École Nationale Supérieure d'Ingénieurs de Caen

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

TP n^o1

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

Problème

Ce TP constitue une introduction à la programmation multithread en langage C via OpenMP ; il a pour objectif de vous faire manipuler :

— les régions parallèles ;
— les données partagées et privées ;
— les différents mécanismes de synchronisation à votre disposition.

Les mécanismes SPMD et MIMD permettant de répartir une charge de travail entre threads seront abordés ultérieurement via deux autres TP spécifiques.

L'archive `tp1.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/ :` vide dans notre cas ;

`src/ :` contient les fichiers sources `serial.c`, `barrier.c` (incomplet pour l'instant), `count.c` (incomplet pour l'instant) et `pi.c` sur lesquels vous allez travailler ;

`CMakeLists.txt :` script permettant de générer le makefile des deux applications via l'utilitaire `cmake` ;

`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`), les caractéristiques de votre processeur (option `-march=native`) et toutes les optimisations possibles (option `-O3`).

1 Exercice

L'application `serial.c` permet d'écrire la valeur d'une variable `var` dans un fichier texte (situé dans votre répertoire courant) dont le nom est `thread_xxx.txt` avec `xxx` représentant le numéro du thread ; la figure 1 présente sa fonction `main`.

```
1  int
2  main() {
3
4    // Une variable entière.
5    int var = 10;
6
7    // Le numéro du thread.
8    int tid = omp_get_thread_num();
9
10   // Ecriture dans un fichier texte.
11   print_to_file(tid, var);
12
13   // C'est terminé.
14   return EXIT_SUCCESS;
15
16 }
```

FIGURE 1 – Fonction `main` de l'application `serial.c`.

La fonction `print_to_file` ne nous intéresse pas. La fonction `omp_get_thread_num`, fournie par la bibliothèque OPENMP, permet d'obtenir le numéro (*thread identifier*) associé à un thread ; cette numérotation commence à zéro.

1.1 Question

Après avoir généré l'exécutable `serial*`, exécutez-le puis assurez vous qu'un fichier texte `thread_0.txt` a bien été créé dans votre répertoire courant, ce dernier contenant la valeur de la variable `var`.

Par défaut votre implémentation OPENMP utilise le nombre de threads logiques de votre machine (par exemple quatre sur un *core-i*5 dépourvu de la technologie *hyperthreading* et huit dans le cas contraire s'il est de dernière génération). Nous souhaitons à présent activer tous ces threads afin de leurs faire créer chacun un fichier : c'est le rôle des régions parallèles.

1.2 Question

Placez les lignes $7 - 11$ de la figure 1 dans une région parallèle. Après avoir regénéré l'exécutable `serial*`, exécutez-le puis assurez-vous que de nouveaux fichiers textes ont bien étés créés dans votre répertoire courant (leur nombre correspondant au nombre de threads logiques de votre machine).

Déclarer la variable `tid` à l'intérieur de votre région parallèle rend celle-ci privée c'est à dire que chaque thread possède son propre exemplaire (et peut donc ainsi lui affecter une valeur différente de celle des autres threads). À l'inverse, la variable `var` est partagée (statut par défaut) par l'ensemble des threads puisqu'elle est déclarée à l'extérieur de la région parallèle. Toute tentative de modification de sa valeur à l'intérieur de notre région parallèle ne peut donc se faire que sous la supervision d'un mécanisme de synchronisation entre threads : l'oublier est une faute courante (appelée *race condition*) difficile à détecter. Si plusieurs threads tentent d'écrire quasi-simultanément (ca ne peut jamais être simultanément car votre machine est MIMD) dans une même cellule mémoire alors le contenu de cette dernière devient indéterminé.

Il peut arriver qu'en parallélisant un code en OPENMP, une variable (par exemple `var`) puisse être privatisée et donc manipulée sans avoir à recourir à des mécanismes de synchronisation. Cependant, la position de sa déclaration rend impossible son placement dans une région parallèle (comme vous l'avez fait pour `tid`). Si vous n'avez pas l'autorisation de modifier le code d'origine (ce qui est le cas général) alors il faut se tourner vers les clauses OPENMP permettant de créer un exemplaire privé d'une variable partagée.

Une clause `private(var)` associée à une région parallèle indique au compilateur qu'il doit créer un exemplaire de la variable partagée `var` dans la mémoire privée de chaque thread exécutant la région (comme si elle était déclarée à l'intérieur), cet exemplaire n'étant pas initialisé (son contenu est indéterminé).

1.3 Question

Privatisez la variable partagée `var` puis affectez lui la valeur de `tid` à l'intérieur de la région parallèle. Après avoir regénéré l'exécutable `serial*`, exécutez-le puis assurez-vous qu'un fichier `thread_xxx.txt` contienne bien la valeur `xxx`.

