TP4 - OpenMP

Parallel Sections, Single, Master, Synchronization

Parallel Programming Report

Mohamed Ayman Bourich

February 2026

Contents

Chapter 1

Exercise 1: Work Distribution with Parallel Sections

1.1 Objective

Initialize an array of size $N = 1\,000\,000$ with random values and use `#pragma omp sections` to compute three quantities in parallel:

- **Section 1:** Sum of all elements
- **Section 2:** Maximum value
- **Section 3:** Standard deviation (must use the sum from Section 1, not recompute it)

1.2 Implementation

The main challenge is that Section 3 (standard deviation) depends on the result of Section 1 (sum), since:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(A_i - \bar{x})^2}, \quad \bar{x} = \frac{\text{sum}}{N}$$

We solve this with a **two-phase approach** inside a single parallel region:

1. **Phase 1:** Two sections compute sum and max in parallel.
2. An implicit barrier at the end of `sections` ensures the sum is ready.
3. A `single` directive computes the mean.
4. **Phase 2:** A section computes the standard deviation using the mean.

```
#pragma omp parallel
{
    /* Phase 1: Sum and Max in parallel */
    #pragma omp sections
    {
        #pragma omp section
        {
            double local_sum = 0.0;
            for (int i = 0; i < N; i++)
                local_sum += A[i];
            sum = local_sum;
        }

        #pragma omp section
        {
            double local_max = A[0];
            for (int i = 1; i < N; i++)
                if (A[i] > local_max)
                    local_max = A[i];
            max_val = local_max;
        }
    }
    /* Implicit barrier: sum is ready */

    #pragma omp single
    { mean = sum / N; }
    /* Implicit barrier: mean is ready */

    /* Phase 2: Standard deviation */
    #pragma omp sections
    {
        #pragma omp section
        {
            double local_stddev = 0.0;
            for (int i = 0; i < N; i++)
                local_stddev += (A[i] - mean) * (A[i] - mean);
            stddev = sqrt(local_stddev / N);
        }
    }
}
```

1.3 Results

Table 1.1: Exercise 1: Sequential vs Parallel Results

Metric	**Sequential**	**Parallel (Sections)**	**Difference**
Sum	499907.929319	499907.929319	0.0
Max	1.000000	1.000000	0.0
Std Dev	0.288695	0.288695	0.0
Time (s)	0.002	0.005	—

1.4 Analysis

- **Correctness:** All results match exactly between sequential and parallel versions.
- **Performance:** The parallel version is slower due to **thread creation overhead**. With $N = 10^6$, the computation is very fast (∼2ms), and the OpenMP overhead dominates.
- **Key insight:** `sections` is designed for coarse-grained task parallelism. For fine-grained numerical work, `parallel for` with reductions would be more efficient.
- The dependency between Section 1 and Section 3 limits the maximum parallelism to 2 concurrent sections in Phase 1.

Chapter 2

Exercise 2: Exclusive Execution - Master vs Single

2.1 Objective

Write a program where:

- A `master` thread initializes a matrix
- A `single` thread prints the matrix
- All threads compute the sum in parallel

Compare execution time with and without OpenMP.

2.2 Implementation

```c
#pragma omp parallel
{
    /* Master thread initializes the matrix */
    #pragma omp master
    {
        init_matrix(N, A);
    }
    #pragma omp barrier  /* Explicit barrier needed after master */

    /* Single thread prints (has implicit barrier) */
    #pragma omp single
    {
        if (N <= 10)
            print_matrix(N, A);
    }
    /* Implicit barrier after single */

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

2.3 Key Difference: Master vs Single

Table 2.1: Comparison of `master` and `single` directives

Property	`master`	`single`
Which thread executes?	Only thread 0	Any one thread
Implicit barrier at end?	**No**	**Yes**
`nowait` clause?	Not applicable	Supported
When to use?	Thread 0 must act	Any thread can act

Important: After `master`, we must add an explicit `#pragma omp barrier` to ensure all threads wait for initialization to complete before proceeding.

2.4 Results

Table 2.2: Exercise 2: Sequential vs Parallel (N=1000)

Version	**Sum**	**Time (s)**
Sequential	999000000.0	0.004
Parallel (8 threads)	999000000.0	0.020

2.5 Analysis

- Results are **correct**: both versions produce the same sum.
- The parallel version is slower because the matrix (1000 $\times$ 1000) is too small to benefit from parallelization. Thread creation and barrier synchronization overhead exceeds the computation time.
- For larger matrices, the parallel sum with `reduction(+:sum)` would show speedup since the work per thread increases.

Chapter 3

Exercise 3: Load Balancing with Parallel Sections

3.1 Objective

Simulate three workloads with different computational costs and measure how `parallel sections` handles the load imbalance:

- Task A (light): N iterations of `sin`
- Task B (moderate): $5N$ iterations of `sqrt` + `cos`
- Task C (heavy): $20N$ iterations of `sqrt` + `cos` + `sin`

3.2 Implementation

3.2.1 Naive Approach

Each task is assigned to one section:

```
#pragma omp parallel sections
{
    #pragma omp section
    { task_light(N); }       // ~1x work
    #pragma omp section
    { task_moderate(N); }    // ~5x work
    #pragma omp section
    { task_heavy(N); }       // ~20x work
}
```

Problem: The total time is bounded by the slowest section (Task C, 20×), so the maximum speedup is $\frac{1+5+20}{20} = 1.3\times$.

3.2.2 Optimized Approach

We split the heavy task into 4 chunks and redistribute work:

```
#pragma omp parallel sections
{
    #pragma omp section
    { task_light(N); heavy_chunk_0(); }  // ~1x + 5x = 6x
    #pragma omp section
    { task_moderate(N); }                // ~5x
    #pragma omp section
    { heavy_chunk_1(); heavy_chunk_2(); } // ~5x + 5x = 10x
    #pragma omp section
    { heavy_chunk_3(); }                        // ~5x
}
```

This distributes work more evenly across 4 sections (∼6x, 5x, 10x, 5x).

3.3 Results

Table 3.1: Exercise 3: Load Balancing Comparison

Version	Time (s)	Speedup
Sequential	0.310	1.00×
Naive Sections	0.262	1.18×
Optimized Sections	0.159	1.95×

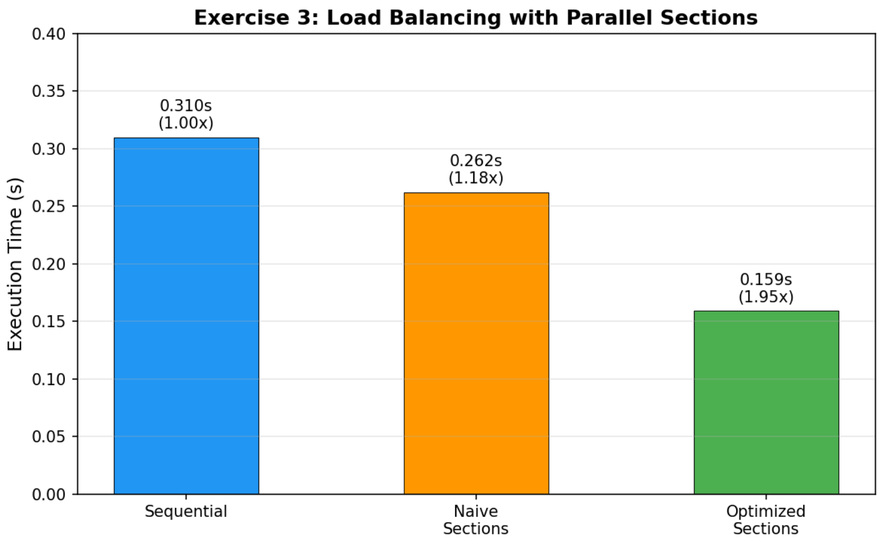

Figure 3.1: Exercise 3: Execution time comparison for different section strategies

3.4 Analysis

- **Naive sections** achieve only 1.18× speedup because the execution time is dominated by the heavy task (20× iterations). While tasks A and B finish quickly, their threads sit idle waiting for task C.
- **Optimized sections** achieve 1.95× speedup by splitting the heavy task and redistributing chunks. This brings us closer to the theoretical maximum of $\frac{26}{6.5} = 4\times$ with 4 balanced sections.
- **Key lesson:** With `parallel sections`, load balancing must be done **manually** by the programmer. Unlike `schedule(dynamic)`, sections do not automatically redistribute work.

Chapter 4

Exercise 4: Synchronization and Barrier Cost

4.1 Objective

Implement a dense matrix-vector multiplication (DMVM) $\vec{y} = A \cdot \vec{x}$ with three parallelization strategies:

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

Matrix dimensions: $m = 600$ rows, $n = 40{,}000$ columns.

4.2 Implementation

4.2.1 Version 1: Implicit Barrier

```
void dmvm_v1(int n, int m, double *lhs, double *rhs, double *mat) {
    #pragma omp parallel for schedule(static)
    for (int c = 0; c < n; ++c) {
        int offset = m * c;
        for (int r = 0; r < m; ++r)
            lhs[r] += mat[r + offset] * rhs[c];
    }
    /* Implicit barrier at end */
}
```

4.2.2 Version 3: Static + Nowait

To avoid race conditions with `nowait`, each thread uses a local accumulator:

```
void dmvm_v3(int n, int m, double *lhs, double *rhs, double *mat) {
    #pragma omp parallel
    {
        double *local_lhs = calloc(m, sizeof(double));

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

        #pragma omp critical
        {
            for (int r = 0; r < m; ++r)
                lhs[r] += local_lhs[r];
        }
        free(local_lhs);
    }
}
```

4.3 Performance Metrics

For the DMVM with m rows and n columns, each iteration performs 1 multiply and 1 add per element:

$$\text{FLOPs} = 2 \times n \times m = 2 \times 40{,}000 \times 600 = 48{,}000{,}000$$

$$\text{MFLOP/s} = \frac{\text{FLOPs}}{t \times 10^6}$$

$$\text{Speedup}(p) = \frac{T_{\text{seq}}}{T_p}, \quad \text{Efficiency}(p) = \frac{\text{Speedup}(p)}{p}$$

4.4 Results

Table 4.1: Exercise 4: V1 (Implicit Barrier) vs V3 (Static + Nowait)

	V1 (Implicit Barrier)				**V3 (Static + Nowait)**			
Threads	Time (ms)	Speedup	Eff (%)	MFLOP/s	Time (ms)	Speedup	Eff (%)	MFLOP/s
1	16.80	1.03	103.2	2857	16.20	1.07	107.0	2963
2	14.45	1.20	60.0	3322	12.45	1.39	69.6	3855
4	14.20	1.22	30.5	3380	12.35	1.40	35.1	3887
8	13.00	1.33	16.7	3692	12.40	1.40	17.5	3871
16	14.15	1.23	7.7	3392	12.60	1.38	8.6	3810

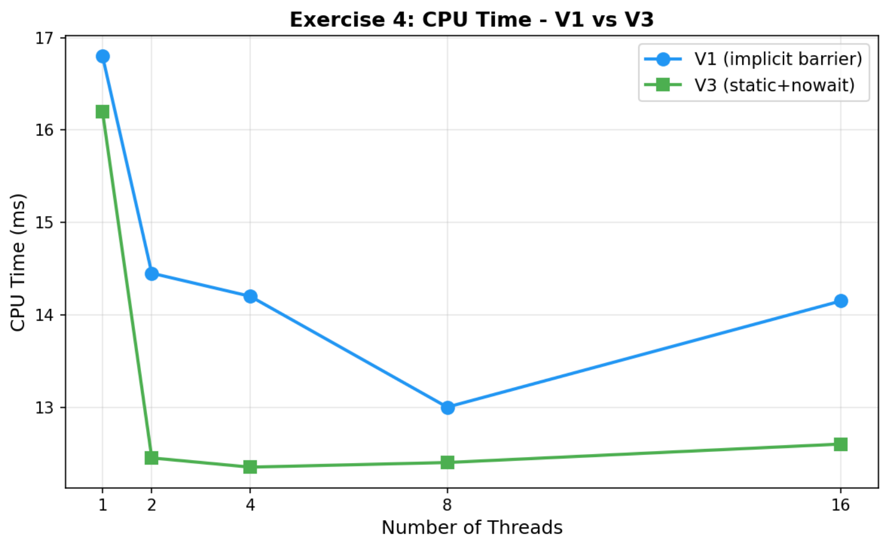

Figure 4.1: CPU Time comparison: V1 (implicit barrier) vs V3 (static + nowait)

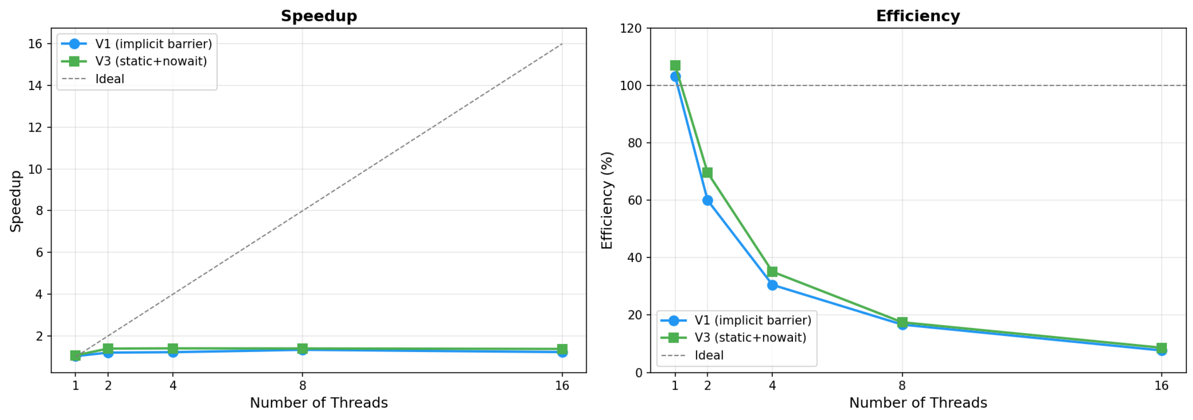

Figure 4.2: Speedup and Efficiency: V1 vs V3

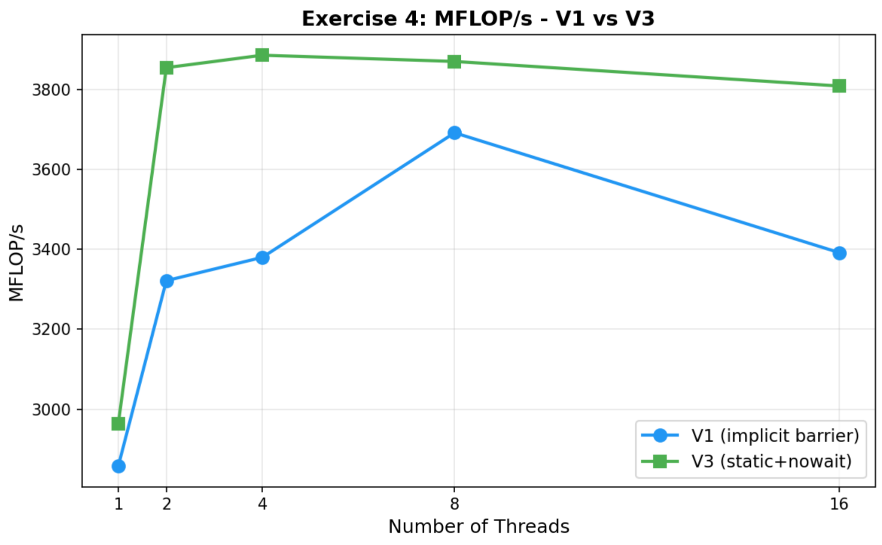

Figure 4.3: MFLOP/s: V1 vs V3

4.5 Analysis

4.5.1 Why Barriers Limit Scalability

- At an implicit barrier, all threads must wait for the **slowest thread** to finish before proceeding. This creates idle time.
- In V1, the barrier at the end of the `parallel for` forces synchronization. If threads have unequal work (e.g. due to OS scheduling), faster threads waste time waiting.
- As the number of threads increases, the probability of one thread being delayed increases, making the barrier cost grow with thread count.
- **Amdahl's Law:** The barrier itself is a serial bottleneck that limits maximum speedup.

4.5.2 When `nowait` Becomes Dangerous

- `nowait` removes the barrier, allowing threads to continue without synchronization.

