Mémoire de master en Informatique

Résolution du *Pigment Sequencing Problem* avec les algorithmes génétiques

Présenté par:
Tafsir GNA
Supervisé par:
Dr Ing. Vinasétan Ratheil HOUNDJI
&
Professeur Mahouton Norbert HOUNKONNOU

Institut de Formation et de Recherche en Informatique (IFRI)

17 septembre 2017

Sommaire

Introduction

- L'échelle de temps ;
- Le nombre de niveaux ;
- Le nombre de produit ;
- Les contraintes de capacité ;
- Les demandes ;
- Les coûts et temps de lancement ou préparation (setup).

- Problèmes de petite taille ou à courtes périodes ;
- Problèmes de grande taille ou à longues périodes ;
- Problèmes de très grande taille ou à très longues périodes.

- Discrete lot-sizing and scheduling problem (DLSP) ;
- Continuous setup lot-sizing problem (CSLP) ;
- Proportional lot-sizing and scheduling problem (PLSP) ;
- General lot-sizing and scheduling problem (GLSP).

Le dimensionnement de lots en planification de production

Classes de problèmes de dimensionnement de lots III

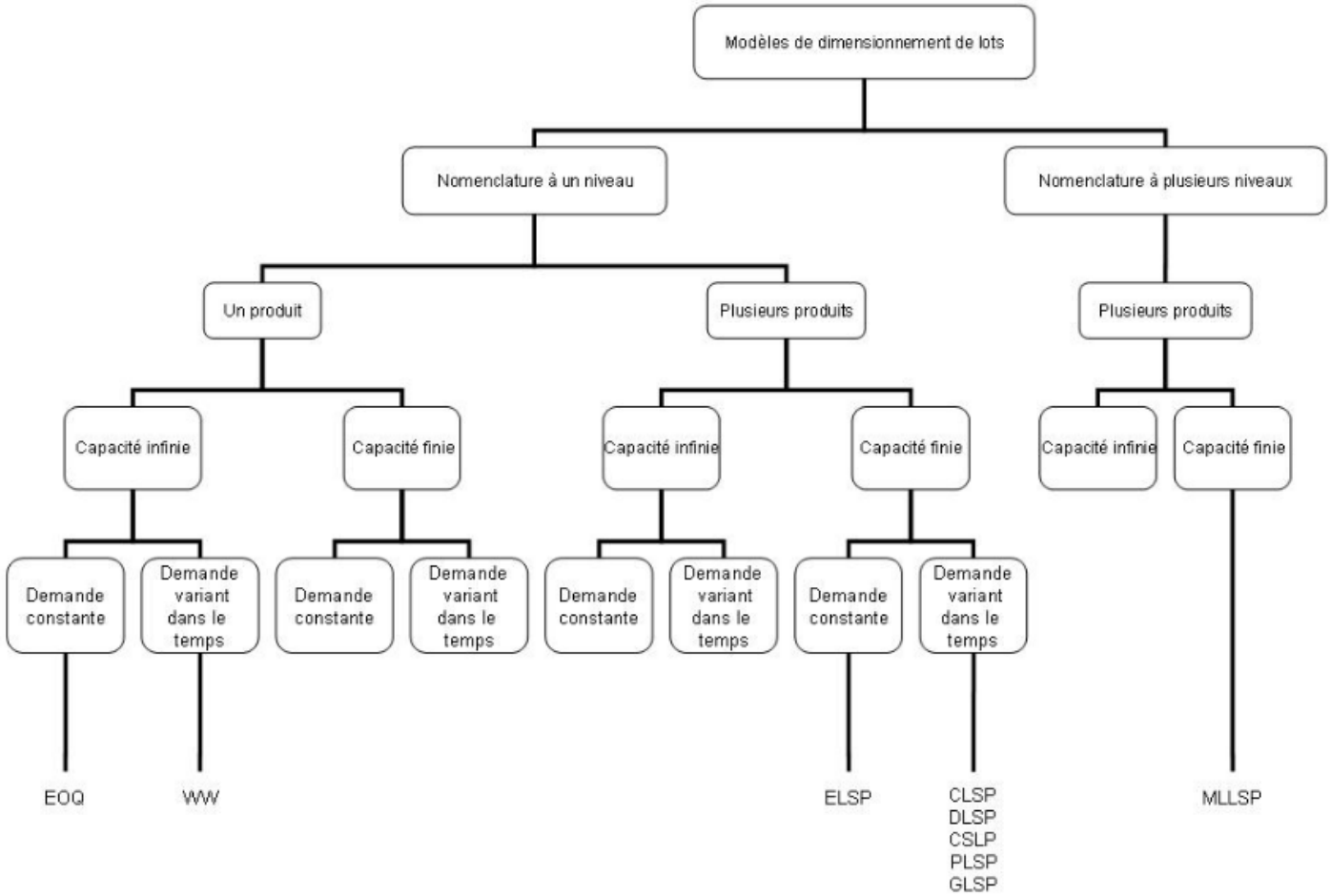


Figure – Exemple de classification des modèles de dimensionnement de lots []

