

OpenMP

cedric.bilat@he-arc.ch

Cedric Bilat
Prof HES
Haute Ecole Arc
cedric.bilat@he-arc.ch
Version 0.0.2

Acronyme

OMP = Open Multi Processor

Historique

Standard industriel depuis 1997

Version :	1.0	2.0	2.5	3.0
Année :	1997	2002	2005	2008
Doc :	85p	106p	250p	326p

Langages

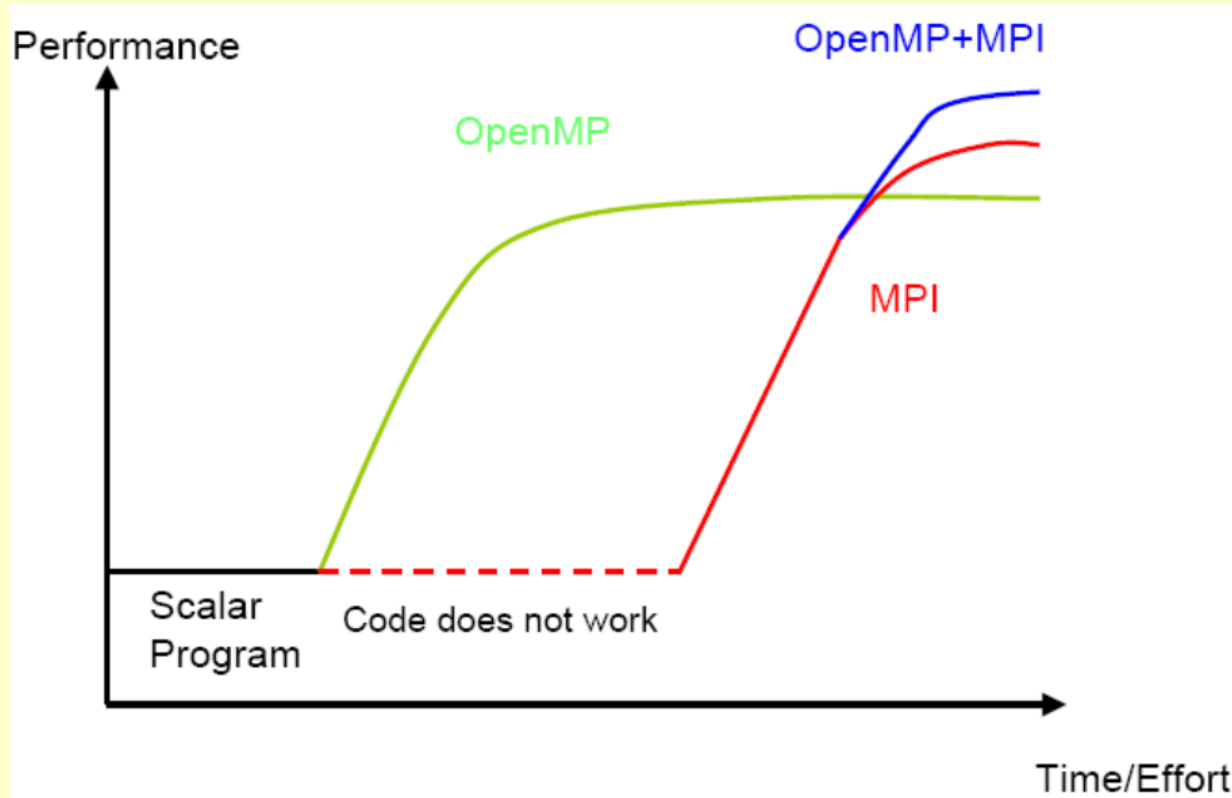
C, Java, Fortran

Compilateur

Visual, Intel compiler, GCC, ...

