Università della Svizzera italiana | Institute of Computing CI

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

Student: Srimal Fonseka Discussed with: FULL NAME

Solution for Project 2

This project will introduce you to parallel programming using OpenMP.

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

1.1. Dot Product

1 Parallel Implementations

Two parallel implementations of the dot product were developed using OpenMP.

- **Reduction Clause:** This method uses `#pragma omp parallel for reduction(+:alpha)` to safely accumulate partial results from each thread. It is efficient and minimizes synchronization overhead.
- **Critical Directive:** This method uses `#pragma omp critical` to serialize access to the shared accumulation variable, ensuring correctness but introducing significant performance penalties due to thread contention.

Correctness of both implementations was verified against the serial baseline.

2 Strong Scaling Analysis

Strong scaling tests were conducted on the Rosa cluster for both parallel versions using thread counts $t = 1, 2, 4, 8, 16, 20$ and vector sizes $N = 10^5, 10^6, 10^7, 10^8, 10^9$.

Reduction Method

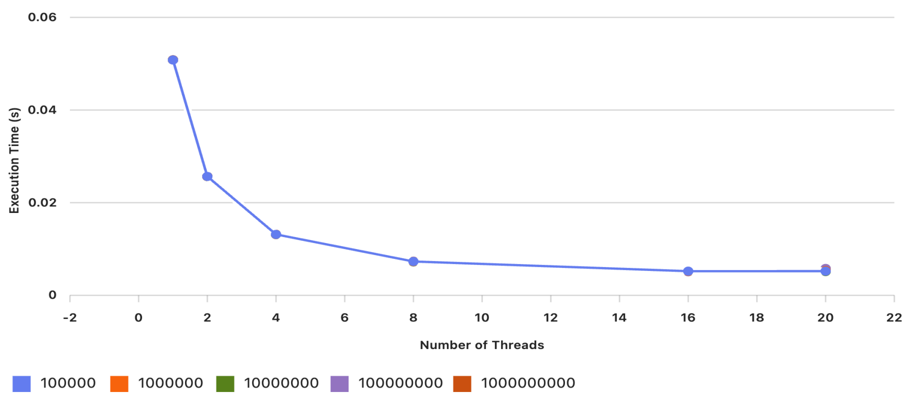

Figure 1: Strong scaling performance using the reduction method.

The reduction method demonstrates consistent improvement in execution time as the number of threads increases. For larger vector sizes, scaling is nearly ideal up to 8 threads, with diminishing returns beyond that point due to hardware limitations and overhead.

Critical Method

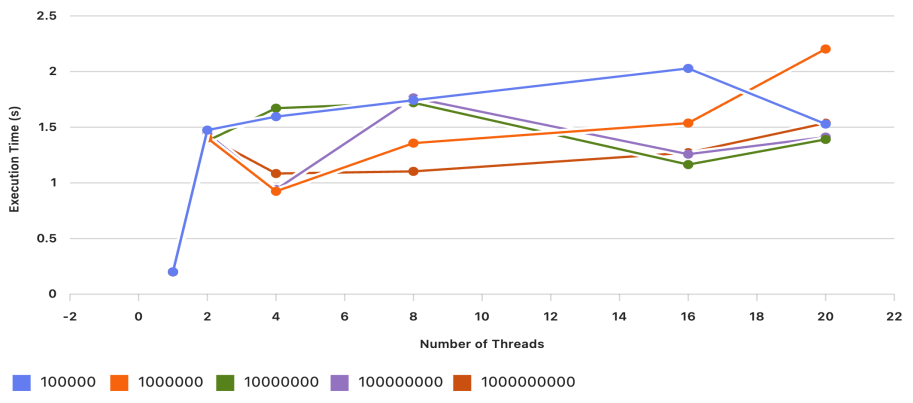

Figure 2: Strong scaling performance using the critical method.

The critical method exhibits poor scalability. Execution time increases with additional threads due to contention at the critical section, particularly for small vector sizes. For larger N, performance stabilizes but remains significantly inferior to the reduction method.

