Université Mohammed VI Polytechnique
College of Computing

TP4
Report

Module: Foundations of Parallel Computing
Instructor: Imad Kissami

Prepared by:
Malak Kably

Academic Year: 2025–2026

Contents

1 Exercise 1: Work Distribution with Parallel Sections

1.1 Implementation

The goal was to parallelize independent computations (sum, max, standard deviation) using `#pragma omp parallel sections`. See `TP4/ex1/code_sections.c` for complete implementation.

1.1.1 Modified Code

```c
// Parallel sections for independent computations
#pragma omp parallel sections
{
    // Section 1: Compute sum
    #pragma omp section
    {
        sum = 0.0;
        for (int i = 0; i < N; i++)
            sum += A[i];
    }

    // Section 2: Compute max
    #pragma omp section
    {
        max = A[0];
        for (int i = 0; i < N; i++)
            if (A[i] > max)
                max = A[i];
    }

    // Section 3: Compute standard deviation
    #pragma omp section
    {
        stddev = 0.0;
        for (int i = 0; i < N; i++)
            stddev += (A[i] - mean) * (A[i] - mean);
    }
}
```

1.2 Justification

Each section executes an independent computation on the array `A`. The `sections` directive allows these tasks to be distributed among available threads, with each thread executing one section. This approach is efficient when tasks are independent and have similar computational costs.

2 Exercise 2: Master and Single Thread Directives

2.1 Implementation

Implemented OpenMP directives to control thread execution: `#pragma omp master` for initialization and `#pragma omp single` for printing. See `TP4/ex2/code_modified.c` and `TP4/ex2/code.c` for full code.

2.1.1 Modified Code

```
#pragma omp parallel
{
    #pragma omp master
    {
        start = (double) clock() / CLOCKS_PER_SEC;

        /* Initialization */
        init_matrix(N, A);
    }

    #pragma omp single
    {
        print_matrix(N, A);
    }
}

/* Sum computation */
sum = sum_matrix(N, A);
```

Key changes:

- Used 1D array with row-major indexing: `A[i*n + j]`
- `master`: Only master thread initializes matrix
- `single`: Any single thread prints (first to reach)
- `reduction(+:sum)`: Parallel reduction for sum computation

2.2 Compilation and Execution

```
gcc -fopenmp -o sequential code.c
gcc -fopenmp -o parallel code_modified.c
./sequential > output_sequential.txt
./parallel > output_parallel.txt
```

2.3 Results Comparison

Sequential Output (`TP4/ex2/output_sequential.txt`):

```
Sum...999000000.000000
Execution_time...0.009131 seconds
```

Parallel Output (`TP4/ex2/output_parallel.txt`):

```
Sum...999000000.000000
Execution_time...0.295051 seconds
```

Both versions produce identical results (sum = 999000000.0), confirming correct synchronization. However, the parallel version is slower due to the overhead of thread creation, matrix printing by a single thread, and the small problem size ($N = 1000$), where parallelization overhead exceeds computation time. This demonstrates that parallelization is only beneficial when computational costs outweigh synchronization and thread management overhead.

3 Exercise 3: Load Balancing with Parallel Sections

3.1 Implementation

Three computational tasks with different workloads were parallelized using OpenMP sections. See `TP4/ex3/code_solution.c`.

3.1.1 Modified Code

```c
const int N = 100000000;

void task_light(int N) {
    double x = 0.0;
    for (int i = 0; i < N; i++)
        x += sin(i * 0.001);
}

void task_moderate(int N) {
    double x = 0.0;
    for (int i = 0; i < N; i++)
        x += sqrt(i * 0.5) * cos(i * 0.001);
}

void task_heavy(int N) {
    double x = 0.0;
    for (int i = 0; i < N; i++)
        x += sqrt(i * 0.5) * cos(i * 0.001) * sin(i * 0.0001);
}

int main() {
    double start, end;

    start = (double) clock() / CLOCKS_PER_SEC;

    #pragma omp parallel sections
    {
        #pragma omp section
```

```
        {
            task_light(N);
        }

        #pragma omp section
        {
            task_moderate(N);
        }

        #pragma omp section
        {
            task_heavy(N);
        }
    }

    end = (double) clock() / CLOCKS_PER_SEC;
    printf("Execution_time...%lf seconds\n", end - start);

    return 0;
}
```

