CPA – Parallel Computing

Informatics Engineering Degree

S2. Programming with OpenMP

J. M. Alonso, F. Alvarruiz, I. Blanquer, D. Guerrero, J. Ibáñez,
E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Year 2016/17

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Content

Section 1

Basic Concepts

- Programming Model
- Simple Example

OpenMP specification

De facto standard for shared memory programming

`http://www.openmp.org`

Specifications:

- Fortran: 1.0 (1997), 2.0 (2000).
- C/C++: 1.0 (1998), 2.0 (2002).
- Fortran/C/C++: 2.5 (2005), 3.0 (2008), 3.1 (2011), 4.0 (2013), 4.5 (2015)

Previous experiences:

- ANSI X3H5 Standard (1994).
- HPF, CMFortran.

Programming Model

OpenMP programming is mainly based on: compiler directives.

Example

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

Advantages:

- It eases the adaptation (the compiler ignores `#pragma`).
- It enables incremental parallelisation.
- It enables compiler-based optimization.

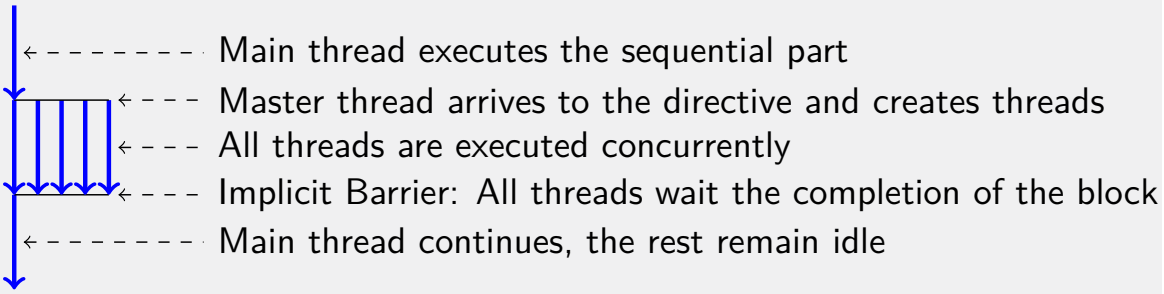
Additionally: functions (see `omp.h`) and environment variables.

Execution Model

OpenMP execution models follows the *fork-join* schema.

There are directives to create threads and splitting the work.

Schema



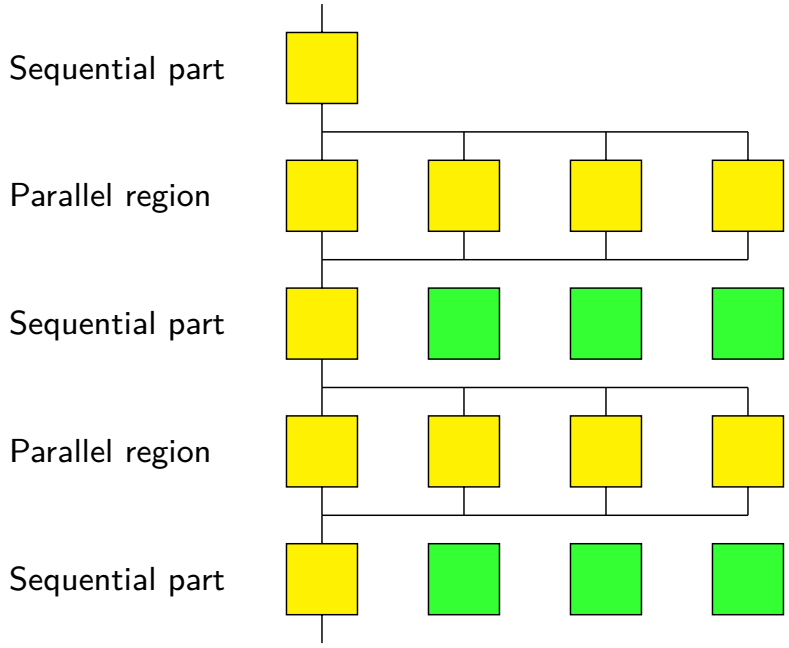
Directives define parallel regions.

