OpenMP Basics

S. Van Criekingen
UPJV / MeCS

www.mecs.u-picardie.fr

December 4, 2014

OpenMP: references

Official website and specifications :

http://openmp.org/

http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Tutorials:

https://computing.llnl.gov/tutorials/openMP/

http://www.idris.fr/data/cours/parallel/openmp/choix_doc.html

http://www.crihan.fr/calcul/tech/doc_ibm_pwr5/MemPart
http://www.crihan.fr/calcul/tech/doc_ibm_pwr5/OpenMP

Layout

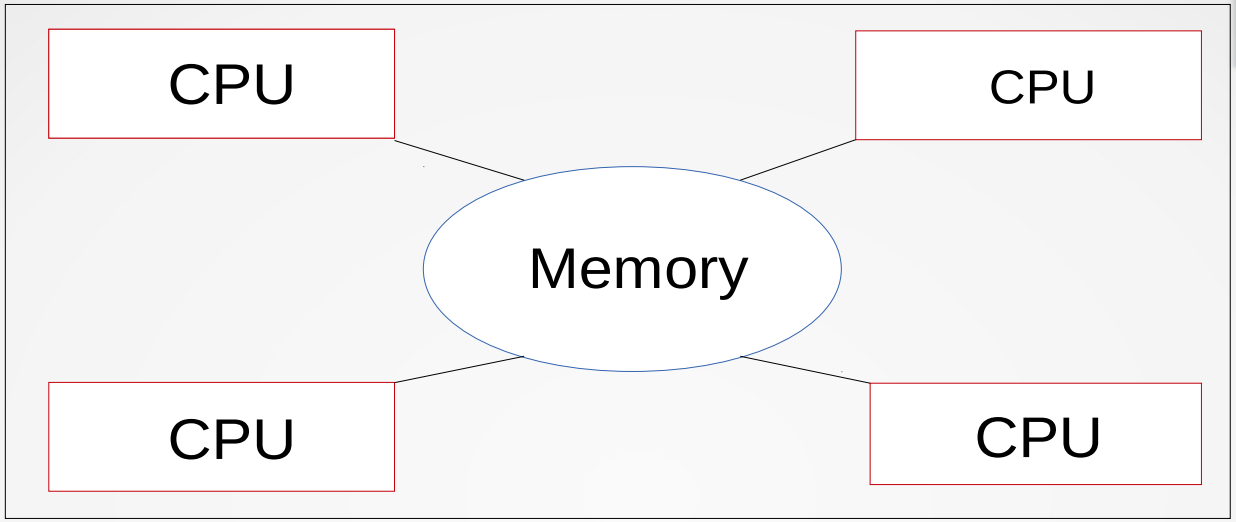
- Introduction & “Hello World”
- Work Sharing
- Synchronization
- Exercises

Layout

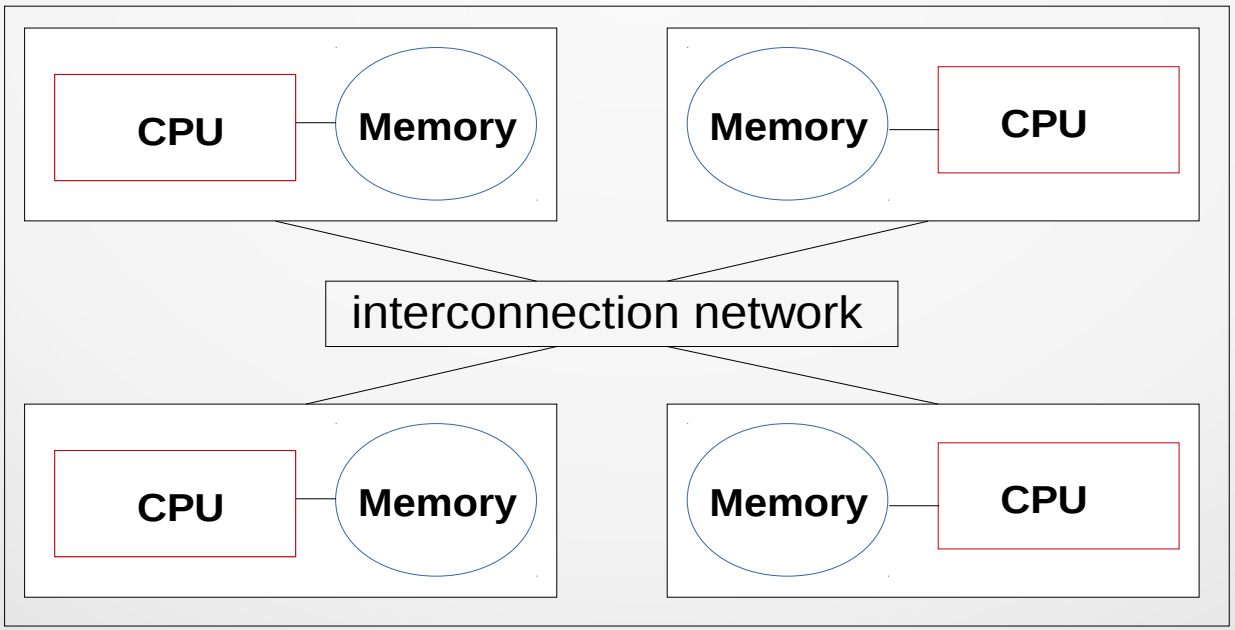
- Introduction & “Hello World”
- Work Sharing
- Synchronization
- Exercises

