CentraleSupélec

aneo

Optimisation de l'ordonnancement d'un graphe de tâches

ST7 - Simulation a haute performance

Réalisé par

Darfaf Yasmine, Knecht Axel, Vallet Alexandre, Wehbe Philémon

Encadré par

Rimmel Arpad, Kirschenmann Wilfried, Delamea Quentin

21 octobre 2025

Table des matières

Abstract

This project investigates the HLFET (Highest Level First with Estimated Times) heuristic for task scheduling. Inefficient task scheduling can lead to significant overheads in both time and cost, particularly in cloud-based infrastructures for high-performance computing.

The objective is to assess the performance of HLFET as a superior alternative to a greedy algorithm, while remaining computationally less expensive than exact algorithms, whose implementation becomes infeasible for large task graphs due to their exponential time complexity.

Following a theoretical study of the algorithm and a formalisation of the scheduling problem, we implement the method on a local machine before deploying it on AWS Lambda to evaluate its scalability and efficiency in a cloud environment. Experiments show that HLFET consistently outperforms the greedy baseline in terms of makespan, while maintaining near-linear runtime scalability on sparse graphs.

Résumé

Ce projet étudie l'heuristique HLFET (Highest Level First with Estimated Times) appliquée à l'ordonnancement de tâches. Un ordonnancement inefficace peut entraîner des surcoûts importants, tant en temps qu'en ressources, notamment dans des infrastructures cloud dédiées au calcul haute performance.

L'objectif est d'évaluer les performances de HLFET en tant qu'alternative pertinente à un algorithme glouton, tout en restant moins coûteuse qu'une méthode exacte, dont l'implémentation devient inenvisageable pour des graphes de tâches de grande taille en raison de leur complexité temporelle exponentielle.

Après une étude théorique de l'algorithme et une formalisation du problème d'ordonnancement, la solution est d'abord implémentée localement, puis déployée sur AWS Lambda afin d'en évaluer la scalabilité et l'efficacité en environnement cloud. Les résultats expérimentaux montrent que HLFET réduit systématiquement le makespan par rapport à l'approche gloutonne, tout en conservant une scalabilité quasi linéaire sur des graphes peu denses.

Chapitre 1

Introduction

1.1 Contexte

L'optimisation de l'ordonnancement des tâches est un enjeu central en calcul haute performance (HPC), où l'exécution efficace de tâches interdépendantes est essentielle pour maximiser l'utilisation des ressources et minimiser le temps total d'exécution. Face à des infrastructures complexes, variables et des contraintes multiples (capacité des nœuds, topologie du réseau, consommation énergétique, délais d'exécution), une planification stratégique est nécessaire pour éviter les goulets d'étranglement, améliorer la performance globale du système et limiter les pertes en temps et en coûts opérationnels.

Dans ce contexte, l'ordonnancement repose sur l'analyse d'un graphe de dépendances entre tâches, où chaque tâche est caractérisée par une durée d'exécution et des prérequis définissant son ordre de traitement. L'infrastructure cible, souvent distribuée sur le cloud, impose une contrainte supplémentaire en nécessitant une allocation intelligente des ressources. L'objectif est d'optimiser la planification afin de minimiser le *makespan* (temps total d'exécution du graphe) tout en prenant en compte la variabilité des performances des nœuds et les coûts liés à l'utilisation des infrastructures.

1.2 Objectifs

Dans le cadre de ce projet, notre objectif est d'étudier un algorithme heuristique visant à optimiser le temps d'exécution d'un graphe tout en maintenant un temps de calcul raisonnable pour l'algorithme lui-même. En complément, nous visons à déployer cette solution sur le cloud et à l'évaluer sur des graphes de grande taille, pouvant atteindre un million de tâches.

1.3 Plan

Après un état de l'art général sur les différentes techniques d'optimisation de l'ordonnancement des graphes de tâches, nous nous concentrerons sur un cas particulier dont les hypothèses et la formalisation mathématique seront définies dans les chapitres suivants. Une fois ce cadre établi, nous réaliserons une étude théorique et numérique de l'algorithme HLFET hors cloud avant de déployer notre solution sur le cloud.

1.4 Contribution

Notre contribution à ce projet s'articule au tour de plusieurs axes. Tout d'abord, nous analysons les enjeux globaux liés à l'ordonnancement des tâches ainsi que les avancées réalisées dans l'optimisation de ce processus. En nous appuyant sur le cours d'optimisation, nous proposons une formalisation du problème, bien que la littérature à ce sujet soit déjà riche et approfondie.

Ensuite, nous étudions le comportement de l'algorithme HLFET en cherchant à en déduire des résultats théoriques par l'expérimentation et en le comparant à d'autres approches existantes.

Enfin, après une phase de tests hors cloud, nous déployons notre solution sur le cloud afin d'évaluer ses performances sur des graphes de très grande taille.

Chapitre 2

Etat de l'art

2.1 Définition et enjeux de l'ordonnancement des tâches

L'ordonnancement des tâches est un problème fondamental dans l'optimisation des systèmes informatiques et industriels. Il consiste à organiser l'exécution d'un ensemble de tâches sur des ressources limitées afin de minimiser des critères tels que le *makespan* (durée totale d'exécution), l'utilisation des ressources ou encore le respect des délais.

Dans un environnement de calcul haute performance (HPC), l'ordonnancement joue un rôle critique en permettant une répartition efficace des charges de travail. De même, en *supply chain*, il optimise la production et la distribution de ressources [5].

2.2 Modèles d'ordonnancement et typologies d'algorithmes

L'ordonnancement peut être classé en plusieurs catégories :

— **Modèles déterministes** : La durée d'exécution des tâches est connue à l'avance [3].

— **Modèles stochastiques** : La durée des tâches est incertaine et suit une distribution de probabilité.

L'optimisation des stratégies d'ordonnancement repose souvent sur des algorithmes heuristiques et métaheuristiques :

— **Heuristiques** : Min-Min, Max-Min, HEFT (*Heterogeneous Earliest Finish Time*), HLFET [3].

— **Métaheuristiques** : Recuit simulé, PSO (*Particle Swarm Optimization*), ACO (*Ant Colony Optimization*), GA (*Genetic Algorithm*) [1][2] .

Les résultats expérimentaux montrent que les approches hybrides combinant plusieurs stratégies améliorent l'efficacité du processus d'ordonnancement [2].

2.3 Simulation et évaluation des performances des algorithmes d'ordonnancement

L'évaluation des algorithmes d'ordonnancement repose sur des simulations permettant d'analyser leur comportement sur des scénarios réalistes. Plusieurs simulateurs ont été développés à cet effet :

— **PYSS** : Simulateur de files d'attente basé sur des traces statiques.

— **Performance Prediction Toolkit (PPT)** : Simulateur basé sur des événements discrets, intégrant des modèles de communication et d'utilisation des ressources [4].

Ces outils permettent de comparer des stratégies d'ordonnancement courantes comme :

— *First Come First Serve* (FCFS)
— *Shortest Job First* (SJF)
— *Score-Based Priority Scheduling*

L'article [4] met en évidence l'impact du placement des tâches et l'intérêt de modèles détaillés pour prédire les performances des algorithmes.

2.4 Perspectives et tendances récentes

Les avancées récentes en intelligence artificielle ont conduit à l'intégration de l'apprentissage automatique dans les algorithmes d'ordonnancement. L'apprentissage par renforcement (*Reinforcement Learning*) est notamment exploré pour adapter dynamiquement les stratégies aux charges de travail évolutives [1].

Enfin, le déploiement d'algorithmes optimisés sur le cloud permet une meilleure gestion des ressources et une réduction des coûts opérationnels [5].

Chapitre 3

Formalisation du problème d'ordonnancement

L'optimisation de l'ordonnancement des tâches est un problème complexe qui peut être formulé sous différentes hypothèses et contraintes. Pour simplifier notre étude, nous adoptons un certain nombre d'hypothèses qui nous permettront de restreindre la complexité du problème et de proposer une approche adaptée.

3.1 Hypothèses du problème

Afin de garantir une modélisation réaliste tout en gardant une complexité raisonnable, nous considérons les hypothèses suivantes :

- **Ressources homogènes et fonctionnelles** : Toutes les ressources (machines ou cœurs) sont identiques en termes de capacités de calcul et ne sont jamais défaillantes.
- **Dépendances entre tâches fixes et connues à l'avance** : Le graphe des dépendances est déterminé avant l'ordonnancement.
- **Durée d'exécution des tâches connue** : La durée d'exécution de chaque tâche est supposée connue et fixe.
- **Temps de transition entre tâches négligé** : Aucun coût de commutation entre tâches n'est pris en compte.
- **Une tâche est exécutée sur un seul cœur** : Chaque tâche est affectée à une seule ressource et ne peut être divisée.

