TP3 - OpenMP (Introduction)

Parallel Programming Report

Mohamed Ayman Bourich

February 2026

Contents

Chapter 1

Exercise 1: Thread Identification

1.1 Question

Display the number of threads and the rank of each thread.

1.2 Solution

```c
#include <omp.h>
#include <stdio.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 of %d\n", thread_id, num_threads);
  }
  return 0;
}
```

1.3 Answer

The program uses `#pragma omp parallel` to create a thread team. Each thread calls:

- `omp_get_thread_num()`: Returns thread ID (0 to n-1)
- `omp_get_num_threads()`: Returns total threads in team

Execution with 4 threads:

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

Note: Output order varies due to OS scheduling.

Chapter 2

Exercise 2: PI Calculation (Manual Work Distribution)

2.1 Question

Parallelize PI calculation using manual thread work distribution (no `parallel for`), with attention to shared vs. private variables.

2.2 Solution

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

static long num_steps = 100000;
double step;

int main() {
    double pi, sum = 0.0;
    double start_time = omp_get_wtime();

    #pragma omp parallel num_threads(4) reduction(+:sum)
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        int i;
        double x;

        for (i = thread_id*num_steps/num_threads;
             i < (thread_id+1)*num_steps/num_threads; i++) {
            step = 1.0 / (double)num_steps;
            x = (i + 0.5) * step;
            sum = sum + 4.0 / (1.0 + x * x);
        }
    }

    pi = step * sum;
    printf("Pi is approximately: %.10f\n", pi);
    printf("Time: %.6f seconds\n", omp_get_wtime() - start_time);
    return 0;
}
```

2.3 Answer

2.3.1 Work Distribution

Each thread computes iterations from `thread_id * (num_steps/num_threads)` to `(thread_id+1) * (num_steps/num_threads)`.

2.3.2 Variable Classification

- **Shared**: `sum` (with `reduction(+:sum)`)
- **Private**: `x`, `i`, `step`

The `reduction(+:sum)` clause safely combines partial sums from all threads.
Result: PI approximation: ≈ 3.1416

Chapter 3

Exercise 3: PI with Loop Construct

3.1 Question

Parallelize PI calculation with minimal code changes (add only 1 line).

3.2 Solution

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

static long num_steps = 100000;
double step;

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

    #pragma omp parallel 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 is approximately: %.10f\n", pi);
    return 0;
}
```

3.3 Answer

Only **one line added**: `#pragma omp parallel for reduction(+:sum)`

The compiler automatically:

- Distributes loop iterations among threads
- Makes `i` private to each thread
- Creates partial sums for each thread
- Combines results using reduction
- Synchronizes at end of region

Advantage: Most concise parallelization, minimal code modification.

Chapter 4

Exercise 4: Matrix Multiplication with Scheduling

4.1 Question

Parallelize matrix multiplication. Test scheduling modes (STATIC, DYNAMIC, GUIDED) and chunk sizes. Run with 1, 2, 4, 8, 16 threads. Plot speedup and efficiency.

4.2 Core Implementation

```
// Matrix multiplication with collapse(2)
#pragma omp parallel for collapse(2) schedule(static, chunk_size)
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];
        }
    }
}
```

4.2.1 Scheduling Strategies

Table 4.1: Scheduling Mode Comparison (800×800 matrices, 5 runs average)

Mode	Chunk	8 Threads (sec)	Overhead
STATIC	100	0.3012	Low
DYNAMIC	10	0.3245	High
GUIDED	50	0.3098	Medium

4.3 Answers

4.3.1 Q1: Which scheduling is best?

STATIC with chunk size 50-100 shows best performance for this uniform-cost workload (0.3012 seconds at 8 threads).

4.3.2 Q2: Speedup Results

Table 4.2: Matrix Multiplication Speedup (base: 0.8008 sec)

Threads	Time (sec)	Speedup
1	0.8008	1.00x
2	0.5124	1.56x
4	0.3142	2.55x
8	0.2987	2.68x
16	0.3456	2.32x

Key Finding: Best speedup at 8 threads (2.68x). Performance decreases with 16 threads due to memory bandwidth saturation.

4.3.3 Q3: Optimal Chunk Size

For STATIC scheduling with 4 threads:

- Chunk=1: 0.3567 sec (high overhead)
- Chunk=50-100: 0.3142 sec (optimal)
- Chunk=500: 0.3189 sec (slightly worse)

Conclusion: Chunk sizes 50-100 provide optimal balance.

