Introduction à openMP

Rapport de TP Par Sébastien Hervieu , 1er Octobre 2017

# Exercice 1 : Multiplication Matricielle

## Ecriture du programme mono thread :

Le programme est implémenté en C++.

Il est constitué d’une classe « Matrix » qui permet de générer une matrice aléatoire étant donné un nombre de lignes et un nombre de colonne. Cette classe comporte aussi une méthode statique multiply() pour effectuer la multiplication de matrices.

Cette classe est mise en œuvre par la fonction main, pour créer deux matrices aléatoires les multiplier.

## Insertion des directives openMP

Une directive « #pragma omp for » est ajoutée sur la boucle for() la plus externe de la multiplication. Cela permet de répartir cette boucle entre les différents thread.

## Mesure des temps d’exécution

Les mesures de temps ont été effectuée en ajoutant des appels à la fonction omp\_get\_wtime() autour de la directive « #pragma omp for ». Toutes les mesures indiquées ci-dessous sont la moyenne de 10 mesures effectuées.

### Mode schedule(static)

Après avoir ajouté au code original les directives openMP, nous avons mesuré le temps d’exécution de la multiplication de matrices avec un mode schedule(static) pour la boucle for la plus extérieure. Nous mesurons sur une machine ayant 4 cœurs physiques, en partant de 1 thread jusqu’à 4 threads, en multipliant 2 matrices carrées de taille croissante (100 x 100, 200 x 200 puis 400 x 400).

Le résultat de cette première mesure est représenté dans la .

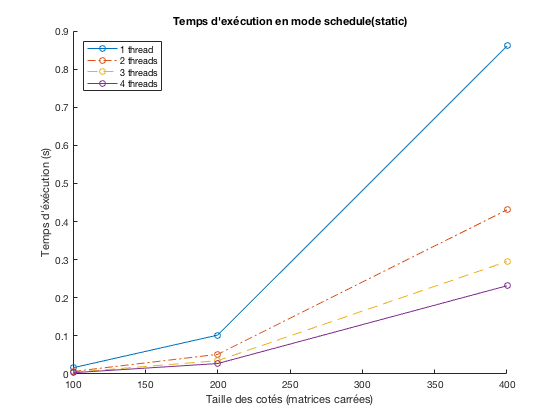


Figure ‑ Temps d'exécution en mode schedule(static)

Nous constatons tout de suite que le temps d’exécution est corrélé avec la taille des matrices : plus les matrices à multiplier sont de grande taille, plus le temps d’exécution est grand.

Nous constatons ensuite que le multithreading donne des temps d’exécution globalement plus courts qu’en single thread. Par exemple pour la multiplication de 2 matrices de 400 x 400, les temps en fonction du nombre de threads sont les suivants :

|  |  |  |
| --- | --- | --- |
| Nombre de threads | Temps d’exécution (s) | Gain par rapport à 1 thread |
| 1 | 0.86 | NA |
| 2 | 0.43 | 2 |
| 3 | 0.3 | 2,86 |
| 4 | 0.23 | 3,34 |

### Mesure avec différentes chunksize en schedule(static)

Nous effectuons ensuite des mesures pour évaluer l’effet d’un changement de valeur de la taille du chunk en mode schedule(static) de la directive « #pragma omp for ». Pour cela nous fixons la taille des matrices à 400 x 400.

Le résultat de ces mesures est indiqué par la .

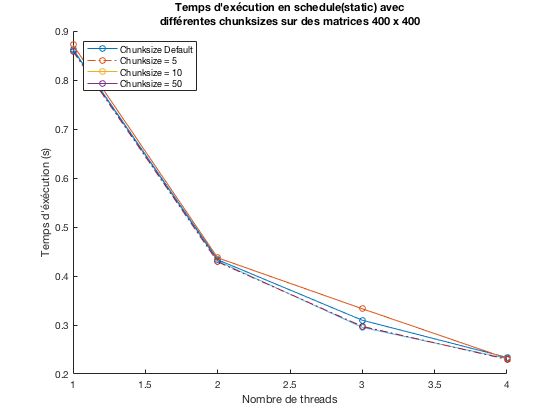


Figure ‑ Temps d'exécution en schedule(static)avec différentes valeurs de chunksize

Nous constatons que dans le mode schedule(static), la taille du chunk n’a aucun pratiquement impact sur le temps d’exécution de la multiplication de matrices.

