PROGRAMMATION A BASE DE THREADS

Marc Pérache marc.perache@cea.fr

Arthur Loussert

Droits de reproduction

Introduction

- OpenMP est un modèle de programmation parallèle qui initialement ciblait uniquement les architectures à mémoire partagée. Aujourd'hui, il cible aussi les accélérateurs, les systèmes embarqués et les systèmes temps réel.
- Les tâches de calcul peuvent accéder à un espace mémoire commun, ce qui limite la redondance des données et simplifie les échanges d'information entre les tâches.
- En pratique, la parallélisation repose sur l'utilisation de processus système légers (ou *threads*), on parle alors de programme *multithreadé*.

Introduction : historique

- La parallélisation multithreadée existait depuis longtemps chez certains constructeurs (ex. CRAY, NEC, IBM, ...), mais chacun avait son propre jeu de directives.
- Le retour en force des machines multiprocesseurs à mémoire partagée a poussé à définir un standard.
- La tentative de standardisation de PCF (Parallel Computing Forum) n'a jamais été adoptée par les instances officielles de normalisation.
- Le 28 octobre 1997, une majorité importante d'industriels et de constructeurs ont adopté OpenMP (Open Multi Processing) comme un standard dit « industriel ».
- Les spécifications d'OpenMP appartiennent aujourd'hui à l'ARB (Architecture Review Board), seul organisme chargé de son évolution.

Introduction : spécifications OpenMP

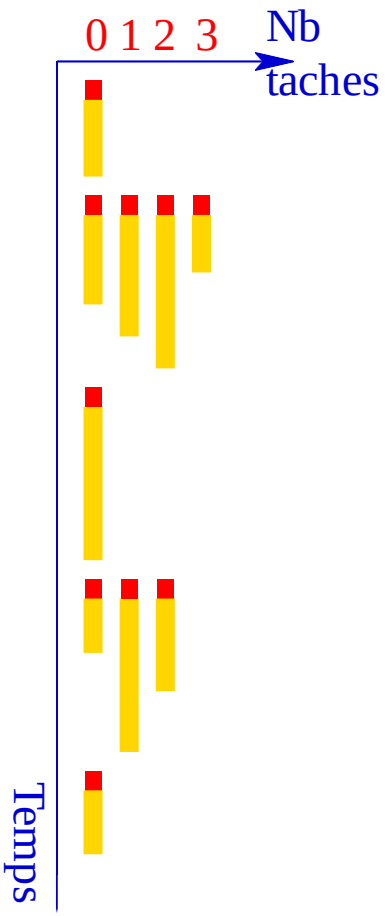
- Une version OpenMP-2 a été finalisée en novembre 2000. Elle apporte surtout des extensions relatives à la parallélisation de certaines constructions Fortran 95.
- La version OpenMP-3 datant de mai 2008 introduit essentiellement le concept de tâche.
- La version OpenMP-4 de juillet 2013 apporte de nombreuses nouveautés, avec notamment le support des accélérateurs, des dépendances entre les tâches, la programmation SIMD (vectorisation) et l'optimisation du placement des threads.

Introduction : terminologie et définitions

- Thread : Entité d'exécution avec une mémoire locale (stack)
- Team : Un ensemble de un ou plusieurs threads qui participent à l'exécution d'une région parallèle.
- Task/Tâche : Une instance de code exécutable et ses données associées. Elles sont générées par les constructions PARALLEL ou TASK.
- Variable partagée : Une variable dont le nom permet d'accéder au même bloc de stockage au sein d'une région parallèle entre tâches.
- Variable privée : Une variable dont le nom permet d'accéder à différents blocs de stockage suivant les tâches, au sein d'une région parallèle.
- Host device : Partie matérielle (généralement nœud SMP) sur laquelle OpenMP commence son exécution.
- Target device : Partie matérielle (carte accélératrice de type GPU ou Xeon Phi) sur laquelle une partie de code ainsi que les données associées peuvent être transférées puis exécutées.

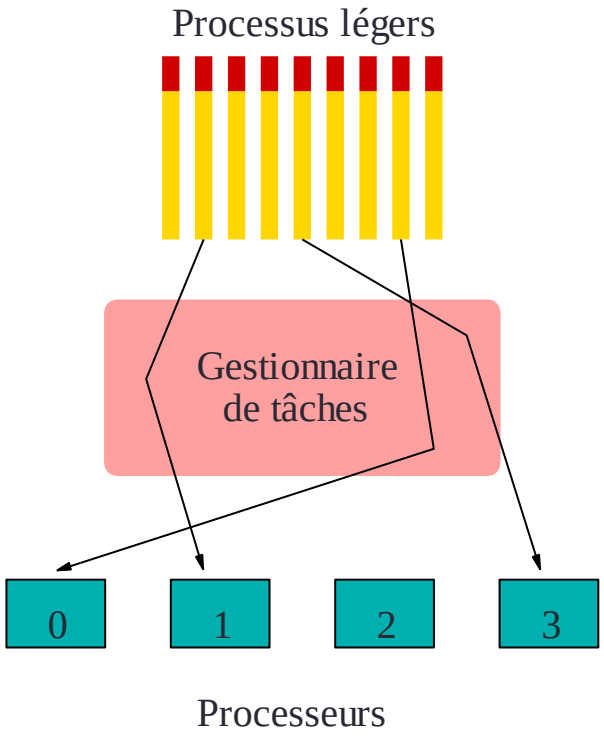
Introduction : concepts généraux

- A son lancement, un programme OpenMP est séquentiel. Il est constitué d'un processus unique, le thread maître dont le rang vaut 0, qui exécute la tâche implicite initiale.
- OpenMP permet de définir des régions parallèles, c'est à dire des portions de code destinées à être exécutées en parallèle.
- Au début d'une région parallèle, de nouveaux processus légers sont créés ainsi que de nouvelles tâches implicites, chaque thread exécutant sa tâche implicite, en vue de se partager le travail et de s'exécuter concurremment.
- Un programme OpenMP est une alternance de régions séquentielles et de régions parallèles.



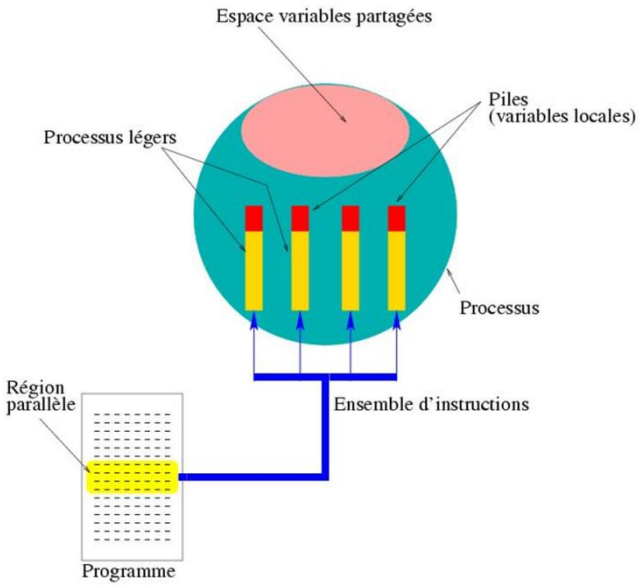
Introduction : concepts généraux

- Chaque processus léger exécute sa propre séquence d'instructions, qui correspond à sa tâche implicite.
- C'est le système d'exploitation ou la bibliothèque de threads qui choisient l'ordre d'exécution des processus (légers ou non) : il les affecte aux unités de calcul disponibles (cœurs des processeurs).
- Il n'y a aucune garantie sur l'ordre global d'exécution des instructions d'un programme parallèle.



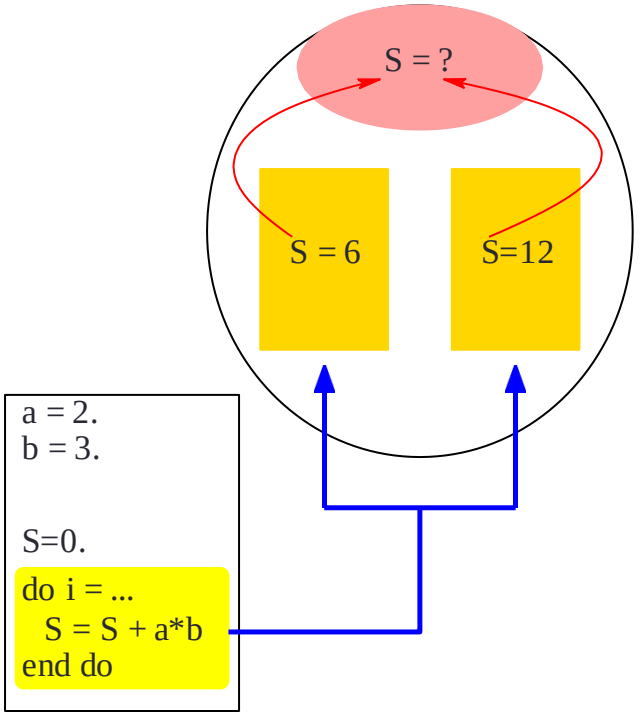
Introduction : concepts généraux

- Les tâches d'un même programme partagent l'espace mémoire de la tâche initiale (mémoire partagée) mais disposent aussi d'un espace mémoire local : la pile (ou stack).
- Il est ainsi possible de définir des variables partagées (stockées dans la mémoire partagée) ou des variables privées (stockées dans la pile de chacune des tâches).



Introduction : concepts généraux

- En mémoire partagée, il est parfois nécessaire d'introduire une synchronisation entre les tâches concurrentes.
- Une synchronisation permet par exemple d'éviter que deux threads ne modifient dans un ordre quelconque la valeur d'une même variable partagée (cas des opérations de réduction).



Introduction : fonctionnalités

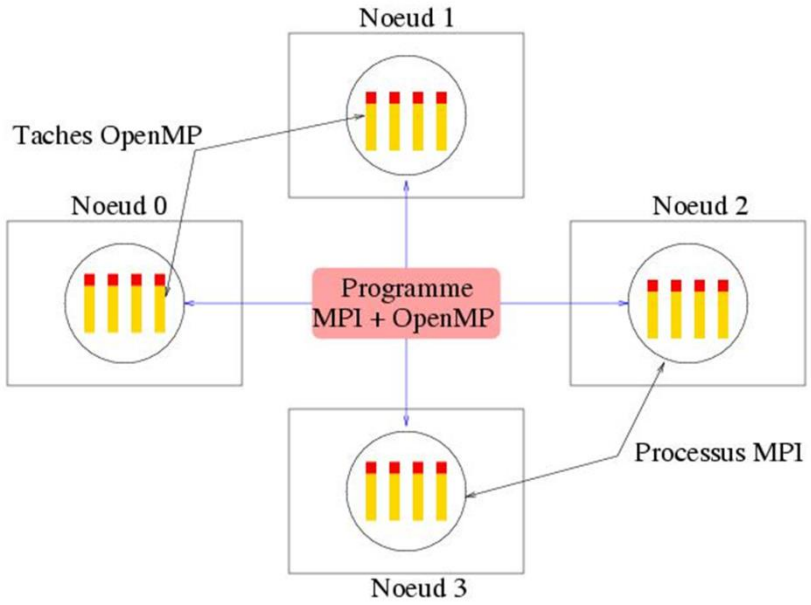
- OpenMP facilite l'écriture d'algorithmes parallèles en mémoire partagée en proposant des mécanismes pour :
  - partager le travail entre les tâches. Il est par exemple possible de répartir les itérations d'une boucle entre les tâches. Lorsque la boucle agit sur des tableaux, cela permet de distribuer simplement le traitement des données entre les processus légers.
  - partager ou privatiser les variables.
  - synchroniser les threads.
- Depuis la version 3.0, OpenMP permet aussi d'exprimer le parallélisme sous la forme d'un ensemble de tâches explicites à réaliser. OpenMP-4.0 permet de décharger une partie du travail sur un accélérateur.

Introduction : OpenMP versus MPI

- Ce sont des modèles de programmation adaptées à deux architectures parallèles différentes :
  - MPI est un modèle de programmation à mémoire distribuée. La communication entre les processus est explicite et sa gestion est à la charge de l'utilisateur.
  - OpenMP est un modèle de programmation à mémoire partagée. Chaque thread a une vision globale de la mémoire.

Introduction : OpenMP versus MPI

- Sur une grappe de machines indépendantes (nœuds) multiprocesseurs à mémoire partagée, la mise en œuvre d'une parallélisation à deux niveaux (MPI et OpenMP) dans un même programme peut être un atout majeur pour les performances parallèles ou l'empreinte mémoire du code.



