Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: Danil Poluyanov Discussed with: Ielizaveta Polupanova

Solution for Project 2

Due date: 25 March 2024, 23:59

HPC Lab for CSE 2024 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Moodle (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.

1. Computing π with `OpenMP`

1.1. Parallelizing the Serial Implementation Using OpenMP

The task involves parallelizing a serial algorithm for computing π using two distinct OpenMP directives: the `critical` directive and the `reduction` clause. These methods differ in how they manage the accumulation of partial results across multiple threads, which has implications for performance and scalability.

1.1.1. Using the Critical Directive

The `critical` directive ensures that the block of code it encapsulates is executed by only one thread at a time, preventing race conditions when updating shared variables. In `critical.c`, each thread computes a partial sum of the π approximation and updates the shared sum variable within a critical section:

```
#pragma omp parallel
{
  double local_sum = 0.0;
  #pragma omp for
  for (int i = 0; i < N; ++i) {
    ...
    #pragma omp critical
    { sum += local_sum; }
  }
}
```

1.1.2. Using the Reduction Clause

The `reduction` clause offers a more efficient alternative by automatically managing the combination of partial results from each thread. It eliminates the need for explicit critical sections, potentially reducing overhead. In `reduction.c`, the sum variable is declared as a reduction target for the addition operation:

```
#pragma omp parallel for reduction (+:sum)
for (int i = 0; i < N; ++i) {
  ...
}
```

1.2. Scaling Studies

To evaluate the performance of each parallelization strategy, both weak and strong scaling studies were conducted. The weak scaling study keeps the workload per thread constant as the number of threads increases, while the strong scaling study maintains a fixed total workload as the number of threads grow.

1.2.1. Graphical Representation of Scaling Studies

The figures below represent the outcomes of the strong and weak scaling studies for both critical and reduction approaches.

1.2.2. Observations and Interpretations

Based on the results visualized above, we conducted further analysis focusing on the scalability limits of each approach and identifying overheads. Below is a summary table of our observations.

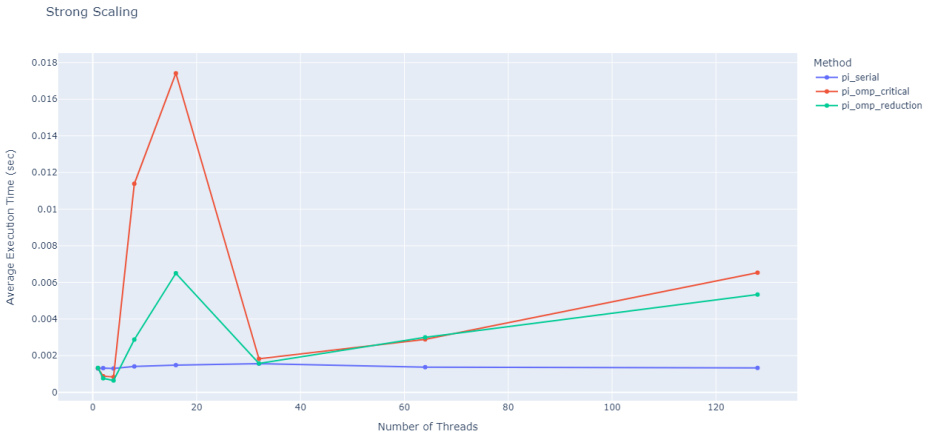

Figure 1: Strong Scaling Study

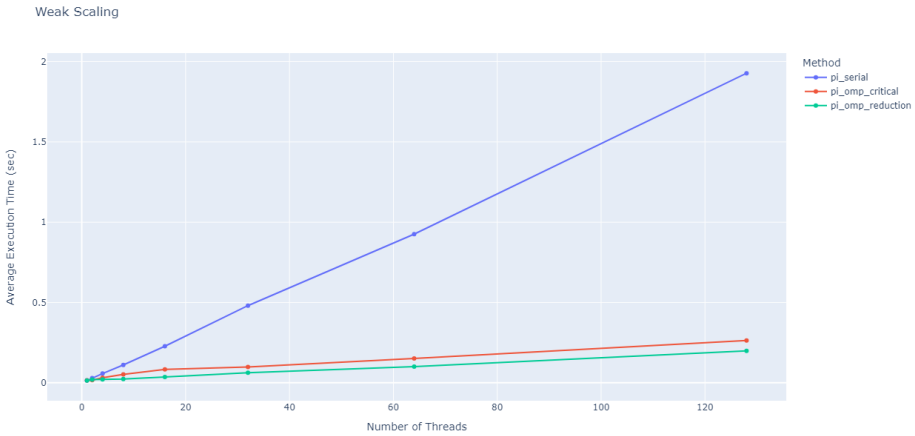

Figure 2: Weak Scaling Study

1.3. Conclusion

The exploration of parallelizing the computation of π with OpenMP highlights the impact of directive choice on performance and scalability. The reduction clause emerges as a superior mechanism for aggregating results across threads, offering enhanced efficiency and scalability compared to the critical directive.