### Mesures du temps d’exécution avec différentes chunksize en mode schedule(dynamic)

Nous effectuons maintenant la même mesure que précédemment (toujours avec des matrices 400 x 400) en mode schedule(dynamic) de la directive « #pragma omp for », pour différentes valeurs du chunksize.

Le résultat est affiché dans la .

Nous constatons encore que la taille du chunksize n’a pas d’influence sur le temps d’exécution de la multiplication avec cette taille de matrice.

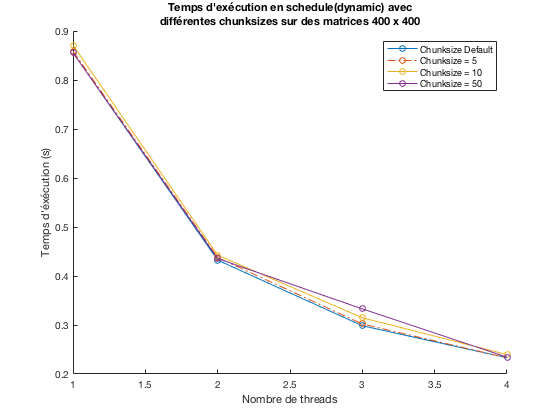


Figure ‑ Temps d'exécution en mode schedule(dynamic) pour chunksize variable (matrices 400 x 400)

Si nous effectuons des mesures similaires avec des matrices de taille 100 x 100, nous obtenons les résultats indiqués par la .

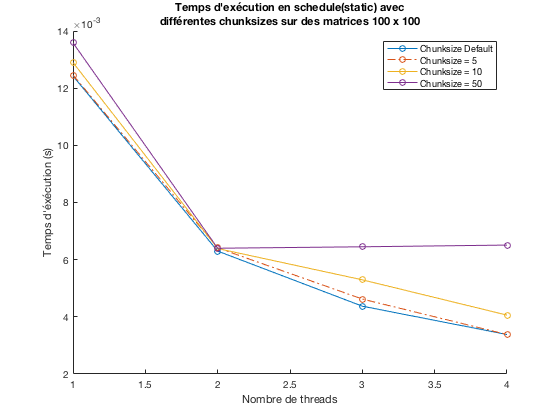


Figure ‑ Temps d'exécution en mode schedule(dynamic) pour chunksize variable (matrices 100 x 100)

Nous constatons donc que dans cette configuration que pour un chunksize de 50, il n’y plus de gain de performance au de-là de 2 threads.

Cela est du au fait que la boucle for() instrumentée par openMP contient 100 entrées. Si le chunksize est de 50, cela veut dire que pour 100 données le système n’aura que 2 chunks à attribuer à 4 threads, et que seuls 2 threads seront actifs.

En conclusion, en utilisant la directive « #pragma omp for », il faut veiller à ce que le chunksize soit correctement réglé par rapport au nombre de boucles à parallèliser. Si le chunksize est trop grand, un ou plusieurs threads n’auront aucun chunk à traiter et ne pourront donc améliorer la performance.

# Exercice 2 : Recherche de nombres premiers

## Ecriture du programme mono thread

Le programme est implémenté en C++.

Tout le programme est dans la fonction main().

Ce programme prend en argument un nombre entier A et vérifie de manière naïve si les entiers N < A sont premiers ou non. Si N est premier, on incrémente une variable globale « foundPrimes ».

On vérifie si un entier N est premier ou non en effectuant une boucle while() en calculant le reste de la division l’entier N par tous les entiers inférieurs. Si ce reste est égal à 0 pour l’un des entiers, N n’est pas premier.

## Insertion des directives openMP

La parallèlisation de ce programme est effectuée en ajoutant un directive « #pragma omp for » autour de la boucle for() la plus externe, qui parcourt les entiers a compris entre 2 et A.

