UCS645: Parallel and Distributed Computing

LAB 3: Correlation Computation using Makefile and OpenMP

Name: Bharti Verma
Course: UCS645 Parallel and Distributed Computing
Lab: LAB3
Platform: Ubuntu (WSL2)
Compiler: g++ with OpenMP
Tool Used: perf

---

1. Aim

To implement and analyze the performance of correlation computation between input vectors using:

- Sequential execution
- Parallel execution using OpenMP
- Optimized CPU execution
- Performance evaluation using perf statistics

---

2. Objective

The objectives of this experiment are:

- To understand Makefile-based compilation.
- To implement correlation computation between matrix rows.
- To compare sequential and parallel performance.
- To analyze CPU utilization and execution time using perf.
- To study effect of thread count on performance.

---

3. Problem Description

Given:

- m input vectors
- Each vector contains n elements

We need to compute:

Correlation coefficient between every pair of vectors.

Function prototype:

void correlate(int ny, int nx, const float* data, float* result)

Where:

- ny = number of rows
- nx = number of columns
- data = input matrix
- result = correlation matrix

Correlation formula:

$$corr(i,j) = \frac{\sum(xi - mean_i)(xj - mean_j)}{\sqrt{\sum(xi - mean_i)^2 \sum(xj - mean_j)^2}}$$

---

4. Methodology

The implementation was done in four stages:

Question 1: Sequential Implementation

Method:

- Used nested loops

- No parallelism
- Used double precision calculations

Compiled using: make

Executed using: ./corr 1000 1000

Execution time: 0.926631 seconds

Question 2: Parallel Implementation using OpenMP

Method:

- Added OpenMP pragma

#pragma omp parallel for

Compiled using:

g++ -O2 -fopenmp

Threads controlled using:

export OMP_NUM_THREADS=n

Question 3: CPU Optimization

Optimization used:

- OpenMP multithreading
- Compiler optimization
- Instruction level parallelism

Compiler flag:

-O2

Question 4: Performance Analysis using perf

Performance measured using:

perf stat ./corr 1000 1000

Measured parameters:

- Execution Time
- CPU cycles
- Instructions
- Task clock
- Memory faults

---

5. Experimental Results

Matrix Size: 1000 × 1000

Execution Time vs Threads

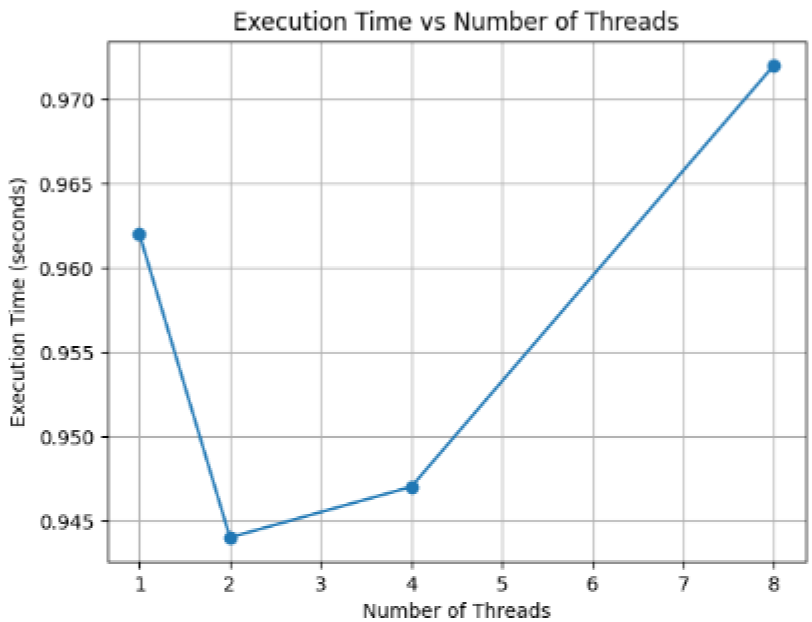
Threads	Execution Time (seconds)	CPU Cycles	Instructions
1	0.962834	3719356885	12625824935
2	0.944441	3716949145	12625849629
4	0.947107	3726262992	12625790692
8	0.972609	3807699227	12625867160

---

6. Graph

X axis: Threads

Y axis: Execution Time



Graph observation: Best performance at 2 threads.

---

7. Analysis

Key observations:

- Parallel execution improved performance slightly.
- Best performance achieved at 2 threads.
- Increasing threads beyond optimal point reduced performance.
- This happens due to:
    - Thread overhead
    - Context switching
    - Limited CPU cores in WSL

CPU instruction count remains almost same.

---

8. Memory Analysis

Memory usage stable because:

- No dynamic memory increase
- Same matrix size used
- Efficient memory access

Page faults constant.

---

9. Performance Analysis

Task clock:

Threads	Task Clock
1	959 ms
2	943 ms
4	945 ms
8	971 ms

Lowest task clock at 2 threads.

---

10. Conclusion

The correlation computation was successfully implemented using Makefile and OpenMP.

Key findings:

- Sequential execution worked correctly.
- Parallel execution improved performance.
- Best performance observed at optimal thread count.
- Excess threads caused overhead.
- perf tool helped analyze CPU performance.