Travaux Pratique 3

Ce sujet est en lien avec le troisième chapitre du cours, et concerne les *patrons de conceptions parallèles*. Ces patrons se retrouvent dans de nombreuses API parallèles : Thrust pour les GPU Nvidia, MPI … mais aussi directement dans la norme du langage C++ ! Cependant, leur introduction en C++ est assez récente (confer *C++ Extensions for Parallelism, ISO/IEC TS 19570:2015*). Ainsi un compilateur à jour est-il nécessaire. Visual C++ 2019 est bienvenu ; idem avec g++ depuis la version 9, que vous trouverez par exemple dans la version `testing` de Debian (nom de code : `bullseye`).

L'idée de travailler avec ces patrons en C++ sur CPU, qui n'est pas terrible comme machine parallèle, et non pas sur GPU qui semble plus adapté, est que tout le monde possède un ordinateur avec CPU (enfin, j'espère), et donc qu'il vous est possible de travailler à la maison.

Dans cette séance, dans un premier temps, l'idée est de travailler avec des patrons simples en C++. Puis, il s'agit d'essayer de les implanter sur une machine classique multicœur, grâce à la gestion des threads proposée par C++. Vous verrez notamment que c'est exactement ce que vous fîtes lors des précédentes séances !

Enfin, rappelez-vous qu'un algorithme parallèle écrit pour p processeurs peut être simulé sur une machine parallèle contenant seulement $p' < p$ processeurs … Souvent le temps d'exécution en est réduit, mais cela améliore l'efficacité !

> Votre travail est à rendre (il le sera à chaque fois) sous la forme du code (et uniquement la partie « student ») accompagnée d'un rapport au format PDF, le tout dans une archive compressée au format ZIP. Ne respecter pas ces contraintes, et votre note en sera diminuée de quelques points.
>
> **Vous ne devez modifier que ce qui se trouve dans le répertoire « `./student/` » !**
>
> Notez que chaque exercice vient avec un squelette, qui s'occupe de la ligne de commande, du lancement de votre code (défini dans une classe particulière), et d'une vérification sommaire du résultat (lorsque c'est possible). Utilisez l'option « `-h` » ou « `--help` » pour connaître le fonctionnement de la ligne de commande …
>
> Deux scripts sont fournis pour compiler : sous Windows, il s'agit de « `./build.bat` » qui génère les exécutables dans le répertoire « `./win32/Release` » ; pour les autres systèmes, utilisez « `./build.sh` » qui génère les exécutables dans « `./linux/` ». Ces scripts utilisent « `cmake` », et soit `GCC` sous linux/mac, soit `Visual C++ 19` sous Windows. Leur comportement a été testé sous Windows, WSL/Debian, et Linux Mint.

Exercice 1

Utilisation d'un MAP, pour calculer le carré des éléments d'un tableau, puis pour une addition de deux vecteurs (deux tableaux). En C++, ce patron s'appelle `transform`. Il est défini dans `algorithm`, et nécessite aussi d'inclure `execution`.

Exercice 2

Utilisation d'un REDUCE, pour calculer la somme des éléments d'un tableau, puis la somme de leur carré. Essayez pour ce dernier deux versions : un `transform` suivi d'un `reduce`, et un unique `transform_reduce`. Notez les différences de temps d'exécution.

Exercice 3

Complétez les deux versions de la fonction « `transform` » afin d'effectuer ce patron parallèle via n threads, ou via un pool de threads[1]. Utilisez vos versions pour refaire le premier exercice, et concluez sur les performances !

Exercice 4

Complétez la fonction « `gather` » afin d'effectuer ce patron parallèle via n threads. Testez cela avec la fonction `run_gather`.

Complétez la fonction « `scatter` » afin d'effectuer ce patron parallèle via n threads. Testez cela avec la fonction `run_scatter`.

Exercice 5

Complétez la fonction « `reduce` » afin d'effectuer ce patron parallèle via n threads. Pour cela, chaque thread doit calculer une partie de la somme. Le thread principal terminera le calcul par sommation standard (le nombre de threads étant généralement limité sur CPU, cela ne pose pas de soucis ici).

Quelle amélioration obtenez-vous en termes de temps de calcul ?

[1] Vous pouvez utiliser la fonction `OPP::ThreadPool& OPP::getDefaultThreadPool()`, qui retourne le pool par défaut (patron singleton). Il est conseillé de prendre un nombre de tâches égal à 4 fois le nombre de processeurs logiques (équilibrage de charge).