CPA – Parallel Computing

Degree in Computer Science

S2. Programming with OpenMP

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

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Year 2024/25

Contents

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), 5.0 (2018)

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 parallelization
- It enables compiler-based optimization

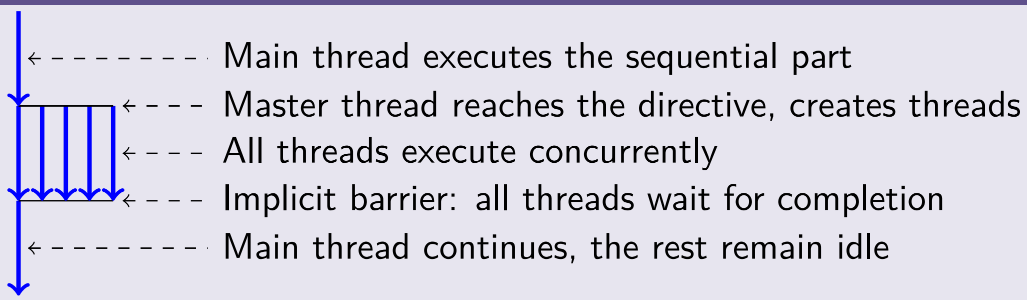
Additionally: functions (see `omp.h`) and environment variables

Execution Model

OpenMP execution models follows the *fork-join* scheme

There are directives to create threads and sharing the work

Scheme



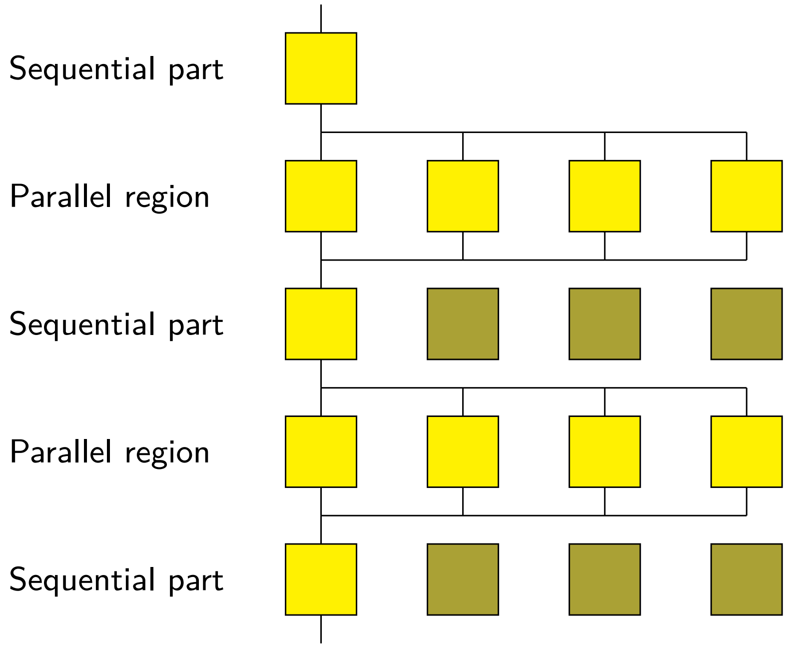
Directives define parallel regions

Other directives/clauses:

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

Execution Model - Threads

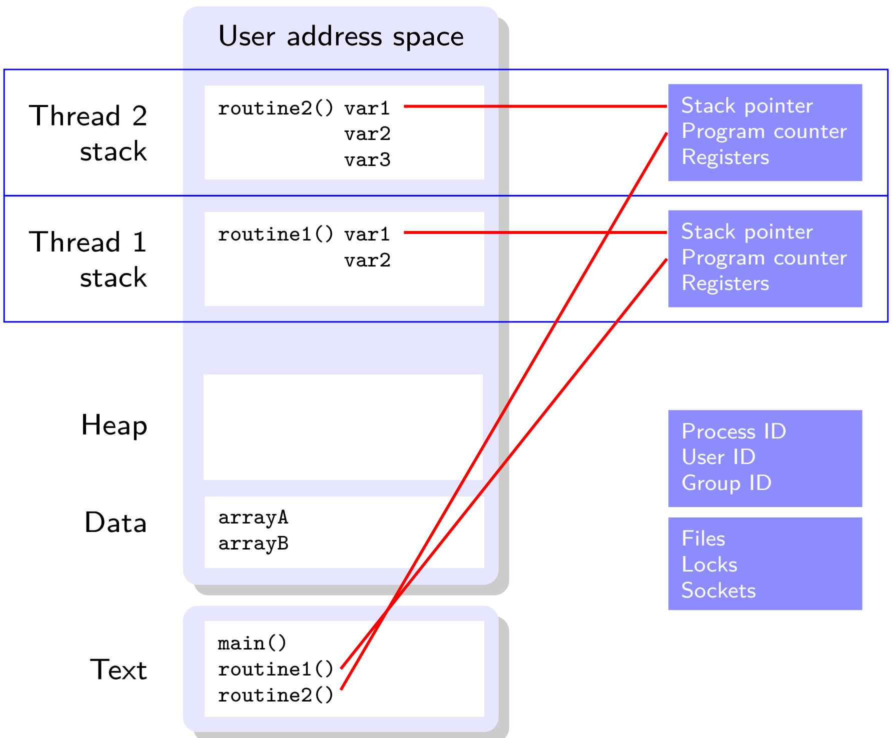
In OpenMP, idle threads are not destroyed, they remain ready for the next parallel region