Pour assurer que la variable « foundPrimes » va bien retourner la somme des nombre d’entiers trouvés par chaque thread, nous complétons la directive par une instruction « reduction(+, foundPrimes).

Lorsque nous exécutons le programme ainsi parallèlisé, nous constatons que les résultats sont similaires à ceux du programme mono thread.

## Mesure du temps d’execution

Les mesures de temps ont été effectuée en ajoutant des appels à la fonction omp\_get\_wtime() autour de la directive « #pragma omp for ». Toutes les mesures indiquées ci-dessous sont la moyenne de 10 mesures effectuées.

### Mesures avec schedule(static) chunksize par defaut.

Nous commençons en mesurant le temps d’exécution de la boucle for() pour diverses valeurs de A, avec un mode schedule(static, default).

Le résultat de cette mesure est indiqué par la .

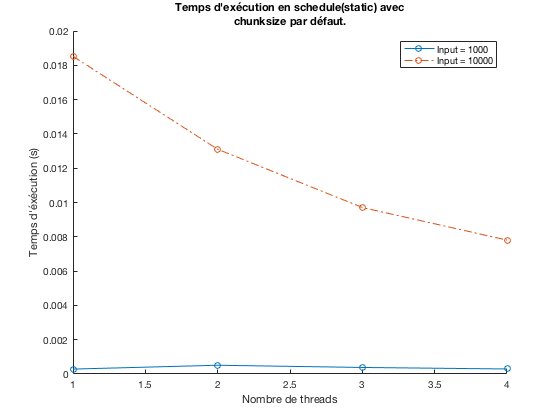


Figure ‑ Temps d'exécution en mode schedule(static,default)

Nous constatons que le gain en performance est réduit par rapport au nombre de threads. Par exemple le gain avec une entrée égale à 10000 n’est que d’environ 2,25 pour 4 threads.

Cela veut dire pour cette configuration que 2 threads sur les 4 ne font pratiquement pas de traitement.

### Mesures en chunksize variable en schedule static

Nous effectuons ensuite une mesure toujours en mode schedule(static) mais pour une seule valeur de A en entrée (10000) et pour des chunksizes de différentes valeurs.

Le graphe résultant de cette mesure est affiché dans la .

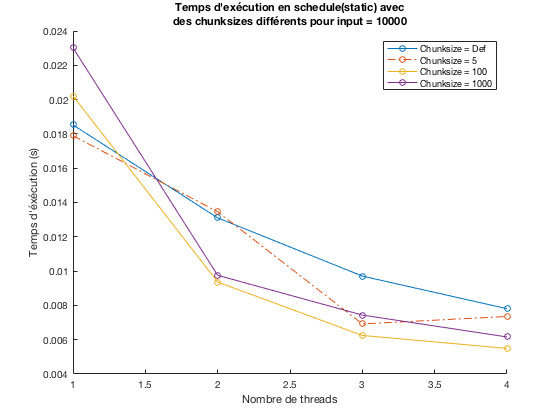


Figure ‑ Temps d'exécution en mode schedule(static) pour différentes chunksizes

Nous constatons que pour certains chunksizes, le gain en performance est légèrement meilleur (jusqu’à 3,1 pour 4 threads avec chunksize = 100), mais que globalement le gain est plutôt minime.

### Mesure avec schedule(dynamic) avec des chunksize variables

Nous effectuons enfin une mesure en mode schedule(dynamic) pour une seule valeur de A en entrée (10000) et pour des chunksizes de différentes valeurs.

Le graphe résultant de cette mesure est affiché dans la .