La clause `firstprivate(var)` opère comme `private(var)` mais initialise l'exemplaire privé de chaque thread à ce que valait la variable partagée `var` en entrée de la région parallèle.

1.4 Question

Remplacez la clause `private(var)` par `firstprivate(var)`. À l'intérieur de la région parallèle ajoutez la valeur de `tid` à `var`. Après avoir regénéré l'exécutable `serial*`, exécutez-le puis assurez-vous qu'un fichier `thread_xxx.txt` contienne bien la valeur `var + xxx`.

Au sein d'une région parallèle il arrive parfois qu'un groupe d'instructions ne puisse être exécuté que par un seul et unique thread, les autres devant attendre que le thread choisi ait terminé son travail. La directive `single`, qui s'utilise comme :

```
#pragma omp single
{
  instruction_1;
  instruction_2;
  ...
  instruction_n;
}
```

pour plusieurs instructions et comme :

```
#pragma omp single
  instruction;
```

pour une seule, permet d'assurer ce type d'opération. Cette directive sélectionne le premier thread disponible (n'importe lequel) pour lui faire exécuter les instructions concernées. Les autres threads attendent que ce dernier ait terminé puis tous reprennent leur exécution. Ce comportement est obtenu grâce à la barrière de synchronisation implicite placée derrière la dernière instruction à exécuter. Cette barrière peut être levée en associant la clause `nowait` à la directive `single`.

Notons que la directive `master` opère de la même façon que `single` mais diffère sur deux points importants :
— le *master thread* est systématiquement sélectionné ;
— les autres threads ne l'attendent pas.
Il faut donc avoir une bonne raison d'utiliser cette directive plutôt que `single`.

1.5 Question

Supprimez dans un premier temps tous les fichiers `thread_xxx.txt` de votre répertoire courant. Dans un second temps retirez la clause `firstprivate(var)` et l'instruction `var += tid;` de la question précédente puis faites en sorte que l'appel à la fonction `print_to_file` ne soit exécuté que par un seul thread (le premier disponible). Enfin, après avoir regénéré l'exécutable `serial*`, exécutez-le puis assurez-vous qu'il n'existe qu'un seul fichier `thread_xxx.txt` dans votre répertoire courant.

Il arrive également que tous les threads d'une région doivent exécuter des instructions modifiant le contenu d'une ou plusieurs variables partagées. Dans ce cas, afin d'éviter une *race condition*, ces instructions ne peuvent être exécutées que séquentiellement c'est à dire par un thread à la fois : c'est le rôle des sections critiques mises en œuvre via la directive `critical` qui s'utilise comme :

```
#pragma omp critical
{
  instruction_1;
  instruction_2;
  ...
  instruction_n;
}
```

L'implémentation OpenMP des sections critiques est cependant très particulière. En effet, afin d'éviter les problèmes d'inter-blocage (problème du « diner des philosophes »), OpenMP implante le même verrou (ou *mutex*, sémaphore, etc.) pour l'ensemble des sections critiques de l'application. Par conséquent, si une application comporte au moins deux sections critiques et qu'un thread est entré dans l'une d'elles mais pas encore ressorti alors aucun thread ne peut entrer dans les autres !

1.6 Question

Supprimez la directive `single` de la question précédente puis installez une section critique contenant :
— un affichage de la valeur de `tid` sur la sortie standard (instruction `printf("%d\n",tid);`) ;
— l'appel de la fonction `print_to_file`.
Après avoir regénéré l'exécutable `serial*`, exécutez-le plusieurs fois et constatez que l'ordre d'exécution d'une section critique est non déterministe c'est à dire qu'il diffère d'une exécution à l'autre selon la disponibilité des threads qui exécutent la région parallèle correspondante.

Les sections critiques concernent des blocs d'instructions. Lorsqu'il s'agit plus simplement pour un thread de modifier le contenu d'une variable partagée par les autres threads de sa région parallèle, il faut se tourner vers d'autres mécanismes de synchronisation beaucoup plus efficaces. Le premier est la directive `atomic` qui s'utilise comme :

```
#pragma omp atomic
  expression;
```

où `expression` est de l'une des formes suivantes :

`x` = `x` binop *expr*
`x` binop= *expr*
`x ++`
`++ x`
`x --`
`-- x`

La directive `atomic` exploite les primitives *lock/unlock* du hardware, ce qui la rend très efficace.

1.7 Question

Supprimez la directive `critical` ainsi que l'appel à la fonction `print_to_file` de la question précédente, pour les remplacer par une incrémentation de la variable partagée `var` protégée par une directive `atomic`. Faites ensuite afficher la valeur de cette dernière sur la sortie standard après la fin de la région parallèle (elle sera donc affichée par le *master thread* désormais seul). Après avoir regénéré l'exécutable `serial*`, exécutez-le puis vérifiez que la valeur affichée est cohérente avec la valeur initiale de `var` et le nombre de threads logiques de votre machine.

