SORBONNE UNIVERSITÉ

Année Universitaire: 2023-2024

Rapport de projet

《 *Task Scheduling*》

Projet réalisé par

Mengqian XU
Yujia TU

Projet encadré par

Didier Smets

29/12/2023

Sommaire

Introduction.....2

Formalisation du Problème.....3

Déroulement du projet.....4

- Définition des Tâches et Création du Graph.....4
- Implémentation de la Simulation Parallèle.....5
- Utilisation de la Topologie.....6

Résultat d'un exemple.....8

Difficultés du projet.....9

Amélioration Possible du projet.....10

Conclusion.....11

Introduction

Le projet vise à résoudre un problème de gestion de tâches avec des dépendances, nécessitant une exécution séquentielle tout en explorant des possibilités d'optimisation via la parallélisation. Nous avons développé un système basé sur des graphes dirigés acycliques (DAG) pour modéliser les dépendances entre les tâches. L'objectif principal est d'optimiser le temps d'exécution total en utilisant des techniques de parallélisation avec Thread.

Les objectifs de notre projet sont multiples :

- Développer un graphe dirigé acyclique (DAG) pour modéliser les dépendances entre les tâches.
- Implémenter des algorithmes de topologie, en mettant particulièrement l'accent sur la méthode basée sur les degrés d'entrée.
- Concevoir un système de simulation parallèle permettant l'exécution simultanée de tâches dépendantes.
- Explorer et comparer les avantages de l'utilisation de threads pour la parallélisation des opérations.
- Mettre en œuvre une allocation dynamique de threads basée sur des considérations de dépendance et de disponibilité de ressources.

Formalisation du Problème

Le problème est modélisé à l'aide d'une structure de graphe, où chaque tâche est représentée par un nœud et les dépendances entre les tâches sont représentées par des arêtes orientées. Il est essentiel de souligner que le graphe est acyclique, garantissant ainsi qu'il existe une séquence d'exécution valide pour toutes les tâches.

Chaque nœud du graphe représente une tâche, et les flèche définissent les dépendances entre les tâches. Par exemple, si la tâche B et C dépend de la tâche A, il y aura une flèche de A vers B et une flèche de A vers C dans le graphe.

Ce modèle permet une représentation claire des dépendances entre les tâches.

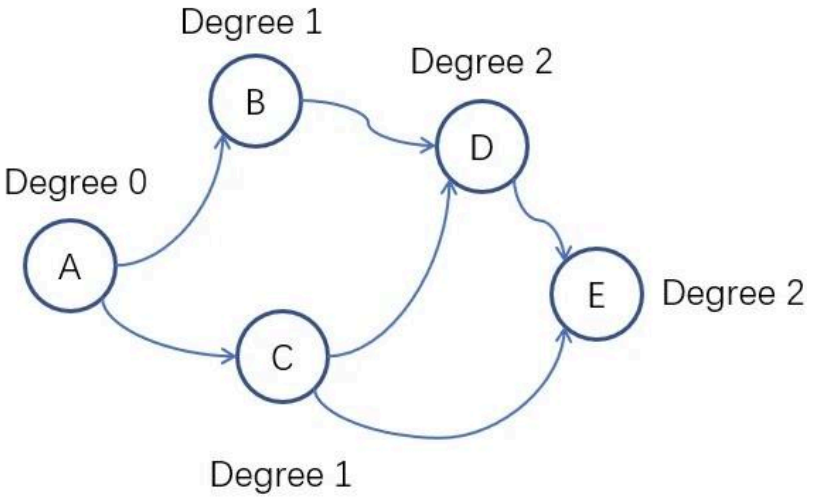

figure 1 : exemple pour DAG

Afin de permettre une exécution parallèle des tâches, nous avons choisi d'adopter une approche basée sur l'utilisation de threads. Chaque tâche a été traitée comme une entité autonome, capable de s'exécuter indépendamment tout en respectant les dépendances définies par le DAG.