Le "Pigment Sequencing Problem" (PSP)

Description

Objectif :

- trouver un plan de production de plusieurs articles à partir d'une machine avec des coûts de transition ;
- respectant la capacité de production de la machine ;
- minimisant les coûts de stockage et de transition.

Le "Pigment Sequencing Problem" (PSP)

Modèles et formulations

- Modèles basés sur la programmation en nombres mixtes[] (MIP1, MIP2, MIP3) ;
- Modèle basé sur la Programmation par contraintes[] ;
- Modèle basé sur le recuit simulé[].

Générer la population initiale P_i
Évaluer la population P_i
while *le critère de terminaison n'est pas satisfait* **do**
    Sélectionner les éléments de P_i à copier dans P_{i+1}
    Appliquer le croisement aux éléments de P_i et les mettre dans P_{i+1}
    Appliquer la mutation aux éléments de P_i et les mettre dans P_{i+1}
    Évaluer la nouvelle population P_{i+1}
    $P_i = P_{i+1}$
end

Algorithm 1: Algorithme génétique standard [?]

- la sélection ;
- le croisement ;
- la mutation.

- Les algorithmes génétiques parallèles de type master-slave ;
- Les algorithmes génétiques parallèles coarse-grained ;
- Les algorithmes génétiques parallèles fine-grained.

- Les algorithmes génétiques parallèles et hiérarchiques entre coarse-grained et master-slave ;
- Les algorithmes génétiques parallèles et hiérarchiques entre coarse-grained et fine-grained.

Outils de test

- Système d'exploitation : Linux Ubuntu 16.04 LTS ;
- Processeur : Intel® Core ™ i7 CPU L 640 @ 2.13GHz x 4 ;
- Mémoire : 3,7 Gio ;
- Type du système d'exploitation : 64 bits.

Modèle et formulation utilisés I

MIP1

$$min \sum_{i,j,t} q^{i,j} \chi_t^{i,j} + \sum_{i,t} h^i s_t^i \tag{1}$$

$$s_0^i = 0, \forall i \tag{2}$$

$$x_t^i + s_{t-1}^i = d_t^i + s_t^i, \forall i, t \tag{3}$$

$$x_t^i \leq y_t^i, \forall i, t \tag{4}$$

$$\sum_i y_t^i = 1, \forall t \tag{5}$$

$$\chi_t^{i,j} = y_{t-1}^i + y_t^j - 1, \forall i, j, t \tag{6}$$

$$x, y, \chi \in \{0,1\}, s \in \mathbb{N}, i \in \{0..NI\}, t \in \{1..NT\} \tag{7}$$

avec les variables de décisions suivantes :

Modèle et formulation utilisés II

MIP1

- x_t^i : variable binaire de production qui vaut 1 si l'article i est produit à la période t et 0 sinon ;
- y_t^i : variable binaire de setup qui vaut 1 si la machine est préparée pour la production de l'article i et 0 sinon ;
- s_t^i : variable entière de stockage qui contient le nombre d'articles i stockés à la période t ;
- $\chi_t^{i,j}$: variable binaire de transition qui vaut 1 si à la période t, on est passé de la production de l'article i à l'article j et 0 sinon.

$$ch_{Tn} = \{(I_{T1}), ..., (I_{T2}), ..., (I_{T3}), (I_{T4}), ..., (I_{T(n-1)}), ..., (I_{Tn})\}$$

où ch_{Tn} est un chromosome dont l'horizon de planification est de Tn périodes et I_{Ti} est la variable entière qui indique l'article produit à la période Ti.

