Modèle de programmation parallèle OpenMP
I. d’Ast - CERFACS

Formation HPC pour tous, OMP, Février 2018
Goal: to learn how to read and write programs for shared memory machines thanks to OpenMP

- 9:00 - 10h30 : Introduction - OpenMP fundamentals - Parallel regions - Shared memory
- 11h - 12h30 : Worksharing and Synchronizations

Lecture notes based on IDRIS (http://www.idris.fr) training sessions
Basic facts related to parallel computing:

- We aim at solving large-scale problems or at processing large amount of data usually in a reduced computing time.
- We want to exploit parallelism of modern processor architectures (multicore, multithreading).
- We want several processes to contribute to the complete computation.
- The main principle behind is thus to share the computing load in a well-balanced way in between the different processes (load balance).
Most modern high-performance computing (HPC) systems are clusters of multiprocessors nodes.
Major parallel programming models:

- **OpenMP** is a standard programming model for shared memory parallel computers.
  - Shared Memory Directives,
  - to define the work decomposition,
  - no data decomposition,
  - synchronization is implicit (can be also user-defined).

- **MPI (Message Passing Interface)** is an open-source, standard message passing library, targeting portability and efficiency for programming distributed memory computers.
  - User specifies how work and data is distributed,
  - User specifies how and when communication has to be done,
  - by calling MPI communication library.
Processes

Each application is a separate process in the OS.

- A process has its own memory space which is not accessible by other running process,
- Processes are ring-fenced from each other: if web browser crashes, it can’t scribble over document stored in the memory your word processor,
- Each process is scheduled to run by the OS.

On a single core, rapid process switching gives illusion applications run concurrently. With a multicore processor, multiple processes can really run at the same time.
Threads

A single process can contain multiple threads.

- Each thread is like a child process contained within parent process,
- All threads in a process have access to the same memory, the memory of the parent process,
- Multiple threads can operate at the same time on the same data on multicore processor.

Threads cannot scale beyond a single node, using multiple nodes requires multiple processes and inter-process communication.
Shared variables parallelism:
- uses threads
- requires shared memory machine
- easy to implement but limited scalability
- in HPC, done using OpenMP compilers

Distributed memory:
- uses processes
- can run on any machine: messages can go over the interconnect
- harder to implement but better scalability
- in HPC, done using the MPI library
OpenMP Agenda

1. Introduction
2. Principles
3. Worksharing
4. Synchronizations
5. Traps
History

- An OpenMP-2 version was finalized in November 2000. In particular, it brought extensions related to the parallelization of certain Fortran 95 constructions.
- OpenMP-3 dating from May 2008, essentially introduces the concept of tasks.
- OpenMP-4 released in July 2013, introduces accelerators offloading, task dependencies, SIMD directives, and thread affinity.
- OpenMP-4.5 was released November 2015. It adds some features as task priorities and improved support for devices.
OpenMP Resources

- Specifications of the OpenMP standard: http://www.openmp.org/
- Website dedicated to the OpenMP users: http://www.compunity.org/
General concepts

- An OpenMP program is executed by one process.
- This process (master thread) activates threads at the entry of a parallel region.
- Each thread executes a group of instructions.
- During the execution of a task by a thread, a variable can be read and/or updated in memory.
- It can be defined either in the stack (local memory space) of a thread: **private variable**, or in a shared-memory space accessible by all the threads: **shared variable**.
An OpenMP program is an alternation of sequential regions and parallel regions.

Sequential region is always executed by the MASTER thread, the one whose rank equals 0.

A parallel region can be executed by many threads at once.

Threads can share the work contained in the parallel region.
The work sharing consists mainly of:

- executing a loop by dividing up iterations between the threads;
- executing many code sections but only one per thread;
- executing many occurrences of the same procedure by different threads (orphaning).

---

**Parallel loop** (Looplevel parallelism)

**Parallel section**

**Orphaning**
General concepts

1. Compilation Directives and Clauses:
   - They serve to create the threads, define the work sharing, the synchronization and the data sharing attributes of variables (shared or private);
   - They are considered by the compiler as comment lines unless a suitable compiler option is specified so that they may be interpreted.

2. Functions and Routines: They are part of a loaded library at link.

3. Environment Variables: Once set, their values are taken into account at execution.
To compile an OpenMP program you need to add a (compiler-specific) flag to your compile and link commands.

