Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Zitian Wang Discussed with: None

Solution for Project 2

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP *(20 Points)*

1.1. Dot Product

1.1.1. Two parallel versions of the dot product

The *reduction* clause automatically handles thread-local partial results and merges them at the end of the parallel section to produce the final outcome

```
1 void dot_product_reduction(double* a, double* b, long double& result){
2 #pragma omp parallel for reduction(+:alpha)
3 for (int i = 0; i < N; i++) {
4     alpha += a[i] * b[i];
5     }
6 }
```

Listing 1: Reduction pragma

The *critical* instruction ensures that updating the shared variable alpha is performed serially, specifically during the summation phase, while the parallel section is responsible for calculating individual results, which involves the multiplication step.

```
void dot_product_critical(double* a, double* b, long double& result,
    int N) {
    long double local_result = 0.0;

    #pragma omp parallel private(local_result)
    {
        local_result = 0.0;

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

        #pragma omp critical
        {
            result += local_result;
        }
    }
}
```

Listing 2: Critical pragma

1.1.2. Strong scaling analysis

To batch execute experiments with varying numbers of threads and different vector lengths, an execution script needs to be designed.

```
#!/bin/bash
VECTOR_LENGTHS=(100000 1000000 10000000 100000000 1000000000)
THREAD_COUNTS=(1 2 4 8 16 20)
g++ -O3 -fopenmp dotProduct.cpp -o dotProduct
if [ $? -ne 0 ]; then
    echo "Compilation failed!"
    exit 1
fi
for N in "${VECTOR_LENGTHS[@]}"; do

    FILENAME="experiment_results_${N}.csv"
    echo "Thread_Count,Method,Execution_Time" > $FILENAME
    for T in "${THREAD_COUNTS[@]}"; do
        export OMP_NUM_THREADS=$T
        echo "Running with Vector Length = $N and Threads = $T"

        ./dotProduct $N $T

        if [ $? -ne 0 ]; then
            echo "Execution failed for Vector Length = $N and Threads =
                $T!"
            exit 1
        fi
    done
done
echo "All experiments completed successfully."
```

Listing 3: Bash Script for Dot Product Experiment

After running the script, three key metrics, Serial Time, Reduction Time, and Critical Time will be obtained.

Table 1: Execution Times (in seconds) for Different Vector Lengths and Thread Counts

Vector Length	**Threads**	**Serial Time**	**Reduction Time**	**Critical Time**
100,000	1	0.0471	0.0118	0.0118
	2	0.0471	0.0061	0.0061
	4	0.0472	0.0034	0.0034
	8	0.0471	0.0025	0.0024
	16	0.0472	0.0028	0.0027
	20	0.0471	0.0040	0.0030
1,000,000	1	0.4716	0.1171	0.1175
	2	0.4710	0.0589	0.0587
	4	0.4713	0.0300	0.0297
	8	0.4714	0.0159	0.0156
	16	0.4710	0.0105	0.0093
	20	0.4710	0.0090	0.0084
10,000,000	1	4.7752	1.3626	1.3630
	2	4.7728	0.6993	0.6985
	4	4.7756	0.6442	0.6440
	8	4.7760	0.4832	0.5681
	16	4.7826	0.5387	0.5407
	20	4.7827	0.5254	0.6447
100,000,000	1	47.8156	13.6160	13.5554
	2	47.8162	6.9151	6.9144
	4	47.8293	6.2994	6.2982
	8	47.7153	5.6159	5.6159
	16	47.7149	4.4565	4.4643
	20	47.7136	5.2087	5.2487
1,000,000,000	1	476.989	133.917	133.921
	2	477.224	69.3902	69.4007
	4	477.007	63.7676	63.7917
	8	476.868	56.4091	56.4636
	16	476.977	45.0795	44.9397
	20	477.347	49.6325	49.7888

Based on the table, we can draw the following conclusions: the execution time of a serial program is linearly proportional to the vector length. The `critical` directive executes faster than the `reduction` clause, with this advantage becoming more pronounced as the vector length increases. The number of threads does not significantly impact the execution time of a serial program. While theoretically, increasing the number of threads should proportionally reduce the execution time for both the `critical` and `reduction` methods, the actual results show that the execution time for these two methods does not decrease by half when the number of threads increases from 8 to 16. Additionally, in some cases, the execution time increases when the number of threads reaches 20. This could be related to the fact that the CPU executing the program has only 20 cores. The following graph is a figurative presentation of the above data, using logarithmic coordinates for ease of presentation.

