

# Parallel programming using OpenMP

## Computer Architecture

J. Daniel García Sánchez (coordinator)  
David Expósito Singh  
Francisco Javier García Blas

ARCOS Group  
Computer Science and Engineering Department  
University Carlos III of Madrid

- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops
- 5 Synchronize with master
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion

# What is OpenMP?

- It is an **language extension** for expressing parallel applications in shared memory systems.
- **Components:**
  - Compiler directives.
  - Library functions.
  - Environment variables.
- Simplifies the way of writing parallel programs.
  - *Mappings* for FORTRAN, C and C++.

# Constructs

## ■ Directives:

**#pragma** omp directive [clause]

- **Example:** Setup the number of threads.

```
#pragma omp parallel num_threads(4)
```

## ■ Library functions:

```
#include <omp.h> // Include to call OpenMP API functions
```

- **Example:** Get the number of threads in use.

```
int n = omp_get_num_threads();
```

# Exercise 1: Sequential

ex1seq.cpp

```
#include <iostream>

int main() {
    using namespace std;

    int id = 0;
    cout << "Hello(" << id << ") ";
    cout << "World(" << id << ")";
    return 0;
}
```

- Print to standard output.

# Exercise 1: Parallel

## ex1par.cpp

```
#include <iostream>
#include <omp.h>

int main() {
    using namespace std;

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        cout << "Hola(" << id << ") ";
        cout << "Mundo(" << id << ")";
    }

    return 0;
}
```

- Compiler flags:
  - gcc: **-fopenmp**
  - Intel Linux: **-openmp**
  - Intel Windows:  
**/Qopenmp**
  - Microsoft Visual Studio: **/openmp**

# Exercise 1

- **Goal:** Verify you have a working environment.
  
- **Activities:**
  - 1 Compile sequential version and run.
  - 2 Compile parallel version and run.
  - 3 Add a call to function `omp_get_num_threads()` to print the number of threads:
    - a) Before the **pragma**.
    - b) Just after **pragma**.
    - c) Within the block.
    - d) Before exiting the program, but outside the block.

# Observations

- A model for multi-threaded shared memory.
  - Communication through shared variables.
- Accidental sharing → **race conditions**.
  - Result depending on threads scheduling.
- Avoiding race conditions.
  - Synchronize to avoid conflicts.
    - Cost of synchronizations.
  - Modify access pattern.
    - Minimize needed synchronizations.



- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops
- 5 Synchronize with master
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion

# Fork-join parallelism

- Sequential application with parallel sections:
  - **Master thread**: Started with main program.
  - A parallel section starts a thread set.
  - Parallelism can be **nested**.
  
- A parallel region is a block marked with the **parallel** directive.
  - **#pragma omp parallel**

# Selecting the number of threads

- Invoking a library function.

## Example

```
// ...  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    // Parallel region  
}
```

- OpenMP directive.

## Example

```
// ...  
#pragma omp parallel num_threads(4)  
{  
    // Parallel region  
}
```

## Exercise 2: Computing $\pi$

- Computing  $\pi$ .

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

- Approximation:

$$\pi \approx \sum_{i=0}^N F(x_i) \Delta x$$

- Adding area of N rectangles:
- **Base:**  $\Delta x$ .
- **Height:**  $F(x_i)$ .

# Exercise 2: Sequential version

## Computing $\pi$ (I)

```

#include <iostream>
#include <iomanip>
#include <chrono>

int main() {
    using namespace std;
    using namespace std::chrono;

    constexpr long nsteps = 10000000;
    double step = 1.0 / double(nsteps);

    using clk = high_resolution_clock;
    auto t1 = clk::now();

```

## Computing $\pi$ (II)

```

    double sum = 0.0;
    for (int i=0; i<nsteps; ++i) {
        double x = (i+0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    double pi = step * sum;

    auto t2 = clk::now();
    auto diff = duration_cast<microseconds>(t2-t1);

    cout << "PI= " << setprecision(10) << pi << endl;
    cout << "Tiempo= " << diff.count() << "us" << endl;

    return 0;
}

