École Nationale Supérieure
d'Ingénieurs de Caen

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

Multi-threading en OpenMP (C et C++)

Niveau	3ème **année**
Parcours	**Électronique et Physique Appliquée**
Unité d'enseignement	2E1AC2 **- Architectures pour le calcul** [**Parallélisme**]
Création	9 **Novembre** 2020
Responsables	**Emmanuel Cagniot** **emmanuel.cagniot@ensicaen.fr** **Hugo Descoubes** **Hugo.Descoubes@ensicaen.fr**

Table des matières

1 Multi-threading en OpenMP (C et C++)

Le modèle de programmation naturel d'une machine SMP est le modèle à parallélisme de processus légers ou multi-threading. Dans ce dernier, un processus lourd est composé de processus légers ou threads, c'est à dire des groupes d'instructions qui s'exécutent de manière séquentielle ou concurrente sur l'ensemble des processeurs de la machine (modèle de programmation dit « à parallélisme de contrôle »).

Les threads communiquent entre eux par lecture/écriture des données du processus père. À tout thread est associée une pile système dans laquelle sont créées ses données privées c'est à dire les paramètres, les résultats et les données locales de ses fonctions.

Bien que ce modèle permette de tirer pleinement partie des possibilités de la machine, il est perçu comme relativement difficile à mettre en œuvre par les non initiés : risques d'inter-blocages, mécanismes d'exclusion mutuelle garantissant la cohérence des écritures, mécanismes de synchronisation, etc.

Une manière de contourner cette difficulté consiste à ne pas écrire directement une application multi-threadée via les bibliothèques de threads standardisés Posix mais plutôt à transformer un code séquentiel existant en un code multi-threadé en y ajoutant des directives destinées au compilateur, ce dernier se chargeant alors de la génération du code multi-threadé correspondant.

Une telle manière de procéder offre des avantages importants :
— l'application est développée en tant qu'application séquentielle, ce qui facilite sa mise au point ;
— les connaissances nécessaires se bornent aux grandes lignes de fonctionnement d'une machine SMP ainsi qu'aux directives proposées et à leurs effets. De fait, tous les publics sont concernés et non plus seulement celui des spécialistes.

Les inconvénients ne sont cependant pas négligeables :
— le seul parallélisme exploitable est celui de l'application séquentielle. Or, un bon algorithme séquentiel se révèle bien souvent être un mauvais algorithme parallèle. De fait, le programmeur doit posséder une certaine connaissance de la machine, notamment du fonctionnement des mémoires caches, afin d'exploiter pleinement le parallélisme. En conséquence, lorsqu'il conçoit son application séquentielle, il doit le faire en prévision de sa parallélisation ;
— le compilateur étant responsable de l'application des directives, l'efficacité de l'application est liée à la qualité de ce dernier.

Ainsi, toute parallélisation basée sur l'utilisation de directives doit être vue comme une façon de tirer un minimum de performances d'une machine SMP.

OpenMP est un standard pour la parallélisation des applications C, Fortran ou C++ par le biais de directives. Adopté en octobre 1997 par une majorité d'industriels et de constructeurs, ses spécifications appartiennent à l'OpenMP Archictecture Review Board, seul organisme habilité à les faire évoluer.

1.1 Éléments de base

1.1.1 Régions parallèles et séquentielles

Le nombre de threads est, par défaut, égal au nombre de processeurs (ou cœurs) de la machine. L'utilisateur peut néanmoins choisir le nombre de threads nécessaires à l'exécution de son application par le biais de la variable d'environnement `OMP_NUM_THREADS`. Le programmeur peut également imposer ce nombre par le biais de la fonction `omp_set_num_threads` tandis que la fonction `omp_get_num_threads` lui permet de connaître le nombre de threads demandés.

Le processus père compte pour un thread : il est appelé thread maître tandis que les autres sont appelés threads fils. Les threads sont affectés aux différents processeurs par le système d'exploitation. Si le nombre de threads indiqué par l'utilisateur est supérieur au nombre de processeurs, plusieurs threads sont affectés

à un même processeur.

Une application OpenMP est caractérisée par une alternance de régions séquentielles et parallèles. Une région séquentielle est exécutée par le thread maître tandis qu'une région parallèle est exécutée par l'ensemble des threads.

La figure 1 présente le principe des régions séquentielles et parallèles dans le cas d'un bi-processeur.

Toute zone de code ne faisant pas l'objet d'une directive désigne une région séquentielle (zones 1 et 3 dans l'exemple de la figure 1).

À l'inverse, toute zone de code à l'intérieur d'un bloc `omp parallel` désigne une région parallèle (zone 2 dans l'exemple de la figure 1). Dans cet exemple particulier, les deux threads se synchronisent à la sortie de la région parallèle puis seul le thread maître poursuit l'exécution de la dernière région séquentielle. Le modèle d'exécution utilisé par OpenMP est donc le modèle classique « *fork-join* ».

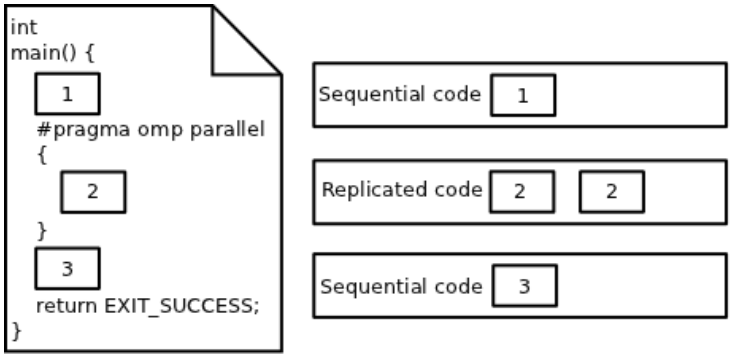

Figure 1 – Régions séquentielles et parallèles sur un bi-processeur.

Chaque thread peut obtenir son identifiant entier via la fonction `omp_get_thread_num`. L'identifiant du thread maître est classiquement la valeur zéro.

Il est possible d'imbriquer des régions parallèles, c'est à dire que les threads qui exécutent une région parallèle peuvent eux-mêmes créer d'autres threads lorsqu'ils rencontrent une région parallèle interne : nous parlons alors de parallélisme imbriqué. Par défaut, ce type de construction est désactivée mais l'utilisateur peut contrôler son activation ou sa désactivation en positionnant respectivement à `TRUE` ou `FALSE` la variable d'environnement `OMP_NESTED`.

Le programmeur peut faire de même via la fonction `omp_set_nested` tandis que `omp_get_nested` lui permet de savoir si la construction est activée ou pas.