Other directives/clauses:

- Type of variable: `private`, `shared`, `reduction`.
- Synchronization: `critical`, `barrier`.

Execution model - Threads

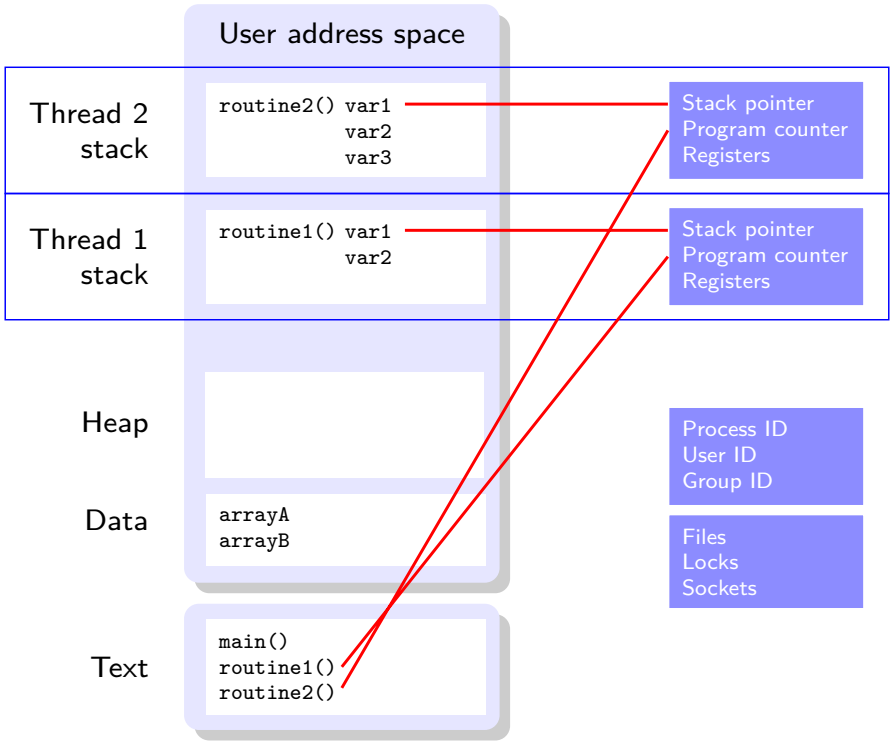
In OpenMP, idle threads are not destroyed. They remain ready for the next parallel region.



Threads created by a directive are called team.

Execution Model - Memory

Each thread has its own execution context (including the stack)



Syntax

Directives:

C/C++

```
#pragma omp <directive>
```

Fortran

```
!$omp <directive>
```

Usage of functions:

```
#include <omp.h>
...
#if _OPENMP
  iam = omp_get_thread_num();
#endif
```

Compilación:

```
gcc-4.2> gcc -fopenmp prg-omp.c
sun> cc -xopenmp -x03 prg-omp.c
intel> icc -openmp prg-omp.c
```


Simple Example

Example

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

- When the execution reaches the directive `parallel`, threads are created (if they have not been created previously).
- Loop iterations are split among the threads.
- By default, all the variables are shared, except for loop iteration variable (`i`) which is always private.
- At the end, all threads are synchronized.

Number and Identifier of Thread

The number of threads can be explicitly defined:

- Using the clause `num_threads`.
- Using the function `omp_set_num_threads()` *before* the parallel region.
- In the execution time, with `OMP_NUM_THREADS`.