3.2 Graphe de tâches

L'ensemble des tâches et leurs dépendances forment un graphe orienté acyclique (**DAG** - Directed Acyclic Graph). Ce graphe est défini par :

$$G = (T, E) \tag{3.1}$$

où :

- $T = \{T_1, T_2, ..., T_n\}$ représente l'ensemble des tâches.
- $E = \{(T_i, T_j) \mid T_i \text{ terminée avant } T_j\}$ représente les arêtes de dépendance entre tâches.

3.2.1 Propriétés des graphes orientés acycliques

Un DAG est un graphe dirigé ne contenant aucun cycle. Cela signifie qu'il existe un ordre partiel entre les tâches, et qu'une tâche T_j ne peut commencer qu'après l'achèvement de toutes ses tâches précédentes T_i. Mathématiquement, cette contrainte est exprimée par :

$$S_j \geq C_i, \quad \forall (T_i, T_j) \in E \tag{3.2}$$

où S_j et C_i sont respectivement lestemps de début ettemps de fin des tâches.

3.2.2 Exemple d'un graphe dirigé acyclique

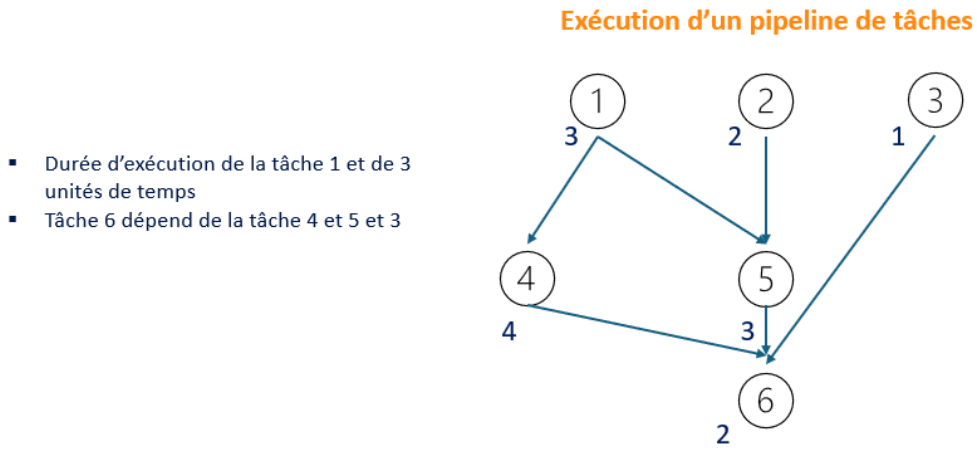

FIGURE 3.1 – Exemple d'un graphe dirigé acyclique

3.3 Notations

Nous introduisons les notations suivantes :
— p_i : durée d'exécution de la tâche T_i.
— $R = \{R_1, ..., R_m\}$: ensemble des m ressources disponibles (machines ou cœurs).
— S_i : instant de début d'exécution de la tâche T_i.
— $C_i = S_i + p_i$: instant de fin d'exécution de la tâche T_i.
— $X_{i,k}$: variable binaire indiquant si T_i est affectée à la ressource R_k :

$$X_{i,k} = \begin{cases} 1 & \text{si } T_i \text{ est affectée à } R_k \\ 0 & \text{sinon} \end{cases} \tag{3.3}$$

3.4 Formalisation sous forme d'un problème d'optimisation

L'objectif est de minimiser le makespan (temps d'achèvement de la dernière tâche) :

$$C_{\max} = \max_{i \in T} C_i \tag{3.4}$$

3.4.1 Contraintes du problème

3.4.1.1 Contraintes de précédence

Une tâche ne peut commencer qu'après la fin de toutes ses tâches précédentes :

$$S_j \geq C_i, \quad \forall (T_i, T_j) \in E \tag{3.5}$$

3.4.1.2 Contraintes d'affectation des tâches aux cœurs

Chaque tâche est exécutée surune seule ressource :

$$\sum_{k=1}^{m} X_{i,k} = 1, \quad \forall T_i \in T \tag{3.6}$$

3.4.1.3 Domaines des variables

$$S_i \geq 0, \quad X_{i,k} \in \{0, 1\} \tag{3.7}$$

3.4.1.4 Contraintes d'exclusivité des ressources

Deux tâches exécutées sur le même cœur ne peuvent pas se chevaucher :

$$S_j \geq C_i - M(1 - Z_{i,j,k}), \quad S_i \geq C_j - M Z_{i,j,k} \tag{3.8}$$

avec :

$$Z_{i,j,k} \geq X_{i,k} + X_{j,k} - 1, \quad Z_{i,j,k} \in \{0, 1\}, \quad \forall i, j, \forall k \tag{3.9}$$

3.5 Complexité du problème

Le problème étudié est NP-Difficile, car il peut être réduit au Job Shop Scheduling Problem. Ce dernier est un problème d'ordonnancement dans lequel un ensemble de tâches doit être exécuté sur un nombre limité de machines, chaque tâche devant respecter un ordre strict d'exécution ainsi que des contraintes de précédence. L'objectif reste identique à celui de notre étude, à savoir la minimisation de la durée totale d'exécution du graphe de tâches. Cependant, la nature du problème fait que celui-ci n'est pas résolvable de manière exacte en temps polynomiale.

Chapitre 4

L'heuristique d'ordonnancement HLFET

4.1 Les algorithmes d'ordonnancement

Dans cette section, nous revenons sur une partie de l'étude présentée dans l'état de l'art concernant les algorithmes d'ordonnancement et les différentes familles de solutions existantes, afin de justifier le choix de l'algorithme HLFET (Highest Level First with Estimated Times).

L'ordonnancement de tâches repose sur des stratégies visant à éviter les approches naïves, souvent coûteuses en termes de temps et de ressources. Pour optimiser l'affectation des tâches, on a recours à des algorithmes d'ordonnancement permettant d'améliorer l'efficacité du processus.

Un algorithme d'ordonnancement peut poursuivre plusieurs objectifs, tels que la minimisation du makespan ou la réduction du nombre de ressources utilisées.

Il existe plusieurs catégories d'algorithmes d'ordonnancement, chacune présentant ses avantages et ses limites.

4.1.1 Algorithmes exacts

Les algorithmes exacts garantissent la détermination d'un ordonnancement optimal en explorant exhaustivement l'espace de solutions. Les approches classiques incluent :

- **Programmation linéaire en nombres entiers (ILP)** : modélise le problème sous forme de contraintes et d'une fonction objectif à optimiser, résolue via des solveurs comme CPLEX ou Gurobi.
- **Branch-and-Bound** : explore l'arbre de recherche en éliminant certaines branches grâce à des bornes sur la fonction objectif.
- **Programmation dynamique** : décompose le problème en sous-problèmes pour éviter les recomputations inutiles.

En raison de leur complexité exponentielle, ces méthodes deviennent rapidement impraticables pour des graphes de grande taille, notamment dans des environnements massivement parallèles tels que le cloud, où l'ordonnancement peut concerner plusieurs millions de tâches. Leur utilisation est donc restreinte à des cas où l'optimalité est impérative ou lorsque la taille du problème demeure modérée.

4.1.2 Les métaheuristiques

Les métaheuristiques sont des algorithmes d'optimisation approchés visant à fournir des solutions de bonne qualité en un temps raisonnable lorsque les méthodes exactes sont inapplicables. Elles reposent sur des stratégies d'exploration et d'exploitation de l'espace de recherche afin d'éviter les minima locaux.

Parmi les approches courantes, on retrouve :

— **Recuit simulé (Simulated Annealing, SA)** : inspiré du refroidissement des métaux, il accepte temporairement des solutions sous-optimales pour échapper aux minima locaux.
— **Algorithmes génétiques (Genetic Algorithms, GA)** : basés sur la sélection naturelle, ils génèrent de nouvelles solutions par mutation et croisement.
— **Recherche tabou (Tabu Search, TS)** : exploite une mémoire adaptative pour éviter la stagnation en interdisant temporairement certains déplacements.

Bien qu'efficaces pour traiter des problèmes complexes, les métaheuristiques ne garantissent pas l'optimalité et leur performance dépend fortement du réglage des paramètres et de la nature du problème.

4.1.3 Les heuristiques

Dans le cadre de notre étude, nous nous intéressons spécifiquement aux heuristiques, qui offrent un compromis entre qualité de solution et temps de calcul. Contrairement aux algorithmes exacts, elles ne garantissent pas l'optimalité, mais permettent d'obtenir des solutions satisfaisantes en un temps réduit, tout en restant plus simples à implémenter.

