Projet OpenMP : Programmation parallèle

Réalisé par :
Iyed Cheberli
Ilyess ElAjroud

Année scolaire : 2018/2019

INTRODUCTION

OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est prise en charge par de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement.

OpenMP est portable et dimensionnable. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel.

Pour mettre en œuvre ce que nous avons étudié pendant le cours de la programmation parallèle, il nous est demandé d'implémenter un programme qui permet de faire le tri en parallèl d'une une très grande base de données. Pour cela on a supposé que cette base de données est constituée d'une liste de nombres réels stockés dans N tableaux (blocs) de taille fixe K. Il s'agit donc en total de N*K nombres réels. On va se baser sur trois grandes fonctions : tri_merge, une méthode de tri de notre choix et une fonction qui va générer la base de données. On a mis à notre disposition un algorithme de tri parallel qu'on va essayer d'implémenter et qui va faire appel à ces trois grandes fonctions.

L'algorithme en question est le suivant :

Entrées :

- N : nombre de tableaux (blocs)
- K : taille des blocs
- B1, B2, … , BN : un ensemble de N tableaux (blocs) non-triés de taille K

Sorties :

- B1, B2, … , BN : un ensemble de N tableaux (blocs) triés de taille K

Pour i=1 à N faire en parallèle

- tri (Entrée : Bi, Sortie : Bi)

Fin pour i

Pour j=1 à N-1 faire

- k=1+ (j%2) // traitement des blocs deux à deux
- Pour i=0 à N/2-1 faire en parallèle
    - b1=1+ (k+2*i)%N
    - b2=1+(K+2*i+1)%N
    - min=min (b1, b2)
    - max=max (b1, b2)
    - tri_merge (Entrées : Bmin, Bmax, Sorties : Bmin, Bmax)
- Fin pour i
- Fin pour j

Sommaire

Table des figures

1- Répartition de travail :

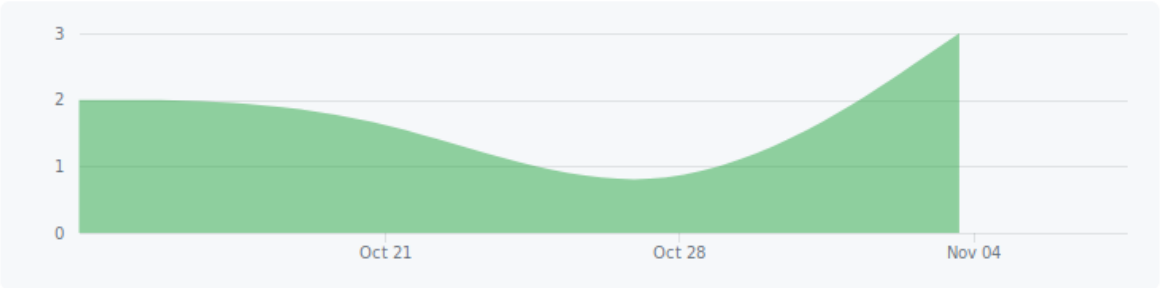
- **Présentation de git :**

GitHub est une plateforme open source de gestion de versions et de collaboration destinée aux développeurs de logiciels.

Git permet de stocker le code source d'un projet et de suivre l'historique complet de toutes les modifications apportées à ce code. Grâce aux outils qu'il fournit, Git permet de gérer les conflits éventuels résultant des changements apportés par plusieurs développeurs, GitHub facilite la programmation collaborative en mettant une interface Web à disposition du référentiel de code de Git, ainsi que des outils d'administration favorisant la collaboration.

Le travail était basé principalement sur git . vous pouvez voir ci-dessous la courbe d'évolution de notre travail :

Il y a une baisse de travail dû à la période de révision .



Commits on Nov 8, 2018

ajout du fichier de partage de rapport
ilyesajroud committed 2 hours ago

Correction de tous les bugs et ajout de la fonctionnalité de création...
Chabroul committed 3 hours ago

implémentation du parallélisme et correction de Bug
Chabroul committed 4 hours ago

Commits on Oct 21, 2018

ajout fonction tri-merge sans parallélisme
ilyesajroud committed 19 days ago

ajout fonction tri-merge (sans parallelisme)
ilyesajroud committed 19 days ago

Commits on Oct 16, 2018

Debut du projet
ilyesajroud committed 23 days ago

Initial commit
ilyesajroud committed 23 days ago

Version finale et stable
Chabroul committed 5 hours ago

Amélioration au niveau du parallél avec ajout de schedule
Chabroul committed 5 hours ago

Changement de la version de tri_merge
Chabroul committed 21 hours ago

Commits on Nov 9, 2018

Amélioration de tri_merge
Chabroul committed 2 days ago

Merge branch 'master' of https://github.com/ilyesajroud/omp
Chabroul committed 2 days ago

2- Stratégie d'implémentation de tri_merge :

Afin de minimiser la complexité en temps et en espace de notre programme nous avons optés pour une stratégie qui fait en sorte qu'on fait le tri au fur et à mesure du traitement des blocs pour qu'on ne fait pas appel à la fonction de tri fusion qui a une complexité de n log (n) .

