Università della Svizzera italiana

Institute of Computing
CI

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

Student: FILIPPO WANG

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 source files (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 introduces parallel programming using OpenMP.

Contents

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

1.1. Dot Product

In this section we analyze two parallel implementations of the dot product operation were developed using OpenMP: one utilizing the `reduction` clause and another using the `critical` directive. The core implementations are shown below:

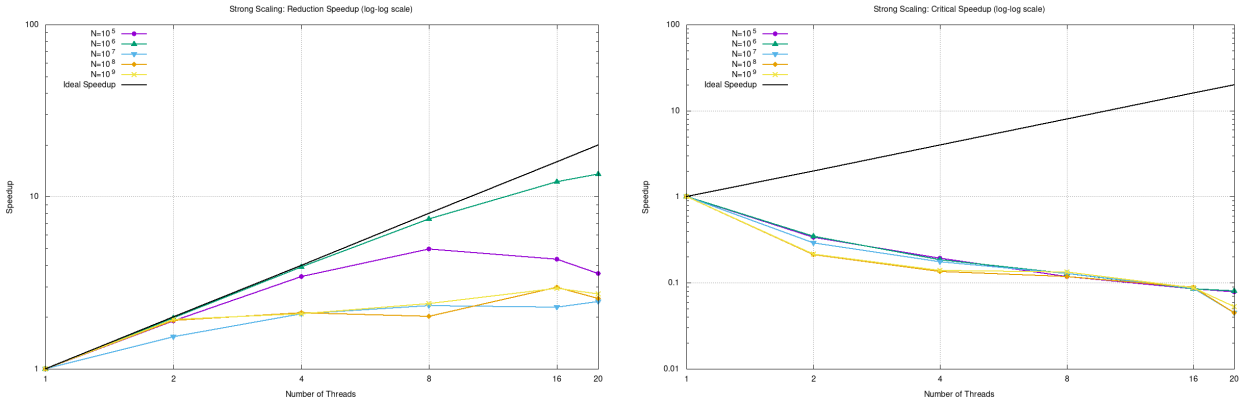
```
#pragma omp parallel for reduction(+:alpha_parallel)
for (int i = 0; i < N; i++) {
    alpha_parallel += a[i] * b[i];
}
```

Listing 1: Dot product implementation using reduction clause

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

Listing 2: Dot product implementation using critical directive

The `reduction` clause provides an efficient mechanism for parallel accumulation by creating private copies of the reduction variable for each thread, which are then combined at the end using a tree-based reduction. In contrast, the `critical` directive forces all threads to serialize access to the shared variable `alpha_parallel`, creating a bottleneck where each thread must wait for exclusive access to perform a single floating-point addition. This approach exhibits embarrassingly poor performance as it combines the overhead of thread creation and execution with essentially serial execution of the actual computation. In general, OpenMP overhead arises from thread management and synchronization. In our case, this overhead is minimal for the `reduction` clause but becomes a severe bottleneck for the `critical` directive.



