Bases de la parallélisation multitâches : OpenMP (Open Multi-Processing)

Intervenant : Imane LASRI (imane.lasri@u-bourgogne.fr)
Centre de Calcul de l'université de Bourgogne (CCuB : ccub@u-bourgogne.fr)

2 février 2016

Sommaire

1 Introduction . . . 3
2 Quelques optimisations avant d'utiliser OpenMP . . . 4
3 Introduction à la parallélisation à mémoire partagée avec OpenMP . . . 7
3.1 Principe . . . 7
3.2 Qu'est ce que le calcul parallèle avec OpenMP ? . . . 7
3.3 Qu'est ce qu'une mémoire partagée et une mémoire privée ? . . . 7
3.4 Différence avec MPI . . . 7
3.5 Les avantages d'OpenMP . . . 8
3.6 Compilation . . . 8
4 Directives OpenMP . . . 8
4.1 Quelques fonctions OpenMP . . . 9
4.2 Définir une région parallèle : directive parallel . . . 9
4.3 Variables privées et partagées . . . 9
4.3.1 Clauses private() et shared() . . . 9
4.3.2 Clause firstprivate() . . . 11
4.3.3 Clause default() . . . 12
4.3.4 Clause lastprivate() . . . 13
4.4 Exécution exclusive : directives master et single . . . 14
4.4.1 Directive master . . . 14
4.4.2 Directive single . . . 15
4.4.3 Clause copyprivate() . . . 16
4.5 Parallélisation de boucles : directive do / for . . . 17
4.5.1 Clause schedule() . . . 19
4.5.2 Clause ordred() . . . 20
4.6 Barrière de synchronisation . . . 22
4.6.1 Directive barrier . . . 22
4.6.2 Clause nowait . . . 23
4.7 Opérations cumulées sur une variable . . . 24
4.7.1 Clause reduction() . . . 24
4.7.2 Directive atomic . . . 25
4.7.3 Directive critical . . . 27
4.8 Sections parallèles : directive sections . . . 27
5 Performances obtenues avec OpenMP . . . 29
6 Bibliographie . . . 30
Annexes . . . 31
Exercices . . . 40
Solutions . . . 47

1 Introduction

Au milieu des années 90 les constructeurs informatiques (Cray, SGI, IBM...) se sont intéressés à la parallélisation multitâches, chaque constructeur avait ses propres versions.

En 1997 DEC, IBM et Intel ont travaillé sur ce qui a été fait formant l'OpenMP ARB pour formaliser une interface de programmation pour Fortran ensuite pour le C/C++ en 1998.

En 2005 les supports Fortran et C/C++ ont été combiné en une seule API : OpenMP version 2.5.

Maintenant OpenMP est géré par l'association à but non lucratif : OpenMP Architecture Review Board (OpenMP ARB). Les membres collaborant à ce projet sont nombreux dont : AMD, Convey Computer , Cray, Fujitsu, HP, IBM, Intel, NEC, NVIDIA, Oracle Corporation, Red Hat, ST Microelectronics, Texas Instruments (membres permanents).

Ce cours s'adresse aux débutants qui s'intéressent à la parallélisation multi-tâches pour machines à mémoire partagée et désirent paralléliser leurs codes Fortran ou C/C++.

De nombreux compilateurs implémentent OpenMP (GNU, Intel, IBM, Oracle, Portland...) dans ce cours on utilisera GNU 4.9.2 et Intel 13.1.3.

La dernière version d'OpenMP à ce jour est la 4.0.

2 Quelques optimisations avant d'utiliser OpenMP

Avant de paralléliser un programme, il faut s'assurer que celui-ci est bien optimisé. Voici quelques règles d'optimisation :

Définir des variables pour éviter la répétition d'un calcul, `exemple` :

✗

```
int x(10), y(2), z(0);

for (int i = 0; i<100; i++)
{
  z= (x/y) + i;
}
```

✓

```
int x(10), y(2), z(0);
int a = x/z;
for (int i = 0; i<100; i++)
{
  z= a + i;
}
```

Changer l'écriture des puissances et divisions, `exemple` :

✗

```
int x(10), y(2);
int z=pow(x,2);
int z_2=z/2;
```

✓

```
int x(10), y(2);
int z=x*x;
int z_2=z*0.5;
```

Sortir des boucles les calculs qui peuvent être réalisés en dehors de celles-ci, `exemple` :

✗

```
int x(10), y(2);
int a, b;

for (int i = 0; i<100; i++)
{
  a=(x/y)*3;
  b=a+i;
}
```

✓

```
int x(10), y(2);
int a, b;

a=(x/y)*3;
for (int i = 0; i<100; i++)
{
  b=a+i;
}
```

Tableaux et ordre des indices des boucles : L'ordre des indices est très important pour la rapidité d'un code de calcul, un tableau mal initialisé peut ralentir jusqu'à 10 fois le code. La gestion de mémoire des tableaux des codes de programmation est différente : en C un tableau est déclaré en mémoire selon les lignes, en Fortran selon les colonnes.

`exemple` : Déclaration de la matrice A_{ij} (3x3)

$$A_{ij} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

programme en C++ :

```
int A [3][3]
```

gestion de la mémoire :

A_{11}	A_{12}	A_{13}
A_{21}	A_{22}	A_{23}
A_{31}	A_{32}	A_{33}

programme en Fortran :

```
integer, dimension (3,3) :: A
```

gestion de la mémoire :

A_{11}	A_{12}	A_{13}
A_{21}	A_{22}	A_{23}
A_{31}	A_{23}	A_{33}

Conséquences :
On peut voir les conséquences de cette gestion de mémoire avec cet exemple en Fortran :

exemple : programme en Fortran90

```
1  program ordre_boucle
2
3    implicit none
4    real*8, dimension (:,:), allocatable :: A
5    integer i,j,N,M
6    real  t1, t2, t3, t_cpus
7
8    N=15000
9    M=15000
10
11   allocate (A(M,N))
12
13   call cpu_time(time=t1)
14   ! initialisation de la matrice A : boucle sur i ensuite j
15   do i=1,M
16      do j=1,N
17               A(i,j)=0.d0
18      enddo
19   enddo
20
21   call cpu_time(time=t2)
22   t_cpus=t2-t1
23
24   write(*,*)"boucle i,j"
25   write(*,*)"temps CPU :",t_cpus  !affiche le temps CPU
26
27   ! initialisation de la matrice A : boucle sur j ensuite i
28   do j=1,N
29      do i=1,M
30          A(i,j)=0.d0
31      enddo
32   enddo
33
34   call cpu_time(time=t3)
35   t_cpus=t3-t2
36
37   write(*,*)"boucle j,i"
38   write(*,*)"temps CPU:",t_cpus   !affiche le temps CPU
39
40   deallocate (A)
41
42 end program ordre_boucle
```

Avec le compilateur GNU 4.4.7 :

```
$ gfortran -o gnu 01_ordre_boucle.f90
$ ./gnu
 boucle i,j
 temps CPU     :    9.5975409
 boucle j,i
 temps CPU     :    1.1278276
```

Avec le compilateur intel 13.1.3 :

```
$ module load intel/13.1.3
$ ifort -o intel 01_ordre_boucle.f90
$ ./intel
 boucle i,j
 temps CPU     :   0.8778670
 boucle j,i
 temps CPU     :   0.4339340
```

Le temps d'exécution du code avec une itération sur des valeurs non contiguës en mémoire (`i` ensuite `j`) est beaucoup plus long qu'avec une itération sur des valeurs contiguës en mémoire.

Le compilateur intel peut faire une optimisation en changeant l'ordre des boucles pour réduire le temps d'exécution.

En C faire une itération sur `i` ensuite sur `j` serait plus rapide qu'une itération sur `j` ensuite `i`.

Application :

Compilez le code `ordre_boucle` avec les options : -O0 -O1 -O2 et -O3 en utilisant les deux compilateurs intel et GNU. Que constatez-vous ?

3 Introduction à la parallélisation à mémoire partagée avec OpenMP

3.1 Principe

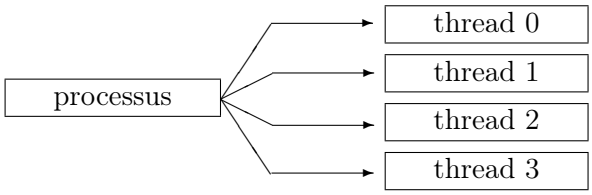
OpenMP est une interface de programmation (API) portable pour le calcul parallèle sur les ordinateurs à mémoire partagée. OpenMP est utilisable avec les langages Fortran, C et C++ sur des platformes Widows et UNIX.

3.2 Qu'est ce que le calcul parallèle avec OpenMP ?

C'est la répartition des charges de calcul sur plusieurs processus légers appelés threads. Le but d'un calcul en parallèle est de diminuer le temps d'exécution du programme.

Un programme est exécuté par une seule tâche (processus). En entrant dans une région parallèle cette tâche active plusieurs "sous-tâches" (processus légers ou threads). chaque thread est exécuté sur un cœur du processeur.

Une tâche en séquentiel est exécutée par le thread maître : thread de rang 0.

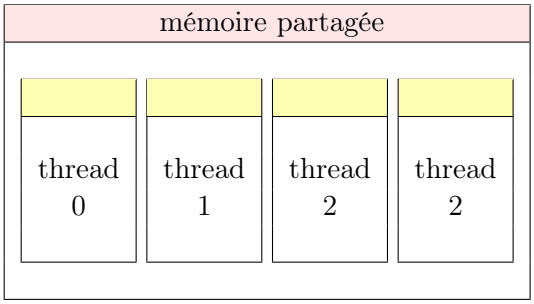


3.3 Qu'est ce qu'une mémoire partagée et une mémoire privée ?

Lors de l'exécution d'une tâche :

• Les variables peuvent être déclarées dans un espace mémoire partagé, c'est à dire que tous les threads ont accès à cet espace mémoire, ces variables sont donc partagées.

• Les variables peuvent être déclarées dans un espace mémoire propre à chaque thread, ces variables sont privées.



3.4 Différence avec MPI

MPI (**M**essage **P**assing **I**nterface) est utilisé sur des machines multiprocesseurs à mémoire distribuée (DMP : Distributed Memory Programming), c-à-d. que la mémoire est répartie sur plusieurs nœuds reliés par un réseau de communication. L'échange de données se fait par "passage de message".

OpenMP (**Open M**ulti-**P**rocessing) est utilisé sur des machines multiprocesseurs à mémoire partagée (SMP : Shared Memory Programming). OpenMP n'est utilisable que sur un seul nœuds de calcul.

3.5 Les avantages d'OpenMP

— Simple à implémenter.
— Les communications sont à la charge du compilateur (communications implicites) ce qui rend son utilisation beaucoup plus facile que MPI.
— Lors de la parallélisation, seules les régions parallèles sont spécifiées, les régions séquentielles restent inchangées.
— Un même programme peut contenir plusieurs régions parallèles et séquentielles.

