I03 : Modèles et techniques en programmation parallèle hybride et multi-cœurs

Romain Horcada

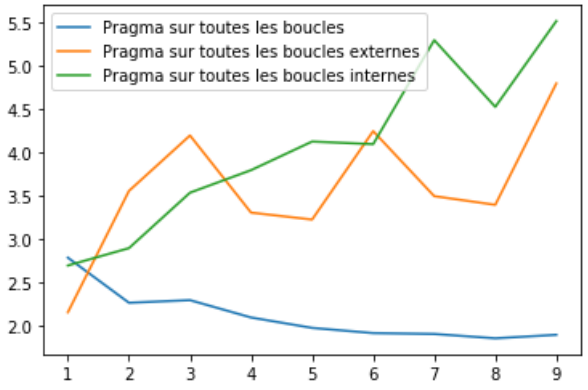
22 décembre 2020

1 Version multithreads avec OpenMP (Grain fin)

Le but de cette section est de trouver les boucles parallélisables afin d'y ajouter des pragma pour accélérer l'éxécution. Tout d'abord, la boucle en temps n'est clairement pas parallélisable, chaque itération modifie le même vecteur solution et chaque itération dépend de la précédente (dans la fonction iteration-domaine). Enfin, en observant la valeur de la variation affichée à l'éxécution, nous n'obtenons pas la bonne solution lorsque l'on place un pragma devant cette boucle.
A certains endroits, plusieurs boucles en espace sont imbriquées, on se demande à quels endroits peut/doit on mettre des pragmas pour que 1) La solution soit la même qu'en séquentiel ? 2) L'éxécution soit la plus rapide possible ?

1.1 Temps d'initialisation

On décide de comparer plusieurs approches, placer un pragma sur les boucles les plus externes, placer un pragma sur toutes les boucles ou placer un pragma sur les boucles internes. Les résultats obtenus sont résumés dans ce graphique :



Le meilleur choix à faire est d'ajouter des pragmas sur toutes les boucles for des fonctions d'initialisation (courbe bleue).

1.2 Temps total

L'essentiel du travail porte sur les boucles en espace de la fonction iteration-domaine qui est appelée à chaque itération en temps. Reste à savoir sur quelles boucles doit-on ajouter des pragmas pour que l'éxécution soit la plus rapide possible.

On va ajouter des pragmas sur toutes les boucles for de la fonction iteration-domaine, en parallélisant de la sorte, on maximise, dans ce cas précis, le rapport temps gagné sur ressource utilisée via une division maximale du travail sur un grand nombre de thread. Néanmoins, l'éxécution n'est pas plus rapide qu'en séquentiel car on créé beaucoup de threads et ce temps n'est pas rattrapé par la division des tâches. La version avec des pragmas partout semble être la moins pire d'après le schéma ci-après.

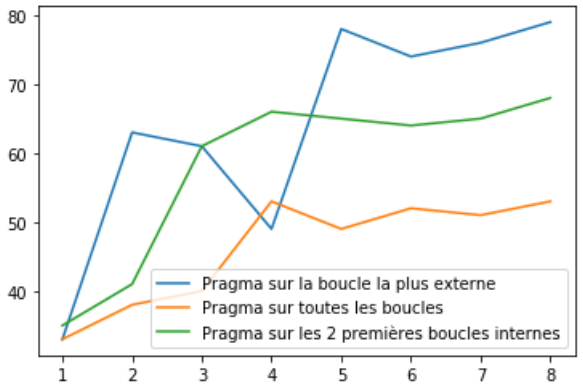
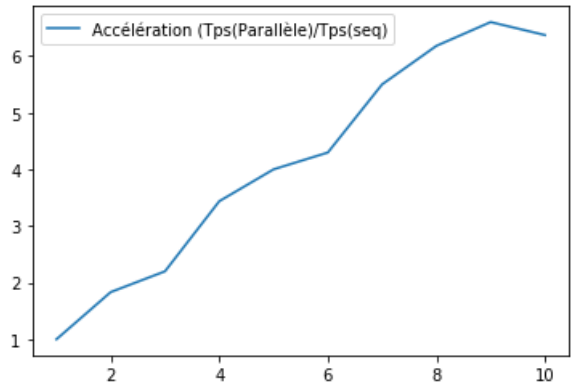


FIGURE 1 – Evolution de l'accélération en fonction du nombre de threads

2 Version multithreads avec OpenMP (Grain grossier)

Le but de cette méthode est de créer des régions parallèles plus grandes et moins nombreuses que pour la version grain fin. On commence par rajouter une région parallèle autour de la boucle en temps du main. On choisit de laisser séquentiel l'affichage des temps d'éxécution, paralléliser cette tâche n'améliore pas la vitesse d'éxécution et permet une meilleur lisibilité des résultats.

Pour suivre la méthode grain grossier, on va utiliser les fonctions iminlocal et imaxlocal pour calculer les intervalles dans lesquels vont évoluer les 3 indices d'espace. On décide de calculer les 6 arguments (2 bornes par dimension) à l'intérieur de la région parallèle mais en amont de la boucle en temps. On modifie la fonction itération pour qu'elle prenne en argument 6 entiers. Ces entiers ont été calculés à partir d'une direction (0,1 ou 2) et d'un numéro de thread obtenu par la fonction 'ompgetthreadnum()'. Néanmoins, la ligne 'C.iteration()' doit être précédée de la ligne "pragma omp single".



La forme de la courbe d'accélération se rapproche de la courbe d'une fonction proportionnelle au nombre de threads utilisés.

Conclusion

Nous avons vu que la méthode de grain grossier était bien plus adaptée pour paralléliser notre script et réduire le temps total d'éxécution. La méthode de grain fin créé un trop grand nombre de thread par rapport à ce qu'ils fournissent comme performance. En revanche, pour la réduction du temps d'initialisation, la méthode de grain fin semble plus adaptée. Pour obtenir une éxécution la plus rapide possible il faudrait alors paralléliser selon un mélange de ces deux approches.