OpenMP 2

1. Imagine you want to compute the minimum axis-aligned bounding box for a list of points in 2D space (tuples of (x, y) coordinates). To do so, you have to find the two tuples $(min(x_i), min(y_i))$ and $(max(x_i), max(y_i))$. Tuples are defined using the structure

```
typedef struct {
  float x;
  float y;
} tuple;
```

Your task is to implement two different versions of such functionality:

a) using a manual reduction, and

b) using a user-defined reduction.

Solution. See `HW-OpenMP2-udr.c`

2. Explain the differences between the following three pieces of code. Are they all correct? Assume the array `v` has length `m` and the values in array `indices` range from `0` to `m`.

1)
```
    #pragma omp parallel for default(none) shared(v, indices, n)
    for (i = 0; i < n; i++)
        v[ indices[i] ] += f(i);
```

2)
```
    #pragma omp parallel for default(none) shared(v, indices, n)
    for (i = 0; i < n; i++)
        #pragma omp critical
        v[ indices[i] ] += f(i);
```

3)
```
    #pragma omp parallel for default(none) shared(v, indices, n)
    for (i = 0; i < n; i++)
        #pragma omp atomic
        v[ indices[i] ] += f(i);
```

Solution. Version 1 is incorrect because multiple `indices[i]` may return the same index, which may lead to race conditions and incorrect results. Versions 2 and 3 are correct. The difference between versions 2 and 3 is that with `critical` the code is sequentialized because, no matter the value of `indices[i]`, only one thread will be executing the critical section, while with `atomic`, threads accessing different indices will be allowed to do it in parallel.

See `08b.histogram-critical.c` and `08c.histogram-atomic.c`.

3. In this task you will estimate the value of π using the Monte Carlo method. Given a probability P, the Monte Carlo method relies on the generation of random samples (events) to compute a numerical approximation of P.

   Consider a circle inscribed in a square (Fig. 1); the method simply consists in generating random points (x, y) within the square range. Given the probability that the points lie within the circle, and the actual number of generated random points that do so, we can estimate π.

   First, what is the probability of random points ending up within the area of the circle? The answer is to find the relationship between the geometry of the square and the circle. The area of the square is $4R^2$ and the area of the circle is πR^2. Now, the ratio of the area of the circle over the area of the square is $\frac{\pi}{4}$. That is, the probability that a random point within the square lies also within the circle is $P = \frac{\pi}{4}$, and thus $\pi = 4P$.

   Write a program that generates np random points using the `drand48` function, computes the ratio q that approximates the probability P, and uses it to estimate the value of π. Base your code on a for loop with $np = 50,000$ iterations. Do not expect more than 1 or 2 decimals of accuracy.

   Parallelize your code using the `parallel` and `for` OpenMP constructs. Pay attention to shared and private variables. You may need to use some of the clauses presented in class.

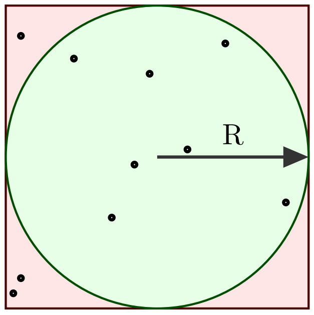

Figure 1: Circle of radius R inscribed in a square of side R^2. Some random points are shown.

Solution. See `HW-OpenMP2-Montecarlo-Pi.c`

4. In this task you will parallelize a given sequential code that solves a steady-state heat conduction problem over a thin square plate. Concretely, the code simulates the diffusion of heat on a plate to determine at which temperature it stabilizes. The initial temperature of the surface is 50 degrees, and a constant temperature is applied at the boundaries. Specifically, the left, top and right boundaries are kept constant at 100 degrees while the bottom boundary is kept at 0 degrees.

