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.

```
// assume there are no OpenMP directives inside these two functions
void do_work(const float a, const float sum);
double new_value(int i);

void time_loop()
{
    float t = 0;
    float sum = 0;

    #pragma omp parallel
    {
        for (int step=0; step<100; step++)
        {
            #pragma omp parallel for nowait
            for (int i=1; i<n; i++)
            {
                b[i-1] = (a[i]+a[i-1])/2.;
                c[i-1] += a[i];
            }

            #pragma omp for
            for (int i=0; i<m; i++)
                z[i] = sqrt(b[i]+c[i]);

            #pragma omp for reduction(+:sum)
            for (int i=0; i<m; i++)
                sum = sum + z[i];

            #pragma omp critical
            {
                do_work(t, sum);
            }
            #pragma omp single
            {
                t = new_value(step);
            }
        }
    }
}
```

1. Remove `nowait` and `parallel` in the first for-loop. **(1 point)**
2. Combine the second and third loops into a single for-loop since the iteration space is the same and the iterations are independent **(1 point)**
3. Remove `critical`, simultaneous calls to `do_work` are allowed **(1 point)**
4. Place a barrier after `do_work` to remove a race condition with **t**, another thread may reach `do_work` after **t** is modified **(1 point)**
5. Add `nowait` to `single` **(1 point)**

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

```c
#define N 1000

extern struct data member[N];// array of structures, defined elsewhere
extern int is_good(int i);// returns 1 if member[i] is "good", 0 otherwise

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))
        {
            good_members[pos] = i;
            #pragma omp atomic
            pos++;
        }
    }
}
```

In order to avoid data races between different updates of global variable pos, the code puts the increment in a atomic construct. However, the code is incorrect, because there is a data race between the read of pos right before the atomic construct and the write of pos within the construct. Changing the body of the if-statement to one of the following gives the correct result:

```c
int mypos;
#pragma omp critical
{
    mypos = pos;
    pos++;
}
    good_members[mypos] = i;
```

```c
int mypos;
#pragma omp atomic capture
mypos = pos++;
good_members[mypos] = i;
```

1. Recognizing there is a race condition for the global variable pos **(1 point)**
2. Proposing any of the two solutions above (or one equivalent) **(1 point)**

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  }
```

1. Lines 6 and 8 can be combined into a single `#pragma omp parallel for`. This reduces the implicit barriers of these two lines from two to one. Same for lines 11 and 13 **(1 point)**
2. Use only a single OpenMP directive (no nested parallelism) by using a collapse clause: `#pragma omp parallel for collapse(2)` **(2 points)**. In this case the code would look as follows:

```
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: Brownian motion (25 points)

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

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

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

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

where $\xi_i^{(t)}$ are independent random variables from the standard normal distribution $\mathcal{N}(0, 1)$ and $\Delta t = t_{\text{max}}/M$. 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:**).
  1. Using `omp_get_wtime()` **(1 point)**
  2. Correct compiler flags **(1 point)**
  3. Correct header **(1 point)**
- 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.
  1. Private instance of `std::default_random_engine` for each thread **(2 points)**
  2. Different seeds for each `std::default_random_engine` **(1 point)**
  3. Private instance of `std::normal_distribution` for each thread **(1 point)**
  4. Swapped loop order: outer loop over particles **i** and inner loop over time **m** **(2 points)**
  5. Directive `#pragma omp for` applied to loop over particles (not time) **(2 points)**
- Parallelize the histogram computation (**TODO 3**).
  (5 points) Either
  1. Private array of size **nb** storing the local part of the histogram for each thread and summation without critical sections
  2. Reduction into one histogram in a critical section or atomic

  or

  1. Addition to a shared histogram with a critical section or atomic for every particle

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.

The strong scaling can be reported either as a plot (speedup, execution time or scaling efficiency) or a table with same quantities. Figure 1 is one example with the speedup T_1/T_P and strong scaling efficiency $T_1/(P\,T_P)$ where T_P is the execution time on P threads.

