Institut
Mines-Télécom

Introduction to OpenMP

Elisabeth Brunet
Télécom SudParis

OpenMP - Agenda

- **Introduction**
- **Execution Model**
- **Structure of an OpenMP code**
  - Construction of a parallel region
- **Scope of variables**
- **SPMD Parallelism SPMD : work and data distribution**
- **Exclusive regions**
- **Synchronization**
- **Nesting of parallel regions**
- **Conclusion**

Introduction : Context

- **Shared memory architectures**
- **Possible strategies of exploitation**
  - Several independent processes
  - Several processes collaborating thanks to data exchanges, e.g. via MPI
  - Intrinsic parallelization
    - Pthreads : POSIX thread library
    - …
    - OpenMP : Open specifications for MultiProcessing

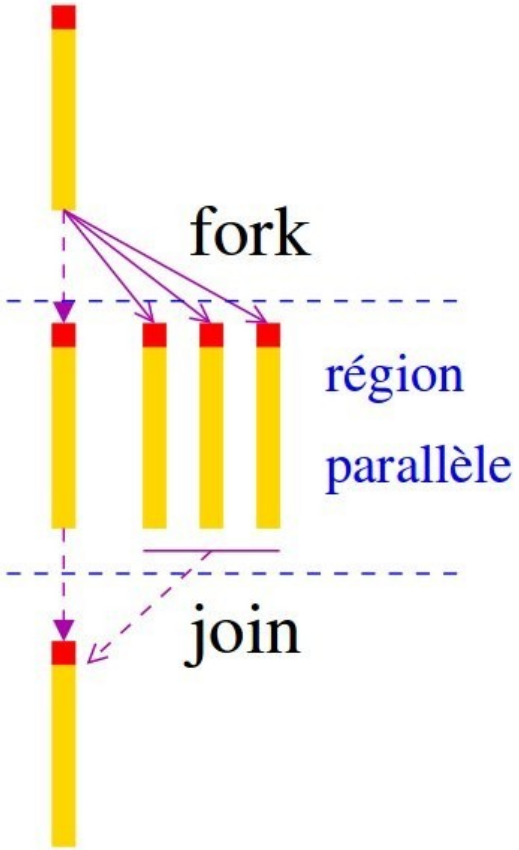
Introduction : Standard OpenMP

- **Language by pragmas allowing the description of the parallel regions in a sequential code**
- **Support for C/C++ and Fortran**
- **Portable on many architectures and OS**

- **Industrial standard introduced in 1997**
- **Specifications established by the ARB (Architecture Review Board) : http://openmp.org**
  - OpenMP2 in 2000 (extension for Fortran 95),
  - OpenMP3 in 2008 (introduction of task),
  - OpenMP4 in 2013 (affinity et support of accelerators)

Execution model

- **Fork-join within a process**
  - Sequential region by the master thread
  - For each parallel region, creation of a team of threads that will share the calculation
  - Implicit synchronization and destruction parallel threads at each end of parallel section



Structure of an OpenMP code : OpenMP pragmas

- **Annotation of sequential code with OpenMP pragmas**
- **Syntax according to the skeleton :**
  - Sentinel directive [clause[clause]...]
  - In C : sentinel = #pragma omp
  - In fortran : sentinel = $!OMP
- **Code region only one entry point and one exit point**
  - i.e. no branching to the inside or outside of the region
  - Valid for all OpenMP constructions
- **Pragmas if OpenMP support is activated at compilation, otherwise it remains comments**
  - Upward portability of programs

Structure of an OpenMP code : Construction of a parallel region

- **Directive parallel**
- **The only directive that creates a threads team**
- **In a parallel section, redundant execution of the code by threads team**

```
int main(){
    // Sequential code
    #pragma omp parallel
    {
    /* Parallel code executed by
    all the threads team*/
    }
    // Sequential code
}
```

Structure of an OpenMP code : Library and environnement

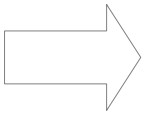
- **Prototypes C/C++ in omp.h, Fortran ones in OMP_LIB**

- **Thread team size**
  - Statically thanks to environment variable OMP_NUM_THREADS
  - Modification within the program
    - Clause num_threads of the directive parallel
    - Function omp_set_num_threads()
  - At runtime, choice of the optimal number of threads
    - If environment variable OMP_DYNAMIC is set to true
    - Call to function omp_set_dynamic()