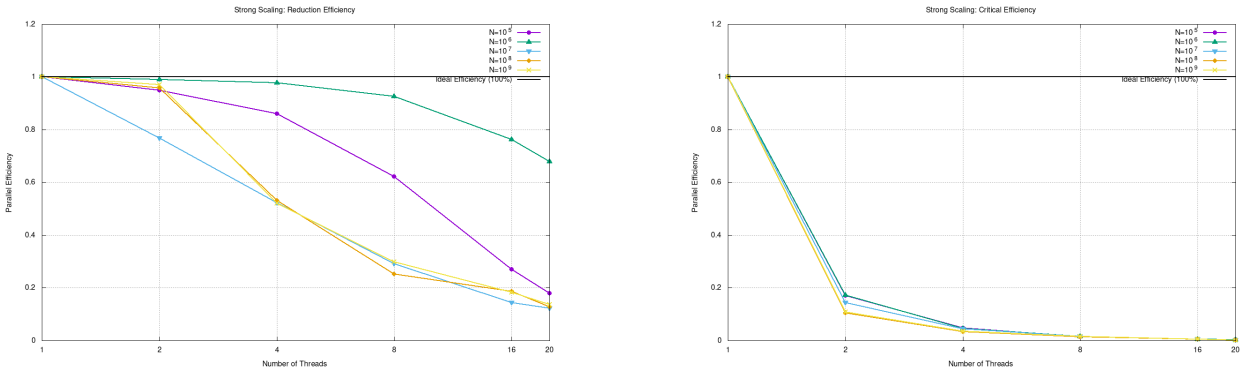
(a) Reduction speedup

(b) Critical efficiency

Figure 1: strong scaling comparison between reduction (left) and critical clause (right) implementations

A strong scaling analysis was performed on the Rosa cluster using compute nodes accessed through `sbatch run_dotProduct.sh`. The experiments tested both implementations with thread counts $t \in \{1, 2, 4, 8, 16, 20\}$ and vector lengths $N \in \{10^5, 10^6, 10^7, 10^8, 10^9\}$. Due to the high runtime of the critical implementation, the number of iterations for the critical version was reduced

from 100 to 5, and the measured time was manually scaled by a factor of 20 to estimate the equivalent 100-iteration runtime, see `dotProduct.cpp` for details. All performance plots were generated using gnuplot.



(a) Reduction efficiency

(b) Critical efficiency

Figure 2: Comparison between reduction (left) and critical clause (right) efficiency in OpenMP

Multithreading the dot product using the reduction implementation becomes beneficial at a vector size at $N = 10^5$ based on our experiments, but it's likely to be beneficial even for smaller arrays, as this is the point where the speedup consistently exceeds 1 for more than one thread. On the other hand, we should never use the critical implementation over the serial version of the dot product since the speedup is always lower than 1 when using multiple threads.

1.2. Approximating π

The value of π can be approximated using numerical integration of the function $f(x) = \frac{4}{1+x^2}$ over the interval $[0, 1]$, which equals π. This can be computed using the midpoint rule:

$$\pi \approx \sum_{i=0}^{N-1} f(x_i) \cdot \Delta x, \quad \text{where} \quad x_i = (i + 0.5) \cdot \Delta x, \quad \Delta x = \frac{1}{N}$$

In this section we implement a serial and a parallel version of the π approximation using OpenMP with $N = 10^{10}$. The parallel version uses the `reduction` clause because it creates private partial sums for each thread and efficiently combines them using a reduction tree.

```
#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);
}
pi_parallel = sum * dx;
```

Listing 3: Parallel π approximation using reduction clause

Speedup measurements were performed for $t \in \{1, 2, 4, 8\}$ threads. The speedup $S(t)$ is defined as the ratio of serial execution time to parallel execution time with t threads: $S(t) = T_{\text{serial}}/T_{\text{parallel}}(t)$. In the context of strong scaling, ideal speedup would be linear: $S(t) = t$. However, in reality it is limited by factors such as overhead, synchronization, and memory bandwidth saturation.

In our experiment we observe an almost-perfect speedup, because the algorithm is embarrassingly parallel. Parallel efficiency is defined as $E(t) = S(t)/t$, representing the fraction of ideal speedup achieved. For perfectly scalable code, $E(t) = 1$, but efficiency typically decreases as thread count increases due to overhead becoming more significant relative to computation. Again, the reduction operation in this π approximation is embarrassingly parallel making it very efficient.

Threads	Serial Time (s)	Parallel Time (s)	Speedup	Efficiency
1	53.931738	53.931362	1.000007	1.000007
2	53.931407	26.968633	1.999783	0.999892
4	53.929562	13.484124	3.999486	0.999872
8	53.931643	6.743289	7.997824	0.999728

