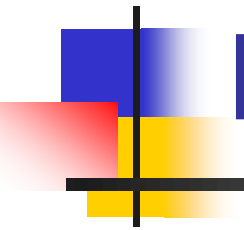
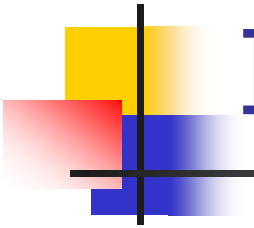


Programmation Parallèle et Distribuée

Cours 9 : Partage de travail en OpenMP

Patrick Carribault
David Dureau
Marc Pérache (marc.perache@cea.fr)



Introduction

- Programmation à mémoire partagée
  - Modèle OpenMP
  - Ensemble de directives + bibliothèque + variables d'environnement
- Modèle fork/join
  - Par défaut le code est séquentiel
  - Directives pour entrer dans une région parallèle
  - Règles strictes sur le flot de données (par défaut *shared*, possibilité de *private* avec/sans initialisation)
- Synchronisations

Introduction

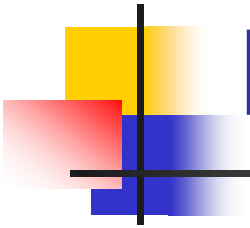
```
#include <omp.h>
#include <stdio.h>

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

Header

Directive

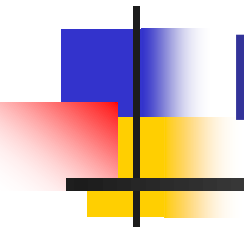
Région parallèle

```
$ 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
```

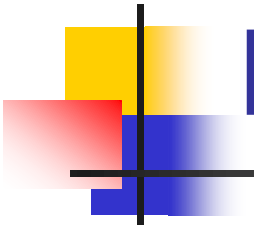


Plan du cours 9

- Partage de travail
  - Boucle parallèle
  - Multiples sections
- Exclusion de travail
  - Single/Master
- Performances
  - Mesure de temps
  - Loi d'Amdhal

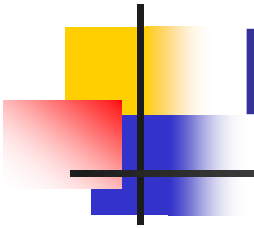


Partage de travail



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.

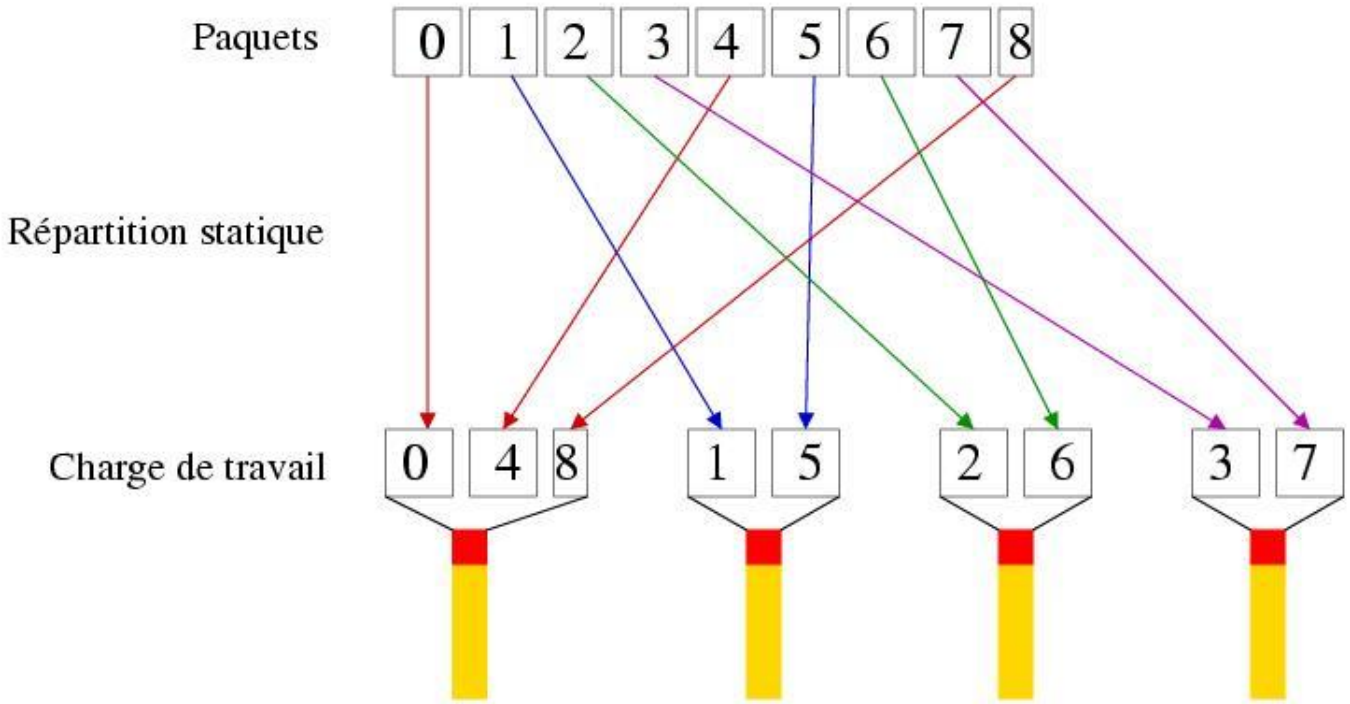
Boucle parallèle

