
    Répéter
        /∗ À une température fixe ∗/
        Générer un voisin aléatoire s′ ;
        $\Delta E = f(s') - f(s)$;
        Si $\Delta E \leq 0$ **Alors** s = s′ /∗ Accepter la solution du voisin ∗/
        Sinon Accepter s′ avec une probabilité $e^{-\frac{\Delta E}{T}}$
    Jusqu'à la condition d'équilibre
      /∗ par exemple. un nombre donné d'itérations exécutées à chaque température T ∗/
      $T = g(T)$; /∗ Mise à jour de la température ∗/
Jusqu'à l'arrêt des critères satisfis /∗ e.g. $T < T_{min}$ ∗/
Sortie: Meilleure solution trouvée.

En plus de la solution actuelle, la meilleure solution trouvée depuis le début de la recherche est stockée. Peu de paramètres contrôlent la progression de la recherche, qui sont la température et le nombre d'itérations effectuées à chaque température.

1.6.4.1 Acceptation de mouvement

Le système peut échapper aux optimums locaux en raison de l'acceptation probabiliste d'un voisin non-amélioré. La probabilité d'accepter un voisin non-amélioré est proportionnelle à la température T et inversement proportionnelle au changement de la fonction objective ΔE

1.6.4.2 Processus de refroidissement

Le programme de refroidissement définit pour chaque étape de l'algorithme i la température T_i. Cela a un grand impact sur le succès de l'algorithme d'optimisation SA. En effet, la performance de SA est très sensible au choix du planning de refroidissement. Les paramètres à prendre en compte pour définir un programme de refroidissement sont la température de départ, l'état d'équilibre, une fonction de refroidissement et la température finale qui définit les critères d'arrêt. Ces paramètres sont expliqués ci-après.

1.6.4.3 Température initiale

Si la température de départ est très élevée, la recherche sera plus ou moins une recherche locale aléatoire. Sinon, si la température initiale est très basse, la recherche sera plus ou moins un premier algorithme de recherche locale en amélioration. Par conséquent, nous devons équilibrer entre ces deux procédures extrêmes. La température de départ ne doit pas être trop élevée pour effectuer une recherche aléatoire pendant une période de temps mais suffisamment élevée pour permettre des déplacements presque à l'état de voisinage. Trois stratégies principales peuvent être utilisées pour traiter ce paramètre:

- *Tout accepter*: La température de démarrage est suffisamment élevée pour accepter tous les voisins pendant la phase initiale de l'algorithme [16]. Le principal inconvénient de cette stratégie est son coût de calcul élevé.
- *Déviation d'acceptation*: La température de départ est calculée par $k\sigma$ en utilisant des expérimentations préliminaires, où σ représente l'écart-type de différence entre les valeurs des fonctions objectives et $k = -3 / \ln(p)$ avec la probabilité d'acceptation de p, qui est supérieure à 3σ [19].
- *Taux d'acceptation*: La température de départ est définie de manière à rendre le taux d'acceptation des solutions supérieur à une valeur prédéterminée a_0

$$T_0 = \frac{\Delta^+}{\ln(m_1(a_0 - 1)/m_2 + a_0)}$$

où m_1 et m_2 sont les nombres de solutions à diminuer et à augmenter dans les expériences préliminaires, respectivement, et $\Delta+$ est la moyenne des valeurs de la fonction objective

augmentée [20]. Par exemple, la température initiale doit être initialisée de telle sorte que le taux d'acceptation soit dans l'intervalle [40%, 50%].

1.6.4.3.1 État d'équilibre

Pour atteindre un état d'équilibre à chaque température, un nombre suffisant de transitions (coups) doit être appliqué. La théorie suggère que le nombre d'itérations à chaque température peut être exponentiel à la taille du problème, ce qui est une stratégie difficile à appliquer dans la pratique. Le nombre d'itérations doit être fixé en fonction de la taille de l'instance du problème et particulièrement proportionnel à la taille du voisinage $|N(s)|$. Le nombre de transitions visitées peut être comme suit:

- *Statique*: dans une stratégie statique, le nombre de transitions est déterminé avant le début de la recherche. Par exemple, une proportion y donnée du voisinage *N(s)* est explorée. Par conséquent, le nombre de voisins générés à partir d'une solution *s* est $y \cdot |N(s)|$. Plus le rapport *y* est important, plus le coût de calcul est élevé et meilleurs sont les résultats.
- *Adaptative*: le nombre de voisins générés dépendra des caractéristiques de la recherche. Par exemple, il n'est pas nécessaire d'atteindre l'état d'équilibre à chaque température. Des algorithmes de recuit simulé sans équilibre peuvent être utilisés: le programme de refroidissement peut être appliqué dès qu'une solution voisine s'améliore. Cette caractéristique peut entraîner la réduction du temps de calcul sans compromettre la qualité des solutions obtenues.

1.6.4.3.2 Refroidissement

Dans l'algorithme SA, la température est diminuée progressivement de telle sorte que

$$T_i > 0, \forall i$$

$$\lim_{i \longrightarrow \infty} T_i = 0$$

Il y a toujours un compromis entre la qualité des solutions obtenues et la vitesse du programme de refroidissement. Si la température diminue lentement, de meilleures solutions sont obtenues mais avec un temps de calcul plus important. La température *T* peut être mise à jour de différentes manières:

- *Linéaire*: dans le plan linéaire trivial, la température T est mise à jour comme suit: $T = T - \beta$, où β est une valeur constante spécifiée. Par conséquent, nous avons

$$T_i = T_0 - i \times \beta$$

où T_i représente la température à l'itération i.

- *Géométrique*: Dans le processus géométrique, la température est mise à jour à l'aide de la formule

$$T = \alpha T$$

où $\alpha \in]0, 1[$. C'est la fonction de refroidissement la plus populaire. L'expérience a montré que α devrait être compris entre 0,5 et 0,99.

1.6.4.3.3 Condition d'arrêt

En ce qui concerne la condition d'arrêt, la théorie suggère une température finale égale à 0. En pratique, on peut arrêter la recherche lorsque la probabilité d'accepter un mouvement est négligeable. Les critères d'arrêt suivants peuvent être utilisés:

- Atteindre une température finale *TF* est le critère d'arrêt le plus populaire. Cette température doit être basse (par exemple, $T_{min} = 0,01$).
- Atteindre un nombre prédéterminé d'itérations sans améliorer la meilleure solution trouvée.
- Atteindre un nombre prédéterminé de fois un pourcentage de voisins à chaque température est acceptée; c'est-à-dire qu'un compteur augmente de 1 chaque fois qu'une température est terminée avec le pourcentage de mouvements acceptés inférieur à une limite prédéterminée et est remis à 0 lorsqu'une nouvelle meilleure solution est trouvée. Si le compteur atteint une limite prédéterminée R, l'algorithme SA est arrêté.

1.7 Métaheuristiques basées sur une population de solution

Les métaheuristiques basées sur une population de solution (P-métaheuristiques) partagent de nombreux concepts communs. Elles peuvent être considérées comme une amélioration itérative d'une population de solutions. D'abord, la population est initialisée. Ensuite, une nouvelle population de solutions est générée. Enfin, cette nouvelle population est intégrée dans la nouvelle population en utilisant des procédures de sélection. Le processus de recherche est arrêté lorsqu'une condition donnée est satisfaite (critère d'arrêt) [1].

1.7.1 Concepts communs fondées sur une population de solution

Les métaheuristiques basées sur la population partent d'une population initiale de solutions. Ensuite, elles appliquent itérativement la génération d'une nouvelle population et le remplacement de la population actuelle (figure 1.14). Dans la phase de génération, une nouvelle population de solutions est créée. Dans la phase de remplacement, une sélection est effectuée à partir des populations actuelles et des nouvelles populations.

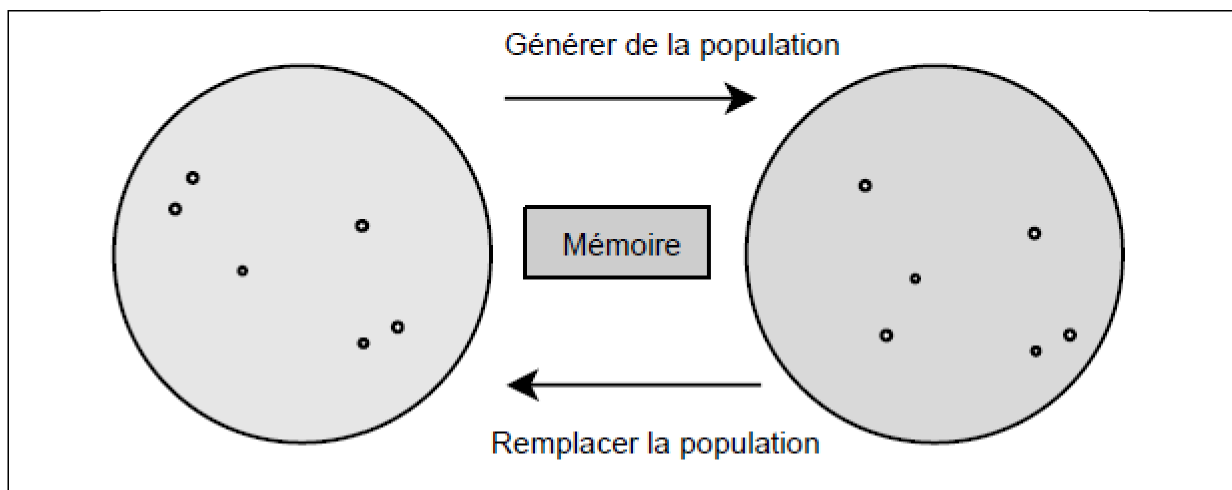


Figure 1-14 Principaux principes de P-métaheuristiques.

Ce processus itère jusqu'à un critère d'arrêt donné. Les phases de génération et de remplacement peuvent être sans mémoire. Dans ce cas, les deux procédures sont basées uniquement sur la population actuelle. Sinon, un historique de la recherche stockée dans une mémoire peut être utilisé dans la génération de la nouvelle population et le remplacement de

l'ancienne population. La plupart des P-métaheuristiques sont des algorithmes inspirés de la nature. Les exemples populaires de P-métaheuristiques sont les algorithmes évolutifs, l'optimisation des colonies de fourmis, la recherche par diffusion, l'optimisation des essaims de particules, la colonie d'abeilles et les systèmes immunitaires artificiels. L'algorithme 1.5 illustre le modèle de haut niveau des P-métaheuristiques.

Algorithme 2.5 Modèle de haut niveau de P-métaheuristique.
$P = P_0$; /∗ Génération de la population initiale ∗/
t = 0 ;
Répéter
Générer($P_{t'}$); /∗ Générer une nouvelle population ∗/
P_{t+1} = Select-Population($P_t \cup P_{t'}$); /∗ Sélectionnez une nouvelle population ∗/
t = t + 1;
Jusqu'à l'arrêt des critères satisfiés
Sortie: Meilleure solution trouvée.

Les P-métaheuristiques diffèrent dans la façon dont ils effectuent la génération et les procédures de sélection et la mémoire de recherche qu'ils utilisent lors de la recherche.

1.7.1.1 Population Initiale

Du fait de la grande diversité des populations initiales, les P-métaheuristiques sont naturellement plus des algorithmes de recherche d'exploration alors que les S-métaheuristiques sont plus des algorithmes de recherche d'exploitation. La détermination de la population initiale est souvent négligée dans la conception d'un P-métaheuristique. Néanmoins, cette étape joue un rôle crucial dans l'efficacité de l'algorithme et son efficacité. Par conséquent, il faut accorder plus d'attention à cette étape [21]. Dans la génération de la population initiale, le principal critère à prendre en compte est la diversification. Si la population initiale n'est pas bien diversifiée, une convergence prématurée peut se produire pour tout P-métaheuristique. Par exemple, ceci peut se produire si la population initiale est générée en utilisant une heuristique gloutonne ou une S-métaheuristique (par exemple, une recherche locale, une recherche de tabous) pour chaque solution de la population.

1.7.1.2 Génération aléatoire

Habituellement, la population initiale est générée de manière aléatoire. Par exemple, en optimisation continue, la valeur réelle initiale de chaque variable peut être générée aléatoirement dans sa gamme possible.

Exemple 1.5 *Population aléatoire uniforme dans l'optimisation globale* : En optimisation continue, chaque variable de décision x_j est définie comme étant dans une plage donnée $[l_j, u_j]$. Générons une population initiale P_0 de taille n distribuée aléatoirement. Chaque solution x_i de la population est un vecteur réel k-dimensionnel x_{ij}, $j \in [1, k]$. Chaque élément du vecteur x_{ij} est généré aléatoirement dans l'intervalle $[l_j, u_j]$, représentant les limites inférieure et supérieure de chaque variable:

$$x_{ij} = l_j + \text{rand}_j[0, 1] \cdot (u_j - l_j),\ i \in [1, n],\ j \in [1, k]$$

où $rand_j$ est une variable aléatoire uniformément distribuée dans l'intervalle [0, 1]. Si les limites (limites inférieure et supérieure) ne sont pas bien définies, les limites doivent être initialisées suffisamment grandes pour englober l'espace de recherche délimité par les solutions optimales.

1.7.1.3 Critères d'arrêt

De nombreux critères d'arrêt basés sur l'évolution d'une population peuvent être utilisés. Certains d'entre eux sont similaires à ceux conçus pour la S-métaheuristique.

- *Procédure statique*: Dans une procédure statique, la fin de la recherche peut être connue a priori. Par exemple, on peut utiliser un nombre fixe d'itérations (générations), une limite sur les ressources CPU, ou un nombre maximum d'évaluations de fonctions objectives.
- *Procédure adaptative*: Dans une procédure adaptative, la fin de la recherche ne peut être connue a priori. On peut utiliser un nombre fixe d'itérations (générations) sans amélioration, lorsqu'une solution optimale ou satisfaisante est atteinte (par ex. une erreur donnée à l'optimum ou une approximation quand une borne inférieure est connu auparavant).

1.7.2 Algorithmes génétiques

Des algorithmes génétiques ont été développés par J. Holland dans les années 1970 pour comprendre les processus adaptatifs des systèmes naturels [31]. Ensuite, ils ont été appliqués à l'optimisation et à l'apprentissage automatique dans les années 1980 [32]. Les GA sont associés à l'utilisation d'une représentation binaire mais de nos jours on peut trouver des GA qui utilisent d'autres types de représentations. Une AG applique généralement un opérateur de croisement à deux solutions qui jouent un rôle majeur, plus un opérateur de mutation qui modifie aléatoirement les contenus individuels pour promouvoir la diversité. Les GA utilisent une sélection probabiliste qui est à l'origine la sélection proportionnelle. Le remplacement (sélection du survivant) est générationnel, c'est-à-dire que les parents sont systématiquement remplacés par les descendants. L'opérateur de croisement est basé sur le croisement à points n ou uniformes tandis que la mutation est un retournement de bit. Une probabilité fixe p_m (resp. P_c) est appliquée à l'opérateur de mutation (resp. croisement). L'algorithme 1.6 illustre le modèle générale des algorithmes génétiques.

Algorithme 1.6 Modèle générale d'un algorithme génétique

Début
Générer la population initiale
Calculer la fonction objective
Répéter
    Sélection
    Croisement
    Mutation
    Calculer la fonction objective
Jusqu'à ce que la population ait convergé
Arrêtez

1.7.3 Algorithme bactérien du recherche de la nourriture (bacterial foraging optimisation algorithm(BFOA))

BFOA est une technique bio inspirée [22]. Elle a été appliquée pour modéliser le comportement de recherche d'Escherichia coli afin de résoudre les problèmes d'optimisation. Les bactéries ont tendance à migrer vers les zones riches en nutriments et ce comportement est appelé «chimiotaxie». Ce mouvement est obtenu en faisant tourner des flagelles en forme de fouet qui sont entraînées par un moteur réversible intégré dans la paroi de la cellule. Escherichia coli a 8-10 flagelles placés au hasard sur un corps cellulaire. Lorsque tous les flagelles tournent dans le sens inverse des aiguilles d'une montre, ils forment un compact qui peut propulser la cellule le long d'une trajectoire hélicoïdale, appelée course. Lorsque les flagelles tournent dans le sens des aiguilles d'une montre, ils tirent tous sur les directions indifférentes de la bactérie et font dégringoler les bactéries. Le cycle d'optimisation peut être divisé en trois parties: chimiotaxie, reproduction, élimination et dispersion.

1.7.3.1 Chimiotaxie (Chemotaxis)

La chimiotaxie est un mouvement cellulaire en réponse à des gradients de concentrations chimiques dans l'environnement. En tant que stratégie de survie, ce mouvement est réalisé en nageant et culbutant à l'aide de flagelles, où chaque flagelle est entraîné comme un moteur biologique. Une bactérie E. coli se déplace en alternant la natation et le culbutage. Il peut soit nager pendant un moment dans la même direction, soit il dégringolera probablement. En basculant entre les deux modes, la bactérie est capable de se déplacer dans des directions aléatoires qui lui permettent de rechercher des nutriments.

1.7.3.2 Essaimage (Swarming)

Alors qu'une bactérie se déplace vers la zone optimale pour la nourriture, il est toujours désireux de trouver la meilleure position ainsi que de produire de forts signaux d'attraction pour d'autres bactéries. Cela leur permettra de se rassembler pour atteindre la zone désirée. Au cours de ce processus, les cellules d'E.coli libèrent un attractif pour les aider à se regrouper. Ensuite, ils se déplacent en mode concentré de population à forte densité.

1.7.3.3 Reproduction

Soit N_c la durée de vie d'une bactérie, mesurée par le nombre d'étapes de chimiotaxie qu'ils exécutent au cours de leur vie. Lorsque toutes les étapes de la chimiotaxie N_c sont terminées, la reproduction est lancée. La valeur sanitaire de la bactérie *i* peut être représentée par la somme de la forme physique de l'étape durant sa vie

1.7.3.4 Élimination et dispersion

Les bactéries seront grandement influencées par l'environnement. Tout comme la population bactérienne va changer en réponse à l'appauvrissement des aliments dû à leur propre consommation, l'opération de dispersion se produira après un certain nombre de processus de reproduction, une probabilité donnée P_{ed} est utilisée pour décrire un choix de bactérie (dispersion ou non). Si certaines bactéries satisfont à la règle de dispersion, elles seront supprimées ou remplacées. Puisque la bactérie essaie d'atteindre le niveau de nutriments sans satisfaction, elle recherchera toujours une plus grande concentration de nutriments. Ainsi, les trois comportements précédents, la chimiotaxie, la reproduction, et le processus d'élimination et de dispersion, sont continus jusqu'à ce que les critères de finition soient atteints.

1.7.4 L'Algorithme de batte (Bat Algorithm)

L'algorithme de batte (BA) est une méta-heuristique bio-inspirée basée sur le système d'écholocation des battes. Elle est utilisée récemment pour résoudre beaucoup de problème d'optimisation [30]. Dans la nature, les battes émettent des impulsions ultrasoniques dans l'environnement environnant à des fins de chasse et de navigation. Après l'émission de ces impulsions, les battes écoutent les échos et, à partir des ces échos, elles peuvent se localiser et identifier les obstacles et les proies. En outre, chaque batte de l'essaim est capable de trouver les zones les plus "nutritives" qui effectuent une recherche individuelle, ou se déplaçant vers un "endroit nutritif précédemment trouvé par l'essaim. L'idée principale de la BA est d'imiter ce système d'écholocation des battes. Quoi qu'il en soit, certaines règles idéalisées doivent être prises en compte pour une adaptation adéquate [23]:

- Toutes les battes utilisent l'écholocation pour détecter la distance, et elles ont une «capacité magique» qui leur permet de faire la différence entre une proie et un obstacle.

- Les battes volent avec une certaine position de la cuve de vitesse x. Ils émettent des impulsions dont la fréquence est ajustée automatiquement et le débit de l'impulsion émise $r \in [0,1]$ est ajusté en fonction de la distance de la batte par rapport à sa cible.
- Le volume des battes varie d'une grande valeur A_0 à une valeur minimale A_{min}.

1.7.5 Recherche de coucou (Cuckoo Search (CS))

La recherche Coucou (CS) est une métaheuristiques inspirés par la nature, développé en 2009 par Xin-She Yang et Suash Deb [24]. CS est basé sur le parasitisme de la couvée de certaines espèces de coucous. De plus, cet algorithme est amélioré par les vols Levy, plutôt que par de simples marches aléatoires isotropes. Les coucous pondent leurs œufs dans des nids communaux, bien qu'ils puissent enlever d'autres œufs pour augmenter la probabilité d'éclosion de leurs propres œufs. Pour simplifier la description de la recherche coucou standard, nous utilisons maintenant les trois règles idéalisées suivantes:

- Chaque coucou pond un œuf à la fois et le dépose dans un nid choisi au hasard;
- Les meilleurs nids avec des œufs de haute qualité seront transférés aux générations suivantes;
- Le nombre de nids d'hôtes disponibles est fixe, et l'œuf pondu par un coucou est découvert par l'oiseau hôte avec une probabilité de probabilité (0,1). Dans ce cas, l'oiseau hôte peut soit se débarrasser de l'œuf, soit simplement abandonner le nid et construire un nid complètement nouveau.

1.8 Conclusion

Nous avons présenté dans ce chapitre les notions lié au problème combinatoire, puis nous avons introduire les deux types des métaheuristiques, S-métaheuristique et P-métaheuristique. Ainsi que leurs concepts communs. Enfin nous avons donné une brève présentation des algorithmes utilisés dans notre contribution.

Chapitre 2: Les systèmes distribués

2.1 Introduction

Les systèmes distribués forment un domaine informatique en constante évolution par exemple pour les réseaux pairs à pairs et les réseaux de capteurs, tandis que d'autres sont devenus beaucoup plus mature, comme les services web et les applications web en général.

2.1.1 Définition d'un système distribué

Diverses définitions des systèmes distribués ont été données dans la littérature, aucun d'eux satisfaisant, et aucun d'entre eux est d'accord avec aucun des autres.

Pour nos besoins, il suffit de donner une caractérisation lâche:

« *Un système distribué est une collection d'ordinateurs indépendants qui apparaît à ses utilisateurs comme un seul système cohérent.* » [85]

« Un système dans lequel des composants matériels ou logiciels situés sur des ordinateurs où équipements intelligents en réseau communiquer et coordonner leurs actions uniquement par la transmission de messages »[34]

Cette définition a plusieurs aspects importants. Le premier est qu'un système distribué est constitué de composants (ordinateurs) autonomes. Un second aspect est que les utilisateurs (qu'ils soient des personnes ou des programmes) pensent qu'ils interagirent avec un seul système. Cela signifie que d'une manière ou d'une autre les composants autonomes besoin de collaborer. Un système distribué peut être composé d’ordinateurs de hautes performances à de petits nœuds dans des réseaux de capteurs. Également, Aucune hypothèse n'est faite sur la manière dont les ordinateurs sont interconnectés.

