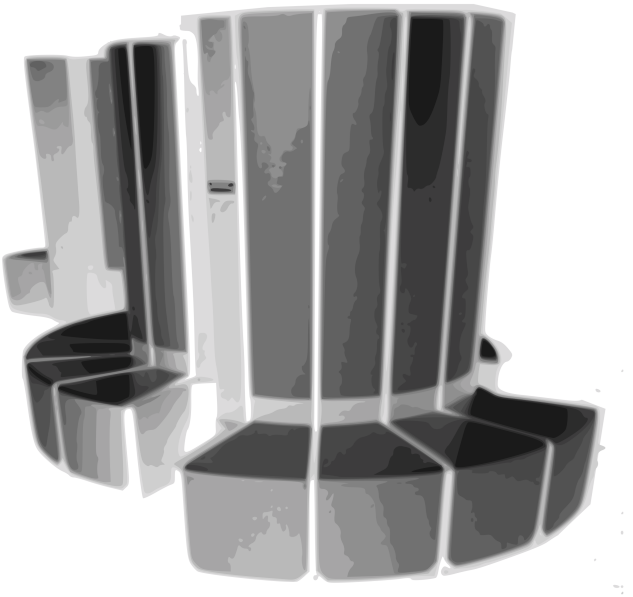
Faculté des Sciences & Techniques

Université de Limoges

Master 1ère année



Parallélisme & Applications — P-F. Bonnefoi

Version du 4 octobre 2024

Table des matières

1 Présentation d'OpenMP 3
Le processus de parallélisation 4
2 Les directives OpenMP 5
3 Le partage du travail entre les threads, «*work sharing*» 9
«WorkSharing» : la directive «`for`» 11
«WorkSharing» : la directive «`sections`» 15
Les fonctions de bibliothèques utilisables durant l'exécution, «runtime» 16
Les variables d'environnement 17
4 La synchronisation 18
5 Les «tasks» 21
6 Le SIMD : l'utilisation d'instructions vectorielles 31




OpenMP utilise trois types de construction pour contrôler la parallélisation d'un programme :

- ⋆ des directives de compilation ;
- ⋆ des fonctions de bibliothèques utilisables lors de l'exécution ;
- ⋆ des variables d'environnement.

Pour compiler un programme OpenMP :

xterm
```
$ gcc -fopenmp -o mon_programme mon_source.c
```

Un exemple de code :

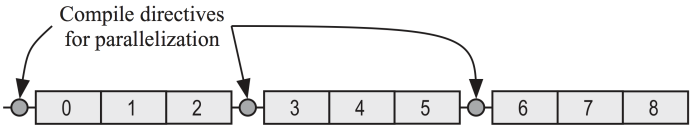
```
 1 #include <omp.h>
 2
 3 int main () {
 4 int var1, var2, var3;
 5
 6 /* Partie séquentielle exécutée par la master thread */
 7
 8 ...
 9
10 #pragma omp parallel private(var1, var2) shared(var3)
11 {
12    Partie parallèle exécutée par toutes les threads
13    ...
14 }
15
16 /* Reprise de la partie séquentielle */
17 }
```

Explications :

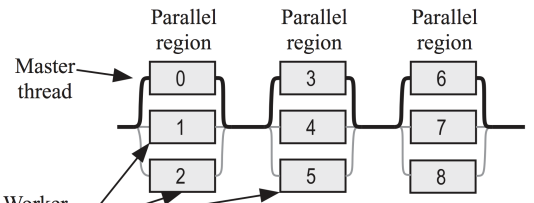
- ▷ la ligne 11 définit le **début du block** correspondant à la partie parallèle :
  - ◇ toutes les *threads* prévues sont lancées, *forked* ;
  - ◇ le block de code est dupliqué entre toutes les threads ;
- ▷ la ligne 14 marque la **fin du block** :
  - ◇ toutes les threads se synchronisent avec la «master thread», *join*, et s'arrêtent ;
  - ◇ la «master thread» reprend l'exécution séquentielle.

Annotation d'un code

OpenMP utilise des directives de compilation pour créer, synchroniser des threads et distribuer le travail qu'elles doivent réaliser :



(a)



(b)

- la version séquentielle est donnée en (a) : elle est composée de différents blocks numérotés ;
- des directives de compilation sont insérées manuellement par le programmeur au début d'un ensemble de block.
  Ces directives ordonnent au compilateur de créer des threads aux endroits indiqués ;
- le schéma (b) indique comment le compilateur crée autant de threads que demandé pour exécuter en parallèle les sections de code ;
- une synchronisation, *join*, est automatiquement ajoutée à la fin de chaque section pour s'assurer que le programme continue son exécution une fois que chaque thread a terminé son travail.
- la «master thread» indiquée par une ligne plus sombre s'occupe de créer les différentes threads.
  Chaque thread est identifiée par un «ID» entier et la «master thread» reçoit le numéro 0.

Le rôle des différentes annotations

- les **directives de compilation** indiquent au compilateur comment paralléliser le code ;
- les **fonctions de bibliothèques** permettent de modifier/connaître le nombre de threads et de connaître le nombre de processeurs/cœurs du système ;
- les v**ariables d'environnement** qui permettent de modifier l'exécution du programme OpenMP.

Les directives contrôlent :
- ▷ la création de threads ;
- ▷ la distribution de la charge de travail, «workload» ;
- ▷ la gestion des données ;
- ▷ la synchronisation entre les threads.

Le format d'une directive est le suivant :

```
#pragma omp nom_directive [clause, …]
```

Exemple ⇒

```
#pragma omp parallel default(shared) private(a,b)
{/* tout le code du bloc est exécuté en parallèle */
   ...
}
```

Directive pragma OpenMP	Description
`#pragma omp parallel`	définie une région parallèle qui doit être **exécutée par plusieurs threads**, le processus original devenant la «master thread». *Toutes les threads exécutent la région parallèle.*
`#pragma omp atomic`	impose la **modification** d'une variable de manière **atomique**
`#pragma omp barrier`	**synchronise** toutes les threads appartenant à la même région parallèle
`#pragma omp critical`	impose que le block de code qui suit soit exécuté par **une seule thread** à la fois
`#pragma omp flush`	opération de **synchronisation** qui garantit que toutes les threads de la région parallèle auront la même valeur pour la variable spécifiée
`#pragma omp for`	indique que les différentes occurrences de la boucle doivent être **exécutées en parallèle** en utilisant les différentes threads
`#pragma omp sections`	**répartie** différents travaux effectué chacun par une thread différente si possible
`#pragma omp section`	définie un travail pour la directive précédente (doit être contenue dans le corps de la directive `sections`)




Il est possible de combiner «région parallèle» et «répartition de travail» :

Full version	Combined construct

#pragma omp parallel
{
  #pragma omp for
  `for-loop`
}
	#pragma omp parallel for
	`for-loop`

#pragma omp parallel
{
#pragma omp sections
{
  [**#pragma omp section**]
  structured block
  [**#pragma omp section**
  structured block]
  ...
}
}
	#pragma omp parallel sections
	{
	  [**#pragma omp section**]
	  structured block
	  [**#pragma omp section**
	  structured block]
	  ...
	}

Cette version peut être plus efficace du point de vue du code généré par le compilateur qui connait à la fois la région parallèle et le travail à répartir.




