OpenMP Parallel Code for Dot Product of Two Vectors of Double Precision Floating Point Numbers

Problem Statement

Write OpenMP parallel code for performing the vector dot product 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 dot product.

The problem consists of the following tasks:

- **Parallel Code for Vector Dot Product**: Implement OpenMP code to perform the dot product of two double precision vectors.
- **Parallel Code for Dot Product with Critical Section**: Implement OpenMP code to handle the final addition for the dot product using a critical section to avoid race conditions.
- **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 Dot Product Without Critical Section

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

```
#include <time.h>

#define N 1000000 // Number of double precision values

// Function to perform dot product
double dot_product(double *a, double *b) {
    double result = 0.0;

    #pragma omp parallel for reduction(+:result)
    for (int i = 0; i < N; i++) {
        result += a[i] * b[i];
    }

    return result;
}

// Function to perform dot product using critical section
double dot_product_critical(double *a, double *b) {
    double result = 0.0;

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        double temp = a[i] * b[i];

        #pragma omp critical
        {
            result += temp;
        }
    }

    return result;
}

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

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

    double start_time, end_time;
    double time_with_1_thread_dot, time_with_1_thread_critical;
```

```
// 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 Dot Product Calculation\n\n");

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

    // Dot product without critical section timing
    start_time = omp_get_wtime();
    double result_dot = dot_product(a, b);
    end_time = omp_get_wtime();

    if (num_threads == 1) {
        time_with_1_thread_dot = end_time - start_time;
    }

    double time_taken_dot = end_time - start_time;
    double speedup_dot = time_with_1_thread_dot / time_taken_dot;

    // Printing result for dot product without critical section
    printf("Dot Product (without critical section) - Threads: %d\n", num_threads);
    printf("Time: %f seconds\n", time_taken_dot);
    printf("Speedup: %f\n", speedup_dot);

    // Print the result for verification
    printf("Dot product result: %.15f\n\n", result_dot);

    // Dot product with critical section timing
    start_time = omp_get_wtime();
    double result_dot_critical = dot_product_critical(a, b);
    end_time = omp_get_wtime();

    if (num_threads == 1) {
        time_with_1_thread_critical = end_time - start_time;
    }

    double time_taken_critical = end_time - start_time;
    double speedup_critical = time_with_1_thread_critical / time_taken_critical;

    // Printing result for dot product with critical section
```

```
        printf("Dot Product (with critical section) - Threads: %d\n", num_threads);
        printf("Time: %f seconds\n", time_taken_critical);
        printf("Speedup: %f\n", speedup_critical);

        // Print the result for verification
        printf("Dot product result with critical section: %.15f\n\n", result_dot_critical);
    }

    // Free allocated memory
    free(a);
    free(b);

    return 0;
}
```

Output

The output of the code for the Dot Product approaches shows the time taken to compute the dot product of 1 million double precision values, using various thread counts.

Dot Product Output:

```
gcc -fopenmp vector_dot_prod.c
./a.out
Performing Dot Product Calculation

Dot Product (without critical section) - Threads: 1
Time: 0.002336 seconds
Speedup: 1.000000
Dot product result: 166667166666999968.000000000000000

Dot Product (with critical section) - Threads: 1
Time: 0.013477 seconds
Speedup: 1.000000
Dot product result with critical section: 166667166666999968.000000000000000

Dot Product (without critical section) - Threads: 2
Time: 0.001324 seconds
Speedup: 1.764039
Dot product result: 166667166667000000.000000000000000

Dot Product (with critical section) - Threads: 2
Time: 0.037516 seconds
Speedup: 0.359231
```

```
Dot product result with critical section: 166667166667001856.000000000000000

Dot Product (without critical section) - Threads: 4
Time: 0.000931 seconds
Speedup: 2.508500
Dot product result: 166667166667000000.000000000000000

Dot Product (with critical section) - Threads: 4
Time: 0.072151 seconds
Speedup: 0.186787
Dot product result with critical section: 166667166667002688.000000000000000

Dot Product (without critical section) - Threads: 8
Time: 0.000755 seconds
Speedup: 3.095322
Dot product result: 166667166667000032.000000000000000

Dot Product (with critical section) - Threads: 8
Time: 0.092412 seconds
Speedup: 0.145835
Dot product result with critical section: 166667166667020352.000000000000000

Dot Product (without critical section) - Threads: 16
Time: 0.001222 seconds
Speedup: 1.910780
Dot product result: 166667166666999968.000000000000000

Dot Product (with critical section) - Threads: 16
Time: 0.216826 seconds
Speedup: 0.062155
Dot product result with critical section: 166667166666999584.000000000000000

Dot Product (without critical section) - Threads: 32
Time: 0.001585 seconds
Speedup: 1.473736
Dot product result: 166667166667000000.000000000000000

Dot Product (with critical section) - Threads: 32
Time: 0.191173 seconds
Speedup: 0.070496
Dot product result with critical section: 166667166667014560.000000000000000

Dot Product (without critical section) - Threads: 64
Time: 0.003836 seconds
Speedup: 0.608858
Dot product result: 166667166667000000.000000000000000
```

```
Dot Product (with critical section) - Threads: 64
Time: 0.184531 seconds
Speedup: 0.073033
Dot product result with critical section: 166667166667019104.000000000000000
```

