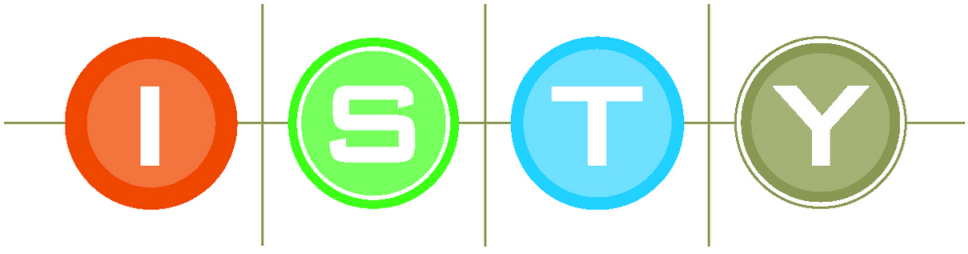
Tri parallèle d'un tableau avec OpenMP

Pierre AYOUB – Océane FLAMANT

10 novembre 2018



INSTITUT DES SCIENCES ET TECHNIQUES DES YVELINES

Table des matières

1 Introduction

Durant la réalisation de ce projet, nous avions comme objectifs, dans un premier temps, d'implémenter en langage C un algorithme de tri parallèle destiné à trier une grande base de données. Dans un second temps, il nous fallait effectuer des mesures du temps d'exécution de notre programme en fonction de plusieurs paramètres, tels que le nombre de threads utilisés ou encore la taille des données.

Plusieurs contraintes nous étaient imposées :

- Le squelette de l'algorithme à suivre était donné dans l'énoncé ;
- Nous avions à utiliser la bibliothèque de programmation parallèle OpenMP pour paralléliser notre algorithme ;
- Nous avions trois fonctions principales à implémenter de la manière suivante :

  generator() : remplit un tableau de taille K avec des nombres réels aléatoires ;

  tri() : trie un tableau de taille K ;

  tri-merge() : à partir de deux tableaux chacun de taille K et triés, renvoie deux tableaux triés vérifiant que toutes les valeurs du premier tableau sont inférieures à celles du deuxième.

2 Développement du programme

Dans cette partie, nous allons expliquer et justifier les choix que nous avons fait afin d'implémenter l'algorithme demandé, ainsi que les modifications qui ont dû être effectuées.

2.1 Structure du programme

Nos fonctions *generator()* et *tri()* suivent scrupuleusement l'algorithme défini dans l'énoncé. Cependant, nous avions le libre arbitre concernant l'implémentation de la fonction *tri()*.

Pour l'algorithme de tri, nous avons choisi le *tri à bulles* (connu aussi sous le nom de *tri par permutation* ou encore *tri par propagation*). Nous reviendrons sur ces performances plus tard, mais nous pouvons souligner le fait que c'est un tri simple à implémenter et qu'il fait partie de la famille des tris en place : il n'a pas besoin de recopier un tableau annexe pour que le tri s'effectue.

2.2 Adaptation de l'algorithme

Nous avons rencontré quelques déconvenues avec l'algorithme donné. En effet, une fois nos fonctions réalisées et vérifiées, nous avons lancé le programme et le résultat obtenu n'était pas le bon. Les valeurs contenues dans les tableaux étaient triés mais les tableaux n'étais pas dans l'odre attendu. Nous avons donc dû effectuer une légère adaptation : en effet, en C, les indices commencent à 0 et non à 1, comme donné dans l'algorithme.

Tout d'abord, chaque *boucle for* commence à 0 et s'arrête à $N-1$. Ensuite nous avons supprimé les $+1$ des variables k, $b1$ et $b2$ car les boucles commencent à 0. Enfin, pour $b2$, le K est devenu k (ce dernier étant une correction d'une erreur de l'énoncé).