Shared vs. Distributed Memory

Shared



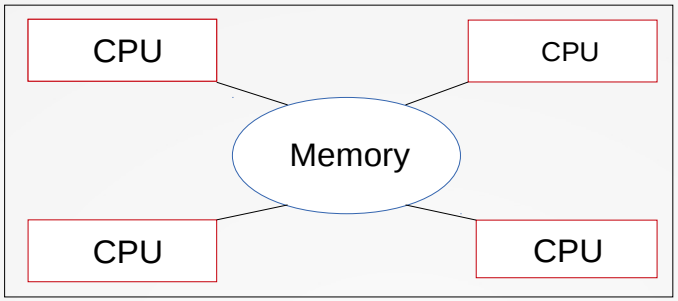
Distributed



Programming models

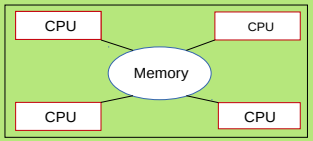
- For shared memory: multi-threading
- For distributed memory: message passing

Multithreading (shared memory)



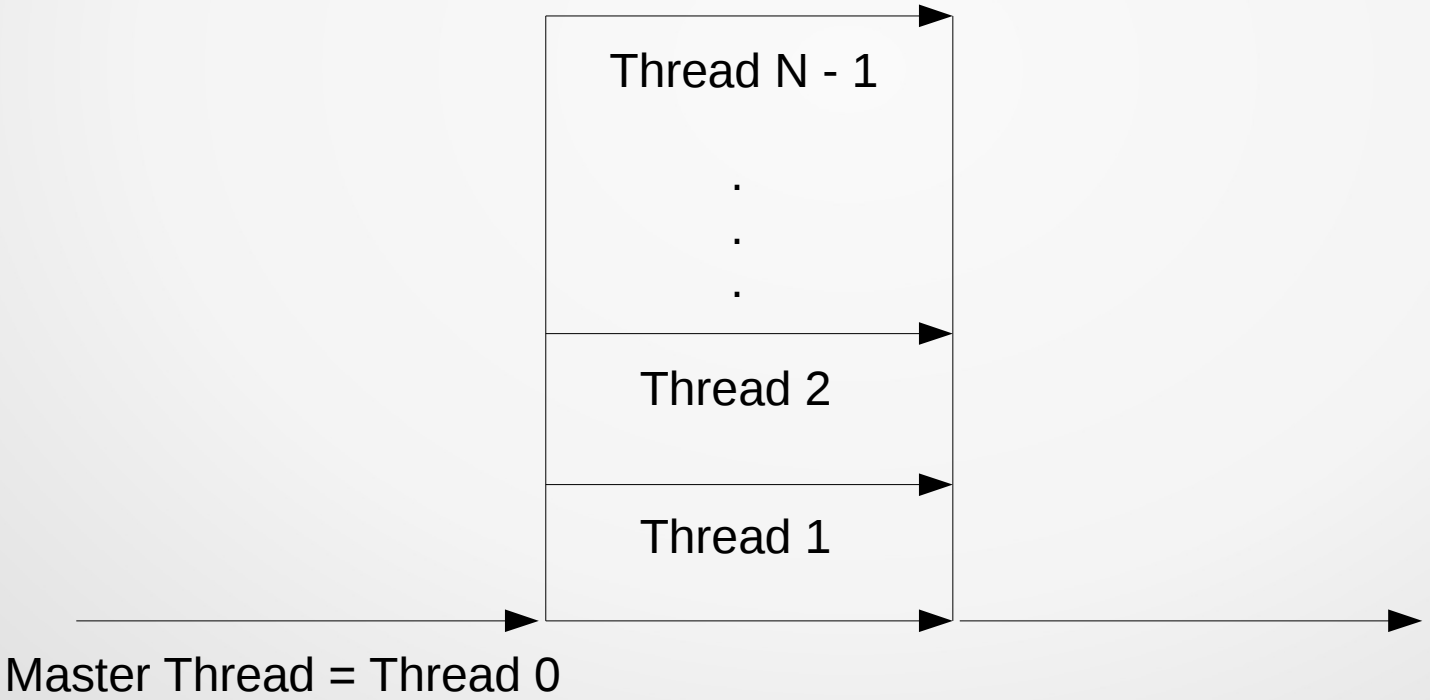
A unique process activates several **threads (*processus légers*)** acting concurrently within the **shared memory**.

