Programmation parallèle (PPAR)
Cours 5 : programmation avec OpenMP

P. Fortin
pierre.fortin @ upmc.fr

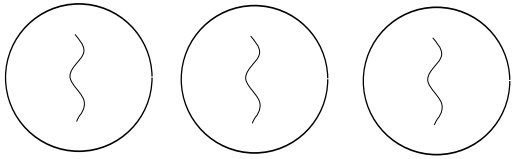
D'après le cours de J.-L. Lamotte

Master 2 Informatique - UPMC

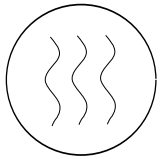
Notions de processus et de thread

Processus : « flot d'exécution » + « espace mémoire »
Thread : « flot d'exécution »

Eléments propres à chaque processus	Eléments propres à chaque thread
Espace d'adressage	Compteur ordinal
Variables globales	Registres
Fichiers ouverts	Pile (dont variables locales)
Processus enfant, signaux...	Etat



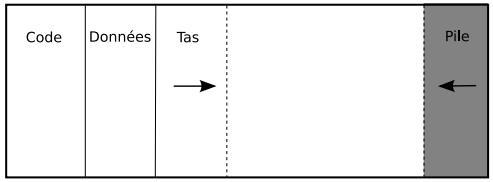
Mode multi-processus



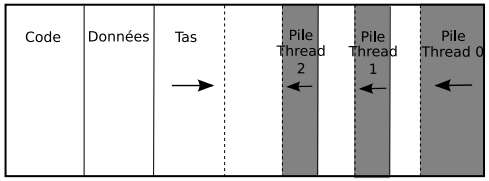
Mode multi-thread

Threads et processus (suite)

Processus mono-thread :



Processus multi-thread :



Toute la mémoire du processus est potentiellement accessible à tous les threads. En pratique, on distingue :

- variables partagées : variables globales ou statiques (segment de données), variables du tas avec pointeur connu de tous les threads
  → Attention aux conflits d'accès à un même emplacement mémoire (*race condition*) ! Exemple : `i = i + 1 ;`
- variables privées : variables locales (pile), variables du tas avec pointeur privé

Programmation en mode multi-thread

Plusieurs expressions possibles du parallélisme :

- parallélisme explicite : threads POSIX
- parallélisme partiellement implicite : OpenMP

⇒ On s'intéresse ici uniquement à OpenMP.

Plusieurs programmations possibles en OpenMP :

- fonctions OpenMP → mode SPMD
  - détermination du travail de chaque thread en fonction de son identifiant (comme MPI, threads POSIX...)
  - efficace mais importantes modifications du code
- directives OpenMP (*#pragma* en C et C++, commentaires en Fortran)
  - objectif : parallélisation des nids de boucles
  - préserve le code initial
  - plus souple (équilibrage de charge)
  - l'efficacité dépend du parallélisme présent dans les boucles

⇒ On privilégie ici les directives OpenMP.

Avantages et inconvénients respectifs d'OpenMP / MPI

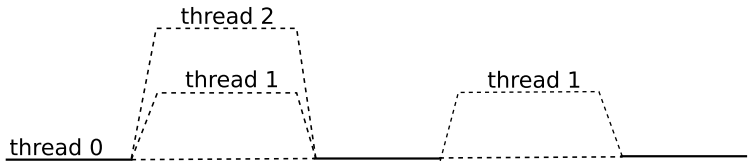
- Avantages d'OpenMP :
  - plus facile à programmer / mettre au point que MPI
  - préserve le code séquentiel original
  - code plus facile à comprendre/maintenir
  - permet une parallélisation progressive
- Inconvénients d'OpenMP :
  - uniquement pour machines à mémoire (virtuellement) partagée
  - actuellement principalement adapté aux boucles parallèles
- Avantages de MPI :
  - s'exécute sur machines à mémoire partagée ou distribuée
  - peut s'appliquer à une gamme de problèmes plus larges qu'OpenMP
  - chaque processus a ses propres variables (pas de conflits)
- Inconvénients de MPI :
  - modifications algorithmiques importantes souvent nécessaires (envoi de messages, recouvrement communications/calcul)
  - peut être plus dur à mettre au point
  - la performance peut dépendre du réseau de communication utilisé

