PMC Lecture 10

Gianmaria Romano

11 November 2025

Chapter 5

Shared memory systems

5.1 The main aspects of the OpenMP API

OpenMP is a multiprocessing API that is normally used to implement shared-memory systems.
In particular, this API aims to decompose a sequential program into simpler components that can be performed in parallel through some compiler directives.
For this reason, OpenMP programs, which are implemented using the Fork/join design pattern, are said to be "globally sequential, locally parallel".

5.2 Pragmas

A pragma is a special preprocessor directive that gives additional information to the compiler in order to instruct it to perform non-basic tasks for parallelizing a sequential code.
N.B.: If the compiler does not know how to read a pragma, it will simply ignore it for portability purposes.
Example: The "parallel" pragma is the simplest directive as it is able to take a sequential code and have it performed in parallel by more threads, which will work on different portions of the data.
For instance, the following code implements a function to print "Hello, World!" from more threads using the OpenMP API:

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

void Hello(void);

int main(int argc, char* argv[]){
    int thread_count = strtol(argv[1], NULL, 10); // Convert the input into a long.

    # pragma omp parallel num_threads (thread_count)
    Hello ();

    return 0;
}

void Hello (void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

5.2.1 Using clauses to specify the number of threads

A clause is a text that allows to modify the behaviour of a compiler directive in order to let the programmer specify how many threads should be used to run an OpenMP program.
Generally speaking, it is possible to specify the number of threads needed to run an OpenMP program using one of the following approaches:

- **Universally:** the number of threads is specified using an environmental variable that indicates the default number of threads for all the parallel regions within the program.
- **Program level:** the number of threads is specified within the program using a dedicated OpenMP function.
- **Pragma level:** the number of threads is specified using a directive that, through a dedicated clause, indicates the number of threads that can be used for a specific parallel region.

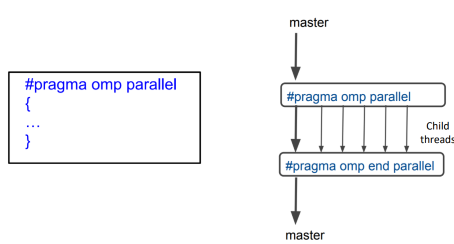
Keep in mind, however, that the actual number of threads that can be used is typically bounded by system and hardware limitations.

5.2.2 Using pragmas to implement mutual exclusion

In order to avoid race conditions, the "critical" pragma allows to implement mutual exclusion to make sure that only one thread at a time can execute the following parallel section.

5.3 Thread terminology in OpenMP APIs

A group of threads that executes a parallel section is known as a "team".
Among the threads of a team, the starting thread, which is known as the "master thread", typically acts as the "parent thread" as, through a directive, it is able to create "child threads" to form the team.



N.B.: When working with variables defined within a parallel construct, each thread works on its own "local" copy, meaning that modifications made by one thread will not be visible to the other threads.

Example: The following code provides an OpenMP implementation for the trapezoidal rule used to approximate integrals:

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0;
    double a, b;
    int n;
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b and n: \n");
    scanf("%lf %lf %d", &a, &b, &n);

    #pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With %d trapezoids, the estimate of the integral is %.14e.\n", n, global_result)

    return 0;
}
```

```c
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b - a) / n;
    local_n = n / thread_count;
    local_a = a + (my_rank * local_n * my_rank);
    local_b = local_a + (local_n * h);
    my_result = (f(local_a) + f(local_b)) / 2.0; // Assume f is a well-known function.
    for (i = 1; i <= local_n - 1; i++) {
        x = local_a + (i * h);
        my_result += f(x);
    }
    my_result = my_result * h;

    #pragma omp critical
    *global_result_p += my_result; // Critical section.
}
```

5.4 Variable scope in OpenMP programs

In the context of serial programming, the scope of a variable simply refers to all sections of a program where that variable can be accessed and used.
Regarding OpenMP programs, however, the scope of a variable indicates both its accessibility and whether the variable is public, which means that all threads can access it, or private, which means that each thread accesses an independent instance.

5.4.1 Scope-modifying clauses

The "default" clause allows the programmer to specify the default scopes of the variables within a parallel construct.
In addition, starting from a situation in which no variable is accessible, the following clauses can be used to handle variable scope:

- **Shared:** the variable is shared among all threads.
  Observe that this is the default behaviour for variables defined outside of a parallel block, although one can explicitly indicate it using the "default" clause.

- **Reduction:** the variable participates in a reduction operation where each thread has a private copy of the variable and the final result is reduced into a single variable found after the parallel block.
- **Private:** each thread receives its separate instance of the variable.
  Keep in mind, however, that, since these copies will be uninitialized, they will not maintain the value of the original variable outside the parallel block.

5.5 Reduction operations in OpenMP programs

In the context of OpenMP programs, a reduction is a computation that repeatedly applies a reduction operator, such as addition or multiplication, to a sequence of operands in order to recover a single result that is obtained by grouping the intermediate results into a dedicated reduction variable.
Reductions are widely used in parallel programming because they can be easily implemented using the "reduction" clause, which allows the OpenMP API to handle computations more efficiently compared to a critical section.