TP4 - OpenMP Advanced Synchronization

Parallel Sections, Single, Master, Synchronization

Analytical Report

February 2026

Exercise 1: Work Distribution with Parallel Sections

My Approach

To solve this problem, I needed to run the Sum, Max, and Standard Deviation calculations at the same time. The main challenge was that the standard deviation usually requires the mean (which comes from the sum), creating a dependency that would force the code to run sequentially.

To fix this and ensure full parallelism, I used the algebraic formula for variance:

$$\sigma^2 = \frac{\sum x_i^2}{N} - \left(\frac{\sum x_i}{N}\right)^2$$

This allowed me to calculate the "Sum of Squares" in Section 3 independently, at the exact same time that Section 1 was calculating the normal Sum.

Code Implementation

I used `omp_set_num_threads(3)` to make sure each section got its own thread. Here is the core logic I wrote:

```
    // Force 3 threads so all sections run simultaneously
    omp_set_num_threads(3);

    #pragma omp parallel sections
    {
        // --- SECTION 1: Compute Sum ---
        #pragma omp section
        {
            double local_sum = 0.0;
            for (int i = 0; i < N; i++) local_sum += A[i];
            sum = local_sum; // No race condition here
        }

        // --- SECTION 2: Compute Max ---
        #pragma omp section
        {
            double local_max = -1.0;
```

```
            for (int i = 0; i < N; i++) {
                if (A[i] > local_max) local_max = A[i];
            }
            max = local_max;
        }

        // --- SECTION 3: Compute Sum of Squares ---
        #pragma omp section
        {
            double local_sq = 0.0;
            // Calculating squares independently to avoid waiting
                for Mean
            for (int i = 0; i < N; i++) local_sq += A[i] * A[i];
            sum_sq = local_sq;
        }
    }

    // Final Step: Combine results instantly
    double mean = sum / N;
    double variance = (sum_sq / N) - (mean * mean);
    stddev = sqrt(variance);
```

Results

I ran the program with $N = 1,000,000$ and obtained the following output:

```
Sum      = 500727.089668
Max      = 0.999999
Std Dev  = 0.288366
Time     = 0.003627 seconds
```

The execution time is very low ($0.003s$), which confirms that the threads were running in parallel. If Section 3 had waited for Section 1, the time would have been the sum of all tasks, but here it corresponds to the duration of the longest individual task.

Exercise 2: Master vs Single - Subtle but Critical Differences

The Three-Phase Structure

Listing 1: Master, Single, and Parallel For

```
#pragma omp parallel
{
    // Phase 1: MASTER initializes
    #pragma omp master
    {
        init_matrix(N, A);
    }
```

```
    #pragma omp barrier  // EXPLICIT barrier needed!

    // Phase 2: SINGLE prints
    #pragma omp single
    {
        print_matrix(N, A);
    }
    // Implicit barrier here

    // Phase 3: ALL threads compute sum
    #pragma omp for reduction(+:sum)
    for (int i = 0; i < N*N; i++) {
        sum += A[i];
    }
}
```

Master vs Single: The Critical Distinction

Aspect	master	single
Which thread?	Always thread 0 (master)	Any one thread (runtime decides)
Implicit barrier?	**NO**	**YES**
Other threads?	Continue immediately	Wait at barrier
Use case	Initialization, I/O setup	One-time computation, printing

Why the Explicit Barrier After Master?

Without the barrier:

```
Time    Master (T0)          Other Threads (T1-T7)
t0      Start init_matrix    Exit master block
t1      Writing A[0]         Read A[0] in sum loop → RACE!
t2      Writing A[1000]      Read A[1000] → RACE!
```

The danger: Other threads don't wait after master block. They immediately proceed to the sum computation, reading uninitialized or partially initialized data!

With explicit barrier:

```
Time    Master (T0)          Other Threads (T1-T7)
t0      Start init_matrix    Hit barrier, WAIT
t1      Writing A[...]       Still waiting
t2      Finish init          Still waiting
t3      Hit barrier          All threads release together
t4      All safely read initialized data
```

Why No Explicit Barrier After Single?

Single has **implicit barrier** at the end. All threads automatically wait for the single block to complete.

This ensures matrix is printed before sum computation begins (though printing happens sequentially anyway).

Performance Comparison

Version	Time (s)	Speedup
Sequential	0.014015	1.00×
OpenMP	0.009000	1.56×

Table 1: N=1000 matrix (1M elements)

Why only 1.56× speedup?

- Initialization: Serial (master only)
- Printing: Serial (single only)
- Sum: Parallel (all threads)

If sum is 50% of total time:

- Serial portion: 50%
- Parallel portion: 50% / 8 threads = 6.25%
- Total: 56.25% of sequential time
- Speedup: $\frac{1}{0.5625} \approx 1.78\times$ (close to observed)

Barrier Semantics Summary

Directive	Implicit Barrier?	Can Add nowait?
`parallel`	Yes (at end)	No
`for`	Yes (at end)	Yes
`sections`	Yes (at end)	Yes
`single`	Yes (at end)	Yes
`master`	**NO**	N/A

Critical rule: If using `master`, almost always need explicit `barrier` afterward!