Table 1: Measured speedup values for the parallel π approximation

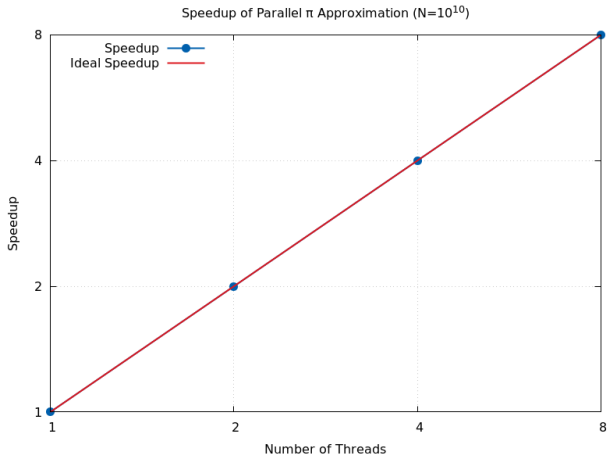


Figure 3: Speedup of the parallel π approximation

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

In this section, we propose a serial and a parallel implementation to compute the Mandelbrot set.

2.1. Serial Implementation

The serial sequential iterates over each pixel in the image. For each complex number $c = c_x + ic_y$ in the parameter space, the algorithm computes the orbit $z_0 = 0, z_{n+1} = z_n^2 + c$ and counts how many iterations are needed before $|z_n| > 2$ or the maximum iteration count is reached. The "orbit simulation" snippet of code is shown in 4, the complete implementation can be found in `mandel_seq_serial.c`.

```
while (n < MAX_ITERS && (x2 + y2) < 4.0) {
  y = 2.0 * x * y + cy; // imaginary part
  x = x2 - y2 + cx;     // real part
  x2 = x * x;
  y2 = y * y;
  n++;
}
```

Listing 4: Serial orbit simulation

2.2. OpenMP Parallelization Strategy

The Mandelbrot computation is embarrassingly parallel: each pixel calculation is completely independent of all others. We parallelize the outer loop (rows) using OpenMP, as shown in Listing 5. The complete implementation is available in `mandel_seq_approx.c`.

```
#pragma omp parallel for private(i, cx, cy, x, y, x2, y2) \
reduction(+:nTotalIterationsCount) schedule(dynamic)
for (j = 0; j < IMAGE_HEIGHT; j++) {
  cy = MIN_Y + j * fDeltaY;  // Directly compute from row j
  cx = MIN_X;
  for (i = 0; i < IMAGE_WIDTH; i++) {
    // ... same orbit simulation as serial version ...
  }
}
```

Listing 5: OpenMP parallel implementation (approximate)

Design decisions:

- **Outer loop parallelization**: Distributing rows among threads provides coarse-grained parallelism, reducing OpenMP overhead and improving cache locality within each row (because of row-major order).
- **Dynamic scheduling**: Different regions of the Mandelbrot set may require different iteration counts that are hard to predict, thus I decided to use `schedule(dynamic)` .
- **Reduction clause**: The variable `nTotalIterationsCount` is updated by all threads. The `reduction(+:nTotalIterationsCount)` clause ensures thread-safe accumulation.
- **Private variables**: Loop variables (`i` , `cx` , `cy` , `x` , `y` , `x2` , `y2`) are declared `private` to ensure each thread has its own copy, avoiding race conditions.
- **Direct coordinate calculation**: Since parallel iterations execute in arbitrary order because of concurrency, we cannot rely on incremental updates. `cy = MIN_Y + j * fDeltaY` directly computes the imaginary part of c from the row index j, ensuring each thread can independently calculate its assigned rows' coordinates regardless of execution order.

2.3. Floating-Point operations are not exact...

While the direct calculation `cy = MIN_Y + j * fDeltaY` is natural for parallel execution, it introduces a subtle issue...I noticed that the images it produces have some extra pixels that belong to the Mandelbrot set with respect to the image produced by the serial implementation (colored in white). Figure 4 shows the difference between the images (you might have to zoom in to notice)

I believe that the floating-point multiplication to compute cy may have rounding errors that can cause pixels to flip between 'in set' and 'out of set',

To address this, I implemented an "exact" (I mean, more precise) version that replaces the multiplication with a for-loop of sums, as shown in 6. The complete implementation is available in `mandel_seq_exact.c` .