Historique d'OpenMP

- Échec des différents essais de normalisation
- Nécessité de développer un standard pour développer des programmes pour des machines parallèles **à mémoire partagée**
- 28 octobre 1997, des industriels et des constructeurs adoptent OpenMP (*Open Multi Processing*) comme un standard industriel.
- Novembre 2000 : définition d'OpenMP-2
- Version présentée ici : version 2.5 (depuis 2005)
- Version 3.0 (2008) : nouvelle notion de tâche (*task*)
- Nouvelle version 4.0 (directives pour génération de code SIMD et de code pour accélérateurs matériels (GPU...)) en cours d'élaboration
- Interface en Fortran, C et C++

Principe de base

- Un processus unique est exécuté sur une machine parallèle à mémoire partagée. Le thread correspondant est le thread ≪ maître ≫ (de numéro 0).
- Des parties de programme sont exécutés en parallèle par des threads selon le modèle *fork and join* :



- La déclaration des zones parallèles se fait à l'aide de directives OpenMP.
- Modèle mémoire OpenMP particulier : le programmeur peut choisir si une variable est *privée* ou *partagée*.

Création d'un programme OpenMP

- Compilation : les directives de compilation (*#pragma* en C et C++, commentaires en Fortran) sont interprétées si le compilateur les reconnaît. Dans le cas contraire, elles sont assimilées à des commentaires. Les directives indiquent au compilateur comment paralléliser le code.
- Édition de liens : bibliothèques particulières OpenMP
- Exécution : des variables d'environnement sont utilisées pour paramétrer l'exécution

Création d'un programme OpenMP

test.c :

```
#ifdef _OPENMP
#include <omp.h>
#endif

int main(){
#pragma omp parallel
  {
  printf("En parallele !\n");
  printf("Toujours //  !\n");
  }

  printf("En sequentiel.\n");
}
```

test2.c :

```
#ifdef _OPENMP
#include <omp.h>
#endif

int main(){
#pragma omp parallel
  printf("En parallele !\n");

  printf("En sequentiel.\n");
}
```


Compilation et exécution

Compilation

```
gcc test2.c -o test2
gcc -fopenmp test2.c -o test2_openmp
```

Exécution 1 :

```
$ ./test2
En parallele !
En sequentiel.
```

Exécution 2 :

```
$ export OMP_NUM_THREADS=4
$ ./test2_openmp
En parallele !
En parallele !
En parallele !
En parallele !
En sequentiel.
```


Compilation conditionnelle - fonctions OpenMP

Compilation conditionnelle :

```
#ifdef _OPENMP

#endif
```

Fonctions OpenMP :

Il existe aussi un certain nombre de fonctions fournies par OpenMP
→ à réserver à un mode de programmation SPMD

- `omp_get_num_threads()`
- `omp_get_thread_num()`
- `omp_set_num_threads()`
- ...

Hello world

Programme

```
#ifdef _OPENMP
#include <omp.h>
#endif

int main(){
#pragma omp parallel
 {
#ifdef _OPENMP
  printf("Hello world thread %d/%d\n",
  omp_get_thread_num(),
  omp_get_num_threads());
#else
  printf("Hello world\n");
#endif
 }
}
```

Exécution

```
$ gcc hello.c -o hello
$ ./hello
Hello world
$ gcc -fopenmp hello.c \
 -o hello
$ export OMP_NUM_THREADS=4
$ ./hello
Hello world thread 0/4
Hello world thread 3/4
Hello world thread 1/4
Hello world thread 2/4
```

Les directives OpenMP

Elles sont délimitées par une sentinelle :

```
#pragma omp directive [clause ]*[clause]
```

Par défaut, il y a une barrière de synchronisation à la fin.