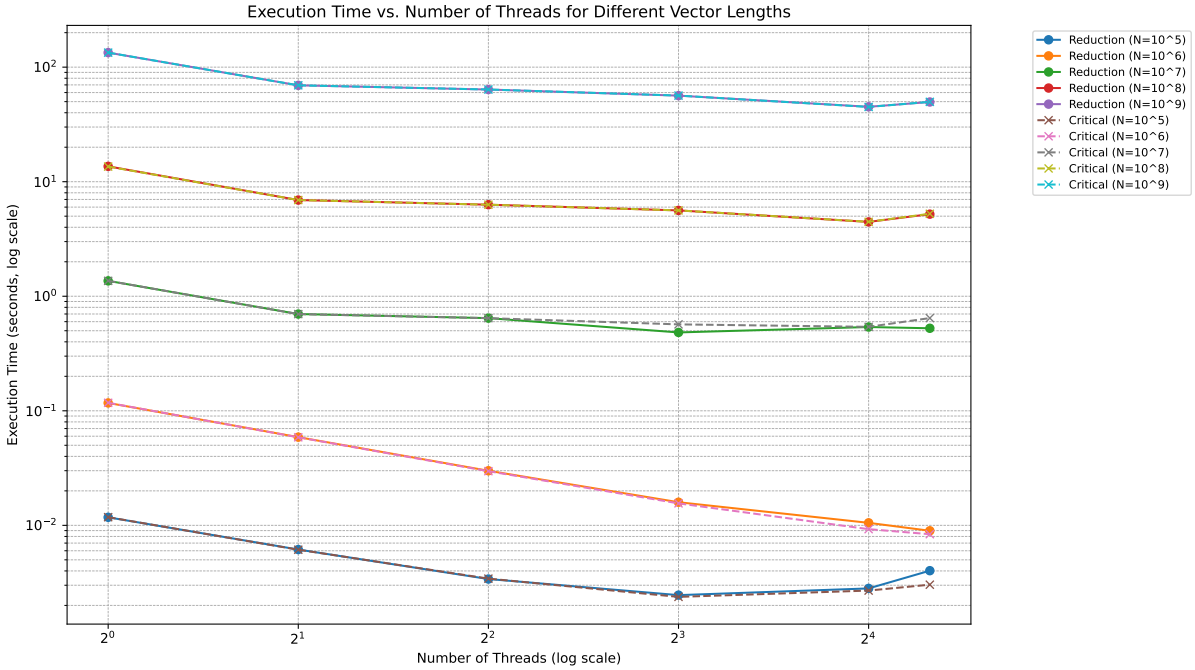

Figure 1: Execution time vs threads

1.1.3. Analysis on the parallel efficiency

In academic English, the definition of *Parallel Efficiency* (Efficiency) can be expressed as follows:
Parallel Efficiency (Efficiency) is defined as the ratio of speedup to the number of threads:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

Where *Speedup* is the ratio of serial execution time to parallel execution time:

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

Ideally, the closer the efficiency value is to 1, the better the parallelization performance[1].

```
for vector_length in vector_lengths:
    serial_time = serial_times[vector_length]
    reduction_speedups = [serial_time / t for t in reduction_times[
        vector_length]]
    reduction_efficiency = [reduction_speedups[i] / threads[i] for i in
         range(len(threads))]
    plt.plot(threads, reduction_efficiency, label=f'Reduction (N=10^{
        int(np.log10(vector_length))})', marker='o')

for vector_length in vector_lengths:
    serial_time = serial_times[vector_length]
    critical_speedups = [serial_time / t for t in critical_times[
        vector_length]]
    critical_efficiency = [critical_speedups[i] / threads[i] for i in
        range(len(threads))]
    plt.plot(threads, critical_efficiency, label=f'Critical (N=10^{int(
        np.log10(vector_length))})', linestyle='--', marker='x')
```

Listing 4: Plotting Parallel Efficiency for Reduction and Critical Methods

The above code is used to calculate Efficiency when drawing a graph.