Objectifs

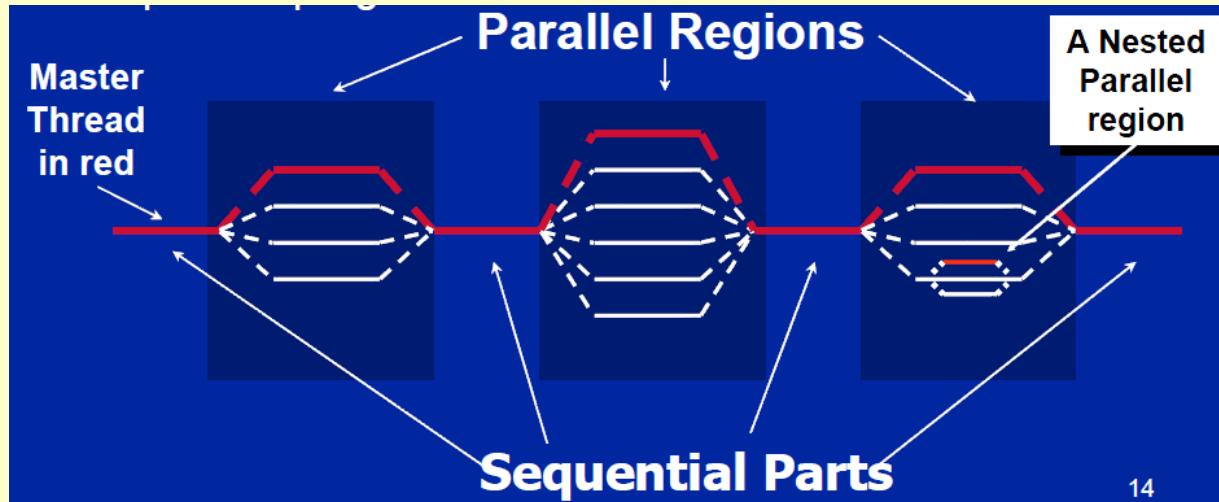
Approche simple de la programmation multi processeurs
pour machine à mémoire partagée
Portabilité!



OMP permet une parallélisation **progressive** d'un code scalaire

Principe de base : Fork/join

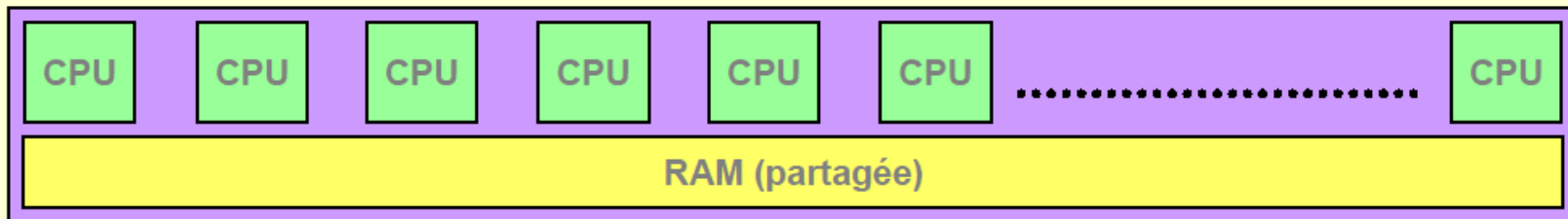
Un processus unique (**master**) est exécuté. Il a le rang(Id) **0**.
Des parties de programmes sont exécutés en parallèle
selon le modèle **fork/join**