```

# Measuring time in C++11

- **include** files:

```
#include <chrono>
```

- Clock type:

```
using clk = chrono::high_resolution_clock;
```

- Get a time point:

```
auto t1 = clk::now();
```

- Time difference (time unit can be specified).

```
auto diff = duration_cast<microseconds>(t2-t1);
```

- Get difference value.

```
cout << diff.count();
```

# Time measurement example

## Example

```
#include <chrono>

void f() {
    using namespace std;
    using namespace std::chrono;

    using clk = chrono::high_resolution_clock;

    auto t1 = clk::now();

    g();

    auto t2 = clk::now();
    auto diff = duration_cast<microseconds>(t2-t1);

    cout << "Time= " << diff.count << "microseconds" << endl;
}
```

# Time measurement in OpenMP

## ■ Time point:

```
double t1 = omp_get_wtime();
```

## ■ Time difference:

```
double t1 = omp_get_wtime();  
double t2 = omp_get_wtime();  
double diff = t2 - t1;
```

## ■ Time difference between two successive ticks:

```
double tick = omp_get_wtick();
```



## Exercise 2

- Create a parallel version from the  $\pi$  sequential version using a **parallel** clause.
- **Observations:**
  - Include time measurements.
  - Print the number of threads in use.
  - Take special care with shared variables.
  - **Idea:** Use an array and accumulate partial sum for each thread in the parallel region.

- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops
- 5 Synchronize with master
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion

# Synchronization mechanisms

- **Synchronization**: Mechanism used to establish constraints on the access order to shared variables.
  - **Goal**: Avoid data races.
  
- **Alternatives**:
  - **High level**: **critical**, **atomic**, **barrier**, **ordered**.
  - **Low level**: **flush**, lock.

# critical

- Guarantees **mutual exclusion**.
- Only a thread at a time can enter the critical region.

## Example

```
#pragma omp parallel  
{  
  for (int i=0;i<max;++i) {  
    x = f(i);  
    #pragma omp critical  
    g(x);  
  }  
}
```

- Calls to **f()** are performed in parallel.
- Only a thread can enter function **g()** at a time.

# atomic

- Guarantees **atomic update** of a single memory location.
- Avoid data races in variable update.

## Example

```
#pragma omp parallel
{
  for (int i=0;i<max;++i) {
    x = f(i);
    #pragma omp atomic
    s += g(x)
  }
}
```

- Calls to **f()** performed in parallel.
- Updates to **s** are *thread-safe*.

## Exercise 3

- Modify program from exercise 2.
- Evaluate:
  - a) Critical section.
  - b) Atomic access.

- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops**
- 5 Synchronize with master
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion

# Parallel for

- **Loop work-sharing**: Splits iterations from a loop among available threads.

## Syntax

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<n; ++i) {
    f(i);
  }
}
```

- **omp for** → For loop work-sharing.
- A private copy of **i** is generated for each thread.
  - Can also be done with **private(i)**



# Example

## Sequential code

```
for (i=0;i<max;++i) { u[i] = v[i] + w[i]; }
```

## Parallel region

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nthreads = omp_get_num_threads();
  int istart = id * max / nthreads;
  int iend = (id==nthreads-1) ?
    ((id + 1) * max / nthreads):max;
  for (int i=istart ; i<iend;++i)
    { u[i] = v[i] + w[i]; }
}
```

## Parallel region + parallel loop

```
#pragma omp parallel
#pragma omp for
for (i=0;i<max;++i)
  { u[i] = v[i] + w[i]; }
```

# Combined construct

- An **abbreviated form** can be used by combining both directives.

