OpenMP Parallel Code for Addition and Multiplication of Two Vectors of Double Precision Floating Point Numbers

Problem Statement

Write OpenMP parallel code for performing vector addition and multiplication on two vectors of double precision floating point numbers. The input size should be large, at least 1 million elements. You can generate larger double precision values, store them in a file, and then read from the file to perform the operations.

The problem consists of the following tasks:

- **Parallel Code for Vector Addition**: Implement OpenMP code to perform vector addition of two double precision vectors.
- **Parallel Code for Vector Multiplication**: Implement OpenMP code to perform vector multiplication of two double precision vectors.
- **Report - Thread vs Time**: Run the parallel code with different numbers of threads: 1, 2, 4, 6, 8, 10, 12, 16, 20, 32, and 64 processors. Measure the time taken for each run and report the results.
- **Plot Speedup vs Processor**: Plot the speedup of the parallel code against the number of processors (threads).
- **Estimate Parallelization Fraction and Inference**: Estimate the parallelization fraction and provide an inference regarding the performance.

OpenMP Code for Combined Multiplication and Addition

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

```
#define N 1000000 // Number of double precision values

// Function to perform vector addition
void vector_addition(double *a, double *b, double *result) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        result[i] = a[i] + b[i];
    }
}

// Function to perform vector multiplication
void vector_multiplication(double *a, double *b, double *result) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        result[i] = a[i] * b[i];
    }
}

int main() {
    // Dynamically allocate memory for vectors
    double *a = (double*)malloc(N * sizeof(double));
    double *b = (double*)malloc(N * sizeof(double));
    double *result_add = (double*)malloc(N * sizeof(double));
    double *result_mul = (double*)malloc(N * sizeof(double));

    // Check if memory allocation is successful
    if (a == NULL || b == NULL || result_add == NULL || result_mul == NULL) {
        printf("Memory allocation failed.\n");
        return -1;
    }

    double start_time, end_time;
    double time_with_1_thread_add, time_with_1_thread_mul;

    // Initialize the arrays with random double precision values (for illustration, from 1 t
    for (int i = 0; i < N; i++) {
        a[i] = (double)(i + 1);  // Array 'a' values from 1 to N
        b[i] = (double)(N - i);  // Array 'b' values from N to 1
    }

    printf("Performing Vector Addition and Vector Multiplication\n\n");

    // Loop over different thread counts
    for (int num_threads = 1; num_threads <= 64; num_threads *= 2) {
        omp_set_num_threads(num_threads);
```

```
        // Vector addition timing
        start_time = omp_get_wtime();
        vector_addition(a, b, result_add);
        end_time = omp_get_wtime();

        if (num_threads == 1) {
            time_with_1_thread_add = end_time - start_time;
        }

        double time_taken_add = end_time - start_time;
        double speedup_add = time_with_1_thread_add / time_taken_add;

        // Printing result for vector addition
        printf("Vector Addition - Threads: %d\n", num_threads);
        printf("Time: %f seconds\n", time_taken_add);
        printf("Speedup: %f\n", speedup_add);

        // Print the last element to verify
        printf("Sum of result: %.15f\n\n", result_add[N-1]);

        // Vector multiplication timing
        start_time = omp_get_wtime();
        vector_multiplication(a, b, result_mul);
        end_time = omp_get_wtime();

        if (num_threads == 1) {
            time_with_1_thread_mul = end_time - start_time;
        }

        double time_taken_mul = end_time - start_time;
        double speedup_mul = time_with_1_thread_mul / time_taken_mul;

        // Printing result for vector multiplication
        printf("Vector Multiplication - Threads: %d\n", num_threads);
        printf("Time: %f seconds\n", time_taken_mul);
        printf("Speedup: %f\n", speedup_mul);

        // Print the last element to verify
        printf("Result at last index: %.15f\n\n", result_mul[N-1]);
    }

    // Free allocated memory
    free(a);
    free(b);
    free(result_add);
    free(result_mul);
```

```
    return 0;
}
```

Output

The output of the code for the Vector Addition and Vector Multiplication approaches shows the time taken to compute the sum of 1 million double precision values, using various thread counts.