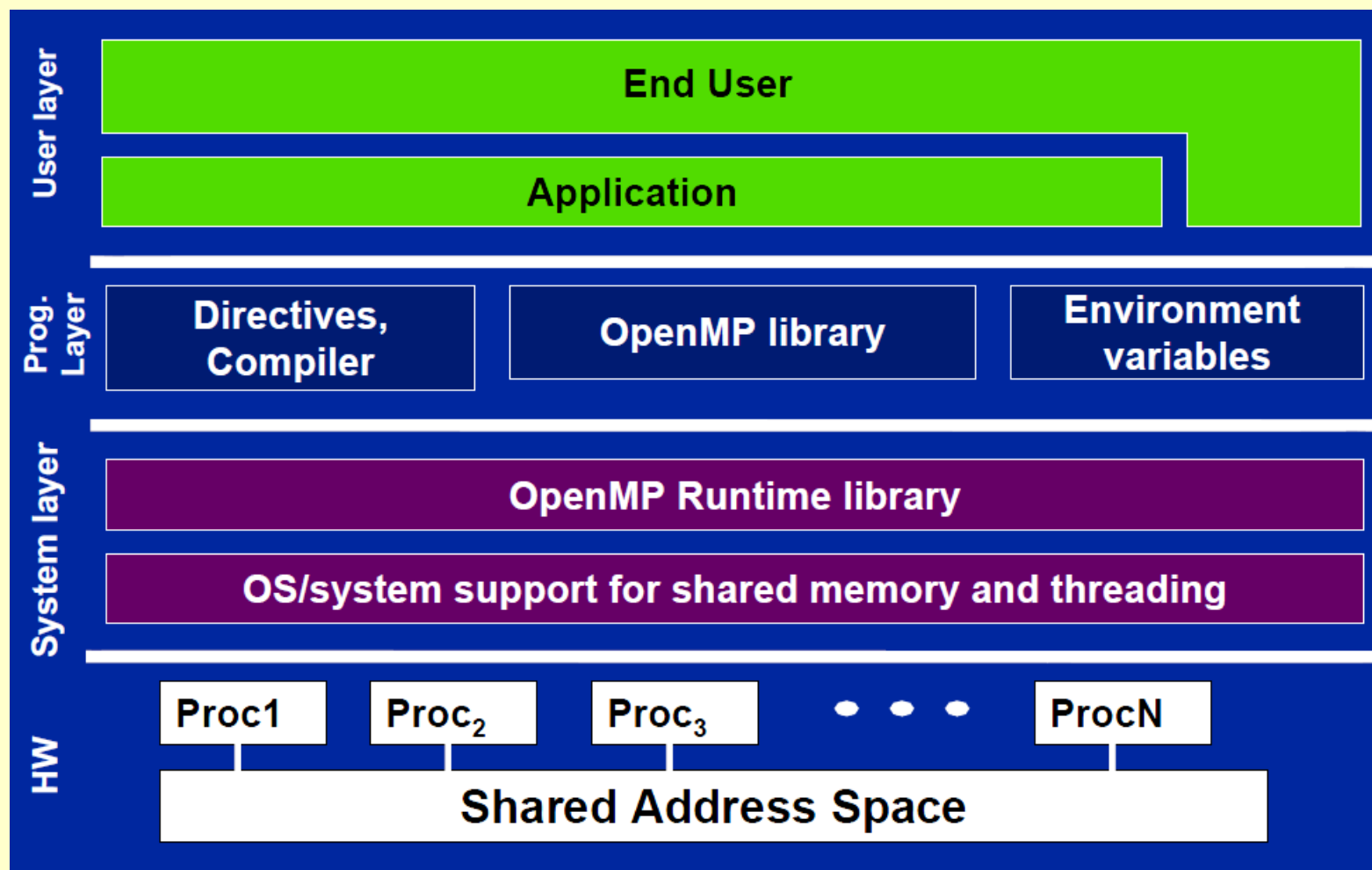
Les variables peuvent être **private** à un thread ou **partagée** entre tous les threads

Une machine à **mémoire partagée** avec

- 1 à n processeurs
- 1 à n cores pour chaque processeur



Tous les core de tous les processeurs accèdent
à une même et unique mémoire!



Windows : Intel (11) / Visual 2010

➤ Compilation	:	omp.h	omp.h
➤ Linkage	:	libiomp5MD.lib	vcomp100.lib
➤ Runtime	:	libiomp5MD.dll	vcomp100.dll

Windows : minGW

➤ Compilation	:	omp.h
➤ <i>Linkage</i>	:	libgomp.lib
➤ Runtime	:	libgomp-1.dll et aussi pthreadXXX.dll libgccXXX.dll

Linux : Intel

➤ Compilation	:	omp.h
➤ Linkage	:	libiomp5.so
➤ Runtime	:	libiomp5.so

Linux : gcc

➤ Compilation	:	omp.h
➤ Linkage	:	libgomp.so
➤ Runtime	:	libgomp.so et aussi librt.so libpthread.so

- Par directives de compilation (**#pragma omp ...**)

Exemples :

(E1) **#pragma omp parallel**

(E2) **#pragma omp parallel for reduction(+:sum, *:add)**

- Par utilisation de fonction de **omp.h**

Exemples :

