Problème d'Ordonnancement des Tâches Pondérées sur des Machines Parallèles

Par

Abderazak HSAINI

Métaheuristiques
Science de la Décision et Recherche Opérationnelle
Institut National de Statistique et d'Économie Appliquée, MA

Novembre 2024

Encadré par M. Oualid BEN BRIK

RÉSUMÉ

Ce rapport présente une étude approfondie de l'ordonnancement des tâches sur des machines parallèles identiques, en minimisant la somme pondérée des temps de fin et le *makespan*. Nous avons formulé un modèle mathématique du problème et exploré plusieurs approches de résolution :

- Une **méthode exacte** implémentée avec MiniZinc, garantissant des solutions optimales mais avec un temps d'exécution élevé pour les grandes instances.
- Une **méthode heuristique** basée sur un ordonnancement pondéré par liste, offrant des solutions rapides mais potentiellement éloignées de l'optimum.
- Une **métaheuristique de recherche tabou**, combinant efficacité et qualité des solutions grâce à des mécanismes de diversification et d'aspiration.

Les résultats expérimentaux montrent que la méthode exacte est préférable pour les petites instances, tandis que les heuristiques et métaheuristiques sont mieux adaptées aux grandes instances où le compromis entre qualité et temps d'exécution est essentiel. Ce travail souligne l'importance d'une approche flexible pour résoudre les problèmes d'ordonnancement, et ouvre la voie à des améliorations futures, notamment à travers l'intégration d'autres techniques de métaheuristiques ou d'algorithmes hybrides.

Table des matières

Chapitre 1

Introduction

1.1 Explication du Problème

L'ordonnancement des tâches pondérées sur des machines parallèles est un problème d'optimisation combinatoire majeur ayant de nombreuses applications pratiques dans divers domaines, tels que la production industrielle, l'informatique répartie et les systèmes de gestion de projets. L'objectif est d'attribuer un ensemble de tâches pondérées à plusieurs machines parallèles de manière à optimiser un critère de performance global, tel que la minimisation du temps d'achèvement pondéré total (makespan pondéré). Chaque tâche possède une pondération qui reflète son importance ou sa priorité dans le processus d'ordonnancement.

1.2 Contexte du Problème

Dans le contexte actuel des systèmes parallèles, le besoin de solutions efficaces à l'ordonnancement des tâches devient crucial pour l'amélioration des performances, la réduction des coûts et l'optimisation des ressources disponibles. Les machines parallèles se retrouvent dans des environnements de production complexes, les centres de données et même dans le calcul haute performance. L'ordonnancement optimal ou presque optimal est essentiel pour garantir une utilisation efficace des ressources et satisfaire les contraintes spécifiques des systèmes.

1.3 Revue de Littérature

Plusieurs approches ont été proposées pour traiter le problème d'ordonnancement des tâches pondérées, incluant des méthodes exactes telles que la programmation linéaire et des méthodes d'approximation basées sur des heuristiques et des métaheuristiques. Les méthodes exactes garantissent une solution optimale mais sont souvent limitées par leur complexité temporelle lorsqu'elles s'appliquent à des instances de grande taille. À l'inverse, les heuristiques et métaheuristiques, telles que le *list scheduling* pondéré, la recherche tabou et les algorithmes génétiques, permettent d'obtenir rapidement des solutions proches de l'optimal dans un délai raisonnable. Toutefois, leur performance dépend souvent du choix des paramètres et de l'initialisation.

Chapitre 2

Le Modèle Mathématique

2.1 Définition du Problème

Un système d'ordonnancement de machines parallèles typique peut être représenté comme illustré dans la Figure 2.1. Ce système comprend m machines et n opérations. Chaque machine est représentée par M_y où $y \in Y$, avec $Y = \{1, 2, \ldots, m\}$. Chaque opération O_j (où $j \in N$, avec $N = \{1, 2, \ldots, n\}$) a un temps de traitement t_j et un poids w_j. Le problème d'ordonnancement consiste à assigner ces opérations aux machines de manière à minimiser la somme pondérée des temps de fin. Dans un ordonnancement réalisable, F_j représente le temps de fin de l'opération j, et F (ou C) désigne la durée totale de la planification (makespan). Nous utilisons de manière interchangeable F_j et C_j, ainsi que F et C. Le planning est non-préemptif, ce qui signifie que chaque machine ne peut traiter qu'une seule opération à la fois. La fonction objectif est de minimiser la somme du temps de fin pondéré de chaque opération, ainsi que le makespan, c'est-à-dire :

$$\Gamma = \sum_{j=1}^{n} w_j F_j + F,$$

où w_j est un nombre non négatif.