Déroulement du projet

Le projet s'est déroulé en plusieurs étapes clés:

- **Définition des Tâches et Création du Graph**
  Avant de commencer le projet, la première étape cruciale a été de déterminer la meilleure façon de représenter le graphique. La représentation du graphique joue un rôle clé dans l'efficacité de la résolution du problème. Après une réflexion approfondie et en se basant sur les enseignements du cours, nous avons opté pour l'utilisation d'une structure de données basée sur la représentation par adjacence.

  Nous avons pris en compte les deux approches classiques pour représenter un graphique, à savoir l'adjacency list (liste d'adjacence) et l'adjacency matrix (matrice d'adjacence). Selon nos analyses, la matrice d'adjacence est généralement utilisée pour les graphiques denses et de grande taille, tandis que la liste d'adjacence est préférée pour les graphiques peu denses, ce qui est le cas de notre application.

  En suivant cette compréhension, nous avons implémenté la représentation par liste d'adjacence, permettant une gestion efficace des dépendances entre les tâches. Nous avons défini chaque tâche en spécifiant son identifiant, sa durée d'exécution, et les tâches sur lesquelles elle dépend.

```
struct Graph {
    int vertices;
    struct Task** tasks;
    int* inDegree;
};
```

figure 2 : struct Graph

  Nous établissons une structure de tâche en attribuant à chaque tâche un identifiant unique, un temps d'exécution, et d'autres informations nécessaires. Nous simulons l'exécution de chaque tâche en attendant la fin de toutes les tâches dépendantes, calculons l'heure de début, exécutons la tâche, enregistrons l'heure de fin, et notifions les autres tâches dépendantes.

- ## Implémentation de la Simulation Parallèle

  Chaque tâche a été simulée en tant que thread indépendant. Les threads ont été créés de manière dynamique, en prenant en compte les dépendances entre les tâches. Ainsi, les tâches peuvent s'exécuter simultanément, respectant les contraintes définies par le graphe.

```
struct Task {
    int id;
    int duration;
    int* dependencies;
    int num_dependencies;
    int start_time;
    int end_time;
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int num_dependencies_left;
};
```

figure 3 : struct Task

La planification et la création de threads suivent les étapes suivantes :

1. Simulation de l'exécution des threads

Chaque tâche est exécutée par un thread, avec la fonction de thread définie comme **“executeTask”**. Cette fonction simule le processus d'exécution de la tâche, notamment en attendant l'achèvement de toutes les tâches dépendantes, en calculant le temps de début de la tâche, en simulant l'exécution de la tâche, en enregistrant le temps de fin, etc.

2. Création dynamique des threads

Les threads sont créés de manière dynamique, chaque tâche correspondant à un thread. Dans la fonction **dynamicThreadAllocation**, un pool de threads est créé à l'aide de la fonction **“pthread_create”**, chaque thread exécutant une tâche. Ainsi, les tâches peuvent être exécutées en parallèle en fonction de leurs dépendances.

3. Calcul du temps de début de la tâche

4. Attente de l'achèvement des tâches dépendantes

Acquérir un mutex avant d'exécuter une tâche. Pendant le processus d'exécution de chaque tâche, le thread attend que toutes les tâches dépendantes soient terminées. Cela est réalisé à l'aide de verrous (type **pthread_mutex_t**) et de variables de condition (type **pthread_cond_t**). La partie attendre que les tâches dépendantes soient terminées se situe autour de l'appel à

```
pthread_mutex_lock(&t->lock);
pthread_cond_wait(&task->cond, &task->lock);
```

pthread_cond_wait.

figure 4 : Application du verrouillage mutex

5. Simulation de l'exécution de la tâche

Utilisation de la fonction **usleep** pour simuler l'exécution de la tâche, avec une durée égale à la durée de la tâche (**duration**).

6. Enregistrement du temps de fin

7. Notification des tâches dépendantes

En adoptant ces pratiques, notre implémentation de la simulation parallèle a permis une exécution efficace des tâches, tout en respectant les contraintes imposées par la structure du graphe.

```
usleep(task->duration * 1000);
```

figure 5 : Simuler l'exécution de la tâche

- ## Utilisation de la Topologie

La topologie du graphe a été exploitée pour réaliser une sorte d'ordonnancement optimal des tâches. En utilisant un tri topologique, nous avons déterminé un ordre d'exécution qui respecte toutes les dépendances, facilitant ainsi une exécution parallèle efficace.