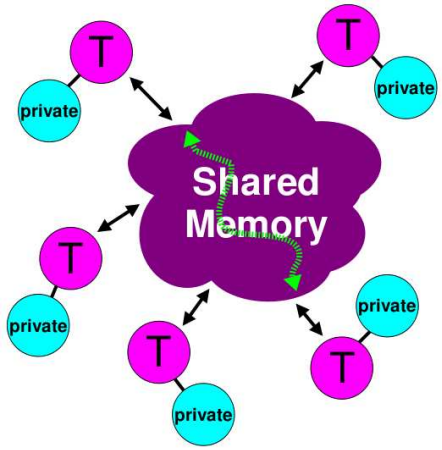
Utilisation des directives OpenMP :

- débranchement externe interdit !
- une seule directive par sentinelle
- majuscule/minuscule importante
- les directives sont : `parallel`, `for`, `sections`, `section`, `single`, `master`, `critical`, `barrier`, `atomic`, `flush`, `ordered`, `threadprivate`

Le modèle mémoire OpenMP

Les variables du code source séquentiel original peuvent être partagées (*shared*) ou privées (*private*) en OpenMP.

- **Variable partagée** : chaque thread accède à la même et unique variable originale.
- **Variable privée** : chaque thread a sa propre copie locale de la variable originale.



(d'après *An Overview of OpenMP 3.0*, R. van der Pas, IWOMP2009)

Le modèle mémoire OpenMP (suite)

- **Par défaut, les variables présentes au début de la région parallèle sont partagées.**
- On peut modifier leur statut avec `private`, `shared`, `firstprivate`, `lastprivate`, `default(shared)`, `default(none)`, `reduction`, `copyin`
- Les variables locales à un thread (i.e. variables locales d'une fonction appellée par un thread depuis une région parallèle, et variables déclarées dans une région parallèle) sont privées.
- **Synchronisations implicites** entre la mémoire principale et la mémoire "locale" de chaque thread (au début et à la fin de certaines directives).

Directive `parallel`

```
#pragma omp parallel [clause ]*[clause]
bloc structuré
```

Définit une région parallèle.

Liste des clauses associées à cette directive :

- `if` (*scalar_expression*) : les threads sont créés si la clause n'est pas présente ou si l'évaluation de *scalar_expression* est non nulle.
- `private` (*liste_de_variables*), `firstprivate` (*liste_de_variables*), `default` (share | none), `copyin`(*liste_de_variables*)
- `reduction`(*opérateur : liste_de_variables*)
- `num_threads`(*expression_entière*) indique explicitement le nombre de threads exécutant cette région parallèle.

```
int main(){
      ...
      initialisation();
#pragma omp parallel ...
 {
      calcul()
}
      post_calcul();
}
```

Il existe en fait plusieurs possibilités pour définir le nombre de threads (par ordre de priorité décroissante) :

#pragma	:	#pragma omp parallel num_threads(16)
au cours de l'execution	:	*omp_set_num_threads(4)*
variable d'environnement	:	export OMP_NUM_THREADS=4

Les variables sont partagées par défaut

Programme

```
#include <omp.h>
#include <stdio.h>

int main()
{
  int c=0;

#pragma omp parallel
  {
    c++;
    printf("c=%d thread %d\n",c, omp_get_thread_num());
  }
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=1 thread 3
c=2 thread 0
c=3 thread 1
c=4 thread 2
```


Attention aux conflits !

Programme

```
#include <omp.h>
#include <stdio.h>

int main()
{
  int i;
  int c=0;

#pragma omp parallel private(i)
  {

    for (i=0; i<100000; i++){ c++; }
    printf("c=%d thread %d\n",c, omp_get_thread_num());
  }
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=100000 thread 0
c=200000 thread 3
c=270620 thread 2
c=286162 thread 1
```


Clause `firstprivate`

Les variables nommées dans `firstprivate` sont locales (privées) aux threads, mais la valeur qu'elles avaient avant d'entrer dans la partie parallèle est conservée

Programme

```
int main(){
  int a;
  a=100;
#pragma omp parallel \
        firstprivate(a)
  {
    a=a+10;
    printf("a=%d\n",a);
  }
  printf("Apres a=%d\n",a);
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
a=110
a=110
a=110
a=110
Apres a=100
```

Clause `private`

Les variables nommées dans `private` sont locales (privées) aux threads.
Elles sont créés à l'initialisation du thread et NON initialisées.
Un exemple de BUG aléatoire :

Programme

```
int main(){
  int a=100;
#pragma omp parallel private(a)
  {
    a=a+10;
    printf("a=%d\n",a);
  }
  printf("Apres a=%d\n",a);
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
a=-1208433038
a=-22
a=-22
a=-22
Apres a=100
```


Variables locales

Toutes les variables locales de fonctions appelées depuis une partie parallèle sont locales (privées) aux threads.
Idem pour les variables déclarées dans le bloc parallèle $\{..\}$.

