J. H. Walther, S. Martin

Fall semester 2022

Set 2 - OpenMP

Issued: October 12, 2022

Hand in (optional): October 25, 2022 23:59

Question 1: OpenMP bug hunting (10 points)

a) Identify and explain any *bugs* in the following OpenMP code. Propose a solution. Assume all headers are included correctly.

```
1  // assume there are no OpenMP directives inside these two functions
2  void do_work(const float a, const float sum);
3  double new_value(int i);
4
5  void time_loop()
6  {
7      float t = 0;
8      float sum = 0;
9
10     #pragma omp parallel
11     {
12         for (int step=0; step<100; step++)
13         {
14             #pragma omp parallel for nowait
15             for (int i=1; i<n; i++)
16             {
17                 b[i-1] = (a[i]+a[i-1])/2.;
18                 c[i-1] += a[i];
19             }
20
21             #pragma omp for
22             for (int i=0; i<m; i++)
23                 z[i] = sqrt(b[i]+c[i]);
24
25             #pragma omp for reduction(+:sum)
26             for (int i=0; i<m; i++)
27                 sum = sum + z[i];
28
29             #pragma omp critical
30             {
31                 do_work(t, sum);
32             }
33             #pragma omp single
34             {
35                 t = new_value(step);
36             }
37         }
38     }
39 }
```

b) Identify and explain any bugs in the following OpenMP code and propose a solution. Assume all headers are included correctly.

```
1  #define N 1000
2
3  extern struct data member[N];// array of structures, defined elsewhere
4  extern int is_good(int i);// returns 1 if member[i] is "good", 0 otherwise
5
6  int good_members[N];
7  int pos = 0;
8
9  void find_good_members()
10 {
11     #pragma omp parallel for
12     for (int i=0; i<N; i++)
13     {
14         if (is_good(i))
15         {
16             good_members[pos] = i;
17             #pragma omp atomic
18             pos++;
19         }
20     }
21 }
```

c) Identify and explain any *improvements* that can be made in the following OpenMP code. Propose a solution. Assume all headers are included correctly.

```
1  void work(int i, int j);
2
3  void nesting(int n)
4  {
5      int i, j;
6      #pragma omp parallel
7      {
8          #pragma omp for
9          for (i=0; i<n; i++)
10         {
11             #pragma omp parallel
12             {
13                 #pragma omp for
14                 for (j=0; j<n; j++)
15                 work(i, j);
16             }
17         }
18     }
19 }
```

Question 2: Brownian motion (25 points)

The Brownian motion of N particles in a one-dimensional space

$$x_i(t) \in \mathbb{R}, \quad i = 1 \ldots N$$

is described with random walks. The algorithm performs M time steps until $t = t_{\mathsf{max}}$. One step is given by

$$x_i(t + \Delta t) = x_i(t) + \xi_i^{(t)} \sqrt{\Delta t}, \quad i = 1 \ldots N,$$

where $\xi_i^{(t)}$ are independent random variables from the standard normal distribution $\mathcal{N}(0, 1)$ and $\Delta t = t_{\mathsf{max}}/M$. The initial positions $x_i(0)$ are sampled from a uniform distribution over $[-\frac{1}{2}, \frac{1}{2}]$.

a) Given a serial implementation of the algorithm provided in the skeleton code, write a parallel version using OpenMP. Proceed in steps:

- Compile the serial version using **make main** and run it with **./main 100000 100** (corresponding to $N = 100000$ and $M = 100$) to produce files **hist_0.dat** and **hist_1.dat**. Use **make plot** to create **hist.pdf** with the histograms of the initial and final configurations of particles. Later you can simply type **make** that by default combines all the above stages.
- Add compiler flags and headers necessary for OpenMP following **TODO 1**. After implementing function **GetWtime()**, check that the program reports non-zero timings for time stepping (**walk:**) and histogram computation (**hist:**).
- Parallelize the time stepping (**TODO 2**) by splitting the particles among multiple threads. Make sure you do not introduce race conditions and each thread uses a separate random generator with a unique seed value. For storing the thread-local data, you may need to use arrays indexed by the thread-id or rely on data-sharing attributes of OpenMP.
- Parallelize the histogram computation (**TODO 3**).

b) Report strong scaling (speedup) on Euler up to 24 cores separately for time stepping (**walk:**) and histogram computation (**hist:**). The values of N and M should be sufficiently large such that the speedup does not depend on them.

c) Answer the following questions:

- Is the amount of computational work equal among all threads (for large N and M)?
- Do you observe perfect scaling of your code? Explain why.
- Run your program with $N = 1000$ and $M = 1$ on 1 thread and 500 threads. Then change the initial seed for the generator and run the program again. Plot the histograms in all four cases and include in your report. Does changing the number of threads have stronger effect on the final histogram than changing the initial seed? Explain why.

