REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE
ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

Groupe : 2CS - SIQ1 **Année : 2024/2025**

TP1 HPC : Programmation parallèle avec Pthreads.

Simulation de la propagation de chaleur dans des structures matérielles

Réalisé par :

- FLISSI Loubna.
- LARBAOUI Yasmine Badr El Houda.

Encadré par :

Madame HAICHOUR Amina Selma.

Table des matières

Introduction

Avec l'augmentation des besoins en simulation numérique, il devient essentiel de développer des méthodes de calcul qui soient non seulement efficaces, mais aussi capables de gérer des volumes de données de plus en plus importants. Dans ce cadre, la parallélisation des algorithmes se présente comme une solution essentielle. En effet, cette technique permet d'exploiter pleinement la puissance de traitement des ordinateurs modernes équipés de plusieurs cœurs.

Ce TP se concentre sur l'application de la parallélisation via les threads (bibliothèque Pthreads) à un problème classique de la physique numérique : **la simulation de la propagation de la chaleur**.

L'objectif principal de ce travail est d'analyser comment cette approche de parallélisation peut améliorer les performances des calculs effectués lors de cette simulation.
Nous chercherons donc à déterminer dans quelle mesure l'utilisation de plusieurs threads peut réduire le temps nécessaire pour réaliser les calculs liés à la diffusion de la chaleur dans des matériaux, en permettant ainsi des simulations plus rapides et plus efficaces.

b. Fonctionnement de l'algorithme séquentiel de simulation de la propagation de chaleur :

Ce programme simule la diffusion de chaleur sur une grille bidimensionnelle de taille fixe. Dans cette approche, les structures matérielles sont modélisées sous forme de grilles, où chaque cellule représente une section de la structure et indique sa température à un instant donné.
Voici les différentes étapes de notre programme :

1. Initialisation de la Grille :

La température initiale de chaque cellule est générée aléatoirement. Cette initialisation permet de créer un état de départ varié pour la simulation, ce qui peut imiter des conditions réelles où les températures ne sont pas uniformes.

2. Calcul des Itérations :

À chaque itération, l'algorithme effectue les opérations suivantes :

- **Création d'une Nouvelle Grille :** Pour chaque cellule, la nouvelle température est calculée en faisant la moyenne des températures des cellules voisines (haut, bas, gauche, droite). Cela permet de simuler le transfert de chaleur entre les cellules.

- **Ajout d'une Composante Sinusoïdale :** Cette composante, basée sur la position de chaque cellule, simule des variations régulières de température, similaires aux changements au cours de la journée.

- **Modélisation de la Dissipation de Chaleur :** En combinant la moyenne des températures voisines avec la composante sinusoïdale, la formule de relaxation simule la diffusion et la dissipation de chaleur dans la grille, montrant ainsi l'évolution de la température dans le temps.

3. Mise à Jour de la Grille :

À chaque étape, la grille principale est mise à jour avec les nouvelles valeurs de température calculées. Cela signifie que l'état de la grille change progressivement, illustrant la manière dont les températures se stabilisent ou varient en fonction des calculs précédents.

4. Analyse de l'Évolution de la Température :

L'évolution des températures dans la grille révèle comment la chaleur se diffuse, permettant d'identifier les zones chaudes et froides et d'évaluer l'efficacité de la dissipation thermique.

En conclusion, ce programme permet ainsi d'explorer comment la température évolue dans la grille, offrant ainsi une simulation précise de la diffusion de chaleur. Cette approche contribue à une meilleure compréhension des principes thermiques et peut être appliquée à divers domaines d'étude, tels que la physique, l'ingénierie thermique, et l'environnement.