"Fork-join" model

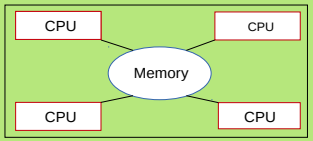


FORK: master thread creates a team of parallel threads
JOIN: threads synchronize at the end of parallel region; then only the master thread continues.

Serial region Parallel region Serial region



Multithreading API's



Application Program Interfaces (API's) enabling multithreading:

- **OpenMP (open Multi-Processing)**
- Pthreads (POSIX threads)
- TBB (Thread Building Blocks)
- ...

OpenMP standard

- OpenMP = Open Multi-Processing
- Uses Fork-Join model
- First specification of standard: 1997 – continue to evolve (OpenMP Architecture Review Board)
- Implemented in many compilers, e.g.,
  - **gcc -fopenmp program.c**
  - **gfortran -fopenmp program.f90**

"Hello world" example (in C)

helloWorld.c

```
#include <omp.h>
#include <stdio.h>
main () {
  int tid ;
  omp_set_num_threads(3);
  #pragma omp parallel private(tid)
  {
     tid = omp_get_thread_num();
     printf("Hello World from thread = %d \n", tid);
  }
  printf("Out of parallel region \n");
}
```

←*set number of threads to 3*

Parallel region within the brackets following ***#pragma omp parallel ...***

```
> gcc -fopenmp helloWorld.c
> ./a.out
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 0
Out of parallel region
```

Order not deterministic

"Hello world" example (in Fortran 90)

helloWorld.f90

```
program main
  use omp_lib
  implicit none
  integer :: tid
  call omp_set_num_threads(3)
  !$omp parallel private (tid)
    tid = omp_get_thread_num()
    write(*,'(a,i2)') 'Hello World from thread ', tid
  !$omp end parallel
 write(*,'(a)') 'Out of parallel region'
end
```

←*set number of threads to 3*

Parallel region between
!$omp parallel …
and
!$omp end parallel

```
> gfortran -fopenmp helloWorld.f90
> ./a.out
Hello World from thread 1
Hello World from thread 2
Hello World from thread 0
Out of parallel region
```

Order not deterministic

Terminology

C code

```
#include <omp.h>
#include <stdio.h>
main () {
  int tid ;
  omp_set_num_threads(3);
  #pragma omp parallel private(tid)
  {
     tid = omp_get_thread_num();
     printf("Hello World from thread = %d \n", tid);
  }
  printf("Out of parallel region \n");
}
```

← *run-time library routine*

← *compiler directive*
(with "private" clause)

← *run-time library routine*

3 ways to change the number of threads

C code

```
[ #include … ]
main ()
{
  int tid ;
  omp_set_num_threads(6);
  #pragma omp parallel private(tid) num_threads(4)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d \n", tid);
  }
}
```

← *run-time library routine*

← *"num_threads" clause*

```
> export OMP_NUM_THREADS=3
> ./a.out
Hello World from thread = 2
Hello World from thread = 0
...
```

← *environment variable :*

- *valid within shell*
- *export / setenv depending on shell*

Precedence : see exercices

Same in Fortran

F90 code

```
program main
  use omp_lib
  implicit none
  integer :: tid
  call omp_set_num_threads(6)
  !$omp parallel private (tid)  num_threads(4)
     tid = omp_get_thread_num()
     write(*,'(a,i2)') 'Hello World from thread ', tid
  !$omp end parallel
end
```

← *run-time library routine*

← *"num_threads" clause*

```
> export OMP_NUM_THREADS=3
> ./a.out
Hello World from thread = 2
Hello World from thread = 0
...
```