Introduction : bibliographie

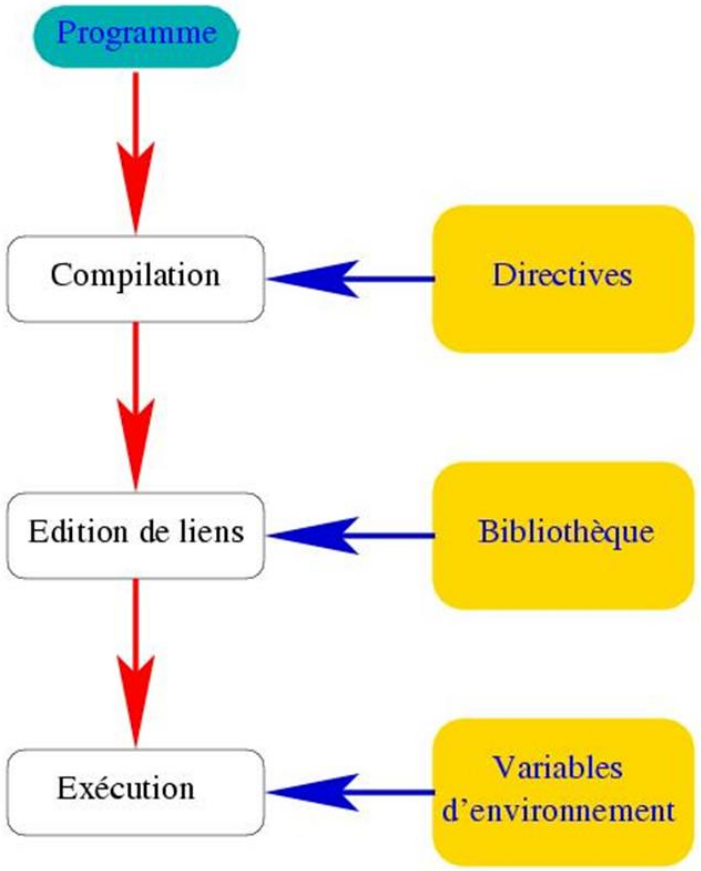
- Premier livre sur OpenMP : R. Chandra & al., Parallel Programming in OpenMP, éd. Morgan Kaufmann Publishers, oct. 2000.
- Livre plus récent sur OpenMP : B. Chapman & al., Using OpenMP, MIT Press, 2008.
- Spécifications du standard OpenMP : http://www.openmp.org/
- Site dédié aux utilisateurs d'OpenMP : http://www.compunity.org/

Principes : interface de programmation

❶ Directives et clauses de compilation : elles servent à définir le partage du travail, la synchronisation et le statut privé ou partagé des données.

❷ Fonctions et sous-programmes : ils font partie d'une bibliothèque chargée à l'édition de liens du programme.

❸ Variables d'environnement : une fois positionnées, leurs valeurs sont prises en compte à l'exécution.



Principes : interface de programmation

- Une directive OpenMP possède la forme générale suivante :

  `sentinelle directive [clause[ clause]...]`

- Les directives OpenMP sont considérées par le compilateur comme des lignes de commentaires à moins de spécifier une option adéquate de compilation pour qu'elles soient interprétées.
- La sentinelle est une chaîne de caractères dont la valeur dépend du langage utilisé.
- Il existe un module Fortran 95 OMP_LIB et un fichier d'inclusion C/C++ omp.h qui définissent le prototype de toutes les fonctions OpenMP. Il est indispensable de les inclure dans toute unité de programme OpenMP utilisant ces fonctions.

Principes : interface de programmation

Pour Fortran, en format libre :

```
!$ use OMP_LIB
...
! $OMP PARALLEL PRIVATE(a,b) &
!$OMP FIRSTPRIVATE(c,d,e)
...
! $OMP END PARALLEL ! C'est un commentaire
```

Pour Fortran, en format fixe :

```
      !$ use OMP_LIB
      ...
C$OMP PARALLEL PRIVATE(a,b)
C$OMP1 FIRSTPRIVATE(c,d,e)
      ...
C$OMP END PARALLEL
```

Pour C et C++ :

```
#include <omp.h>
...
#pragma omp parallel private(a,b) firstprivate(c,d,e)
{ ... }
```

Principes : interface de programmation

Voici les options de compilation permettant d'activer l'interprétation des directives OpenMP par certains compilateurs :

Compilateur GNU : `-fopenmp`

```
gfortran -fopenmp prog.f90 # Compilateur Fortran
```

Compilateur Intel : `-fopenmp` ou `-qopenmp`

```
ifort -fopenmp prog.f90 # Compilateur Fortran
```

Compilateur IBM : `-qsmp=omp`

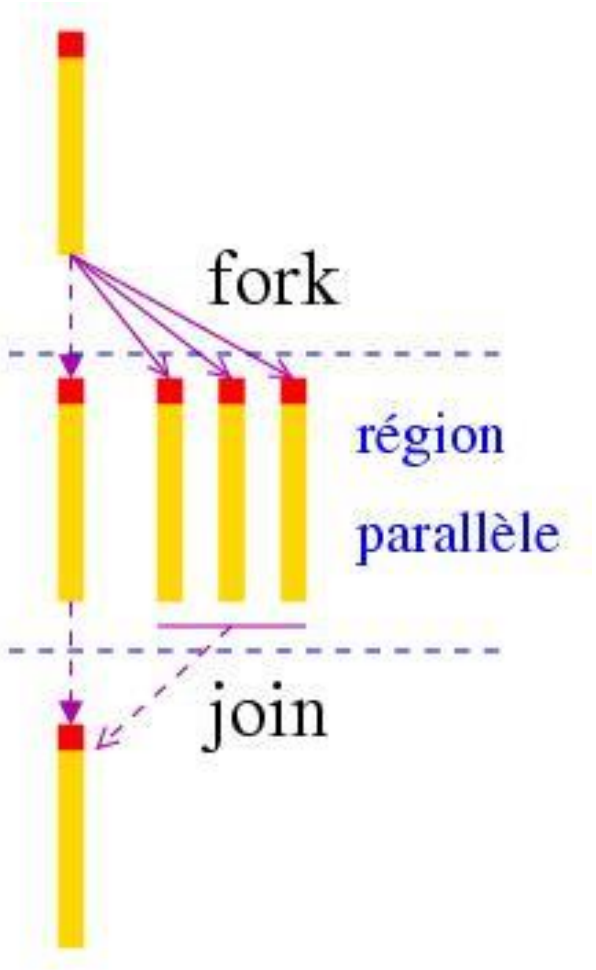
```
xlf_r -qsuffix=f=f90 -qnosave -qsmp=omp prog.f90 # Compilateur Fortran
```

Exemple d'exécution :

```
export OMP_NUM_THREADS=4                    # Nombre de threads souhaité
./a.out                                     # Exécution
```

Principes : construction d'une région parallèle

- Un programme OpenMP est une alternance de régions séquentielles et parallèles (modèle « fork and join »)
- A l'entrée d'une région parallèle, le thread maître (celui de rang 0) crée/active (fork) des processus « fils » (processus légers) et autant de tâches implicites. Ces processus fils exécutent leur tâche implicite puis disparaissent ou s'assoupissent en fin de région parallèle (join).
- En fin de région parallèle, l'exécution redevient séquentielle avec uniquement l'exécution du thread maître.



Premier programme

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

void main() {
  #pragma omp parallel
    {
      printf( "Hello from thread %d\n",
        omp_get_thread_num() ) ;
    }
}
```

Header

Directive

Région parallèle

```
$ gcc –o test –fopenmp test.c

$ export OMP_NUM_THREADS=4
$ ./test
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 0
```

Premier programme

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


void main() {
 #pragma omp parallel
  {
    printf( “Hello from thread %d\n”,
      omp_get_thread_num() ) ;
  }
}
```

- Comment écrire ce code en pthreads

Principes : Construction d'une région parallèle

- Au sein d'une même région parallèle, toutes les tâches concurrentes exécutent le même code.
- Il existe une barrière implicite de synchronisation en fin de région parallèle.
- Il est interdit d'effectuer des branchements (ex. GOTO, CYCLE, etc.) vers l'intérieur ou vers l'extérieur d'une région parallèle ou de toute autre construction OpenMP.

Second programme

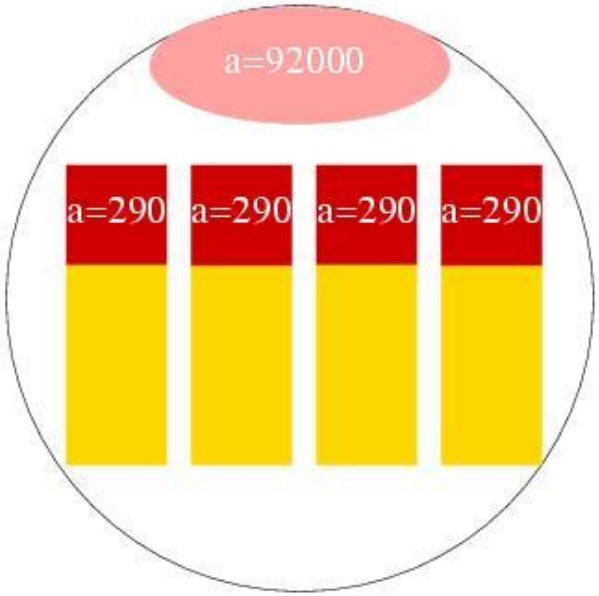
```
#include <stdio.h>
#include <omp.h>

int main() {
  float a;
  int   p;
  a = 92290. ; p = 0;
#pragma omp parallel
  {
#ifdef _OPENMP
    p=omp_in_parallel();
#endif
    printf("a vaut : %f ; p vaut : %d\n",a,p);
  }
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
```

Fonction OpenMP

Flot de données

- Dans une région parallèle, par défaut, le statut des variables est partagé.
- Il est possible, grâce à la clause DEFAULT, de changer le statut par défaut des variables dans une région parallèle.
- Si une variable possède un statut privé (PRIVATE), elle se trouve dans la pile de chaque tâche. Sa valeur est alors indéfinie à l'entrée d'une région parallèle (dans l'exemple ci-contre, la variable a vaut 0 à l'entrée de la région parallèle).



Données privées

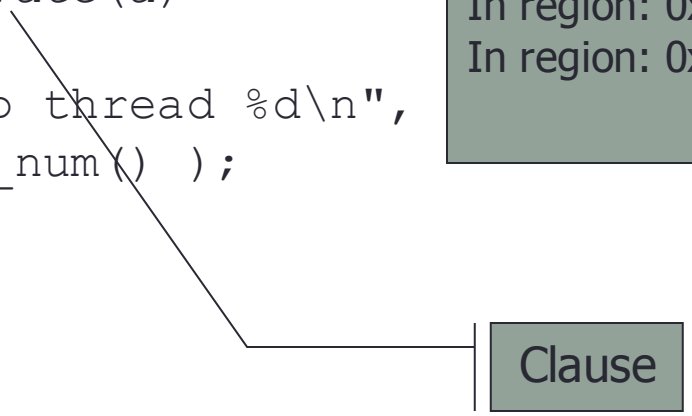
```
#include <stdio.h>
#include <omp.h>

int main() {
  float a;
  a = 92000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

```
$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ ./test
Out region: 0xbf8d9a9c
In region: 0xbf8d9a9c thread 1
In region: 0xbf8d9a9c thread 2
In region: 0xbf8d9a9c thread 3
In region: 0xbf8d9a9c thread 0
```

Données privées

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

int main() {
  float a;
  a = 92000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel private(a)
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