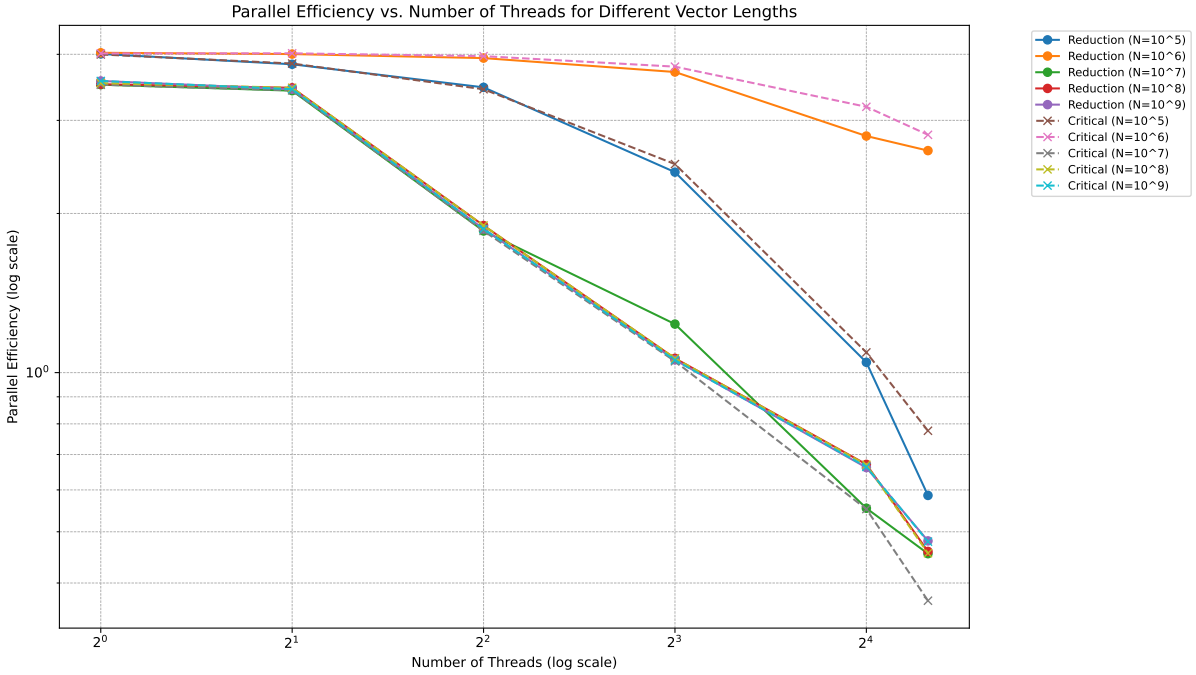

Figure 2: Parallel efficiency time vs threads

The graph with the logarithm of efficiency as the vertical axis more clearly illustrates the observations described in the previous question. By plotting the logarithm of efficiency, the deviations from ideal parallel performance become more apparent, especially as the number of threads increases beyond the optimal range, highlighting the diminishing returns in efficiency.

Initially, I used `#pragma omp critical` within each loop iteration, which resulted in a very time-consuming process. Since then, I have modified the approach. Each thread now computes a partial dot product in the local variable `local_result` and then accumulates it into the final result using `#pragma omp critical`. By doing this, the critical section is executed only once after each thread completes its computation, significantly reducing the number of lock contentions and, consequently, the synchronization overhead. Therefore, in this exercise, the *reduction* method demonstrates lower parallel efficiency compared to the *critical* method.

