Travaux Pratique 1

Contexte : Les données indiquées dans ce rapport ont été obtenues avec un Processeur 6 cœurs et 12 processeurs logiques.

Exercice 1 :

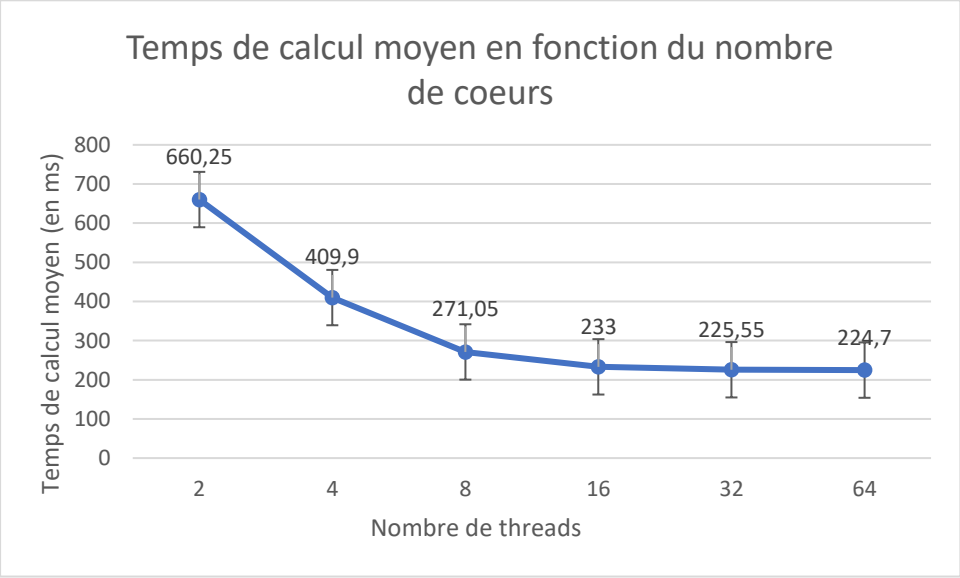
Il y a 3 threads : le thread principal, le thread first qui exécute foo et le thread second qui exécute bar. Les deux derniers sont lancés par le thread principal.

Exercice 2 :

Globalement, plus on augmente le nombre de threads, plus le calcul est effectué rapidement.

Pour analyser cela, j'ai fait des statistiques avec un échantillon de 20 exécutions pour chaque nombre de threads.

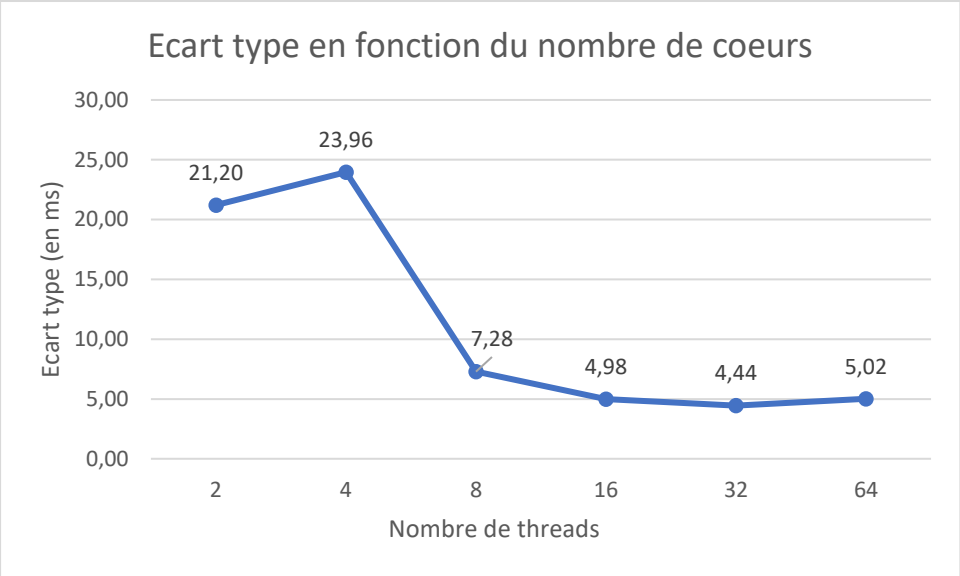
On le voit sur le graphe 1 suivant :



Graphe 1 : Temps de calcul moyen en fonction du nombre de cœurs

On constate que le temps moyen diminue en fonction du nombre de threads. Cependant, on atteint assez rapidement un palier vers 225 ms. J'émet l'hypothèse que ce palier est atteint car on a atteint un nombre de threads supérieur ou égal au nombre de processeurs logiques et donc on ne peut pas gagner plus de temps en augmentant encore le nombre de threads (ou très peu).