Question 3: Julia Set(25 points)

Romeo recently heard about the Julia set. Fascinated by its beauty, he decided to impress his beloved Juliet by painting the set on the wall of a large house across the street of her balcony. In order to do so he has to know what color goes where and how much paint of each color he will need. The job is not trivial, as he wants a very high resolution painting.

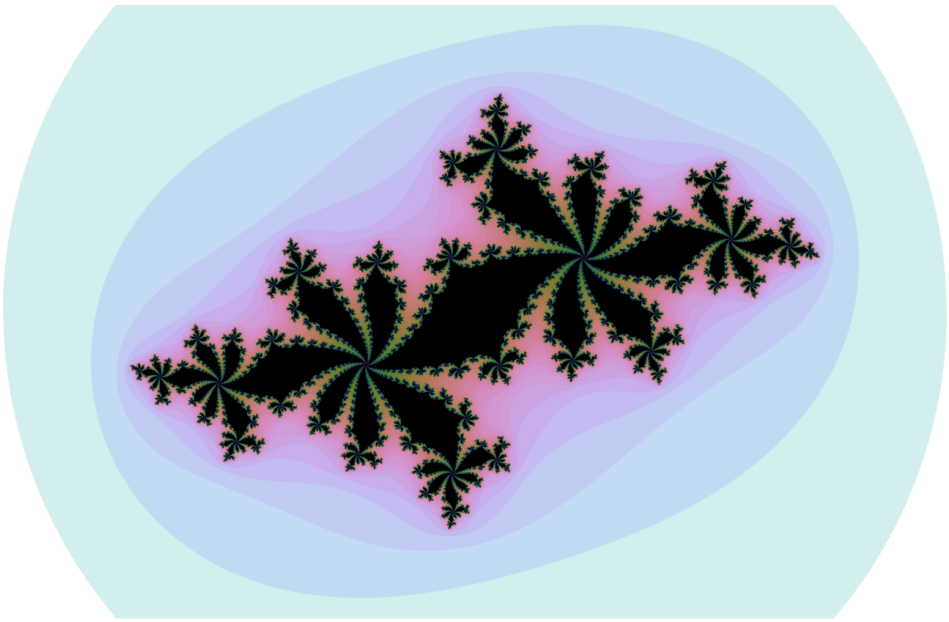


Figure 1: Visualization of the Julia set for $c = -0.624 + 0.435i$.

Romeo had some training in the arts of programming, but he managed only to write a slow serial code for computing the colors. Help him by a) parallelizing the code, b) computing the required amount of each color.

For the purpose of this exercise, we define the Julia set as the set of complex numbers w for which the numbers z_n, defined as

$$z_0 = w, \quad z_{n+1} = z_n^2 + c, \tag{1}$$

do not diverge for $n \to \infty$, where c is some fixed complex number. It can be shown that the divergence occurs if $|z_n|$ becomes larger than 2 for some n (if $|c| < 2$, which is true in our case). In our visualization in Figure 1 each pixel represents one value of w, where the x coordinate determines the real component and y the imaginary component of w. The pixel color is determined by the minimum n for which $|z_n| > 2$. For some pixels such n either does not exist or is too large, so we stop iterating after $n = 1000$.

You are given a C++ code that computes the iteration count for each pixel and a plotting script for generating the visualization.

a) Parallelize the function `julia_set` using OpenMP. Use `make && make run && make plot` to compile the code, run it and plot the set. Report execution time for 1, 2 and 4 threads (use `make run`). Use a small resolution for developing, large for benchmarking. Note that

due to the symmetry, the code computes only the bottom half of the image. Furthermore, the plotting script will likely run longer than the C++ code.

b) By definition, the number of iterations may differ from pixel to pixel, which causes load imbalance. Improve your parallelization to fix the load imbalance issue, without adding a large overhead. Describe your approach. Report execution time again. Do you get the desired speed-up?

   Hint: Run `lscpu` to get the number of cores. Important fields are *thread(s) per core*, *core(s) per socket* and *socket(s)*.

c) Shakespeare told you it is a good practice to avoid accessing the same cache lines by different threads. Propose a simple way to incorporate that into your code.

d) Implement the function `compute_histogram` that will help the prince determine how much paint of each color he needs to buy. Minimize the usage of critical regions, locks and atomics. See the skeleton code for details.