Regarding the relationship between the number of threads, parallel efficiency, and workload size, both *reduction* and *critical* methods show significant efficiency improvements when the thread count is low. However, as the thread count increases to 16 or 20, the overhead from synchronization and scheduling begins to reduce efficiency. Larger vector lengths lead to higher parallel efficiency, particularly in multi-threaded scenarios, as the overhead between threads becomes relatively smaller, and the proportion of computation increases.

The benefits of multi-threading are evident when the vector length is large (e.g., $N \geq 10^6$), where multi-threading achieves significant speedup. Conversely, for smaller vector lengths, using more threads yields diminishing returns or even inefficiencies due to synchronization overhead, making multi-threading less advisable for small-scale problems.

1.2. Approximating π

1.2.1. Serial and parallel versions method

```
/* TODO Serial Version of Computing Pi*/
time = omp_get_wtime();
sum = 0.0;
for (long int i = 0; i < N; i++) {
    double x = (i + 0.5) * dx;
    sum += 4.0 / (1.0 + x * x);
}
```

Listing 5: Serial Version of Computing Pi

```
/* TODO Parallel Version of Computing Pi */
time = omp_get_wtime();
sum = 0.;
#pragma omp parallel for reduction(+:sum)
for (long int i = 0; i < N; i++) {
    double x = (i + 0.5) * dx;
    sum += 4.0 / (1.0 + x * x);
}
```

Listing 6: Parallel Version of Computing Pi

To parallelize the code, the `#pragma omp parallel for reduction(+:sum)` directive is used because it automatically handles the local accumulation for each thread and then combines the results from all threads at the end. This approach significantly reduces the overhead of manually managing locks.

1.2.2. Compare with parallel implementation and the serial version

```
#!/bin/bash
g++ -O3 -fopenmp pi.cpp -o pi
if [ $? -ne 0 ]; then
    echo "Compilation failed!"
    exit 1
fi
THREAD_COUNTS=(1 2 4 8)
for T in "${THREAD_COUNTS[@]}"; do
    export OMP_NUM_THREADS=$T
    echo "Running with $T thread(s):"
    ./pi
    if [ $? -ne 0 ]; then
    echo "Execution failed for $T thread(s)!"
    exit 1
    fi
    echo ""
done
echo "All experiments completed successfully."
```

Listing 7: Bash Script for Running Pi Program with OpenMP Support

The plot generated from the experimental data reveals a diagonal line trending upwards, indicating that the speedup increases nearly linearly with the number of threads. This represents an ideal scenario, where parallel efficiency approaches 1. However, in many real-world situations (as discussed in the previous problem), achieving linear speedup becomes increasingly difficult as the number of threads grows. This is because managing threads introduces overheads associated with synchronization and communication, which tend to increase as the number of threads rises.

Table 2: Serial and Parallel Execution Times for Approximating π with Different Thread Counts

Threads	Serial Time (s)	Parallel Time (s)	Speedup
1	53.9293	53.9290	1.00
2	53.9291	27.0123	2.00
4	53.9331	13.5054	3.99
8	53.9288	6.7539	7.98

