Tâches OpenMP

1 Hello tasks !

Dans ce programme `no-hello.c` on crée autant de threads qu'il y a de cœurs logiques sur la machine grâce à la directive `pragma omp parallel`.

fichiers/no–hello.c

```
int main()
{
  #pragma omp parallel
  {
    printf("Bonjour de la part de %d\n",omp_get_thread_num() );
    printf("Au revoir de la part de %d\n",omp_get_thread_num() );
  }
  return 0;
}
```

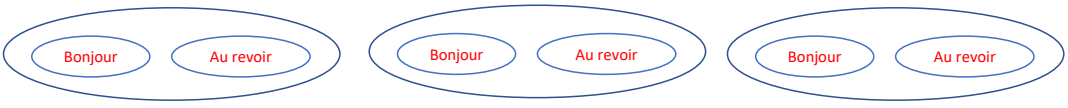
D'après la norme, et par construction, chaque thread son code au sein d'une *tâche implicite* qui lui est automatiquement associée. Ici chaque thread/tâche implicite réalise deux `printf()`. Pour 3 threads ceci produit le graphe de tâches suivant :

Une tâche explicite est un bloc d'instructions que l'on souhaite exécuter en parallèle avec d'autres traitements. Pour créer des tâches explicites on utilise le pragma `omp task` comme dans le programme suivant :

fichiers/hello.c

```
int main()
{
  #pragma omp parallel
  {
    #pragma omp task
    printf("Bonjour de la part de %d\n",omp_get_thread_num() );

    #pragma omp task
    printf("Au revoir de la part de %d\n",omp_get_thread_num() );
  }
  return 0;
}
```



Ici chaque thread exécute sa tâche implicite qui crée explicitement 2 tâches filles. Chaque tâche explicite affiche le numéro du thread qui l'exécute. Compiler le programme et exécuter le plusieurs fois pour voir ses comportements possibles.

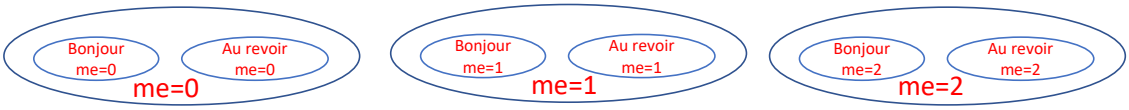
Ensuite, pour mieux visualiser le travail de chaque thread, on utilise une variable `me` contenant le numéro du thread initiateur de la tâche. On passe ensuite cette variable initialisée aux tâches filles grâce à la clause `firstprivate(me)`. Cette clause indique, à l'image d'un paramètre de fonction passé par valeur, qu'il faut créer une copie de la variable pour la tâche et capturer sa valeur, et ce dès la création de la tâche.

fichiers/analyse–hello.c

```
int main()
{
  #pragma omp parallel
  {
    int me = omp_get_thread_num();

    #pragma omp task firstprivate (me)
    printf("Bonjour de %d exécuté par %d\n", me, omp_get_thread_num());

    #pragma omp task firstprivate (me)
    printf("Au revoir de %d exécuté par %d\n", me, omp_get_thread_num());
  }
  return 0;
}
```



Exécuter et analyser le comportement du programme ci-dessus. On pourra utiliser plus de threads que de cœurs pour renforcer le caractère non déterministe de ce programme : chaque thread crée deux tâches qui sont chacune exécutée par n'importe quel thread de l'équipe et ce dans n'importe quel ordre. Seule une tâche implicite est associée à un thread, toutes les tâches explicites sont associées à l'équipe de threads.

La norme OpenMP précise qu'une tâche explicite peut être exécutée immédiatement après sa création ou être différée jusqu'à son exécution par n'importe quel thread de l'équipe du thread créateur. La norme précise aussi qu'un thread peut laisser de coté une tâche en cours d'exécution pour en débuter une autre et que, par défaut [1], toute tâche commencée par un thread doit être terminée par celui-ci.

2 L'idiome parallel - single

Pour exécuter des tâches en parallèle, il est nécessaire de créer une équipe de threads. Afin de maitriser la génération du parallélisme, les programmeurs OpenMP ont l'habitude de faire en sorte qu'un seul thread lance la première tâche. Le nombre de tâches lancées est alors indépendant du nombre de threads ce qui est bien commode. On utilise pour cela la directive `single` (un seul thread exécute le bloc) ou bien la directive `master` (seul le maître exécute le code).

Exécuter le programme suivant puis modifier le afin que chaque affichage soit pris en charge par une tâche dédiée à cela.