3.6 Compilation

Pour compiler un programme parallélisé en OpenMP il faut définir le nombre de threads avec la variable d'environnement `OMP_NUM_THREADS`. Par exemple pour utiliser 4 threads :

```
$ export OMP_NUM_THREADS=4
```

Compilation avec GNU :

programme en C :

```
$ gcc -fopenmp programme.c
```

programme en C++ :

```
$ g++ -fopenmp programme.cpp
```

programme en Fortran :

```
$ gfortran -fopenmp programme.f
```

Compilation avec Intel :

programme en C :

```
$ icc -openmp programme.c
```

programme en C++ :

```
$ icpc -openmp programme.cpp
```

programme en Fortran :

```
$ ifort -openmp programme.f
```

4 Directives OpenMP

Pour utiliser OpenMP il faut inclure le module `OMP_LIB` pour Fortran et le fichier d'inclusion `omp.h` pour le C/C++.

programme en C/C++ :

```
#include "omp.h"
```

programme en Fortran90 :

```
!$ use OMP_LIB
```

Remarques :

1) Les instructions OpenMP en Fortran 77 commencent par : `*$` ou `c$`.

2) Attention à ne pas mettre d'espace entre les caractères `*,c,!` et `$` et à mettre un espace entre $ et `use` (pour `!$ use OMP_LIB`)

3) Les instructions OpenMP sont ignorées par le compilateur si le programme est compilé sans faire appel à l'option OpenMP (`-openmp` pour intel et `-fopenmp` pour GNU)

4) Contrairement au C/C++, Fortran ne fait pas la diférence entre majuscule et miniscule

4.1 Quelques fonctions OpenMP

`omp_get_num_threads()` : retourne le nombre total de threads utilisés
`omp_set_num_threads(int)` : spécifie un nombre de thread dans une région parallèle
`omp_get_thread_num()` : retourne le numéro du thread courant
`omp_in_parallel()` : retourne `F` en séquentiel et `T` en parallèle
`omp_get_num_procs()` : retourne le nombre de threads disponibles sur la machine
`omp_get_wtime()` : retoune le temps écoulé d'un certain point

Exemple d'utilisation des fonctions OpenMP en Annexe 1

4.2 Définir une région parallèle : directive parallel

Les fonctions OpenMP se trouvent toujours dans une région parallèle. On peut définir plusieurs régions parallèles dans un programme à l'aide de la directive suivante :

programme en C/C++ :

```
#pragma omp parallel
 {
   ....
 }
```

programme en Fortran90 :

```
!$OMP PARALLEL
  ....
  ....
!$OMP END PARALLEL
```

Remarque :
Attention à ne pas mettre d'espace entre `!$` et `OMP`

exemple 1 : programme en Fortran90

```
1  program hello_world
2
3    !$ use OMP_LIB
4    implicit none
5
6    !$OMP PARALLEL
7    write (*,*) "hello world !"
8    !$OMP END PARALLEL
9
10 end program hello_world
```

```
$ gfortran -fopenmp
    01_hello_world.f90
$ export OMP_NUM_THREADS=4
$ ./a.out
hello world !
hello world !
hello world !
hello world !
```

exemple 1 : programme en C++

```
1   #include <omp.h>
2   #include <cstdio>
3
4   main ()
5   {
6     # pragma omp parallel
7     {
8       printf("hello world ! \n");
9     }
10  }
```

```
$ gfortran -fopenmp
    01_hello_world.C
$ export OMP_NUM_THREADS=4
$ ./a.out
hello world !
hello world !
hello world !
hello world !
```

4.3 Variables privées et partagées

4.3.1 Clauses private() et shared()

Les clauses `private()` et `shared()` permettent de définir si la variable est privée ou partagée. Par défaut, toutes les variables sont partagée.

Dans cet exemple, la valeur finale de y change à chaque exécution du programme, elle dépend de l'ordre de l'exécution de chaque thread qui est complètement aléatoire.

exemple 2 : programme en Fortran90

```
program var_prive_partage

        !$ use OMP_LIB
        implicit none

        integer :: x=1, y=10
        integer :: num_thread

        print *, " Dans la region sequentielle, x=", x, "y=", y
        print *, " Dans la region parallele:"

        !$OMP PARALLEL PRIVATE (x,num_thread), SHARED (y)
        num_thread = OMP_GET_THREAD_NUM()
        x= 2 + num_thread
        y= 2 + num_thread
        print *, "    avec le thread num",num_thread,"x=", x,"y=", y
        !$OMP END PARALLEL

        print *, " Dans la region sequentielle, x=", x, "y=", y

end program var_prive_partage
```

```
$ gfortran -fopenmp 02_var_prive_partage.f90
$ export OMP_NUM_THREADS=4
$ ./a.out
Dans la region sequentielle, x=1  y=10
Dans la region parallele:
  avec le thread num  2   x=4   y=4
  avec le thread num  3   x=5   y=4
  avec le thread num  1   x=3   y=4
  avec le thread num  0   x=2   y=2
Dans la region sequentielle, x=1  y=2
```

exemple 2 : programme en C++

```
#include <iostream>
#include <cstdio>
#include "omp.h"
using namespace std;


main ()
{
  int x=1, y=10;
  int num_thread;
  cout << "Dans la region sequentielle x="<< x <<" y="<< y <<endl<<endl;
  cout << "Dans le region parallelle:" << endl;

  # pragma omp parallel private (x,num_thread) shared (y)
  {
    num_thread = omp_get_thread_num();
    x= 2 + num_thread;
    y= 2 + num_thread;
    printf("avec le thread %d x=%d y=%d \n", num_thread, x, y);
  }

  cout<<"Dans la region sequentielle x=" << x << " y=" <<y<< endl<< endl;
  return 0;
}
```

```
  $ g++ -fopenmp 02_var_prive_partage.C
  $ export OMP_NUM_THREADS=4
  $ ./a.out
Dans la region sequentielle x=1    y=10
Dans le region parallelle:
avec le thread 2 x=4 y=4
avec le thread 3 x=5 y=5
avec le thread 0 x=2 y=2
avec le thread 1 x=3 y=3
Dans la region sequentielle x=1    y=5
```

4.3.2 Clause firstprivate()

Quand une variable est déclarée privée sa valeur n'est pas définie à l'entrée d'une région parallèle.

Pour garder la dernière valeur affectée à une variable avant d'entrer dans la région parallèle il faut utiliser la clause : `firstprivate`

exemple 3 : programme en Fortran90

```
1  program var_parallele
2    !$use OMP_LIB
3    implicite none
4    integer :: x
5    x=10
6    !$ OMP PARALLEL PRIVATE (x)
7    print *, "x=", x
8    !$ OMP END PARALLEL
9  end program var_parallel
```

```
$ export OMP_NUM_THREADS=4
$ gfortran -fopenmp
  03.var_parallele.f90
$ ./a.out
 x=            0
 x=        32527
 x=            0
 x=            0
```

Lorsqu'une variable privée est initialisée en dehors d'une région parallèle, sa valeur n'est pas prise en compte dans la région parallèle.

la clause `firstprivate(a)` permet d'avoir la dernière valeur de la variable `a` avant d'entrer dans la région parallèle. Cette variable sera privée.

```
1  program var_parallele
2    !$use OMP_LIB
3    implicite none
4    integer :: x
5    x=10
6    !$ OMP PARALLEL FIRSTPRIVATE(x)
7    print *, "x=", x
8    !$ OMP END PARALLEL
9  end program var_parallel
```

```
$ export OMP_NUM_THREADS=4
$ gfortran -fopenmp
  03_var_parallele.f90
$ ./a.out
 x=           10
 x=           10
 x=           10
 x=           10
```

exemple 3 : programme en C++

```
1  #include <iostream>
2  #include <cstdio>
3  #include "omp.h"
4  using namespace std;
5  int main ()
6  {
7    int x=10;
8    # pragma omp parallel private (x)
9    {
10     printf("x=%d\n",x);
11   }
12 }
```

```
$ export OMP_NUM_THREADS=4
$ g++ -fopenmp 03_var_parallele.C
$ ./a.out
 x=            0
 x=           57
 x=            0
 x=            0
```

```cpp
#include <iostream>
#include <cstdio>
#include "omp.h"
using namespace std;
int main ()
{
  int x=10;
  # pragma omp parallel
    firstprivate(x)
  {
    printf("x=%d\n",x);
  }
}
```

```
$ export OMP_NUM_THREADS=4
$ g++ -fopenmp 03_var_parallele.C
$ ./a.out
 x=          10
 x=          10
 x=          10
 x=          10
```

4.3.3 Clause default()

Par défaut toutes les variables sont partagée. Le statut par défaut des variables peut être changé avec la clause `default()` cette clause prend comme argument : `private, shared, firstprivate` ou `none` en Fortran et `shared` ou `none` en C/C++.

`default(none)` permet à l'utilisateur de spécifier le statut de toutes les variables utilisées dans la région parallèle.

exemple 4 : programme en Fortran90

```fortran
program var_prive_partage

        !$ use OMP_LIB
        implicit none
        integer :: x=1, y=10
        integer :: num_thread

        print *, " Dans la region sequentielle, x=", x, "y=", y
        print *, " Dans la region parallele:"

        !$ OMP PARALLEL DEFAULT (none) PRIVATE(x,y,num_thread)
        num_thread = OMP_GET_THREAD_NUM()
        x= 2 + num_thread
        y= 2 + num_thread
        print *, "    avec le thread num",num_thread,"x=", x,"y=", y
        !$OMP END PARALLEL

        print *, " Dans la region sequentielle, x=", x, "y=", y

end program var_prive_partage
```

```
  $ g++ -fopenmp 04_default.f90
  $ export NUM_THREADS=4
  $ ./a.out
  Dans la region sequentielle, x=1  y=10
  Dans la region parallele:
    avec le thread num  0   x=2   y=2
    avec le thread num  2   x=4   y=4
    avec le thread num  1   x=3   y=3
    avec le thread num  3   x=5   y=5
  Dans la region sequentielle, x=1  y=10
```

exemple 4 : programme en C++

```
#include <iostream>
#include <cstdio>
#include "omp.h"
using namespace std;

main ()
{
  int x=1, y=10;
  int num_thread;

  cout << "Dans la region sequentielle x="<< x << " y=" << y <<endl<<endl;
  cout << "Dans le region parallelle:" << endl;

  # pragma omp parallel default (none) private (num_thread,x,y)
  {
    num_thread = omp_get_thread_num();
    x= 2 + num_thread;
    y= 2 + num_thread;
    printf("avec le thread %d x=%d y=%d \n", num_thread, x, y);

  }

  cout <<endl<<"Dans la region sequentielle x="<< x <<"y="<< y<<endl<<endl;

  return 0;
}
```

```
$ g++ -fopenmp 04_default.C
$ export NUM_THREADS=4
$ ./a.out
Dans la region sequentielle, x=1  y=10
Dans la region parallele:
  avec le thread num  0   x=2   y=2
  avec le thread num  2   x=4   y=4
  avec le thread num  1   x=3   y=3
  avec le thread num  3   x=5   y=5
Dans la region sequentielle, x=1  y=10
```

4.3.4 Clause lastprivate()

La clause `lastprivate()` permet de donner la dernière valeur d'une variable dans une région parallèle, elle n'est utilisable qu'avec les directives : `do` / `for` et `section`.