```
#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
```

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

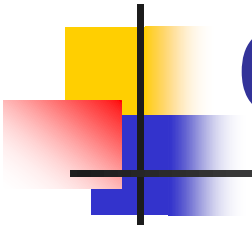
```
#include <stdio.h>

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
```



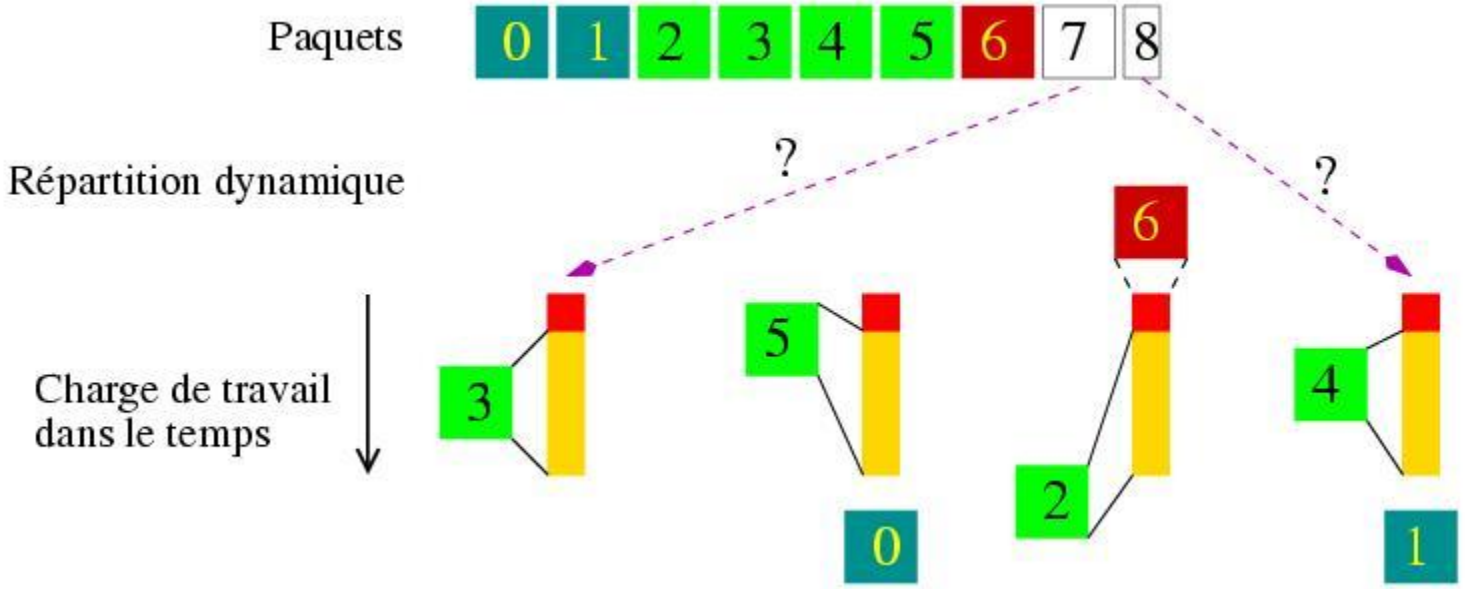
Clause schedule

