Question 1: Parallel Monte Carlo using OpenMP

Question 1a)

The following figures illustrate the used programs to compute the Monte Carlos simulation.

main.cpp > ...

```cpp
#include <omp.h>
#include <random>
#include <stdio.h>
#include <stdlib.h>

// Integrand
inline double F(double x, double y)
{
    if (x * x + y * y < 1.) { // inside unit circle
        return 4.;
    }
    return 0.;
}

// Method 0: serial
double C0(size_t n)
{
    // random generator with seed 0
    std::default_random_engine g(0);
    // uniform distribution in [0, 1]
    std::uniform_real_distribution<double> u;

    double s = 0.; // sum
    for (size_t i = 0; i < n; ++i)
    {
        double x = u(g);
        double y = u(g);
        s += F(x, y);
    }
    return s / n;
}

```

Figure 1: Serial version of the Monte Carlo simulation (provided by the TAs).

```cpp
// Method 1: openmp, no arrays
// TODO: Question 1a.1
double C1(size_t n)
{
    // Variable to store the result.
    double sum = 0.0;

    // Set the number of threads used to test the code.
    // omp_set_num_threads(1);

    // Print out number of threads.
    int nthreads;
    #pragma omp parallel
    #pragma omp master
    nthreads = omp_get_num_threads();

    #pragma omp parallel
    {
        // Get the id of the thread.
        const int tid = omp_get_thread_num();

        // Create individual random generator for each thread.
        // The thread id is used for individual seeds.
        std::default_random_engine generator;
        generator.seed(tid);
        std::uniform_real_distribution<double> u;

        // Compute sum of the thread.
        #pragma omp for reduction(+:sum) nowait
        for (int i = 0; i < n; ++i)
        {
            double x = u(generator);
            double y = u(generator);
            sum += F(x, y);
        }
    }

    return sum/n;
}

```

Figure 2: Parallelized Monte Carlo simulation without the usage of arrays.

```cpp
// Method 2, only `omp parallel for reduction`, arrays without padding
// TODO: Question 1a.2
double C2(size_t n)
{
    // Set the number of threads used to test the code.
    // omp_set_num_threads(1);

    int nthreads;
    #pragma omp parallel
    #pragma omp master
    nthreads = omp_get_max_threads();

    // Allocate memory to store the individual results of each thread.
    // Note: No padding is used for this array, thus, false sharing will be caused.
    double *sumAr = new double[nthreads];

    #pragma omp parallel
    {
        // Get the id of the thread.
        int tid = omp_get_thread_num();

        // Create individual random generator for each thread.
        // The thread id is used for individual seeds.
        std::default_random_engine generator;
        generator.seed(tid);
        std::uniform_real_distribution<double> u;

        #pragma omp for
        for (int i = 0; i < n; i++) {
            double x = u(generator);
            double y = u(generator);
            sumAr[tid] += F(x, y);
        }
    }

    double sum = 0.0;
    for (int i = 0; i < nthreads; i++) {
        sum += sumAr[i];
    }

    return sum/n;
}
```

Figure 3: Parallelized version of the Monte Carlo simulation, usage of arrays without padding.

```cpp
// Method 3, only `omp parallel for reduction`, arrays with padding
// TODO: Question 1a.3
double C3(size_t n)
{
    // Set the number of threads used to test the code.
    omp_set_num_threads(1);

    // Get the number of the threads in use.
    int nthreads;
    #pragma omp parallel
    #pragma omp master
    nthreads = omp_get_max_threads();

    // Allocate memory to store the individual results of each thread.
    // Array padding is used to avoid false sharing.
    double *sumAr = new double[nthreads*8];

    #pragma omp parallel
    {
        // Get the id of the thread.
        int tid = omp_get_thread_num();

        // Create individual random generator for each thread.
        // The thread id is used for individual seeds.
        std::default_random_engine generator;
        generator.seed(tid);
        std::uniform_real_distribution<double> u;

        #pragma omp for nowait
        for (int i = 0; i < n; i++)
        {
            double x = u(generator);
            double y = u(generator);
            sumAr[tid*8] += F(x, y);
        }
    }

    double sum = 0.0;
    for (int i = 0; i < nthreads*8; i+=8) {
        printf("%d = %f\n", i, sumAr[i]);
        sum += sumAr[i];
    }

    return sum/n;
}
```

Figure 4: Parallelized version of the Monte Carlo simulation, usage of arrays with padding.