En principe, les systèmes distribués devraient également être relativement faciles à étendre. Cette caractéristique est une conséquence directe de l'existence d'ordinateurs indépendants. Un système distribué sera normalement disponible en permanence, bien que certaines parties soient peut-être temporairement hors service où les utilisateurs et les applications ne devraient pas remarquer que les pièces sont remplacées ou réparées, ou que de nouvelles pièces sont ajoutées pour servir plus d'utilisateurs ou d'applications.

Afin de prendre en charge des ordinateurs et des réseaux hétérogènes tout en offrant une vue unique, les systèmes distribués sont souvent organisés au moyen d'une couche de logiciel, c'est-à-dire logiquement placé entre une couche de plus haut niveau composée d'utilisateurs et des applications, et une couche inférieure constituée de systèmes d'exploitation et des infrastructures de communication, comme le montre la figure 2.1 . En conséquence, une telle distribution de système est parfois appelé middleware. Le middleware est une couche qui s'étend sur plusieurs machines et offre à chaque application la même interface.

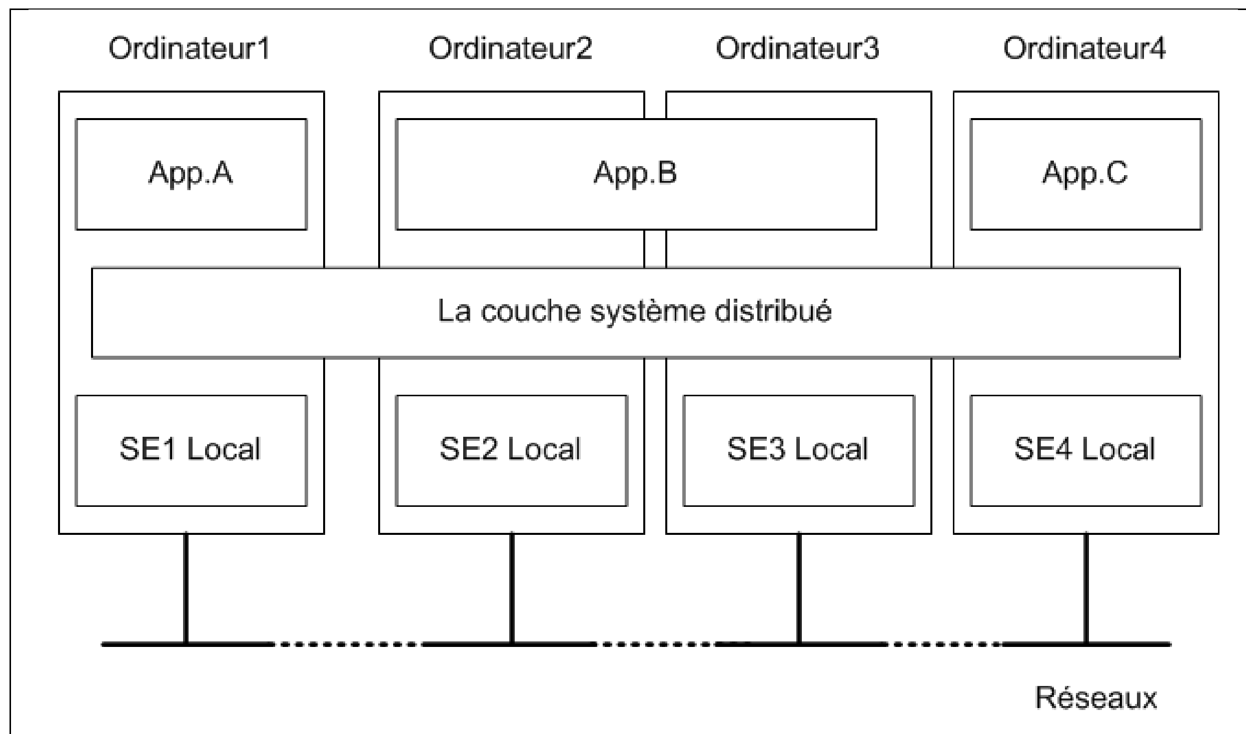


Figure 2-1 Un système distribué organisé en middleware.

La figure 2.1 montre quatre ordinateurs en réseau et trois applications, dont l'application B est réparti entre les ordinateurs 2 et 3. La même interface est offerte à chaque application. Le système distribué fournit les moyens pour les composants des applications distribuées pour communiquer les uns avec les autres, mais aussi pour laisser les différentes applications se communiquent. En même temps, il cache les différences dans le matériel et les systèmes d'exploitation de chaque application.

2.2 L'objectif d'un système distribué

- *Rendre les ressources accessibles* : L'objectif principal d'un système distribué est de le rendre facile pour les utilisateurs et les applications d'accéder à des ressources distantes et les partager de manière contrôlée et efficace. Les ressources peuvent être à peu près tout, mais les exemples typiques comprennent des choses comme les imprimantes, les ordinateurs, les données, les fichiers, les pages Web et les réseaux, pour n'en nommer que quelques-uns. Il y a plusieurs raisons de vouloir partager des ressources. Une raison évidente est celle de l'économie. Par exemple, il est moins onéreux de laisser une imprimante partagée par plusieurs utilisateurs dans un petit bureau que d'avoir à acheter et à entretenir une imprimante distincte pour chaque utilisateur. De même, il est économiquement sensé de partager des ressources coûteuses telles que des superordinateurs, des systèmes de stockage à haute performance, des dispositifs de mise en image et d'autres périphériques coûteux.

  La connexion des utilisateurs et des ressources facilite également la collaboration et l'échange d'informations, comme l'illustre clairement le succès d'Internet avec ses protocoles simples d'échange de fichiers et de courrier, documents, audio et vidéo. La connectivité d'Internet conduit maintenant à de nombreuses organisations virtuelles dans lesquelles des groupes de personnes géographiquement dispersés travaillent ensemble au moyen d'un groupware, c'est-à-dire d'un logiciel d'édition collaborative, de téléconférence, etc. De même, la connectivité Internet a permis au commerce électronique de nous permettre d'acheter et de vendre toutes sortes de biens sans avoir à aller dans un magasin avant de partir.

  Cependant, à mesure que la connectivité et le partage augmentent, la sécurité devient de plus en plus importante. Dans la pratique actuelle, les systèmes offrent peu de protection contre l'écoute indiscrète ou l'intrusion dans la communication. Les mots de passe et autres informations sensibles sont souvent envoyés en tant que texte non cryptés à travers le réseau, ou stockés sur des serveurs que nous pouvons seulement espérer être dignes de confiance. En ce sens, il y a beaucoup de place à l'amélioration. Par exemple, il est actuellement possible de commander des marchandises en fournissant simplement un numéro de carte de crédit.

- *Transparence de la distribution* : Un objectif important d'un système distribué est de cacher le fait que ses processus et les ressources sont physiquement distribués sur plusieurs ordinateurs, un tel système est dit transparent. Le concept de transparence peut être appliqué à plusieurs aspects d'un système distribué, telle que celles représentés sur la Tableau 2.1.

Tableau 2-1 Différentes formes de transparence dans un système distribué (ISO, 1995).

Transparence	**Description**
Accès	Cacher les différences dans la représentation des données et la façon dont une ressource est accessible
Localisation	Cacher où une ressource est situé
Migration	Cacher qu'une ressource peut se déplacer vers un autre emplacement
Relocalisation	Cacher qu'une ressource peut être déplacée vers un autre emplacement en cours d'utilisation
Réplication	masquer qu'une ressource est répliqué
Concurrence	cacher qu'une ressource peut être partagé par plusieurs utilisateurs compétitifs
Echec	cacher l'échec et la récupération de la ressource

- *Ouverture* : Un autre objectif important des systèmes distribués est l'ouverture. Un système distribué est ouvert et offre des services selon des règles standard décrire la syntaxe et la sémantique de ces services. Par exemple, dans les architectures réseaux, les règles standard régissent le format, le contenu et la signification des messages envoyé et reçu. Ces règles sont formalisées dans des protocoles.
- *Évolutivité (scalability)* : l'évolutivité d'un système peut être mesurée selon au moins trois dimensions différentes [86]. Tout d'abord, un système peut être évolutif par rapport à sa taille, ce qui signifie que nous pouvons facilement ajouter plus d'utilisateurs et de ressources au système. Deuxièmement, un système géographiquement évolutif est celui dans lequel les utilisateurs et les ressources peuvent se situer loin d'une part. Troisièmement, un système peut être administrativement évolutif, sachant qu'il peut encore être facile à gérer même s'il couvre de nombreuses organisations administratives indépendantes. Malheureusement, un système qui est évolutif dans une ou plusieurs de ces dimensions souvent présente une perte de performance à mesure que le système augmente.

- *Problèmes d'évolutivité (mise à l'échelle)* : lorsqu'un système doit évoluer, des types de problèmes très différents doivent être résolus. Considérons d'abord la mise à l'échelle en fonction de la taille. Si plus d'utilisateurs ou de ressources on besoin d'être soutenu, nous sommes souvent confrontés aux limites de la centralisation des services, données et algorithmes (voir Tableau 2.2). Par exemple, de nombreux services sont centralisés dans le sens où ils sont mis en œuvre au moyen d'un seul serveur fonctionnant sur une machine spécifique dans le système distribué. Le problème avec ce schéma est évident: le serveur peut devenir un goulot d'étranglement quand le nombre d'utilisateurs et les applications se développent. Même si nous avons une capacité de traitement et de stockage pratiquement illimitée, la communication avec ce serveur finira par interdire la poursuite de la croissance.

Tableau 2-2 Exemples de limitations d'évolutivité

Concept	Exemple
Services centralisés	Un seul serveur pour tous les utilisateurs
Données centralisés	Un seul annuaire téléphonique en ligne
Algorithmes centralisés	Faire le routage basé sur des informations complètes

L'évolutivité géographique a ses propres problèmes. L'une des principales raisons pour lesquelles il est actuellement difficile d'adapter les systèmes distribués existants conçus pour les réseaux locaux et qui sont basés sur une communication synchrone. Dans cette forme de communication, une partie demandant un service, généralement appelée client, se bloque jusqu'à ce qu'une réponse soit renvoyée. Cette approche fonctionne généralement bien dans les réseaux locaux où la communication entre deux machines est généralement au pire occupe quelques centaines de microsecondes. Cependant, dans un système étendu, nous devons prendre en compte le fait que la communication interprocessus peut prendre plusieurs centaines de millisecondes, trois fois plus lente. Construire des applications interactives en utilisant la communication synchrone dans des systèmes étendus nécessite beaucoup de soin.Un autre problème qui entrave l'évolutivité géographique est que la communication dans les réseaux étendus est intrinsèquement peu fiable et pratiquement toujours point à point. En revanche, les réseaux locaux fournissent généralement une communication très fiable qui basées sur la diffusion, ce qui facilite beaucoup le développement de systèmes distribués. Par exemple, considérez le problème de localisation d'un service. Dans un

système local, un processus peut simplement diffuser un message à la machine, demandant si elle est exécuter le service dont il a besoin. Seules les machines qui ont le service répondent, chacun fournissant son adresse réseau dans le message de réponse. Un tel système de localisation est impensable dans un système étendu: imaginez ce qui se passerait si nous essayions pour localiser un service de cette façon sur Internet. Au lieu de cela, les services de localisation spéciaux ont besoin être conçu, qui peut avoir besoin de s'étendre dans le monde entier et être capable de milliards d'utilisateurs

- Techniques de mise à l'échelle : dans la plupart des cas, les problèmes d'évolutivité dans les systèmes répartis apparaissent comme des problèmes de performance causés par la capacité limitée des serveurs et du réseau. Pour remédier au problème de d'évolutivité, trois techniques sont utilisées. La première consiste à cacher les latences de communication, la deuxième sur la distribution et la troisième sur la réplication [86].

2.2.1 Point Faible

Cependant, les systèmes distribués diffèrent des systèmes traditionnels car les composants sont dispersés sur un réseau. Ce qui veut dire, ne pas prendre en compte cette dispersion pendant la conception rend un tel système inutilement complexe avec des résultats pleins d’erreurs qui doivent être corrigées plus tard. Les hypothèses fausses suivantes que tout le monde le fait lors du développement d'une application distribuée pour la première fois et qui nous devons les évités:

- Le réseau est fiable.
- Le réseau est sécurisé.
- Le réseau est homogène.
- La topologie ne change pas.
- La latence est nulle.
- La bande passante est infinie.
- Le coût du transport est nul.
- Il y a un administrateur.

Notez comment ces hypothèses se rapportent aux propriétés uniques des systèmes distribués suivantes: la fiabilité, la sécurité, l'hétérogénéité et la topologie du réseau; la latence et la

bande passante; les coûts de transport; et enfin les domaines administratifs. Par exemple, si les réseaux ne sont pas fiables, cela va conduire à l'impossibilité de la transparence. Si les réseaux ne sont pas l'hétérogène, cela va conduire à des systèmes limités. Aussi, la réplication assure l'évolutivité, la latence assure la disponibilité et la bande passante assure la qualité de service. La sécurité assure la protection des données et des ressources, les coûts de transport inclure la charge sur le réseau et enfin les domaines administratifs qui supportent réellement les unités organisationnelles sous réseau.

2.3 Types de systèmes distribués

Différents types de systèmes distribués peuvent être cité. Dans ce qui suit, nous faisons une distinction entre les systèmes de calcul distribués, les systèmes d'information distribués, et systèmes distribués pervasifs.

2.3.1 Systèmes de calcul distribués

Une classe importante de systèmes distribués est celle utilisée pour supporter les tâches informatiques de haute performance. Aussi, On peut faire une distinction entre deux sous-groupes. Le premier groupe englobe les systèmes informatiques de cluster. Dans ces systèmes, le matériel de calcul sous-jacent consiste en une collection de postes de travail similaires ou PC, étroitement liés au moyen d'un réseau local de haut débit. En outre, chaque nœud exécute le même système d'exploitation. La situation devient très différente dans le cas de l'informatique en grille. Ce sous-groupe comprend les systèmes distribués qui sont souvent construits en tant que systèmes informatiques fédéraux, où chaque système peut relever d'une administration du domaine, et peut être très différent en ce qui concerne le matériel, les logiciels et la technologie de réseau déployée.

- *Systèmes informatiques de cluster* : les systèmes informatiques en cluster sont devenus populaires a cause du leurs rapport prix / performance d'ordinateurs personnels et de postes de travail améliorés. À un certain moment, il est devenu financièrement et techniquement attrayant pour construire un superordinateur en connectant simplement une collection d'ordinateurs relativement simples dans un réseau à haute vitesse. Dans presque tous les cas, l'informatique en cluster (voir Figure 2.2) est utilisée pour accélérer l'exécution des programmes dans laquelle un seul programme (calcul intensif) est exécuté en parallèle sur plusieurs machines.

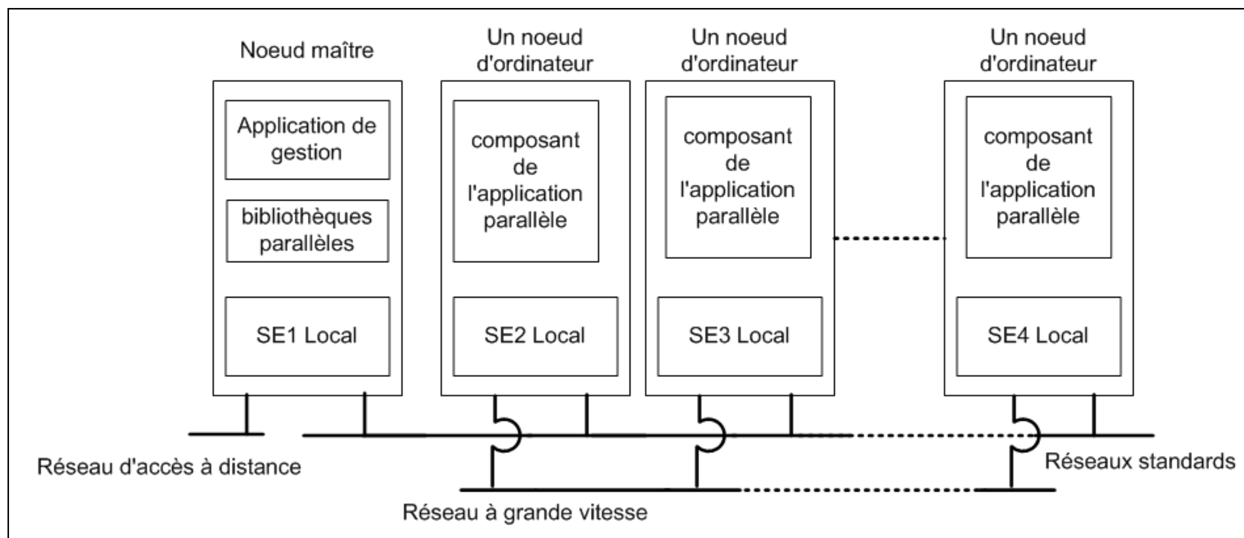


Figure 2-2 Un exemple de système informatique en cluster

- *Systèmes informatiques de grille* : une caractéristique des systèmes informatique en grille est son homogénéité. Dans la plupart des cas, les ordinateurs dans un cluster sont en grande partie les mêmes, ils ont tous le même fonctionnement, et sont tous connectés via le même réseau. En revanche, les systèmes informatiques de grille ont un haut degré d'hétérogénéité: aucune hypothèse n'est faite concernant le matériel, les systèmes d'exploitation, les réseaux, les domaines administratifs, la sécurité politiques, etc. Un problème clé dans un système informatique en grille est que les ressources de différentes organisations sont réunies pour permettre la collaboration d'un groupe de personnes ou institutions. Une telle collaboration est réalisée sous la forme d'une organisation virtuelle. Les personnes appartenant à la même organisation virtuelle ont des droits d'accès aux ressources qui sont fournis à cette organisation. Typiquement, les ressources consistent en serveurs de calcul (y compris les superordinateurs, éventuellement mis en œuvre en tant qu'ordinateurs de cluster). En outre, des appareils en réseau spéciaux tels que des télescopes, des capteurs, etc., peuvent également être fournis. Compte tenu de sa nature, une grande partie du logiciel pour réaliser l'informatique en grille évolue autour de l'accès aux ressources de différents domaines administratifs, et uniquement aux utilisateurs et aux applications appartenant à une organisation virtuelle spécifique. Pour cette raison, l'accent est souvent mis sur les

problèmes architecturaux. Une architecture proposée par [87] est montré dans la Figure 2.3

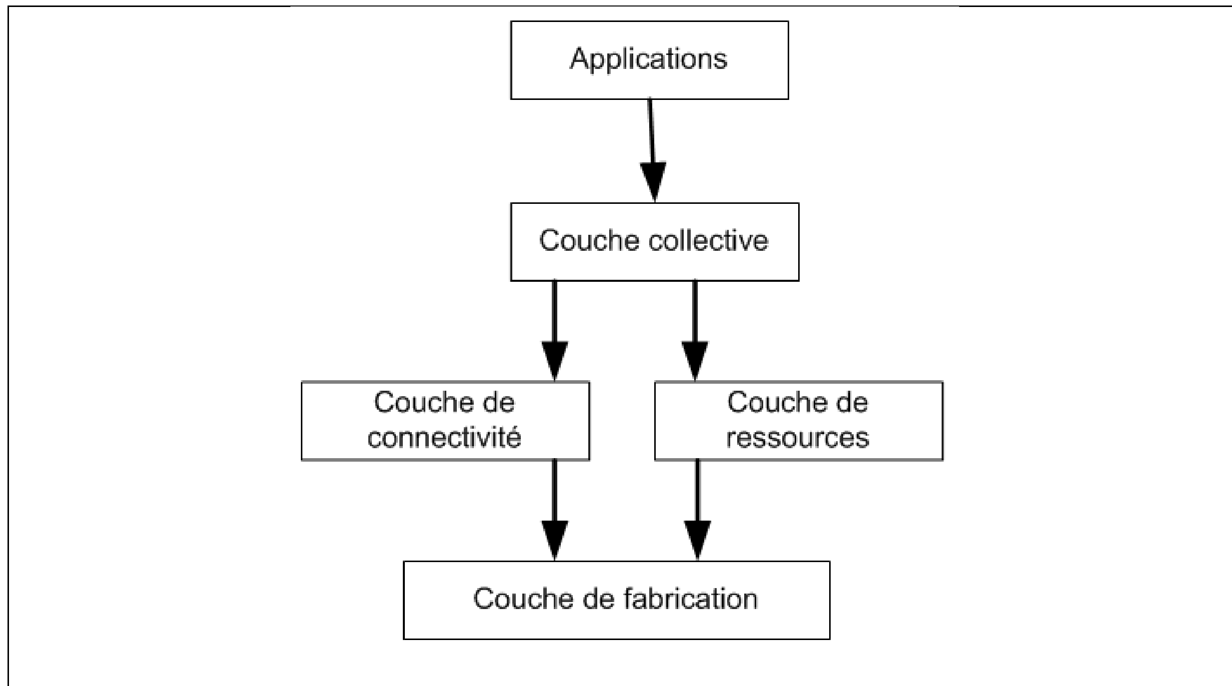


Figure 2-3 Une architecture en couches pour les systèmes de calcul en grille

.

L'architecture est composée de quatre couches. La couche de fabrication (fabric layer) fournit des interfaces aux ressources locales sur un site spécifique. Notez que ces interfaces sont adaptées pour permettre le partage des ressources au sein d'une organisation virtuelle. Typiquement, elles vont fournir des fonctions pour interroger l'état et les capacités d'une ressource, ainsi que des fonctions pour la gestion de ressources réelles (par exemple, le verrouillage de ressources).

La couche de connectivité (connectivity layer) est constituée de protocoles de communication destinés aux transactions de grille qui couvrent l'utilisation de plusieurs ressources. Par exemple, les protocoles sont nécessaires pour transférer des données entre les ressources, ou simplement accéder à une ressource d'un emplacement distant. En outre, la couche de connectivité contiendra des protocoles de sécurité pour authentifier les utilisateurs et les ressources.