```
$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ ./test
Out region: 0xbf887e2c
In region: 0xbf887dfc thread 0
In region: 0xb67cf2ec thread 3
In region: 0xb6fd02ec thread 2
In region: 0xb77d12ec thread 1
```



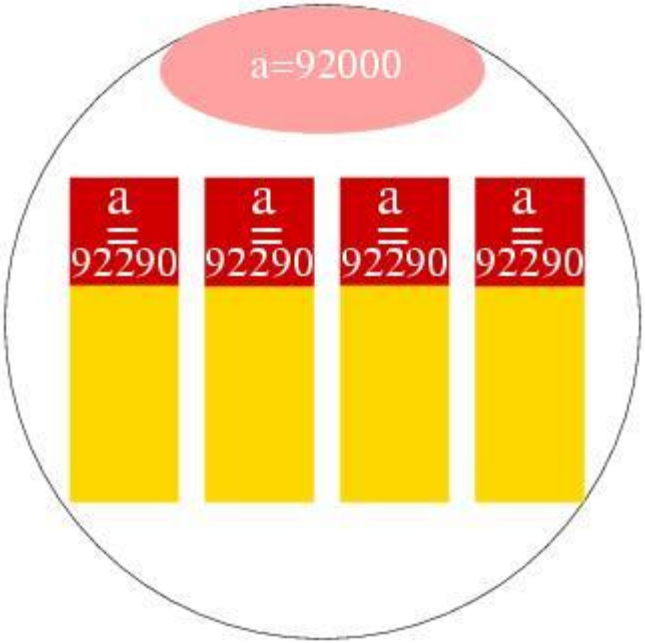
Données privées

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

int main() {
  float a;
  a = 92000.;
  printf( “Out region: %p\n”, &a);
#pragma omp parallel private(a)
  {
    printf(“In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

- Comment écrire ce code en pthreads

Données privées initialisées

- Cependant, grâce à la clause FIRSTPRIVATE, il est possible de forcer l'initialisation de cette variable privée à la dernière valeur qu'elle avait avant l'entrée dans la région parallèle.



Données privées initialisées

```
#include <stdio.h>

int main() {
  float a;
  a = 92000.;
#pragma omp parallel default(none) \
    firstprivate(a)
  {
    a = a + 290.;
    printf("a vaut : %f\n",a);
  }
  printf("Hors region, a vaut : %f\n",
    a);
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
a vaut : 92290.
a vaut : 92290.
a vaut : 92290.
a vaut : 92290.
Hors region, a vaut : 92000.
```

Données privées initialisées

```
#include <stdio.h>

int main() {
  float a;
  a = 92000.;
#pragma omp parallel default(none) \
    firstprivate(a)
  {
    a = a + 290.;
    printf("a vaut : %f\n",a);
  }
  printf("Hors region, a vaut : %f\n",
    a);
  return 0;
}
```

- Comment écrire ce code en pthreads

Étendue d’une région parallèle

- L'étendue d'une construction OpenMP représente le champ d'influence de celle-ci dans le programme.
- L'influence (ou la portée) d'une région parallèle s'étend aussi bien au code contenu lexicalement dans cette région (étendue statique), qu'au code des sous-programmes appelés.
  - L'union des deux représente « l'étendue dynamique ».

Étendue d'une région parallèle

```
#include <omp.h>

void sub(void);

int main() {
#pragma omp parallel
  {
    sub();
  }
  return 0;
}
```

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

void sub(void) {
  int p=0;
#ifdef _OPENMP
  p = omp_in_parallel();
#endif
  printf("Parallele ? : %d\n",
  p);
}
```

```
$ gcc –o prog -fopenmp prog.c sub.c
$ export OMP_NUM_THREADS=4
$ ./prog
Parallele ? : 1
Parallele ? : 1
Parallele ? : 1
Parallele ? : 1
```

Étendue d'une région parallèle

- Dans un sous-programme appelé dans une région parallèle, les variables locales et automatiques sont implicitement privées à chacune des tâches

```
int main() {
#pragma omp parallel
  default(shared)
{
   sub();
}
return 0;
}
```

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

void sub(void) {
int a;

a=92290;
a = a +
  omp_get_thread_num();
printf("a vaut : %d\n", a);
}
```

```
$ ./prog
a vaut : 92293
a vaut : 92291
a vaut : 92292
a vaut : 92290
```

Transmission par argument

- Dans une procédure, toutes les variables transmises par argument par référence, héritent du statut défini dans l'étendue lexicale (statique) de la région.

```
#include <stdio.h>
int main() {
  int a, b;
  a = 92000;
#pragma omp parallel shared(a) \
  private(b)
  {
    sub(a, &b);
    printf("b vaut : %d\n",b);
  }
  return 0;
}
```

```
#include <omp.h>

void sub(int x, int *y)
{
*y = x +
omp_get_thread_num();
}
```

```
$ ./prog
b vaut : 92003
b vaut : 92001
b vaut : 92002
b vaut : 92000
```

Variables statiques

- Une variable est statique si son emplacement en mémoire est défini à la déclaration par le compilateur.

- En Fortran, c'est le cas des variables apparaissant en COMMON ou contenues dans un MODULE ou déclarées SAVE ou initialisées à la déclaration (ex. PARAMETER, DATA, etc.).

- En C, c'est le cas des variables externes ou déclarées STATIC.

Variables statiques

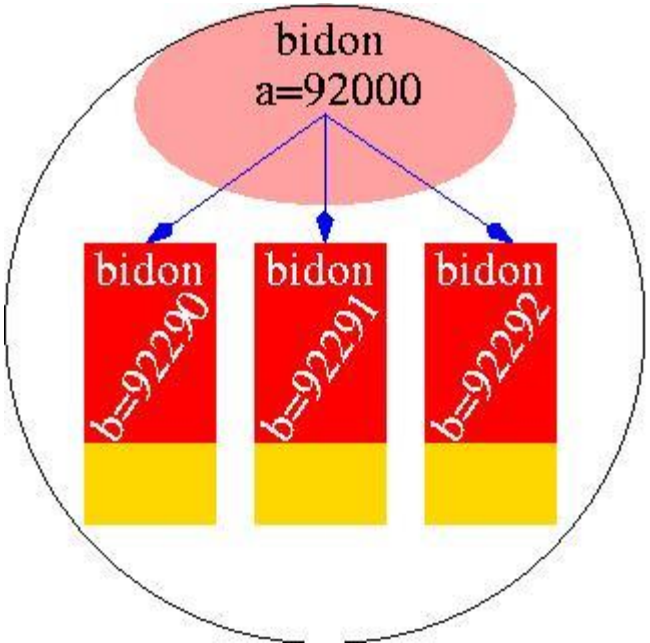
```
#include <omp.h>
float a;

int main() {
  a = 92000;
#pragma omp parallel
  {
    sub();
  }
  return 0;
}
```

```
#include <stdio.h>
float a;

void sub(void) {
  float b;

  b = a + 290.;
  printf(
    "b vaut : %f\n",b);
}
```

```
$ export OMP_NUM_THREADS=2
$ ./prog
B vaut : 92290
B vaut : 92290
```

Variables statiques

- L'utilisation de la directive THREADPRIVATE permet de privatiser une instance statique et faire que celle-ci soit persistante d'une région parallèle à une autre.
- Si, en outre, la clause COPYIN est spécifiée alors la valeur des instances statiques est transmise à toutes les tâches.



Variables statiques

```
#include <stdio.h>
#include <omp.h>
int a;
#pragma omp threadprivate(a)

int main() {
  a = 92000;
  #pragma omp parallel copyin(a)
  {
    a = a + omp_get_thread_num();
    sub();
  }
  printf(
   "Hors region, A vaut: %d\n",a);
  return 0;
}
```

```
#include <stdio.h>

int a;
#pragma omp threadprivate(a)

void sub(void) {
  int b;
  b = a + 290;
  printf("b vaut : %d\n",b);
}
```

```
$ OMP_NUM_THREADS=4 ./prog
B vaut : 92290
B vaut : 92291
B vaut : 92292
B vaut : 92293
Hors region, A vaut : 92000
```

Allocation dynamique

- L'opération d'allocation/désallocation dynamique de mémoire peut être effectuée à l'intérieur d'une région parallèle.

- Si l'opération porte sur une variable privée, celle-ci sera locale à chaque tâche.

- Si l'opération porte sur une variable partagée, il est alors plus prudent que seule une tâche (ex. la tâche maître) se charge de cette opération

Allocation dynamique

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
  int n, debut, fin, rang, nb_taches, i;
  float *a;
  n=1024, nb_taches=4;
  a=(float *) malloc(n*nb_taches*sizeof(float));
  #pragma omp parallel default(none) \
     private(debut,fin,rang,i) shared(a,n) if(n > 512)
  {
    rang=omp_get_thread_num();
    debut=rang*n;
    fin=(rang+1)*n;
    for (i=debut; i<fin; i++)
       a[i] = 92291. + (float) i;
    printf("Rang : %d ; A[%.4d],...,A[%.4d] : %#.0f,...,%#.0f \n",
           rang,debut,fin-1,a[debut],a[fin-1]);
  }
  free(a);
  return 0;
}
```

```
$ OMP_NUM_THREADS=4  ./prog
Rang : 3 ; A[3072],...,A[4095]  : 95363.,...,96386.
Rang : 0 ; A[0000],...,A[1023]  : 92291.,...,93314.
Rang : 1 ; A[1024],...,A[2047]  : 93315.,...,94338.
Rang : 2 ; A[2048],...,A[3071]  : 94339.,...,95362.
```

Compléments

- La construction d'une région parallèle admet deux autres clauses :
  - REDUCTION : pour les opérations de réduction avec synchronisation implicite entre les tâches ;
  - NUM_THREADS : elle permet de spécifier le nombre de tâches souhaité à l'entrée d'une région parallèle de la même manière que le ferait le sous-programme OMP_SET_NUM_THREADS.
    - → Exemple précédent avec l'utilisation de cette clause/fonction
- D'une région parallèle à l'autre, le nombre de tâches concurrentes peut être variable si on le souhaite. Pour cela, il suffit d'utiliser le sous-programme OMP_SET_DYNAMIC ou de positionner la variable d'environnememt OMP_DYNAMIC à true.
- Il est possible d'imbriquer (nesting) des régions parallèles, mais cela n'a d'effet que si ce mode a été activé à l'appel du sous-programme OMP_SET_NESTED ou en positionnant la variable d'environnement OMP_NESTED.

Compléments

```
#include <omp.h>
int main() {
  int rang;
  #pragma omp parallel private(rang) \
     num_threads(3)
  {
    rang=omp_get_thread_num();
    printf(
      "Mon rang dans region 1 : %d \n",rang);
    #pragma omp parallel private(rang) \
      num_threads(2)
    {
      rang=omp_get_thread_num();
      printf(
       "    Mon rang dans region 2 : %d \n",rang);
    }
  }
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c
$ export OMP_DYNAMIC=true
$ export OMP_NESTED=true
$ ./prog

Mon rang dans region 1 : 0
        Mon rang dans region 2 : 1
        Mon rang dans region 2 : 0
Mon rang dans region 1 : 2
        Mon rang dans region 2 : 1
        Mon rang dans region 2 : 0
Mon rang dans region 1 : 1
        Mon rang dans region 2 : 0
        Mon rang dans region 2 : 1
```

Partage du travail

- En principe, la construction d'une région parallèle et l'utilisation de quelques fonctions OpenMP suffisent à eux seuls pour paralléliser une portion de code.
  - Mais il est, dans ce cas, à la charge du programmeur de répartir aussi bien le travail que les données et d'assurer la synchronisation des tâches.
  - Exemple : addition de deux vecteurs en Pthreads
- Heureusement, OpenMP propose deux directives (FOR, SECTIONS) qui permettent aisément de contrôler assez finement la répartition du travail et des données en même temps que la synchronisation au sein d'une région parallèle.
- Par ailleurs, il existe d'autres constructions OpenMP qui permettent l'exclusion de toutes les tâches à l'exception d'une seule pour exécuter une portion de code située dans une région parallèle.
- ATTENTION : le partage du travail concerne des parties indépendantes de code. Ni le compilateur, ni le runtime ne vérifie que ces parties sont bien parallélisables !

Boucle parallèle

- C'est un parallélisme par répartition du domaine d'itérations d'un nid de boucle
  - Concerne un ensemble de boucles parfaitement imbriquées
  - Restrictions sur la structure des boucles (pas de boucle *while* ni de boucles irrégulières)
  - Les indices de boucles sont des variables entières privées.
- Le mode de répartition des itérations peut être spécifié dans la clause SCHEDULE.
  - Le choix de l'ordonnancement par défaut n'est pas imposé
- Le choix du mode de répartition permet de mieux contrôler l'équilibrage de la charge de travail entre les tâches.
- Par défaut, une synchronisation globale est effectuée en fin de construction à moins d'avoir spécifié la clause NOWAIT.

Boucle parallèle

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4  ./a.out
Thread 0 running iteration 0
Thread 0 running iteration 1
Thread 0 running iteration 2
Thread 3 running iteration 9
Thread 2 running iteration 6
Thread 2 running iteration 7
Thread 2 running iteration 8
Thread 1 running iteration 3
Thread 1 running iteration 4
Thread 1 running iteration 5
```

Boucle parallèle

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
          omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
   }
```

- Comment écrire ce code en pthreads

Fusion de boucles

- Dans le cas de boucles parfaitement imbriquées et sans dépendances, il peut être intéressant de les fusionner pour obtenir un espace d'itération plus grand.
- Ainsi, on augmente la granularité de travail de chacun des threads ce qui peut parfois améliorer significativement les performances.
- La clause COLLAPSE(n) permet de fusionner les nids boucles imbriquées qui suivent immédiatement la directive. Le nouvel espace d'itération est alors partagé entre les threads suivant le mode de répartition choisi.

