OpenMP
www.openmp.org

- Une API pour la programmation parallèle en mémoire partagée
  - C, C++, Fortran
  - Portable
  - Porté par l'*Architecture Review Board* (Intel, IBM, AMD, Microsoft, Cray, Oracle, NEC…)
- Basée sur des annotations : #pragma omp *directive*
- et des fonctions: omp_*fonction*()
- Permet de paralléliser un code de façon plus ou moins intrusive un code
  - Plus on en dit plus on a de performance
  - Facile à mettre en œuvre par un non spécialiste… Trop facile ?
- Ne permet pas de créer ses propres outil de synchronisation

- Cours sur internet :
  - https://computing.llnl.gov/tutorials/openMP/
  - http://software.intel.com/en-us/search/site/language/en/type/bds_video?query=openmp

Hello world !

```
int main()
{
 printf("bonjour\n");
 printf("au revoir\n") ;

 return EXIT_SUCCESS;
}
```

```
> gcc bon.c
>./a.out
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>

int main()
{
#pragma omp parallel
 printf("bonjour\n");

 printf("au revoir\n") ;

 return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>

int main()
{
#pragma omp parallel
 printf("bonjour\n");

 printf("au revoir\n") ;

 return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>OMP_NUM_THREADS=3 ./a.out
bonjour
bonjour
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>

int main()
{
#pragma omp parallel
  printf("bonjour\n");

 printf("au revoir\n") ;

  return EXIT_SUCCESS;
}
```

```
> export OMP_NUM_THREADS=3
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>
int main()
{
#pragma omp parallel num_threads(3)
  printf("bonjour\n");

 printf("au revoir\n") ;
  return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
au revoir
>
```

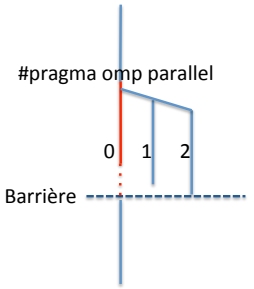
Hello world !

```
#include <omp.h>
int main()
{
omp_set_num_threads(3);
#pragma omp parallel
  printf("bonjour\n");

 printf("au revoir\n") ;
  return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
au revoir
>
```

Parallélisme fork-join

- Un unique thread exécute séquentiellement le code de *main*.
- Lors de la rencontre d'un bloc parallèle, tout thread
  - crée une *équipe* de threads
  - la charge d'exécuter une fonction correspondant au bloc parallèle
  - rejoint en tant que *maître* l'équipe.
- À la fin du bloc parallèle
  - les threads d'une même équipe s'attendent au moyen d'une barrière *implicite*
  - les threads esclaves sont démobilisés
  - le thread maître retrouve son équipe précédente



Hello world !

```
#include <omp.h>
int main()
{
#pragma omp parallel
 {
    printf("bonjour\n");
    // barrière implicite
 }
printf("au revoir\n") ;
 return EXIT_SUCCESS;
}
```

Calcul en parallèle de Y[i] = f(T,i) distribution d'indices

```
double input[NB_ELEM], output[NB_ELEM];

int main()
{
…
 #pragma omp parallel
 #pragma omp for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

 …
}
```

Calcul en parallèle de Y[i] = f(T,i) distribution d'indices

```
double input[NB_ELEM], output[NB_ELEM];

int main()
{
…
 for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

 …
}
```

Calcul en parallèle de Y[i] = f(T,i) distribution d'indices

```
double input[NB_ELEM], output[NB_ELEM];

int main()
{
…
 #pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

 …
}
```

Calculer Y[i] = f^k(T,i)

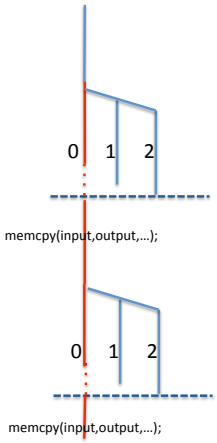
```
int main()
{
…
for (int etape = 0; etape < k; etape++)
{
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

memcpy(input,output,…);
}
 …
}
```

Calculer Y[i] = f^k(T,i)

```
int main()
{
…
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

memcpy(input,output,…);
}
 …
}
```

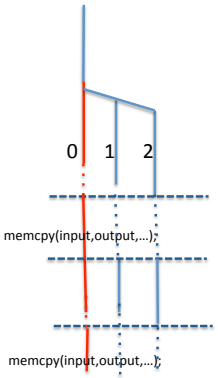
Calculer Y[i] = f^k(T,i) OpenMP

```
int main()
{
…
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

memcpy(input,output,…);
}
 …
}
```



