OpenMP Parallel Code for Sum of N - Double Precision Floating Point Numbers

Problem Statement

Write OpenMP parallel code for summing N double precision floating point numbers. The input size should be large, at least 1 million numbers. You can generate and store larger double precision values in a file and then read from it to perform the summation.

The problem consists of the following tasks:

- **Parallel Code Using Reduction Construct**: Implement OpenMP code using the reduction construct for parallel summation.
- **Parallel Code Using Critical Section**: Implement OpenMP code using the critical section to perform parallel summation.
- **Report - Thread vs Time**: Run the parallel code with different numbers of threads: 1, 2, 4, 8, 16, 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 Using Critical Section

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

#define N 1000000 // Number of double precision values

int main() {
```

```
double *arr = (double*)malloc(N * sizeof(double));
double sum = 0.0;
double start_time, end_time;

// Initialize the array with large double precision values
for (int i = 0; i < N; i++) {
    // You can use random values or simple values for testing
    arr[i] = i + 1;  // Array values from 1 to N (just an example)
}

// Variable to store the time with 1 thread for calculating speedup
double time_with_1_thread;

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

    start_time = omp_get_wtime();

    sum = 0.0;

    // Parallel region with critical section
    #pragma omp parallel
    {
        double local_sum = 0.0;

        #pragma omp for
        for (int i = 0; i < N; i++) {
            local_sum += arr[i];
        }

        #pragma omp critical
        {
            sum += local_sum;
        }
    }

    end_time = omp_get_wtime();

    // Capture the time with 1 thread for calculating speedup
    if (num_threads == 1) {
        time_with_1_thread = end_time - start_time;
    }

    double time_taken = end_time - start_time;
    double speedup = time_with_1_thread / time_taken;
```

```
        printf("Threads: %d, Time: %f seconds, Speedup: %f\n",
            num_threads, time_taken, speedup);
        printf("Sum: %.15f\n", sum);
    }

    free(arr);
    return 0;
}
```

OpenMP Code Using Reduction Construct

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

#define N 1000000 // Number of double precision values

int main() {
    double *arr = (double*)malloc(N * sizeof(double));
    double sum = 0.0;
    double start_time, end_time;

    // Initialize the array with large double precision values
    for (int i = 0; i < N; i++) {
        // You can either use random values or simple values
        // For demonstration, I'll use values from 1 to N
        arr[i] = i + 1;  // Array values from 1 to N (just an example)
    }

    // Variable to store the time with 1 thread for calculating speedup
    double time_with_1_thread;

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

        start_time = omp_get_wtime();

        sum = 0.0;

        // Parallel region with reduction construct
        #pragma omp parallel for reduction(+:sum)
        for (int i = 0; i < N; i++) {
```

```
            sum += arr[i];
        }

        end_time = omp_get_wtime();

        // Capture the time with 1 thread for calculating speedup
        if (num_threads == 1) {
            time_with_1_thread = end_time - start_time;
        }

        double time_taken = end_time - start_time;
        double speedup = time_with_1_thread / time_taken;

        printf("Threads: %d, Time: %f seconds, Speedup: %f\n",
            num_threads, time_taken, speedup);
        printf("Sum: %.15f\n", sum);
    }

    free(arr);
    return 0;
}
```

Output

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

Critical Section Output:

```
gcc -fopenmp sumOfN_critical_test.c
./a.out
Threads: 1, Time: 0.002548 seconds, Speedup: 1.000000
Sum: 500000500000.000000000000000
Threads: 2, Time: 0.001680 seconds, Speedup: 1.516887
Sum: 500000500000.000000000000000
Threads: 4, Time: 0.000947 seconds, Speedup: 2.689885
Sum: 500000500000.000000000000000
Threads: 8, Time: 0.000925 seconds, Speedup: 2.755814
Sum: 500000500000.000000000000000
Threads: 16, Time: 0.001694 seconds, Speedup: 1.503913
Sum: 500000500000.000000000000000
Threads: 32, Time: 0.002382 seconds, Speedup: 1.069573
Sum: 500000500000.000000000000000
Threads: 64, Time: 0.004427 seconds, Speedup: 0.575580
```

```
Sum: 500000500000.000000000000000
```

Reduction Output:

```
gcc -fopenmp sumOfN_reduction_test.c
./a.out
Threads: 1, Time: 0.002835 seconds, Speedup: 1.000000
Sum: 500000500000.000000000000000
Threads: 2, Time: 0.001514 seconds, Speedup: 1.872457
Sum: 500000500000.000000000000000
Threads: 4, Time: 0.000994 seconds, Speedup: 2.851106
Sum: 500000500000.000000000000000
Threads: 8, Time: 0.000789 seconds, Speedup: 3.590671
Sum: 500000500000.000000000000000
Threads: 16, Time: 0.001165 seconds, Speedup: 2.433018
Sum: 500000500000.000000000000000
Threads: 32, Time: 0.001630 seconds, Speedup: 1.739345
Sum: 500000500000.000000000000000
Threads: 64, Time: 0.003510 seconds, Speedup: 0.807626
Sum: 500000500000.000000000000000
```

