TP1 - Parallélisme et efficacité

L'objectif est d'observer grossièrement l'efficacité du parallélisme.
Copiez le répertoire `~sathibau/PAP/TP1` chez vous.

1 Mesure du temps

Tout d'abord, essayons différentes façons de mesurer le temps. Compilez le programme `temps` en lançant `make temps`. Il existe en gros trois méthodes, de la plus grossière à la plus fine :

- La commande `time` : lancez `time ./temps` : quelque chose comme
  ```
  real    0m3.628s
  user    0m3.616s
  sys     0m0.004s
  ```
  s'affiche : vous obtenez ainsi le temps passé dans le programme lui-même (ici 3.616), le temps passé en mode noyau (ici $0,004$) et le temps final.
- La fonction `gettimeofday` : décommentez l'affichage de `TIME_DIFF()` et relancez le programme. Notez que la précision est bien meilleure.
- Le compteur de cycles du processeur : décommentez l'affichage de `TICK_DIFF()` et relancez le programme. Obtenez depuis `/proc/cpuinfo` la fréquence du processeur en `MHz` pour en déduire le temps en s. On ne peut avoir plus précis comme mesure !

Selon la finesse voulue, on pourra donc utiliser l'une ou l'autre de ces méthodes.

2 Somme de cos/sin

Une version séquentielle de la somme de cosinus et sinus de matrices est fournie dans `mat.c`

Lisez, compilez et lancez (plusieurs fois) ce programme, vérifiez que le nombre de cycles reste en gros le même.

Juste pour voir, essayez d'échanger les deux boucles `for` à l'intérieur de la fonction `addmatrix`. Quel est le résultat ? Cela est dû aux effets de cache, qui seront vus ultérieurement en cours.

3 Version parallèle

Adaptez ce programme pour obtenir une version parallèle utilisant P threads qui chacun traitent une partie des matrices (pour les CSI, il y a un exemple de programmation de threads

dans le fichier `exemple.c`). La structure générale de `main()` sera donc de la forme :

- générer les matrices (déjà écrit),
- lancer les P threads en leur passant en paramètre leur numéro.
- attendre la terminaison de ces P threads,

Tandis que celle de la fonction `thread_fonction()` exécutée par les threads sera :

- récupérer le numéro de thread (déjà écrit)
- en déduire la partie de matrice à traiter
- une double boucle for pour effectuer le traitement.

Pour la « tranche de matrice à traiter », on aura plusieurs approches. Pour commencer, divisez simplement les N lignes (boucle `i`) en P tranches, que vous confiez à chaque thread.

Comparez les temps de calcul de la version séquentielle et de la version parallèle. Y a-t-il un gain à paralléliser ? De combien de processeurs dispose la machine sur laquelle vous travaillez (cherchez dans `/proc`) ? Peut-on généraliser les résultats obtenus ?

Donnez à P différentes puissances de 2, et calculez à chaque fois le *speedup* (pourquoi des puissances de 2 d'ailleurs ?) Vous pouvez réessayez sur d'autres machines multiprocesseurs (`hagrid`, `fridulva`, `dudley1` ou `dudley2` par exemple, utilisez `top` pour être sûr d'être seul à calculer sur la machine). Juste pour voir, essayez en répartissant les colonnes au lieu des lignes. Cela change-t-il quelque chose ?

Il vous est conseillé de noter le nom des machines sur lesquelles vous travaillez, ou de les retrouver via Nagios (`https://services.emi.u-bordeaux1.fr/nagios3/`) pour les allumer au besoin et vous y connecter à distance en étant déjà réparti sur les machines parallèles du CREMI pour vos travaux futurs.

4 Ordonnancement dynamique

Une autre approche est de répartir dynamiquement les indices : écrivez une fonction `get_indice()` qui « distribue les indices » : elle possède une variable statique `i` initialisée à 0 (la première ligne de la matrice), et à chaque fois qu'on l'appelle, retourne `i++`. Il faudra bien sûr utiliser un mutex (pourquoi ?) (Pour les CSI, il y a un exemple de programmation de mutexes dans le fichier `exemple-mutex.c`)

La structure de la fonction `thread_fonction()` sera alors simplement :

- tant que `get_indice()` retourne une valeur plus petite que N, effectuer le calcul pour le numéro de ligne renvoyé.

Mesurez le temps obtenu (faites varier P). Est-ce intéressant ? Pourquoi ?

5 Amdahl en force

De la même façon que vous avez parallélisé `addmatrix` en répartissant simplement les lignes en P tranches, parallélisez `totalmatrix()`. Il faudra bien sûr utiliser un mutex pour protéger l'accès à la variable `sum`.

Mesurez le temps obtenu. Est-ce intéressant ? Pourquoi ? Comment faire pour éviter le problème ?