Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Student: Yannick Ramic Discussed with: Carla Lopez

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 [20 points]

The very first exercise task, comprises of two subtasks, where the goal of this omp introductory exercise is to implement two versions of parallelized code. The underlying serial implementation is already given and provides the template on how to compute π numerically, by approximating the integral with the midpoint rule, as explained in the exercise sheet. To speed up this task we should implement 2 different parallelized versions, one with the *critical* directive and the other one with the *reduction* clause. Besides the lecture notes the information to solve this task was taken from the introductory book by Hager and Wellein [1].

As described in [1], OpenMP provides a more elegant way, compared to an implementation with a critical region, in order to update a variable. Thus, I will start with explaining the advantage of using the *Reduction* clause. The *Reduction* clause expects a definition of the used operator, which is in our case the + operator because we accumulate values at each iteration, by simply summing over them. Besides the operator we have to define our target value, which is in this case *sum*. As a result, the specified variable will be privatized and initialized with a sensible initial value. In the end, obviously we need to synchronize each thread again. Thus, in the end all partial results in this case will be accumulated and stored in the variable sum. [1]

Another possibility, also described in [1], is to define critical regions, in order to achieve a fast parallelized version of a serial code implementation. Critical regions can become necessary, in order to avoid concurrent overwriting and sharing a variable. This problem is called race condition and can be avoided by allowing only one thread executing code in this defined code part. The order in which threads access this region of code is undefined but also here are possibilities if necessary. At this point it's important to mention that a wrong use of the *critical* directive can lead to another problem called deadlocks, where one or more threads become inactivate and wait for resources that become never available. This problem typically occurs when two or more critical regions are badly arranged. For instance, when a thread encounters another critical directive inside a critical region, it will be blocked forever. In order to avoid this, OpenMP also provides the solution of naming the critical region, which offers the opportunity to distinguish each region. [1]

```
# Critical Parallelization Version

#pragma omp parallel
{