Les heuristiques reposent sur des règles de décision basées sur des critères locaux, sans exploration exhaustive de l'espace de recherche. Parmi les approches courantes en ordonnancement, on retrouve :

— **Plus court d'abord (Shortest Processing Time, SPT)** : priorise les tâches ayant la plus faible durée d'exécution.
— **Premier arrivé, premier servi (First Come, First Served, FCFS)** : exécute les tâches dans l'ordre de leur arrivée.
— **Heuristique de liste (List Scheduling)** : attribue une priorité aux tâches et les affecte aux ressources disponibles en fonction de cette priorité.

Bien que rapides et faciles à mettre en œuvre, les heuristiques peuvent produire des solutions de qualité variable, fortement dépendantes de la structure du problème et des choix heuristiques adoptés.

4.2 L'algorithme HLFET

4.2.1 Principe

L'algorithme HLFET (Highest Level First with Estimated Times) appartient à la famille des heuristiques, et plus précisément aux heuristiques de listes. Il repose sur un ordonnancement prioritaire des tâches en fonction d'un critère appelé niveau, qui est défini dans le paragraphe suivant.

Son fonctionnement suit deux étapes principales :

1. **Ordonnancement des tâches** : les tâches sont triées par ordre décroissant de leur niveau, ce qui privilégie l'exécution des tâches considérées comme les plus critiques pour le respect des dépendances. A niveau egal et en prenant en compte les dépendances, on privilégie la tâche avec la durée d'exécution la plus longue.
2. **Affectation aux ressources** : chaque tâche est attribuée à la ressource disponible qui minimise son temps de fin estimé , optimisant ainsi l'utilisation des ressources.

Ce principe permet d'exploiter au mieux le parallélisme tout en minimisant le retard des tâches critiques.

4.2.1.1 Le niveau d'une tâche

Le niveau d'une tâche est une mesure de son importance structurelle dans le graphe de dépendances. Différentes définitions existent, mais il est généralement calculé en fonction du chemin critique, qui correspond à la plus longue séquence de dépendances menant à la fin du graphe. Plus précisément, le niveau d'une tâche peut être défini comme :

— **Niveau descendant (Bottom Level, BL)** : somme des temps d'exécution sur le plus long chemin entre la tâche et la dernière tâche du graphe qui lui est liée.

$$noveau(T_i) = p_i + \max_{T_j \in Succ(T_i)} niveau(T_j) \tag{4.1}$$

où :
— p_i est le temps d'exécution de la tâche T_i.
— $Succ(T_i)$ est l'ensemble des tâches successeurs immédiats de T_i.
— Si T_i est une tâche terminale (sans successeur), alors $niveau(T_i) = p_i$.
Cette approche permet de reflèter l'impact d'une tâche sur la durée totale d'exécution.

4.2.2 Explication de l'algorithme par le chemin critique

L'algorithme HLFET exploite la notion de chemin critique, qui correspond à la séquence de tâches dont la durée totale détermine le makespan du graphe d'ordonnancement. En priorisant les tâches situées sur ce chemin, il cherche à réduire au maximum le temps total d'exécution du graphe et à le faire converger vers le makespan.

L'efficacité de HLFET repose sur la capacité à bien estimer les niveaux des tâches et à effectuer une affectation optimale aux ressources. Toutefois, son efficacité peut être limitée par des déséquilibres de charge entre les ressources, notamment si plusieurs tâches critiques se retrouvent affectées à une même ressource du au fait qu'il affecte les tâches à la première ressource disponible minimisant le temps de fin de la ressource et en tenant compte des dépendances.

4.2.3 Complexité de l'algorithme HLFET

L'étude de Norre [3] met en évidence deux phases distinctes dans l'heuristique HLFET. La première consiste à calculer le plus long chemin dans le graphe de précédence (*niveau de chaque tâche*), et présente une complexité en $\mathcal{O}\left(T^2\right)$, considérant qu'on peut atteindre $E \approx T^2$ dans le pire des cas. La seconde phase, l'ordonnancement proprement dit, requiert un tri des tâches de complexité $\mathcal{O}\left(T \log T\right)$. L'algorithme HLFET se ramène donc à une complexité globale en $\mathcal{O}\left(T^2\right)$.

4.2.4 Motivations du choix de cette méthode

Le choix de l'algorithme HLFET (*Highest Level First with Estimated Times*) repose sur les critères suivants :

4.2.4.1 Qualité des solutions et efficacité prouvée

L'étude de [3] montre que HLFET produit des solutions proches de l'optimal, avec un écart moyen de 6% par rapport à la borne inférieure. Bien que des méthodes comme le recuit simulé puissent améliorer légèrement ce résultat (écart moyen de 4%), cet écart reste modéré au regard du surcoût computationnel de ces méthodes avancées.

4.2.4.2 Complexité algorithmique maîtrisée

HLFET a une complexité en $\mathcal{O}(n^2)$, ce qui le rend bien plus rapide que les méthodes exactes ($\mathcal{O}(2^n)$). Il est donc particulièrement adapté aux grands graphes de tâches (10^6 tâches), où les méthodes exactes deviennent impraticables.

4.2.4.3 Robustesse face à l'augmentation du nombre de tâches

L'étude de [3] met en évidence que l'efficacité de HLFET est stable même lorsque le nombre de tâches augmente, contrairement aux métaheuristiques dont le gain décroît sur des instances de grande taille. Cette scalabilité en fait un bon candidat pour les environnements HPC et cloud.

Chapitre 5

Etude expérimentale

5.1 Approche globale

5.1.1 Objectifs

Dans cette partie, nous allons étudier les résultats obtenus suite à l'étude de l'algorithme HLFET. On verra en premier lieu, le comportement de l'algorithme sur une machine locale, avant de le déployer sur le cloud. Dans la première partie, on compare l'efficacité de l'algorithme HLFET par rapport à un algorithme glouton, mais aussi d'examiner l'évolution du makespan en fonction du nombre de ressources disponibles et l'évolution du temps d'exécution en fonction du nombre de tâches et des dépendances. Dans la seconde partie, on souhaite évaluer le fonctionnement de l'algorithme pour un graphe de tâches d'ordre de 10^6 tâches avec un certain nombre de ressources.

5.2 Implémentation de l'algorithme HLFET

L'algorithme HLFET (*Highest Level First with Estimated Times*) a été implémenté pour l'ordonnancement de tâches représentées sous forme de graphe acyclique dirigé (DAG).

L'implémentation commence par la construction du DAG à partir des données d'entrée, suivie du calcul du *niveau descendant pondéré* (*Bottom Level*) de chaque tâche. Ce niveau est défini récursivement comme la somme de la durée de la tâche et du niveau le plus élevé parmi ses successeurs directs. Il reflète la contribution potentielle de chaque tâche au chemin critique global.

Le cœur de l'algorithme repose sur une boucle de planification itérative :

— À chaque itération, les tâches admissibles (dont toutes les dépendances ont été planifiées) sont identifiées.
— Parmi elles, celle ayant le niveau descendant le plus élevé est sélectionnée. En cas d'égalité, on choisit la tâche de plus longue durée.
— On détermine le *plus tôt début possible* en fonction de la complétion des dépendances.
— La tâche est affectée au cœur permettant de minimiser sa date de fin, en tenant compte de la disponibilité des cœurs.

L'ordonnancement est représenté par une structure regroupant, pour chaque cœur, la liste des tâches triées par ordre de début. Le *makespan* final correspond à la date de fin la plus tardive observée parmi tous les cœurs.

5.3 Étude du comportement de l'algorithme

5.3.1 Démarche expérimentale

Dans le cadre de cette étude, nous avons comparé plusieurs algorithmes d'ordonnancement appliqués à des graphes acycliques (DAG) de tâches, afin d'évaluer leur performance en termes de *makespan* (temps d'exécution total) et de temps de calcul.

5.3.1.1 Algorithmes testés

- **Algorithme glouton naïf (Greedy)** : cet algorithme traite les tâches dans l'ordre d'apparition, sans critère de priorité. À chaque itération, les tâches prêtes sont affectées au cœur disponible le plus tôt. Il s'agit d'une approche simple, non guidée par une estimation de criticité.
- **Algorithme HLFET (Highest Level First with Estimated Times)** : dans cette implémentation, la priorité est donnée aux tâches les plus critiques selon leur contribution au chemin critique. Chaque tâche est associée à un *niveau descendant pondéré* (*Bottom Level*), défini comme la somme de sa durée et du niveau maximal de ses successeurs. L'ordonnancement s'effectue en planifiant en priorité les tâches ayant le niveau le plus élevé, dans le but de minimiser le *makespan*.
- **Algorithme exact par backtracking** : pour des graphes de très petite taille , une approche exhaustive par backtracking est utilisée afin de déterminer l'ordonnancement optimal minimisant le *makespan*. L'algorithme explore récursivement l'ensemble des séquences admissibles de tâches, en respectant les dépendances, et teste toutes les affectations possibles aux cœurs. À chaque étape, les configurations sous-optimales sont élaguées (*pruning*) si le makespan partiel dépasse le meilleur trouvé jusque-là. Bien que garantissant l'optimalité, cette méthode devient rapidement inapplicable pour de grands graphes en raison de sa complexité exponentielle ($O(n! \cdot m^n)$).

