TP3 - OpenMP Analysis

Understanding Parallel Computing Fundamentals

Kawtar Labzae

February 2026

1 Exercise 1: Fork-Join Model and Thread Management

1.1 Essential Parallelization Code

```c
#pragma omp parallel private(rank)
{
    rank = omp_get_thread_num();
    printf("Hello␣from␣rank␣%d\n", rank);
    if (rank == 0)  // Only master thread
        nthreads = omp_get_num_threads();
}
```

1.2 Conceptual Framework

Fork-Join Execution Model:

1. **Sequential region**: Single master thread
2. **Fork**: Master spawns worker threads at parallel directive
3. **Parallel region**: All threads execute concurrently
4. **Join**: Implicit barrier, workers terminate, master continues

Why Restrict Thread 0? The assignment `nthreads = ...` creates a write operation. If all N threads write simultaneously:

- Creates unnecessary memory bus contention
- Triggers cache coherence protocol overhead
- Violates single-responsibility principle

Non-deterministic Execution: Thread output order varies because OS scheduler controls:

- Core assignment

- Context switching timing
- Priority levels

Critical Design Principle: Parallel algorithms must be **order-independent**. Any dependency on execution order creates race conditions.

2 Exercise 2: Reduction Patterns and Work Distribution

2.1 Core Challenge

Computing $\pi \approx \sum_{i=0}^{n-1} \frac{4}{1+x_i^2} \Delta x$ requires accumulating partial sums without race conditions.

```
#pragma omp parallel reduction(+:global_sum)
{
    id = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    // Cyclic distribution
    for (i = id; i < num_steps; i += nthreads) {
        x = (i + 0.5) * step;
        global_sum += 4.0 / (1.0 + x * x);
    }
}
```

2.2 Reduction Mechanism Internals

The Race Condition Without Reduction:

```
Time    Thread 0        Thread 1        Memory(sum)
t0      read sum=0      read sum=0      0
t1      compute +5      compute +3      0
t2      write sum=5     write sum=3     5 (or 3!)
```

Result: Lost update! Final sum is wrong.
How `reduction(+:var)` Solves This:

1. **Privatization**: Creates `sum_private[nthreads]`
2. **Initialization**: Each `sum_private[i] = 0` (identity for +)
3. **Local accumulation**: Thread i updates only `sum_private[i]`
4. **Barrier**: Implicit synchronization
5. **Final reduction**: `sum = sum_private[0] + sum_private[1] + ...`

2.3 Work Distribution Strategies

Strategy	Iteration Assignment	Memory Access
Cyclic (used)	T0:0,4,8... T1:1,5,9...	Identical (Compute-bound)
Block	T0:0-24999, T1:25000-49999	Identical (Compute-bound)

Correction on Cache: Previous assumptions about stride are incorrect here. Since x is calculated mathematically ($x = (i + 0.5) * step$) rather than loaded from an array, there is no memory stride penalty. Both strategies have identical cache behavior.

Why Cyclic Here? Demonstrates manual control. It ensures load balancing if iterations had variable costs, but for this uniform loop, it adds complexity.

3 Exercise 3: Abstraction vs Manual Control

3.1 The Minimal Parallelization

```c
// Before: Serial
for (i = 0; i < num_steps; i++) {
    sum += 4.0 / (1.0 + x * x);
}

// After: One-line addition
#pragma omp parallel for reduction(+:sum) private(x)
for (i = 0; i < num_steps; i++) {
    sum += 4.0 / (1.0 + x * x);
}
```

3.2 What OpenMP Automates

Aspect	Automatic Handling by `parallel for`
Thread creation	Fork-join managed internally
Work distribution	Default: static block scheduling
Variable scoping	Loop index `i` automatically private
Synchronization	Implicit barrier before reduction

3.3 Performance Comparison

	Ex2 (Manual)	Ex3 (Auto)
Time	0.001373s	0.002336s
Distribution	Cyclic	Block

Why is Auto Slower? (Runtime Overhead) The manual version is faster at this small scale because it avoids the abstraction tax of OpenMP.

1. **Library Calls:** `pragma omp for` invokes runtime library functions to calculate loop bounds.
2. **Safety Checks:** The runtime manages implicit barriers and safety checks that the manual `int i = id` logic skips.
3. **Scale:** At 100k iterations, the computation is so fast that this runtime overhead (0.5-1ms) becomes a significant percentage of total time.

