# Parallel Programming Course. OpenMP.

Obolenskiy Arseniy, Nesterov Alexander

Nizhny Novgorod State University

April 2, 2025

# Today

# What is OpenMP?

- Brief overview
- Importance in parallel computing
- Use cases

# What is OpenMP?

- Open standard for parallel programming
- Supports multi-platform shared-memory multiprocessing
- Used in computational science, engineering, and simulations

# Your First OpenMP Program

```c
#include <omp.h>
#include <stdio.h>

int main() {
  #pragma omp parallel
  {
    printf("Hello from thread %d\n", omp_get_thread_num());
  }
  return 0;
}
```

# OpenMP Features

- Compiler directives
- Runtime library functions
- Environment variables

# Basic OpenMP Features

OpenMP provides three primary mechanisms to express parallelism clearly and effectively:

- Compiler directives (`#pragma omp`)
- Runtime library functions
- Environment variables

# Compiler Directives

Compiler directives guide the compiler to parallelize sections of code. General syntax:

```
#pragma omp directive [clauses]
```

Commonly used directives:

- `parallel` — Creates parallel threads.
- `for` — Parallelizes loop iterations.
- `section` — Defines parallel sections.

Example:

```
#pragma omp parallel for
for(int i = 0; i < N; i++) {
  a[i] = b[i] + c[i];
}

```

# The parallel Directive

The `parallel` directive starts a parallel region executed by multiple threads.
Syntax:

```
#pragma omp parallel [clauses]
{
  // Code executed in parallel
}
```

Example:

```
#pragma omp parallel
{
  printf("Thread %d is running\n", omp_get_thread_num());
}

```

The `for` directive parallelizes loops among threads.
To take an effect it must be within an existing parallel region:
Syntax:

```
#pragma omp for [clauses]
for (init; condition; increment) {
  // Loop body
}
```

Example:

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < N; i++) {
    array[i] = compute(i);
  }
}

```

# The parallel for Directive

Combines the parallel and for directives, simplifying syntax.
Syntax:

```
#pragma omp parallel for [clauses]
for (init; condition; increment) {
  // Loop body
}
```

Example:

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
  data[i] = process(i);
}

```

This is equivalent to a parallel region with a single for loop.

# Clauses: private and shared

Applicable to directives:

- parallel, for, parallel for

Controls the scope of variables:

- shared(var): Variable shared among threads (default).
- private(var): Each thread gets its own private copy.

Example (parallel for):

```
int temp = 0;
#pragma omp parallel for private(temp)
for(int i = 0; i < N; i++) {
    temp = compute(i);
    result[i] = temp;
}
```

# Clause: schedule

Applicable to directives:

- for, parallel for

Controls iteration distribution among threads:
Syntax:

```
schedule(type, chunk_size)
```

Types:

- static (default)
- dynamic
- guided

Example (parallel for):

```
#pragma omp parallel for schedule(dynamic,4)
for(int i = 0; i < N; i++) {
  heavy_computation(i);
}

```

# Clause: reduction

Applicable to directives:

- parallel, for, parallel for

Combines thread results safely into one variable.

Syntax:

```
reduction(operator: variable)
```

Common operators: +, -, *, max, min

Example (parallel for):

```c
int total = 0;
#pragma omp parallel for reduction(+:total)
for(int i = 0; i < N; i++) {
    total += array[i];
}
printf("Sum = %d\n", total);

```

# Clause: num_threads

Applicable to directives:

- parallel, parallel for

Sets number of threads explicitly:

Syntax:

```
num_threads(number_of_threads)
```

Example (parallel for):

```
#pragma omp parallel for num_threads(8)
for(int i = 0; i < N; i++) {
    compute(i);
}

```

Overrides default thread count and environment settings.

# Sections

Use the sections directive to run independent tasks in parallel:

```
#pragma omp parallel sections
{
  #pragma omp section
  {
    compute_task_A();
  }
  #pragma omp section
  {
    compute_task_B();
  }
  #pragma omp section
  {
    compute_task_C();
  }
}

```

# Synchronization and Data Sharing

- Barrier
- Critical sections
- Atomic operations
- Built-in reduction operation
- OpenMP locks (similar to mutex)

# OpenMP Barrier (barrier)

Synchronizes threads explicitly; threads wait at the barrier until all threads arrive.
Syntax:

```
#pragma omp barrier
```

Example:

```
#pragma omp parallel
{
    compute_part1();

    #pragma omp barrier  // All threads wait here

    compute_part2();  // Starts only after all threads
                      // finish compute_part1()
}

```

Barrier ensures correct sequence in parallel regions.