Comme ces directives sont expliquées dans la suite de ce cours, l'exemple d'utilisation de la clause `lastprivate` est donné en annexe page II

4.4 Exécution exclusive : directives master et single

Dans une région parallèle il est possible d'exécuter une partie du code avec un seul thread avec les directives `master` ou `single`

4.4.1 Directive master

Le code est exécuté avec le thread maître : thread numéro 0

programme en C/C++ :

```
#pragma omp master
{
 ....
}
```

programme en Fortran90 :

```
!$OMP MASTER
....
....
!$OMP END MASTER
```

exemple 5 : programme en Fortran90

```
1  program master
2    !$ use OMP_LIB
3    implicit none
4    integer thread_num
5    !$OMP PARALLEL PRIVATE (thread_num)
6    !$ thread_num=OMP_GET_THREAD_NUM();
7    write(*,*) "hello depuis le thread numero", thread_num
8    !$OMP MASTER
9    write(*,*) "special hello depuis le thread maitre, thread :", thread_num
10   !$OMP END MASTER
11   !$OMP END PARALLEL
12 end program master
```

```
$ export OMP_NUM_THREADS=4
$ gfortran -fopenmp 05_master.f90
$ ./a.out
 hello depuis le thread numero           1
 hello depuis le thread numero           0
 special hello depuis le thread maitre, thread :           0
 hello depuis le thread numero           3
 hello depuis le thread numero           2
```

exemple 5 : programme en C++

```
1  #include <cstdio>
2  #include "omp.h"
3  using namespace std;
4  main ()
5  {
6    int thread_num;
7    # pragma omp parallel private (thread_num)
8    {
9      thread_num = omp_get_thread_num();
10     printf("hello depuis le thread numero %d \n", thread_num);
11     # pragma omp master
12     {
13        printf("special hello depuis le thread maitre, thread : %d \n",
       thread_num);
14     }
15   }
16   return 0;
17 }
```

```
$ export OMP_NUM_THREADS=4
$ g++ -fopenmp 05_master.C
$ ./a.out
hello depuis le thread numero 0
special hello depuis le thread maitre, thread : 0
hello depuis le thread numero 2
hello depuis le thread numero 3
hello depuis le thread numero 1
```

4.4.2 Directive single

Le code est exécuté par le premier thread arrivé à la directive single. Le code poursuit son exécution uniquement lorsque tous les threads arrivent à la fin de la directive `single`.

programme en C/C++ :

```
#pragma parallel single
{
 ....
}
```

programme en Fortran90 :

```
!$OMP SINGLE
....
....
!$OMP END SINGLE
```

exemple 6_1 : programme en Fortran90

```
1  program single
2   !$use OMP_LIB
3   integer :: a, rang
4   !$OMP PARALLEL PRIVATE (a,rang)
5   rang=OMP_GET_THREAD_NUM()
6   a=1
7   !$OMP SINGLE
8   a=2
9   !$OMP END SINGLE
10  write(*,*) "thread numero:", &
11    rang,"a=",a
12  !$OMP END PARALLEL
13 end program single
```

```
$ export OMP_NUM_THREADS=4
$ gfortran -fopenmp 06_1_single.f90
$ ./a.out
 thread numero:       0 a=       1
 thread numero:       2 a=       1
 thread numero:       1 a=       1
 thread numero:       3 a=       2
```

exemple 6_1 : programme en C++

```
1  #include <cstdio>
2  #include "omp.h"
3  using namespace std;
4  main ()
5  {
6   int a, rang;
7    # pragma omp parallel private
       (a,rang)
8    {
9    rang = omp_get_thread_num();
10   a=1;
11   # pragma omp single
12     {
13       a=2;
14     }
15     printf("thread numero: %d,
       a=%d \n", rang,a);
16   }
17     return 0;
18 }
```

```
$ export OMP_NUM_THREADS=4
$ g++ -fopenmp 06_1_single.C
$ ./a.out
 thread numero: 0, a=1
 thread numero: 1, a=2
 thread numero: 2, a=1
 thread numero: 3, a=1
```

Dans cet exemple la valeur de `a` change pour un seul thread seulement, pour que cette valeur soit transmise à tous les autres threads il faut utiliser la clause `copyprivate`.

4.4.3 Clause copyprivate()

Cette clause n'est utilisable qu'à la fin de la directive single. Elle permet de transmettre à tous les threads la nouvelle valeur d'une variable privée affectée par un seul thread.

programme en C/C++ :

```
#pragma omp single copyprivate(x)
{
 ....
}
```

programme en Fortran90 :

```
!$OMP SINGLE
....
....
!$OMP END SINGLE COPYPRIVATE (x)
```

exemple 6_2 : programme en Fortran90

```
1  program single
2   !$use OMP_LIB
3   integer :: a, rang
4   !$OMP PARALLEL PRIVATE (a,rang)
5   rang=OMP_GET_THREAD_NUM()
6   a=1
7   !$OMP SINGLE
8   a=2
9   !$OMP END SINGLE COPYPRIVATE(a)
10  write(*,*) "thread numero:", &
11    rang,"a=",a
12  !$OMP END PARALLEL
13 end program single
```

```
$ export OMP_NUM_THREADS=4
$ gfortran -fopenmp 06_2_single.f90
$ ./a.out
 thread numero:       2 a=       2
 thread numero:       0 a=       2
 thread numero:       3 a=       2
 thread numero:       1 a=       2
```

exemple 6_2 : programme en C++

```
1  #include <cstdio>
2  #include "omp.h"
3  using namespace std;
4  main ()
5  {
6   int a, rang;
7    # pragma omp parallel private
       (a,rang)
8    {
9    rang = omp_get_thread_num();
10   a=1;
11   # pragma omp single
       copyprivate(a)
12     {
13       a=2;
14     }
15     printf("thread numero: %d,
       a=%d \n", rang,a);
16   }
17     return 0;
18 }
```

```
$ export OMP_NUM_THREADS=4
$ g++ -fopenmp 06_2_single.C
$ ./a.out
thread numero: 3, a=2
thread numero: 0, a=2
thread numero: 2, a=2
thread numero: 1, a=2
```

4.5 Parallélisation de boucles : directive do / for

La parallélisation de boucles en OpenMP se fait par la directive `DO` en Fortran et `for` en C/C++.

Les boucles `do while` ne sont pas parallélisée avec OpenMP.

Plusieurs directives `DO` ou `for` peuvent être mise dans une région parallèle.

La boucle `do/for` vient immédiatement après la directive.

programme en C/C++ :

```
# pragma omp for
    ....
```

programme en Fortran90 :

```
!$OMP DO
  ....
!$OMP END DO
```

La directive `omp parallel do/for` est la fusion des deux directives `parallel` et `do/for`

exemple 7 : programme en Fortran90

```
1  program initialisation_matrice
2    !$ use OMP_LIB
3    implicit none
4    real*8, dimension (:,:), allocatable :: A
5    integer :: i,j,N,M, t1, t2, ir
6    real    :: cpu_1, cpu_2, t_cpus, t_total
7
8    N=50000
9    M=50000
10   allocate (A(M,N))
11
12   call cpu_time(time=cpu_1)
13   call system_clock(count=t1, count_rate=ir)
14
15   !$OMP PARALLEL DO PRIVATE(i,j)
16    do j=1,N
17      do i=1,M
18        A(i,j)=0.d0
19      enddo
20     enddo
21   !$OMP END PARALLEL DO
22
23   call system_clock(count=t2, count_rate=ir)
24   t_total=real(t2 - t1,kind=8)/real(ir,kind=8)
25
26   call cpu_time(time=cpu_2)
27   t_cpus=cpu_2-cpu_1
28
29   write(*,*)"temps reel    :",t_total      !affiche le temps reel
30   write(*,*)"temps CPU     :",t_cpus      !affiche le temps CPU
31
32   deallocate (A)
33 end program initialisation_matrice
```

Compilation avec GNU en séquentiel puis en parallèle

```
$ gfortran 07_initialisation_matrice_parallel.f90
$ ./a.out
temps reel     :    9.5970001
temps CPU      :    9.5975399

$ gfortran -fopenmp 07_initialisation_matrice.f90
$ ./a.out
temps reel     :    2.8099999
temps CPU      :    11.243291
```

exemple 7 : programme en C++

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <sys/time.h>
#include <omp.h>
using namespace std;
int main()
{
    long double **A;
    int N=50000;
    int M=50000;
    // essayer d'allouer de la memoire
    try
    {
        // dimension 1
        A = new long double*[M];
        // initialisations des pointeurs a 0
        fill_n (A, M, static_cast<long double*>(0));
        // dimansion 2
        for (int i=0; i<M; i++)
        {
            A[i] = new long double [N];
        }
        }
        // en cas d'erreur d'allocation desaloue tout ce qui a ete aloue
        catch (const bad_alloc &)
    {
        for (int i=0; i<M; i++)
        {
            delete []A[i];
        }
        delete []A;
        cerr << " ERREUR : Probleme d'allocation de memoire de la matrice
    A" << endl;
        exit;
    }
    // initialisation de la matrice A :
    double t0 = omp_get_wtime();
    double t0_cpu = clock();

    # pragma omp parallel for
    for (int i=0; i<M; i++)
    {
        for (int j=0; j<N; j++)
        {
            A[i][j]=0.0;
        }
    }
    double t1 = omp_get_wtime();
    double t1_cpu = clock();

    double temps_reel=t1-t0;
    double temps_CPU=(t1_cpu-t0_cpu)/CLOCKS_PER_SEC;

    cout << " temps reel : " << temps_reel << endl;
    cout << " temps CPU : " << temps_CPU << endl;

    // desalocation de la memoire
    for (int i=0; i<M; i++)
    {
        delete []A[i];
    }
    delete []A;
}
```

Compilation avec GNU en séquentiel puis en parallèle

```
$ g++ -fopenmp 07_initialisation_matrice_parallel.C
$ export OMP_NUM_THREADS=1
$ ./a.out
  temps reel : 24.6523
  temps CPU : 24.66

$ export OMP_NUM_THREADS=4
$ ./a.out
  temps reel : 7.33658
  temps CPU : 28.72
```

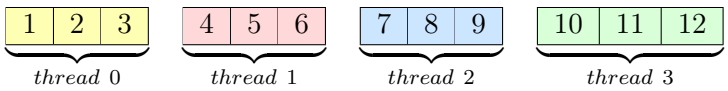
4.5.1 Clause schedule()

Les itérations sont réparties automatiquement par le compilateur sur chaque thread. Cette répartition peut être contrôlée avec la clause `schedule()`. Cette clause prend la forme suivante :

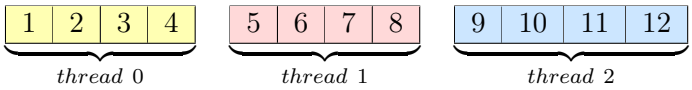
- `schedule(type,N)` :
  - `type` : le mode de répartition des paquets d'itérations : `static, dynamic, guided` ou `runtime`
  - `N` : entier qui détermine le nombre d'itérations dans un paquet (facultatif, prend la valeur 1 s'il n'est pas spécifié)

On veut paralléliser une boucle de 12 itérations sur 4 threads.

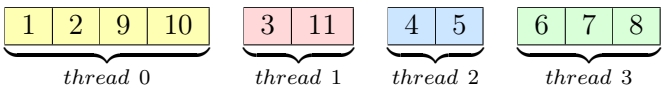
• `schedule(static)` : les itérations sont réparties sur chaque thread par paquets en proportion égale, sans accorder de priorité aux threads.



Pour définir un nombre d'itération sur chaque paquet, il faut ajouter une valeur à `N`, exemple pour 4 itérations par paquet : `schedule(static,4)`



• `schedule(dynamique)` : Les paquets d'itérations sont distribués sur les threads les plus rapides. A chaque fois qu'un thread termine un paquet d'itérations il reçoit directement un autre paquet.



• `guided` : la taille des paquets d'itérations attribués à chaque thread décroit exponentiellement, chaque paquet est attribué au thread le plus rapide.

• `runtime` : permet de de choisir le mode de répartition des itérations avec la variable d'environnement `OMP_SCHEDULE`