(E3) **int tid=omp_get_thread_num()** //id du thread

(E4) **omp_set_num_threads(4)** //spécifier le #thread

(E5) **omp_get_num_procs()** // #cores

(E6) **omp_in_parallel()** //1 si région parallèle

Hello : Region Parallel

Code (hello.cpp)

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

void main(void)
{
    int nbThread= omp_get_num_procs(); // Facultatif
    omp_set_num_threads(nbThread);    // Facultatif
    #pragma omp parallel
    {
        int id=omp_get_thread_num();
        std::cout<< " Hello(" <<id<< " ) " <<std::endl;
    } // Implicit barrier
}
```

Région Parallèle

Le thread courant (avant l'appel #pragma) devient le master thread.

Il est utilisé dans le pool de thread exécutant la région parallèle. Le master thread à l'id=0

Compilation Flag

Visual : /openmp

Intel : /openmp (windows) -fopenmp (Linux)

Mingw : -fopenmp

Gcc : -fopenmp

Execution

Hello(0)

Hello(1)

Avant de redonner la main au thread courant, tous les threads du pool s'attendent les uns les autres à la barrière

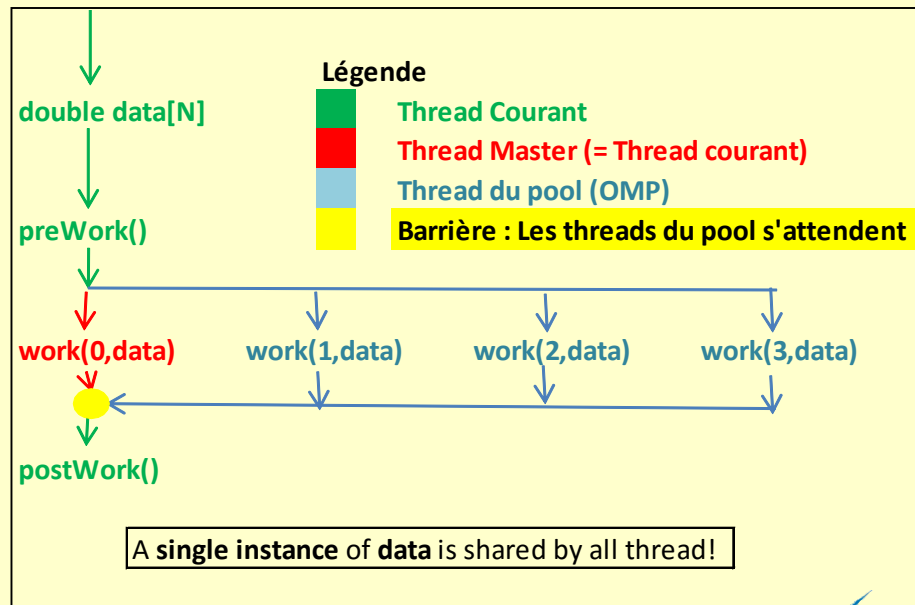
Hello : Flux Exécution

```
double data[N]=...;
prework();
#pragma omp parallel num_threads(4)
{
    int id=omp_get_thread_num();
    work(id,data);
} // Implicit barrier
postWork();
```

// une autre manière de spécifier le #thread

Région Parallèle

Chaque thread exécute
le même code redondant!
Paradigme **SPMD**
Single **P**rogram **M**ultiple **D**ata



Une des cibles OMP est la **parallelisation des boucles**.

OMP permet une approche **incrémentale** du parallélisme

- Peu de changement du code source!
- Changement progressif du code source!

```
//Programme Séquentiel
double data[N]=...;
for(int i=1;i<=N;i++)
{
    work(data[i-1]);
}
postWork();
```

```
//Programme Parallèle
double data[N]=...;
#pragma omp parallel for
for(int i=1;i<=N;i++)
{
    work(data[i-1]);
} // Implicit barrier
postWork();
```

Découper en
plusieurs threads.

Tous les threads s'attendent mutuellement les uns les autres ici,
avant de redonner la main au thread courant (thread master)
qui executera seul postwork()

Les variables peuvent être **private** à un thread ou **shared** entre tous les threads. Par défaut les variables sont **shared** sauf la variable de boucle *i* d'une région parallèle `for`, qui est évidemment **private**.

Les deux codes suivants sont équivalents:

```
//Programme Parallèle
double data[N]=...;
#pragma omp parallel for
for(int i=1;i<=N;i++)
{
    work(data[i-1]);
} // Implicit barrier
postWork();
```

```
//Programme Parallèle (équivalent)
double data[N]=...;
int i; // Sans initialisation!!
#pragma omp parallel for private(i) shared(data,N)
for(i=1;i<=N;i++)
{
    work(data[i-1]);
} // Implicit barrier
postWork();
```

La variable de boucle *i* est implicitement **private**.
Ainsi la pile de chaque thread possède
sa propre instance de variable *i*

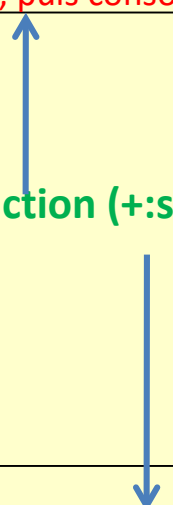
La variable *data* est implicitement **shared**.
Chaque thread accède à une même
et unique instance!

L'exemple précédent est peu représentatif des problèmes réels.
En pratique $work(data[i-1])$ fournit un résultat qu'il s'agit ensuite de combiner avec les autres $work$ pour établir un résultat final.
Une étape de **synchronisation/consolidation** est nécessaire!

La pile du thread k contient une instance de `sum` initialisé à 0.
`sum` est private jusqu'à la barrière, puis consolidé!

```
//Programme Séquentiel
double data[N]=...;
double sum=0;
for(int i=1;i<=N;i++)
{
    sum+=work(data[i-1]);
}
```

```
//Programme Parallèle
double data[N]=...;
double sum; // init inutile!!
#pragma omp parallel for reduction (+:sum)
for(int i=1;i<=N;i++)
{
    sum+=work(data[i-1]);
} // implicit barrier
```



En fin d'exécution, une barrière bloque tous les threads.
La règle de consolidation associée consiste alors à additionner entre elles, les variables `sum` associés au pile de chacun des threads.

3 possibilités de répartition de task dans le pool de thread OMP :

- ❑ Paradigme **SPMD** = **S**ingle **P**rogramme **M**ultiple **D**ata

```
#pragma omp parallel for
for(int i=1;i<=N;i++)
{
    work(data[i-1]);
} // Implicit barrier
```

Boucle Parallèle

Le code de tous les works est le même !
(**S**ingle **P**rogramme)

- ❑ Paradigme **MPMD** = **M**ultiple **P**rogramme **M**ultiple **D**ata

```
#pragma omp parallel sections
{
    #pragma omp section
    workA(...);

    #pragma omp section
    workB(...);
} //Implicit barrier
```

Sections Parallèles

WorkA et Work B s'exécutent en parallèle.
Deux travaux indépendant !
WorkA a un code différent de workB !
(**M**ultiple **P**rogramme)

- ❑ Répartition **basique**

```
#pragma omp parallel
{
    work();
}
```

Région Parallèle

Tous les threads du pool exécutent work !

Un peu de recul

Axiome :

Un code **work** ne peut s'exécuter en parallèle que s'il se trouve dans une région parallèle.

```
#....parallel...  
{  
  work();  
}
```

Des raccourcis syntaxique existe !

Full Version

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

```
#pragma omp parallel  
{  
  #pragma omp sections  
  {  
    #pragma omp section  
    structured-block  
  
    #pragma omp section  
    structured-block  
  }  
}
```

Light Version

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

```
#pragma omp parallel sections  
{  
  #pragma omp section  
  structured-block  
  
  #pragma omp section  
  structured-block  
}
```

Parfois la syntaxe light n'est pas utilisable.