    double partial_sum = 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;
    partial_sum += 4.0 / (1.0 + x*x);
    }
    #pragma omp critical
        sum += partial_sum;
} // Parallelization ends here!
pi = sum*h;
```

```
# Reduction Parallelization Version

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

Figure 1: Critical and Reduction Based Parallelization Version

Before presenting the strong and weak scaling analysis, the underlying code for both versions can be found in 1. As mentioned in the last task it's necessary to benchmark the performance of each parallelized version. Weak scaling asks the question, how well does the prallel fraction scale among p processors? code.

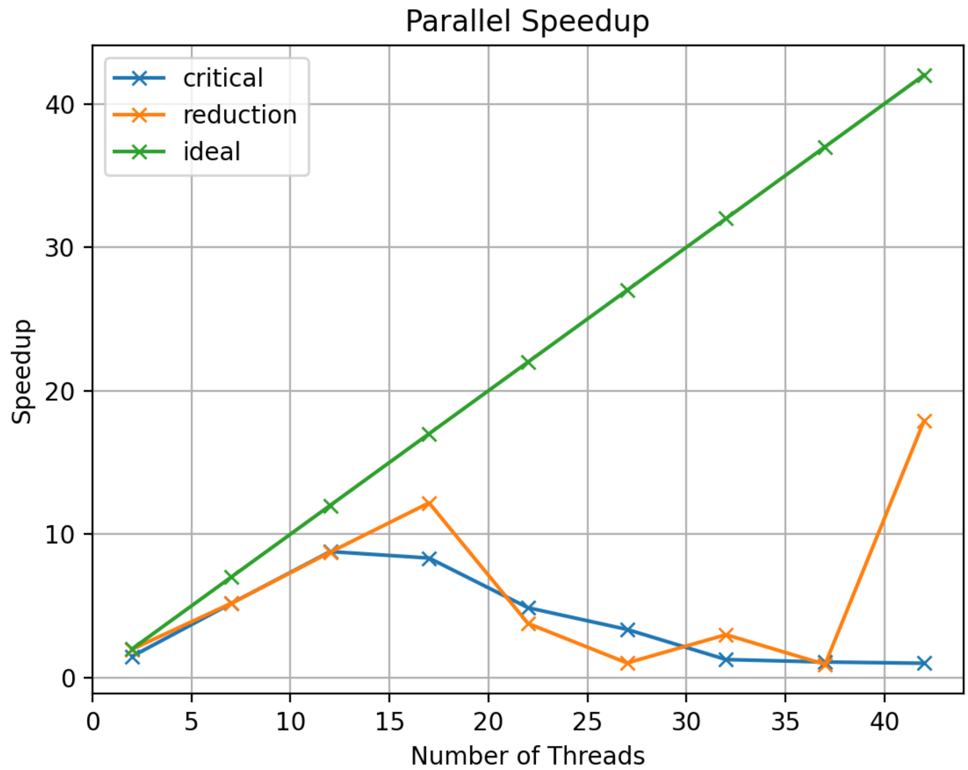

Figure 2: Strong Scaling Analysis: Parallel Speedup

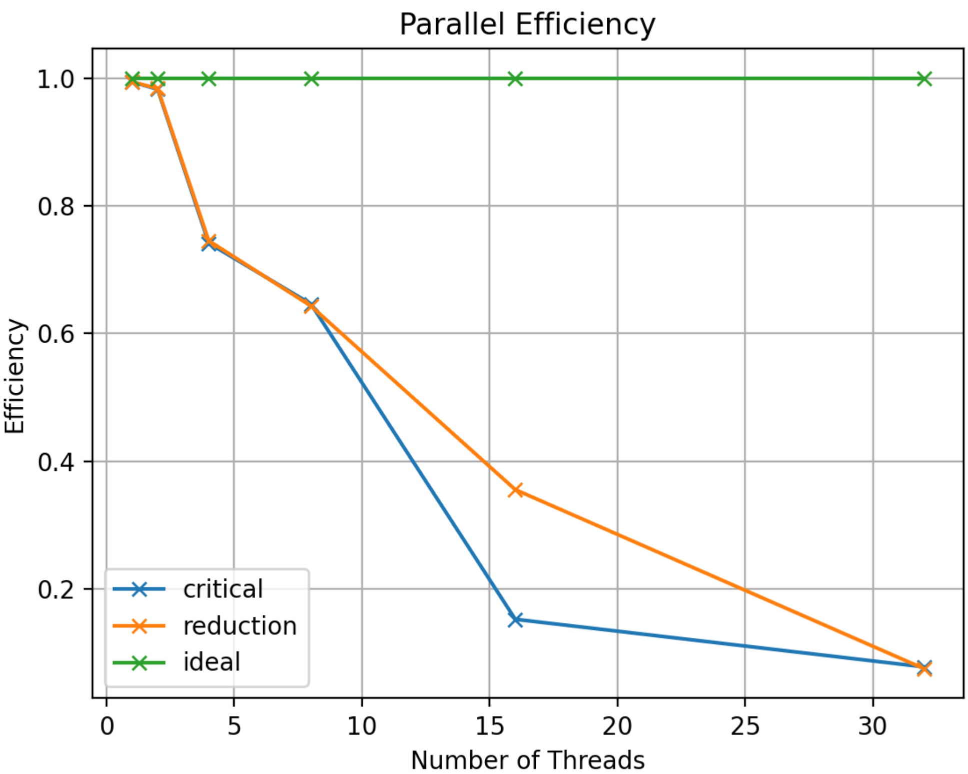

Figure 3: Weak Scaling Analysis: Parallel Efficiency

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

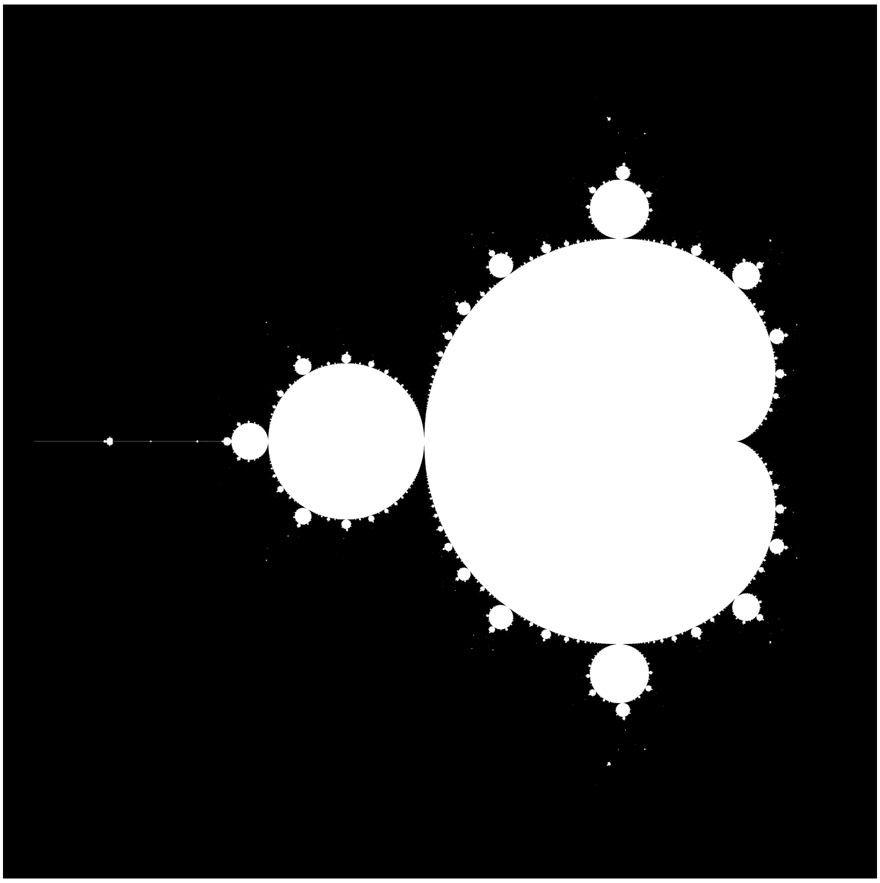

Figure 4: Quicksort algorithm, retrived from [2]

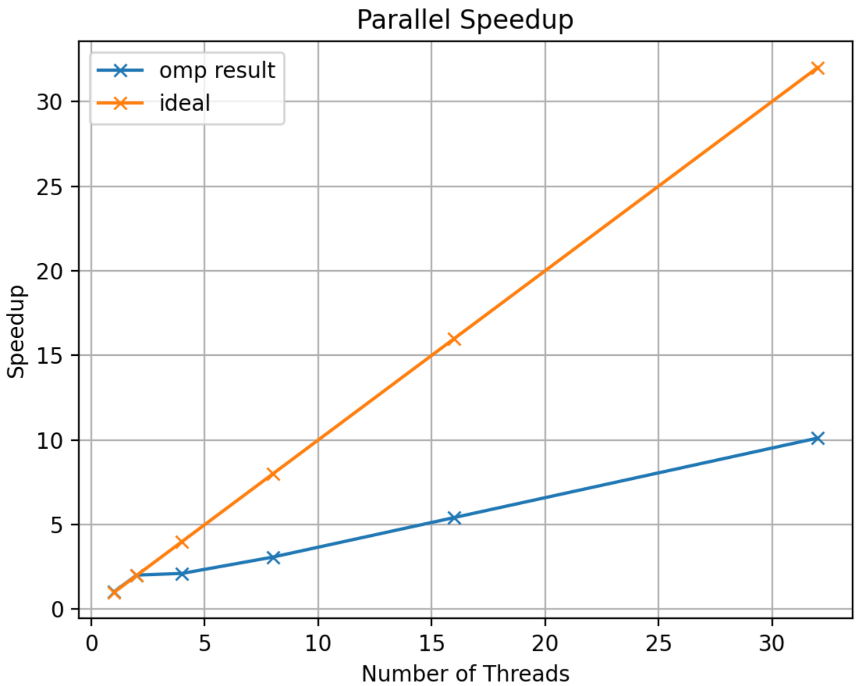

Figure 5: Quicksort algorithm, retrived from [2]

3. Bug hunt [10 points]

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

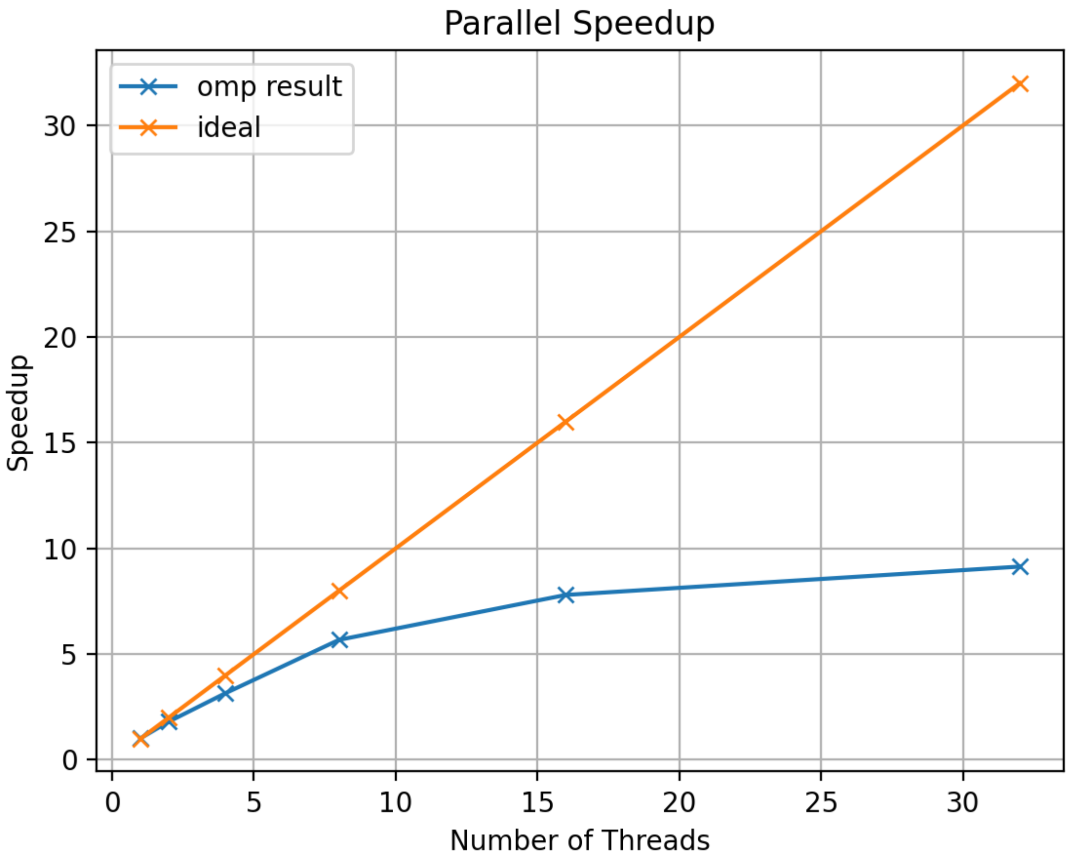

Figure 6: Quicksort algorithm, retrived from [2]

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