Programme

```
void func()
{
  int a=10;
  a+=omp_get_thread_num();
  printf("a=%d\n",a);
}
int main(){
#pragma omp parallel
  func();
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ test2
10
11
12
13
```


La directive `for`

`#pragma omp for` *[clause]*[clause]*
boucle for

Clauses associées :

- `private` (*liste_de_variables*), `firstprivate` (*liste_de_variables*),`lastprivate` (*liste_de_variables*)
- `reduction`(*opérateur : liste_de_variables*)
- `ordered`
- `schedule`(*type*, *taille*)
- `nowait`

Cette directive doit être placée juste avant une boucle `for`.

La directive `for` (2)

La boucle doit être de la forme canonique suivante :

`for` (*expr_init* ; *expr_logique* ; *increment*)

- L'indice est de type entier.
- L'incrémentation est de la forme $++$, $--$, $+=$, $-=$, $var = var + inc$, $var = inc + var$, $var = var - inc$, avec un incrément entier.
- Test : $<$, $>$, $<=$, $>=$. La borne est une expression invariante.
- De plus : pas de sortie prématurée de la boucle (`break`, `return`, `exit`).

Conséquences de la directive `for` :

- Barrière implicite à la fin du `for` (sauf si `nowait`).
- **L'indice est une variable privée.**

Exemple

```
#include <omp.h>

int main(){
  int i, t[100];
#pragma omp parallel
  {
#pragma omp for
    for (i=0; i<100; i++){
      t[i] = i;
    }
  }
}
```

Avec 4 threads, le premier peut par exemple calculer les t[i] de 0 a 24, le second de 25 a 49, ...

Clause `lastprivate`

Les variables nommées dans `lastprivate` sont locales (privées) aux threads. A la fin de la partie parallèle, elles sont affectées avec les valeurs obtenues par le thread $N-1$.

Programme

```
  int a,i;

#pragma omp parallel
#pragma omp  for lastprivate(a)
  for(i=0;i<4;i++){
      a=i*10
      printf("PAR a=%d thread %d \n",
              a, omp_get_thread_num());
  }
  printf("SEQ a=%d %d\n",a);
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
PAR a=0 thread 0
PAR a=10 thread 1
PAR a=20 thread 2
PAR a=30 thread 3
SEQ a=30
```


Forme raccourcie pour la directive `for`

`#pragma omp parallel for` *[clause]*[clause]*
boucle for

Cette directive admet toutes les clauses de `parallel` et de `for` à l'exception de `nowait`.
Elle doit se situer sur la ligne juste avant une boucle `for`.

exemple :

```
#pragma omp parallel for lastprivate(a) private(j)
```


La clause `reduction`

Programme

```
int main(){
  int a[4][4],s=0, i,j;
  for(i=0;i<4;i++)
    for(j=0;j<4;j++)
      a[i][j]=i*4+j;
#pragma omp parallel
  {
#pragma omp  for reduction(+:s) private(j)
    for(i=0;i<4;i++){
      for(j=0;j<4;j++) s=s+a[i][j];
      printf("PAR=%d : i=%3d  s=%d\n",
     omp_get_thread_num(),i,s);
    }
  }
  printf("SEQ s=%d\n",s);
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
PAR=0 : i=  0  s=6
PAR=1 : i=  1  s=22
PAR=2 : i=  2  s=38
PAR=3 : i=  3  s=54
SEQ s=120
```

Partage des calculs et ordonnancement des threads

2 clauses : `schedule` et `nowait`

- clause `nowait` :
  Par défaut, il y a une synchronisation à la fin de la boucle.
- clause `schedule(mode, chunk_size)` :
  4 modes : `static, dynamic, guided, runtime`

  Par défaut, le choix dépend de l'implémentation d'OpenMP utilisée.