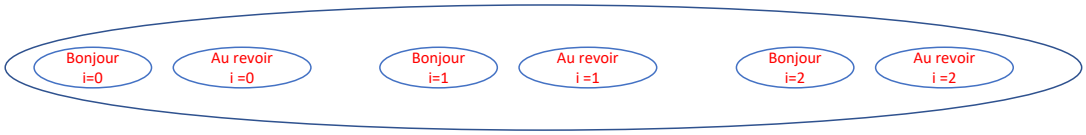
fichiers/single–hello.c

```
int main()
{
#pragma omp parallel
  {
#pragma omp single
    {
      for (int i = 0 ; bonjour[i] != NULL; i++)
        printf("%s (%d)\n",bonjour[i], omp_get_thread_num());

      for (int i = 0 ; aurevoir[i] != NULL; i++)
```

1. Seules les tâches créées à l'aide de la clause `untied` peuvent être reprises par tout thread équipier

```
        printf("%s (%d)\n",aurevoir[i], omp_get_thread_num());
    }
  }
  return 0;
}
```



3 Synchronisation de tâches

À la base d'OpenMP, il y a l'espoir de produire un programme parallèle à partir d'un programme séquentiel en utilisant des directives pour ajouter du parallélisme ou, à l'opposé, le contraindre. Le concept de tâche permet d'exposer facilement du parallélisme et nous avons jusqu'ici produit des ensembles de tâches dont l'exécution est finalement désordonnées, non déterministe. Nous allons voir ici comment mettre un peu d'ordre dans ces ensembles et y introduire une dose de déterminisme.

3.1 Directive taskwait

On désire maintenant attendre la terminaison de toutes les tâches « bonjour » avant de créer les tâches « au revoir ». Pour cela on va insérer la ligne `pragma omp taskwait` entre les deux `for`. Cette directive bloque l'exécution de la tâche en cours jusqu'à la terminaison de toutes ses tâches filles. Tester cette solution.

Ensuite on souhaite paralléliser le programme suivant :

fichiers/simulation–calcul.c

```
int f(int i){
  sleep(1);
  return 2*i;
}

int g(int i){
  sleep(1);
  return 2 * i + 1;
}

int main()
{
  int x = f(2);
  int y = g(3);
  printf("résultat %d\n", x+y);
  return 0;
}
```

Pour cela on va utiliser une tâche pour calculer `f(2)` et modifier `x`[2]. On précisera grâce à la clause `shared(x)` que la variable `x` de la fonction `main()` doit être modifiée par effet de bord par la tâche. La clause `shared(x)` correspond à un passage par adresse d'un paramètre d'une fonction.

Vérifier que le programme est bien parallélisé (durée de l'ordre de la seconde) et que le résultat affiché est correct.

2. On peut éventuellement ajouter une seconde tâche pour calculer `g(3)` et modifier `y`.

3.1.1 taskwait vs taskgroup

Le programme `task-wait.c` montre que le comportement de la directive `taskwait` est uniquemet déterminé par l'état de la tâche qui l'exécute. Ici seule les tâches n'ayant pas généré de tâche se terminent rapidement.

version épurée de `task-wait.c`

```
void creer_tache(char *nom, int maman)
{
#pragma omp task firstprivate(nom,maman)
      {
        sleep(1);
        printf("%s [fille de %d]\n",nom,maman);
      }
}

int main()
{
#pragma omp parallel num_threads(4)
  {
    int me ;
    #pragma omp atomic capture
    me = id++;
#pragma omp single nowait
    {
      printf("tache %d va créer A et B \n", me);
      creer_tache("A",me);
      creer_tache("B",me);
    }

#pragma omp taskwait
    printf("tache %d a passé taskwait \n", me);
  }
 }
```

La directive `taskwait` ne concerne que les tâches filles. Pour s'en convaincre, comparer les comportements des programmes `task-group.c` et `task-wait.c`. Modifier ensuite le programme `task-group.c` en utilisant la directive `taskgroup` pour attendre la terminaison de toute la descendance d'une tâche. Cette directive porte sur un bloc d'instructions (à la manière des directives `critical`, `master`,...).

3.1.2 Tâches et durée de vie des variables locales

La directive `taskwait` permet aussi de maîtriser la durée de vie des variables locales, comme l'illustre le programme suivant : Exécuter le programme suivant :

extrait de fichiers/duree–vie–locales.c

```
void generer()
{
  char chaine[]="jusqu'ici tout va bien";

  for(int i = 0; i < 10 ; i++)
#pragma omp task shared(chaine) firstprivate(i)
    printf("tache %d par thread %d >>>> %s <<<< \n",
           i,omp_get_thread_num(), chaine);

#pragma omp taskwait
}
```

1. Exécuter le progamme `duree-vie-locales` et vérifier qu'il se déroule normalement.
2. Supprimer la directive puis observer le comportement du programme. Expliquer.
3. Remplacer la clause `shared(chaine)` par une directive `firstprivate(chaine)` puis observer le comportement du programme. Expliquer.