- 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.

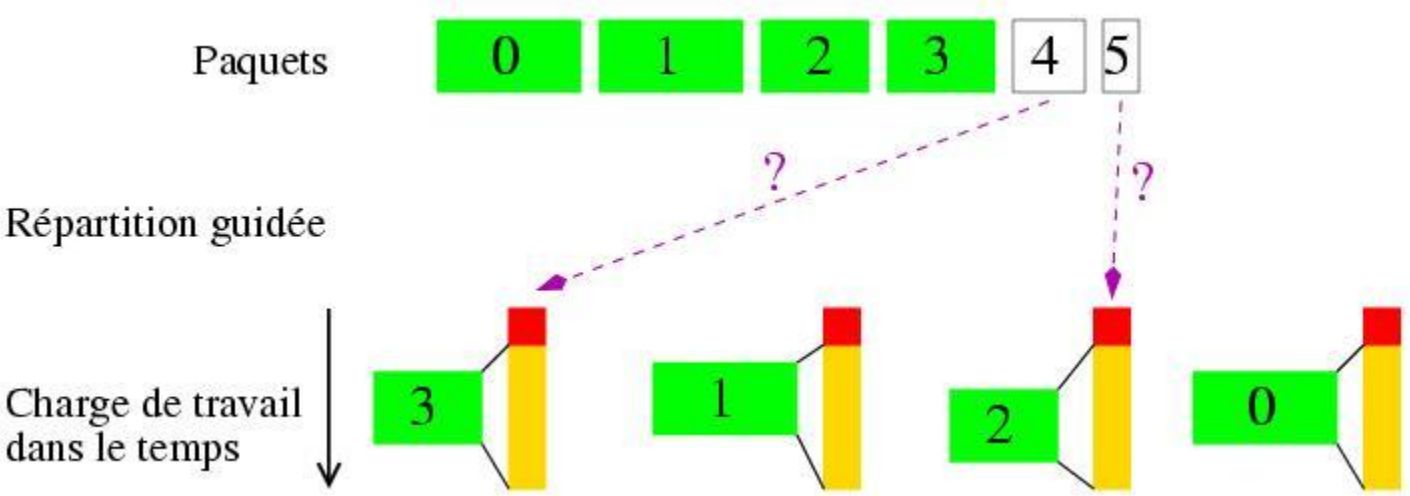
```
$ export OMP_SCHEDULE="DYNAMIC,480"
$ export OMP_NUM_THREADS=4 ; ./prog
```

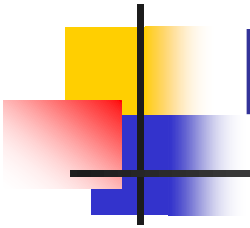


Clause schedule

- 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
```



