Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Student: Pranjal Mishra   Discussed with: Panagiotis Minos, Yanki Kiran, Yannick Ramic

Solution for Project 2

Due date: 25 March 2024, 23:59

1. Computing π with `OpenMP` [20 points]

To start with, after running with `pi_serial` code, the execution time was $T_{\text{serial}} = 0.00175748$ secs. Then the task was to paralleize the serial code and perform the strong and weak scaling studies.

1.1. Speedup

Speedup ratio is a ration of T_1 over T_1 , where T_1 is the time needed to execute serial application, T_{N} is the time needed to execute the application with a certain number of processor.

$$S = \frac{T_1}{T_N}$$

1.2. Strong and weak scaling studies

In a strong scaling, the problem size of the program stays fixed but the number of processing elements are increased. By this we use problem size of 100000000. In strong scaling a program is considered to scale linearly if the speed up is equal with the number processing used. As we can see, it is difficult to achieve linear speed up because of overhead increases together with the growth of number of processes used.

In weak scaling, the problem size assigned keep increasing along with the growth of threads number. Therefore linear scaling should be achieved if the run time stays constant while the number of workload and threads are doubled. However in this case, we are not getting the constant value of run time

```
#pragma omp parallel
  {
    double sum_private = 0.;
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int i_beg = tid * N / nthreads;
    int i_end = (tid + 1) * N / nthreads;
    for (int i = i_beg; i < i_end; ++i)
    {
      double x = (i + 0.5) * h;
      sum_private += 4.0 / (1.0 + x * x);
    }

#pragma omp critical
    sum += sum_private;
  }
```

The reduction clause is applied to the variable sum, indicating that each thread will have a private copy of sum, and at the end of the parallel region

```
#pragma omp parallel for reduction(+ : sum)
  for (int i = 0; i < N; ++i) {
    double x = (i + 0.5) * h;
    sum += 4.0 / (1.0 + x * x);
  }
```

Within the parallel region, each thread calculates a partial sum (sum_private) independently. The critical directive ensures that only one thread at a time can execute the critical section, which in this case is the addition of each thread's partial sum to the global sum (sum). This prevents race conditions and ensures the correctness of the final result.