La couche de ressources (ressource layer) est responsable de la gestion d'une seule ressource à la fois. Elle utilise les fonctions fournies par la couche de connectivité et

appelle directement les interfaces fournissent par la couche de fabrication. Par exemple, cette couche offrira des fonctions pour obtenir des informations de configuration sur une ressource spécifique, ou, en général, des opérations spécifiques telles que la création d'un processus ou la lecture de données. La couche ressource est donc considérée comme responsable du contrôle d'accès, et dépendra donc de authentification effectuée dans le cadre de la couche de connectivité.

La couche suivante dans la hiérarchie est la couche collective (collective layer). Elle traite la manipulation d'accès aux plusieurs ressources et elle se compose généralement des services pour la découverte de ressource, allocation et ordonnancement de tâches sur plusieurs ressources, réplication de données, etc. Contrairement à la couche de connectivité et de ressource, qui consiste de petite collection de protocoles standards, la couche collective peut consister de nombreux protocoles différents pour de nombreux objectifs différents, reflétant le large spectre des services qu'il peut offrir à une organisation virtuelle.

Enfin, la couche application (application layer) comprend les applications qui fonctionnent dans une organisation virtuelle et qui font usage de l'environnement informatique de grille.

2.3.2 Systèmes d'information distribués

Une autre classe importante de systèmes distribués se trouve dans les organisations qui ont été confrontés à une multitude d'applications en réseau, mais pour lesquelles l'interopérabilité avéré être une expérience dure. Beaucoup de middleware existant sont le résultat et de travailler avec une infrastructure dans laquelle il était plus facile d’intégrer des applications dans un système d'information à l'échelle de l'entreprise

2.3.3 Systèmes distribués pervasifs

Les systèmes distribués sont caractérisés par leur stabilité: les nœuds sont fixes et ont un fonctionnement plus ou moins permanent avec une connexion à un réseau de haute qualité. Dans une certaine mesure, cette stabilité a été réalisée à travers les différentes techniques qui viser la transparence de la distribution, ce qui permet aux utilisateurs et aux applications de crois que les nœuds restent en place.

Cependant, les choses sont devenues très différentes avec l'introduction des dispositifs informatiques embarqués et mobiles. Nous sommes maintenant confrontés à des systèmes distribués dans lequel l'instabilité est le comportement par défaut, ce qui référer à des systèmes omniprésents (pervasif) distribués, qui se caractérisent souvent par être petit, alimenté par batterie, mobile, et ayant seulement une connexion sans fil.

Un système distribué pervasif est devenu une partie de notre environnement avec une caractéristique importante qui est le manque de contrôle administratif humain. Où l'intervention humaine apparait uniquement lors de la configuration de ces appareils, mais sinon ils ont besoin de découvrir automatiquement leur environnement en formulant les trois exigences suivantes pour les applications pervasifs [88]:

- Embrasser les changements contextuels.
- Encourager la composition ad hoc.
- Reconnaissez le partage par défaut.

L'adoption de changements contextuels signifie qu'un périphérique doit être continuellement conscient du fait que son environnement peut changer tout le temps. L'un des plus simples changements est la découvre qu'un réseau n'est plus disponible parce qu'un utilisateur se déplace entre les stations de base. Dans un tel cas, l'application devrait réagir, éventuellement en se connectant automatiquement à un autre réseau ou en autres actions appropriées.

Encourager la composition ad hoc fait référence au fait que de nombreux dispositifs dans les systèmes pervasifs seront utilisés de différentes manières par différents utilisateurs. Par conséquent, il devrait être facile de configurer la suite d'applications exécutées sur un périphérique, soit par l'utilisateur ou par interposition automatique (mais contrôlée).

2.4 Architectures des systèmes distribués

Les systèmes distribués sont souvent des logiciels complexes dont les composants sont par définition dispersés sur plusieurs machines. Pour maîtriser leur complexité, il est crucial que ces systèmes soient correctement organisés. L'organisation des systèmes distribués concerne principalement les composants logiciels qui constituent le système. Ces architectures logicielles nous disent comment divers composants logiciels doivent être organisés et comment ils doivent interagir.

2.4.1 Styles Architecturaux

La notion de style architectural est importante. Elle est formulé en termes de composants, la façon dont les composants sont connectés les uns aux autres, les données échangées entre les composants, et enfin comment ces éléments sont configurés conjointement dans un système. Un composant est une unité modulaire avec des interfaces entrées et sorties bien définies qui peut être remplacée dans son environnement. Un connecteur est généralement décrit comme un mécanisme qui assure la communication, la coordination ou la coopération entre les composants. En utilisant des composants et des connecteurs, nous pouvons arriver à différentes configurations, qui, à leur tour, ont été classés en styles architecturaux. Plusieurs styles peut être identifiés, dont les plus importants pour les systèmes distribués sont:

- Les architectures en couches
- Architectures basées sur des objets
- Architectures centrées sur les données
- Architectures basées sur des événements

L'idée de base pour le style en couches est simple: les composants sont organisés dans un mode en couches où un composant à la couche L; est autorisé à appeler des composants à la couche sous-jacente L_i, mais pas l'inverse, comme le montre la figure 2.4 (a). Dans les architectures à base d'objets, comme il est illustré sur la figure 2.4 (b), chaque objet correspond à un composant, et ces composants sont connectés via un mécanisme d'appel de procédure distant.

Les architectures centrées sur les données évoluent autour de l'idée que les processus communiquent à travers un référentiel commun (passif ou actif). Par exemple, les systèmes distribués basés sur le web sont largement centrés sur les données: les processus communiquent grâce à l'utilisation de services de données web partagés.

Dans les architectures basées sur les événements, les processus communiquent essentiellement à travers la propagation d'événements, qui portent également éventuellement des données, comme le montre la figure 2.5. L'idée de base est que les processus publient des événements après lesquels le middleware s'assure que seuls les processus qui ont souscrit à ces événements les recevront. Le principal avantage des systèmes basés sur les événements est que les processus sont faiblement couplés.

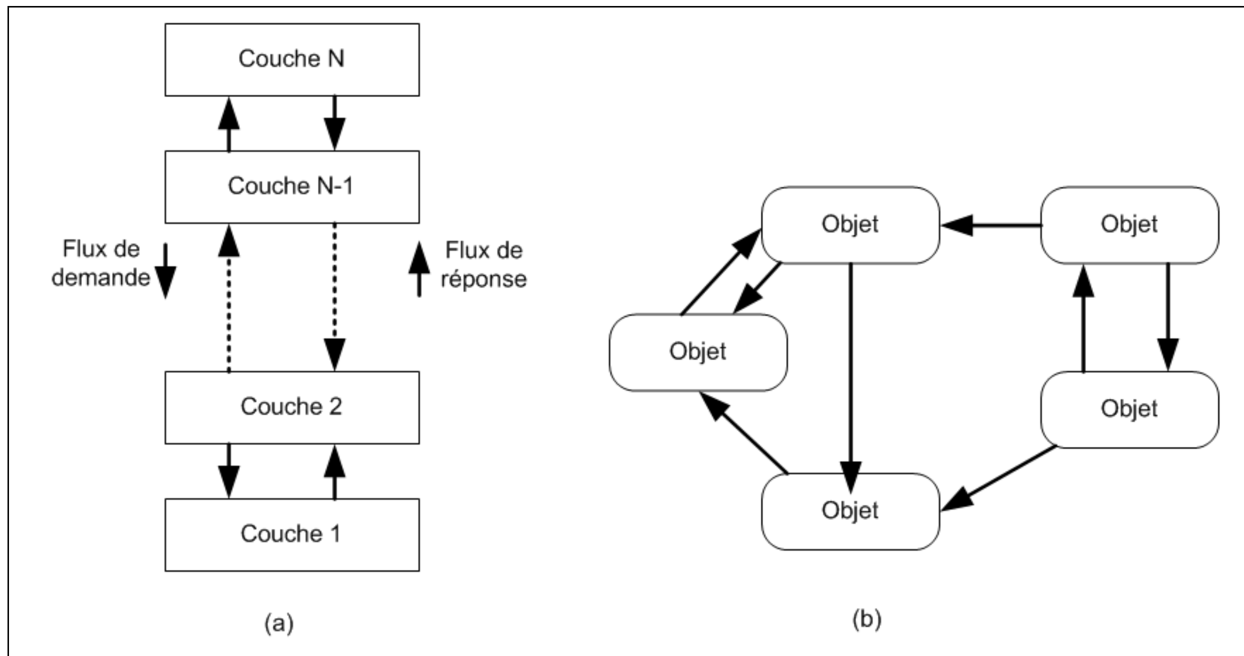


Figure 2-4 Le style (a) basé sur couche (b) a basé sur objet.

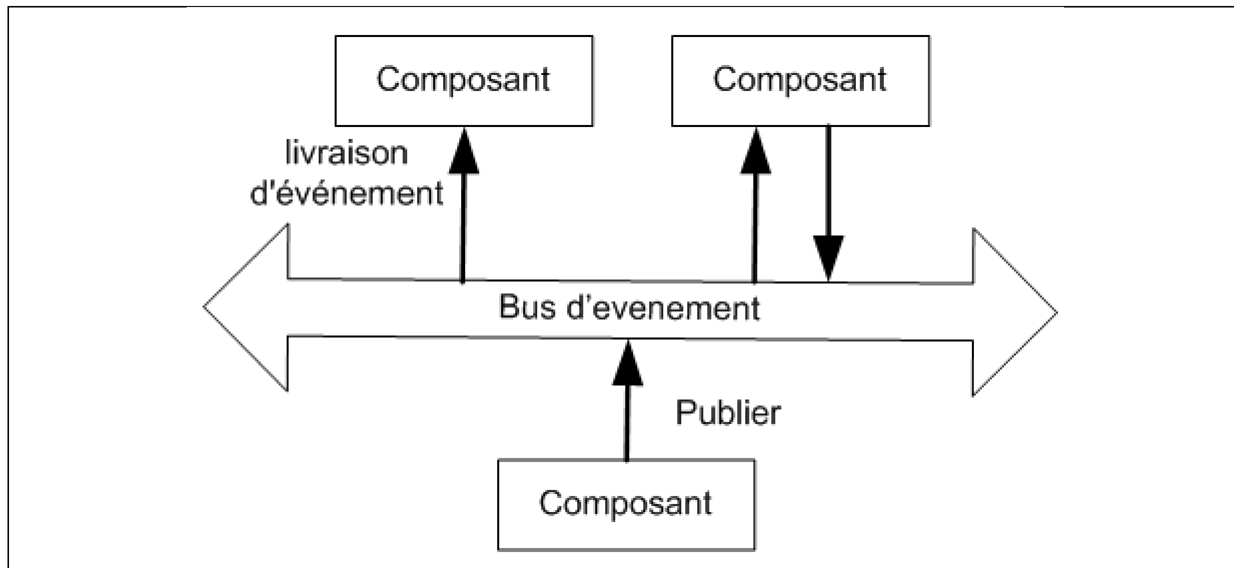


Figure 2-5 Le style basé sur l'événement

2.5 Processus

Le concept d'un processus provient du domaine des systèmes d'exploitation où il est généralement défini comme un programme en cours d'exécution. Du point de vue du système

d'exploitation, la gestion d'ordonnancement des processus est peut-être le problème le plus important à traiter. Cependant, quand il vient à des systèmes distribués, d'autres questions deviennent également ou plus important. Par exemple, pour organiser efficacement les systèmes client-serveur, il est souvent pratique faire usage de techniques de multithreading. Une contribution principale des threads dans les systèmes distribués est qu'ils permettent aux clients et les serveurs à se construire de telle sorte que la communication et le traitement local peuvent chevaucher résultant en un niveau de performance élevé.

2.5.1 Threads

Bien que les processus forment un bloc de construction dans les systèmes distribués, la pratique indique que la granularité des processus telle qu'elle est fournie par les systèmes d'exploitation sur lesquels les systèmes distribués sont construits n'est pas suffisante. Au lieu de cela, il s'avère qu'ayant une granularité plus fine sous la forme de plusieurs threads de contrôle par processus aide à construire des applications distribuées et à atteindre de meilleures performances.

2.5.1.1 Introduction to Threads

Pour comprendre le rôle des threads dans les systèmes distribués, il est important de comprendre ce qu'est un processus et comment se rapportent les processus et les threads. Pour exécuter un programme, un système d'exploitation crée un certain nombre de processeurs virtuels, chacun pour exécuter un programme différent. Pour garder une trace de ces processeurs virtuels, le système d'exploitation a une table de processus, contenant des entrées pour stocker les valeurs de registre CPU, des cartes mémoire, des fichiers ouverts, des informations de comptabilité, privilèges, etc. Un processus est souvent défini comme un programme en cours d'exécution, c'est-à-dire un programme en cours d'exécution sur l'un des processeurs virtuels du système d'exploitation. Comme un processus, un thread exécute son propre morceau de code, indépendamment des autres threads. Cependant, contrairement aux processus, aucune tentative n'est faite pour atteindre un degré élevé de transparence de concurrence si cela entraîne une dégradation des performances. Par conséquent, un système de threads ne conserve généralement que les informations minimales pour permettre à un processeur d'être partagé par plusieurs threads. En particulier, un contexte de thread se compose souvent de rien de plus que le contexte de la CPU, ainsi que d'autres informations pour la gestion des threads. Pour cette raison, la protection des données contre un accès

inapproprié par les threads au sein d'un seul processus est entièrement laissée aux développeurs d'applications.

2.5.1.2 Threads dans les systèmes distribués

Une propriété importante des fils est qu'ils peuvent fournir un moyen pratique de permettre le blocage des appels système sans bloquer tout le processus dans lequel le thread est en cours d'exécution. Cette propriété rend les threads particulièrement attrayants dans les systèmes distribués, car il est beaucoup plus facile d'exprimer la communication sous la forme de maintenir plusieurs connexions logiques en même temps.

- *Clients multithread* : Pour établir un degré élevé de transparence de la distribution, les systèmes distribués qui fonctionnent sur des réseaux étendus peuvent avoir besoin de masquer le temps long de la propagation des messages interprocessus. La manière habituelle de masquer les latences de communication est d'initier la communication et de continuer immédiatement avec autre chose. Pour comprendre les avantages des threads lors de l'écriture du code du serveur, considérez l'organisation d'un serveur de fichiers qui doit parfois bloquer l'attente du disque. Le serveur de fichiers attend normalement une requête entrante pour une opération de fichier, exécute ensuite la requête, puis renvoie la réponse. Une organisation possible et particulièrement populaire est illustrée à la Figure. 2.6

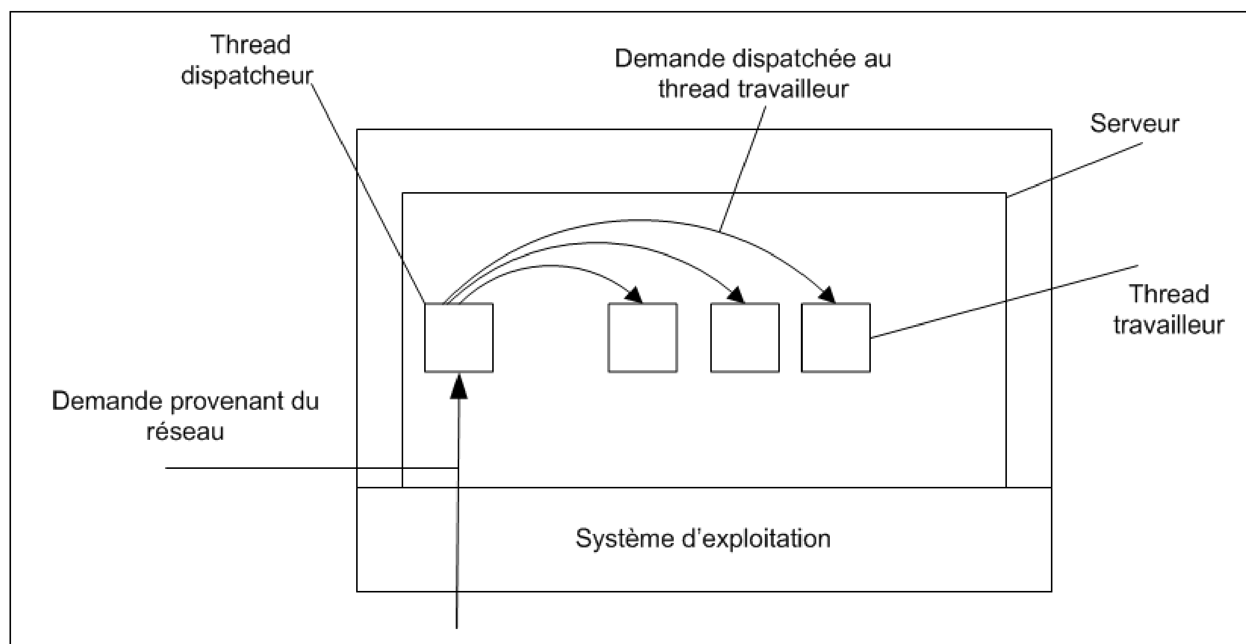


Figure 2-6 Un serveur multithread organisé dans un modèle dispatcheur / Travailleur.

Le thread travailleur procède en effectuant une lecture bloquante sur le système de fichiers local, ce qui peut entraîner la suspension du thread jusqu'à ce que les données soient extraites du disque. Si le thread est suspendu, un autre thread est sélectionné pour être exécuté..

2.5.2 La communication dans les systèmes distribués

La Communication en systèmes distribués est toujours basé sur le passage de messages de bas niveau tel qu'il est proposé par le réseau sous-jacent voir Figure 3.7. L'expression de la communication par le biais du passage de message est plus difficile que d'utiliser des primitives basées sur la mémoire partagée, comme dans les plates-formes non distribuée.

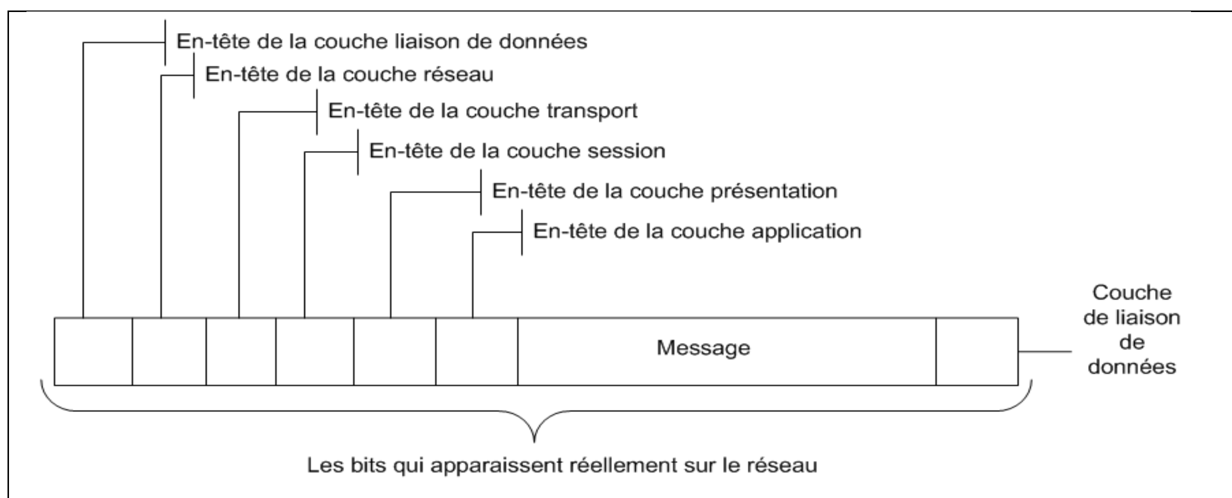


Figure 2-7 Un message typique tel qu'il apparaît sur le réseau.

Les systèmes distribués se composent souvent de milliers ou même des millions de processus dispersés à travers un réseau avec une communication non fiable comme Internet. Nous présentant ensuite trois modèles de communication largement utilisés: appel de procédure distante (RPC, Remote Call Procedure), Le middleware orienté message (MMO, Message-Oriented Middleware), et le streaming de données.

2.5.2.1 Communication à base des RPC dans les environnements de calcul distribués (DCE)

Les appels de procédure à distance ont été largement adoptés comme une base de communication pour les middlewares et les systèmes distribués en général. Le système RPC de DCE comprend un certain nombre de composants, notamment des langages, des bibliothèques, des démons et des programmes utilitaires. Ce, qu'ils permettent d'écrire des

clients et des serveurs. L'ensemble du processus d'écriture et d'utilisation d'un client et d'un serveur RPC est résumé à la figure 2.8.

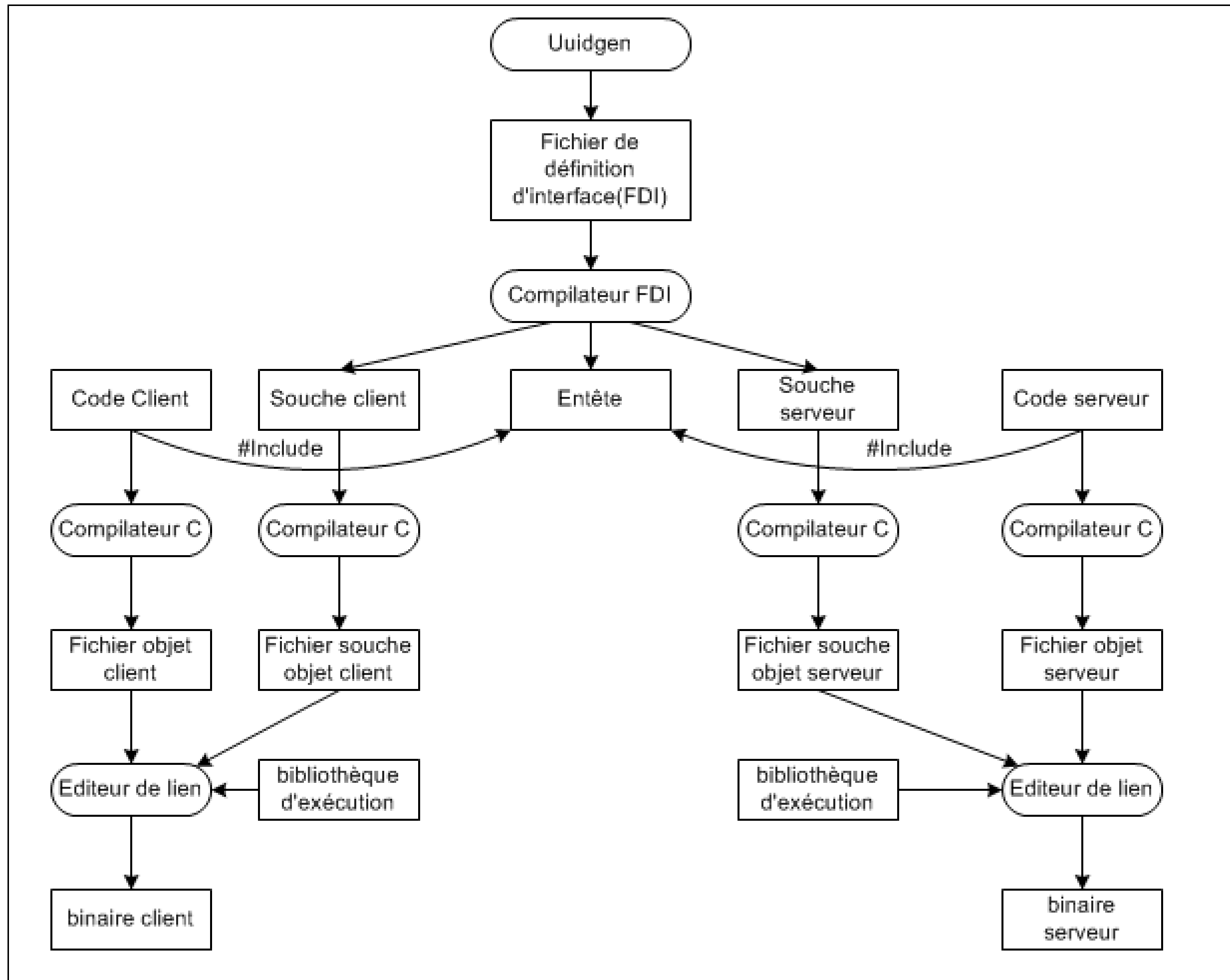


Figure 2-8 Les étapes de l'écriture d'un client et d'un serveur dans DCE RPC.

