



École Nationale Supérieure d'Ingénieurs de CAEN

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

TP n°1

Niveau	2 ^{ème} année
Parcours	Informatique
Unité d'enseignement	2I2AC3 - Architectures parallèles
Responsables	Emmanuel CAGNIOT Emmanuel.Cagniot@ensicaen.fr Hugo DESCoubes Hugo.Descoubes@ensicaen.fr

1 Multi-threading en OPENMP

Le modèle de programmation naturel d'une machine SMP est le modèle à parallélisme de processus légers communicants 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. 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 sont eux aussi non 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 Architecture Review Board, seul organisme habilité à les faire évoluer. Sa dernière norme 4.0 a été publiée courant juillet 2013.

1.1 Éléments de base

1.1.1 Création des threads

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 notre cas, 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.

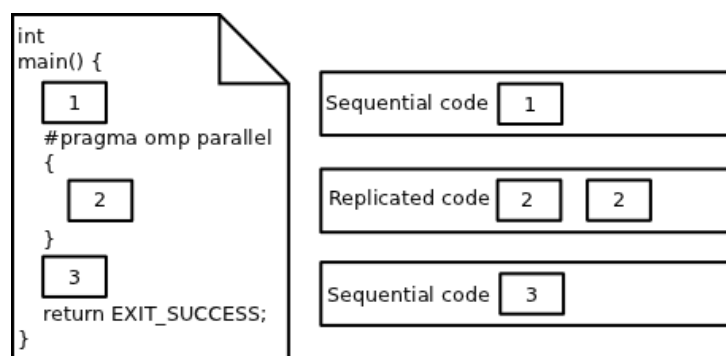


FIGURE 1 – Régions séquentielles et parallèles sur un bi-processeur.

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 la fonction `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 booléenne. Si cette condition est vérifiée alors la région est considérée comme parallèle. Dans le cas contraire, elle est considérée comme séquentielle. De même, il est possible de fixer le nombre de threads qui vont exécuter une région parallèle en adjoignant la clause `num_threads(n)` où `n` désigne le nombre maximum de threads autorisés.

1.1.2 Partage de la charge de travail

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

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`.

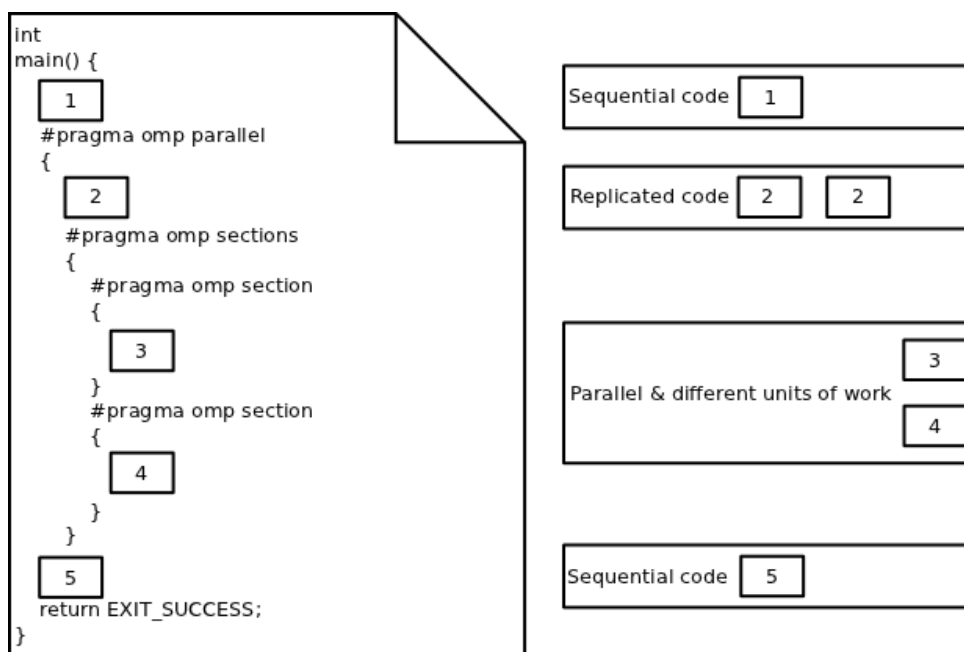


FIGURE 2 – Sections parallèles sur un bi-processeur.

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.

Boucles de type for : ces boucles ne peuvent être parallélisées que sous deux conditions. Primo, 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 calculé directement. Secundo, les itérations de la boucle doivent être indépendantes les unes des autres, c'est à dire qu'une itération ne doit pas nécessiter des résultats calculés lors des itérations précédentes. 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 ou SPMD.