1. Scaling data with at least two values (e.g. 1 and 24 threads) **(1 point)**
2. Scaling data with more than 6 values (covering range between 1 and 24 threads) **(1 point)**
3. Several samples for each number of threads, possibly with averaging over them **(1 point)**

c) Answer the following questions:

- Is the amount of computational work equal among all threads (for large N and M)?
  Yes, there is no dependencies between particles and the same operations are performed for every particle **(1 point)**
- Do you observe perfect scaling of your code? Explain why.
  The code should have good scaling (in the order of 70% on 24 cores) for both the random walk (**walk:**) and the histogram (**hist:**) as there is no interactions between particles and each random walk requires a limited number of memory accesses (all time steps work with data in cache).
  1. Explanation of any observed scaling **(1 point)**
  2. Scaling in the order of 70% on 24 cores **(2 points)**
- 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.
  Both should have similar effect as seen from Figure 2. Large differences in `hist_1` with 500 threads may result from incorrect seeding of the generators.
  1. Explanation of any observed result **(1 point)**
  2. All histograms `hist_1` look similar **(1 point)**

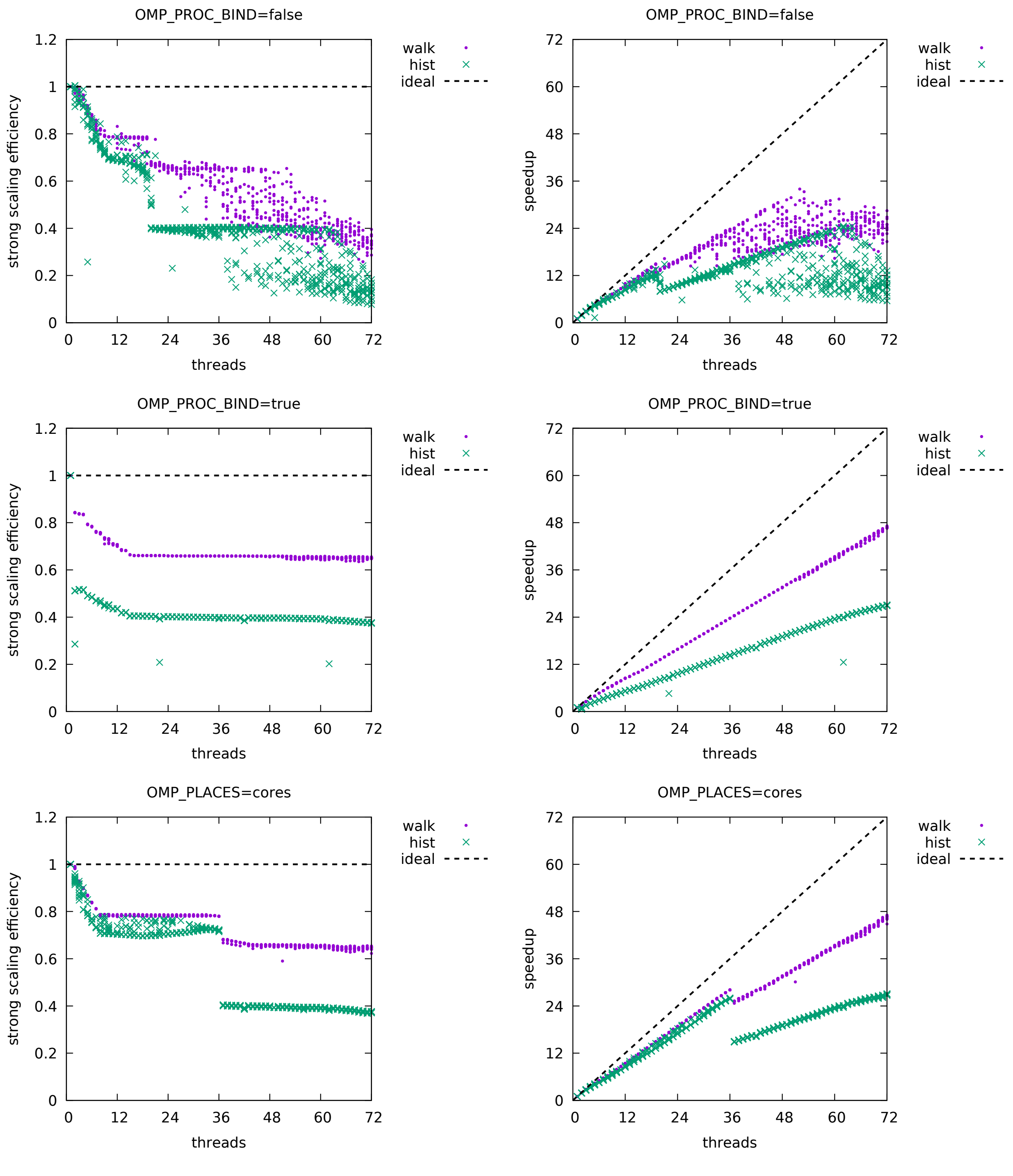

Figure 1: Speedup on a 36-core compute node of Piz Daint with two `Intel Xeon E5-2695`. Different options for `OMP_PLACES` and `OMP_PROC_BIND`. Samples from 10 runs for each number of threads. Produced with $N = 10^7$ and $M = 10$ and the histogram computation repeated 100 times.

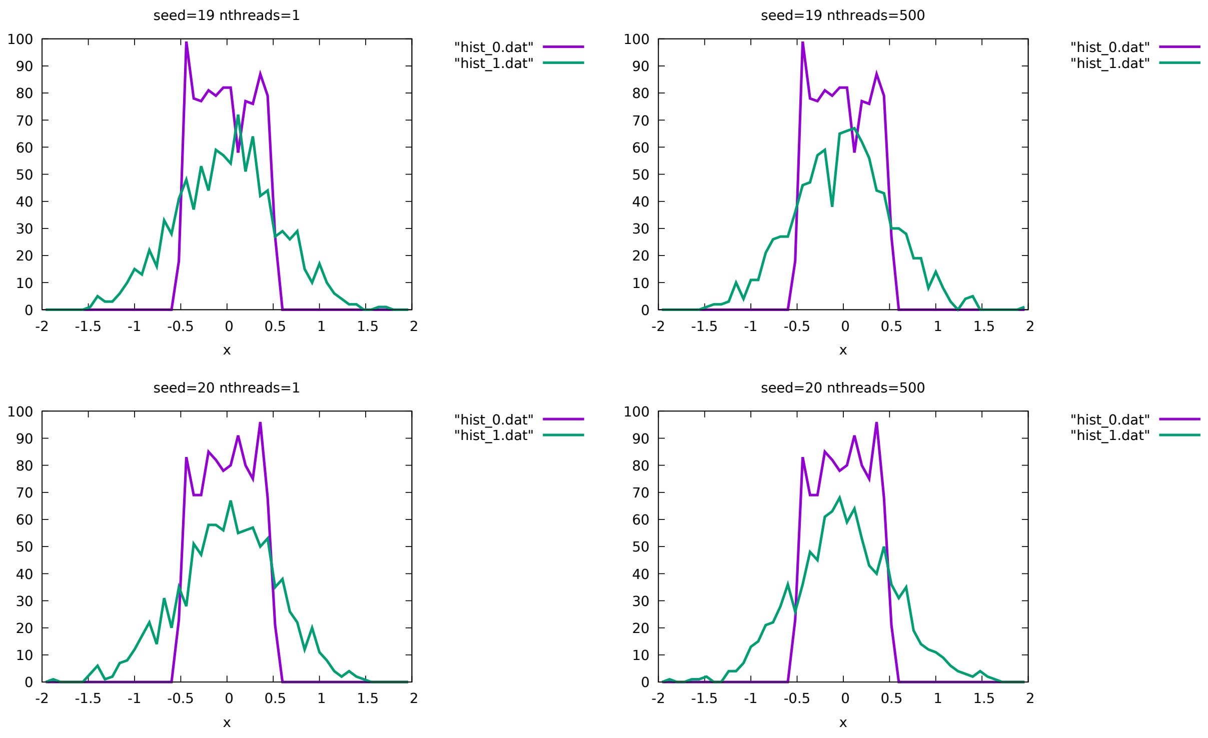

Figure 2: Histograms produced with $N = 1000$ and $M = 1$ for different initial seeds and the number of threads.