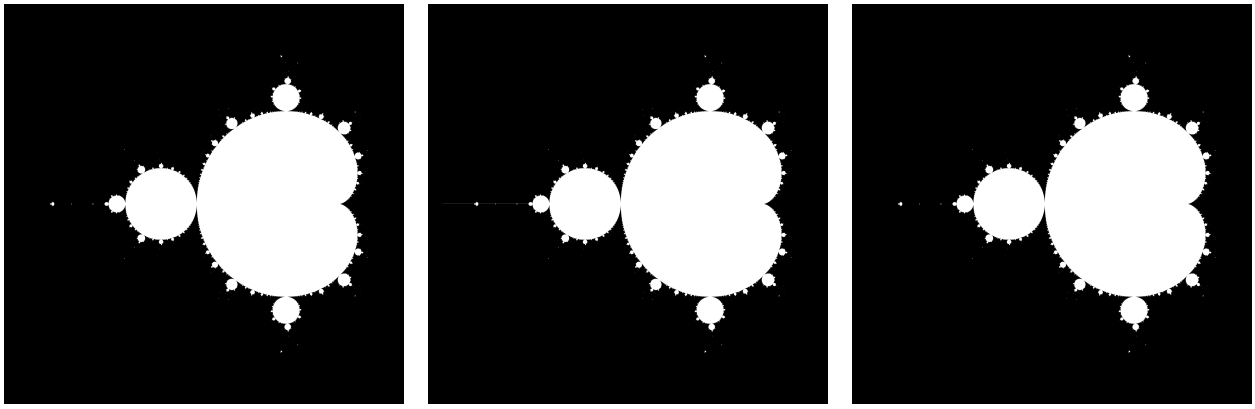
```
for (j = 0; j < IMAGE_HEIGHT; j++) {
  // accumulate j additions
  cy = MIN_Y;
  for (int k = 0; k < j; k++) {
    cy += fDeltaY;
  }
  // ...
  }
```

Listing 6: multiplication as sum

This approach performs j additions of `fDeltaY` for each row j, ($O(n^2)$ additions instead of $O(n)$ multiplications). Though this overhead is almost negligible compared to the number of iterations in the Mandelbrot computation itself, as shown in the strong-scaling plots in 5

2.4. Visual Comparison

Figure 4 shows the three implementations rendering the same region of the Mandelbrot set. While the overall structure is similar, there are subtle differences in center row pixels between the approximate parallel version and the serial/exact versions. The images are consistent between runs of different amount of threads. Tests were performed using $t = \{1, 2, 4, 8, 16, 20\}$ number of threads. I tried some experiments with images of size 2048x2048, and the approximation issue is still present.



(a) Serial (b) Parallel (approximate) (c) Parallel (exact)

Figure 4: Mandelbrot set images (4096×4096 pixels) generated by three implementations.

2.5. Strong Scaling Performance

Figure 5 presents strong scaling results for both parallel implementations with a fixed problem size of 4096×4096 pixels.

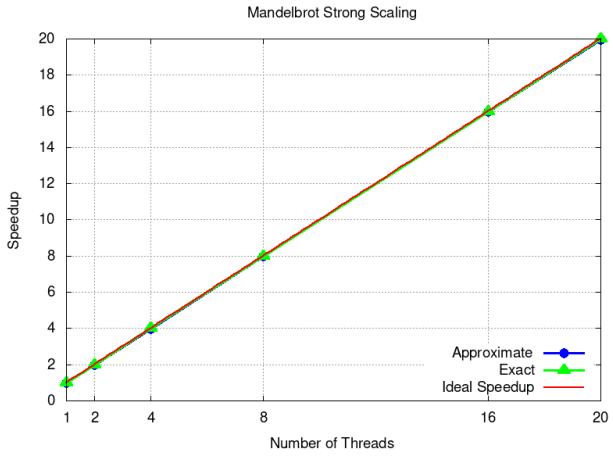


Figure 5: Strong scaling performance of parallel Mandelbrot implementations

Both implementations achieve near-linear speedup, confirming the above discussion.