Malgré son efficacité, la directive `atomic` ne peut gérer un grand nombre de modifications d'une variable partagée. Supposons que nous devions calculer la somme des éléments d'un tableau de taille N et que

nous disposions de P threads. Une manière de procéder consiste à fractionner (virtuellement) ce tableau en tronçons de taille N/P et à les répartir sur nos threads (selon un mécanisme que nous aborderons dans le prochain TP). Nous pourrions ainsi espérer une accélération P mais il n'en sera rien car la variable partagée devant accueillir le résultat devra être mise à jour N fois : en d'autre termes, nos threads passerons plus de temps à attendre de pouvoir mettre à jour cette variable qu'à calculer.

Un autre mécanisme de synchronisation permet de gérer ce cas de figure : la réduction. Ce mécanisme est mis en œuvre via une clause `reduction(op:var)` accolée à une région parallèle. Cette clause indique au compilateur qu'il doit :

1. considérer qu'une clause `private(var)` est accolée à la région. Par conséquent, chaque thread va travailler sur un exemplaire local de `var` initialisé à l'élément neutre de l'opération `op` ;
2. combiner la valeur initiale de `var` en entrée de région avec celle de tous les exemplaires locaux via l'opérateur `op` pour mettre à jour la variable partagée `var` en sortie de région. Cette opérateur doit obligatoirement être associatif et commutatif car l'ordre d'arrivée des threads en sortie de région est indéterminé.

1.8 Question

Remplacez la directive `atomic` de la question précédente par une clause `reduction`. Après avoir regénéré l'exécutable `serial*`, exécutez-le puis vérifiez que vous obtenez bien le même résultat.

2 Exercice

La directive `barrier` représente le dernier mécanisme de synchronisation que nous allons aborder. La plupart des directives OpenMP (à l'exception de `master`) telles que `parallel`, `single`, etc. installent une barrière de synchronisation implicite à leur sortie : tous les threads exécutant la région parallèle concernées doivent mutuellement s'attendre sur cette barrière avant de reprendre leur exécution.

Dans un modèle d'exécution MIMD les threads ne peuvent, au mieux, que s'exécuter quasi-simultanément (les threads de l'application mais également ceux des autres applications ainsi que ceux du système d'exploitation se partagent les cœurs de la machine et peuvent même migrer d'un cœur à un autre en fonction de leurs niveaux de charge). Par conséquent, il peut arriver que le programmeur ait besoin d'installer explicitement une barrière de synchronisation afin, par exemple, de s'assurer que tous les theads de son application aient pris connaissance de la valeur d'une variable partagée avant de la modifier : c'est le rôle de la directive `barrier`.

2.1 Question

L'application `barrier.c` déclare un tableau d'entiers `values` dont la taille vaut le nombre de threads disponibles pour exécuter une région parallèle. Complétez son code afin que dans l'ordre :

1. chaque thread inscrive son identifiant `tid` à l'emplacement d'indice `tid` du tableau ;
2. l'ensemble des threads de la région effectuent une permutation circulaire du tableau afin qu'en sortie `values[tid]` contiennent la valeur initiale de son successeur (utilisez une copie locale de ce successeur pour effectuer l'opération).

3 Exercice

Nous nous intéressons à différentes techniques permettant de paralléliser des algorithmes exploitant des listes chaînées en OpenMP. Il s'agit d'un problème difficile pour plusieurs raisons :

- les listes, contrairement aux tableaux, ne sont pas des espaces d'adresses contigues. Par conséquent, l'accès à un élément est en temps linéaire (il faut partir de la tête de liste et suivre le chaînage jusqu'à atteindre cet élément) ce qui empêche l'utilisation des boucles `for` parallèles ;
- le coût du parcours de la liste peut être supérieur au travail effectué sur chaque élément de cette liste ;
- le nombre d'éléments de la liste n'est pas forcément connu ;
- etc.

Cependant, dans certains cas bien précis, il est possible de s'affranchir de toutes ces limitations et proposer des implémentations parallèles efficaces.

Soit la structure de liste simplement chaînée de la figure 2. Il s'agit d'une structure de cellule assez classique dans laquelle nous trouvons :

- la charge « utile » de la cellule c'est à dire l'élement lui-même (ici de type entier) ;
- un pointeur vers la cellule suivante, la valeur spéciale `NULL` étant utilisée pour signifier l'absence de voisin.

```
1  /**
2   * Cellule d'une liste d'entiers simplement chaînée.
3   */
4  struct slist_t {
5
6    /** La charge utile : l'élément lui-même. */
7    int elt;
8
9    /** Un pointeur vers la cellule suivante. */
10   struct slist_t* next;
11
12 };
```

Figure 2 – Structure de liste simplement chaînée.

