École Nationale Supérieure d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, FRANCE

TP $n^o 2$

Niveau	**$2^{ème}$ année**
Parcours	**Informatique**
Unité d'enseignement	**2I2AC3 - Architectures parallèles**
Responsable	**Emmanuel Cagniot** **Emmanuel.Cagniot@ensicaen.fr**

1 Problème

Nous souhaitons proposer plusieurs variantes OpenMP de l'algorithme de multiplication matrice-vecteur de la figure 1. Sa simplicité offre plusieurs possibilités de parallélisation/vectorisation au niveau des lignes et/ou des colonnes de la matrice.

L'archive `tp2.tar.gz`, accessible via la page web décrivant l'unité d'enseignement, contient un squelette incomplet des applications à réaliser. Sa structure est la suivante :

`src/include/ :` contient les déclarations des différentes versions de notre produit matrice-vecteur ;

`src/ :` contient les définitions des différentes versions de notre produit matrice-vecteur ainsi que les deux programmes principaux `bench.c` et `bench_vs.c` ;

`CMakeLists.txt :` script permettant de générer le makefile de l'application via l'utilitaire `cmake`. Six exécutables différents (ils permettent de vérifier la validité du résultat) sont générés grâce à l'utilisation conjointe de `bench.c` et de directives pré-processeur :

- `bench` qui représente le programme exploitant une version canonique du produit matrice-vecteur ;
- `bench_omp_for_i` qui exploite une parallélisation sur les lignes de la matrice ;
- `bench_omp_for_j1` qui exploite une parallélisation sur les colonnes de la matrice ;
- `bench_omp_for_j2` qui exploite une parallélisation sur les colonnes de la matrice avec maximisation de la région parallèle associée ;
- `bench_omp_simd` qui exploite une vectorisation sur les colonnes de la matrice ;
- `bench_omp_for_simd` qui exploite une parallélisation sur les lignes de la matrice et une vectorisation sur les colonnes.

Cinq autres exécutables permettant de comparer versions séquentielle et parallèle/vectorielle sont générés à partir de `bench_vs.c`. Ces exécutables sont nommés `bench_vs_xxx` où `xxx` est le nom de l'une des quatre versions parallèle/vectorielle décrites ci-dessus ;

`Lisezmoi.txt :` fichier texte décrivant la procédure de génération du makefile et celle de la configuration du fichier `CMakeCache.txt` produit par `cmake`. Une fois ce fichier modifié, le compilateur est autorisé à exploiter sa bibliothèque OpenMP (option `-fopenmp`) et ses extensions SIMD (option `-fopenmp-simd`), les caractéristiques de votre processeur (option `-march=native`) et toutes les optimisations possibles (option `-O3`).

Le nombre de threads disponibles se fixe via la variable d'environnement `OMP_NUM_THREADS`. Par défaut, sa valeur est le nombre de cœurs physiques du CPU ($\times 2$ si celui-ci propose l'hyperthreading c'est à dire deux threads logiques se partageant un cœur physique - cas des core $i3$ et $i7$ et de certains $i5$ sur ordinateurs portables). Par exemple, si vous ne souhaitez que deux threads alors il suffit de valider la commande shell `export OMP_NUM_THREADS=2` avant d'exécuter votre application. Notez que rien ne garantit que ces deux threads s'exécuteront sur des cœurs différents car ils doivent se partager les cœurs du CPU avec les threads d'autre applications et ceux du système d'exploitation.

La figure 1 présente la forme canonique (séquentielle et sans optimisation manuelle) de notre algorithme de multiplication matrice-vecteur. OpenMP n'accepte de paralléliser les boucles `for` que si la condition de continuation n'utilise pas les opérateurs égalité (`==`) ou différence (`!=`), ce qui permet à l'implémentation de pouvoir pré-calculer le nombre d'itérations à répartir sur les threads disponibles. Par conséquent, la boucle extérieure sur les lignes de la matrice et celle intérieure sur les colonnes utilisent toutes deux l'opérateur strictement inférieur à (`<`). La clause `reduction` ne pouvant être appliquée à un élément de tableau que depuis la version 4.5 d'OpenMP (celle qui accompagne les dernières versions de compilateurs), nous préférons utiliser une variable scalaire `acc` servant d'accumulateur dans le calcul de la valeur de chaque composante du vecteur b.

```
1  void
2  matvec(const float A[], const float x[], float b[], const unsigned size) {
3
4    // Accumulateur permettant de calculer la composante b[i].
5    float acc;
6
7    // Boucle sur les lignes de la matrice.
8    for (unsigned i = 0; i < size; i ++) {
9
10     // Initialisation de l'accumulateur.
11     acc = 0.0;
12
13     // Boucle sur les colonnes de la matrice.
14     for (unsigned k = 0; k < size; k ++) {
15       acc += A[i * size + k] * x[k];
16     }
17
18     // C'est terminé pour b[i].
19     b[i] = acc;
20
21   }
22
23 }
```

FIGURE 1 – Forme canonique de l'algorithme de multiplication matrice-vecteur.