Principe :

On a créé un tableau intermédiaire qui va contenir les données de bin1 et bin2 triées. Puisque bin1 et bin2 sont déjà triées on va comparer à chaque fois bin1[i] et bin2[j] et on met dans le tableau intermédiaire la valeur la plus petite de tel sorte qu'on aura finalement dans le tableau intermédiaire les valeurs triées. Finalement on fait un parcours du tableau intermédiaire pour remettre dans bin1 les K premières valeurs petites et dans bin2 les K dernières valeurs plus grandes que bin2

Illustration :

1 3 6 7 **2** 4 5 8
-> **2 > 1** on insère 1 dans Tab_inter et on incrémente le compteur de bin1

Tab_inter = {1}
3 6 7 **2** 4 5 8

-> 2<3 on insère 2 dans Tab_inter et on incrémente le compteur de bin2
Tab_inter = {1,2}

3 6 7 **4** 5 8

4 > 3 on insère 3 dans Tab_inter et on incrémente le compteur de bin1
Tab_inter = {1,2,3}

6 7 **4** 5 8

4 < 6 on insère 4 dans Tab_inter et on incrémente le compteur de bin2

Tab_inter = {1,2,3,4}

6 7 **5** 8

……. jusqu'à compteur de bin1 et bin2 soit égale à K (taille des deux tableaux)

Finalement dans Tab_inter on aura la combinaison de bin1 et bin2 triées. Donc maintenant on doit juste remettre dans bin1 les K premières valeurs et dans bin2 les K dernières valeurs et retourner bin1 et bin2.

Avec cette stratégie on a pu trancher les valeurs des tableaux en entrée pour satisfaire le besoin du projet SANS faire appel à une fonction de Tri .

Tri_merge n'est qu'une partie de notre projet .La suite du projet comporte une fonction de tri ainsi que la fonction qui permet de paralléliser le processus de tri de blocs N*K . Ci dessous un exemple d'exécution de notre programme pour N= 100 et K= 100

En effet, notre programme nous demande d'abord le nombre de threads qui vont faire le travaille et affiche le temps d'exécution de la partie parallèle et grâce à une fonction qu'on a implémenté vérifie si les blocs ils sont bien triés ou non.

```
chabroul@chabroul-K55VD:~/Bureau/Projet Parallél/omp$ make
rm -f omp
echo "compiling"
compiling
gcc -g -Wall -fopenmp omp.c -o omp
chmod u+x omp
chabroul@chabroul-K55VD:~/Bureau/Projet Parallél/omp$ ./omp 100 100
Base de données d'entiers de 100 x 100 éléments
Entrez le nombre de threads qui vont exécuté le tri parallél
4
Generation du tableau de bloc  FINI
TRI parallélFINI
////////////////////////////////////////////////////////
Le temps d'execution du tri parallél est 0.023318
////////////////////////////////////////////////////////
k= 10000
Tri parallél fonctionnel à 100/100
 chabroul@chabroul-K55VD:~/Bureau/Projet Parallél/omp$
```

3- Tests et courbes :

Tout nos test et courbes ont été réalisés sur un ordinateur avec les caractéristiques techniques ci dessous :

Product: Intel(R) Pentium(R) CPU B950 @ 2.10GHz

Mémoire : 8 Gb

Nombre de cœurs : 2

Os : Ubuntu 18.04 64 bits

N	K	time	Nombre de threads	Charge des threads	Charges des cœurs
5	5	0,00103	5	0,000206	0,000515
10	10	0,001549	5	0,0003098	0,0007745
15	15	0,001769	5	0,0003538	0,0008845
50	50	0,008044	5	0,0016088	0,004022
100	100	0,035916	5	0,0071832	0,017958
200	200	0,162921	5	0,0325842	0,0814605
300	300	0,484706	5	0,0969412	0,242353
400	400	1,142695	5	0,228539	0,5713475
500	500	2,244723	5	0,4489446	1,1223615
1000	1000	18,749779	5	3,7499558	9,3748895
1200	1200	37,445171	5	7,4890342	18,7225855
1300	1300	97,888885	5	19,577777	48,9444425

