Introduction à OpenMP

Jérôme Lelong

Ensimag

Année 2013-2014

Généralités

- Ensemble de directives de compilations et de fonctions utilisées pour créer des programme parallèles sur les systèmes à mémoire partagée.
- La définition formelle de OpenMP est donnée dans des spécifications pour le Fortran et le C/C++
- Basé sur le un modèle de type **fork join**.
  - La tâche principale s'exécute de manière séquentielle jusqu'à ce quelle rencontre une directive parallèle.
  - Le programme principal crée des branches (forks) complémentaires qui s'exécutent en parallèle.
  - Les tâches s'exécutent dans une **région parallèle**.
  - A la fin de la région parallèle, les tâches attendent jusqu'à ce que l'ensemble des tâches d'un même groupe se terminent.
  - La tâche principale continue son exécution jusqu'à la prochaine région parallèle.
- L'objectif est de faire en sorte que les programmeurs puissent paralléliser leur code facilement.
  Le programme parallélisé avec OpenMP doit être performant, mais pas au détriment de la complexité de sa programmation ou de sa maintenance.

OpenMP en bref

- API de parallélisation pour la mémoire partagée (multi-thread)
- OpenMP est basé sur un modèle "fork join"
  - La thread principale s'exécute séquentiellement jusqu'à la première directive parallèle
  - Fork : La thread principale crée un ensemble de threads qui vont s'exécuter parallèlement
  - Join : Lorsque tous les threads ont terminé leur tâche, synchronisation puis la thread principale reprend séquentiellement.
- Avantages :
  - Espace d'adressage globale, partage des données facile
  - code peu différent du code séquentiel.
  - pas de surcoût de communication
- Inconvénients :
  - Mémoire limitée : mauvais passage à l'échelle
  - Conflits d'accès aux variables partagées.

Deux types d'approche

- Parallélisme de boucle : la boucle placée après cette directive est automatiquement parallélisée, chaque thread en réalise une partie. Le reste du code s'exécute de manière séquentielle et inchangée.
- Parallélisme de région : une région du code s'exécute simultanément sur plusieurs threads.

Compilation

En C/C++,

- inclure le fichier `omp.h`
- passer le flag `-fopenmp` à gcc/g++ pour la compilation **et** l'édition de liens. Dans un Makefile standard

```
CFLAGS=-fopenmp
CXXFLAGS=-fopenmp
LDFLAGS=-fopenmp
```

- la macro `_OPENMP` est automatiquement définie et peut servir à une compilation conditionnelle

```
#ifdef _OPENMP
partie parallèle
#endif /* _OPENMP */
```

- Sous Mac OS X, pas de support OpenMP par défaut.
- Pour l'instant, le compilateur LLVM ne supporte pas les directives OpenMP.

La directive parallel

- La directive parallel crée une région parallèle.

```
#pragma omp parallel
{
  // région parallèle.
  printf("Hello World\n");
}
```

- Chaque thread va exécuter l'intégralité du code se trouvant dans la région parallèle.
- Par défaut, le nombre de threads créées est égales au nombre de coeurs.

La directive parallel for

- La directive parallel for indique que la boucle qui suit doit être exécutée en parallèle

```
#pragma omp parallel for
for ( i=0 ; i<n ; i++ )  a[i] = b[i] + c[i];
```

```
int i, n = 4;
#pragma omp parallel for
for ( i=0 ; i<n ; i++ )
  printf("thread : %d, loop : %d\n",
         omp_get_thread_num(), i);
```

- S'il y a 2 threads, chacune exécute $n/2$ itérations.

parallel-for-1.c

Les directives parallel et for

- Il est possible de séparer la création des threads (parallel) et la parallélisation de la boucle (for).

```
#pragma omp parallel
{
  #pragma omp for
  for ( i=0 ; i<n ; i++ )
    printf("thread : %d, loop : %d\n",
           omp_get_thread_num(), i);
}
```

- Ne pas mettre deux fois la directive parallel, sinon ...