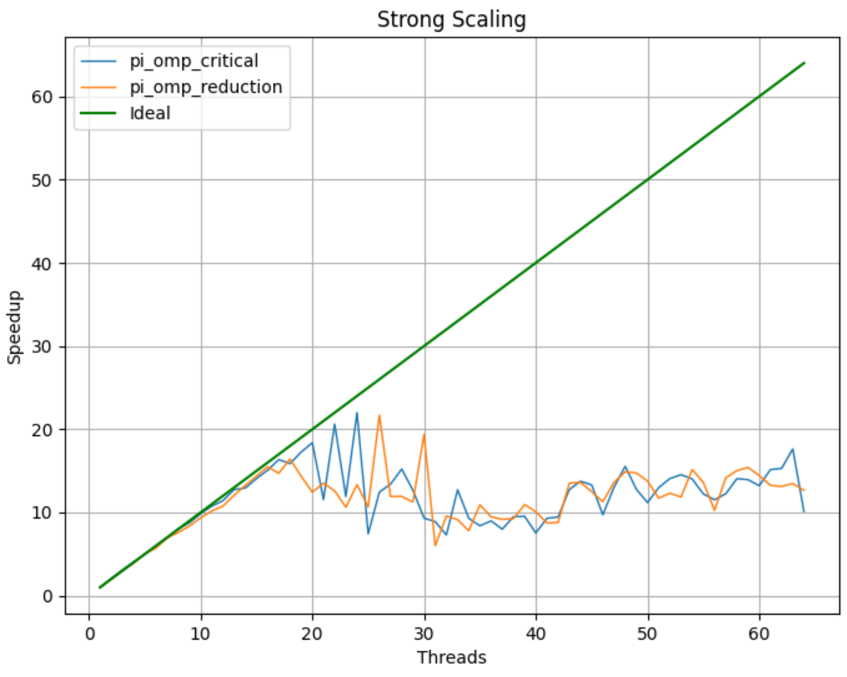

Figure 1: Strong scaling analysis

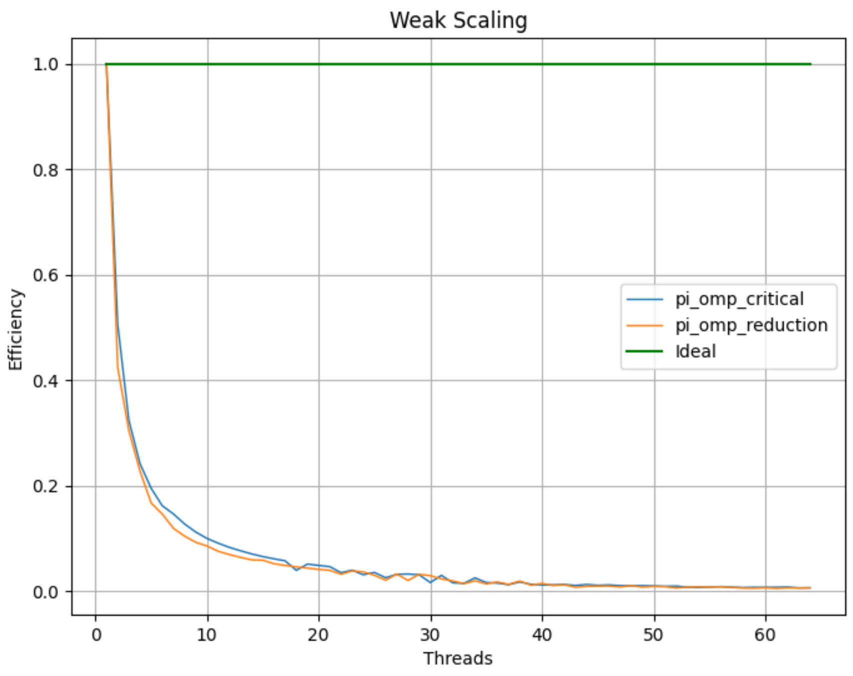

Figure 2: Strong scaling analysis

2. The Mandelbrot set using `OpenMP` [20 points]

First, TODO code for the `mandel_seq` is here:

```
do {
       x_temp = x*x - y*y + cx;
       y = 2*x*y + cy;
       x = x_temp;
       x2 = x * x;
       y2 = y * y;
       n++;
} while ((x2 + y2 < 2*2) && n < MAX_ITERS);

#pragma omp atomic
       nTotalIterationsCount += n;
```