Les clauses utilisables dans les directives de compilation

Certaines des directives de compilation utilisent une ou plusieurs clauses :

- l'ordre dans lequel est écrit les clauses n'est pas important ;
- la plupart des clauses acceptent une liste d'éléments séparés par des virgules ;
- les clauses gèrent les partages des données entre les threads ;
- certaines clauses gèrent la copie d'une variable privée d'une thread à une variable correspondante d'une autre thread.

Directive	**Clause**
Parallel	Copying, default, private, firstprivate, reduction, shared
Sections	Private, firstprivate, lastprivate, reduction, schedule, nowait
Section	Private, firstprivate, lastprivate, reduction
Critical	Aucune
Barrier	Aucune
Atomic	Aucune
Flush (*liste*)	Aucune
Ordered	Aucune
Threadaptive (*liste*)	Aucune

Clause	**Description**
default(*mode*)	permet de contrôler le mode par défaut pour le partage de l'accès aux variables, le mode peut être *private*, *shared*, *none*
shared(*liste*)	donne la liste des variables à partager entre les différentes threads créées par la directive «parallel». Exemple : `#pragma omp parallel default(shared)`
copyin(*liste*)	copie la valeur des variables de la liste depuis la «master thread» vers les autres threads
num_threads(n *entier*)	requiert la création de n threads

- `single` : une section de code ne doit être **exécutée que par un seule thread** et pas nécessairement la `master` ;
- `critical` : le code ne doit être exécuté que par **une thread à la fois** :

```
1 int a=0, b=0;
2 #pragma omp parallel num_threads(4)
3 {
4     #pragma omp single
5     a++;
6     #pragma omp critical
7     b++;
8 }
9 printf("single: %d -- critical: %d\n", a, b);
```

xterm

```
$ single_vs_critical
single: 1 -- critical: 4
```

Master *vs* atomic

- `master` similaire à single avec **deux différences** :
  - `master` est exécuté uniquement par la thread «*master*» alors que `single` est exécuté par la première thread atteignant la région ;
  - `single` a une **barrière de synchronisation implicite** à la fin de la région où toutes les threads attendent la fin de la région alors que `master` n'en a pas ;
- `atomic` est similaire à `critical` mais est **restraint** à de **simples opérations** :
  - `atomic update` celle par défaut ,
  - `atomic read`,
  - `atomic write`
  - `atomic capture`.




- Les directives de «partage de travail» contrôlent quelles threads exécutent quelles instructions.
- Ces directives ne créent pas de threads.
- Ces directives sont au nombre de deux : «`#pragma omp for`» et «`#pragma omp sections`».

La directive `#pragma omp for`

Il existe différentes clauses possibles pour cette directive, en particulier :

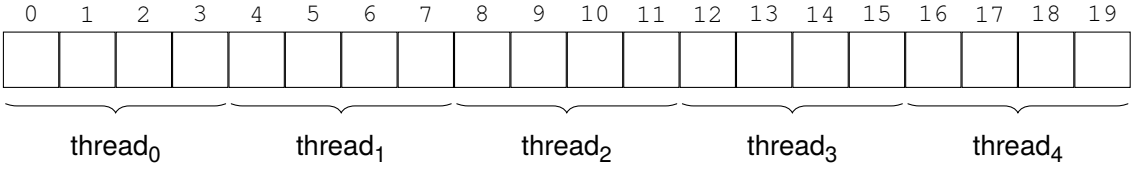
- `schedule(static, 3)` indique que le nombre total d'itérations de la boucle doit être divisé par 3, et la valeur obtenue représente le nombre d'itérations à répartir entre les différentes threads.
  Le «static», indique que cette répartition doit être fait de manière **régulière** et **cyclique** entre les différentes threads classées par leur ID ;
- `schedule(dynamic, 3)` le découpage est similaire à l'exemple précédent, mais le «dynamic» indique que la répartition des itérations doit être fait **en fonction de la première thread disponible** : quand une thread termine sa tâche, elle cherche à prendre le prochain bloc de travail, «chunk», disponible.

Clauses pour le `#pragma omp for`	**Description**
schedule(type [, chunk size])	l'ordonnancement peut être statique ou dynamique. Elle décrit comment le travail est réparti entre les threads, le nombre d'itération de la boucle réalisé par chaque thread est égal à «chunk size».
private(liste)	la liste des variables «privées» de chaque thread
firstprivate(liste)	variables initialisées avec la valeur connue avant le début de la région ou du block
lastprivate(liste)	variables mises à jour à la sorite de la région ou du block
shared(liste)	variables partagées entre les threads
reduction(operator:list)	réalise une «réduction» sur les variables indiquées dans la liste en utilisant l'opérateur `+ * - & | ^ && ||`. Cet opérateur travaille sur la sortie de chaque thread une fois qu'elles ont fini leur exécution.
nowait	Les threads ne se synchronisent pas à la fin de la région «omp for».



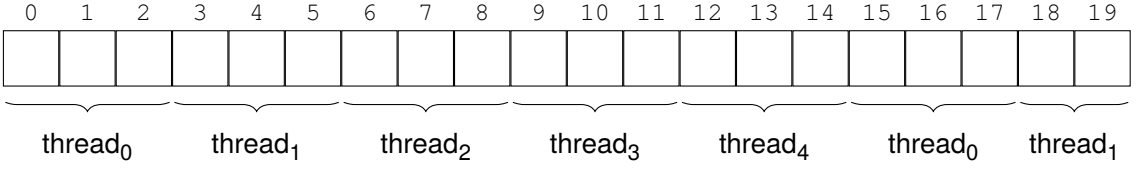

Répartition «*static*»

20 cases pour 5 threads $\Longrightarrow$ 4 cases par threads :



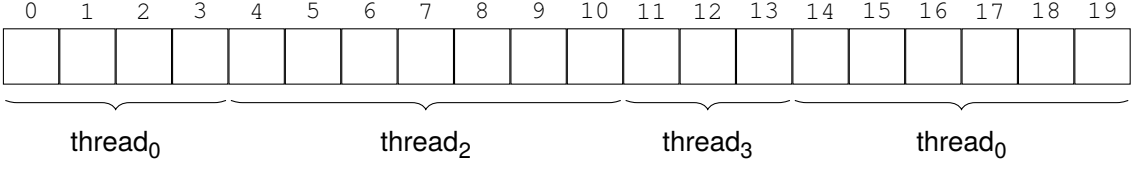
Répartition «*static*» par «*chunk*»

20 cases pour 5 threads par chunk de 3 :



La répartition du travail est cyclique : une thread reçoit un bloc de cases à traiter, puis un autre jusqu'à épuisement global des cases.

