Università della Svizzera italiana

Institute of Computing CI

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

Student: Metehan Dündar Discussed with: Only TAs

# Solution for Project 2

This project will introduce you to parallel programming using OpenMP.

## 1. Parallel reduction operations using OpenMP [10 points]

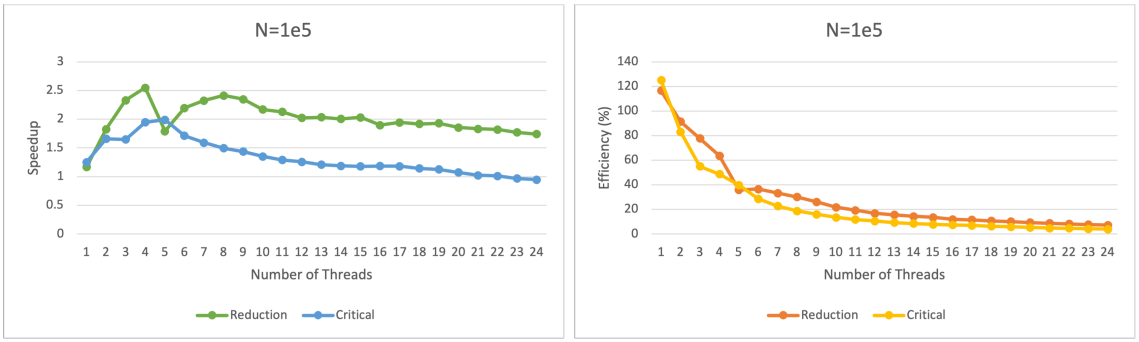


Figure 1: Parallel speedup and efficiency for dot products, $N = 100,000$.

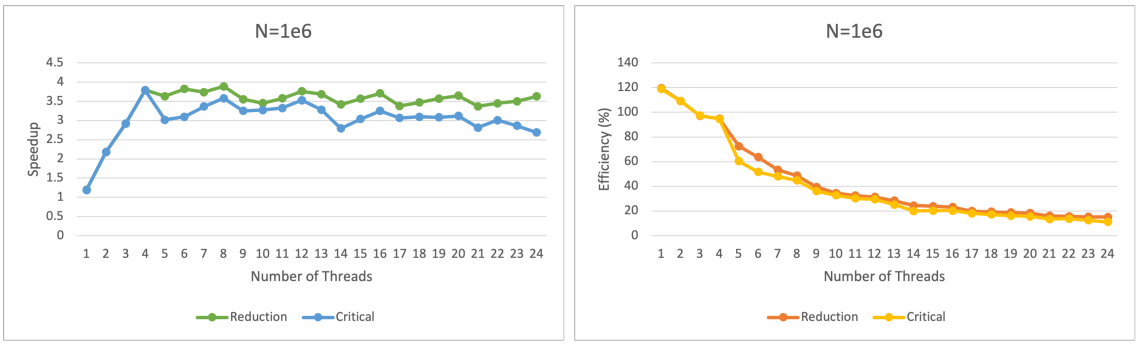


Figure 2: Parallel speedup and efficiency for dot products, $N = 1,000,000$.

- Figure 1 - 4 depict the results. The performance scalings of the two versions are comparable since the OpenMP implementations of `reduction` and `critical` are almost equal.
- For a parallel job, we can calculate the speedup $S$ and the efficiency $E$ by comparing the run-time on one core $T_1$ and on $n$ cores $T_n$.