L'utilisation du parallélisme imbriqué est problématique car la création d'un thread est une opération coûteuse qui doit être rentabilisée dans la quantité de travail que ce thread devra effectuer. Par conséquent, créer des threads supplémentaires lorsque la charge de travail est insuffisante ne peut amener qu'à une dégradation des performances de l'application. Lorsque la construction est désactivée, le compilateur ignore simplement la directive.

La clause `if (condition)` peut être ajoutée à la directive `omp parallel`. Son argument est une condition booléene. Si cette condition est vérifiée alors la région est considérée comme parallèle. Dans le cas contraire, elle est considérée comme séquentielle.

Il est également possible de fixer le nombre de threads qui vont exécuter une région parallèle en adjoi-

gnant le clause `num_threads(n)` où `n` désigne le nombre maximum de threads autorisés.

Les threads qui exécutent une région parallèle peuvent se répartir la charge de travail de cette dernière. Ce partage peut prendre trois formes.

1.1.2 Sections parallèles

Un bloc `omp sections` annonce la découpe d'une région parallèle en plusieurs zones de code délimitées par des sous-blocs `omp section` (zones 3 et 4 dans l'exemple de la figure 2).

Par défaut, une barrière de synchronisation implicite est placée à la fin d'un bloc `omp sections`, cette barrière ne pouvant être franchie que lorsque tous les threads l'ont atteinte. Pour des considérations d'optimisation, il est possible de supprimer cette barrière via la clause `nowait`.

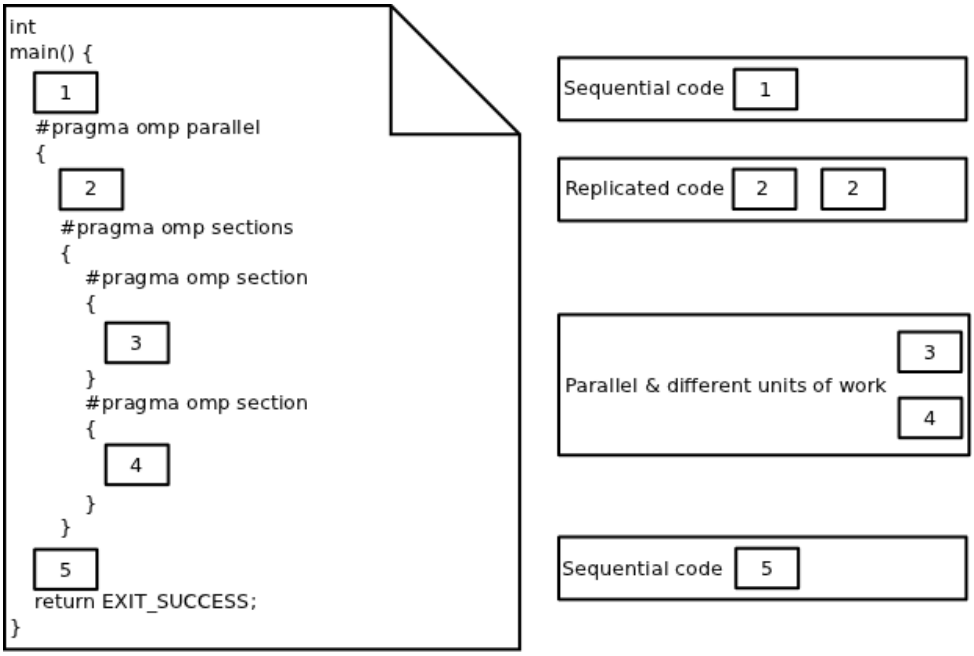

FIGURE 2 – Sections parallèles sur un bi-processeur.

Le nombre de sous-blocs `omp section` fixe le nombre de threads qui travaillent simultanément. Si ce nombre est inférieur au nombre de threads qui exécutent la région parallèle englobante alors ceux qui ne travaillent pas attendent les autres sur la barrière de synchronisation évoquée ci-dessus. Si le nombre de sous-blocs est supérieur au nombre de threads qui exécutent la région parallèle alors les threads disponibles terminent un sous-bloc avant d'en exécuter un autre (sérialisation).

La figure 3 présente la parallélisation d'un algorithme récursif par le biais de sections parallèles. Si le parallélisme imbriqué n'est pas autorisé alors un maximum de deux threads exécutent les sections. Inversement, si le parallélisme imbriqué est autorisé alors il faut prévoir une clause `if` permettant d'arrêter de créer des threads lorsque la taille du tableau à trier devient insuffisante. Si une région parallèle ne contient qu'un seul et unique sous-bloc `omp section` alors l'ensemble des directives peut être fusionné en `omp parallel section`.

L'utilisation de sections parallèles traduit le modèle d'exécution MIMD. L'un des principaux intérêts de cette construction réside en la possibilité de pratiquer le recouvrement communication/calcul, c'est à dire que certains threads communiquent pendant que d'autres calculent, ce qui évite les temps morts et donc

améliore la performance des applications parallèles. Le principal inconvénient est que cette construction est purement statique, c'est à dire fixée dès la compilation.

```
1  void
2  quicksort(double t[], const int& a, const int& b) {
3    const double pivot = t[(a + b) / 2];
4    int i = a, j = b;
5    do {
6      while (t[i] < pivot) {
7        i ++;
8      }
9      while (t [j] > pivot) {
10       j --;
11     }
12     if (i <= j) {
13       std::swap(t[i], t[j]);
14       i ++;
15       j --;
16     }
17   } while (i <= j);
18 #pragma omp parallel sections
19   {
20 #pragma omp section
21     if (a < j) {
22       quicksort(t, a, j);
23     }
24 #pragma omp section
25     if (i < b) {
26       quicksort(t, i , b);
27     }
28   }
29 }
```

FIGURE 3 – Parallélisation d'un *quicksort* à l'aide de sections parallèles.

Le mécanisme des sections parallèles pose problème dans le cas des algorithmes récursifs puisqu'il suppose d'activer le parallélisme imbriqué. Or, dans la plupart des cas, ce mécanisme conduit à une baisse dramatique des performances de l'application pour les raisons suivantes :

- une mauvaise implémentation pour la création et la destruction dynamique de threads (OpenMP est un standard et non une bibliothèque : par conséquent, vous êtes à la merci de votre compilateur et des possibilités de votre machine) ;
- l'overhead induit par la synchronisation d'un trop grand nombre de threads ;
- la difficulté de contrôler le nombre total de threads à un instant donné ;
- etc.

Il est possible de limiter la création de nouveaux threads via la clause additionnelle `if (...)` mais la valeur de coupure doit être fournie par le programmeur, ce qu'il fera quasi systématiquement de manière empirique. Une autre façon de limiter ce nombre consiste à fixer le nombre maximum de threads (initiaux plus dynamiques) via la variable d'environnement `OMP_THREAD_LIMIT` (par exemple `OMP_THREAD_LIMIT=4`). Lorsque cette limite est atteinte, plusieurs blocs `omp_section` peuvent être exécutés séquentiellement par les threads disponibles.