Exemple `schedule`

```
#define MAX 10
int main(){
  int a[MAX],i, imin, imax;
#pragma omp parallel private(imin,imax)
    {
      imax=0;
      imin=MAX;
#pragma omp for schedule(...)
      for(i=0;i<MAX;i++){
        imin=i<imin?i:imin;
        imax=i>imax?i:imax;
        a[i]=1;
        sleep(1); /* pour augmenter la charge */
        printf("%3d:%3d\n",omp_get_thread_num(),i);
      }
      printf("thread %d imin=%d imax=%d\n",
              omp_get_thread_num(),imin,imax);
    }
}
```


Clauses `static` et `dynamic`

schedule(static)

```
  2:  6
  3:  9
  0:  0
  1:  3
  2:  7
  0:  1
  1:  4
  2:  8
  0:  2
  1:  5
thread 1 imin=3 imax=5
thread 3 imin=9 imax=9
thread 0 imin=0 imax=2
thread 2 imin=6 imax=8
```

schedule(static,2)

```
  0:  0
  2:  4
  3:  6
  1:  2
  0:  1
  3:  7
  2:  5
  1:  3
  0:  8
  0:  9
thread 0 imin=0 imax=9
thread 3 imin=6 imax=7
thread 2 imin=4 imax=5
thread 1 imin=2 imax=3
```

schedule(dynamic,2)

```
  3:  6
  1:  2
  2:  4
  0:  0
  1:  3
  2:  5
  0:  1
  3:  7
  1:  8
  1:  9
thread 1 imin=2 imax=9
thread 2 imin=4 imax=5
thread 0 imin=0 imax=1
thread 3 imin=6 imax=7
```


schedule

- `schedule (static,n)` : `n` indique la taille des paquets (*chunk*).
  La distribution des paquets est ensuite cyclique entre les threads.
  Valeur de *n* par défaut $\approx \frac{\text{Nbre d'iterations}}{\text{Nbre de threads}}$.
  $\rightarrow$ équilibrage de charge statique
- Le remplacement de `static` par `dynamic` change la politique l'affectation d'un paquet de données à un thread. Dans le cas `dynamic`, les blocs sont affectés aux premiers threads disponibles.
  Valeur de *n* par défaut : 1.
  $\rightarrow$ équilibrage de charge dynamique
- `guided` : équilibrage de charge dynamique avec une taille de bloc proportionnelle au nombre d'itérations encore non attribuées divisé par le nombre de threads (taille décroissante vers 1)
- `runtime` : le choix (`static, dynamic` ou `guided`) est reporté à l'exécution $\rightarrow$ à privilégier
  Exemple : `export OMP_SCHEDULE="static,1"`

Clause et directive `ordered`

Clause `ordered` et directive `ordered` $\longrightarrow$ exécution séquentielle (débogage)

```
#pragma omp parallel private(imin,imax)
    {
      imax=0;
      imin=MAX;
#pragma omp for ordered schedule(static,2)
      for(i=0;i<MAX;i++){
        int t;
        imin=i<imin?i:imin;
        imax=i>imax?i:imax;
        a[i]=i;
#pragma omp ordered
        printf("%3d:%3d\n",omp_get_thread_num(),i);
      }
      printf("thread %d imin=%d imax=%d\n",
             omp_get_thread_num(),imin,imax);
    }
```


Directive `sections`

Chaque section est exécutée par un unique thread.

```
#pragma omp sections [clause ]*[clause]
{
[#pragma omp section ]
bloc structuré
[#pragma omp section
bloc structuré]
...
}
```

Listes des clauses possibles :

- `private` (*liste_de_variables*), `firstprivate` (*liste_de_variables*),`lastprivate` (*liste_de_variables*)
- `reduction`(*opérateur : liste_de_variables*)
- `nowait`

Directives `sections`

Programme

```
#pragma omp parallel
  #pragma omp sections
    {
    #pragma omp section
      printf("section 1 thread %d\n",
             omp_get_thread_num());
    #pragma omp section
      printf("section 2 thread %d\n",
             omp_get_thread_num());
    #pragma omp section
      printf("section 3 thread %d\n",
             omp_get_thread_num());
    #pragma omp section
      printf("section 4 thread %d\n",
             omp_get_thread_num());
    }
```

Exécution

```
$ export OMP_NUM_THREADS=3
section 1 thread 0
section 2 thread 0
section 3 thread 1
section 4 thread 2
```


Directive `single`

```
#pragma omp single  directive [clause ]*[clause]
bloc structuré
```

But : définir, dans une région parallèle, une portion de code qui sera exécutée par un seul thread.