En résumé :
— $k = 1 + (j \bmod 2) \Rightarrow k = (j \bmod 2)$
— $b_1 = 1 + (k + 2 * i) \bmod N \Rightarrow b_1 = (k + 2 * i) \bmod N$
— $b_2 = 1 + (K + 2 * i + 1) \bmod N \Rightarrow b_2 = (k + 2 * i + 1) \bmod N$

3 Performances

Nous allons maintenant discuter des performances de notre programme. D'un point de vue algorithmique, dû à notre choix de *tri à bulle*, les performances ne devraient pas être excellentes. En effet, le *tri à bulle* a une complexité en moyenne en $\mathcal{O}(n^2)$, ce qui est bien moins bon qu'un algorithme tel que le *tri rapide* ayant une complexité en moyenne en $\mathcal{O}(n.\log n)$. Cependant, étant donné que nous sommes dans une projet de programmation parallèle et non d'algorithmique, il y a deux raisons qui justifient le choix du *tri à bulle* :

— Une « mauvaise » complexité algorithmique nous permettra de mieux mettre en évidence l'importance du parallélisme ;
— L'algorithme parcours de multiples fois le tableau sur toute sa longueur, ainsi cela va rendre possible la mise en avant des défauts de cache.

3.1 Mesures

Avant de commenter nos mesures, il nous semble bon d'expliquer un minimum ce qui va suivre. Notre programme voit son comportement modifié par 4 paramètres :

Taille de la base de données : correspond au nombre de tableaux multiplié par la taille d'un tableau ($N * K$). Il faut multiplier ce nombre par la taille d'un réel double précision pour l'exprimer en octet ;

N : nombre de tableaux ;

K : taille d'un tableau ;

OMP_NUM_THREADS : nombre de threads exécutant le programme.

Il est important de se rendre compte que si $N <$ OMP_NUM_THREADS, alors les OMP_NUM_THREADS $- N$ threads ne pourront pas obtenir de travail.

De tous les paramètres des trois graphiques suivants, rien n'as été choisi au hasard. Par défaut :

— OMP_NUM_THREADS est égal aux nombre de cœurs de la machine hôte ;