$$S = \frac{T_1}{T_n}, E = \frac{100 \cdot T_1}{n \cdot T_n}$$

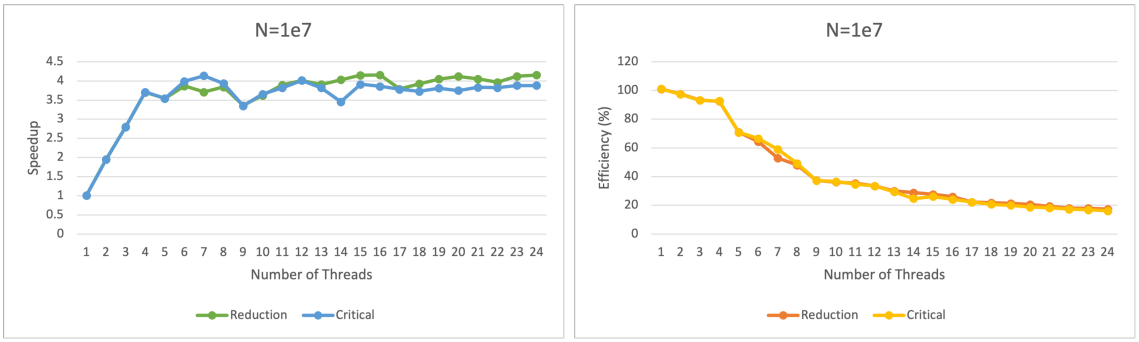


Figure 3: Parallel speedup and efficiency for dot products, $N = 10,000,000$.

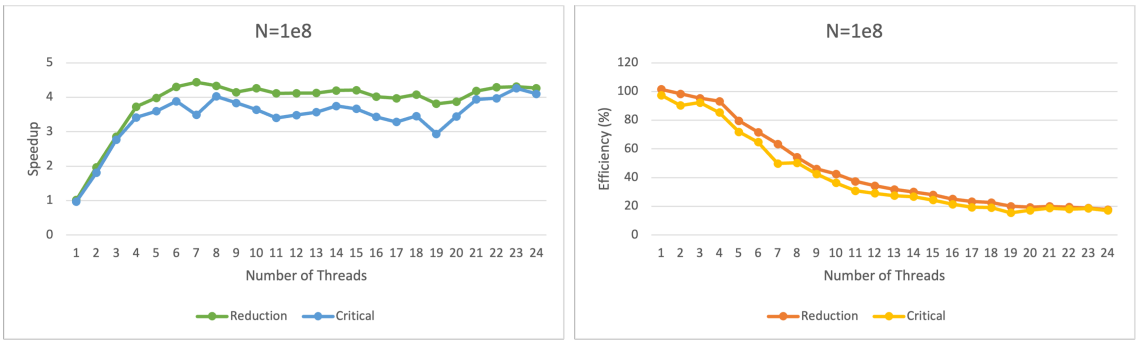


Figure 4: Parallel speedup and efficiency for dot products, $N = 100,000,000$.

- For $N = 100,000$, parallel efficiency is quite poor. Because the working set is tiny, the overhead of parallelization consumes a considerable percentage of run-time and degrades performance.
- For $N = 1,000,000$, the working set is around 8MB. When the number of threads is high enough, it can fit inside the L3 cache and even the L2 cache of each thread. Aside from the large cache bandwidth, the overhead of parallelization is modest when the issue size is considered. In this situation, parallel efficiency is high.
- For $N = 10,000,000, N = 100,000,000$, because the working set size exceeds the capacity of the cache, it must be read from and written to main memory. Because all threads share the same path to main memory, performance is constrained by the memory bus's bandwidth.
- This explains why the performance scaling trend of the code with three distinct issue sizes is identical.
- To conclude, $N = 1,000,000$, a parallelized version is advantageous since the parallelization overhead is small and numerous threads do not compete for the fixed memory bandwidth.

## 2. The Mandelbrot set using OpenMP [30 points]

- The results are shown in Figure 5. Programs of different image sizes show almost identical performance scaling pattern when it is tested.
- In serial version, the program performs about $3\times10^8$ iterations per second and thus 2400MFlops/s.
- In dynamic schedule version (Figure 5), the workload distribution among threads is in balance, and thus the performance scaling is perfect.