Structure of an OpenMP code : Thread team size

```
export OMP_NUM_THREADS = 4;
```

```
#pragma omp parallel
  printf(« Section 1\n»)

#pragma omp parallel num_threads(2)
  printf(« Section 2\n »)

omp_set_num_threads(3) ;
#pragma omp parallel
  printf(« Section 3\n»)
```



Section 1
Section 1
Section 1
Section 1
Section 2
Section 2
Section 3
Section 3
Section 3

Structure of an OpenMP code : Thread numbering

- **The master thread is numbered 0**

- Size of a new thread team
  omp_get_max_threads()
- #concurrent threads in the region
  omp_get_num_threads()

- Thread id : omp_get_thread_num()

Structure of an OpenMP code : Compilation et execution

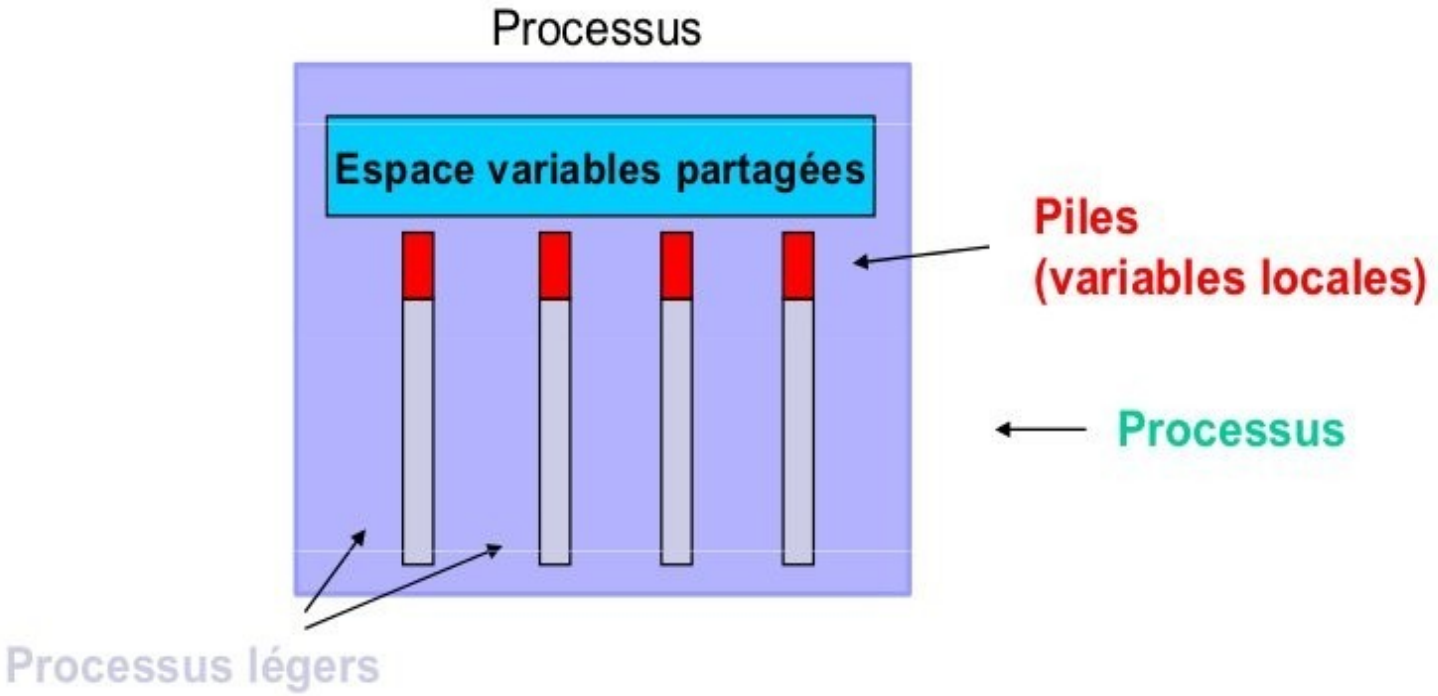
- **Compilation :**
  - Support in most classical compilers
  - Option to be specified at compile time
  - In C : gcc -fopenmp prog.c -o prog
  - In fortran : gfortran -fopenmp -std=f95 prog.f
  - Protect calls to OpenMP library functions for when OpenMP support is disabled

```
#ifdef _OPENMP
    omp_get_num_threads()
#endif
```

