Parallel and Distributed Computing

Assignment 1

102303654

Ques1. DAXPY Vector Operation

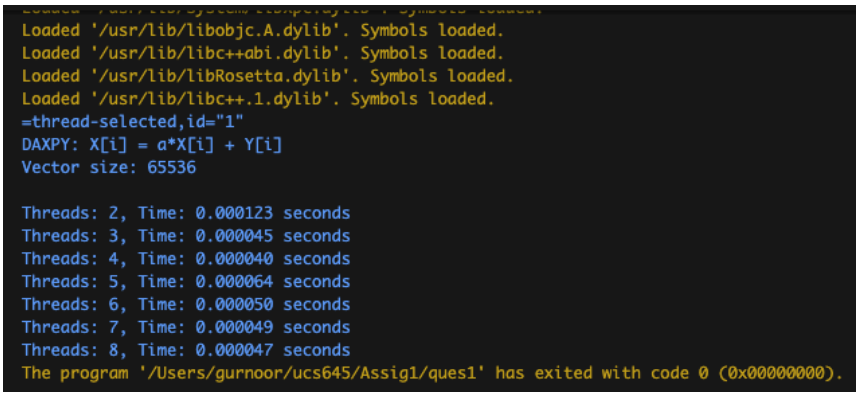
Operation: X[i]=a×X[i]+Y[i]

Code:-

Assig1 > ques1.cpp > main()

```c
#include <stdio.h>
#include <omp.h>

#define N 65536
int main() {
    double X[N], Y[N];
    double a = 2.5;
    double start_time, end_time;

    for (int i = 0; i < N; i++) {
        X[i] = i * 1.0;
        Y[i] = i * 0.5;
    }

    printf("DAXPY: X[i] = a*X[i] + Y[i]\n");
    printf("Vector size: %d\n\n", N);

    // Test different thread counts
    for (int threads = 2; threads <= 8; threads++) {
        for (int i = 0; i < N; i++) {
            X[i] = i * 1.0;
        }

        omp_set_num_threads(threads);
        start_time = omp_get_wtime();

        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
            X[i] = a * X[i] + Y[i];
        }

        end_time = omp_get_wtime();

        printf("Threads: %d, Time: %f seconds\n", threads, end_time - start_time);
    }

    return 0;
}
```

Output:-



Speed-Up Calculation:-

Speed-Up = TN threads/T2 threads

(Here, T2=0.000123 seconds)

Threads	Time(s)	Speed-Up
2	0.000123	1.00
3	0.000045	2.73
4	0.000040	3.08
5	0.000064	1.92
6	0.000050	2.46
7	0.000049	2.51
8	0.000047	2.61

Observation:
Speed-up increases gradually from thread 2-4, achieving maximum speedup at around 4 threads. Here, the cores of cpu are bring fully utilized.
Then after 4 threads, speedup becomes irregular. At 5 threads , speedup decreases gradually. Whereas , the speedup makes some recovery in threads 6-8 but couldnt match the speedup achieved on 4 threads.

Conclusion:-
So the best performance of DAXPY operation is achieved at 4 threads.

Q2. Matrix Multiplication using OpenMP

Matrix size: 1000 × 1000

Two parallel strategies were implemented:

1. 1D Threading (outer loop parallelization)
2. 2D Threading using collapse(2)

Code:-

Assig1 > ques2.cpp > main()

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define SIZE 1000

int main() {
    // Allocate matrices dynamically (on heap, not stack)
    double **A = (double **)malloc(SIZE * sizeof(double *));
    double **B = (double **)malloc(SIZE * sizeof(double *));
    double **C = (double **)malloc(SIZE * sizeof(double *));

    for (int i = 0; i < SIZE; i++) {
        A[i] = (double *)malloc(SIZE * sizeof(double));
        B[i] = (double *)malloc(SIZE * sizeof(double));
        C[i] = (double *)malloc(SIZE * sizeof(double));
    }

    double start_time, end_time;

    // Initialize matrices
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            A[i][j] = i + j;
            B[i][j] = i - j;
            C[i][j] = 0;
        }
    }

    printf("Matrix Multiplication - 1000x1000\n");
    printf("================================\n\n");

    // ========== 1D THREADING ==========
    printf("1D Threading (Outer Loop Parallelization):\n");
    printf("-----------------------------------------\n");

    for (int threads = 2; threads <= 8; threads++) {
        omp_set_num_threads(threads);
        start_time = omp_get_wtime();

        #pragma omp parallel for
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                C[i][j] = 0;
                for (int k = 0; k < SIZE; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }

        end_time = omp_get_wtime();
        printf("Threads: %d, Time: %f seconds\n", threads, end_time - start_time
    }

    printf("\n");

    // ========== 2D THREADING ==========
    printf("2D Threading (Nested Loop Parallelization with collapse):\n");
    printf("---------------------------------------------------------\n");

```