   The code relies on the iterative computation of the Poisson equation $\Delta\phi = f$ using finite differences. The surface is discretized in a grid of $n \times n$ grid points. At each iteration, the interior grid points[1] $\phi_{i,j}$ are updated following the rule:

$$\phi(i,j) = \frac{\phi(i-1,j) + \phi(i,j-1) + \phi(i+1,j) + \phi(i,j+1)}{4}.$$

   The sequential program provided represents the discretized grid as a matrix of size $n \times n$. The matrix is initialized according to the conditions described above, and then the program performs an indefinite number of timesteps (iterations). At each iteration, the program estimates the new temperature at timestep t from that of the previous (old) iteration $t-1$. When the difference between *every* pair $(\phi_{old_{i,j}}, \phi_{new_{i,j}})$ from two consecutive iterations is below a given threshold, the program has found the steady state and it terminates.

   You have to parallelize the given code using OpenMP. The goal of is to take the code given in `HW-OpenMP-2.4.c` and produce multiple parallel versions using OpenMP. After testing the code for its correctness, you will collect timings to study the attained speedups, using 1, 2, 4, and 8 threads. You have to code the following 4 versions of the function `compute_*`.

   (a) Parallelize the outermost loop in the matrix update (lines 63–68).

   (b) Parallelize the innermost loop in the matrix update (lines 63–68).

      - Which of these two versions do you expect to be faster? Why?
      - Run the code and look at the timings. Was your prediction correct/accurate?

   (c) Take the fastest of the two versions above and parallelize also the computation of the convergence. Use two separate parallel regions.

   (d) Rewrite version "(c)" using one single parallel region.

   Create the 4 different functions (`compute_outer`, `compute_inner`, `compute_conv` and `compute_single_region`, respectively), and add the necessary code to the `main` function to evaluate the timings and speedups. Run experiments for $n \in [100, 500, 1000]$ using 1, 2, 4, and 8 threads.

   We suggest you run the job in the university's cluster using the *jobscript* provided:

```
bsub < HW-OpenMP-2.4-jobscript
```

[1]Remember that the boundaries are kept constant at a temperature of either 0 or 100 degrees.

Solution. See `HW-OpenMP2-Heat.c`

5. In this task you will practice with data-sharing attributes, race conditions, and synchronization constructs. We provide you with the program `HW-OpenMP-2.5.c`[2] which contains at least 4 bugs. We ask you to identify and fix these bugs. We also ask you to list the bugs you found and explain which problem they were causing and how you fixed them. Instructions on how to run the program and the expected result are given at the beginning of the file. Test with different number of threads.

   Solution. The bugs are identified and explained in the solution code. See `HW-OpenMP2-Mandelbrot.c` and `HW-OpenMP2-Mandelbrot-v2.c`.

6. Given the following code, assume that the code is executed by two threads, iterations are distributed in two big chunks, one per thread, and function `f(i)` takes i ms to execute. Answer the questions below.

```
    #pragma omp parallel
    {
        #pragma omp for schedule(static)
        for (i = 1; i < n; i++)
            f(i);
        #pragma omp for schedule(static)
        for (i = 1; i < n; i++)
            f(n-i);
    }
```

   a) How long would each thread take to execute the parallel region? How much of that time is spent in waiting for the other thread?
   b) How would the execution time change if we used `schedule(static,1)` in both `for` directives?
   c) Would it improve if we used `schedule(dynamic, 1)`?
   d) Is there an OpenMP clause that allows to eliminate the waiting time? How long would each thread take when using this clause?

   Solution.

   a) How long would each thread take to execute the parallel region? How much of that time is spent in waiting for the other thread?

[2]This program was written by Mark Bull and Tim Mattson and is often used in OpenMP tutorials to illustrate a number of common bugs.

Because of the implicit barriers at the end of each worksharing directive, both threads take the same time to execute the parallel region. Thread 1 spends $t = \sum_1^{\frac{n}{2}} i$ ms to execute the first loop, while thread 2 takes $3t$. In the second loop, threads exchange this behavior. Thus, in total, both threads take $6t$ ms to compute the parallel region, where $4t$ ms are spent in computation and $2t$ ms are spent in waiting for the other thread to complete.

b) How would the execution change time if we used `schedule(static,1)` in both `for` directives?