2.2 Paramètres du Problème

Les paramètres du problème comprennent :

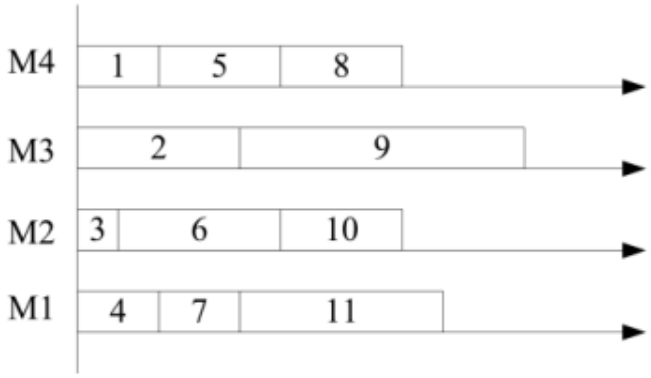

Figure 2.1: Le système d'ordonnancement des machines parallèles

- N: Ensemble des indices pour les n opérations, $N = \{1, 2, \ldots, n\}$.
- Y: Ensemble des indices pour les m machines, $Y = \{1, 2, \ldots, m\}$.
- t_j: Temps de traitement de l'opération j.
- B_j: Ensemble contenant tous les indices des opérations inférieures à j, $B_j = \{1, 2, \ldots, j-1\}$.
- w_j: Poids de l'opération j.

2.3 Variables de Décision

Les variables de décision du problème comprennent :

- F_j: Temps de fin de l'opération j.
- F: Makespan ou temps de réalisation maximal du planning des machines parallèles.

- x_{iy}: Variable binaire indiquant si l'opération i est affectée à la machine y,

$$x_{iy} = \begin{cases} 1 & \text{si l'opération } i \text{ est affectée à la machine } y, \\ 0 & \text{sinon.} \end{cases} \quad \forall i \in N, \forall y \in Y$$

- a_{ijy}: Variable binaire indiquant si les opérations i et j sont affectées à la machine y,

$$a_{ijy} = \begin{cases} 1 & \text{si les opérations } i \text{ et } j \text{ sont affectées à la machine } y, \\ 0 & \text{sinon.} \end{cases} \quad \forall i \in B_j, \forall j \in N, \forall y \in Y$$

2.4 Modèle Mathématique

L'objectif est de minimiser :

$$\text{Minimiser} \quad \sum_{j=1}^{n} w_j F_j + F$$

Sous les contraintes suivantes :

$$a_{ijy} - x_{iy} - x_{jy} \geq -1, \quad \forall i \in B_j, \forall j \in N, y \in Y \tag{2.1}$$

$$-a_{ijy} + x_{iy} \geq 0, \quad \forall i \in B_j, \forall j \in N, y \in Y \tag{2.2}$$

$$-a_{ijy} + x_{jy} \geq 0, \quad \forall i \in B_j, \forall j \in N, y \in Y \tag{2.3}$$

$$\sum_{y \in Y} x_{iy} = 1, \quad \forall i \in N \tag{2.4}$$

$$F_j - \sum_{i \in B_j} \sum_{y \in Y} t_i a_{ijy} - t_j = 0, \quad \forall j \in N \tag{2.5}$$

$$F \geq F_j, \quad \forall j \in N \tag{2.6}$$

$$F_i \geq 0, \quad \forall i \in N \tag{2.7}$$

$$a_{ijy} \in \{0, 1\}, \quad \forall j \in N, i \in B_j, y \in Y \tag{2.8}$$

$$x_{iy} \in \{0, 1\}, \quad \forall i \in N, y \in Y \tag{2.9}$$

Chapitre 3

Implémentation du Modèle sur MiniZinc

Dans ce chapitre, nous détaillons l'implémentation de notre modèle d'ordonnancement des tâches pondérées sur des machines parallèles à l'aide du langage de modélisation MiniZinc. L'objectif de cette implémentation est de résoudre le problème en minimisant la somme pondérée des temps de fin des opérations ainsi que le makespan (temps de réalisation maximal).