Répartition «*dynamic*»



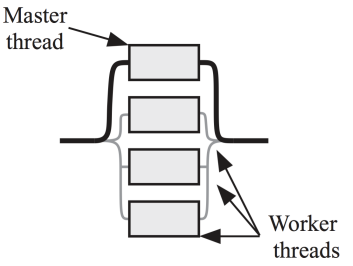
La répartition se fait suivant la disponibilité des différents cœurs.



«WorkSharing» : la directive «`for`»

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main (int argc, char *argv[])
6  {  int i = 0, n = 100;
7     float a[100], b[100], sum;
8     for (i=0; i < n; i++)
9        a[i] = b[i] = i*1.0;
10    sum = 0.0;
11
12    #pragma omp parallel for schedule(dynamic,16) reduction(+:sum)
13    for (i = 0; i < 64; i++)
14       sum = sum + (a[i]* b[i]);
15
16    printf( "sum = %f\n", sum);
17 }
```

Le schéma de parallélisation :



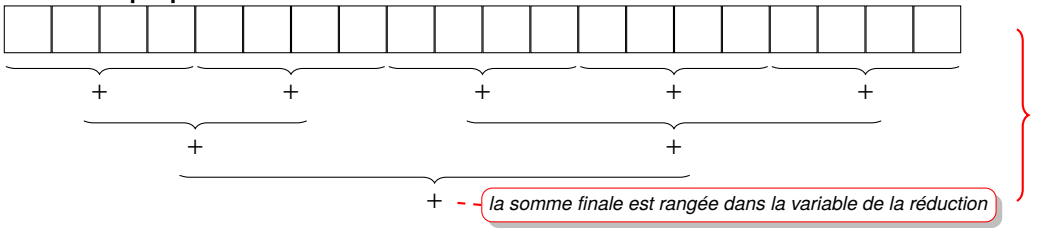
- ligne 13 : la directive de compilation ordonne l'exécution en parallèle des différentes itérations (qui sont indépendantes les unes des autres).

Attention

OpenMP ne garantie/vérifie pas la capacité d'un code à être exécuté en parallèle.

- la répartition est «*dynamic*», chaque cœur disponible reçoit, comme travail, 16 itérations de boucle.
- l'opération de **réduction** applique l'addition sur les différentes valeurs de la variable «`sum`», calculées par chacune des threads.

Autre exemple pour la réduction des valeurs d'un tableau :



Les sommes successives sont faites suivant les disponibilités des cœurs

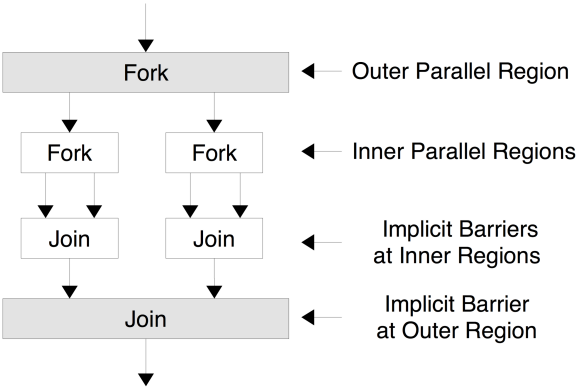
Un autre exemple

```
1  #pragma omp parallel default (none) \
2     shared (x, m, n) private (i, j, h1, h2, y1, y2)
3  { /* début de la région parallèle  */
4     #pragma omp for nowait
5     for (i = 0; i < m; i++) {
6        for (j = 0; j < n; j++)                    ❶
7           y1(i) = y1(i) + h1(j) *x(i-j);
8        }
9     #pragma omp for nowait
10    for (i = 0; i < m; i++) {
11       for (j = 0; j < n; j++)                    ❷
12          y2(i) = y2(i) + h2(j) *x(i-j);
13       }
14    } /* fin de la région parallèle */
```

- la région parallèle débute en ligne 3 et finit en ligne 14 ;
- la clause «`default(none)`» (l'autre option possible serait «shared») définit la portée par défaut des variables dans chaque thread ; *none* indique que c'est uniquement le programmeur qui décide du partage ou non, *y compris pour la variable utilisée pour la boucle* ;
- en ligne 2, la clause «`shared(x, m, n)`» indique que ces trois variables données dans la liste doivent être partagées par toutes les threads ;
- la clause «`private (i, j, h1, h2, y1, y2)`» indique que ces variables doivent être privées pour chaque thread.
- la ligne 4 est une directive utilisée pour paralléliser la boucle ❶.
  Les itérations de la boucle vont être réparties entre les différentes threads, et chaque thread va exécuté la boucle imbriquée, *nested*, (en ligne 6) de manière séquentielle.
- la clause «nowait» indique que les threads ne se synchroniseront pas à la fin de la boucle extérieur, ce qui évite la mise en œuvre implicite de la barrière de synchronisation à la fin de la parallélisation de la boucle.
- la ligne 9 est similaire à la ligne 4, pour la boucle ❷.



Le «nested parallelism» ou parallélisme imbriqué



Le «parallélisme imbriqué», ou «*nested parallelism*» peut être activé :

- en l'activant avec l'appel à la fonction de la bibliothèque `omp_set_nested()`.
  Dans le programme, son activation peut être testée par la fonction `omp_get_nested()`.
- en positionnant la variable d'environnement `OMP_NESTED` à `TRUE` ;

```
1  #pragma omp parallel
2  {
3     #pragma omp for private(i, j)
4     for (i=0; i <n; i++) {
5        #pragma omp parallel ----- requis
6        {
7           #pragma omp for private(j)
8           for (j=0; i<n; j++) {
9              do_work(i,j)
10          }
11       }
12    }
13 }
```

barrière implicite

barrière implicite

Attention

- le nombre de threads utilisées va croître de manière exponentielle ;
- seules les **régions parallèles** peuvent être imbriquées et non les directives «worksharing» de répartition de travail (il faut obligatoirement le mot clé `parallel` dans la directive imbriquée) ;
- les barrières de synchronisation implicites de fin de région ne peuvent être supprimées (pas de `nowait` possible) ;
- les directives de barrière, `#pragma omp barrier`, et de section critique, `#pragma omp critical`, ne peuvent être imbriquées dans une directive «woksharing».