Nous avons exploré deux approches fondamentales pour la mise en œuvre de

l'algorithme de topologie, nécessaire à l'ordonnancement des tâches dans notre graphe dirigé acyclique (DAG). Ces deux méthodes sont basées sur la recherche en profondeur (DFS) et sur les degrés d'entrée des nœuds.

1. **Méthode basée sur DFS**
   Nous avons initialement considéré l'approche basée sur la recherche en profondeur (DFS) pour obtenir un ordre topologique des tâches. Cela est en adéquation avec notre programme de cours et a été une première étape naturelle dans notre processus de conception.

2. **Méthode basée sur les Degrés d'Entrée**
   En poursuivant notre exploration, nous avons également expérimenté la méthode basée sur les degrés d'entrée des nœuds. Après une analyse approfondie, nous avons constaté que cette méthode était particulièrement adaptée à la représentation en liste d'adjacence (Adjacency List).

- **Choix Final - Adaptation à la Structure du Graph**
  Après avoir évalué ces deux approches, nous avons pris la décision d'adopter la méthode basée sur les degrés d'entrée, en raison de son ajustement optimal à la structure spécifique de notre graphe, représenté en liste d'adjacence. Cette décision découle de la nature des opérations sur notre graphe, où la méthode basée sur les degrés d'entrée s'est révélée plus efficace pour la gestion des dépendances dans une structure de liste d'adjacence.

  Notre recherche a confirmé que la méthode basée sur DFS était plus adaptée aux représentations en matrice d'adjacence, alors que la méthode basée sur les degrés d'entrée était plus efficace avec les listes d'adjacence, en accord avec la littérature spécialisée.

  Ce choix stratégique a été crucial pour le succès global de notre projet, soulignant l'importance de l'alignement entre la méthode algorithmique choisie et la structure spécifique du graphe que nous avons manipulé dans le cadre de notre projet de simulation parallèle.

Résultat d'un exemple

Le programme affichera les heures de début et de fin de chaque tâche, ainsi que le temps total nécessaire à l'exécution de toutes les tâches.

Exemple 1: pour n = 5 , on initialise **duration = 300**

```
Exécution de la tâche 0, durée 300, heure de début 0
Exécution de la tâche 1, durée 300, heure de début 300
Exécution de la tâche 2, durée 300, heure de début 300
Exécution de la tâche 3, durée 300, heure de début 600
Exécution de la tâche 4, durée 300, heure de début 600
Heure de début de la tâche 0 : 0, heure de fin : 300
Heure de début de la tâche 1 : 300, heure de fin : 600
Heure de début de la tâche 2 : 300, heure de fin : 600
Heure de début de la tâche 3 : 600, heure de fin : 900
Heure de début de la tâche 4 : 600, heure de fin : 900
Temps total : 900
```

figure 6 : exemple1 pour exécuter la fonction

Exemple 2: Pour n = 7, on initialise **duration = rand() % 900 + 100**

```
Exécution de la tâche 1, durée 935, heure de début 0
Exécution de la tâche 0, durée 449, heure de début 0
Exécution de la tâche 3, durée 170, heure de début 0
Exécution de la tâche 2, durée 139, heure de début 935
Exécution de la tâche 6, durée 312, heure de début 935
Exécution de la tâche 4, durée 289, heure de début 1074
Exécution de la tâche 5, durée 483, heure de début 1363
Heure de début de la tâche 0 : 0, heure de fin : 449
Heure de début de la tâche 1 : 0, heure de fin : 935
Heure de début de la tâche 2 : 935, heure de fin : 1074
Heure de début de la tâche 3 : 0, heure de fin : 170
Heure de début de la tâche 4 : 1074, heure de fin : 1363
Heure de début de la tâche 5 : 1363, heure de fin : 1846
Heure de début de la tâche 6 : 935, heure de fin : 1247
Temps total : 1846
```

figure 7 : exemple 2 pour exécuter la fonction

Difficultés du projet

Nous avons principalement rencontré des difficultés dans deux parties. Dans la première partie, nous devons représenter les tâches et les dépendances sous forme de graphe acyclique dirigé (DAG) et utiliser un algorithme de tri topologique pour trouver un ordonnancement de tâches raisonnable. Dans la deuxième partie, nous devons prendre en compte la durée et la parallélisation des tâches, ainsi que la minimisation du temps total d'achèvement.