parallel-twice.c

La synchronisation

- Par défaut, synchronisation à la fin de la boucle for.
- Utiliser la clause nowait pour ne pas synchroniser

```
#pragma omp parallel
{
#pragma omp for nowait
    for ( i=0 ; i<n ; i++ )
      printf("First For -- thread : %d, loop : %d\n",
             omp_get_thread_num(), i);
#pragma omp for
    for ( i=0 ; i<n ; i++ )
      printf("Second For -- thread : %d, loop : %d\n",
             omp_get_thread_num(), i);
}
```

parallel-nowait.c

Variables privées ou partagées

Les variables peuvent être

- globales/partagées : elles sont partagées par toutes les threads. On les déclare avec shared(var1, var2)
- locales : chaque thread possède sa propre version de la variable, elles sont “allouées” dans une espace mémoire local. On les déclare avec private(var1, var2)

Par défaut, toute variable est partagée.

- On peut changer la règle par défaut avec default(private)
- On peut supprimer toute règle par défaut avec default(none). Dans ce cas, toutes les variables doivent apparaître explicitement dans shared() ou private().

Variables privées ou partagées

- Echange de tableau

```
#pragma omp parallel for private(tmp)
for ( i=0 ; i<n ; i++ )
{
tmp = b[i]; b[i] = a[i]; a[i] = tmp;
}
```

- On peut spécifier l’état de chaque variable

```
#pragma omp parallel for default(none) \
private(tmp) shared(a,b,n)
for ( i=0 ; i<n ; i++ )
{
tmp = b[i]; b[i] = a[i]; a[i] = tmp;
}
```

- Les variables de boucle sont gérées automatiquement par le compilateur.

Variables privées

- firstprivate : initialise une variable privée avec sa valeur avant le bloc parallèle.
- lastprivate : la variable privée conserve sa valeur après le bloc parallèle. La valeur dépend de la dernière thread exécutée
- Une variable allouée dynamiquement ne peut être privées mais les différentes threads peuvent faire appel à l’allocation dynamique pour créer des pointeurs en propre.

La clause Réduction

```
#pragma omp parallel for shared(n, a)
for ( i=0 ; i<n ; i++ )
  sum += a[i] ;
```

Quel statut pour `sum`

- private : les calculs des différentes threads sont perdus une fois les threads terminées. Même avec lastprivate, on ne récupère que la valeur calculée par la dernière thread.
- shared : toutes les threads écrivent à la même adresse mémoire. **Grave erreur**.
- reduction : il faut déclarer la variable `sum` comme étant une réduction : `reduction(op : var)`, ici cela donne

```
#pragma omp parallel for shared(n, a) \
reduction(+ : sum)
for ( i=0 ; i<n ; i++ )
  sum += a[i] ;
```

Les différents types de variables

- 3 types de variables : privées, partagées, les réductions.
- Les variables déclarées à l'intérieur d'un bloc parallèle sont obligatoirement privées.

```
#pragma omp parallel
{
  double lsum = 0.;
  #pragma omp for
  for ( i=0 ; i<N ; i++ )
    lsum += a[i];
  #pragma omp atomic
  sum += lsum;
}
```

lsum est automatiquement privée.

- De même, lors de l'appel à une fonction externe, les variables locales déclarées dans cette fonction sont automatiquemet privées.

Threadprivate

```
  int  b, tid;
  static double x;  static int a;
#pragma omp threadprivate(a,x)
  printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
  {
    tid = omp_get_thread_num();
    a = tid;
    b = tid;
    x = 10 + tid;
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }
  printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }
```


Threadprivate

- S'applique à des variables globales à un fichier ou statique (y compris dans une fonction). Par défaut ces variables sont partagées.
- La directive threadprivate permet de déclarer que les variables listées seront privées à chaque thread mais persistantes d'une région parallèle à une autre. Elles gardent leur valeur entre 2 régions parallèles.
- La directive threadprivate doit apparaître après la déclaration des variables mais avant qu'elles ne soient référencées.
- Si de plus cette variable est précisée dans copyin, alors elle sera initialisée dans chaque thread par sa valeur dans la thread principale.
- La répartition entre les threads doit être statique et le même nombre de threads doivent s'exécuter dans chaque région parallèle.