3.2 Graphe de tâches inter-dépendantes

On désire affiner le comportement du programme : la seule régle est que pour toute langue le bonjour doit s'afficher avant l'au revoir correspondant.

3.2.1 Imbrication de tâches

Voici une solution où l'on utilise une tâche intermédiaire pour regrouper les couples de tâches :

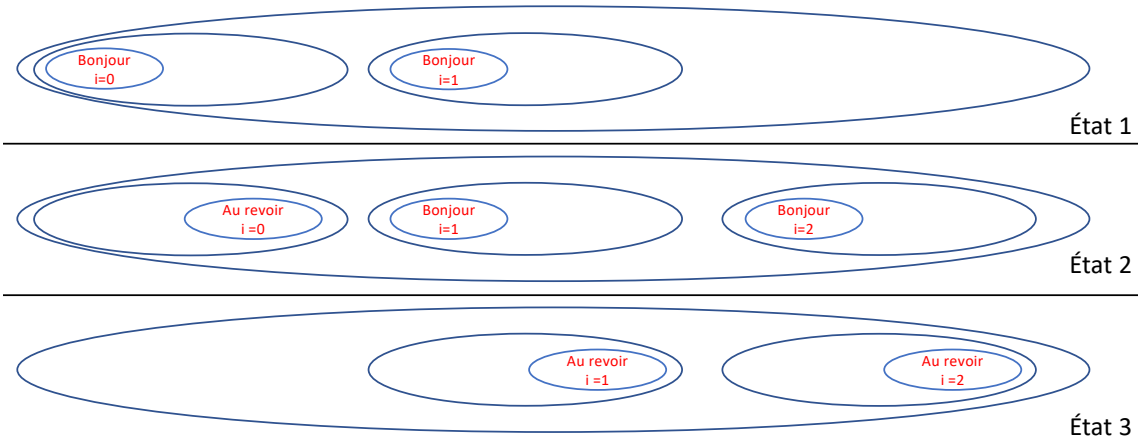
fichiers/tw–hello.c

```
int main()
{
   #pragma omp parallel
  {
   #pragma omp single
    {
      for (int i = 0 ; bonjour[i] != NULL; i++)
      #pragma omp task firstprivate(i)
      {
          #pragma omp task firstprivate(i)
          printf("%s (%d)\n",bonjour[i], omp_get_thread_num());

          #pragma omp taskwait

          #pragma omp task firstprivate(i)
          printf("%s (%d)\n",aurevoir[i], omp_get_thread_num());
        }
      }
  }
  return 0;
}
```

Ci-dessous figure un exemple de trois états successifs des tâches en attente d'exécution, on observe que pour un `i` donné la tâche au revoir apparaît toujours après la terminaison de la tâche bonjour correspondante.



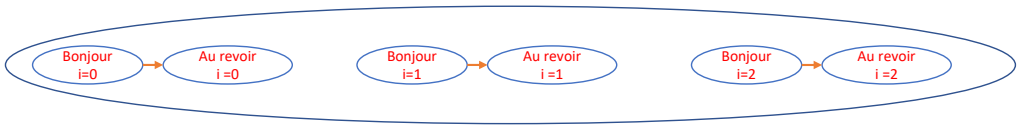
3.2.2 Dépendance de données

Une solution plus élégante est d'utiliser la capacité d'OpenMP à calculer automatiquement les dépendances entre les tâches à partir d'indications fournies par le programmeur. On transforme un programme séquentiel (fonctionnel) en une séquence de création de tâches et pour chaque tâche on précise quelles sont ses données accédées en lecture seule, écriture seule et lecture/ écriture.

Comme les tâches sont insérées séquentiellement le support d'exécution et capable d'ajouter automatiquement des dépendances entre une tâche qui produit un objet et celles qui le consomme.

Rependre le fichier `single-hello.c` pour mettre en place ces dépendances.

Ici nous allons ruser en ajoutant la clause `depend(out: bonjour[i])` aux pragmas des tâches bonjour et `depend(in: bonjour[i])` à ceux des tâches au revoir. On indique au support d'exécution que chaque tâche bonjour produit la donnée `bonjour[i]` et que celle-ci est nécessaire à l'exécution de la tâche au revoir correspondante. Cela produit un graphe de dépendances à l'image du graphe ci-dessous.



Notons qu'il est possible d'utiliser le mode `inout` pour qualifier les objets en lecture écriture et que les dépendances ne sont calculées qu'entre tâches sœurs.

