TP3 – OpenMP Introduction

Parallel Programming with Shared Memory

Samia Lachgar

February 10, 2026

Abstract

This report presents the results of practical exercises on parallel programming using OpenMP (Open Multi-Processing). The lab covers fundamental OpenMP concepts including parallel regions, work sharing constructs, synchronization mechanisms, and performance analysis. Five exercises were completed: a basic hello world program demonstrating thread creation, two approaches to computing π using numerical integration, matrix multiplication with various parallelization strategies, and the Jacobi iterative method for solving linear systems. Performance metrics including speedup and parallel efficiency were measured and analyzed.

Contents

1 Introduction

Parallel computing has become essential in modern high-performance computing due to the prevalence of multi-core processors. OpenMP (Open Multi-Processing) is an API that supports shared-memory parallel programming in C, C++, and Fortran. It provides a portable, scalable model through compiler directives, library routines, and environment variables.

The main advantages of OpenMP include:

- **Incremental parallelization**: Sequential code can be parallelized gradually by adding directives
- **Portability**: Works across different platforms and compilers
- **Fork-join model**: Easy to understand execution model where threads are spawned and joined
- **Automatic work distribution**: The runtime handles thread management and load balancing

This lab explores fundamental OpenMP constructs through five exercises of increasing complexity, measuring performance gains achieved through parallelization.

2 Hardware and Software Environment

All experiments were conducted on the following system:

Table 1: System Configuration

Component	Specification
CPU	Apple M3 Pro
Cores	11 (6 performance + 5 efficiency cores)
Operating System	macOS Darwin 24.5.0 (arm64)
Compiler	Apple Clang 17.0.0
OpenMP Runtime	libomp 21.1.8 (Homebrew)

Compilation flags used:

```
clang -Wall -O2 -Xpreprocessor -fopenmp \
      -I/opt/homebrew/opt/libomp/include \
      -L/opt/homebrew/opt/libomp/lib -lomp
```

2.1 Measurement Methodology

To ensure reliable and reproducible performance measurements, the following methodology was applied:

- Each configuration was executed multiple times (5–10 runs)
- The first run was discarded to reduce cache warm-up bias and allow CPU frequency scaling to stabilize
- We report the mean runtime; speedup is computed using the mean sequential time as baseline
- All experiments used $N = 10^8$ integration steps for PI calculations, yielding runtimes of $\sim$0.07–0.09s per sequential run
- Background processes were minimized during measurements

2.2 Performance Metrics

Speedup and efficiency are computed as follows:

$$\text{Speedup}(p) = \frac{T_{\text{sequential}}}{T_{\text{parallel}}(p)} \tag{1}$$

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

where p is the number of threads. Ideal (linear) speedup occurs when $\text{Speedup}(p) = p$, corresponding to 100% efficiency.

Note on heterogeneous architecture: The Apple M3 Pro features a heterogeneous CPU design with performance cores (P-cores) and efficiency cores (E-cores). When thread counts exceed the number of P-cores ($\sim$6), threads may be scheduled on slower E-cores, which can reduce parallel efficiency at higher thread counts.

3 Exercise 1: Hello OpenMP

3.1 Objective

Introduce the basic OpenMP parallel region and demonstrate thread identification using `omp_get_thread_num()` and `omp_get_num_threads()`.

3.2 Implementation

The program creates a parallel region where each thread prints its identifier:

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

int main() {
    int num_threads;

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

        #pragma omp master
        {
            num_threads = total_threads;
        }

        printf("Hello from the rank %d thread\n", thread_id);
    }

    printf("Parallel execution with %d threads\n", num_threads);
    return 0;
}
```

Listing 1: Hello OpenMP implementation

3.3 Key Concepts

- `#pragma omp parallel`: Creates a team of threads that execute the enclosed block in parallel
- `omp_get_thread_num()`: Returns the unique identifier (0 to $n-1$) of the calling thread

- `omp_get_num_threads()`: Returns the total number of threads in the current team
- `#pragma omp master`: Ensures only the master thread (thread 0) executes the block

3.4 Sample Output