Illustration avec les demandes $D_{I1} = (0, 1, 0, 0, 1)$ et $D_{I2} = (1, 0, 0, 0, 1)$

T1	T2	T3	T4	T5
2	1	0	2	1

T1, T2, T3,T4,T5 : Période 1,2,3,4,5

Figure – Représentation génétique adoptée

Aspects généraux aux deux méthodes de recherche proposées

Initialisation I

Algorithme : Processus de génération de la population initiale

Données : instance de PSP à traiter, taille de la population

Résultat : Population initiale constituée

$queue \leftarrow []$

$noeud \leftarrow nouveauNoeud()$

tant que *taille(populationInitiale) est inférieure à taillePopulation* **faire**

- **si** *noeud.chromosome est prêt* **alors**
  - populationInitiale.ajouter(noeud.chromosome)
- **sinon**
  - $noeudFils \leftarrow noeud.obtenirSuccesseurs()$
  - noeudFils.trier(decroissant)
  - queue.ajouter(noeudFils)
  - **si** *queue est vide* **alors**
    - **retourner** populationInitiale
  - **fin**
  - $noeud \leftarrow queue.dernier()$
- **fin**

fin

retourner populationInitiale

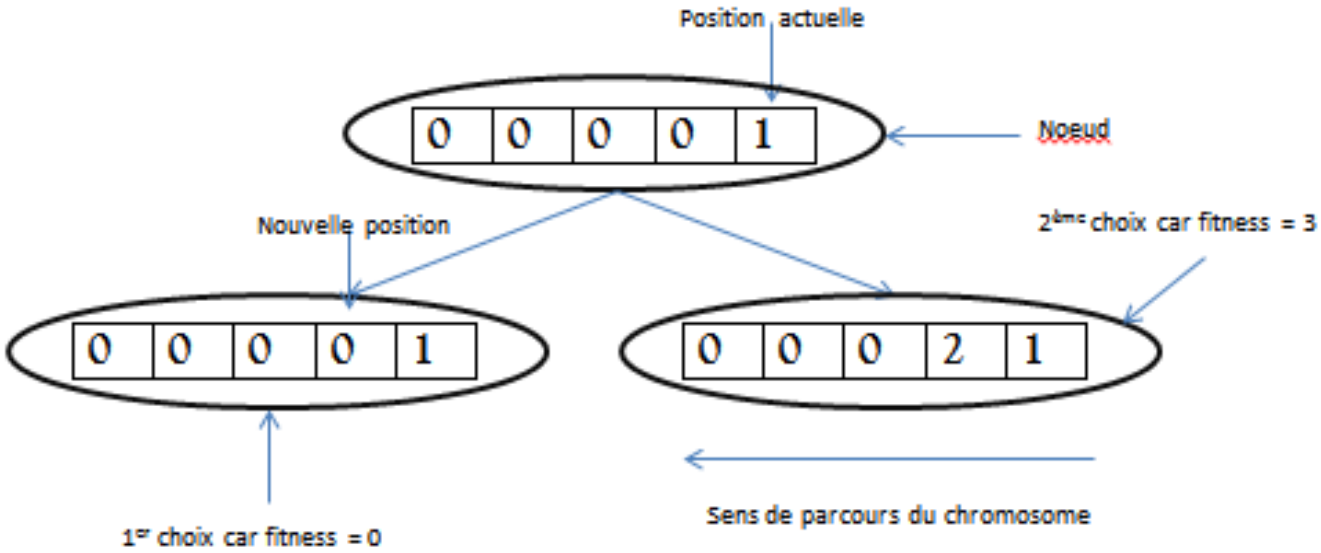


Figure – Schéma illustratif de l'application de l'algorithme du Hill climbing à une instance de PSP