The obtained visualization from the serial code can be seen here

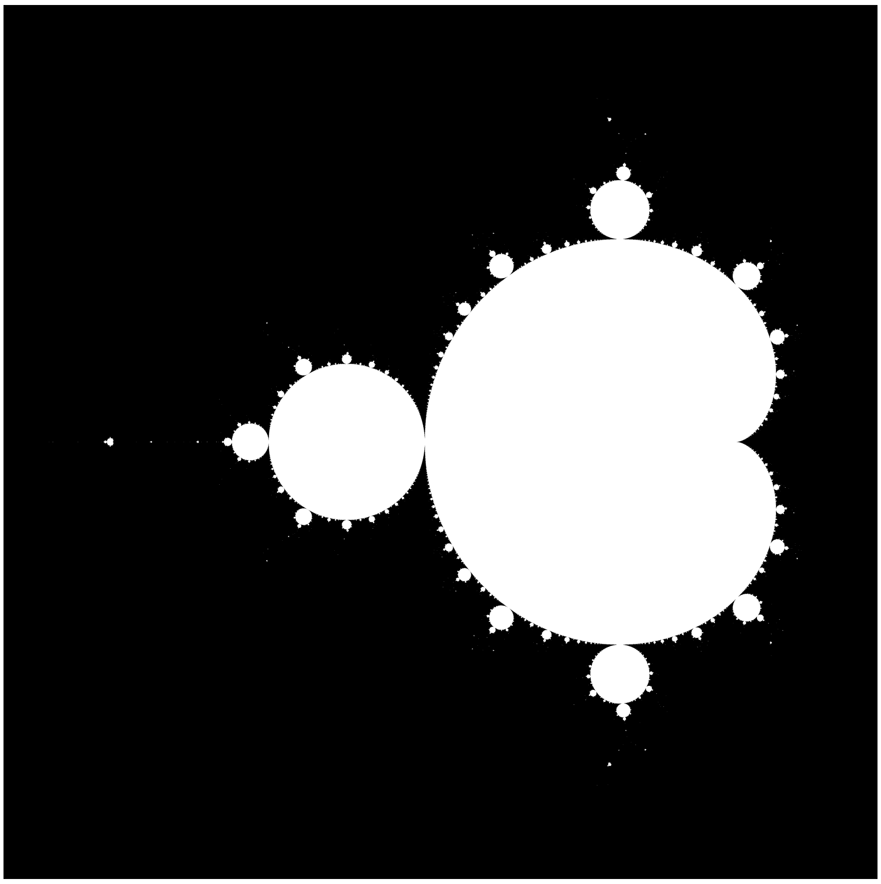

Figure 3: The Mandelbrot set

Moreover, the output of the code (with `sbatch run_scaling.sh`) is here

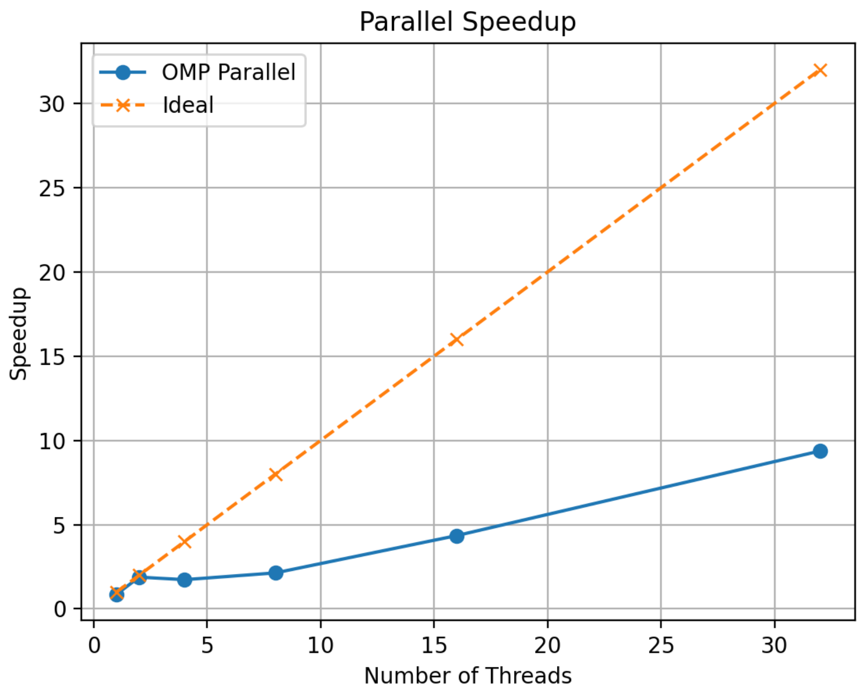

Figure 4: Strong scaling study: Speedup vs Number of Threads

```
Mandel Serial Implementation:
Total time:                   331.241 seconds
Image size:                   4096 x 4096 = 16777216 Pixels
Total number of iterations: 113624527400
Avg. time per pixel:          1.97435e-05 seconds
Avg. time per iteration:      2.91522e-09 seconds
Iterations/second:            3.43027e+08
MFlop/s:                      2744.22
```