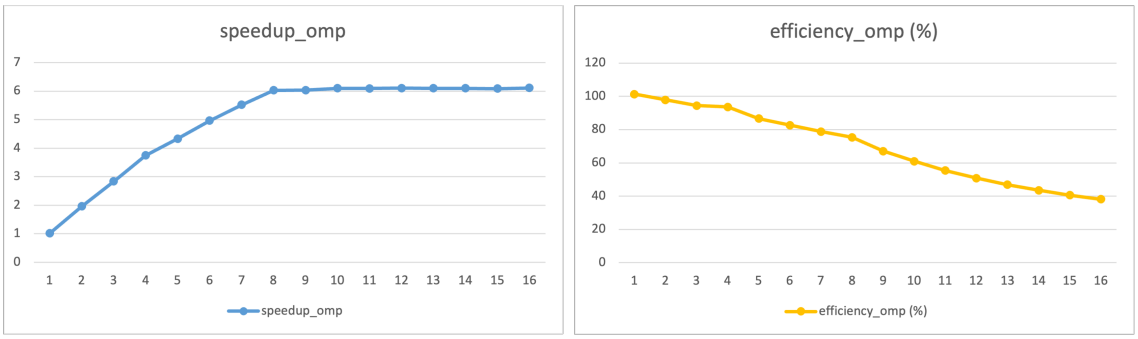


Figure 5: Parallel speedup and efficiency with dynamic schedule for Mandelbrot set.

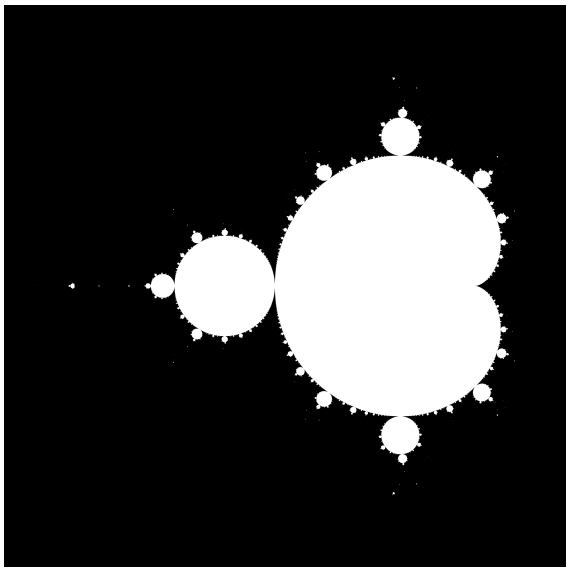

Figure 6: The Mandelbrot set.

# 3. Bug hunt [15 points]

## 3.1. bug1

The problem with the supplied code is that the variable `tid` is declared as private but is utilised outside of the parallel zone, where it is no longer valid. To resolve this issue, `tid` should be declared outside the parallel area so that its value is retained after the parallel for construct.

## 3.2. bug2

The variable `total` is defined outside the parallel zone and shared by all threads, potentially resulting in a race scenario. `total` should be specified as a private variable to ensure that each thread gets its own copy. It is possible to achieve this by including a `private` clause in the pragma directive for the parallel area. By including the `private(tid, total)`clause, you guarantee that each thread has its own private copy of `tid` and `total`, avoiding race situations and data discrepancies.

## 3.3. bug3

The problem comes with where the obstacles are placed. The `nowait` phrase in the `#pragma omp sections nowait` directive allows threads to continue without waiting for other threads to finish their sections. `#pragma omp barrier` is introduced after each section's computation to guarantee that the threads synchronise before advancing. This guarantees that all threads complete their parts before proceeding to the next phase.

### 3.4. bug4

The problem with the supplied code is that the array `a[N][N]` is too huge for the stack. When a huge array like this is required, it is best to allocate it on the heap using dynamic memory allocation routines like `malloc` and `free`. This should prevent the stack segmentation fault produced by a big array.

### 3.5. bug5

The problem in the code is a classic case of a deadlock caused by a cyclic locking pattern. The stalemate arises because each thread acquires locks in a different sequence.

If thread 1 obtains `locka` and then attempts to acquire `lockb`, and thread 2 acquires `lockb` and then attempts to acquire `locka`, they will be waiting for each other to release the locks, resulting in a deadlock.

To resolve this issue, make sure that all threads acquire locks in the same sequence. For example, in both parts, you might opt to obtain `locka` before `lockb`. This modification guarantees that the locking sequence is constant and avoids the risk of a stalemate.

