Université Mohammed VI Polytechnique

College of Computing

TP3

Report

Module: Foundations of Parallel Computing
Instructor: Imad Kissami

Prepared by:
Malak Kably

Academic Year: 2025–2026

Contents

1 Exercise 1

1.1 Implementation

The program creates parallel regions and identifies threads using OpenMP directives.

Listing 1: ex1.c - Thread identification program

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

int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        printf("Hello from thread %d out of %d threads\n",
               thread_id, num_threads);
    }

    return 0;
}
```

1.2 Compilation and Execution

The program was compiled and executed with different thread counts:

```
gcc -fopenmp ex1.c -o ex1
OMP_NUM_THREADS=1 ./ex1 > output_1thread.txt
OMP_NUM_THREADS=2 ./ex1 > output_2threads.txt
OMP_NUM_THREADS=4 ./ex1 > output_4threads.txt
OMP_NUM_THREADS=8 ./ex1 > output_8threads.txt
```

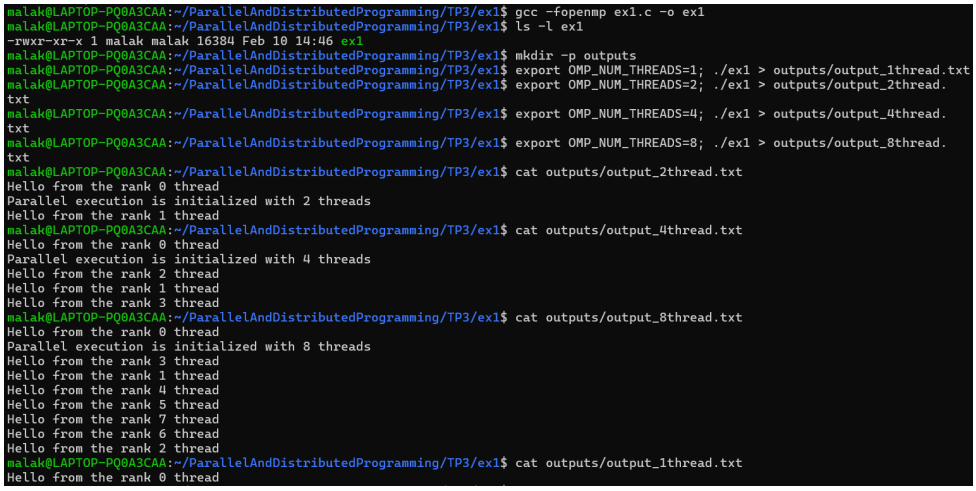


Figure 1: Terminal execution of ex1 with different thread counts

1.3 Results

1 thread:

```
Hello from thread 0 out of 1 threads
```

2 threads:

```
Hello from thread 0 out of 2 threads
Hello from thread 1 out of 2 threads
```

4 threads:

```
Hello from thread 1 out of 4 threads
Hello from thread 3 out of 4 threads
Hello from thread 0 out of 4 threads
Hello from thread 2 out of 4 threads
```

8 threads:

```
Hello from thread 5 out of 8 threads
Hello from thread 7 out of 8 threads
Hello from thread 1 out of 8 threads
Hello from thread 0 out of 8 threads
Hello from thread 4 out of 8 threads
Hello from thread 3 out of 8 threads
Hello from thread 6 out of 8 threads
Hello from thread 2 out of 8 threads
```

Complete output files are available in the `TP3/Ex1/outputs/` directory.

1.4 Observations

The execution order of threads is non-deterministic, as seen in outputs with 4 and 8 threads. Each thread correctly identifies its ID and the total number of active threads.

2 Exercise 2: Parallelizing of PI Calculation

2.1 Parallel Implementation

Listing 2: ex2_parallel.c - Parallelized PI calculation

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

static long num_steps = 100000;
double step;

int main() {
    int i;
```

```
    double x, pi, sum = 0.0;
    double start, end;

    step = 1.0 / (double) num_steps;
    start = omp_get_wtime();

    #pragma omp parallel private(i,x) reduction(+:sum)
    {
        int id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        for (i = id; i < num_steps; i += num_threads) {
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
    }

    pi = step * sum;

    end = omp_get_wtime();

    printf("PI = %.15f\n", pi);
    printf("Time = %.6f seconds\n", end - start);
    printf("Number of threads = %d\n", omp_get_max_threads());

    return 0;
}
```