3. Bug hunt *(15 Points)*

This section analyzes five buggy OpenMP programs (`omp_bug1.c` through `omp_bug5.c`). We try to find them and propose a fix.

3.1. Bug 1: Incorrect parallel For syntax

The `parallel for` construct is a combined directive that creates a parallel region and distributes loop iterations among threads. It must be immediately followed by a `for` statement, not a compound block.

Fix: Separate the `parallel` and `for` directives to allow `tid` assignment in the parallel region

```
#pragma omp parallel shared(a, b, c, chunk) private(i, tid)
{
  tid = omp_get_thread_num();
  #pragma omp for schedule(static, chunk)
  for (i = 0; i < N; i++) {
  // ...
  }
}
```

3.2. Bug 2: Variable scope violation

The variable `total` is implicitly `private` due to the `#pragma omp for` construct—each thread maintains its own private copy, which is discarded after the parallel region. These variables should be accessed only within the parallel region where they are valid, while in the code, it tries to print it outside the parallel region.

Fix: Declare `total` as `shared` with a `reduction` clause

3.3. Bug 3: Waiting forever

There is a `barrier` directive placed inside the `print_results()` function that is only called by a subset of threads, because the `#pragma omp sections` construct assigns each section to a single thread. With two sections, only two threads execute and call `print_results()`. However, a `barrier` requires *all* threads in the team to reach it for synchronization. The remaining threads never enter the function, never encounter the `barrier`, resulting in a deadlock as the two threads wait indefinitely.

Fix: Remove the `barrier` from the function and place it in the main parallel region.

3.4. Bug 4: Stack Overflow

A large array ($1048 \times 1048 \times 8$ bytes ≈ 8.8 MB) is declared as `private`, so each thread receives its own copy allocated on the thread's stack. As explained in the Stack Overflow discussion, thread stacks usually have a limited size (typically 1-4 MB by default, so it causes a stack overflow and segmentation fault when each thread attempts to allocate its private copy.

Fix: we can manually increase the stack size using `OMP_STACKSIZE=16M`, but we would have to handle the main thread since "This value does not affect the stack size of the main thread." (from Stack Overflow discussion).

3.5. Bug 5: Classic Deadlock

Two sections acquire the same locks in opposite order, creating a classic deadlock scenario. It may happend that Thread 1 acquires `locka` and Thread 2 simultaneously acquires `lockb`, both threads then attempt to acquire the other's lock.

Fix: acquire (both) locks in the same sequence across all threads, perform the computation on both variables, and then release the 2 locks.

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

4.1. Implementation

In this section, we try to parallelize the computation of a histogram.

```
long dist[BINS];
for (int i = 0; i < BINS; ++i) {
  dist[i] = 0;
}

for (long i = 0; i < VEC_SIZE; ++i) {
  dist[vec[i]]++;  // Increment bin counter
}
```

Listing 7: Sequential histogram implementation (`hist_seq.cpp`)

To parallelize this computation we make each thread maintain its own local histogram during the computation phase, then combines results in a critical section. This approach avoids race conditions 8.

```
#pragma omp parallel
{
  // Each thread gets its own private histogram
  long thread_dist[BINS] = {0};

  // Distribute work across threads
  #pragma omp for
  for (long i = 0; i < VEC_SIZE; ++i) {
    thread_dist[vec[i]]++;
  }

  // Combine private histograms
  #pragma omp critical
  {
    for (int i = 0; i < BINS; ++i) {
      dist[i] += thread_dist[i];
    }
  }
}
```

Listing 8: Parallel histogram with private histograms (`hist_omp.cpp`)

The `#pragma omp for` directive distributes the 10^9 elemements array across threads, with each thread updating only its private `thread_dist` array. The `#pragma omp critical` section performs a reduction involving 16 additions per thread.

4.2. Performance Analysis

The sequential implementation completes in 0.829155 seconds, while the single-threaded parallel version requires 0.832413 seconds, representing only 0.4% overhead, we can infer that there is minimal OpenMP overhead.