- **Execution**
  - Positioning of environment variables
    - OMP_NUM_THREADS, OMP_DYNAMIC, ...
  - Regular launching : ./prog

Scope of variables

- **Usual access to memory from threads**
  - Access to the virtual memory of the process
  - Access to their respective stacks



Scope of variables:

Transmission to parallel sections

- **By default, variables are shared**

- **Clause shared**
  - Shared variable that remains in global memory

- **Clause private**
  - Variable duplicated in each stack but !! not initialized !!!
- **Clause firstprivate**
  - duplicated variable in each stack initialized by the original variable value

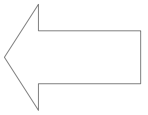
Scope of variables: Directive threadprivate

- **Make persist a variable private to thread from a region parallel to another one**
- **Clause copyin sets the variable to the value held by thread 0 for all the threads**

```
export OMP_NUM_THREADS = 3
```

```
int a = 10, b = 40;

#pragma omp threadprivate (a, b)
```

```
S1 - 2 - 2, ?
S1 - 0 - 0, 40
S1 - 1 - 1, ?
S2 – 2 – 2, 40
S2 – 1 – 1, 40
S2 – 0 - 0, 40
```

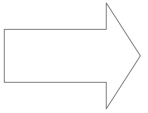


```
#pragma omp parallel private (tid)
{
  tid = omp_get_thread_num() ;
  a = tid ;
  printf(« S1 - %d – %d, %d\n », tid, a , b) ;
}
#pragma omp parallel private (tid) copyin(b)
{
  tid = omp_get_thread_num() ;
  printf(« S2 - %d – %d, %d\n », tid, a , b) ;
}
```

Scope of variables: Argument transmission

- **Inherits the scope defined for the current parallel region**

```
void function (int a, int *b){
  *b = a + omp_get_thread_num();
}


int main(){
  int a = 5, b;
  omp_set_num_threads(6);

#pragma omp parallel private(b)
  {
    function(a, &b);
    printf("%d - a=%d, b=%d\n", omp_get_thread_num(), a, b);
  }
  return EXIT_SUCCESS;
}
```



```
3 - a=5, b=8
4 - a=5, b=9
0 - a=5, b=5
1 - a=5, b=6
2 - a=5, b=7
5 - a=5, b=10
```

Recap

- **Directive parallel**

#pragma omp parallel [clause ...]

if (scalar_expression)

private (list)

shared (list)

default (shared | none)

firstprivate (list)

reduction (operator: list)

copyin (list)

num_threads (integer)

{

structured_block

}

- **Directive threaprivate**

#pragma omp threadprivate (list)

- **Specification of the team thread size**
- **Thread numbering**
- **Compilation/execution**

SPMD parallelism : Work and data distribution

- **Fine distribution of**
  - computation
  - Data assigned to each worker

- **Directives**
  - For loop
  - Mutual exclusions: master and single directives

Work distribution: For directive

- **Distribution of the iterations of a loop following**
  - Default scheduling defined at the installation of the environment
  - Environment variable OMP_SCHEDULE
  - *Schedule* clause
- **Scope of variables maintained**
  - *Lastprivate* clause : keeps the value of the last iteration after the loop
- ***Nowait* clause**
  - disable synchronization at the end of the region
- **Warning**
  - Loop indices are integer and private
  - No infinite loop
  - No loop while

```
#pragma omp for [clause ...]
                schedule (type [,chunk])
                ordered
                collapse(n)
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                nowait

  for_loop
```

- **« *Parallel for* » directive = merge of « *parallel* » and « *for* » directives**

Work distribution: For directive – Schedule clause(1/2)

- **Specification of the iteration distribution mode**
- **Function omp_get_schedule(omp_sched_t *kind, int *modifier)**
  - kind={omp_sched_static|omp_sched_dynamic| omp_sched_guided|omp_sched_auto}
    modifier = iteration size
- **Clause schedule (static|dynamic|guided|runtime[, taille])**

#pragma omp for schedule(static, 2)