Calculer Y[i] = f^k(T,i) OpenMP

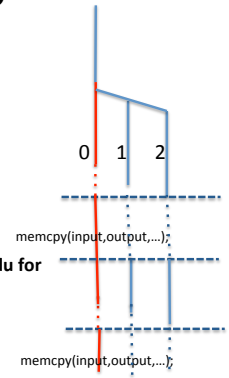
```
int main()
{
…
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

memcpy(input,output,…);
}
 …
}
```



Réduire le coût de la parallélisation

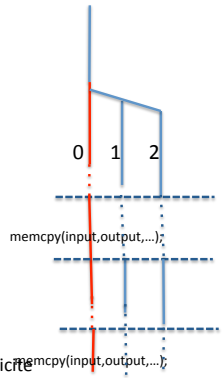
Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
#pragma omp parallel
for (int etape = 0; etape < k; etape++)
{
#pragma omp for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);
 // barrière implicite placée par le compilateur à la suite du for

 if (omp_get_thread_num() == 0)
     memcpy(input,output,…);
 #pragma omp barrier
}
 …
}
```



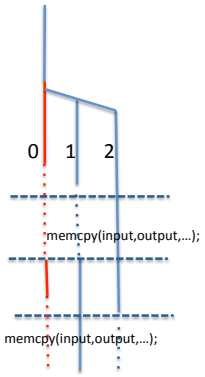
Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
#pragma omp parallel
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);
 // barrière implicite
#pragma omp master
 memcpy(input,output,…);
 #pragma omp barrier // nécessaire car pas de barrière implicite
}
 …
}
```



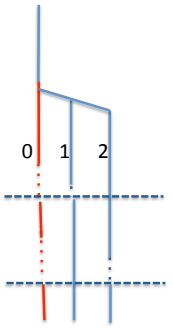
Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
#pragma omp parallel
for (int etape = 0; etape < k; etape++)
{
#pragma omp for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

#pragma omp single
 memcpy(input,output,…);
// le compilateur place ici une barrière implicite
}
 …
}
```



Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
 double *entree = input;
 double *sortie = output;
…
#pragma omp parallel firstprivate(entree, sortie)
for (int etape = 0; etape < k; etape++)
{
#pragma omp for
for( int n= debut; n < fin; n++)
  sortie[n] = f(entree,n);
 echange (&entree,&sortie);
}
 …
}
```



omp parallel

#pragma omp parallel

- barrière implicite inévitable à la fin de la section parallèle
- nombre de threads : num_threads(n)
  - Dépendant de l'implémentation
  - Au maximum n threads exécuteront la section

- clauses sur le partage des variables
  - default (none | shared | private | firstprivate)
  - private(…) shared(…)
  - fisrtpivate(…) : la valeur de la variable est recopiée à l'initialisation du thread
  - lastprivate(…) : affectée par le thread exécutant la dernière affectation relativement au programme séquentiel

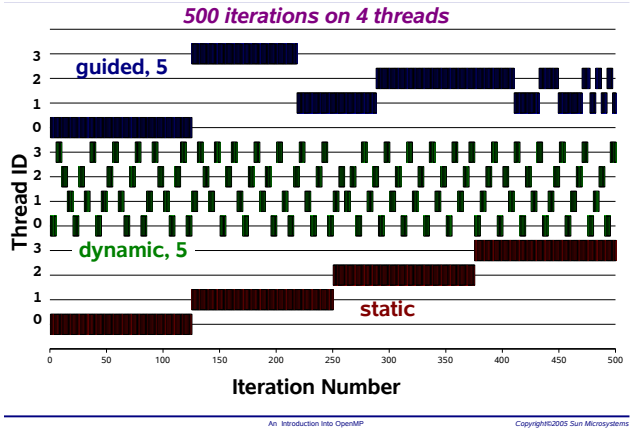
Il faut privilégier la clause *firstprivate* pour les variables fréquemment accédées car la clause *shared* limite les possibilité d'optimisations sur cette variable.

omp for

#pragma omp for

- nowait
  - Suppression de la barrière implicite
- schedule(mode,taille)
  - (**static**) : distribution par bloc
  - (static,1) : distribution cyclique
  - (static,n) : cyclique par tranche de n
  - (**dynamic**,n) : à la demande par tranche de n
  - (**guided**,n) : à la demande, par tranches de tailles décroissantes = MAX(n, (nb indices restants) / nb_threads)
  - (**runtime**) : suivant la valeur de la variable OMP_SCHEDULE
  - (**auto**) : suivant le compilateur et/ou le support d'exécution