3.2 Solution Approach

- **Task distribution**: Three sections for light, moderate, and heavy tasks
- **Load balancing**: OpenMP runtime assigns sections to threads

3.3 Compilation and Execution

```
gcc -fopenmp -O2 -o code_solution code_solution.c -lm
./code_solution > output.txt
```

3.4 Results

From `TP4/ex3/output.txt`:

```
Execution_time...5.276509 seconds
```

4 Exercise 4: Dense Matrix-Vector Multiplication and Barrier Analysis

4.1 Problem Description

Implement and analyze three versions of dense matrix-vector multiplication (DMVM) with different synchronization strategies:

- **Version 1**: Implicit barrier (baseline)
- **Version 2**: Dynamic scheduling with `nowait`

- **Version 3**: Static scheduling with `nowait`

Matrix dimensions: $n = 40000$ columns, $m = 600$ rows. Total FLOPs: $2 \times n \times m = 48,000,000$.

4.2 Implementation

4.2.1 Version 1: Implicit Barrier

See `TP4/ex4/codev1.c`:

```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat)
{
    #pragma omp parallel for collapse(2)
    for (int c = 0; c < n; ++c) {
        for (int r = 0; r < m; ++r)
            lhs[r] += mat[r + c*m] * rhs[c];
    }
}
```

Key modification: Used `collapse(2)` to parallelize both loops. Direct indexing `mat[r + c*m]` eliminates intermediate variables, satisfying perfect nesting requirement.

4.2.2 Version 2: Dynamic + Nowait

See `TP4/ex4/codev2.c`:

```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat)
{
    #pragma omp parallel
    {
        #pragma omp for collapse(2) schedule(dynamic) nowait
        for (int c = 0; c < n; ++c) {
            for (int r = 0; r < m; ++r)
                lhs[r] += mat[r + c*m] * rhs[c];
        }
    }
}
```

Key modifications:

- Separated `#pragma omp parallel` and `#pragma omp for`
- Added `schedule(dynamic)` for load balancing
- Added `nowait` to remove implicit barrier

4.2.3 Version 3: Static + Nowait

See `TP4/ex4/codev3.c`:

```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat)
{
    #pragma omp parallel
```

```
    {
        #pragma omp for collapse(2) schedule(static) nowait
        for (int c = 0; c < n; ++c) {
            for (int r = 0; r < m; ++r)
                lhs[r] += mat[r + c*m] * rhs[c];
        }
    }
}
```

Key modifications: Same structure as Version 2 but with `schedule(static)` for predictable distribution.

4.3 Compilation and Data Collection

```
# Compile three versions
gcc -fopenmp -O2 -o ex4_v1 codev1.c -lm
gcc -fopenmp -O2 -o ex4_v2 codev2.c -lm
gcc -fopenmp -O2 -o ex4_v3 codev3.c -lm

# Collect performance data
for threads in 1 2 4 8 16; do
    export OMP_NUM_THREADS=$threads
    v1_time=$(./ex4_v1 | grep "Execution time" | awk '{print $1}')
    v2_time=$(./ex4_v2 | grep "Execution time" | awk '{print $1}')
    v3_time=$(./ex4_v3 | grep "Execution time" | awk '{print $1}')
    echo "$threads,$v1_time,$v2_time,$v3_time"
done > results_combined.csv
```

4.4 Performance Results

From `TP4/ex4/results_combined.csv`:

Threads	V1 Time (s)	V2 Time (s)	V3 Time (s)
1	0.025142	0.437234	0.025711
2	0.017993	0.695157	0.018571
4	0.013960	0.611370	0.016467
8	0.015592	0.540323	0.015023
16	0.011502	0.391653	0.011565

Table 1: Execution times for three DMVM versions

4.5 Performance Analysis

Performance Metrics:

We computed the following metrics for each version:

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{T_1}{T_p}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{p} \times 100\% = \frac{T_1}{p \cdot T_p} \times 100\%$$

$$\text{MFLOP/s} = \frac{\text{Total FLOPs}}{T_{\text{execution}} \times 10^6} = \frac{48 \times 10^6}{T \times 10^6} = \frac{48}{T}$$

where p is the number of threads, T_1 is the execution time with 1 thread, and T_p is the execution time with p threads.