Voici les difficultés que nous avons rencontrées et leurs solutions :

- **Comprendre les graphes et le tri topologique**
  N'étant pas familiers avec le concept de tri topologique, nous avons recherché des algorithmes classiques à partir desquels nous pourrions apprendre. Nous avons représenté le graphe à l'aide de listes d'adjacences.

- **Parallélisation des tâches et optimisation du temps total d'exécution**
  C'est un défi car nous n'avons pas d'expérience en programmation multithread ou en gestion de la concurrence. Nous avons dû apprendre à créer et gérer des threads, ainsi qu'à synchroniser leur exécution pour respecter les dépendances entre les tâches. De plus, minimiser le temps total d'exécution est également un défi, car cela nécessite de trouver une allocation optimale des tâches aux threads.

- **Gestion de la mémoire**
  Travaillant avec des structures de données dynamiques, nous devons être attentifs à la gestion de la mémoire. Cela comprend l'allocation et la libération appropriées de la mémoire pour éviter les fuites. Parallèlement, nous avons mis en place un avertissement en cas de fuite de mémoire.

- **Tests et débogage**
  Tester et déboguer notre code est un défi, surtout lorsque nous travaillons avec des structures de données complexes et de la programmation multithread. Nous avons effectué des tests en personnalisant les tâches et leur séquence, et en définissant aléatoirement les tâches et leur séquence.

Amélioration Possible du projet

- **Limite de Temps pour les Threads**
  Pour renforcer la sécurité du système et prévenir les situations potentiellement malveillantes, une amélioration possible serait l'ajout d'une limite de temps d'exécution pour chaque thread. Si un thread dépasse cette limite, il pourrait être automatiquement interrompu ou tué. Cela éviterait les situations où un thread monopolise excessivement les ressources du système.

- **Utilisation d'une Barrière pour Protéger le Système**
  Une barrière de protection pourrait être mise en place pour éviter que le système ne continue à exécuter indéfiniment certaines tâches.

- **Incorporation de la notion de Quantum Temps**
  En s'inspirant des concepts issus des systèmes d'exploitation, une amélioration significative pourrait consister à introduire la notion de "quantum temps" pour chaque tâche. Chaque fois qu'une tâche est élue pour l'exécution, elle se voit attribuer un quantum de temps. Si la tâche n'utilise pas complètement ce quantum lors d'une exécution, le reste du quantum pourrait être attribué à d'autres tâches. Cela contribuerait à une utilisation plus équitable des ressources et éviterait la monopolisation excessive par une seule tâche.

Ces améliorations pourraient renforcer la robustesse et la sécurité de notre système de gestion de tâches, tout en s'inspirant des principes éprouvés de la gestion des ressources dans les systèmes d'exploitation. En incorporant ces mécanismes, le système deviendrait plus résilient face à des comportements potentiellement nuisibles tout en optimisant l'utilisation des ressources disponibles.

Conclusion

En conclusion, ce projet a démontré une approche efficace pour la résolution de problèmes complexes en utilisant des structures de données avancées et des techniques de traitement parallèle. L'utilisation de graphes dirigés acycliques, de threads et de méthodes de programmation dynamique a permis une gestion optimale des tâches interdépendantes, contribuant ainsi à l'efficacité globale du système. Ce projet a enrichi notre compréhension des techniques de parallélisation et de gestion des dépendances dans un environnement computationnel. En outre, il a constitué une opportunité précieuse pour approfondir nos connaissances en lien avec le cours, notamment en matière de création de graphes dirigés acycliques, renforçant ainsi notre maîtrise des concepts fondamentaux du domaine.