TP4 - OpenMP

Parallel Sections, Single, Master, Synchronization

BENFELLAH ikram

February 2026

Contents

1 Exercise 1: Work Distribution with Parallel Sections

1.1 Objective

Implement parallel computation of sum, maximum, and standard deviation using `#pragma omp sections`. The key constraint is that Section 3 (standard deviation) must use the sum computed in Section 1, without recomputing it.

1.2 Implementation

The program uses three parallel sections:

- **Section 1**: Computes the sum of all elements and signals completion
- **Section 2**: Computes the maximum value (independent)
- **Section 3**: Waits for Section 1, then computes standard deviation using the computed sum

Key synchronization mechanism:

```
#pragma omp sections
{
    #pragma omp section
    {
        // Section 1: Compute sum
        for (int i = 0; i < N; i++)
            local_sum += A[i];
        par_sum = local_sum;
        par_mean = par_sum / N;
        sum_ready = 1;  // Signal completion
    }

    #pragma omp section
    {
        // Section 3: Wait for sum, then compute stddev
        while (!sum_ready) { }  // Busy wait
        double local_mean = par_sum / N;
        // ... compute standard deviation
    }
}
```

1.3 Results

Metric	Sequential	Parallel (Sections)
Sum	499907.929319	499907.929319
Max	1.000000	1.000000
Std Dev	0.288695	0.288695
Time (s)	0.018	0.037

Table 1: Exercise 1 Results (N = 1,000,000)

1.4 Analysis

The parallel version is slower than sequential due to:

1. **Data dependency**: Section 3 must wait for Section 1 to complete

2. **Synchronization overhead**: The busy-wait mechanism adds delay

3. **Limited parallelism**: Only 3 sections, but Section 3 is blocked

4. **Overhead vs. work ratio**: For simple operations like sum/max, thread creation and synchronization overhead exceeds computation time

The results are correct (values match), demonstrating that the synchronization mechanism works properly.

2 Exercise 2: Exclusive Execution - Master vs Single

2.1 Objective

Demonstrate the difference between `#pragma omp master` and `#pragma omp single`, and parallelize matrix sum computation.

2.2 Key Differences

Feature	master	single
Executed by	Thread 0 only	Any one thread
Implicit barrier	No	Yes (unless nowait)
Use case	Thread 0 specific tasks	One-time initialization

Table 2: Comparison of master and single directives

2.3 Implementation

```
#pragma omp parallel shared(A, par_sum)
{
    // Master thread initializes (no implicit barrier!)
    #pragma omp master
    {
        init_matrix(N, A);
    }
    #pragma omp barrier  // Explicit barrier needed

    // Single thread prints (has implicit barrier)
    #pragma omp single
    {
        // print_matrix(N, A);
    }

    // All threads compute sum with reduction
    #pragma omp for reduction(+:par_sum)
    for (int i = 0; i < N * N; i++)
        par_sum += A[i];
}
```

2.4 Results

Version	Time (s)	Sum
Sequential	0.019	999000000.0
Parallel (8 threads)	0.013	999000000.0
Speedup	**1.46x**	

Table 3: Exercise 2 Results (N = 1000 × 1000)

2.5 Analysis

- The parallel version achieves a **1.46x speedup** with 8 threads
- The speedup is limited because:
  - Matrix initialization is sequential (master thread only)
  - Only the sum computation is parallelized
  - Memory bandwidth limits scaling for simple addition operations
- Results match perfectly, verifying correctness

3 Exercise 3: Load Balancing with Parallel Sections

3.1 Objective

Implement task scheduling with unbalanced workloads and optimize the distribution.

3.2 Workload Characteristics

- **Task A (light)**: 100,000 iterations → 0.002s
- **Task B (moderate)**: 500,000 iterations → 0.016s
- **Task C (heavy)**: 2,000,000 iterations → 0.112s

Load imbalance ratio: Heavy task takes 56× longer than light task.