```
$ export OMP_SCHEDULE="static,4"
```

4.5.2 Clause ordred()

Très utile pour le débogage, la clause `ordered()` permet l'exécution de la boucle dans l'ordre des indices.

exemple 8 : programme en Fortran90

```fortran
program schedule_ordered

  !$ use OMP_LIB
  implicit none

  real*8, dimension (:,:), allocatable :: A
  integer :: i, j, N, M
  integer :: num_du_thread

  N=12
  M=12

  allocate (A(M,N))

  !$OMP PARALLEL DEFAULT (PRIVATE) SHARED(M,N)
  !$OMP DO SCHEDULE (RUNTIME) ORDERED
  do j=1,N
    do i=1,M
      A(i,j)=0.d0
    enddo
    !$ num_du_thread=OMP_GET_THREAD_NUM()
    !$OMP ORDERED
    write(*,*) "j=",j,"id thread :", num_du_thread
    !$OMP END ORDERED
  enddo
  !$OMP END DO
  !$OMP END PARALLEL

  deallocate (A)

end program schedule_ordered
```

exemple 8 : programme en C++

```cpp
#include <omp.h>
#include <cstdio>
#include <iostream>
using namespace std;

int main()
{
  int const N=12, M=12;
  long double A[N][M];
  int i,j;
  int num_du_thread;

  #pragma omp parallel for ordered schedule (runtime) private
    (i,j,num_du_thread)
  for (i=0; i<M; i++)
  {
    for (j=0; j<N; j++)
    {
      A[i][j]=0.0;
    }
    num_du_thread=omp_get_thread_num();
    # pragma omp ordered
    printf("i=%d  id thread :%d \n", i, num_du_thread);
  }
}
```

Compilation avec Intel

```
$ export OMP_NUM_THREADS=4
$ export OMP_SCHEDULE="static"
$ icpc -o intel -openmp
      08_schedule_ordered.C
$ ./intel
 i=0  id thread : 0
 i=1  id thread : 0
 i=2  id thread : 0
 i=3  id thread : 1
 i=4  id thread : 1
 i=5  id thread : 1
 i=6  id thread : 2
 i=7  id thread : 2
 i=8  id thread : 2
 i=9  id thread : 3
 i=10 id thread : 3
 i=11 id thread : 3
```

Compilation avec GNU

```
$ export OMP_NUM_THREADS=4
$ export OMP_SCHEDULE="static"
$ g++ -o gnu -fopenmp
     08_schedule_ordered.C
$ ./gnu
 i=0  id thread : 0
 i=1  id thread : 1
 i=2  id thread : 2
 i=3  id thread : 3
 i=4  id thread : 0
 i=5  id thread : 1
 i=6  id thread : 2
 i=7  id thread : 3
 i=8  id thread : 0
 i=9  id thread : 1
 i=10 id thread : 2
 i=11 id thread : 3
```

On peut changer la valeur de la variable OMP_SCHEDULE et ré-exécuter le programme :

```
$ export OMP_SCHEDULE="static,2"
$ ./intel
 i=0 id thread : 0
 i=1 id thread : 0
 i=2 id thread : 1
 i=3 id thread : 1
 i=4 id thread : 2
 i=5 id thread : 2
 i=6 id thread : 3
 i=7 id thread : 3
 i=8 id thread : 0
 i=9 id thread : 0
 i=10 id thread : 1
 i=11 id thread : 1
```

```
$ export OMP_SCHEDULE="static,2"
$ ./gnu
 i=0 id thread : 0
 i=1 id thread : 0
 i=2 id thread : 1
 i=3 id thread : 1
 i=4 id thread : 2
 i=5 id thread : 2
 i=6 id thread : 3
 i=7 id thread : 3
 i=8 id thread : 0
 i=9 id thread : 0
 i=10 id thread : 1
 i=11 id thread : 1
```

```
$ export OMP_SCHEDULE="dynamic,2"
$ ./intel
 i=0 id thread : 0
 i=1 id thread : 0
 i=2 id thread : 2
 i=3 id thread : 2
 i=4 id thread : 0
 i=5 id thread : 0
 i=6 id thread : 2
 i=7 id thread : 2
 i=8 id thread : 0
 i=9 id thread : 0
 i=10 id thread : 2
 i=11 id thread : 2
```

```
$ export OMP_SCHEDULE="dynamic,2"
$ ./gnu
 i=0 id thread : 2
 i=1 id thread : 2
 i=2 id thread : 3
 i=3 id thread : 3
 i=4 id thread : 0
 i=5 id thread : 0
 i=6 id thread : 1
 i=7 id thread : 1
 i=8 id thread : 2
 i=9 id thread : 2
 i=10 id thread : 3
 i=11 id thread : 3
```

4.6 Barrière de synchronisation

A la fin de certaines directives (directive `do` par exemple) le programme ne peut continuer son exécution que si tous les threads ont terminé leur travail. Par exemple pour afficher un résultat obtenu avec une boucle, il faut que tous les threads terminent le calcul pour afficher le résultat final. Il y a deux types de barrières de synchronisation :

Barrière de synchronisation implicite, qu'on trouve dans les directives :

- `parallel`
- `do / for`
- `ordered`
- `section`
- `critical`
- `single`

Barrière de synchronisation explicite : directive `barrier`

4.6.1 Directive barrier

exemple 9 : programme en Fortran90

```
1  program barrier
2    !$ use OMP_LIB
3    implicit none
4    integer thread_num
5
6    !$OMP PARALLEL PRIVATE (thread_num)
7    thread_num=OMP_GET_THREAD_NUM();
8    !$OMP SINGLE
9    write(*,*) "single : hello depuis le thread numero", thread_num
10   !$OMP END SINGLE
11   write(*,*) "region parallel : hello depuis le thread numero", thread_num
12
13   !$OMP MASTER
14   write(*,*) "master : hello depuis le thread numero", thread_num
15   !$OMP END MASTER
16   !$OMP END PARALLEL
17
18 end program barrier
```

```
  $ ifort -openmp 09_barrier.f90
  $ export OMP_NUM_THREADS=4
  $ ./a.out
 single : hello depuis le thread numero           1
 region parallel : hello depuis le thread numero           2
 region parallel : hello depuis le thread numero           0
 master : hello depuis le thread numero           0
 region parallel : hello depuis le thread numero           3
 region parallel : hello depuis le thread numero           1
```

On remarque dans cet exemple que l'affichage à l'écran ne se fait pas dans l'ordre, le thread le plus rapide affiche le message en premier. L'ordre de l'affichage est aléatoire à chaque exécution du programme.

En ajoutant la directive `!$OMP BARRIER` à la ligne 12 le programme va s'exécuter dans l'ordre de son écriture :

```
  $ ifort -openmp 09_barrier.f90
  $ ./a.out
 single : hello depuis le thread numero           1
 region parallel : hello depuis le thread numero           2
 region parallel : hello depuis le thread numero           3
 region parallel : hello depuis le thread numero           1
 region parallel : hello depuis le thread numero           0
 master : hello depuis le thread numero           0
```

Il n'y a pas d'intérêt à mettre la directive `!$OMP BARRIER` après `!$OMP END SINGLE` car cette dernière contient une barrière de synchronisation implicite.

4.6.2 Clause nowait

Dans certains cas, imposer une barrière implicite peut s'avérer inutile et ralentit un peu le code. Pour éviter cette attente, il faut utiliser la clause `nowait` qui vient directement après la fin d'une directive contenant une barrière de synchronisation implicite (à part la directive `parallel` qui contient une barrière implicite qui ne peut être levée).

programme en C/C++ :

```
# pragma omp for nowait
```

programme en Fortran90 :

```
!$OMP DO
  ....
!$OMP END DO NOWAIT
```

On peut vérifier avec l'exemple précédent en rajoutant `nowait` à la fin de la directive single :

```
  $ ifort -openmp 09_barrier.f90
  $ ./a.out
 region parallel : hello depuis le thread numero           0
 master : hello depuis le thread numero          0
 region parallel : hello depuis le thread numero           2
 region parallel : hello depuis le thread numero           3
 single : hello depuis le thread numero          1
 region parallel : hello depuis le thread numero           1
```

exemple 9 : programme en C++

```
1  #include <cstdio>
2  #include <omp.h>
3  using namespace std;
4
5  int main ()
6  {
7    # pragma omp parallel
8    {
9      int thread_num = omp_get_thread_num();
10     # pragma omp single nowait
11     {
12       printf("single : hello depuis le thread numero %d \n", thread_num);
13     }
14     printf("region parallel : hello depuis le thread numero %d \n",
       thread_num);
15     # pragma omp barrier
16     # pragma omp master
17     {
18       printf("master : hello depuis le thread numero %d \n", thread_num);
19     }
20   }
21 }
```

4.7 Opérations cumulées sur une variable

OpenMP propose plusieurs solutions pour protéger la modification de variables partagées afin de cumuler des opérations sur des variables.

4.7.1 Clause reduction()

Cette clause est utilisée pour calculer une somme, un produit, un maximum... etc. de variables dont la valeur change avec les indices d'une boucle (par exemple) et chaque nouvelle valeur dépend de la valeur précédente. Cette clause prend la forme suivante :

programme en C/C++ :

```
# pragma omp for reduction(op:var)
```

programme en Fortran90 :

```
!$OMP DO REDUCTION(op:var)
  ....
!$OMP END DO
```

`reduction(operation:variable)`

— opération : l'opération de réduction utilisée (résumé dans les tableaux ci-dessous)
— variable : la variable sur quoi l'opération est effectuée

Opérations arithmétiques		
	Fortran	**C/C++**
sommation	+	+
soustraction	-	-
produit	*	*
division	/	non-utilisé

Opérations logiques		
	Fortran	**C/C++**
et	.and.	&&
ou	.or.	||
équivalence	.eqv.	non-utilisé
non-équivalence	.neqv.	non-utilisé

Fonctions intrinsèques		
	Fortran	**C/C++**
maximum	max	non-utilisé
minimum	min	non-utilisé
et binaire	iand	&
ou inclusif binaire	ior	|
ou exclusif binaire	ieor	$\wedge$

exemple 10 : programme en Fortran90