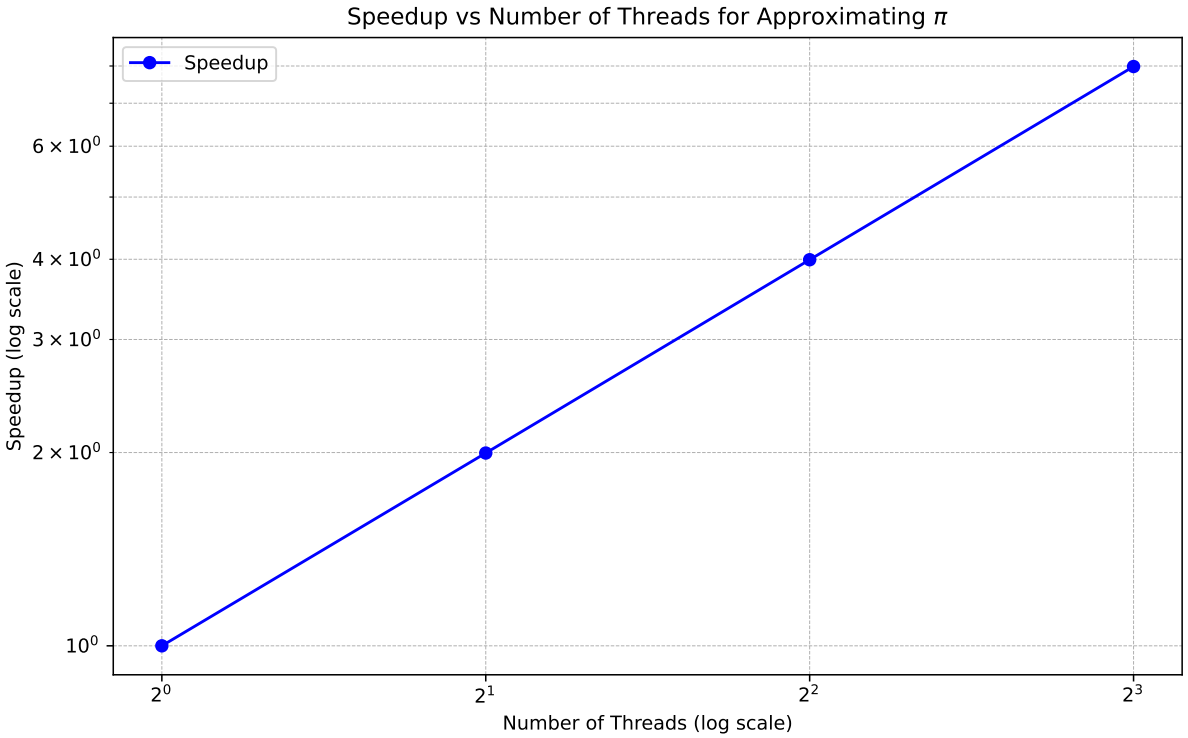

Figure 3: Pi speedup

2. The Mandelbrot set using OpenMP *(20 Points)*

2.1. Computation kernel of the Mandelbrot set and number of iterations

```
while ((x2 + y2 <= 4.0) && (n < MAX_ITERS)) {
    y = 2 * x * y + cy;
    x = x2 - y2 + cx;
    x2 = x * x;
    y2 = y * y;
    n++;
    nTotalIterationsCount++;
}
```

Listing 8: Mandelbrot Iteration Loop

2.2. Parallelize the Mandelbrot code

```
#pragma omp parallel for private(x, y, x2, y2, cx, cy) reduction(+:
    nTotalIterationsCount)
for (long j = 0; j < IMAGE_HEIGHT; j++) {
```

```
    cy = MIN_Y + j * fDeltaY;
    for (long i = 0; i < IMAGE_WIDTH; i++) {
        cx = MIN_X + i * fDeltaX;
        x = cx;
        y = cy;
        x2 = x * x;
        y2 = y * y;
        int n = 0;
        while ((x2 + y2 <= 4.0) && (n < MAX_ITERS)) {
            y = 2 * x * y + cy;
            x = x2 - y2 + cx;
            x2 = x * x;
            y2 = y * y;
            n++;
            nTotalIterationsCount++;
        }

        int c = ((long)n * 255) / MAX_ITERS;
        #pragma omp critical
        png_plot(pPng, i, j, c, c, c);
    }
}
```

Listing 9: Parallel Mandelbrot Calculation with OpenMP

Table 3: Benchmark Results for Serial and Parallel Mandelbrot Set Computation

Threads	**Serial Time (s)**	**Parallel Time (s)**	**Speedup**	**Iterations/Second**	**MFlop/s**
1	156.967	314.236	0.500	2.06×10^8	1648.12
2	156.970	235.837	0.666	2.74×10^8	2194.17
4	156.962	233.083	0.673	2.78×10^8	2220.10
8	156.968	208.487	0.753	3.10×10^8	2482.00
16	156.963	186.563	0.841	3.47×10^8	2773.68
20	156.979	181.296	0.866	3.57×10^8	2854.26

