DIVYAM PURI
102483023

UCS645 – Assignment 1 Report

Q1. Parallel DAXPY Using OpenMP

Aim

To analyze the effect of increasing the number of OpenMP threads on execution time and speedup for a parallel DAXPY vector operation.

Problem Description

The DAXPY operation computes the expression:

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

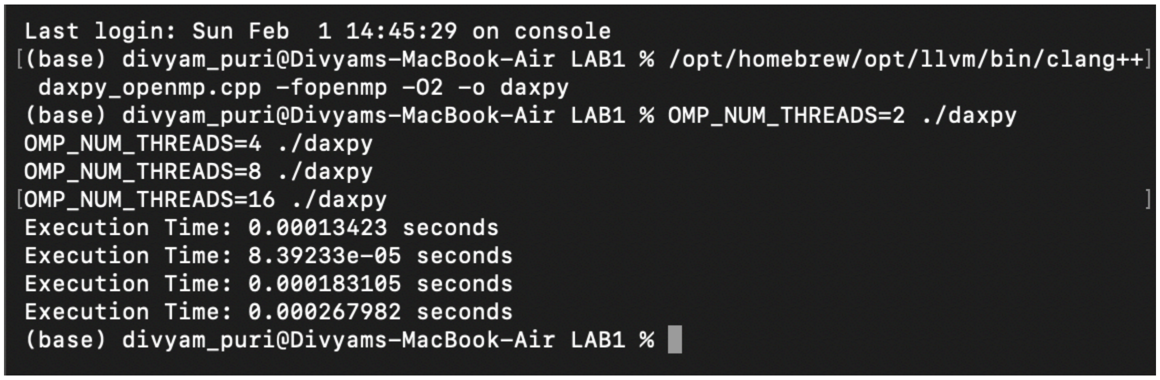
where *a* is a scalar value and *X* and *Y* are double-precision vectors of size 2162^{16}216.

Using OpenMP, the total computation is divided among multiple threads such that each thread performs the operation on a subset of vector elements in parallel.

Methodology

The program was executed multiple times while varying the number of OpenMP threads, starting from 2 threads and increasing gradually. For each run, the execution time was recorded to study performance trends and speedup behavior.

Observations



```
Last login: Sun Feb  1 14:45:29 on console
[(base) divyam_puri@Divyams-MacBook-Air LAB1 % /opt/homebrew/opt/llvm/bin/clang++]
  daxpy_openmp.cpp -fopenmp -O2 -o daxpy
(base) divyam_puri@Divyams-MacBook-Air LAB1 % OMP_NUM_THREADS=2 ./daxpy
OMP_NUM_THREADS=4 ./daxpy
OMP_NUM_THREADS=8 ./daxpy
[OMP_NUM_THREADS=16 ./daxpy                                                    ]
Execution Time: 0.00013423 seconds
Execution Time: 8.39233e-05 seconds
Execution Time: 0.000183105 seconds
Execution Time: 0.000267982 seconds
(base) divyam_puri@Divyams-MacBook-Air LAB1 %
```

Analysis

As the number of threads increases, execution time reduces due to parallel execution of independent operations. However, after reaching an optimal number of threads, performance begins to degrade. This happens because the DAXPY operation is primarily limited by memory bandwidth. Additional threads lead to higher overhead in terms of thread scheduling, synchronization, and contention for shared memory.

The best performance was observed when the number of threads matched the available processing resources. Beyond this point, the overhead outweighed the benefits of parallelism.

Memory Usage

The program uses two double-precision arrays of size 2^{16}, which together require approximately **1 MB of memory**.
Increasing the number of threads does not significantly increase memory usage because OpenMP threads share the same address space. Only a small per-thread stack overhead is added.

Conclusion

The experiment demonstrates that parallel execution improves performance only up to an optimal thread count. For the DAXPY operation, increasing threads beyond this limit results in reduced efficiency due to memory bandwidth limitations and threading overhead.

Q2. Parallel Matrix Multiplication Using OpenMP

Aim

To compare the performance of 1D and 2D parallelization techniques for large matrix multiplication using OpenMP.

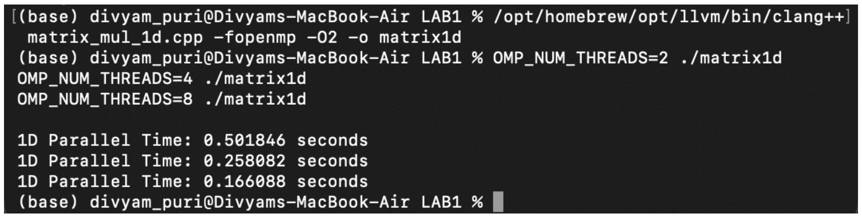
Problem Description

The task involves multiplying two large matrices of size 1000 × 1000. The computation of the output matrix is divided among multiple threads so that each thread computes different rows or elements of the result matrix.