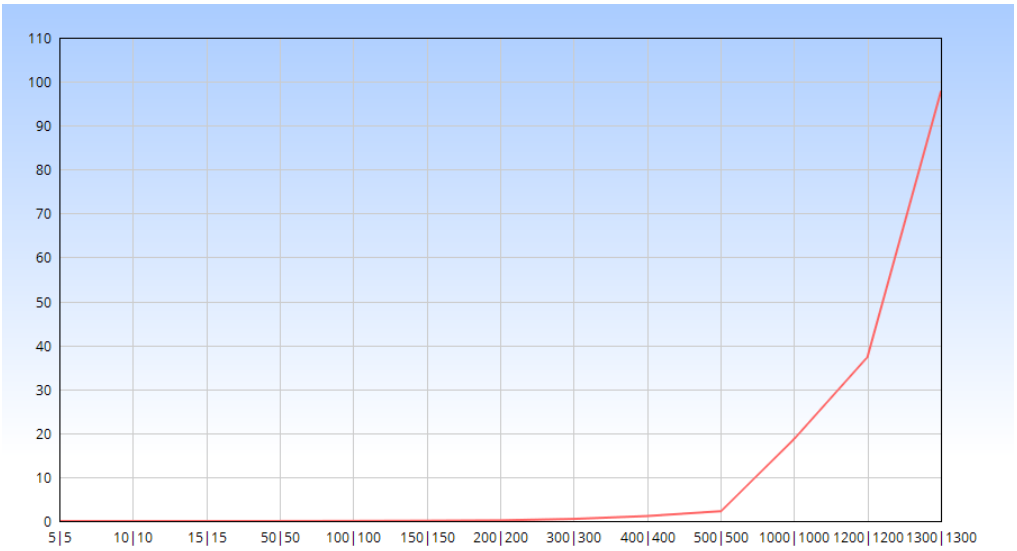


Figure 1 Temps d'exécution en fonction de taille total de données (N*K)

- **Interprétation :**

Lors de notre premier test , nous avons voulu voir le comportement du processeur avec une large quantité de donnés . Nous avons choisi un nombre de threads fixe égale à 5 et nous avons varier et N et K de tel sorte a augmenté considérablement le nombre de cases traité. Comme vous pouvez le remarquer , au dessous de 800 tableaux de 800 cases , le processeur gère trés bien la quantité de données imposée avec un temps d'exécution qui ne dépasse pas la barre des 10 secondes .

Pour 1000 tableaux de 1000 cases , Le temps d'exécution met quasiment 2 fois le temps d'exécution de 500 *500 .

Pour 1690000 cases(1300*1300) , le temps d'exécution explose avec plus 90 secondes avec 100 % de charge sur le processeur bi-cœur .

- **Conclusion :**

La taille des données a un effet crucial sur le temps d'exécution.

Test 2 :

N	K	time	Nombre de threads	Charge des threads	Charges des cœurs
10000	10	21,006252	2	10,503126	10,503126
10000	10	23,085171	5	11,542585	4,6170342
10000	10	23,864788	10	11,932394	2,3864788
10000	10	23,901779	15	11,95089	1,593451933
10000	10	23,972765	20	11,986382	1,19863825
10000	10	24,415558	30	12,207779	0,813851933
10000	10	24,695694	40	12,347847	0,61739235
10000	10	25,319275	50	12,659637	0,5063855

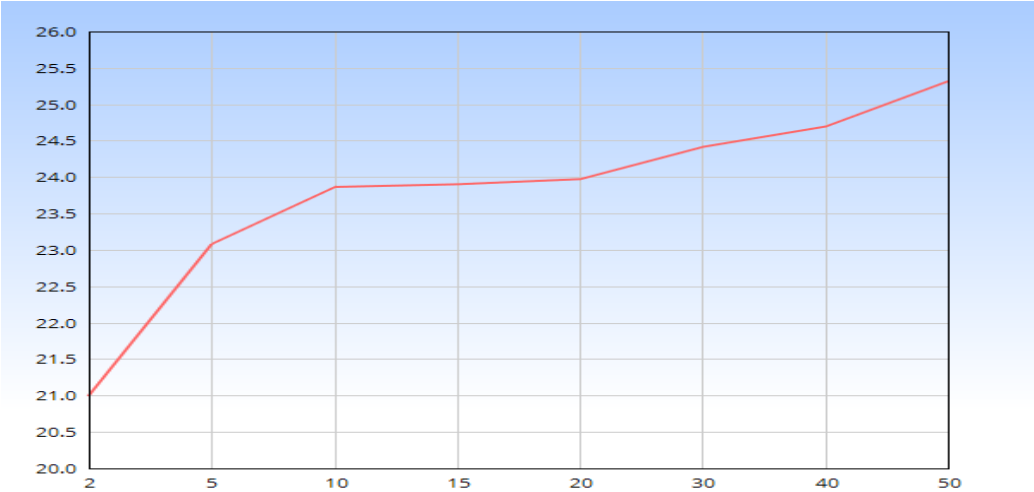


Figure 2 Pour N*K fixé, la variation du temps d'exécution en fonction de nombre de Threads

- **Interprétation :**

Lors de notre deuxième test nous avons travaillé avec 10 000 tableaux de 10 cases ce qui est équivalent à traiter (N*K)=10*10000=100000 cases.