2.5.2.2 Communication orientée message

Dans de nombreuses applications distribuées, la communication ne suit pas plutôt un modèle strict d'interaction client-serveur. Dans ces cas, il s'avère que penser en termes de messages est plus approprié. Cependant, la communication de bas niveau dans les réseaux informatiques sont inadaptées en raison de la transparence de la distribution. Une alternative consiste à utiliser une file d'attente de messages de haut niveau, dans lequel la communication se déroule sensiblement dans les mêmes systèmes. L'idée de base derrière un système de mise

en file d'attente de messages est que les applications communiquent en insérant des messages dans des files d'attente spécifiques. Ces messages sont transférés sur une série de serveurs de communication et sont finalement livrés à la destination, même si elle était en panne lorsque le message a été envoyé voir figure 2.9.

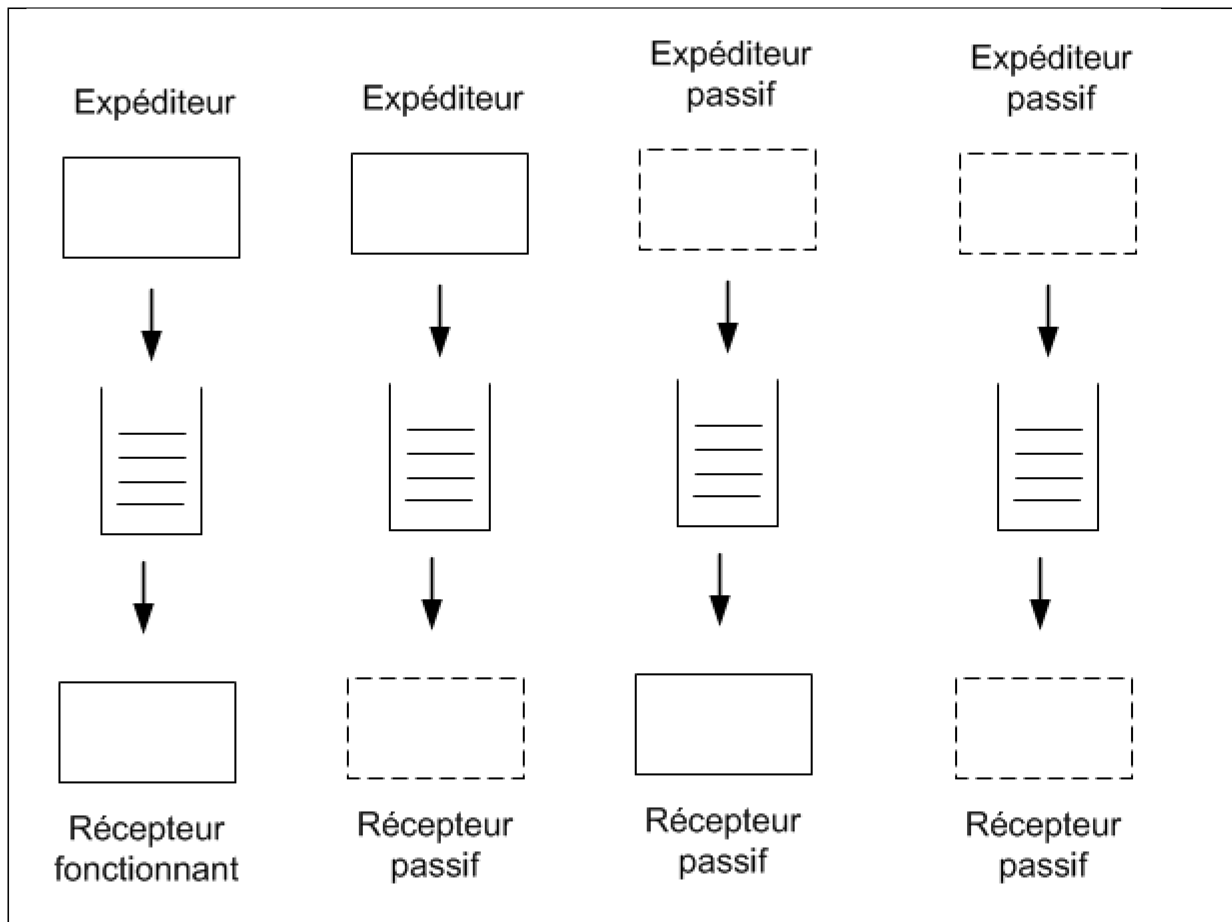


Figure 2-9 Quatre combinaisons pour des communications faiblement couplées utilisant

2.5.2.3 Communication orientée en flux

Avec l'avènement des systèmes distribués multimédias, il est devenu évident que de nombreux systèmes manquaient de support pour la communication de médias continus, comme l'audio et la vidéo. Ce qu'il faut, c'est la notion de flux qui peut soutenir le flux continu de messages, soumis à diverses contraintes de synchronisation. Les flux peuvent être simples ou complexes. Un flux simple consiste en une seule séquence de données, alors qu'un

flux complexe est constitué de plusieurs flux, appelés sous-flux. Du point de vue des systèmes distribués, nous pouvons distinguer plusieurs éléments nécessaires pour supporter les flux. Pour simplifier, nous nous concentrons sur le streaming de données stockées, par opposition à la diffusion de données en direct. Dans le dernier cas. Les données sont capturées en temps réel et envoyées sur le réseau aux destinataires. La principale différence entre les deux est que le streaming de données en direct laisse moins de possibilités de réglage d'un flux. Une esquisse d'une architecture client-serveur générale pour prendre en charge des flux multimédias continus [89], peut être illustrée comme dans la Figure 2.10.

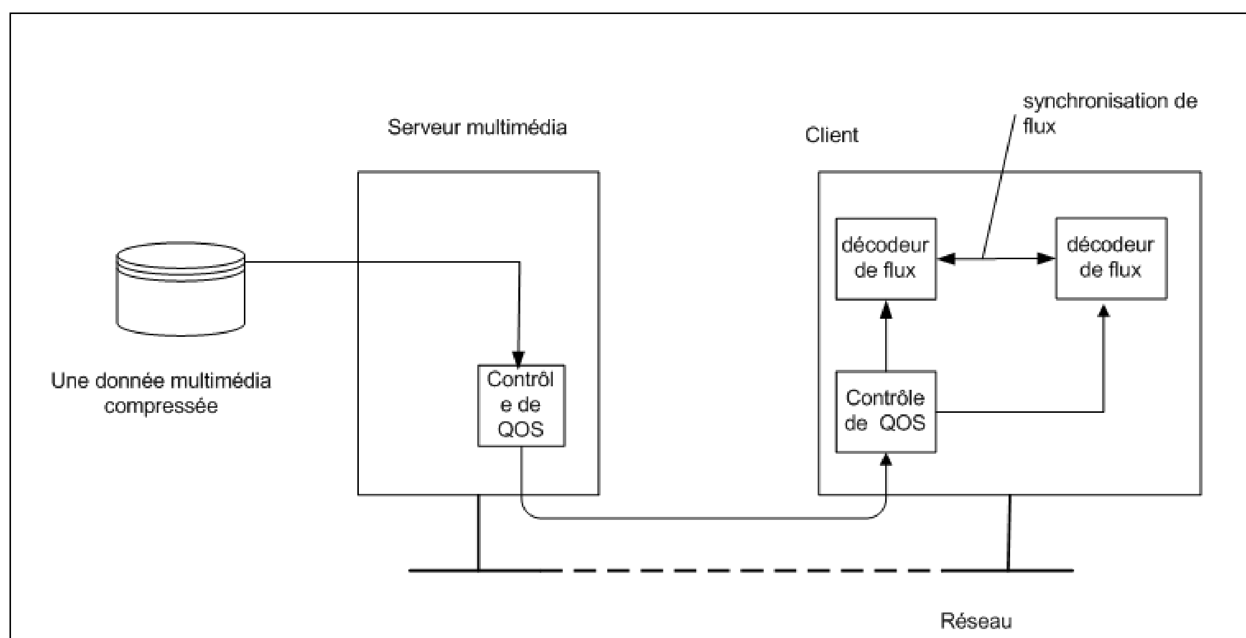


Figure 2-10 Une architecture générale pour le streaming de données multimédia stockées sur un réseau.

2.6 Spécification de système distribué utilisés

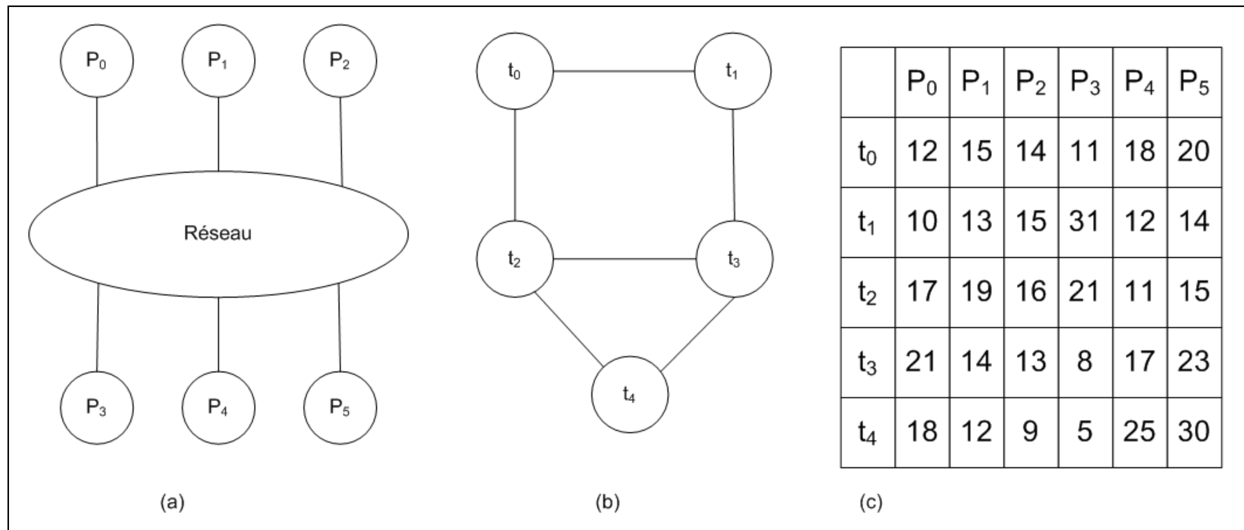
2.6.1 Modèle d'architecture :

Le système informatique réparti est supposé consister en un ensemble P = $\{p_0, p_1, ..., p_{k-1}\}$ de processeurs hétérogènes interconnectés par un réseau de communication arbitraire, comme le montre la figure 2.11 (a). Les chemins de communication entre différents processeurs peuvent avoir des débits de transfert de données différents, modélisés comme une matrice $K \times K$, W, où W_{lk} représente le taux de transfert de données entre les processeurs p_l et p_k. Les entrées

diagonales de la matrice *W* sont réglées à l'infini, car le débit de transfert de données du bus intra-processeur est beaucoup plus rapide que celui du réseau inter-processeur. Les processeurs peuvent avoir des tailles de mémoire, des capacités de traitement et des taux de défaillance différents. De même, les chemins de communication entre différents processeurs peuvent avoir des taux de défaillance différents. Les défaillances des composants (processeurs et voies de communication) dans le système sont supposées être statistiquement indépendantes. En outre, le système cible est supposé contenir un matériel de communication dédié, de sorte que le calcul peut se chevaucher avec la communication.

2.6.2 Modèle d'application

Une application distribuée est décrite par un graphe d'interaction de tâches (GIT), sous cette forme G (T, E), où T = $\{t_{0,}\ t_1, ..., t_{N-1}\}$ représente un ensemble de tâches partitionnées à partir d'une application et E est un ensemble d'arêtes représentant les communications inter-tâches, comme illustré à la figure 2.11 (b). Comme les processeurs du système ont des capacités de traitement différentes, les temps d'exécution d'une tâche s'exécutant sur différents processeurs sont différents. Pour modéliser ceci, une matrice EC de $N \times K$ temps attendu est utilisée, où ec_{ik} représente le temps d'exécution attendu de la tâche ti sur le processeur p_k, comme le montre la Figure 2.11 (c). L'exécution d'une tâche consomme une quantité spécifique de ressources de mémoire et de calcul à partir de son processeur affecté. L'exigence de communication entre tâches est représentée par une matrice $N \times N$, D, où d_{ij} est la quantité de données à transférer entre les tâches t_i et t_j. De même, les communications inter-tâches peuvent prendre des temps de communication différents si elles sont transmises par des chemins de communication différents. Il n'y a pas de relation de précédence entre les tâches. De nombreux problèmes du monde réel peuvent être modélisés comme des GIT tels que la solution itérative des systèmes d'équations, les simulations de systèmes d'alimentation et les programmes de simulation VLSI [50].



	P_0	P_1	P_2	P_3	P_4	P_5
t_0	12	15	14	11	18	20
t_1	10	13	15	31	12	14
t_2	17	19	16	21	11	15
t_3	21	14	13	8	17	23
t_4	18	12	9	5	25	30

Figure 2-11 Un exemple de système distribué hétérogène et un graphe d'interaction de tâches. (a) Système distribué hétérogène, (b) un graphe d'interaction de tâche (GIT), et (c) Temps d'exécution attendus (expected execution times).

2.6.3 Modèle de fiabilité

La fiabilité d'un système informatique distribué pour une application donnée est la probabilité que l'application, affectée aux processeurs du système par une tâche *X*, puisse fonctionner avec succès pendant la durée d'exécution [60,61]. La fiabilité du système est le produit de la probabilité que chaque processeur est fonctionnel pendant l'exécution des tâches qui lui sont assignées et de la probabilité que chaque chemin de communication soit fonctionnel durant la période active de communication inter-tâches entre les processeurs terminaux du chemin. Sous une allocation de tâches $X = \{x_{ik}\}$ $(0 \le i \le N-1, 0 \le k \le K-1)$, la fiabilité du processeur p_k pendant un intervalle de temps t suit la distribution de Poisson, où λ_k est le taux de défaillance du processeur.

Puisque le temps écoulé total pour l'exécution des tâches assignées au processeur pk, est $\sum_{i=0}^{N-1} x_{ik} ec_{ik}$

Ainsi, la fiabilité du processeur correspondant peut être calculée par la formule (1)

$$R_k(X) = e^{-\lambda_k \sum_{i=0}^{N-1} x_{ik} ec_{ik}} \tag{1}$$

De même, soit µlk désigne le taux de défaillance du chemin entre les processeurs terminaux pl et pk. Ainsi la fiabilité du chemin est donnée par la formule (2)

$$R_{lk}(X) = e^{-\mu_{lk}\sum_{i=0}^{N-1}\sum_{i\neq j} x_{il}x_{jk}\left(d_{ij}/w_{kl}\right)} \tag{2}$$

La fiabilité du système est donnée par la formule (3)

$$R(X) = \prod_{k=0}^{K-1} R_k(X) \prod_{k=0}^{K-2} \prod_{l>k} R_{kl}(X) \tag{3}$$

Donc, maximise la fiabilité du système équivaut à minimiser le coût suivant,

$$Z(X) = \sum_{k=0}^{K-1}\sum_{i=0}^{N-1} \lambda_k x_{ik} ec_{ik} + \sum_{k=0}^{K-2}\sum_{k>l}\sum_{i=0}^{N-1}\sum_{i\neq j} \mu_{kl} x_{il} x_{kl} \left(d_{ij}/w_{kl}\right) \tag{4}$$

2.6.4 Modèle d'allocation des taches sous les contraintes de capacité processeurs est mémoires

Soit mi la quantité de mémoire requise par la tâche ti, M_k la capacité mémoire disponible sur le processeur p_k, li la demande de charge de traitement de la tâche t_i et L_k la capacité de traitement disponible sur le processeur p_k. Sous les contraintes de ressources système, le problème d'allocation de tâches adressées est formulé par le problème de programmation quadratique 0-1 suivant:

$$\text{Min } Z(X) \tag{5}$$

$$\text{Sujet à } \sum_{k=0}^{K-1} x_{ik} = 1 \qquad \forall i = 0,1,\ldots,N-1, \tag{6}$$

$$\sum_{i=0}^{N-1} m_i x_{ik} \leq M_k \qquad \forall k = 0,1,\ldots,K-1, \tag{7}$$

$$\sum_{i=0}^{N-1} l_i x_{ik} \leq L_k \qquad \forall k = 0,1,\ldots,K-1, \tag{8}$$

$$x_{ik} \in \{1,0\} \qquad \forall i,k, \tag{9}$$

La contrainte (6) stipule que chaque module doit être affecté à un seul processeur. Les contraintes (7) et (8) garantissent que la capacité de mémoire et de ressources de calcul de chaque processeur n'est pas inférieure à la quantité totale de ressources requises de tous ses modules affectés. La contrainte (9) garantit que x_{ik} sont des variables binaires. La formulation ci-dessus est un problème de programmation 0-1 avec une fonction objective quadratique et il est connu qu'elle est NP-difficile [83]. Par conséquent, la résolution de ce problème est prohibitive en raison d'énormes calculs. Bien que des algorithmes exacts proposés dans [60, 81, 82], la complexité du cas le plus défavorable reste exponentielle et se limite à de petits problèmes d'allocation de tâches. Une alternative consiste à trouver des solutions approchées en utilisant efficacement les méta-heuristiques.

2.7 Un état de l’art

2.7.1 Introduction

Optimiser l'allocation des tâches dans un système distribué hétérogène en assurant une haute qualité de service est connu comme un problème NP-difficile. Beaucoup de critère de qualité sont étudiés parmi eux on trouve le critère de la fiabilité. Ce critère a attiré l’attention d’une grande communauté des chercheurs, que se soit seul ou le considéré avec d’autres critères. Nous allons présenter quelques approches de la littérature qui prennent la fiabilité comme un objectif ou une partie des objectifs (multi objectif) à maximiser en utilisant des techniques approximatives (heuristiques où méta-heuristiques).

2.7.2 Approches approximatives (un état de l’art)

Le problème de l'allocation des tâches dans un système distribué hétérogène est un problème NP-difficile. Ainsi, différentes techniques approchées sont proposées dans la littérature, chacune diffère selon l'objectif ou le domaine d'application, comme la réduction des coûts de reconfiguration [35], l'optimisation des besoins en bande passante [36] consommation d'énergie sur les données à grande échelle [37], amélioration de la performance en utilisant la technologie des agents [38], amélioration de la concurrence des tâches multiples [39],efficacité sociale [40], réduction de la consommation d'énergie globale [41], répartition des charges pour l'optimisation des performances, minimisation de puissance et optimisation du rapport coût / performance [42]le traitement des cœurs reconfigurables avec une prise en compte dynamique des coûts de communication [43], minimisant la somme totale des temps d'exécution et de communication [44 - 48] ou en minimisant le délai d’exécution d'une

application [49-53]. Les systèmes distribués sont plus complexes que les systèmes centralisés. Dans un système aussi vaste, les pannes de machine et de réseau sont inévitables et peuvent avoir un effet négatif sur les applications exécutées sur le système, comme les réseaux véhiculaires sans pilote [54], les réseaux maillés sans fil [55,56], les réseaux corporels sans fil [57] et d'autres domaines émergents. Par conséquent, assurer la fiabilité des systèmes distribués est d'une importance cruciale avec l'attribution des tâches, en particulier pour certaines applications critiques, telles que le contrôle aéronautique, le contrôle des processus industriels et les systèmes bancaires. Par conséquent, il est de plus en plus nécessaire de développer des techniques d'attribution de tâches pour obtenir une fiabilité maximale du système, à savoir minimiser la probabilité de défaillance d'un système pendant l'exécution d'une application, en particulier pour certaines applications critiques. Une façon de prendre en compte les défaillances est d'utiliser des algorithmes d'allocation fiables dans lesquels les tâches d'une application sont assignées aux processeurs les mieux adaptés pour minimiser la probabilité de défaillance de cette application. Plusieurs algorithmes ont été proposés dans la littérature pour l'allocation des tâches dans le but de maximiser la fiabilité avec un ou plusieurs objectifs. Par exemple, optimiser le coût et la fiabilité [58, 59], maximiser la fiabilité [25, 28-30, 60-63], en optimisant la fiabilité, le coût et le temps d’exécution [26, 64], ou en optimisant la fiabilité et la consommation d'énergie [65-67].