Threads created by a directive are called team

Execution Model - Memory

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



Syntax

Directives:

```
#pragma omp <directive> [clause [...]]
```

Usage of functions:

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

Conditional compilation: the `_OPENMP` macro contains the date of the supported OpenMP version, e.g. 201107

Compilation:

```
gcc> gcc -fopenmp prg-omp.c
sun> cc -xopenmp -xO3 prg-omp.c
intel> icc -qopenmp 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 `parallel` directive is reached, threads are created (if they have not been created previously)
- Loop iterations are shared among the threads
- By default, all the variables are shared, except for the loop variable (`i`) that is always private
- At the end, all threads synchronize

Number of Threads and Thread Identifier

The number of threads can be specified:

- Using the `num_threads` clause
- Calling the `omp_set_num_threads()` function *before* the parallel region
- At run time, with `OMP_NUM_THREADS`

Useful functions:

- `omp_get_num_threads()`: returns the number of threads
- `omp_get_thread_num()`: 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 Parallelization

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

The parallel for Directive

The loop below the directive is parallelized

C/C++

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

OpenMP imposes restrictions to the loop type, for instance:

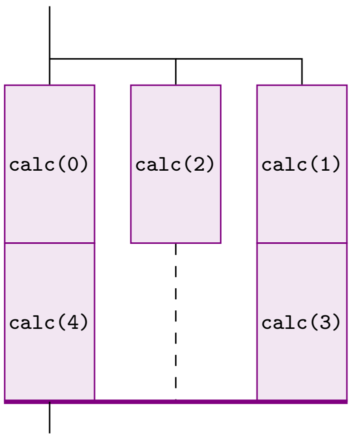
```
for (i=0; i<n && !found; i++)
  if (x[i]==elem) found=1;
```


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 construct

Variables:

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

Variable Scope

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: choose an incorrect scope

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

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

`private`, `shared`, `default`

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

Exceptions (`private`):

- Index variable of the parallelized 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`

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

Wrong: after the loop, only the sum of the main thread is available - moreover, copies of each thread are not initialized

`shared`

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

Wrong: race condition when reading/writing `sum`

`reduction`

To perform reductions with 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 computes part of the sum, at the end all parts are combined in the total sum

It works as a private variable, but:

- At the end, the private values are combined
- Correctly initialized (to the neutral element of the operation)

firstprivate, lastprivate

Private variables are created without an initial value and after the block its value is undefined

- `firstprivate`: initializes to 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 has value n */
```

Default behaviour tries to avoid unnecessary copies

Guarantee Sufficient Work

Loop parallelization introduces an overhead: activation and deactivation of threads, synchronization

In simple loops, the overhead could be even larger than the compute time

if clause

```
    #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 parallelized

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 shared; each thread executes the whole `j` loop
- In the second case, at each iteration of `i` threads are activated and deactivated; there are `n` synchronizations

Nested Loops - Exchange

Usually recommended to parallelize the outer loop

- When data dependencies prevent from parallelizing the outer 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 exchanged loops