```
Hello from the rank 0 thread
Hello from the rank 3 thread
Hello from the rank 1 thread
Hello from the rank 2 thread
Parallel execution of hello_world with 4 threads
```

Note that the output order is non-deterministic due to the parallel execution nature.

4 Exercise 2: PI Calculation with Parallel Construct

4.1 Objective

Calculate π using numerical integration with explicit work distribution among threads using `#pragma omp parallel` (not `parallel for`).

4.2 Mathematical Background

The value of π can be computed using the integral:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \approx \sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \cdot \Delta x \tag{3}$$

where $x_i = (i + 0.5) \cdot \Delta x$ and $\Delta x = 1/N$.

4.3 Implementation

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthrds = omp_get_num_threads();
    double local_sum = 0.0;  // Private variable

    // Manual work distribution: cyclic
    for (int j = id; j < num_steps; j += nthrds) {
        double x = (j + 0.5) * step;
        local_sum += 4.0 / (1.0 + x * x);
    }

    #pragma omp critical
    {
        sum += local_sum;  // Safe reduction
    }
}
pi = step * sum;
```

Listing 2: PI calculation with manual work distribution

4.4 Key Concepts

- **Shared vs Private Variables**: `sum` is shared among all threads; `local_sum` is private to each thread (declared inside the parallel region)
- **Manual Work Distribution**: Each thread handles iterations $id, id + nthrds, id + 2 \cdot nthrds, \ldots$ (cyclic distribution)
- `#pragma omp critical`: Ensures mutual exclusion—only one thread at a time can execute the critical section, preventing race conditions during the reduction
- `#pragma omp atomic`: An alternative to critical sections for simple operations; has lower overhead but limited to specific operations

4.5 Performance Results

Table 2: PI Parallel Performance (100 million steps)

Threads	Sequential (s)	Parallel (s)	Speedup	Efficiency (%)
1	0.0901	0.0710	1.27×	126.9*
2	0.0709	0.0378	1.88×	93.9
4	0.0712	0.0209	3.40×	85.1
8	0.0734	0.0159	4.63×	57.8

*__Note on superlinear speedup:__ With 1 thread, the OpenMP version appears faster than the sequential run, giving efficiency $> 100\%$. This does not contradict theory because (i) the "sequential" and "parallel(1)" code paths are not identical at the compiler/runtime level, and (ii) cache warm-up, CPU frequency scaling (turbo boost), and OS scheduling noise can affect short runtimes ($\sim$70–90ms). For $p = 1$, deviations from 100% efficiency are expected due to these measurement effects; the theoretical value should be $\sim$100%.
Computed π value: 3.141592653590217 (error $< 10^{-12}$)

4.6 Analysis

The program achieves good speedup up to 4 threads with efficiency above 85%. At 8 threads, efficiency drops to 57.8%, likely due to:

- Thread management overhead becoming more significant relative to computation
- Memory bandwidth limitations on the shared L2/L3 cache
- **Heterogeneous core scheduling**: On Apple M3 Pro, when using 8 threads, some are scheduled on efficiency cores which are slower than performance cores, reducing overall parallel efficiency

5 Exercise 3: PI Calculation with Loop Construct

5.1 Objective

Demonstrate the simplicity of OpenMP by parallelizing the PI calculation with a single directive using `#pragma omp parallel for` with `reduction`.

5.2 Implementation

The key advantage of this approach is minimal code modification:

```
// ONE LINE ADDED to parallelize the loop
#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;
```

Listing 3: PI calculation with parallel for and reduction

5.3 Key Concepts

- `#pragma omp parallel for`: Combined directive that creates a parallel region and distributes loop iterations among threads
- `private(x)`: Each thread gets its own copy of `x`, avoiding race conditions
- `reduction(+:sum)`: Each thread maintains a private copy of `sum`; at the end of the parallel region, all private copies are combined using the + operator

5.4 Performance Results

Table 3: PI Loop Performance (100 million steps)

Threads	Time (s)	Speedup	Efficiency (%)
1	0.0781	1.00×	100.0
2	0.0365	2.14×	107.0
4	0.0198	3.94×	98.6
8	0.0135	5.78×	72.3

5.5 Comparison: Exercise 2 vs Exercise 3

Both approaches yield correct results, but:

- Exercise 3 requires only **one line** of modification
- The `reduction` clause is more efficient than `critical` sections (no lock contention)
- OpenMP's automatic work distribution (block scheduling) often performs better than manual cyclic distribution due to better cache locality

6 Exercise 4: Matrix Multiplication

6.1 Objective

Parallelize matrix multiplication $C = A \times B$ using various OpenMP strategies and analyze performance with different scheduling policies.