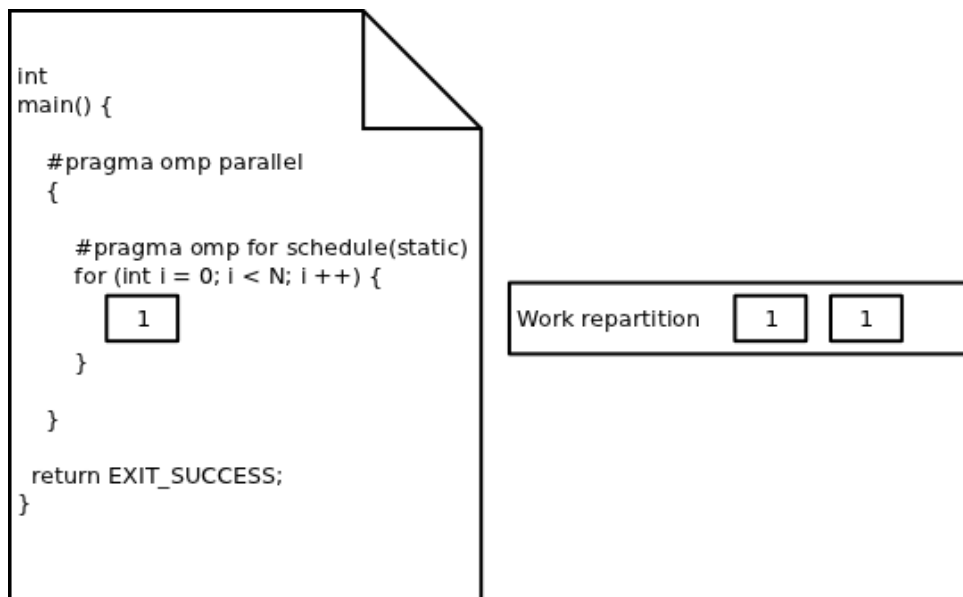


FIGURE 4 – Parallélisation d'une boucle de type `for` sur un bi-processeur.

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

static : les itérations sont réparties cycliquement par blocs de taille `size`. Ce type de répartition concerne les corps de boucles dont le temps de calcul est fixe ;

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 le temps de calcul n'est pas fixe (par exemple des appels de fonctions). Comme les blocs d'itérations sont implantés dans une file partagée par l'ensemble des threads, celle-ci 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 la taille des blocs décroît exponentiellement jusqu'à `size` ;

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.

1.1.3 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.

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. Ces deux directives utilisent une barrière de synchronisation implicite pour que les threads qui ne travaillent pas attendent celui qui exécute le bloc. Cette barrière peut être levée via la clause **nowait**. Bien que **single** et **master** soient relativement semblables, la première doit être la règle et la seconde l'exception. En effet, dans le premier cas, la zone est exécutée par n'importe quel thread pourvu qu'il soit le premier arrivé. Dans le second cas, nous imposons que cette zone ne soit exécutée que par le seul thread maître. Or, ce dernier peut ne pas encore être disponible alors que d'autres threads le sont et attendent.

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 même et 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 ...