- Barrière implicite à la fin de `single`.
- `nowait` et `copyprivate` sont incompatibles.
- Rien n'est dit sur le thread qui exécute la directive.

Listes des clauses possibles :

- `private` (*liste_de_variables*), `firstprivate` (*liste_de_variables*), `copyprivate` (*liste_de_variables*)
- `nowait`

Directive `single`

Programme

```
int main(){
  int a=10;
#pragma omp parallel firstprivate(a)
    {
#pragma omp single
      a=20;

      printf("thread %d a=%d\n",
     omp_get_thread_num(),a);
    }
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
thread 0 a=20
thread 1 a=10
thread 3 a=10
thread 2 a=10
```


Directive `single` et clause `copyprivate`

Programme

```
int main(){
#pragma omp parallel
    {
      int a=10;
#pragma omp single copyprivate(a)
      a=20;

      printf("thread %d a=%d\n",
     omp_get_thread_num(),a);
    }
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
thread 0 a=20
thread 1 a=20
thread 3 a=20
thread 2 a=20
```


Directive `master`

```
#pragma omp master
bloc structuré
```

- Pas de clause
- Pas de barrière implicite
- Seul le thread 0 (master) exécute le code associé

Synchronisations en OpenMP

Plusieurs possibilités :

- barrières
- directives `atomic` et `critical`
- verrous via fonctions OpenMP (non traitées ici) :
  `omp_init_lock()`
  `omp_{set,test}_lock()`
  `omp_unset_lock()`
  `omp_destroy_lock()`

Directive `barrier`

```
#pragma omp barrier
```

Lorsqu'un thread rencontre un point de synchronisation, il attend que tous les autres soient arrivés à ce même point.

Remarque : pour des problèmes de syntaxe C :

```
if (n!=0)
     #pragma omp barrier
```

est incorrecte

```
if (n!=0) {
     #pragma omp barrier
}
```

est correcte

Directive `atomic`

```
#pragma omp atomic
expression-maj
```

- La mise à jour est atomique.
- L'effet ne concerne que l'instruction suivante.
- *expression-maj* est de la forme :
  - `++x, x++, --x, x--`
  - *x binop= expr*,
    - *binop* ne doit pas être surchargé (C++, Fortran90/95),
    - *binop* $\in \{+, *, -, /, \&, \wedge, |, >>, <<\}$,
    - l'expression *expr* ne doit pas faire référence à *x*.
- Seul le chargement et la mise à jour de la variable forment une opération atomique, l'évaluation de l'expression *expr* ne l'est pas.

Programme

```
#include <omp.h>
#include <stdio.h>

int main()
{
  int i;
  int c=0;

#pragma omp parallel private(i)
  {

    for (i=0; i<100000; i++){
#pragma omp atomic
       c++;
    }
    printf("c=%d thread %d\n",c, omp_get_thread_num());
  }
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=100000 thread 0
c=294308 thread 2
c=394308 thread 3
c=400000 thread 1
```


Directive `critical`

```
#pragma omp critical [nom]
bloc structuré
```

- Un seul thread à la fois peut exécuter le bloc d'une directive `critical`
- Le thread est bloqué à l'entrée du bloc structuré tant qu'un autre thread exécute un bloc portant le même nom.
- Le nom est utile pour l'édition multi-fichiers et pour distinguer des sections critiques indépendantes.