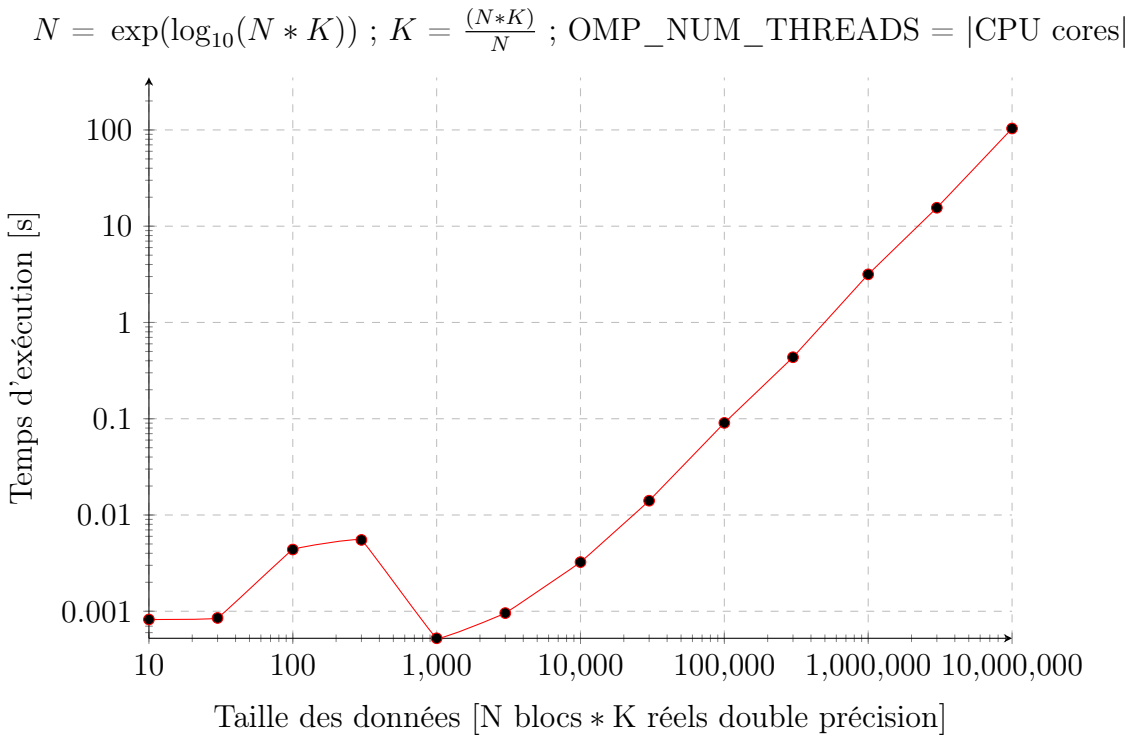


FIGURE 1 – Temps d'exécution en fonction de la taille des données

— N est calculé à partir d'une formule déterminée empiriquement, permettant d'approcher la valeur optimale de N pour un $N * K$ donné après une série de mesures sur une de nos machines ;
— K prend sa valeur en fonction de N.

Enfin, vous noterez que les échelles de nos graphiques sont des échelles logarithmiques, avec un facteur 10 entre chaque graduation.

Comme nous pouvons le constater sur la Figure 1, le temps d'exécution varie énormément en fonction de la taille des données, malgré que les 3 paramètres présentés ci-dessus aient été choisis de manière optimale. En dessous d'une taille de 100 000 nombres, le temps d'exécution est négligeable pour un utilisateur. Cependant, à partir d'une taille d'environ 30 000 nombres, nous dépassons définitivement le seuil des 0,01s : cela s'explique par le fait que notre base de données ne tient plus dans nos mémoires caches L_1 et L_2. De plus, à partir de ce seuil là se créé une droite de fonction linéaire, avec un