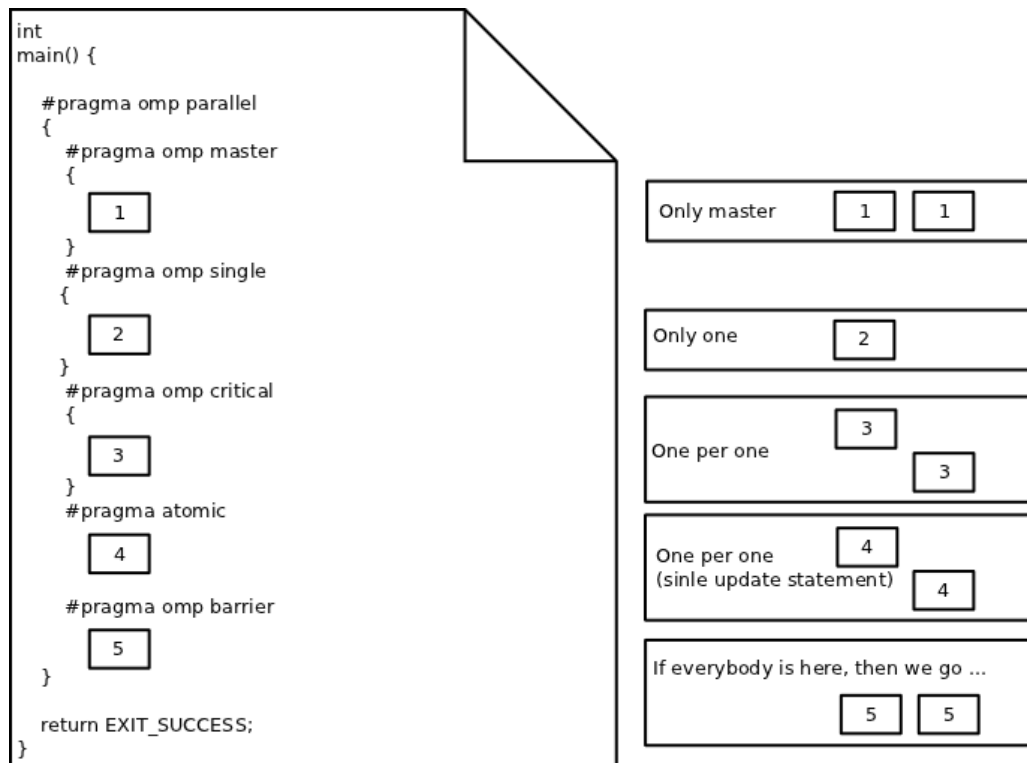


FIGURE 6 – Mécanismes de synchronisation explicites sur un bi-processeur.

La directive `omp atomic` est destinée aux opérations d'incrément ou de décrémentation, c'est à dire des expressions exploitant les opérateurs de la forme $\theta =$ 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 verrouillage/déverrouillage 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.

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

```

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.

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 8.

```

1 #pragma omp parallel
2 {
3 #pragma omp for schedule(auto)
4     for (size_t i = 0; i < n; i++) {
5 #pragma omp cancellation point for
6         if (t[i] == 1) {
7             std::cout << omp_get_thread_num() << std::endl;
8 #pragma omp cancel for
9         }
10     }
11 }

```

FIGURE 8 – Point de vérification d’abandon explicite.

1.1.4 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 cependant 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 ;

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(MaClasse::attribut) : une instance de l'attribut de classe **MaClasse::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 multicœurs 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. Cette opération peut être arithmétique ou logique. Chaque thread calcule un résultat partiel indépendant des autres (variable locale). L'ensemble des threads se synchronise ensuite pour mettre à jour le résultat final. La variable sur laquelle porte la réduction ne peut être un tableau ni un élément de tableau. En outre, l'instance de la variable **a** locale au thread doit être initialisée à l'élément neutre pour l'opérateur concerné.

Le programmeur dispose maintenant de la possibilité de 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-identifiant : typename-list : combiner )
                               [ initializer(initialize-clause) ]
```

avec :

reduction-identifiant : 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 protected :
9   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

Dans toute optimisation d’un code de calcul, la vectorisation, c’est à dire l’optimisation des boucles via les opérateurs câblés pipelinés et/ou les jeux d’instructions SIMD, doit toujours précéder la parallélisation. La mise en œuvre de la vectorisation en OPENMP est très similaire à ces des boucle parallèles, la syntaxe étant 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 apparaître 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.

```

1  somme = 0;
2  #pragma omp simd linear(somme : 2)
3  for (size_t i = 0; i < n; i++) {
4      t[i] = somme;
5      somme += 2;
6  }
```

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.

```
1 #pragma omp declare simd uniform(x)
2 int
3 square(const int& x) {
4     return x * x;
5 }
6
7 ...
8
9 #pragma omp simd
10 for (size_t i = 0; i < n; i++) {
11     t[i] = square(i);
12 }
```

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.

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 crée 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`. Cependant, si la région parallèle englobante déclare certaines données avec la clause `shared`, alors ces données sont considérées comme partagées par la tâche.

```

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 barrière de synchronisation implicite placée à la fin d'une région parallèle garantit que toutes les tâches associées à cette région seront exécutées quand la région se terminera. Cependant, une tâche de cette région (nous dirons une tâche mère) peut donner naissance à d'autres 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 crée 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 quel 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.

Notons enfin que la norme ne précise rien sur la façon dont son gère 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 Dépendances entre tâches

La clause `depend(dependence-type : list)` permet d'exprimer des dépendances fines entre tâches affectées à une même région 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  // 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.

1.3.2 Groupes de tâches

En règle générale, les tâches associées à une région parallèle peuvent être considérées 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 créées 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.