3.1 Description du Code

Le modèle MiniZinc utilisé pour l'implémentation est le suivant :

```
include "globals.mzn";
include "data.dzn";

int: m;                          % Nombre de machines
int: n;                          % Nombre d'op rations
set of int: Machines = 1..m;     % Ensemble des machines
set of int: Operations = 1..n;   % Ensemble des op rations

array[Operations] of int: t;     % Temps de traitement pour chaque
    op ration
array[Operations] of int: w;     % Poids de chaque op ration

% Variables de d cision
```

```
13 array[Operations, Machines] of var bool: x;  % x[i, y] = 1 si l'op ration
       i est affect e  la machine y, 0 sinon
14 array[Operations, Operations, Machines] of var bool: a; % a[i, j, y] = 1 si
       i et j sont affect es  la m me machine y, 0 sinon
15
16 array[Operations] of var int: F;             % Temps de fin de chaque
       op ration
17 var int: C;                                  % Makespan (temps de fin
       maximal de toutes les op rations)
18
19 % Fonction objectif : minimiser la somme pond r e des temps de fin et du
       makespan
20 var int: objective = sum(j in Operations)(w[j] * F[j]) + C;
21 solve minimize objective;
22
23 % Contraintes
24 constraint forall(i in Operations) (
25     sum(y in Machines) (x[i, y]) = 1            % (4) Chaque op ration
       est affect e  exactement une machine
26 );
27
28 constraint forall(i in Operations, j in Operations where i < j, y in
       Machines) (
29     a[i, j, y] - x[i, y] - x[j, y] >= -1        % (1)
30     /\ -a[i, j, y] + x[i, y] >= 0               % (2)
31     /\ -a[i, j, y] + x[j, y] >= 0               % (3)
32 );
33
34 constraint forall(j in Operations) (
35     F[j] = sum(i in 1..j-1, y in Machines) (t[i] * a[i, j, y]) + t[j]    %
       (5) Calcul du temps de fin de chaque op ration
36 );
37
```

```
38 constraint forall(j in Operations) (
39     C >= F[j]                                    % (6) Makespan est
       sup rieur ou  gal  au temps de fin de chaque op ration
40 );
41
42 constraint forall(i in Operations) (
43     F[i] >= 0                                    % (7) Temps de fin de
       chaque op ration est non n gatif
44 );
45
46 % Contraintes d'int grit  pour les variables binaires
47 constraint forall(i in Operations, j in Operations where i < j, y in
       Machines) (
48     a[i, j, y] in 0..1                           % (8) a[i, j, y] est
       binaire
49 );
50
51 constraint forall(i in Operations, y in Machines) (
52     x[i, y] in 0..1                              % (9) x[i, y] est binaire
53 );
```

Listing 3.1: Implémentation de l'heuristique en Python

3.2 Explication du Modèle

3.2.1 Inclusions et Paramètres

Le modèle commence par inclure les bibliothèques nécessaires :

- `globals.mzn` : Cette bibliothèque fournit des contraintes globales utiles pour l'optimisation.

- `data.dzn` : Ce fichier contient les données d'entrée, telles que le nombre de machines et d'opérations, les temps de traitement et les poids.

Les paramètres du problème incluent :

- m : Nombre de machines.
- n : Nombre d'opérations.
- `Machines` et `Operations` : Ensembles représentant les indices des machines et des opérations respectivement.
- `t` : Tableau contenant les temps de traitement des opérations.
- `w` : Tableau contenant les poids associés à chaque opération.

