Machine Parallèle

Calculs des décimales de π multi-threadé

Etienne PENAULT 17805598

Semestre 5

2019-2020

Introduction:

Le but de ce Projet est de trouver un moyen pour calculer les N-ièmes décimales de π de manière optimale et multi-threadée.
Pour cela, je me suis renseigné à propos de la formule qui m'a été conseillé par le professeur: Bailey-Borwein-Plouffe ou autrement appelé BBP.

Le principe de cette formule repose sur une somme, c'est à dire qu'il faudra appliquer la formule autant de fois que de décimales souhaitées et ainsi additionner tous les résultats afin d'arriver à la valeurs de π, avec la précision des décimals voulues.

Cette formule est assez intéressante à utiliser, car contrairement au crible d'Eratosthène utilisé sur le TP du calcul des N-ièmes nombres premiers, nous n'avons pas besoin du terme précédent pour calculer le suivant. Cela implique que nous pouvons calculer chaque valeurs, indépendamment les unes des autres, et les additionner toute à la fin afin de retomber sur la valeur de π voulue.

La formule que nous allons utiliser est la suivante :

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

Optimisation:

Hors threading, nous pouvons utiliser une optimisation qui peut nous faire gagner jusqu'au double du temps de calcul sur chaque élément de la somme. Pour cela, au lieu de calculer $\frac{1}{16^k}$ lors de chaque itération, nous opterons pour une autre manière de calculer cette fraction.

La façon de calculer revient à appliquer la puissance x^k à l'ensemble de la fraction tel que $\left(\frac{1}{16}\right)^k$ et non uniquement au nombre 16.

Par ce procédé, nous calculons directement la valeur finale de la fraction et ne calculons pas 16^k qui donne un résultat vraiment grand et qui alourdit le calcul et par conséquent la vitesse de calcul et la vitesse d'exécution de notre programme.

Le but de cette optimisation est donc de passer par les nombres les plus petits possibles.

$\forall\ k \in [0 : \infty]$

$$\left(\frac{1}{16}\right)^k < 16^k$$

Notre nouvelle formule à appliquer est donc:

$$\pi = \sum_{k=0}^{\infty} \left[\left(\frac{1}{16}\right)^k \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right) \right]$$

Thread:

Pour pouvoir multi-threader notre calcul de π, j'ai opté pour la technique suivante: Lancer le calcul en faisant commencer chaque thread par son numéro de thread, c'est à dire à quelle valeur k doit être assignée au début pour tel thread... (par exemple nous avons 4 threads, le premier thread va commencer par calculer la formule avec $k = 0$, le second avec $k = 1$, le troisième avec $k = 2$, et le quatrième avec $k = 3$).

Chaque thread se voit attribuer le même décalage (le nombre total de thread) afin de scinder le travail. Un thread, au lieu d'avancer d'un en un lors de chaque itérations, va donc avancer de notre nombre total de thread. Par ce biais, cette méthodes nous garanti que la formule sera appliquée $\forall\ k \in [0 : l]$ avec l notre limite, et que chaque valeur de k sera utilisée une seule fois.