Figure 6 presents the strong scaling behavior across 1 to 128 threads. We can notice two clearly distinct areas in the plot.

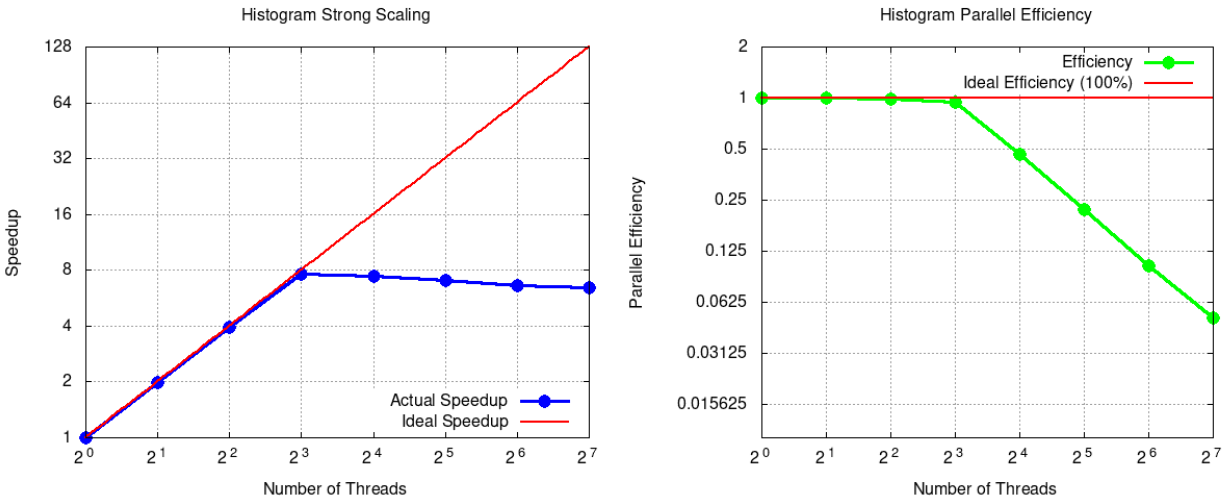


Figure 6: Strong scaling (left) and parallel efficiency (right) of histogram computation.

Near-linear scaling (1–8 threads): The implementation achieves almost-linear speedup in this range, reaching 7.6× on 8 threads with 95% efficiency.

Performance degradation (16–128 threads): Beyond 8 threads, despite the Intel Xeon E5-2650 v3 having 10 physical cores, speedup plateaus at approximately 7.4× and efficiency drops dramatically to 46% (16 threads), 22% (32 threads), and just 5% (128 threads).

The reason might be related to memory bandwidth saturation.

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

This problem contains a loop-carried dependency where each iteration depends on the value of `Sn` from the previous iteration, preventing straightforward parallelization. The recurrence `Sn *= up` creates a dependency chain that must be broken to achieve parallel execution.

Initially I tried using `firstprivate` and `lastprivate`, but I failed to achieveperformance improvements. Then I tried closed-form approach computing `opt[n] = Sn_initial * pow(up, n)` at every iteration, but it was *extremely* slower than sequential execution.

In the next approach, I partitioned the iterations evenly among threads, with each thread computing its starting value using `pow()` once, then iterating sequentially for their chunk

```
double Sn_initial = Sn;
#pragma omp parallel
{
  // chunk the iterations evenly among the threads ...

  // One pow() call per thread to initialize
  double thread_Sn = Sn_initial * pow(up, start_index);

  for (int n = start_index; n < end_index; ++n) {
    opt[n] = thread_Sn;
    thread_Sn *= up;
  }
}
```

This implementation achieves near-linear speedup up to 16 threads (15× speedup on 16 threads), calling `pow()` only once per thread rather than per iteration. The parallel result differs minimally

from sequential (485165097.62503749 vs. 485165097.62511122, I believe due to the `pow()` operations. I accepted this negligible error giving more priority to the parallel efficiency.

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