Dynamic scheduling is implemented to accommodate variations in thread execution times, as certain threads may complete their tasks more rapidly, necessitating reassignment of new iterations. We can see that the parallel mandelbrot is roughly ten times faster.

```
Mandel Parallelized with 32 Threads:
Total time:                   35.3486 seconds
Image size:                   4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:          2.10694e-06 seconds
Avg. time per iteration:      3.11024e-10 seconds
Iterations/second:            3.21519e+09
MFlop/s:                      25721.5
```

The results we obtained help us create the strong scaling analysis shown in Figures 6 and 7. These figures clearly show that our code benefits a lot from parallelization. When we increase the number of threads, the speedup improvement becomes less noticeable, which means the efficiency drops a bit. However, even with this decrease, the efficiency is still quite good, showing that parallel processing is really helpful in speeding up our code.

3. Bug hunt [10 points]

3.1. Bug 1

Moving the `tid` variable inside the loop to ensure it's accessible within each iteration, fixing the bug identified in the original version.

3.2. Bug 2

In the provided code, the total variable is defined within the parallel region, making it private to each thread. However, since each thread is updating its own private copy of total, the final result will be the sum of all the individual thread's total values.

3.3. Bug 3

This code is blocked by the barriers. To fix this the barriers could be removed so that the output is given whenever each thread finishes instead of just the first two.

3.4. Bug 4

This code throws a segmentation fault. This is caused by the stack size being too small and could be fixed by modifying the omp stack size variable.

3.5. Bug 5

This code deadlocks. To fix this the threads could initialize and then wait for the other to finish, producing a program in which one thread can acquire both locks.