Allocation dynamique

- Des opérations d'allocation/libération dynamique peuvent être effectuées au sein d'une région parallèle.
- Si l'opération porte sur une variable privée, alors l'allocation réservera une zone mémoire différente dans chaque thread. Chaque thread doit libérer la mémoire allouée avant de se terminer

```
#pragma omp parallel private (rng)
{
  rng = pnl_rng_create (PNL_RNG_MERSENNE);
  ...
  pnl_rng_free (&rng);
}
```

Allocation dynamique

- Des opérations d'allocation/libération dynamique peuvent être effectuées au sein d'une région parallèle.
- Si l'opération porte sur une variable privée, alors l'allocation réservera une zone mémoire différente dans chaque thread. Chaque thread doit libérer la mémoire allouée avant de se terminer

```
#pragma omp parallel
{
  PnlRng *rng;
  rng = pnl_rng_create (PNL_RNG_MERSENNE);
  ...
  pnl_rng_free (&rng);
}
```

- Si l'opération porte sur une variable partagée, alors il faut s'assurer qu'une seule thread se charge de l'allocation et idem pour la libération. Utiliser les directives master, single.

Informations sur les threads

- `int omp_get_thread_num()` renvoie le numéro du thread entre 0 et $p-1$ s'il y a p threads
- `omp_set_num_threads(int)` fixe le nombre de threads. Cette fonction doit être appelée dans une zone **séquentielle** avant la région parallèle
- `int omp_get_num_threads()` retourne le nombre de threads utilisées dans une région parallèle.
- `int omp_int_parallel()` renvoie un entier non nul si l'appel se fait dans une zone parallèle.
- `double omp_get_wtime()` mesure le temps (de la pendule) écoulée depuis une date fixe du passé. Permet par différence de mesurer des speed-ups.
- `omp_set_dynamic(0 ou 1)` active ou non les threads dynamiques

La directive Parallel

- On peut paralléliser autre chose que de simples boucles avec la directive

```
#pragma omp parallel
```

- Mais dans ce cas il faut gérer à la main la synchronisation et la quantité de travail que chacun fait.

```
#pragma omp parallel default(none) shared(n) \
private (i, id, nthreads) reduction(+:sum)
{
  nthreads = omp_get_num_threads() ;
  id = omp_get_thread_num ();
  for (i=id*(n/nthreads) ; i<(id+1)*n/nthreads ; i++)
  {
    sum += i;
  }
}
```

▸ *parallel-sansfor.c*

La directive Barrier

La directive barrier permet de synchroniser tous les threads en un point donné d'une région parallèle

```
#pragma omp parallel
{
  faire_quelque_chose
  for ( i=0 ; i<n ; i++ )
  {
    agir sur a[i]
  }
  #pragma omp barrier
  travailler sur a
}
```


Les directives Master et Single

- La directive master permet de spécifier qu'une sous partie d'un bloc parallèle ne doit être exécutée que par le maître.

```
#pragma omp parallel shared(a, acopy)
  {
    for ( i=0 ; i<n ; i++ )
    {
      agir sur a[i]
    }
#pragma omp barrier
#pragma omp master
    {
      memcpy (acopy, a, n*sizeof(double);
    }
    travailler sur a
  }
```

- On peut remplacer master par single ; dans ce cas, le code n'est exécuté qu'une seule fois mais mas forcément par le maître.

La directive Ordered

Il peut arriver que l'on souhaite exécuter une partie d'une boucle parallèle de manière séquentielle (ordonnée)

```
#pragma omp parallel for private(tmp) ordered
for ( i=0 ; i<n ; i++ )
{
  tmp = une_fonction (i)
  #pragma omp ordered
  {
    printf ("%d, %f\n", i, tmp);
  }
}
```

parallel-ordered.c

La directive Critical

La directive critical permet de s'assurer qu'une seule thread à la fois exécutera le code. Similaire à ordered mais sans garantir l'ordre d'exécution.

```
#pragma omp parallel default(none) shared(n,sum) \
private (i, id, nthreads, lsum)
{
  nthreads = omp_get_num_threads() ;
  id = omp_get_thread_num ();
  lsum = 0;
  for (i=id*(n/nthreads) ; i<(id+1)*n/nthreads ; i++)
    {
      lsum += i;
    }
  #pragma omp critical
  sum += lsum;
}
```

critical.c

La directive Atomic

- Le directive assure qu'une variable partagée est lue et modifiée en mémoire par une seule tâche à la fois.
- Son effet est local à l'instruction qui suit immédiatement la directive.
- L'instruction doit être de la forme `x op = exp` où `op` est une opération élémentaire +,-,×,/ et `exp` une expression indépendante de `x`.
- Les performances de la directive atomic sont bien meilleures que celles de la directive critical. Ne pas se servir de critical pour une unique instruction.