Practical Lesson: `parallel for` is safer and cleaner, but for extremely small loops, manual distribution can be slightly faster by bypassing runtime management.

4 Exercise 4: Matrix Multiplication - Memory Bounds and Scheduling

4.1 Parallelization Structure

```
#pragma omp parallel for collapse(2) schedule(runtime)
for (int i = 0; i < 1000; i++) {
    for (int j = 0; j < 1000; j++) {
        for (int k = 0; k < 1000; k++) {
            c[i*m + j] += a[i*n + k] * b[k*m + j];
        }}}
```

4.2 Why `collapse(2)`?

Without collapse:

- Parallelizes only i-loop: 1000 iterations
- With 16 threads: 62 iterations/thread

With collapse(2):

- Merges i and j: $1000 \times 1000 = 1,000,000$ iterations
- With 16 threads: 62,500 iterations/thread

Benefit: Finer granularity → better load balance when threads iterations.

Why NOT collapse(3)? The k-loop accumulates into `c[i][j]`. Multiple threads would write to same location → requires synchronization → kills performance.

4.3 Experimental Results Analysis

Threads	Sched	Chunk	Time (s)	Speedup
1	static	100	7.640	1.00
2	static	100	3.373	2.27
4	guided	100	2.620	2.92
8	dynamic	10	2.453	3.11
16	guided	1	2.748	2.78

Table 1: Best configuration for each thread count

4.4 Critical Performance Insights

4.4.1 Why Only 3.11x Speedup with 8 Threads?

1. Memory Bandwidth Saturation

Matrix size: $3 \times 1000^2 \times 8$ bytes = 24 MB
Each element of C requires:

- 1000 reads from A (row i)
- 1000 reads from B (column j)
- 1 write to C

Total memory traffic: $1000^3 \times (2 \times 8 \text{ bytes read} + 8 \text{ write}) = 16$ GB
Typical memory bandwidth: 20-40 GB/s per socket
Implication: Adding threads doesn't help when all threads are waiting for memory!
2. Cache Inefficiency

- Matrix B accessed column-wise: `b[k][j]`
- Non-contiguous access → new cache line every iteration
- L1 cache (32 KB) $\ll$ column size (8000 bytes)
- Cache miss rate ≈ 90-95%

3. False Sharing

Adjacent elements of C (e.g., `c[i][j]` and `c[i][j+1]`) often share a 64-byte cache line. When different threads write nearby elements:

- Each write invalidates other cores' caches
- Cache coherence protocol overhead
- Ping-ponging of cache lines between cores

4.5 Scheduling Strategy Analysis

Schedule	Characteristics
`static`	Chunks assigned at compile/start time. **Lowest overhead**, best for uniform work.
`dynamic`	Runtime work queue. Thread finishes chunk → gets next from queue. **Higher overhead** but handles load imbalance.
`guided`	Starts with large chunks, decreases exponentially. Compromise between static and dynamic.

Why dynamic(10) wins at 8 threads?

- Matrix multiplication has relatively uniform work per iteration
- But cache effects create some variance
- Dynamic handles variance without excessive overhead
- Chunk=10 balances scheduling cost vs granularity

4.6 The 16-Thread Performance Drop

Speedup at 8 threads: 3.11x
Speedup at 16 threads: 2.78x (slower!)
Root Causes:

1. **Memory bandwidth fully saturated**: All 16 threads fighting for same bus
2. **Thread overhead**: Context switching, synchronization costs scale with thread count

4.7 Key Lessons

1. **Identify the bottleneck**: CPU-bound vs memory-bound vs synchronization-bound
2. **Memory bandwidth is finite**: Adding threads to memory-bound code doesn't help
3. **More threads ≠ faster**: Overhead and contention can reduce performance

5 Exercise 5: Parallelizing Jacobi Method with OpenMP

5.1 Code Implementation

The Jacobi method solves a system of linear equations iteratively. The algorithm contains two main computational intense loops that were parallelized:

1. Computing the new vector approximation ($x_{courant}$).
2. Checking for convergence (finding the maximum error).

Below are the relevant modified sections of the code using OpenMP directives:

Listing 1: Parallelized Jacobi Iteration

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

    // Parallelize the calculation of x_courant
    // 'j' must be private to avoid race conditions in the inner
       loops
    #pragma omp parallel for private(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];
    }

    double absmax = 0;