← *environment variable*

Precedence : see exercices

Shared vs. **Private** clause

C code

```
[ #include … ]
main ()
{
  int tid, nthreads ;
  omp_set_num_threads(3);
  #pragma omp parallel private(tid) shared(nthreads)
  {
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    printf("Hello from thread = %d out of %d \n", tid, nthreads);
  }
}
```

```
> ./a.out
Hello from thread = 1 out of 3
Hello from thread = 2 out of 3
Hello from thread = 0 out of 3
```

In parallel region :

- tid private

to each thread

- nthreads shared

among threads

Shared vs. **Private** clause (in Fortran)

F90 code

```
program main
  use omp_lib
  implicit none
  integer :: nthreads, tid
  call omp_set_num_threads(3)
  !$omp parallel private (tid) shared (nthreads)
     tid = omp_get_thread_num()
     nthreads = omp_get_num_threads()
     write(*,'(a,i2,a,i2)') 'Hello from thread ', tid, ' out of ', nthreads
  !$omp end parallel
end
```

```
> ./a.out
Hello from thread = 1 out of 3
Hello from thread = 2 out of 3
Hello from thread = 0 out of 3
```

In parallel region :

- tid private

to each thread

- nthreads shared

among threads

Dynamic change of number of threads

C code

```
main ()  {
 int nthreads;
 #pragma omp parallel shared(nthreads) num_threads(3)
 {
   nthreads = omp_get_num_threads();
   printf("Part A: number of threads = %d\n", nthreads);
 }
 #pragma omp parallel shared(nthreads) num_threads(2)
 {
   nthreads = omp_get_num_threads();
   printf("Part B: number of threads = %d\n", nthreads);
 }
}
```

This requires setting OMP_DYNAMIC environment variable to TRUE.
Note: implementation dependent.

```
> export OMP_DYNAMIC=TRUE
> ./a.out
Part A : number of threads = 3
Part B : number of threads = 2
```

← *environment variable*

Also possible with **omp_set_dynamic()** ←*run-time library routine*

Components of OpenMP API (C)

OpenMP is an API with 3 components :

- **Compiler directives (with clauses):**

```
#pragma omp parallel private(tid) num_threads(4)
#pragma omp parallel shared(nthreads) num_threads(3)
#pragma omp barrier
...
```

Note : **shared** by default

- **Run-time library routines :**

```
void omp_set_num_threads(int nthreads)
int omp_get_thread_num(void)
int omp_get_num_threads(void)
void omp_set_dynamic(int dynamic_threads)     (implementation dependent)
double omp_get_wtime(void)
...
```

- **Environment variables :**

```
OMP_NUM_THREADS
OMP_DYNAMIC     (implementation dependent)
...
```

Components of OpenMP API (Fortran 90)

OpenMP is an API with 3 components :

- **Compiler directives (with clauses):**

!$omp parallel private(tid) num_threads(4)
!$omp parallel shared(nthreads) num_threads(3)
!$omp barrier
...

Note : **shared** by default

- **Run-time library routines :**

subroutine **omp_set_num_threads**(integer)
integer function **omp_get_thread_num**()
integer function **omp_get_num_threads**()
subroutine **omp_set_dynamic**(logical) (implementation dependent)
double precision function **omp_get_wtime**()
...

- **Environment variables :**

OMP_NUM_THREADS
OMP_DYNAMIC (implementation dependent)
...

Layout

- Introduction & “Hello World”
- Work Sharing
- Synchronization
- Exercises

Layout

- Introduction & “Hello World”
- Work Sharing
  - For/do loop
  - Sections
  - Single/Master
- Synchronization
- Exercises

Layout

- Introduction & “Hello World”
- Work Sharing
  - For/do loop
  - Sections
  - Single/Master
- Synchronization
- Exercises

"**for**" loop (C): principle

C code

```
...
#pragma omp parallel private(i)
{
  ...
  #pragma omp for schedule(type, [chunk])
   for (i=0; i < N; i++) {...}
   ...
}
...
```

← ***For*** *loop within parallel region*

Loop iterations are divided among threads according to the **schedule** clause

- *type:* **static, dynamic, ...**
- *chunk:* number of iterations assigned to one thread at a time

Static, 5 iterations on 4 threads, 2 by 2

C code

```
main (){
 int tid, i, N=5, chunk=2 ;
 omp_set_num_threads(4);
 #pragma omp parallel private(tid,i)  shared(chunk,N)
 {
   tid = omp_get_thread_num();
   #pragma omp for schedule(static, chunk)
   for (i=0; i < N; i++)
     printf("In the loop: i=%d handled by thread %d \n", i, tid);
  }
}
```