```
for (n = 0; n <= N; ++n) {
    opt[n] = Sn;
    Sn *= up;
}
```

Figure 7: Loop Dependencies Problem Description

```
#pragma omp parallel shared(opt) private(n)
{
#pragma omp for firstprivate(lastn) lastprivate(Sn)
    for (n = 0; n <= N; ++n) {
        if (lastn == n - 1) {
            // Use the fast version!
            Sn *= up;
        } else {
            // Use the slow version!
            // Note that S0 = up!
            Sn = up * pow(up, n);
        }
        opt[n] = Sn;
        // Update lastn!
        lastn = n;
    }
} // End OMP
```

Figure 8: Parallelized Code Snippet

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

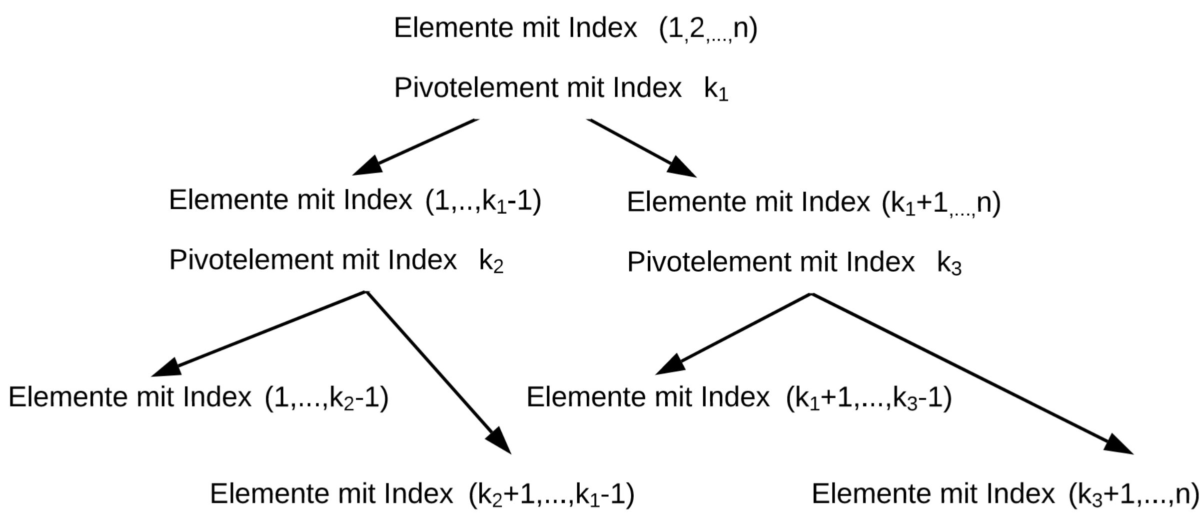

Figure 9: Quicksort algorithm, retrived from [2]

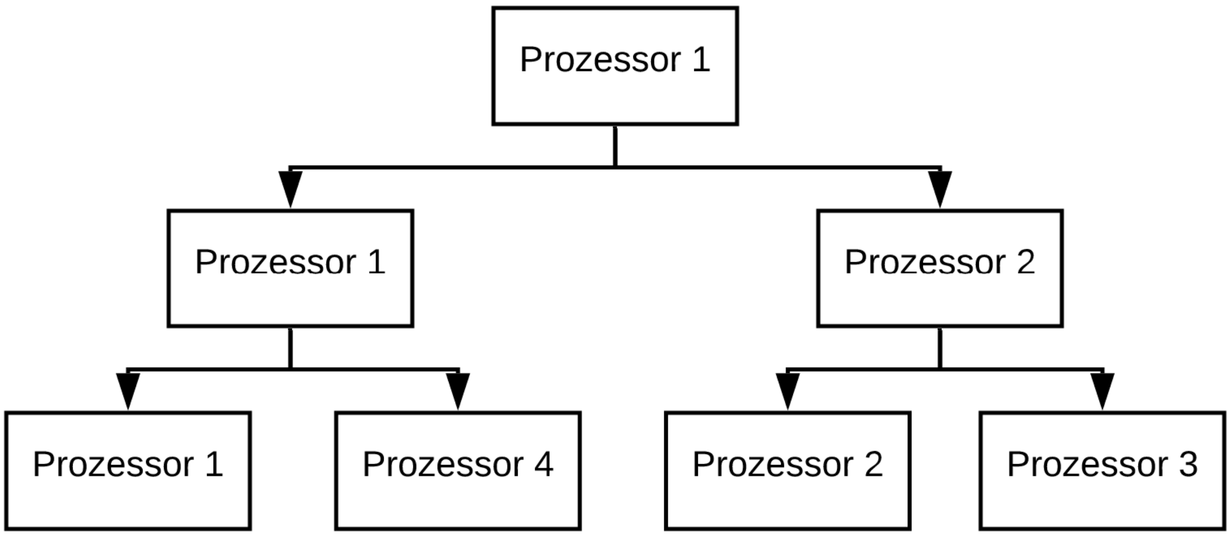

Figure 10: Parallelized quicksort algorithm, retrived from [2]

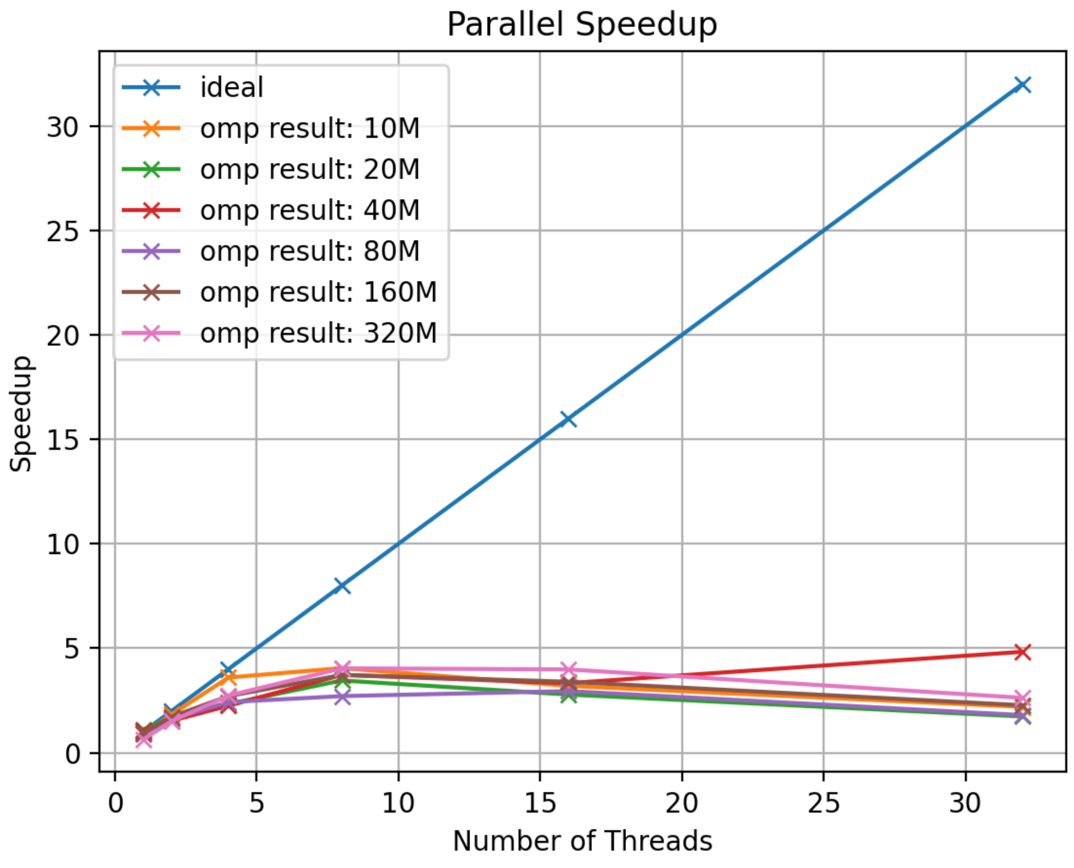

Figure 11: Quicksort algorithm, retrived from [2]

7. Appendix

In this section all results and the complete output data can be found.