```

1  #pragma omp parallel
2  {
3  #pragma omp task
4      {
5          // T1.
6      }
7
8  #pragma omp task
9      {
10         // T2.
11
12     #pragma omp task
13         {
14             // T3.
15         }
16
17     }
18
19     ...
20     // Point du code.
21
22 #pragma omp taskwait
23     ...
24 }
```

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

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`.

1.4 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. La race condition désigne l'écriture simultanée d'une variable partagée par plusieurs threads. La figure 18 présente un exemple non trivial.

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  #pragma omp parallel
2  {
3
4  #pragma omp taskgroup
5  {
6
7  #pragma omp task
8  {
9      // T1.
10 }
11
12 #pragma omp task
13 {
14     // T2.
15
16 #pragma omp task
17 {
18     // T3.
19 }
20 }
21
22 } // omp taskgroup
23
24 // Point du code.
25 ...
26 }

```

FIGURE 17 – Utilisation de la directive `taskgroup`.

```

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_`.

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.

2 Exercice

L'algorithme `count_if` (module `algorithm` de la bibliothèque standard C++) permet de comptabiliser les éléments d'un conteneur satisfaisant un prédicat unaire. La figure 20 présente la définition de cet algorithme.

Le prédicat unaire `pred` est invoqué sur tous les éléments de l'intervalle `[first, last[`, `first` et `last` étant deux itérateurs de types `InputIterator`. Rappelons que les opérations permises par ce type d'itérateur sont la pré-incrémentation (`++ first`), la post-incrémentation (`first ++`), le dé-référencement `*first`, le test d'égalité (`first == last`) et celui de l'inégalité (`first != last`).

Nous souhaitons paralléliser l'algorithme `count_if` via OpenMP. Cependant, les problèmes posés sont multiples :

1. la condition de continuation utilise un test d'inégalité (`first != last`). Comme les itérateurs de classe `InputIterator` ne proposent que des tests d'égalité ou d'inégalité, il n'est à priori pas possible de transformer la boucle `while` en une boucle `for` puis de paralléliser cette dernière via une directive de type `omp for`;
2. le nombre d'éléments de l'intervalle `[first, last[` ne peut être calculé directement (il faut parcourir les éléments correspondants afin de les comptabiliser);
3. la nature du prédicat unaire `pred` est inconnue (s'agit-il d'une fonction dont le temps de calcul est fixe quel que soit son argument ou au contraire d'une fonction dont le temps de calcul dépend de son argument?).

