TP1 - Efficacité des programmes séquentiels et premières expériences parallèles

L'objectif est de mettre en œuvre des techniques d'optimisation pour les programmes séquentiels. Copiez le répertoire `/net/cremi/rnamyst/etudiants/pmg/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. On ne peut avoir plus précis comme mesure ! Ce compteur n'est cependant pas forcément synchronisé entre les différents cœurs et processeurs.

Calculez expérimentalement la fréquence du cœur utilisé pour l'exécution du programme. Comparez ce résultat à la fréquence du processeur écrite dans le fichier `/proc/cpuinfo` [1]. Notez que la fréquence du processeur peut éventuellement changer dynamiquement.

2 Fusion de boucle

Observez le programme `boucles.c`. Quelle optimisation auriez-vous naturellement tendance à faire ? Le gain obtenu est-il décevant, correct, ou plus que satisfaisant ? Comment l'expliquer ?

Reproduire l'expérience en ajoutant une troisième puis une quatrième boucle, obtenez-vous une accélération encore meilleure ? Pourquoi ?

3 Déroulement de boucle

Tout d'abord, observons le coût d'une boucle : le programme `deroulement.c` effectue un calcul tout bête au sein d'une boucle qui a un nombre d'itérations connu. Tel quel, chaque mesure prend quelques secondes.

1. La fréquence y figure deux fois : l'une en commentaire du modèle de cpu, l'autre en tant que fréquence instantanée.

Déroulez la boucle, c'est-à-dire par exemple définir `D` à 2 pour faire 2 fois moins d'itérations, mais en répliquant le contenu de la boucle pour lui faire faire deux fois le calcul par itération. Attention aux indices de tableau (*vérifiez* que le résultat obtenu est bien le même). Essayez en déroulant 2 itérations, 4 et 8.

Est-il intéressant de dérouler indéfiniment ? Quelle combinaison d'options de gcc permet de dérouler les boucles ?

4 Multiplication et somme de matrices

Les programmes `mul_mat.c` et `som_mat.c` effectuent sommes et multiplications de matrices de façon très basique. Cette façon de faire est loin d'être optimale.

1. Lancer plusieurs fois le programme `som_mat` pour vérifier que les procédures `somMat` et `somMat2` ont un temps d'exécution similaire.
2. Modifier la procédure `somMat2` en permutant l'ordre des boucles sur i et sur j. Mesurer l'accélération obtenue.

   Cette accélération est due à une meilleure exploitation de la mémoire cache qui, grossièrement, charge dans sa mémoire, non seulement la valeur de la case mémoire demandée par le processeur, mais aussi celles de ses cases voisines.
3. A votre avis, quelle case de `t[i][j+1]` ou de `t[i+1][j]` est rangée en mémoire à coté de `t[i][j]` ? Est-ce vraiment toujours le cas ?
4. Recommencer l'expérience en faisant varier la taille de la matrice (prendre N = 1024, N = 256, N = 64). Expliquer les résultats obtenus en vous aidant de la commande lstopo pour connaitre la taille des caches.

On s'intéresse maintenant au produit de matrices.

1. Modifier le code de `prodMat2` afin d'utiliser plus efficacement le cache du processeur. Le gain obtenu est-il décevant, correct ou plus que satisfaisant ?
2. Il est probable que quelques défauts de cache évitables subsistent dans votre code. Les repérez-vous ? Quelle permutation des boucles sur i, j, k induit le plus petit nombre de défauts de cache ? Modifier votre code en conséquence.
3. Lorsque N est assez grand il est probable quelques défauts de cache évitables subsistent dans votre code. Supposons que le cache fasse 8 Mo pour quelle valeur de N apparaissent ces défauts de cache ?

5 Parallélisation de boucles for

Il s'agit de recueillir et de comparer les performances de différentes techniques de répartition d'un ensemble d'indices :

— répartition par bloc : chaque thread exécute une tranche d'indices contigus, chaque tranche contenant le même nombre d'éléments ;
— répartition cyclique : chaque thread exécute les indices congrus au numéro de leur thread, les threads étant numérotés de 0 à $P - 1$;
— répartition dynamique : les threads piochent les indices au fur et à mesure.

Pour évaluer les différentes politiques de répartition on dispose dans le fichier `scheduling.c` de quatre fonctions :

- `int fpetit(int i)` : fonction constante de très faible durée
- `int fconstant(int i)` : fonction constante
- `int fcroissant(int i)` : fonction dont la complexité croît quadatriquement par rapport à i
- `int fperiodique(int i)` : fonction dont la complexité varie cycliquement selon une période de 200.

1. Compléter la fonction `lance(thread_fun)` afin que celle-ci lance `P` threads exécutant la fonction `thread_fun(&id[i])` où id[i] désigne le numéro du thread.
2. Vérifier le bon fonctionnement de votre programme en l'exécutant. Les temps affichés correspondent au temps mis par une version séquentielle et au temps de création/destruction des P threads pour chaque politique de répartition. Ces temps sont ils similaires ?
3. Après avoir observé la fonction sequentielle, compléter la fonction `void *cyclique(void *param)` afin que celle-ci exécute `fun(i)` pour tous les indices i congrus à l'identité du thread. Exécuter le programme. Faire afficher les speed-up.
4. De même, compléter les fonctions `block` et `dynamique`.

Interpréter les résultats obtenus sur votre machine. Tester sur différentes machines (telles `cocatrix`, `boursouf`, `infini1` ou encore `jolicoeur`), on pourra utiliser la commande `ssh machine $PWD/cmd`.

On pourra également faire varier N pour voir son impact sur les speed-up.