Question 1

a) 1) the for loop in line 12 is run by each thread created in line 10

↳ write #pragma omp parallel for in front of the line 12 could work but then each for loop inside tries to get parallel again.

Therefore I think it makes sense, to run the outer most for loop in serial.
Also fix the structure inside !

→ add pragma omp parallel before line 14

↳ therefore also delete "parallel" in line 14

→ also delete nowait as this might lead to issues as we use b[i] and c[i] on the next loop

My proposed solution is:

```
1  void do_work(const float a, const float sum);
2  double new_value(int i);
3
4  void time_loop()
5  {
6      float t = 0;
7      float sum = 0;
8
9      for (int step = 0; step < 100; step++)
10     {
11         #pragma omp parallel
12         {
13             #pragma omp for
14             for (int i = 1; i < n; i++)
15             {
16                 b[i - 1] = (a[i] + a[i - 1]) / 2.;
17                 c[i - 1] += a[i];
18             }
19
20             #pragma omp for
21             for (int i = 1; i < m; i++)
22                 z[i] = sqrt(b[i] + c[i]);
23
24             #pragma omp for reduction(+ : sum)
25             for (int i = 0; i < m; i++)
26                 sum += z[i];
27
28             #pragma omp critical
29             {
30                 do_work(t, sum);
31             }
32             #pragma omp single
33             {
34                 t = new_value(step);
35             }
36         }
37     }
```

b) All threads can write into the same position in the array good_members.

↳ to fix this, the atomic section should include the write

A solution could be:

```c
#define N 1000

extern struct data member[N];
extern int is_good(int i);

int good_members[N];
int pos = 0;

void find_good_members()
{
    #pragma omp parallel for
    for (int i=0; i<N; i++)
    {
        if(is_good(i))
        {
            #pragma omp atomic
            {
                good members[N] = is_good(i);
                pos++;
            }
        }
    }
}
```

c) 1) Nesting is inefficient cause every threads tries to open a new family of threads

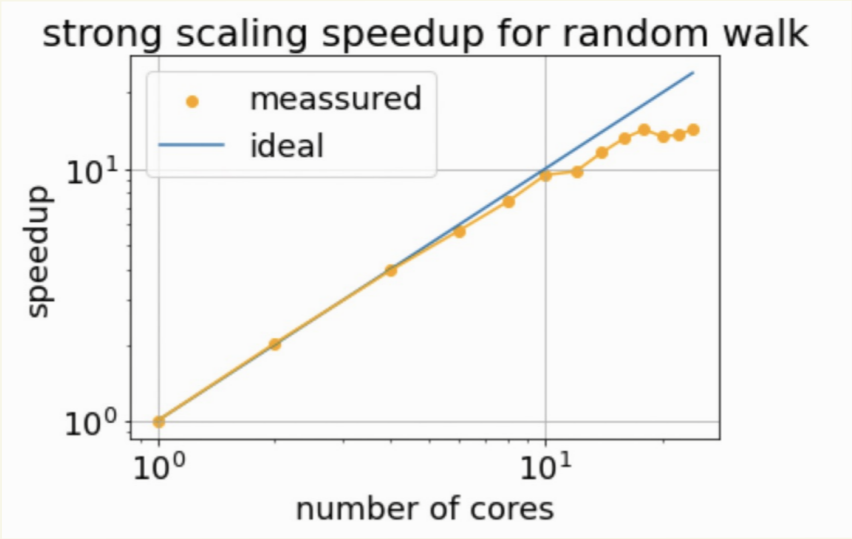
↳ a solution could look like:

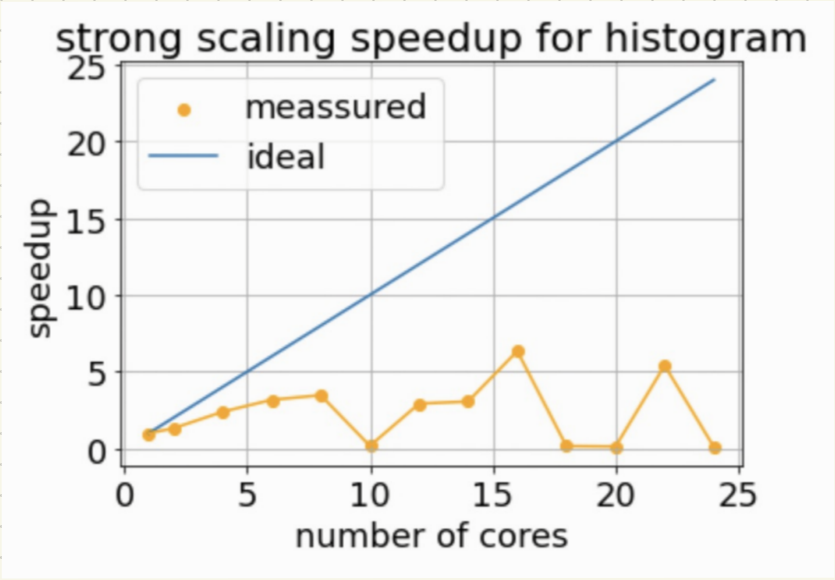
```c
void work(int i, int j);

void nesting(int n)
{
    int i, j;
    #pragma omp parallel for collapse(2)
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            work(i,j);
}
```

Question 2

b)


strong scaling speedup for random walk
meassured
ideal
speedup
10^1
10^0
10^0
10^1
number of cores

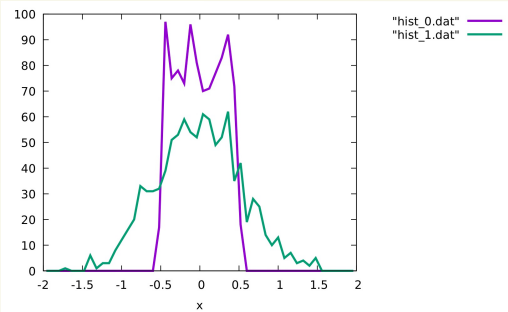

strong scaling speedup for histogram
meassured
ideal
speedup
25
20
15
10
5
0
0
5
10
15
20
25
number of cores

c)
- Yes it is N and M don't influence the splitting of work.
- For the time stepping I observe almost perfect scaling. This is due to the good implementation of the random walks in parallel using thread local seeds.

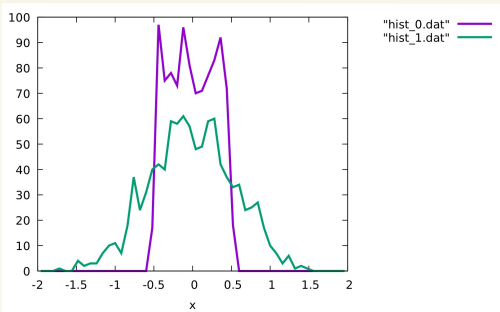
  For the hist. method the speedup isn't close to perfect. In fact there is no speedup at all.

  So something with the code doesn't work here.