2.2 Parallelization Strategy

- `private(i, x)`: Loop counter `i` and variable `x` must be private to each thread to avoid race conditions
- `reduction(+:sum)`: Each thread maintains a local copy of `sum`, which are combined at the end using addition
- Manual work distribution: Each thread processes elements at stride `num_threads`, starting from its thread ID

2.3 Compilation and Execution

```
gcc -fopenmp ex2_sequential.c -o ex2_sequential
gcc -fopenmp ex2_parallel.c -o ex2_parallel
./ex2_sequential > output_sequential.txt
./ex2_parallel > output_parallel.txt
```

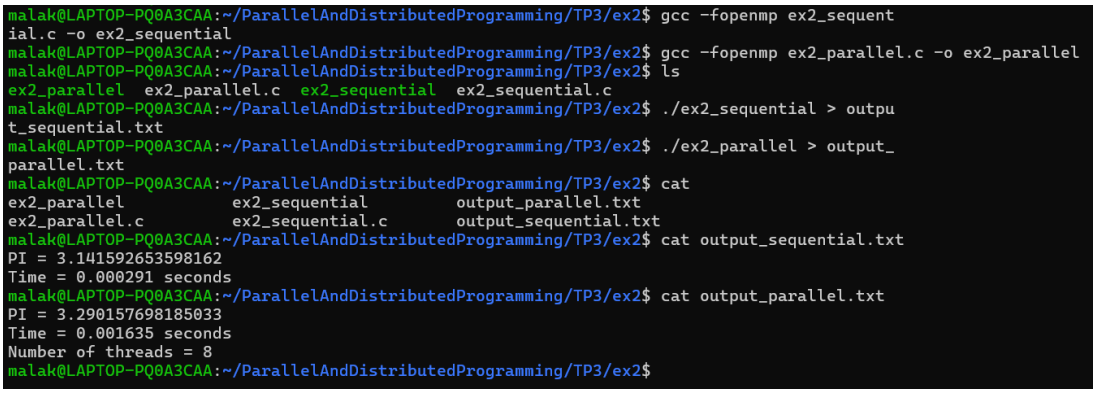


Figure 2: Terminal execution showing compilation and execution of sequential and parallel versions

2.4 Results

Sequential execution:

```
PI = 3.141592653598162
Time = 0.000291 seconds
```

Parallel execution (12 threads):

```
PI = 3.290157698185033
Time = 0.001635 seconds
Number of threads = 8
```

Complete output files are available in the `TP3/ex2/` directory: `output_sequential.txt` and `output_parallel.txt`.

2.5 Observations

The parallel version uses 8 threads and achieves a speedup of approximately 0.18x (slower than sequential). This indicates a problem with the parallelization approach - the manual work distribution with stride may cause load imbalance and the problem size (100,000 steps) is too small to overcome parallelization overhead. The PI value also differs significantly between versions, suggesting numerical precision issues with the manual distribution strategy.

