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. Thus, Wermelinger [2] describes weak scaling by asking the question: How well does the prallel fraction scale among p processors? According to [2] weak scaling can be summarized with a fixed execution time and identifying, how much longer it takes for the problem without parallelism. Whereas strong scaling has a fixed problem size and answers the question, how much does parallelism reduce the execution time of a problem.

$$S_p = \frac{w/t_p}{w/t_1} = \frac{t_1}{t_p}$$

The equation presented above describes the calculation of the speedup considering the strong scaling analysis, where w is the associated work or in other words the problem size, t_1 the time needed to complete the serial task and t_p the time necessary to complete the described work in a parallelized fashion. On the other hand, in the equation beneath, E_w represents the efficiency by using parallelism and this is considered as the weak scaling analysis. The big difference is that for the weak scaling we actually need to compute a solution for the serial implementation for different problem sizes. This is not necessary for the strong scaling, since the problem size is fixed. Thus, week scaling requires to compare the time it takes between the serial implementation for a specific problem size with the parallelized solution for the same problem size. From a starting value it is necessary to scale the problem up in size and with used threads. How this was done exactly for the underlying problem will become clear, by inspecting the provided output files. [1], [2]

$$S_p = \frac{p \cdot w/t_p}{w/t_1} = p \cdot \frac{t_1}{t_p}$$

$$E_w = \frac{S_p}{p} = \frac{t_1}{t_p}$$

The results from the provided codes, can be seen for both scaling analysis in the figures 2 and 3. It becomes evident from figure 2 that initially the speedup is close to the ideal line, but at some point the results become worse. There are several reasons for this phenomena. First, the problem size is still too small to analyse the problem properaly. Even, with a size of 10 million iterations the code only takes 0.18 seconds to run. Since creating the team of threads and the two operations fork and join also require some time, this will result in the fact that splitting up the task between different threads takes more time than just solving the problem in a sequential and naive way. This is true for both, the critical and the reduction version. Also due to the waiting of threads in the critical case, ideally the reduction clause should lead to better and faster results. Eventhough I took 100 experiments and averaged the time, it becomes evident from both figures 2 and 3, that the results for the reduction based solution zigzags around and this effect can't be seen. Mentionable is that at the last measurement in figure 2 for the reduction based solution there is a spike up and a massive increasement in speedup.