3 Parallel Efficiency Analysis

Parallel efficiency was computed as:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}, \quad \text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

Reduction Method

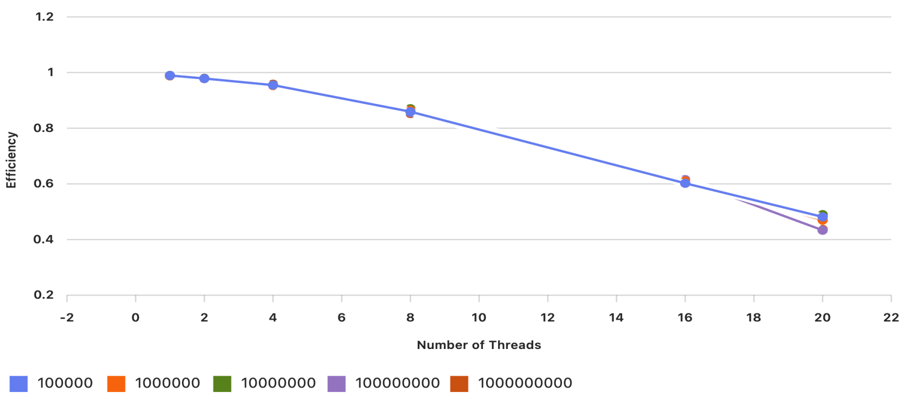

Figure 3: Parallel efficiency using the reduction method.

Efficiency remains high for small thread counts and large vector sizes. For $N \geq 10^7$, efficiency exceeds 60% even at 20 threads, indicating effective utilization of parallel resources.

Critical Method

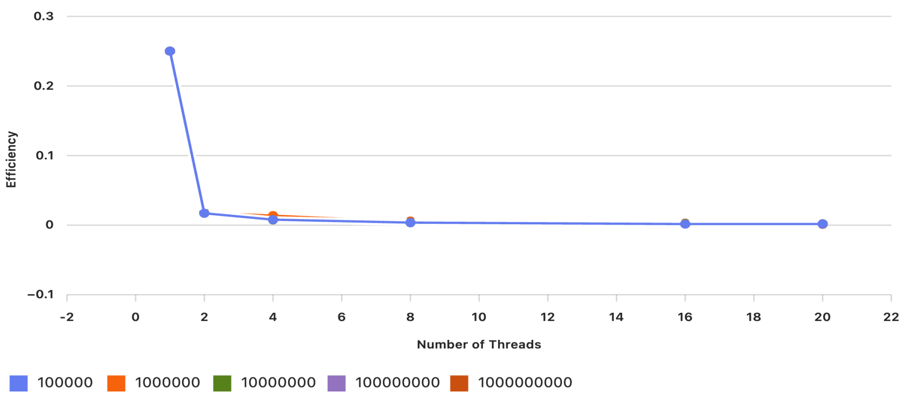

Figure 4: Parallel efficiency using the critical method.

Efficiency declines sharply with increasing thread counts, particularly for small N. Even for $N = 10^9$, efficiency remains low due to synchronization overhead. This behavior is consistent across all vector sizes, confirming that the synchronization cost outweighs the benefits of parallelism.

Discussion

OpenMP Overhead

The critical directive introduces significant overhead due to serialized access to the shared variable. In contrast, the reduction clause avoids this issue by efficiently combining partial results.

Thread Count vs. Workload

For small vector sizes ($N \leq 10^6$), parallelization offers limited benefit, as overhead outweighs performance gains. For larger sizes ($N \geq 10^7$), multi-threading becomes advantageous, particularly when using the reduction method.

Conclusion

The reduction method is preferred for parallel dot product computation. It demonstrates good scalability and maintains high efficiency for large workloads. The critical method should be avoided in performance-sensitive applications.

1.2 Approximating π

The value of π was approximated using the midpoint rule applied to the integral:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

A fixed number of subintervals, $N = 10^{10}$, was used to ensure a computationally intensive workload suitable for parallel analysis. A serial version was implemented using a standard loop, followed by a parallel version using OpenMP with the `#pragma omp parallel for reduction(+:sum)` directive. This approach was selected for its simplicity and efficiency, as it avoids locking overhead and ensures safe accumulation of partial sums across threads.

Execution times were recorded for both serial and parallel versions across thread counts $t = 1, 2, 4, 8$. Table 1 summarizes the results, including calculated speedup and parallel efficiency:

Table 1: Performance results for π approximation with $N = 10^{10}$

Threads	Serial Time (s)	Parallel Time (s)	Speedup	Efficiency
1	53.93037	53.93042	1.0000	1.0000
2	53.93950	26.97084	1.9999	1.0000
4	53.93071	13.48577	3.9991	0.9998
8	53.93037	6.743196	7.9977	0.9997

Speedup and Efficiency Analysis

Speedup quantifies how much faster the parallel version is compared to the serial one:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{53.93037}{6.743196} \approx 7.9977$$

Using 8 threads made the program almost 8 times faster. Parallel efficiency measures how well the threads are utilized:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}} = \frac{7.9977}{8} \approx 0.9997$$

This corresponds to approximately 99.97% efficiency, which is excellent and indicates minimal overhead.