La redondance du système est la technique traditionnelle pour atteindre la tolérance aux pannes et donc la fiabilité. Plusieurs objectifs peuvent être associés à une technique de redondance fondée sur la fiabilité. Par exemple, assurer la fiabilité et le temps d’ordonnancement [68,75], optimisant la fiabilité avec le ratio de longueur du temps d’ordonnancement et l'accélération [76], ou améliorant à la fois la réduction du temps et la fiabilité [77], optimiser le temps d’ordonnancement sous les contraintes de fiabilité et de température [78]. Le système informatique distribué est redondant s'il possède une redondance logicielle (par exemple réplication de tâches parmi les nœuds de traitement) et / ou redondance matérielle (par exemple, plusieurs processeurs sur un nœud de traitement et plusieurs canaux de communication reliant chaque nœud) [79], cependant, il s'agit d'une approche coûteuse. De plus, dans de nombreuses situations, la configuration du système est fixe et nous n'avons pas la liberté d'introduire la redondance du système. Par conséquent, nous devons nous tourner vers une stratégie d'allocation des tâches pour atteindre une grande fiabilité du système. Cette thèse étudie le problème d'allocation des tâches afin de maximiser

la fiabilité du système d'un système réparti sans coûts matériels et / ou logiciels supplémentaires.

Dans [80], une nouvelle technique basée sur l'algorithme d'optimisation de l'accouplement des abeilles (HBMO) dans le but de maximiser la fiabilité du système. L'approche basée sur HBMO combine la puissance du recuit simulé, des algorithmes génétiques avec une heuristique de recherche locale spécifique. Dans [81], deux algorithmes pour trouver une allocation de tâches optimale et sous-optimale sont présentés. Le premier est un algorithme exact qui est basé sur la technique brancher et lier avec des sous-estimations pour réduire les calculs dans la recherche d'une allocation optimale des tâches. L'algorithme réorganise la liste des modules pour permettre à un sous-ensemble de modules qui ne communiquent pas entre eux d'être attribués en dernier, afin de réduire davantage les calculs d'allocation optimale des tâches pour maximiser la fiabilité. Le deuxième, est une heuristique qui obtient des allocations de tâches sous-optimales dans un temps de calcul raisonnable. Dans [82,83], le problème d'allocation des tâches dans des systèmes distribués hétérogènes dans le but de maximiser la fiabilité du système est abordé. Un modèle d'allocation fiable basé sur une fonction de coût représentant le manque de fiabilité provoqué par l'exécution de tâches sur les processeurs système et le manque de fiabilité causé par le temps de communication inter processeur soumis aux contraintes imposées par l'application et les ressources système. Aussi, une heuristique dérivé de la technique de recuit simulé (SA) pour résoudre rapidement le problème mentionné. Dans [84], un algorithme génétique (GA) simple est utilisé pour optimiser la fiabilité du système distribué par l'allocation des tâches.

2.8 Conclusion

Nous avons présenté dans ce chapitre les systèmes distribués ainsi que leurs caractéristique en citant quelque types des ces systèmes. Aussi nous avons présenté quelque technique de communication dans les systèmes distribués. Ensuite nous avons présenté nos modèles utilisés pour simuler l’application distribuée, la communication entre les parties de l’application, l’architecture du système et son fiabilité. Pius, nous avons présenté le model du problème d’allocation et distribution des taches sous contrainte du demande mémoire et processeur dans le but de maximiser la fiabilité du système distribué. Enfin nous avons présenté un état de l’art en liaison direct avec notre problème. Les chapitres suivants vont présenter les résultats de notre recherche qui sont concrétisés par deux algorithmes à base de métaheuristiques.

Chapitre 3: Algorithme modifié de la recherche bactérienne

3.1 Introduction

Les métaheuristiques ont démontré leurs efficacités pour résoudre des problèmes complexes. Dans ce chapitre nous allons présenter le premier algorithme amélioré [28] qui émule le processus de recherche de la nourriture pour les bactéries pour résoudre le problème d'allocation des taches afin de maximiser la fiabilité du système distribué.

3.2 L'algorithme modifié de la recherche bactérienne (modified bacterial foraging algorithm)

3.2.1 L'algorithme de la recherche bactérienne classique

Le cycle d'optimisation pour l'algorithme de la recherche bactérienne classique [22] peut être divisé en trois parties: la chimiotaxie (le culbutage (tumbling) et natation (swimming), la reproduction, et l'élimination et la dispersion.

3.2.1.1 chimiotaxie

La bactérie peut être modélisée par $\theta^{i}(jk\ ,l)$ ce qui signifie la $i^{ème}$ bactérie au $j^{ième}$ chimiotaxie, $k^{ième}$ reproduction et $l^{ième}$ élimination et dispersion. La direction peut être ajustée en utilisant la formule (10)

$$\theta^{i}(j+1,k,l)=\theta^{i}(j,k,l)+c(i)\frac{\Delta(i)}{\sqrt{\Delta^{T}(i)\Delta(i)}} \quad (10)$$

Où, *c (i)* est la longueur de marche unitaire et Δ (i) est l'angle de direction de la $j^{ième}$ marche. Le comportement du processus de recherche de nourriture bactérienne (comme la natation et le culbutage) peut être considéré comme un processus d'optimisation.

3.2.1.2 l'essaimage (Swarming)

Alors qu'une bactérie se déplace vers la zone optimale pour la nourriture, il est toujours désireux de trouver la meilleure position ainsi que de produire de forts signaux d'attraction

pour d'autres bactéries. Cela leur permettra de se rassembler pour atteindre la zone désirée. Au cours de ce processus, les cellules d'E.coli libèrent un attractif pour les aider à se regrouper. Ensuite, ils se déplacent en mode concentré de population à forte densité. Le mécanisme du signal interne dans un essaim de E. coli peut être exprimé par la formule (11)

$$J_{cc}\left(\theta, P\left(i,j,k\right)\right)=\sum_{i=0}^{S-1} J_{cc}\left(\theta,\theta_i\left(j,k,l\right)\right)=\sum_{i=0}^{S-1}\left[-d_{attract}\exp\left(-w_{attract}\sum_{m=0}^{n-1}\left(\theta_m-\theta_{i,m}\right)^2\right)\right]+$$

$$\sum_{i=0}^{S-1}\left[h_{repellant}\exp\left(-w_{repellant}\sum_{m=0}^{n-1}\left(\theta_m-\theta_{i,m}\right)^2\right)\right] \quad (11)$$

Où, *Jcc (θ, P (i, j, k))* est la valeur de la fonction objective à ajouter à la fonction objective originale, *S* est le nombre total du bactéries, *n* est le nombre de variables présentes dans chaque bactérie à optimiser , et $\theta = [\theta 1, \theta 2, ..., \theta n]^T$ est un point dans un espace de recherche à *n* dimensions. En outre, $d_{attract}$ et $w_{attract}$ sont la profondeur d'un attractant libéré par une cellule, et la largeur du signal attractif, respectivement. Au contraire, $d_{repellant}$ est la hauteur de la magnitude de l'effet répulsif, et $w_{repellant}$ est la largeur du répulsif.

3.2.1.3 Reproduction

Soit N_c la durée de vie d'une bactérie, mesurée par le nombre d'étapes de chimiotaxie qu'ils exécutent au cours de sa vie. Lorsque toutes les étapes de la chimiotaxie *Nc* sont terminées, la reproduction est lancée. La valeur sanitaire de la bactérie *i* peut être représentée par la somme de la fonction de qualité (fitness) de l'étape au cours de sa vie, comme dans la formule (12)

$$J_{health}^{i}=\sum_{i=0}^{N_c-1} J\left(i,j,k,l\right) \quad (12)$$

Les valeurs de la fonction de qualité (fitness) des bactéries sont triées par ordre croissant en fonction de leur état de santé lorsque l'objectif recherché devrait être minimal. Seules les meilleures bactéries survivent et la moitié restante sera supprimée. Ensuite, le nouveau groupe de bactéries qui est numériquement $S_r = S / 2$ est autorisé à se reproduire par fission.

3.2.1.4 Élimination et dispersion

Les bactéries seront grandement influencées par l'environnement. Tout comme la population bactérienne qui va changer en réponse à l'appauvrissement des aliments dû à leur propre consommation, l'opération de dispersion se produira après un certain nombre de processus de reproduction, une probabilité donnée P_{ed} est utilisée pour décrire un choix de bactérie (dispersion ou non). Si certaines bactéries satisfont à la règle de dispersion, elles seront supprimées ou remplacées. Puisque la bactérie essaie d'atteindre le niveau de nutriments sans satisfaction, elle recherchera toujours une plus grande concentration de nutriments. Ainsi, les trois comportements précédents, la chimiotaxie, la reproduction, et le processus d'élimination et de dispersion, vont continuer jusqu'à ce que les critères de finition soient atteints.

3.2.2 Algorithme proposé

Afin de bénéficier de l'avantage de l'algorithme bactérien, un algorithme amélioré de recherche bactérien (modified bacterial foraging algorithm(MBFA)) est proposé en ajoutant une technique de recuit simulé (voir algorithme 4.2). Le détail de MBFA est décrit ci-dessous:

3.2.2.1 Représentation de la bactérie (solution)

Chaque bactérie devrait représenter une solution pour le problème d'allocation des tâches, donc chaque bactérie B_i est représentée avec un vecteur de *N* composantes (i.e. nombre de tâches), où chaque composante contienne le processeur affecté à la tâche t_j, $j = 0, .. ., N-1$.

	B_i^0	B_i^1	B_i^2	B_i^3	B_i^4	B_i^5
B_i :	0	2	0	1	1	2

Figure 4-1 Représentation de la $i^{ème}$ bactérie

Chaque processeur est représenté par une valeur entière comprise entre 0 et *K-1*. La figure 4.1 montre un exemple illustratif de la bactérie B_i qui décrit une allocation de tâche qui affecte six tâches à trois processeurs. Ainsi, $B_{i1} = 2$ signifie que la tâche t_1 est assignée au processeur avec l'indice deux qui est le troisième dans la ième bactérie. On peut basculer entre la

représentation B_i et la représentation matricielle binaire *X*, où chaque élément de *X* est noté avec x_{ij} en utilisant le formule (13)

$$B_i^j = k \Leftrightarrow x_{jk} = 1 \wedge x_{jl} = 0, \forall l \neq k \middle| l, k \in [0, K-1], j \in [0, N-1] \quad (13)$$

Pour gérer efficacement le problème d'allocation des tâches, deux fonctions de mappage sont définies, M et Q. M est défini de l'intervalle entier [0, K-1] à l'intervalle réel [0, 1] qui affecte à chaque processeur *h* une position θ.

$$M : [0, K-1] \rightarrow [0,1]$$
$$h \mapsto \theta$$

$$M(h) = a + \beta(b-a), \begin{cases} a = \dfrac{h}{K}, b = \dfrac{h+1}{K} - \varepsilon, 0 \leq h < K-2 \\ a = \dfrac{h}{K}, b = \dfrac{h+1}{K}, K-2 \leq h \leq K-1 \end{cases} \quad (14)$$

Où, β est un nombre aléatoire dans [0, 1], ε est une petite valeur utilisée pour interdire d'atteindre la limite supérieure ($\varepsilon = 10^{-K}$).

Aussi, Q est destiné à assigner à chaque position θ un indice de processeur *h*, et il est défini comme suit:

$$Q : [0,1] \rightarrow [0, K-1]$$
$$\theta \mapsto h$$

$$Q(\theta) = \begin{cases} \lceil (K-1)\theta \rceil & , 0 \leq \theta \leq 1 \\ \lceil (K-1)\beta \rceil & , \text{Autrement} \end{cases} \quad (15)$$

3.2.2.2 Modèle de mouvement

Dans l'algorithme original de recherche de bactéries, les bactéries $S_r = S / 2$ les plus faibles meurent, et parmi les bactéries les plus saines, chaque bactérie se divise en deux bactéries placées au même endroit. Bien que cela assure que la population de bactéries reste constante, elle entraîne une perte de diversité de solution. Dans la colonie bactérienne proprement dite, certaines bactéries faibles savent que si elles se dirigent vers une bactérie plus forte, elles trouveront une meilleure surface nutritive, pour modéliser cela, la formule (16) est proposée.

$$\begin{cases} \delta = weak / S \\ \tau_k^{weak} = \dfrac{\left(\theta_k^{strong} - \theta_k^{weak}\right)}{\sqrt{\sum_{k=0}^{K-1}\left(\theta_k^{strong} - \theta_k^{weak}\right)^2}} \\ \theta_k^{weak} = \theta_k^{weak} + \delta \times \tau_k^{weak} \end{cases} \quad (16)$$

Où, θ_k^{weak} est la kième position de la bactérie la plus faible avec l'indice "*weak*", $k = 0, ..., K-1$, $weak = S / 2, ..., S$. La position la plus forte θ_k^{strong} est utilisée pour améliorer la plus faible. En outre, le taux du rang, δ, pour chaque bactérie faible est calculé. Enfin, chaque bactérie faible est améliorée par son échelle de taux τ_k^{weak}.

3.2.2.3 Fonction d'évaluation (fitness function)

La bactérie est en compétition pour la meilleure solution au cours de l'évolution selon une mesure de qualité de la solution, appelée fitness, de sorte que l'évolution de l'essaim est dirigée vers la solution optimale par la meilleure bactérie. La fonction objective originale définie en (4) ne peut donner aucune information sur la validité de la solution sous les contraintes (7, 8), puisque les contraintes (6, 9) sont toujours satisfaites si le schéma de représentation de la bactérie est adopté. Par conséquent, une fonction de pénalité est adoptée pour estimer le niveau d'infaisabilité d'une bactérie sous contraintes (6, 9), et elle est donnée pour une bactérie B_i en utilisant la formule (17)

$$\Phi_{\alpha,\beta}(B_i)=\sum_{k=0}^{K-1}\left(\alpha\max\left(0,\sum_{\{j|B_i^j=k\}} m_j-M_k\right)+\beta\max\left(0,\sum_{\{j|B_i^j=k\}} l_j-L_k\right)\right) \quad (17)$$

Le terme *max* de gauche calcule la quantité d'insuffisance de ressources mémoire sur le processeur *k*, si l'exigence de mémoire encourue par toutes les tâches en cours d'exécution allouées au processeur *k* dépasse la capacité mémoire du processeur. Sinon, il renvoie zéro. De même, le terme *max* de droite calcule la quantité d'insuffisance de ressources de calcul sur le processeur *k*, si l'exigence de traitement de charge encourue par toutes les tâches en cours d'exécution allouées au processeur *k* dépasse sa capacité de traitement. Sinon, il renvoie zéro. α et β sont des facteurs de pénalité $\Phi_{\alpha,\beta}(B^i)$ dont sa valeur est prise à l'échelle de *Z (X)*, (*X* est une forme matricielle binaire de la bactérie B_i générée à l'aide de (13)), de sorte que la tendance d'évolution sera influencée par la pénalité encourue et se diriger vers des solutions valables et s'éloigner des invalides. Les deux facteurs de pénalité α et β sont fixés à 0,5. La fonction objective globale est donnée en sommant la fonction objective (4) avec la fonction de pénalité (17) et la fonction calculant l'attractif de signalisation de cellule à cellule, et elle est définie dans la formule (18).

$$F\left(B_i\right)=Z\left(X\right)+\Phi\left(B_i\right)_{\alpha,\beta}+J_{cc}\left(\theta,P\left(i,j,k\right)\right) \quad (18)$$

Par conséquent, faible est (4) et proche de zéro est (17) le mieux est (18), alors meilleure est la qualité de la bactérie B_i.

3.2.2.4 Algorithme de recuit simulé (simulated annealing SA)

Pour renforcer la recherche de meilleure bactérie, l'algorithme de SA est appliqué dans la dernière étape de l'algorithme 5.2 de recherche de nourriture bactérienne à chaque bactérie B_i pour améliorer sa fiabilité. SA calcule l'aptitude de chaque bactérie F (énergie) en fixant une température initiale T puis une stratégie de recherche de voisin est invoquée pour générer une solution voisine B_n à la solution courante B_i et calcule la fonction objective correspondante F_i en changeant une valeur de processeur par une autre. Si le fitness F_n de la solution voisine B_n

est inférieur à celui de B_i, la solution voisine est acceptée comme solution courante. Sinon, une fonction de probabilité exp ($-\Delta$ / T) est évaluée pour déterminer si la solution voisine peut être acceptée comme une solution courante, où $\Delta = F_n - F_i$. Après l'atteinte d'un équilibre thermique à la température T actuelle, la valeur de T est diminuée d'un facteur de refroidissement α. L'algorithme continue à partir du point de solution actuel à la recherche d'un équilibre thermique au nouveau niveau de température. L'ensemble du processus se termine lorsque le nombre d'itérations est atteint. SA est décrite dans la Algorithme 4.1

Algorithme 4.1 Pseudo code de **SA**

Entrée: bactérie B_i ;
Sortie: le meilleur voisin de Bi
Calculer la fonction objective F_i en utilisant (18) ;
Sélectionnez une température initiale T ; //T=10000;
Sélectionnez un facteur de refroidissement $\alpha < 1$; // (α =0.003);
Répéter
Sélectionnez une solution voisine B_n à B_i;
Calculer sa fonction objective F_n en utilisant (18) ;
$\Delta = F_n - F_i$;
Si $\Delta < 0$ **Alors**
$B_i = B_n$; $F_i = F_n$;
Sinon
Générer une valeur aléatoire r dans (0, 1);
Si $r < exp(-\Delta/T)$ **Alors**
$B_i = B_n$; $F_i = F_n$;
Finsi
Finsi
$T = \alpha \times T$;
Jusqu'à T>1

3.2.3 Evaluations expérimentales

Pour évaluer les performances de MBFA décrit dans l’algorithme 4.2, des instances de problèmes générées de manière aléatoire sont utilisées, car un ensemble de référentiels standard généralement accepté n'existe pas. Pour tester le MBFA, un large éventail de scénarios, similaires à ceux utilisés par d'autres chercheurs, sont utilisés [60,61]. Un ensemble de données de simulation est créé en faisant varier un ensemble de paramètres qui détermine

les caractéristiques des instances de problème générées. Les paramètres sont décrits dans Tableau 3.1

Tableau 3-1Désignations des paramètres utilisées

Paramètre	**Désignation**
N	Nombre de tâches
K	Nombre de processeurs
DS	Densité d'interaction de la tache
CCR	Rapport de communication au temps calcul
ec_{ik}	Temps d'exécution attendu de la tâche t_i sur le processeur p_k
w_{lk}	Taux de transfert de données entre les processeurs p_l et p_k
d_{ij}	Quantité de données transférées entre les tâches t_i et t_j
m_i	Mémoire requise par la tâche t_i
M_k	Capacité de mémoire disponible sur le processeur p_k
l_i	Charge de traitement demandé
L_k	Capacité de traitement disponible sur le processeur p_k
λ_k	Taux d'échec du processeur p_k
μ_{lk}	Taux d'échec du lien entre p_k et p_l

Les valeurs des paramètres sont générées au hasard à partir des distributions uniformes comme dans [61,80] et décrits dans le tableau 3.2.

Tableau 3-2 Valeurs des paramètres utilisées

Paramètre	**Valeurs possibles**
N	(20,40,60,80)
K	(4,6,8,10)
DS	(0.3,0.5,0.8)
d_{ij}	Généré aléatoirement de sorte que le CCR et 0.5, 1.0 ou 2.0
ec_{ik}	Uniforme (15,25)
w_{lk}	Uniforme (1,10)
λ_k	Uniforme (0.00005, 0.00010)
μ_{lk}	Uniforme (0.00015, 0.00030)
m_i	Uniforme (15,25)
M_k	Varie de R/K to 2R/K, ou $R = \sum_{i=0}^{N-1} m_i$

l_i	Uniforme (15,25)
L_k	Varie de R/K to 2R/K, ou $R = \sum_{i=0}^{N-1} l_i$

Les paramètres de MBFA, HPSO et GAA sont pris comme dans le tableau 3.3

Tableau 3-3 Paramètres et valeurs utilisées pour MBFA, HPSO et GAA

Paramètre	**MBFA**	**HPSO**	**GAA**
Population	50	50	50
Nombre d'itération		80000	6000
Coefficients cognitifs (ξ1, ξ2)		(2.05, 2.05)	
Mutation et croisement (m_{pr} , c_{pr})			(0.2, 0.8)
Longueur de début du chimiotaxie (C)	0.2		
Etapes de chimiotaxie (N_c)	7		
Etapes de nage (N_s)	5		
Etapes de reproduction (N_{re})	4		
Etapes d'élimination et de dispersion (N_{ed})	4		
Probabilité d'élimination et de dispersion (P_{ed})	0.25		
Profondeur et largeur d'un attractant ($d_{attract}$, $w_{attract}$)	(0.1, 0.2)		
Hauteur et largeur de répulsif ($d_{repellant}$, $w_{repellant}$)	(0.1, 10)		

Tous les algorithmes sont codés en JAVA 1.8 et exécutés sur un processeur AMD 1,00 GHz avec une mémoire principale de 4 Go fonctionnant sous environnement Windows 8.1. Dans toutes les expériences, chaque algorithme se termine lorsque le nombre maximal d'itérations est atteint.