c. La modélisation algorithmique de la problématique choisie le pseudo-code :

```
Définir TAILLE = 1000  // Taille de la grille
Définir ITERATIONS = 1000  // Nombre d'itérations de la simulation

Début du programme
|       1. Initialiser grille et nouvelle_grille en tant que matrices de taille TAILLE x TAILLE.
|       2. Initialiser le générateur de nombres aléatoires avec l'heure actuelle.
|       3. Remplir chaque cellule de grille avec une valeur aléatoire entre 0 et 99.
|       4. Pour chaque itération de 0 à ITERATIONS - 1 :
|           |
|           |   Calculer les nouvelles valeurs de la grille :
|           |       |
|           |       |   Pour chaque cellule de grille (i, j), sauf les bords :
|           |       |       nouvelle_grille[i][j] = 0.25 * (grille[i-1][j] +
|           |       |                                       grille[i+1][j] +
|           |       |                                       grille[i][j-1] +
|           |       |                                       grille[i][j+1])+
|           |       |                                       10 * sin(i * 0.01) * cos(j * 0.01)
|           |       |
|           |   Mettre à jour grille avec les valeurs de nouvelle_grille
|           |
|           |                                   pour chaque cellule intérieure (i, j).
|           |   Afficher l'état de l'itération (numéro de l'itération terminée).
|           |
|       5. Libérer la mémoire allouée pour grille et nouvelle_grille.
Fin du programme
```

d. La pertinence de la parallélisation dans l'optimisation des simulations de chaleur :

- **Exploitation du parallélisme dans l'algorithme de simulation de chaleur :**

  La pertinence de la parallélisation de la simulation de chaleur réside dans la nature du problème, qui implique des calculs répétitifs et indépendants sur une grille de données, permettant une division efficace des tâches.
  À chaque itération, chaque cellule de la grille met à jour sa valeur en fonction de ses quatre voisines. Ce calcul est indépendant pour chaque cellule, ce qui constitue un parallélisme naturel dans le problème.

- **Identification des éléments de parallélisme :**

  L'algorithme séquentiel présente plusieurs caractéristiques favorables à la parallélisation :

- **Indépendance des calculs locaux:** Le calcul de la nouvelle température d'une cellule ne dépend que des valeurs de ses voisines à l'itération précédente.
- **Répétition de tâches identiques:** Le même calcul est répété pour chaque cellule à chaque itération.
- **Absence de dépendances temporelles fortes:** Les calculs d'une itération peuvent être effectués en parallèle, sous réserve d'une synchronisation appropriée.

  Ces éléments de parallélisme intrinsèque offrent de nombreuses opportunités pour accélérer les calculs en répartissant la charge de travail sur plusieurs cœurs de processeur.

- **Évaluation de la pertinence de la parallélisation :**

  La parallélisation de cet algorithme s'avère particulièrement pertinente pour les raisons suivantes :

- **Amélioration des performances:** En exploitant le parallélisme, on observe une accélération significative des calculs, notamment pour de grandes grilles et un nombre élevé d'itérations.
- **Scalabilité:** La parallélisation permet de tirer parti de l'augmentation de la puissance de calcul en ajoutant des cœurs de processeur.
- **Simplicité de mise en œuvre:** La répartition des itérations entre les threads est une approche relativement simple à mettre en œuvre.

e. La stratégie de parallélisation :

La stratégie de parallélisation adoptée dans ce code consiste à diviser le calcul de la propagation de la chaleur dans la grille entre plusieurs threads, permettant à chaque thread de prendre en charge un ensemble d'itérations indépendantes. Cette approche vise à réduire considérablement le temps d'exécution total en exploitant le parallélisme offert par les architectures multi-cœurs. Les dépendances entre les cellules voisines traitées par différents threads sont gérées par la barrière de synchronisation **pthread_barrier_wait** à la fin de chaque itération.