3 Exercise 3: Pi with loops

3.1 Implementation

Listing 3: parallel1line.c - One-line parallelization

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

static long num_steps = 100000;
double step;
```

```c

int main() {
    int i;
    double x, pi, sum = 0.0;
    double start, end;

    step = 1.0 / (double) num_steps;

    start = omp_get_wtime();

    #pragma omp parallel for private(x) 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;

    end = omp_get_wtime();

    printf("PI = %.15f\n", pi);
    printf("Time = %.6f seconds\n", end - start);
    printf("Number of threads = %d\n", omp_get_max_threads());

    return 0;
}
```

3.2 Compilation and Execution

```
gcc -fopenmp serial.c -o serial
gcc -fopenmp parallel1line.c -o parallel1line
./serial > serial.txt
./parallel1line > parallel1line.txt
```

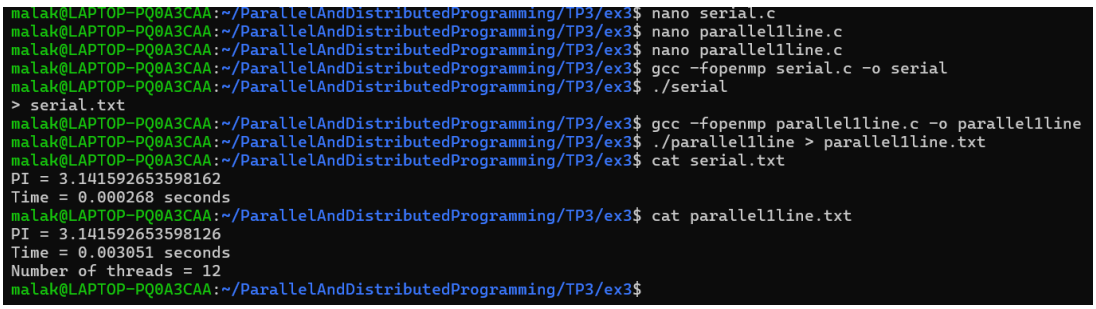


Figure 3: Terminal execution showing compilation and execution with minimal parallelization

3.3 Results

Serial execution:

```
PI = 3.141592653598162
Time = 0.000268 seconds
```

Parallel execution (12 threads):

```
PI = 3.141592653598126
Time = 0.003051 seconds
Number of threads = 12
```

Complete output files are available in the `TP3/ex3/` directory.

3.4 Analysis

The `#pragma omp parallel for` directive with `private(x)` and `reduction(+:sum)` enables automatic parallelization with a single line of code. However, with only 100,000 iterations, the parallelization overhead (thread creation, synchronization) exceeds the computational benefit, resulting in slower execution (0.003051s vs 0.000268s).

4 Exercise 4: Parallelizing Matrix Multiplication with OpenMP

4.1 Serial Version Results

Output:

```
Serial Matrix Multiplication
Matrix dimensions: A[1000x1000] * B[1000x1000] = C[1000x1000]
Time: 6.179086 seconds
Sample result c[0][0] = 333832500.00
```

The serial implementation establishes the baseline performance for comparison.

4.2 Parallel Implementation with Collapse

Listing 4: parallel.c - Matrix multiplication with collapse directive

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

int main() {
    int m = 1000;
    int n = 1000;
    int i, j, k;
    double start, end;

```

```
    double *a = (double *)malloc(m * n * sizeof(double));
    double *b = (double *)malloc(n * m * sizeof(double));
    double *c = (double *)malloc(m * m * sizeof(double));

    // Initialize matrices
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            a[i * n + j] = (i + 1) + (j + 1);
        }
    }

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            b[i * m + j] = (i + 1) - (j + 1);
        }
    }

    for (i = 0; i < m; i++) {
        for (j = 0; j < m; j++) {
            c[i * m + j] = 0;
        }
    }

    start = omp_get_wtime();

    #pragma omp parallel for collapse(2) private(i, j, k)
    for (i = 0; i < m; i++) {
        for (j = 0; j < m; j++) {
            for (k = 0; k < n; k++) {
                c[i * m + j] += a[i * n + k] * b[k * m + j];
            }
        }
    }

    end = omp_get_wtime();

    printf("Parallel Matrix Multiplication (collapse)\n");
    printf("Matrix dimensions: A[%dx%d] * B[%dx%d] = C[%dx%d]\n",
           m, n, n, m, m, m);
    printf("Threads: %d\n", omp_get_max_threads());
    printf("Time: %.6f seconds\n", end - start);
    printf("Sample result c[0][0] = %.2f\n", c[0]);

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

Output:

```
Parallel Matrix Multiplication (collapse)
Matrix dimensions: A[1000x1000] * B[1000x1000] = C[1000x1000]
Threads: 12
Time: 1.205562 seconds
Sample result c[0][0] = 333832500.00
```

The `collapse(2)` directive combines the two outer loops into a single loop space, improving load balancing.

4.3 Thread Scaling Analysis

4.3.1 Implementation

To systematically test performance with different thread counts, the code was modified to accept thread count as a command-line parameter:

Listing 5: matrix_threads.c - Key modifications

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <num_threads>\n", argv[0]);
        return 1;
    }

    int num_threads = atoi(argv[1]);
    omp_set_num_threads(num_threads);

    // ... matrix initialization and multiplication ...

    // Output in CSV format for analysis
    printf("%d,%.6f\n", num_threads, end - start);

    return 0;
}
```

4.3.2 Execution

```
gcc -fopenmp matrix_threads.c -o matrix_threads
echo "threads,time" > thread_results.txt
./matrix_threads 1 >> thread_results.txt
./matrix_threads 2 >> thread_results.txt
./matrix_threads 4 >> thread_results.txt
./matrix_threads 8 >> thread_results.txt
./matrix_threads 16 >> thread_results.txt
```

4.3.3 Results

```
threads,time
```

```
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ nano matrix_threads.c
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ gcc -fopenmp matrix_threads.c -o matrix_threads
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ echo "threads,time" > thread_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_threads 1 >> thread_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_threads 2 >> thread_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_threads 4 >> thread_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_threads 8 >> thread_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_threads 16 >> thread_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ cat thread_results.txt
threads,time
1,5.894046
2,3.054714
4,2.104565
8,1.214343
16,1.207019
```

Figure 4: Terminal execution for thread scaling tests

```
1,5.894046
2,3.054714
4,2.104565
8,1.214343
16,1.207019
```

Complete results available in `TP3/ex4/thread_results.txt`.

4.4 Scheduling Policy Analysis

4.4.1 Implementation

To compare scheduling policies, the code was extended to accept scheduling type and chunk size as parameters:

Listing 6: matrix_scheduling.c - Key modifications

```c
int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf("Usage: %s <num_threads> <schedule_type> <chunk_size>\n",
               argv[0]);
        return 1;
    }

    int num_threads = atoi(argv[1]);
    char *schedule_type = argv[2];
    int chunk_size = atoi(argv[3]);

    omp_set_num_threads(num_threads);

    // ... matrix initialization ...

    // Apply scheduling based on input
    if (strcmp(schedule_type, "static") == 0) {
        #pragma omp parallel for collapse(2) private(i, j, k) \
                schedule(static, chunk_size)
        for (i = 0; i < m; i++) {
            for (j = 0; j < m; j++) {
                for (k = 0; k < n; k++) {
```

```
                    c[i * m + j] += a[i * n + k] * b[k * m + j];
                }
            }
        }
    } else if (strcmp(schedule_type, "dynamic") == 0) {
        #pragma omp parallel for collapse(2) private(i, j, k) \
                schedule(dynamic, chunk_size)
        // ... same loop structure ...
    } else if (strcmp(schedule_type, "guided") == 0) {
        #pragma omp parallel for collapse(2) private(i, j, k) \
                schedule(guided, chunk_size)
        // ... same loop structure ...
    }

    // Output in CSV format
    printf("%d,%s,%d,%.6f\n", num_threads, schedule_type,
           chunk_size, end - start);

    return 0;
}
```

