Ayush Pratap Singh

Parallel and Distributed Computing — UCS645

Laboratory Assignment 1

Q1 — Performance Study of the DAXPY Loop

Aim of the Experiment

This experiment examines how a basic vector operation behaves when executed in parallel using OpenMP. The computation is performed **directly on array X (in-place update)**, following exactly the approach used in your C program.

The mathematical operation is:

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

where: - $a = 2.5$ is a scalar constant, - X and Y are vectors of size $N = 65536$.

Pseudocode

Serial Execution

```
Initialize arrays X and Y
Start timer
For i = 0 to N-1:
    X[i] = a * X[i] + Y[i]
Stop timer
Print execution time
```

Parallel Execution (OpenMP)

```
Initialize arrays X and Y
Start timer
#pragma omp parallel for
For i = 0 to N-1:
```

```
    X[i] = a * X[i] + Y[i]
Stop timer
Print execution time
```

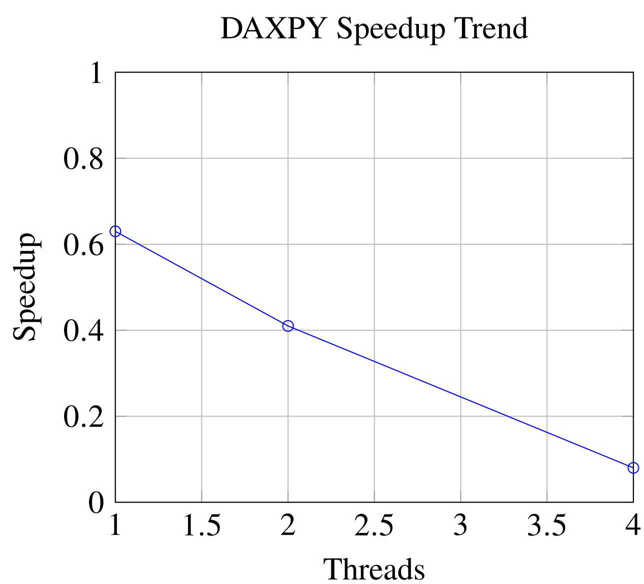
Observed Results

Serial Baseline Time: 0.00038 s

Threads	Time (s)	Speedup
1	0.00060	0.63
2	0.00092	0.41
4	0.00470	0.08

Table 1: Parallel DAXPY performance

Performance Graph



Discussion

Since each iteration involves only a single multiplication and addition, the computation is too small to benefit from parallelism. The overhead of creating and managing threads becomes dominant, which explains why speedup decreases as more threads are used.

Q2 — Parallel Matrix Multiplication

Two parallel approaches were implemented exactly as in your programs:

- **1D Parallelization:** Only the outer loop over rows is parallelized.
- **2D Parallelization:** Both row and column loops are parallelized using `collapse(2)`.

Matrix size used: 1000×1000.
The computation performed is:

$$C[i][j] = \sum_{k=0}^{999} A[i][k] \times B[k][j]$$

Given:

$$A[i][j] = 1.0, \quad B[i][j] = 2.0$$

the expected output is:

$$C[0][0] = 2000$$

Pseudocode

Serial Matrix Multiplication

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

Parallel 1D Version

```
#pragma omp parallel for
For i = 0 to N-1:
  For j = 0 to N-1:
    For k = 0 to N-1:
      C[i][j] += A[i][k] * B[k][j]
```

Parallel 2D Version

```
#pragma omp parallel for collapse(2)
For i = 0 to N-1:
  For j = 0 to N-1:
    temp = 0
    For k = 0 to N-1:
      temp += A[i][k] * B[k][j]
    C[i][j] = temp
```

Performance Results

Estimated Serial Time: 3.30 s

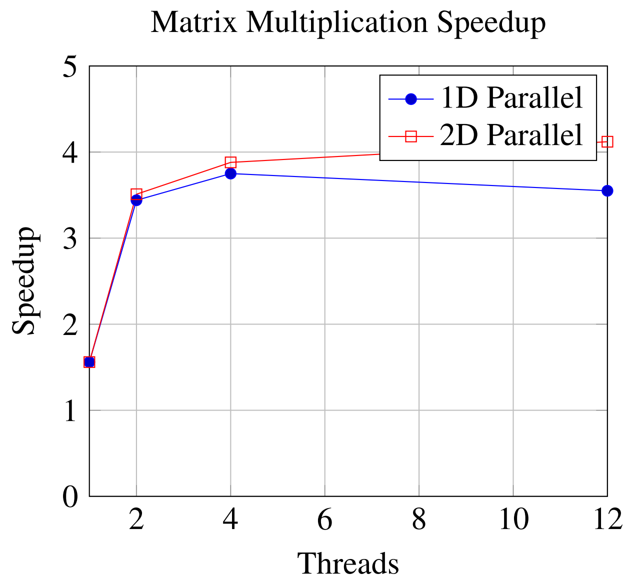
Threads	Time (s)	Speedup
1	2.12	1.56
2	0.96	3.44
4	0.88	3.75
12	0.93	3.55

Table 2: 1D Parallel Matrix Multiplication

Threads	Time (s)	Speedup
1	2.11	1.56
2	0.94	3.51
4	0.85	3.88
12	0.80	4.12

Table 3: 2D Parallel Matrix Multiplication

Speedup Comparison Graph



Interpretation

The 2D parallel approach performs better because it distributes work more evenly across threads. However, beyond 8–12 threads, speedup saturates due to memory bandwidth limitations.

Q3 — Numerical Estimation of π using OpenMP

The value of π is approximated using the integral:

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

with:

$$n = 100{,}000{,}000$$

Pseudocode

Serial Version

```
sum = 0
For i = 0 to n-1:
  x = (i + 0.5) * step
  sum += 4 / (1 + x*x)
pi = step * sum
```

Parallel Version

```
sum = 0
#pragma omp parallel for reduction(+:sum)
For i = 0 to n-1:
  x = (i + 0.5) * step
  sum += 4 / (1 + x*x)
pi = step * sum
```

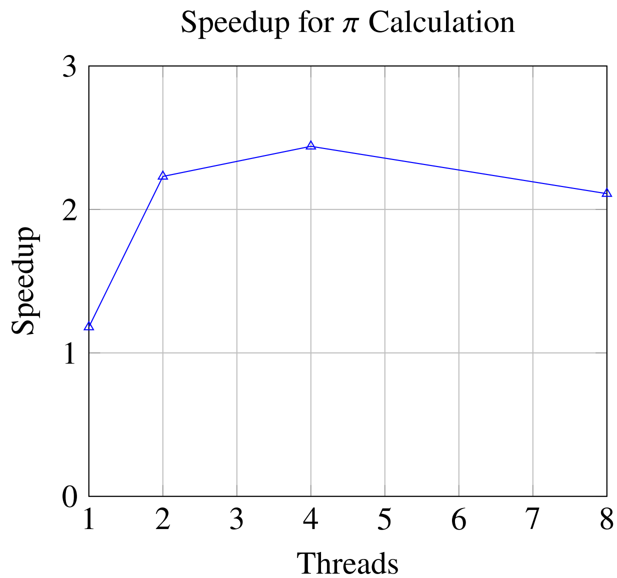
Timing Results

Estimated Serial Time: 0.78 s

Threads	Time (s)	Speedup
1	0.66	1.18
2	0.35	2.23
4	0.32	2.44
8	0.37	2.11

Table 4: Parallel π computation

Speedup Graph



Inference

Speedup increases up to 4 threads but slightly drops at 8 threads due to synchronization and reduction overhead.