1.1.3 Boucles parallèles (de type `for`)

Une boucle de type `for` ne peut être parallélisée que sous trois conditions :
— les itérations doivent pouvoir être réalisées dans n'importe quel ordre (boucle de type « *forall* ») ;
— son indice doit être un entier (ou implicitement convertible en entier) ou un itérateur de type tableau (`RandomAccessIerator`, C++) ;
— la condition de continuation ne doit faire intervenir que les opérateurs `<`, `<=`, `>` ou `>=` (les opérateurs `==` et `!=` sont exclus) afin que le nombre d'itérations puisse être pré-calculé par l'implémentation OpenMP.

La parallélisation d'une boucle de type `for`, effectuée via la directive `omp for` (Figure 4), signifie que les threads qui exécutent la région parallèle englobante se répartissent les itérations de cette boucle. De fait, cette parallélisation traduit le modèles d'exécution SIMD.

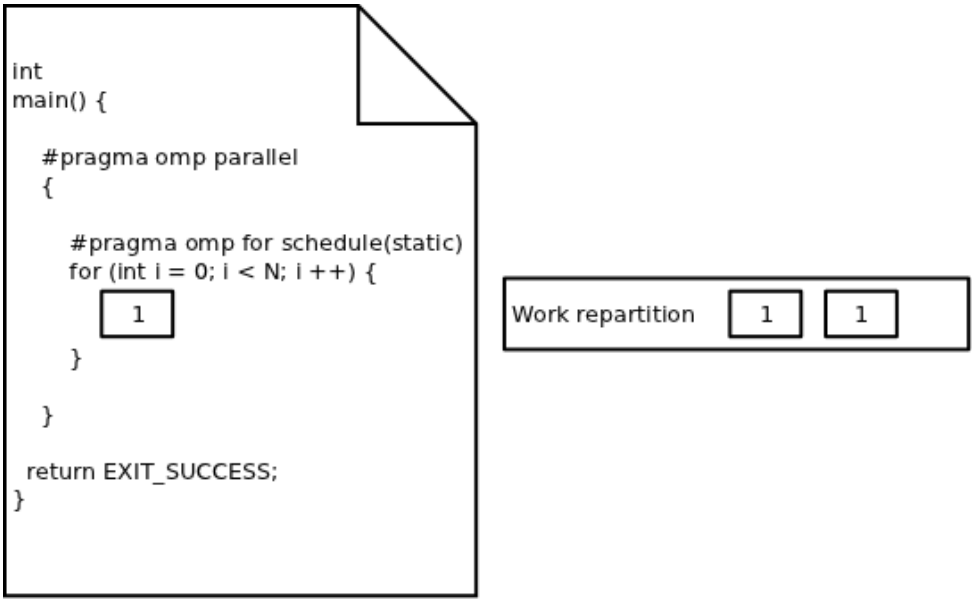

Figure 4 – Parallélisation d'une boucle de type `for` sur un bi-processeur.

Lorsqu'une région parallèle ne contient qu'une seule et unique directive `omp for`, il est possible de fusionner le tout en `omp parallel for`.

Une directive `omp for` est souvent accompagnée de la clause `schedule(type [, size])` qui précise la manière de répartir les itérations entre les différents threads.

Le paramètre optionnel `size` indique le nombre d'itérations consécutives qu'un thread doit traiter : lorsqu'il est omis, la valeur par défaut de ce paramètre est fixée à une itération (sauf dans le cas `static`, voir ci-dessous).

Le paramètre `type` désigne la façon de répartir les itérations. Ses valeurs peuvent être :

`static` : les itérations sont réparties cycliquement par blocs de taille `size`. Lorsque le paramètre `size` est absent, sa valeur est fixée par l'implémentation à N/P où N et P désignent respectivement le nombre d'itérations de la boucle et le nombre de threads disponibles dans la région parallèle. Ce type de répartition concerne les corps de boucles dont la durée de calcul est fixe (homogène) ou quasi fixe (quasi-homogène) ;

`dynamic` : les itérations sont réparties indistinctement par blocs de taille `size` : dès qu'un thread a

terminé son bloc, il commence le prochain bloc disponible. Ce type de répartition concerne les corps de boucles dont la durée de calcul varie fortement d'une itération à l'autre (hétérogène). Comme les blocs d'itérations sont implantés dans une file partagée par l'ensemble des threads exécutant la région parallèle, cette file est protégée par une exclusion mutuelle d'où un *overhead* important lié à la synchronisation. Par conséquent, le programmeur doit tenter de déterminer au mieux la valeur du paramètre `size` permettant de compenser cet *overhead* ;

`guided` : même comportement que le type `dynamic` mais l'implémentation tente d'adapter la valeur du paramètre `size` au cours des itérations (sa valeur décroit exponentiellement jusqu'à 1). Cela peut tout aussi bien réduire drastiquement la durée d'exécution de la boucle que l'aggraver fortement ;

`runtime` : le choix est reporté au moment de l'exécution en positionnant la variable d'environnement `OMP_SCHEDULE`, par exemple : `export OMP_SCHEDULE="guided, 4"`. Ce réglage est commun à toutes les clauses exploitant le type de répartition `runtime` ;

`auto` : laisse le soin à l'implémentation de choisir la meilleure répartition des itérations possible en fonction des seules informations disponibles à la compilation (le paramètre `size` n'est pas mentionné) ;

`ordered` : force l'ordre d'évaluation des itérations de la boucle comme si cette dernière était placée dans une région séquentielle. Ce type de répartition n'est utilisé qu'à des fins de *debugging* (le paramètre `size` n'est pas mentionné).

Par défaut, une barrière de synchronisation implicite est placée à la sortie de la boucle parallèle. Pour des considérations d'optimisation, il est possible de la supprimer via la clause `nowait`.

Considérons l'exemple d'une boucle séquentielle extérieure encageant une boucle parallèle intérieure. La norme actuelle d'OpenMP garantit que le thread ayant traité l'itération j de la boucle parallèle à l'itération i de la boucle séquentielle, continue de le faire à l'itération $i+1$ de cette dernière. Par conséquent, la localité des données au sein du cache privé de niveau 1 de chaque cœur est correctement exploitée : cette caractéristique est appelée « *cache affinity* ».

Enfin, la figure 5 présente la façon de paralléliser efficacement des boucles imbriquées. La clause `collapse(profondeur)` dans laquelle `profondeur` représente un niveau d'imbrication, demande au compilateur de fusionner les espaces d'itération des boucles correspondantes en un seul espace. Les conditions à respecter sont :

— des espaces d'itération n-aires (rectangulaire dans le cas 2D) ;
— des instructions uniquement localisées dans le corps de la boucle la plus profonde.

```
1  const int p = 10;
2  const int q = 20;
3  int m[p][q];
4
5  #pragma omp parallel for schedule(auto) collapse(2)
6  for (int i = 0; i < p; i ++) {
7    for (int j = 0; j < q; j ++) {
8      m[i][j] = i + j;
9    }
10 }
```

FIGURE 5 – Parallélisation de boucles imbriquées.