```

1 template < typename InputIterator , typename UnaryPredicate >
2 typename iterator_traits<InputIterator >::difference_type
3 count_if(InputIterator first , InputIterator last , UnaryPredicate pred) {
4     typename iterator_traits<InputIterator >::difference_type ret = 0;
5     while (first != last) {
6         if (pred(*first)) ret ++;
7         first ++;
8     }
9     return ret;
10 }

```

FIGURE 20 – Algorithme `count_if` de la bibliothèque standard C++.

Nous sommes à priori mal partis mais tout n'est cependant pas si sombre. En effet, l'algorithme `count_if` peut être instancié par des itérateurs de classe `InputIterator` mais également pas des itérateurs de classes dérivées, notamment `RandomAccessIterator`. Ces derniers sont l'apanage des structures de données où l'accès à un élément via son rang s'effectue en temps constant grâce à l'utilisation d'une zone de mémoire contiguë. Sont concernés les tableau de capacité fixe (classiques ou de classe `array`) et ceux de capacité dynamique (classe `vector`). Outre les possibilités des `InputIterator`, les `RandomAccessIterator` proposent, entre autres, les opérateurs relationnels manquants (<, <=, > et >=) ainsi que la différence entre deux itérateurs (`last - first`) dont le résultat est le nombre d'éléments de l'intervalle semi-ouvert correspondant. Par conséquent, il est possible de proposer une spécialisation relativement simple de l'algorithme `count_if` pour les `RandomAccessIterator`. Pour toutes les autres classes d'itérateurs (le cas général), nous allons devoir nous torturer sérieusement les méninges ...

Deux possibilités d'implémentation s'offrent à nous :

1. sous forme de fonctions génériques (implémentation analogue à celle du module `algorithm` de la bibliothèque standard C++);
2. sous forme de méthodes de classes (mot-clé `static`) génériques d'une classe non générique (implémentation analogue à celle des classes `Arrays` ou `Collections` de la bibliothèque standard JAVA).

Nous optons pour la seconde solution qui offre (selon nous) d'avantage de sécurité grâce à l'utilisation des droits d'accès.

Architecture incomplète

L'archive `tp1.tar.gz`, accessible via la page web décrivant l'unité d'enseignement, contient un squelette incomplet de l'application à réaliser. Sa structure est la suivante :

`src/include/Metrics.hpp` : fichier en-tête contenant la déclaration d'une classe `Metrics` permettant de calculer des facteurs d'accélération et d'efficacité;

`src/include/CountIf.hpp` : fichier en-tête contenant le squelette incomplet de notre version parallèle de l'algorithme `count_if`. Cette classe non générique propose une méthode générique `apply` qui opère en deux temps :

1. elle instancie une sous-classe générique `Impl` avec le type des itérateurs reçus (il existe une sous-classe `Impl` dédiée au cas général des `InputIterator` et une autre dédiée au cas particulier des `RandomAccessIterator`);
2. elle invoque la méthode générique `apply` de cette sous-classe.

La sous-classe `Impl` dédiée aux `InputIterator` propose trois méthodes génériques `strategyA`, `strategyB` et `strategyC`, chacune implémentant une stratégie de parallélisation spécifique. Sa méthode `apply` invoquera l'une des ces trois méthodes afin d'évaluer la pertinence de chaque.

`src/Metrics.cpp` : définition de la classe `Metrics`;

`src/testCountIf.cpp` : programme de démonstration de l'algorithme parallèle `CountIf`. Cette application teste (calcul des facteurs d'accélération et d'efficacité) la version parallèle de notre algorithme

sur deux types de conteneurs (listes et tableaux de taille fixe) et deux types de prédicats unaires. Le temps de calcul du premier (`estPair`) est fixe tandis que celui du second (`pgcd21Vaut3`) varie très fortement ;

`CMakeLists.txt` : script permettant de générer le makefile de l'application 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`.

2.1 Question

Décompressez l'archive `tp1.tar.gz`. En fonction des instructions du fichier `Lisezmoi.txt`, créez le makefile de l'application, configurez le fichier `CMakeCache.txt` puis lancez la compilation.

2.2 Question

Comme elle représente une spécialisation totale, la définition de la sous-classe `Impl` dédiée au cas particulier des `RandomAccessIterator` est placée à l'extérieur de la classe `CountIf` (ainsi que l'exige la norme C++). Complétez la définition de sa méthode `apply` via une boucle `for` parallèle et la clause la plus adéquate pour la répartition des itérations.

2.3 Question

Que concluez-vous des performances de cette première version parallèle de l'algorithme `count_if` ?

2.4 Question

Concernant les `InputIterator`, une première stratégie de parallélisation consiste, dans un premier temps, à constituer un tableau (de taille fixe) dont les éléments contiennent des pointeurs vers les éléments de l'intervalle `[first, last[`. Dans un second temps, les éléments de ce tableau sont traités via une boucle parallèle `for` et la clause de répartition des itérations la plus adéquate.

Le problème posé par cette stratégie est que le nombre d'éléments de l'intervalle `[first, last[` ne peut être calculé à priori à moins de le parcourir (ce qui s'avérerait exorbitant en termes d'overhead). Par conséquent, il est impossible d'utiliser un tableau de taille fixe. Envisager un tableau de taille dynamique (classe `vector`) serait suicidaire en raison des opérations de ré-allocation-recopie enclenchées lorsque la taille du tableau vient à excéder sa capacité.