Algorithme 4.2 Pseudo code de l'algorithme **MBFA**
/*Étape d'initialisation*/ Nombre de bactéries à utiliser dans la recherche (S). Nombre d'étapes chimiotactiques (N_c), longueur de nage maximale (N_s), Nombre d'étapes de reproduction (Nre), Nombre d'événements de dispersion d'élimination (N_{ed}), Probabilité d'élimination et de dispersion (P_{ed}),

Pas de chimiotaxie de longueur constante (C),

/* Étapes de traitement*/

(1) Générer un essaim initial de bactéries N au hasard, où chaque bactérie devrait respecter la représentation comme dans la figure 4.1.
(2) Début de la boucle d'élimination-dispersion: $l = l + 1$
(3) Début de la boucle de reproduction: $k = k + 1$
(4) Début de la boucle de chimiotaxie: $j = j + 1$
(a) **Pour Chaque** $i = 0,1,...,S-1$ **Faire**
Prendre une étape de chimiotaxie pour une bactérie B_i comme suit :
(a.1) Calculer la fonction objective de B_i , $J(i,j,k,l)$ en utilisant (18)
(a.2) $J_{last} = J(i,j,k,l)$ et $B_{last} = B_i$ pour sauvegarder la valeur de la fonction objective et la solution puisque nous pouvons trouver de meilleures solutions via une exécution.
(a.3) Culbuter: Générer un vecteur aléatoire $\Delta(i) \in R^N$ où chaque élément $\Delta_m(i)$, $m = 0,...,N-1$ est un nombre aléatoire dans [-1,1]
(a.4) Déplacer: $c(i) = C$
(a.4.1) **Pour chaque** processeur B_i^p de la bactérie B_i **Faire**
- Calculer sa position en utilisant (14)
- Mettre à jour cette position en utilisant (10)
- Remplacer le processeur actuel par le nouveau généré à partir de la position mise à jour en utilisant (15)

Cela résulte en une étape de taille de $c(i)$ dans le sens de la dégringolade pour la bactérie B_i .
(a.5) Calculer la nouvelle fonction objective $J(i,j+1,k,l)$ de B_i
(a.6) Nager:
(a.6.1) Let $m = 0$ (compteur pour la longueur de nage)
(a.6.2) **Tanque** $m < N_s$ **Faire** (si vous n'avez pas descendu trop longtemps)
(a.6.2.1) **Si** $J(i,j+1,k,l) < J_{last}$ **Alors**
(a.6.2.1.1) $J_{last} = J(i,j+1,k,l)$
(a.6.2.1.2) $B_{last} = B_i$
(a.6.2.1.3) (10) sera repris, et les nouvelles positions générées $\{\theta_p^i(j+1,k,l)\}, 0 \le p \le N-1$ sont utilisées pour calculer $J(i,j+1,k,l)$,en suite mètre à jour J_{last} et B_{last} comme nous l'avons fait ci-dessus.
(a.6.2.1.4) $m = m + 1$.
(a.6.2.2) **Sinon** let $m = N_s$.
Finsi
FinTanque.
(a.7) **Aller à** la prochaine bactérie $i+1$: if $i \neq S$ **Aller à** (a.2) pour traiter la prochaine bactérie
(5) **Si** $j==N_c$ **Alors Aller à** (3). Dans ce cas, continuez la boucle de chimiotaxie, puisque la vie de la bactérie n'est pas terminée.
(6) **Pour** k et l données, et **Pour chaque** $i = 0,1,...,S-1$ **Faire**
Soit J_{health}^i, la santé de la i^{ème} bactérie. Les bactéries sont triées par ordre croissant en utilisant (12)
/* améliorer les faibles bactéries en utilisant la fonction de rapprochement */
(6.1) **Pour chaque** bactérie , B_i , $i = S/2,...,S-1$ **Faire**
Pour chaque processeur B_i^p de B_i,
- Utiliser (16) pour générer la nouvelle position, puis utiliser (15) pour obtenir le nouveau processeur
- Calculer la nouvelle fonction objective B_i

(7) Reproduction :
(7.1) **Pour** k et l Données, et **Pour chaque**, $i = 0,1,...,S-1$
Soit J^{i}_{health}, la santé de la i[ème] bactérie. Les bactéries sont triées par ordre croissant en utilisant (12)
(7.2) Les $S_r = S/2$ bactéries avec les valeurs les plus hautes de J_{health} meurent et les autre, qui ont les valeurs minimales, se divisent et leurs copies sont maintenant placés au même endroit que leur parent.
(8) **Si** $k < N_{re}$ **Alors Aller à** (2). Dans ce cas, le nombre d'étapes de reproduction spécifiées n'est pas atteint, donc nous commençons la prochaine génération en boucle chimiotactique.
(9) Élimination-dispersion:
Pour $m = 0,...,S-1$, **Faire**
Avec une probabilité P_{ed} chaque bactérie est éliminée et dispersée à un endroit aléatoire tout en maintenant le nombre de la population constante. Si une bactérie est éliminée, une nouvelle sera dispersée dans un emplacement aléatoire du domaine d'optimisation.
(10) **Si** $I<N_{ed}$ **Alors**
améliorer chaque bactérie de l'essaim en utilisant **SA** et **Aller à** (2),
Sinon arrêtez.

3.2.3.1 Test des principaux paramètres

Pour tester l'efficience et l'efficacité de MBFA avec celles des HPSO et GAA, plusieurs paramètres sont choisis: nombre de tâches (N), densité (DS), rapport de communication au calcul (CCR), taille de l'essaim (S) et nombre de processeur (PR). Finalement, la performance de tous les algorithmes est rapportée. Afin d'analyser les résultats de fiabilité donnés par MBFA avec ceux de HPSO et GAA, l'information d'amélioration moyenne en pourcentage (average percentage improvement (API)) est utilisée. Par exemple, l'API entre HPSO et MBFA est calculée comme suit :

$$API = \left(1 - \frac{\mathrm{Re}\,liability_{HPSO}}{\mathrm{Re}\,liability_{MBFA}}\right) \times 100\%$$

3.2.3.1.1 Nombre de paramètres de tâches (N)

La figure 4.2 montre que, lorsque N = 20, la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 97%, 93% et 93%. Ainsi, l'API indique que MBFA surpasse HPSO et GAA de 4%. En outre, lorsque N = 40, MBFA surpasse HPSO de 20% et GAA de 16%, et la fiabilité donnée par MBFA, HPSO et GAA est de 94%, 75% et 79%, respectivement. Lorsque N = 60, API indique que MBFA surpasse HPSO de 35% et GAA de 32%, et la fiabilité donnée par MBFA, HPSO et GAA est de 93%, 60% et 62%, respectivement. Enfin, lorsque N

= 80, API montre que MBFA surpasse HPSO de 56% et GAA de 50%, et la fiabilité donnée par MBFA, HPSO et GAA est de 90%, 39% et 44%, respectivement. Par conséquent, la meilleure fiabilité est celle de MBFA. En outre, à partir des résultats ci-dessus, MBFA montre la meilleure évolutivité sur HPSO et GAA. Ceci est dû au mécanisme de recherche efficace et indépendant de la recherche de nourriture bactérienne pour l'exploration et au recuit simulé pour l'exploitation qui permet une convergence rapide vers la meilleure solution.

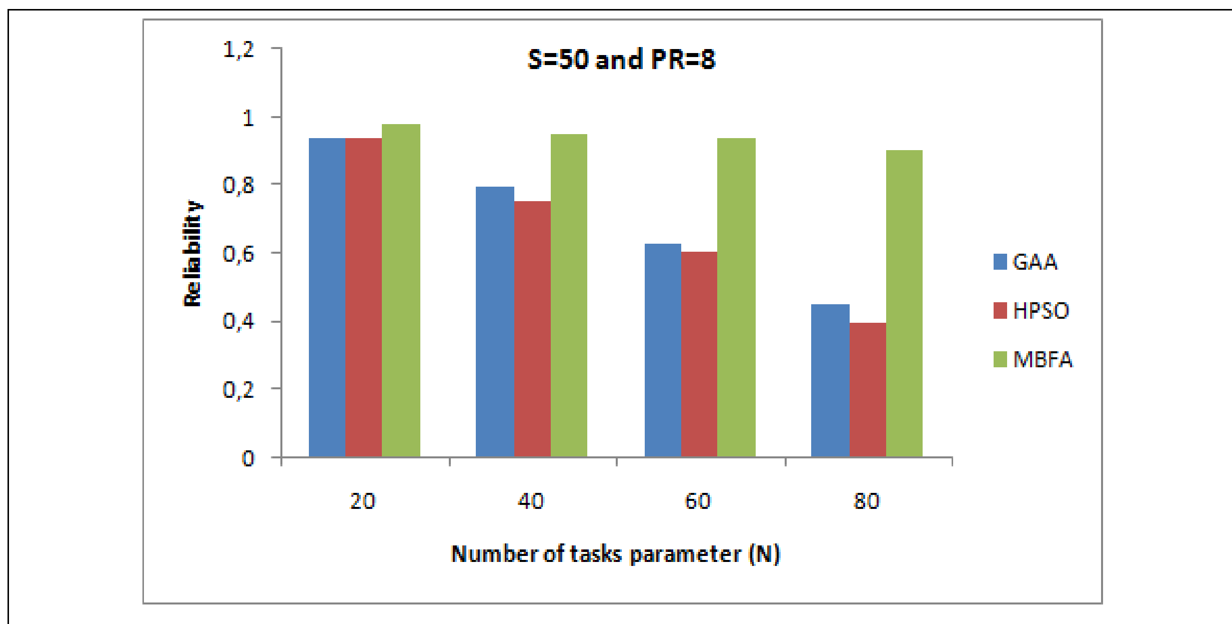


Figure 4-2 Comparaison de fiabilité sous le paramètre nombre de tâches

3.2.3.1.2 Paramètre de densité (DS)

La figure 4.3 met en évidence que, pour les problèmes avec moins de complexité (DS = 0,3), MBFA, HPSO et GAA0, la fiabilité donnée est de 97%, 93% et 93%, respectivement. Ainsi, l'API montre que MBFA surpasse HPSO et GAA de 4%. En outre, pour les problèmes de complexité moyenne (DS = 0,5), MBFA surpasse HPSO et GAA de 6%, et la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 96%, 90% et 90%. Enfin, pour les problèmes de complexité élevée (DS = 0,8), API indique que MBFA surpasse HPSO de 5% et GAA de 6%, et la fiabilité donnée par MBFA, HPSO et GAA respectivement de 97%, 92% et 91%. A partir des résultats ci-dessus, MBFA confirme sa supériorité sur les approches publiées en considérant le paramètre densité. Cela est dû à la modification apportée à MBFA.

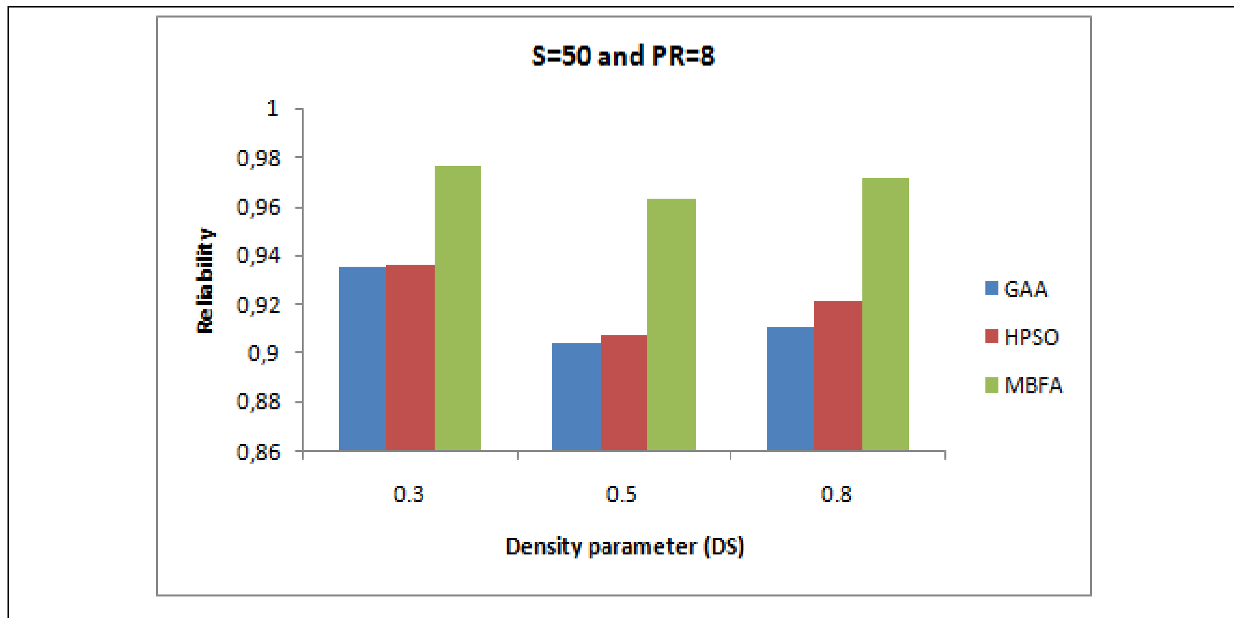


Figure 4-3 Comparaison de fiabilité sous le paramètre de densité

3.2.3.1.3 Rapport de Communication au Calcul (CCR)

La figure 4.4 montre, pour les problèmes orientés calculs (CCR = 0.5), la fiabilité suivante: 97%, 93% et 93% qui correspond respectivement à MBFA, HPSO et GAA. API montre que MBFA surpasse à la fois HPSO et GAA de 4%. De même, pour les problèmes avec la même quantité de calcul et de communication (CCR = 1.0), MBFA surpasse HPSO de 6% et GAA de 8%. La fiabilité donnée par MBFA, HPSO et GAA est respectivement de 97%, 91% et 89%. Enfin, pour les problèmes orientées communications (CCR = 2.0), API indique que MBFA surpasse HPSO de 16% et GAA de 14%, et la fiabilité donnée par MBFA, HPSO et GAA respectivement de 97%, 81% et 83%. Ainsi, MBFA est plus pratique pour tout type d'application. C'est parce que MBDA utilise une technique adaptée, ce qui n'est pas le cas pour HPSO et GAA.

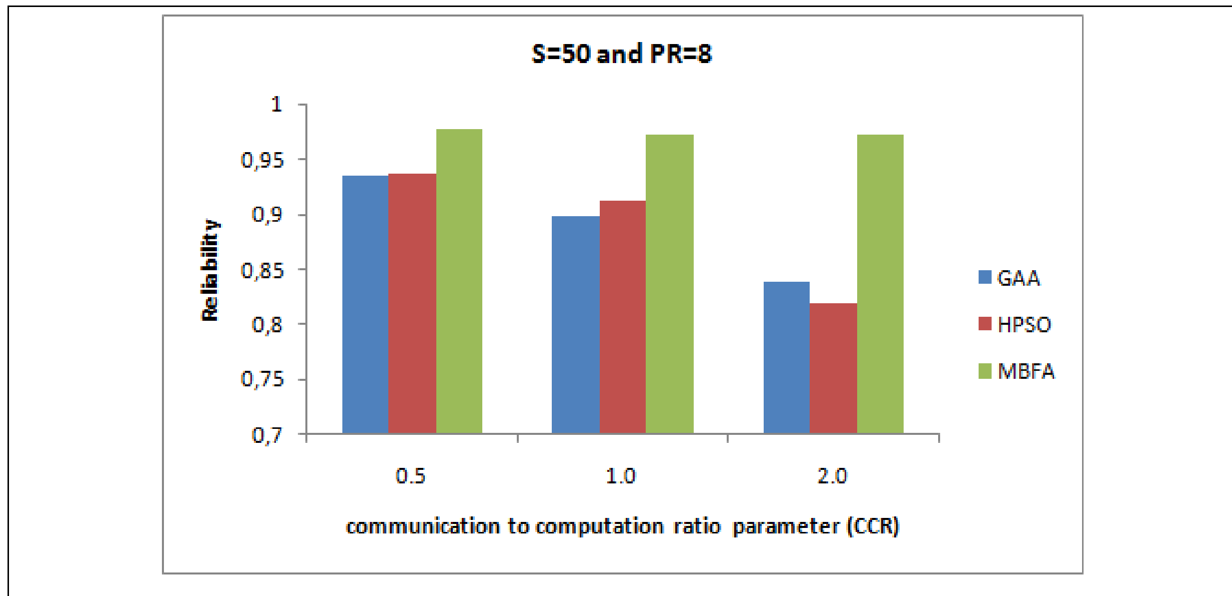


Figure 4-4 Comparaison de fiabilité sous paramètre CCR

3.2.3.1.4 Taille de la population (Swarm (S))

La figure 4.5 montre l'impact de la modification de la taille de l'essaim sur la fiabilité. Ainsi, lorsque S = 20, la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 87%, 26% et 31%. Ainsi, MBFA surpasse HPSO de 70% et GAA de 64% en utilisant les informations API. De même, pour S = 40, la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 85%, 34% et 31%. Ainsi, MBFA surpasse HPSO de 60% et GAA de 63%. Lorsque S = 60, la fiabilité calculée par MBFA, HPSO et GAA est respectivement de 89%, 30% et 27%. Par conséquent, MBFA surpasse HPSO de 66% et GAA de 69%. De même, lorsque S = 80, la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 90%, 30% et 28%. Enfin, lorsque S = 100, la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 86%, 25% et 24%. Ainsi, MBFA surpasse HPSO de 70% et GAA de 72%. Ainsi, la taille de l'essaim a moins d'impact sur MBFA. C'est parce que MBFA convertit vers la meilleure solution tôt.

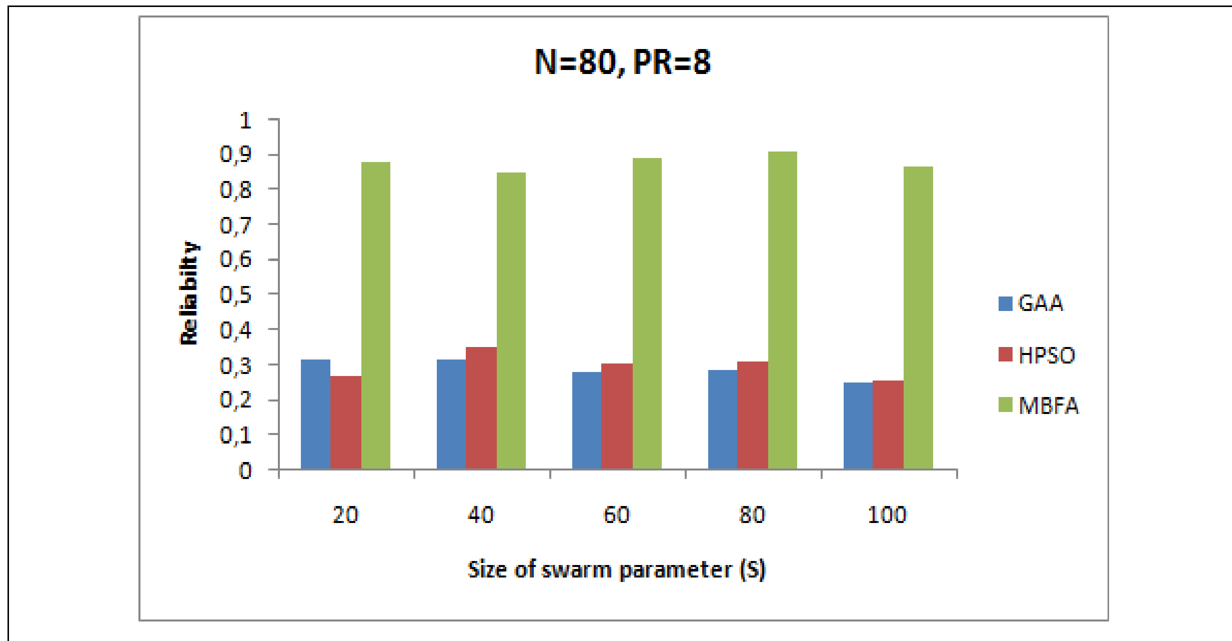


Figure 4-5 Comparaison de fiabilité sous la taille du paramètre de la population

3.2.3.1.5 Nombre de processeur (PR)

La figure 4.6 montre l'impact de l'évolution du nombre de processeurs sur la fiabilité. Pour cela, cinq valeurs pour les processeurs sont utilisées. Lorsque PR = 2, la fiabilité calculée par MBFA, HPSO et GAA est de 91%, 68% et 70%, respectivement. En outre, lorsque PR = 4, la fiabilité donnée par MBFA, HPSO et GAA est de 89%, 49% et 41%, respectivement. Lorsque PR = 6, la fiabilité donnée par MBFA, HPSO et GAA est de 84%, 41% et 34%, respectivement. De même, pour PR = 8, la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 87%, 34% et 26%. Enfin, pour PR = 10, la fiabilité donnée par MBFA, HPSO et GAA est respectivement de 85%, 32% et 33%. Ainsi, MBFA est plus évolutif (scalable) que HPSO et GAA. Ceci est dû à la modification apportée à l'algorithme bactérien de recherche de la nourriture.

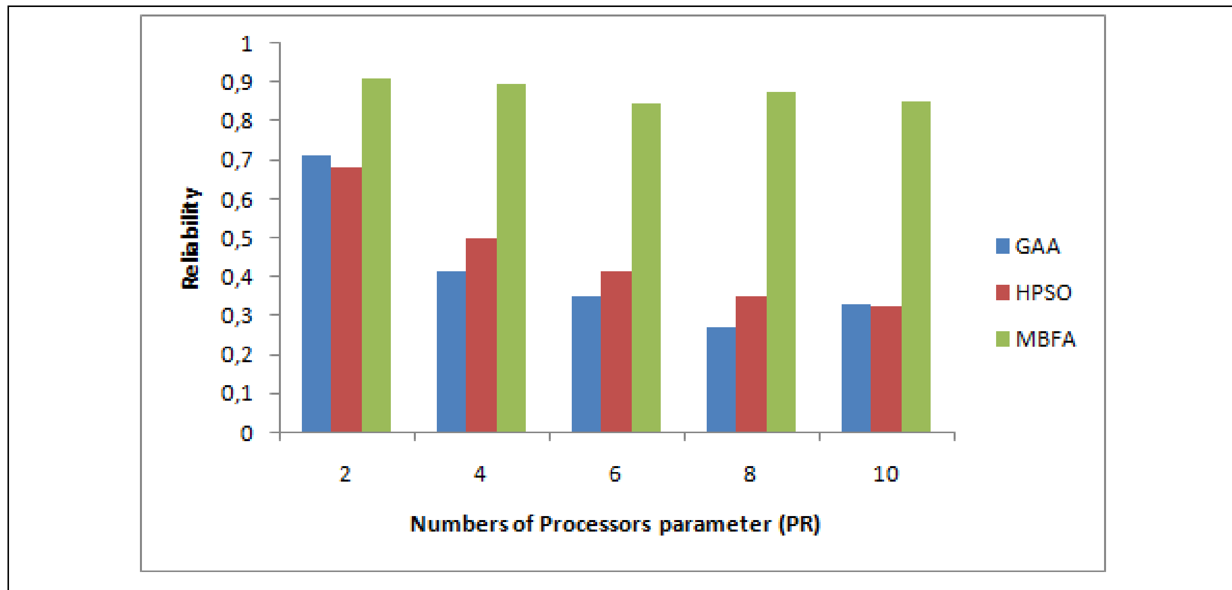


Figure 4-6 Comparaison de fiabilité sous le paramètre du nombre de processeurs

3.2.3.1.6 Étude de performance

La figure 4.7 montre la durée de fonctionnement de MBFA sous 2, 4, 6, 8 et 10 processeurs qui est de 1, 5, 12, 23, 37 seconde (s), respectivement. De même, la figure 4.7 met en évidence le temps de fonctionnement de HPSO qui est respectivement de 0, 7, 26, 73, 163 secondes sous 2, 4, 6, 8 et 10 processeurs. Enfin, le temps de fonctionnement de GAA montré est 2, 11, 28, 49, 80 seconde (s) pour 2, 4, 6, 8 et 10 processeurs, respectivement. En fait, plus de processeurs sont impliqués, moins de temps de fonctionnement est effectué ce qui n'est pas le cas dans cette étude. Cela peut s'expliquer par le flux de communication impliquée en raison des tâches réparties sur différents processeurs. Malgré cela, MBFA affiche la meilleure performance par rapport à HPSO et GAA. Comme il est montré dans les tests précédents, MBFA a besoin de moins d'itération pour converger vers la solution optimale. Cela est dû à son mécanisme de recherche efficace.