In this case the workload would be much better balanced. Thread 1 would take $t_2 = \sum_1^{\frac{n}{2}} 2 * i - 1$ and thread 2 would take $t_3 = \sum_1^{\frac{n}{2}} 2 * i$, which is only $\frac{n}{2}$ ms more than thread 1. Thus, waiting time is reduced from $O(n^2)$ to $O(n)$.

c) Would it improve if we used `schedule(dynamic, 1)`?

In this case, the execution and waiting times for the first loop would be the same as in case "b)". In the second loop, the work would be perfectly balanced: every four chunks of size 1, thread 1 would take 1st and 4th, thread 2 would take 2nd and 3rd.

d) Is there an OpenMP clause that allows to eliminate the waiting time? How long would each thread take when using this clause?

Yes, we can use the `nowait` clause in the first loop (also in the second) to remove the implicit barriers. In this specific code, we are guaranteed that both threads will be assigned the same logical iterations in both loops, and by removing the barriers we achieve a perfect work balancing, where each thread takes $4t$ to complete.

7. Write an OpenMP program to determine the default scheduling of your OpenMP implementation of choice. Assume it is one of `static`, `dynamic`, or `guided`.

   Solution. See `HW-OpenMP2-default-schedule.c`

8. In this task you will reason about and test different scheduling schemes in OpenMP. We provide you with the program `HW-OpenMP-2.8.c`, which runs a number of *for* loops, each with a different distribution of workload per iteration. More specifically, the program contains 4 independent (non-nested) *for* loops, each of them with the following type of distribution of work:

   - **Constant**: all iterations perform the same amount of work.
   - **Small fluctuations**: all iterations perform a similar amount of work, with fluctuations of a small percent.

- **Large fluctuations**: most iterations perform little work, while a few of them, concentrated in the beginning of the loop, may require several orders of magnitude more work than the baseline.
- **Increasing**: each iteration performs an amount of work proportional to the index of the loop, which ranges from 0 to some n.

Each work distribution is simulated by four different functions that execute the necessary amount of work in each case, based on the iteration number i and the number of iterations n.

Before proceeding to experimentally observe the behavior for different scheduling schemes, we ask you to look into the code of each of these functions, and produce four plots illustrating the work distribution. Represent the iteration number in the x-axis and the amount of work in the y-axis. As an example, the plot for the *constant* distribution is a horizontal line. No need for wasting time in fancy plots, you can use any plotting program or even handmade and scanned plots :)

Then, still before running any code, you must argue which between `static` and `dynamic` schemes with different *chunk sizes* should be faster in each case. Explain your choice in a few lines.

Finally, run the code and compare with your predictions. In case you were mistaken in any of your predictions, try to explain why this was the case. To run the code, we provide you with a *script* which you can run by simply typing

```
./HW-OpenMP-2.8-script
```

in your shell. Following the example in the script, run the code with different schemes and chunk sizes.

We suggest you run the job in the university's cluster using the *jobscript* provided:

```
bsub < HW-OpenMP-2.8-jobscript
```

By default, the script uses 4 threads. Feel free to try with a larger number of threads. If you use more than 8 threads, remember to change the line "`#BSUB -n 8`" in the jobscript to request a larger node.

Solution. See files `HW-OpenMP2-workloads.txt` and `HW-OpenMP2-workloads.output.txt`.

9. In this task you will use OpenMP `sections` to parallelize a recursive algorithm that computes the Fibonacci numbers. You have to modify the function `fibonacci` in `HW-OpenMP-2.9.c` to use the constructs `sections` and `section` in order to execute each recursive call in parallel. *Keep in mind that by no means this is the best way to compute Fibonacci numbers. The goal is simply to practice the use of* `sections`.

Solution. See `HW-OpenMP2-Fibonacci-sections.c`