    // Parallelize the convergence check
    // Use reduction to safely find the global maximum error
    #pragma omp parallel for reduction(max: absmax)
    for (i = 0; i < n; i++) {
        double curr = fabs(x[i] - x_courant[i]);
        if (curr > absmax)
            absmax = curr;
    }

    norme = absmax / n;
    // ... convergence check and swap ...
}
```

5.2 Performance Analysis

The code was executed with $N = 2000$ iterations using 1, 2, 4, 8, and 16 threads.

5.2.1 Experimental Data

$$\text{Speedup}(S_p) = \frac{T_1}{T_p} \quad , \quad \text{Efficiency}(E_p) = \frac{S_p}{p}$$

Threads (p)	**Time (s)**	**Speedup (S_p)**	**Efficiency (E_p)**
1	58.43	1.00	1.00 (100%)
2	36.22	1.61	0.81 (81%)
4	26.74	2.19	0.55 (55%)
8	23.67	2.47	0.31 (31%)
16	25.69	2.27	0.14 (14%)

Table 2: Performance Metrics for Jacobi Method ($N = 2000$)

5.2.2 Speedup and Efficiency Plots

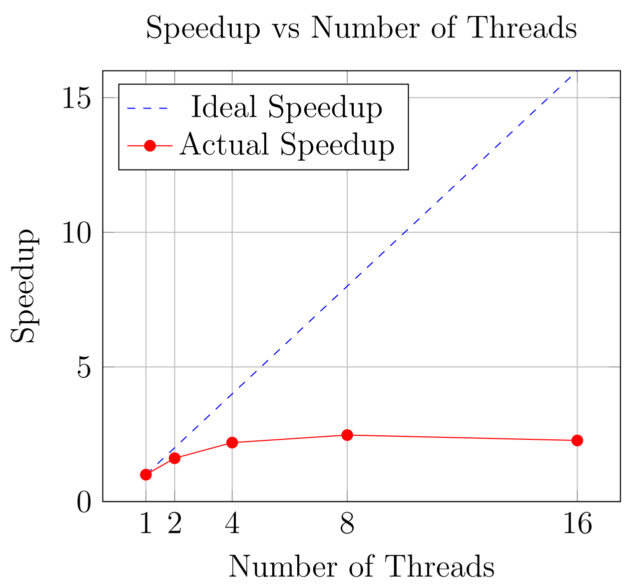

Figure 1: The speedup plateaus significantly after 4 threads and degrades at 16.

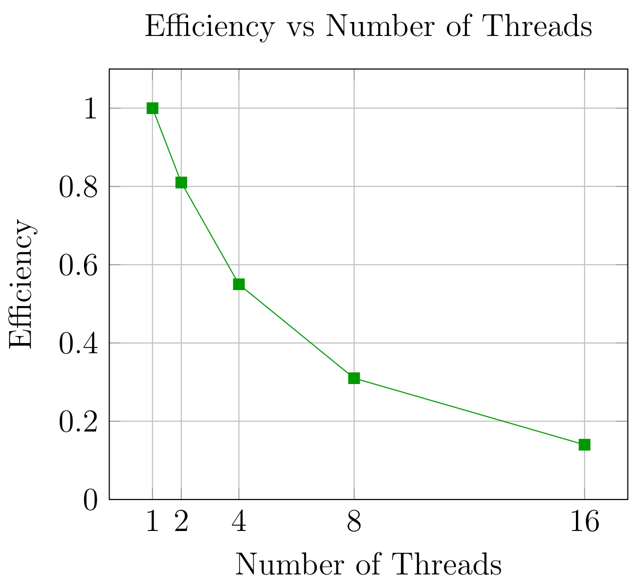

Figure 2: Efficiency drops rapidly as thread count increases.

5.3 Discussion

Observation: The speedup improves up to 8 threads ($S_8 \approx 2.47$) but fails to scale linearly. Notably, at 16 threads, the performance actually **degrades** ($25.69s$ vs $23.67s$ at 8 threads).

Root Causes:

- **Memory Bound Algorithm:** The Jacobi method involves dense matrix-vector multiplication. The CPU is likely waiting for data from RAM rather than performing calculations. Adding more threads just increases contention for memory bandwidth, saturating the bus.
- **Parallel Overhead:** At 16 threads, the cost of managing threads (creation, synchronization barriers at the end of loops) outweighs the benefit of splitting the work further.
- **False Sharing / Cache Issues:** Although 'x_courant' is accessed distinctly by index i, the frequent reads of the shared matrix a and vector x across all cores cause cache thrashing.