## Two directives

```
vector<double> vec(max);  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i=0; i<max; ++i) {  
        vec[i] = generate(i);  
    }  
}
```

## Combined directive

```
vector<double> vec(max);  
#pragma omp parallel for  
for (i=0; i<max; ++i) {  
    vec[i] = generate(i);  
}
```

# Reductions

## Example

```
double sum = 0.0;
vector<double> v(max);
for (int i=0; i<max; ++i) {
    sum += v[i];
}
```

### ■ Effects:

- Private copy for each variable.
- Local copy updated in each iteration.
- Local copies combined at the end.

- A **reduction** performs a reduction on the variables that appear in its list in a parallelized loop.
- Reduction clause: **reduction (op1:var1, op2:var2)**

## Example

```
double sum = 0.0;
vector<double> v(max);
#pragma omp parallel for reduction(+:sum)
for (int i=0; i<max; ++i) {
    sum += v[i];
}
```

# Reduction operation

- Associative operations.

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

- Initial value defined by the operation.

- **Basic operators:**

- **+** (initial value: **0**).
- **\*** (initial value: **1**).
- **-** (initial value: **0**).

- **Advanced operators:**

- **&** (initial value: **0**).
- **|** (initial value: **0**).
- **^** (initial value: **0**).
- **&&** (initial value: **1**).
- **||** (initial value: **0**).

## Exercise 4

- Modify the  $\pi$  computation program.
- Transform the program for obtaining a similar version to the original sequential program.

- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops
- 5 Synchronize with master**
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion

# Barriers

- Allows to synchronize all threads in a point.
  - Wait until all threads arrive to the **barriers**.

## Example

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    v[id] = f(id);
    #pragma omp barrier

    #pragma omp for
    for (int i=0;i<max;++i) {
        w[i] = g(i);
    } // Implicit barrier

    #pragma omp for nowait
    for (int i=0;i<max;++i) {
        w[i] = g(i);
    } // nowait -> No implicit barrier

    v[i] = h(i);
} // Implicit barrier
```

# Single execution: master

- The **master** clause marks a block that is only executed in the *master* thread.

## Example

```
#pragma omp parallel
{
  f(); // In all threads
  #pragma omp master
  {
    g(); // Only in master
    h(); // Only in master
  }
  i(); // In all threads
}
```



## Single execution: single

- The **single** clause marks a block that is only executed in one thread.
  - Does not need to be the master thread.

### Example

```
#pragma omp parallel
{
  f(); // In all threads
  #pragma omp single
  {
    g(); // Only in one thread
    h(); // Only in one thread
  }
  i(); // In all threads
}
```

# Ordering

- An **ordered** region is executed in sequential order.

## Example

```
#pragma omp parallel
{
  #pragma omp for ordered reduction(+:res)
  for (int i=0;i<max;++i) {
    double tmp = f(i);
    #pragma ordered
    res += g(tmp);
  }
}
```

# Simple locks

- Locks in the OpenMP library.
  - Also nested locks.

## Example

```
omp_lock_t l;  
omp_init_lock(&l);  
  
#pragma omp parallel  
{  
    int id = omp_get_thread_num();  
    double x = f(i);  
    omp_set_lock(&l);  
    cout << "ID=" << id << " tmp= " << tmp << endl;  
    omp_unset_lock(&l);  
}  
omp_destroy_lock(&l);
```

## Other library functions

- **Nested locks:**
  - `omp_init_nest_lock()`, `omp_set_nest_lock()`,  
`omp_unset_nest_lock()`, `omp_test_next_lock()`,  
`omp_destroy_nest_lock()`.
- **Processor query:**
  - `omp_num_procs()`.
- **Number of threads:**
  - `omp_set_num_threads()`, `omp_get_num_threads()`,  
`omp_get_thread_num()`, `omp_get_max_threads()`.
- **Test for parallel region:**
  - `omp_in_parallel()`.
- **Dynamic selection of number of threads:**
  - `omp_set_dynamic()`, `omp_get_dynamic()`.

# Environment variables

- Default number of threads:
  - **OMP\_NUM\_THREADS**
  
- Scheduling mode:
  - **OMP\_SCHEDULE**

- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops
- 5 Synchronize with master
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion

# Storage attributes

- Programming model in **shared memory**:
  - **Shared variables**.
  - **Private variables**.
  
- **Shared**:
  - Global variables (file scope and name space)
  - **static** variables.
  - Objects in dynamic memory (**malloc()** and **new**).
  
- **Private**:
  - Local variables in functions invoked from a parallel region.
  - Local variables defined within a block.

# Modifying storage attributes

- Attributes in parallel clauses:
  - **shared**.
  - **private**.
  - **firstprivate**.
- **private** creates a new local copy per thread.
  - Value of copies is not initialized.

## Example

