OpenMP Implementation for Matrix Multiplication and Knapsack

Assem 120210321

1 Part A: Dense Matrix Multiplication

1.1 Implementation Approach

For the matrix multiplication problem ($N = M = L = 256$), we implemented both sequential and parallel versions to multiply two matrices $A(N \times M)$ and $B(M \times L)$ producing result matrix $C(N \times L)$.

The standard matrix multiplication algorithm is:

```
for i = 0 to N-1:
    for j = 0 to L-1:
        C[i][j] = 0
        for k = 0 to M-1:
            C[i][j] += A[i][k] * B[k][j]
```

1.2 Parallelization Strategy

1. **Outer Loop Parallelization:** We parallelized the outermost loop (the i loop) using OpenMP's `#pragma omp parallel for` directive because:
   - Each row of the result matrix can be computed independently
   - It minimizes synchronization overhead
   - It provides good load balancing with identical workload per iteration
2. **Thread Count Testing:** The implementation tests with varying thread counts (2, 4, 8, 16) to analyze scalability.
3. **Data Sharing:** Matrices A and B are shared across threads (read-only), while each thread writes to its own portion of matrix C, avoiding race conditions.

1.3 Performance Results

1.4 Analysis

The implementation shows excellent parallel scaling up to 8 threads, with near-linear speedup progression from 2 threads (1.56x) to 8 threads (4.67x). This confirms our parallelization strategy is effective for this problem size.

Configuration	Execution Time (ms)	Speedup
Sequential	14	1.00x
2 threads	9	1.56x
4 threads	6	2.33x
8 threads	3	4.67x
16 threads	4	3.50x

Table 1: Matrix Multiplication Performance Results

Key observations:

- **Positive scaling trend:** Performance improves significantly as threads increase from 2 to 8
- **Peak efficiency at 8 threads:** Optimal performance is achieved with 8 threads, matching the hardware's effective parallel processing capacity
- **Diminishing returns at 16 threads:** Performance slightly degrades when moving from 8 to 16 threads (4.67x to 3.5x speedup), likely due to thread management overhead and resource contention

2 Part B: Pseudo-Polynomial Knapsack

2.1 Implementation Approach

The knapsack problem ($N = C = 1024$) is implemented using a 2D dynamic programming table where:

- `dp[i][w]` represents the maximum value attainable with first i items and weight capacity w
- The recurrence relation is:

```
dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i]] + value[i])
```

2.2 Parallelization Strategy

1. **Inner Loop Parallelization:** We parallelized the inner loop (over weights) using OpenMP:

   ```
   #pragma omp parallel for
   for (int w = 0; w <= capacity; w++)

   ```

   This choice was made because:

   - The outer loop has dependencies between iterations (`dp[i]` depends on `dp[i-1]`)

- The inner loop iterations at each level i are completely independent
- This provides significant parallelism (1024 iterations) at each step

2. **Data Dependencies:** The algorithm respects dependencies between item levels by keeping the outer loop sequential
3. **Thread Count Testing:** Various thread counts are tested to measure scalability

2.3 Performance Results

Configuration	Execution Time (ms)	Speedup
Sequential	3	1.00x
2 threads	25	0.12x
4 threads	33	0.09x
8 threads	47	0.06x
16 threads	77	0.04x

Table 2: Knapsack Performance Results

2.4 Analysis

The knapsack implementation shows significant performance degradation when parallelized, with execution times increasing as more threads are added.

Key observations:

- **Fast sequential performance:** The sequential version completes in just 3 ms, indicating high efficiency for this problem size
- **Parallelization overhead:** The parallel versions are consistently slower than sequential, with slowdown worsening as thread count increases
- **Causes of poor performance:**
  - Thread creation overhead exceeds computational benefits
  - Work per thread is too small to amortize parallel execution costs
  - Dynamic programming table access patterns may cause cache thrashing
  - Implicit barriers at parallel region boundaries add overhead

3 Comment

These results show an important principle: **not all problems benefit equally from parallelization**. Key lessons:

1. **Problem size matters:** For small problems, parallelization overhead can outweigh benefits
2. **Algorithm characteristics affect parallelizability:**
   - Matrix multiplication has independent computations
   - Knapsack DP has fine-grained dependencies challenging effective parallelization
3. **Parallelization strategy is critical:**
   - Row-based decomposition works well for matrix multiplication
   - Inner-loop parallelization was ineffective for knapsack
4. **Measurement is essential:** Always verify that parallelization actually improves performance