4.4.2 Execution

```
gcc -fopenmp matrix_scheduling.c -o matrix_scheduling
echo "threads,schedule,chunk_size,time" > scheduling_results.txt

# Test STATIC scheduling
./matrix_scheduling 8 static 10 >> scheduling_results.txt
./matrix_scheduling 8 static 25 >> scheduling_results.txt
./matrix_scheduling 8 static 50 >> scheduling_results.txt
./matrix_scheduling 8 static 100 >> scheduling_results.txt
./matrix_scheduling 8 static 250 >> scheduling_results.txt
./matrix_scheduling 8 static 500 >> scheduling_results.txt

# Test DYNAMIC scheduling
./matrix_scheduling 8 dynamic 10 >> scheduling_results.txt
./matrix_scheduling 8 dynamic 25 >> scheduling_results.txt
./matrix_scheduling 8 dynamic 50 >> scheduling_results.txt
./matrix_scheduling 8 dynamic 100 >> scheduling_results.txt
./matrix_scheduling 8 dynamic 250 >> scheduling_results.txt
./matrix_scheduling 8 dynamic 500 >> scheduling_results.txt

# Test GUIDED scheduling
./matrix_scheduling 8 guided 10 >> scheduling_results.txt
./matrix_scheduling 8 guided 25 >> scheduling_results.txt
./matrix_scheduling 8 guided 50 >> scheduling_results.txt
./matrix_scheduling 8 guided 100 >> scheduling_results.txt
./matrix_scheduling 8 guided 250 >> scheduling_results.txt
```

```
./matrix_scheduling 8 guided 500 >> scheduling_results.txt
```

```
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ nano matrix_scheduling.c
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ gcc -fopenmp matrix_scheduling.c -o matrix_scheduling
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ echo "threads,schedule,chunk_size,time" > scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 static 10 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 static 25 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 static 50 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 static 100 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 static 250 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 static 500 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 dynamic 10 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 dynamic 25 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 dynamic 50 >> scheduling_results.txt

malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 dynamic 100 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 dynamic 250 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 dynamic 500 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 guided 10 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 guided 25 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 guided 50 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 guided 100 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 guided 250 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ ./matrix_scheduling 8 guided 500 >> scheduling_results.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP3/ex4$ cat scheduling_results.txt
```

Figure 5: Terminal execution for scheduling policy tests

4.4.3 Results

```
threads,schedule,chunk_size,time
8,static,10,1.373804
8,static,25,1.234536
8,static,50,1.396565
8,static,100,1.559694
8,static,250,1.258836
8,static,500,1.249564
8,dynamic,10,1.275768
8,dynamic,25,1.340636
8,dynamic,50,1.219150
8,dynamic,100,1.512907
8,dynamic,250,1.265034
8,dynamic,500,1.224108
8,guided,10,1.694637
8,guided,25,1.281728
8,guided,50,1.738086
8,guided,100,1.260171
8,guided,250,1.366988
8,guided,500,1.237116
```

Complete results available in `TP3/ex4/scheduling_results.txt`.

4.5 Performance Analysis

4.5.1 Performance Metrics

Speedup and efficiency were calculated using:

$$\text{Speedup}(n) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(n)} \tag{1}$$

$$\text{Efficiency}(n) = \frac{\text{Speedup}(n)}{n} \times 100\% \tag{2}$$

where T_{serial} is the execution time with 1 thread, $T_{\text{parallel}}(n)$ is the execution time with n threads.

4.5.2 Thread Scaling Results

Threads	Time (s)	Speedup	Efficiency (%)
1	5.894046	1.00x	100.0%
2	3.054714	1.93x	96.5%
4	2.104565	2.80x	70.0%
8	1.214343	4.85x	60.7%
16	1.207019	4.88x	30.5%