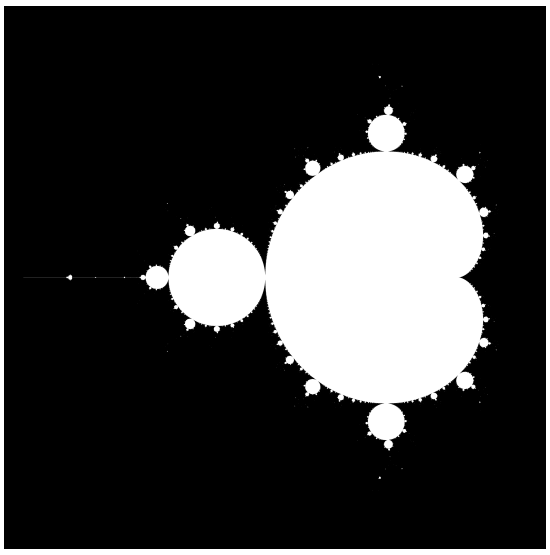

Figure 4: The Mandelbrot set

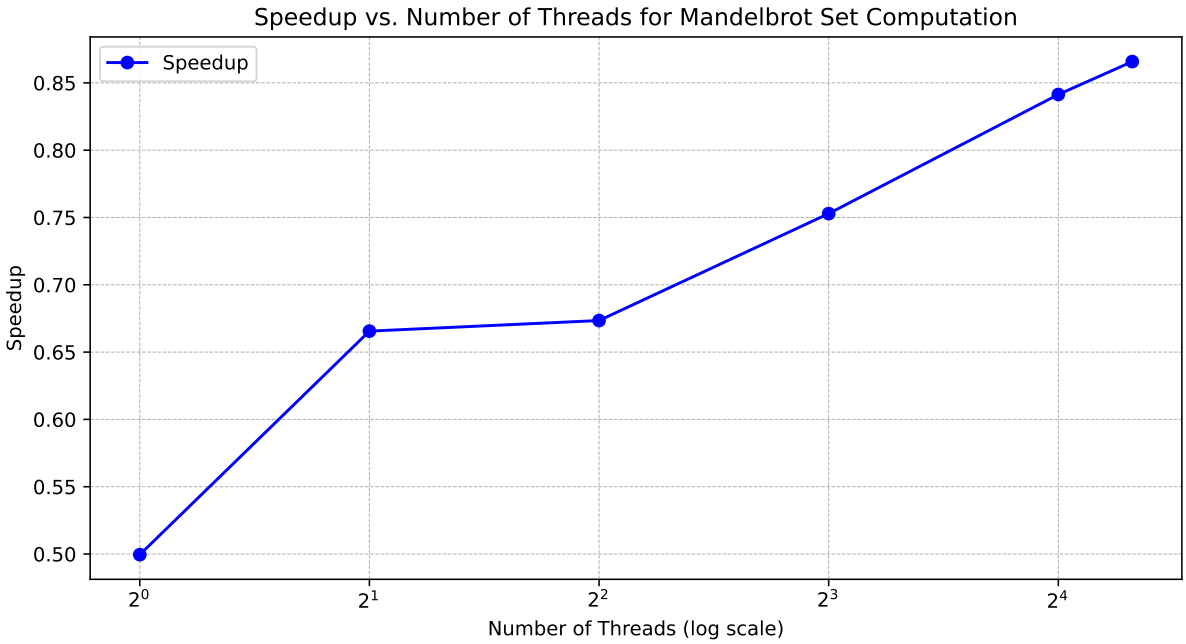

Figure 5: Speedup

The speedup shows little improvement when the number of threads increases from 2 to 4, likely due to the computational nature of the Mandelbrot set, which varies in complexity across different regions of the complex plane. Some regions require more iterations to determine whether they belong to the Mandelbrot set, while others require fewer iterations. This variation can cause an imbalance in workload distribution among the threads, with some handling more complex regions and others dealing with simpler regions. As a result, some threads may finish earlier, leading to increased waiting time for the remaining threads, thereby affecting overall parallel efficiency. However, as the number of threads increases to 8, 16, or even 20, the speedup begins to grow more linearly. This is likely because each thread handles a smaller workload, resulting in more efficient task processing.