```
Sequential Program      OMP_NUM_THREADS=8       OMP_NUM_THREADS=32
dist[0]=93              dist[0]=93              dist[0]=93
dist[1]=3285            dist[1]=3285            dist[1]=3285
dist[2]=85350           dist[2]=85350           dist[2]=85350
dist[3]=1260714         dist[3]=1260714         dist[3]=1260714
dist[4]=10871742        dist[4]=10871742        dist[4]=10871742
dist[5]=54586161        dist[5]=54586161        dist[5]=54586161
dist[6]=159818688       dist[6]=159818688       dist[6]=159818688
dist[7]=273378694       dist[7]=273378694       dist[7]=273378694
dist[8]=273376196       dist[8]=273376196       dist[8]=273376196
dist[9]=159818440       dist[9]=159818440       dist[9]=159818440
dist[10]=54574824       dist[10]=54574824       dist[10]=54574824
dist[11]=10876078       dist[11]=10876078       dist[11]=10876078
dist[12]=1261215        dist[12]=1261215        dist[12]=1261215
dist[13]=85046          dist[13]=85046          dist[13]=85046
dist[14]=3397           dist[14]=3397           dist[14]=3397
dist[15]=77             dist[15]=77             dist[15]=77
Time: 2.65926 sec       Time: 0.468622 sec      Time: 0.290979 sec
```

Figure 12: Parallelized Code Snippet