Useful functions:

- `omp_get_num_threads()`: It returns the number of threads.
- `omp_get_thread_num()`: It returns the identifier of the thread (starting from 0, main thread is always 0).

```
omp_set_num_threads(3);
printf("threads before = %d\n",omp_get_num_threads());
#pragma omp parallel for
for (i=0; i<n; i++) {
  printf("threads = %d\n",omp_get_num_threads());
  printf("I am %d\n",omp_get_thread_num());
}
```


Section 2

Loop Parallelisation

- Directive `parallel for`
- Variable Types
- Performance Improvement

Directive `parallel for`

Next loop is parallelised:

C/C++

```
#pragma omp parallel for [clause [clause ...]]
for (index=first; test_expr; increment_expr) {
    // loop body
}
```

OpenMP imposes restrictions to the loop type, for instance:

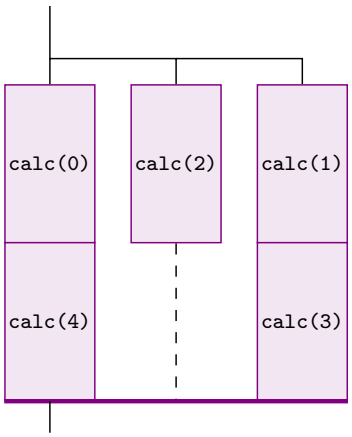
```
for (i=0; i<n && !encontrado; i++)
  if (x[i]==elem) encontrado=1;
```


Example of `parallel for`

Consider a possible execution with 3 threads

Simple loop

```
#pragma omp parallel for
for (i=0; i<5; i++) {
  a[i] = calc(i);
}
```



Implicit barrier at the end of the `parallel for` construction

Variables:

- `a`: concurrent access, but not more than one thread accessing the same position
- `i`: different value in each thread $\rightarrow$ need a private copy

Variable types

Variables are classified according to their (*scope*):

- Private: each thread has a different replica.
- Shared: all threads can read and write.

Typical source of errors: Incorrect scope.

The scope can be modified with clauses added to the directives:

- `private`, `shared`.
- `reduction`.
- `firstprivate`, `lastprivate`.

`private`, `shared`

If the scope of a variable is not defined, by default it is `shared`.

Exceptions (`private`):

- Index variable of the parallelised loop.
- Local variables of the called subroutines, (except if they are declared `static`).
- Automatic variables declared inside the loop.

Clause `default`

- `default(none)` forces to specify the scope of all variables.

private, shared

private

```
suma = 0;
#pragma omp parallel for private(suma)
for (i=0; i<n; i++) {
    suma = suma + x[i]*x[i];
}
```

Wrong: after the loop, only the sum of the main thread is available - moreover, copies of each thread have no initial value.

shared

```
sum = 0;
#pragma omp parallel for shared(suma)
for (i=0; i<n; i++) {
    sum = sum + x[i]*x[i];
}
```

Wrong: race condition at reading/writing `sum`.

Reduction

To perform reduction of commutative and associative operators (`+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`, `max`, `min`)

reduction(redn_oper: var_list)

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<n; i++) {
    sum = sum + x[i]*x[i];
}
```

Each thread performs a part of the sum, and all parts are combined at the end in the total sum.

It works as a private variable, but it is correctly initalized.

`firstprivate`, `lastprivate`

When created, private variables do not have an initial value and after the completion of the `parallel` block, its value is undefined.

- `firstprivate`: Sets up as initial value the value of the main thread at the begining of the block.
- `lastprivate`: The value of the variable after the block is the one of the "last" iteration of the loop.

Example

```
alpha = 5.0;
#pragma omp parallel for firstprivate(alpha) lastprivate(i)
for (i=0; i<n; i++) {
    z[i] = alpha*x[i];
}
k = i;    /* i values n */
```

Default behaviour focus on avoiding unnecessary copies.

Guarantee enough computing load

Loop parallelisation introduces an *overhead*: Creation and termination of threads, synchronization.

In simple loops, overhead could be even larger than the workload.

Clause `if`