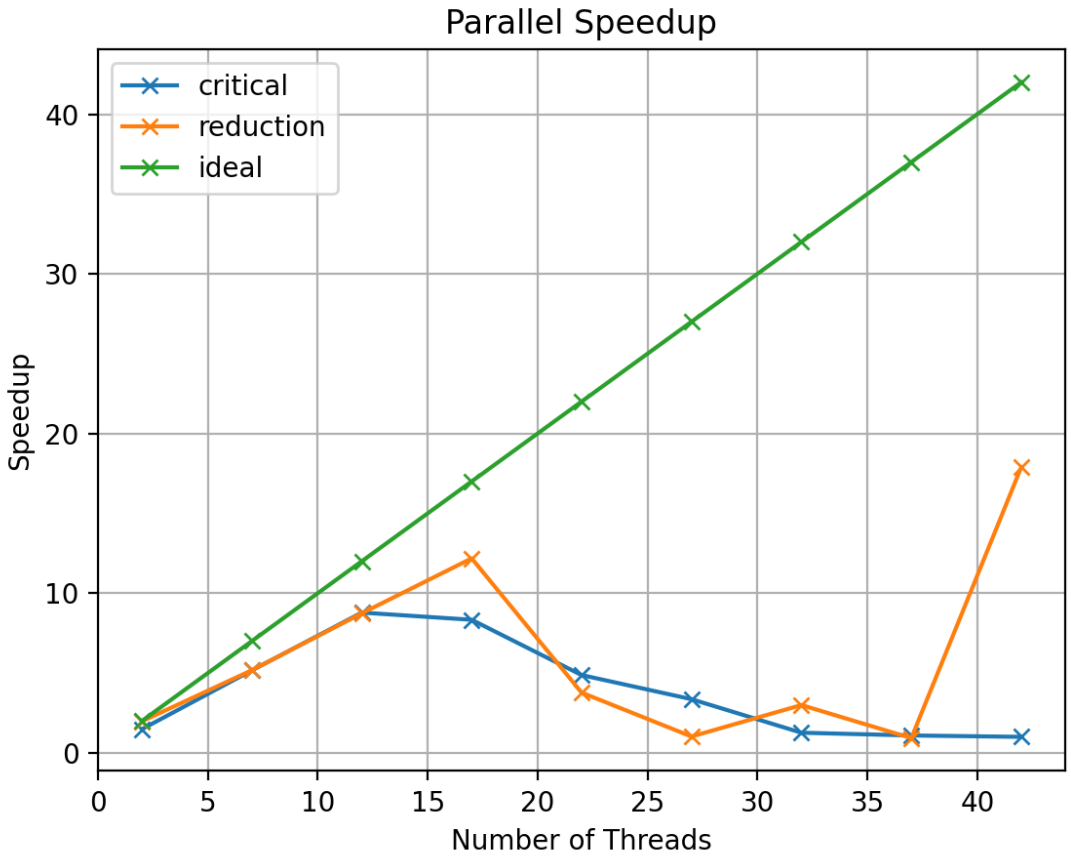

Figure 2: Strong Scaling Analysis: Parallel Speedup

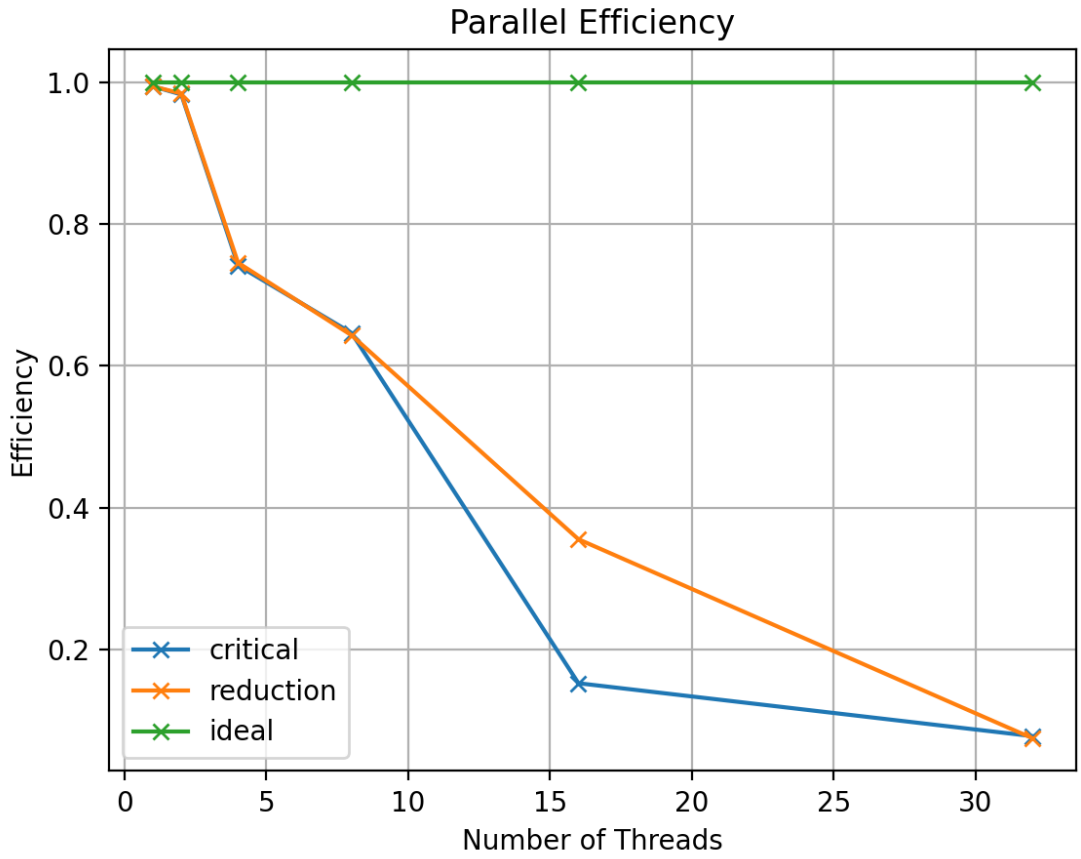

Figure 3: Weak Scaling Analysis: Parallel Efficiency

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

The goal of this task is to visualize the Mandelbrot set and achieve this in a parallelized fashion. The visualized result can be seen in figure 4. It becomes evident that on the left half plane there is a white line, which shouldn't be there compared to the sequential solution. The reason for this phenomena is simply double precision floating point errors that result in a few wrong results, where the underlying pixel doesn't have the value of 0, but some really small number instead.