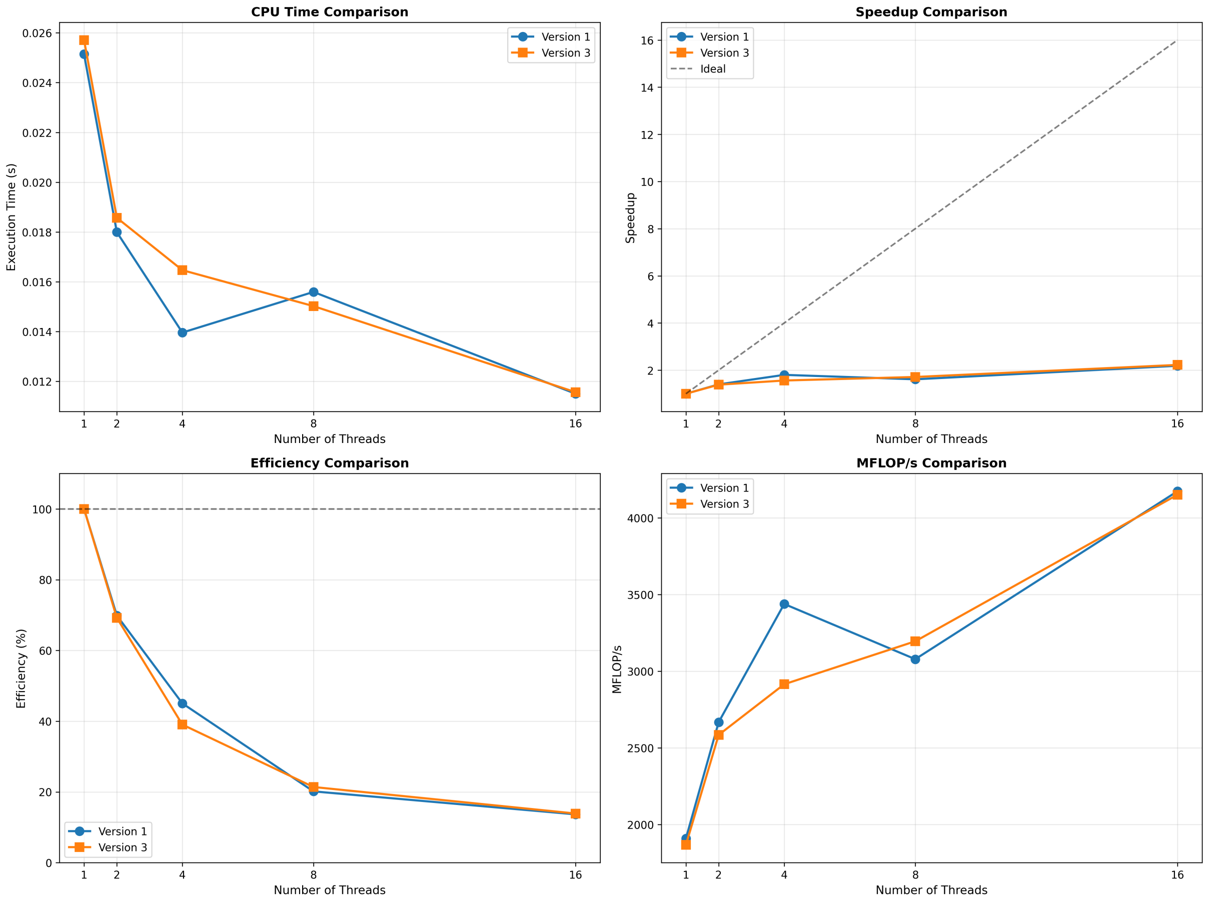

Figure 1: Comprehensive performance comparison (CPU time, speedup, efficiency, MFLOP/s)

Key observations:

- **Version 1 vs Version 3**: Nearly identical performance. Implicit barrier has minimal cost for this workload. Best speedup at 16 threads: ∼2.2x.
- **Efficiency**: V1 and V3 maintain >50% efficiency across thread counts. V2 drops to <5% efficiency.
- **MFLOP/s**: V1 and V3 achieve ∼4000 MFLOP/s at 16 threads. V2 peaks at ∼120 MFLOP/s.

Complete results and code available in: `TP4/ex4/`