# Critical Sections (critical)

**Purpose:** Ensures only one thread executes a code region at a time, preventing race conditions.
Syntax:

```
#pragma omp critical [name]
{
  // critical section
}
```

Example:

```
#pragma omp parallel
{
  #pragma omp critical
  {
    sum += compute_value();
  }
}

```

Usage of this directive ensures safe access to the shared variables within block boundaries.

# Named Critical Sections

Multiple named critical sections prevent unnecessary waiting.

**Syntax:**

```
#pragma omp critical(name)
{
  // named critical section
}
```

Example:

```
#pragma omp parallel
{
  #pragma omp critical(update_sum)
  {
    sum += compute_sum();
  }

  #pragma omp critical(update_max)
  {
    max_val = max(max_val, compute_val());
  }
}
```

Different named critical regions do not block each other.

Purpose: Enforces atomicity of a single memory operation.
Syntax:

```
#pragma omp atomic
  expression;
```

Supported operations: +, -, *, /, &, |, ^, ++, -
Example:

```
#pragma omp parallel for
for(int i = 0; i < N; i++) {
  #pragma omp atomic
  count += array[i];
}
```

It is more efficient than `critical` for simple arithmetic.

# `critical` vs. `atomic`

Key differences between these synchronization methods:

- Critical Sections:
    - Allows arbitrary blocks of code.
    - More general-purpose, but potentially slower due to locking overhead.
- Atomic Operations:
    - Limited to single, simple memory operations.
    - Faster, uses hardware-level instructions.

Use `atomic` for simple operations, `critical` for more complex sections.

# OpenMP Functions: Thread Management

Control and query the number of threads.

Commonly used functions:

- `omp_set_num_threads(int n)`
- `omp_get_num_threads()`
- `omp_get_thread_num()`
- `omp_get_max_threads()`

Example:

```
omp_set_num_threads(4);
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  printf("Hello from thread %d\n", tid);
}

```

# OpenMP Locks

Purpose: Provide explicit, fine-grained control of synchronization for critical regions.

API:

- omp_init_lock() — Initializes a lock
- omp_set_lock() — Locks (blocks if unavailable)
- omp_unset_lock() — Releases a lock
- omp_destroy_lock() — Frees lock resources

Example:

```
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel for
for(int i = 0; i < N; i++) {
    omp_set_lock(&lock);
      sum += compute(i);
    omp_unset_lock(&lock);
}

omp_destroy_lock(&lock);
```

Explicit locking provides precise synchronization control.

# OpenMP Functions: Timing

Useful functions for measuring execution time:

- omp_get_wtime() — returns current time in seconds.
- omp_get_wtick() — precision of timer.

Example:

```
double start = omp_get_wtime();

#pragma omp parallel for
for(int i = 0; i < N; i++) {
  heavy_computation(i);
}

double end = omp_get_wtime();
printf("Elapsed time: %f seconds\n", end-start);

```

# Environment Variables in OpenMP

Environment variables control OpenMP runtime behavior without recompilation.
Common environment variables include:

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_DYNAMIC
- OMP_NESTED

# OMP_NUM_THREADS

Specifies the default number of threads.
Example usage:

```
export OMP_NUM_THREADS=8
./my_program
```

Overrides default or explicitly set number of threads within code unless set otherwise by `num_threads` clause.

# OMP_SCHEDULE

Sets default scheduling policy for loops with the `schedule(runtime)` clause.

Syntax:

```
export OMP_SCHEDULE="type,chunk"
```

Example:

```
export OMP_SCHEDULE="dynamic,4"
./my_program
```

Affects loops declared as:

```
#pragma omp parallel for schedule(runtime)
```

# OMP_DYNAMIC and OMP_NESTED

Enables dynamic thread adjustment (true/false).
Example:

```
export OMP_DYNAMIC=true
```

# OMP_DYNAMIC and OMP_NESTED

Allows nested parallelism (true/false).
Example:

```
export OMP_NESTED=true
```

Nested parallel regions:

```
#pragma omp parallel num_threads(2)
{
  #pragma omp parallel num_threads(2)
  {
    // Nested region, total 4 threads
  }
}

```

Thank You!

# References

- OpenMP Official Specification:
  https://www.openmp.org/specifications/
- OpenMP Reference Guides:
  https://www.openmp.org/resources/refguides/