Compared to Chapter 1 the results seem more "stable". In figure 5 we see a linear instead of a zigzagging behaviour, by increasing the number of threads for a fixed problem size. For the same reasons as explained already in Chapter 1, that the process to fork the master thread into a team of threads and in the end joining them back together into the master thread again requires some time itself and is not a "perfect" operation without any time loss. Hence, there will be always a gap between the ideal and the actual behaviour of the parallelization speedup. Further results such as the underyling output file or a snapshot of it can be found in the Appendix, more precisly figure 14.

It is worth noting that for updating the variable nTotalIterationsCount, racing condition needs to be avoided. Instead of critical pragma here the pragma atomic was chosen, which share a lot of similarities. To run the code with multiple threads the pragma omp for was chosen without a collapse clause to make the code even faster. The colapse pragma would allow us to run both for loops in parallel and not only the first one. The main difficulty of this task was to think about classifying variables as private and shared. Also, the two values cx and cy needed to be modified since those two variables were loop dependent. It turned out that correct results could only be achieved by applying everything in correct order. Otherwise, results were completly wrong. The underlying codes can be found in the mandel repository.

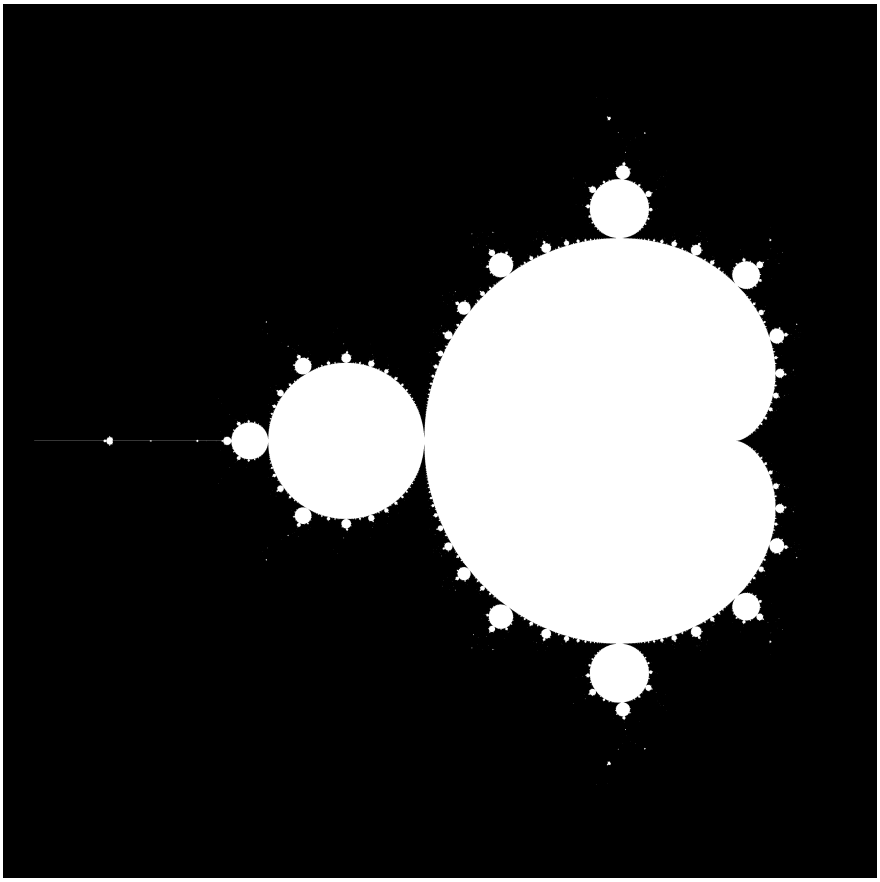

Figure 4: Parallelized Visualization Result

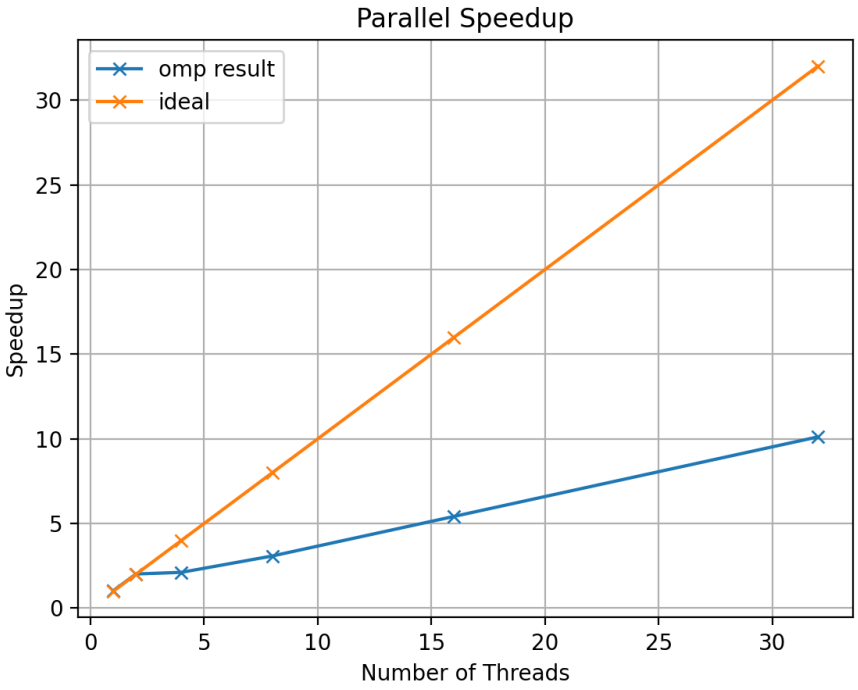

Figure 5: Strong Scaling Analysis of the Mandelbrot Set Simulation

3. Bug hunt [10 points]

3.1. Bug 1

The problem here is the combination of #pragma omp parallel and #pragma omp for, which leads to an error in this implementation. Also another problem is that the #pragma omp for should be followed directly with a for loop and no code in between. Thus, this problem can be resolved by first defining the omp parallel region followed by the omp for pragma as illustrated in figure 6.

```
#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++) {
            c[i] = a[i] + b[i];
            printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
        }
}
```

Figure 6: Bug 1 resolved

3.2. Bug 2