Schedule (d'après sun)



Somme

- *Il s'agit de paralléliser le plus efficacement possible la boucle suivante (en modifiant au besoin le code):*

```
for(i=0 ; i < 1000 ; i++)
        s += f(i) ;
```

  - *En supposant que le temps de calcul de f(i+1) est toujours (très) supérieur à celui de f(i).*

- Trois solutions pour mettre à jour la variable s:
  - Utilisation d'une section critique pour chaque indice
  - Utilisation d'une variable atomique pour chaque indice
  - Utilisation d'une variable locale et une seule mise à jour par thread

Somme

```
#pragma omp parallel for shared(s) schedule(dynamic)
    for(i=999 ; i >0 ; i--)
        {
        int x = f(i);
        #pragma omp critical
            s += x;
        }
#pragma omp parallel for shared(s) schedule(dynamic)
    for(i=999 ; i >0 ; i--)
        {
         int x = f(i);
        #pragma omp atomic
            s += x;
        }

#pragma omp parallel for reduction(+:s) schedule(dynamic)
    for(i=999 ; i >0 ; i--)
        s += f(i) ;
```

Critical vs atomic

#pragma omp critical [*identificateur*]

- Sans identificateur un mutex par défaut est utilisé
- Pas de barrière implicite

- #pragma omp atomic
  - Remplacée au besoin par un critical
  - Possibilité de faire des instructions du type fetch_and_add()

```
#pragma omp atomic capture
{v=x; x--;}
```

```
#pragma omp parallel for
   for (i = 0; i < 2000; i++)
    for (j = 0; j < 2000; j++)
#pragma omp atomic
       x++;
```

Nb threads	critical	atomic
sans openMP	0,018	0,018
1	0,2	0,15
2	1,7	0,7
3	2,4	0,9
6	6	1,2
12	12	1,2
24	24	1,2
48	47	1,2

Parallélisme de données Vectorisation

- Vectorisation = appliquer une même opération à un vecteur de n données
  - Nos compilateurs savent souvent le faire automatiquement
  - Ce pragma autorise le compilateur à *vectoriser* (si possible) la boucle en fermant les yeux quant aux éventuelles dépendances de données inter itérations

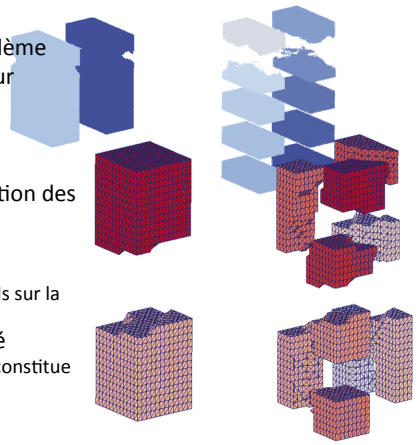
```
#pragma omp simd
    for(i=1 ; i <1000 ; i++)
        c[i]=a[i]+b[i] ;
```

```
#pragma omp simd for
    for(i=1 ; i <1000 ; i++)
        c[i]=a[i]+b[i] ;
```

Parallélisme imbriqué

- Exprimer plus de parallélisme
  - Adapter le parallélisme au problème
  - Méthode récursive « diviser pour régner »

- Difficultés:
  - Surcoût à la création lié à la gestion des équipes
    - Support d'exécution
    - Ordonnanceur système
    - Répartition des équipe de threads sur la machine
  - Déterminer la bonne granularité
    - Une technique en plus, mais ne constitue pas une solution en soit



Parallélisme imbriqué
Diviser pour régner

```
void tsp (int hops, int len, Path_t path, int *cuts)

{
    #pragma omp parallel for
    for(i …)
        …
        tsp(hops+1, len+dist, path, cuts) ;
        …
}

Int main()
{
omp_set_nested(1);
…
}
```

Parallélisme imbriqué
Diviser pour régner

```
void tsp (int hops, int len, Path_t path, int *cuts)

{
    #pragma omp parallel for
    for(i …)
        …
        tsp(hops+1, len+dist, path, cuts) ;
        …
}

Int main()
{
omp_set_nested(1);
…
}
```

Comment ne pas trop générer de threads ?
Comment allouer la mémoire des chemins ?