5.3.1.2 Expériences et mesures

Les expériences consistent à générer des DAGs aléatoires dont la taille (nombre de tâches) et le nombre maximal de dépendances sont variables. Pour chaque configuration, plusieurs itérations sont effectuées afin de lisser l'effet de la génération aléatoire. Les mesures principales sont :

- Le **makespan** du planning obtenu.
- Le **temps d'exécution** de l'algorithme (évaluation de la complexité computationnelle).

5.3.2 Résultats

Cette section présente et analyse les résultats expérimentaux obtenus.

5.3.2.1 Évolution du makespan en fonction du nombre de cœurs

Cette expérience vise à étudier l'impact du nombre de cœurs sur le *makespan* obtenu par l'algorithme HLFET. Un DAG aléatoire de taille fixe (500 tâches) est généré, avec un maximum de 3 dépendances par tâche.

Pour chaque valeur du nombre de cœurs (entre 1 et 16), l'ordonnancement est exécuté 5 fois, et le makespan moyen est calculé.

Le code simule un environnement à ressources homogènes, sans latence de communication ni surcharge d'ordonnancement.

Résultats

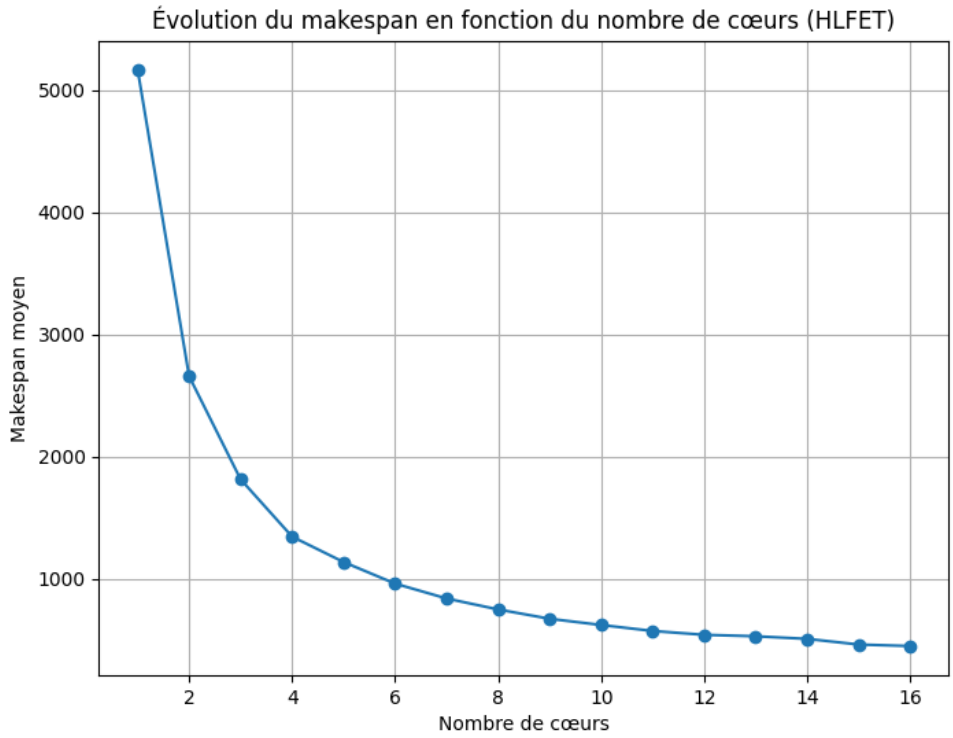

FIGURE 5.1 – Evolution du makespan en fonction du nombre de ressources

Le résultat met en évidence la décroissance du makespan avec l'augmentation du nombre de cœurs, jusqu'à un certain seuil où le chemin critique devient la borne inférieure. C'est en accord avec les études théoriques.

5.3.2.2 Influence des paramètres du graphe sur la complexité de l'algorithme

Nous avons évalué son comportement en mesurant :

— L'évolution du **makespan** en fonction du nombre de cœurs disponibles ;
— L'évolution du **temps d'exécution** en fonction du nombre de tâches et du nombre de dépendances ;
— La **scalabilité** de l'algorithme lorsque le nombre de ressources augmente.

L'étude a été réalisée sur une machine locale en mesurant le temps d'ordonnancement pour différentes configurations de graphes de tâches (DAG) et différentes ressources. On effectue nos tests à partir d'un premier graphe de 5000 tâches, 2 dépendances par tâches et 5 coeurs.

Résultats

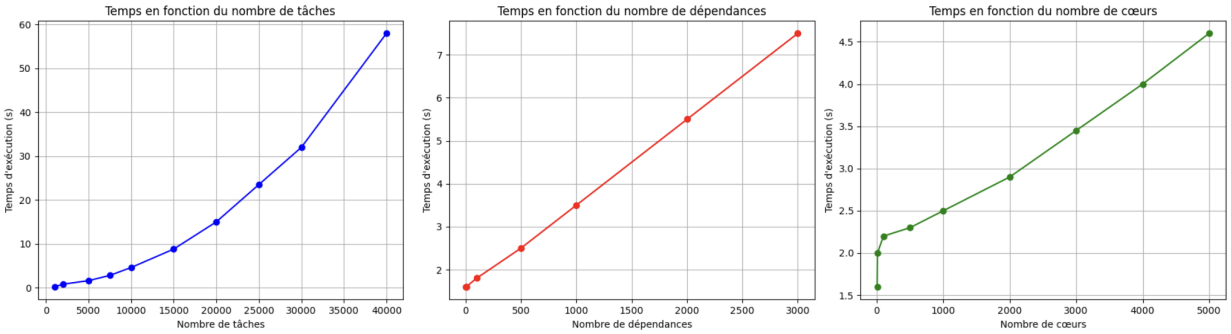

FIGURE 5.2 – Evolution du temps d'exécution de notre algorithme

Influence du nombre de tâches (n)

Complexité théorique : $O(n \log n)$
Observation : Le temps de calcul augmente de manière super-linéaire.
Conclusion : La croissance du temps d'exécution n'est pas strictement conforme à la complexité théorique en raison de l'overhead mémoire et du scheduling.

Influence du nombre de dépendances (m)

Complexité théorique : $O(nm)$
Observation : Jusqu'à $m = 100$, l'impact est faible ; au-delà de $m = 1000$, le temps double.
Conclusion : L'impact des dépendances devient significatif lorsque leur densité est élevée.

Influence du nombre de cœurs (num_cores)

Complexité théorique : $O(num\_cores)$
Observation : L'augmentation du nombre de cœurs réduit le makespan jusqu'à un certain seuil, au-delà duquel l'overhead de synchronisation réduit les gains.
Conclusion : La parallélisation est efficace jusqu'à un certain seuil, après quoi elle devient contre-productive.

5.3.2.3 Evaluation comparative de HLFET et d'un ordonnanceur glouton sur des graphes de tâches acycliques

Protocole expérimental

Les expériences consistent à comparer les performances de deux algorithmes d'ordonnancement sur des DAGs aléatoires de tailles croissantes :

— **HLFET (Highest Level First with Estimated Times)**, basé sur le niveau dans le DAG et la durée des tâches ;
— **Un ordonnancement glouton** basé sur le *Static Level*, i.e., la durée d'une tâche augmentée de l'effet de ses dépendants.

Pour chaque taille de graphe (de 10 à 400 tâches), un DAG est généré aléatoirement avec :

— des durées de tâches entières uniformes entre 1 et 20 ;
— un maximum de 3 dépendances par tâche vers des prédécesseurs (DAG peu dense).

Chaque expérience est répétée trois fois. Pour chaque algorithme, sont mesurés :

— le **makespan** obtenu ;
— le **temps d'exécution réel** (en secondes).

Les tests sont réalisés sur 4 cœurs homogènes. Les résultats sont présentés sous forme de courbes en fonction du nombre de tâches.