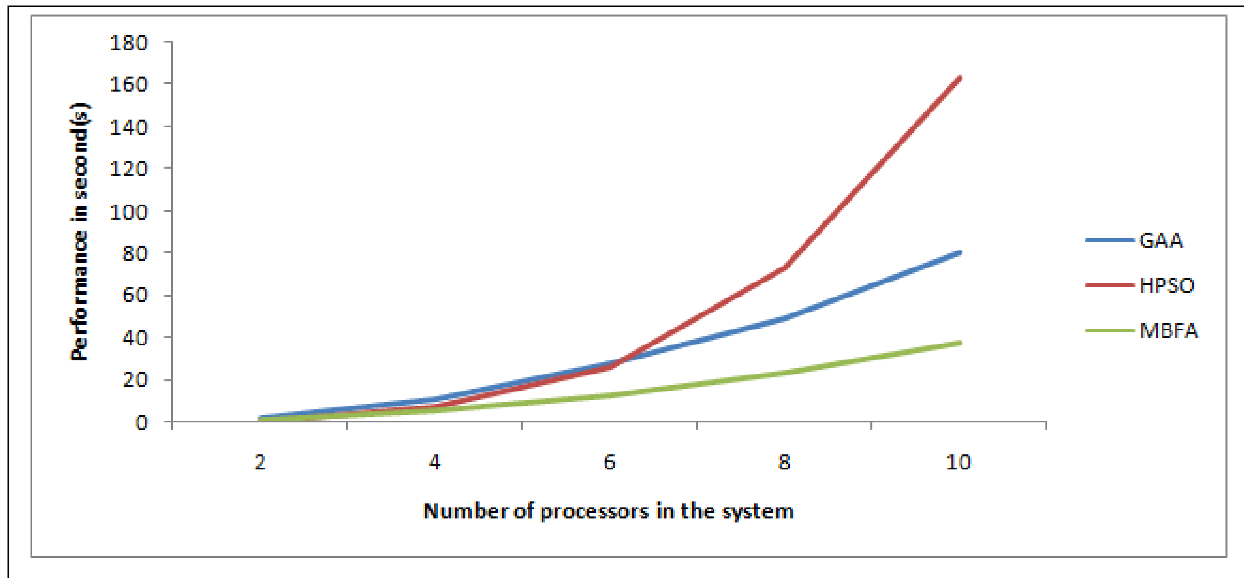


Figure 4-7 Comparaison de la performance entre MBFA, HPSO et GAA

3.2.4 Conclusion de la contribution

Résoudre le problème d'allocation des tâches pour maximiser la fiabilité sans duplication des ressources est l'objectif principal de ce document. MBFA est proposé et testé sous plusieurs paramètres utilisant des problèmes aléatoires. Les résultats de la simulation montrent que MBFA peut atteindre une plus grande fiabilité sous tous les paramètres. La fiabilité donnée par MBFA est supérieure à celle de l'algorithme HPSO et GAA. De plus, les performances de MBFA sont supérieures à celles de HPSO et de GAA. Par conséquent, MBFA est considérablement efficace et faisable pour l'utilisation dans le problème d'allocation de tâches pour maximiser la fiabilité dans des systèmes distribués hétérogènes.

3.3 Conclusion

Dans ce chapitre nous avons présenté l'algorithme MBFA, puis on a lui testé avec des algorithmes similaires qui existent dans la littérature. Dans le chapitre suivant nous allons présenter le deuxième algorithme qui s'appelle l'algorithme modifié de la recherche coucou.

Chapitre 4: Algorithme modifié de recherche discrète du coucou

4.1 Introduction

Dans ce chapitre nous allons présenter la deuxième technique améliorée [29] qui émule le processus de survivance chez les coucous pour résoudre le problème d'allocation des taches afin de maximiser la fiabilité du système distribué.

4.2 L'algorithme de recherche de coucou classique

Nous pouvons utiliser les représentations simples suivantes où chaque œuf représente une solution, et chaque coucou ne peut pondre qu'un seul œuf, le but est d'utiliser les meilleures solutions (coucous) pour remplacer les mauvaises dans les nids. CS utilise une combinaison équilibrée entre une marche aléatoire locale et autre aléatoire globale, contrôlée par un paramètre de commutation p_a. La marche aléatoire locale peut être modélisée par (10) comme dans [79]

$$x_i^{t+1} = x_i^t + \alpha s \otimes H\left(p_a - \varepsilon\right) \otimes \left(x_l^t - x_j^t\right) \tag{10}$$

Où x_l^t et x_j^t sont deux solutions différentes choisies aléatoirement par permutation aléatoire, H (u) est une fonction de Heaviside, ε est un nombre aléatoire tiré d'une distribution uniforme et *s* est la taille du pas. D'autre part, la marche aléatoire globale est effectuée en utilisant les vols de Lévy comme dans (11):

$$x_i^{t+1} = x_i^t + \alpha L\acute{e}vy\left(s, \theta\right) \tag{11}$$

Où, α est le facteur d'échelle de taille de pas supérieur à zéro qui devrait être lié aux échelles du problème d'intérêt. Dans la plupart des cas, on peut utiliser α = O (L / 10), où L est l'échelle caractéristique du problème d'intérêt, alors que dans certains cas α = O (L / 100) peut être plus efficace et évite de voler trop loin.

$$L\acute{e}vy(s,\theta) = \frac{\lambda \Gamma(\theta) \sin(\pi\theta/2)}{\pi} \frac{1}{s^{1+\theta}} \quad (, s >> s_0 > 0) \tag{12}$$

Ici, Γ (θ) est la distribution gamma et θ est le paramètre d'échelle avec $1 < \theta \leq 3$.

4.2.1 Représentation de coucou (solution)

Chaque coucou devrait représenter une solution pour le problème d'allocation des tâches, donc chaque coucou B^i est représentée avec un vecteur de *N* composantes (i.e. nombre de tâches), lorsque chaque composante contient le processeur affecté à la tâche t_j, *j = 0, .. ., N-1.* Chaque processeur est représenté par une valeur entière comprise entre 0 et K-1. La figure 5.1 montre un exemple illustratif de coucou B^i qui décrit une allocation de tâche qui affecte six tâches à trois processeurs. Ainsi, $B_1^i = 2$ signifie que la tâche t_1 est assignée au processeur avec l'indice deux qui est le troisième dans la ième coucou. On peut basculer entre la représentation B^i et la représentation matricielle binaire *X*, où chaque élément de *X* est noté avec x_{ij} en utilisant le formule (13)

$$B_j^i = k \Leftrightarrow x_{\mathrm{jk}} = 1 \wedge x_{jl} = 0, \forall l \neq k \,\middle|\, l,k \in [0, K-1], j \in [0, N-1] \quad (13)$$

5.2.2 Fonction d'évaluation

Chaque coucou est en compétition pour la meilleure solution au cours de l'évolution selon une mesure de qualité de la solution, appelée fitness, de sorte que l'évolution de l'essaim est dirigée vers la solution optimale par la meilleure coucou. La fonction objective originale définie en (4) ne peut donner aucune information sur la validité de la solution sous les contraintes (7, 8), puisque les contraintes (6, 9) sont toujours satisfaites si le schéma de représentation de la coucou est adopté. Par conséquent, une fonction de pénalité est adoptée pour estimer le niveau d'infaisabilité d'un coucou sous contraintes (6, 9), et elle est donnée pour une coucou B^i en utilisant la formule (17)

$$\Phi_{\alpha,\beta}(B^i) = \sum_{k=0}^{K-1} \left(\alpha \max\left(0, \sum_{\{j | B_j^i = k\}} m_j - M_k\right) + \beta \max\left(0, \sum_{\{j | B_j^i = k\}} l_j - L_k\right) \right) \quad (14)$$

Le terme *max* de gauche calcule la quantité d'insuffisance de ressources mémoire sur le processeur *k*, si l'exigence de mémoire encourue par toutes les tâches en cours d'exécution allouées au processeur *k* dépasse la capacité mémoire du processeur. Sinon, il renvoie zéro.

De même, le terme *max* de droite calcule la quantité d'insuffisance de ressources de calcul sur le processeur *k*, si l'exigence de traitement de charge encourue par toutes les tâches en cours d'exécution allouées au processeur k dépasse sa capacité de traitement. Sinon, il renvoie zéro. α et β sont des facteurs de pénalité dont sa valeur est prise à l'échelle de Z (X), (X est une forme matricielle binaire de la coucou B^i générée à l'aide de (13)), de sorte que la tendance d'évolution sera influencée par pénalité encourue et se diriger vers des solutions valables et s'éloigner des invalides. Les deux facteurs de pénalité α et β sont fixés à 0,5. La fonction globale est donnée par la somme de la fonction de pénalité avec la fonction d'objective d'origine et elle est définie en utilisant la formule (15).

$$F\left(B^i\right) = Z\left(X\right) + \Phi\left(B^i\right)_{\alpha,\beta} \tag{15}$$

Par conséquent, faible est (4) et proche de zéro est (14) le mieux est (15), alors meilleure est la qualité de coucou B^i.

4.3 Algorithme modifié de recherche discrète du coucou (modified discrete cuckoo search (MDCS))

Comme nous traitons le coucou discret avec la structure définie dans la figure 4.1, nous devons adapter une fonction de mappage *f* pour transformer chaque indice de processeur *k* en une valeur réelle *x* adaptée.

$$\begin{aligned} & f : \mathbb{N} \to [0,1] \\ & k \mapsto \begin{cases} x = \dfrac{1}{k}, k \neq 0 \\ x = 0, k = 0 \end{cases} \end{aligned} \tag{16}$$

Pour le mappage inverse, la fonction *g* est présentée comme suit:

$$\begin{aligned} & g : [0,1] \to \mathbb{N} \\ & x \mapsto \begin{cases} k = \lceil |x \times K| \rceil, |x| < 1 \\ k = \lceil |x \times K| \rceil \text{modulo } \text{K}, \text{sinon} \end{cases} \end{aligned} \tag{17}$$

Où, K est le nombre de processeurs dans le système et k est l'indice du processeur après de mapper la valeur x en utilisant la fonction g.

Pour améliorer la formule de marche aléatoire locale, nous ajoutons une stratégie de mise à jour dynamique pour les paramètres α et p_a. En outre, nous introduisons deux nombres aléatoires β_1 et β_2 pour assurer plus de variabilité entre les valeurs des composants des nids. Ainsi (10) devient

$$\begin{cases} \delta_i = \left(fit_i - fit_{best}\right)/\left(fit_{best} - fit_{worst}\right) \\ \eta = \left(fit_{avg} - fit_{best}\right)/\left(fit_{best} - fit_{worst}\right) \\ \alpha_i = \alpha_{\min} + \delta_i\left(\alpha_{\max} - \alpha_{\min}\right) \\ p_a = p_{\min} + \eta\left(p_{\max} - p_{\min}\right) \\ x_{i,k}^{t+1} = x_{i,k}^{t} + \alpha_i s \otimes H\left(p_a - \varepsilon\right) \otimes \left(\beta_1 x_{l,k}^{t} - \beta_2 x_{j,k}^{t}\right) \end{cases} \tag{18}$$

Où, fit_i, fit_{best} , fit_{agv} and fit_{worst} sont : la valeur de la fonction d'évaluation du nid i, la valeur de la fonction d'évaluation moyenne des nids actuels, la valeur de la fonction d'évaluation du meilleur et pire nid, respectivement. Aussi, $x_{i,k}^{t} = f(B_k^i)$, $x_{l,k}^{t} = f(B_k^l)$,et $x_{j,k}^{t} = f(B_k^j)$, où B^l and B^j sont deux nids choisis aléatoirement. La taille de pas s et le processus de marche aléatoire global sont calculés comme dans [27].

Aussi, pour améliorer la formule de marche aléatoire globale, nous ajoutons la stratégie de mise à jour dynamique pour le paramètre θ. Donc (11) devient.

$$\begin{cases} \delta_i = \left(fit_i - fit_{best}\right)/\left(fit_{best} - fit_{worst}\right) \\ \theta_i = \theta_{\min} + \delta_i\left(\theta_{\max} - \theta_{\min}\right) \\ \alpha_i = \alpha_{\min} + \delta_i\left(\alpha_{\max} - \alpha_{\min}\right) \\ x_{i,k}^{t+1} = x_{i,k}^{t} + \alpha_i Lévy(s, \theta_i) \end{cases} \tag{19}$$

Algorithm 5.1 Pseudo code de **MDCS**

(1) Initialiser les nids B^i (i = 1, 2, …, n) comme dans la figure 4.1

(2) Calculer la function objective F_i pour chaque nid i en utilisant equ.(15)

(3) Tanque (t < Itération Max ou critère d'arrêt) **Faire**

(4) Générer un coucou aléatoire B^i

(5) Mapper chaque composant B_k^i à une valeur réelle $x_{i,k}^t$ en utilisant l'équation (16)

(6) Appliquer une marche aléatoire globale à chaque valeur $x_{i,k}^t$ en utilisant l'équation (19)

(7) Mapper la nouvelle valeur $x_{i,k}^{'}$ à un index de processeur en utilisant l'équation (17)

(8) Evaluer sa fonction objective F_i en utilisant l'équation (15)

(9) Choisissez un nid parmi *n,* soit B^i au hasard

(10) **Si** ($F_i < F_j$) **Alors** Remplacer B^j par B^i ;

(11) Abandonner une fraction p_a des nids les plus mauvais

(12) Choisissez au hasard deux nids B^m and B^h du reste

(13) Mapper chaque composant de B^i, B^m and B^h en utilisant l'équation (16)

(14) Appliquer la marche locale à $x_{i,k}^t$ en utilisant les valeurs de $x_{m,k}^t$ et $x_{h,k}^t$ dans l'équation (18)

(15) Appliquer (17) à de nouvelles valeurs pour obtenir le nouveau nid

(16) Répétez depuis (12) jusqu'à ce que tous les nids abandonnés soient remplacés

(17) Gardez les meilleurs nids en utilisant l'équation (15)

(18) Classez les nids et trouvez le meilleur

(19) FinTanque

4.4 Evaluation de performance

Pour évaluer les performances de MDCS, dans des systèmes distribués hétérogènes, des instances de problèmes générées de manière aléatoire sont utilisées, car un ensemble de repères standard généralement accepté n'existe pas. Nous avons choisi de tester l'algorithme 5.1 en simulant un large éventail de scénarios similaires à ceux utilisés par d'autres chercheurs [61,80]. Un ensemble de données de simulation est créé en faisant varier un ensemble de paramètres qui détermine les caractéristiques des instances des problèmes générées. Les principaux paramètres du problème et du système sont décrits dans le tableau 5.1 et leurs valeurs dans le tableau 5.2.

Tableau 4-1 Désignation des paramètres du système et du problème

Parameter	**Designation**
N	Nombre de tâches
K	Nombre de processeurs
DS	Densité d'interaction de la tache
CCR	Rapport de communication au de temps calcul
ec_{ik}	Temps d'exécution attendu de la tâche t_i sur le processeur p_k
w_{lk}	Taux de transfert de données entre les processeurs p_l et p_k
d_{ij}	Quantité de données transférées entre les tâches t_i et t_j
m_i	Mémoire requise par la tâche t_i
M_k	Capacité de mémoire disponible sur le processeur p_k
l_i	Charge de traitement demandé t_i
L_k	Capacité de traitement disponible sur le processeur p_k
λ_k	Taux d'échec du processeur p_k
μ_{lk}	Taux d'échec du lien entre p_k et p_l

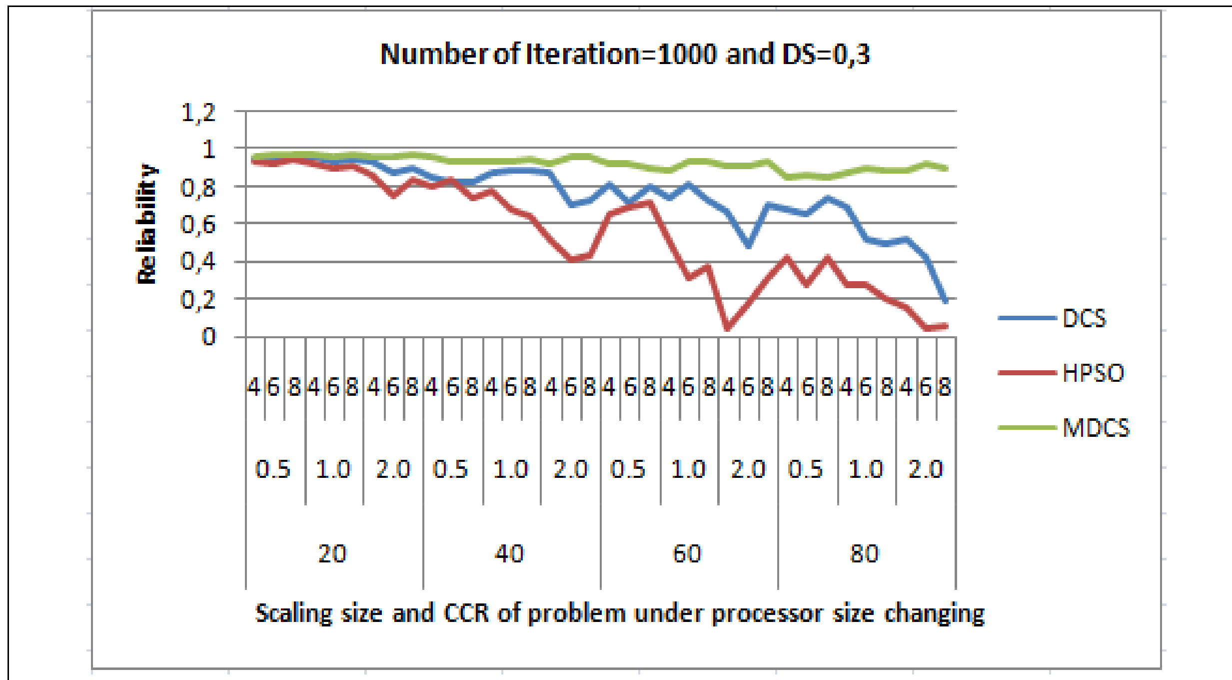
Tableau 4-2 Valeurs des paramètres système et problème.

Parameter	**Possible value**
N	(20,40,60,80)
K	(4,6,8)
DS	(0.3,0.5,0.8)
d_{ij}	Généré aléatoirement de sorte que le CCR et 0.5, 1.0 ou 2.0
ec_{ik}	Uniforme (15,25)
w_{lk}	Uniforme (1,10)
λ_k	Uniforme (0.00005, 0.00010)
μ_{lk}	Uniforme (0.00015, 0.00030)
m_i	Uniforme (15,25)
M_k	Varie de *R/K* to *2R/K*, ou $R = \sum_{i=0}^{N-1} m_i$
li	Uniforme (15,25)
L_k	Varie de *R/K* to *2R/K*, ou $R = \sum_{i=0}^{N-1} l_i$

Tableau 4-3 Paramètres des MDCS et l'HPSO

Parameter	**MDCS**	**HPSO**
n (population)	20	20
itr_{max} (nombre d'itérations maximal)	1000	1000
$(\alpha_{\min}, \alpha_{\max})$	(0.01, 0.1)	
$(p_{\min}, p_{\max})$	(0.1, 0.8)	
$(\theta_{\max}, \theta_{\min})$	(1,3)	
(ζ_1, ζ_2)		(2.05, 2.05)

Les paramètres de MDCS et HPSO sont également présentés dans le tableau 5.3. La figure 5.1 montre que pour DS = 0.3 (problème de faible complexité), la fiabilité donnée par MDCS est comprise entre 96% et 85%, alors que pour DCS, elle est comprise entre 96% et 19%, et pour HPSO est entre 95% et 4%.



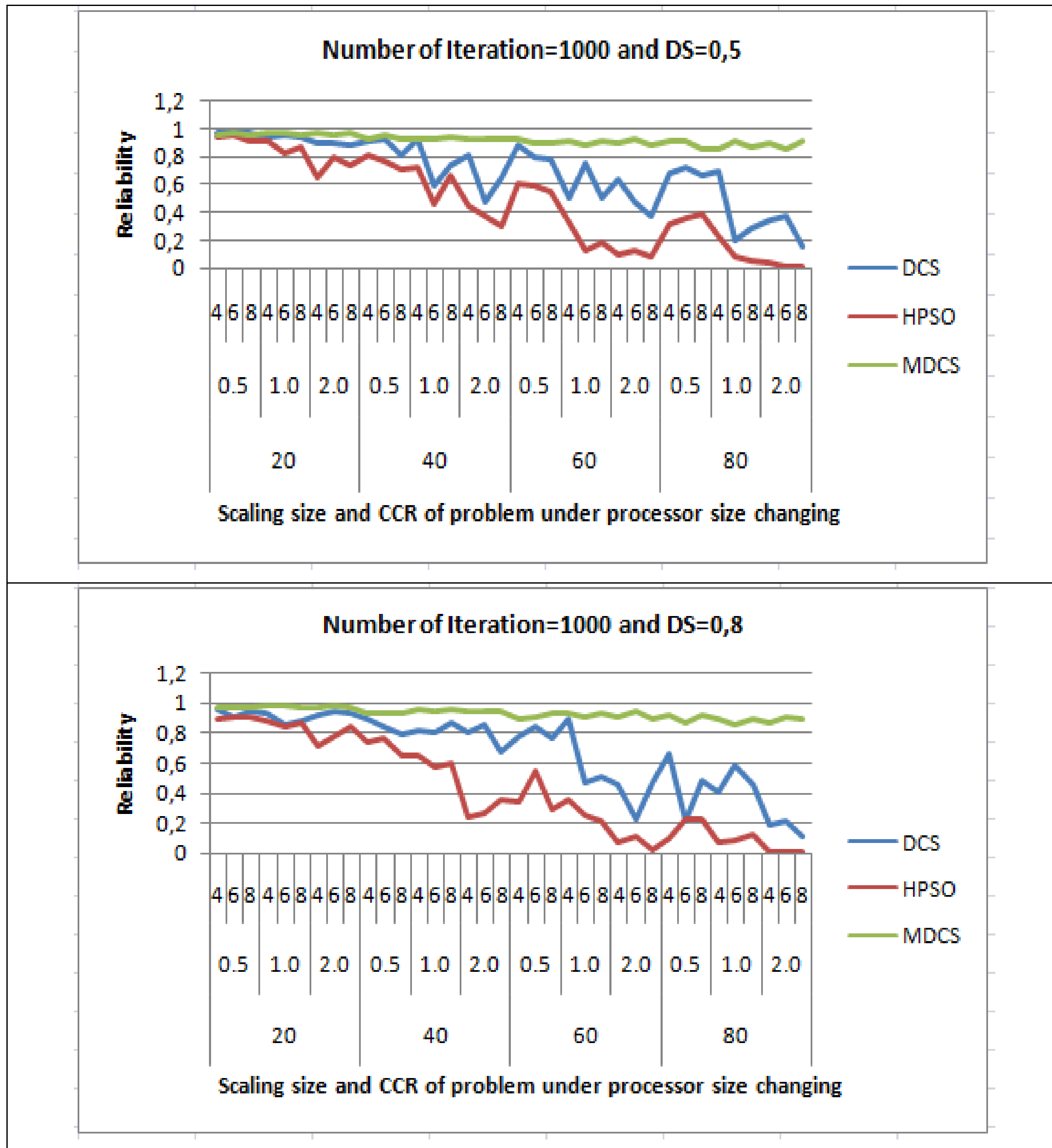


Figure 5-1 Comparaison de la fiabilité de DCS, MDCS and HPSO

Ainsi, MDCS est plus fiable que ses concurrents pour les problèmes de faible complexité. Aussi pour DS = 0,5 (problème de complexité moyenne) la fiabilité donnée par MDCS est comprise entre 98% et 86%, alors que pour DCS, elle est comprise entre 96% et 16%, et pour HPSO, elle est entre 96% et 1%. Ainsi, MDCS est plus fiable que ses concurrents pour les problèmes de complexité moyenne. Enfin, pour DS = 0,8 (problème à haute complexité), la

fiabilité donnée par MDCS est comprise entre 98% et 86%, alors que pour DCS, elle est comprise entre 96% et 11% et pour HPSO, elle est entre 96% et 0,4%. Ainsi, MDCS est plus fiable que ses concurrents pour les problèmes de grande complexité.