```
1  program reduction
2    use OMP_LIB
3    implicit none
4    integer :: i,M,a
5    M=10
6    a=2
7    !$OMP PARALLEL DO PRIVATE (i) FIRSTPRIVATE (M) REDUCTION (+:a)
8    do i=1,M
9       a=a+1
10   enddo
11   !$OMP END PARALLEL DO
12   write(*,*)"a final =",a
13 end program reduction
```

```
$ gfortran -fopenmp 10_reduction.f90
$ ./a.out
  a final =      12
```

exemple 10 : programme en C++

```
#include <cstdio>
#include "omp.h"
using namespace std;

main ()
{
        int M=10;
        int a=2;

        # pragma omp parallel
        {
                int thread_num=omp_get_thread_num();
                # pragma omp for firstprivate(M) reduction(+:a)
                for (int i=0; i<M; i++)
                {
                        a=a+1;
                }
        }
        printf("a final = %d \n",a);
}
```

```
$ g++ -fopenmp 10_reduction.C
$ ./a.out
  a final = 12
```

4.7.2 Directive atomic

Permet la mise à jour atomique[2] d'un emplacement mémoire. Cette directive ne prend en charge aucune clause et s'applique sur l'instruction qui vient juste après. Atomic a de meilleur performances que la directive critical. Elle prend la forme suivante :

programme en C/C++ :

```
# pragma omp atomic
 x (operation) = qqch
```

programme en Fortran90 :

```
!$OMP ATOMIC
  x = operation sur x
```

x : variable scalaire.

operation :

En fortran, l'une des opérations de réduction (voir tableaux § 4.7.1 pour Fortran)

En C/C++, l'une des opérations suivantes : $+, -, *, /, \&, \wedge, |, <<, >>$ on peut aussi utiliser : $x++, x--, ++x, --x$

Opérations arithmétiques		exemple d'utilisation
sommation	+	x++ , x+= , ++x
soustraction	-	x- - , x-= , - -x
produit	*	x*=
division	/	x/=

Remarques :

- La directive atomic ne peut être utilisée que pour des accès mémoire, elle ne peut inclure des appels de fonctions, les indices de tableaux, la surcharge d'opérateurs.

- A partir de OpenMP 4.0 (voir standard OpenMP 4.0) on peut utiliser en C/C++ :

`x = x (operation) qqch`

2. L'atomicité désigne une opération ou un ensemble d'opérations d'un programme qui s'exécutent entièrement sans pouvoir être interrompues avant la fin de leur déroulement. Une opération qui vérifie cette propriété est qualifiée d'«atomique», ce terme dérive de «atomos» qui signifie «que l'on ne peut diviser» (wiki)

exemple 11 : programme en Fortran90

```
program atomic
  !$ use OMP_LIB
  implicit none
  integer :: i,M,a
  M=10
  a=2
  !$OMP PARALLEL DO PRIVATE (i), SHARED (M,a)
  do i=1,M
     !$OMP ATOMIC
     a=a+1
  enddo
  !$OMP END PARALLEL DO
  write(*,*)"a final =",a
end program atomic
```

```
$ gfortran -fopenmp 11_atomic.f90
$ ./a.out
  a final =      12
```

exemple 11 : programme en C++

```
#include <omp.h>
#include <iostream>
#include <cstdio>
using namespace std;

int main ()
{
        int M=10, a=2;
        # pragma omp parallel
        {
                int num_thread=omp_get_thread_num();
                # pragma omp for //reduction (+:a)
                for (int i=0; i<M; i++)
                {
                        # pragma omp atomic
                        a+=1;
                        printf("thread num: %d a= %d \n", num_thread,a);
                }
        }
        cout << "a final =" << a << endl;
}
```

```
$ gfortran -fopenmp 11_atomic.C
$ ./a.out
  a final =      12
```

4.7.3 Directive critical

S'applique sur un bloc de code, celui-ci ne sera exécuté que par un seul thread à la fois, une fois qu'un thread a terminé un autre thread peut donc avoir accès à cette région critique.

`atomic` et `reduction` sont plus restrictives mais plus performantes que critical.

programme en C/C++ :

```
# pragma omp critical
{
 ....
}
```

programme en Fortran90 :

```
!$OMP CRITICAL
  ....
  ....
!$OMP END CRITICAL
```

exemple 12 : programme en Fortran90

```
1  program critical
2    !$ use OMP_LIB
3    implicit none
4    integer :: i,M,a
5    M=10
6    a=2
7    !$OMP PARALLEL PRIVATE (i) FIRSTPRIVATE (a,M)
8    !$OMP CRITICAL
9    call calcul(a,M)
10   !$OMP END CRITICAL
11   !$OMP SINGLE
12   write(*,*) "a final =",a
13   !$OMP END SINGLE
14   !$OMP END PARALLEL
15
16 end program critical
17
18 SUBROUTINE calcul(a,M)
19   implicit none
20   integer, intent(out)::a
21   integer ::M,i
22   do i=1,M
23     a=a+1
24   enddo
25 END SUBROUTINE calcul
```

```
$ gfortran -fopenmp 12_critical.f90
$ ./a.out
  a final =     12
```

4.8 Sections parallèles : directive sections

Une section parallèle n'est exécutée que par un seul et unique thread (tout comme `single`). Ces deux directives sont différentes car elle ne font pas tout à fait la même chose : la directive `sections` peut prendre les clauses `lastprivate` et `reduction` et la directive `single` est la seule à prendre la clause `copyprivate`.

programme en C/C++ :

```
# pragma omp sections
{
  # pragma omp section
     ....
  # pragma omp section
     ....
}
```

programme en Fortran90 :

```
!$OMP SECTIONS

  !$OMP SECTION
    ....
  !$OMP SECTION
    ....
!$OMP END SECTIONS
```

Remarques :

1) Plusieurs sections parallèles peuvent être déclarées au seins de la directive `sections`

2) `omp parallel sections` est la fusion entre les deux directives : `parallel` et `sections`

3) Tout comme la directive `parallel` , la barrière implicite de `omp parallel sections` ne peut pas être levée par la clause nowait

5 Performances obtenues avec OpenMP

Pour une bonne performance avec OpenMP il faut toujours s'assurer de :
- Paralléliser la boucle la plus externe et la plus coûteuse
- Utiliser `atomic` ou `reduction` à la place de `critical` quand c'est possible
- Supprimer les barrières de synchronisation quand elles sont inutiles
- Faire attention au statut des variables : privé, partagé, etc.
- Toujours mettre les indices de boucles en statut privé
- Minimiser les régions parallèles
- Utiliser les fusions `omp parallel do/for` ou `omp parallel sections` quand c'est possible

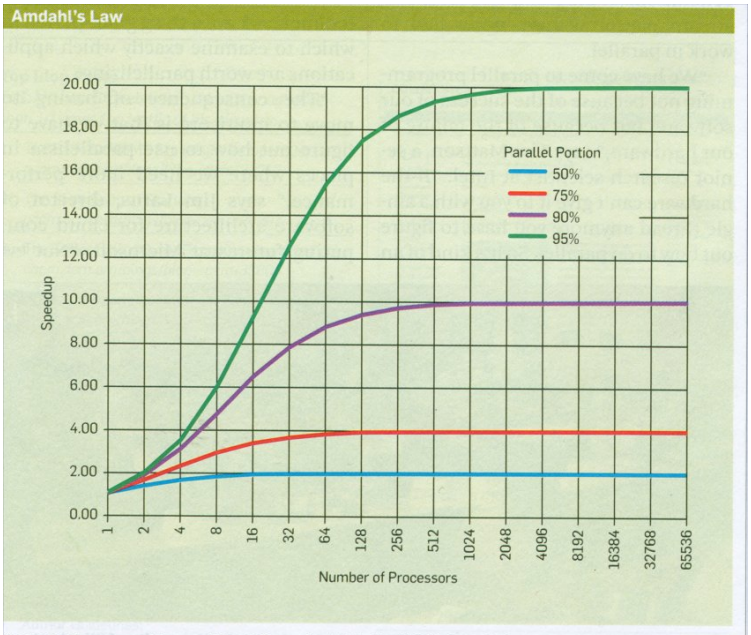
Les performances obtenues pour un code parallèle sont estimées par rapport à sa version séquentielle. L'accélération du code dépend du nombre de threads utilisés, de la machine où le code a été exécuté, du compilateur et de la bonne programmation.

Le gain de performance est exprimé par la loi d'Amdahl :

$$SpeedUp = \frac{T_{seq}}{T_{par}} = \frac{1}{(1 - frac) + \frac{frac}{N}}$$

T_{seq} : Le temps d'exécution du programme en séquentiel
T_{par} : Le temps d'exécution du programme en parallèle
$frac \in [0, 1]$ fraction du temps parallèle $= \frac{1}{T_{par}}$
N : nombre de threads

Ainsi, quand le nombre de threads tend vers l'infini, le SpeedUp devient constant



Named after computer architect Gene Amdahl, Amdahl's Law is frequently used in parallel programming to predict the theoretical maximum speedup using multiple processors.

6 Bibliographie

Le site officiel d'OpenMP : www.openmp.org

Standard OpenMP 3.1 : juillet 2011
(http ://www.openmp.org/mp-documents/OpenMP3.1.pdf)

Standard OpenMP 4.0 : juillet 2013
(http ://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)

Exemples OpenMP version 4.0.0 : novembre 2013
(http ://openmp.org/mp-documents/OpenMP4.0.0.Examples.pdf)

Exemples OpenMP version 4.0.1 : fevrier 2014
(http ://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf)

Parallel Programming in Fortran 95 using OpenMP : février 2002
(http ://www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf)

L'IDRIS : http ://www.idris.fr/data/cours/parallel/openmp/

Annexes

I. Exemple de l'utilisation de quelques fonctions OpenMP

```
1  program fonctions_openmp
2          !$ use OMP_LIB
3          implicit none
4          integer :: nbr_threads, nbr_de_coeurs, thread_num
5          logical :: parallel
6          real    :: t0, t1, t_total
7          integer :: i, a
8
9          t0=OMP_GET_WTIME()
10         nbr_de_coeurs=OMP_GET_NUM_PROCS()
11         parallel=OMP_IN_PARALLEL()
12         write (*,*) "Region parallel:", parallel
13
14         !$OMP PARALLEL
15         nbr_threads=OMP_GET_NUM_THREADS()
16         !$OMP END PARALLEL
17
18         write(*,*) "Nombre de coeurs disponibles sur la machine:",&
19                    nbr_de_coeurs
20         write(*,*) "Nombre de threads utilises:", nbr_threads
21
22         !$OMP PARALLEL PRIVATE (thread_num,parallel)
23         parallel=OMP_IN_PARALLEL()
24         thread_num=OMP_GET_THREAD_NUM()
25         write(*,*) "hello word !, depuis le thread numero :", &
26                 thread_num, "Region parallel:",parallel
27         !$OMP END PARALLEL
28
29         call OMP_SET_NUM_THREADS(3)
30         !$OMP PARALLEL private (thread_num)
31         thread_num=OMP_GET_THREAD_NUM()
32         write(*,*) "coucou depuis", thread_num
33         !$OMP END PARALLEL
34
35         call OMP_SET_NUM_THREADS(2)
36         !$OMP PARALLEL private (thread_num)
37         thread_num=OMP_GET_THREAD_NUM()
38         write(*,*) "bonjour depuis", thread_num
39         !$OMP END PARALLEL
40
41         t1=OMP_GET_WTIME()
42         t_total=t1-t0
43         write(*,*) "temps d'excecution du programme:",t_total
44
45 end program fonctions_openmp
```