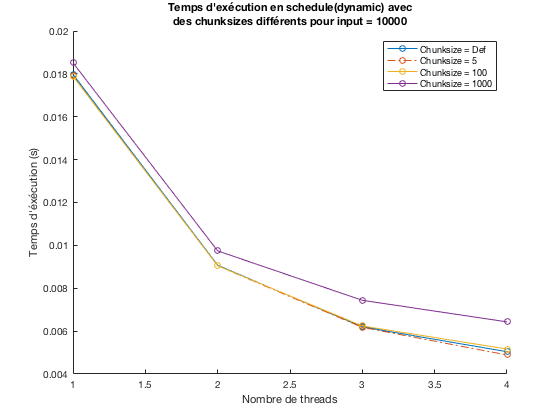


Figure ‑ Temps d'exécution en mode schedule(dynamic) et diverses valeurs de chunksize

Nous constatons dans ce cas que pour des chunksizes plutôt petits, le gain de performance est légèrement améliorée (x 3,1 en 4 threads et pour chunksize 1, 5 et 100).

## Analyse et conclusion

L’algorithme naïf de recherche de nombre premier est caractérisé par une très grande variabilité de la durée de la vérification.

Par exemple, il sera très rapide de vérifier qu’un nombre pair n’est pas premier (1 boucle while()) alors qu’un nombre effectivement premier prendra beaucoup plus de temps.

Ensuite, la boucle commence systématiquement par des recherches sur les nombres les plus petits. Un thread qui a cette part de traitement static traite ce chunk beaucoup plus rapidement que les autres threads, qui auront des threads plus lourds.

Donc une allocation statique des chunk sur cet algorithme n’est pas efficace car il n’est pas possible d’anticiper la durée d’une boucle, si on considère que les nombres premiers sont répartis de manière inconnue sur l’intervalle considéré.

Il est bien plus efficace d’utiliser le mode schedule(dynamic) : lorsqu’un thread a fini un chunk, il attend généralement moins que dans le cas static pour recevoir un autre chunk.

# Exercice 3 : Histogramme

## Ecriture du programme mono thread

Le programme est écrit en C++ et est implémenté entièrement dans la fonction main().

Le tableau de données est initialisé pour est initialisé automatiquement avec des valeurs float comprises entre 0 et 10.

Un tableau de 10 compteurs « histogram[] » est ensuite initialisé avec des valeurs zéro ; le tableau de donnée est ensuite parcouru dans son entièreté par une boucle for() qui incrémente la valeur de histogram[] en fonction de la valeur courante du tableau de donnée.

## Insertion des directives openMP

La parallèlisation de l’algorithme est effectuée par l’ajout d’une directive « #pragma omp for » autour de la boucle for la plus externe.

Il faut cependant protégé l’accès en écriture du tableau « histogram[] » au moment de l’incrémentation de l’index à chaque itération pour éviter que les threads tentent de mettre à jour le même index en même temps, ce qui résulterait en un perte de donnée.

## Mesure des temps d’exécution

Nous commençons par mesurer le temps d’exécution de l’algorithme histogramme sur un tableau de données de taille 10000 x 10000 en **mono thread**. Le temps d’exécution dans ce cas est de **0,67 seconde**.

### Mesure avec directive « atomic »

Nous implémentons ensuite l’algorithme multi thread en insérant les directives openMP comme décrit ci-dessus, et en protégeant l’accès au tableau « histogram[] » par une directive « #pragma omp atomic »

Le temps d’exécution en mode schedule(static) avec 4 threads et les mêmes données que précédemment est dans ce cas de **2,18 secondes**. Il est de **5,6 secondes** si le mode est schedule(dynamic).

Nous constatons que la performance du programme est considérablement dégradée avec l’utilisation de la directive « #pragma omp atomic ». En effet, vu que tous les threads doivent incrémenter constamment l’un des index du tableau « histogram[] », il y a constamment au moins un thread qui possède la section « atomique ». Dans ce les trois autres threads ne peuvent accéder à cette section et sont donc bloqués.

Il ne faut donc pas utiliser la directive « atomic » avec ce programme.

### Modification du programme avec des tableaux privés

Nous modifions donc le programme de manière à ce que chaque thread dispose de son propre tableau « privateHisto[] », lesquels sont ensuite additionnés dans « histogram[] » après la fin de la boucle for().

Le temps d’exécution dans ce cas est de **0,181 seconde**, pour un tableau de donnée de 10000 x 10000, et un mode schedule(static).