1.1.4 Synchronisation des threads

Les directives `omp parallel`, `omp sections` et `omp for` utilisent des barrières de synchronisation implicites qu'il est possible de lever via la clause `nowait`. Il est également possible d'utiliser les mécanismes de synchronisation explicites présentés en figure 6.

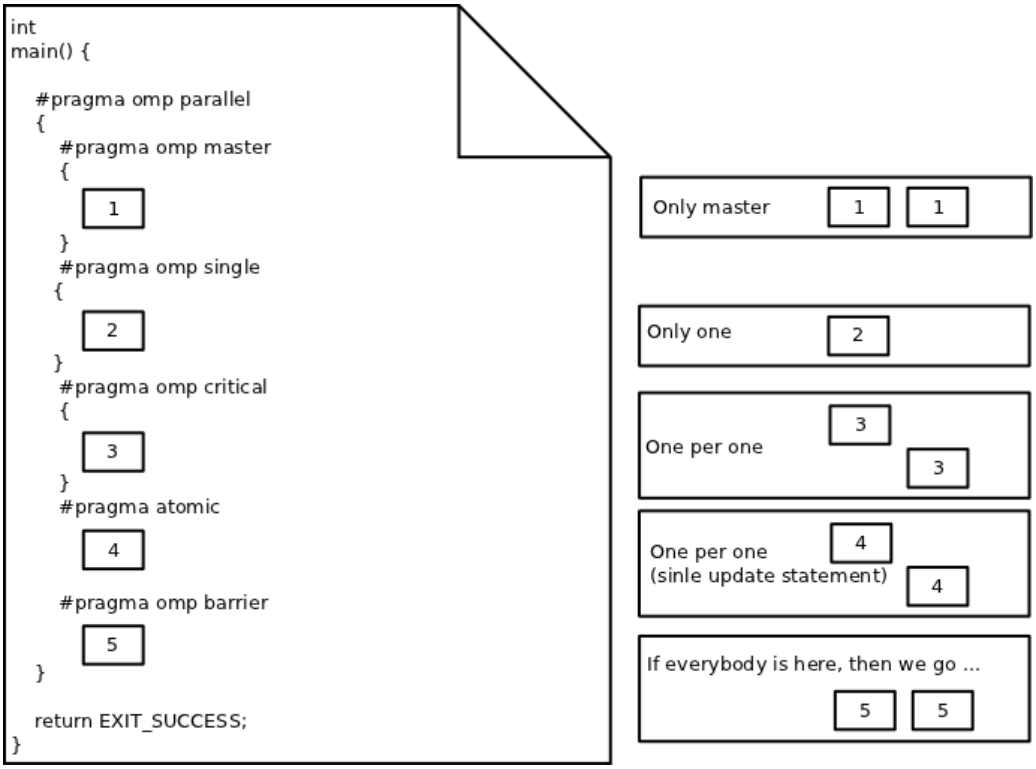

FIGURE 6 – Mécanismes de synchronisation explicites sur un bi-processeur.

Une zone de code délimitée par un bloc `omp master` n'est exécutée que par le seul thread maître. De manière analogue, une zone délimitée par un bloc `omp single` n'est exécutée que par un seul thread, en fait le premier arrivé. La directive `omp single` impose une barrière de synchronisation implicite en sortie pour que les threads qui ne travaillent pas attendent celui qui exécute le bloc. Cette barrière peut être levée via la clause `nowait`. À l'inverse, la directive `omp master` n'impose aucune barrière de synchronisation implicite.

Une zone de code délimitée par un bloc `omp critical` désigne une section critique qui ne peut être exécutée que par un seul thread à la fois. Une barrière de synchronisation explicite est placée à la sortie de ce bloc, barrière qu'il est possible de lever via la clause `nowait`. Il faut se méfier de ce mécanisme de synchronisation car pour garantir l'absence d'interblocage, les implémentations OPENMP utilisent un unique verrou pour l'ensemble des sections critiques de l'application : par conséquent, lorsqu'un thread se trouve dans l'une de ces sections (n'importe laquelle), tous les autres sont mis en attente ...

La directive `omp atomic` est destinée aux opérations d'incrémentation ou de décrémenentation, c'est à dire des expressions exploitant les opérateurs de la forme θ`=` ou θ désigne un opérateur binaire, les opérateurs de pré et post-incrémentation `++` ainsi que les opérateurs de pré et post-décrémentation `--`. Cette directive, qui ne peut être associée qu'à une seule et unique opération, exploite les possibilités de vérouillage/déverouillage du matériel pour la mener à bien. Bien que leur principe soit identique (une barrière de synchronisation implicite est placée en sortie, barrière qu'il est possible de lever via la clause

`nowait`), la directive `omp atomic` doit être systématiquement préférée à la directive `omp critical` à chaque fois que cela est possible.

La directive `omp barrier` représente une barrière de synchronisation explicite.

Bien qu'il ne s'agisse pas à proprement parler d'un mécanisme de synchronisation, la norme OpenMP propose maintenant un moyen permettant de forcer les threads qui exécutent une portion de code parallèle à l'abandonner proprement. Ce mécanisme concerne :
- les régions parallèles ;
- les sections parallèles ;
- les boucles ;
- les groupes de tâches.

et permet de faire l'économie d'un montage complexe basé sur une réduction. La syntaxe de la directive correspondante est :

`#pragma omp cancel` *`construct-type-clause`* `[ [,]` *`if-clause`* `]`

La figure 7 présente une exploitation de ce mécanisme.

```
1  #pragma omp parallel
2    {
3  #pragma omp for schedule(auto)
4      for (size_t i = 0; i < n; i ++) {
5        if (t[i] == 1) {
6          std::cout << omp_get_thread_num() << std::endl;
7  #pragma omp cancel for
8        }
9      }
10   }
```

FIGURE 7 – Mécanisme d'abandon de l'exécution.

Dans cet exemple, le premier thread qui découvre un élément du tableau `t` valant 1 affiche son numéro sur la sortie standard puis abandonne l'exécution de la boucle. Il saute alors à la barrière de synchronisation implicite placée à la sortie de cette boucle. À chaque itération, les threads qui exécutent la boucle vérifient que l'abandon n'a pas encore été activé à un point de vérification placé implicitement en début de boucle. Si tel est le cas, ils abandonnent l'exécution de la boucle et vont se placer en attente sur la barrière de synchronisation correspondante. Dans le cas contraire, ils exécutent l'itération courante.