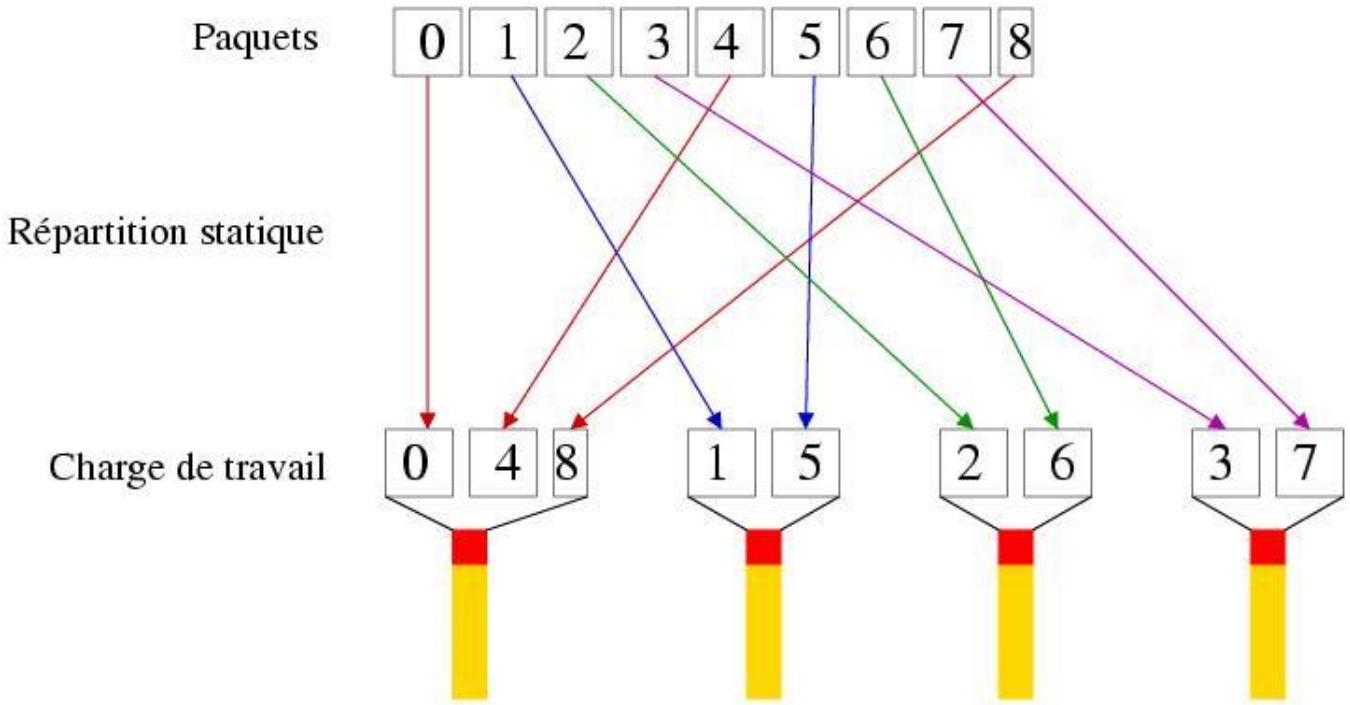
```
#pragma omp for collapse(2)
for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    f(i, j);
```

≡

```
#pragma omp for
for (i = 0; i < 100; i++)
  f(i/10, i%10);
```

Ordonnancement statique

- L'ordonnancement STATIC consiste à diviser les itérations en paquets d'une taille donnée appelé *chunk* (sauf peut-être pour le dernier).
  - Par défaut, le taille du *chunk* est maximale
  - Il est ensuite attribué, d'une façon cyclique à chacune des tâches, un ensemble de paquets suivant l'ordre des tâches jusqu'à concurrence du nombre total de paquets.



Ordonnancement statique

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,1)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4  ./a.out
Thread 0 running iteration 0
Thread 0 running iteration 4
Thread 0 running iteration 8
Thread 3 running iteration 3
Thread 3 running iteration 7
Thread 2 running iteration 2
Thread 2 running iteration 6
Thread 1 running iteration 1
Thread 1 running iteration 5
Thread 1 running iteration 9
```

Ordonnancement statique

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,2)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4  ./a.out
Thread 0 running iteration 0
Thread 0 running iteration 1
Thread 0 running iteration 8
Thread 0 running iteration 9
Thread 3 running iteration 6
Thread 3 running iteration 7
Thread 1 running iteration 2
Thread 1 running iteration 3
Thread 2 running iteration 4
Thread 2 running iteration 5
```

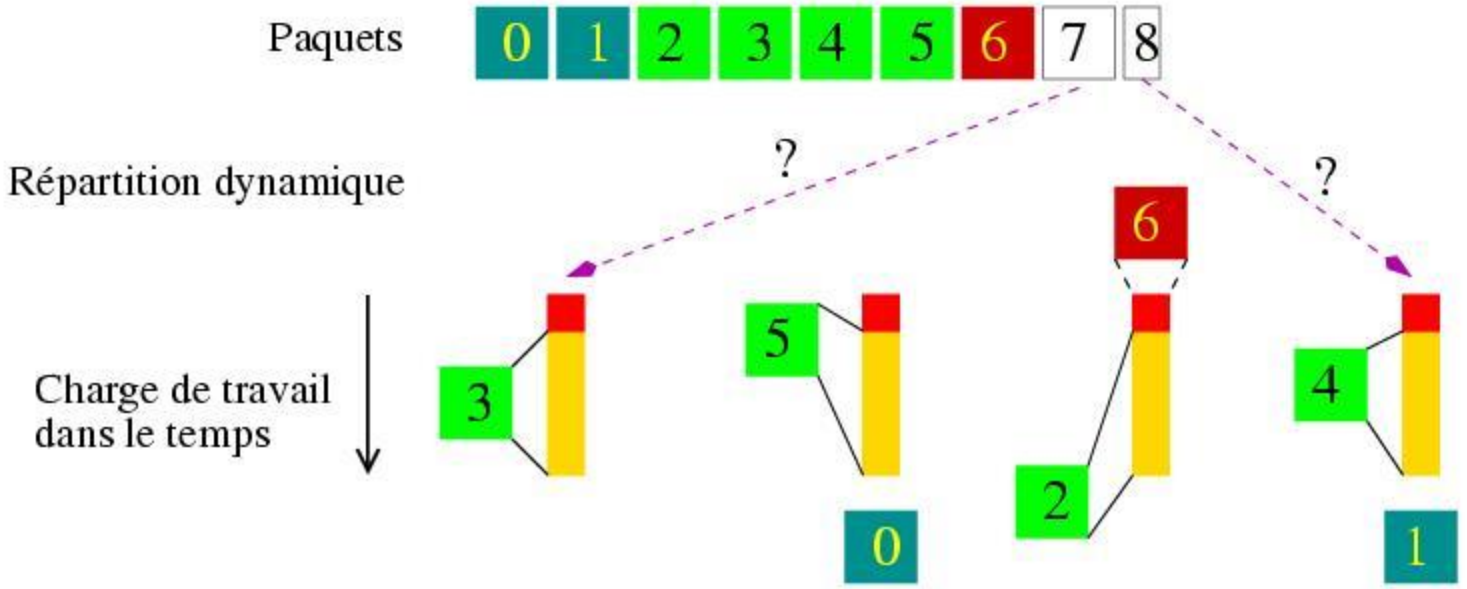
Clause schedule

- Nous aurions pu différer à l'exécution le choix du mode de l'ordonnancement des itérations à l'aide de la variable d'environnement OMP_SCHEDULE.
  - Fonction également disponible `omp_set_schedule()`
- Le choix de l'ordonnancement des itérations d'une boucle peut être un atout majeur pour l'équilibrage de la charge de travail sur une machine dont les processeurs ne sont pas dédiés.
  - La taille du *chunk* joue également un rôle important dans les performances

Clause schedule

- DYNAMIC : les itérations sont divisées en paquets de taille donnée. Sitôt qu'une tâche épuise ses itérations, un autre paquet lui est attribué.

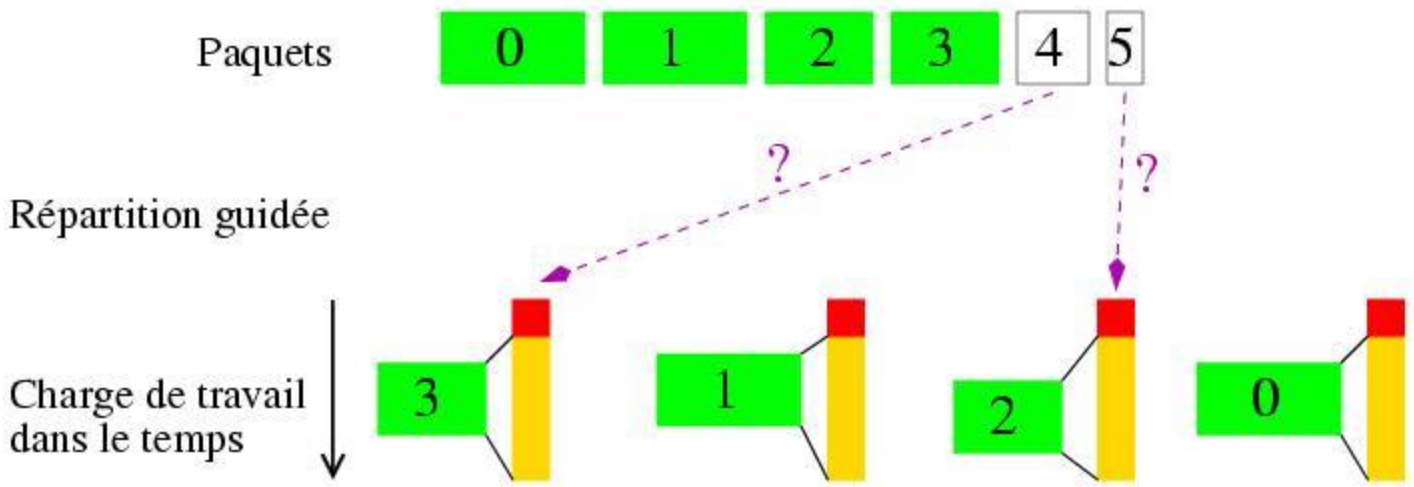
```
$ export OMP_SCHEDULE="DYNAMIC,480"
$ export OMP_NUM_THREADS=4 ; ./prog
```



Clause schedule

- GUIDED : les itérations sont divisées en paquets dont la taille décroît exponentiellement. Tous les paquets ont une taille supérieure ou égale à une valeur donnée à l'exception du dernier dont la taille peut être inférieure. Sitôt qu'une tâche finit ses itérations, un autre paquet d'itérations lui est attribué.

```
> export OMP_SCHEDULE="GUIDED,256"
> export OMP_NUM_THREADS=4 ; ./prog
```



Exécution ordonnée

- Il est parfois utile d'exécuter une partie d'une boucle d'une façon ordonnée.
  - Pour le débuggage
  - Pour les I/O
- L'ordre des itérations sera alors identique à celui correspondant à une exécution séquentielle.
  - Seul le bout de code en question est concerné

Exécution ordonnée

```
#include <stdio.h>
#include <omp.h>
#define N 9
int main() {
    int i, rang;

    #pragma omp parallel default(none) private(rang,i)
    {
        rang=omp_get_thread_num();
        #pragma omp for schedule(runtime) ordered nowait
        for (i=0; i<N; i++) {
            #pragma omp ordered
            {
                printf("Rang : %d ; iteration : %d\n",rang,i);
            }
        }
    }
    return 0;
}
```

Réduction

- Une réduction est une opération associative appliquée à une variable partagée.
- L'opération peut être :
  - arithmétique : +, --, * ;
  - logique : .AND., .OR., .EQV., .NEQV. ;
  - une fonction intrinsèque : MAX, MIN, IAND, IOR, IEOR.
- Chaque tâche calcule un résultat partiel indépendamment des autres. Elles se synchronisent ensuite pour mettre à jour le résultat final.