```
$ gfortran -fopenmp fonctions_openmp.f90
$ export OMP_NUM_THREADS=4
$ ./a.out
  Region parallel: F
 Nombre de coeurs disponibles sur la machine:          40
 Nombre de threads utilises:           4
 hello word !, depuis le thread numero :           0 Region parallel: T
 hello word !, depuis le thread numero :           3 Region parallel: T
 hello word !, depuis le thread numero :           2 Region parallel: T
 hello word !, depuis le thread numero :           1 Region parallel: T
 coucou depuis           0
 coucou depuis           2
 coucou depuis           1
 bonjour depuis           0
 bonjour depuis           1
 temps d'excecution du programme:   0.0000000
```

II. Exemple de l'utilisation de la clause lastprivate

exemple : programme en Fortran90

```
1  program lastprivate
2    !$ use OMP_LIB
3    implicit none
4    integer :: i,a
5    a=33
6    !$OMP PARALLEL
7    !$OMP SINGLE
8    write (*,*) "a avant le do=",a
9    !$OMP END SINGLE
10   !$OMP DO LASTPRIVATE(a)
11   do i=1,10
12           a=i
13   enddo
14   !$OMP END DO
15   !$OMP END PARALLEL
16   write (*,*) "a apres le do=",a
17 end program lastprivate
```

```
$ gfortran -fopenmp lastprivate
$ export OMP_NUM_THREADS=4
$ ./a.out
 a avant le do=           33
 a apres le do=           10
```

Sans la clause `lastprivate(a)`, la valeur de la variable `a` après la sortie de la boucle `do` change à chaque éxécution du programme.
Cette clause permet donc d'envoyer la valeur d'une variable calculée dans la boucle.

exemple : programme en C++

```
1  #include <iostream>
2  #include <cstdio>
3  #include "omp.h"
4  using namespace std ;
5
6  main ()
7  {
8   int i, a(33);
9   # pragma omp parallel
10  {
11   # pragma omp single
12   {
13     printf ("avant le for, a=%d
       \n",a);
14   }
15   # pragma omp for lastprivate (a)
16   for (i=0; i<10; i++)
17   {
18     a=i;
19   }
20  }
21  printf ("apres le for, a=%d\n",a);
22 }
```

```
$ gfortran -fopenmp lastprivate
$ export OMP_NUM_THREADS=4
$ ./a.out
 a avant le do=           33
 a apres le do=           10
```

Sans la clause `lastprivate(a)`, la valeur de la variable `a` après la sortie de la boucle `for` change à chaque éxécution du programme.
Cette clause permet donc d'envoyer la valeur d'une variable calculée dans la boucle.

Exercices

Exercice 1 : Equation de la chaleur 2D

On souhaite paralléliser un programme séquentiel pour la résolution de l'équation de la chaleur en 2D par la méthode des différences finies.

$$\frac{\partial u}{\partial t}u(x,y,t) = \alpha\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)u(x,y,t) \tag{1}$$

Les coordonées spatiale (x,y) et tomporelles (t) sont discrétisées uniformément tel que :

$$\begin{aligned} x_i &= x_0 + idx \quad , \quad 0 < i \leq M \quad , \quad dx = (x_{max} - x_0)/M \\ y_i &= x_0 + idy \quad , \quad 0 < i \leq N \quad , \quad dy = (y_{max} - y_0)/N \\ t &= t + dt \quad , \quad t_0 = 0 < t \leq t_{max} \quad , \quad dt = 0.5 \end{aligned}$$

$\alpha > 0$: coefficient de diffusivité thermique pour calculer les conditions CFL :

$$CFL_x = \alpha\frac{dt}{dx^2}$$
$$CFL_y = \alpha\frac{dt}{dy^2}$$

La solution de l'équation (1) est approchée par le shéma explicite suivant :

$$du_x(i,j) = \left[u^n_{i+1,j} - 2u^n_{ij} + u^n_{i-1,j}\right]$$
$$du_y(i,j) = \left[u^n_{i,j+1} - 2u^n_{ij} + u^n_{i,j-1}\right]$$

$$u^{n+1}_{ij} = u^n_{ij} + CFL_x\ du_x + CFL_y\ du_y \tag{2}$$

Conditions aux limites :

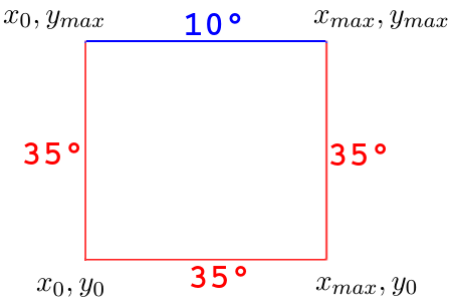


FIGURE 1 – shéma représentant la figure avec les conditions aux limites

1) Paralleliser ce code en utilisant OpenMP

2) Ecrire le nombre total des threads disponible sur la machine

3) Utiliser les fonctions OpenMP de sorte a ce que le code utilise toujours 4 threads (même si la variable OMP_NUM_THREADS a une valeur differente de 4)

4) Ecrire le code de tel sorte a ce qu'il soit executable en sequentiel et en parallel

```
program equation_chaleur_2D
      implicit none
      real(kind=8)       :: alpha=0.000127  ! Diffusivite thermique
                                             or = 1.27E10-4 (m2/s)
      integer, parameter :: m=10000,n=10000 ! nbr de points suivant x, y
      integer            :: x0=0, xmax=10   ! x0:x initial, xmax:x final
      integer            :: y0=0, ymax=10   ! y0:y initial, ymax:y final
      real               :: dx, dy          ! discretisation spatial
      real               :: t, tmax=10
      real               :: dt=0.5          ! discretisation temporel
      real(kind=8)       :: cfl_x, cfl_y    ! coefficients CFL
      real(kind=8), dimension(m+1,n+1)    :: u
      real(kind=8), dimension(m+1,n+1)    :: du_x, du_y
      integer            :: i,j
      integer            :: t1,t2,ir
      real               :: temps
      character (len = 10) :: resultat_u

      call system_clock(count=t1, count_rate=ir)

      dx=(xmax-x0)/real(m)
      write (*,*) "dx=", dx
      dy=(ymax-y0)/real(n)
      write (*,*) "dy=", dy

      cfl_x=alpha*dt/(dx*dx)
      cfl_y=alpha*dt/(dy*dy)

      ! initialisation de la matrice u
      do j=1,n+1
                do i=1,m+1
                        u(i,j)=0.0
                end do
      end do
      ! Conditions aux limites
      do i=2,m
               u(i,1)=35.0
               u(i,n)=35.0
      end do
      do j=1,n+1
               u(m,j)=35.0
               u(1,j)=10.0
      end do
      t=0.0
      do while (t<tmax)
               t=t+dt
               do j=2, n
                       do i=2, m
                               du_x(i,j)=u(i+1,j)-2.0*u(i,j)+u(i-1,j)
                               du_y(i,j)=u(i,j+1)-2.0*u(i,j)+u(i,j-1)
                        end do
               end do
               do j=2, n
                       do i=2, m
                             u(i,j)=u(i,j)+cfl_x*du_x(i,j)+cfl_y*du_y(i,j)
                       end do
               end do
       end do
      call system_clock(count=t2, count_rate=ir)
      temps=real(t2 - t1,kind=8)/real(ir,kind=8)
      write (*,*) "temps d'excecution du programme:",temps
      open (unit=10,file="resultat_u",action="write",status="new")
      write (unit=10,fmt=*) u

end program equation_chaleur_2D
```

Exercice 2 : Calcul de π

On souhaite paralléliser un programme séquentiel pour le calcul de π par la méthode des rectangles sur `n` intervalles avec un pas `h`

$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx$$

- Paralléliser le code séquentiel du calcul de PI en utilisant OpenMP

```
program pi
  !                    --
  !  But : calcul de || par la methode des rectangles (point milieu).
  !
  !                     / 1
  !                    |       4              --
  !                    |   ----------  dx = ||
  !                    |    1 + x**2
  !                   / 0

  implicit none

  integer, parameter :: n=100000000
  double precision   :: f, x, a, h
  double precision   :: Pi_calcule, Pi_reel, ecart
  integer            :: i, k
  integer            :: t1, t2, ir
  real               :: temps


  ! Fonction instruction a integrer
  f(a) = 4.0_16 / ( 1.0_16 + a*a )

  ! valeur reel de Pi avec 15 chiffres apres la virgule
  Pi_reel=3.141592653589793

  ! Longueur de l'intervalle d'integration
  h = 1.0_8 / real(n,kind=8)

  ! Temps elapsed de reference
  call system_clock(count=t1, count_rate=ir)

  do k=1,2
     ! Calcul de Pi
     Pi_calcule = 0.0_16
     do i = 1, n
        x = h * ( real(i,kind=16) - 0.5_16 )
        Pi_calcule = Pi_calcule + f(x)
     end do
     Pi_calcule = h * Pi_calcule

  enddo

  ! Temps final
  call system_clock(count=t2, count_rate=ir)
  temps=real(t2-t1)/real(ir)

  ! Ecart entre la valeur reelle et la valeur calculee de Pi.
  ecart = abs(Pi_reel - Pi_calcule)

  ! Impression du resultat.
  write (*,*) "Nombre d intervalles      : ",n
  write (*,*) "| Pi_reel - Pi_calcule |  : ",ecart
  write (*,*) "Temps reel                : ",temps

end program pi
```

Exercice 3 : Nombres premiers

On souhaite paralléliser un programme séquentiel qui compte le nombre des nombres premiers entre 1 et un entier fourni par l'utilisateur. Pour tester si un entier N est premier ou non, on vérifie s'il est divisible par les entiers plus petits.

- Écrire les directives et clauses OpenMP pour que la recherche des nombres premiers se fasse en parallèle.