La directive Atomic (exemple)

```
#pragma omp parallel default(none) shared(n,sum) \
private (i, id, nthreads, lsum)
{
  nthreads = omp_get_num_threads() ;
  id = omp_get_thread_num ();
  lsum = 0;
  for ( i=id*(n/nthreads) ; i<(id+1)*n/nthreads ; i++ )
    {
      lsum += i;
    }
  #pragma omp atomic
  sum += lsum;
}
```

atomic.c

La directive Section

- Comment exécuter des blocs de code différents de manière parallèle ? Utiliser les sections.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    tache_1
    #pragma omp section
    tache_2
  }
}
```

- `tache_1` et `tache_2` sont exécutées en parallèle par 2 threads différentes.

Ordonnancement

Lors de l'utilisation de #pragma omp for, la clause schedule(type[,chunk]) définit comment les itérations de la boucle sont réparties entre les threads. type peut valoir :

- static : les itérations sont divisées en blocs de taille *chunk* puis assignées de manière statique. Si *chunk* n'est pas précisé, les itérations sont découpées de manière contiguë entre les threads.
- dynamic les itérations sont divisées en blocs de taille *chunk*. Dès qu'une thread a fini, on lui assigne un autre bloc. Par défaut, chunk=1.
- guided : si chunk=k, la taille de chaque bloc est proportionnelle au nombre d'itérations restantes divisées par le nombre de threads. cette taille décroît jusqu'à k (sauf pour la dernière thread).
- runtime définit par la variable OMP_SCHEDULE. Pas d'argument *chunk*.
- auto : le scheduling est laissé au compilo et/ou à l'OS.

Les variables d'environement

- OMP_SCHEDULE
- OMP_NUM_THREADS (entier) : nombre de threads à utiliser dans les régions parallèles si aucune autre valeur n'est spécifiée.
- OMP_DYNAMIC (TRUE ou FALSE) : active ou non l'ajustement automatique du nombre de threads disponible pour l'exécution de régions parallèles
- OMP_NESTED (TRUE ou FALSE) : autorise ou non le parallélisme imbriqué. Dépend des implémentations.
- OMP_MAX_ACTIVE_LEVEL (entier) : si OMP_NESTED=TRUE, définit le nombre maximal de niveaux d'imbrication.
- OMP_THREAD_LIMIT (entier) : nombre maximal de threads utilisables.

Sortie d'une boucle

Recherche de la présence d'une sous-chaîne dans une chaîne de caractères

```
for(int p = 0; p < size; p++)
  if(str[p:p+n] == substr)
    {
      return str + p;
    }
  return NULL;
```

On ne peut pas sortir d'une boucle parallel for avec un break ou un return.
Quelles solutions pour paralléliser ?

- Utiliser pthreads et la fonction `pthread_cancel()` au lieu d'OpenMP
- Utiliser un flag pour détecter qu'une thread a trouvé et paralléliser "à la main".

Sortie d'une boucle

```
const char* result = NULL; bool done = false;
#pragma omp parallel {
  int this_thread = omp_get_thread_num();
  int num_threads = omp_get_num_threads();
  int begin = this_thread * size / num_threads;
  int end   = (this_thread+1) * size / num_threads;
  for(int p = begin; p < end; ++p) {
    if(str[p:p+n] == substr) { result = str+p; break; }
  }
}
```

search-naif.c

- Toutes les threads s'exécutent jusqu'à trouver une occurrence de la sous-chaine. Pas d'arrêt quand une thread a trouvé
- Pas de gain par rapport à la version séquentielle si l'on teste juste la présence de la sous-chaîne ou non

Sortie d'une boucle