One major problem here is the fact that further specifications of private and shared variables is missing. Since the pragma omp for is used the variable total is unique for each thread and as a result we would run into a racing problem. This can be avoided by introducing a total variable that is private for each thread and in the end define a critical region or use the atomic pragma for the global total variable. Furthermore, every variable declared outside of the omp region is by default shared. Since, we need to update total in the end, this variable needs to be shared. This is not true though for the variable tid, which needs to private and not shared, since each thread should hold its own tid value. Otherwise there would be problems with the if condition. In addition, another way instead of defining a critical region would be to work with the reduction method presented in Chapter 1.

3.3. Bug 3

the pragma omp sections allows us to define one or more specific sections where only one thread executes the region. Thus the pragma omp barrier in line 79 of the provided file `omp_bug3.c` will lead to a stop of the whole program, since not more than one thread are already in one region and the function print_results is only called within a section. As a result by only deleting this one line, the code should work as intended.

3.4. Bug 4

Since given the hint from stackoverflow we can identify that the underlying problem here is a segmentation fault. One way to avoid this problem is by controlling the stack size limit. The issue here is that the variable a is huge and it's a private variable. Thus, each thread holds a copy of the variable a, which leads to the described problem. One way to easily resolve this issue is by granting each thread the possibility of allocating more memory to fit the requirement. This could be done for instance by defining in the bashfile a propriate amount with the command *export OMP_STACKSIZE*. For this specific case at least 8.8MB of memory are required, since the problem size N is equal to 1048. Moreover, due to the fact that 1 double requires 8 Bytes, the total amount of memory necessary would be $1048 \cdot 1048 \cdot 8$ Bytes = 8.786432 MB.

3.5. Bug 5

The last bug is the issue of a so called deadlock occuring. As already explained an omp section can only by entered by one single thread. Hence, in the first section this single thread first sets locka and then lockb, where in the second section another single thread sets concurrently lockb and then locka. As a result, both threads are waiting for the other one to unset the underlying lock and the code runs infinatelly long. This problem can easily be avoided by rearranging the locks that each thread is able to lock and unlock the given section. More precisely, declaring the array both threads need to lock and unlock this process concurrently. Same thing needs to happen for the second part of the code where the declaration of the other array follows concurrently.