```
Mandel Serial Implementation:
Total time:                 339.969 seconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113624527400
Avg. time per pixel:        2.02637e-05 seconds
Avg. time per iteration:    2.99204e-09 seconds
Iterations/second:          3.34221e+08
MFlop/s:                    2673.77
---------------------------------------------
Mandel Parallelized with 1 Thread:
Total time:                 331.488 seconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:        1.97582e-05 seconds
Avg. time per iteration:    2.91669e-09 seconds
Iterations/second:          3.42855e+08
MFlop/s:                    2742.84
---------------------------------------------
Mandel Parallelized with 2 Threads:
Total time:                 168.342 seconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:        1.0034e-05 seconds
Avg. time per iteration:    1.4812e-09 seconds
Iterations/second:          6.75128e+08
MFlop/s:                    5401.02
---------------------------------------------
Mandel Parallelized with 4 Threads:
Total time:                 160.61 seconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:        9.57308e-06 seconds
Avg. time per iteration:    1.41317e-09 seconds
Iterations/second:          7.07631e+08
MFlop/s:                    5661.05
---------------------------------------------
Mandel Parallelized with 8 Threads:
Total time:                 110.289 seconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:        6.57374e-06 seconds
Avg. time per iteration:    9.70408e-10 seconds
Iterations/second:          1.03049e+09
MFlop/s:                    8243.96
---------------------------------------------
Mandel Parallelized with 16 Threads:
Total time:                 62.7268 seconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:        3.73881e-06 seconds
Avg. time per iteration:    5.51918e-10 seconds
Iterations/second:          1.81186e+09
MFlop/s:                    14494.9
---------------------------------------------
Mandel Parallelized with 32 Threads:
Total time:                 33.6183 seconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:        2.00381e-06 seconds
Avg. time per iteration:    2.958e-10 seconds
Iterations/second:          3.38067e+09
MFlop/s:                    27045.3
```

Figure 13: Parallelized Code Snippet