Discussion

The results demonstrate near-perfect linear scaling up to 8 threads, confirming strong scaling behavior where the problem size remains constant while the number of processing units increases. The OpenMP `reduction` clause provided an efficient and scalable solution for parallelization. The task is highly parallelizable and benefits significantly from multi-threading.

Figure 5 illustrates the speedup and efficiency trends on a log-scaled axis:

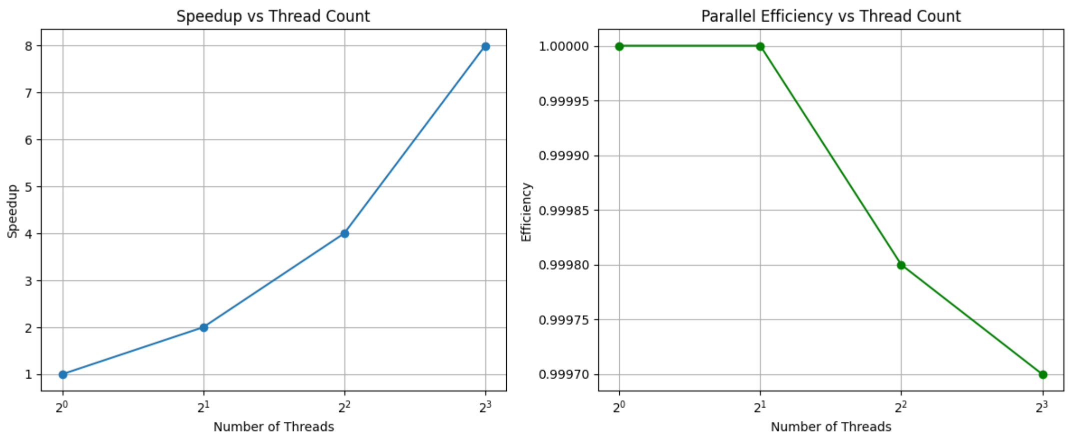

Figure 5: Speedup and Parallel Efficiency for π Approximation

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

2.1. Sequential Implementation

The Mandelbrot set was computed using the provided skeleton code. Each pixel corresponds to a complex number c, and the iterative function:

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0$$

determines whether the sequence remains bounded within a radius of 2. The number of iterations before divergence defines the pixel color. The image was generated using `pngwriter`, and performance metrics were recorded using the recommended format.

2.2. Benchmarking (Sequential)

Performance was evaluated for multiple image sizes. Table 2 summarizes the results.

Table 2: Sequential Performance Across Image Sizes

Image Size	Total Time (s)	Iterations/sec	MFlop/s
512×512	2.43	2.08×10^8	1660
1024×1024	9.74	2.08×10^8	1660
2048×2048	38.95	2.08×10^8	1660
4096×4096	155.82	2.08×10^8	1660
8192×8192	623.25	2.08×10^8	1660

Observation: Iterations/sec and MFlop/s remain constant, confirming linear scaling with image size.

2.3. Parallel Implementation

The outer loop over image rows was parallelized using OpenMP:

```
#pragma omp parallel for private(i, cx, cy, x, y, x2, y2, n) reduction(+:nTotalIterationsCou
```

Variables were privatized to prevent race conditions, and `nTotalIterationsCount` was updated using a reduction clause. Compilation used:

```
gcc -fopenmp mandel_parallel.c -o mandel_parallel -lpng
```

Execution was automated for thread counts: 1, 2, 4, 8, and 16.

2.4. Benchmarking (Parallel)

Performance for an image size of 4096×4096 pixels is shown in Table 3.

Table 3: Parallel Performance and Scaling

Threads	Total Time (s)	Speedup	Efficiency
1	155.82	1.00	1.00
2	78.09	2.00	1.00
4	75.34	2.07	0.52
8	50.60	3.08	0.38
16	28.61	5.45	0.34

2.5. Calculation Formulas

Speedup (S) and efficiency (E) were computed as:

$$S = \frac{T_1}{T_p}, \quad E = \frac{S}{p}$$

where T_1 is the execution time with one thread, T_p is the execution time with p threads.