```
#include <stdio.h>

int main() {
  #pragma omp parallel
  {
    #pragma omp for
      for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
          printf("The loop iterators are %d and %d.\n", i, j);
  }
}
```

```
$ gcc -o my_program my_program.c
-Wall -fopenmp
$ ./my_program
The loop iterators are 2 and 0.
The loop iterators are 2 and 1.
The loop iterators are 2 and 2.
The loop iterators are 0 and 0.
The loop iterators are 0 and 1.
The loop iterators are 0 and 2.
The loop iterators are 1 and 0.
The loop iterators are 1 and 1.
The loop iterators are 1 and 2.
```

Utilisation de la clause `collapse(n)`

```
#include <stdio.h>

int main() {
  #pragma omp parallel
  {
    #pragma omp for collapse(2)
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        printf("The loop iterators are %d and %d.\n", i, j);
  }
}
```

```
$ gcc -o my_program my_program.c
-Wall -fopenmp
$ ./my_program
The loop iterators are 2 and 2.
The loop iterators are 0 and 0.
The loop iterators are 2 and 1.
The loop iterators are 0 and 1.
The loop iterators are 2 and 0.
The loop iterators are 1 and 2.
The loop iterators are 0 and 2.
The loop iterators are 1 and 0.
The loop iterators are 1 and 1.
```

$\Longrightarrow$ Le «*collapse*» transforme une **double** imbriquée en une **seule** boucle imbriquée avec tous les couples de valeurs de (i, j).



«WorkSharing» : la directive «sections»

```
1  #pragma omp parallel
2  {
3     #pragma omp sections [clause [ ···]]
4     {
5        #pragma omp section
6        {
7           ...                    ❶
8        }
9        #pragma omp section
10       {
11          ...                    ❷
12       }
13       #pragma omp section
14       {
15          ...                    ❸
16       }
17    }
18 }
```

- la ligne 1 définit la région parallèle ;
- la ligne 3 ordonne au compilateur d'exécuter chacune des «section» dans une thread séparée :
  - la ligne 5 définit la section ❶ exécutée par une thread ;
  - la ligne 9 définit la section ❷ exécutée par une seconde thread ;
  - la ligne 13 définit la section ❸ exécutée par une troisième thread.

Utilisation du «nowait»

La clause «nowait» permet aux threads qui exécutent :

▷ **une section** et qui finissent plus rapidement que celles s'occupant d'une autre section ;

▷ **aucune section** ;

de passer au travail situé après la partie «sections» et contenu dans la région parallèle globale.



Le fichier d'entête «`omp.h`» contient les prototypes des différentes fonctions.

Ces fonctions :

- ▷ contrôlent l'environnement d'exécution ;
- ▷ contrôlent et surveillent les threads ;
- ▷ contrôlent et surveillent les processeurs.

Exemples de fonction

`void omp_set_num_threads(int num_threads);` contrôle le nombre de threads utilisées pour la région parallèle qui sera définie après et qui ne possède pas de clause `num_threads`.

Fonction	Description
`double omp_get_wtime(void);`	retourne le temps réel, *elapsed wall clock*, en secondes. *Ce temps prends en compte le temps utilisateur, le temps système ainsi que les ralen tissements dus à la surcharge du système.*
`double omp_get_wtick(void);`	retourne la précision du temps utilisé par la fonction `omp_get_wtime`
`omp_set_num_threads`	définit le nombre de threads pour la région parallèle
`omp_get_num_threads`	retourne le nombre de threads
`omp_get_thread_num`	retourne le numéro de la thread courante
`omp_get_num_procs`	retourne le nombre de processeur/cœur
`omp_in_parallel`	retourne vrai si la fonction est appelée depuis une région parallèle



Les variables d'environnement

Des variables d'environnement sont utilisées pour modifier l'exécution de programmes OpenMP.

Les usages de certaines variables d'environnement :

- ◇ le nombre de threads ;
- ◇ le type de politique d'ordonnancement, «scheduling policy» ;
- ◇ le parallélisme imbriqué, «nested parallelism» ;
- ◇ la limite du nombre de threads.

Les variables d'environnement possèdent :

- ⋆ des noms tout en majuscules ;
- ⋆ des valeurs insensibles à la casse (pas de différences minuscules/majuscules).

Clauses `#pragma omp for`	**Description**
`OMP_NUM_THREADS` *nombre*	définit le nombre de threads de la région parallèle
`OMP_DYNAMIC` *true/false*	ajuste dynamiquement le nombre de threads de la région parallèle
`OMP_THREAD_LIMIT` *limite*	contrôle le nombre maximum de threads utilisées dans le programme OpenMP

Pour définir une variable d'environnement :

```
$ export OMP_NUM_THREADS=4
```




Des fonctions basées sur l'utilisation de «loquets» permettent de :

- ▷ synchroniser les threads entre elles ;
- ▷ garantir l'intégrité des données lors de leur écriture/lecture simultanée par les différentes threads.

La syntaxe est la suivante :

`void omp_`*fonction*`_lock(omp_lock_t *lck)` où *fonction* peut être : `init`, `destroy`, `set`, `unset` et `test`.

```
 1 #include <omp.h>
 2 #include <stdio.h>
 3 int main() {
 4    omp_lock_t loquet;
 5    omp_init_lock(&loquet);
 6    #pragma omp parallel shared(loquet)
 7    {
 8       int id = omp_get_thread_num();
 9       omp_set_lock(&loquet);
10       printf("Mon numero de thread est %d\n",id);
11       omp_unset_lock(&loquet);
12       while(!omp_test_lock(&loquet))
13          autre_travail(id);
14       travail(id);
15       omp_unset_lock(&loquet);
16    }
17    omp_destroy_lock(&loquet);
18 }
```

Deux types de fonctionnement :

- ⋆ **Non bloquant** : l'opération `omp_test_lock` permet de tester et de positionner le loquet :
  - ◇ s'il est occupé la fonction retourne faux ;
  - ◇ s'il est libre, la fonction positionne le loquet et retourne vrai.
- ⋆ **Bloquant** : l'opération `omp_set_lock` bloque la thread tant qu'il n'est pas possible de positionner le loquet.

Sur l'exemple ci-contre, une thread réalise un `autre_travail` de manière répétitive en attendant d'obtenir le loquet.

Il est possible d'utiliser une version autorisant l'usage imbriqué où la même thread peut positionner le loquet plusieurs fois en suffixant par `_nest`*, comme par exemple :* `omp_set_nest_lock`*.*

OpenMP offre des directives de compilation pour synchroniser l'exécution des threads.

Attention : Le programmeur doit éviter inter-blocages qui peuvent découler de ces synchronisations.

Il existe 5 directives de synchronisation :

- ◇ critical ;
- ◇ atomic ;
- ◇ barrier.
- ◇ ordered ;
- ◇ flush ;

La directive `barrier`

Cette directive, `#pragma omp barrier`, synchronise **toutes** les threads :

- ◇ lorsqu'une thread atteint la barrière, elle attend que toutes les autres threads atteignent aussi la barrière.
- ◇ lorsque toutes les threads ont atteint la barrière, elles reprennent l'exécution du code situé après la barrière.

La directive critical

Cette directive définit une section critique, c-à-d que les threads exécutant du code code parallèle vont suspendre leur exécution à leur arrivée à cette directive : une thread va exécuter la section de code suivant la directive seulement lorsqu'aucune autre thread ne l'exécute.