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

The relevent code snippet for `hist_omp.c`

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

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

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

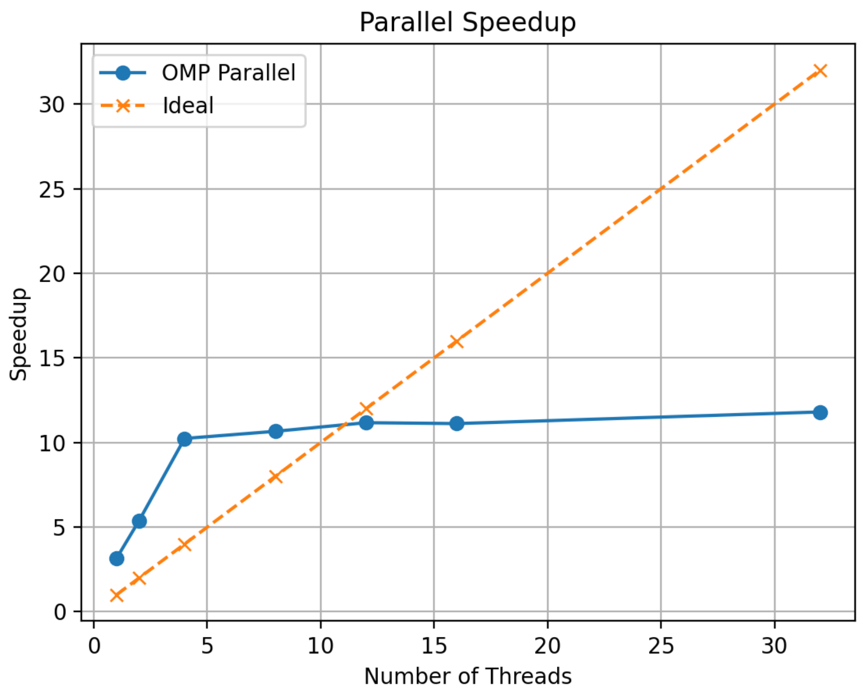

Figure 5: Strong scaling study: Speedup vs Number of Threads

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

The code was parallelized by adding a pragma omp for with first private taking in the base and up variables from the global and a last private updating the S_N value at the end of the loop. The output is

```
Running the Sequential Program
Sequential RunTime:  7.662206 seconds
Final Result Sn    :  485165097.62511122
Result ‖opt‖^2_2 :  5884629305179574.000000

---------------------------------------------
Running with OMP_NUM_THREADS=2
Parallel RunTime   :  0.468264 seconds
Final Result Sn    :  7.3890560091157873
Result ‖opt‖^2_2 :  13.399537

---------------------------------------------
Running with OMP_NUM_THREADS=4
Parallel RunTime   :  0.285648 seconds
Final Result Sn    :  7.3890560091184163
Result ‖opt‖^2_2 :  13.399537

---------------------------------------------
Running with OMP_NUM_THREADS=8
Parallel RunTime   :  0.168647 seconds
Final Result Sn    :  7.3890560091177599
Result ‖opt‖^2_2 :  13.399537

---------------------------------------------
Running with OMP_NUM_THREADS=16
Parallel RunTime   :  0.116720 seconds
Final Result Sn    :  7.3890560091169588
Result ‖opt‖^2_2 :  13.399537

---------------------------------------------
Running with OMP_NUM_THREADS=32
Parallel RunTime   :  0.101026 seconds
Final Result Sn    :  7.3890560091173505
Result ‖opt‖^2_2 :  13.399537

---------------------------------------------
Running with OMP_NUM_THREADS=64
Parallel RunTime   :  0.078584 seconds
Final Result Sn    :  7.3890560091172963
Result ‖opt‖^2_2 :  13.399537

---------------------------------------------
```

6. Quicksort using `OpenMP` tasks [20 points]

The `pragma omp task` directive creates two tasks in parallel to execute the quicksort function on different portions of the array concurrently.

The `firstprivate` clause ensures that each task has its own private copy of the specified variables. In this case, the data array is marked as `firstprivate` , meaning that each task will work on a private copy of the array.

By parallelizing the partitioning step of the Quicksort algorithm using tasks, this code leverages task-level parallelism to potentially improve the efficiency of sorting large arrays on multi-core systems. Each task independently sorts a portion of the array, allowing for more parallelism and potentially faster execution.

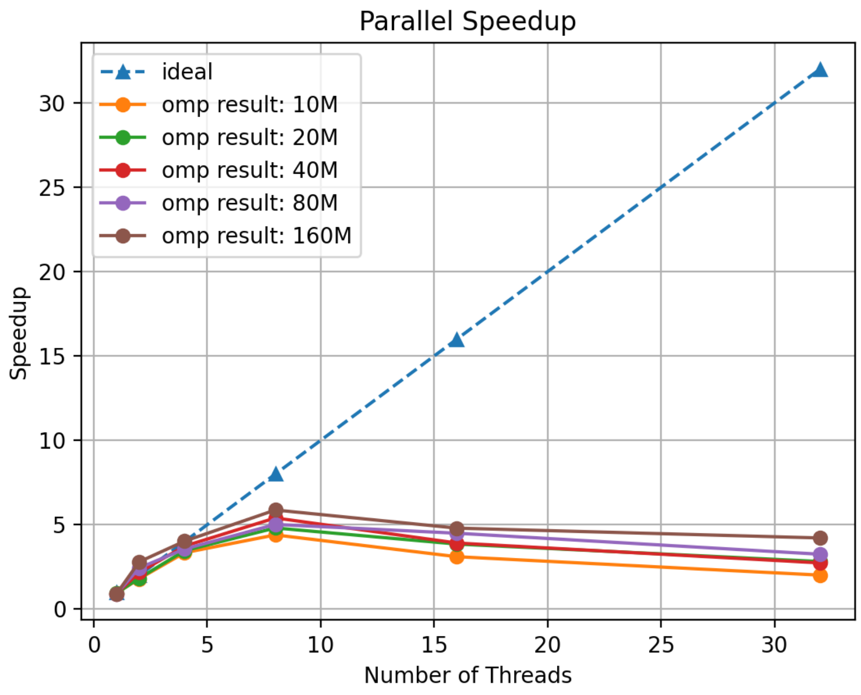

Figure 6: Strong scaling study: Speedup vs Number of Threads