3.2.2 Variables de Décision

Les principales variables de décision du modèle sont :

- `x[i, y]` : Variable binaire indiquant si l'opération i est affectée à la machine y.
- `a[i, j, y]` : Variable binaire indiquant si les opérations i et j sont affectées à la même machine y.
- `F[j]` : Temps de fin de l'opération j.
- `C` : Makespan, ou temps de fin maximal parmi toutes les opérations.

3.2.3 Fonction Objectif

La fonction objectif à minimiser est définie par :

$$\texttt{objective} = \sum_{j \in \texttt{Operations}} (w[j] \cdot F[j]) + C$$

Cette fonction cherche à minimiser la somme pondérée des temps de fin des opérations ainsi que le makespan.

3.2.4 Contraintes du Modèle

Le modèle impose plusieurs contraintes pour garantir un ordonnancement valide :

- **Affectation des opérations** : Chaque opération doit être affectée à exactement une machine.

$$\sum_{y \in \texttt{Machines}} \texttt{x[i, y]} = 1, \quad \forall i \in \texttt{Operations}$$

- **Compatibilité des affectations** : Les contraintes `(1)`, `(2)` et `(3)` garantissent la cohérence des affectations des opérations sur les machines.

- **Calcul du temps de fin** : Le temps de fin de chaque opération est calculé en tenant compte de l'ordre et des temps de traitement.

$$\texttt{F[j]} = \sum_{i \in 1..j-1, y \in \texttt{Machines}} (\texttt{t[i]} \cdot \texttt{a[i, j, y]}) + \texttt{t[j]}, \quad \forall j \in \texttt{Operations}$$

- **Makespan** : Le makespan doit être supérieur ou égal à chaque temps de fin.

$$\texttt{C} \geq \texttt{F[j]}, \quad \forall j \in \texttt{Operations}$$

- **Non-négativité et binarité** : Les temps de fin doivent être non négatifs, et les variables `x` et `a` sont binaires.

3.3 Exécution et Résultats

L'implémentation est testée sur des petites et grandes instances de données à partir du fichier `data.dzn`. MiniZinc fournit des outils pour visualiser les résultats, ce qui permet d'observer la répartition des opérations sur les machines et de comparer les temps de fin et le makespan obtenus.

—

Ce chapitre décrit en détail le modèle d'implémentation en MiniZinc, les contraintes du modèle et les objectifs poursuivis, offrant une base solide pour des analyses et des comparaisons ultérieures.

Chapitre 4

Résolution par Heuristique

Ce chapitre présente l'implémentation et l'application d'une approche heuristique pour résoudre le problème d'ordonnancement des tâches pondérées sur des machines parallèles. L'heuristique choisie est l'*ordonnancement par liste pondérée* (*Weighted List Scheduling*). Cette méthode trie les tâches en fonction d'un ratio de priorité (poids/temps de traitement) afin de minimiser la somme pondérée des temps de fin, tout en réduisant le makespan.

4.1 Description de l'Heuristique Utilisée

L'algorithme d'ordonnancement par liste pondérée repose sur les étapes suivantes :

1. Calcul du ratio de priorité pour chaque tâche, défini comme le ratio entre le poids et le temps de traitement de la tâche.
2. Tri des tâches par ordre décroissant de leur ratio de priorité.
3. Affectation de chaque tâche à la machine disponible ayant le temps de fin minimal.
4. Mise à jour du temps de fin de la machine après l'achèvement de la tâche.
5. Calcul du temps de fin de chaque tâche, de la somme pondérée des temps de fin et du makespan (le temps de fin maximal des machines).

