Università della Svizzera italiana

Institute of Computing
CI

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

Student: FULL NAME Discussed with: FULL NAME

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 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 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

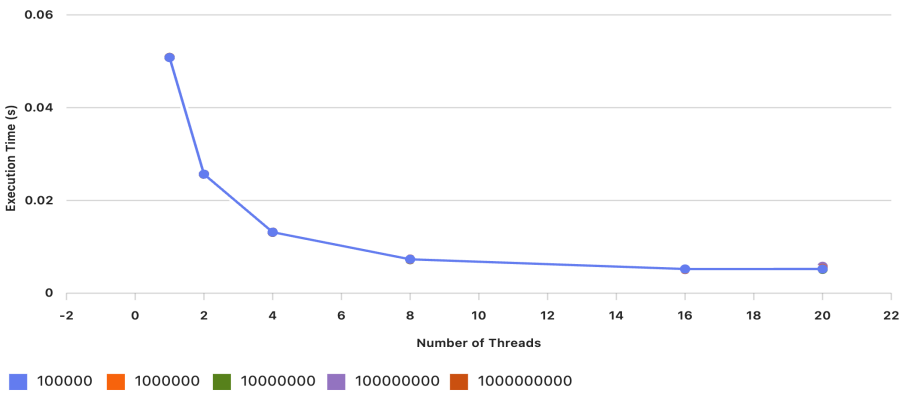

Figure 1: Strong scaling performance using the reduction method.

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

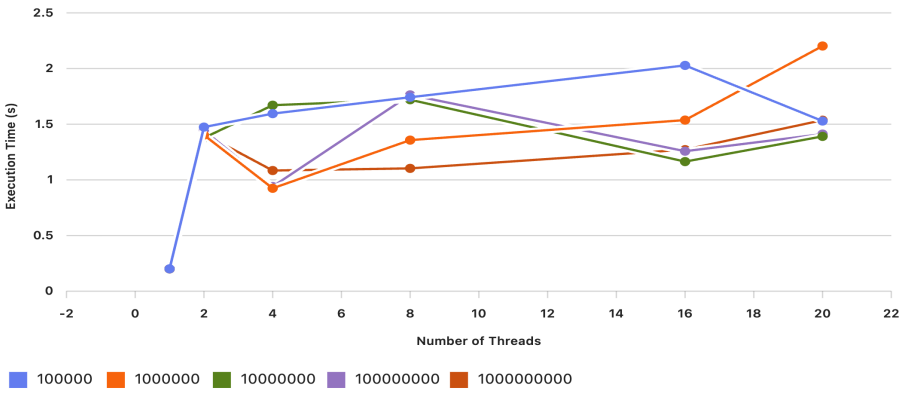

Figure 2: Strong scaling performance using the critical method.

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

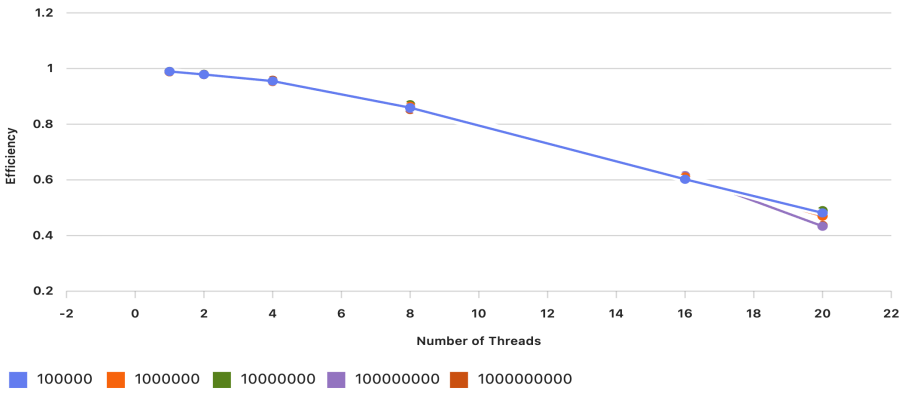

Figure 3: Parallel efficiency using the reduction method.

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

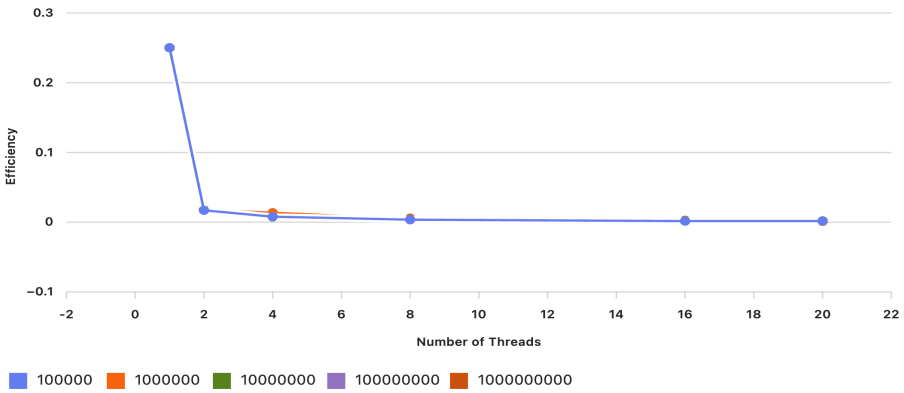

Figure 4: Parallel efficiency using the critical method.