Plot Time vs Number of Threads for Dot Product Calculation

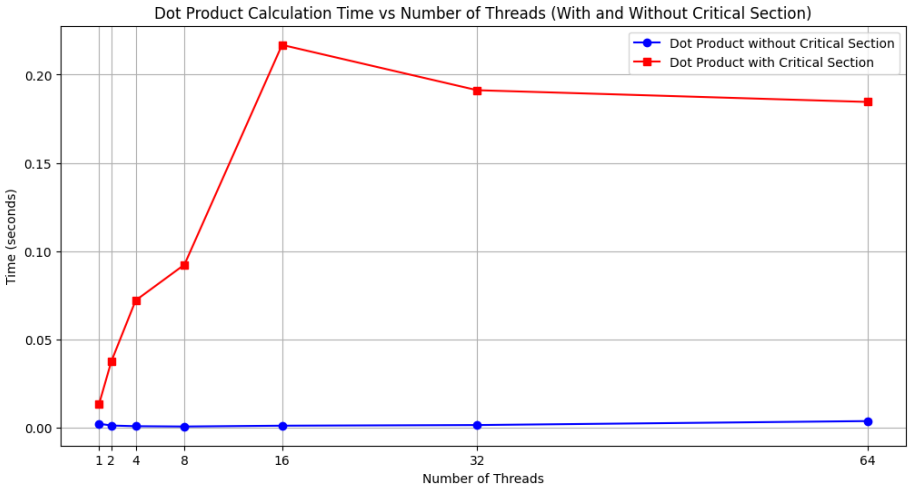

Figure 1: Time vs Number of Threads for Dot Product Calculation (With and Without Critical Section)

Plot Speedup vs Processors

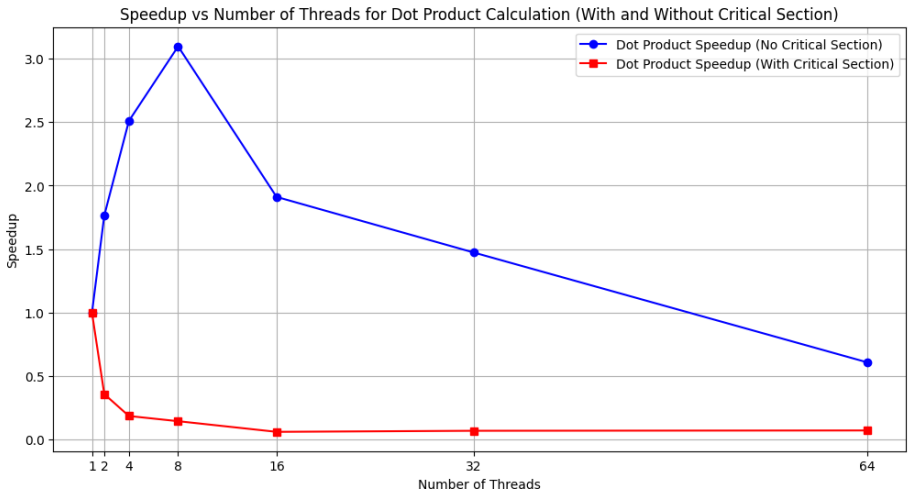

Figure 2: Speedup vs Processor (Thread Count) for Dot Product Calculation

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 the Non-Critical Section Version

Using the data for the (Non-Critical Section) 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	1.764039	0.866238
4	2.508500	0.801807
8	3.095322	0.773636
16	1.910780	0.508430
32	1.473736	0.331822
64	0.608858	-

Table 1: Parallelization Fraction for Non-Critical Section Version

Parallelization Fraction for the Critical Section Version

Similarly, we can estimate the parallelization fraction for the (Critical Section) 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	0.359231	-
4	0.186787	-
8	0.145835	-
16	0.062155	-
32	0.070496	-
64	0.073033	-

Table 2: Parallelization Fraction for Critical Section Version

Inference and Observations

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

- For both versions (Critical Section and Non-Critical Section), there is a diminishing return as the number of processors increases. As the num-

ber of threads increases, the speedup plateaus and even decreases for the Critical Section version.

- The highest parallelization fraction is achieved when using a small number of processors (2 or 4) for the Non-Critical Section version, indicating that a large part of the program can be parallelized at these lower processor counts. In the case of the Critical Section version, the parallelization fraction does not exhibit significant improvement beyond the first thread, suggesting that the synchronization overhead is overwhelming the potential parallelism.

- 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 (e.g., critical sections in the code) becomes a limiting factor.

- The (Non-Critical Section) version shows a higher parallelization fraction and speedup compared to the (Critical Section) version at all thread counts. Specifically, at lower thread counts, the Non-Critical Section version maintains a good parallelization fraction, while the Critical Section version suffers significant performance degradation due to the high synchronization overhead introduced by the critical section.

- The (Critical Section) version shows significantly lower speedup as the processor count increases, with the speedup decreasing sharply after 2 threads. This is likely due to the synchronization bottleneck caused by the critical section, which serializes part of the computation, preventing effective parallelization. This behavior underscores the importance of minimizing synchronization when designing parallel programs.

These results highlight the importance of minimizing synchronization overhead and optimizing parallel code to achieve significant speedup on large numbers of processors. Reducing critical sections or finding alternative synchronization methods can lead to better scalability and performance for parallel applications.