- `fopenmp` for gcc/gfortran
- `qopenmp` for Intel compilers
- `mp` for PGI compilers

The number of threads which will be used is determined at runtime by the `OMP_NUM_THREADS` environment variable; set this before you run the program.

```
export OMP_NUM_THREADS = 4
```

Run in the same way you would a sequential program.

```
./a.out
```
OpenMP directive format: Fortran

sentinel directive_name [ clause [ [ , ] clause ] ... ]

Treated as Fortran comments - not case sensitive

- Directive sentinels:

  Fixed source form:
  !$OMP | C$OMP | *$OMP [starting at column 1]

  Free source form:
  !$OMP [may be preceded by white space]

- Conditional compilation:

  Fixed source form: !$ | C$ | *

  Free source form: !$

  Example: !$ write(*,*) OMP_GET_NUM_PROCS(),’ avail. processors’

- Include file for library routines: include ’omp_lib.h’ or use OMP_LIB
OpenMP directive format: C/C++

```c
#pragma omp directive_name [ clause [ [ , ] clause ] ... ] new-line
```

The `#pragma` directive is case sensitive.

- **Conditional compilation:**
  ```c
  #ifdef _OPENMP
  printf("%d avail.processors", OMP_GET_NUM_PROCS());
  #endif
  ```

- **Include file for library routines:**
  ```c
  #ifdef _OPENMP
  #include <omp.h>
  #endif
  ```
In a parallel region, by default, the data sharing attribute of the variables is shared.

Within a single parallel region, all the concurrent threads execute the same code.

There is an implicit synchronization barrier at the end of the parallel region.

A program that branches (e.g. GOTO, CYCLE) into or out of a parallel region is non conforming.

```fortran
program parallel
  implicit none
  real :: a
  logical :: p
  a = 92290. ; p=.false.
  !$OMP PARALLEL
  !$ p = OMP_IN_PARALLEL ()
  print *,"A = ",a, "; p = " ,p
  !$OMP END PARALLEL
end program parallel
```
It is possible, with the DEFAULT clause, to change the default data-sharing attribute of the variables in a parallel region.

If a variable has a private data-sharing attribute (PRIVATE), it will be stored in the stack of each thread. Its value in this case is indeterminate at the entry of a parallel region.

```c
#include <stdio.h>

int main() {
    float a=92000;
    #pragma omp parallel private(a)
    {
        a= a + 290; // dangerous
        printf ("a = %f",a);
    }
    printf("Out of region, A= %f",a);
    return 0;
}
```
However, with the FIRSTPRIVATE clause, it is possible to force the initialization of this private variable with the last value it had before the entry in the parallel region.

```fortran
program parallel
implicit none
real :: a
a = 92000.
!$OMP PARALLEL \textbf{DEFAULT(NONE)} &
!$OMP \textbf{FIRSTPRIVATE(a)}
a = a + 290.
print *, "A = ", a
!$OMP END PARALLEL
print *, "Out of region, A = ", a
end program parallel
```

export OMP_NUM_THREADS=4
a.out
Out of region, A =?
Parallel region extent

- The extent of an OpenMP construct is the range of its influence in the program.
- The influence (or the scope) of a parallel region extends to the code lexically contained in this region, as well as to the code of the called routines. The union of the two represents the dynamic extent.

```plaintext
program parallel
  implicit none
  !$OMP PARALLEL
  call sub()
  !$OMP END PARALLEL
end program parallel

subroutine sub()
  implicit none
  logical :: p
  !$ p = OMP_IN_PARALLEL()
  !$ print *,"Parallele ?:", p
end subroutine sub
```
In a called routine inside a parallel region, the local and automatic variables are implicitly private to each thread (they are stored in the stack of each thread).

```fortran
program parallel
  implicit none
  !$OMP PARALLEL DEFAULT(SHARED)
  call sub()
  !$OMP END PARALLEL
end program parallel

subroutine sub()
  !$use OMP_LIB
  implicit none
  integer :: a
  a = 92290
  !$ a = a + OMP_GET_THREAD_NUM()
  print *, "A = ", a
end subroutine sub
```
Argument-passing case

In a called routine inside a parallel region, all the dummy arguments passed by reference inherit the data-sharing attribute of the actual associated argument.
Static variables case

A variable is static if its location in memory is defined at declaration by the compiler.

This is the case of variables appearing in COMMON or contained in a MODULE or declared with SAVE or initialized when declared (ex. PARAMETER, DATA, etc.).