←5 loop iterations, assigned 2 by 2

←4 threads

*←**static** assignment*

```
> ./a.out | sort
In the loop: i=0 handled by thread 0
In the loop: i=1 handled by thread 0
In the loop: i=2 handled by thread 1
In the loop: i=3 handled by thread 1
In the loop: i=4 handled by thread 2
```

4th thread (#3) gets no work

Dynamic, 5 iterations on 4 threads, 2 by 2

C code

```
main (){
 int tid, i, N=5, chunk=2 ;
 omp_set_num_threads(4);
 #pragma omp parallel private(tid,i)  shared(chunk,N)
 {
   tid = omp_get_thread_num();
   #pragma omp for schedule(dynamic, chunk)
   for (i=0; i < N; i++)
     printf("In the loop: i=%d handled by thread %d \n", i, tid);
  }
}
```

←5 loop iterations, assigned 2 by 2

←4 threads

*←**dynamic** assignment*

```
> a.out | sort
In the loop: i=0 handled by thread 0
In the loop: i=1 handled by thread 0
In the loop: i=2 handled by thread 2
In the loop: i=3 handled by thread 2
In the loop: i=4 handled by thread 3
```

```
> a.out | sort
In the loop: i=0 handled by thread 1
In the loop: i=1 handled by thread 1
In the loop: i=2 handled by thread 3
In the loop: i=3 handled by thread 3
In the loop: i=4 handled by thread 2
```

Not deterministic

"**for**" loop: **schedule** types

C code

```
#pragma omp for schedule(type, [chunk])
 for (i=0; i < N; i++) {...}
```

Type can (mainly) be:

- **static:** first chunk of iterations assigned to thread 0, next one to thread 1, … If *chunk* not specified : even distribution of loops among threads.
- **dynamic:** iterations again assigned chunk by chunk to threads, but in any order. When a thread finishes its chunk of iterations, it can be assigned another. Default *chunk* = 1.

Same in Fortran : "**do**" loop

F90 code

```
...
!$omp parallel private (i)
{
  ...
  !omp do schedule(type, [chunk])
  do i = 1, N
    ...
  enddo
  !omp enddo
  ...
}
...
```

Same possibilities for ***type*** and ***chunk*****.**

Static, 5 iterations on 4 threads, 2 by 2

F90 code

```
program main
  use omp_lib
  implicit none
  integer :: tid, i, N=5, chunk=2;
  call omp_set_num_threads(4)
  !$omp parallel private (tid,i) shared(chunk,N)
    tid = omp_get_thread_num()
    !$omp do schedule(static,chunk)
    do i = 1,N
       write(*,'(a,i2,a,i2)') 'In the loop: i=',i,' handled by thread', tid
    enddo
    !$omp end do
  !$omp end parallel
end
```

```
> ./a.out | sort
In the loop: i= 1 handled by thread 0
In the loop: i= 2 handled by thread 0
In the loop: i= 3 handled by thread 1
In the loop: i= 4 handled by thread 1
In the loop: i= 5 handled by thread 2
```

Layout

- Introduction & “Hello World”
- Work Sharing
  - For/do loop
  - Sections
  - Single/Master
- Synchronization
- Exercises

Sections: principle

C code

```
…
#pragma omp parallel
{
   …
   #pragma omp sections
   {
      #pragma omp section
      {…}
      #pragma omp section
      {…}
   }
}
…
```

Each section will be executed by one thread in the team.

Ex: 2 sections shared among 4 threads

C code

```
main(){
  int tid;
  omp_set_num_threads(4);
  #pragma omp parallel private(tid) {
    tid = omp_get_thread_num();
    #pragma omp sections
    {
      #pragma omp section
      {
        printf("Thread %d doing section 1\n",tid);
      }
      #pragma omp section
      {
        printf("Thread %d doing section 2\n",tid);
      }
}}}
```

```
> a.out
Thread 2 doing section 1
Thread 3 doing section 2
```

```
> a.out
Thread 0 doing section 1
Thread 2 doing section 2
```

Not deterministic

Ex: 4 sections shared among 2 threads

C code

```
main(){
  int tid;
  omp_set_num_threads(2);
  #pragma omp parallel private(tid) {
    tid = omp_get_thread_num();
    #pragma omp sections
    {
      #pragma omp section
      {printf("Thread %d doing section 1\n",tid);}
      #pragma omp section
      {printf("Thread %d doing section 2\n",tid);}
      #pragma omp section
      {printf("Thread %d doing section 3\n",tid);}
      #pragma omp section
      {printf("Thread %d doing section 4\n",tid);}
}}}
```

```
> a.out
Thread 0 doing section 1
Thread 0 doing section 3
Thread 0 doing section 4
Thread 1 doing section 2
```

```
> a.out
Thread 1 doing section 2
Thread 1 doing section 3
Thread 1 doing section 4
Thread 0 doing section 1
```