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

The idea, how to parallelize the problem on the right hand side of the figure 7, is represented on the left hand side of the same image. The idea is to allocate n number of threads and for each thread a distribution variable (dist_local) should be created, where each thread should iterate over a truncated series of the total vector (#pragma omp for). As a result, in the end I still need to avoid racing when updating the target variable (dist). This can be done by making use of the pragma omp atomic. An advantage of this pragma is that it allows not only the control of concurrent threads, but also grants read and access rights to only a specific memory location, thus sharing similarities with the critical pragma. To make the code on the left hand side of image 7 even more efficient, it would be possible to use a reduction clause for the variable dist.

```
// Parallelized Code

 #pragma omp parallel shared(vec, dist)
  {
    long dist_local[BINS];
    for (int bin = 0; bin < BINS; ++bin) {
      dist_local[bin] = 0;
    }

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

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

```
// Sequential Code

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

Figure 7: Parallelization of the Histogram Calculation

Again a strong scaling analysis was conducted and the result can be observed in figure 8. From this plot it becomes evident that the speedup approaches a stationary value around 10. Initally the speedup is again close to the ideal value but then becomes worse over an increasing number of threads. One reason also here is the increasing overhead to reach the goal with an increasing amount of threads. The teamsize, as well as the two operations forking and joining becomes more effort and requires more time. Since our problem size stays the same this can lead to an increased overhead and makes the overall code even less efficient in some cases (see previous examples). Furthermore, a small snapshot of the results can be found in figure 13, while the complete output file is available in the respective coding directory.