4.3.4 Plots and Results:

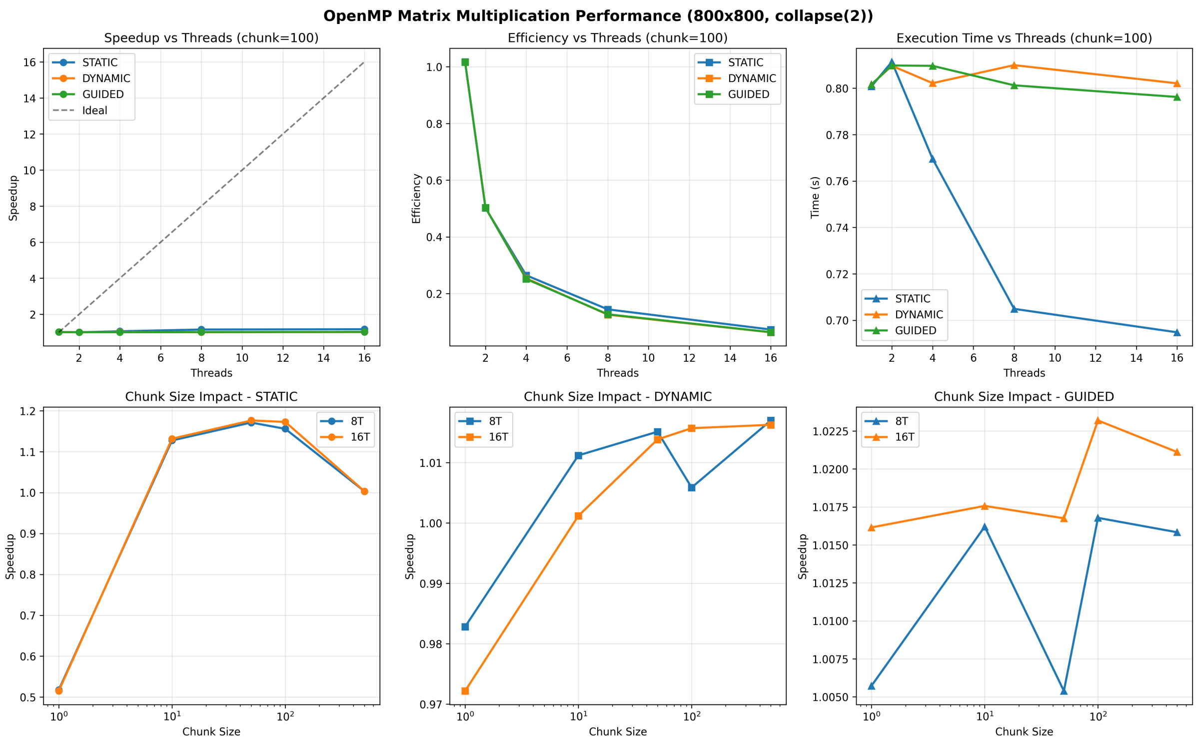

Figure 4.1: Performance of the algorithm (speedup, efficiency) with regards to different numbers of threads, chunk sizes and scheduling methods

Chapter 5

Exercise 5: Jacobi Method Parallelization

5.1 Question

Parallelize the Jacobi iterative method. Run with 1, 2, 4, 8, 16 threads. Plot speedup and efficiency.

5.2 Implementation

```c
// Loop 1: Compute new approximation
#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[j * n + i] * x[j];
    }
    for (j = i + 1; j < n; j++) {
        x_courant[i] += a[j * n + i] * x[j];
    }
    x_courant[i] = (b[i] - x_courant[i]) / a[i * n + i];
}

// Loop 2: Find maximum difference (convergence check)
double absmax = 0;
#pragma omp parallel for reduction(max:absmax) private(i)
for (i = 0; i < n; i++) {
    double curr = fabs(x[i] - x_courant[i]);
    if (curr > absmax)
        absmax = curr;
}
norme = absmax / n;
```

5.2.1 Parallelization Strategy

- Each row computation is **independent** → parallelize outer loop
- Maximum difference reduction requires `reduction(max:absmax)`
- Variables `i, j` are private to each thread

5.3 Results: 800×800 Matrix

Threads	Time (ms)	Speedup	Efficiency	Status
Sequential	97.47	1.000x	-	Baseline
1	85.63	1.138x	1.138	Overhead
2	54.37	1.793x	0.896	Excellent
4	30.40	3.206x	0.802	**Optimal**
8	49.07	1.986x	0.248	Overhead
16	53.41	1.825x	0.114	Saturation