La solution à ce problème consiste à tronçonner l'intervalle `[first, last[` en segments d'une taille arbitraire fixe (éventuellement calculée en fonction du nombre de threads disponibles). Notre stratégie de parallélisation revient alors à constituer un segment de pointeurs, à traiter ce segment via une boucle parallèle `for` et à recommencer ainsi jusqu'à avoir traité l'intégralité des éléments de l'intervalle `[first, last[`.

Complétez la définition de la méthode générique `strategyA` qui implémente cette stratégie de parallélisation. Dans un premier temps, la boucle de tronçonnage sera séquentielle tandis que seul le traitement du tronçon fera l'objet d'une région parallèle (`pragma omp parallel for schedule(...)`).

2.5 Question

Faites en sorte que la méthode `apply` invoque la méthode `strategyA` et observez ses performances en faisant éventuellement varier la taille des tronçons. Qu'en concluez-vous ?

2.6 Question

L'écriture de la méthode générique `strategyA` demandée précédemment vous place à la merci de votre compilateur (n'oubliez pas qu'OpenMP n'est pas une bibliothèque mais un standard). En effet, il peut traduire ce code de façon intelligente ou particulièrement stupide :

- s'il est intelligent, il ré-écrit l'intégralité de la boucle de tronçonnage pour l'incorporer dans une région parallèle. Par conséquent, les threads créés en entrée de cette région seront disponibles tout le temps d'exécution de la boucle de tronçonnage ;
- s'il est stupide, il traduira votre code mot pour mot c'est à dire que des threads risquent d'être créés en entrée de la boucle parallèle `for` puis détruits à l'issue et ce à chaque itération de la boucle de tronçonnage (c'est pour cette raison que la norme OpenMP recommande aux développeurs de toujours maximiser leurs régions parallèles).

Ré-écrivez la définition de la méthode générique `strategyA` afin que la boucle de tronçonnage soit placée à l'intérieur d'une région parallèle (une attention très particulière devra être apportée à la synchronisation de vos différents threads ...).

2.7 Question

Une seconde stratégie de parallélisation consiste à exploiter la notion de tâche introduite par la norme 3.0 d'OpenMP pour tenter de reproduire la clause de répartition des itérations `schedule(dynamic, size)` des boucles parallèles `for`. Une taille `size = 1` n'est bien sûr pas envisageable puisque le coup de création et de synchronisation d'un nombre de tâches trop important serait alors largement supérieur aux gains produit par la parallélisation. Nous allons donc fixer la taille du paramètre `size` à une valeur arbitraire (pouvant éventuellement dépendre du nombre de threads disponibles) et ainsi tronçonner l'intervalle `[first, last[`. Chaque tâche ainsi créée invoquera l'algorithme `count_if` de la bibliothèque standard sur son sous-intervalle. Le résultat de cet appel sera ensuite cumulé dans un compteur global commun à toutes les tâches (attention : la clause `reduction` n'est malheureusement applicable qu'aux threads mais pas aux tâches).

Complétez la définition de la méthode générique `strategyB` qui implémente cette stratégie de parallélisation. Le tronçonnage de l'intervalle `[first, last[` et la création des tâches correspondantes seront l'œuvre d'un seul et unique thread.

2.8 Question

Il est peut-être possible d'optimiser la méthode `strategyB` de la question précédente comme suit :

1. les tâches étant créées à l'intérieur d'un bloc `single`, tous les autres threads sont placés en attente sur la barrière de synchronisation implicite placée à la sortie de ce bloc. Par conséquent, ces derniers sont obligés d'attendre que toutes les tâches soient créées pour commencer à les traiter. Une clause `nowait` placée derrière le mot-clé `single` permet de lever cette barrière ;
2. lorsqu'une tâche commence son exécution avec un thread et qu'elle est mise en sommeil pour une raison particulière, elle ne peut reprendre son exécution qu'avec le même thread. Par conséquent, tant que ce thread n'est pas disponible, elle reste en attente même si elle est prête à reprendre son exécution. Toute implémentation prévoit un nombre maximum de tâches en attente dans une file de tâches à exécuter. Lorsque ce plafond est atteint, le thread chargé de créer les tâches suspend le processus et commence lui-même à traiter les tâches en attente. Quand un certain nombre de tâches ont été sorties de la file, il reprend le processus de création. Par conséquent, si ce thread particulier a commencé à traiter des tâches puis repris le processus de création, toutes les tâches qu'il a commencées mais qu'il n'a pas terminées restent en attente. La clause `untied` placée derrière le mot clé `task` permet de briser le lien entre une tâche et son thread : une tâche qui a commencé son exécution avec un thread puis qui a été mise en sommeil peut être reprise par un autre thread.

Faites en sorte que la méthode `apply` invoque la méthode `strategyB` et observez ses performances en faisant éventuellement varier la taille des tronçons. Qu'en concluez-vous ?

2.9 Question

Une troisième stratégie de parallélisation consiste à reproduire la clause de répartition des itérations `schedule(static, 1)` des boucles parallèles `for`, c'est à dire une répartition cyclique des itérations de l'intervalle `[first, last[` 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 alors chaque

thread doit avancer de \mathcal{P} positions dans l'intervalle `[first, last[`, traiter l'élément correspondant et recommencer ainsi jusqu'à dépasser la position `last`.

Complétez la définition de la méthode générique `strategyC` qui implémente cette stratégie de parallélisation.

2.10 Question

Faite en sorte que la méthode `apply` invoque la méthode `strategyC` et observez ses performances. Qu'en concluez-vous ?