By default, a static variable is a shared variable.

```fortran
module var_stat
  real :: c
end module var_stat
```

```fortran
program parallel
  use var_stat
  implicit none
  real :: a
  common /bidon/a
  !$OMP PARALLEL
  call sub()
  !$OMP END PARALLEL
end program parallel
```

```fortran
subroutine sub()
  use var_stat
  implicit none
  real :: a, b=10.
  integer :: rank
  common /bidon/a
  !$OMP BARRIER
  print *, "values of A, B and C : ", a, b, c
end subroutine sub
```

```fortran
module var_stat
real :: c
end module var_stat
```
Exercises

1. Hello world : shared and private memory
2. Loop : Parallel region and work sharing
3. Memory Allocation
OpenMP proposes three directives (DO, SECTIONS and WORKSHARE) to control the work and data distribution inside a parallel region:

- DO
- SECTIONS
- WORKSHARE

In addition, there are OpenMP constructs which enable the exclusion of all the threads except for one thread, in order to execute a code portion located in a parallel region:

- SINGLE
- MASTER
Parallel loops

- A parallel loop is a loop for which all iterations are independent of each other.
- The parallelized loop is the one which comes immediately after the DO directive.
- The distribution mode of the iterations can be specified with the SCHEDULE clause,
- The loop indices are private integer variables.
- By default, a global synchronization is done at the end of a END DO construct unless the NOWAIT clause is specified.
- It is possible to introduce as many DO constructs as we want (one after another) in a parallel region.
Parallel loops syntax

**FORTRAN:**

```
!$OMP DO [clauses]
do loop
!$OMP END DO [nowait]
```

Clauses:
- private, firstprivate, lastprivate
- reduction
- schedule, collapse, ordered

**C/C++:**

```
#pragma omp for [clauses]
for loop
```

Clauses:
- private, firstprivate, lastprivate
- reduction
- schedule, collapse, ordered
- nowait
program parallel
$ use OMP_LIB
implicit none
integer, parameter :: n=4096
real, dimension(n) :: a
integer :: i, i_min, i_max, rank, nb_threads
$OMP PARALLEL PRIVATE(rank,nb_threads,i_min,i_max)
$ rank= OMP_GET_THREAD_NUM() ; nb_threads= OMP_GET_NUM_THREADS()
i_min=n ; i_max=0
$OMP DO SCHEDULE(STATIC,n/nb_threads)
do i = 1, n
a(i) = 92290. + real(i) ; i_min=min(i_min,i) ; i_max=max(i_max,i)
end do
$OMP END DO NOWAIT
print *,"Rank : ",rank,"; i_min :",i_min,"; i_max :",i_max
$OMP END PARALLEL
end program parallel
Schedule Clause

For reference:

- **Fortran:** `SCHEDULE (kind[, chunksize])`
- **C/C++:** `schedule (kind[, chunksize])`

- **static:** With no chunksize specified, the iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread in order.
- **dynamic:** divides the iteration space up into chunks of size chunksize , and assigns them to threads on a first-come-first-served basis. Default chunksize : 1
- **guided:** similar to DYNAMIC, but the chunks is reduced in an exponentially decreasing manner. Default chunksize : 1 (here, chunk specifies the smallest piece)
- **auto:** Lets the runtime choose the schedule.
- **runtime:** The schedule is choosed at run time with 
  `OMP_SCHEDULE` environment variable.
Ordered execution

- It is sometimes useful (when debugging) to execute a loop in an orderly way.
- The order of iterations will then be identical to the one occurring during a sequential execution.

Fortran:

```fortran
!$OMP DO ORDERED
.. 
!$OMP ORDERED ! don’t forget
block
!$OMP END ORDERED
```

C/C++:

```c
#pragma omp for ordered
.. 
#pragma omp ordered // don’t forget
block
```
Reduction

- Perform reduction on the shared variables that appear in list, with the operator (+,*,-,max,min...)
- Each thread computes a partial result independently of the others. They are then synchronized in order to update the final result.
- Variables can be arrays in Fortran

Fortran:
```fortran
 !$OMP DO 
 reduction (operator : list)
```

C/C++:
```c
#pragma omp for 
 reduction (operator : list)
```

Combined parallel do/for directive

- Shortcut form for specifying a parallel region that contains a single do/for directive with the union of their respective clauses
- The termination directive END PARALLEL DO includes a global synchronization barrier and cannot admit the NOWAIT clause.