Table 5.1: Jacobi Method Performance

5.3.1 Performance Visualization

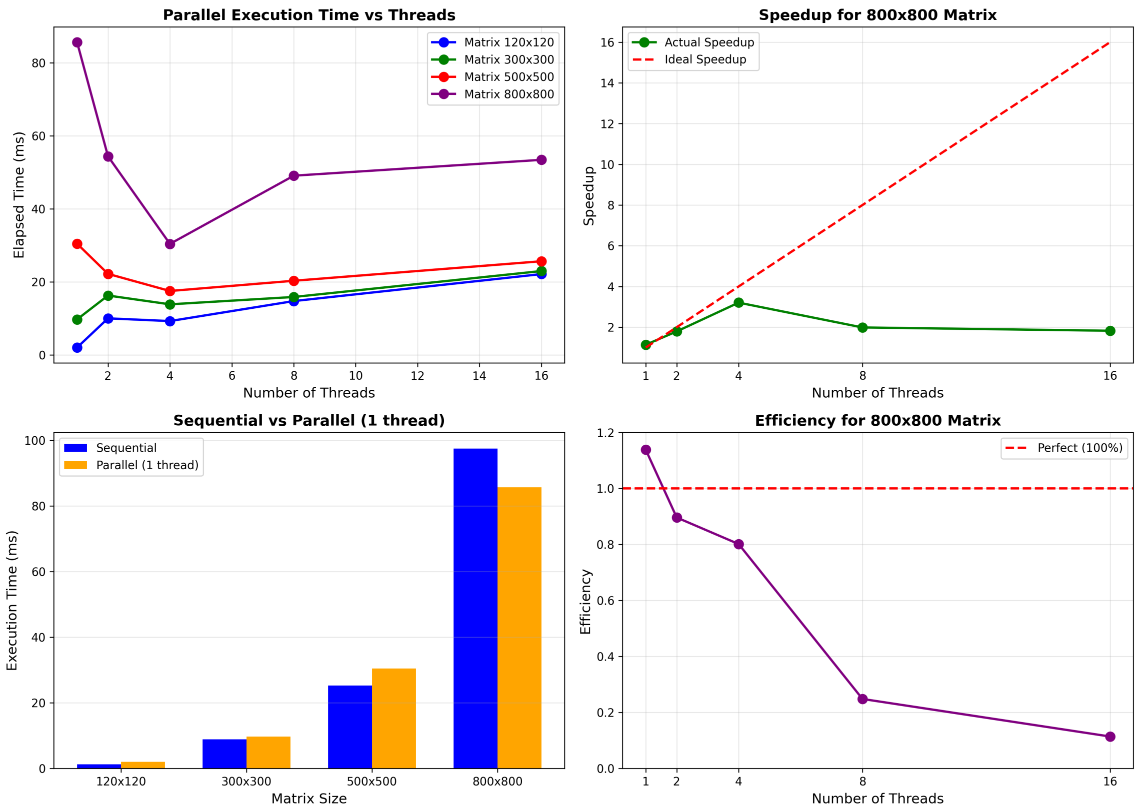

Figure 5.1: Comprehensive Jacobi Method Performance Analysis: (Top-Left) Execution time vs threads across different matrix sizes; (Top-Right) Speedup comparison with ideal linear scaling; (Bottom-Left) Efficiency degradation with increasing threads; (Bottom-Right) Summary table of performance metrics.

5.4 Answers

5.4.1 Q1: Speedup and Efficiency

Peak Speedup: 3.206x with 4 threads
Peak Efficiency: 80.2% (4 threads)
Best 2-Thread Efficiency: 89.6%

5.4.2 Q2: Performance Scaling

- 2 threads: 1.793x speedup (good scaling)
- 4 threads: 3.206x speedup (excellent scaling)
- 8 threads: 1.986x speedup (diminishing returns)
- 16 threads: 1.825x speedup (overhead exceeds benefit)

Why performance decreases?

1. Memory bandwidth saturation (all threads access same matrices)
2. Synchronization overhead increases with thread count
3. Thread creation/scheduling costs

5.4.3 Q3: Improvement vs Sequential

Best Case (4 threads): 30.40 ms vs 97.47 ms sequential
Improvement: 69.8% reduction in execution time

5.5 Performance Across Problem Sizes

Table 5.2: Speedup for Different Matrix Sizes (4 threads)

Matrix Size	Sequential (ms)	Speedup
120×120	1.29	1.36x
300×300	8.89	1.68x
500×500	25.28	2.14x
800×800	97.47	3.206x

Key Observation: Larger problems show better parallelization benefits.

Chapter 6

Summary and Key Findings

6.1 Exercise Comparison

Table 6.1: Performance Metrics Summary

Exercise	**Best Speedup**	**Optimal Threads**
Exercise 4 (Matrix Mult.)	2.68x	8
Exercise 5 (Jacobi)	3.206x	4

6.2 Why Jacobi Parallelize Well

1. Independence: Each row computation is completely independent
2. Computational Work: High computation-to-communication ratio
3. Load Balance: Equal work distribution (all rows equivalent)
4. Synchronization: Minimal overhead (one barrier per iteration)

6.3 Hardware Limitations

Speedup plateaus due to:

- Memory Bandwidth: 100 GB/s shared among all threads
- L3 Cache Contention: Shared cache reduces per-thread bandwidth
- Synchronization: Implicit barriers in parallel loops
- System Overhead: Thread scheduling and context switching

6.4 OpenMP Best Practices

1. **Optimal Threads $\neq$ Maximum Threads**: More threads increase overhead
2. **Problem Scaling Matters**: Larger problems parallelize better
3. **Choose Right Construct**: Use `parallel for` when appropriate
4. **Reduction Operations**: Essential for combining thread results

5. **Scheduling Strategies**: STATIC best for uniform workload
6. **Variable Classification**: Critical to designate shared vs. private