Il est possible de fixer explicitement un point de vérification via la syntaxe suivante :

`#pragma omp cancellation point` *`construct-type-clause`*

La figure 8 présente une utilisation de cette possibilité pour l'exemple de la figure 7.

1.1.5 Attributs et manipulation des données

Dans une application C++/OpenMP, les attributs de classes sont systématiquement partagés par l'ensemble des threads exécutant une région parallèle. De même, toute variable déclarée à l'extérieur d'une région parallèle est considérée par défaut comme partagée par les threads qui exécutent cette région.

Il est possible de modifier ce statut en ajoutant les clauses suivantes à une directive `omp parallel` :

`private(a, b, c)` : une instance de chaque variable partagée `a`, `b` et `c` est créée dans la mémoire locale de chaque thread. Cependant, ces instances ne sont pas initialisées ;

```
#pragma omp parallel
  {
#pragma omp for schedule(auto)
    for (size_t i = 0; i < n; i ++) {
#pragma omp cancellation point for
      if (t[i] == 1) {
        std::cout << omp_get_thread_num() << std::endl;
#pragma omp cancel for
      }
    }
  }
```

FIGURE 8 – Point de vérification d'abandon explicite.

`firstprivate(a, b, c)` : même comportement que `private` mais les instances locales sont initialisées à partir des valeurs de `a`, `b` et `c` ;

`lastprivate(a, b, c)` : même comportement que `private` mais les variables partagées originales `a`, `b` et `c` sont mises à jour à partir de leurs homologues locales les plus récemment calculées. Cette clause doit être maniée avec beaucoup de prudence et peut concerner, par exemple, le résultat d'une boucle de calcul ;

`default(global)` : où `global` peut prendre comme valeur `private`, `firstprivate`, `shared` ou `none`. Cette clause définit la conduite à tenir pour toutes les variables déclarées à l'extérieur de la région parallèle. La clause `shared` est présentée ci-dessous. La valeur `none` impose de re-déclarer chaque variable déclarée en dehors de la région parallèle avec la clause `shared`. La clause `default` doit être vue comme un outil de contrôle strict du statut de chaque variable mis entre les mains du programmeur ;

`shared(d, e)` : annule l'effet d'une clause `default` avec les arguments `private`, `firstprivate` ou `none` pour les variables `d` et `e` ;

`threadprivate(MaClassee::attribut)` : une instance de l'attribut de classe `MaClassee::attribut` est systématiquement créée dans la mémoire locale de chaque thread pour toutes les régions parallèles rencontrées. Cependant, comme pour `private`, ces instances ne sont pas initialisées ;

`copyin(MaClasse::attribut)` : demande à ce que l'attribut de classe `MaClasse::attribut` ayant fait l'objet d'une clause `threadprivate` soit maintenant initialisé à partir de l'instance de cet attribut possédée par le thread maître ;

`proc_bind(policy)` : qui gère l'affectation des threads (thread affinity) aux différents cœurs d'une machine de type CC-NUMA en fonction de la valeur de la variable d'environnement `OMP_PLACE`, celle-ci définissant la granularité du *cluster* (nœud de calcul, processeurs multi-cœurs de ce nœud ou bien encore cœurs de ce processeur). Le paramètre `policy` peut prendre l'une des valeurs suivantes :

`master` : les threads sont affectés au même emplacement que le thread maître ;

`close` : les threads sont affectés à des emplacement consécutifs depuis celui du thread maître ;

`spread` : les threads sont répartis uniformément sur l'ensemble des emplacements.

Des clauses telles que `firstprivate` ou `copyin` permettent de recopier le contenu de données partagées dans des données locales aux threads. OpenMP propose également des mécanismes permettant d'effectuer le chemin inverse :

`copyprivate(a, b)` : systématiquement associée aux directives `omp master` ou `omp single`, cette clause permet au thread qui exécute le bloc de diffuser, à l'issue, le contenu de ses variables locales `a` et `b` aux autres threads qui exécutent la même région parallèle ;

`flush(a, b)` : demande à ce que la cohérence des caches soient rétablie pour toutes les instances locales des variables partagées `a` et `b`. Cette directive doit être maniée avec prudence et n'existe

que pour des considérations d'optimisation. Elle peut être utilisée sans argument auquel cas elle s'applique à toutes les données locales aux threads ;

`reduction(opérateur : a)` : effectue une réduction sur la variable partagée `a`. Une réduction est une opération associative appliquée à une variable partagée (scalaire ou tableau). Cette opération peut être arithmétique ou logique. Chaque thread calcule un résultat partiel indépendant des autres (variable locale initialisée à l'élément neutre de l'opération). L'ensemble des threads se synchronise ensuite pour mettre à jour la variable partagée concernée en combinant les exemplaires locaux avec la valeur initiale de cette variable.

Le programmeur peut définir ses propres opérateurs de réduction. Pour ce faire, il doit tout d'abord déclarer les caractéristiques de cet opérateur avant de pouvoir l'exploiter dans une clause `reduction`. Cette déclaration est effectuée par le biais d'une directive dont la syntaxe est :

```
#pragma omp declare reduction( reduction-identifier : typename-list : combiner )
                        [ initializer(initialize-clause) ]
```

avec :

`reduction-identifier` : le nom que donne le programmeur à son opérateur ;

`typename-list` : une liste de types sur lesquels peuvent être appliqués l'opérateur, chaque nom étant séparé du précédent par une virgule ;

`combiner` : une expression décrivant comment combiner la copie locale détenue par chaque thread avec le nouvel élément. La copie locale est désignée par le mot-clé `omp_out` tandis que le nouvel élément est désigné par `omp_in` ;

`initialize-clause` : cette clause optionnelle décrit la façon dont la (ou les) copie locale (désignée par le mot-clé `omp_priv`) à chaque thread doit être initialisée. La donnée partagée correspondante est désignée par le mot-clé `omp_orig`. En l'absence de cette clause, chaque donnée locale est initialisée par défaut.

La figure 9 présente la définition *inline* d'une classe `Long` destinée à illustrer notre propos. Cette classe propose un constructeur par défaut/logique, un accesseur ainsi qu'une surcharge de l'opérateur `+` qui retourne la somme de deux instances de la classe.

```
1  class Long {
2  public:
3    Long(const long& value = 0): value_(value) { }
4  public:
5    const long& value() const { return value_; }
6  public:
7    Long operator+(const Long& rhs) const { return Long(value_ + rhs.value_); }
8  private:
9    const long value_;
10 };
```

FIGURE 9 – Définition *inline* de la classe `Long`.