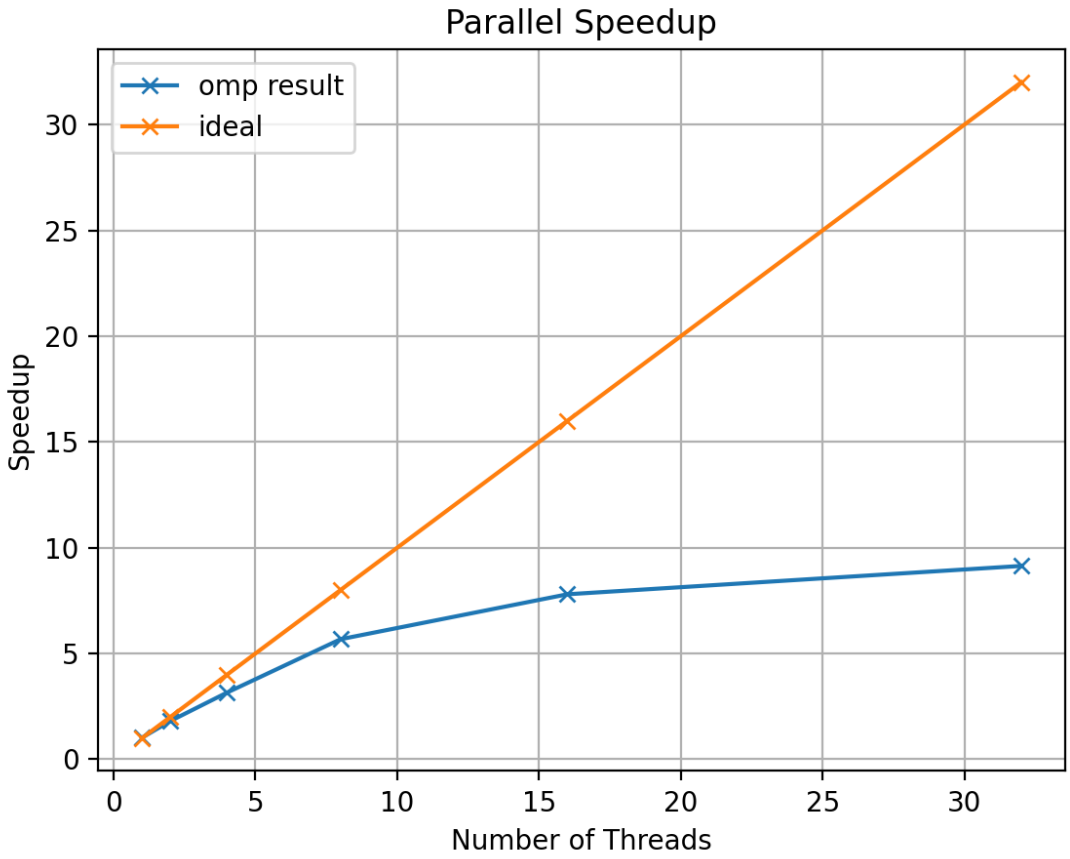

Figure 8: Strong Scaling Analysis for the Parallel Histogram Calculation

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

This problem was actually introduced in the book [1], as a problem which first appeared on the official OpenMP mailing in list in 2007. The difficulty in this task is the fact that there is a loop-carried dependency. Each iteration result depends on the previous one, which makes this task especially challenging. Since we truncate/particion the task, it becomes evident that a more general formulation is necessary. Thus, the problem can be rewritten as: opt[n] = pow(up, n). The problem of this code is that it's computationally expensive and we want to avoid it. But still at the first time the code starts and each thread runs the task the first time, we want that this computationally more expensive part should be used. From that point on, we continue with the faster an computationally less expensive algorithm.