```
Running the Sequential Program
Sequential RunTime:  0.903069 seconds
Final Result Sn    :  7.3890560830036707
Result ||opt||^2_2 :  13.399537

Running with OMP_NUM_THREADS=1
Parallel RunTime  :  0.913387 seconds
Final Result Sn    :  7.3890560091131112
Result ||opt||^2_2 :  13.399537

Running with OMP_NUM_THREADS=2
Parallel RunTime  :  0.520727 seconds
Final Result Sn    :  7.3890560091157873
Result ||opt||^2_2 :  13.399537

Running with OMP_NUM_THREADS=4
Parallel RunTime  :  0.300407 seconds
Final Result Sn    :  7.3890560091184163
Result ||opt||^2_2 :  13.399537

Running with OMP_NUM_THREADS=8
Parallel RunTime  :  0.174071 seconds
Final Result Sn    :  7.3890560091177599
Result ||opt||^2_2 :  13.399537

Running with OMP_NUM_THREADS=16
Parallel RunTime  :  0.088380 seconds
Final Result Sn    :  7.3890560091169588
Result ||opt||^2_2 :  13.399537

Running with OMP_NUM_THREADS=32
Parallel RunTime  :  0.081424 seconds
Final Result Sn    :  7.3890560091173505
Result ||opt||^2_2 :  13.399537
```

Figure 14: Parallelized Code Snippet

```
Running the Sequential Program
Size of dataset: 10000000, elapsed time[s] 2.182783e+00
Running with OMP_NUM_THREADS=1
Size of dataset: 10000000, elapsed time[s] 2.343903e+00
Running with OMP_NUM_THREADS=2
Size of dataset: 10000000, elapsed time[s] 1.195091e+00
Running with OMP_NUM_THREADS=4
Size of dataset: 10000000, elapsed time[s] 6.066376e-01
Running with OMP_NUM_THREADS=8
Size of dataset: 10000000, elapsed time[s] 5.392190e-01
Running with OMP_NUM_THREADS=16
Size of dataset: 10000000, elapsed time[s] 6.830470e-01
Running with OMP_NUM_THREADS=32
Size of dataset: 10000000, elapsed time[s] 9.954015e-01
```

Figure 15: Parallelized Code Snippet

References

[1] Gerhard Wellein Georg Hager. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press: Taylor & Francis Group, Boca Raton, FL, 2011.

[2] Parallel quicksort, 2024. [online]. available: `https://de.wikipedia.org/wiki/Parallel_Quicksort`. accessed: 2024-03-21.