Nous souhaitons effectuer des réductions avec l'opérateur `+` sur le type `Long`. Bien que cet opérateur soit explicitement surchargé dans notre classe, la norme OPENMP nous interdit de le faire figurer dans une clause `reduction` puisque `Long` n'est pas un type fondamental. La seule solution consiste à déclarer explicitement un opérateur de réduction puis à l'exploiter.

La figure 10 présente la déclaration du notre. Ici, l'opérateur de réduction `sumLong` n'est autorisé à travailler que sur le type `Long`. Comme ce dernier surcharge l'opérateur `+`, la réduction est tout simplement définie comme : `omp_out = omp_out + omp_in`. Afin de rester compatible avec les réductions sur

les types fondamentaux, nous initialisons volontairement chaque copie locale avec la valeur initiale de la variable partagée correspondante.

```
1 #pragma omp declare reduction(sumLong : Long : omp_out = omp_out + omp_in) \
2   initializer(omp_priv(omp_orig))
```

FIGURE 10 – Déclaration de notre opérateur de réduction `sumLong` sur le type `Long`.

Ne reste plus qu'à exploiter classiquement notre opérateur, ce que montre la figure 11.

```
1   Long somme;
2 #pragma omp parallel for schedule(auto) reduction(sumLong : somme)
3   for (long i = 0; i < n; i ++) {
4     somme = somme + t[i];
5   }
```

FIGURE 11 – Exploitation de l'opérateur de réduction `sumLong` de la figure 10.

1.2 Vectorisation

En règle générale, la vectorisation est activée via les options d'optimisation transmises au compilateur. Elle peut également être mise en œuvre par des directives OPENMP dont la syntaxe est de l'une des formes suivantes :

```
#pragma omp simd [ clause-list ]
#pragma omp for simd [ clause-list ]
```

La première forme demande à ce que l'ensemble de la boucle `for` qui suit soit simplement vectorisée. Dans le second cas, les itérations de cette boucle sont réparties équitablement entre les threads disponibles et les sous-boucles correspondantes sont vectorisées. Les clauses qui peuvent apparaitre derrière une directive de vectorisation sont :

`safelen(n)` : où `n` est un entier positif indiquant le nombre d'itérations maximum de la boucle pouvant être traitées simultanément. Par défaut, cette longueur est tout simplement celle de l'espace d'itération de la boucle ;

`linear(list [: linear-step ])` : une variable partagée définie à l'extérieur d'une boucle vectorielle ne doit pas voir sa valeur modifiée à l'intérieur de cette boucle, sans quoi le comportement du programme correspondant n'est pas défini. Si le programmeur veut modifier le contenu de cette variable à l'intérieur de sa boucle alors il peut le faire via la clause `linear` ou la clause `reduction` que nous avons déjà rencontrée. La clause `linear` indique tout simplement que la valeur de la variable en question dépend de l'indice de boucle selon la relation :

$$x_i = x_{orig} + i \times linear - step$$

La valeur du pas par défaut est 1 ;

`aligned(list [: alignment ])` : qui indique que les constantes ou variables définies dans la liste possèdent des alignements particuliers en mémoire. La valeur par défaut est celle prévue par l'architecture pour ces données ;

`private` : déjà décrite ;

`lastprivate` : déjà décrite ;

`reduction` : déjà décrite ;

`collapse` : déjà décrite.

La figure 12 présente un exemple de vectorisation.

```
  somme = 0;
#pragma omp simd linear(somme : 2)
  for (size_t i = 0; i < n; i ++) {
    t[i] = somme;
    somme += 2;
  }
```

FIGURE 12 – Exemple de vectorisation

Lorsqu'un corps de boucle invoque une fonction avec des arguments de type scalaire, le compilateur ne peut pas la vectoriser car il faudrait que ces arguments soient des tableaux. La norme propose un mécanisme permettant de traiter ce problème : il suffit de déclarer la fonction concernée comme étant destinée à être invoquée dans une boucle vectorielle. Le compilateur génère alors autant de fonctions particulières que d'appels dans la boucle vectorisée. Cette déclaration est effectuée via la directive suivante placée devant la déclaration (ou la définition) de la fonction :

`#pragma omp declare simd [` *`clause-list`* `]`

Les clauses pouvant apparaître dans cette déclaration sont :

`simdlen(n)` : génère une fonction dont les arguments sont des tableaux de taille `n` ;

`uniform(argument-list)` : indique que tous les paramètres de la fonction mentionnés dans la liste se comportent comme des constantes ;

`inbranch` : indique que la fonction doit être invoquée dans un bloc `if then else`. Par défaut, la fonction peut être invoquée dans ou à l'extérieur d'une alternative ;

`notinbranch` : indique que la fonction ne doit pas être invoquée à l'intérieur d'un bloc `if then else`. Par défaut, la fonction peut être invoquée dans ou à l'extérieur d'une alternative ;

`linear(argument-list [: linear-step ])` : déjà décrite ci-dessus et s'applique aux arguments de la fonction mentionnés dans la liste ;

`aligned(argument-list [: alignment ])` : déjà décrite ci-dessus et s'applique aux arguments de la fonction mentionnés dans la liste.

La figure 13 présente un exemple d'invocation de fonction au sein d'une boucle vectorielle.

```
#pragma omp declare simd uniform(x)
int
square(const int& x) {
  return x * x;
}

...

#pragma omp simd
  for (size_t i = 0; i < n; i ++) {
    t[i] = square(i);
  }
```

FIGURE 13 – Invocation d'une fonction au sein d'une boucle vectorielle.

1.3 Tâches

Les tâches sont un apport majeur dans la norme OpenMP, apport destiné à pallier les faiblesses des versions antérieures en matière de parallélisation d'algorithmes récursifs, d'algorithmes manipulant des structures de données irrégulières ou bien encore de parallélisation de boucles d'un autre type que `for`.

Considérons l'algorithme récursif présenté en figure 3. Avant l'introduction des tâches, sa parallélisation passait par les sections parallèles. Or, le coût de création des threads demeurait suffisamment dissuasif pour ne pas recourir au parallélisme imbriqué. Par conséquent, le programmeur ne pouvait pas exploiter tout le parallélisme potentiel de cet algorithme. La figure 14 présente la parallélisation du même algorithme à l'aide de tâches.

```
1  void
2  quicksort(double t[], const int& a, const int& b) {
3    const double pivot = t[(a + b) / 2];
4    int i = a, j = b;
5    do {
6      while (t[i] < pivot) {
7        i ++;
8      }
9      while (t [j] > pivot) {
10       j --;
11     }
12     if (i <= j) {
13       std::swap(t[i], t[j]);
14       i ++;
15       j --;
16     }
17   } while (i <= j);
18 #pragma omp task shared(a) if (b - a > 100)
19   if (a < j) {
20     quicksort(t, a, j);
21   }
22 #pragma omp task shared(b) if (b - a > 100)
23   if (i < b) {
24     quicksort(t, i , b);
25   }
26 #pragma omp taskwait
27 }
28
29 void
30 parquicksort(double t[], const int& a, const int& b) {
31 #pragma omp parallel
32   {
33 #pragma omp single
34     quicksort(t, a, b);
35   }
36 }
```

Figure 14 – Parallélisation d'un quicksort à l'aide de tâches.