Not deterministic

Sections in Fortran

F90 code

```
...
!$omp parallel
   ...
   !$omp sections
      !$omp section
        ...
      !$omp section
        ...
   !$omp end sections
  ...
!$omp end parallel
...
```

Ex: 2 sections shared among 4 threads

F90 code

```
program main
 use omp_lib
 implicit none
 integer :: tid
 call omp_set_num_threads(4)
 !$omp parallel private (tid)
  tid = omp_get_thread_num()
  !$omp sections
   !$omp section
    write(*,'(a,i2,a)') 'Thread ',tid,' doing section 1'
   !$omp section
    write(*,'(a,i2,a)') 'Thread ',tid,' doing section 2'
  !$omp end sections
 !$omp end parallel
end
```

```
> a.out
Thread  0 doing section 1
Thread  2 doing section 2
```

```
> a.out
Thread  0 doing section 1
Thread  1 doing section 2
```

Not deterministic

Layout

- Introduction & “Hello World”
- Work Sharing
  - For/do loop
  - Sections
  - Single/Master
- Synchronization
- Exercises

single: principle

C code

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

Code enclosed in "**single**" will be executed by one thread in the team (but you do not control which one)

master: principle

C code

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

Code enclosed in "**master**" will be executed by the master thread

Note: the "**single**" and "**master**" directives can be useful for I/O

Ex: **single**

C code

```
main () {
 int tid;
 omp_set_num_threads(4);
 #pragma omp parallel private(tid)
 {
   tid = omp_get_thread_num();
   #pragma omp single
   {
     printf("Only thread %d doing this part\n",tid);
   }
   printf("Thread %d doing this part\n",tid);
}}
```

```
> ./a.out  | sort
Only thread 3 doing this part
Thread 0 doing this part
Thread 1 doing this part
Thread 2 doing this part
Thread 3 doing this part
```

```
> ./a.out  | sort
Only thread 0 doing this part
Thread 0 doing this part
Thread 1 doing this part
Thread 2 doing this part
Thread 3 doing this part
```

Not deterministic

Ex: **master**

C code

```
main () {
 int tid;
 omp_set_num_threads(4);
 #pragma omp parallel private(tid)
 {
   tid = omp_get_thread_num();
   #pragma omp master
   {
     printf("Only thread %d doing this part\n",tid);
   }
   printf("Thread %d doing this part\n",tid);
}}
```

```
> ./a.out  | sort
Only thread 0 doing this part
Thread 0 doing this part
Thread 1 doing this part
Thread 2 doing this part
Thread 3 doing this part
```

Deterministic

Single/Master in Fortran

F90 code

```
...
!$omp parallel
  ...
  !$omp single
    ...
  !$omp end single
 ...
!$omp end parallel
...
```

F90 code

```
...
!$omp parallel
  ...
  !$omp master
    ...
  !$omp end master
 ...
!$omp end parallel
...
```

Ex: **single**

F90 code

```
program main
  use omp_lib
  implicit none
  integer :: tid
  call omp_set_num_threads(4)
  !$omp parallel private (tid)
    tid = omp_get_thread_num()
    !$omp single
      write(*,'(a,i2,a)') 'Only thread ',tid,' doing this part'
    !$omp end single
    write(*,'(a,i2,a)') 'Thread ',tid,' doing this part'
  !$omp end parallel
end
```

```
> ./a.out  | sort
Only thread  2 doing this part
Thread  0 doing this part
Thread  1 doing this part
Thread  2 doing this part
Thread  3 doing this part
```

```
> ./a.out  | sort
Only thread  0 doing this part
Thread  0 doing this part
Thread  1 doing this part
Thread  2 doing this part
Thread  3 doing this part
```

Not deterministic

Layout

- Introduction & “Hello World”
- Work Sharing
- Synchronization
- Exercises

Layout

- Introduction & “Hello World”
- Work Sharing
- Synchronization
  - Barrier
  - Critical & Atomic
  - Reduction
- Exercises

Layout

- Introduction & “Hello World”
- Work Sharing
- Synchronization
  - Barrier
  - Critical & Atomic
  - Reduction
- Exercises

barrier: principle

C code

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

Each thread waits for all the others before going further.

barrier example (in C)