```
 1 #include <omp.h>
 2
 3 int main ()
 4 {    int x = 0;
 5    #pragma omp parallel
 6    {
 7       /* instructions */
 8       …
 9       #pragma omp critical
10       x = x+1;
11       /* instructions */
12    }
13 }
```

- ◇ la ligne 5 définit une section de code qui doit être exécutée par toutes les threads ;
- ◇ la ligne 9 indique que l'instruction de la ligne 9 ne peut exécutée que par une seule thread **à la fois** et que toutes les threads qui ont atteint cette ligne doivent attendre.

Ainsi, la directive `critical` garantit qu'une section de code ne peut être exécutée que par une thread à la fois, et ce, sans interruption des autres threads.

Les directives «orphelines» ou «orphan directives»

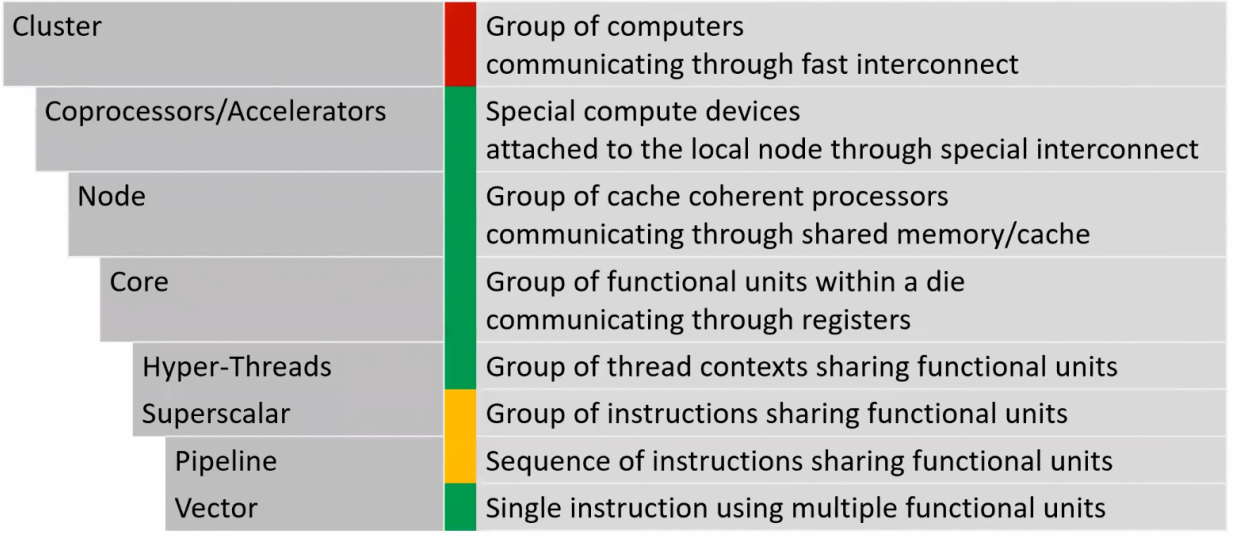
Ce sont des directives qui sont définies **en dehors d'une région parallèle**, mais qui seront **activées** lorsqu'une région parallèle fera appel à la **fonction qui les contient**.

⟹*Utile pour définir des fonctions de bibliothèque utilisable en séquentiel ou en parallèle.*



OpenMP

Levels of Parallelism in OpenMP 5.0



https://www.youtube.com/watch?v=YZCWPkKLVYM

- la directive «task» crée une **tâche** de manière **explicite** ;
- toutes les threads se **partagent le travail disponible** parmi **l'ensemble des tâches créées** ;
- la directive «taskwait» **garantie** que toutes les tâches enfants crées dans la même tâche courante **ont fini**.

Fonctionnement :

▷ lorsqu'une tâche est créée, elle rejoint un «*pool*» de tâches ;

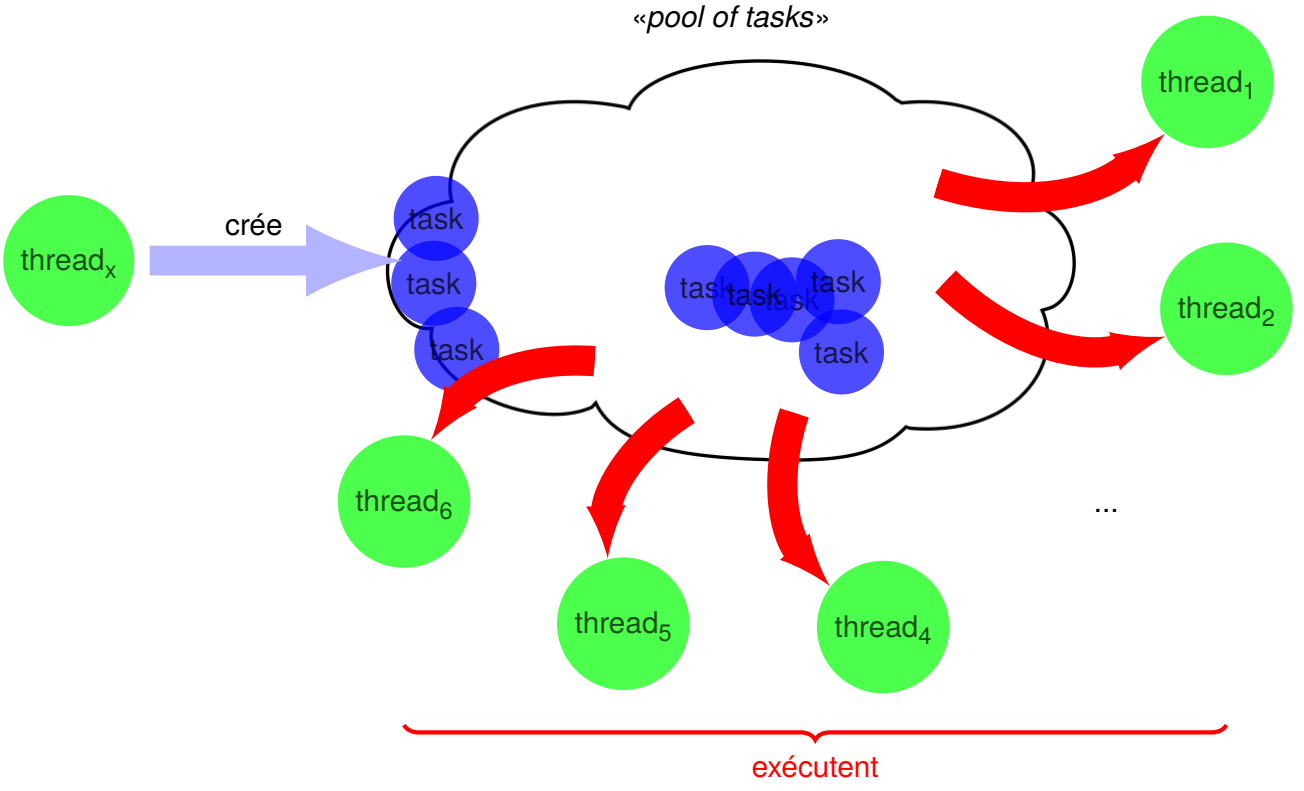
▷ chaque thread matérielle peut exécuter une tâche du «*pool*».

Exemple :