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:

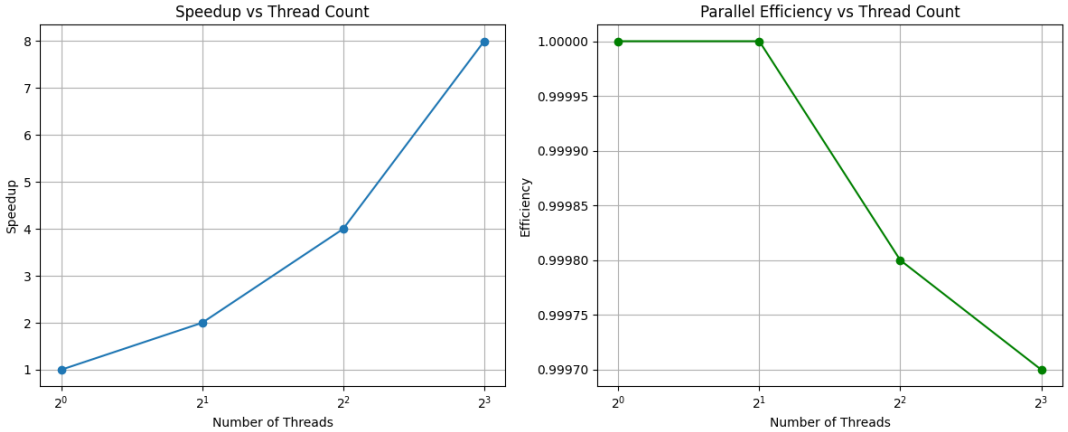

Figure 5: Speedup and Parallel Efficiency for π Approximation

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

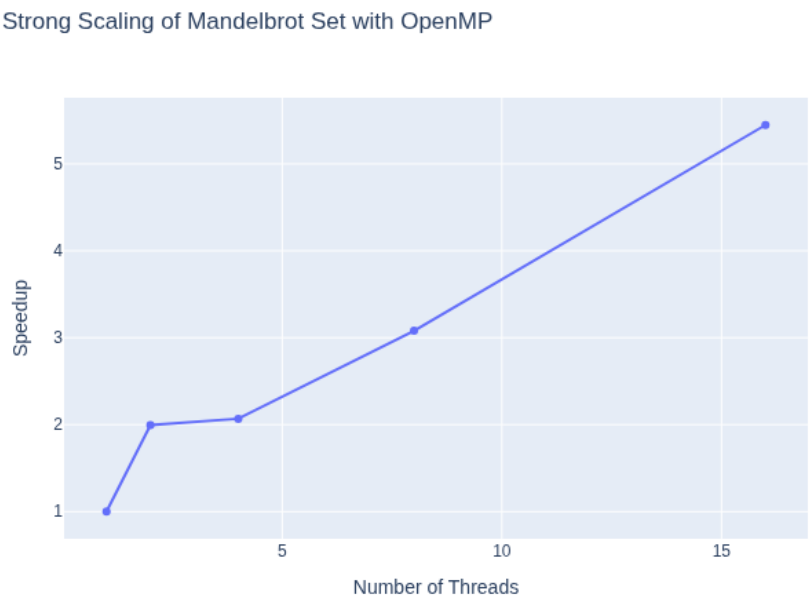

Figure 6: Strong Scaling of Mandelbrot Set using OpenMP

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.

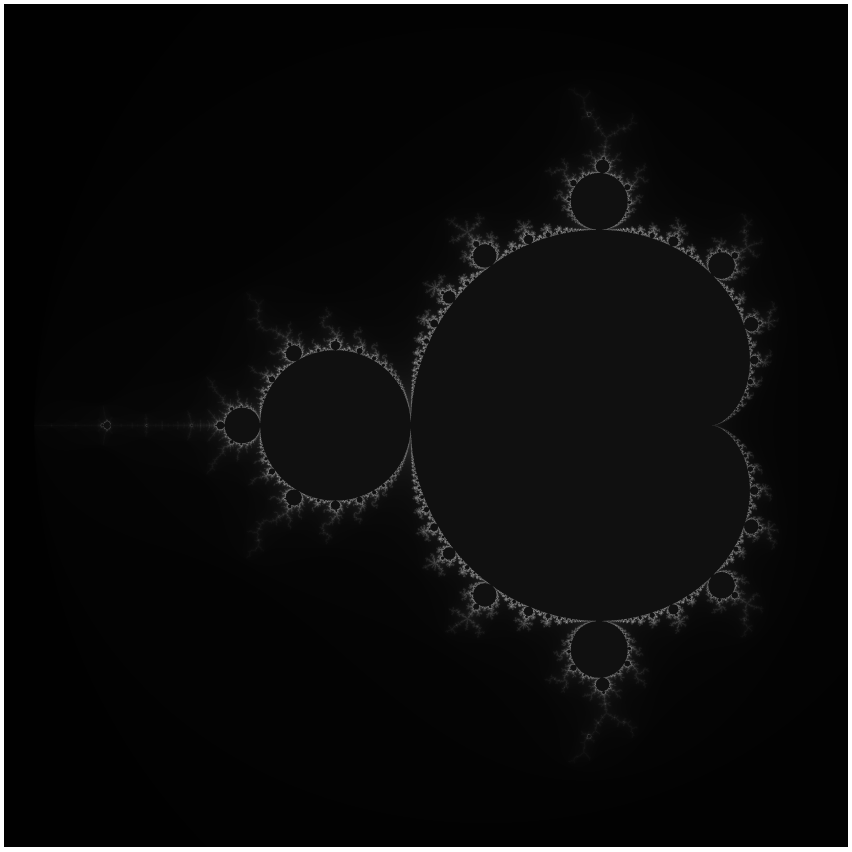

Figure 7: Mandelbrot Set rendered using OpenMP with 8 threads (4096×4096 pixels)

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.

```
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.

```
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.

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

Listing 5: Buggy Code

Fix: Declare `c` as shared.

```
#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.

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

Listing 7: Buggy Code

Fix: Make `a` shared.

```
#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)*

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

6. Quality of the Report *(15 Points)*