```
#pragma omp parallel for if(n>5000)
for (i=0; i<n; i++)
    z[i] = a*x[i] + y[i];
```

If the expression is false, the loop is executed sequentially.

This clause can be also used to avoid data dependencies detected at execution time.

Nested loops

We must put the directive before the loop to be parallelised.

Case 1

```
#pragma omp parallel for \
        private(j)
for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
        // loop body
    }
}
```

Case 2

```
for (i=0; i<n; i++) {
    #pragma omp parallel for
    for (j=0; j<m; j++) {
        // loop body
    }
}
```

- In the first case, the iterations of `i` are split; the same thread executes the complete `j` loop.
- In the second case, at each iteration of `i` threads are activated and deactivated; there are `n` synchronizations.

Nested loops - exchange

The best performance is normally obtained when parallelising the external loop:

- When data dependencies prevent from parallelising the external loop, loop exchange may be convenient.

Sequential code

```
for (j=1; j<n; j++)
    for (i=0; i<n; i++)
        a[i][j] = a[i][j] + a[i][j-1];
```

Parallel code with loop exchange

```
#pragma omp parallel for private(j)
for (i=0; i<n; i++)
    for (j=1; j<n; j++)
        a[i][j] = a[i][j] + a[i][j-1];
```

However, these changes may impact on cache locality.

Scheduling

Ideally, all iterations cost the same and each thread has approximately the same number of iterations.

In reality, a workload unbalance may appear, therefore reducing performance.

In OpenMP it is possible to select the best scheduling.

Scheduling can be:

- Static: iterations are assigned to threads at coding time.
- Dynamic: assignement of iterations to threads progressively adapts to the current execution.

The scheudling considers contiguous ranges of iterations (*chunks*).

Scheduling - `schedule` Clause

Syntax of the scheduling clause:

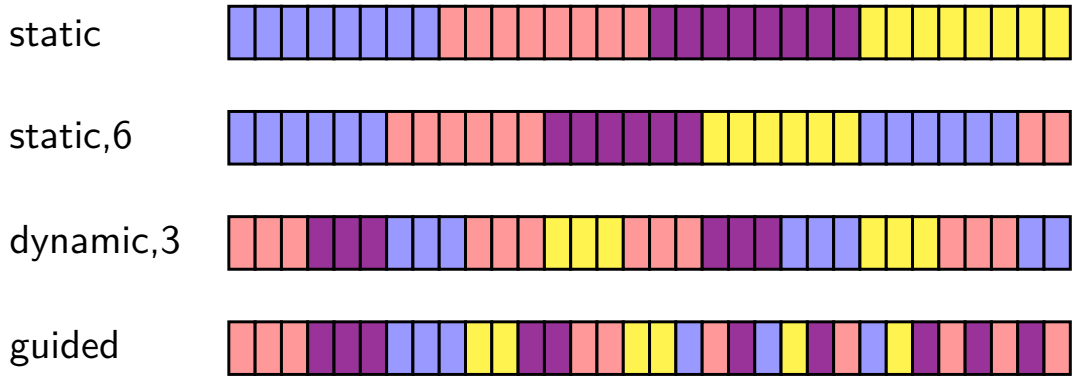
```
schedule(type[,chunk])
```

- `static` (without `chunk`): Each thread receives an iteration range of similar size.
- `static` (woth `chunk`): cyclic (*round-robin*) assignment of ranges of size `chunk`.
- `dynamic` (`chunk` optional, 1 by defalut): ranges are being assigned as required (*first-come, first-served*).
- `guided` (`chunk` minimum size optional): same as `dynamic` but the size of the iteration range decreases exponentially ($\propto n_{rest}/n_{threads}$) with the loop progress.
- `runtime`: the scheduling is defined by the value of the environment variable `OMP_SCHEDULE`.

Scheduling - Example

Example: Loop of 32 iterations executed with 4 threads.

