TP3 — OpenMP: Introduction

Parallel Programming Lab Report

February 2026

Mouad Elansari

College of Computing — UM6P

Contents

Exercise 1: Hello World with OpenMP

1.1 Objective

Write an OpenMP program that displays the rank of each thread, then test it with different numbers of threads.

1.2 Implementation

The program creates a parallel region with `#pragma omp parallel num_threads(N)` and uses `omp_get_thread_num()` to retrieve each thread's rank.

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

int main(){
    #pragma omp parallel num_threads(5)
    {
        int rank;
        rank = omp_get_thread_num();
        printf("Hello from the rank %d thread \n", rank);
    }
    return 0;
}
```

Listing 1: ex1.c — Hello World with OpenMP

1.3 Results

The program was tested with **5 threads** and **20 threads**.

Output with 5 threads:

```
Hello from the rank 4 thread
Hello from the rank 0 thread
Hello from the rank 1 thread
Hello from the rank 2 thread
Hello from the rank 3 thread
```

Output with 20 threads (excerpt):

```
Hello from the rank 19 thread
Hello from the rank 2 thread
Hello from the rank 4 thread
Hello from the rank 5 thread
Hello from the rank 6 thread
Hello from the rank 1 thread
Hello from the rank 3 thread
... (all 20 threads print, in non-deterministic order)
Hello from the rank 0 thread
```

1.4 Analysis

The output order is non-deterministic in both cases: threads are scheduled independently by the OS, so no fixed ordering exists between them. All N expected threads execute,

but never in the same sequence across runs. This is a fundamental property of parallel execution — ordering between independent threads requires explicit synchronisation.

Exercise 2: Parallelizing PI Calculation — `parallel` Construct

2.1 Objective

Parallelize the PI computation using `#pragma omp parallel` (without `parallel for`), with careful attention to shared vs. private variables.

2.2 Background

The numerical integration formula used is the midpoint rectangle rule:

$$\pi \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{4}{1 + \left(\frac{i + 0.5}{N}\right)^2}$$

The step count was set to $N = 10^8$ to make timing differences clearly measurable.

2.3 Implementation

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

static long num_steps = 100000000;
double step;

int main () {
    double start = omp_get_wtime();
    int i; double x, pi, sum = 0.0;
    step = 1.0 / (double) num_steps;

    #pragma omp parallel private(x)
    {
        #pragma omp for reduction(+:sum)
        for (i = 0; i < num_steps; i++) {
            x = (i + 0.5) * step;
            sum = sum + 4.0 / (1.0 + x * x);
        }
    }

    pi = step * sum;
    printf("PI : %f \n", pi);
    double end = omp_get_wtime();
    printf("Time : %f \n", end - start);
}
```

Listing 2: Parallel PI with `parallel` + `for` + `reduction`

Variable classification:

- `x` — **private**: each thread computes its own x value per iteration independently.

- `sum` — **reduction(+:sum)**: each thread keeps a private partial sum; OpenMP accumulates them atomically at the barrier, avoiding race conditions.
- `step`, `num_steps` — **shared** (read-only): identical for all threads.

2.4 Results

Version	PI value	Time (s)
Without parallel	3.141593	0.342378
With parallel	3.141593	0.167890

Table 1: PI computation time with $N = 10^8$ steps

2.5 Analysis

The parallel version achieves approximately **2**× speedup over the serial version. The `reduction` clause is critical: without it, multiple threads would concurrently write to `sum`, producing a *race condition* and an incorrect result. Both versions yield the same PI value to 6 decimal places, confirming numerical correctness.

Exercise 3: PI with Loop Construct — `parallel for`

3.1 Objective

Parallelize the serial PI program by adding only **one line**, using the combined `#pragma omp parallel for` directive.

3.2 Implementation

```
    #pragma omp parallel for   /* <-- only line added */
        for (i = 0; i < num_steps; i++) {
            x = (i + 0.5) * step;
            sum = sum + 4.0 / (1.0 + x * x);
        }
```

Listing 3: Minimal parallelisation — one pragma line added

3.3 Analysis

The `#pragma omp parallel for` directive combines the `parallel` and `for` constructs into a single line, making it the minimal change to the serial code. However, this version contains a latent **race condition** on `sum`: multiple threads write to it concurrently with no reduction clause, so results may be incorrect. Exercise 2 shows the fully correct approach using `reduction(+:sum)` and `private(x)`. The lesson is that minimal changes enable parallelism quickly, but correctness still requires explicit reasoning about data sharing.

