TP 1 : Optimisations & Parallélisation

Récupérer le code à compléter sur https://www.lri.fr/~bagneres/index.php?section=teaching&page=2016_Et4_archi_parallele

Le projet utilise *CMake* https://fr.wikipedia.org/wiki/CMake pour lancer la compilation des différents tests. Le fichier `README.txt` donne les différentes commandes pour compiler les différents tests (pour les systèmes comme GNU/Linux, sinon : https://www.lri.fr/~bagneres/index.php?section=tools&page=cmake_and_cpp).
Pour lancer un test en particulier : `make && ./test__nom_du_test`

Pour les différentes versions et les différentes tailles, faire un graphique des temps obtenus. Pour une version et une taille donnée, exécuter plusieurs fois le programme (au moins 5 fois) et prendre le temps minimum. Expliquer les résultats obtenus.

Faire les exécutions à comparer sur la même machine et dans les mêmes conditions. Donner les caractéristiques du système d'exploitation, du compilateur et du processeur (modèle, nombre de cœurs, hyperthread, taille des caches, ... (la commande `cat /proc/cpuinfo` est un bon début)).

0.1 Donner deux utilisations/intérêts d'utiliser différents threads dans un même programme.

1 Multiplication De Matrices

Le fichier `tests/matmul.cpp` implémente une multiplication de matrice (`R = alpha * A * B + beta`) :

```
for (size_t i = 0; i < NB_ROW_R; i++)
{
    for (size_t j = 0; j < NB_COL_R; j++)
    {
        R[i][j] = beta;
        for (size_t k = 0; k < NB_COL_A; ++k)
        {
            R[i][j] += alpha * A[i][k] * B[k][j];
        }
    }
}
```

1.1 Expliquer pourquoi ce code a de mauvaises performances.

Une première optimisation possible est de "transposer" la matrice `B` tel que `B_ji[i][j] = B[j][i]`.

1.2 Dans le fichier `tests/matmul_B_ji.cpp` : remplir `B_ij` (ligne 72) et écrire la multiplication en utilisant `B_ij` au lieu de `B` (ancienne ligne 76). Vérifier que le code est équivalent en comparant les fichiers `.log` écrits avec un outil comme *diff* ou *kompare*.

1.3 Expliquer comment cette optimisation améliore les performances.

1.4 Peut-on appliquer cette optimisation dans tous les cas ? Expliquer pourquoi.

Une autre optimisation possible est d'inverser les boucles `j` et `k`.

1.5 Dans le fichier `tests/matmul_ikj.cpp` : écrire la nouvelle version de la multiplication (ligne 68). Vérifier que le code est équivalent.

1.6 Paralléliser les trois versions avec OpenMP. Expliquer pourquoi la parallélisation est possible et le gain obtenu (par rapport à celui pouvant être naïvement espéré).

Comparer les différentes versions (`matmul`, `matmul_B_ji`, `matmul_ikj`), avec et sans OpenMP, en `-O3 -ffast-math` pour les tailles de matrices suivantes : 250, 500, 750, 1000, 1250, 1500.

1.7 Faire un graphique et expliquer les résultats obtenus.

2 Parcourir Les Éléments : `std::vector` Vs `std::list`

On cherche à calculer l'écart type (*standard deviation* en anglais) d'une suite de valeurs. Les valeurs pourront être contenu dans un `std::vector` (http://en.cppreference.com/w/cpp/container/vector) ou une `std::list` (http://en.cppreference.com/w/cpp/container/list).
Avec `std::vector`, on peut utiliser des indices mais pour parcourir et modifier une `std::list`, on est obligé d'utiliser ses itérateurs (http://en.cppreference.com/w/cpp/concept/Iterator).

Dans `tests/vector.cpp` : calculer la somme des éléments du vecteur (l. 44).

2.1 Paralléliser ce calcul. Expliquer pourquoi la parallélisation est possible.

2.2 Si le code est bien parallélisé, le test échoue alors que les valeurs affichées sont identiques : expliquer pourquoi.

(Modifier la précision d'affichage avec `std::cout.precision(std::numeric_limits<double>::max_digits10);`)

2.3 La parallélisation apporte-t-elle un réel gain ? Expliquer pourquoi.

Dans `tests/liste.cpp` : calculer la somme des éléments de la liste (l. 49).

2.4 La parallélisation est-elle possible ? Pourquoi ?

Le calcul de la variance est le suivant : on accumule dans une variable le carré de chaque élément moins la moyenne des valeurs : $variance = variance + (élément - moyenne)^2$.

2.5 Dans `tests/vector.cpp` : calculer la variance (ancienne l. 76). Paralléliser ce calcul.

2.6 Dans `tests/list.cpp` : calculer la variance (ancienne l. 81).

Comparer les performances maximales obtenues avec `std::vector` (version parallélisée si plus performante) et `std::list` en `-O3 -ffast-math` pour les tailles suivantes : 1000×1000, 5000×1000, 9000×1000.

2.7 Faire un graphique et expliquer les résultats obtenus.

3 Ajouts, Suppression, Tri : `std::vector` Vs `std::list` (Bonus)

Le test `tests/add_and_remove.cpp` utilise la programmation générique disponible en C++ pour tester les performances d'ajout et suppression d'élément de `std::vector` et `std::list`.
On s'intéresse à la fonction `add_and_remove_and_sort`.

3.1 Comment fonctionne un ajout d'élément dans un `std::vector` ? dans une `std::list` ? En théorie, lequel est le plus efficace ? Expliquer pourquoi.

3.2 Mêmes questions avec la suppression d'un élément.

Comparer les performances maximales obtenues avec `std::vector` et `std::list` en `-O3 -ffast-math` pour les tailles suivantes : 1000×1000, 5000×1000, 9000×1000.

3.3 Faire un graphique et expliquer les résultats obtenus.

3.4 Quelle est la différence entre les itérateurs de `std::vector` et ceux de `std::list` ?

3.5 Comment fonctionne l'idiome `erase-remove` (l. 36 - 40) ? Comment pourrait-on paralléliser ce code ?

3.6 Quel est le pire cas pour `std::vector` où l'on supprime un élément ? Expliquer pourquoi.

Écrire un nouveau test (relancer `cmake ..`) qui compare les performances avec (`std::vector` et `std::list`) pour ce cas avec différentes tailles.

3.7 Faire un graphique et expliquer les résultats obtenus.

3.8 Réfléchir et décrire un conteneur qui serait performant pour ce cas (et aussi les cas précédents). Expliquer pourquoi.