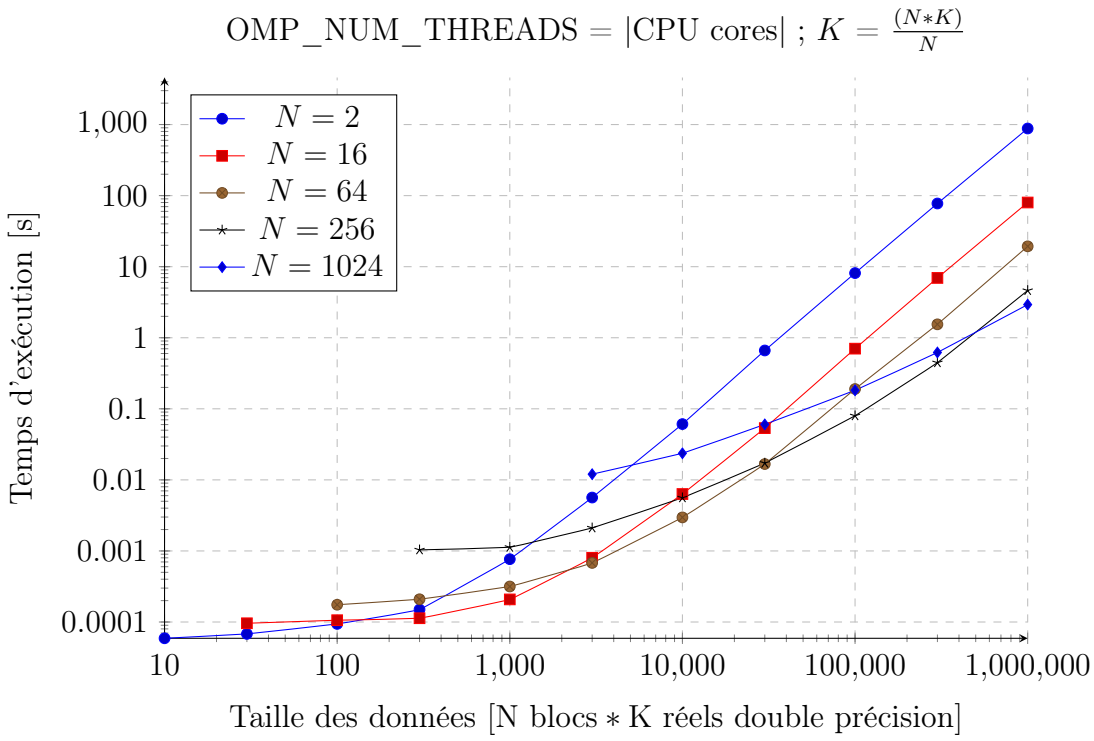


FIGURE 2 – Temps d'exécution en fonction de la taille des données et du nombre de blocs

coefficient directeur environ égal à 1.5. Cela paraît peu, mais cela n'est pas à négliger puisque nous sommes sur des échelles logarithmiques : c'est-à-dire que lorsque l'on multiplie la taille de nos données par 10, le temps d'exécution est environ multiplié par 30. Le temps d'exécution va donc croître très rapidement. Par exemple, pour 1 000 000 de nombres, nous étions à 3s d'exécution. A 10 000 000, nous étions maintenant à 100s. Nous pourrions largement imaginer qu'à 100 000 000, nous serions à 3000s.

La Figure 2 nous permet de mesurer l'influence du paramètre N sur le temps d'exécution de notre programme. Notez tout d'abord que si $N > N*K$, alors le test ne peut pas avoir lieu : nous avons donc des courbes qui ne commencent pas à une taille de 10. Nous observons que pour des petites tailles de base de données (< 100), un N petit est optimal. Cependant, un N petit montre très vite ses limites et est dépassé par des valeurs de N supérieures.