Parallélisme imbriqué
Diviser pour régner

- Limiter le nombre de threads

```
void tsp ((int hops, int len, Path_t path, int *cuts)

{
    #pragma omp parallel for if (p < PROFMAX)
    for(i …)
        …
        tsp(hops+1, len+dist, path, cuts) ;
        …
}
```

L'expérience montre qu'il vaut mieux dupliquer le code et ne pas user de la clause if (pour le moment)

Parallélisme imbriqué
Diviser pour régner

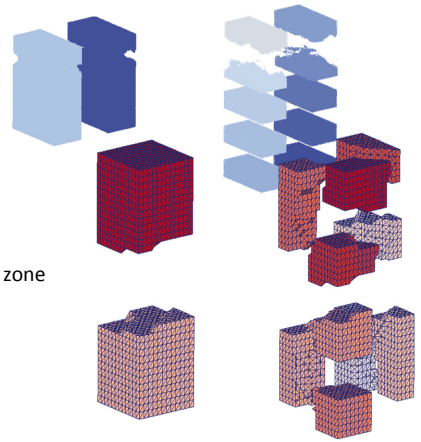
- Allouer la mémoire :

```
void tsp (int hops, int len, Path_t path, int *cuts)
{   …
    if (hops < PROFMAX)
    {
        #pragma omp parallel for
        for(i …)
        {
            int mycuts[NrTowns];
            int mypath[NrTowns];
            memcpy(mypath, path, … )
            …
            tsp(hops+1, len+dist, mypath, mycuts) ;
            …
        }
```

Il suffit d'allouer dans la pile car la variable path n'est pas modifiée même si elle est partagée par les différent threads.

Parallélisme imbriqué

```
omp_set_nested(1);

#omp parallel num_threads(externe)
{
// distributions des zones de travail

#omp parallel for num_threads(interne)
 {
 // parallélisation du travail au sein d'une zone
  }
```



Parallélisme imbriqué
Comment fixer les threads sur l'architecture ?

- Pour les implémentations actuelles, il ne vaut mieux pas utiliser plus de threads que de cœurs.

GNU

GOMP_CPU_AFFINITY=0,1,2,3,..,15 = *0-15*

Utile pour un seul niveau de parallélisme

OMP 4.0 (à venir)

OMP_PLACES={0,1,2,3}, {4,5,6,7}, ..., , {28,29,30,31}= *{0:4}:4:8*

OMP_PROC_BIND="spread,close"

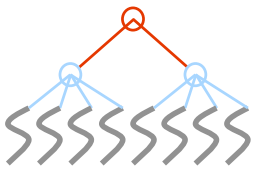
```
#omp parallel proc_bind(spread) num_threads(externe)
{
// distributions des zones de travail

#omp parallel proc_bind(close)  num_threads(interne)
 {
 // parallélisation du travail au sein d'une zone
  }
```

Exemple : le benchmark BT-MZ

- Parallélisation à deux niveaux
  - Plusieurs simulations sur des zones différentes
  - Plusieurs threads pour traiter chacune des zones
  - Imbrication de régions parallèles OpenMP
  - Allocation mémoire: *first-touch*
- Parallélisme externe irrégulier
  - Différentes charges de travail selon les zones
- Parallélisme interne régulier
  - Les threads d'une même zone ont la même quantité de travail à effectuer

2 zones traitées en parallèle

4 threads par zone



Arbre de threads obtenu lors d'une exécution « 2x4 » du benchmark BT-MZ

BT-MZ: Résultats expérimentaux

Outer x Inner	GOMP 3	Intel	Cache	
			Original	Info de charge
4 x 4	9.4	13.8	14.1	14.1
16 x 1	**14.1**	**13.9**	14.1	14.1
16 x 4	11.6	6.1	14.1	14.9
16 x 8	11.5	4.0	14.4	**15.0**
32 x 1	12.6	10.3	13.5	13.8
32 x 4	11.2	3.4	14.3	14.8
32 x 8	10.9	2.8	**14.5**	14.7

Accélérations obtenues sur la machine 16 cœurs de type opteron avec la classe C du benchmark BT-MZ

Parallélisme imbriqué
Qualité du support d'exécution

	GOMP 3	ICC	Forest
atomic	0,52	0,87	0,49
barrier	75,51	26,98	27,56
critical	12,90	39,01	4,13
for	80,44	28,17	27,33
lock	4,69	4,41	4,06
parallel	3209,75	304,94	171,66
parallel for	3222,49	311,58	170,56
reduction	3220,41	454,20	171,58

Benchmark Nested-EPCC : surcout (µs) de l'invocation des mot-clés OpenMP en contexte de parallélisme imbriqué, sur une machine à 16 coeurs