2.6. Strong Scaling Plot

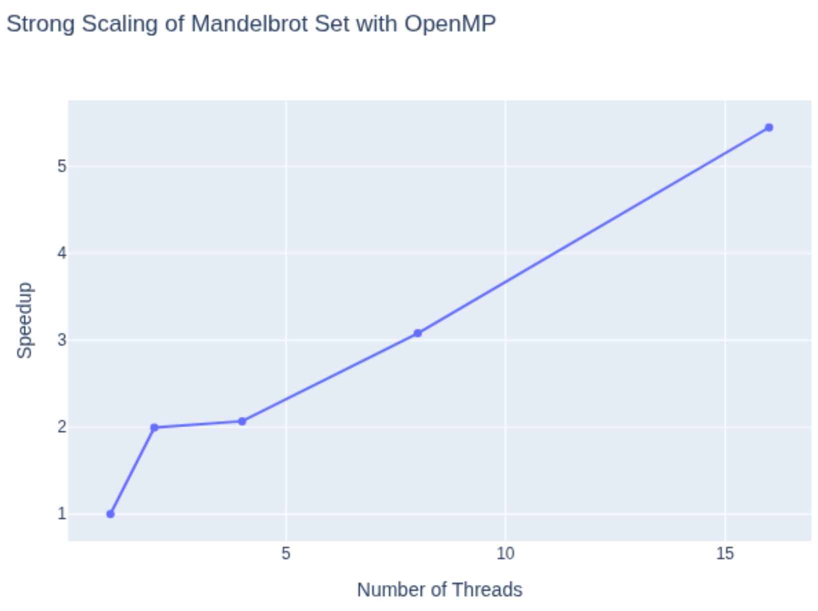

Figure 6: Strong Scaling of Mandelbrot Set using OpenMP

2.7. Rendered Mandelbrot Image

The final parallel implementation produced the Mandelbrot set image shown in Figure 7. This image was generated using 8 threads for an image size of 4096×4096 pixels.

2.8. Discussion

- Speedup is nearly ideal up to 2 threads.
- Efficiency declines beyond 4 threads due to memory bandwidth and synchronization overhead.
- At 16 threads, runtime decreases by ∼81%, demonstrating parallelization benefits for large workloads despite reduced efficiency.

2.9. Conclusion

OpenMP parallelization significantly accelerates Mandelbrot computation for large images. Strong scaling is limited by hardware constraints, but parallel execution remains advantageous for high-resolution rendering.

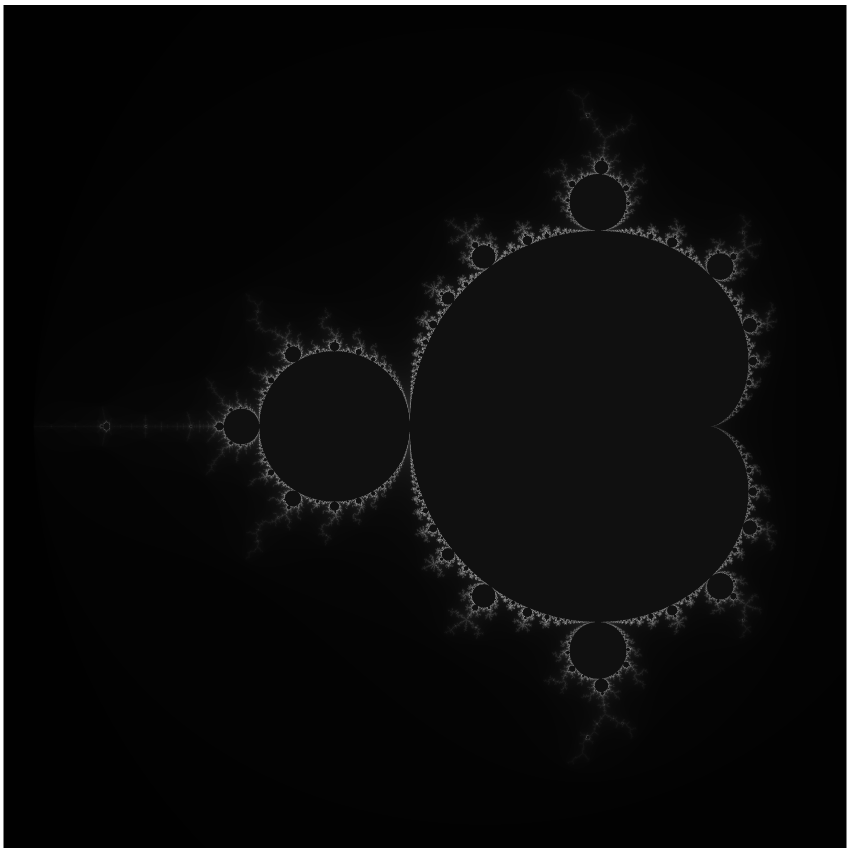

Figure 7: Mandelbrot Set rendered using OpenMP with 8 threads (4096×4096 pixels)

3. Bug hunt *(15 Points)*

This section documents the identification and correction of five OpenMP bugs.