Résultats

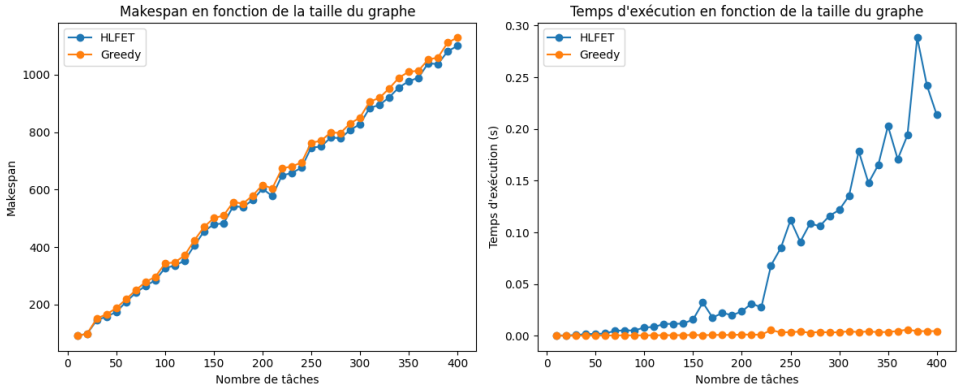

FIGURE 5.3 – Makespan et Temps d'exécution de l'algorithme HLFET et glouton en fonction du nombre de tâches

Avec un nombre maximal de dépendances fixé par tâche dans le graphe, on constate que l'algorithme HLFET produit un makespan légèrement inférieur à celui de l'algorithme glouton. En revanche, son temps d'exécution est nettement plus élevé, avec une croissance de complexité quadratique clairement observable. Cette tendance reste toutefois sujette à des variations liées

5.3.2.4 Comparaison avec un algorithme exact

Cette expérience vise à évaluer la qualité des heuristiques HLFET et Greedy en les comparant à un algorithme exact de référence, basé sur un backtracking exhaustif. Ce dernier explore toutes les planifications possibles et fournit un makespan optimal, mais il est limité à de petits graphes (≤ 15 tâches) en raison de sa complexité exponentielle.

Chaque algorithme est exécuté sur des DAGs aléatoires générés avec un maximum de 3 dépendances par tâche. Pour chaque taille, l'expérience est répétée trois fois afin de lisser les mesures. Les performances sont évaluées en fonction du **makespan moyen** obtenu.

Résultats

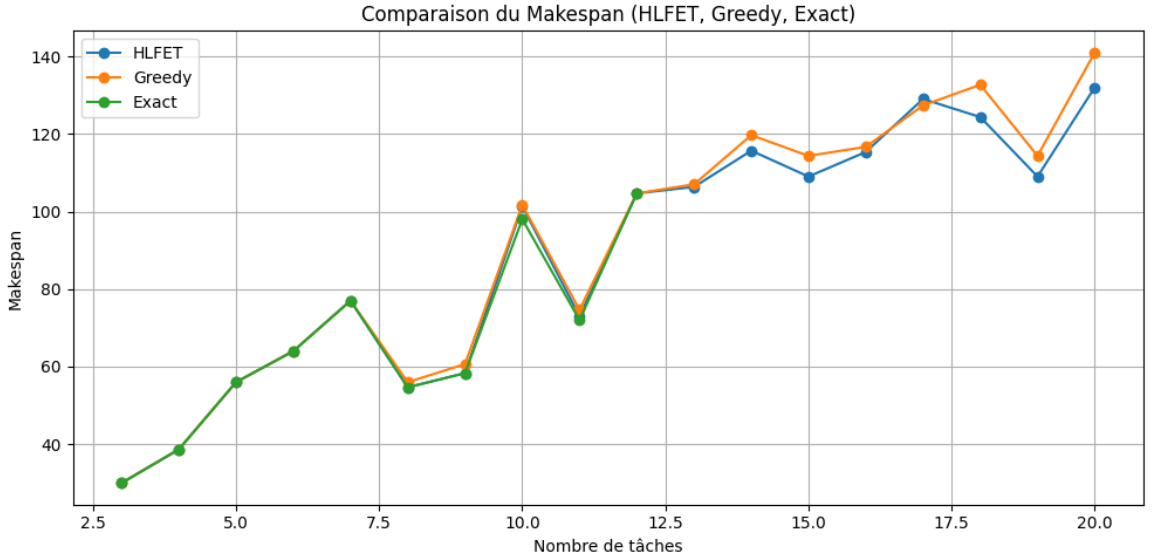

FIGURE 5.4 – Comparaison des performances entre un algorithme exact, glouton et HLFET

Comme la complexité d'un algorithme exact est exponentielle en fonction du nombre de tâches, les tests ne peuvent être effectués que sur de très petits graphes. L'itération a été limitée à 12 tâches pour l'algorithme exact, car au-delà, le temps de calcul devient prohibitif.

On observe que pour un nombre très réduit de tâches, les trois algorithmes produisent des résultats identiques. Cependant, au-delà de 7 tâches, bien que les résultats de l'algorithme glouton et de HLFET continuent de coïncider, l'algorithme exact fournit des makespans systématiquement meilleurs, comme attendu compte tenu de sa capacité à explorer l'espace complet des ordonnancements admissibles et à garantir la solution optimale.

5.3.2.5 Impact du nombre de dépendances et de la taille du DAG sur les performances

Cette expérience vise à analyser l'effet combiné de deux facteurs :

— le nombre de tâches dans le DAG (`task_sizes`) ;
— le nombre maximal de dépendances par tâche (`max_dependencies`).

Pour chaque combinaison de ces paramètres, des DAGs aléatoires sont générés et ordonnancés à l'aide des algorithmes HLFET et Greedy. L'expérience est répétée trois fois pour lisser les variations, et les métriques suivantes sont collectées :

— le **makespan** moyen ;
— le **temps d'exécution** moyen de l'algorithme.

Les résultats sont représentés sous forme de nuages de points 3D afin de visualiser les évolutions du makespan et du temps d'exécution en fonction de la densité du graphe et de sa taille.

Résultats

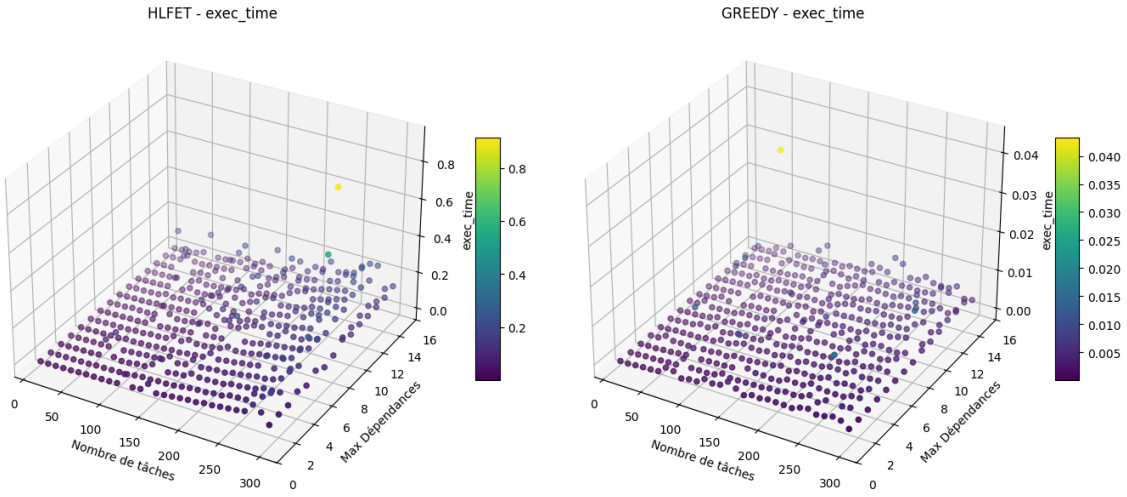

FIGURE 5.5 – Evolution du temps d'exécution en fonction de la tailles de graphes et du nombre de dépendances maximales pour l'algorithme HLFET et glouton

L'algorithme HLFET étant plus sophistiqué, il est tout à fait normal que son temps d'exécution soit supérieur à celui de l'algorithme glouton, en particulier lorsque le nombre de dépendances augmente. Néanmoins, à l'échelle des tests réalisés (nombre de tâches inférieur à 300), les écarts de temps restent modérés et ne permettent pas d'observer des différences marquées en pratique.