```
$ OMP_NUM_THREADS=4 OMP_SCHEDULE=guided ./prog
```



Section 3

Parallel Regions

- Directive `parallel`
- Workload Splitting

`parallel` Directive

The block is executed in a replicated way

C/C++

```
#pragma omp parallel [clause [clause ...]]
{
    // block
}
```

Some of the accepted clauses are: `private`, `shared`, `default`, `reduction`, `if`

Example - prints as many lines as threads

```
#pragma omp parallel private(myid)
{
  myid = omp_get_thread_num();
  printf("I am thread %d\n",myid);
}
```


Workload splitting

Along with the replicated execution, splitting the work among the threads is also needed often.

- Each thread can work on a part of the data structure.
- Each thread can perform a different operation.

Possible mechanisms for the workload splitting:

- Queue of parallel tasks.
- According to the thread identifier.
- Using OpenMP specific constructions.

Splitting according to the thread identifier

The next functions are used:

- `omp_get_num_threads()`: It returns the number of threads.
- `omp_get_thread_num()`: It returns the thread identifier.

Which can be used to define which part of the workload should be done by each thread.

Identifiers start on 0, and the main thread is always 0.

Example - thread identifiers

```
#pragma omp parallel private(myid)
{
    nthreads = omp_get_num_threads();
    myid = omp_get_thread_num();
    dowork(myid, nthreads);
}
```


Splitting using a queue of parallel tasks

A queue of parallel tasks is a data structured shared among the threads that stores a list of "tasks" to be performed.

- Tasks can be processed concurrently.
- Any task can be run by any thread.

```
int get_next_task() {
    static int index = 0;
    int result;
    #pragma omp critical
    {   if (index==MAXIDX) result=-1;
        else { index++; result=index; }
    }
    return result;
}
...
    int myindex;
    #pragma omp parallel private(myindex)
    {   myindex = get_next_task();
        while (myindex>-1) {
            process_task(myindex);
            myindex = get_next_task();
        }
    }
```

Work splitting constructions

Previous solutions are quite primitive.

- Programmer normally splits the workload.
- Ofuscated programming and complex for large codes.

OpenMP has specific constructions (*work-sharing constructs*)

Three types:

- `for` construct to split iterations of loops.
- Sections to distinguise different parts of the code.
- Code to be executed by a single thread.

Implicit barrier at the end of the block.

`for` construction

Enables splitting automatically the iterations of a loop.

Example of shared loop

```
#pragma omp parallel
{
    ...
    #pragma omp for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

The loop iterations are not replicated but *shared* among the threads.

`parallel` and `for` directives can be combined in one.

Loop construction - Clause `nowait`

When several independent loops take place in the same parallel region, `nowait` removes the implicit barrier at the end of the loop.

Loops without a barrier

```
void a8(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```


`sections` Construction

Suitable for independent but difficult for parallelizing code blocks.

- Individual workload is reduced.
- Each fragment is essentially sequential.

They can be combined with `parallel`

Example of sections

```
#pragma omp parallel sections
{
    #pragma omp section
      Xaxis();
    #pragma omp section
      Yaxis();
    #pragma omp section
      Zaxis();
}
```

A thread may execute more than one section.

Clauses: `private`, `first/lastprivate`, `reduction`, `nowait`.

`single` construction

Code fragments that can be executed by a single thread.

Ejemplo `single`

```
#pragma omp parallel
{
    #pragma omp single
      printf("work1 Starts\n");
    work1();

    #pragma omp single
      printf("work1 Ends\n");

    #pragma omp single nowait
      printf("work1 ended, work2 starts\n");
    work2();
}
```

Some accepted clauses: `private`, `firstprivate`, `nowait`

Section 4

Synchronization

- Mutual Exclusion
- Event-based Synchronization

Race condition (1)

Next example shows a race condition.

Maximum searching

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        cur_max = a[i];
    }
}
```

Sequence with a wrong result:

Thread 0: reads a[i]=20, reads cur_max=15
Thread 1: reads a[i]=16, reads cur_max=15
Thread 0: checks a[i]>cur_max, writes cur_max=20
Thread 1: checks a[i]>cur_max, writes cur_max=16

Race condition (2)

There are cases where concurrent access does not produce a race condition

Example of concurrent access without race condition

```
found = 0;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] == value) {
        found = 1;
    }
}
```

Even if several threads will write simultaneously, the result would be correct.