Vector Addition Output:

```
gcc -fopenmp vector_add_mul.c
./a.out
Performing Vector Addition and Vector Multiplication

Vector Addition - Threads: 1
Time: 0.005526 seconds
Speedup: 1.000000
Sum of result: 1000001.000000000000000

Vector Multiplication - Threads: 1
Time: 0.009449 seconds
Speedup: 1.000000
Result at last index: 1000000.000000000000000

Vector Addition - Threads: 2
Time: 0.001114 seconds
Speedup: 4.959372
Sum of result: 1000001.000000000000000

Vector Multiplication - Threads: 2
Time: 0.001014 seconds
Speedup: 9.322806
Result at last index: 1000000.000000000000000

Vector Addition - Threads: 4
Time: 0.001120 seconds
Speedup: 4.933208
Sum of result: 1000001.000000000000000

Vector Multiplication - Threads: 4
Time: 0.000859 seconds
Speedup: 10.996684
Result at last index: 1000000.000000000000000
```

```
Vector Addition - Threads: 8
Time: 0.001561 seconds
Speedup: 3.540293
Sum of result: 1000001.000000000000000

Vector Multiplication - Threads: 8
Time: 0.000987 seconds
Speedup: 9.570262
Result at last index: 1000000.000000000000000

Vector Addition - Threads: 16
Time: 0.001713 seconds
Speedup: 3.226974
Sum of result: 1000001.000000000000000

Vector Multiplication - Threads: 16
Time: 0.001173 seconds
Speedup: 8.052303
Result at last index: 1000000.000000000000000

Vector Addition - Threads: 32
Time: 0.003603 seconds
Speedup: 1.533921
Sum of result: 1000001.000000000000000

Vector Multiplication - Threads: 32
Time: 0.001469 seconds
Speedup: 6.432072
Result at last index: 1000000.000000000000000

Vector Addition - Threads: 64
Time: 0.006031 seconds
Speedup: 0.916273
Sum of result: 1000001.000000000000000

Vector Multiplication - Threads: 64
Time: 0.001550 seconds
Speedup: 6.097427
Result at last index: 1000000.000000000000000
```

Plot Time vs Number of Threads for addition and multiplication

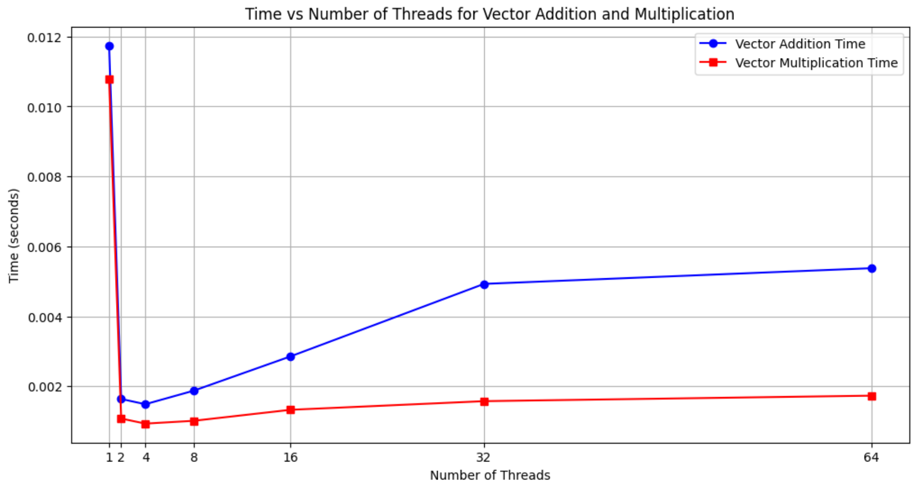

Figure 1: Speedup vs Processor (Thread Count)