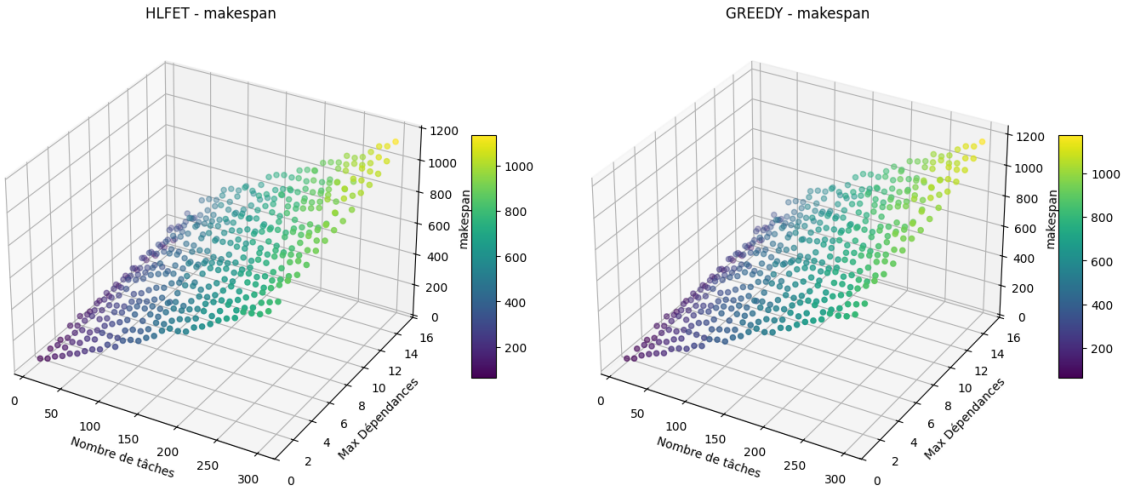

FIGURE 5.6 – Evolution du makespan en fonction de la tailles de graphes et du nombre de dépendances maximales pour l'algorithme HLFET et glouton

Comme dans les expériences précédentes de comparaison de makespans, on observe une légère amélioration avec l'algorithme HLFET, bien que cette différence ne soit pas encore significative à l'échelle des graphes considérés.

5.3.3 Conclusion de l'étude expérimentale hors cloud

L'ensemble des expériences menées confirme l'intérêt d'utiliser des heuristiques guidées par la structure du graphe pour améliorer la qualité de l'ordonnancement. L'algorithme HLFET, en exploitant une mesure du niveau critique des tâches (Bottom Level), permet globalement d'obtenir de meilleurs makespans que l'approche gloutonne naïve, en particulier lorsque la taille des graphes augmente.

Cependant, cette amélioration en qualité s'accompagne d'un surcoût en temps de calcul, plus sensible lorsque le nombre de dépendances est élevé. À l'échelle des graphes testés (jusqu'à 300 tâches), ce surcoût reste modéré et tolérable dans un contexte non temps réel.

La comparaison avec un algorithme exact a également permis de valider la pertinence des heuristiques utilisées. Pour de petits graphes, HLFET et Greedy coïncident souvent avec la solution optimale, mais au-delà d'un certain seuil (7 à 10 tâches), seul l'algorithme exact parvient systématiquement à minimiser le makespan.

Enfin, l'analyse conjointe de la taille du graphe et de sa densité montre que le parallélisme est mieux exploité lorsque le nombre de dépendances reste faible, ce qui renforce l'intérêt d'utiliser HLFET dans des cas peu contraints structurellement.

5.4 Déploiement de l'algorithme sur le cloud

5.4.1 Démarche

Pour tester la scalabilité de l'algorithme HLFET en environnement cloud, nous avons utilisé AWS Lambda, qui permet l'exécution de fonctions sans gestion de serveurs. L'objectif était d'évaluer les performances de l'algorithme sur un graphe de 1 000 000 tâches.

En raison des limitations de mémoire (10GB de RAM) et de durée d'exécution sur Lambda (15 minutes), il n'a pas été possible de traiter un graphe de 10^6 tâches. Le test s'est donc concentré sur un DAG de **100 000 tâches**.

L'algorithme a été déployé sous forme de fonction Python sur AWS Lambda, avec un fichier de graphe (au format `JSON`) transféré manuellement depuis une machine locale, afin de minimiser le temps de calcul sur le cloud. Ce fichier contient la structure d'un DAG généré en amont, la création du graphe étant assez coûteuse en temps.

Le test a consisté à exécuter l'algorithme HLFET sur Lambda pour 3 critères différents :

- le nombre de tâches pour un même nombre de cœurs (5 cœurs, avec une durée maximale de 20 par tâche et un maximum de 3 dépendances par tâche)
- le nombre de cœurs simulés pour un même graphe (de 100 000 tâches, avec une durée maximale de 20 par tâche et un maximum de 3 dépendances par tâche)
- le nombre de dépendances pour un même nombre de tâches (10 000 tâches, avec une durée maximale de 20 par tâche, 5 cœurs)

en mesurant :

- le makespan produit par l'algorithme
- le temps d'exécution réel de la fonction d'ordonnancement (mesuré via `time` sur Python).

5.4.2 Résultats

5.4.2.1 Variation du nombre de tâches

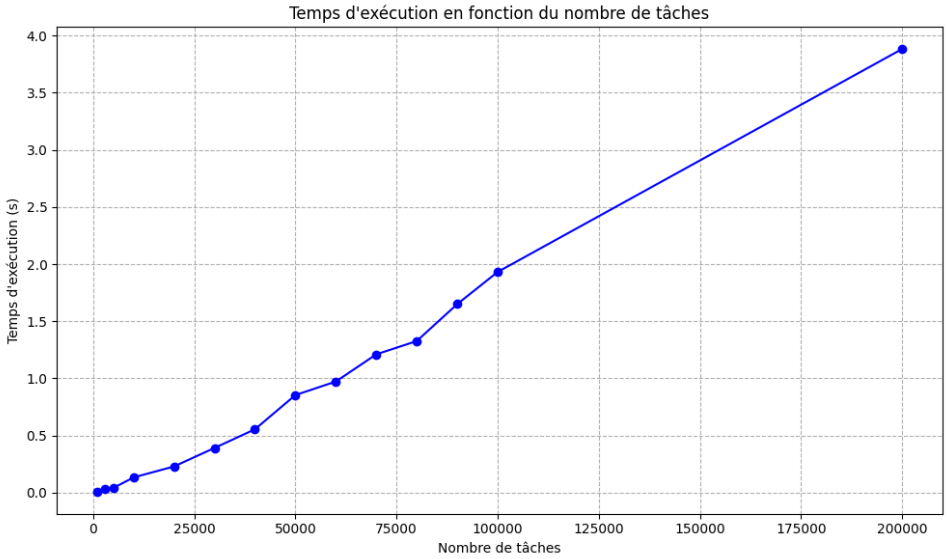

FIGURE 5.7 – Temps d'exécution de l'algorithme en fonction du nombre de tâches

La figure 5.7 montre une augmentation quasi linéaire du temps d'exécution en fonction du nombre de tâches, avec une croissance légèrement convexe. Ce resultat est quelque peut incohérent avec avec la complexité réelle de l'algorithme qui est en $\mathcal{O}(n^2)$.

Ce résultat pourrait s'expliquer par l'insuffisance du nombre de points pris pour le tracé de la courbe et par le petit nombre de dépendances (3 dépendances par tâches au maximum).

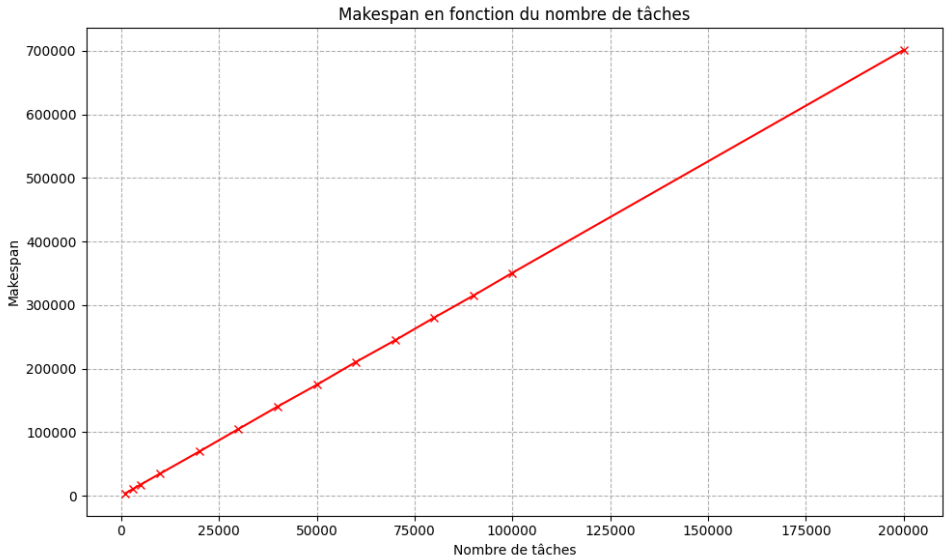

FIGURE 5.8 – Makespan de l'algorithme en fonction du nombre de tâches

La figure 5.8 présente une évolution linéaire du makespan avec le nombre de tâches. Cette

croissance est attendue : plus le graphe contient de tâches, plus la durée globale d'exécution augmente, même avec un degré de parallélisme constant.

