NOM / PRENOM_Etudiant 1 : ABOUD Ibrahim

NOM / PRENOM_Etudiant 2 : BOUYAKOUB Rayane

Groupe : SD1

TP N° 2
Programmation parallèle-OpenMP

Objectifs :

1. Identifier le parallélisme dans un code donné pour une exécution sur une architecture parallèle à mémoire partagée.
2. Utiliser effectivement les directives de **OpenMP** pour paralléliser un programme en vue d'une exécution sur une architecture parallèle à mémoire partagée.
3. Comparer les exécutions séquentielle et parallèle d'un même programme afin de déterminer l'accélération obtenue.

Outil : Langage C et API **OpenMP**.

L'avènement des nouvelles technologies de vision par ordinateur et de graphisme a nourri le besoin incessant d'effectuer des opérations de plus en plus complexes sur des matrices et ce en un minimum de temps. En effet, les matrices jouent un rôle crucial dans les graphiques. En partant du principe que toute image est représentée par une matrice où chaque chiffre représente l'intensité d'une certaine couleur à un certain point de grille, la reconnaissance d'images est ainsi réduite en grande partie, à des opérations matricielles telles que les inversions, les décompositions, etc. Néanmoins, le graphisme est loin d'être le seul domaine d'utilité des matrices, surtout avec l'ampleur qu'ont pris les Big Data qui sont une collection de vecteurs reliant plusieurs points de données. De nombreux algorithmes de classification et d'analyse de données reposent sur des systèmes linéaires et des opérations matricielles.

Ces raisons ont donc motivé le choix du problème à traiter qui s'oriente vers une opération matricielle (la décomposition Gaussienne d'une matrice).

La décomposition Gaussienne d'une matrice, dite échelonnement par ligne, est une méthode très utile pour la résolution des systèmes linéaires avec la méthode de Gauss. Elle est également utilisée lors de la factorisation LU d'un matrice qui est à son tour un outil puissant pour les algorithmes d'inversion de matrices et de résolution de systèmes complexes.

En algèbre linéaire, une matrice carrée est dite échelonnée en lignes, si elle est triangulaire supérieure, c'est-à-dire le nombre de zéros précédant la première valeur non nulle d'une ligne, augmente ligne par ligne jusqu'à ce qu'il ne reste éventuellement plus que des zéros. Toute matrice peut être transformée en sa matrice échelonnée réduite au moyen d'opérations élémentaires sur les lignes, qui sont :

- Permuter deux lignes.
- Multiplier une ligne par une constante non nulle.
- Ajouter à une ligne le multiple d'une autre ligne.

Parmi les algorithmes existants pour échelonner une matrice, il y a l'élimination de Gauss sans permutations, dont le pseudo code se présente comme suit :

Gauss

Pour i allant de 1 à n − 1, on effectue les calculs suivants :

| On ne change pas la i-ème ligne (qui est la ligne du pivot)

| Pour k allant de i + 1 à n :

| $L_{k,i} = U_{k,i} / U_{i,i}$

| Pour j allant de i + 1 à n :

| | | $U_{k,j} = U_{k,j} - L_{k,i}U_{i,j}$

| | Fin pour

| Fin pour

Fin Pour

Fin Gauss

La complexité algorithmique asymptotique de l'élimination de Gauss est $O(n^3)$, telle que $n \times n$ est la taille de la matrice et le nombre d'instructions à réaliser est proportionnel à n^3. Cet algorithme est généralement utilisé sur un ordinateur pour des systèmes avec des milliers d'inconnues et d'équations d'où la nécessité de paralléliser pour améliorer les performances.

Le document texte fourni comporte un code séquentiel écrit en langage C et formé de trois fonctions :

- **print_matrix()** affiche une matrice carrée d'ordre *size* passé en paramètre.
- **gaussian()** fait la transformation en matrice échelonnée réduite Gaussienne d'une matrice donnée en entrée d'ordre *size* suivant l'algorithme présenté plus haut.
- **random_fill()** remplit aléatoirement une matrice d'ordre *size* passé en paramètre.

1. Examinez ce code séquentiel et proposez trois versions parallèles en utilisant les directives OpenMP, tout en permettant une variation du nombre de threads.