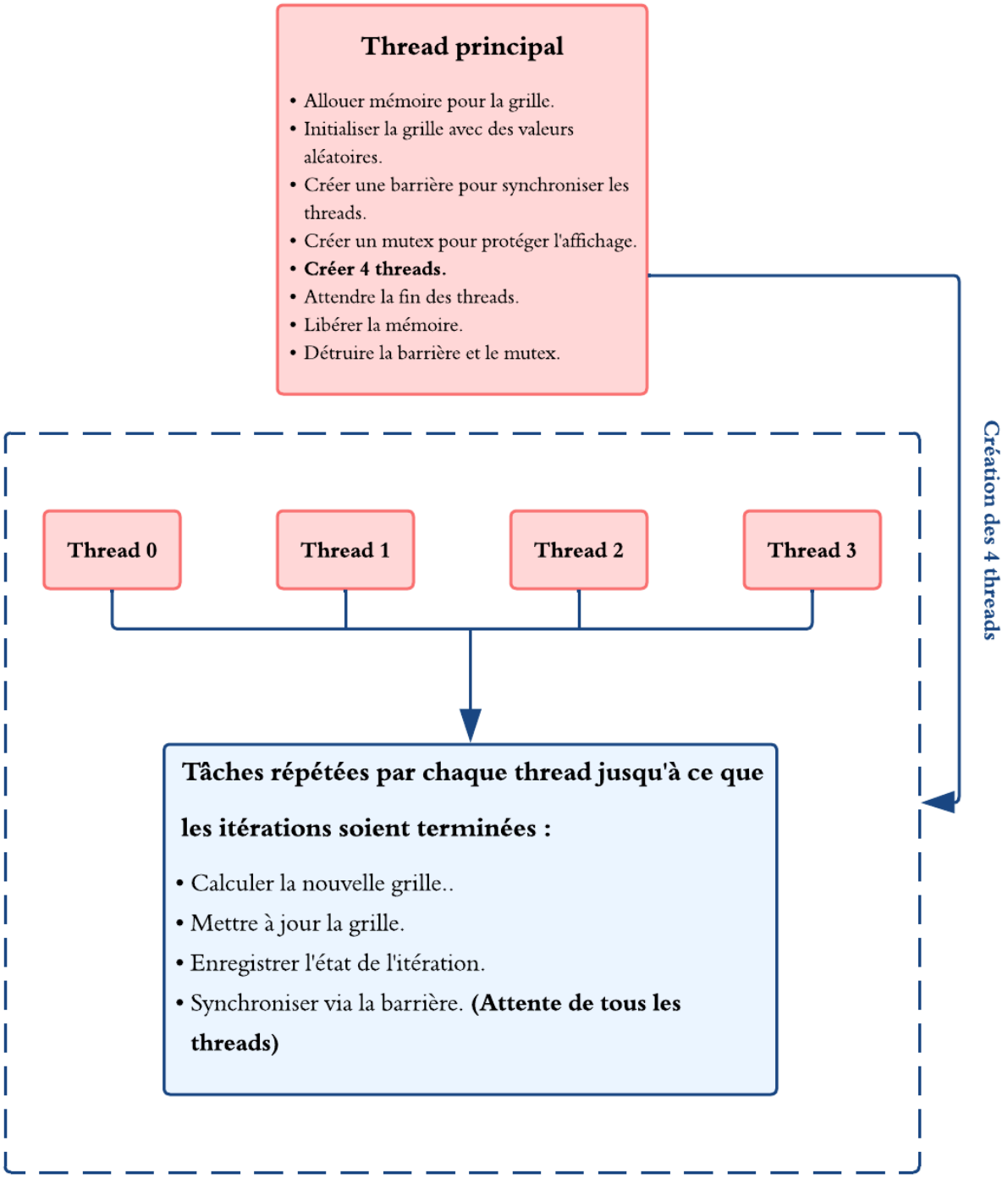
Les principaux hotspots de ce programme se situent dans la fonction **simuler_chaleur_partie**, précisément dans le calcul des valeurs de la nouvelle grille et la mise à jour de la grille principale. Ces étapes sont particulièrement critiques car elles sont très coûteuses en temps de calcul, et leur complexité quadratique $O(n^2)$ peut rapidement devenir un goulet d'étranglement lorsque la taille de la grille augmente, alors que le calcul de la chaleur dans une grille est un problème clairement parallèle.
En effet, chaque cellule peut être mise à jour de manière indépendante des autres, tant que les mises à jour n'impliquent pas des cellules voisines lors de la même itération. De plus, l'utilisation de plusieurs threads permet d'exploiter efficacement les ressources processeur disponibles et d'améliorer les performances globales, ce qui accélère les calculs sur de grandes grilles où une exécution séquentielle ne serait pas appropriée.