Fortran:

```
!$OMP PARALLEL DO [clauses]
do loop
!$OMP END PARALLEL DO
```

C/C++:

```
#pragma omp parallel for [clauses]
for loop
```
Parallel Sections

1. The SECTIONS construct defines structured blocks to be distributed among and executed by the threads. Each structured block introduced by SECTION is executed once by one of the threads in the team.

2. The goal is to distribute the execution of many independent code portions (structured blocks) on different threads.

3. The NOWAIT clause is allowed at the end of the END SECTIONS construct to remove the implicit synchronization barrier.

4. The PARALLEL SECTIONS directive is a combination of PARALLEL and SECTIONS directives with the union of their respective clauses. It cannot admit the NOWAIT clause.
3- Parallel programming model OpenMP

3.2- Worksharing

FORTRAN:

```fortran
!$OMP SECTIONS [clauses]
!$OMP SECTION
block1
!$OMP SECTION
block2
...
!$OMP END SECTIONS [nowait]
```

Clauses:
- private, firstprivate, lastprivate
- reduction

C/C++:

```c
#pragma omp sections [clauses]
{
#pragma omp section
block1
#pragma omp section
block2
...}
```

Clauses:
- private, firstprivate, lastprivate
- reduction
- nowait
program parallel
implicit none
integer, parameter :: n=513, m=4097
real, dimension(m,n) :: a, b
real, dimension(m) :: coord_x
real, dimension(n) :: coord_y
real :: step_x, step_y
integer :: i
!
$OMP PARALLEL
!
$OMP SECTIONS
!
$OMP SECTION
!$OMP NOWAIT

!$OMP SECTION
!$OMP NOWAIT
!
$OMP SECTION
call read_initial_x(a)
!
$OMP SECTION
call read_initial_y(b)
!
$OMP SECTION
step_x = 1./real(m-1)
step_y = 2./real(n-1)
coord_x(:) = (/real(i-1)*step_x,i=1,m)/
coord_y(:) = (/real(i-1)*step_y,i=1,n)/
!
$OMP END SECTIONS
!
$OMP END PARALLEL
end program parallel

subroutine read_initial_x(x)
implicit none
integer, parameter :: n=513, m=4097
real, dimension(m,n) :: x
!$OMP NOWAIT
!
$OMP SECTION
call random_number(x)
!
$OMP END SECTION
end subroutine read_initial_x

subroutine read_initial_y(y)
implicit none
integer, parameter :: n=513, m=4097
real, dimension(m,n) :: y
!
$OMP SECTION
call random_number(y)
!
$OMP END SECTION
end subroutine read_initial_y
Exclusive Execution

In order to execute a block of code by only one thread, use:

- **SINGLE directive**: the first thread to reach the SINGLE directive will execute the block, all the other threads wait until block has been executed unless nowait is specified. The clauses are private, firstprivate, copyprivate, nowait.

- **MASTER directive**: the master thread (thread 0) executes the block, there is no synchronisation at the end of the block.

FORTRAN:

```fortran
!$OMP SINGLE [clauses]
block
!$OMP END SINGLE [NOWAIT]
```

C/C++:

```c
#pragma omp single [clauses]
block
```

FORTRAN:

```fortran
!$OMP MASTER
block
!$OMP END MASTER
```

C/C++:

```c
#pragma omp master
block
```
Exercises

1. Counter: Variable status and Worksharing
2. sections: Worksharing
3. flowdep: Flow dependance
4. antidep: Anti dependance
Synchronization becomes necessary in the following situations:

- to make sure that all the concurrent threads have reached the same instruction point in the program (global barrier);
- to order the execution of all the concurrent threads when they have to execute the same code portion affecting one or many shared variables whose coherence in memory (read or write) has to be guaranteed (mutual exclusion);
- to synchronize at least two concurrent threads among the others (lock mechanism).
BARRIER

- The BARRIER directive synchronizes all the concurrent threads within a parallel region.
- Each thread waits until all the other threads reach this point of synchronization in order to continue, together, the program execution.