L'implémentation de l'heuristique est réalisée en Python comme suit :

```
import heapq

# Donn es d'entr e du probl me
n = 20  # Nombre d'op rations (t ches)
m = 10  # Nombre de machines
processing_times = [6, 10, 3, 8, 7, 5, 12, 4, 9, 11, 13, 2, 15, 8, 6, 7, 14,
    3, 9, 10]  # Temps de traitement de chaque op ration
weights = [4, 2, 6, 1, 5, 3, 4, 6, 2, 3, 7, 5, 1, 8, 6, 3, 4, 5, 2, 7]  #
    Poids de chaque op ration

# Calcul du ratio poids/temps de traitement pour trier les op rations par
    ordre de priorit
tasks = sorted([(i, processing_times[i], weights[i], weights[i] /
    processing_times[i]) for i in range(n)],
               key=lambda x: x[3], reverse=True)

# Liste pour enregistrer les temps de fin des machines (initialement toutes
    0)
machine_times = [0] * m  # Temps de fin de chaque machine

# Variables pour stocker le temps de fin de chaque op ration et la valeur
    du makespan
completion_times = [0] * n  # Temps de fin de chaque op ration
total_weighted_completion_time = 0  # Somme pond r e des temps de fin

# Appliquer l'algorithme de List Scheduling Pond r
for task in tasks:
    i, processing_time, weight, priority = task

    # Trouver la machine avec le temps de fin minimal
    min_machine = min(range(m), key=lambda x: machine_times[x])

    # Affecter l'op ration    cette machine
```

```
28      start_time = machine_times[min_machine]
29      end_time = start_time + processing_time
30      machine_times[min_machine] = end_time
31
32      # Mettre    jour le temps de fin de l'op ration et la somme pond r e
        des temps de fin
33      completion_times[i] = end_time
34      total_weighted_completion_time += weight * end_time
35
36  # Calcul du makespan (temps de fin maximal des machines)
37  makespan = max(machine_times)
38
39  # Affichage des r sultats
40  print("Temps de fin de chaque op ration :", completion_times)
41  print("Somme pond r e des temps de fin :", total_weighted_completion_time)
42  print("Makespan :", makespan)
```

Listing 4.1: Implémentation de l'heuristique en Python

4.2 Explication du Code

L'algorithme commence par initialiser les paramètres du problème, notamment le nombre de machines (m), le nombre d'opérations (n), les temps de traitement et les poids associés à chaque opération.

4.2.1 Calcul du Ratio Poids/Temps de Traitement

Pour chaque opération, le ratio de priorité est calculé comme suit :

$$\text{Ratio} = \frac{\text{Poids de l'opération}}{\text{Temps de traitement de l'opération}}$$

Ce ratio permet de classer les tâches de manière à privilégier celles ayant un impact élevé sur la fonction objectif.

4.2.2 Affectation des Tâches aux Machines

Les tâches sont ensuite triées par ordre décroissant de leur ratio. Chaque tâche est affectée à la machine ayant le temps de fin minimal, afin de minimiser l'accumulation du temps de traitement global. Ce choix permet de réduire la somme pondérée des temps de fin tout en optimisant le makespan.

4.3 Résultats et Analyse

Les résultats obtenus par l'heuristique incluent :

- **Temps de fin de chaque opération** : Le temps de fin pour chaque opération est stocké dans un tableau `completion_times`.
- **Somme pondérée des temps de fin** : Cette valeur est calculée à l'aide de la formule :

$$\sum_{j=1}^{n} w_j \cdot F_j$$

où w_j et F_j représentent respectivement le poids et le temps de fin de l'opération j.

- **Makespan** : Le makespan correspond au temps de fin maximal parmi toutes les machines, calculé par :

$$\text{Makespan} = \max(\texttt{machine\_times})$$

L'heuristique d'ordonnancement par liste pondérée offre une solution rapide et efficace

pour des problèmes de taille importante, tout en fournissant une bonne approximation du coût total d'ordonnancement et du makespan.

Chapitre 5

Résolution par Métaheuristique

Dans ce chapitre, nous présentons la résolution du problème d'ordonnancement par une approche métaheuristique basée sur la *recherche tabou*. La recherche tabou est une méthode itérative qui utilise une stratégie d'exploration des voisins d'une solution courante tout en évitant de revenir sur des solutions déjà explorées, grâce à l'utilisation d'une liste tabou.