C code

```
main ()
{
  int tid ;
  omp_set_num_threads(3);
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello from thread = %d \n", tid);
    #pragma omp barrier
    printf("Goodbye from thread = %d \n", tid);
  }
  printf("Out of parallel region \n");
}
```

Without barrier :

```
> ./a.out
Hello from thread = 2
Goodbye from thread = 2
Hello from thread = 1
Goodbye from thread = 1
Hello from thread = 0
Goodbye from thread = 0
Out of parallel region
```

With barrier :

```
> ./a.out
Hello from thread = 2
Hello from thread = 1
Hello from thread = 0
Goodbye from thread = 2
Goodbye from thread = 0
Goodbye from thread = 1
Out of parallel region
```

barrier: notes

- A barrier is implicitly present at the end of workshare constructs (for, sections,...) unless **nowait** clause.
- A barrier can be useful for clean output.

Barrier in Fortran

F90 code

```
...
!$omp parallel
    ...
    !$omp barrier
    ...
!$omp end parallel
...
```

Layout

- Introduction & “Hello World”
- Work sharing
- Synchronization
  - Barrier
  - Critical & Atomic
  - Reduction
- Exercises

critical: principle

C code

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

Code enclosed in « critical » will be executed one thread at a time.

critical: example

C code

```
main ()
{
 int x=0;
 omp_set_num_threads(4);
 #pragma omp parallel shared(x)
 {
   #pragma omp critical
   {
     x = x+1;
   }
 }
 printf("At the end x= %d \n",x);
}
```

With « critical » :

```
> ./a.out
At the end x= 4
```

Without « critical » :

```
> ./a.out
At the end x= 4
```

```
> ./a.out
At the end x= 3
```

Not deterministic

Why?

critical: more on example

C code

```
main ()
{
 int x=0;
 omp_set_num_threads(4);
 #pragma omp parallel shared(x)
 {
   #pragma omp critical
   {
     x = x+1;
   }
 }
 printf("At the end x= %d \n",x);
}
```

Each thread

- copies **x** into a register
- performs addition in register
- puts back sum value into **x**

Thus: copy to a register should not be done while another thread is operating on **x** in another register (mutual exclusion needed).

Otherwise : « *race condition* »

Same in Fortran

F90 code

```
program main
 use omp_lib
 implicit none
 integer :: x=0
 call omp_set_num_threads(4)
 !$omp parallel shared (x)
   !$omp critical
     x= x+1
   !$omp end critical
 !$omp end parallel
 write(*,'(a,i2)') 'At the end x=',x
end
```

With « critical » :

```
> ./a.out
At the end x= 4
```

Without « critical » :

```
> ./a.out
At the end x= 4
```

```
> ./a.out
At the end x= 3
```

Not deterministic

atomic: principle

C code

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

« Atomic = mini-critical »

Same effect as critical but only on directly following instruction (- would have been sufficient for our example).

atomic/critical: example

C code

```
main ()
{
 int x=0;
 omp_set_num_threads(4);
 #pragma omp parallel shared(x)
 {
   #pragma omp critical
   {
     x = x+1;
   }
 }
 printf("At the end x= %d \n",x);
}
```

C code

```
main ()
{
 int x=0;
 omp_set_num_threads(4);
 #pragma omp parallel shared(x)
 {
   #pragma omp atomic
      x = x+1;
 }
 printf("At the end x= %d \n",x);
}
```

```
> ./a.out
At the end x= 4
```

Same in Fortan

F90 code

```
program main
 use omp_lib
 implicit none
 integer :: x=0
 call omp_set_num_threads(4)
 !$omp parallel shared (x)
   !$omp atomic
     x= x+1
   !$omp end atomic
 !$omp end parallel
 write(*,'(a,i2)') 'At the end x=',x
end
```

← *optional*

Layout

- Introduction & “Hello World”
- Work sharing
- Synchronization
  - Barrier
  - Critical & Atomic
  - Reduction
- Exercises

reduction: principle

Idea : to "reduce" a given shared variable within a parallel region, by applying a given operator (sum, product, minimum,...) to the set of values of this variable in each thread.

Typically used with for/do loops.

reduction: principle

C code

```
#pragma omp parallel ...
{
  ...
  #pragma omp for schedule(static) reduction(+:var)
  for (i=0; i < N; i++)
  {
     var = var + 1;
  }
  ...
}
```

Chunk not specified → even distribution of loops among threads.

← Here **var** = sum of the values of var on all threads

Note

The **reduction** clause takes care of mutual exclusion.