3.3 Results

Version	Time (s)	Speedup	Strategy
Sequential	0.117	1.00x	-
Parallel (basic)	0.118	0.99x	3 sections
Parallel (optimized)	0.065	1.80x	Split heavy task
OpenMP Tasks	0.181	0.65x	Dynamic task creation

Table 4: Exercise 3 Results

3.4 Optimization Strategy

The basic parallel sections achieve no speedup because:

- Total time ≈ max(section times) = heavy task time

- Light and moderate tasks finish early and threads sit idle

The optimized version:

1. Combines light + moderate tasks (0.017s total)
2. Splits heavy task into 2 parts (≈0.06s each)
3. All sections now have similar execution time

```
#pragma omp parallel sections
{
    #pragma omp section
    { task_light(n); task_moderate(n); }  // Combined: ~0.017s

    #pragma omp section
    { task_heavy_part(n, 0, 2); }  // First half: ~0.06s

    #pragma omp section
    { task_heavy_part(n, 1, 2); }  // Second half: ~0.065s
}
```

3.5 Analysis

- **Load balancing is crucial**: Unbalanced sections waste parallel resources
- **Optimal strategy**: Split heavy tasks to match lighter task durations
- **OpenMP tasks overhead**: Dynamic task creation adds overhead for fine-grained tasks
- The optimized version achieves **1.80x speedup** vs 0.99x for basic sections

4 Exercise 4: Synchronization and Barrier Cost

4.1 Objective

Analyze the cost of barriers in dense matrix-vector multiplication with:

- Version 1: Implicit barrier (default)
- Version 2: schedule(dynamic) with nowait
- Version 3: schedule(static) with nowait

4.2 Matrix-Vector Multiplication

For a matrix $A \in \mathbb{R}^{m \times n}$ and vector $x \in \mathbb{R}^n$:

$$y = Ax \quad \text{where} \quad y_i = \sum_{j=1}^{n} A_{ij} x_j$$

FLOPs: $2 \times m \times n = 2 \times 600 \times 40000 = 48,000,000$

4.3 Results

Threads	Version	Time (s)	Speedup	Efficiency	MFLOP/s
1	V1 (barrier)	0.644	0.32	0.32	74.53
1	V3 (static)	0.568	0.36	0.36	84.51
2	V1 (barrier)	0.373	0.55	0.27	128.69
2	V3 (static)	0.264	0.77	0.39	181.82
4	V1 (barrier)	0.221	0.92	0.23	217.19
4	V3 (static)	0.224	0.91	0.23	214.29
8	V1 (barrier)	0.162	1.26	0.16	296.30
8	V3 (static)	0.177	1.15	0.14	271.19
16	V1 (barrier)	0.194	1.05	0.07	247.42
16	V3 (static)	0.171	1.19	0.07	280.70