Exercise 4: Parallelizing Matrix Multiplication with OpenMP

4.1 Objective

Parallelize a matrix multiplication $C = A \times B$ (with $m = n = 200$) using `collapse(2)`, then analyse speedup and efficiency over 1, 2, 4, 8, and 16 threads.

4.2 Implementation

All initialization loops and the multiplication itself are parallelized over the outer two indices using `collapse(2)`, which fuses the two loops into one larger iteration space for better load balancing.

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

int main() {
    int m = 200, n = 200;
    double *a = (double *)malloc(m * n * sizeof(double));
    double *b = (double *)malloc(n * m * sizeof(double));
    double *c = (double *)malloc(m * m * sizeof(double));

    double start = omp_get_wtime();

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            a[i * n + j] = (i + 1) + (j + 1);

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            b[i * m + j] = (i + 1) - (j + 1);

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            c[i * m + j] = 0;

    // Outer two loops are independent -- safe to collapse
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            for (int k = 0; k < n; k++)
                c[i * m + j] += a[i * n + k] * b[k * m + j];

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

    free(a); free(b); free(c);
    return 0;
}
```

Listing 4: ex.c — Matrix Multiplication with `collapse(2)`

4.3 Results

Threads (p)	**Time (s)**	**Speedup** $S(p)$	**Efficiency** $E(p)$
1	0.047008	1.000	1.000
2	0.037259	1.262	0.631
4	0.086735	0.542	0.135
8	0.043462	1.082	0.135
16	0.042478	1.107	0.069

Table 2: Matrix multiplication performance ($m = n = 200$). $S(p) = T_1/T_p$, $E(p) = S(p)/p$.

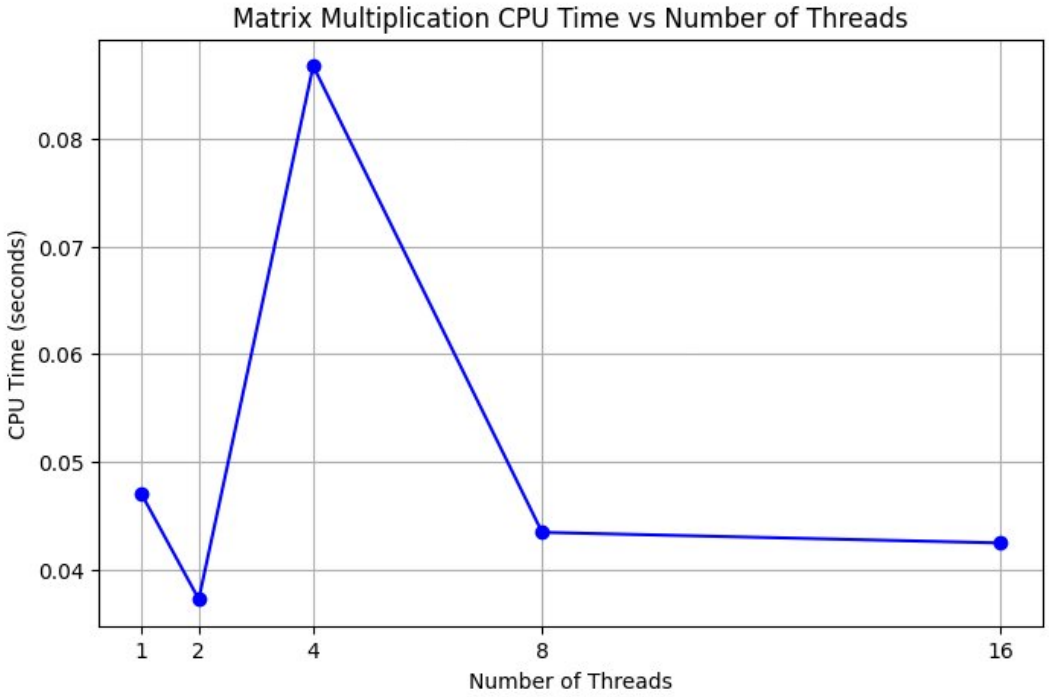

Figure 1: CPU execution time vs. number of threads (matrix multiplication, 200×200)