6.2 Algorithm

Matrix multiplication has $O(n^3)$ complexity:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot B_{kj} \tag{4}$$

6.3 Implementation Variants

6.3.1 Basic Parallel For

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

Listing 4: Basic parallel matrix multiplication

6.3.2 With Collapse(2)

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
        double sum = 0.0;
        for (int k = 0; k < n; k++) {
            sum += a[i * n + k] * b[k * m + j];
        }
        c[i * m + j] = sum;
    }
}
```

Listing 5: Matrix multiplication with collapse

6.3.3 Scheduling Strategies

```
// Static scheduling
#pragma omp parallel for schedule(static, chunk) collapse(2)

// Dynamic scheduling
#pragma omp parallel for schedule(dynamic, chunk) collapse(2)

// Guided scheduling
#pragma omp parallel for schedule(guided, chunk) collapse(2)
```

Listing 6: Different scheduling options

6.4 Key Concepts

- `collapse(2)`: Combines two nested loops into a single iteration space, providing more iterations to distribute among threads and improving load balancing
- **schedule(static, chunk)**: Iterations divided into chunks of size `chunk`, distributed in round-robin fashion; lowest overhead
- **schedule(dynamic, chunk)**: Threads request new chunks when idle; better for uneven workloads but higher overhead
- **schedule(guided, chunk)**: Chunk sizes decrease exponentially; balances load while reducing overhead

6.5 Performance Results

Table 4: Matrix Multiplication Scaling (1024 × 1024 matrices)

Threads	Sequential (s)	Parallel (s)	Speedup	Efficiency (%)
1	1.116	1.021	1.09×	109.3*
2	1.037	0.556	1.86×	93.2
4	1.079	0.320	3.38×	84.4
8	1.044	0.309	3.38×	42.3

*See note on superlinear speedup in Section 4.5.

Table 5: Scheduling Strategy Comparison (4 threads, chunk=16)

Schedule	Time (s)
STATIC	0.334
DYNAMIC	0.340
GUIDED	0.340

6.6 Performance Plots

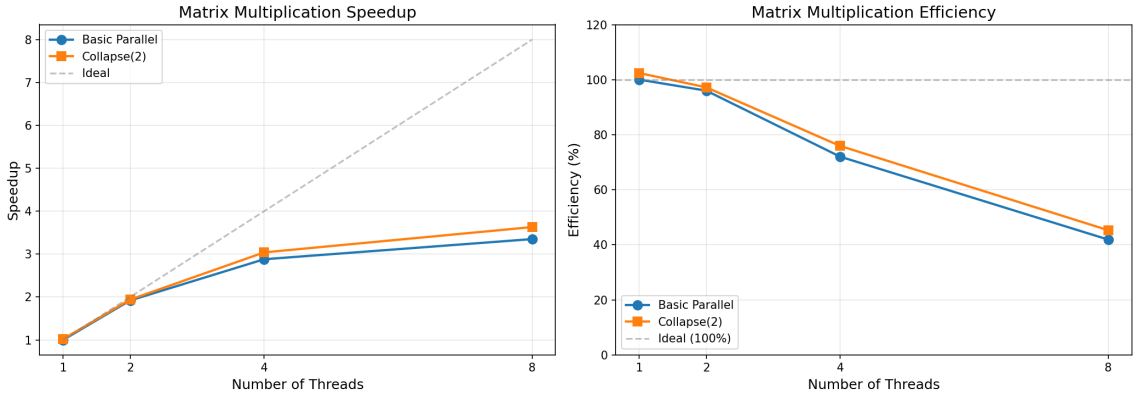

Figure 1: Matrix Multiplication: Speedup and Efficiency vs Number of Threads