```
60
61      for (int threads = 2; threads <= 8; threads++) {
62          omp_set_num_threads(threads);
63          start_time = omp_get_wtime();
64
65          #pragma omp parallel for collapse(2)
66          for (int i = 0; i < SIZE; i++) {
67              for (int j = 0; j < SIZE; j++) {
68                  C[i][j] = 0;
69                  for (int k = 0; k < SIZE; k++) {
70                      C[i][j] += A[i][k] * B[k][j];
71                  }
72              }
73          }
74
75          end_time = omp_get_wtime();
76          printf("Threads: %d, Time: %f seconds\n", threads, end_time - start_time);
77      }
78
79      // Free allocated memory
80      for (int i = 0; i < SIZE; i++) {
81          free(A[i]);
82          free(B[i]);
83          free(C[i]);
84      }
85      free(A);
86      free(B);
87      free(C);
88
89      return 0;
90  }
```

Output:

```
=thread-selected,id="1"
Matrix Multiplication - 1000x1000
================================

1D Threading (Outer Loop Parallelization):
------------------------------------------
Threads: 2, Time: 3.824080 seconds
Threads: 3, Time: 2.792589 seconds
Threads: 4, Time: 2.138586 seconds
Threads: 5, Time: 1.783573 seconds
Threads: 6, Time: 1.517723 seconds
Threads: 7, Time: 1.306858 seconds
Threads: 8, Time: 1.178423 seconds

2D Threading (Nested Loop Parallelization with collapse):
---------------------------------------------------------
Threads: 2, Time: 3.931052 seconds
Threads: 3, Time: 2.803235 seconds
Threads: 4, Time: 2.126501 seconds
Threads: 5, Time: 1.746498 seconds
Threads: 6, Time: 1.489473 seconds
Threads: 7, Time: 1.303326 seconds
Threads: 8, Time: 1.181371 seconds
The program '/Users/gurnoor/ucs645/Assig1/ques2' has exited with code 0 (0x00000000).
```

Speed-Up Calculation:-

Speed-Up = TN threads/T2 threads

(Here, T2=3.824080 s)

For 1-D Threading Speedup:-

Threads	Time(s)	Speed-Up
2	3.824080	1.00
3	2.792589	1.37
4	2.138586	1.79
5	1.783573	2.14
6	1.517723	2.52
7	1.306858	2.93
8	1.178423	3.25

For 2-D Threading Speedup:-

Threads	Time(s)	Speed-Up
2	3.931052	1.00
3	2.803235	1.40
4	2.126501	1.85
5	1.746498	2.25
6	1.489473	2.64
7	1.303326	3.01
8	1.181371	3.33

Observation:-
For both 1-D and 2-D execution , total time decreases continuously as no of threads increases.2D threading performs slightly better than the 1D threading. At higher thread counts, the speedup continues to increase but the rate of improvement reduces

Conclusion:-
The 2D threading provides better performance than 1D threading for Matrix Multiplication.

Q3.Computation of π using OpenMP

The value of π is calculated using,

π=∫4/(1+ x^2) dx

Number of steps: 100000

Code:

Assig1 > ques3.cpp > main()

```c
#include <stdio.h>
#include <omp.h>

#define NUM_STEPS 100000

int main() {
    double step = 1.0 / (double)NUM_STEPS;
    double pi, sum;
    double start_time, end_time;

    printf("Pi Calculation using Integration\n");
    printf("Number of steps: %d\n\n", NUM_STEPS);

    // Test different thread counts
    for (int threads = 2; threads <= 8; threads++) {
        sum = 0.0;
        omp_set_num_threads(threads);
        start_time = omp_get_wtime();

        #pragma omp parallel for reduction(+:sum)
        for (int i = 0; i < NUM_STEPS; i++) {
            double x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }

        pi = step * sum;
        end_time = omp_get_wtime();

        printf("Threads: %d, Pi: %.10f, Time: %f seconds\n",
               threads, pi, end_time - start_time);
    }

    return 0;
}
```

Output:

```
=thread-selected,id="1"
Pi Calculation using Integration
Number of steps: 100000

Threads: 2, Pi: 3.1415926536, Time: 0.000271 seconds
Threads: 3, Pi: 3.1415926536, Time: 0.000142 seconds
Threads: 4, Pi: 3.1415926536, Time: 0.000148 seconds
Threads: 5, Pi: 3.1415926536, Time: 0.000094 seconds
Threads: 6, Pi: 3.1415926536, Time: 0.000107 seconds
Threads: 7, Pi: 3.1415926536, Time: 0.000086 seconds
Threads: 8, Pi: 3.1415926536, Time: 0.000194 seconds
The program '/Users/gurnoor/ucs645/ques3' has exited with code 0 (0x00000000).
```

Speed-Up Calculation:-

Speed-Up = TN threads/T2 threads

(Here,T2=0.000271 seconds)

Threads	Time(s)	Speed-Up
2	0.000271	1.00
3	0.000142	1.91
4	0.000148	1.83
5	0.000094	2.88
6	0.000107	2.53
7	0.000086	3.15
8	0.000104	2.61

Observation:-
The execution time decreases significantly as the number of threads increases.The best speedup is observed around 7 threads, where the computation achieves a speedup of approximately 3.15.

Conclusion:-
Best performance is achieved using 7 threads.