In general, synchronization mechanisms are needed:

- Mutual exclusion.
- Event-based synchronization.

Mutual exclusion

Mutual exclusion when accessing shared variables prevents any race condition.

OpenMP provides three different constructions:

- Critical sections: `critical` directive.
- Atomic Operations: `atomic` directive.
- Locks: `*_lock` routines.

`critical` directive (1)

in previous example, access in mutual exclusion to the variable. `cur_max` prevents the race condition to happen.

Maximum searching, without race condition

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    #pragma omp critical
    if (a[i] > cur_max) {
        cur_max = a[i];
    }
}
```

When a loop reaches the `if` block (the critical section), it waits until no other thread is executing the block at the same time.

OpenMP guarantees the progress (at least one waiting thread enters in the critical section), but not timeouts.

`critical` directive (2)

In practice, previous example is executed sequentially.

Considering than `cur_max` is never decreased, the following improvement can be introduced.

Improved maximum search

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        #pragma omp critical
        if (a[i] > cur_max)
            cur_max = a[i];
    }
}
```

Second `if` is required since `cur_max` is read out of the critical section.

This solution enteres in the critical section less frequently.

Named `critical` directive

The use of named critical section enables including several, unrelated, critical sections.

Searching minimum and maximum

```
cur_max = -100000;
cur_min = 100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        #pragma omp critical (maxlock)
        if (a[i] > cur_max)
            cur_max = a[i];
    }
    if (a[i] < cur_min) {
        #pragma omp critical (minlock)
        if (a[i] < cur_min)
            cur_min = a[i];
    }
}
```

atomic directive

It is possible to introduce processor-specific instruction to update a memory location (e.g. CMPXCHG in Intel).
In C/C++ syntax is:

```
#pragma omp atomic
x <binop>= expr
```

```
#pragma omp atomic
x++, ++x, x--, --x
```

where <binop> can be +, *, -, /, %, &, |, ^, <<, >>

Example

```
#pragma omp parallel for shared(x, index, n)
for (i=0; i<n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
}
```

The code is much more efficient than the one using `critical` and enables updating the elements of `x` in parallel.

Event-based synchronization

Mutual exclusion constructions provide exclusive access but they do not impose an execution order in the critical sections.

Event-based synchronization enables defining the order of the execution of the loop iterations.

- Barriers: `barrier` directive.
- Ordered sections: `ordered` directive.
- `master` directive.

barrier barrier

When a barrier is reached, threads wait for all the threads to arrive.

Barrier example

```
#pragma omp parallel private(index)
{
    index = generate_next_index();
    while (index>0) {
        add_index(index);
        index = generate_next_index();
    }
    #pragma omp barrier
    index = get_next_index();
    while (index>0) {
        process_index(index);
        index = get_next_index();
    }
}
```

It is used to guarantee that all threads in a phase have been completed before a new parallel phase is started.

ordered directive

It makes a portion of the code to be executed in the original sequential order.

Example ordered

```
#pragma omp parallel for ordered
for (i=0; i<n; i++) {
    a[i] = ...   /* complex computation */
    #pragma omp ordered
    fprintf(fd, "%d %g\n", i, a[i]);
}
```

Restrictions:

- If a parallel loop includes a ordered directive, clause ordered should be added to the loop too.
- Only one ordered section is permitted by iteration.

`master` directive

Identifies a code block inside a parallel region to be executed only by the *master* thread.

Example `master`

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    calc1();     /* make computations */
    #pragma omp master
        printf("Intermediate results: ...\n", ...);
    calc2();     /* make more computations */
}
```

Differences with `single`:

- It does not require all threads to reach this construction.
- There is no implicit barrier (other threads simply skip this code).