Réduction

```
#include <stdio.h>
#define N 5
int main()
{
  int i, s=0, p=1, r=1;

  #pragma omp parallel
  {
    #pragma omp for reduction(+:s) reduction(*:p,r)
    for (i=0; i<N; i++) {
      s = s + 1;
      p = p * 2;
      r = r * 3;
    }
  }
  printf("s = %d ; p = %d ; r = %d\n",s,p,r);
  return 0;
}
```

```
$ gcc –o prog –fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
s = 5 ; p = 32 ; r = 243
```

Flot de données

- La construction FOR accepte également des clauses concernant le flot des données
  - PRIVATE : pour attribuer à une variable un statut privé ;
  - FIRSTPRIVATE : privatise une variable partagée dans l'étendue de la construction DO et lui assigne la dernière valeur affectée avant l'entrée dans cette région ;
  - LASTPRIVATE : privatise une variable partagée dans l'étendue de la construction DO et permet de conserver, à la sortie de cette construction, la valeur calculée par la tâche exécutant la dernière itération d'une boucle.

Région combinée

- La directive PARALLEL FOR est une fusion des directives PARALLEL et FOR munie de l'union de leurs clauses respectives.
- La fin du bloc inclut une barrière globale de synchronisation et ne peut admettre la clause NOWAIT.

Sections parallèles

- Une section est une portion de code exécutée par une et une seule tâche.
- Plusieurs portions de code peuvent être définies par l'utilisateur à l'aide de la directive SECTION au sein d'une construction SECTIONS.
- Le but est de pouvoir répartir l'exécution de plusieurs portions de code indépendantes sur les différentes tâches.
- La clause NOWAIT est admise en fin de construction pour lever la barrière de synchronisation implicite.

Sections parallèles

```
int main() {
    int i, rang;
    float pas_x, pas_y;
    float coord_x[M],coord_y[N];
    float a[M][N], b[M][N];

    #pragma omp parallel private(rang) num_threads(3)
    {
        rang=omp_get_thread_num();
        #pragma omp sections nowait
        {
            #pragma omp section
            {
                lecture_champ_initial_x(a);
                printf("Tâche numéro %d : init. champ en X\n",rang);
            }
            #pragma omp section
            {
                lecture_champ_initial_y(b);
                printf("Tâche numéro %d : init. champ en Y\n",rang);
            }
        }
    }
    return 0;
}
```

Sections parallèles

- Toutes les directives SECTION doivent apparaître dans l'étendue lexicale de la construction SECTIONS.
- Les clauses admises dans la directive SECTIONS sont celles que nous connaissons déjà :
  - PRIVATE ;
  - FIRSTPRIVATE ;
  - LASTPRIVATE ;
  - REDUCTION.
- La directive PARALLEL SECTIONS est une fusion des directives PARALLEL et SECTIONS munie de l'union de leurs clauses respectives.

EXÉCUTION EXCLUSIVE

Exécution exclusive

- Il arrive que l'on souhaite exclure toutes les tâches à l'exception d'une seule pour exécuter certaines portions de code incluses dans une région parallèle.
- Pour se faire, OpenMP offre deux directives SINGLE et MASTER.
- Bien que le but recherché soit le même, le comportement induit par ces deux constructions reste assez différent.

Construction *master*

- La construction MASTER permet de faire exécuter une portion de code par la tâche maître seule.
- Cette construction n'admet aucune clause.
- Il n'existe aucune barrière de synchronisation ni en début (MASTER) ni en fin de construction (END MASTER).

Construction *master*

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 92290.;

        #pragma omp master
        {
            a = -92290.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
}
```

Construction *master*

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 92290.;

        #pragma omp master
        {
            a = -92290.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
}
```

- Comment écrire ce code en pthreads

Construction *single*

- La construction SINGLE permet de faire exécuter une portion de code par une et une seule tâche sans pouvoir indiquer laquelle.
  - Généralisation de MASTER
- En général, c'est la tâche qui arrive la première sur la construction SINGLE mais cela n'est pas spécifié dans la norme.
- Toutes les tâches n'exécutant pas la région SINGLE attendent, en fin de construction, la terminaison de celle qui en a la charge, à moins d'avoir spécifié la clause NOWAIT.

Construction *single*

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 92290.;

        #pragma omp single
        {
            a = -92290.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
}
```

Construction *single*

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 92290.;

        #pragma omp single
        {
            a = -92290.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
   }
```

- Comment écrire ce code en pthreads

Construction *single*

- Une clause supplémentaire admise est la clause COPYPRIVATE.
- Elle permet à la tâche chargée d'exécuter la région SINGLE, de diffuser aux autres tâches la valeur d'une liste de variables privées avant de sortir de cette région.
- Les autres clauses admises par la directive SINGLE sont PRIVATE et FIRSTPRIVATE.

Construction *single*

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 92290.;

        #pragma omp single copyprivate(a)
        {
            a = -92290.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
}
```

SYNCHRONISATIONS

Synchronisations

- La synchronisation devient nécessaire dans les situations suivantes :
  1. pour s'assurer que toutes les tâches concurrentes aient atteint un même niveau d'instruction dans le programme (barrière globale)
  2. pour ordonner l'exécution de toutes les tâches concurrentes quand celles-ci doivent exécuter une même portion de code affectant une ou plusieurs variables partagées dont la cohérence (en lecture ou en écriture) en mémoire doit être garantie (exclusion mutuelle).
  3. pour synchroniser au moins deux tâches concurrentes parmi l'ensemble (mécanisme de verrous).

Synchronisations

- Il est possible d'imposer explicitement une barrière globale de synchronisation grâce à la directive BARRIER.

- Le mécanisme d'exclusion mutuelle (une tâche à la fois) se trouve, par exemple, dans les opérations de réduction (clause REDUCTION) ou dans l'exécution ordonnée d'une boucle (directive DO ORDERED). Dans le même but, ce mécanisme est aussi mis en place dans les directives ATOMIC et CRITICAL.

- Des synchronisations plus fines peuvent être réalisées soit par la mise en place des mécanismes de verrous (cela nécessite l'appel à des sous-programmes de la bibliothèque OpenMP), soit par l'utilisation de la directive FLUSH.

Barrière

- Principe implicite
  - Chaque construction OpenMP (région parallèle, partage de travail, …) implique une barrière implicite à la fin
  - Cette barrière concerne tous les threads de la même équipe (*team*)
- La directive BARRIER synchronise l'ensemble des tâches concurrentes dans une région parallèle.

```
#pragma omp barrier
```

- Chacune des tâches attend que toutes les autres soient arrivées à ce point de synchronisation pour poursuivre, ensemble, l'exécution du programme.

Mise à jour atomique

- La directive ATOMIC assure qu'une variable partagée est lue et modifiée en mémoire par une seule tâche à la fois.

```
#pragma omp atomic
```

- Son effet est local à l'instruction qui suit immédiatement la directive.

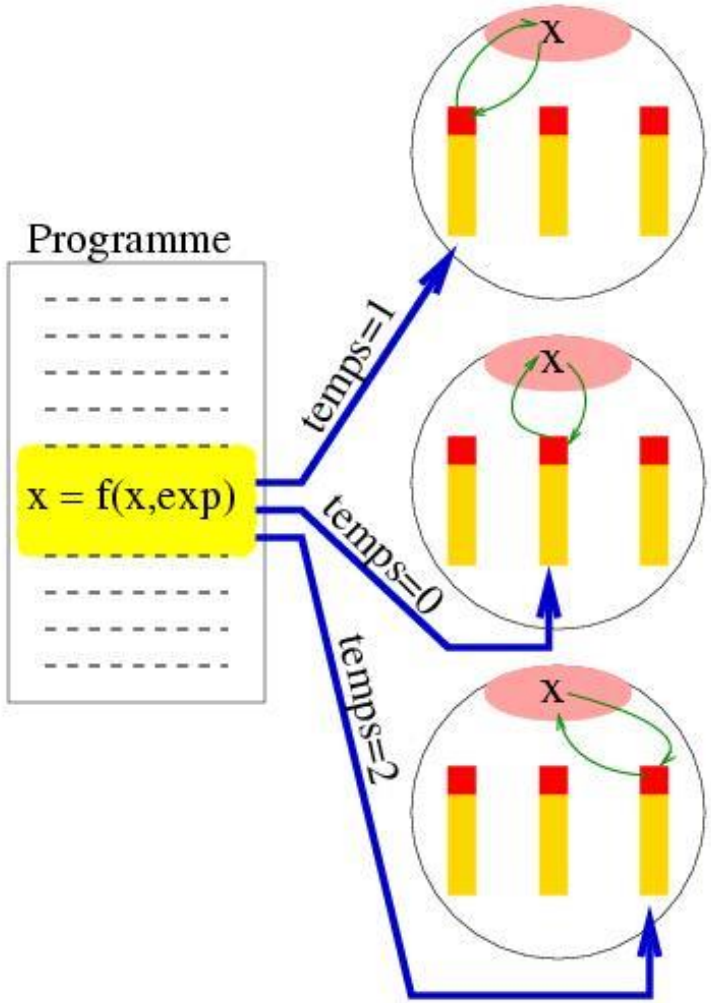
Mise à jour atomique

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

int main() {
    int compteur, rang;

    compteur = 92290;
    #pragma omp parallel private(rang)
    {
        rang=omp_get_thread_num();

        #pragma omp atomic
          compteur++;

        printf("Rang : %d ; compteur vaut : %d\n",rang,compteur);
    }
    printf("Au total, compteur vaut : %d\n",compteur);
    return 0;
}
```

Mise à jour atomique

- L'instruction en question doit avoir l'une des formes suivantes :
  - x=x~(op)~exp ;
  - x=exp~(op)~x ;
  - x=f(x,exp) ;
  - x=f(exp,x) ;
- `(op)` représente l'une des opérations suivantes : +, --, *, /, .AND., .OR., .EQV., .NEQV..
- `f` représente une des fonctions intrinsèques suivantes : MAX, MIN, IAND, IOR, IEOR.
- `exp` est une expression arithmétique quelconque indépendante de x.

Mise à jour atomique



Régions critiques

- Une région critique peut être vue comme une généralisation de la directive ATOMIC bien que les mécanismes sous-jacents soient distincts.
  - Les tâches exécutent cette région dans un ordre non-déterministe mais une à la fois.
  - Son étendue est dynamique.

- Une région critique est définie grâce à la directive CRITICAL et s'applique à une portion de code

```
#pragma omp critical
```

- Possibilité de créer des région critiques *nommées*
- Pour des raisons de performances, il est déconseillé d'émuler une instruction atomique par une région critique.

Régions critiques

```
#include <stdio.h>

int main()
{
    int s, p;

    s = 0, p = 1;
    #pragma omp parallel
    {
        #pragma omp critical
        {
            s++;
            p*=2;
        }
    }
        printf("Somme et produit finaux : %d, %d\n",s,p);
    return 0;
}
```

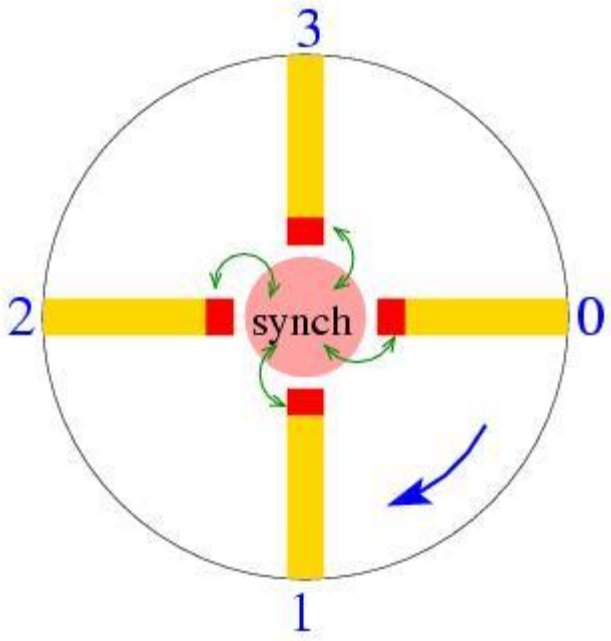
Régions critiques

```
#include <stdio.h>

int main()
{
    int s, p;

    s = 0, p = 1;
    #pragma omp parallel
    {
        #pragma omp critical
        {
            s++;
            p*=2;
        }
    }
        printf("Somme et produit finaux : %d, %d\n",s,p);
    return 0;
}
```

