Parallel and Distributed Computing
Assignment 1 – OpenMP Programming

Ramyak Sharma

Introduction

This assignment evaluates performance gains obtained using OpenMP parallel programming. Three computational problems were parallelized and execution times were compared for varying thread counts.

1 Question 1: DAXPY Loop

Problem Statement

The DAXPY operation performs

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

on vectors of size 2^{16}. Execution time and speedup were measured while increasing thread count.

Implementation

Sequential execution was first measured. The loop was then parallelized using OpenMP and executed with thread counts ranging from 2 to 12.

Results

Execution statistics from the program 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

Observation

Sequential execution time is extremely small. Parallel execution introduces thread creation and synchronization overhead, causing parallel runtimes to become larger than sequential runtime.

Consequently, speedup remains below 1 for all thread counts. The DAXPY computation is memory-bound, meaning threads compete for memory bandwidth rather than computation time.

Conclusion

Parallelization does not improve performance for this workload due to low computation per iteration and memory access limitations.

2 Question 2: Matrix Multiplication

Problem Statement

Matrix multiplication of large matrices was parallelized using two methods:

- 1D threading
- 2D threading

Implementation

Sequential matrix multiplication was implemented first. Two parallel strategies were tested:

- 1D threading parallelizes rows so each thread computes separate output rows.
- 2D threading distributes work across rows and columns simultaneously.

Execution time was measured across multiple thread counts.

Results

1D Threading

```
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

```
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

Observation

1D threading provides strong speedup, reaching approximately 4.7× improvement at 12 threads. Work distribution across rows minimizes scheduling overhead.

2D threading improves load distribution but introduces extra scheduling overhead, resulting in smaller speedups compared to 1D threading.

Performance scaling slows after several threads due to memory bandwidth and cache limitations.

Conclusion

Matrix multiplication benefits significantly from parallelization. 1D threading proved more efficient on the tested hardware due to lower overhead.

3 Question 3: Calculation of π

Problem Statement

The value of π is computed using numerical integration:

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

Implementation

The computation loop was parallelized using OpenMP reduction to combine partial sums computed by threads.

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

Observation

Parallel execution shows strong improvement, reaching speedups above 5× around 7–12 threads. Performance variation occurs due to scheduling and memory contention effects.

Reduction overhead is minimal compared to computation cost, allowing efficient scaling.

Conclusion

Parallel reduction effectively accelerates numerical integration. Optimal speedup occurs when thread count matches hardware capabilities.

Overall Conclusion

OpenMP significantly improves performance for compute-intensive tasks such as matrix multiplication and numerical integration. However, memory-bound operations like

DAXPY may not benefit due to bandwidth limitations and thread overhead. Optimal thread count typically corresponds to available hardware cores.