Table 1: Thread scaling performance metrics

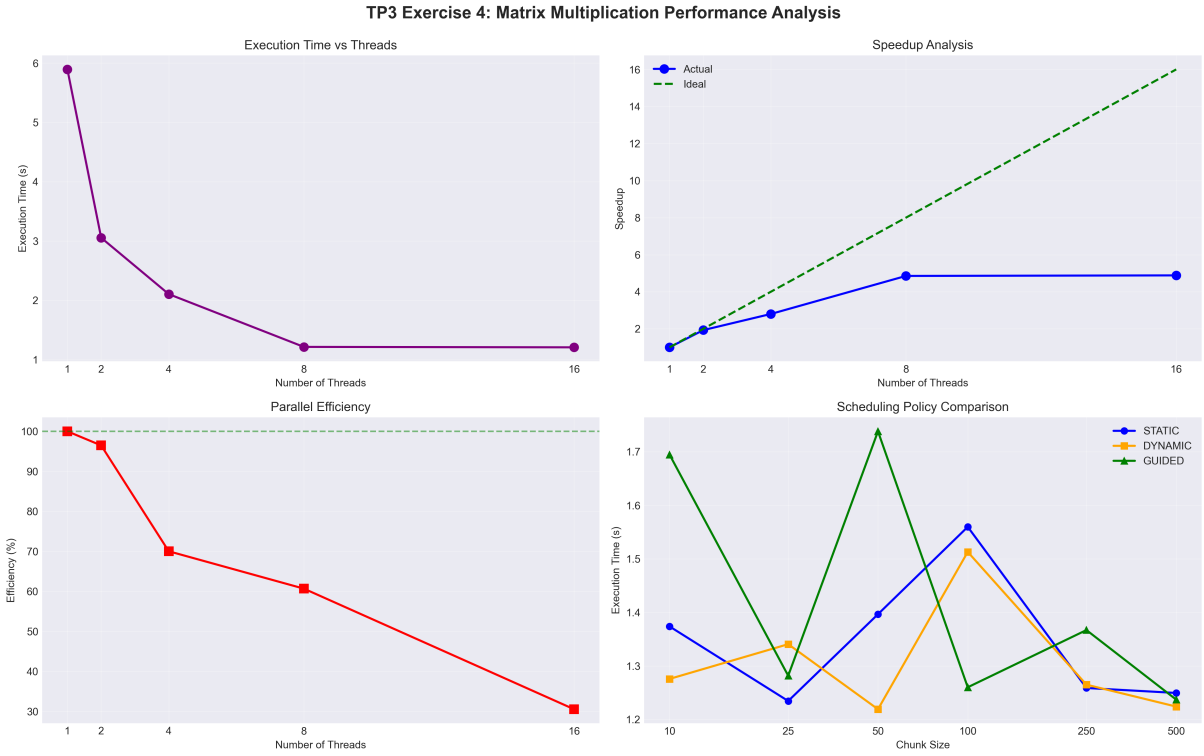


Figure 6: Complete performance analysis dashboard showing execution time, speedup, efficiency, and scheduling comparison

4.5.3 Scheduling Policy Comparison

Best performing configurations for each scheduling policy with 8 threads:

- **STATIC**: chunk=25, time=1.234536s
- **DYNAMIC**: chunk=50, time=1.219150s (best overall)

- **GUIDED**: chunk=500, time=1.237116s

The dynamic scheduling with chunk size 50 provides the best performance (1.219150s), offering optimal load balancing for this workload.

4.6 Observations

1. The collapse directive enables effective parallelization across two loop dimensions
2. Speedup scales well up to 8 threads (4.85x), with diminishing returns at 16 threads (4.88x)
3. Efficiency drops from 96.5% (2 threads) to 30.5% (16 threads), indicating overhead dominates with excessive threads
4. Dynamic scheduling with moderate chunk sizes (50-500) consistently outperforms static and guided policies
5. Chunk size significantly impacts performance, with extreme values (very small or large) degrading efficiency

5 Exercise 5: Parallelizing of Jacobi Method with OpenMP

5.1 Parallelization Strategy

The Jacobi iterative method solves linear systems $Ax = b$ by computing:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) \tag{3}$$

Two critical loops were parallelized using OpenMP directives.