Le partitionnement dans notre solution parallèle utilisant la bibliothèque pthreads consiste à répartir les itérations entre les threads. Chaque thread reçoit un certain nombre d'itérations à traiter (nombre d'itérations / nombre de threads), ce qui permet de répartir équitablement la charge de travail. Cette méthode facilite la gestion des threads, car chaque thread travaille sur des sections différentes sans risquer de conflits d'accès aux données, tant qu'il respecte les limites de la grille. La synchronisation par la barrière garantit que tous les threads ont terminé leur tâche avant de mettre à jour la grille principale, ce qui assure la cohérence des données.

En conclusion, cette stratégie de parallélisation, basée sur une distribution efficace des calculs et une synchronisation précise, améliore la simulation de la propagation de la chaleur tout en préservant l'intégrité des données par l'utilisation des threads.

f. Représentation schématique de la solution parallèle :

Ce schéma montre la solution parallèle implémentée, en soulignant les tâches réalisées et les communications et synchronisations essentielles entre les threads.



h. Comparaison des Performances :

Voici un tableau comparant le temps d'exécution de la solution séquentielle et de la solution parallèle en fonction du nombre de threads et de la taille des données :

Nombre de threads	Taille de la grille	Temps d'exécution
Code séquentiel	100 x 100	`execution time : 2.043 s`
	1000 x 1000	`execution time : 102.809 s`
2 threads	100 x 100	`execution time : 2.211 s`
	1000 x 1000	`execution time : 54.234 s`
4 threads	100 x 100	`execution time : 2.081 s`
	1000 x 1000	`execution time : 31.838 s`
6 threads	100 x 100	`execution time : 2.054 s`
	1000 x 1000	`execution time : 25.937 s`
8 threads	100 x 100	`execution time : 2.085 s`
	1000 x 1000	`execution time : 23.010 s`
16 threads	100 x 100	`execution time : 1.996 s`
	1000 x 1000	`execution time : 21.973 s`
24 threads	100 x 100	`execution time : 1.968 s`
	1000 x 1000	`execution time : 20.922 s`
50 threads	100 x 100	`execution time : 2.016 s`
	1000 x 1000	`execution time : 21.524 s`

- **Comparaison par rapport à la taille de la grille :**

  Pour une grille de 100 x 100, le temps d'exécution reste à peu près le même, autour de 2 secondes, peu importe le nombre de threads utilisés. Cela indique qu'avec des grilles plus petites, le coût de gestion des threads n'est pas nécessaire, car le calcul est suffisamment rapide même en traitement séquentiel.
  En revanche, pour une grille de 1000 x 1000, les temps d'exécution augmentent de manière significative, passant de 102,809 secondes en mode séquentiel à des temps beaucoup plus réduits lorsque le nombre de threads augmente. Cela indique que les tâches parallèles utilisent de manière plus efficace la puissance de calcul disponible avec des grilles de plus grande taille.

- **Comparaison par rapport au nombre de threads :**

  L'utilisation de 2 à 8 threads montre une amélioration progressive du temps d'exécution pour la grille de 1000 x 1000. Par exemple, le temps d'exécution passe de 102.809 secondes (programme séquentiel) à 54.234 secondes (2 threads), et continue de diminuer jusqu'à 23.10 secondes (8 threads). Cela indique une bonne scalabilité de l'algorithme en parallèle. Cependant, lorsque le nombre de threads dépasse 8, les améliorations de performance commencent à diminuer. Par exemple, avec 16, 24 et 50 threads, le temps d'exécution pour la grille de 1000 x 1000 reste constant entre 20 et 22 secondes. Cela montre un effet de saturation, où le coût de gestion des threads (communication et synchronisation) annule les bénéfices de la parallélisation.

Les résultats montrent que la parallélisation est très efficace pour des tâches lourdes avec une grande taille de données. En utilisant plusieurs threads, le temps d'exécution pour des tâches complexes diminue de manière significative, bien qu'il existe un nombre maximum de threads où les améliorations en efficacité commencent à diminuer.