## Conclusion

La directive « atomic » doit être utilisée avec beaucoup de précaution, uniquement lorsque le temps de traitement d’une boucle est très supérieur au temps passé dans la directive « atomic ».

# Exercice 4 : Tri par transposition impaire-paire

## Ecriture du programme mono thread

Le programme est écrit en C++. Il est constitué d’une fonction sortArray() qui implémente l’algorithme de tri par transposition impaire-paire. Cette fonction est invoqué directement par la fonction main(), après avoir crée le tableau avec des valeurs aléatoires. Une fonction checkArraySorted() est ensuite invoquée pour vérifier que le tableau est bien trié.

## Insertion des directives openMP

L’algorithme de tri par transposition impaire-paire est divisé en deux phases distinctes : la phase impaire et la phase pair. Chacune de ces phases inclut une boucle for() qui va parcourir le tableau pour effectuer la comparaison / transposition.

Une directive « #pragma omp for » est ajouté à l’intérieur des phases impaire-pair, à l’extérieur de la boucle for().

## Mesure des temps d’exécution

Les mesures de temps ont été effectuée en ajoutant des appels à la fonction omp\_get\_wtime() autour de l’appel à la fonction sortArray(). Toutes les mesures indiquées ci-dessous sont la moyenne de 10 mesures effectuées.

### Mesures avec mode schedule(static,default)

Nous commençons par mesurer le temps d’exécution de l’application avec différentes tailles de listes à trier, en mode schedule(static,default).

Le résultat de ces mesures est affiché .

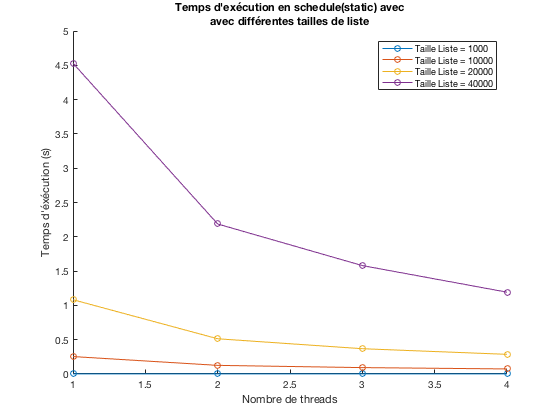


Figure ‑ Temps d'exécution en mode schedule(static, default)

Nous constatons que la performance augmente généralement avec le nombre de threads. Pour 4 threads, avec une liste de 40000 entrées, le gain est de performance est de 4,5 / 1,5 = 3.

### Mesure en mode schedule(dynamic, default)

Nous recommençons la mesure précédente, mais en utilisant cette fois-ci le mode schedule(dynamic,default).

Le résultat de cette mesure est montré dans la .

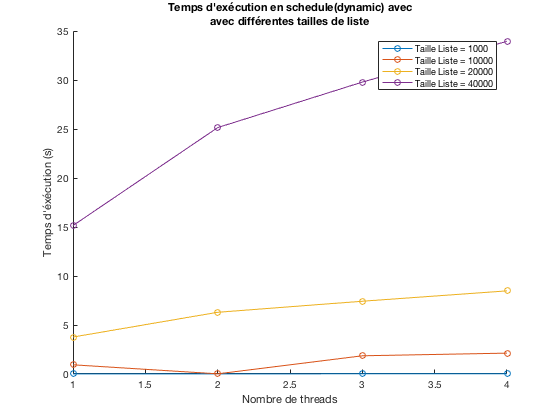


Figure ‑ Temps d'exécution en mode schedule(dynamic,default)

Nous constatons ici que la performance est dégradée en multithread par rapport au monothread.

## Analyse et conclusion

La dégradation des performances en mode schedule(dynamic, default) est expliqué par le fait que la boucle for() parallèlisée ne consiste qu’en une comparaison et une inversion de deux valeurs, ce qui est très rapide à executer.

En mode schedule(dynamic, default), le chunksize est de 1, ce qui veut dire qu’à chaque itération chaque thread va exécuter l’algorithme d’allocation dynamique de chunks de openMP, ce qui représente un overhead non négligeable.