Dans la suite de cet énoncé, vous vérifierez systématiquement la validité des fonctions écrites avant de comparer versions séquentielle et parallèle/vectorielle. Par exemple, si vous venez de compléter le fichier source `matvec_omp_simd.c` alors vous devez d'abord exécuter `bench_omp_simd` (vérification du résultat de votre application parallèle) avant `bench_vs_omp_simd` (comparaison des durées d'exécution).

1.1 Question

Complétez la définition de la fonction `matvec_omp_for_i` qui implémente une parallélisation sur les lignes de la matrice. La clause de répartition des itérations utilisée est `runtime`, ce qui signifie que la façon de répartir les itérations sera fournie via la variable d'environnement `OMP_SCHEDULE`.

1.2 Question

Quel type de répartition (paramètre *type* de la clause `schedule`) est logiquement le plus adapté pour notre fonction `matvec_omp_for_i` (justifiez votre réponse).

1.3 Question

En supposant que votre réponse à la question précédente soit `dynamic` (ce n'est qu'un exemple), faites plusieurs essais d'exécution de `bench_vs_omp_for_i` en augmentant progressivement la valeur du paramètre *size* de la clause `schedule` (par exemple `export OMP_SCHEDULE="dynamic"` puis `export OMP_SCHEDULE="dynamic,1024"` et ainsi de suite). Que remarquez-vous ?

La fonction `matvec_omp_for_i` implique une écriture dans l'élément de tableau `b[i]`, d'où un risque de false sharing. Dans un cas relativement simple comme le notre, il est préférable de laisser le compilateur choisir la façon de répartir les itérations en utilisant le type `auto` ; celui-ci parviendra à déterminer la valeur du paramètre *type* de la clause `schedule` (celle que vous auriez utilisée) mais également celle du paramètre *size* (celle que vous ne parveniez pas à fixer). Cependant, OpenMP n'est pas une bibliothèque mais un standard, c'est à dire que l'implémentation des spécifications du *consortium* OpenMP *Architecture Review* diffère d'un compilateur à l'autre.

1.4 Question

Le compilateur installé sur vos machine est GNU `gcc`. Refaites deux exécutions de votre application `bench_vs_omp_for_i`, l'une avec `OMP_SCHEDULE="static"` et l'autre avec `OMP_SCHEDULE="auto"` ; que remarquez-vous ?

1.5 Question

Confirmez vos soupçons liés à la question précédente en recherchant comment le type `auto` est implémenté dans `gcc` (tapez simplement les mots-clés `gcc openmp schedule auto` dans votre navigateur).

1.6 Question

Complétez la définition de la fonction `matvec_omp_for_j1` qui implémente une parallélisation au niveau des colonnes de la matrice en choisissant la meilleure façon de répartir les itérations de la boucle intérieure (vous devez être capable de jusitifier votre choix).

Vous avez remarqué que votre nouvelle implémentation `matvec_omp_for_j1` est beaucoup moins performante que `matvec_omp_for_i`. Avant de la jeter aux orties, n'oubliez pas qu'OpenMP est un standard et, par conséquent, vous êtes à la merci de votre compilateur. En effet, il peut traduire votre code de façon intelligente ou particulièrement stupide :

- s'il est intelligent il le ré-écrira pour qu'une région parallèle contienne à la fois la boucle extérieure séquentielle sur les lignes et votre boucle intérieure parallèle sur les colonnes. De fait, les threads étant crées en entrée de la région et détruits en sortie, ils sont disponibles tout le temps du calcul ;
- inversement, s'il est stupide alors il traduira votre code mot pour mot c'est à dire que des threads risquent d'être créés en entrée de la boucle parallèle sur les colonnes puis détruits à l'issue et ce à chaque itération de la boucle extérieure sur les lignes de la matrice : c'est pour cette raison que les développeurs doivent toujours maximiser leurs régions parallèles.