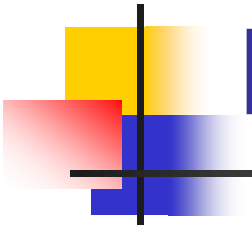


Exécution ordonnée

- 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;
}
```



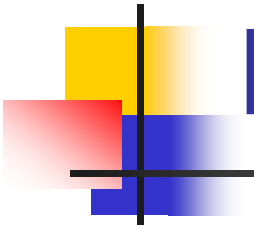
Réduction

- 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

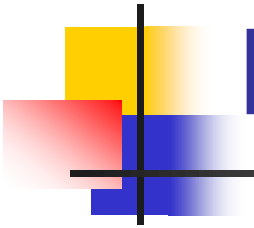
```
#include <stdio.h>
#define N 5
int main()
{
  int i, s=0, p=1, r=1;

  #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
```



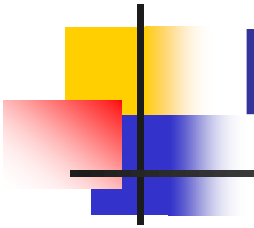
Flot de données

- 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.



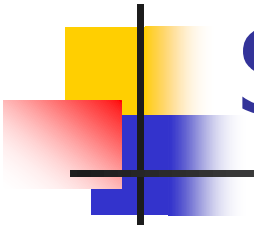
Région combinée

- 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.



Nid de boucles

- Depuis la norme 3.0, il est possible de partager le travail d'un ensemble de boucles imbriqués (nid de boucles)
- Utilisation de la clause collapse(int)
  - L'entier en paramètre définit la profondeur du nid de boucles càd le nombre de boucles imbriqués
  - Le nid de boucles doit être parfait

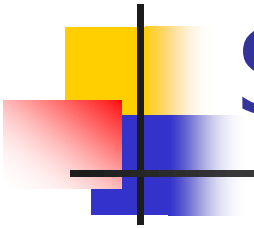


Sections parallèles

- 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.

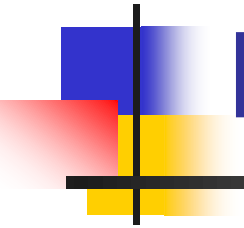
Sections parallèles

```
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;
}
```

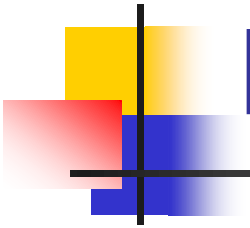


Sections parallèles

- 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.

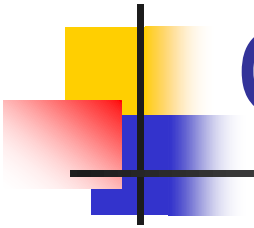


Exécution exclusive



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.

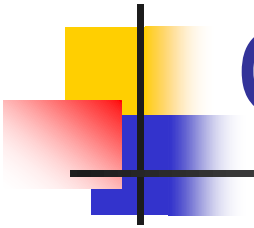


Construction *master*

- 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;
}
```

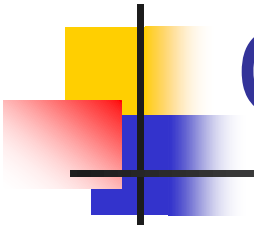


Construction *single*

- 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;
}
```

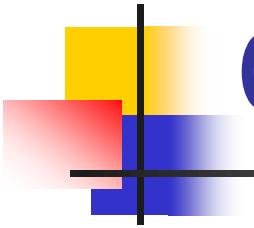


Construction *single*

- 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;
}
```

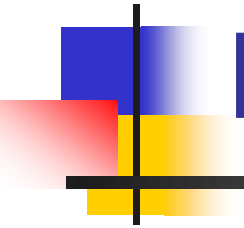


Quelques pièges

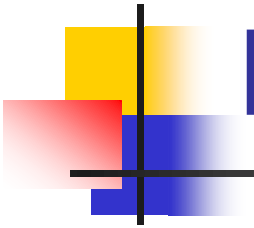
- Dans le second exemple ci-contre, il se produit un effet de course entre les tâches qui fait que l'instruction « print » n'imprime pas le résultat escompté de la variable « s » dont le statut est SHARED. Il se trouve ici que NEC et IBM fournissent des résultats identiques mais il est possible et légitime d'obtenir un résultat différent sur d'autres plateformes. La solution est de glisser, par exemple, une directive BARRIER juste après l'instruction « print ».

Quelques pièges

```
#include <stdio.h>

int main()
{
    float s;

    #pragma omp parallel default(none) shared(s)
    {
        #pragma omp single
        {
            s=1.;
        }
        printf(s = %f\n",s);
        s=2.;
    }
    return 0;
}
```

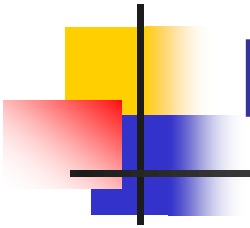


Performances



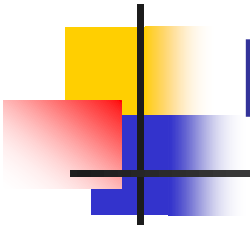
Performances

- 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.

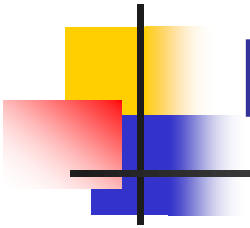


Règles de bonnes performances

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.