Table 5: Exercise 4 Performance Metrics

4.4 Performance Plots

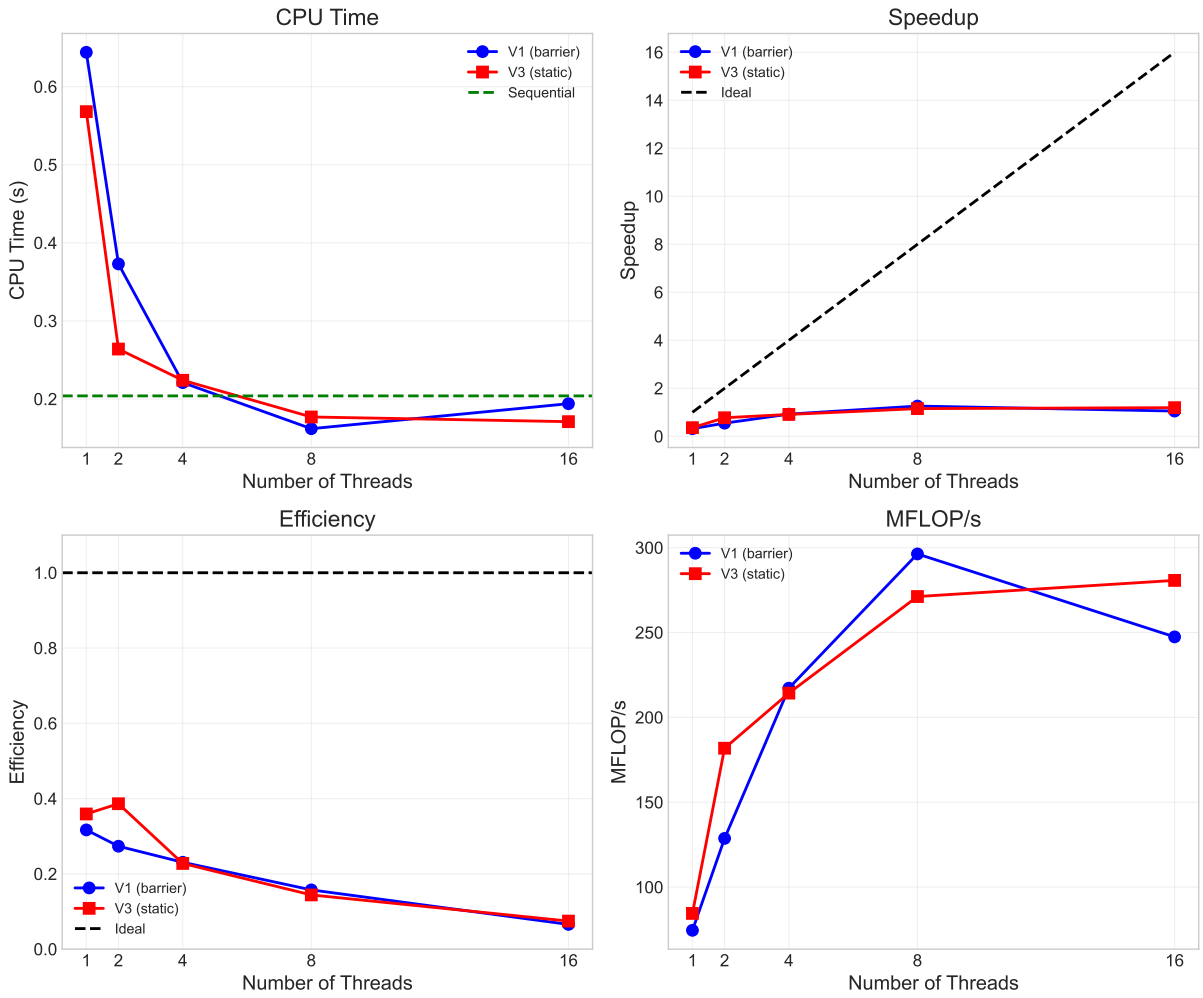

Figure 1: Performance comparison: CPU Time, Speedup, Efficiency, and MFLOP/s

4.5 Analysis: Why Barriers Limit Scalability

1. **Synchronization overhead**: All threads must wait at the barrier for the slowest thread to complete. This creates idle time proportional to load imbalance.
2. **Increasing contention**: As thread count increases, the probability of one slow thread increases, and the barrier synchronization itself becomes a bottleneck.
3. **Memory bandwidth saturation**: Matrix-vector multiplication is memory-bound. Adding more threads doesn't help when memory bandwidth is saturated.
4. **Cache effects**: Barriers can cause cache invalidation and cold cache misses when threads resume.

4.6 Analysis: When `nowait` Becomes Dangerous

The `nowait` clause removes the implicit barrier, which is dangerous when:

1. **Data dependencies exist**: If subsequent code depends on results from the parallel loop, removing the barrier causes race conditions.

```
// DANGEROUS: next_computation depends on lhs values
#pragma omp for nowait
for (int i = 0; i < n; i++)
    lhs[i] = compute(i);

result = next_computation(lhs);  // Race condition!

```

2. **Shared data is modified**: When multiple parallel regions modify the same data without synchronization.
3. **Reduction operations follow**: If a reduction depends on the parallel loop's completion.

In this exercise, `nowait` is SAFE because:

- Each row computation is independent
- Each thread writes to different `lhs[r]` elements (no overlap)
- No subsequent code depends on the parallel region results before the parallel block ends

4.7 Observations

- **Peak performance** at 8 threads (296 MFLOP/s for V1)
- **Performance degradation** at 16 threads due to overhead exceeding benefits
- **Efficiency drops** significantly as threads increase ($0.32 \rightarrow 0.07$)
- **V3 (static nowait)** slightly outperforms V1 at higher thread counts due to reduced synchronization overhead
- The sequential baseline (0.204s) is faster than 1-thread parallel versions due to OpenMP overhead

5 Conclusion

This TP explored key OpenMP concepts:

1. **Parallel Sections**: Useful for task-level parallelism but limited by the slowest section. Data dependencies require explicit synchronization.
2. **Master vs Single**: `master` has no implicit barrier (requires explicit synchronization), while `single` does. Choose based on synchronization needs.
3. **Load Balancing**: Critical for parallel efficiency. Splitting heavy tasks and combining light tasks can dramatically improve speedup (0.99x $\rightarrow$ 1.80x in our case).
4. **Barriers and nowait**: Barriers are necessary for correctness but add overhead. Use `nowait` only when data independence is guaranteed.

Key takeaways:

- Parallelization overhead can exceed benefits for small workloads
- Memory bandwidth often limits scaling before CPU becomes the bottleneck
- Proper load balancing is as important as the parallelization strategy
- Always verify correctness when using `nowait`