1.7 Question

Complétez la fonction `matvec_omp_for_j2` en tenant compte des remarques précédente (une attention très particulière devra être apportée à la synchronisation de vos différents threads).

1.8 Question

Après avoir testé votre nouvelle implémentation `matvec_omp_for_j2`, expliquez d'où provenaient les faiblesses de `matvec_omp_for_j1` par rapport à `matvec_omp_for_i` (le ratio des performances pouvait-il être prévu grosso modo ?).

Lorsque l'espace d'itération d'une boucle n'est pas très grand, il peut être plus intéressant de la vectoriser (optimisation mono-cœur) que de la paralléliser. L'idée est ici de demander au compilateur non pas d'exploiter des threads mais l'un des jeux d'instructions vectoriel disponibles sur le CPU.

1.9 Question

Complétez la fonction `matvec_omp_simd` qui vectorise la boucle intérieure sur les colonnes de la matrice.

1.10 Question

Vous avez remarqué que votre nouvelle implémentation `matvec_omp_simd` n'apporte rien de plus par rapport à la forme canonique `matvec`. Que recouvrent les options d'optimisation `-O3 -march=native` fournies lors de la compilation ? Déduisez-en le domaine d'usage de la clause `simd`.

Lorsque l'espace d'itération d'une boucle devient très grand, il peut être intéressant de combiner à la fois parallélisme et vectorisation. L'idée est ici de fractionner équitablement l'espace d'itération entre les threads disponibles puis de demander à chacun de vectoriser la boucle sur le sous-espace qui lui est associé.

1.11 Question

Complétez la fonction `matvec_omp_for_simd` qui parallélise la boucle extérieure sur les lignes de la matrice tout en vectorisant la boucle intérieure sur ses colonnes.

1.12 Question

Comparez votre nouvelle implémentation `matvec_omp_for_simd` avec `matvec_omp_for_i` : sont-elles équivalentes en performances ?

2 Exercice

Les mots de passe sont la plupart du temps protégés en base de données. On utilise généralement une fonction de hachage H pour laquelle la propriété de non-inversibilité nous garantit qu'à partir de $H(m)$, m ne peut pas être trouvé efficacement.

Supposons qu'un attaquant mette la main sur une liste de mots de passe protégés, $H(m_1)$, ..., $H(m_n)$. Afin de retrouver le maximum de mots de cette liste, l'attaquant cherche les entrées m_1, ..., m_n qui produisent les mêmes sorties. Pour cela, il génère des mots candidats c_1, ..., c_m en entrée et calcule les sorties correspondantes $H(c_1)$, ..., $H(c_m)$ qu'il compare aux $H(m_i)$. Si $H(c_i) = H(m_i)$, alors on sait que $m_i = c_i$ (avec une très grande probabilité).

Une attaque séquentielle sur un haché $H(m)$ consiste à générer un candidat c, à calculer $H(c)$ et à comparer $H(c)$ avec$H(m)$. Si $H(c) = H(m)$, l'attaque est terminée, sinon on recommence avec le c suivant. Les candidats peuvent être générés selon différentes stratégies. La plus connue est la stratégie dite « force brute » (aussi appelée naïve ou exhaustive). Elle consiste à générer les candidats dans un ordre lexicographique : a, b, c, ..., aa, ab, ac, ..., aaa, aab, aac, ..

Nous souhaitons proposer une implémentation « force brute » OpenMP permettant de retrouver un mot de passe hashé via l'algorithme (désormais obsolète) $SHA1$. L'archive `tp2.tar.gz` intègre le squelette incomplet de cette application. Plus précisément :

- le sous-répertoire `src/include/` contient un fichier en-tête `word_gen.h` contenant les déclarations de deux fonctions `word_gen_generate` et `word_gen_omp_generate`. La première représente une fonction permettant de générer tous les mots possibles d'une longueur fixée à partir d'un ensemble de caractères. Chaque mot généré est transmis à une fonction *callback* qui permet à l'utilisateur de traiter ce mot tout en indiquant s'il désire ou non générer le mot suivant. La fonction `word_gen_omp_generate` représente une version parallèle OpenMP de cette fonction ;
- le sous-répertoire `src` comporte :
  - le fichier `word_gen.c` contenant la définition de nos deux fonctions. Celle de `word_gen_generate` est donnée tandis que vous devrez compléter celle de `word_gen_omp_generate` ;
  - l'application `gen_word.c` qui illustre l'utilisation de `word_gen_generate` et plus précisément la façon d'écrire sa fonction *callback* ;
  - l'application `brute_force.c` dont vous aurez à compléter la fonction *callback*.