3. Bug hunt *(15 Points)*

bug1 The `tid` is used to print the thread number, but due to the nesting of the {} block and the inner `for` loop, the `tid` may not be unique within the loop, potentially causing confusion regarding thread numbers. To resolve this, move the `tid` fetching operation inside the `for` loop body so that the current thread's number is fetched again each time the loop iterates, ensuring the uniqueness of the thread identifier.

bug2 The variable `total` is declared and used in the parallel region, but it is shared among threads, causing data contention as each thread attempts to modify it. This issue can be resolved by utilizing OpenMP's `reduction(+:total)` mechanism, which converts `total` into a local cumulative variable for each thread. After all threads complete their calculations, the `reduction` directive combines the individual `total` values, ensuring accurate final results without data contention.

bug3 The `#pragma omp barrier` cannot be used within a `section` structure inside a parallel region because it can lead to a deadlock. This occurs when some threads are still executing sections while others reach the barrier, causing all threads to wait indefinitely for one another,

halting execution. To prevent such mis-synchronization and avoid potential deadlocks, remove the `#pragma omp barrier` from the `print_results()` function.

bug4 For an array `a` of size $N \times N$, where $N = 1048$, each thread needs to allocate a large chunk of memory for its own copy. This can exceed the available stack space, leading to a segmentation fault. To resolve this issue, the stack size for all additional threads can be adjusted via the environment variable `OMP_STACKSIZE`, which allows for controlling and increasing the stack size limit to prevent such errors.

bug5 In the code, the two `#pragma omp sections` are executed by different threads, both of which involve two locks: `locka` and `lockb`. If the threads execute simultaneously, with one thread holding `locka` and the other holding `lockb`, they can end up waiting for each other to release their respective locks, resulting in a deadlock. To prevent deadlocks, ensure that both threads acquire the locks in the same order. This way, both threads will follow a consistent locking sequence, avoiding situations where one thread waits indefinitely for the other.

4. Parallel histogram calculation using OpenMP *(15 Points)*

```
time_start = walltime();
long local_histograms[BINS * omp_get_max_threads()] = {0};

#pragma omp parallel shared(local_histograms)
{
    int tid = omp_get_thread_num();
    long* local_hist = &local_histograms[tid * BINS];

    #pragma omp for
    for (long i = 0; i < VEC_SIZE; ++i) {
        dist[vec[i]]++;
    }
}
```

Listing 10: Parallel Histogram Calculation with OpenMP

Table 4: Strong Scaling Results for Histogram Calculation

Number of Threads	Parallel Time (s)	Serial Time (s)	Speedup
1	1.85422	1.85422	1.00
2	1.58072	1.85422	1.17
4	1.01373	1.85422	1.83
8	0.840241	1.85422	2.21
16	0.831195	1.85422	2.23
32	0.81123	1.85422	2.29
64	0.80345	1.85422	2.31
128	0.79932	1.85422	2.32

In line with the previous conclusion, the marginal benefit of performance improvement decreases significantly once the number of threads exceeds 8. Beyond this point, additional threads provide diminishing returns in terms of speedup, likely due to increased overhead from thread management, synchronization, and reduced workload per thread. In particular, the performance gain from increasing the number of threads from 1 to 2 is not significant, likely because the synchronization overhead between threads constitutes a larger percentage of the total runtime,