5.4.2.2 Variation du nombre de cœurs pour un même graphe

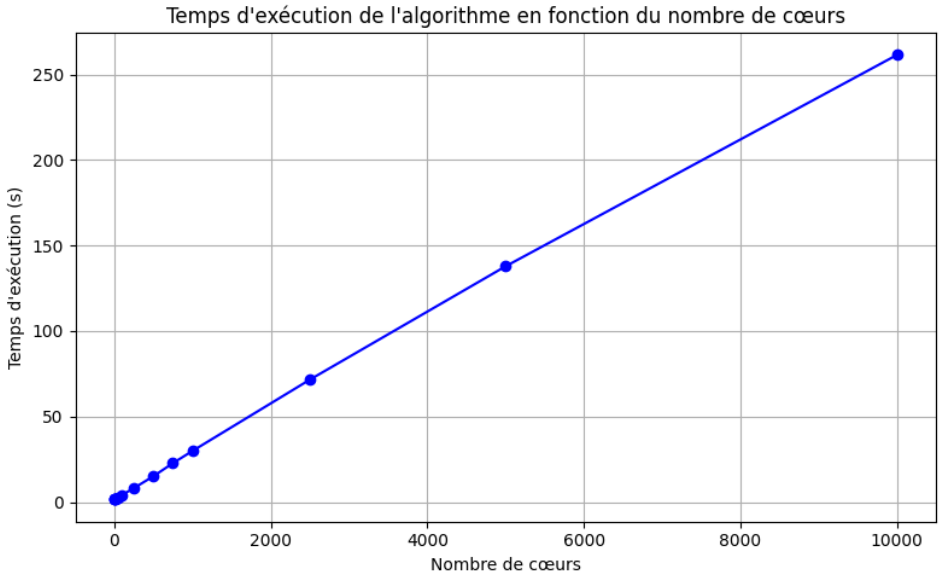

FIGURE 5.9 – Temps d'exécution de l'algorithme en fonction du nombre de cœurs (échelle linéaire)

Comme le montre la figure 5.9, le temps d'exécution augmente de façon quasi linéaire avec le nombre de cœurs. La complexité est donc proportionnelle au nombre de cœurs. Elle est cohérente avec le fonctionnement de HLFET, dans lequel chaque tâche est affectée à la ressource minimisant son temps de fin estimé. Lorsque le nombre de cœurs augmente, le nombre d'opérations de sélection croît, rendant la phase d'affectation plus coûteuse. Ainsi, bien que le nombre de tâches soit constant, le coût global du scheduling augmente. L'algorithme reste stable mais devient progressivement plus coûteux en exécution à mesure que les ressources augmentent. Ce résultat met en lumière une limite à l'augmentation du nombre de cœurs : le gain marginal devient rapidement négligeable face à l'augmentation du coût de planification.

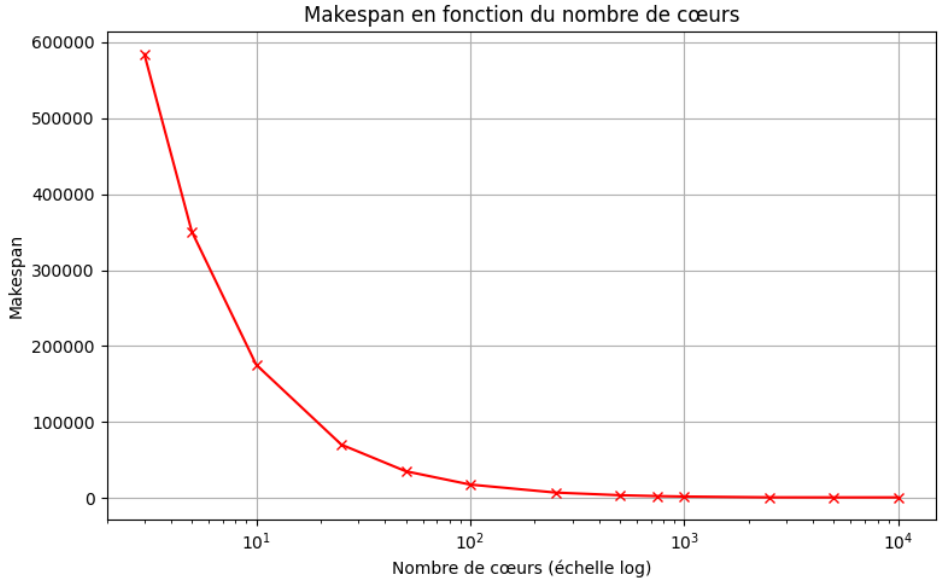

FIGURE 5.10 – Makespan en fonction du nombre de cœurs (échelle logarithmique)

Cependant, la figure 5.10 indique que le makespan se stabilise à partir de 1000 cœurs, illustrant une saturation du parallélisme. Au-delà d'un certain seuil, le chemin critique devient le facteur limitant : les dépendances empêchent une exécution pleinement parallèle, même si davantage de cœurs sont disponibles.

5.4.2.3 Variation du nombre de dépendances maximal

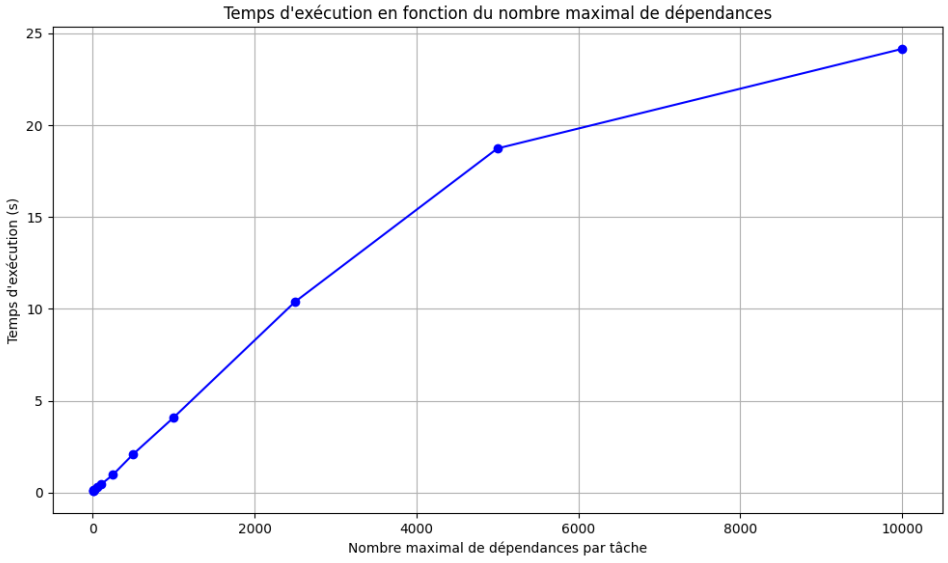

FIGURE 5.11 – Temps d'exécution de l'algorithme en fonction du nombre de dépendances max

La figure 5.11 montre une croissance significative du temps d'exécution avec le nombre maximal de dépendances par tâche. En effet, l'augmentation du nombre d'arêtes dans le graphe

implique une densité plus élevée, rendant plus complexe le calcul des niveaux critiques ainsi que le respect des contraintes de précédence dans le planning.

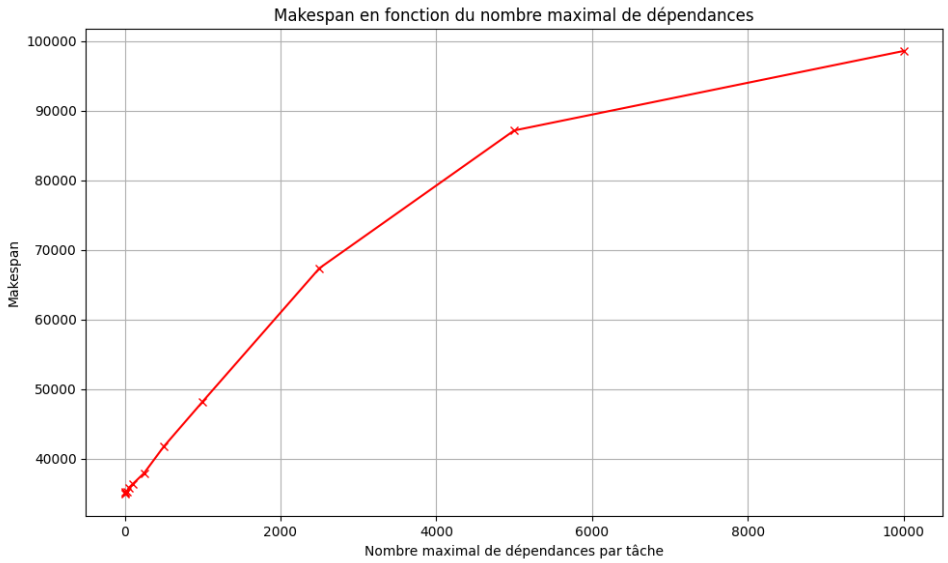

FIGURE 5.12 – Makespan de l'algorithme en fonction du nombre de dépendances max

La figure 5.12 montre que le makespan augmente lui aussi avec la densité des dépendances. Plus les tâches ont de prédécesseurs, plus le graphe devient contraint, ce qui ralentit l'exécution globale. Toutefois, cette croissance est moins brutale que celle du temps d'exécution. Elle est sous-linéaire, indiquant une convergence de ce makespan vers une borne supérieure qui n'est que le chemin critique du graphe qui est irréductible.

