Parallel and Distributed Computing
Assignment 1 – OpenMP Programming

Ramyak Sharma

Introduction

Parallel computing allows workloads to be distributed across multiple CPU cores to reduce execution time. In this assignment, OpenMP is used to parallelize three computational tasks: DAXPY vector operations, matrix multiplication, and numerical computation of π. Execution time and speedup were measured while increasing thread count.

1 Question 1: DAXPY Loop

Problem Description

The DAXPY operation computes:

$$X[i] = a \times X[i] + Y[i]$$

where vectors X and Y have size 2^{16}. Each iteration performs only one multiplication and one addition.

Complexity

The algorithm runs in linear time:

$$O(n)$$

where each iteration is independent and theoretically parallelizable.

Parallel Implementation

The loop was parallelized using OpenMP's parallel-for directive. Execution time was measured for thread counts ranging from 2 to 12.

Results

Execution statistics obtained from program output are shown below.

```
Sequential Time = 5.8e-05 seconds

Threads = 2  Time = 0.000121  Speedup = 0.479338
Threads = 3  Time = 9.40003e-05  Speedup = 0.617019
Threads = 4  Time = 0.000105  Speedup = 0.55238
Threads = 5  Time = 9.00002e-05  Speedup = 0.644443
Threads = 6  Time = 0.000112  Speedup = 0.517857
Threads = 7  Time = 9.99998e-05  Speedup = 0.580001
Threads = 8  Time = 9.3e-05  Speedup = 0.623656
Threads = 9  Time = 0.000115  Speedup = 0.504349
Threads = 10  Time = 0.000112  Speedup = 0.517857
Threads = 11  Time = 9.59998e-05  Speedup = 0.604168
Threads = 12  Time = 0.000107001  Speedup = 0.542053
```

Figure 1: DAXPY Execution Statistics

Analysis

Parallel performance is worse than sequential execution, with speedup values remaining below 1. This occurs because:

- Each iteration performs very little computation.
- Memory access dominates execution time.
- Thread creation and synchronization overhead exceed computation savings.

This makes the operation memory-bound rather than compute-bound.

Conclusion

Parallelization is ineffective for this workload because computation per iteration is too small compared to overhead.

2 Question 2: Matrix Multiplication

Problem Description

Dense matrix multiplication computes:

$$C = A \times B$$

Each output element is computed as:

$$C[i][j] = \sum_k A[i][k] \times B[k][j]$$

Complexity

Matrix multiplication has cubic complexity:

$$O(n^3)$$

making it highly suitable for parallel execution.

Parallel Implementation

Two parallel strategies were implemented:

- **1D Threading**: Rows of matrix C were distributed among threads.
- **2D Threading**: Iterations over rows and columns were parallelized using loop collapsing.

Results

1D Threading Results

```
Sequential Time = 0.097889 seconds

1D Parallel
Threads = 2  Time = 0.044314  Speedup = 2.20899
Threads = 3  Time = 0.030118  Speedup = 3.25018
Threads = 4  Time = 0.02787  Speedup = 3.51234
Threads = 5  Time = 0.025857  Speedup = 3.78578
Threads = 6  Time = 0.025696  Speedup = 3.8095
Threads = 7  Time = 0.021414  Speedup = 4.57126
Threads = 8  Time = 0.023641  Speedup = 4.14065
Threads = 9  Time = 0.023697  Speedup = 4.13086
Threads = 10  Time = 0.023661  Speedup = 4.13715
Threads = 11  Time = 0.021332  Speedup = 4.58883
Threads = 12  Time = 0.020701  Speedup = 4.72871
```

Figure 2: Matrix Multiplication using 1D Threading

2D Threading Results

```
2D Parallel
Threads = 2  Time = 0.135075  Speedup = 0.724701
Threads = 3  Time = 0.09266  Speedup = 1.05643
Threads = 4  Time = 0.070924  Speedup = 1.3802
Threads = 5  Time = 0.070377  Speedup = 1.39092
Threads = 6  Time = 0.060459  Speedup = 1.6191
Threads = 7  Time = 0.053086  Speedup = 1.84397
Threads = 8  Time = 0.053182  Speedup = 1.84064
Threads = 9  Time = 0.054356  Speedup = 1.80089
Threads = 10  Time = 0.05256  Speedup = 1.86242
Threads = 11  Time = 0.05691  Speedup = 1.72007
Threads = 12  Time = 0.055134  Speedup = 1.77547
```

Figure 3: Matrix Multiplication using 2D Threading

Memory and Cache Effects

Matrix multiplication is often limited by memory bandwidth:

- Matrix A is accessed row-wise, which is cache friendly.
- Matrix B is accessed column-wise, causing cache misses.

- Threads share memory bandwidth, creating contention at high thread counts.