```
#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];
```

These changes may have an impact on cache use (locality)

Scheduling

Ideally, all iterations cost the same and each thread gets assigned approximately the same number of iterations

In reality, a workload imbalance may appear, therefore reducing performance

---

In OpenMP it is possible to select the scheduling

Scheduling can be:

- Static: iterations are assigned to threads a priori
- Dynamic: assignment adapts to the current execution

The scheduling is done on the basis of contiguous ranges of iterations (*chunks*)

Scheduling - `schedule` Clause

Syntax of the scheduling clause:

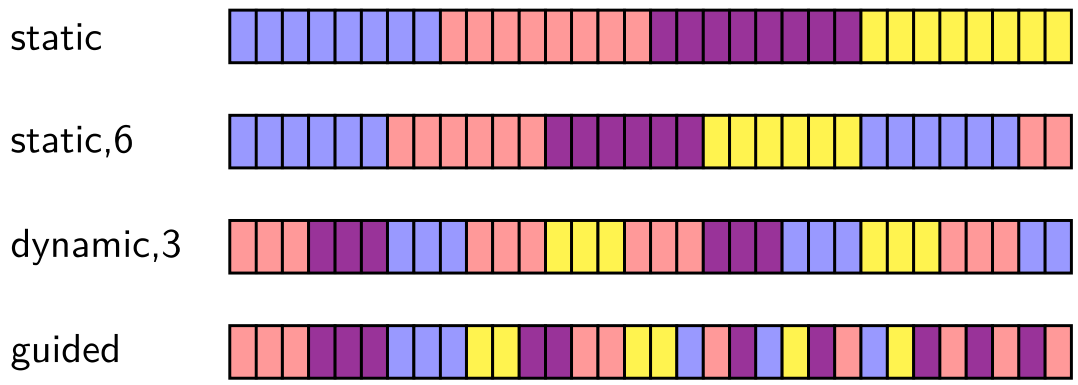
```
schedule(type[,chunk])
```

- `static` (without `chunk`): each thread receives an iteration range of similar size
- `static` (with `chunk`): cyclic assignment (*round-robin*) of ranges of size `chunk`
- `dynamic` (optional `chunk`, 1 by defalut): ranges are being assigned as required (*first-come, first-served*)
- `guided` (optional minimum `chunk`): same as `dynamic` but the size of the iteration range decreases exponentially ($\propto n_{rem}/n_{threads}$) 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

- `parallel` Directive
- Work Sharing

The `parallel` Directive

The block below the directive is executed in a replicated way

C/C++

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

Some of the allowed 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);
}
```


Work Sharing

Along with the replicated execution, it is often necessary to share the work among the threads

- Each thread works on a part of the data structure, or
- Each thread performs a different operation

Possible mechanisms for worksharing:

- Based on the thread identifier
- Parallel task queue
- Using OpenMP specific constructs

Sharing Based on Thread Identifier

We use the following functions:

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

to determine which part of the workload is done by each thread

Example - thread identifiers

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


Sharing Using a Parallel Task Queue

A parallel task queue is a shared data structure containing 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();
        }
    }
```

Worksharing Constructs

The solutions mentioned before are quite primitive

- Programmer is in charge of splitting the workload
- Obscure and complicated code in large programs

OpenMP offers specific worksharing constructs

Three types:

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

Implicit barrier at the end of the block

The `for` construct

Automatically distributes 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 Construct - `nowait` Clause

When several independent loops appear in the same parallel region, `nowait` removes the implicit barrier

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]);
    }
}
```


The `sections` Construct

For independent pieces of code difficult to parallelize

- Individually they represent little work, or
- Each fragment is inherently sequential

It can also 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`

The `single` Construct

Code fragments that must be executed by a single thread

Example of `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 allowed clauses: `private`, `firstprivate`, `nowait`

Section 4

Synchronization

- Mutual Exclusion
- Other Type of Synchronization

Race Condition (1)

The following example illustrates a race condition

Find the maximum value

```
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 write at once, the result is correct

In general, synchronization mechanisms are needed:

- Mutual exclusion
- Other type of synchronization

Mutual Exclusion

Mutual exclusion when accessing shared variables prevents any race condition

OpenMP provides three different constructs:

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

The `critical` Directive (1)

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

Find the maximum value, 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 thread reaches the `if` block (the critical section), it waits until no other thread is executing it at the same time

OpenMP guarantees progress (at least one waiting thread enters the critical section), but not limited wait time

The `critical` Directive (2)

In practice, the previous example is executed sequentially

Considering that `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];
    }
}
```

The second `if` is required since `cur_max` is read outside the critical section

This solution enters the critical section less frequently

Named `critical` Directive

By adding a name, we can have several unrelated critical sections

Find the minimum and maximum values

```
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];
    }
}
```

The atomic Directive

Atomic load-modify-store operations

```
#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 using `critical` and enables updating the elements of `x` in parallel

The barrier Directive

When reaching a barrier, threads wait for the rest 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 a phase has been finished completely before proceeding to the next phase

The ordered Directive

To make sure that a portion of the code of the iterations is 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 an `ordered` directive, the `ordered` clause should also be added to the loop
- Only one `ordered` section is allowed per iteration