```
const char* result = NULL; bool done = false;
 #pragma omp parallel {
   int this_thread = omp_get_thread_num();
   int num_threads = omp_get_num_threads();
   int begin = this_thread * size / num_threads;
   int end   = (this_thread+1) * size / num_threads;
   for(int p = begin; p < end; ++p) {
     if(done) break;
     if(str[p:p+n] == substr) {
       done = true; result = str+p; break;
     }
   }
 }
```

search-1.c

La synchronisation des variables partagées n'est pas immédiate. Il faut la forcer avant lecture et écriture.

Sortie d'une boucle

```
const char* result = NULL; bool done = false;
 #pragma omp parallel {
   int this_thread = omp_get_thread_num();
   int num_threads = omp_get_num_threads();
   int begin = this_thread * size / num_threads;
   int end   = (this_thread+1) * size / num_threads;
   for(int p = begin; p < end; ++p) {
#pragma omp flush (done)
     if(done) break;
     if(str[p:p+n] == substr) {
       done = true;
#pragma omp flush (done)
       result = str+p; break;
     }
   }
 }
```

search.c

Produit Matrice - Vecteur I

```
#pragma omp parallel for shared(A,x,res) private(j)
for ( i=0 ; i<A->m ; i++ )
  {
    double tmp = 0.;
    for ( j=0 ; j<A->n ; j++ )
      {
        tmp += MGET(A,i,j) * GET(x, j);
      }
    LET(res, i) = tmp;
  }
```


Produit Matrice - Vecteur II

```
  for ( i=0 ; i<A->m ; i++ )
   {
     tmp = 0.;
#pragma omp parallel for reduction(+:tmp)
     for ( j=0 ; j<A->n ; j++ )
       {
         tmp += MGET(A, i, j) * GET(x, j);
       }
     LET(res, i) = tmp;
   }
```

Produit Matrice - Vecteur III

```
 #pragma omp parallel private(i,j,tmp)
  for ( i=0 ; i<A->m ; i++ )
    {
      tmp = 0.;
#pragma omp single
      LET(res,i) = 0.;
#pragma omp for
      for ( j=0 ; j<A->n ; j++ )
        {
          tmp += MGET(A, i, j) * GET(x, j);
        }
#pragma omp atomic
      LET(res, i) += tmp;
    }
```



```
my_mat_vect : 44.670248
mat_vect_pnl : 21.018873
my_mat_vect_omp_inner : 31.645028
my_mat_vect_omp_inner_2 : 32.285124
my_mat_vect_omp_outer : 9.683041
```

Speedup super-linéaire : effet important de la taille du cache. Calcul réalisé sur `ensipsys` avec une matrice de taille 2,000 x 60,0000.

Flush

Comment assurer qu'une variable partagée a la même valeur dans toutes les threads.

- Permet de rafraîchir l'état d'une variable partagée
- Elle assure la cohérence entre la tâche exécutante et les différents niveaux de la hiérarchie mémoire.
- La directive flush est implicite dans les cas suivants
  - directive barrier
  - début et fin d'une région parallel, critical ou ordered
  - fin d'une région for sections ou single
  - sauf si nowait est utilisée.

Les verrous

Permettent d'éviter que 2 threads exécutent un même code simultanément

- void omp_init_lock(omp_lock_t *lock) : initialise le verrou associé au paramètre lock.
- void omp_destroy_lock(omp_lock_t *lock) : détruit la variable lock, uniquement possible si le verrou n'est pas actif
- void omp_set_lock(omp_lock_t *lock) : rend actif le verrou lock
- void omp_unset_lock(omp_lock_t *lock) : rend inactif le verrou lock
- int omp_test_lock(omp_lock_t *lock) : essaye d'activer le verrou mais ne se bloque pas si ce n'est pas possible. Renvoie un entier non nul si un verrou a été activé et 0 sinon.

Les verrous

- Permettent d'émuler une région critique, ... mais plus finement
- On peut sortir d'un verrou par un saut
- On peut utiliser plusieurs verrous pour exclure mutuellement certaines threads
- En attendant d'acquérir un verrou, une thread peut faire autre chose.