Il existe différentes stratégies de parallélisation d'une application « force brute ».

L'une d'elle, relativement simple, consiste à ne pas tenter de paralléliser le générateur d'entrées (ici des mots) mais simplement l'utiliser en implémentant une approche de type `SPMD` (*Single Program Multiple Data*) comme celle utilisée dans le TP précédent pour l'algorithme `count` et plus spécialement le calcul de π. Supposons que notre alphabet comporte N caractères et que nous disposions de P threads ; nous pouvons faire gérer des sous-ensembles d'environ N/P caractères consécutifs à chaque thread si $N > P$ ou un seul caractère si $N \leq P$. Plus précisément, il s'agit de confier à chaque thread la génération et le test de tous les mots dont la première lettre appartient à son sous-ensemble de caractères.

Comme le thème de notre TP est la parallélisation des boucles `for`, nous supposons que nous avons accès au code du générateur et l'autorisation de le modifier.

2.1 Question

Complétez la définition de la fonction `word_gen_omp_generate` afin de lui faire implémenter la stratégie décrite précédemment mais au niveau de la boucle `for` de notre algorithme récursif (ici la fonction interne `word_gen_generate_impl`). Vous aurez deux écueils à surmonter :

- bien choisir la clause de répartition des itérations ;
- modifier la condition de continuation de la boucle qui empêche l'implémentation de pré-calculer le nombre d'itérations à répartir (à cause du *flag* `again` qui permet de l'arrêter prématurément lorsque le *callback* retourne `false`). Il vous faudra faire usage des mécanismes d'interruption de threads proposés dans la dernière norme OpenMP et décrits en page 10 du support écrit de TP. L'effet de ces mécanismes est désactivé par défaut mais vous pouvez les activer (entre autres) en fixant à la valeur `true` la variable d'environnement `OMP_CANCELLATION` dans votre terminal (par exemple : `export OMP_CANCELLATION=true`).

2.2 Question

En vous inspirant du *callback* de l'application `word_generator.c`, complétez celui de notre application principale `brute_force.c`. Vous devez pour cela :

- récupérer le hashé du mot de passe mystère qui vous arrive via le paramètre `param` (castez le en `unsigned char*`) afin de le rendre exploitable ;
- calculer le hashé du nouveau mot (paramètre `word`) via les fonctions `SHA1_Init`, `SHA1_Update` et `SHA1_Final` de la bibliothèque OpenSSL (jetez un peu coup d'oeil sur Internet pour savoir comment faire) ;
- comparer les deux hashés via la fonction `strncmp` de la bibliothèque standard (pas `strcmp` car les deux hashés ne sont pas terminés par le caractère spécial `'\0'`) ;

— afficher le mot sur la sortie standard et retourner `false` si les deux hashés sont égaux ou simplement retourner `true` s'ils diffèrent.

L'application `brute_force.c` prend trois arguments via sa ligne de commandes :
— l'alphabet sous forme d'une chaîne de caractères (par exemple `cba`. L'ordre d'énumération des caractères sert de base pour l'ordre lexicographique du générateur) ;
— la longueur des mots à générer (seuls les mots de cette longueur sont générés à partir de l'alphabet) ;
— la signature $SHA1$ du mot de passe mystère.
Les temps d'exécution des versions séquentielle et parallèle sont ensuite comparés par le biais du *speedup* et de l'*efficiency*.

2.3 Question

Testez l'application `brute_force`. Pour générer la signature $SHA1$ du mot de passe mystère (par exemple `passwd`, un mot de passe très sécurisé ...), tapez simplement :

```
echo -n passwd | sha1sum | awk '{ print $1 }'.
```

Les temps de calcul précédents montrent que même pour une version séquentielle, les fonctions de hachage classiques (dites cryptographiques) sont extrêmement rapides en exécution. Par conséquent, pour peu que l'on mette du parallélisme dans la partie, l'attaquant a systématiquement l'avantage. On préférera donc des fonctions lentes et coûteuses en mémoire pour stocker les mots de passe en base de données (par exemple `bcrypt`, `argon2` ou `scrypt`).