```
1  program nombres_premiers
2
3          integer :: nombre, compteur, nombre_premier=0
4          integer :: i,j, reste, imax
5
6          write(*,*) "Entrez un nombre :"
7          read(*,*) nombre
8
9          do j=2,nombre
10                 compteur=1
11                 imax=floor(sqrt(real(j)))
12                 do i=2, imax
13                         if (mod(j,i)==0) then
14                                 compteur=0;
15                                 exit;
16                         end if
17                 end do
18                 nombre_premier=nombre_premier+compteur
19         end do
20
21         write (*,*) "dans", nombre, "il y a", nombre_premier, "nombres
               premiers"
22
23 end program nombres_premiers
```

```
$ ifort nombres_premiers.f90
$ a.out
 Entrez un nombre :
50
 dans           50 il y a           15 nombres premiers
```

Exercice 4 : Equation de la chaleur stationnaire (Programme séquentiel à paralléliser)

```
program equation_chaleur
    real                        :: diff
    real                        :: epsilon = 0.001
    integer                     :: i, j, iterations, iterations_print
    real                        :: valeur_initiale=0
    integer, parameter          :: m=1500, n=1500
    real, dimension (m+1,n+1)    :: u, w
    integer                     :: ir, t1, t2
    real                        :: temps
    character (len = 10) :: resultat_w
    call system_clock(count=t1, count_rate=ir)
!       conditions aux limites
!                    W = 0
!              +------------------+
!              |                  |
!     W = 100  |                  | W = 100
!              |                  |
!              +------------------+
!                     W = 100
!                       I = 1
!           [1][1]--------------[1][N]
!              |                  |
!       J = 1  |                  |  J = N
!              |                  |
!           [M][1]--------------[M][N]
!                       I = M
!    a l'equilibre La solution de l'equation de la chaleur est donnee par :
!       W[Central] = (1/4) * ( W[Nord] + W[Sud] + W[Est] + W[Ouest] )
!              +---------------------------+
!              |   |   |   | N |   |   |   |   N : Nord, S : Sud
!              +---------------------------+   O : Ouest, E : Est
!              |   |   | O | X | E |   |   |   X : W[Central]
!              +---------------------------+
!              |   |   |   | S |   |   |   |
!              +---------------------------+
!       Dans ce programme :
!       diff : norme du chgmt de la solution d'une iteration a la suivante
!       valeur_initiale : la moyenne des valeurs aux limites, utilise pour
!           initialiser les valeurs de la solution a l'interieu du domaine.
!       u(m,n) : la solution a l'iteration precedente
!       w(m,n) : la solution a la derniere iteration

    write(*,*) " Resolution de l'eq de la chaleur a l'etat stationnaire*"
    write(*,*) " sur une plaque rectangulaire de taille M*N=",M,N
    write(*,*) " *etat stationnaire : pas d'evolution temporelle"
    write(*,*) " "
    write(*,*) " les iterations seront repetees jusqu'a ce que le chgmt de"
    write(*,*) " temperature soit inferieur a epsilon, epsilon=",epsilon
    !$ write(*,*) "nbr de threads disponibles sur la machine :", nbr_procs
    !$ write(*,*) "nombre de threads utilises :", nbr_threads

!       Conditions aux limites :
!       -----------------------
    do i=2,m
        w(i,1)=100.0
    end do
    do i=2,m
        w(i,n)=100.0
    end do
    do j=1,n+1
        w(m,j)=100.0
    end do
    do j=1,n+1
        w(1,j)=0.0
    end do
```

```
!       Initialisation des valeurs a l'interieur du domaine (en prenant la
!       moyenne des valeurs aux limites)
!       ------------------------------------------------------------------
    do i=2,m
            valeur_initiale=valeur_initiale+w(i,1)+w(i,n)
!                +------------------+
!               | |                  | |
!    W(i,1)=100 | |                  | | W(i,n)= 100
!               | |                  | |
!                +------------------+
    end do
    do j=1,n+1
            valeur_initiale=valeur_initiale+w(m,j)+w(1,j)
!    W(n,j) = 0  +------------------+
!                +------------------+
!                |                   |
!                |                   |
!                |                   |
!   W(1,j) = 100 +------------------+
!                +------------------+
    end do
    valeur_initiale=valeur_initiale/real(2*m+2*n-4)
    write (*,*) "valeur_initiale=", valeur_initiale
    do i=2,m
            do j=2,n
                    w(i,j)=valeur_initiale
            end do
    end do
    iterations=0
    diff=epsilon
    do while (epsilon <= diff)
        ! enregistrer la solution precedente dans U
        do j=1,n+1
                do i=1,m+1
                        u(i,j)=w(i,j)
                end do
        end do
        ! calcul des nouvelles valeurs de W
        ! (W[Central] = (1/4) * (W[Nord] + W[Sud] + W[Est] + W[Ouest] )
        do j=2,n
            do i=2,m
                    w(i,j)=(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))*0.25
            end do
        end do
        diff=0.0
        my_diff=0.0
        do j=2,n
                do i=2,m
                        if (my_diff < abs(w(i,j)-u(i,j))) then
                                my_diff = abs(w(i,j)-u(i,j))
                        end if
                end do
                if (diff<my_diff) then
                        diff=my_diff
                end if
        end do
        iterations=iterations+1
        write(*,*) "iteration=", iterations, "diff=", diff
    end do
    open (unit=10,file="resultat_w",action="write",status="new")
    write (unit=10,fmt=*) w
    call system_clock(count=t2, count_rate=ir)
    temps=real(t2 - t1,kind=8)/real(ir,kind=8)
    ! Temps de calcul
     write(*,*) "Temps reel", temps
end program equation_chaleur
```

Solutions

Exercice 1 : Equation de la chaleur 2D

```
program equation_chaleur_2D

      implicit none
      real(kind=8)       :: alpha=0.000127  ! Diffusivite thermique
                                             or = 1.27E10-4 (m2/s)
      integer, parameter :: m=10000,n=10000 ! nbr de points suivant x, y
      integer            :: x0=0, xmax=10   ! x0:x initial, xmax:x final
      integer            :: y0=0, ymax=10   ! y0:y initial, ymax:y final
      real               :: dx, dy          ! discretisation spatial
      real               :: t, tmax=10
      real               :: dt=0.5          ! discretisation temporel
      real(kind=8)       :: cfl_x, cfl_y    ! coefficients CFL
      real(kind=8), dimension(m+1,n+1)    :: u
      real(kind=8), dimension(m+1,n+1)    :: du_x, du_y
      integer            :: i,j
      integer            :: t1,t2,ir
      real               :: temps
      character (len = 10) :: resultat_u

      call system_clock(count=t1, count_rate=ir)

      dx=(xmax-x0)/real(m)
      write (*,*) "dx=", dx
      dy=(ymax-y0)/real(n)
      write (*,*) "dy=", dy

      cfl_x=alpha*dt/(dx*dx)
      cfl_y=alpha*dt/(dy*dy)

      ! Conditions aux limites

      !$ nbr_de_coeurs=OMP_GET_NUM_PROCS()
      !$ write(*,*) "Nombre de coeurs disponibles sur la machine:",
          nbr_de_coeurs
      !$OMP PARALLEL PRIVATE (i,j)
      !$ call OMP_SET_NUM_THREADS(4)
      !$OMP DO
      do j=1,n+1
              do i=1,m+1
                      u(i,j)=0.0
              end do
      end do
      !$OMP END DO
      !$OMP DO
      do i=2,m
              u(i,1)=35.0
              u(i,n)=35.0
      end do
      !$OMP END DO nowait
      !$OMP DO
      do j=1,n+1
              u(m,j)=35.0
              u(1,j)=10.0
      end do
      !$OMP END DO
      !$OMP END PARALLEL
      t=0.0
      do while (t<tmax)
              t=t+dt
              !$OMP PARALLEL private (i,j)
              !$OMP DO
              do j=2, n
                      do i=2, m
                              du_x(i,j)=u(i+1,j)-2.0*u(i,j)+u(i-1,j)
                              du_y(i,j)=u(i,j+1)-2.0*u(i,j)+u(i,j-1)
```

```
64                          end do
65                  end do
66                  !$OMP END DO
67                  !$OMP DO
68                  do j=2, n
69                          do i=2, m
70                                  u(i,j)=u(i,j)+cfl_x*du_x(i,j)+cfl_y*du_y(i,j)
71                          end do
72                  end do
73                  !$OMP END DO
74                  !$OMP END PARALLEL
75          end do
76
77          call system_clock(count=t2, count_rate=ir)
78          temps=real(t2 - t1,kind=8)/real(ir,kind=8)
79          write (*,*) "temps d'excecution du programme:",temps
80          open (unit=10,file="resultat_u",action="write",status="new")
81          write (unit=10,fmt=*) u
82
83  end program equation_chaleur_2D
```

```
$ ifort -openmp
$ a.out
$ export OMP_NUM_THREADS=6
 dx=  9.9999998E-03
 dy=  9.9999998E-03
 Nombre de coeurs disponibles sur la machine:          32
 temps d'excecution du programme:   112,9404
```

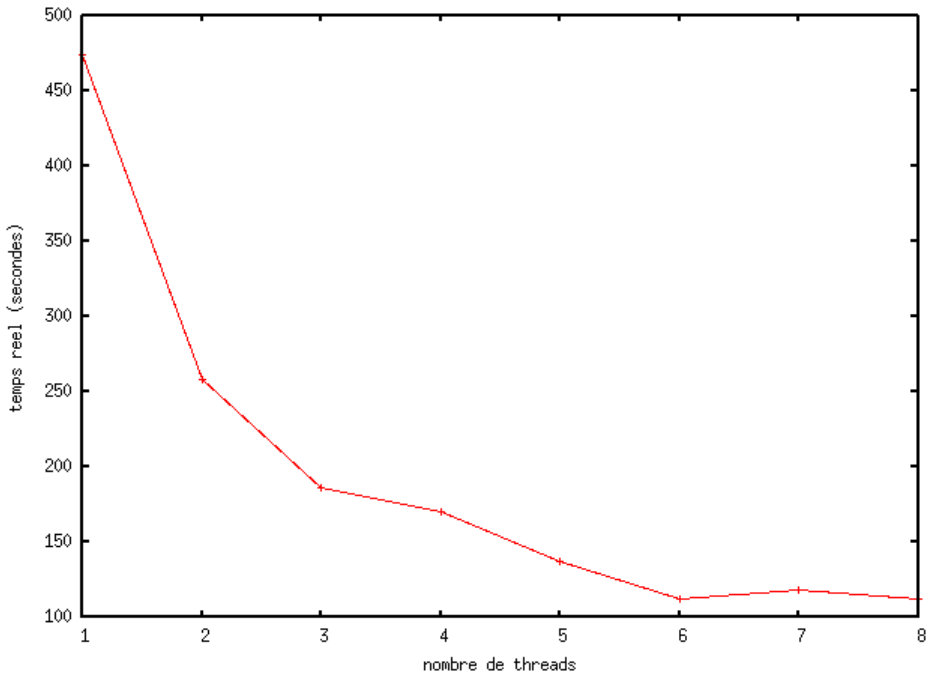


FIGURE 2 – Temps de l'exécution du programme en fonction du nombre de threads utilisés

Exercice 2 : Calcul de π

```
1  program pi
2
3    !$ use OMP_LIB
4    implicit none
5    integer, parameter :: n=100000000
6    double precision   :: f, x, a, h
7    double precision   :: Pi_calcule, Pi_reel, ecart
8    integer            :: i, k
9    integer            :: t1, t2, ir
10   real               :: temps
11
12  ! Fonction instruction a integrer
13   f(a) = 4.0_16 / ( 1.0_16 + a*a )
14
15   ! valeur reel de Pi avec 16 chiffres apres la virgule
16   Pi_reel=3.141592653589793
17
18   ! Longueur de l'intervalle d'integration
19   h = 1.0_8 / real(n,kind=8)
20
21   ! Temps initial
22   call system_clock(count=t1, count_rate=ir)
23
24   do k=1,2
25      ! Calcul de Pi
26      Pi_calcule = 0.0_16
27      !$OMP PARALLEL DO PRIVATE(i,x) REDUCTION(+ : Pi_calcule)
28      do i = 1, n
29         x = h * ( real(i,kind=16) - 0.5_16 )
30         Pi_calcule = Pi_calcule + f(x)
31      end do
32      !$OMP END PARALLEL DO
33      Pi_calcule = h * Pi_calcule
34   enddo
35
36   ! Temps final
37   call system_clock(count=t2, count_rate=ir)
38   temps=real(t2-t1)/real(ir)
39
40   ! Ecart entre la valeur estimee et la valeur calculee de Pi.
41   ecart = abs(Pi_reel - Pi_calcule)
42
43   ! Impression du resultat.
44 write (*,*) "Nombre d intervalles       : ",n
45 write (*,*) "| Pi_reel - Pi_calcule |   : ",ecart
46 write (*,*) "Temps reel                 : ",temps
47
48 end program pi
```