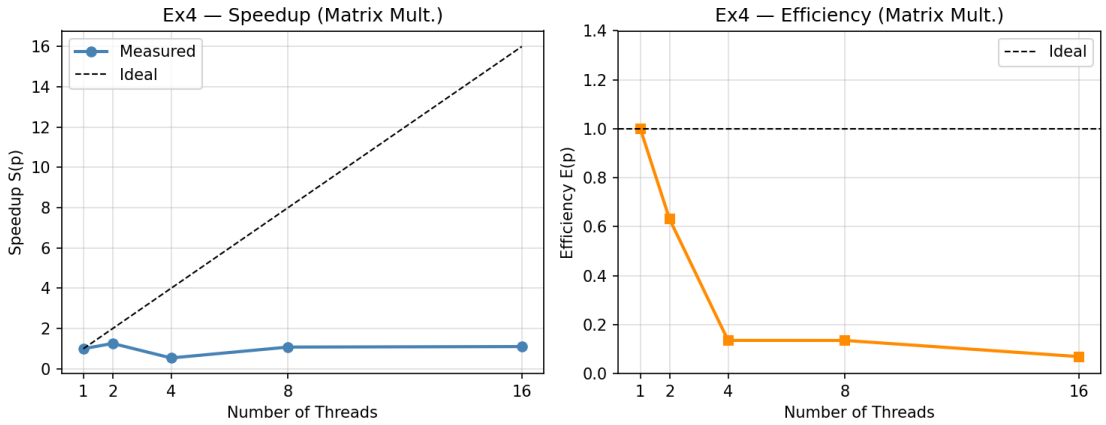

Figure 2: Speedup and efficiency for matrix multiplication ($m = n = 200$)

4.4 Analysis

The results show irregular behaviour across thread counts for this small matrix:

- At **2 threads** a modest speedup of 1.26× is achieved.
- At **4 threads** performance *degrades* (time spikes to 0.087 s), likely due to thread creation overhead and cache contention dominating on a small problem.
- From 8 onwards, execution time stabilises around 0.043 s, suggesting the computation is short enough that spawning threads dominates but all finish roughly in parallel.
- Efficiency drops sharply: at 16 threads it reaches only 6.9%, meaning 93% of thread capacity is wasted on overhead.

For larger matrices the computation-to-overhead ratio increases and both speedup and efficiency would improve significantly.

Exercise 5: Parallelizing the Jacobi Method with OpenMP

5.1 Objective

Solve a linear system $Ax = b$ using the Jacobi iterative method in parallel ($n = 120$, diagonal dominance = 80), then measure speedup and efficiency over 1, 2, 4, 8, 16, and 32 threads.

5.2 Serial Version Overview

The serial code (`exx.c`) implements the standard Jacobi update:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ji}\, x_j^{(k)} \right)$$

Convergence is tested each iteration using the infinity norm scaled by n:

$$\text{norme} = \frac{\|x^{(k+1)} - x^{(k)}\|_\infty}{n} \leq \varepsilon_{\text{machine}}$$

5.3 Parallel Implementation

```
while (1) {
    iteration++;

    // Each row i is independent of other rows => safe to parallelise
    #pragma omp parallel for private(j)
    for (i = 0; i < n; i++) {
        x_courant[i] = 0;
        for (j = 0; j < n; j++) {
            if (j == i) continue;
            x_courant[i] += a[j * n + i] * x[j];
        }
        x_courant[i] = (b[i] - x_courant[i]) / a[i * n + i];
    }

```