## 4. Parallel histogram calculation using OpenMP [15 points]

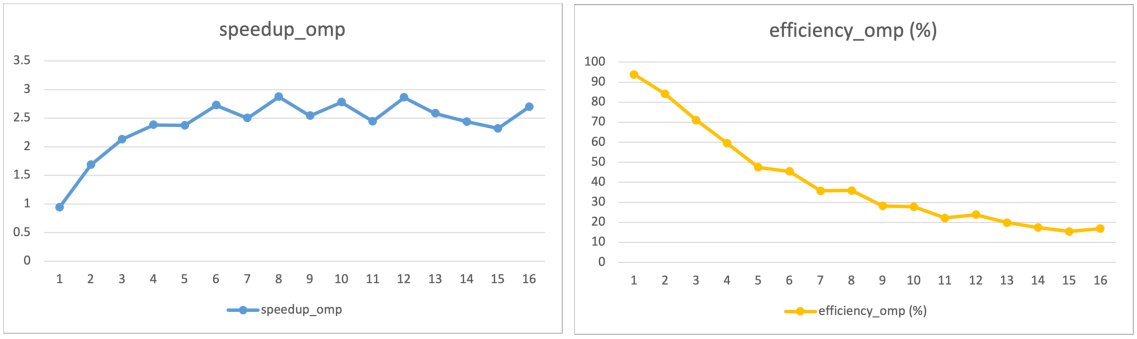


Figure 7: Parallel speedup and efficiency for histogram calculation.

- Figure 7 depicts the performance scaling. Data from the 1-thread version demonstrates the overhead caused by OpenMP when compared to the original.
- The for-loop across the vecsize might then be performed concurrently. It is looping over the number of bins in a key region to aggregate the previously computed findings. Because the number of bins is minimal in relation to the vecsize, this worked successfully.
- The speedup of odd-number threads is less than that of neighbouring even-number threads. However, there is still an overall workload imbalance, which leads to low parallel efficiency.

## 5. Parallel loop dependencies with OpenMP [15 points]

- Adding parallelization to the code considerably enhanced the code's computing performance by making the process quicker. Based on the original/serial points plotted, the result was created in a specific length of time (approximately 5 seconds).
- However, by introducing parallel programming into this process, the logic of the programme allows us to run distinct loop iterations concurrently, which clearly speeds up the process as the number of threads rises.
- The open mp framework parallelizes loops by spawning threads and dividing loop iterations among them. This utility has accelerated the computation of the blocks.

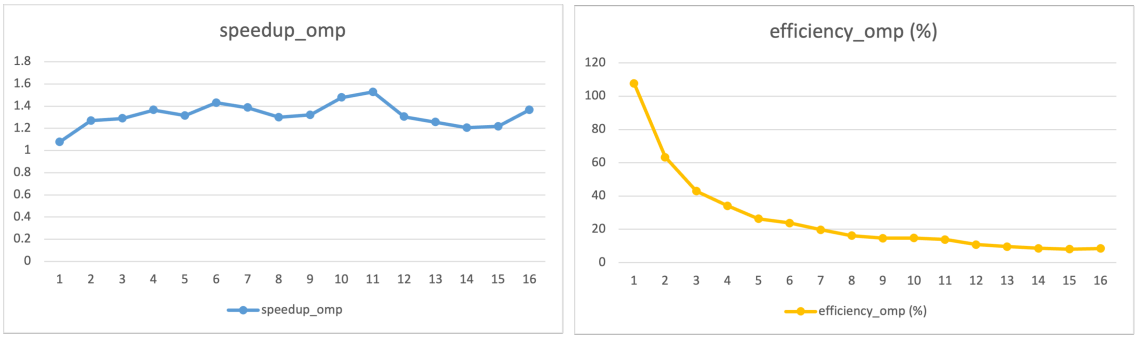


Figure 8: Parallel speedup and efficiency for loop dependencies.

# 6. Task: Quality of the Report [15 Points]