```
int tab[N]=...
int a;
#pragma omp parallel shared (a,b) private(i)
{
    #pragma omp single
    {
        a=10;
    } //implicite barrier

    #pragma omp for
    for(int i=1;i<=N;i++)
    {
        tab[i-1]=a;
    } // Implicit barrier

} // Implicit barrier
```

} Exécuter que par 1 Thread

Pas forcément en premier dans la région parallèle !
On peut imaginer du code avant et après !

Identique à :

```
Int tab[N]=...
int a=10;
#pragma omp parallel for shared (a,b) private(i)
for(int i=1;i<=N;i++)
{
    tab[i-1]=a;
} // Implicit barrier
```



```
#pragma omp parallel
```

```
{  
    [green box]  
}
```

Forms a team of threads and starts parallel execution!

Le bloc vert est exécuté en parallèle, autant de fois qu'il y a de thread. Le développeur doit s'arranger que chaque instance de thread qui exécute ce bloc vert travaille sur un jeu de données différent (par exemple en jouant avec tid)

```
#pragma omp parallel for
```

```
for (int i=0;i<n;i++)
```

```
{  
    [green box]  
}
```

Shortcut for specifying a parallel execution!

Le bloc vert est exécuté en parallèle, autant de fois qu'il y a de thread

```
#pragma omp parallel
```

```
{  
    [green box]  
    #pragma omp for  
    for (int i=0;i<n;i++)  
    {  
        [green box]  
    }  
    [green box]  
}
```

Forms a team of threads and starts parallel execution! (#)

Specifies that the iterations of associated loops will be executed in parallel, by threads in the active context create in (#)

<http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf>
<http://openmp.org/mp-documents/OpenMP-4.0-C.pdf>

Par défaut, les variables d'une région parallèle sont **share** (sauf la variable de boucle de **omp parallel for** qui est private)

On peut changer ce statut avec :

```
#pragma omp parallel default(private)
```

Toutes les variables deviennent maintenant private !

- shared** : Variable de type **readable** normalement, car partagé par tout les threads !
- private** : Variable de type **readable / writable**
dont chaque thread du pool OMP possède une instance dans sa pile.
L'initialisation doit se faire à l'intérieur du thread, ie dans la région parallèle !
Si l'initialisation dépend d'une valeur connue avant la section parallèle,
Il faut utiliser **firstprivate**.
- threadprivate** : Les **variable globale** (de type non pointeur) sont par défaut shared (assez intuitif !)
Threadprivate permet de changer ce comportement. Une copie de la variable globale est alors utilisée !

Firstprivate : Variable **private** s'initialisant avec la valeur de la variable de même nom déclarée avant la section parallèle !

Lastprivate

: La dernière valeur de la variable est accessible après la section parallèle. Par dernière, on entend la valeur de la variable si le code n'avait pas été parallèle.

Autrement dit, OMP découpe en tranche l'intervalle de travail $[0, n[$. La variable *lastprivate* aura pour valeur après la boucle, la valeur de l'instance a du thread qui s'est occupé de la dernière tranche de travail

Exemple

```
int n=5;
int a;
#pragma omp parallel for default(shared) private (i) lastprivate(a) shared(n)
for(int i=0;i<n;i++)
{
    a=i+1;
    cout<< " Thread" <<omp_get_thread_num()<< " a=" <<a<< " i=" <<i<<endl;
}
cout<< " After : a= " <<endl;
```

Exécution

Thread 0 : a=1 : i=0
Thread 0 : a=2 : i=1
Thread 2 : a=5 : i=4
Thread 1 : a=3 : i=2
Thread 1 : a=4 : i=3
After : a=5

Goal

Organiser l'accès à **des données partagées** par plusieurs threads

Implicite / Explicite

Les techniques vues jusqu'ici possèdent des synchronisations (barrière) implicite. Parfois il est nécessaire d'effectuer des synchronisations explicites !

Conseil

Evitez au plus les synchronisations (dans la mesure du possible).

Les synchronisations ruinent les performances des codes parallèles !!

OMP

Technique existantes :

- Barrier
- Section critique
- Atomic assignment
-

Exemple

```
#pragma omp parallel private (tid)
{
  int tid=omp_get_thread_num();
  sleep(1000*tid); //ms
  cout<< " tid= " <<tid <<" Before" <<endl;
  #pragma omp barrier
  cout<< " tid= " <<tid <<" After" <<endl;
}
```

Les threads s'attendent ici,
Avant de repartir !

Exécution

tid=1 Before
tid=2 Before
tid=3 Before

tid=3 After
tid=2 After
tid=1 After

Exemple