```
 1 #include <omp.h>
 2 #include <stdio.h>
 3 void main()
 4 {
 5    int i;
 6    #pragma omp parallel private(i)
 7    {   for(i=0; i<10; i++)
 8       #pragma omp task
 9       {      printf("Task %d: %i\n",
10              omp_get_thread_num(),i);
11       }
12    }
13 }
```

Grâce à la directive « `#pragma omp task` *», une* ***tâche*** *est ajoutée à chaque occurence de la boucle* `for`.

Les «*tasks*» et le «*pool of tasks*»



- Une thread$_x$ crée un ensemble de «*tasks*» qui rejoignent un «*pool of tasks*» ;
- l'ensemble des threads disponibles, y compris la thread$_x$, exécutent une plus ou plusieurs «*tasks*», jusqu'à vider le *pool*.

Exemples d'interactions

```
#include <stdio.h>
int main () {

   #pragma omp parallel
   {
      #pragma omp single
      {
         printf("Hello ,\n");

         printf("world!\n");

      }
   }
   return 0;
}
```

```
#include <stdio.h>

int main () {
   #pragma omp parallel
   {
       #pragma omp single
      {
         #pragma omp task
         printf("Hello ,\n");
         #pragma omp task
         printf("world!\n");
      }
   }
   return 0;
}
```

```
#include <stdio.h>
int main () {
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
         printf("Hello ,\n");
         #pragma omp task
         printf("world!\n");
         printf("Bye\n");
      }
   }
   return 0;
}
```

```
#include <stdio.h>
int main () {
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
         printf("Hello ,\n");
         #pragma omp task
         printf("world!\n");
         #pragma omp taskwait
         printf("Bye\n");
      }
   }
   return 0;
}
```

Quelles sont les sorties ?

Les nombres de Fibonacci

```
int fib (int n)
{
   int x,y;
   if ( n < 2 ) return n;
   x = fib(n-1);
   y = fib(n-2);
   return x+y;
}

int main() {
   int N = 45;
   fib(N);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Implémentation récursive inefficace en $O(n^2)$

```
xterm
pef@cube $ time ./fib
1134903170./fib  14.28s user 0.00s system
99% cpu 14.286 total
```

Une implémentation parallèle

```
long int fib ( int n )
{
   long int x,y;
   if ( n < 2 ) return n;
   #pragma omp task shared(x)
   x = fib(n-1);
   #pragma omp task shared(y)
   y = fib(n-2);
   #pragma omp taskwait
   return x+y;
}
int main()
{ int N=35;
    #pragma omp parallel
    {
        #pragma omp master
        printf("%ld\n",fib(N));
    }
}
```

Attention, c'est bien 35 !

```
xterm
pef@cube $ time ./fib_omp
9227465
./fib_omp  43.64s user 3.30s system 273%
cpu 17.151 total
```

Le temps augmente !

Si on supprime le «*taskwait*» :

```
xterm
pef@cube $ time ./fib_omp
140521173189380
./fib_omp  1.41s user 0.00s system 364% cpu
0.385 total
```

$\Longrightarrow$ la valeur calculée est **fausse** !

$\Longrightarrow$ Attention aux **mauvaises utilisations** du parallélisme...

Limiter la création de «*tasks*» par l'utilisation d'une directive «*if*»

```
long int fib ( int n )
{
   int x,y;
   if ( n < 2 ) return n;
   #pragma omp task shared(x) if(n>30)
   x = fib(n-1);
   #pragma omp task shared(y) if(n>30)
   y = fib(n-2);
   #pragma omp taskwait
   return x+y;
}
int main()
{ int N=45;
    #pragma omp parallel
    {
      #pragma omp master
      printf("%ld\n",fib(N));
   }
}
```

désactive la directive si faux

valeur 45

On arrête de créer des tâches à partir d'un certain niveau de l'arbre récursif :

xterm

```
pef@cube $ time ./fib_omp2
1134903170
./fib_omp2  173.46s user 0.04s system 272%
cpu 1:03.61 total
```

Pour une condition de «*if (n>35)*» :

xterm

```
pef@cube $ time ./fib_omp2
1134903170
./fib_omp2  179.28s user 0.03s system 272%
cpu 1:05.78 total
```

On limitant encore plus la création de «*tasks*»...

Pour la directive «*if (n>40)*»

xterm

```
pef@cube $ time ./fib_omp2
1134903170
./fib_omp2  143.49s user 0.01s system 210% cpu
1:08.26 total
```

```
long int fib ( int n )
{
   int x,y;
   if ( n < 2 ) return n;
   #pragma omp task shared(x) if(n>40)
   x = fib(n-1);
   #pragma omp task shared(y) if(n>40)
   y = fib(n-2);
   #pragma omp taskwait
   return x+y;
}
...
```


Création de tâches dans une boucle

```
#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < n; i++) {
        #pragma omp task
        task_implementation(data[i]);
    }
}
```

*La granularité de parallélisme peut être **trop fine*** ⟹ *il peut être plus efficace de regrouper des instances de boucles ensemble.*

Répartition de travail entre les tâches

```
#pragma omp parallel
#pragma omp single
{
    int per_task = 10;
    for (int i = 0; i < n; i += per_task) {
        #pragma omp task
        {
            for (int j = 0; j < per_task && i+j < n; j++)
                task_implementation(data[i+j]);
        }
    }
}
```

Compliqué et difficile de gérer la granularité...