- **Première version :** Les directives **#pragma omp parallel {}** et **#pragma omp for** doivent être intégrées dans la fonction **gaussian()**.
- **Deuxième version :** Intégrez la directive **#pragma omp parallel {}** dans la fonction **main()**, tandis que la directive **#pragma omp for** sera incluse dans la fonction **gaussian()**.
- **Troisième version :** Les directives **#pragma omp parallel {}** et **#pragma omp for** doivent toutes deux figurer dans la fonction **main()**.

Il est primordial de réaliser ces trois versions avec le minimum de modifications possibles.

2. Reporter les temps nécessaires aux exécutions séquentielles et parallèles pour différents nombres de threads et différentes valeurs de *N* (ordre de la matrice), comme suit :

N	100	250	500	710
Séquentiel	2 ms	30 ms	138 ms	355 ms
2 threads	1 ms	20 ms	125 ms	304 ms
4 threads	~0 ms	15 ms	70 ms	207 ms
8 threads	1 ms	24 ms	80 ms	196 ms
12 threads	2 ms	35 ms	110 ms	210 ms
16 threads	2 ms	39 ms	202 ms	387 ms

3. Discuter les résultats obtenus.

3.1. Impact de la taille des matrices sur les performances

Les résultats montrent une différence notable dans l'impact de la parallélisation en fonction de la taille de la matrice. Par exemple, pour une matrice de taille N=500, on observe une réduction du temps d'exécution significative entre l'exécution séquentielle (138 ms) et 4 threads (70 ms, soit une réduction de 50%). Cependant, pour une matrice plus petite (N=100), cette réduction est moins marquée, passant de 2 ms en séquentiel à environ 1 ms avec 4 threads.

Cette différence de performance s'observe également pour les autres tailles de matrices. En augmentant la taille de la matrice, l'impact de la parallélisation devient plus important. Cela s'explique par plusieurs facteurs :

- Un meilleur ratio entre le temps de calcul et le surcoût de la parallélisation.
- Une utilisation plus optimale des ressources de calcul avec une charge plus importante.
- Une meilleure répartition de la charge entre les threads.

Ces observations montrent que la parallélisation devient plus efficace lorsque la taille de la matrice (et donc la quantité de données à traiter) augmente.

3.2. Dégradation des performances et limitations matérielles

On observe une dégradation des performances au-delà de 8 à 12 threads, qui peut être expliquée par plusieurs facteurs. La machine de test dispose de 4 cœurs physiques, ce qui offre un maximum de 8 threads logiques au total.

- Limitations matérielles et planification : La machine ne disposant que de 8 threads logiques, toute utilisation au-delà nécessite une planification par le système d'exploitation. Cela oblige certains threads à attendre leur tour dans une file d'attente, diminuant ainsi les gains de parallélisation.

- Surcoût de gestion : Avec un nombre de threads supérieur à 8, le système doit gérer des changements de contexte fréquents, ce qui consomme des ressources supplémentaires et réduit les performances globales.

- Autres facteurs de dégradation : Plus le nombre de threads augmente, plus le coût lié à leur gestion (création, synchronisation et destruction) s'élève, entraînant une dégradation des performances et limitant le gain en accélération apporté par la parallélisation.

3.3. Nombre optimal de threads

En fonction de l'architecture de la machine (8 threads logiques disponibles), les résultats montrent qu'un nombre optimal de threads se situe entre 4 et 8 :

- À 4 threads : On obtient un excellent compromis pour les matrices moyennes (N = 250), avec un temps de 15 ms, soit une réduction de plus de 50% par rapport au temps séquentiel (30 ms).

- À 8 threads : L'accélération est maximale pour certaines tailles de matrices, notamment pour N=500, mais l'efficacité diminue pour les plus petites matrices.

- Au-delà de 8 threads : L'augmentation du nombre de threads au-delà de la limite matérielle de 8 threads logiques provoque une dégradation des performances, le temps d'exécution ayant tendance à augmenter (par exemple, 16 threads pour N=500 augmente à 202 ms par rapport à 80 ms pour 8 threads).