From image 9, it is illustrated to face this problem that two important clauses were used. One additional clause to the prama for is, firstprivate(lastn). This clause assigns the initial value of lastn to its private copies when the parallel region starts [1]. This copy could also have been done manually, but this is just a more efficient and automized way. Next clause is lastprivate(Sn) which ensures that Sn has the same value as in the serial case after the loop is finished [1]. Since no results were expected according to the task description, I still put the output of my parallelized results into the Appendix, more precisely, figure 15.

```
// Parallelized Code:

#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
```

```
// Sequential Code:

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

Figure 9: Parallelized Code Snippet

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

Eventhough figure 10 is in german, it becomes clear how this code can be parallelized. The quicksort algorithm seperates the code at each step into a left and right branch, we can use this fact to parallelize each division and increase the number of threads while going down the tree. Since this problem can be described in form of a binary tree structure.

The underlying code can be found in the respective repository and is named as `quicksort_omp.c`. All the data and also the length of the container that needs to be sorted, needs to be shared hence they are declared as such. First used pragma is omp single which identifies a section of the code that must be run by a single available thread. Moreover, the additional clause nowait, is used to avoid a barrier at the end of the single directive. One approach to solve this problem and the idea described above, is to use omp sections. Since I struggled and got bad results with that appraoch I used the omp task pragma instead. This pragma can be used to explicitly define a task. According to the IBM website this code gets used a lot to parallelize irregular algorithms. Which is definatelly the case for this binary tree. Since it is typically unbalanced as well instead of following a balanced tree structure. Again the additional clause firstprivate was used to create copies. For more information return to Chapter 5.

The resulting figure 11 shows that overhead doesn't get less with an increased problem size. It becomes evident that the following parallelized code performs poorly for an increasing number of threads and is far away from the theoretical ideal result. The complete output results can be found in the respective repository but for a fixed problem size the results can also be found in the Appendix within figure 12.

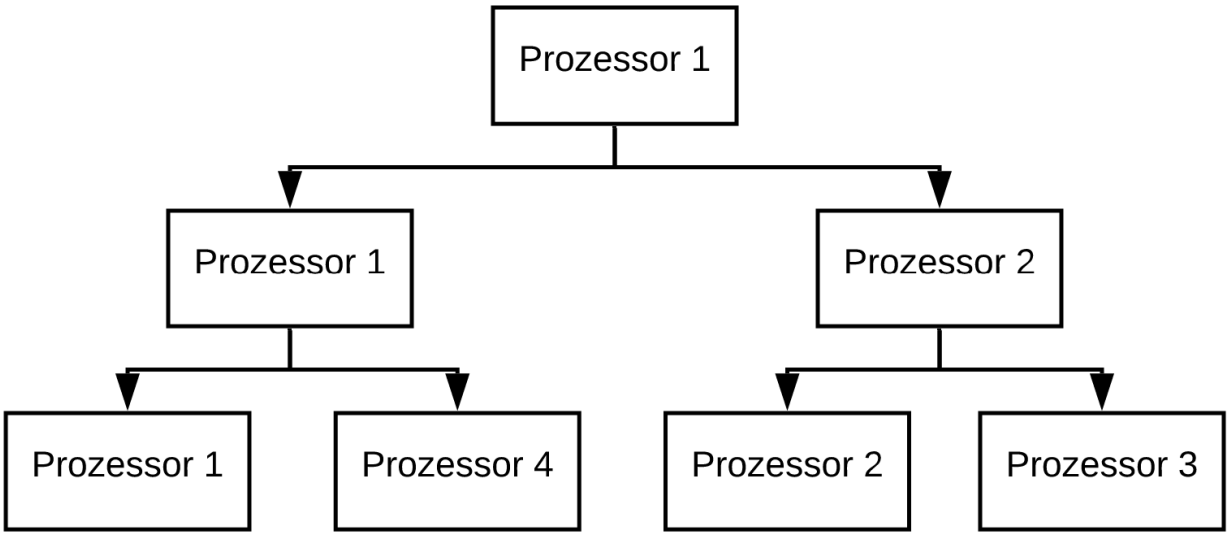

Figure 10: Parallelized quicksort algorithm, retrived from [3]

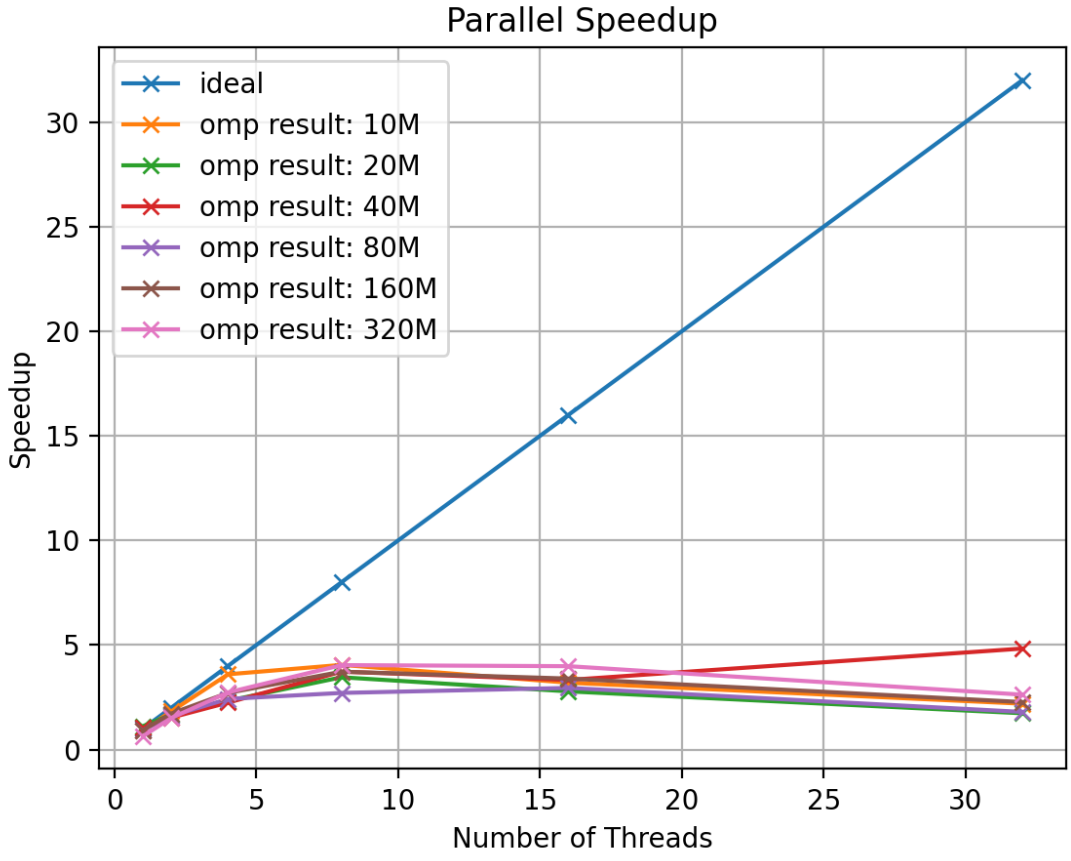

Figure 11: Result of the Parallelized Quicksort Algorithm for different Problem Sizes

7. Appendix

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

```
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 12: Output File for the Quicksort Problem

```
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 13: Output File for the Histogram Calculation Problem

```
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 14: Output File for the Mandelbrot Set Problem

```
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 15: Output File for the Loop-Dependencies Problem

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] High performance computing for science and engineering i: Strong and weak scaling, fabian wermelinger, computational science & engineering laboratory, 2018. [online]. available: `https://www.cse-lab.ethz.ch/wp-content/uploads/2018/11/amdahl_gustafson.pdf`. accessed: 2024-03-21.

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