- Comment écrire ce code en pthreads

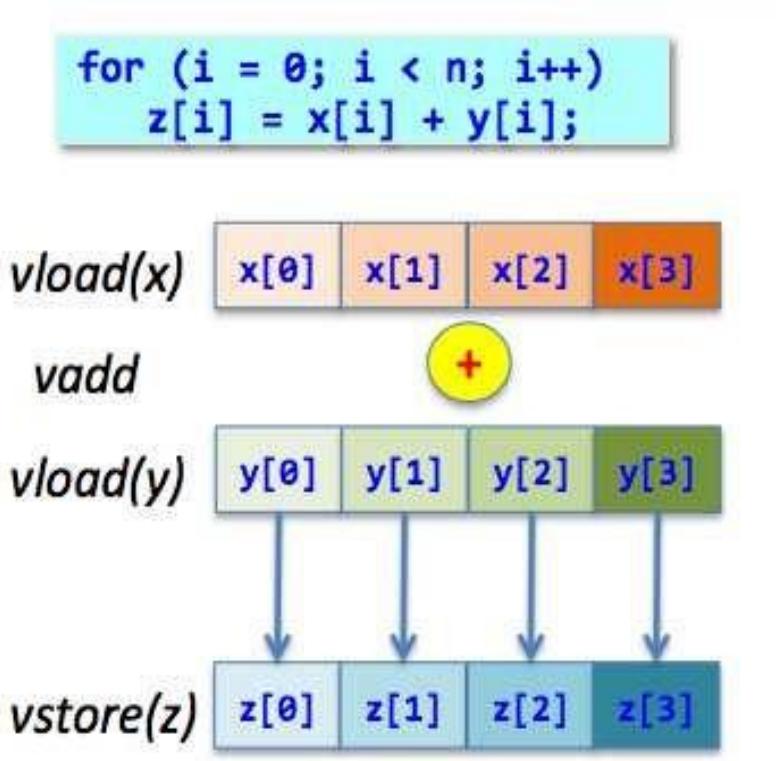
Directive FLUSH

- Elle est utile dans une région parallèle pour rafraîchir la valeur d'une variable partagée en mémoire globale.
- Elle est d'autant plus utile que la mémoire d'une machine est hiérarchisée.
- Elle peut servir à mettre en place un mécanisme de point de synchronisation entre les tâches.



Vectorisation SIMD : introduction

- SIMD = Single Instruction Multiple Data
- Une seule instruction/opération agit en parallèle sur plusieurs éléments.
- Avant OpenMP 4.0, les développeurs devaient soit se reposer sur le savoir-faire du compilateur, soit avoir recours à des extensions propriétaires (directives ou fonctions intrinsèques).
- OpenMP 4.0 offre la possibilité de gérer la vectorisation SIMD de façon portable et performante en utilisant les instructions vectorielles disponibles sur l'architecture cible.



Vectorisation SIMD : vectorisation SIMD d'une boucle

- La directive SIMD permet de découper la boucle qui la suit immédiatement en morceaux dont la taille correspond à celle des registres vectoriels disponibles sur l'architecture cible.
- La directive SIMD n'entraine pas la parallélisation de la boucle.
- La directive SIMD peut ainsi s'utiliser aussi bien à l'intérieur qu'à l'extérieur d'une région parallèle.

```
program boucle_simd
implicit none
integer(kind=8) :: i
integer(kind=8), parameter :: n=500000
real(kind=8), dimension(n) :: A, B
real(kind=8)               :: somme
...
somme=0
!$OMP SIMD REDUCTION(+:somme)
do i=1,n
   somme=somme+A(i)*B(i)
enddo
...
end program boucle_simd
```

Vectorisation SIMD : parallélisation et vectorisation

- La construction DO SIMD est une fusion des directives DO et SIMD munie de l'union de leurs clauses respectives.
- Cette construction permet de partager le travail et de vectoriser le traitement des itérations de la boucle.
- Les paquets d'itérations sont distribués aux threads en fonction du mode de répartition choisi. Chacun vectorise le traitement de son paquet en le subdivisant en bloc d'itérations de la taille des registres vectoriels, blocs qui seront traités l'un après l'autre avec des instructions vectorielles.
- La directive PARALLEL DO SIMD permet en plus de créer la région parallèle.

```
program boucle_simd
implicit none
integer(kind=8) :: i
integer(kind=8), parameter :: n=500000
real(kind=8), dimension(n) :: A, B
real(kind=8)               :: somme
...
somme=0
!$OMP PARALLEL DO SIMD REDUCTION(+:somme)
do i=1,n
   somme=somme+A(i)*B(i)
enddo
...
end program boucle_simd
```

Vectorisation SIMD : vectorisation SIMD de fonctions scalaires

- Le but est de créer automatiquement une version vectorielle de fonctions scalaires. Les fonctions ainsi générées pourront être appelées à l'intérieur de boucles vectorisées, sans casser la vectorisation.
- La version vectorielle de la fonction permettra de traiter les itérations par bloc et non plus l'une aprés l'autre...
- La directive DECLARE SIMD permet de générer une version vectorielle en plus de la version scalaire de la fonction dans laquelle elle est déclarée.

```
program boucle_fonction_simd
implicit none
integer, parameter :: n=1000
integer :: i
real, dimension(n) :: A, B
real :: dist_max
...
dist_max=0
 !$OMP PARALLEL DO SIMD REDUCTION(max:dist_max)
do i=1,n
   dist_max=max(dist_max,dist(A(i),B(i)))
enddo
 !$OMP END PARALLEL DO SIMD

print *,"Distance maximum = ",dist_max

contains

real function dist(x,y)
  !$OMP DECLARE SIMD (dist)
  real, intent(in) :: x, y
  dist=sqrt(x*x+y*y)
end function dist

end program boucle_fonction_simd
```

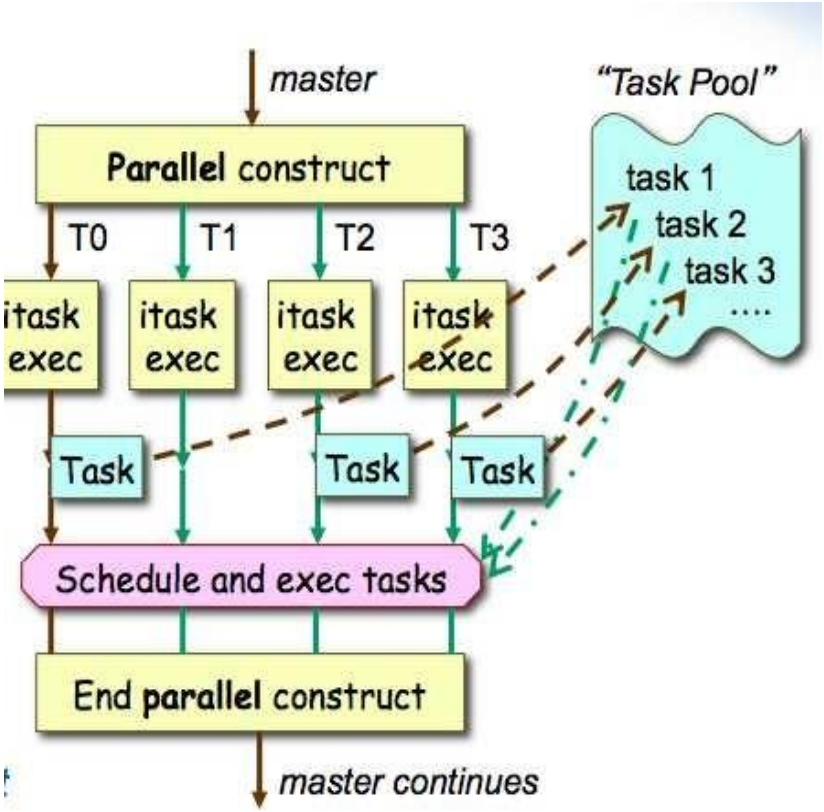
Les tâches OpenMP : introduction

- Le modèle « fork and join » associé aux constructions de partage du travail est limitatif.
- En particulier, il n'est pas adapté aux problématiques dynamiques (boucles while, recherche en parallèle dans un arbre, etc.) ou aux algorithmes récursifs.
- Un nouveau modèle basé sur la notion de tâches a été introduit avec la version OpenMP 3.0. Il est complémentaire de celui uniquement axé sur les threads.
- Il permet l'expression du parallélisme pour les algorithmes récursifs ou à base de pointeurs, couramment utilisés en C/C++.
- La version OpenMP 4.0 permet de gérer des constructions de génération et de synchronisation de tâches explicites (avec ou sans dépendances).

Les tâches OpenMP : les bases du concept

- Une tâche OpenMP est constituée d'une instance de code exécutable et de ses données associées. Elle est exécutée par un thread.
- Deux types de tâches existent :
  - Les tâches implicites générées par la directive PARALLEL
  - Les tâches explicites générées par la directive TASK
- Plusieurs types de synchronisation sont disponibles :
  - Pour une tâche donnée, la directive TASKWAIT permet d'attendre la terminaison de tous ses fils (de première génération).
  - La directive TASKGROUP/END TASKGROUP permet d'attendre la terminaison de tous les descendants d'un groupe de tâches.
  - Des barrières implicites ou explicites permettent d'attendre la terminaison de toutes les tâches explicites déjà créées.
- Les variables (et leur statut associé) sont relatives à une tâches, sauf pour la directive THREADPRIVATE qui est, elle, associée à la notion de thread.

Les tâches OpenMP : le modèle d'exécution des tâches



- *implicit tasks cannot be deferred*
- *explicit tasks could be deferred*

Les tâches OpenMP : le modèle d'exécution des tâches

- L'exécution commence avec le thread master seul.
- A la rencontre d'une région parallèle (PARALLEL) :
  - Création d'une équipe de threads.
  - Création des tâches implicites, une par thread, chaque thread exécutant sa tâche implicite.
- A la rencontre d'une construction de partage du travail :
  - Distribution du travail aux threads (ou aux tâches implicites)
- A la rencontre d'une construction TASK :
  - Création de tâches explicites.
  - L'exécution de ces tâches explicites peut ne pas être immédiate.
- Exécution des tâches explicites :
  - A des points du code appelés task scheduling point (TASK, TASKWAIT, BARRIER), les threads disponibles commencent l'exécution des tâches en attente.
  - Un thread peut passer de l'exécution d'une tâche à une autre.
- A la fin de la région parallèle :
  - Toutes les tâches terminent leur exécution.
  - Seul le thread master continue l'exécution de la partie séquentielle.

Les tâches OpenMP : quelques exemples

```
program task_print
implicit none

print *,"Un "
print *,"grand "
print *,"homme "

end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
```

```
program task_print
implicit none

!$OMP PARALLEL
print *,"Un "
print *,"grand "
print *,"homme "
!$OMP END PARALLEL

end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
Un
homme
grand
homme
```

Les tâches OpenMP : quelques exemples

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
print *,"grand "
print *,"homme "
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
```

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
grand
homme
```

```
Un
homme
grand
```

Les tâches OpenMP : quelques exemples

- Si on rajoute un print juste avant la fin de la région SINGLE, ça ne marche pas !
- En effet, les tâches explicites ne sont exécutables qu'aux task scheduling point du code (TASK, TASKWAIT, BARRIER)...

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
a marche sur la lune
homme
grand
```

```
Un
a marche sur la lune
grand
homme
```

Les tâches OpenMP : quelques exemples

- La solution consiste à introduire un task scheduling point avec la directive TASKWAIT pour exécuter les tâches explicites, puis attendre que ces dernières aient terminé avant de continuer.
- Si on veut imposer un ordre entre « grand » et « homme », il faut utiliser la clause DEPEND introduite dans OpenMP 4.0.

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
!$OMP TASKWAIT
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
homme
grand
a marche sur la lune
```

```
Un
grand
homme
a marche sur la lune
```

Les tâches OpenMP : dépendance entre tâches

- La clause DEPEND(type dependance:list) permet de gérer des dépendances entre des tâches explicites ayant le même père (i.e. générées par la même tâche).
- Une tâche T1 qui dépend de la tâche T2 ne pourra commencer à s'exécuter que lorsque l'exécution de T2 sera terminée.
- Il existe trois types de dépendance :
  - IN : la tâche générée sera une tâche dépendante de toutes les tâches précédemment générées par le même père, qui référencent au moins un élément en commun dans la liste de dépendance de type OUT ou INOUT.
  - INOUT et OUT : la tâche générée sera une tâche dépendante de toutes les tâches précédemment générées par le même père, qui référencent au moins un élément en commun dans la liste de dépendance de type IN, OUT ou INOUT.
- La liste de variables de la directive DEPEND correspond à une adresse mémoire et peut être un élément d'un tableau ou une section de tableau.