- **Danger 1 – Race conditions:** If the next operation reads data written by the parallel loop, removing the barrier causes data races. In our DMVM, multiple threads writing to the same `lhs[r]` would corrupt results without local accumulators.
- **Danger 2 – Data dependencies:** If a subsequent loop depends on results from the previous one, `nowait` can produce incorrect results.
- **Safe usage:** `nowait` is safe when each thread works on independent data and there are no cross-thread dependencies before the next synchronization point.

4.5.3 Why Speedups Are Limited

The DMVM operation is **memory-bound**: the computation ($2nm$ FLOPs) is small compared to the data movement ($nm+n+m$ doubles read). On modern hardware, the memory bandwidth is the bottleneck, not the CPU. Adding more threads does not increase memory bandwidth, so speedup plateaus quickly. This is consistent with the observed results where speedup saturates around 1.3–1.4$\times$ regardless of thread count.

Chapter 5

Conclusion

This TP explored several important OpenMP constructs for parallel programming:

1. **Parallel Sections** (`#pragma omp sections`) enable coarse-grained task parallelism. They are useful when different independent tasks can run concurrently, but require manual load balancing.

2. **Master vs Single:**

   - `master` – only thread 0 executes, no implicit barrier
   - `single` – any thread executes, has an implicit barrier

   The choice depends on whether you need guaranteed execution by thread 0 and whether synchronization is needed.

3. **Load Balancing:** Naive use of sections can lead to poor load balance when tasks have unequal costs. Splitting heavy tasks across multiple sections significantly improves performance ($1.18\times \rightarrow 1.95\times$).

4. **Barriers and Synchronization:**

   - Implicit barriers ensure correctness but limit scalability
   - `nowait` can improve performance but risks race conditions
   - Memory-bound operations show limited speedup regardless of synchronization strategy
   - Local accumulators are essential when using `nowait` to avoid data races

Key takeaway: The choice of synchronization strategy must balance **correctness** (barriers ensure data consistency) with **performance** (barriers introduce idle time). Memory-bound operations require algorithmic changes (blocking, cache optimization) rather than just more threads to achieve significant speedup.