La fonction `parquicksort` constitue le point d'entrée de cet algorithme. Nous y trouvons une région parallèle englobant un bloc `omp single`. L'appel de la fonction `quicksort` sur l'ensemble du tableau à trier est donc effectué par un seul et unique thread. Chaque appel de la fonction `quicksort` a pour effet non pas son exécution mais la création de deux tâches, l'une rappelant la fonction `quicksort` sur la moitié gauche du tableau et l'autre sur la moitié droite. Ces tâches sont placées au fur et à mesure

de leur construction dans une file de tâches. Lorsque la dernière tâche est créée, le bloc `omp single` se termine et tous les threads qui exécutent la région parallèle commence à exécuter les tâches placées dans la file. La barrière de synchronisation `omp taskwait` permet à une tâche d'attendre la fin de l'exécution des tâches qu'elle a engendrées.

L'exemple de la figure 14 illustre la philosophie des tâches en OpenMP. Une tâche est à l'image d'un thread, un bloc d'instructions avec un environnement, c'est à dire des données privées. Lorsqu'un thread exécutant une région parallèle rencontre un constructeur de tâche, il créé une instance de cette tâche puis la place dans une file associée à la région parallèle. Si ce thread n'exécute pas un bloc `omp single` alors l'exécution de la tâche peut démarrer immédiatement ou plus tard. Cette exécution est assurée par l'équipe de threads qui exécute la région parallèle englobante. Une tâche est affectée à un thread unique et seul ce dernier, par défaut, peut l'exécuter. Il peut cependant la mettre en sommeil pour la reprendre ultérieurement. Lorsqu'une clause `if (...)` est associée au constructeur de tâche et que la condition correspondante est fausse alors le thread qui rencontre ce constructeur doit exécuter immédiatement la tâche correspondante.

Le statut des données manipulées par une tâche est par défaut `firstprivate` pour toutes les données locales à la région parallèle, les autres ayant par défaut le statut `shared`. Une tâche peut demander à partager une donnée `data` locale à une région parallèle en faisant suivre son constructeur de la clause `shared(data)`.

La barrière de synchronisation implicite placée à la fin d'une région parallèle garantit que toutes les tâches associées à cette région seront exécutées quant la région se terminera. Cependant, une tâche de cette région (nous dirons une tâche mère) peut donner naissance à d'autre tâches (nous dirons des tâches filles) en leur fournissant des arguments qui correspondent à certaines de ses données locales. L'exécution des tâches pouvant être différée, il est possible que l'exécution de la tâche mère se termine alors que celle des tâches filles n'a pas encore commencé. Par conséquent, si des alias ou des pointeurs ont été fournis aux tâches filles lors de leur création, ces derniers ne référencent plus rien lorsque les tâches filles commencent leur exécution. De fait, il faut absolument que la tâche mère soit exécutée après ses tâches filles. Cet ordonnancement est assuré par la barrière de synchronisation explicite `omp taskwait`.

Ainsi, dans l'exemple de la figure 14, les paramètres de la fonction `quicksort` sont respectivement un tableau et deux alias. La clause `firstprivate` est implicite pour le tableau mais seul le pointeur associé (l'adresse de base) est capturée par la tâche et non pas le contenu. Pour les deux alias, le compilateur exige qu'ils fassent l'objet d'une clause `shared`, ce qui n'est pas un problème puisqu'il s'agit de constantes. Lorsque la tâche mère créée deux tâches filles, elle leur fournit (via des alias) une nouvelle borne de tableau calculée à partir de l'ancienne. Par conséquent, cette nouvelle borne est locale à la tâche mère et celle-ci doit donc suspendre son exécution afin de laisser terminer ses tâches filles. Notons que la directive `omp taskwait` ne concerne que les descendants directs et pas les descendants des descendants.

Il est possible, pour des considérations d'optimisation (et plus précisément d'équilibrage de charge), de demander à ce qu'une tâche puisse commencer son exécution avec un thread, puis, si elle est mise en sommeil, de pouvoir être réveillée et ré-affectée à un autre thread qui serait immédiatement disponible. Cette possibilité, à manier avec beaucoup de prudence, est activée via la clause `untied`. Les tâches peuvent être créées dans n'importe quelle mécanisme de répartition du travail (boucle de type `for`, sections parallèles) au sein d'une région parallèle. Elles permettent également de paralléliser des boucles de type `for` dont la borne supérieure est inconnue et plus généralement n'importe quel type de boucle.

La clause `reduction` n'a pas de sens pour les tâches : il faut donc utiliser la directive `atomic` pour obtenir le résultat correspondant.

Notons enfin que la norme ne précise rien sur la façon dont son gérée les tâches en interne, notamment si la file de tâches associée à une région parallèle est unique ou au contraire distribuée sur l'ensemble des cœurs de la machine. Le problème d'une file unique est que pour un nombre élevé de threads, celle-ci

devient un sérieux point de contention. Ce problème peut être résolu si le scheduler utilisé par l'implémentation exploite un algorithme de work-stealing emprunté à Cilk et repris par Intel Threading Building Blocks.

1.3.1 Dépéndances entre tâches

La clause `depend( dependence-type : list )` permet d'exprimer des dépendances fines entre tâches affectées à une même régions parallèles. Ses paramètres sont :

- *`list`* est une liste de variables et/ou de constantes pouvant contenir des tableaux ou des tranches de tableaux ;
- *`dependence-type`* exprime le type de dépendance liant les tâches filles à leur mère. Les valeurs permises sont :

  `in` qui indique que la tâche (appelons la $\mathcal{T}$) qui va être créée ne pourra être exécutée tant que toutes les tâches définies précédemment et mentionnant au moins dans leur clause `depend` l'une des variables ou constantes de la clause de $\mathcal{T}$ avec les modes `out` ou `inout`, n'auront pas terminé ;

  `out` qui indique que la tâche $\mathcal{T}$ à créer ne pourra être exécutée tant que toutes les tâches définies précédemment et mentionnant au moins une variable ou constante de $\mathcal{T}$ (quel que soit le mode) n'auront pas terminé ;

  `inout` : même comportement que le mode `out`.