5.1 Présentation de l'Algorithme de Recherche Tabou

La recherche tabou est une métaheuristique de recherche locale qui cherche à améliorer une solution initiale par l'exploration d'un voisinage défini. Elle utilise une liste tabou pour empêcher l'algorithme de revenir à des solutions déjà visitées, ce qui lui permet d'échapper aux optima locaux.

5.2 Implémentation

L'algorithme est implémenté en Python et suit les étapes suivantes :

1. Génération d'une solution initiale à l'aide d'une heuristique d'ordonnancement par liste pondérée.
2. Calcul de la fonction objectif, qui consiste à minimiser le *makespan* et la somme

pondérée des temps de fin.

3. Génération des voisins de la solution courante.

4. Application des critères de sélection, d'aspiration, et mise à jour de la liste tabou.

5. Mise à jour des meilleures solutions trouvées.

```python
import heapq
import random
import copy
from collections import deque

# Param tres du probl me
n = 20  # Nombre d'op rations
m = 10  # Nombre de machines
processing_times = [6, 10, 3, 8, 7, 5, 12, 4, 9, 11, 13, 2, 15, 8, 6, 7, 14, 3, 9, 10]  # Temps de traitement
weights = [4, 2, 6, 1, 5, 3, 4, 6, 2, 3, 7, 5, 1, 8, 6, 3, 4, 5, 2, 7]  # Poids

# Param tres de la recherche tabou
tabu_tenure = 5               # Dur e de la liste tabou
max_iter = 100                # Nombre d'it rations maximum
no_improvement_limit = 20     # Limite de non-am lioration
num_neighbors = min(10, n)    # Nombre de voisins explor s

# 1. Solution Initiale avec l'Heuristique List Scheduling Pond r
def initial_solution():
    tasks = sorted([(i, processing_times[i], weights[i], weights[i] / processing_times[i]) for i in range(n)],
                   key=lambda x: x[3], reverse=True)
    machine_times = [0] * m
    assignment = [[] for _ in range(m)]
    completion_times = [0] * n
```

```
25
26      for task in tasks:
27          i, processing_time, weight, priority = task
28          min_machine = min(range(m), key=lambda x: machine_times[x])
29          start_time = machine_times[min_machine]
30          end_time = start_time + processing_time
31          machine_times[min_machine] = end_time
32          assignment[min_machine].append(i)
33          completion_times[i] = end_time
34
35      makespan = max(machine_times)
36      weighted_completion = sum(weights[i] * completion_times[i] for i in
    range(n))
37      return assignment, makespan, weighted_completion
38
39  # Calcul de l'objectif : Makespan et Somme Pond r e des Temps de Fin
40  def calculate_objective(assignment):
41      machine_times = [0] * m
42      completion_times = [0] * n
43
44      for y in range(m):
45          for i in assignment[y]:
46              completion_times[i] = machine_times[y] + processing_times[i]
47              machine_times[y] += processing_times[i]
48
49      makespan = max(machine_times)
50      weighted_completion = sum(weights[i] * completion_times[i] for i in
    range(n))
51      return makespan, weighted_completion
52
53  # G n ration des voisins
54  def generate_neighbors(assignment):
55      neighbors = []
```

```
    for y1 in range(m):
        for y2 in range(m):
            if y1 != y2 and assignment[y1]:  # Transf rer une t che de y1 y2
                for i in range(len(assignment[y1])):
                    new_assignment = copy.deepcopy(assignment)
                    task = new_assignment[y1].pop(i)
                    new_assignment[y2].append(task)
                    neighbors.append(new_assignment)
    return neighbors

# 2. M taheuristique de Recherche Tabou
def tabu_search():
    current_assignment, current_makespan, current_weighted_completion = initial_solution()
    best_assignment = current_assignment
    best_makespan = current_makespan
    best_weighted_completion = current_weighted_completion

    tabu_list = deque(maxlen=tabu_tenure)  # Liste Tabou avec taille limit e
    iter_no_improve = 0

    for iteration in range(max_iter):
        if iter_no_improve >= no_improvement_limit:
            break  # Arr t si pas d'am lioration

        # G n ration et Filtrage des Voisins
        neighbors = generate_neighbors(current_assignment)
        neighbors = [neighbor for neighbor in neighbors if neighbor not in tabu_list]

        best_neighbor = None
```

```
85          best_neighbor_makespan = float('inf')
86          best_neighbor_weighted_completion = float('inf')
87
88          # 3. Crit re d'Aspiration et S lection du Meilleur Voisin
89          for neighbor in neighbors:
90              makespan, weighted_completion = calculate_objective(neighbor)
91              if (weighted_completion < best_neighbor_weighted_completion or
92                  (weighted_completion == best_neighbor_weighted_completion and
    makespan < best_neighbor_makespan)):
93                  best_neighbor = neighbor
94                  best_neighbor_makespan = makespan
95                  best_neighbor_weighted_completion = weighted_completion
96
97          if best_neighbor is None:
98              break  # Aucun voisin valide trouv
99
100         # Mise    jour de la solution actuelle
101         current_assignment = best_neighbor
102         current_makespan = best_neighbor_makespan
103         current_weighted_completion = best_neighbor_weighted_completion
104
105         # 4. Mise    jour de la Liste Tabou
106         tabu_list.append(best_neighbor)
107
108         # 5. Mise    jour des Meilleures Solutions
109         if (current_weighted_completion < best_weighted_completion or
110             (current_weighted_completion == best_weighted_completion and
    current_makespan < best_makespan)):
111             best_assignment = current_assignment
112             best_makespan = current_makespan
113             best_weighted_completion = current_weighted_completion
114             iter_no_improve = 0  # R initialisation en cas d'am lioration
115         else:
```

```
116             iter_no_improve += 1  # Incr mentation en cas de non-
    am lioration
117
118     return best_assignment, best_makespan, best_weighted_completion
119
120 # Ex cution de la Recherche Tabou
121 best_assignment, best_makespan, best_weighted_completion = tabu_search()
122 print("Meilleur assignement des op rations:", best_assignment)
123 print("Makespan optimal trouv :", best_makespan)
124 print("Somme pond r e des temps de fin optimale:",
    best_weighted_completion)
```