Example (sum)

C code

```
[...] N=5, var=0;
omp_set_num_threads(4);
#pragma omp parallel shared(var,N) private(i,tid)
{
  tid = omp_get_thread_num();
  #pragma omp for schedule(static) reduction(+:var)
  for (i=0; i < N; i++)
  {
    var = var + 2;
  }
  printf("On thread %d, var=%d \n", tid, var);
}
```

```
> ../a.out | sort
On thread 0, var=10
On thread 1, var=10
On thread 2, var=10
On thread 3, var=10
```

Note

Without the **reduction** clause, you might get in trouble.

Failure example (sum)

```
[...] N=5000, var=0;
omp_set_num_threads(4);
#pragma omp parallel shared(var,N) private(i,tid)
{
  tid = omp_get_thread_num();
  #pragma omp for schedule(static) reduction(+:var)
  for (i=0; i < N; i++)
  {
    var = var + 2;
  }
  printf("On thread %d, var=%d \n", tid, var);
}
```

← N must be sufficiently large to observe failure (machine dependent)

```
> ../a.out | sort
On thread 0, var=3854
On thread 1, var=3854
On thread 2, var=3854
On thread 3, var=3854
```

Not deterministic

Example (sum) in Fortran 90

F90 code

```
[…] N=5, var=0;
call omp_set_num_threads(4)
!$omp parallel shared(var,N) private(tid,i)
   tid = omp_get_thread_num()
   !$omp do schedule(static) reduction(+:var)
   do i = 1,N
      var = var + 2
   enddo
   !$omp end do
   write(*,'(a,i2,a,i2)') 'On thread=', tid, ' var=',var
 !$omp end parallel
```

```
> ../a.out | sort
On thread 0, var=10
On thread 1, var=10
On thread 2, var=10
On thread 3, var=10
```

Note

Typical use of reduction : **dot product**

```
#pragma omp for schedule(static) reduction(+:var)
  for ( i=0; i < N; ++i ) {
    dot_product += x[i] * y[i];
  }
```

Layout

- Introduction & “Hello World”
- Work sharing
- Synchronization
- Exercises

Exercise 1: *Hello World*

Compile and run the *Hello World* program (slides 11/12) and determine the order of precedence between the different ways to fix the number of threads (slides 14/15).

Print the total number of threads using a shared variable (slides 16/17).

Exercise 2: *For/do loop*

Compile and run the **for/do loop** example, modify the parameters **chunk**, **N** and schedule type (**static** and **dynamic**), and observe the changes (slides 25-26 / 28-29).

Exercise 3: *for/do loop* with scaling

InitVec.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main () {
   int i, j;
   double *x, startTime, endTime;
   int N = 5e6;
   int nloops = 500;
   x = (double*) malloc (N*sizeof(double));
   startTime = omp_get_wtime();
   for ( j=0; j < nloops; ++j ) {
    for ( i=0; i < N; ++i ) {
       x[i] = 1.;
    }
   }
   endTime = omp_get_wtime();
   printf("Global time = %10.5f\n", endTime - startTime);
   free (x);
}
```

The ***initVec*** code initializes a vector of size n, this initialization being repeated **nloops** times.

- Compile and run the code
- Parallelize the initialization **for** loop on i
- compare timing performances on 1, 2, 4,... threads ;
  (Take chunk = N/nthreads.)

Exercise 3: *for/do loop* with scaling

InitVec.f90

```
[...]
 integer :: i, j
 double precision, allocatable :: x(:)
 double precision startTime, endtime
 integer :: N = 5e6
 integer :: nloops = 500
 allocate(x(N))
 startTime = omp_get_wtime()
 do j = 1,nloops
   do i = 1,N
     x(i) = 1.
   enddo
 enddo
 endTime = omp_get_wtime()
 deallocate(x)
 write(*,'(a,f10.5)') 'Global time =', endTime - startTime
```

The ***initVec*** code initializes a vector of size n, this initialization being repeated **nloops** times.

- Compile and run the code
- Parallelize the initialization **for** loop on i
- compare timing performances on 1, 2, 4,... threads ;
  (Take chunk = N/nthreads.)

Exercise 4: *reduction*

Compile and run the *reduction* example (slide 62/64).

Then replace the sum reduction by a product reduction as follows:

- Initialize **var** at 1 instead of 0
- Replace **reduction(+:var)** by **reduction(*:var)**
- In the loop, replace **var+= 2** by **var *=2**

Observe and explain the result.

Observe the failure when the *reduction* clause is removed (you might have to increase N to observe it).

Solutions of exercises 3 and 4
will be posted on

www.mecs.u-picardie.fr

(click on “séminaire”)