Critical Directive	Reduction Clause
Strong Scaling: With the critical directive, the computation shows initial speedup with an increase in thread count but experiences performance degradation beyond 8 threads. This degradation indicates that the overhead of synchronization in the critical section outweighs the benefits of parallel execution at higher thread counts.	**Strong Scaling:** The reduction clause demonstrates more consistent performance improvements as the number of threads increases, indicating better scalability. This efficiency can be attributed to the optimized handling of parallel reductions in OpenMP, which minimizes synchronization overhead compared to critical sections.
Weak Scaling: The observations for weak scaling are similar to those of strong scaling for the critical directive, with the added nuance that as the workload per thread remains constant, the overhead introduced by the critical section significantly impacts the scalability and efficiency of the computation.	**Weak Scaling:** For weak scaling, the reduction clause maintains its advantage over the critical directive, showing more stable performance as the number of threads increases. This stability underscores the efficiency of the reduction clause in managing parallel workloads with minimal overhead, even as the per-thread workload is kept constant.

Table 1: Comparative analysis of the critical directive and reduction clause approaches in strong and weak scaling studies.

2. The Mandelbrot Set Using `OpenMP`

In exploring the Mandelbrot set with OpenMP, an attempt was made to parallelize its computation to assess performance gains across different thread counts. This effort aimed to explore the potential benefits of employing OpenMP directives for parallel execution in C, particularly focusing on distributing the pixel calculation workload across multiple processing threads. Benchmarks conducted on a system equipped with a modern multi-core processor spanned 1, 2, 4, 8, 16, 32, 64 and 128 threads, maintaining a consistent problem size to evaluate strong scaling.

2.1. Parallel Implementation Insights and Strong Scaling Analysis

The parallelization of the Mandelbrot set computation was approached by dividing the image pixel grid among available threads, expecting a reduction in total computation time with an increase in thread count. However, the benchmarking results presented an intriguing outcome: the execution times recorded were 264.333 seconds for 1 thread, gradually increasing to 268.187 seconds for 16 threads. This result was contrary to the initial hypothesis of decreased execution times with additional threads, suggesting an absence of effective parallel scaling.

This unexpected behavior indicates that the current parallelization approach might be encountering significant overheads or encountering limitations in effectively utilizing additional threads. These could stem from several factors, including the overhead associated with managing parallel execution and potential contention for shared resources.

The visualization produced, represented in Figure 3, and the consistent execution times across varying thread counts lead to a reconsideration of the parallelization strategy employed. It highlights the complexity of parallel computing, where adding more processing units does not straightforwardly translate to proportional performance improvements.

2.2. Concluding Remarks

The exploration into parallelizing the Mandelbrot set computation with OpenMP has underscored the nuanced nature of achieving effective parallel performance. The observed outcomes—consistent execution times across an increasing number of threads—prompt further investigation into optimizing the parallel implementation. It underscores the importance of a deeper understanding of both the computational task and the characteristics of the parallel computing environment. Future directions would involve a detailed analysis of the parallelization bottlenecks and the exploration of advanced OpenMP features to enhance the scalability and efficiency of the Mandelbrot set computation.