3.2.3 Génération d'un graphe de tâches

Dans le programme `depend.c` les tâches peuvent être exécutées dans un ordre aléatoire. Lorsqu'on exécute ce code, les entiers de 0 à $T^2 - 1$ apparaissent exactement une fois dans le tableau dans un ordre quelconque.

version épurée de `depend.c`

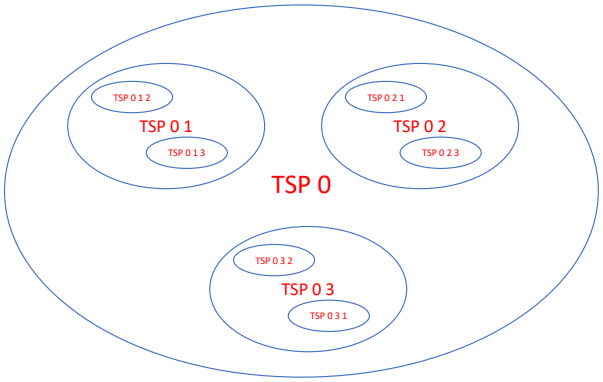
```
int A[T][T];
int k = 0;
#pragma omp parallel
#pragma omp single
  for (i=0; i < T; i++ )
    for (j=0; j < T; j++ )
#pragma omp task firstprivate(i,j)
#pragma omp atomic capture
    A[i][j] = k++;
```

Il s'agit de faire en sorte qu'une tâche traitant le couple d'indices `(i,j)` attendent que les tâches traitant les couples `(i-1,j)` lorsque `i > 0` et `(i,j-1)` lorsque `j > 0` soient terminées pour pouvoir être exécutée. Autrement dit à la fin on devrait obsterver la propriété suivante : `A[i][j] >= A[x][y]` lorsque `i >= x` et `j >= y`.

Utiliser le mécanisme des dépendances pour obtenir le comportement attendu. Étrangement, pour simplifier le code on aura intérêt à modifier ainsi `int A[T][T+1]` la déclaration de `A`. Pourquoi ?

4 Parallélisation du TSP à l'aide de tâches OpenMP

Voici le graphe de tâches imbriquées attendu pour 4 villes :



Nous utilisons le code développé lors du précédent TP en nous appuyant sur le code de la fonction `tsp_ompfor()` :

code `tsp_ompfor()`

```
1  void tsp_ompfor(int etape, int lg, chemin_t chemin, int mask)
2  {
3      if (etape == nbVilles)
4          verifier_minimum(lg, chemin);
5      else if (etape > grain)
6          tsp_seq(etape, lg, chemin, mask);
7      else
8      {
9          int ici = chemin[etape - 1];
10 #pragma omp parallel for schedule(dynamic)
11         for (int i = 1; i < nbVilles; i++)
12         {
13             if (!present(i, mask))
14             {
15                 chemin_t mon_chemin;
16                 memcpy(mon_chemin, chemin, etape * sizeof (*chemin));
17                 mon_chemin[etape] = i;
18                 int dist = distance[ici][i];
19                 tsp_ompfor(etape + 1, lg + dist, mon_chemin, mask | (1 << i));
20             }
21         }
22     }
23 }
```

Concrétement, il s'agit de :

1. dupliquer le code de la fonction et la nommer `tsp_task()` (ne pas oublier de changer l'appel récursif en ligne 19).
2. déplacer la création des threads (ligne 19) au niveau du `main()` en faisant en sorte qu'un seul thread lance la recherche ;
3. insérer une directive après la ligne 13 pour créer une tâche pour chaque appel récursif à la procédure `tsp_task()`.
4. insérer une clause `taskwait` à la fin de la fonction pour éviter la perte du tableau `chemin`.

4.1 Mesure de perfomances sans l'optimisation algorithmique

Tester et mesurer les performances de votre programme avec le cas « 13 villes et graine 1234 » avec des grains de plus en plus grands. Utiliser le script pour comparer les performances à la version `collapse(3)`.

4.2 Mesure de perfomances avec l'optimisation algorithmique

Tester et mesurer les performances de votre programme avec le cas « 15 villes et graine 1234 » avec des grains de plus en plus grands. Utiliser le script pour comparer les performances à la version `collapse(3)` et, pourquoi pas, une version `collapse(4)` que vous ne manquerez pas d'écrire.

4.3 Pour aller plus loin

Il est possible de supprimer la directive `taskwait` qui prolonge la durée de vie de la variable `chemin` soit en allouant dynamiquement une copie du tableau `chemin` avant la création de la tâche, soit en allouant dans la pile une copie du tableau `chemin` dans une variable locale `monchemin` et en en utilisant la clause `firstprivate(monchemin)`.