Considérons le petit exemple de la figure 15 :

- la tâche `A` est définie la première et sa clause `depend` mentionne deux variables partagées `x` et `y` en mode `out`. Par conséquent, le compilateur regarde si d'autres tâches définies précédemment dans la même région parallèle mentionne ces deux variables dans leur clause `depend` quel que soit leur mode. Comme il n'y en a pas, la tâche `A` peut être exécuté immédiatement ;
- la tâche `B`, définie en second position, mentionne la variable `x` avec le mode `in`. Par conséquent, `B` dépend de `A` ;
- la tâche `C`, définie en troisième position, mentionne la variable `y` avec le mode `in`. Par conséquent, `C` dépend de `A` mais pas de `B` : les deux tâches peuvent donc être exécutées simultanément dès que `A` a terminé ;
- la tâche `D`, définie en dernière position, mentionne les deux variables en mode `out`. Par conséquent, `D` dépend à la fois de `B` et de `C` : c'est donc la tâche qui sera exécutée en dernier.

1.3.2 Groupes de tâches

En règle générale, les tâches associées à une région parallèle peuvent être considérée comme un même groupe et toutes se termineront sur la barrière de synchronisation placée à la sortie de cette région. Cependant, il est parfois nécessaire de définir explicitement un groupe de tâches afin de lui faire faire une action particulière. Dans ce cas, comme une tâche isolée, il doit être possible de :

- interrompre toutes les tâches du groupe ;
- garantir que toutes les tâches de ce groupes, initiales comme celles créés dynamiquement, ont achevé leur exécution à un point donné du code.

La notion de groupe de tâches était absente des normes d'OpenMP antérieure à la version 4.0. Il n'existait pas de clause d'abandon et la seule directive permettant de synchroniser des tâches sur un point particulier du code était `omp taskwait`. Or, cette dernière ne synchronise que les tâches définies au niveau de la région parallèle mais pas leur descendantes. Ainsi, dans l'exemple de la figure 16, seules les tâches `T1` et `T2` seront synchronisées sur la barrière `taskwait` mais pas `T3` puisque celle-ci est créée par `T2`.

La nouvelle directive `taskgroup` permet de résoudre les deux problèmes mentionnée ci-dessous, ainsi que le montre la figure 17. Dans ce nouvel exemple, les tâches `T1`, `T2` et `T3` sont considérées comme appartenant au même groupe. Par conséquent, elle achève leur exécution et s'attendent sur la barrière de synchronisation implicite placée à la sortie de la directive `omp taskgroup`.

```
1   // Variable partagée x.
2
3   int x = 0, y = 100;
4
5  #pragma omp parallel
6   {
7
8     // Tâche A.
9
10 #pragma omp task depend (out : x, y)
11    {
12      x = 1;
13      y = 101;
14    }
15
16    // Tâche B.
17
18 #pragma omp task depend(in : x)
19    {
20      x = 2;
21    }
22
23    // Tâche C.
24
25 #pragma omp task depend(in : y)
26    y = 202;
27
28    // Tâche D.
29
30 #pragma omp task depend(out : x, y)
31    {
32      x = 3;
33      y = 303;
34    }
35
36  }
```

FIGURE 15 – Dépendances entre tâches.

```
#pragma omp parallel
  {
#pragma omp task
    {
      // T1.
    }

#pragma omp task
    {
      // T2.

#pragma omp task
      {
        // T3.
      }

    }

  ...
  // Point du code.

#pragma omp taskwait
  ...
  }
```

FIGURE 16 – Utilisation de la directive `omp taskwait`.

```
#pragma omp parallel
  {

#pragma omp taskgroup
    {

#pragma omp task
      {
        // T1.
      }

#pragma omp task
      {
        // T2.

#pragma omp task
        {
          // T3.
        }
      }

    } // omp taskgroup

  // Point du code.
  ...
  }
```

FIGURE 17 – Utilisation de la directive `taskgroup`.

1.4 Pièges à éviter

Sont résumés ici les pièges les plus courants.

1.4.1 *Race condition*

Il s'agit du premier piège dans lequel tombe les débutants mais aussi les programmeurs plus chevronnés (lorsqu'ils utilisent des objets par exemple).

La *race condition* désigne soit l'écriture quasi-simultanée, soit la lecture/écriture quasi-simultanée d'une variable partagée par plusieurs threads. La figure 18 présente un exemple non trivial.

```
1  class MaClasse {
2  public:
3    MaClasse const int& attribut) : attribut_(attribut) { }
4  public:
5    const int& attribut() const { return attribut_; }
6  public:
7    void add(const int& valeur) { attribut_ += valeur; }
8  protected:
9    int attribut_;
10 };
11 ...
12 int
13 main ( ) {
14   MaClasse objet(1);
15 #pragma omp parallel
16   {
17       objet.add(10);
18   }
19   return EXIT_SUCCESS;
20 }
```

FIGURE 18 – Race condition sur l'attribut `MaClasse::attribut_`.

Dans cet exemple, l'instance `objet` de classe `MaClasse` est partagée par tous les threads qui exécutent la région parallèle. Par conséquent, l'attribut `attribut_` de l'instance objet l'est également. Or, chaque thread invoque la méthode `add` qui modifie la valeur de cet attribut. La solution consiste ici soit à :

- encager l'invocation de la méthode `add` dans un bloc `omp critical` : c'est la solution coté utilisateur si la classe n'est pas prévue pour un usage dans un contexte multithreadé ;
- faire précéder l'expression `attribut_ += valeur` de la directive `omp atomic` : c'est la solution coté développeur si la classe est prévue pour être exploitée dans un contexte multithreadé.

1.4.2 *False sharing*

Le *false sharing* désigne l'écriture par plusieurs threads de données stockées sur la même ligne d'un cache partagé. Lorsqu'un thread accède à une donnée de cette ligne en lecture, le processeur sous-jacent la recopie dans son propre cache. Si un autre thread modifie la donnée dans le cache partagé alors toutes les données de la ligne doivent être invalidées dans les caches locaux. Par conséquent, si plusieurs threads modifient les données de la ligne, il se produit un phénomène de ping pong entre le cache partagé et les caches locaux, ce qui peut nuire gravement aux performances de l'application.

La figure 19 présente un exemple trivial. Dans ce dernier, nous déclarons un tableau dont la taille est le nombre de threads par défaut exécutant une région parallèle. Chaque thread récupère ensuite son identifiant pour mettre à jour la case correspondante du tableau.

```
1  int threads ;
2
3  #pragma omp parallel
4  #pragma omp single
5  threads = omp_get_num_threads();
6
7  int valeurs[threads];
8  #pragma omp parallel
9    {
10     const int id = omp_get_thread_num();
11     valeurs[id] = id * 10 + 5;
12   }
```

FIGURE 19 – *False sharing* induit par un tableau.

Il est évident ici que le tableau étant de petite taille, ses éléments partageront une même ligne du cache partagé, provoquant ainsi du *false sharing* lors de chaque écriture. Le false sharing est un problème difficile à éradiquer mais la conduite à tenir pour l'éviter consiste à utiliser un maximum de données locales aux threads.