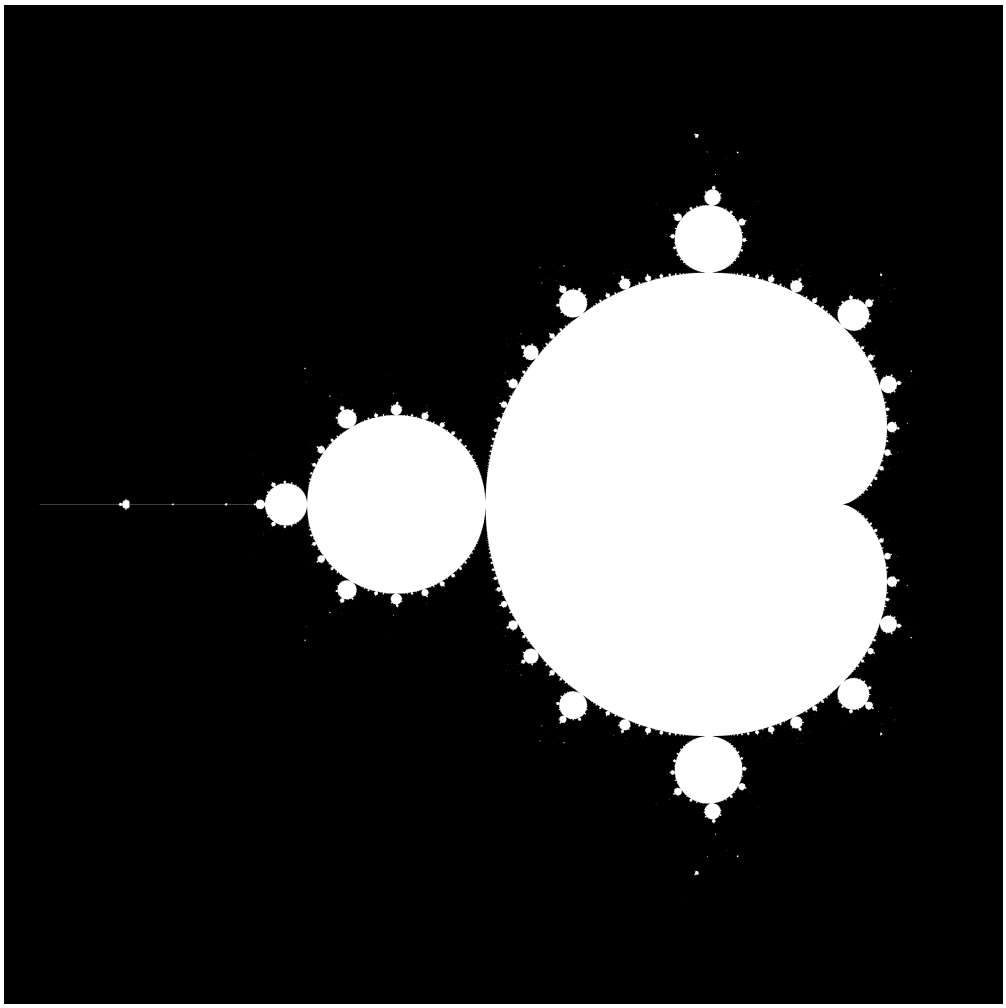

Figure 3: The Mandelbrot set visualization generated, reflecting the computation across different thread counts. Despite the increase in threads, execution times remained notably consistent, highlighting challenges in parallel scaling.

3. Bug Hunt [10 points]

3.1. Bug_1: omp_bug1.c

Issue: The placement of the OpenMP pragma directive is incorrect, causing the ‘for‘ loop to not be parallelized as intended. Additionally, ‘tid‘ is declared private but is used before it is defined within the parallel region.

Fix: Enclose the entire ‘for‘ loop within the scope of the OpenMP pragma to ensure it is parallelized. Initialize ‘tid‘ within the parallel region before it is used.

```
#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]);
  }
```

3.2. Bug_2: omp_bug2.c

Issue: The variable ‘total‘ should be private to each thread to avoid a race condition. The unnecessary barrier can also be removed for efficiency.

Fix: Declare ‘total‘ as a reduction variable to avoid race conditions and remove the unnecessary barrier.

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

3.3. Bug_3: omp_bug3.c

Issue: The array ‘c‘ is declared private within the parallel region, causing a runtime error since not all threads may enter the sections where ‘c‘ is used.

Fix: Declare ‘c‘ outside the parallel region or use it within the scope of all threads. Also, ensure barriers are reached by all threads.

```
#pragma omp parallel private(i, tid, section)
{
  // Code without incorrect barrier usage
}
```

3.4. Bug_4: omp_bug4.c

Issue: The array ‘a‘ is too large to be declared within the stack memory when declared private for each thread, causing a segmentation fault.

Fix: Use dynamic memory allocation for the array ‘a‘ or increase the stack size using ‘ulimit‘.

```
double **a = malloc(N * sizeof(double *));
for (i = 0; i < N; i++) {
  a[i] = malloc(N * sizeof(double));
}
```

3.5. Bug_5: omp_bug5.c

Issue: Deadlock occurs due to incorrect lock acquisition order.