```
$ ifort -openmp pi.SOLUTION.f90
$ export OMP_NUM_THREADS=8
$ a.out
 Nombre d intervalles       :    100000000
 | Pi_reel - Pi_calcule |   :   8.742275836581825E-008
 Temps reel                 :    4.627500
```

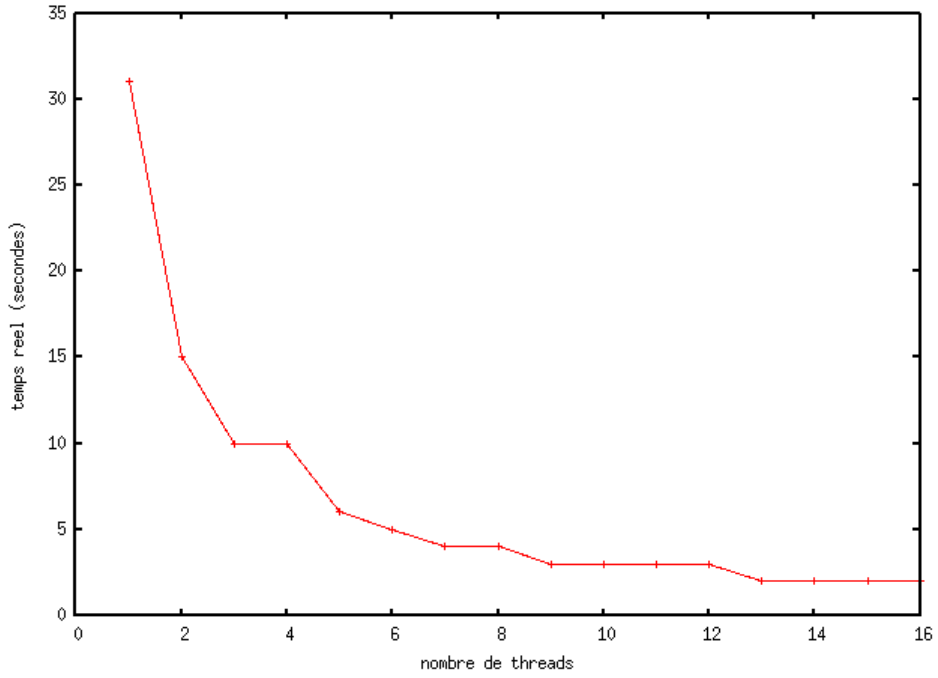


FIGURE 3 – Temps de l'exécution du programme en fonction du nombre de threads utilisés

Exercice 3 : Nombres premiers

```
1  program nombres_premiers
2
3          !$ use OMP_LIB
4          integer :: nombre, compteur, nombre_premier=0
5          integer :: i,j, reste, imax
6          !$ integer :: num_thread
7
8
9          !$OMP PARALLEL
10         num_thread=OMP_GET_NUM_THREADS()
11
12         !$OMP SINGLE
13         write(*,*) "ce programme utilise : ", num_thread, "threads"
14         write(*,*) "Entrez un nombre :"
15         read(*,*) nombre
16         !$OMP END SINGLE
17
18         !$OMP DO PRIVATE(i,j, compteur) REDUCTION(+:nombre_premier)
19         do j=2,nombre
20                 compteur=1
21                 imax=floor(sqrt(real(j)))
22                 do i=2, imax
23                         if (mod(j,i)==0) then
24                                 compteur=0;
25                                 exit;
26                         end if
27                 end do
28                 nombre_premier=nombre_premier+compteur
29         end do
30         !$OMP END DO
31         !$OMP END PARALLEL
32         write (*,*) "dans", nombre, "il y a", nombre_premier, "nombres
               premiers"
33
34 end program nombres_premiers
```

Exercice 4 : Équation de chaleur

```
program equation_chaleur

    real                    :: diff
    real                    :: epsilon = 0.001
    integer                 :: i, j, iterations, iterations_print
    real                    :: valeur_initiale=0
    integer, parameter      :: m=1500, n=1500
    real, dimension (m+1,n+1)   :: u, w
    integer                 :: ir, t1, t2
    real                    :: temps
    character (len = 10) :: resultat_w
    call system_clock(count=t1, count_rate=ir)

    write(*,*) " Resolution de l'eq de la chaleur a l'etat stationnaire*"
    write(*,*) " sur une plaque rectangulaire de taille M*N=",M,N
    write(*,*) " *etat stationnaire : pas d'evolution temporelle"
    write(*,*) " "
    write(*,*) " les iterations seront repetees jusqu'a ce que le chgmt de"
    write(*,*) " temperature soit inferieur a epsilon, epsilon=",epsilon

    !$OMP PARALLEL private (i,j)
    !$ nbr_procs=OMP_GET_NUM_PROCS()
    !$ nbr_threads=OMP_GET_NUM_THREADS()

    !$OMP SINGLE
    !$ write(*,*) "nbr de threads disponibles sur la machine :", nbr_procs
    !$ write(*,*) "nombre de threads utilises :", nbr_threads
    !$OMP END SINGLE

!   Conditions aux limites :
!   ------------------------
    !$OMP DO
    do i=2,m
            w(i,1)=100.0
    end do
    !$OMP END DO nowait
    !$OMP DO
    do i=2,m
            w(i,n)=100.0
    end do
    !$OMP END DO nowait
    !$OMP DO
    do j=1,n+1
            w(m,j)=100.0
    end do
    !$OMP END DO nowait
    !$OMP DO
    do j=1,n+1
            w(1,j)=0.0
    end do
    !$OMP END DO

!   Initialisation des valeurs a l'interieur du domaine (en prenant la
!   moyenne des valeurs aux limites)
!   ---------------------------------------------------------------------
    !$OMP DO REDUCTION (+:valeur_initiale)
    do i=2,m
            valeur_initiale=valeur_initiale+w(i,1)+w(i,n)
    end do
    !$OMP END DO nowait
    !$OMP DO REDUCTION (+:valeur_initiale)
    do j=1,n+1
            valeur_initiale=valeur_initiale+w(m,j)+w(1,j)
    end do
    !$OMP END DO
```

```
66      !$OMP SINGLE
67      valeur_initiale=valeur_initiale/real(2*m+2*n-4)
68      write (*,*) "valeur_initiale=", valeur_initiale
69      !$OMP END SINGLE
70
71      !$OMP DO
72      do i=2,m
73              do j=2,n
74                      w(i,j)=valeur_initiale
75              end do
76      end do
77      !$OMP END DO
78      !$OMP END PARALLEL
79
80      diff=epsilon
81      do while (epsilon <= diff)
82          !$OMP PARALLEL PRIVATE(i,j, my_diff)
83          ! enregistrer la solution precedente dans U
84          !$OMP DO
85          do j=1,n+1
86                  do i=1,m+1
87                          u(i,j)=w(i,j)
88                  end do
89          end do
90          !$OMP END DO
91
92          ! calcul des nouvelles valeurs de W
93          ! (W[Central] = (1/4) * (W[Nord] + W[Sud] + W[Est] + W[Ouest]
94          !$OMP DO
95          do j=2,n
96                  do i=2,m
97                          w(i,j)=(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))*0.25
98                  end do
99          end do
100         !$OMP END DO
101
102         diff=0.0
103         my_diff=0.0
104         !$OMP DO
105         do j=2,n
106                 do i=2,m
107                         if (my_diff < abs(w(i,j)-u(i,j))) then
108                                 my_diff = abs(w(i,j)-u(i,j))
109                         end if
110                 end do
111                 !$OMP CRITICAL
112                 if (diff<my_diff) then
113                         diff=my_diff
114                 end if
115                 !$OMP END CRITICAL
116         end do
117         !$OMP END DO
118         !$OMP END PARALLEL
119         iterations=iterations+1
120         write(*,*) "iteration=", iterations, "diff=", diff
121     end do
122
123     call system_clock(count=t2, count_rate=ir)
124     temps=real(t2 - t1,kind=8)/real(ir,kind=8)
125
126     open (unit=10,file="resultat_w",action="write",status="new")
127     write (unit=10,fmt=*) w
128
129     ! Temps de calcul
130     write(*,*) "Temps reel", temps
131
132 end program equation_chaleur
```

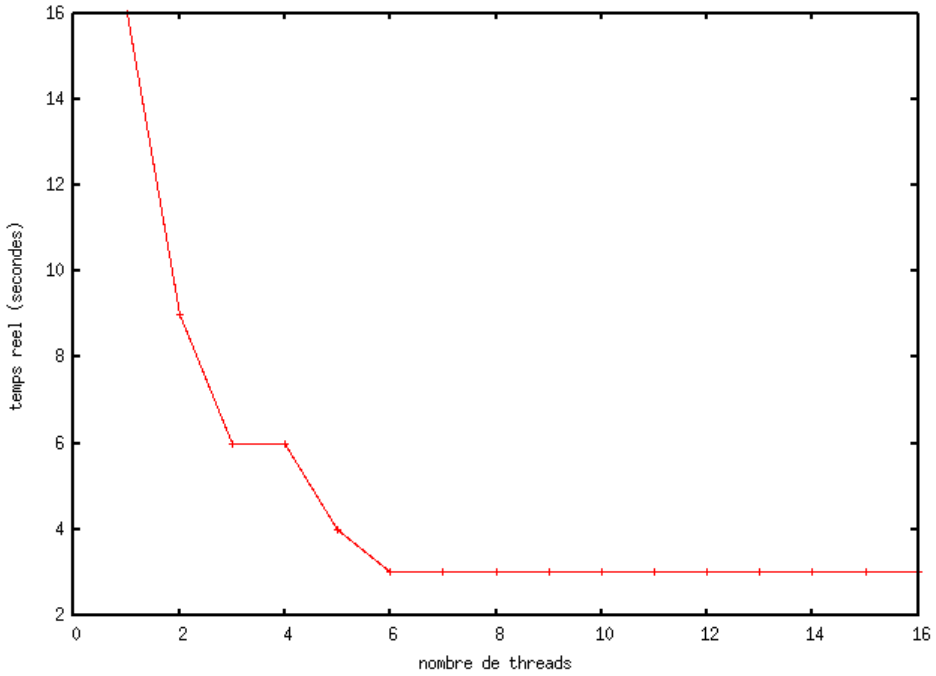


FIGURE 4 – Temps de l'exécution du programme en fonction du nombre de threads utilisés