- For 1 core

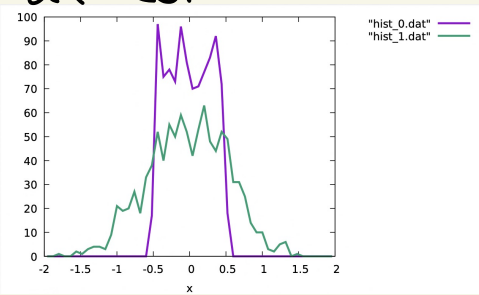


1 core, different seed

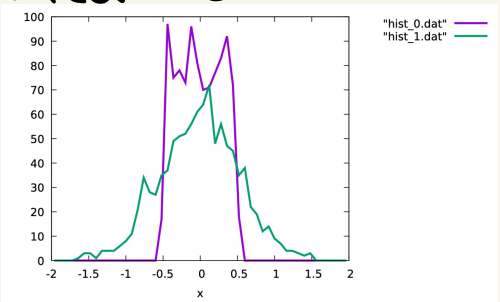


500 cores took to long and wasn't executed. therfore I just used 24 cores

24 cores



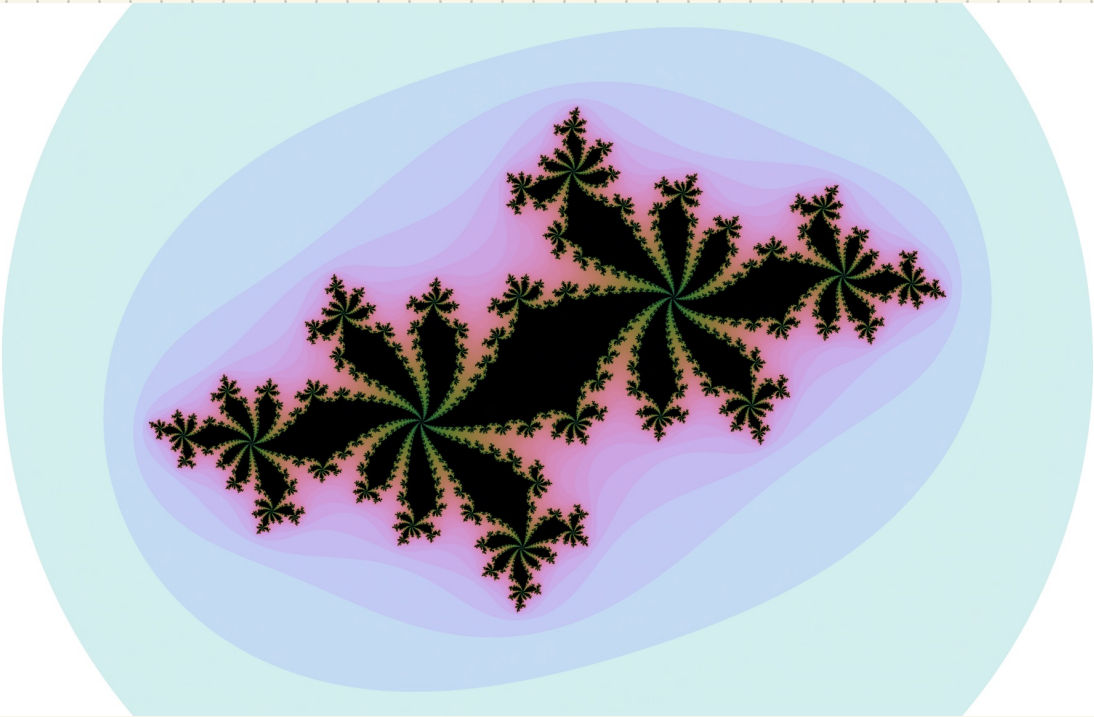
24 cores & different seed



Problem 3

a)

run time	1,94s	1,78s	1,20s
threads	1	2	4

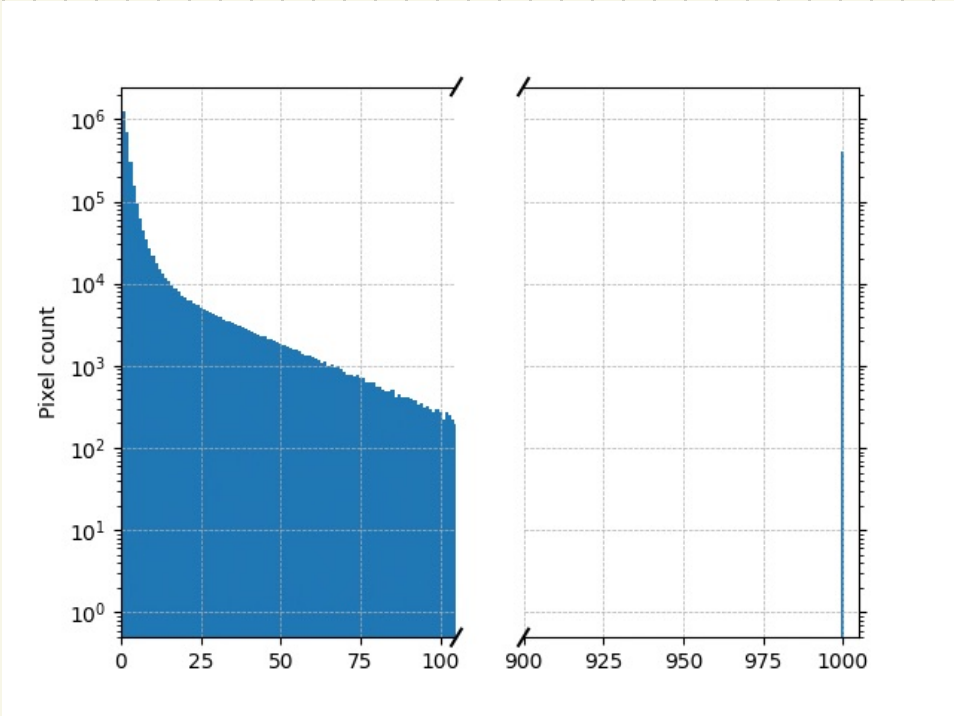


b)

run-time	2,1s	1,3s	0,81s
threads	1	2	4

So the speedup is indeed faster than before in subtask a)

d)


Pixel count
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0
25
50
75
100
900
925
950
975
1000