Listing 5.1: Implémentation de l'heuristique en Python

5.3 Description de l'Algorithme

L'algorithme de recherche tabou suit les étapes suivantes :

- **Initialisation :** Une solution initiale est générée à l'aide d'une heuristique d'ordonnancement par liste pondérée.
- **Évaluation :** La fonction objectif calcule le *makespan* et la somme pondérée des temps de fin.
- **Génération des voisins :** Les voisins sont créés en transférant une tâche d'une machine à une autre.
- **Filtrage Tabou :** Les solutions déjà visitées sont exclues à l'aide de la liste tabou.
- **Critère d'aspiration :** Le meilleur voisin est sélectionné s'il améliore la solution courante, même s'il est tabou.

- **Mise à jour :** La meilleure solution et la liste tabou sont mises à jour à chaque itération.

5.4 Résultats et Analyse

Les résultats obtenus par la recherche tabou incluent :

- **Assignement des opérations :** Une affectation optimale ou proche de l'optimal pour chaque tâche sur les machines.
- **Makespan optimal :** Temps de fin maximal trouvé parmi les machines.
- **Somme pondérée des temps de fin :** Valeur minimale atteinte de la somme pondérée des temps de fin.

La recherche tabou offre une approche puissante pour échapper aux optima locaux et obtenir des solutions de qualité pour des problèmes complexes d'ordonnancement.

Chapitre 6

Comparaison des Méthodes

Dans ce chapitre, nous comparons les performances des méthodes exactes, heuristiques et métaheuristiques présentées précédemment pour résoudre le problème d'ordonnancement des tâches. Cette comparaison est effectuée en utilisant deux critères principaux :

- **Qualité de la solution :** Mesurée par le *makespan* et la somme pondérée des temps de fin (valeur optimale atteinte).
- **Temps d'exécution :** Temps nécessaire pour trouver la solution.

Nous comparons les performances sur plusieurs instances de tailles différentes. Les résultats sont présentés dans les tableaux suivants.

6.1 Qualité des Solutions

Le Tableau 6.1 présente les valeurs optimales (makespan et somme pondérée des temps de fin) obtenues par les trois méthodes pour différentes instances.