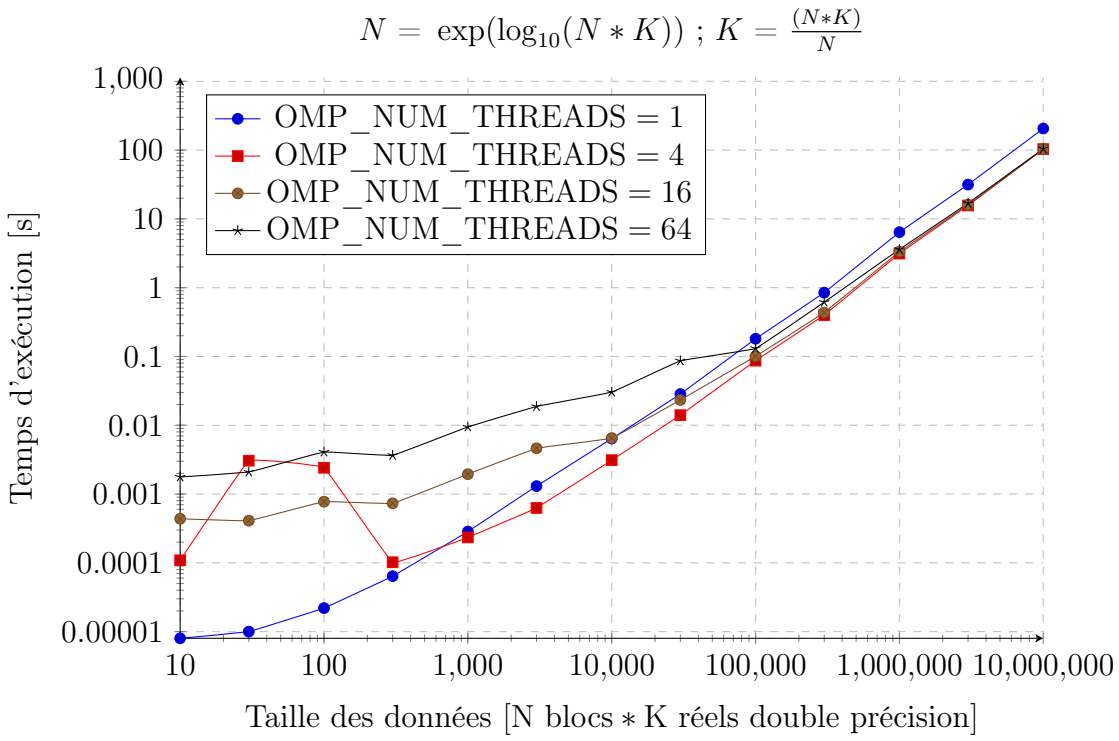


FIGURE 3 – Temps d'exécution en fonction de la taille des données et du nombre de threads

En revanche, les grands N, dû aux changements constant de tableau en cours de traitement, sont nettement moins adaptés pour des petites tailles de base de données (le temps d'exécution peut être multiplé par 10 par rapport à un N petit), mais devient très vite optimal pour de grandes bases de données ($> 100\,000$). Les différences les plus extrêmes de notre graphique se situent à une taille de base de données de $1\,000\,000$, où un $N = 2$ donne un temps d'exécution de 1 000s, tandis qu'un $N = 1024$ donne un temps d'exécution de 3s : c'est une différence d'un facteur d'environ 333 !

La Figure 3 nous informe de la différence du temps d'exécution en fonction du nombre de threads se chargeant du tri. Comme attendu, un grand nombre de threads sur un petit nombre de données augmente le temps d'exécution dû aux nombreux changements de contextes inutiles. Cependant, avoir plusieurs threads devient bénéfique à partir d'une taille comprise entre 1 000

et 10 000 nombres. En revanche, nous pouvons remarquer que le temps d'exécution est similaire entre 4 et 64 threads à partir d'une taille d'1 000 000 de données : la cause étant que la machine ne disposant que de 4 cœurs. Si nous avions un CPU disposant de plus de cœurs, alors nous aurions vu le temps d'exécution continuer de diminuer en augmentant le nombre de threads. La différence entre 1 et 4 threads pour 1 000 000 de nombres est tout de même non-négligeable, passant respectivement d'un temps d'exécution de 200s à 100s. Nous pourrions nous attendre à un facteur 4, mais nous n'observons qu'un facteur 2 : l'origine de ce phénomène vient de *l'HyperThreading*, exposant au système d'exploitation 4 cœurs logiques tandis que le CPU ne possède seulement que 2 cœurs physiques.

3.2 Perspectives d'amélioration

Plusieurs moyens peuvent être mis en œuvre pour améliorer les performances de notre programme :

- Augmenter le degré de parallélisme : paralléliser de nouvelles boucles, découper notre tableau sur de nouveaux niveaux.
- Améliorer la gestion de la mémoire cache : effectuer des benchmarks pour déterminer la taille précise à partir de laquelle nos *cache miss* augmentent de manière notable.
- Améliorer la complexité algorithmique : utiliser un algorihtme de tri plus efficace (tel que le *tri fusion*, le *tri rapide* ou encore le *tri par tas*).

4 Conclusion

Les mesures de performance de notre programme auront été une tâche intéréssante, car plusieurs problématiques ont été introduites pour y répondre convenablement. Certains choix n'auront pas été de toute simplicité, tel que le choix de l'échelle des axes, du nombre de graphiques, des échantillons représentatifs, etc. De plus, nous avons créé un script shell permettant de recréer et/ou modifier nos expériences de manière très simple.

Concernant la partie analyse, il a été instructif de comprendre les correlations entre les différences de performance induites par chaque paramètre. Nous pouvons aussi noter à quel point, sur une machine donnée, un paramètre peut-être bien plus important qu'un autre (ici, N est bien plus influant que OMP_NUM_THREADS).

Nous pouvons donc en conclure que pour obtenir un programme performant, il faut à la fois une étude algorithmique poussée pour obtenir la meilleure complexité, mais aussi prendre en compte les caractéristiques de l'algorithme qui pourraient influencer le matériel.