Programme

```
int main(){
  int cpt=0, rang,i,index[1024],nbt;
#pragma omp parallel private(rang)
    {
      rang= omp_get_thread_num();
      #pragma omp critical
      {
         index[cpt]=rang;
         cpt++;
      }
      #pragma omp master
        nbt=omp_get_num_threads();
    }
    for(i=0;i<nbt;i++)
      printf("i=%d index[]=%d\n",i, index[i]);
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
index[0]=0
index[1]=3
index[2]=1
index[3]=2
```


Différence `atomic, critical`

`atomic :`

- destinée à la mise à jour de variables
- dépend du matériel et du système d'exploitation (instructions atomiques du processeur)

`critical :`

- destinée à englober une partie plus importante de code
- implémentation avec des verrous a priori

Directive `threadprivate`

Permet de définir le statut des variables statiques dans les threads.
Une variable `threadprivate` ne doit pas apparaître dans une autre clause sauf pour `copyin, copyprivate, schedule, num_thread, if`.
Une variable `threadprivate` ne doit pas être une référence (C++).

Directive `threadprivate`

Programme

```
static int a=10;
#pragma omp threadprivate(a)


int main(){
  int rang;

#pragma omp parallel copyin(a) private(rang)
    {
      rang= omp_get_thread_num();
      a+=rang;
      printf("thd=%d a=%d\n",
     rang,a);
    }
    printf("SEQ a=%d\n",a);
}
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
thd=0 a=10
thd=3 a=13
thd=1 a=11
thd=2 a=12
SEQ a=10
```

Parallélisme imbriqué (nesting)

Il doit respecter

- une directive `parallel` dans une directive `parallel`
- un certain nombre de règles sont à respecter (cf OpenMP Ref. Manuel)
- une variable d'environnement : `OMP_NESTED` doit prendre la valeur TRUE (ou FALSE) pour autoriser (ou non) le parallélisme imbriqué (non autorisé par défaut).

Parallélisme imbriqué

Programme

```
#define MAX 4
  int i,j;
#pragma omp parallel
 {
   #pragma omp for
   for(i=0;i<MAX;i++){
     #pragma omp parallel
      {
       #pragma omp for
       for(j=0;j<MAX;j++){
 printf("%3d:%3d \n",
            omp_get_thread_num(),
            i*MAX+j);
       }
      }
   }
 }
```

Exécution

```
$ export OMP_NUM_THREADS=4
$ export OMP_NESTED=TRUE
  0:  0    3:  7
  1:  1    1: 13
  2:  2    2: 14
  3:  3    3: 15
  0:  4    0:  8
  0: 12    1:  9
  1:  5    2: 10
  2:  6    3: 11
```


Bibliothèque OpenMP

Liste des fonctions :

- `void omp_set_num_threads(int` *num_thread*`)` : fixe le nombre de threads utilisable par le programme. L'appel doit se situer dans une région séquentielle. Dans une région parallèle, le comportement est inconnu.
- `int omp_get_num_threads(void)`
- `int omp_get_max_threads(void)` : retourne le nombre de threads maximum qui sera utilisé pour la prochaine région parallèle (valable si la clause `num_threads` n'est pas utilisée).
- `int omp_get_thread_num(void)`
- `int omp_get_num_procs(void)`
- `int omp_in_parallel(void)` : retourne un entier non nul si l'appel a lieu dans une région parallèle.

Bibliothèque OpenMP (2)

Liste des fonctions :

- `void omp_set_nested(int` *nested*`)` : *nested* prend pour valeur 0 pour désactiver l'imbrication du parallélisme. L'appel doit se situer dans une région séquentielle. Dans une région parallèle, le comportement est inconnu.
- `int omp_get_nested(void)`
- `double omp_get_wtime(void)` la différence entre deux appels permet de calculer le *wall-clock time* en secondes.
- `int omp_get_wtick(void)`

Les variables d'environnement

- `OMP_NUM_THREADS`
- `OMP_SCHEDULE` :
  `export OMP_SCHEDULE=''guided,4''`
  `export OMP_SCHEDULE=''dynamic''`
- `OMP_NESTED`
- ...

Références

- Site officiel et spécifications : `http://www.openmp.org`
- Communauté des utilisateurs : `http://www.compunity.org`
- Cours de l'IDRIS sur OpenMP :
  `http://www.idris.fr/data/cours/parallel/openmp/`