```
omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel
{
  if ( ! omp_test_lock(&lock) )  do_something;
  omp_unset_lock(&lock);
  do_something_else
}
omp_destroy_lock(&lock);
```

lock.c

Quelques conseils

- Minimiser le nombre de régions parallèles dans le code (temps passé à la gestion des threads)
- La directive nowait peut permettre d'éviter une synchronisation inutile.
- Les directives atomic et reduction sont plus performantes que critical
- Réaliser trop de travail dans une section critical ou trop de sections critical
- Trop de variables partagées peuvent sensiblement ralentir un code.

Maximiser les régions parallèles

```
#pragma omp parallel
{
  #pragma omp for /* boucle 1 */
  { ..... }

  #pragma omp for /* boucle 2 */
  { ..... }

  #pragma omp for /* boucle 3 */
  { ..... }
}
```

Il faut préférer les grandes régions parallèles

- Meilleure utilisation du cache
- Moins de temps perdu à la gestion des threads
- Meilleures opitmisaitons du compilateur.

Les boucles internes

```
  for ( i=0 ; i<n ; i++ )
    for ( j=0 ; j<m ; j++)
#pragma omp parallel for
      for ( k=0 ; k<p ; k++ ) {
        .....
      }
```

```
  #pragma omp parallel private (i, j, k)
  for ( i=0 ; i<n ; i++ )
    for ( j=0 ; j<m ; j++)
#pragma omp for
      for ( k=0 ; k<p ; k++ ) {
        .....
      }
```


Trop de travail dans une région critique

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
  #pragma omp critical {
    if (arr[i] > max) max = arr[i];
  }
}
```


Trop de travail dans une région critique

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
  #pragma omp critical {
    if (arr[i] > max) max = arr[i];
  }
}
```

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
  #pragma omp flush(max)
  if (arr[i] > max) {
    #pragma omp critical {
      if (arr[i] > max) max = arr[i];
    }
  }
}
```


Une meilleure solution à la recherche du maximum

```
#pragma omp parallel
{
 int priv_max;
 #pragma omp for
  for (i = 0; i < N; ++i)
    if (arr[i] > priv_max) priv_max = arr[i];
  #pragma omp flush (max)
  if (priv_max > max) {
    #pragma omp critical {
      if (priv_max > max) max = priv_max;
    }
  }
}
```

C'est une réduction de type max. Cette réduction n'existe qu'en Fortran (pas d'opérateur max en C).

▸ *max-reduction.c*

Quelques dangers

- L'utilisation de librairies qui ne sont pas "thread-safe", par exemple `random` : à cause de variables statiques.
  $\rightsquigarrow$ Solution : au lieu d'utiliser une variable interne pour stocker l'état du générateur, le passer en argument des différentes fonctions.
- Un temps de calcul notablement plus élevé que pour le code séquentiel est (souvent) le signe d'apparition d'accès concurrents non gérés par le programmeur.
- Les variables privées masquent les variables globales.
- Dans

```
#pragma parallel for
for ( i=0 ; i<N ; i++ )
```

  N ne peut pas être donné par un #define

Mesure de performance

- Pour mesurer le temps d'exécution d'une partie du code, utiliser omp_get_wtime qui renvoie un temps en secondes (en double)

```
start = omp_get_wtime ()
... parallel stuff
end = omp_get_wtime ()
printf ("temps calcul : %f \n", end - start);
```

- La mesure obtenue peut varier d'une exécution à l'autre en fonction de la répartition du travail entre les threads et de la charge de la machine : on mesure un temps réel et non un temps CPU.
- On mesure la performance par

$$S(N) = \frac{\text{temps code séquentiel}}{\text{temps code parallèle avec N threads}}$$

  On a toujours $S \leq$ nombre de threads.

Alignement mémoire

- Privilégier une parallélisation qui permet à chaque thread de manipuler un bloc contigu de mémoire.
  Par exemple : en C paralléliser les opérations sur des lignes différentes d'une matrice.
- Utiliser le paramètre chunk pour que la taille des blocs manipulés soit inférieure à la taille du cache.
- Dans un produit matrice vecteur, quelle boucle paralléliser ? D'une manière générale la boucle externe.