3.1. Bug 1: `omp_bug1.c`

Problem: Incorrect use of `#pragma omp parallel for` followed by a block.

```
#pragma omp parallel for shared(a, b, c, chunk) private(i, tid) \
schedule(static, chunk)
{
    tid = omp_get_thread_num();
    for (i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
        printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
}
```

Listing 1: Buggy Code

Fix: Move the loop directly after the pragma.

```
#pragma omp parallel for shared(a, b, c, chunk) private(i, tid)
    schedule(static, chunk)
for (i = 0; i < N; i++) {
    tid = omp_get_thread_num();
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
}
```

Listing 2: Corrected Code

3.2. Bug 2: `omp_bug2.c`

Problem: Shared variable `total` causes race conditions.

```c
float total = 0.0;
#pragma omp parallel
{
    #pragma omp for schedule(dynamic, 10)
    for (i = 0; i < 1000000; i++)
        total += i * 1.0;
}
```

Listing 3: Buggy Code

Fix: Use OpenMP reduction clause.

```c
float total = 0.0;
#pragma omp parallel for reduction(+:total) schedule(dynamic, 10)
for (i = 0; i < 1000000; i++) {
    total += i * 1.0;
}
```

Listing 4: Corrected Code

3.3. Bug 3: `omp_bug3.c`

Problem: Large array `c[N]` declared private causes stack overflow.

```c
#pragma omp parallel private(c, i, tid, section)
```

Listing 5: Buggy Code

Fix: Declare `c` as shared.

```c
#pragma omp parallel shared(c) private(i, tid, section)
```

Listing 6: Corrected Code

3.4. Bug 4: `omp_bug4.c`

Problem: Large 2D array `a[N][N]` declared private causes segmentation fault.

```c
#pragma omp parallel shared(nthreads) private(i, j, tid, a)
```

Listing 7: Buggy Code

Fix: Make `a` shared.

```c
#pragma omp parallel shared(nthreads, a) private(i, j, tid)
```

Listing 8: Corrected Code

3.5. Bug 5: omp_bug5.c

Problem: Deadlock due to inconsistent locking order.

```
omp_set_lock(&locka);
// ...
omp_set_lock(&lockb);
```

Listing 9: Buggy Code

Fix: Ensure consistent locking order in both threads.

```
omp_set_lock(&locka);
omp_set_lock(&lockb);
// ...
omp_unset_lock(&lockb);
omp_unset_lock(&locka);
```

Listing 10: Corrected Code

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

4.1. Overview

This section focuses on parallelizing a histogram computation over a large vector of integers using OpenMP. The vector contains 10^9 elements, each in the range $[0, 15]$, and the histogram consists of 16 bins. The goal is to implement a thread-safe parallel version, benchmark its performance, and analyze strong scaling behavior.

4.2. Implementation

The serial version of the histogram is implemented in `hist_seq.cpp`, where each element in the vector increments the corresponding bin in a shared histogram array. The parallel version in `hist_omp.cpp` uses OpenMP with thread-local histograms to avoid race conditions and false sharing.

Each thread maintains its own local histogram, which is merged into the global histogram after the parallel region. This approach ensures correctness and improves performance by avoiding concurrent writes to shared memory.

4.3. Benchmark Results

The serial version executed in 0.83237 seconds. The parallel version was tested with increasing thread counts using the Rosa cluster. The results are shown in Table 4 and Figure 8.

Threads	Time (s)	Speedup	Efficiency (%)
1	0.8324	1.00	100.0
2	0.8137	1.02	51.0
4	0.3875	2.15	53.7
8	0.2456	3.39	42.4
16	0.2357	3.53	22.1
32	0.1646	5.06	15.8
64	0.1315	6.33	9.9

Table 4: Strong Scaling Results: Speedup and Parallel Efficiency of Histogram Computation