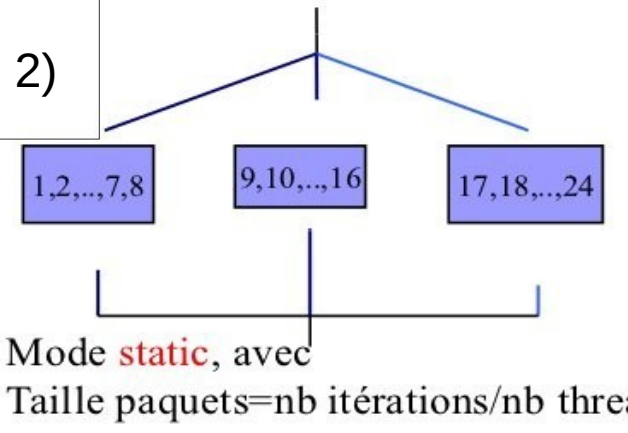
Work distribution: For directive - Schedule clause (1/2)

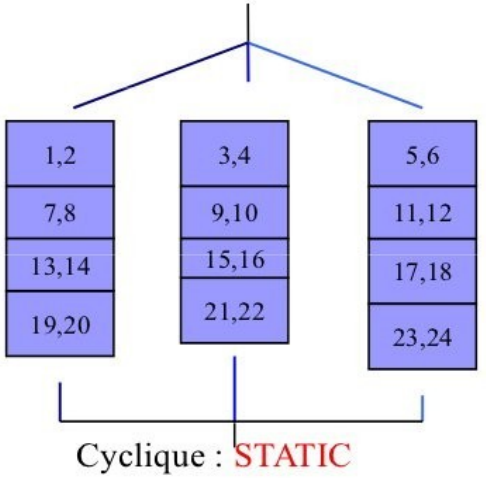
- **Specification of the iteration distribution mode**
- **Function omp_get_schedule(omp_sched_t *kind, int *modifier)**
  - kind={omp_sched_static|omp_sched_dynamic| omp_sched_guided|omp_sched_auto}
    modifier = iteration size
- **Clause schedule (static|dynamic|guided|runtime[, taille])**
  - **static** : divides the iterations by slices of the given size then cyclic distribution
    - if unspecified, size=nb_iter/nb_threads

Ex : 24 iterations, 3 threads

#pragma omp for schedule(static)

#pragma omp for schedule(static, 2)





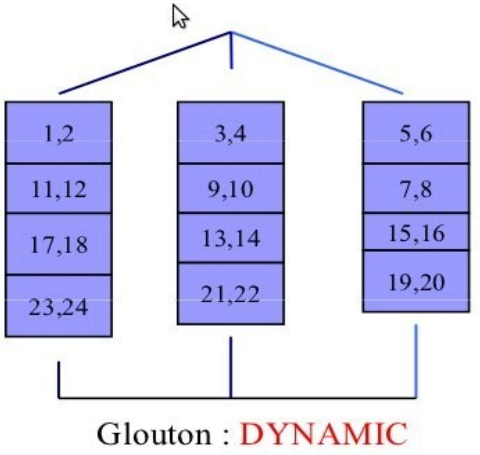
Work distribution: For directive - Schedule clause (2/2)

- **dynamic** : slices of the specified size Are distributed among threads dynamically
  - If unspecified, size=1

Ex : 24 iterations, 3 threads

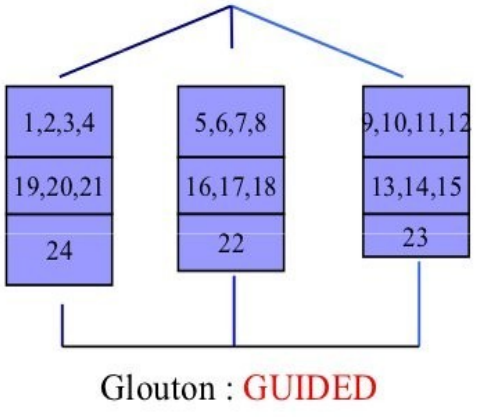
#pragma omp for schedule(dynamic,2)

#pragma omp for schedule(guided, 4)



- **guided** : iterations are divided in slices whose size decreases exponentially.
  - If unspecified, size=1

- **runtime** :
  - Schedule choice postponed at runtime
  - OMP_SCHEDULE environment variable
  - Ex : export OMP_SCHEDULE= "dynamic, 2"



Work distribution: For directive – Collapse clause

- **Specifies a number of loops to unroll in order to create a wider iteration space**

- **Increases the degree of parallelism**

```
#pragma omp parallel for collapse(2)
  for(i=0 ; i < M ; i++)
    for(j=0 ; j < N ; j++)
      for(k=0 ; k < K ; k++){
        fonc(i, j, k) ;
      }
```