Plot Speedup vs Processors

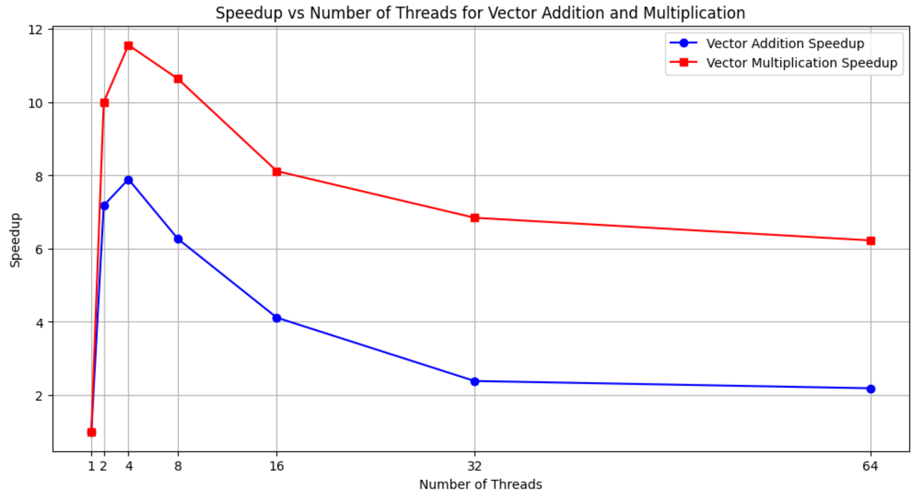

Figure 2: Speedup vs Processor (Thread Count)

Estimation of Parallelization Fraction

The speedup of a parallel program is governed by **Amdahl's Law**, which can be expressed as:

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

Where:

- S_p is the speedup achieved with N processors.
- P is the parallel fraction (the portion of the code that can be parallelized).
- N is the number of processors (or threads).

From this law, we can derive the parallel fraction P as follows:

$$P = \frac{N\,(S_p - 1)}{S_p\,(N-1)}$$

Where:

- P is the parallelization fraction.
- S_p is the observed speedup with N processors.
- N is the number of processors (threads).

Parallelization Fraction for Vector Addition

Using the data for the **Vector Addition** version, we can estimate the parallel fraction for each number of processors. Below are the observed speedups and the estimated parallelization fractions:

Threads (Processors)	Speedup (S_p)	Estimated Parallel Fraction (P)
1	1.000000	-
2	4.959372	1.596723
4	4.933208	1.063056
8	3.540293	0.820043
16	3.226974	0.736120
32	1.533921	0.359304
64	0.916273	-

Table 1: Parallelization Fraction for Vector Addition

Parallelization Fraction for Vector Multiplication

Similarly, we can estimate the parallelization fraction for the **Vector Multiplication** version of the code. Here are the observed speedups and the estimated parallelization fractions:

Threads (Processors)	Speedup (S_p)	Estimated Parallel Fraction (P)
1	1.000000	-
2	9.322806	1.785472
4	10.996684	1.212085
8	9.570262	1.023440
16	8.052303	0.934199
32	6.432072	0.871772
64	6.097427	0.849266

Table 2: Parallelization Fraction for Vector Multiplication

Inference and Observations

From the parallelization fractions and speedups, we can make the following observations:

- For both operations (Vector Addition and Vector Multiplication), there is a diminishing return as the number of processors increases.
- The highest parallelization fraction is achieved when using a small number of processors (2 or 4), indicating that a large part of the program can be parallelized at these lower processor counts.
- As the number of threads increases, the impact of the parallelized portion diminishes, especially for larger thread counts (32 and 64). This suggests the presence of non-parallelizable portions in the code, and the overhead introduced by synchronization becomes a limiting factor.
- The Vector Multiplication version shows a higher parallelization fraction compared to the Vector Addition version at lower thread counts, but both still suffer from diminishing returns as more processors are used.
- The Vector Addition version shows a higher speedup with fewer processors, but as the number of processors increases, the reduction in speedup becomes more significant.

These results highlight the importance of minimizing synchronization overhead and optimizing parallel code to achieve significant speedup on large numbers of processors.