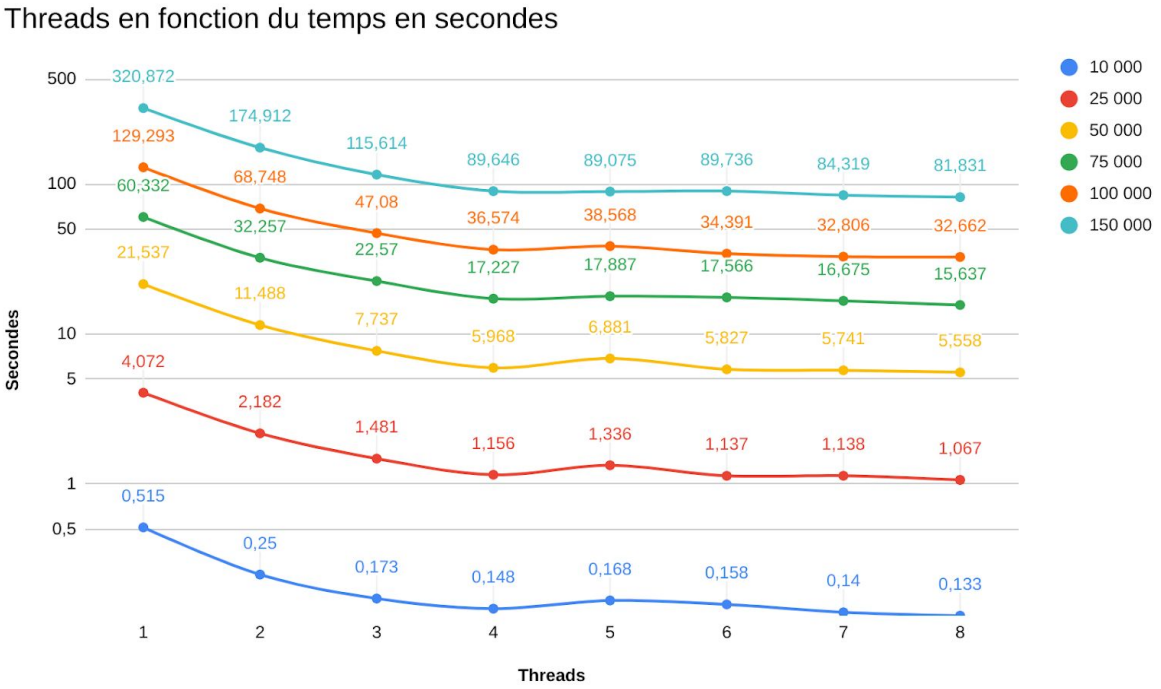
De plus, afin de stocker les résultats des calculs d'un thread, chaque thread va additionner au fur et à mesure tous ses résultats indépendamment des résultats des autres threads.
En effet, il faut séparer les résultats de chaque thread et non pas additionner tous les résultats de tous les thread dans une seule et unique variable, sinon cela fausserait les résultats de la valeur de π à cause d'une data race.

Pour palier à ça, nous créons au préalable un vecteur de résultats où nous stockerons les résultats de chaque thread. Chaque thread calcule donc son résultat dans le même tableau, mais à un index différent, qui sera tout simplement le numéro du thread actuel (qui est aussi notre valeur de départ pour k, par exemple nous avons 4 threads, le premier thread va utiliser l'index 0, le second l'index 1, le troisième l'index 2, et le quatrième l'index 3).

Une fois tous les threads terminés, nous pouvons additionner les résultats de tous les threads afin d'avoir notre valeur de π à la décimale souhaitée et ensuite nous écrivons le résultat dans un fichier :).

Tableau/Graphique des performances:

Nombre de décimales/ thread	10 000	25 000	50 000	75 000	100 000	150 000
1	0,515s	4,072s	21,537s	1m0,332s	2m9,293s	5m20,872s
2	0,250s	2,182s	11,488s	32,257s	1m8,748s	2m54,912s
3	0,173s	1,481s	7,737s	22,570s	47,080s	1m55,614s
4	0,148s	1,156s	5,968s	17,227s	36,574s	1m29,646s
5	0,168s	1,336s	6,881s	17,887s	38,568s	1m29,075s
6	0,158s	1,137s	5,827s	17,566s	34,391s	1m29,736s
7	0,140s	1,138s	5,741s	16,675s	32,806s	1m24,319s
8	0,133s	1,067s	5,558s	15,637s	32,662s	1m21,831s



Avec ces données interprétées sous formes de tableau et de graphique, nous pouvons en conclure que le gain de temps maximal se trouve lors de l'utilisation de 4 threads (mon nombre de coeurs physique), ou encore lors de l'utilisation de 8 threads (ou plus...). Nous remarquons ce phénomène sur tous les nombres de décimales maximales saisies, cela semble universel.