4.4.1 Conclusion de la contribution

Dans cette partie, l’algorithme modifié de recherche discret du coucou (MDCS) est présenté. MDCS est appliqué au problème de l'allocation des tâches qui maximise la fiabilité du système distribué hétérogène. Pour montrer l'efficacité et l'efficience de MDCS, MDCS est testé sous des problèmes aléatoires en utilisant les paramètres de densité et de CCR. L'évolutivité de MDCS est testée en faisant varier la taille du problème et du processeur. MDCS est comparé à l'algorithme hybride d'optimisation de l'essaim de particules (HPSO) et DCS, qui est similaire à MDCS sans formule d'amélioration. Les résultats obtenus montrent clairement que le MDCS est plus fiable que HPSO et DCS tous les cas de tests.

4.5 Conclusion

Dans ce chapitre nous avons présenté l’algorithme MDCS qui une version amélioré du l’algorithme classique en l’adaptant pour le problème d’allocation des taches dans un système distribué. Dans ce qui suit une conclusion générale qui résume nos contributions ainsi que les futures pistes est introduite.

Conclusion générale et perspectives

Les travaux présentés dans cette thèse s'inscrivent dans l'objectif global de la conception de système distribué de haute fiabilité sans redondance matérielle ou logicielle en basant sur les métaheuristiques. La partie concernée est celle de l'ordonnancement et la distribution des taches (allocation spatiale et temporelle des taches). Les métaheuristiques sont choisies pour leurs efficacités à résoudre des problèmes NP-complet comme le notre dans un temps raisonnable avec une qualité de solution qui est proche de la solution optimale.

Les résultats des travaux de recherche effectués ont donné lieu à deux algorithmes améliorés permettant de concevoir un noyau distribué qui peut gérer une allocation des taches sur un système distribué hétérogène tout en assurant sa fiabilité.

Le premier algorithme que nous avons appelé, l'algorithme modifié de la recherche bactérienne (modified bacterial foraging algorithm (MBFA), qui est une version améliorée de l'algorithme classique de recherche bactérienne par un modèle qui prend en considération les solutions de faible fiabilité et il essaie de les améliorer pour quelle soient de bonne fiabilité. Aussi, cet algorithme est adapté au problème de distribution et d'ordonnancement fiable en utilisant une nouvelle technique de mappage. Pour augmenter la précision de recherche de MBFA, MBFA est renforcé par une métaheuristique de recuit simulé.

Le deuxième algorithme est appelé, l'algorithme modifié de recherche discret du coucou (modified discret cuckoo search algorithm (MDCS)). Dans cet algorithme nous avons présenté des modèles qui permettent une mise à jour dynamique des paramètres de l'algorithme MDCS avec une nouvelle technique de mappage.

Les deux algorithmes présentés offrent une meilleur fiabilité pour le problème de distribution et ordonnancement sans aucune redondance matérielle ou logicielle en profitant de la simplicité de conception et d'implémentation offertes par les métaheuristiques.

Enfin, les travaux présentés dans cette thèse offrent un certain nombre de perspectives :

- Avant tout, les deux algorithmes, MBFA et MDCS, doivent être testés plus extensivement sur des graphes d'algorithmes et d'architectures avec de grande taille afin de confirmer la stabilité de ces algorithmes.
- Ensuite, ces algorithmes doivent être testés sur des cas réels afin d'observer leurs vrais comportement et leur niveau de confiance qui est lié à leurs fiabilité.

- Les deux algorithmes peuvent être étendus pour résoudre des problèmes multi-objectifs comme l'optimisation de l'énergie en assurant une grande fiabilité avec un temps d'exécution minimal.

Références

[1] El-Ghazali Talbi, Metaheuristics from design to implementation,Wiley, 2009

[2] D. P. Bertsekas. Nonlinear Programming. Athena Scientific, Belmont, MA, 2004.

[3] G. Nemhauser and L. Wolsey. Integer and Combinatorial Optimization. Wiley, 1999.

[4] C. H. Papadimitriou and K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, New York, 1982.

[5] R. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artificial Intelligence, 27(1):97–109, 1985.

[6] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, 1995.

[7] R. Bellman. Dynamic Programming. Princeton University Press, Princeton, NJ, 1957.

[8] E. Falkenauer. Genetic Algorithms and Grouping Problems. Wiley, 1998.

[9] M. Gen and R. Cheng. A survey of penalty techniques in genetic algorithms. In T. Fukuda and T. Furuhashi, editors, International Conference on Evolutionary Computation,Nagoya, Japan, 1996, pp. 804–809.

[10] J. T. Richardson, M. R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In 3rd International Conference on Genetic Algorithms (ICGA'3), 1989, pp. 191–197.

[11] J. Joines and C. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with genetic algorithms. In D. Fogel, editor, 1st IEEE Conference on Evolutionary Computation. IEEE Press, 1994, pp. 579–584.

[12] M. Gendreau, A. Hertz, and G. Laporte. A tabu search heuristic for the vehicle routing problem. Management Science, 40:1276–1290, 1994.

[13] D. Dasgupta and M. Michalewicz. Evolutionary Algorithms in Engineering Applications. Springer, 1997.

[14] T. G. Crainic and M. Toulouse. Parallel strategies for metaheuristics. In F.W. Glover and G. A. Kochenberger, editors, Handbook of Metaheuristics. Springer, 2003, pp. 475–513.

[15] E. H. L. Aarts and J. K. Lenstra. Local Search in Combinatorial Optimization. Wiley, 1997.

[16] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing.Science, 220(4598):671–680, 1983.
[17] V. Cerny. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of Optimization Theory and Applications, 45:41–51, 1985.
[18] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. Journal of Chemical Physics, 21:1087–1092,1953.
[19] M. D. Huang, F. Romeo, and A. L. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. In IEEE International Conference on Computer-Aided Design, Santa Clara, CA, 1986, pp. 381–384.
[20] E. H. L. Aarts and R. J. M.Van Laarhoven. Simulated Annealing: Theory and Applications. Reidel, Dordrecht, 1987.
[21] H. Maaranen, K. Miettinen, and A. Penttinen. On initial populations of a genetic algorithm for continuous optimization problems. Journal of Global Optimization, 37:405–436,2007.
[22] Passino, K.M. (2002) 'Biomimicry of bacterial foraging for distributed optimization and control', IEEE Transactions on Control Systems Magazine, Vol. 22, No.3, pp.52–67.
[23] X.S.Yang, A new metaheuristic bat-inspired algorithm, in: J.R. Gonzalez, et al.(Eds.), Nature Inspired Cooperative Strategies for Optimization (NISCO 2010).Studies in Computational Intelligence, Springer Berlin, 284, Springer, Berlin, pp. 65–74,2010.
[24] Yang .X.-S, S. Deb. 2009. Cuckoo search via Levy flights. In: Proc. of World Congress on Nature & Biologically Inspired Computing (NaBIC 2009), December ,India,
[25] Abbache, F. and Kalla, H. Maximizing Reliability of Heterogeneous Distributed System Using an Adapted Discrete Flower Pollination Algorithm for Task Allocation Problem, SCS-NCC' 2018, Proceedings of the 21st Saudi Computer Society National Computer Conference, Riyadh, Kingdom of Saudi Arabia, 25-26 April, 2018,
[26] Kumar,H.,Chauhan,N. and Yadav,P. (2018) 'A task allocation model for minimising system cost and maximising reliability of distributed computing system', International Journal of Communication Networks and Distributed Systems , Vol. 20 No. 2,pp. 226-243.

[27] Yang .X.-S, S. Deb. 2010. Engineering optimization by cuckoo search, Int. J. Mathematical Modelling and Numerical Optimisation,Vol. 1, No. 4, 330-343

[28] Abbache, F. and Kalla, H. 'Task allocation-based modified bacterial foraging algorithm for maximising reliability of a heterogeneous distributed system', Int. J.Communication Networks and Distributed Systems, (paper in press)

[29] Abbache, F. and Kalla, H. 'Maximizing Reliability of Heterogeneous Distributed System Using Bio-Inspired Technique for Task Allocation Problem', CSAI 2017 Proceedings of the 2017 International Conference on Computer Science and Artificial Intelligence, Jakarta, Indonesia — December 05 - 07, 2017 ,Pages 131-136,

[30] Abbache, F. and Kalla, H.' Task allocation Based Modified Discrete Bat Algorithm for Maximizing Reliability of a Heterogeneous Distributed System',ICNAS 2017, Proceedings of the 3rd International Conference on Networking and Advanced Systems, Annaba,Algeria, Decembre 13-14,2017, pages 25-30

[31] J. H. Holland. Adaptation in Natural and Artificial Systems.University ofMichigan Press, Ann Arbor, MI, 1975.

[32] D. E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison- Wesley, 1989.

[33] A. B. Hadj-Alouane and J. C. Bean. A genetic algorithm for the multiple-choice integer program. Operations Research, 45(1):92–101, 1997.

[34] Coulouris.G, Dollimore.J, Kindberg.T, Blair.G(2012) DISTRIBUTED SYSTEMS,Concepts and Design Fifth Edition

[35] Daniel, M. et al. (2015) 'Task allocation in organic computing systems: networks with configurable helper units', Int. J. of Autonomous and Adaptive Communications Systems,Vol.8,No.1, pp.60 – 80

[36] Kousalya,A., Radhakrishnan,R. (2017) 'Hybrid algorithm based on genetic algorithm and PSO for task scheduling in cloud computing environment', Int. J. of Networking and Virtual Organisations,Vol.17, No.2/3, pp.149 – 157

[37] Althebyan, Q. et al.(2017) 'A scalable map reduce tasks scheduling: a threading-based approach', Int. J. of Computational Science and Engineering,Vol.14, No.1, pp.44 – 54.

[38] Yongho, K., Eric, T. (2016) 'A realistic decision making for task allocation in heterogeneous multi- agent systems', Procedia Computer Science, Vol.94, pp. 386-391.

[39] Wanqing, Z. and Qinggang, M. and Paul, W. H. C. (2016) ‘A heuristic distributed task allocation method for multivehicle multitask problems and its application to search and rescue scenario’, IEEE Transactions on Cybernetics ,Vol.46, No. 4,pp. 902 – 915.
[40] Wanyuan, W.,Yichuan, J. (2015) ‘Multiagent-based allocation of complex tasks in social networks’, IEEE Transactions on Emerging Topics in Computing, Vol.3, No. 4, pp.571 – 584.
[41] Wenzhong, .G et al.(2015) ‘A PSO-Optimized Real-Time Fault-Tolerant Task Allocation algorithm in Wireless Sensor Networks’, IEEE Transactions on Parallel and Distributed Systems,Vol.26, No. 12,pp.3236 – 3249.
[42] Keqin L. (2017) ‘Optimal task dispatching on multiple heterogeneous multiserver systems with dynamic speed and power management’, IEEE Transactions on Sustainable Computing,Vol.2, No.2, pp.167 – 182.
[43] Yoosefi, .A, Naji, .H (2017) ‘A clustering algorithm for communication-aware scheduling of task graphs on multi-core reconfigurable systems’, IEEE Transactions on Parallel and Distributed Systems,Vol.28, No.10,pp.2718 – 2732.
[44] Chen, W.H. and Lin, C.S. (2000) ‘A hybrid heuristic to solve a task allocation problem’, Comput. Oper. Res,Vol. 27,No. 3, pp.287–303.
[45] Salcedo-Sanz, S., Xu, Y and Yao, X. (2006) ‘Hybrid meta-heuristics algorithms for task assignment in heterogeneous computing systems’. Comput. Oper. Res,Vol.33, No. 3,pp.820–835
[46] Ucar, B., Aykanat, C., Kaya, K. and Ikinci, M.(2006).’Task assignment in heterogeneous computing systems’, J. Parallel Distrib. Comput, Vol.66,pp.32–46.
[47] Yin, P.Y, Yu S.S., Wang, P.P. and Wang, Y.T. (2006). ‘A hybrid particle swarm optimization algorithm for optimal task assignment in distributed systems’, Comput. Stand.Interfaces ,Vol. 28,pp.441–450.
[48] Hamam, Y. and Hindi, K.S. (2000) ‘Assignment of program modules to processors: a simulated annealing approach’, European J. Oper. Res.Vol.122, pp. 509 –513.
[49] Ahmad, and Dhodhi, M.K. (1995). ‘Task assignment using problem-space genetic algorithm, concurrency’, Pract. Exp. Vol 7, pp.411–428.
[50] Salman, A., Ahmad, I., and Al-Madani, S. (2002). ‘Particle swarm optimization for task assignment problem’, Microprocess. Microsyst.Vol. 26,pp.363–371.

[51] Shen, C. and Tsai, W. (1985) 'A graph matching approach to optimal task assignment in distributed computing systems using minimax criterion',IEEE Trans. Comput,Vol. 34,pp.197– 203.
[52] Kafil, M. and Ahmad, I. (1998) 'Optimal task assignment in heterogeneous distributed computing systems', IEEE Concurr, Vol. 6, pp.42–51.+++Vol 54,pp.530–548.
[54] Misra,S. et al.,(2017) 'Learning automata-based fault-tolerant system for dynamic autonomous unmanned vehicular networks', IEEE Systems Journal,Vol.11, No.4,pp.2929 – 2938
[55] Kuang, Z. and Chen, Z.(2017) 'A high reliability and low latency routing algorithm in cognitive wireless mesh networks', Int. J. Communication Networks and Distributed Systems, Vol. 18, No. 1, pp.58–82.
[56] Kar,P. and Misra,S. (2016) 'Reliable and efficient data acquisition in wireless sensor networks in the presence of transfaulty nodes', IEEE Transactions on Network and Service Management,Vol.13, No. 1, pp.99 – 112
[57] Moulik, S., Misra, S. and Das, (2017) 'D.AT-MAC: Adaptive MAC-frame payload tuning for reliable communication in wireless body area networks', IEEE Transactions on Mobile Computing,Vol. 16, No.06, pp: 1516-1529
[58] Hsieh, C.C. and Hsieh, Y.C (2003) ' Reliability and cost optimization in distributed computing systems', Comput. Oper. Res, Vol. 30, pp.1103–1119.
[59] Kumar,H.,Chauhan,N. and Yadav,P. (2018) 'A task allocation model for minimising system cost and maximising reliability of distributed computing system', International Journal of Communication Networks and Distributed Systems , Vol. 20 No. 2,pp. 226-243.
[60] Shatz, S.M, Wang J.P. and Goto M. (1992) 'Task allocation for maximizing reliability of distributed computer systems', IEEE Trans Comput.Vol.1,pp.1156–1168.
[61] Attiya, G. and Hamam, Y. (2006) 'Task allocation for maximizing reliability of distributed systems: a simulated annealing approach', J. Parallel Distrib.Comput. , Vol.66, pp.1259–1266.
[62] Kang, Q., Hong, H. and Jun, W. (2013) 'An effective iterated greedy algorithm for reliability-oriented task allocation in distributed computing systems', J. Parallel Distrib. Comput, Vol.73, pp.106–1115

[63] Rahimzadeh, F., Mohammad, L., Mahan, K. (2015) ‘ High reliable and efficient task allocation in networked multi-agent systems autonomous agents and multi-agent systems’, Vol. 29, No.6,pp.1023–1040.
[64] Wilson,K,J. and Quigley,J.(2016) ‘Allocation of tasks for reliability growth using multi-attribute utility’, European Journal of Operational Research,Vol. 255, No. 1, pp.259-271.
[65] Li,K et al. (2015) ‘Maximizing reliability with energy conservation for parallel task scheduling in a heterogeneous cluster’, Information Sciences,Vol. 319, pp.113-131.
[66] Li,K.,Zhang,L.,Xua and Y.Joint(2016) ‘Optimization of energy efficiency and system reliability for precedence constrained tasks in heterogeneous systems’, International Journal of Electrical Power & Energy Systems,Vol. 78,pp.499-512.
[67] Moghaddas,V.,Fazeli M. And Patooghy,A.(2016) ‘Reliability-oriented scheduling for static- priority real-time tasks in standby-sparing systems’, Microprocessors and icrosystems,Vol.45,pp.208-215.
[68] Xiao,Y. et,al. (2017) ‘A novel task allocation for maximizing reliability considering fault-tolerant in VANET real time systems’, IEEE 28th Annual International Symposium on Personal,Indoor, and Mobile Radio Communications (PIMRC),pp.1 – 7
[69] Elegbede C,A.O, Chu ,C., Adjallah, K.H., Yalaoui, F. (2003) ‘ Reliability allocation through cost minimization’. IEEE Trans. Reliab. Vol.52, pp.106–111.
[70] Girault, A. and Kalla, H. (2009) ‘A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate’, IEEE Transactions on Dependable and Secure Computing ,Vol.6, pp. 241– 254.
[71] Laiping, Z. , Yizhi, R. ,and Kouichi S. (2013) ‘Reliable workflow scheduling with less resource redundancy’, Parallel Computing, Vol. 39, pp. 567– 585.
[72] Kumar, A., and Agrawal, D.P(1993) ‘A generalized algorithm for evaluating distributed program reliability’, IEEE Trans. Reliab, Vol. 42, pp. 416– 426.
[73] Tom, P.A. and Murthy, C. (1998). ‘Algorithms for reliability-oriented module allocation in distributed computing systems’, J. Syst. Softw, Vol.40,pp.125–138.
[74] Kim ,H. and Kim ,P. (2017) ‘Reliability–redundancy allocation problem considering optimal redundancy strategy using parallel genetic algorithm’, Reliability Engineering & System Safety,Vol. 159, pp.153-160.

[75] Li ,K. ,Wang,S. ,Jing, M., Guoqing, X. and Keqin, L. (2017) ‘A reliability- aware task scheduling algorithm based on replication on heterogeneous computing systems’, Journal of Grid Computing, Vol. 15, No. 1, pp. 23–39.

[76] Chen, C. (2016) ‘Task scheduling for maximizing performance and reliability considering fault recovery in heterogeneous distributed systems’, IEEE Transactions on Parallel and Distributed Systems, Vol. 27, No. 2, pp. 521 – 532.

[77] Azimzadeh,F. and Biabani,F. (2017) ‘Multi-objective job scheduling algorithm in cloud computing based on reliability and time’, IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC), pp.169 – 176.

[78] Wei, T. et al. (2017) ‘Reliability and temperature constrained task scheduling for makespan minimization on heterogeneous multi-core platforms’, Journal of Systems and Software,Vol.133,pp.1-16.

[79] Hsieh, C.C. (2003) ‘Optimal task allocation and hardware redundancy policies in distributed computing systems’, European J. Oper. Res, Vol.147, pp. 430–447.

[80] Kang, Q.M, He, H., Song, H.M. and Deng, R. (2010). ‘Task allocation for maximizing reliability of distributed computing systems using honeybee mating optimization’, J. Syst.Softw,Vol.83, pp.2165–2174.

[81] Kartik, S. and Murthy, C.S.R. (1997) ‘Task allocation algorithms for maximizing reliability of distributed computing systems’, IEEE Trans. Comput. , Vol. 46, pp. 719–724.

[82] Mahmood, A. (2001). ‘Task allocation algorithms for maximizing reliability of heterogeneous distributed computing systems’, Control Cybernet,Vol. 30,pp. 115–130.

[83] Lin, M.S., and Chen, D.J (1997) ‘The computational complexity of the reliability problem on distributed systems’. Information Processing Letters ,Vol. 64,pp. 143–147.

[84] Vidyarthi, D.P. and Tripathi, A.K. (2001).’ Maximizing reliability of distributed computing systems with task allocation using simple genetic algorithm’, J. Syst. Archit,Vol. 47, pp.549–554.

[85] Andrew S. Tanenbaum (2006) DISTRIBUTED SYSTEMS ,principles and paradigms, second edition

[86] Neuman, B.: "Scale in Distributed Systems." In Casavant, T. and Singhal, M. (eds.),Readings in Distributed Computing Systems, pp. 463-489. Los Alamitos, CA: IEEE Computer Society Press, 1994. Cited on pages 9, 12,624.

[87] Foster, I., Kesselman, C., and Tuecke, S.: "The Anatomy of the Grid, Enabling

Scalable Virtual Organizations." Journal of Supercomputer Applications, (15)3:200-222, Fall 2001. Cited on page 19.
[88] Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., and Wetherall, D.: "System Support for Pervasive Applications." ACM Trans. Compo Syst., (22)4:421-486, Nov. 2004. Cited on page 25.
[89] Wu, D., Hou, Y., Zhu, W., Zhang, Y., and Peha, J.: "Streaming Video over the Internet:Approaches and Directions." IEEE Trans. Circuits & Syst. Video Techn., (11)1:1-20, Feb. 2001.Cited on page 159.