- No triangular iteration space
- Perfect nesting: no interleaved Operation between loops

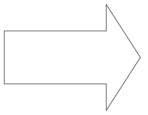
Without collapse :
- Repartition following i → max M threads
- Each thread executes the entire 2nd and 3rd loops

Withe collapse :
- Repartition following i and j → max M*N threads
- Each thread executes the entire 3rd loop

For directive – Ordered clause

- **Executes the loop sequentially**
  - For debugging
  - Whatever the scheduling policy set up
  - Also available as a directive

```
A[4] = {1,2,3,4} ;
#pragma omp parallel for schedule(dynamic)
  for(i=0 ; i < 4 ; i++)
     printf(« B1 - %d\n », A[i]) ;

#pragma omp parallel for schedule(dynamic) ordered
  for(i=0 ; i < 4 ; i++)
     printf(« B2 - %d\n », A[i]) ;
```



```
B1 - 2
B1 - 3
B1 - 1
B1 - 4
B2 – 1
B2 – 2
B2 – 3
B2 - 4
```

Nesting of parallel constructions

- **Parallel regions can be nested if**
  - OMP_NESTED env variable set up
  - Prior call to omp_set_nested()

```
#pragma omp parallel
{
  # pragma omp for
  for (i=0 ; i < 10 ; i++){
     ...
    #pragma omp parallel
    # pragma omp for
    for(j=0 ; j < 10 ; j++){
       ...
    }
  }
 }
```

- Nesting induces the use of a new parallel region

Exclusive regions

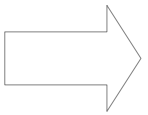
- Master directive
- Single directive
- Critical directive

Exclusive regions: Master directive

- **Execution by the master thread only**
- **No synchronization with other threads**
  - No wait for start or end of execution

```
#pragma omp master

  structured_block
```

```
int a, tid ;
#pragma omp parallel private (a, tid)
{
  a = 20 ;
  tid = omp_get_thread_num() ;

  #pragma omp master
  {
   a = 10 ;
  }

  printf(« %d, a=%d\n », tid, a) ;
}
```



```
1- 20
2 - 20
0 - 10
```

Exclusive regions: Single directive

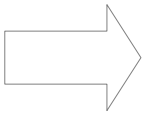
- **Execution only by the first thread that arrives at the construction**
- **Implicit barrier at end of construction**
  - Threads not participating wait for the end of the execution before continuing their execution
  - Unless nowait clause

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate (list)
                   copyprivate(list)
                   nowait

    structured_block
```

- **Copyprivate clause**
  - Update private variables of threads at the directive end

Exclusive regions: Illustration

```
int main(){
  int tid, a;

  omp_set_num_threads(4);

#pragma omp parallel private(tid, a)
  {
    tid = omp_get_thread_num();
    a = 888;

#pragma omp single
    a = 999;

    printf("%d - a = %d\n", tid, a);

#pragma omp master
    printf("%d – The End \n", tid) ;
  }
  return EXIT_SUCCESS ;
}
```



```
0 - a = 888
3 - a = 888
2 - a = 999
1 - a = 888
0 – The End
```

Exclusive regions: Critical directive

- **All threads participate in the construction each in turn**
- **Non-deterministic order**
  - ➔ first come, first served

```
#pragma omp critical [name]
  structured_block
```

Synchronization

- **Barrier directive**

```
#pragma omp barrier
```

  - All threads must have reached the barrier before they can continue
  - Implicit barrier at the end of the parallel regions
    - Possible release with nowait clause

- **Atomic directive**

```
#pragma omp atomic
  statement_expression
```

  - Atomic update in memory
  - Variable modification by one thread at a time

- **Flush directive**

```
#pragma omp flush (list)
```

  - Ensures that shared variables are refreshed at the thread level
    - Cached variables for performance purposes
    - Particularly useful on NUMA machines

Conclusion

- **OpenMP is a high level language**
- **Allows to quickly parallelize a program written in sequential**

- **Data-oriented parallelism with data and work distribution**
- **Scope of variables**
- **Untreated**
  - Task-oriented parallelism
  - Accelerator, simd constructions, cancelation, thread affinity redefinition of reduction operator, sub-tables,...
- **Reference Card :**
  - openmp.org/mp-documents/OpenMP4.0-CCard.pdf