Considérons à présent un algorithme de traitement de ces listes, par exemple `count`, un algorithme permettant de compatibiliser le nombre d'occurrences d'un certain élément. La figure 3 présente sa définition.

Comme tout algorithme de manipulation de liste chaînée, une grosse partie de l'implémentation est dédiée au parcours des éléments de la liste (boucle `while`) depuis sa tête (ici le paramètre `head`). Le corps de cette boucle contient le traitement effectué sur chaque élément de la liste (ici nous comptabilisons le nombre d'occurrences de l'élément `elt`). L'observation du code de la figure 3 illustre les difficultés décrites au début de cet énoncé :

- le nombre d'éléments de la liste est inconnu à priori : la boucle de parcours des éléments de la liste s'arrête lorsque le pointeur vers l'élément suivant vaut la valeur spéciale `NULL` ;
- le travail sur chaque élément de la liste peut être plus ou moins lourd en termes de durées de calcul (dans notre cas c'est trop peu pour pouvoir espérer une parallélisation efficace mais il s'agit tout de même d'un bon exercice).

Une technique de parallélisation très simple et très efficace consiste à reproduire la clause de répartition des itérations `schedule(static, 1)` des boucles parallèles `for`, c'est à dire une répartition cyclique des

```
unsigned long
count(struct slist_t* head, int elt) {

  unsigned long res = 0;
  struct slist_t* cell = head;

  while (cell != NULL) {
    if (cell->elt == elt) {
      res ++;
    }
    cell = cell->next;
  }

  return res;

}
```

FIGURE 3 – Algorithme `count`.

itérations de la boucle sur l'ensemble des threads disponibles. Pour cela, chaque thread doit d'abord se placer sur son premier élément. Ensuite, si $\mathcal{P}$ désigne le nombre de threads disponibles dans la région parallèle alors chaque thread doit avancer de $\mathcal{P}$ positions dans la liste, traiter l'élément correspondant et recommencer ainsi jusqu'à atteindre la valeur spéciale `NULL`.

Les seuls outils dont nous avons besoin pour implémenter cette stratégie sont in fine la région parallèle et les données privées c'est à dire toute donnée (variable ou constante) déclarée au sein de cette région. Dans notre cas, ces données sont respectivement :

— la constante entière `steps` qui représente le nombre de threads disponibles dans la région ;
— la constante entière `tid` qui représente le numéro du thread, la valeur 0 étant celle associée au *master thread* ;
— le pointeur `cell` permettant de se déplacer de cellule en cellule.

3.1 Question

Complétez la définition de la fonction `omp_count` du fichier `count.c` en implémentant la technique décrite ci-dessus. Les fonctions de bibliothèque `int omp_get_num_threads()` et `int omp_get_thread_num()` permettent d'initialiser respectivement les constantes `steps` et `tid` au sein de la région parallèle.

4 Exercice

La méthode des trapèses est une méthode d'intégration numérique assez basique qui, sur un intervalle $[a, b]$, approxime l'intégrale $I = \int_a^b f(x)dx$ par la quantité

$$I \simeq (b-a)\frac{f(a)+f(b)}{2}. \tag{4.1}$$

Nous comptons l'appliquer à l'approximation numérique de l'intégrale

$$\pi = \int_0^1 \frac{4}{1+x^2}dx. \tag{4.2}$$

L'application `pi.c` approxime la valeur de π par la méthode des trapèzes et un nombre de points d'intégration donnés via la ligne de commandes. La figure 4 présente le bloc d'instructions relatif au calcul proprement dit.

```
// Résultat final.
double res = 0.0;

// Pas d'integration.
const double h = 1.0 / nb_points;

// C'est parti.
for (long i = 0; i < nb_points; i ++) {
  const double x = h * (i - 0.5);
  res += 4.0 / (1.0 + x * x);
}
res *= h;
```

FIGURE 4 – Application de la méthode des trapèzes pour l'approximation de π.

Nous souhaitons paralléliser ce calcul avec pour seul outil la région parallèle. Si N et P représentent respectivement le nombre de points d'intégration et le nombre de threads disponibles (numérotés de 0 à $P-1$) :

- attribuer $(N/P)+1$ points d'intégration à chaque thread dont le numéro appartient à l'intervalle $[0, P-2]$, le dernier thread traitant le reliquat de points ;
- faire calculer un résultat local à chaque thread ;
- sommer les résultats locaux pour obtenir le résultat global.

4.1 Question

Modifiez l'application `pi.c` afin qu'elle implémente la stratégie décrite ci-dessus.