Utilisation de `taskloop`

Le `taskloop` permet de résoudre ce problème :

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskloop
    for (int i = 0; i < n; i++)
        task_implementation(data[i]);
}
```

*Une **barrière de synchronisation** est **implicite** avec `taskloop`, ce qui assure que le travail de toutes les tâches associées à la boucle est fini avant de passer à la suite.*

```
if([ taskloop :] scalar-expression)
shared(list), private(list)
firstprivate(list), lastprivate(list)
reduction([default ,]reduction-identifier : list)
in_reduction(reduction-identifier : list)
default(shared or none)
grainsize(grain-size)
num_tasks(num-tasks)
collapse(n)
final(scalar-expr)
priority(priority-value)
untied
mergeable
nogroup
allocate([allocator :] list)
```

nombre d'itérations par task

nombre de tasks à générer

```
#include <stdio.h>
int main() {
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        {
            #pragma omp task
            printf("Hello.\n");
            printf("Hi.\n");
        }
        #pragma omp taskwait
        printf("Goodbye.\n");
    }
    return 0;
}
```

```
xterm
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
Hi.
Goodbye.
Hello.
```

La tâche affichant `Hello` est une descendante de celle affichant `Hi` ⟹ elle peut afficher après le `taskwait`.

Utilisation `taskgroup`

```
#include <stdio.h>

int main() {
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        {
            #pragma omp task
            printf("Hello.\n");

            printf("Hi.\n");
        }

        printf("Goodbye.\n");
    }
    return 0;
}
```

```
xterm
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
Hi.
Hello.
Goodbye.
$ ./my_program
Hello.
Hi.
Goodbye.
```

Le `Goodbye` est affiché après la fin du `taskgroup`

```
#include <stdio.h>

int main() {
    #pragma omp parallel
    #pragma omp single nowait
    {
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                #pragma omp task
                printf("Hello.\n");

                printf("Hi.\n");
            }
        }
        printf("Goodbye.\n");
    }
    return 0;
}
```

Tasks et réduction


```
void example() {
    int result = 0;
   #pragma omp parallel   // create parallel team
   #pragma omp single     // have only one task creator
    {
        #pragma omp taskgroup task_reduction(+:result)
        {
            while(have_to_create_tasks()) {
                #pragma omp task in_reduction(+:result)
                {   // this tasks contribute to the reduction
                    result = do_something();
                }
                #pragma omp task firstprivate(result)
                {   // this task does not contribute to the reduction
                    result = do_something_else();
                }
            }
        }
    }
}
```

la variable recevant la contribution de chaque task

une task contribuant à la réduction

depend([depend-modifier,]dependence-type : locator-list)

in
out
inout
mutexinoutset
depobj

```c
#include <stdio.h>

int main() {
  int number;

  #pragma omp parallel
  #pragma omp single nowait
  {
    #pragma omp task
    number = 1;

    #pragma omp task
    {
      printf("The number is %d\n", number);
      number++;
    }
    #pragma omp task
    printf("The final number is %d\n", number);
  }
  return 0;
}
```

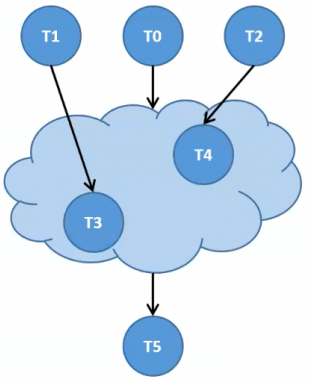
```
xterm
$ gcc -o my_program my_program.c -Wall -fopenmp
$ ./my_program
The number is 1
The final number is 1
$ ./my_program
The final number is 1
The number is 1
```

```c
#include <stdio.h>

int main() {
  int number;

  #pragma omp parallel
  #pragma omp single nowait
  {
    #pragma omp task depend(out: number)
    number = 1;

    #pragma omp task depend(inout: number)
    {
      printf("The number is %d\n", number);
      number++;
    }

    #pragma omp task depend(in: number)
    printf("The final number is %d\n", number);
  }

  return 0;
}
```

```
xterm
$ gcc -o my_program my_program.c -Wall
-fopenmp
$ ./my_program
The number is 1
The final number is 2
```

New Task Dependencies

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)  //T0
  res = 0;

  #pragma omp task depend(out: x)  //T1
  long_computation(x);

  #pragma omp task depend(out: y)  //T2
  short_computation(y);

  #pragma omp task depend(in: x) depend(mutexinoutset: res) //T3
  res += x;

  #pragma omp task depend(in: y) depend(mutexinoutset: res) //T4
  res += y;

  #pragma omp task depend(in: res)  //T5
  std::cout << res << std::endl;
}
```



Code scalaire :

- exécute le code une seule instruction à la fois ;
- toutes les instructions scalaires n'ont pas d'équivalent vectoriel ;

  Code vectoriel :
  - exécute le code plusieures instructions à la fois ;
  - SIMD, «*Single Instruction Multiple Data*» ;
  - toutes les instructions vectorielles n'ont pas d'équivalents (exemple : `shuffle`)

AVX : Advanced Vector Extension

Scalaire	1 élément à la fois
	`addss xmm1, xmm2`
SSE	4 élément à la fois
	`addps xmm1, xmm2`
AVX	8 éléments à la fois
	`vaddps ymm1, ymm2, ymm3`
MIC / AVX-512	16 éléments à la fois
	`vaddps zmm1, zmm2, zmm3`

SIMD

Version Hardware :

Où chaque couleur exprime un «flot SIMD».

Version software :

```
for (int i = 0; i < size; i++) {
   C[i] = A[i] + B[i];
}
```

Par groupe de 4 :

```
for (int i=0; i < size; i += 4) {
   for( int j = 0; j < 4; j++)
      C[i+j] = A[i+j] + B[i+j];
}
```

```c
#include <immintrin.h>  // Include the AVX header file

int main() {
  float a[8] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
  float b[8] = {8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0};
  float c[8];  // The result array

  __m256 a_avx = _mm256_loadu_ps(a);  // Load the first array into an AVX register
  __m256 b_avx = _mm256_loadu_ps(b);  // Load the second array into an AVX register

  __m256 c_avx = _mm256_add_ps(a_avx, b_avx);  // Add the two arrays using AVX instructions

  _mm256_storeu_ps(c, c_avx);  // Store the result into the result array

  // Print out the result
  for (int i = 0; i < 8; i++) {
    printf("%f ", c[i]);
  }
  printf("\n");

  return 0;
}
```

On utilise des **instructions spéciales** du processeur définies dans le fichier `immintrin.h`.