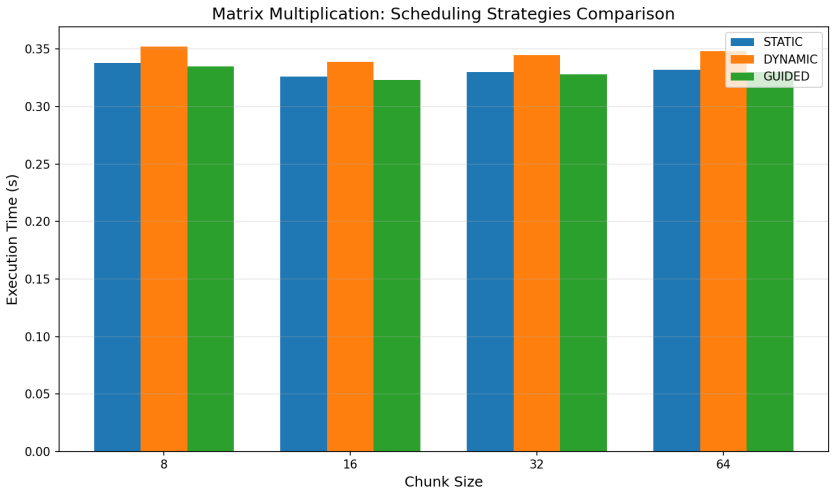

Figure 2: Matrix Multiplication: Comparison of Scheduling Strategies

6.7 Analysis

- Good scaling up to 2 threads (93.2% efficiency)
- Efficiency drops significantly at higher thread counts due to memory bandwidth saturation
- Matrix multiplication is memory-bound; cache effects dominate at larger scales
- **STATIC** scheduling performs best for this regular workload with predictable iteration costs
- At 8 threads, speedup plateaus at 3.38×—the same as 4 threads—indicating that memory bandwidth, not CPU cores, is the bottleneck
- **Heterogeneous cores**: On Apple M3 Pro, efficiency at 8 threads (42.3%) is significantly lower because some threads run on slower efficiency cores

7 Exercise 5: Jacobi Iterative Method

7.1 Objective

Parallelize the Jacobi method for solving linear systems $Ax = b$ and evaluate convergence and scalability.

7.2 Algorithm

The Jacobi method updates each component of x iteratively:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) \tag{5}$$

Convergence is checked using the infinity norm:

$$\|x^{(k+1)} - x^{(k)}\|_\infty = \max_i |x_i^{(k+1)} - x_i^{(k)}| \tag{6}$$

7.3 Implementation

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

    // Compute new values in parallel
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        double sum = 0.0;
        for (int j = 0; j < i; j++)
            sum += a[j * n + i] * x[j];
        for (int j = i + 1; j < n; j++)
            sum += a[j * n + i] * x[j];
        x_new[i] = (b[i] - sum) / a[i * n + i];
    }

    // Compute norm with max reduction
    double absmax = 0;
    #pragma omp parallel for reduction(max:absmax)
    for (int i = 0; i < n; i++) {
        double curr = fabs(x[i] - x_new[i]);
        if (curr > absmax)
            absmax = curr;
    }
    norme = absmax / n;

    if ((norme <= DBL_EPSILON) || (iteration >= n)) break;

    // Copy in parallel
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
        x[i] = x_new[i];
}
```