```
double sum=0;
double tab[n]=....;
#pragma omp parallel private (tid)
{
    long tid=omp_get_thread_num();
    double sumLocal=0;
    #pragma omp for //Initialize by each thread !
                    //Distributed over the thread
    for(int i=1;i<=n;i++)
    {
        sumLocal+=tab[i-1];
    }
    #pragma omp critical (updateSum)
    {
        sum+=sumLocal;
        cout<< " tid= " <<tid << " sumLocal= " << sumLocal <<endl;
    }
}
cout<< « After parallel region : sum=" <<sum<<endl;
```

Sans **parallel** !!!

Identifiant de la section critique

Un thread à la fois
exécute la section critique !

Exécution

tid=1 sumLocal=36

tid=2 sumLocal=200

tid=3 sumLocal=300

After parallel region : sum=536

Note

Cet exemple effectue une **réduction** à l'aide d'une section critique.

En pratique utilisez la boucle avec réduction pour une telle tâche !

#pragma omp parallel for reduction (+:sum)

Exemple

```
double sum=0;
double tab[N]=...;
#pragma omp parallel shared(sum,tab)
{
    double sumLocal=0;           //Initialize by each thread !
    #pragma omp for              //Distributed over the thread
    for(int i=1;i<=n;i++)
    {
        sumLocal+=tab[i-1];
    }
    #pragma omp atomic
    sum=sum+sumLocal;
}
cout<< « After parallel region : sum=" <<sum<<endl;
```

shared facultatif ! variables sont shared par default.

Sans **parallel** !!!

L'action **read-add-write**
est exécutée par un seul thread
à la fois! Ce qui garantit l'intégrité
de la variable sum!

Note

Cet exemple effectue une **réduction** à l'aide d'une opération atomique.
En pratique utilisez la boucle avec réduction pour une telle tâche !

#pragma omp parallel for reduction (+:sum)

Synchronisation : Atomic Assignment : Autres exemples

cedric.bilat@he-arc.ch

Exemple 1

```
int nbThread=2*omp_get_num_procs(); //Disons 4
omp_set_num_threads(nbThread);
int n=10;
int counter=0;

#pragma omp parallel shared(n,counter) private(i)
{
    for(int i=1;i<=n;i++)
    {
        #pragma omp atomic
        counter=counter+1;
    }
}
cout<< " counter=" <<counter<<endl;
```

Private facultatif ! variables boucle sont private par default.

L'action **read-add-write**
est exécutée par un seul thread
à la fois! Ce qui garantit l'intégrité
de la variable sum

Exécution

counter=10*4= 40

Exemple 2

```
int nbThread=2*omp_get_num_procs();
omp_set_num_threads(nbThread);
int n=10;
int counter=0;
#pragma omp parallel shared(n,counter) private(i)
{
    for(int i=1;i<=n;i++)
    {
        #pragma omp atomic
        counter=counter+function(i);
    }
}
cout<< " counter=" <<counter<<endl;
```

Seul **read-add-write** est atomic!
L'évaluation de **function(i)**
s'effectue en parallèle !!!

Exemple

```
double tab[N]=...;

int nbThread=2*omp_get_num_procs();
omp_set_num_threads(nbThread);
double tabSumThread[nbThread];
Init(tabSumThread,nbThread,0);

#pragma omp parallel shared(tabSumThread,tab,n)
{
    long tid=omp_get_thread_num();
    #pragma omp for //Distributed over the thread
        for(int i=1;i<=n;i++)
        {
            tabSumThread[tid]=tab[i-1];
        }
}

//Reduction (hors OMP)
double sum=0;
for(int s=1;s<=nbThread;s++) // pas expensive, nbThreads petit
{
    sum+=tabSumThread[s-1];
}

cout<< " After parallel region : sum=" <<sum<<endl;
```

Sans **parallel** !!!

Préparation tableau auxiliaire

tabSumThread[nbThread]

Chaque thread stocke

Sa somme dans la case tabSumThread[tid]

Réduction sans synchronisation

de tabSumThread

Les bases sont là !

A vous de creuser d'avantage selon vos besoins !