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.

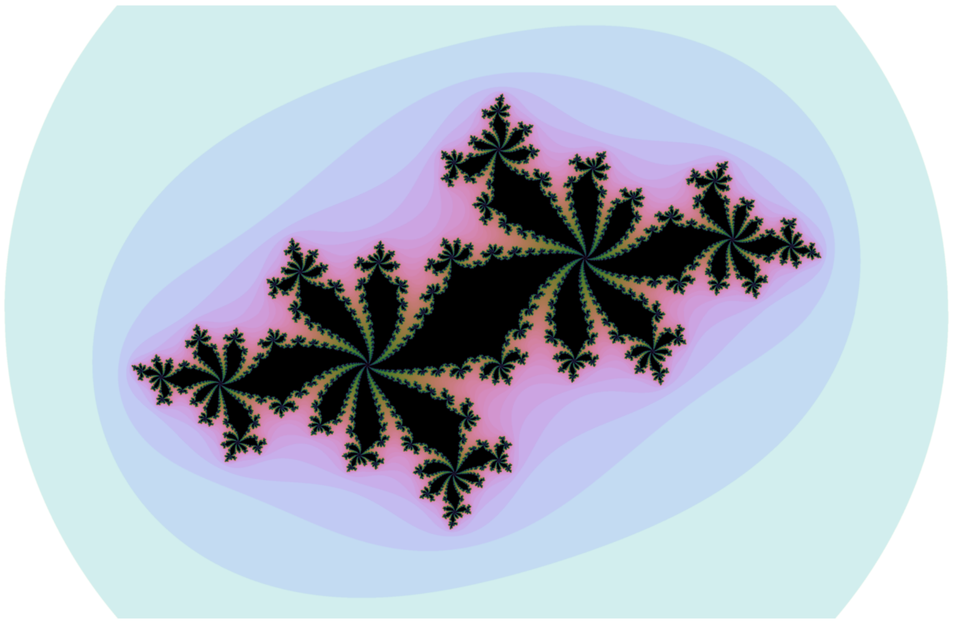

Figure 3: Visualization of the Julia set for $c = -0.624 + 0.435i$.

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 3 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.

1. Parallelized loop simply by adding `pragma omp parallel for`. Not necessary to add `collapse(2)`. **(4 points)**
2. Time measurement **(1 point)**

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)*.

Add `schedule(static, chunk_size)` or `schedule(dynamic)`. Dynamic schedule without a chunk size is fine because the overhead is not that large (few percent). Sample running times: 1.37s, 0.69s, 0.44s for 1, 2, 4 threads (on 2-core Mac!).

1. Use of static schedule (with a chunk size) or of a dynamic schedule **(4 points)**
2. Time measurement **(1 point)**

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.

Use a chunk size which is a multiple of 32 with e.g. `schedule(static, 32)` and align the array with `alignas(64)`.

- Mention chunk size **(3 points)**.
- Mention array alignment **(2 points)**.

  OR
- **(2 points instead of 5)** for solution where splitting by rows or padding of arrays was done.

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.

A serial solution is a double `for` loop that traverses all pixels and increases the counter corresponding to the iteration count:

```
for (int i = 0; i < HEIGHT; ++i)
    for (int j = 0; j < WIDTH; ++j)
        ++result[image[i][j]];
```

To parallelize, put the for loops in a parallel region and add `pragma omp for` before the outer loop. To avoid race condition, use a local copy of `result` and perform the reduction after the for loops are done. Alternatively, replace `std::vector<int>` with `int[MAX_ITERATIONS + 1]` for the `result` variable and simply use `pragma omp parallel for reduction(+: result)`.

- **10 points** for a parallelized loop with reduction or a parallelized loop with atomics or complexity `O(MAX_ITERATIONS * HEIGHT * WIDTH)`.

  OR
- **5 points** for a serial version or parallelized with a race condition