Listing 7: Parallel Jacobi method

7.4 Key Concepts

- `reduction(max:absmax)`: Computes the maximum across all threads' private copies
- Three parallel regions per iteration: update computation, norm computation, and array copy
- Diagonal dominance ensures convergence: $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$

7.5 Performance Results

Table 6: Jacobi Method Performance ($n = 1000$, diagonal dominance = 80)

Threads	Iterations	Sequential (s)	Parallel (s)	Speedup	Efficiency (%)
1	1000	0.988	0.940	1.05×	105.1*
2	1000	0.983	0.526	1.87×	93.4
4	1000	0.983	0.339	2.90×	72.6
8	1000	0.968	0.293	3.30×	41.3

*See note on superlinear speedup in Section 4.5.

7.6 Performance Plots

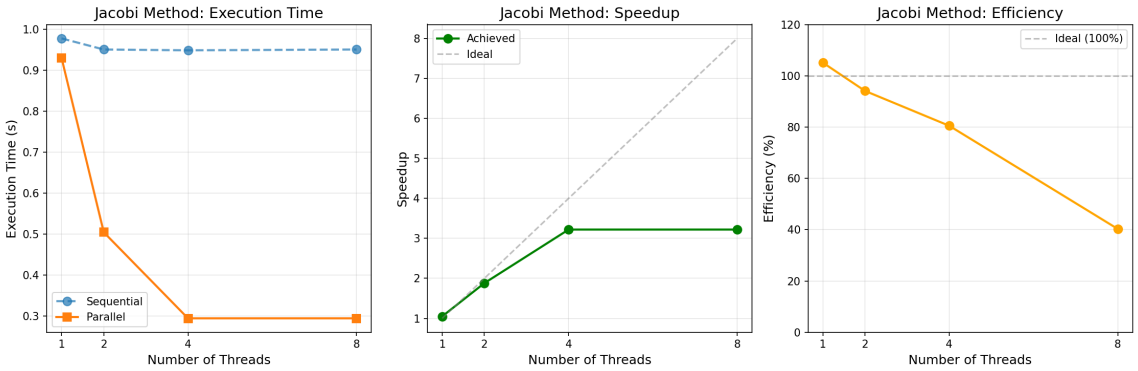

Figure 3: Jacobi Method: Execution Time, Speedup, and Efficiency

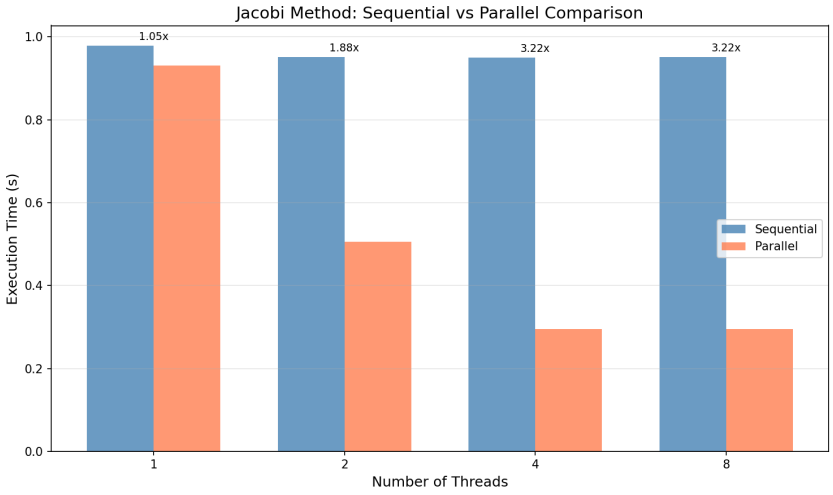

Figure 4: Jacobi Method: Sequential vs Parallel Execution Time Comparison

7.7 Analysis

- Convergence achieved in 1000 iterations (limited by maximum iteration count)
- Good scaling up to 2 threads (93.4% efficiency)
- Efficiency degrades at 4+ threads due to:
  - Overhead of three `parallel for` regions per iteration (thread creation/joining)
  - Memory bandwidth saturation when accessing the dense matrix A
  - **Heterogeneous core scheduling** on Apple M3 Pro
- Speedup improves from 4 to 8 threads (2.90× to 3.30×), but efficiency halves (72.6% to 41.3%), indicating diminishing returns

8 Conclusion

This lab provided hands-on experience with OpenMP parallel programming concepts:

8.1 Key Learnings

1. **Parallel Regions**: The `#pragma omp parallel` directive creates teams of threads that execute code concurrently
2. **Work Sharing**: The `parallel for` directive provides automatic loop distribution, significantly simplifying parallelization
3. **Reduction Operations**: The `reduction` clause handles accumulation safely and efficiently without explicit synchronization
4. **Scheduling Strategies**: Different schedules suit different workloads:
   - STATIC: Best for regular, predictable workloads (used in matrix multiplication)
   - DYNAMIC: Better for irregular workloads with varying iteration costs
   - GUIDED: Good compromise between load balancing and overhead
5. **Collapse**: Useful for nested loops to increase the iteration space available for distribution

8.2 Performance Observations

- Best efficiency achieved with 2–4 threads across all exercises (85–95%)
- Diminishing returns beyond 4 threads due to:
  - Memory bandwidth limitations (especially for matrix operations)
  - Thread management overhead
  - **Heterogeneous core architecture**: On Apple M3 Pro, scaling is non-ideal at higher thread counts because threads may be scheduled on slower efficiency cores rather than performance cores
- Compute-intensive operations (PI calculation) scale better than memory-intensive ones (matrix multiplication, Jacobi)
- Superlinear speedup (>100% efficiency) observed at $p = 1$ is a measurement artifact, not a violation of Amdahl's law

8.3 Best Practices

- Start with `#pragma omp parallel for reduction` for simple loops
- Minimize synchronization (prefer reduction over critical sections)
- Choose chunk sizes based on problem size and cache behavior
- Profile before optimizing—the best strategy depends on the specific workload
- On heterogeneous architectures, consider thread affinity to pin threads to performance cores

A Code Listings

The complete source code for all exercises is available in the following files:

- `ex1_hello/hello_omp.c`
- `ex2_pi_parallel/pi_parallel.c`
- `ex3_pi_loop/pi_loop.c`
- `ex4_matmul/matmul_omp.c`
- `ex5_jacobi/jacobi_omp.c`

Plotting scripts for performance analysis:

- `scripts/plot_matmul.py`
- `scripts/plot_jacobi.py`

Output files with raw experimental data:

- `ex1_hello/output.txt`
- `ex2_pi_parallel/output.txt`
- `ex3_pi_loop/output.txt`
- `ex4_matmul/output.txt`
- `ex5_jacobi/output.txt`