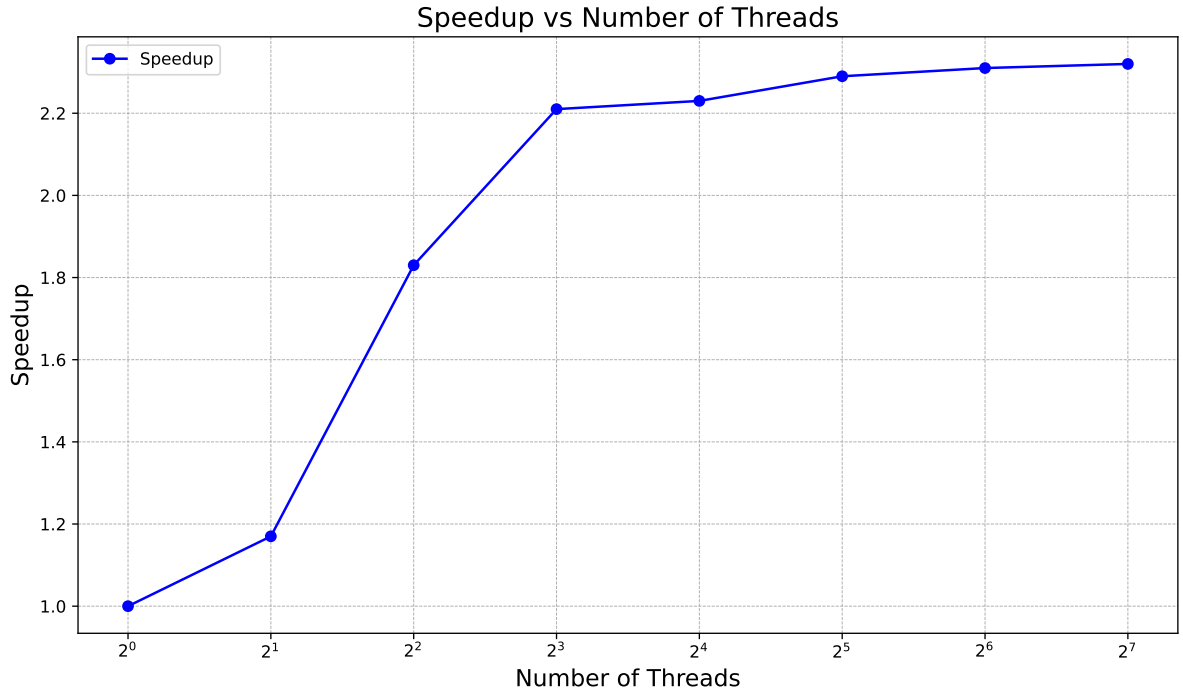

Figure 6: Histogram speedup

especially when updating the histogram. It is only when the number of threads reaches a certain threshold that the computational benefits begin to outweigh the synchronization costs, leading to more noticeable performance improvements.

5. Parallel loop dependencies with OpenMP *(15 Points)*

```
#pragma omp parallel for firstprivate(Sn) lastprivate(Sn)
for (n = 0; n <= N; ++n) {
    opt[n] = Sn;
    Sn *= up;
}
```

Listing 11: loop parallelization

`firstprivate(Sn)` ensures that each thread starts with the initial value of `Sn` from the serial portion when entering the parallel region, guaranteeing consistent initial states for all threads, while `lastprivate(Sn)` ensures that the value of `Sn` from the last iteration is saved back to the serial portion after the parallel region.

```
#pragma omp parallel for reduction(+:temp)
for (n = 0; n <= N; ++n) {
    temp += opt[n] * opt[n];
}
```

Listing 12: reduction

Since all threads need to accumulate the value of `temp`, the `reduction(+:temp)` clause is used, which allows for safe accumulation in a parallel environment and ultimately combines the partial sums from each thread.

Table 5: Comparison of Sequential and Parallel Execution for Recursion Problem

Metric	**Sequential**	**Parallel (OMP)**
RunTime (s)	6.736231	0.410847
Final Result S_n	485165097.62511122	2.7182818255291776
Result $\|\|opt\|\|_2^2$	5.8846×10^{15}	3.194528

References

[1] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.