```
    // Parallel max-reduction for convergence check
    double absmax = 0;
    #pragma omp parallel for reduction(max:absmax)
    for (i = 0; i < n; i++) {
        double curr = fabs(x[i] - x_courant[i]);
        if (curr > absmax)
            absmax = curr;
    }

    norme = absmax / n;
    if ((norme <= DBL_EPSILON) || (iteration >= n)) break;
    memcpy(x, x_courant, n * sizeof(double));
}
```

Listing 5: parallel_ex.c — key parallelised sections of Jacobi

Parallelisation decisions:

- The outer loop over `i` is parallelised because each $x_i^{(k+1)}$ depends only on $x^{(k)}$ (already fixed at iteration start), so all rows are **independent**.
- `j` is declared **private** to avoid sharing the inner-loop index between threads.
- The norm computation uses **reduction(max:absmax)** to safely find the global maximum convergence criterion across all threads.

5.4 Results

Threads (p)	Elapsed time (s)	Speedup $S(p)$	Efficiency $E(p)$
1	0.002917	1.000	1.000
2	0.003378	0.864	0.432
4	0.002064	1.413	0.353
8	0.007389	0.395	0.049
16	0.035070	0.083	0.005
32	0.047580	0.061	0.002

Table 3: Jacobi method performance ($n = 120$). $S(p) = T_1/T_p$, $E(p) = S(p)/p$.

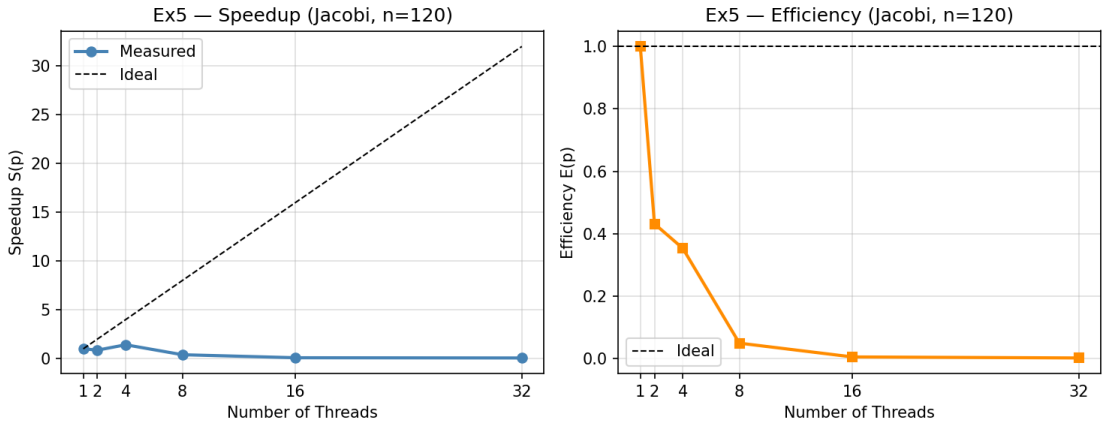

Figure 3: Speedup and efficiency for the Jacobi method ($n = 120$)

5.5 Analysis

The Jacobi results clearly demonstrate the **overhead-dominated** regime for small problem sizes:

- The best result is at **4 threads** with speedup 1.41×.
- With **8 threads** speedup collapses to 0.40× — the parallel version is already *slower* than serial.
- With **16 threads** the elapsed time is 12× worse than the single-thread baseline, and with 32 threads it degrades further to 0.048 s.
- This behaviour is explained by the **very small per-thread workload**: only $\sim 120/p$ row updates per iteration. The cost of spawning threads at every loop entry inside the `while` loop, plus the implicit barrier at the end of each `#pragma omp for`, far outweighs the computation.
- For a larger system (e.g. $n \geq 1000$), the computation would dominate and the parallel version would scale much more favourably.

Conclusion

This lab covered the core OpenMP building blocks for shared-memory parallelism. Exercise 1 established that threads execute concurrently and in non-deterministic order. Exercises 2 and 3 demonstrated PI parallelisation, showing both the correct approach (`reduction`, `private`) and the minimal single-line approach, with the parallel version achieving $\sim 2\times$ speedup on 10^8 iterations. Exercise 4 (Matrix Multiplication) and Exercise 5 (Jacobi Method) both illustrated the same fundamental principle: for small problem sizes, thread-management overhead dominates and parallel performance is poor or even negative. Efficiency dropped to below 10% for 8+ threads in both cases. The overarching lesson is that **problem size is the decisive factor**: OpenMP parallelism is only beneficial when the computation per thread is large enough to amortise the cost of thread creation, synchronisation, and memory-bandwidth contention.