Les résultats montrent que la méthode exacte offre les meilleures solutions en termes de qualité (valeurs optimales) mais au prix d'un temps d'exécution potentiellement élevé. Les méthodes heuristiques et métaheuristiques, quant à elles, offrent des solutions proches de l'optimal, bien que parfois légèrement inférieures.

6.2 Temps d'Exécution

Le Tableau 6.2 présente les temps d'exécution (en secondes) des trois méthodes pour les mêmes instances.

Les résultats montrent que les méthodes heuristiques et métaheuristiques sont beaucoup plus rapides que la méthode exacte. La méthode exacte devient impraticable pour des instances de grande taille en raison de l'explosion combinatoire du temps de résolution. Les méthodes heuristiques et métaheuristiques offrent un compromis intéressant en termes de temps d'exécution, tout en produisant des solutions de bonne qualité.

6.3 Analyse Globale

De manière générale, la méthode exacte garantit l'obtention de la solution optimale mais au prix d'un temps de calcul très élevé, notamment pour les instances de grande taille. Les méthodes heuristiques et métaheuristiques, quant à elles, permettent de trouver des solutions de bonne qualité en un temps réduit, ce qui les rend particulièrement adaptées pour les problèmes à grande échelle. La recherche tabou (métaheuristique) parvient souvent à trouver de meilleures solutions que les méthodes heuristiques simples, tout en maintenant un temps d'exécution raisonnable.

En conclusion, le choix de la méthode dépendra du compromis recherché entre qualité de la solution et temps d'exécution.

Table 6.1: Comparaison des valeurs optimales obtenues par les méthodes exacte, heuristique et métaheuristique

Instance	**Méthode exacte**	**Méthode heuristique**	**Métaheuristique**
Petite	9	9	9
Grande	72	25	23

Table 6.2: Comparaison des temps d'exécution des méthodes exacte, heuristique et métaheuristique

Instance	**Méthode exacte**	**Méthode heuristique**	**Métaheuristique**
Petite	0.274s	0.112s	0.144s
Grande	6m 45s	0.122s	0.197s

Chapitre 7

Conclusion

Ce rapport a abordé le problème d'ordonnancement des opérations sur des machines parallèles identiques en se concentrant sur l'optimisation de la somme pondérée des temps de fin ainsi que du *makespan*. Nous avons proposé plusieurs approches pour résoudre ce problème, allant de la méthode exacte à des méthodes d'approximation telles que les heuristiques et métaheuristiques.

Dans un premier temps, nous avons formulé un modèle mathématique rigoureux pour le problème et avons implémenté une méthode exacte utilisant le langage de modélisation MiniZinc. Cette méthode garantit une solution optimale, mais son temps d'exécution augmente de manière exponentielle avec la taille des instances du problème, ce qui la rend impraticable pour les cas de grande envergure.

Pour pallier cette limitation, nous avons développé une approche heuristique basée sur un ordonnancement pondéré par liste. Cette méthode offre une solution rapide, même pour des instances de grande taille, mais la qualité des solutions obtenues peut s'éloigner de l'optimal.

Afin d'améliorer la qualité des solutions tout en conservant un temps d'exécution raisonnable, nous avons ensuite exploré une métaheuristique basée sur la recherche tabou. Cette méthode a montré son efficacité en trouvant des solutions proches de l'optimum avec un coût computationnel bien inférieur à celui de la méthode exacte. La recherche tabou exploite intelligemment les solutions voisines et utilise des mécanismes de diversification pour éviter

le piégeage dans des optima locaux.

Les résultats expérimentaux présentés dans ce rapport montrent clairement l'importance de choisir la bonne méthode en fonction des exigences spécifiques du problème. Si l'optimalité stricte est requise et que le temps de calcul n'est pas une contrainte, la méthode exacte est la meilleure option. En revanche, pour des problèmes de grande taille nécessitant des solutions de bonne qualité en temps limité, les méthodes heuristiques et métaheuristiques offrent un compromis satisfaisant.