Dans le cas présent où le traitement d’une itération est très court, l’overhead du mode schedule(dynamic, default) est mis en valeur.

Il faut donc veiller à ce que la durée de traitement d’un chunk en mode dynamic soit suffisamment longue pour éviter qu’un overhead considérable apparaissent.

**Modification de l’algorithme lorsque la liste est de taille impaire :**

Oui il faut modifier l’algorithme dans ce cas. Il faut veiller à ce que les indexes en bout de liste ne sortent pas du buffer, et il faut ajouter une itération supplémentaire par rapport à la taille de la liste.

# Exercice 5 : Dynamique moléculaire

## Analyse du programme originel mono thread

Le programme originel md.cpp est composé principalement de deux parties :

* une classe « Particle » qui fournit les fonctionnalités nécessaires au programme pour effectuer les calculs d’énergie cinétique, de changement de vecteur position, de changement de vecteur vitesse et de changement de force
* la fonction main(), qui met en œuvre la classe « Particle » pour initialiser les particules et les faire évoluer au sein d’une boucle while(){}.

La majeur partie de la simulation est implémentée dans cette boucle while() :

- une boucle for(){} parcourt les particules une à une et calcule son interaction avec toutes les autres particules. L’état global du système (énergie potentielle totale et énergie cinétique totale) est représenté par les deux variables « ke » et « pe », qui sont cumulée lors du calcul de l’interaction de chaque particule avec toutes les autres.

## Insertion des directives openMP

Les directives openMP sont ajoutées dans la boucle while() :

* l’essentiel de la parallèlisation est effectuée par une directive « #pragma omp for » pour répartir le calcul des interactions de chaque particule entre tous les threads.
* l’initialisation de « ke » et « pe » est effectuée uniquement par le thread master.
* Chaque thread dispose de ses propres variables privées « private\_ke » et « private\_pe » pour cumuler l’énergie cinétique et l’énergie potentielle pour les particules qui leur sont allouées.
* Ces variables privées sont enfin additionnées au sein d’une section critique pour reconstituer « ke » et « pe ».

## Mesure des temps d’exécution

Après avoir compilé md.cpp,et en exécutant le programme sans modification le résultat est le suivant :

18015619@me041005:~/Documents/master2/ProgParallele/introOpenMP/05\_dynmoleculaire> ./mdsingle

Time t = 0.00000 pe = 1823511.31853 ke = 0.00000 total = 1823511.31853

Time t = 0.01000 pe = 1823421.25986 ke = 90.05546 total = 1823511.31531

Time t = 0.02000 pe = 1823145.88474 ke = 365.39093 total = 1823511.27567

Time t = 0.03000 pe = 1822674.71990 ke = 836.44495 total = 1823511.16485

Time t = 0.04000 pe = 1821989.41249 ke = 1521.52846 total = 1823510.94096

Time t = 0.05000 pe = 1821073.06136 ke = 2437.49925 total = 1823510.56061

Time taken was 41.46307 seconds

Après avoir inséré les directives openMP, après avoir compilé et en exécutant le programme sur une machine 4 cœurs, nous obtenons le résultat suivant :

> g++ -fopenmp mdparallelized.cpp -O4 -o mdparallel

> ./mdparallel

Time t = 0.00000 pe = 1823511.31853 ke = 0.00000 total = 1823511.31853

Time t = 0.01000 pe = 1823421.25986 ke = 90.05546 total = 1823511.31531

Time t = 0.02000 pe = 1823145.88474 ke = 365.39093 total = 1823511.27567

Time t = 0.03000 pe = 1822674.71990 ke = 836.44495 total = 1823511.16485

Time t = 0.04000 pe = 1821989.41249 ke = 1521.52846 total = 1823510.94096

Time t = 0.05000 pe = 1821073.06136 ke = 2437.49925 total = 1823510.56061

Time taken was 11.39883 seconds

Le gain en performance est donc de 41,46 / 11,4 = 3,64.