5.4.3 Conclusion

Pour conclure :

— L'augmentation du nombre de tâches impacte à la fois le makespan (de façon linéaire) et le temps d'exécution (sous-linéaire à quadratique).
— L'augmentation du nombre de cœurs réduit significativement le makespan jusqu'à un certain seuil, mais augmente le temps d'exécution en raison du coût croissant de l'ordonnancement.
— L'augmentation du nombre maximal de dépendances rend le graphe plus dense, augmentant à la fois le makespan et le temps d'exécution de manière sous-linéaire.

Bien que la complexité théorique de l'algorithme HLFET soit quadratique dans le pire des cas (notamment en raison du calcul des niveaux et de l'ordonnancement avec contraintes), les résultats expérimentaux montrent une croissance quasi linéaire du temps d'exécution sur le cloud. Cela peut s'expliquer par plusieurs facteurs :

— **La structure des graphes testés** : ceux-ci sont peu denses, avec un nombre limité (3) de dépendances par tâche. Dans ce contexte, la complexité effective du calcul est inférieure à la borne théorique.
— **L'allocation dynamique de ressources sur le cloud** : AWS Lambda augmente automatiquement la puissance CPU avec la mémoire allouée, ce qui permet de traiter

plus efficacement les cas volumineux et masque partiellement la croissance de la charge algorithmique.
— **L'optimisation de l'implémentation** : l'utilisation de structures de données efficaces (comme des tas (heap) pour sélectionner les ressources disponibles) limite l'impact du nombre de cœurs ou de tâches sur le coût de l'ordonnancement.

Ainsi, la courbe quasi linéaire obtenue expérimentalement reflète davantage le comportement moyen dans un contexte cloud bien dimensionné, que la complexité asymptotique pessimiste du modèle.

En comparaison avec l'implémentation hors cloud, l'implémentation cloud permet de traiter des graphes bien plus grands (jusqu'à 100 000 tâches testées ici, la configuration de l'environnement ne permettant pas de passer à 1 million de tâches) avec une performance stable. Le coût en temps d'exécution est naturellement plus élevé, en raison des limitations d'environnement (mémoire, latence réseau), mais reste raisonnable pour des graphes peu denses. Pour des graphes plus denses, la complexité se rapprocherait de $\mathcal{O}(n^2)$, la complexité pire cas de HLFET. Contrairement à l'implémentation locale, le cloud offre une flexibilité pour tester à grande échelle, au prix d'un surcoût en gestion des ressources. En revanche, l'implémentation locale est mieux adaptée aux graphes de taille modérée, avec une exécution plus rapide mais des limitations mémoire et CPU dès que la taille ou la densité augmente significativement.

Chapitre 6

Conclusion

6.1 Synthèse de l'étude

Ce projet a permis d'explorer la problématique d'ordonnancement de tâches sous contraintes de dépendances en modélisant le problème sous forme de graphe acyclique dirigé (DAG). L'algorithme HLFET, fondé sur une hiérarchisation des tâches via le niveau descendant pondéré (Bottom Level), a été implémenté et évalué dans différents environnements.

L'étude expérimentale a d'abord été menée en local, en comparaison avec un ordonnanceur glouton naïf et un algorithme exact par backtracking. Ces analyses ont permis de confirmer la supériorité de HLFET en termes de makespan, notamment sur des graphes de taille croissante et de structure peu contrainte. L'algorithme exact, bien qu'inapplicable à grande échelle, a servi de référence pour valider la qualité des heuristiques sur de petits graphes.

Le déploiement sur le cloud a ensuite permis d'évaluer la scalabilité de l'algorithme sur des DAGs massifs, jusqu'à 100 000 tâches. Malgré des contraintes liées à la mémoire et au temps d'exécution, les résultats ont montré un comportement stable de l'algorithme, avec une croissance linéaire du temps de calcul dans la plupart des cas. Ces résultats suggèrent que l'implémentation cloud est adaptée à des cas de grande ampleur, à condition que la densité des dépendances reste modérée.

En définitive, le projet a montré que l'heuristique HLFET constitue un bon compromis entre qualité de l'ordonnancement et temps de calcul, tout en offrant une robustesse suffisante pour être envisagée dans des environnements distribués. Ce travail ouvre la voie à des extensions vers des modèles plus complexes (ressources hétérogènes, priorités dynamiques, contraintes temps réel) et à une exploitation plus poussée du cloud computing pour les problèmes d'ordonnancement à grande échelle.

6.2 Limites et perspectives

6.2.1 Limites

Ce travail, bien qu'abouti sur plusieurs aspects, présente un certain nombre de limites :

— **Scalabilité partielle sur le cloud** : en raison des limitations imposées par l'environnement AWS Lambda (temps d'exécution, mémoire, taille de fichier), il n'a pas été possible de tester l'algorithme HLFET sur un graphe de 10^6 tâches comme initialement prévu. Le test s'est limité à 100 000 tâches.

— **Génération de graphes coûteuse** : la génération aléatoire de DAGs très larges et cohérents (sans cycles) s'est révélée coûteuse en mémoire et en temps. Elle a nécessité une préparation locale, ce qui limite l'automatisation complète des tests sur le cloud.
— **Modèle simplifié** : le modèle étudié suppose des ressources homogènes, sans surcharge de communication ni préemption. Il ne prend pas en compte des contraintes plus réalistes comme les fenêtres temporelles, les priorités ou les coûts de transfert de données.
— **Absence d'optimisation parallèle de l'algorithme** : bien que HLFET soit testé sur un environnement parallèle (multi-cœurs), l'algorithme lui-même reste séquentiel dans son implémentation, ce qui limite les gains en temps d'exécution.

6.2.2 Perspectives

Plusieurs pistes d'amélioration et d'extensions peuvent être envisagées :
— **Implémentation distribuée de HLFET** : paralléliser l'algorithme lui-même pour répartir le coût du calcul des niveaux et de l'affectation des tâches, en particulier dans un environnement de type cluster ou cloud fonctionnel.
— **Comparaison avec d'autres heuristiques avancées** : tester d'autres algorithmes classiques (HEFT, ETF, MCP, etc.) pour positionner HLFET de manière plus fine dans l'écosystème des heuristiques d'ordonnancement.
— **Extension à des modèles hétérogènes** : introduire des cœurs aux vitesses différentes, des tâches avec priorités ou des dépendances pondérées par des délais, pour approcher des cas industriels plus complexes.
— **Exploration des topologies de graphes spécifiques** : analyser l'impact de structures particulières (arbres, chaînes, séries-parallèles, DAGs aléatoires très denses) sur la performance des algorithmes.
— **Optimisation automatique des paramètres** : utiliser des techniques d'apprentissage ou de métaheuristiques (recherche tabou, algorithmes génétiques) pour ajuster dynamiquement les priorités ou critères de tri des tâches.

Bibliographie

[1] Nisha Devi, Sandeep Dalal, Kamna Solanki, Surjeet Dalal, Umesh Kumar Lilhore, Sarita Simaiya, and Nasratullah Nuristani. A systematic literature review for load balancing and task scheduling techniques in cloud computing. *Artificial Intelligence Review*, 57 :276, 2024.

[2] Heba M. Fadhil. Optimizing task scheduling and resource allocation in computing environments using metaheuristic methods. *Fusion : Practice and Applications*, 15(1) :157–179, 2024.

[3] S. Norre. Ordonnancement de tâches sur un système multiprocesseur - modèles déterministes et modèles stochastiques. *Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle*, 28(3) :221–253, 1994.

[4] Mohammad Abu Obaida and Jason Liu. Simulation of hpc job scheduling and large-scale parallel workloads. In W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, editors, *Proceedings of the 2017 Winter Simulation Conference*, pages 920–931. IEEE, 2017.

[5] PlaniSense. Ordonnancement : Méthodes et outils pour une gestion optimisée des tâches, 2024. Consulté le 19 mars 2025.