Exercise 3: Load Balancing with Parallel Sections & Tasks

Approach

To address the load balancing challenge presented by three tasks with vastly different execution times (Light, Moderate, Heavy), five different parallelization strategies were implemented and compared:

- **Strategy 1 (Basic Sections):** Naive distribution of tasks to threads using `#pragma omp sections`.
- **Strategy 2 (Manual Grouping):** Grouping Light and Moderate tasks onto a single thread to balance the workload against the Heavy task.
- **Strategy 3 (Tasks):** Utilizing the OpenMP `task` construct for dynamic scheduling.
- **Strategy 4 (Priority Tasks):** Assigning higher priority to the Heavy task to ensure it is scheduled first.
- **Strategy 5 (Data Parallelism):** Parallelizing the internal loop of the Heavy task itself, rather than distributing the functions.

Code Implementation

The following C code integrates all five strategies for performance comparison.

Listing 2: Comparison of Load Balancing Strategies

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

// --- Task Definitions ---
void task_light(int N) {
    double x = 0.0;
    for (int i = 0; i < N; i++) x += sin(i * 0.001);
}

void task_moderate(int N) {
    double x = 0.0;
    for (int i = 0; i < 5*N; i++) x += sqrt(i * 0.5) * cos(i *
       0.001);
}

void task_heavy(int N) {
    double x = 0.0;
    for (int i = 0; i < 20*N; i++)
        x += sqrt(i * 0.5) * cos(i * 0.001) * sin(i * 0.0001);
}

```

```
int main() {
    int N = 1000000;
    double start, end;

    // ... [Sequential Baseline Code Omitted] ...

    // --- Strategy 1: Basic Sections ---
    #pragma omp parallel sections
    {
        #pragma omp section
        task_light(N);
        #pragma omp section
        task_moderate(N);
        #pragma omp section
        task_heavy(N);
    }

    // --- Strategy 3: Dynamic Tasks ---
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            task_light(N);
            #pragma omp task
            task_moderate(N);
            #pragma omp task
            task_heavy(N);
        }
    }

    // --- Strategy 4: Priority Tasks ---
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task priority(100) // Start Heavy task
                first
            task_heavy(N);
            #pragma omp task priority(50)
            task_moderate(N);
            #pragma omp task priority(10)
            task_light(N);
        }
    }

    // --- Strategy 5: Parallel Loop (Data Parallelism) ---
    // Run Light/Mod sequentially, but split Heavy across all
       threads
    task_light(N);
    task_moderate(N);
```

```
72
73      double x = 0.0;
74      #pragma omp parallel for reduction(+:x) schedule(dynamic,
            1000)
75      for (int i = 0; i < 20*N; i++) {
76          x += sqrt(i * 0.5) * cos(i * 0.001) * sin(i * 0.0001);
77      }
78
79      return 0;
80  }
```

Results Analysis

The code was executed with $N = 1,000,000$ on an 8-thread system.

```
=== Load Balancing with OpenMP ===
N = 1000000, Threads = 8

Sequential: 0.535063 seconds

--- Strategy 1: Basic Parallel Sections ---
Time: 0.489296 s, Speedup: 1.09x

--- Strategy 2: Balanced Sections (Combine A+B) ---
Time: 0.532013 s, Speedup: 1.01x

--- Strategy 3: Dynamic Task Scheduling ---
Time: 0.452350 s, Speedup: 1.18x

--- Strategy 4: Priority-based Scheduling ---
Time: 0.449185 s, Speedup: 1.19x

--- Strategy 5: Parallelize Heavy Task Internally ---
Time: 0.194987 s, Speedup: 2.74x
```

Analysis of Functional Parallelism (Strategies 1-4): All section and task-based strategies yielded poor speedup ($\approx 1.1x$). This limited performance is due to the "Critical Path" problem: `task_heavy` consumes approximately 90% of the total execution time. Even if Tasks A and B complete instantly on other threads, the program execution time is bounded by the single thread processing Task C. The implicit barrier at the end of the parallel region forces all other threads to idle until Task C completes.

Analysis of Data Parallelism (Strategy 5): Strategy 5 proved strictly superior ($\approx 2.74x$ speedup) by shifting from Task Parallelism to Data Parallelism. Instead of assigning the heavy function to a single thread, the internal loop of `task_heavy` was parallelized using `#pragma omp parallel for`. This allowed all 8 threads to contribute to the heaviest computation, significantly reducing the bottleneck that plagued the previous strategies.

Exercise 4: Synchronization Overhead and the `nowait` Clause

The Dense Matrix-Vector Multiplication

Computing: $\text{lhs}[r] = \sum_{c=0}^{n-1} \text{mat}[r][c] \times \text{rhs}[c]$

Algorithm structure:

```
for (int c = 0; c < n; ++c) {            // Columns
    for (int r = 0; r < m; ++r) {        // Rows
        lhs[r] += mat[r][c] * rhs[c];
    }
}
```

Matrix: 600 rows × 40,000 columns

Key observation: Outer loop over columns, inner over rows.

Version 1: Implicit Barrier (Safe)

Listing 3: Synchronization every column

```
#pragma omp parallel
{
    for (int c = 0; c < n; ++c) {
        #pragma omp for schedule(static)
        for (int r = 0; r < m; ++r) {
            lhs[r] += mat[r + c*m] * rhs[c];
        }
        // IMPLICIT BARRIER after each column!
    }
}
```

Result: 0.0834s, Speedup 0.95× (SLOWER than sequential!)

What's happening?

- 40,000 columns → 40,000 barriers!
- Each barrier: all 8 threads synchronize
- Barrier overhead: ∼1-5 μs per barrier
- Total barrier cost: $40,000 \times 2\mu s = 80ms$
- Actual computation: $\sim 79ms$
- Overhead exceeds computation!

Barrier mechanism cost:

1. Each thread sets "arrived" flag
2. Threads spin-wait checking if all arrived
3. Cache line bouncing between cores
4. Context switches if waiting too long

Version 2: Dynamic + nowait (UNSAFE!)

Listing 4: Race condition catastrophe

```
#pragma omp parallel
{
    for (int c = 0; c < n; ++c) {
        #pragma omp for schedule(dynamic) nowait
        for (int r = 0; r < m; ++r) {
            lhs[r] += mat[r + c*m] * rhs[c];
        }
    }
}
```

Result: 0.4077s, Speedup 0.19× - TERRIBLE!
The race condition:

```
Time    Thread 0 (dynamic)      Thread 1 (dynamic)
t0      c=0, gets rows 0-99     c=0, gets rows 100-199
t1      Finishes column 0       Still working on c=0
t2      c=1, gets rows 0-99     Still on c=0
        Read lhs[50]
t3      Compute lhs[50]+=...    Finishes c=0
t4      Write lhs[50]           c=1, gets rows 0-99
                                Read lhs[50] - RACE!
```

Why dynamic makes it worse:

- Static: Thread X always owns rows 0-74
- Dynamic: Thread X might get rows 0-99 in c=0, then rows 50-149 in c=1
- Overlapping row assignments across columns → multiple threads update same `lhs[r]`

Result: Catastrophic race conditions. Wrong answer + terrible performance from cache thrashing.

Version 3: Static + nowait (SAFE & FAST!)

Listing 5: The correct optimization

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

Result: 0.0738s, Speedup 1.07× - BEST!
Why this is safe:

```
Thread assignment with static scheduling:
Thread 0: Always rows 0-74   (all columns)
Thread 1: Always rows 75-149 (all columns)
...
Thread 7: Always rows 525-599 (all columns)
```

Critical property: Each thread owns a fixed set of rows across ALL column iterations.

- Thread 0 updates `lhs[0..74]` for c=0
- Thread 0 updates `lhs[0..74]` for c=1
- No other thread touches `lhs[0..74]`
- No race condition!

Performance benefit:

- Removed 39,999 barriers (kept only final one at parallel region end)
- Each thread processes its rows for all columns without waiting
- Better cache locality (each thread keeps its rows in cache)
- Minimal synchronization overhead

Performance Summary

Version	Time (s)	Speedup	Efficiency	MFLOP/s	Status
Sequential	0.0792	1.00×	100%	606	OK
V1 (Sync)	0.0834	0.95×	12%	576	OK
V2 (Dyn)	0.4077	0.19×	2%	118	RACE
V3 (Static)	0.0738	1.07×	13%	650	OK

FLOPs calculation: $2 \times 600 \times 40000 = 48M$ FLOPs (multiply + add)

When is `nowait` Safe?

Safe conditions:

1. **Static scheduling** with non-overlapping work partitions
2. **Read-only** shared data
3. **Thread-private** results combined later

Dangerous conditions:

1. Dynamic/guided scheduling with shared writes
2. Dependency between loop iterations
3. Relying on results from previous iteration

Barrier Cost Analysis

Theoretical barrier overhead:
Assume 2μs per barrier:

- V1: 40,000 barriers $\times$ 2μs = 80ms overhead
- V3: 1 barrier $\times$ 2μs = 0.002ms overhead
- Savings: 80ms

Observed difference: 83.4ms - 73.8ms = 9.6ms
Why less than theoretical?

- Some barriers overlap with computation
- Cache effects provide other benefits in V3
- Measurement variance

Critical Design Principles

1. **Minimize synchronization points**: Each barrier is expensive
2. **Use nowait carefully**: Only when guaranteed safe
3. **Static scheduling for predictable access patterns**: Enables safe nowait
4. **Understand data access patterns**: Who reads/writes what, when?
5. **Profile before optimizing**: V1 showed synchronization was the bottleneck