Avec une variation du nombre de thread entre 2 et 50, nous remarquons d'exécution augmente en augmentons les thread .

Contrairement à toute attente, l'augmentation du nombre de threads n'implique pas une augmentation de performance mais une détérioration assez importante. Dans notre cas , notre processeur bi-cœurs est 20 % plus performant sur 2 threads que sur 50 threads . En effet , le processeur met plus de temps à créer les threads , à partager le travail entre les threads , à les ordonnancer , attendre la fin des exécutions ,faire la réduction de tous les résultats et enfin terminer tous les threads .

- **Conclusion :**

L'augmentation du nombre de threads n'implique pas nécessairement l'amélioration de performances.

Test 3 :

N	K	time	nombre de threads	charge des cœurs	charge des threads
4	125	0,000587	2	0,000293	0,0002935
10	50	0,000745	2	0,000372	0,0003725
2	250	0,000817	2	0,000408	0,0004085
125	4	0,005888	2	0,002944	0,002944
5	100	0,006042	2	0,003021	0,003021
20	25	0,007633	2	0,003817	0,0038165
100	5	0,007916	2	0,003958	0,003958

1	500	0,010316	2	0,005158	0,005158
50	10	0,015721	2	0,007861	0,0078605
250	2	0,018628	2	0,009314	0,009314
500	1	0,040233	2	0,020117	0,0201165

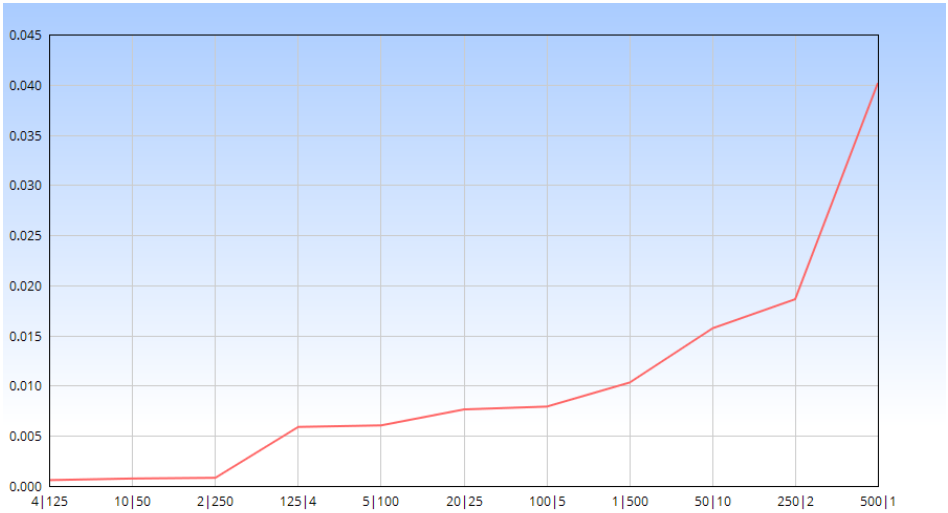


Figure 3 Pour N*K fixé, la variation du temps d'exécution en fonction N et K

- **Interprétation :**

Afin de voir l'influence des variables N et K indépendamment l'une de l'autre sur le temps d'exécution , nous avons fixé le nombre de cases à 500 et le nombre de threads à 2 .La courbe nous montre qu'en augmentant le nombre tableaux (N) ,le temps d'exécution augmente .

Ceci s'explique par la stratégie choisie pour la fonction Tri_merge .En effet , nous avons choisi le tri fusion pour sa complexité assez réduite et son optimisation pour les tableaux de grande taille .Néanmoins chaque exécution de fonction coute un temps précieux.

- **Conclusion :**

Dans notre cas , l'algorithme est optimal pour des tableaux de grande tailles (N petit et K grand) et n'est pas optimal pour un grand nombre de tableaux avec une taille réduite (N grand et K petit) .

CONCLUSION

Tout au long de ce projet nous avons été confronté à plusieurs problèmes liées à la programmation parallèle (Multi-threading). La protection des variables partagées ainsi que l'accès aux tableaux était l'un des problèmes majeure.

OpenMP nous offre des méthodes prédéfinis pour protéger les variables sensibles .la clause #pragma omp critical nous permet de mettre en place des sections critiques dans les blocs à traiter par les threads.

Grâce aux différents tests effectués lors de ce projet, nous pouvons conclure que les performances du Multi-Thread dépend de l'architecture des processeurs et principalement le nombre de coeurs . De ce fait, nous pouvons conclure que le multi-threading est assez performant par rapport à l'exécution séquentielle néanmoins il a des limites matérielle et logiciels .La taille des données à traiter ont un rôle très important pour mesurer les performances du multi-threading