Sélection

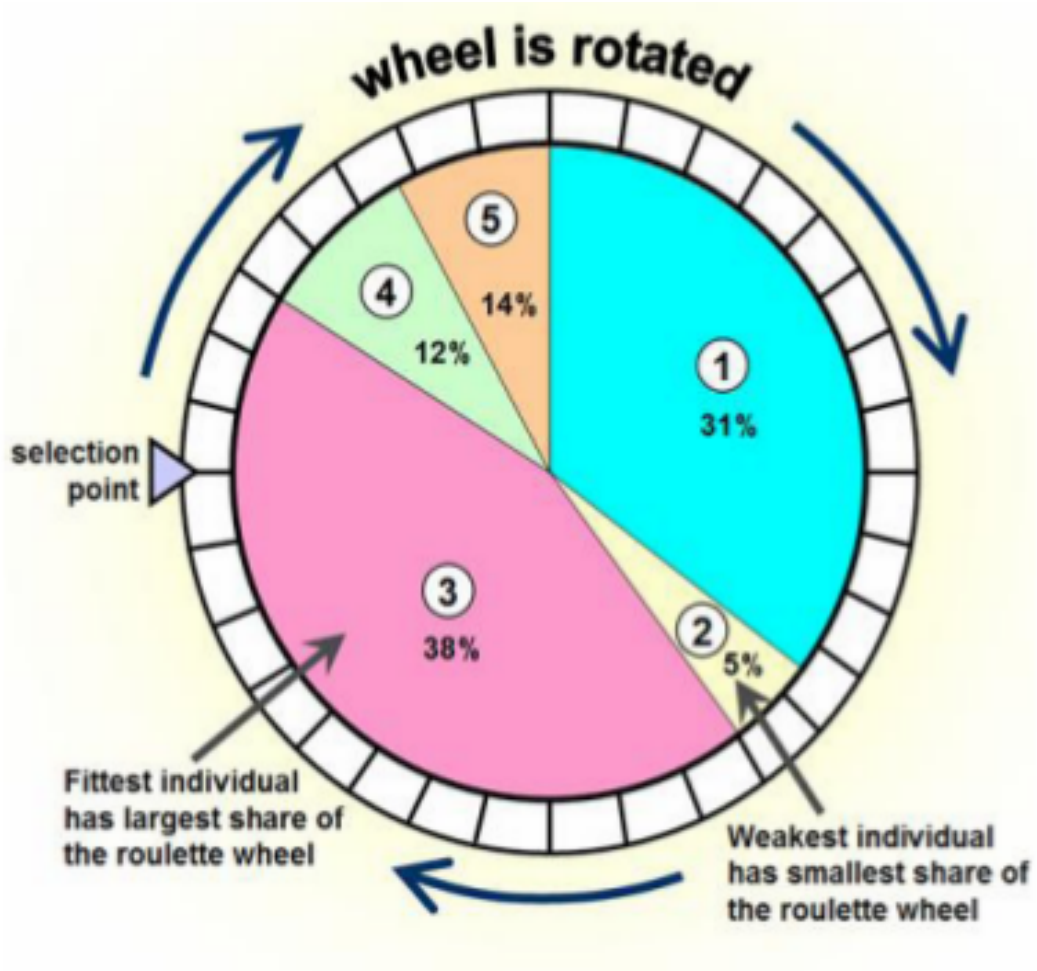


Figure – Schéma illustratif de la méthode de *roulette wheel*

Croisement

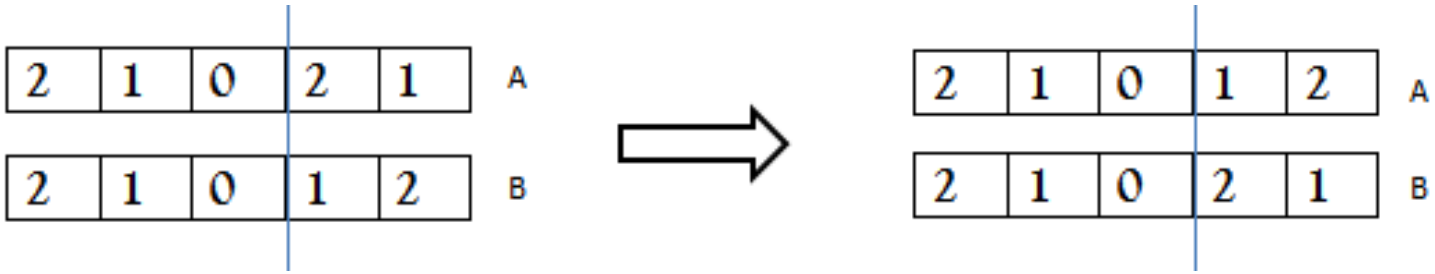


Figure – Schéma illustratif de la méthode de croisement utilisée

Mutation

Figure – Schéma illustratif de la méthode de mutation utilisée

Aspects généraux aux deux méthodes de recherche proposées

Evaluation

Algorithme : Algorithme utilisé dans le processus d'évaluation d'un chromosome

Données : chromosome, cout_stockage, cout_transition

Résultat : eval

```
eval ← 0
//On calcule le cout de stockage de chaque production
pour gene in chromosome faire
    si gene.value != 0 alors
        date_limite ← getDateLimite(gene)
        temp ← (date_limite − gene.periode) ∗ cout_stockage(gene.value)
        evaluation ← evaluation + temp
    fin
fin
//On calcule le cout de transition de entre deux productions
pour gene in chromosome faire
    si gene.valeur != 0 alors
        next_gene ← obtenirProchainGene(chromosome)
        si transition(gene, prochain_gene) est vrai alors
            temp ← cout_transition(gene, prochain_gene)
            evaluation ← evaluation + temp
        fin
    fin
fin
retourner evaluation
```

Deux critères de terminaison

- Convergence de la population sur une solution ;
- Absence de meilleures solutions à partir de celle trouvée.