Les tâches OpenMP : quelques exemples

- Introduisons une dépendance entre les tâches explicites pour que la tâche T1 : print *,”grand ” s’exécute avant la tâche T2 : print *,”homme ”.
- On peut par exemple utiliser la clause DEPEND(OUT:T1) pour la tâche T1 et DEPEND(IN:T1) pour la tâche T2.

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK DEPEND(OUT:T1)
print *,"grand "
!$OMP END TASK
!$OMP TASK DEPEND(IN:T1)
print *,"homme "
!$OMP END TASK
!$OMP TASKWAIT
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
a marche sur la lune
```

Les tâches OpenMP : statut des variables

- Le statut par défaut des variables est :
  - SHARED pour les tâches implicites
  - Pour les tâches explicites :
    - Si la variable est SHARED dans la tâche père, alors elle hérite de son statut SHARED.
    - Dans les autres cas, le statut par défaut est FIRSTPRIVATE.
- Lors de la création de la tâche, on peut utiliser les clauses SHARED(list), PRIVATE(list), FIRSTPRIVATE(list) ou DEFAULT(PRIVATE|FIRSTPRIVATE|SHARED|NONE) (en C/C++ uniquement DEFAULT(PRIVATE|NONE)) pour spécifier explicitement le statut des variables qui apparaissent lexicalement dans la tâche.

Les tâches OpenMP : clauses FINAL et MERGEABLE

- Dans le cas d'algorithmes récursifs de type « Divide and Conquer », le volume du travail de chaque tâche (i.e. la granularité) diminue au fil de l'exécution. C'est la principale raison pour laquelle le code précédent n'est pas performant.
- Les clauses FINAL et MERGEABLE sont alors très utiles : elles permettent au compilateur de pouvoir fusionner les nouvelles tâches créées.
- Malheureusement, ces fonctionnalités ne sont que très rarement implémentées de façon efficace.

Affinités : affinité des threads

- Par défaut, le système d'exploitation ou la bibliothèque de threads choisit le cœur d'exécution d'un thread.
- Celui-ci peut changer en cours d'exécution, au prix d'une forte pénalité.
- Pour pallier ce problème, il est possible d'associer explicitement un thread à un cœur pendant toute la durée de l'exécution : c'est ce que l'on appelle le binding.
- Avec les compilateurs GNU, l'association thread/cœur d'exécution peut se faire avec la variable d'environnement GOMP CPU AFFINITY.
- Avec les compilateurs Intel, l'association thread/cœur d'exécution peut se faire avec la variable d'environnement KMP AFFINITY (cf. Intel Thread Affinity Interface).

Affinités : affinité des threads

- OpenMP4.0 introduit la notion de places qui définissent des ensembles de cœurs logiques ou physiques qui seront associés à l'exécution d'un thread.
- Les places peuvent être définies explicitement par l'intermédiaire d'une liste, ou directement avec les mots clés suivants :
  - threads : chaque place correspond à un cœur logique de la machine,
  - cores : chaque place correspond à un cœur physique de la machine,
  - sockets : chaque place correspond à un socket de la machine.
- Exemples pour une architecture bi-sockets quadri-cœurs avec hyperthreading :
  - OMP PLACES=threads : 16 places correspondant a un cœur logique
  - OMP PLACES="threads(4)" : 4 places correspondant a un cœur logique
  - OMP PLACES="{0,8,1,9},{6,14,7,15}" : 2 places, la première sur le premier socket, la seconde sur le deuxième.

Affinités : affinité des threads

- La clause PROC BIND de la construction PARALLEL ou la variable d'environnement
- OMP PROC BIND permettent de choisir l'affinité parmi les choix suivants :
  - SPREAD répartition équitable des threads sur les différentes places définies
  - CLOSE regroupement des threads au plus près du master thread
  - MASTER les threads s'exécutent sur la même place que celle du master

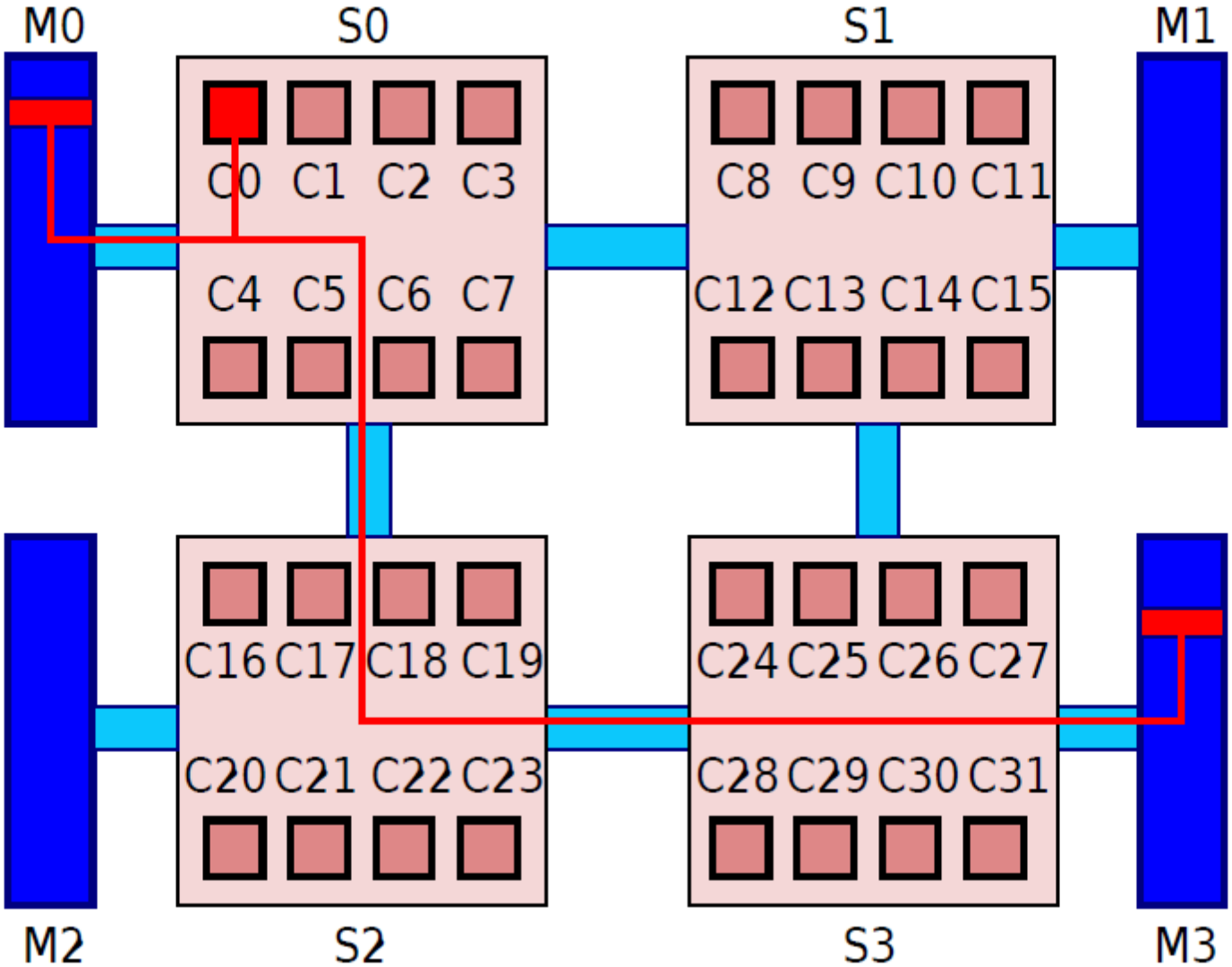
```
export OMP_PLACES="{0,8,1,9},{2,10,3,11},{4,12,5,13},{6,14,7,15}"
Soit 4 places p0={0,8,1,9}, p1={2,10,3,11}, p2={4,12,5,13} et p3={6,14,7,15}
 ! $OMP PARALLEL PROC_BIND(SPREAD) NUM_THREADS(2)
    ! $OMP PARALLEL PROC_BIND(CLOSE) NUM_THREADS(4)
....
Dans la premiere region parallele
Th0 s'executera sur p0 avec une partition de place =p0p1
Th1 s'executera sur p2 avec une partition de place =p2p3
Dans la seconde region parallele
Th00 et Th01 s'executeront sur p0
Th02 et Th03 s'executeront sur p1
Th10 et Th11 s'executeront sur p2
Th12 et Th13 s'executeront sur p3
```

Affinités : affinité mémoire

- Les nœuds multi-socket modernes sont fortement NUMA (Non Uniform Memory Access), le temps d'accès à une donnée est variable suivant l'emplacement du banc mémoire où elle est stockée.
- La localité du stockage en mémoire des variables partagées (sur la mémoire locale du socket qui exécute le thread ou sur la mémoire distante d'un autre socket) va fortement influer sur les performances du code.
- Le système d'exploitation essaie d'optimiser ce processus d'allocation mémoire en privilégiant, lorsque cela est possible, l'allocation dans la mémoire locale du socket qui est en charge de l'exécution du thread. C'est ce que l'on appelle l'affinité mémoire.

Affinités : affinité mémoire

Architecture simplifiée d'une machine fortement NUMA (quadri-sockets, octo-cœurs).



Affinités : affinité mémoire

- Pour les tableaux, l'allocation réelle de la mémoire se fait à l'exécution, page par page, lors du premier accès à un élément de ce tableau.
- Suivant les caractéristiques des codes (memory bound, CPU bound, accès mémoire aléatoires, accès mémoire suivant une dimension privilégiée, etc.), il vaut mieux regrouper tous les threads au sein du même socket (répartition de type compact) ou au contraire les répartir sur les différents sockets disponibles (répartition de type scatter).
- En général, on essaiera de regrouper sur un même socket des threads qui travaillent sur les mêmes données partagées.

Affinités : stratégie

- Pour optimiser l'affinité mémoire dans une application, il est très fortement recommandé d'implémenter une stratégie de type « First Touch » : chaque thread va initialiser la partie des données partagées sur lesquelles il va travailler ultérieurement.
- Si les threads sont bindés, on optimise ainsi les accès mémoire en privilégiant la localité des accès.
- Avantage : gains substantiels en terme de performance.
- Inconvénient :
  - aucun gain à escompter avec les scheduling DYNAMIC et GUIDED
  - aucun gain à escompter si la parallélisation utilise le concept des tâches explicites.

Affinités : exemples d'impact sur les performances

Code « *Memory Bound* » s'exécutant avec 4 threads sur des données privées.

```
program SBP
...
! $OMP PARALLEL PRIVATE(A,B,C)
do i=1,n
    A(i) = A(i)*B(i)+C(i)
enddo
!$OMP END PARALLEL
...
end program SBP
```

```
>export OMP_NUM_THREADS=4
>export KMP_AFFINITY=compact
>a.out
```

```
Temps elapsed = 116 s.
```

```
>export OMP_NUM_THREADS=4
>export KMP_AFFINITY=scatter
>a.out
```

```
Temps elapsed = 49 s.
```

☞ Pour optimiser l'utilisation des 4 bus mémoire, il est donc préférable de binder un thread par socket. Ici le mode *scatter* est 2.4 fois plus performant que le mode *compact* !

Affinités : exemples d'impact sur les performances

```
program NoFirstTouch
 implicit none
 integer, parameter    :: n = 30000
 integer :: i, j
 real, dimension(n,n) :: TAB

 ! Initialisation de TAB
 TAB(1:n,1:n)=1.0

 !$OMP PARALLEL
 ! Calcul sur TAB
 ! $OMP DO SCHEDULE(STATIC)
 do j=1,n
    do i=1,n
       TAB(i,j)=TAB(i,j)+i+j
    endd
 o enddo
 !$OMP END PARALLEL
end program NoFirstTouch
```

```
>export OMP_NUM_THREADS=32; a.out
```

```
Temps elapsed = 98.35 s.
```

```
program FirstTouch
 implicit none
 integer, parameter    :: n = 30000
 integer :: i, j
 real, dimension(n,n) :: TAB
 !$OMP PARALLEL
 ! Initialisation de TAB
 ! $OMP DO SCHEDULE(STATIC)
 do j=1,n
    TAB(1:n,j)=1.0
 enddo
 ! Calcul sur TAB
 ! $OMP DO SCHEDULE(STATIC)
 do j=1,n
    do i=1,n
       TAB(i,j)=TAB(i,j)+i+j
    endd
 o enddo
 !$OMP END PARALLEL
end program FirstTouch
```