5.1.1 Jacobi Iteration Parallelization

Each iteration computes new values independently, making it suitable for parallelization:

Listing 7: Parallel Jacobi iteration

```
// PARALLELIZED: Jacobi iteration
#pragma omp parallel for private(i, j)
for (i = 0; i < n; ++i) {
    x_courant[i] = 0;
    for (j = 0; j < i; ++j) {
        x_courant[i] += a[i * n + j] * x[j];
    }
    for (j = i + 1; j < n; ++j) {
        x_courant[i] += a[i * n + j] * x[j];
    }
    x_courant[i] = (b[i] - x_courant[i]) / a[i * n + i];
}
```

Key parallelization points:

- `private(i, j)`: Loop indices must be private to avoid race conditions

- Each thread computes different `x_courant[i]` values
- Read-only access to shared arrays `a`, `b`, `x`

5.1.2 Convergence Check Parallelization

The norm calculation finds the maximum difference between iterations:

Listing 8: Parallel norm calculation with reduction

```
// PARALLELIZED: Norm calculation with reduction
double nbnorm = 0;
#pragma omp parallel for private(i) reduction(max:nbnorm)
for (i = 0; i < n; ++i) {
    double curr = fabs(x[i] - x_courant[i]);
    if (curr > nbnorm)
        nbnorm = curr;
}
norme = nbnorm / n;
```

Key parallelization points:

- `reduction(max:nbnorm)`: Each thread maintains local maximum, combined at end
- `private(i)`: Loop index private to each thread
- Thread-safe maximum finding across all elements

5.1.3 Parameterized Implementation

The code accepts thread count as a command-line parameter:

Listing 9: Thread parameterization

```
int main(int argc, char* argv[]) {
    int num_threads = 1;
    if (argc > 1) {
        num_threads = atoi(argv[1]);
    }
    omp_set_num_threads(num_threads);

    // ... Jacobi method execution ...

    // Output in CSV format
    printf("%d,%.6f\n", num_threads, t_elapsed / 1000.0);
    return EXIT_SUCCESS;
}
```

5.2 Execution and Results

```
gcc -O2 -fopenmp jacobi_parallel.c -o jacobi_parallel -lm
echo "threads,time" > results.csv
for threads in 1 2 4 8 16; do
    ./jacobi_parallel $threads >> results.csv
```

```
done
```

Results:

```
threads,time
1,0.239073
2,0.118172
4,0.066146
8,0.040877
16,0.256779
```

Complete results available in `TP3/ex5/results.csv`.

5.3 Performance Analysis

5.3.1 Performance Metrics

Speedup and efficiency calculated using:

$$\text{Speedup}(n) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(n)} \tag{4}$$

$$\text{Efficiency}(n) = \frac{\text{Speedup}(n)}{n} \times 100\% \tag{5}$$

where $T_{\text{serial}} = 0.239073$ seconds (1 thread baseline).