Algorithme : Algorithme utilisé comme fonction de faisabilité

Données : chromosome, deadlines

```
// On commence par reduire le surplus de production ;
pour i ← 1 à Nombre_Periodes faire
    article ← chromosome.obtenirArticle(i) ;
    si estEnSurplus(article, deadline(i)) alors
        supprimerProduction(article) ;
    fin
fin
// On compense le manque de production ;
pour article in liste_articles faire
    article_deadlines ← deadlines(article) ;
    pour deadline in article_deadlines faire
        si nonProduit(dealine) alors
            produire(article);
        fin
    fin
fin
```

Problématiques :

- Fréquence de migration ;
- Choix et nombre de migrants ;
- Topologie de connexions ;
- Méthode d'intégration des migrants ;

Problématiques :

- Fréquence de migration ;
- Choix et nombre de migrants ;
- Topologie de connexions ;
- Méthode d'intégration des migrants ;

Méthodes de recherche proposées

Autres algorithmes implémentés I

- Hybridation

Algorithme : Algorithme utilisé comme fonction de faisabilité

Données : chromosome, deadlines

```
// On commence par reduire le surplus de production ;
pour i ← 1 à Nombre_Periodes faire
    article ← chromosome.obtenirArticle(i) ;
    si estEnSurplus(article, deadline(i)) alors
        supprimerProduction(article) ;
    fin
fin
// On compense le manque de production ;
pour article in liste_articles faire
    article_deadlines ← deadlines(article) ;
    pour deadline in article_deadlines faire
        si nonProduit(dealine) alors
            produire(article);
        fin
    fin
fin
```

- Table de hash

Algorithme : Algorithme implémentant l'utilisation de la table de hash dans nos méthodes proposées

Données : chromosome, table_hash, max_len

```
solution ← chromosome.solution;
si solution in table_hash alors
    indice ← table_hash[solution] ;
    chromosome.valeurFitness ← table_hash[solution][valeurFitness] ;
sinon
    chromosome.valeurFitness ← evaluation(solution) ;
    table_hash.ajouter([solution, chromosome.valeurFitness]);
    si longueur(table_hash) > max_len alors
        table_hash.enleverDernier() ;
    fin
fin
```

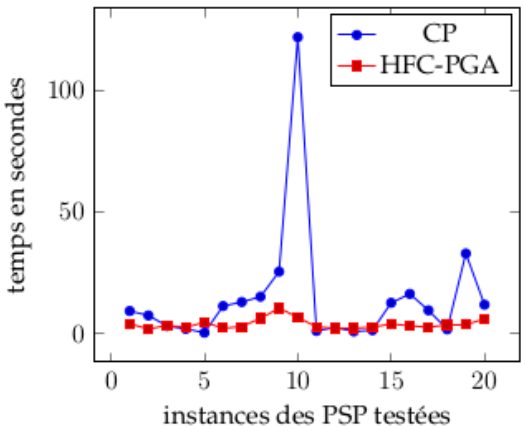
Résultats et discussion

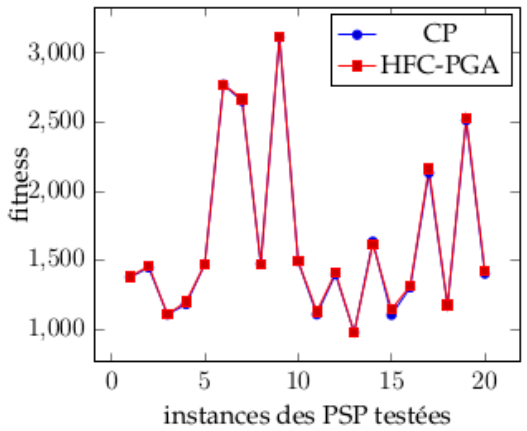
Données et paramètres de test

- Taille de la population : 25 individus par processus ;
- Probabilité de mutation : 5% ;
- Probabilité de croisement : 80% ;
- Nombre de migrants : 1 individu ;
- Nombre de processus esclaves : 2 processus ;
- Nombre de processus principaux : 2 processus ;
- Nombre de générations avant migration : 0 génération (la migration intervient après une convergence).

Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques fine-grained et coarse-grained

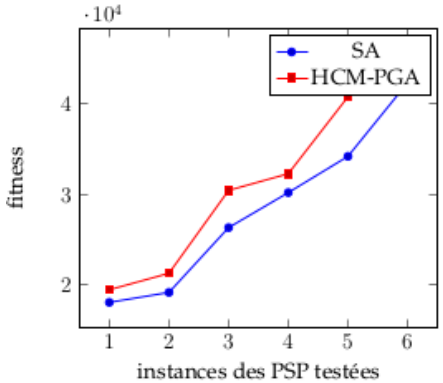
HFC-PGA et CP

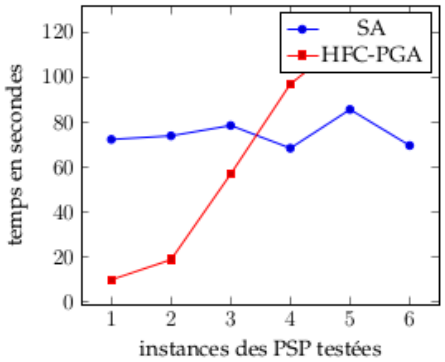




Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques fine-grained et coarse-grained

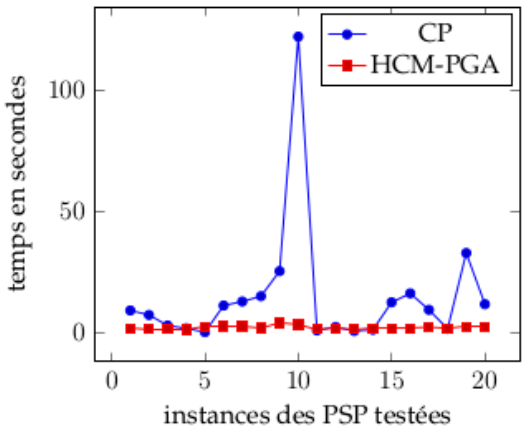
HFC-PGA et SA

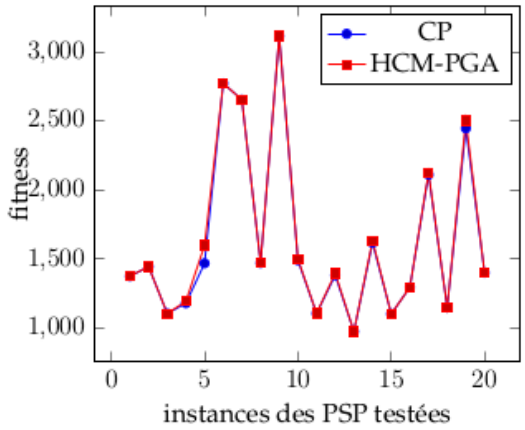




Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques master-slave et coarse-grained

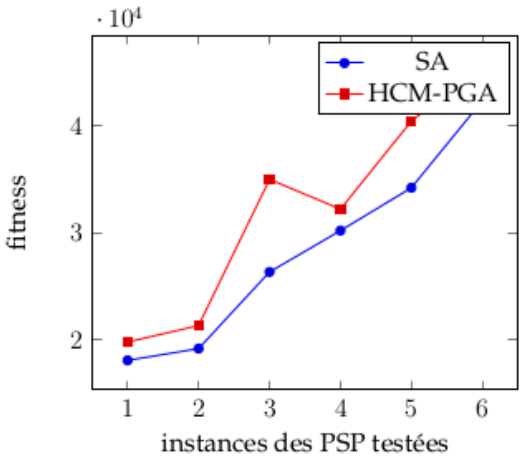
HCM-PGA et CP

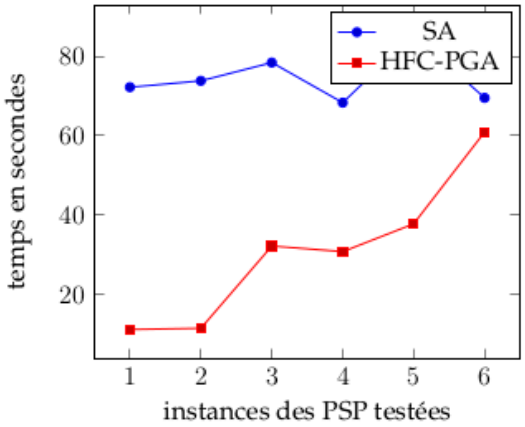




Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques master-slave et coarse-grained

HCM-PGA et SA





Conclusion

Merci pour votre attention