Plot Time vs Number of Threads for Reduction and Critical Section

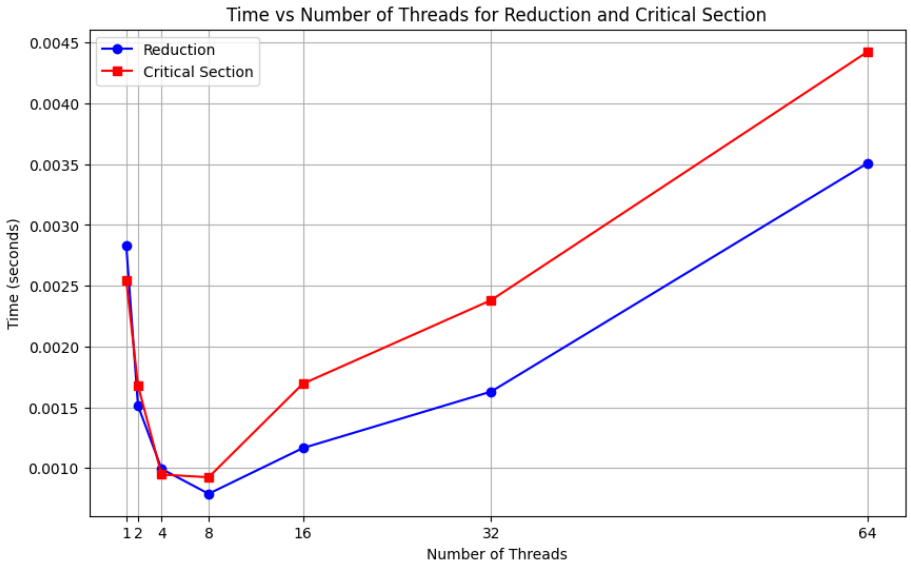

Figure 1: Speedup vs Processor (Thread Count)

Plot Speedup vs Processors

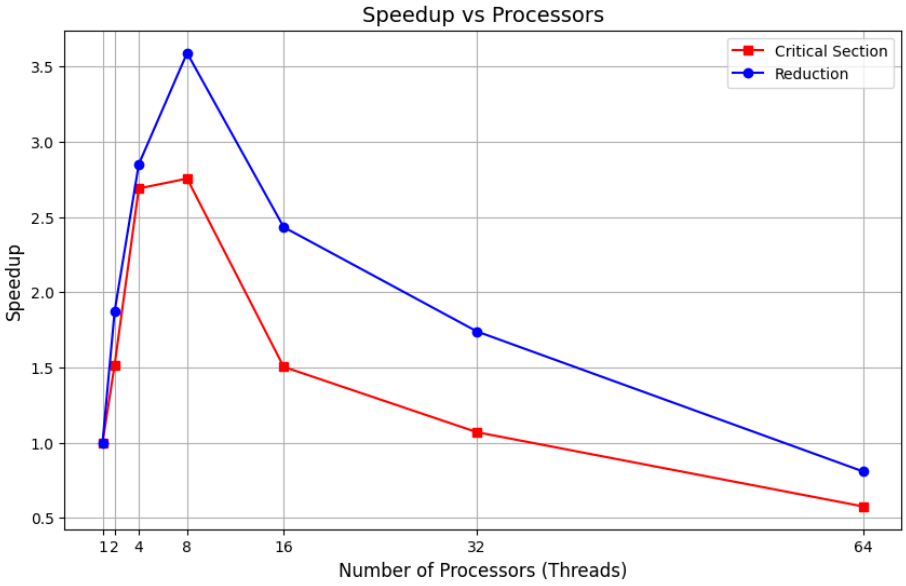

Figure 2: Speedup vs Processor (Thread Count)

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 the Critical Section Version

Using the data for the 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	1.516887	0.6815
4	2.689885	0.8376
8	2.755814	0.7281
16	1.503913	0.3574
32	1.069573	0.0671
64	0.575580	-

Table 1: Parallelization Fraction for Critical Section Version

Parallelization Fraction for the Reduction Version

Similarly, we can estimate the parallelization fraction for the Reduction 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	1.872457	0.9318
4	2.851106	0.8656
8	3.590671	0.8245
16	2.433018	0.6282
32	1.739345	0.4387
64	0.807626	-

Table 2: Parallelization Fraction for Reduction Version

Inference and Observations

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

- For both versions (Critical Section and Reduction), 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 (e.g., critical sections in the code) becomes a limiting factor.
- The reduction version shows a higher parallelization fraction compared to the critical section version at lower thread counts, but it still suffers from diminishing returns as more processors are used.
- The critical section version shows lower speedup as the processor count increases, likely due to the high synchronization overhead incurred when accessing shared data.

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