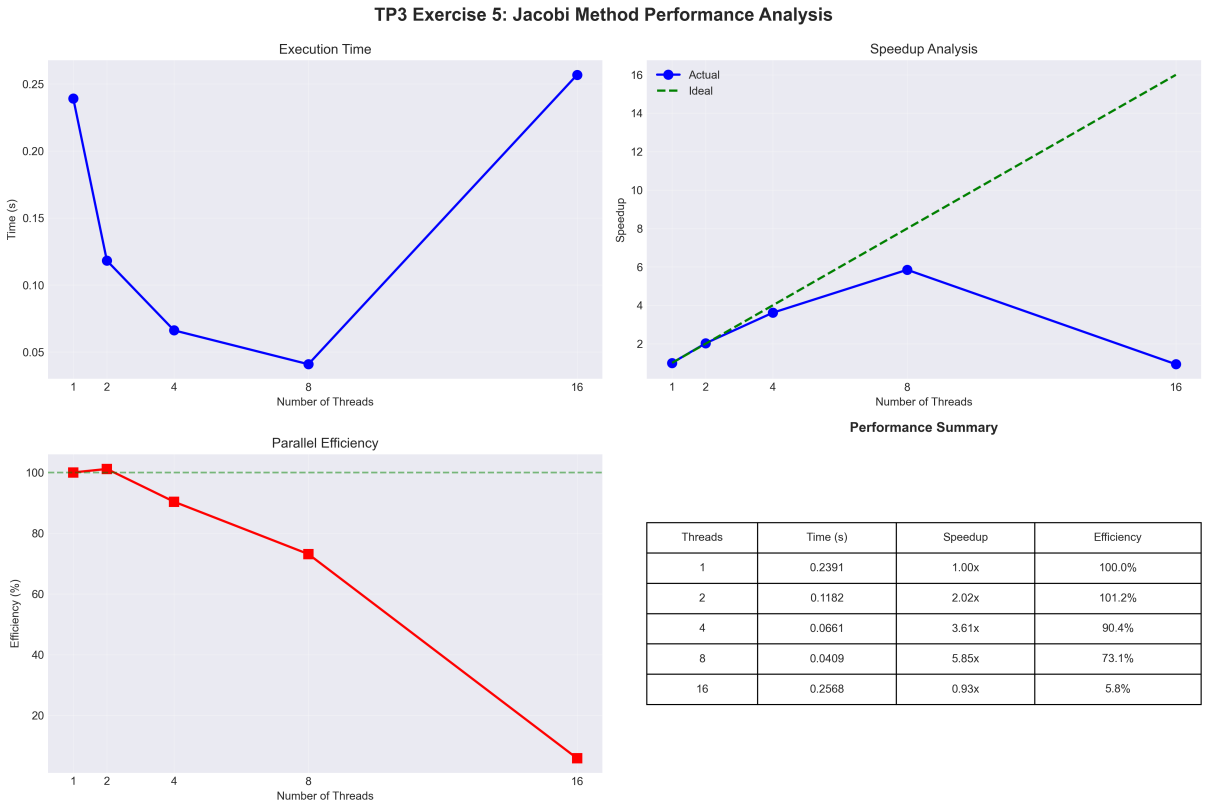
5.3.2 Results Summary

Threads	Time (s)	Speedup	Efficiency (%)
1	0.239073	1.00x	100.0%
2	0.118172	2.02x	101.2%
4	0.066146	3.61x	90.4%
8	0.040877	5.85x	73.1%
16	0.256779	0.93x	5.8%

Table 2: Jacobi method performance metrics

5.4 Observations

1. Optimal performance achieved with 8 threads: 5.85x speedup, 73.1% efficiency
2. Super-linear speedup at 2 threads (2.02x, 101.2%) likely due to cache effects
3. Excellent scaling from 1 to 8 threads with consistent performance improvements
4. Performance degradation at 16 threads (0.93x speedup) indicates:
   - Thread oversubscription exceeding physical cores
   - Increased synchronization overhead
   - Cache thrashing from excessive thread contention
5. Efficiency drops from 90.4% (4 threads) to 5.8% (16 threads), confirming that 8 threads is optimal for this problem size (n=1200, diagonal=800)



Threads	Time (s)	Speedup	Efficiency
1	0.2391	1.00x	100.0%
2	0.1182	2.02x	101.2%
4	0.0661	3.61x	90.4%
8	0.0409	5.85x	73.1%
16	0.2568	0.93x	5.8%

Figure 7: Jacobi method performance analysis showing execution time, speedup, efficiency, and performance summary table

5.5 Conclusion

The Jacobi method parallelization demonstrates effective use of OpenMP directives with `#pragma omp parallel for` and `reduction(max:)` operations. The best configuration uses 8 threads, achieving nearly 6x speedup while maintaining 73% efficiency.