FORTRAN:

```fortran
!$OMP BARRIER
```

C/C++:

```c
#pragma omp barrier
```

```c
program parallel
implicit none
real, allocatable, dimension(:) :: a, b
integer :: n, i
n = 5
!$OMP PARALLEL
!$OMP SINGLE
allocate(a(n), b(n))
!$OMP END SINGLE
!$OMP MASTER
read(9) a(1:n)
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO SCHEDULE(STATIC)
do i = 1, n
  b(i) = 2.*a(i)
end do
!$OMP END PARALLEL
print *, "B = ", b(1:n)
deallocate(a, b)
end program parallel
```
ATOMIC

- The ATOMIC update ensures that a shared variable is read and modified in memory by one and only one thread at a time.
- It applies only to a single instruction.

```plaintext
program parallel
$ use OMP_LIB
implicit none
integer :: counter, rank
counter = 92290
$OMP PARALLEL PRIVATE(rank)
$rank = OMP_GET_THREAD_NUM()
$OMP ATOMIC
counter = counter + 1
print *, "Rank : ", rank, "; counter = ", counter
$OMP END PARALLEL
print *, "In total, counter = ", counter
end program parallel

export OMP_NUM_THREADS=4
./a.out
In total, counter = ?
```
FORTRAN:

\$OMP ATOMIC

Instruction:

Instruction: $x = x \text{ op } \exp$, $x = \exp \text{ op } x$, $x = f(x, \exp)$, $x = f(\exp, x)$

\text{op} : +, *, -, /, .and., .or., .eqv., or .neqv.

\text{f} \text{ is one of MAX, MIN, IAND, IOR or IEOR}

\exp \text{ is any arithmetic expression independent of } x.$

C++:

#pragma omp atomic

Instruction:

Instruction: $x++$, $++x$, $x--$, $--x$ or $x \text{ binop } = \exp$

\text{binop is one of +, *, -, /, &, ^, <<, or >>}

\exp \text{ is a scalar expression.}$
A critical section is a block of code which can be executed by only one thread at a time.

Can be used to protect updates to shared variables.

The CRITICAL directive allows critical sections to be named.

If one thread is in a critical section with a given name, no other thread may be in a critical section with the same name (though they can be in critical sections with other names).
program parallel
implicit none
integer :: s, p
s=0
p=1
!
$OMP PARALLEL
  !$OMP CRITICAL
s = s + 1
  !$OMP END CRITICAL
  !$OMP CRITICAL (RC1)
p = p * 2
  !$OMP END CRITICAL (RC1)
  !$OMP CRITICAL
s = s + 1
  !$OMP END CRITICAL
  !$OMP END PARALLEL
print *, "s= ",s, " ; p= ",p
end program parallel

export OMP_NUM_THREADS=3
./a.out
s=? p=?
Parallelization

- The overhead of executing a parallel region is typically in the tens of microseconds range. So the sequential execution time of a section of code has to be several times this to make it worthwhile parallelising.
- Is my loop is parallelisable? Run the loop in reverse order!
- Does my OpenMP code give the same numerical results as the sequential one? What about reproducibility?
- Be careful with reduction operations. Floating-point operations are non-associative operations due to round-off errors.
- Note that the default schedule for loops with no schedule clause is implementation defined. Don’t write code that relies on a particular mapping of iterations to threads.
SINGLE or MASTER

- Both constructs cause a code block to be executed by one thread only, while the others skip it: which should you use?
- MASTER has lower overhead (it is just a test, whereas SINGLE requires some synchronisation).
- But beware that MASTER has no implied barrier!
- If you expect some threads to arrive before others, use SINGLE, otherwise use MASTER.
Data sharing

- Don’t forget that private variables are uninitialised on entry to parallel regions!
- Always use default(none) when debugging.
- Avoid data race: two threads access the same shared variable and at least one thread modifies the variable and the accesses are concurrent, i.e. unsynchronized.
- Compiling with OpenMP flag forces all local variables to be stack allocated and not heap allocated: contents of local variables are not preserved between function calls.
- The size of thread stack apart from the master thread can be controlled by the OMP_STACKSIZE environment variable. The size of the master thread’s stack is controlled in the same way as for sequential program.
Time measurements

Use OMP_GET_WTIME in order to measure the elapsed time in seconds. This measure can vary from one execution to another according to the machine workload and the distribution of threads on the cores.
Exercises

1. \texttt{pi_openmp}: Data sharing and reduction operations
2. Dependency: Parallelization of recurrence relation
• OpenMP requires a multi-processor computer with shared memory.
• Parallelization is relatively easy to implement, even when starting from a sequential program and allows incremental parallelization.
• Within the parallel regions, the work can be shared through the parallel loops and the parallel sections. However, a thread can also be singled out for a particular work.
• Point-to-point or global explicit synchronizations are sometimes necessary in the parallel regions.
• Special attention must be payed to the definition of the data-sharing attribute of used variables in a construct.