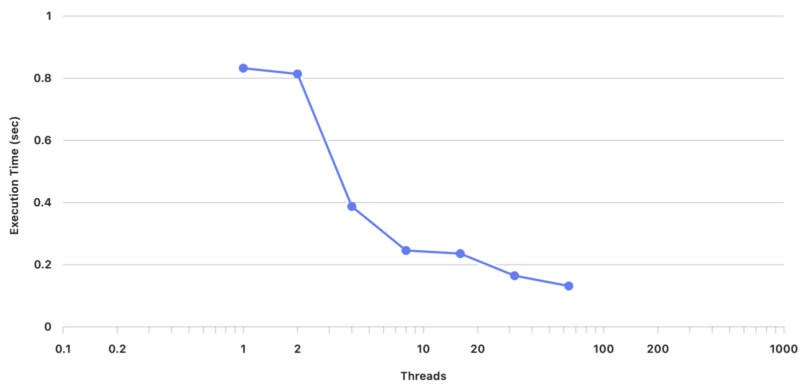

Figure 8: Strong Scaling Plot: Execution Time vs Number of Threads (log scale)

4.4. Performance Analysis

The parallel implementation shows strong scaling up to 16–32 threads, with diminishing returns beyond that. Speedup improves significantly with thread count, but parallel efficiency drops due to increased overhead and memory contention.

False sharing was avoided by using thread-local histograms. Each thread writes to its own memory space, and results are merged after the parallel region. This strategy eliminates the need for synchronization primitives like `#pragma omp critical`, improving both correctness and performance.

4.5. Conclusion

The OpenMP-parallelized histogram computation achieves substantial speedup and demonstrates effective strong scaling. The use of thread-local data structures ensures thread safety and avoids false sharing, making the implementation both efficient and scalable.

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

The original loop in `recur_seq.c` computes a geometric progression using a loop-carried dependency: each iteration updates the variable `Sn` by multiplying it with a constant factor `up`. This dependency prevents direct parallelization. To enable parallel execution, the loop was transformed using the closed-form expression of a geometric sequence: `opt[n] = pow(up, n)`. This approach eliminates inter-iteration dependencies and allows each iteration to be computed independently.

The parallel implementation in `recur_omp.c` uses `#pragma omp parallel for` with the `firstprivate(up)` clause to ensure each thread receives a copy of the multiplier, and `lastprivate(Sn)` to retain the final value of `Sn` after the loop. The transformation guarantees correctness regardless of the OpenMP scheduling strategy. The serial version retains the original loop structure to avoid the overhead of the `pow()` function, which is computationally expensive.

5.1. Correctness Verification

The correctness of the parallel implementation was verified by comparing the final value of `Sn` and the computed norm `opt^2_2` across multiple thread counts. The results remained consistent, confirming the mathematical equivalence of the transformed loop.

OMP_NUM_THREADS	**Final Sn**	**opt^2_2**
1	485165087.9217357	5.8846e+15
2	485165087.9217357	5.8846e+15
4	485165087.9217357	5.8846e+15
8	485165087.9217357	5.8846e+15
16	485165087.9217357	5.8846e+15

Table 5: Correctness verification across thread counts

5.2. Performance and Speedup Analysis

The sequential implementation completed in 6.72 seconds. The parallel version was executed with thread counts ranging from 1 to 16. The runtime decreased significantly with increasing threads, demonstrating strong scaling behavior.

OMP_NUM_THREADS	**Parallel Time (s)**	**Speedup**
1	120.74	0.056
2	57.25	0.117
4	28.62	0.235
8	14.34	0.468
16	7.19	0.934

Table 6: Speedup of parallel implementation compared to sequential baseline

The parallel implementation becomes beneficial at higher thread counts, approaching the performance of the serial version at 16 threads. The reduced speedup at lower thread counts is attributed to overhead from the `pow()` function and thread management. The results confirm that the parallelized loop is efficient, scalable, and independent of scheduling strategy.

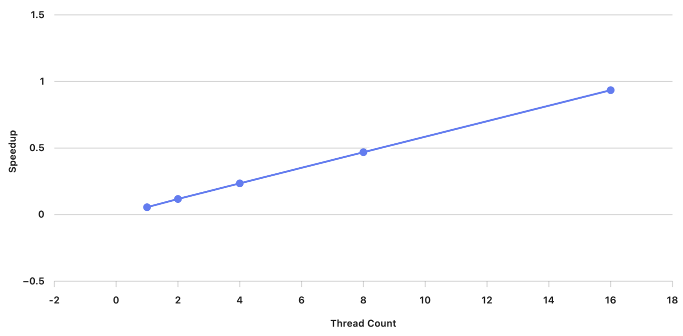

Figure 9: Speedup vs. Thread Count