```
>export OMP_NUM_THREADS=32; a.out
```

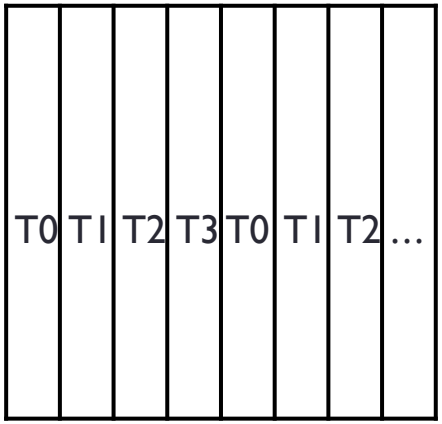
```
Temps elapsed = 10.22 s.
```

☞ L'utilisation de la stratégie de type « *First Touch* » permet un gain de l'ordre d'un facteur 10 sur cet exemple !

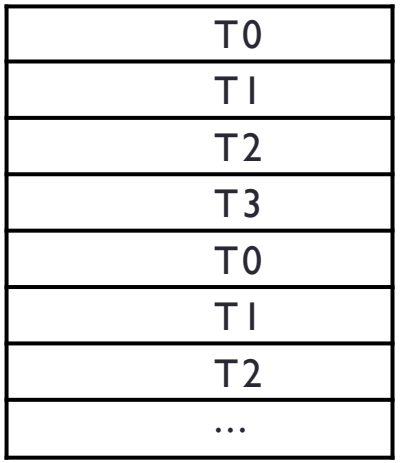
Affinités : exemples d'impact sur les performances

- Code de type « directions alternées » s'exécutant avec 4 threads sur un tableau 2D partagé, tenant dans le cache L3 d'un socket. C'est un exemple pour lequel il n'y a pas de localité *thread d'exécution/donnée*.
  - Aux itérations paires, chaque thread travaille sur des colonnes du tableau partagé.
  - Aux itérations impaires, chaque thread travaille sur des lignes du tableau partagé.

Itération paire



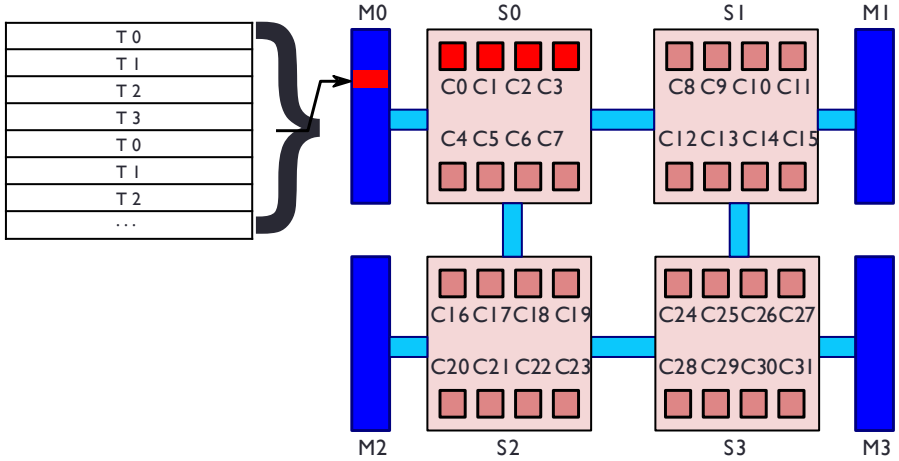
Itération impaire



- La stratégie « *First Touch* » est utilisée.
- On va comparer un binding de type *compact* avec un binding de type *scatter*.

Affinités : exemples d'impact sur les performances

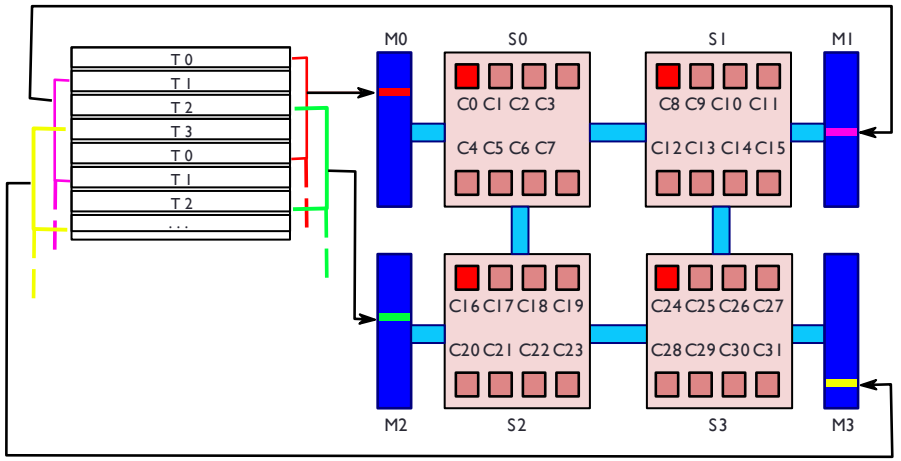
☞ Binding de type *compact*




```
>export MP_NUM_THREADS=4 ; a.out
```

```
Temps elapsed = 33.46 s.
```

☞ Binding de type *scatter*




```
>export MP_NUM_THREADS=4 ; a.out
```

```
Temps elapsed = 171.52 s.
```

☞ Dans cet exemple, le mode *compact* est plus de 5 fois plus performant que le mode *scatter* !

PERFORMANCES

Performances

- En général, les performances dépendent de l'architecture (processeurs, liens d'interconnexion et mémoire) de la machine et de l'implémentation OpenMP utilisée.
- Il existe, néanmoins, quelques règles de « bonnes performances » indépendantes de l'architecture.
- En phase d'optimisation avec OpenMP, l'objectif sera de réduire le temps de restitution du code et d'estimer son accélération par rapport à une exécution séquentielle.

Règles de bonnes performances

1. Maximiser la taille des régions parallèles.
2. Adapter le nombre de tâches demandé à la taille du problème à traiter afin de minimiser les surcoûts de gestion des tâches par le système.
3. Dans la mesure du possible, paralléliser la boucle la plus externe.
4. Utiliser la clause SCHEDULE(RUNTIME) pour pouvoir changer dynamiquement l'ordonnancement et la taille des paquets d'itérations dans une boucle.
5. La directive SINGLE et la clause NOWAIT peuvent permettre de baisser le temps de restitution au prix, le plus souvent, d'une synchronisation explicite.
6. La directive ATOMIC et la clause REDUCTION sont plus restrictives mais plus performantes que la directive CRITICAL.

Règles de bonnes performances

7. Utiliser la clause IF pour mettre en place une parallélisation conditionnelle (ex. sur une architecture vectorielle, ne paralléliser une boucle que si sa longueur est suffisamment grande).
8. Éviter de paralléliser la boucle faisant référence à la première dimension des tableaux (en Fortran) ou la seconde (en C/C++) car c'est celle qui fait référence à des éléments contigus en mémoire.

Règles de bonnes performances

```
#include <stdio.h>
#include <stdlib.h>
#define N 1025

int main() {
 int i, j;
 float a[N][N],  b[N][N];

 for(j=0; j<N; j++)
     for(i=0; i<N; i++)
         a[i][j] = (float) drand48();

 #pragma omp parallel do schedule(runtime)  if(N > 514)
 {
        for(j=1; j<N-1; j++)
            for(i=0; i<N; i++)
                b[i][j] = a[i][j+1]  - a[i][j-1];
 }
 return  0;
}
```

Règles de bonnes performances

- Les conflits inter-tâches (de banc mémoire sur une machine vectorielle ou de défauts de cache sur une machine scalaire), peuvent dégrader sensiblement les performances.
- Sur une machine multi-noeuds à mémoire virtuellement partagée (ex. SGI-O2000), les accès mémoire (type NUMA : Non Uniform Memory Access) sont moins rapides que des accès intra-noeuds.
- Indépendamment de l'architecture des machines, la qualité de l'implémentation OpenMP peut affecter assez sensiblement l'extensibilité des boucles parallèles.

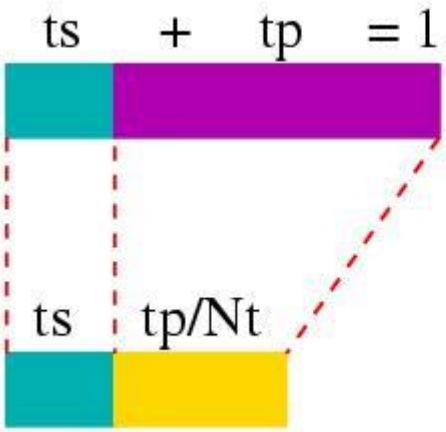
Mesure du temps

- OpenMP offre deux fonctions :
  1. OMP_GET_WTIME pour mesurer le temps de restitution en secondes ;
  2. OMP_GET_WTICK pour connaître la précision des mesures en secondes.
- Ce que l'on mesure est le temps écoulé depuis un point de référence arbitraire du code.
- Cette mesure peut varier d'une exécution à l'autre selon la charge de la machine et la répartition des tâches sur les processeurs.

Accélération

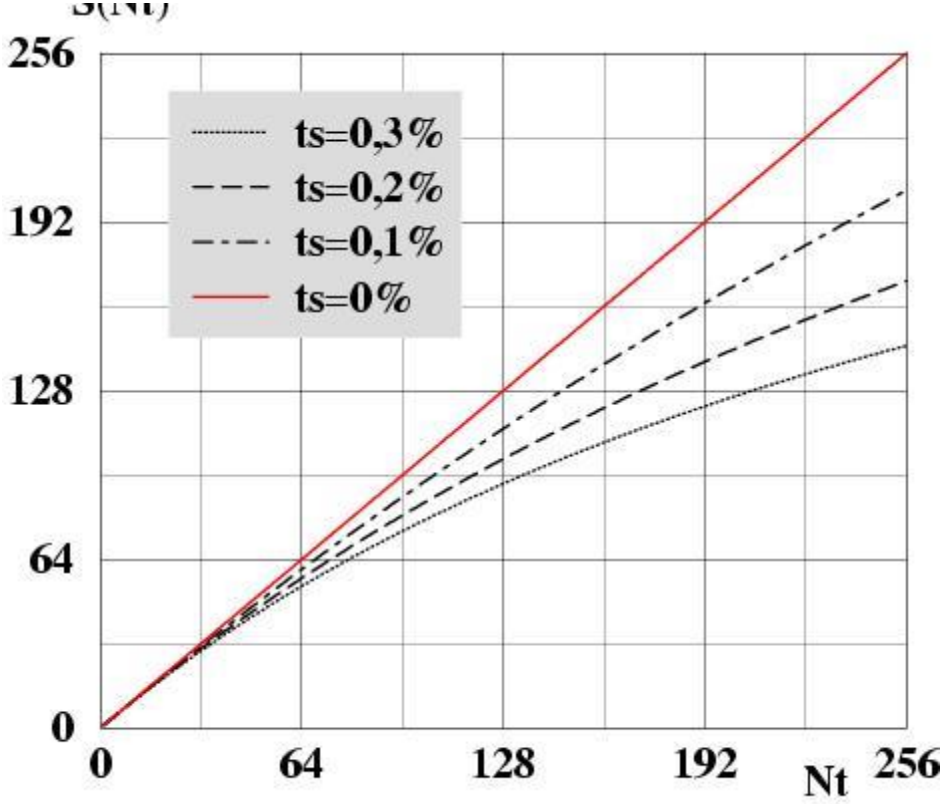
- Le gain en performance d'un code parallèle est estimé par rapport à une exécution séquentielle.
- Le rapport entre le temps séquentiel Ts et le temps parallèle Tp sur une machine dédiée est déjà un bon indicateur sur le gain en performance. Celui-ci définit l'accélération S(Nt) du code qui dépend du nombre de tâches Nt.
- Si l'on considère Ts=ts+tp=1 (ts représente le temps relatif à la partie séquentielle et tp celui relatif à la partie parallélisable du code), la loi dite de « Amdhal » S(Nt)=1/(ts+(tp/Nt)) indique que l'accélération S(Nt) est majorée par la fraction séquentielle 1/ts du programme.

Accélération



$$S(Nt) = \frac{1}{ts + \frac{tp}{Nt}}$$

Accélération



Résumé

- OpenMP
  - Modèle de programmation parallèle à mémoire partagée
  - Modèle *fork/join*
- Possibilité de partager du travail
  - Boucle *for*
- Gestion de tâches explicites

ACTIVITÉ DE R&D HPC