Pour compiler :

xterm

```
$ gcc -O3 -mavx2 -o exemple_simd exemple_simd.c
```

L'option `-mavx2` permet d'activer la compilation des instructions AVX.

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

int main() {
  float a[8] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
  float b[8] = {8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0};
  float c[8];  // The result array

  #pragma omp parallel for
  for (int i = 0; i < 8; i += 8) {
    __m256 a_avx = _mm256_loadu_ps(&a[i]);  // Load the first array into an AVX register
    __m256 b_avx = _mm256_loadu_ps(&b[i]);  // Load the second array into an AVX register

    __m256 c_avx = _mm256_add_ps(a_avx, b_avx);  // Add the two arrays using AVX instructions

    _mm256_storeu_ps(&c[i], c_avx);  // Store the result into the result array
  }

  // Print out the result
  for (int i = 0; i < 8; i++) {
    printf("%f ", c[i]);
  }
  printf("\n");

  return 0;
}
```

Pour compiler :

xterm

```
$ gcc -fopenmp -O3 -mavx2 -o exemple_omp_simd exemple_omp_simd.c
```

Sur cet exemple, la taille du tableau étant de 8 il n'y a du travail que pour une thread...

Le SIMD un exemple plus gros

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

#define SIZE 1073741824

float a[SIZE];
float b[SIZE];
float c[SIZE];  // The result array

int main() {
  double start,end;

  #pragma omp parallel for
  for(int i=0; i < SIZE; i ++)
  {
    a[i] = 10.0;
    b[i] = 3.0;
  }
  printf("OMP for static + instructions AVX2\n");
  start = omp_get_wtime();

  #pragma omp parallel for schedule(static)
  for (int i = 0; i < SIZE; i += 8) {
    __m256 a_avx = _mm256_loadu_ps(&a[i]);  // Load the first array into an AVX register
    __m256 b_avx = _mm256_loadu_ps(&b[i]);  // Load the second array into an AVX register
    __m256 c_avx = _mm256_mul_ps(a_avx, b_avx);  // Add the two arrays using AVX instructions
    _mm256_storeu_ps(&c[i], c_avx);  // Store the result into the result array
  }

  end = omp_get_wtime();
  // Print out the result
  for (int i = 0; i < 16; i++) {
    printf("%.1f ", c[i]);
  }
  printf("\n");
  printf("Duration %lf\n", end-start);
```


Le SIMD un exemple plus gros (suite)


```
// Execution omp uniquement
printf("\nOMP for dynamic\n");
start = omp_get_wtime();

#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < SIZE; i ++) {
  c[i] = a[i] * b[i];
}

end = omp_get_wtime();
// Print out the result
for (int i = 0; i < 16; i++) {
  printf("%.1f ", c[i]);
}
printf("\n");
printf("Duration %lf\n", end-start);

// Execution omp uniquement
printf("\nOMP for static\n");
start = omp_get_wtime();

#pragma omp parallel for schedule(static)
for (int i = 0; i < SIZE; i ++) {
  c[i] = a[i] * b[i];
}

end = omp_get_wtime();
// Print out the result
for (int i = 0; i < 16; i++) {
  printf("%.1f ", c[i]);
}
printf("\n");
printf("Duration %lf\n", end-start);
```


Le SIMD un exemple plus gros (suite et fin)


```
// Utilisation de omp simd
printf("\nOMP + simd\n");
start = omp_get_wtime();

int j;
#pragma omp parallel for private(j)
for (int i = 0; i < SIZE; i +=8) {
  #pragma omp simd
  for(j=0;j<8;j++)
  c[i+j] = a[i+j] * b[i+j];
}

end = omp_get_wtime();
// Print out the result
for (int i = 0; i < 16; i++) {
  printf("%.1f ", c[i]);
}
printf("\n");
printf("Duration %lf\n", end-start);
return 0;
}
```

Pour compiler :

xterm

```
$ gcc -fopenmp -O3 -mavx2 -mcmodel=large -o omp_avx omp_avx.c
```


Le SIMD un exemple plus gros (exécution)

xterm

```
bonnep02@fst-o-i-212-06:~/OPENMP$ ./omp_avx
OMP for static + instructions AVX2
30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0
Duration 0.465490

OMP for dynamic
30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0
Duration 27.500794

OMP for static
30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0
Duration 0.564993

OMP + simd
30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0 30.0
Duration 0.564375
```

On peut s'assurer que le code compilé contient bien des instructions SIMD :

xterm

```
$ objdump -d omp_avx | cut -f 3 | grep '^v' | head -n 10
vmovsd %xmm0,0x28(%rsp)
vmovsd %xmm0,0x38(%rsp)
vxorpd %xmm1,%xmm1,%xmm1
vcvtss2sd -0x4(%r14),%xmm1,%xmm0
vmovsd 0x38(%rsp),%xmm5
vsubsd 0x28(%rsp),%xmm5,%xmm0
vmovsd %xmm0,0x38(%rsp)
vmovsd %xmm0,0x50(%rsp)
vxorpd %xmm2,%xmm2,%xmm2
vcvtss2sd -0x4(%r14),%xmm2,%xmm0
```


Le SIMD : l'utilisation d'instructions vectorielles

```
for ( int i = 0; i < N; i++ ) {
   a[i] = b[i] * scalar + c[i];
}
```

En openMP 4.5 :

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i+=16) {
   #pragma vector nontemporal(a)
   #pragma omp simd aligned(a,b,c)
   for(j = 0; j < 16; j++) {
      a[i+j] = b[i+j] * scalar + c[i+j];
   }
}
```

En openMP 5 :

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i += 16) {
   #pragma omp simd aligned(a,b,c) nontemporal(a)
   for(j = 0; j < 16; j++) {
      a[i+j] = b[i+j] * scalar + c[i+j];
   }
}
```

Pour compiler :

xterm

```
gcc -Wall -fopenmp -march=core-avx2 -o omp_simd omp_simd.c
```

Pour vérifier la présence des instructions vectorielles :

xterm

```
$ objdump -d omp_simd | cut -f 3 | grep '^v'
vmovq  %xmm0,%rax
vmovq  %xmm0,%rax
vmovq  %rax,%xmm2
vsubsd -0x50(%rbp),%xmm2,%xmm1
...
```