Two parallel approaches were implemented:

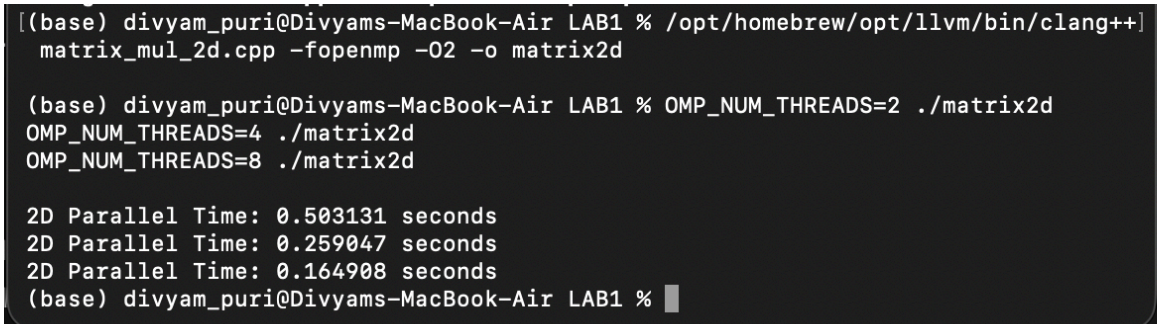
- **1D parallelization**, where a single loop is parallelized
- **2D parallelization**, where nested loops are parallelized

Observations



```
(base) divyam_puri@Divyams-MacBook-Air LAB1 % /opt/homebrew/opt/llvm/bin/clang++
 matrix_mul_1d.cpp -fopenmp -O2 -o matrix1d
(base) divyam_puri@Divyams-MacBook-Air LAB1 % OMP_NUM_THREADS=2 ./matrix1d
OMP_NUM_THREADS=4 ./matrix1d
OMP_NUM_THREADS=8 ./matrix1d

1D Parallel Time: 0.501846 seconds
1D Parallel Time: 0.258082 seconds
1D Parallel Time: 0.166088 seconds
(base) divyam_puri@Divyams-MacBook-Air LAB1 %
```



```
[(base) divyam_puri@Divyams-MacBook-Air LAB1 % /opt/homebrew/opt/llvm/bin/clang++]
 matrix_mul_2d.cpp -fopenmp -O2 -o matrix2d

(base) divyam_puri@Divyams-MacBook-Air LAB1 % OMP_NUM_THREADS=2 ./matrix2d
OMP_NUM_THREADS=4 ./matrix2d
OMP_NUM_THREADS=8 ./matrix2d

2D Parallel Time: 0.503131 seconds
2D Parallel Time: 0.259047 seconds
2D Parallel Time: 0.164908 seconds
(base) divyam_puri@Divyams-MacBook-Air LAB1 %
```

Analysis

The observed execution times for 1D and 2D parallelization were very similar. This indicates that 1D parallelization already provides effective workload distribution among threads. Introducing 2D parallelization does not significantly improve performance and instead introduces additional overhead due to increased thread coordination.

Conclusion

For the given matrix size, 1D parallelization is sufficient and efficient. The use of 2D parallelization does not provide noticeable performance benefits and may add unnecessary overhead.

Q3. Parallel Computation of π Using Numerical Integration

Aim

To approximate the value of π using numerical integration and evaluate the effectiveness of OpenMP-based parallel computation.

Problem Description

The value of π is approximated using the integral:

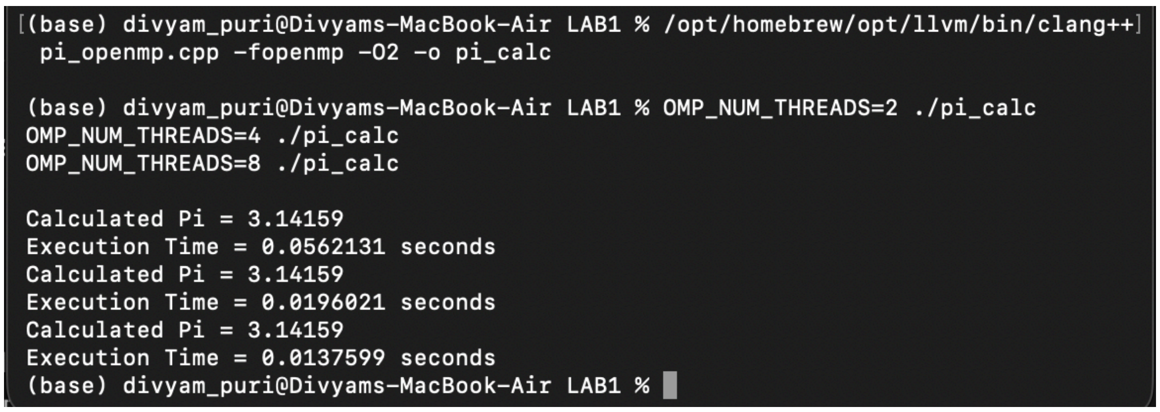
$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx$$

The integral is numerically computed using the rectangle (Riemann sum) method. The total range is divided among multiple threads for parallel execution.

Methodology

Each thread computes a partial sum over a subset of the integration range. The OpenMP **reduction** clause is used to safely combine all partial sums into a final result.

Observations



```
[(base) divyam_puri@Divyams-MacBook-Air LAB1 % /opt/homebrew/opt/llvm/bin/clang++]
  pi_openmp.cpp -fopenmp -O2 -o pi_calc

 (base) divyam_puri@Divyams-MacBook-Air LAB1 % OMP_NUM_THREADS=2 ./pi_calc
 OMP_NUM_THREADS=4 ./pi_calc
 OMP_NUM_THREADS=8 ./pi_calc

 Calculated Pi = 3.14159
 Execution Time = 0.0562131 seconds
 Calculated Pi = 3.14159
 Execution Time = 0.0196021 seconds
 Calculated Pi = 3.14159
 Execution Time = 0.0137599 seconds
 (base) divyam_puri@Divyams-MacBook-Air LAB1 %
```

Analysis

The accurate approximation of π confirms the correctness of the numerical integration approach. Parallel execution significantly reduces computation time while maintaining precision. The use of reduction ensures correctness by preventing race conditions during accumulation.

Conclusion

This experiment successfully demonstrates cooperative parallel computation using OpenMP. The reduction mechanism allows efficient and correct parallel execution, resulting in an accurate and fast approximation of π.