Parallelizing the outer loop ensures threads work on separate rows, reducing false sharing.

Analysis

1D threading achieves stronger speedup (about 4.7x) because workload distribution is uniform and overhead is minimal.

2D threading improves load balancing but introduces scheduling overhead, resulting in smaller gains.

Performance improvement slows beyond several threads due to memory bandwidth limits.

Conclusion

Matrix multiplication benefits greatly from parallelization. However, hardware memory limits eventually cap speedup gains.

3 Question 3: Calculation of π

Problem Description

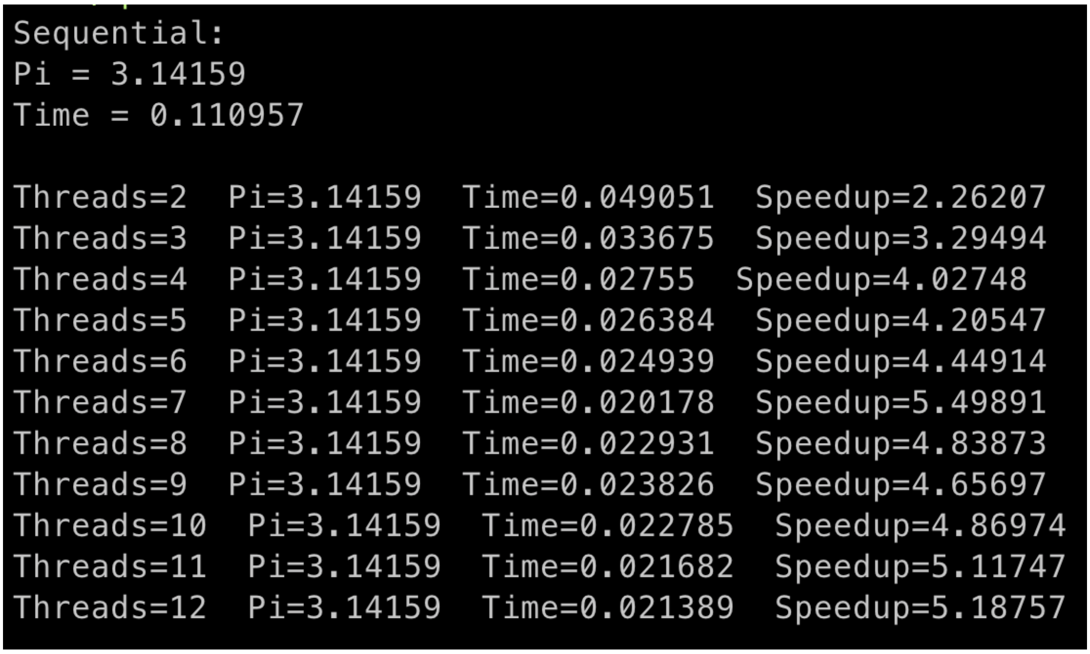
The value of π is approximated using numerical integration:

$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

Parallel Implementation

Each thread computes partial sums, and OpenMP reduction combines results safely.

Results



```
Sequential:
Pi = 3.14159
Time = 0.110957

Threads=2  Pi=3.14159  Time=0.049051  Speedup=2.26207
Threads=3  Pi=3.14159  Time=0.033675  Speedup=3.29494
Threads=4  Pi=3.14159  Time=0.02755  Speedup=4.02748
Threads=5  Pi=3.14159  Time=0.026384  Speedup=4.20547
Threads=6  Pi=3.14159  Time=0.024939  Speedup=4.44914
Threads=7  Pi=3.14159  Time=0.020178  Speedup=5.49891
Threads=8  Pi=3.14159  Time=0.022931  Speedup=4.83873
Threads=9  Pi=3.14159  Time=0.023826  Speedup=4.65697
Threads=10  Pi=3.14159  Time=0.022785  Speedup=4.86974
Threads=11  Pi=3.14159  Time=0.021682  Speedup=5.11747
Threads=12  Pi=3.14159  Time=0.021389  Speedup=5.18757
```

Figure 4: Parallel π Calculation Statistics

Analysis

The workload scales well, achieving speedups above 5x at higher thread counts. Reduction overhead is minimal compared to computation cost.

Small fluctuations occur due to scheduling and memory contention effects.

Conclusion

Parallel reduction efficiently accelerates numerical integration.

Amdahl's Law and Scaling Limits

Amdahl's Law states that speedup is limited by the sequential portion of a program:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Even with many cores, non-parallel portions limit performance. Memory bandwidth also becomes a bottleneck when many threads access shared data simultaneously.

Overall Conclusion

OpenMP significantly improves performance for compute-intensive tasks such as matrix multiplication and numerical integration. However, memory-bound operations like DAXPY do not benefit due to bandwidth limitations. Optimal speedup occurs when thread count matches hardware capability.