Travaux Pratique 4

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, l'objectif est d'implanter les patrons SCAN en version inclusive et exclusive, 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.

NB : dans les exercices suivants, utilisez l'option « -s=16 » afin de faciliter le déverminage.

Exercice 1

L'objectif ici est d'implanter deux versions du SCAN INCLUSIF, pour des types et opérateurs quelconques. Ces deux versions sont :

1. La première version est en séquentiel, ce qui est trivial.
2. La seconde version est en parallèle (à écrire dans le fichier `inclusive_scan.h`) via le pool de threads par défaut (méthode `OPP::getDefaultThreadPool()`).

Quelle est l'accélération pour n éléments ? Dans le rapport, donnez aussi le nombre d'applications de la fonction utilisée (bref la complexité en nombre d'appel de l'opérateur).

Exercice 2

L'objectif ici est d'implanter deux versions du SCAN EXCLUSIF, là aussi pour des types et opérateurs quelconques. Ces deux versions sont :

1. La première version est en séquentiel, ce qui est trivial.
3. La seconde version est en parallèle (dans le fichier `exclusive_scan.h`) via le pool de threads par défaut (méthode `OPP::getDefaultThreadPool()`).

Quelle est l'accélération pour n éléments ? Dans le rapport, donnez aussi le nombre d'applications de la fonction utilisée.