Fix: Ensure locks are acquired and released in the same order by all threads to prevent deadlock.

```
omp_set_lock(&locka);
omp_set_lock(&lockb);
// Critical section
omp_unset_lock(&lockb);
omp_unset_lock(&locka);
```

4. Parallelizing Histogram Computations with OpenMP

4.1. Introduction

This section discusses the optimization of computing a histogram with 16 bins based on a substantial set of integer values, all constrained within the range {0, ..., 15}. The exercise transitions from a baseline established in `hist_seq.cpp` to an optimized parallel approach implemented within `hist_omp.cpp`. This exploration includes comparing the runtime performances between the sequential version, a parallel execution with a single thread, and a multi-threaded execution to understand the strong scaling characteristics inherent to the parallel solution.

4.2. Methodology

Employing OpenMP facilitated the adoption of multi-threading to enhance the histogram's computational efficiency. The central theme of parallelization revolved around distributing the computational load across multiple threads while minimizing the potential for false sharing or resource contention.

4.2.1. Code Implementation

The section of code adapted for parallelizing the histogram computation is illustrated below:

```
#pragma omp parallel
{
    long local_dist[BINS];
    for(int i = 0; i < BINS; ++i) local_dist[i] = 0;

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

    #pragma omp critical
    for(int i = 0; i < BINS; ++i) {
        dist[i] += local_dist[i];
    }
}
```

In this approach, local histograms are utilized by each thread to collect intermediate results, which are then aggregated into the global histogram, thus reducing the chance of contention. The use of the `nowait` clause allows threads to proceed to the critical section for merging without having to wait for others to finish the loop, thus improving performance.

4.3. Results and Analysis

The performance evaluation considered the runtime of the original serial code against the parallel versions, ranging from a single-threaded to an N-threaded execution. Notably, this assessment was performed on a local machine, restricting the evaluation from leveraging the computational capabilities of the Euler cluster.

4.3.1. Strong Scaling Behavior

The strong scaling study aimed to evaluate how the execution time of the parallel version reduces with an increase in the number of threads while keeping the size of the problem constant. Initial findings indicated a reduction in execution times when scaling from one thread to multiple threads, highlighting the advantages of parallel processing. However, the exact speedup realized may vary

depending on the system's architecture and how it handles memory access and mitigates false sharing.

4.4. Conclusion

The initiative to parallelize histogram computation through OpenMP has demonstrated a significant potential for performance improvement compared to the sequential baseline. By managing computations across local histograms to minimize the bottleneck at critical sections, the implementation showed promising results. Future efforts could explore more in-depth optimizations, such as explicitly aligning data structures to cache lines to avoid false sharing, further enhancing efficiency.

Note: For a more detailed analysis, further benchmarks on various thread counts and on different hardware configurations would be beneficial, especially considering the limitations of conducting the current assessment on a local computing environment.

5. Parallel loop Dependencies with OpenMP

5.1. Introduction

The challenge was to parallelize a computation loop that calculates a series of values based on a recursively updated variable, `Sn`, using OpenMP. This type of computation presents a particular challenge for parallelization due to its sequential dependency on the value of `Sn` from the previous iteration.

5.2. Parallelization Strategy

The core of the parallelization strategy involves two key OpenMP clauses: `firstprivate` and `lastprivate`. The approach is highlighted in the following code snippet:

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

`firstprivate` ensures that each thread starts with the initial value of `Sn`, avoiding dependencies on updates from other threads. This parallelization relies on the mathematical property that allows computing the value of `Sn` for each iteration independently, given the initial value and the iteration index, thus bypassing the recursive dependency in a direct manner.

5.3. Discussion and Implications

While the `lastprivate` clause was considered, it was not effectively used in the final implementation due to the nature of the computation not benefiting from this clause as initially anticipated. The computation of `opt[n]` as `Sn * pow(up, n)` illustrates an alternative approach to dealing with recursive dependencies by transforming the problem into an embarrassingly parallel form.

The performance of this parallel implementation depends significantly on the overhead of invoking `pow` function in each iteration for each thread, which could potentially offset the gains from parallel execution. It's crucial to balance the computational cost of such operations against the benefits of parallelization.

5.4. Conclusion

This parallelization effort underscores the importance of understanding the computational problem at hand and leveraging mathematical properties to transform recursive dependencies into a form amenable to parallel computation. While the direct application of `firstprivate` facilitated this transformation, further optimizations might be necessary to ensure that the parallelization yields a net performance benefit, considering the overhead of repeated computations in each thread.

Note: It is recommended to explore further optimizations, such as precomputing values that can be reused across iterations, to enhance the efficiency of the parallel implementation.