En analysant l'écart type sur le graphe 2, on constate aussi que celui-ci diminue en fonction du nombre de threads lancés. Donc avec plus de threads, le temps de calcul est plus stable. Même si par rapport à la moyenne, cette différence est assez faible. En effet, en calculant le coefficient de variation ($\frac{Ecart\ type \times 100}{Moyenne}$), on obtient 3% à 2 threads, 5% à 4 threads et environ 2% pour 8 threads et plus.



Graphe 2 : Ecart type en fonction du nombre de cœurs

Exercice 3 :

En utilisant une unique variable globale avec un mutex pour contrôler son accès, j'obtiens des résultats similaires à l'exercice 2 que ce soit en temps de calcul ou en écart-type.

Exercice 4 :

J'ai créé une classe MoniteurPi qui a un mutex et un double en attribut pour stocker le résultat. L'accès en lecture et en écriture est contrôlé par le mutex. Ce mutex est déclaré comme un pointeur dans la classe MoniteurPi. Cela permet de déclarer des accesseurs constants comme le getResult, et pouvoir quand même modifier l'état du mutex. Donc en plus du constructeur et du destructeur, on a un accesseur « getResult » qui encapsule la section critique pour l'accès en lecture et un mutateur « add » qui encapsule la section critique pour l'accès en écriture. Dans ce cas, on est obligé de faire un mutateur qui prend en paramètres l'incrément et pas la nouvelle valeur, car l'addition doit se faire dans la même section critique que la lecture de la valeur courante. Si on la faisait en dehors, alors

Ici, une méthode pour l'écriture et une méthode pour la lecture suffise car nous n'avons pas besoin de gérer des écritures et des lectures concurrentes. Seules les écritures sont concurrentes, contrairement au patron lecteur/rédacteur. La lecture n'a lieu qu'à la fin du programme, quand tous les threads se sont exécutés, par le thread principal.

Exercice 5 :

Implantation

J'utilise donc deux moniteurs pour réaliser cet exerice. MoniteurIntervalle me permet de garder l'accès à la valeur courante en section critique. Cette valeur est donnée aux threads pour qu'il effectue le calcul des nombres premiers jumeaux dessus. Elle est incrémentée dans la section critique à chaque fois qu'on veut récupérer la valeur. Le maximum peut ne pas être en section critique car il n'est jamais modifié et est même déclaré constant. Comme c'est entier et pas un pointeur sur un objet, cela ne pose pas de problème. (Après expérimentation, j'ai pu constater que je

ne gagnais pas de temps d'exécution ou très peu. C'est logique car il n'est récupéré qu'une fois par thread.)

Mon deuxième moniteur, MoniteurPremier, stocke les paires de nombres entiers premiers jumeaux trouvés. L'ajout d'une paire se fait en section critique car la méthode emplace_back de std::vector n'est pas thread safe. L'accesseur sur le résultat se fait en section critique même si ce n'est pas vraiment nécessaire car seul le thread principal appelle cette méthode, une fois que tous les threads ont fini de travailler. La fonction exécutée par les threads, calculate, effectue une boucle tant que le nombre courant du moniteur d'intervalle et inférieur à son maximum – 1 (pour ne pas calculer une paire qui sorte de l'intervalle donné). Le thread vérifie si le nombre « curr » qu'il a récupéré est premier, si c'est le cas, il vérifie si « curr » +2 est premier. Si c'est encore le cas alors on ajoute la paire « curr », « curr+2 » au moniteur de nombres premiers. Avant la fin de la boucle, on récupère le nouvel élément courant du moniteur d'intervalle.

Résultat

En testant avec l'intervalle [10000000 ;10001000], j'obtiens les couples (10000139,10000141), (10000451,10000453), (10000721,10000723) et (10000871,10000873) en un peu moins de 400 ms avec 12 threads et plus. On observe encore ce palier lié au nombre de processeurs logiques, je suis à plus d'une seconde pour 2 threads.