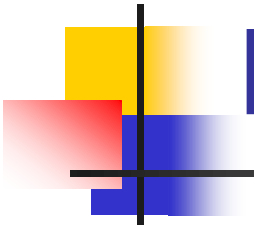
Règles de bonnes performances

```
#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;
}
```



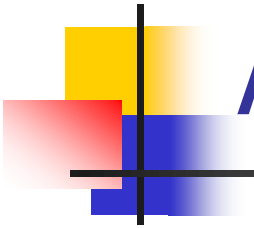
Règles de bonnes performances

- 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

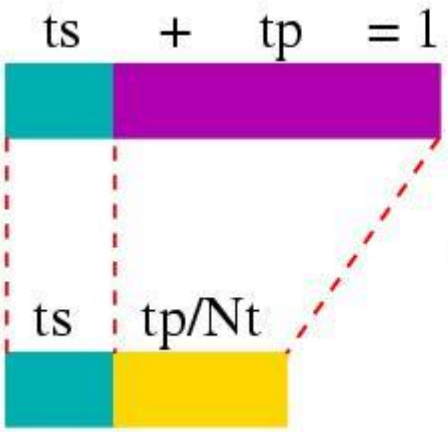
- OpenMP offre deux fonctions :
  1. OMP_GET_WTIME pour mesurer le temps de restitution en secondes ;
  2. OMP_GET_WTICK pour connaître la précision des mesures en secondes.
- Ce que l'on mesure est le temps écoulé depuis un point de référence arbitraire du code.
- Cette mesure peut varier d'une exécution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



Accélération

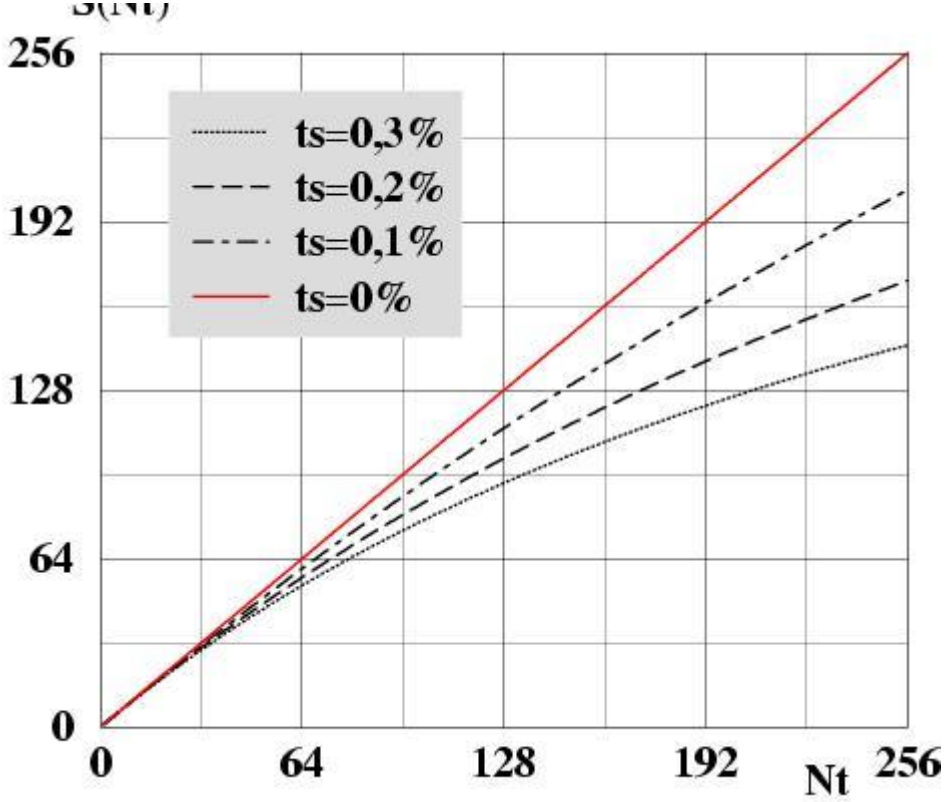
- Le gain en performance d'un code parallèle est estimé par rapport à une exécution séquentielle.
- Le rapport entre le temps séquentiel Ts et le temps parallèle Tp sur une machine dédiée est déjà un bon indicateur sur le gain en performance. Celui-ci définit l'accélération S(Nt) du code qui dépend du 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.

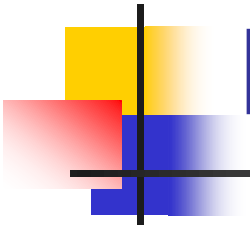
Accélération



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

Accélération





Résumé

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