```
void f() {  
    int x = 17;  
    #pragma omp parallel for private(x)  
    for (int i=0;i<max;++i) {  
        x += i; // x not initialized  
    }  
    cout << x << endl; // x==17  
}
```



# firstprivate

- Particular case of **private**.
  - Each private copy is initialized with the value of the variable of the **master** thread.

## Example

```
void f() {  
    int x = 17;  
    #pragma omp parallel for firstprivate (x)  
    for (long i=0;i<maxval;++i) {  
        x += i; // x is initially 17  
    }  
    std::cout << x << std::endl; // x==17  
}
```

# lastprivate

- Pass the value of the private variable of the last **sequential** iteration to the global variable.

## Example

```
void f() {  
    int x = 17;  
    #pragma omp parallel for firstprivate (x) lastprivate (x)  
    for (long i=0;i<maxval;++i) {  
        x += i; // x is initially 17  
    }  
    std::cout << x << std::endl; // x value in iteration i==maxval-1  
}
```

- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops
- 5 Synchronize with master
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion

# Sections

- Defines a set of code sections.
- Each section is passed to a different thread.
- Implicit barrier at the end of the **section** block.

## Example

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    f();
    #pragma omp section
    g();
    #pragma omp section
    h();
  }
}
```

# Loop scheduling

- **schedule(static) | schedule(static,n):**
  - Schedules iteration blocks (size  $n$ ) for each thread.
- **schedule(dynamic) | schedule(dynamic,n):**
  - Each thread takes a block of  $n$  iterations from a queue until all have been processed.
- **schedule(guided) | schedule(guided,n):**
  - Each thread takes an iteration block until all have been processed. Starts with a large block size and it is decreased until size  $n$  is reached.
- **schedule(runtime) | schedule(runtime,n):**
  - Uses scheduling specified by **OMP\_SCHEDULE** or the runtime library.

- 1 Introduction
- 2 Threads in OpenMP
- 3 Synchronization
- 4 Parallel loops
- 5 Synchronize with master
- 6 Data sharing
- 7 Sections and scheduling
- 8 Conclusion**

# Summary

- **OpenMP** allows to annotate sequential code to make use of **fork-join parallelism**.
  - Based in the concept of parallel region.
- Synchronization mechanisms may be **high level** or **low level**.
- Parallel loops combined with reductions allow to preserve original code for many algorithms.
- **Storage attributes** allow to control copies and data sharing in parallel regions.
- OpenMP offers multiple scheduling approaches.

# References

## ■ Books:

- *An Introduction to Parallel Programming*. P. Pacheco. Morgan Kaufmann, 2011. (Cap 5).
- *Multicore and GPU Programming*. G. Barlas. Morgan Kaufmann. 2014. (Cap 4).

## ■ Web:

- **OpenMP**: <http://www.openmp.org>.
- **Lawrence Livermore National Laboratory Tutorial**: <https://computing.llnl.gov/tutorials/openMP/>.



# Parallel programming using OpenMP

## Computer Architecture

J. Daniel García Sánchez (coordinator)  
David Expósito Singh  
Francisco Javier García Blas

ARCOS Group  
Computer Science and Engineering Department  
University Carlos III of Madrid