Question 2a)

Contrasting the results of the parallel versions of the Monte Carlo simulation it is clearly visible that the performance of code using arrays without padding is clearly worse than the other two options. The reason for this performance drawback can be explained by 'False sharing'. As the sums of the individual threads are stored in the same cache line a regular updating of cache lines is caused. Therefore, the concept of locality cannot be exploited and the code performs considerably worse.

Question 3a)

The amount of computational work was equally distributed between the number of threads for samples tested in a range of 1e6 to 1e9. The following figures display the printed results of work sharing between threads. A counter variable cnt was used to identify how many iterations each thread executes.

(a)

```
argc = 2
argv = 1
C1
Total number of threads = 8
I am thread ID = 0
I am thread ID = 7
I am thread ID = 2
I am thread ID = 1
I am thread ID = 6
I am thread ID = 4
I am thread ID = 3
I am thread ID = 5
I am thread 0 - cnt = 12500000
I am thread 6 - cnt = 12500000
I am thread 2 - cnt = 12500000
I am thread 3 - cnt = 12500000
I am thread 5 - cnt = 12500000
I am thread 7 - cnt = 12500000
I am thread 1 - cnt = 12500000
I am thread 4 - cnt = 12500000
res:  3.14197543999999995279
ref:  3.14159265358979311600
error: 3.827864102068367913O3e-04
time: 0.52992218296276405454
```

(b)

```
The output (if any) follows:

argc = 2
argv = 1
C1
Total number of threads = 8
I am thread ID = 0
I am thread ID = 7
I am thread ID = 4
I am thread ID = 2
I am thread ID = 1
I am thread ID = 5
I am thread ID = 6
I am thread ID = 3
I am thread 6 - cnt = 125000000
I am thread 7 - cnt = 125000000
I am thread 0 - cnt = 125000000
I am thread 5 - cnt = 125000000
I am thread 1 - cnt = 125000000
I am thread 2 - cnt = 125000000
I am thread 3 - cnt = 125000000
I am thread 4 - cnt = 125000000
res:  3.14163318800000013198
ref:  3.14159265358979311600
error: 4.053441020701598063l7e-05
```

Figure 5: Workload distribution of a sample size of **(a)** n = 1e8 and a sample size of **(b)** n = 1e9.

Considering the results of Figure 6 it is clearly visible that perfect scaling is observed for the parallelized programs 'No arrays' and 'Arrays (padding)'. The speed up is computed by dividing the old execution time, in this case the time it takes to compute the Monte Carlo simulation with one thread, i.e. a serial program, by the newly measured computation time. As the results show that the computation time is cut by about a factor of ca. two by doubling the number of threads, the code can be described to scale perfectly. Perfect scaling can be observed due to the fact that the code is very well parallelizable. The crucial code segment is the computation of the different estimated values inside the circle. As neither atomized executions nor critical sections are involved in the parallelized for loop, the different threads do not have to wait for each other during the iterations and can proceed without having to communicate with the other threads. Thus, no performance bottleneck due to thread communication can be observed that would decrease performance with an increase in the number of threads. Furthermore, the addition of the individual thread sums and the branching statements can be described as negligible compared to the up to 1e10 for loop iterations, as far as performance is concerned, causing no hurdles for perfect scaling.

The code was run twice under the exact same conditions to test for performance changes. With regards to the execution time plotted in the figure down below, it is to say that slight changes can be observed. One reason why programs have different execution times when run multiple times is the time-sharing of OS kernels like Linux. Such kernels handle several different processes at the same

time so that other processes may run between execution phases of a particular program. Such processes can impact the overall time of code. The more cores that are used for a program the more likely such performance impacts become. A way around this problem is to measure the used CPU time. Performance changes due to cached files and cached data should not be an issue on the Euler cluster as lots of programs are run all the time making it very unlikely that the same node with prior cached files is used for a re-run of a particular program falsifying performance.

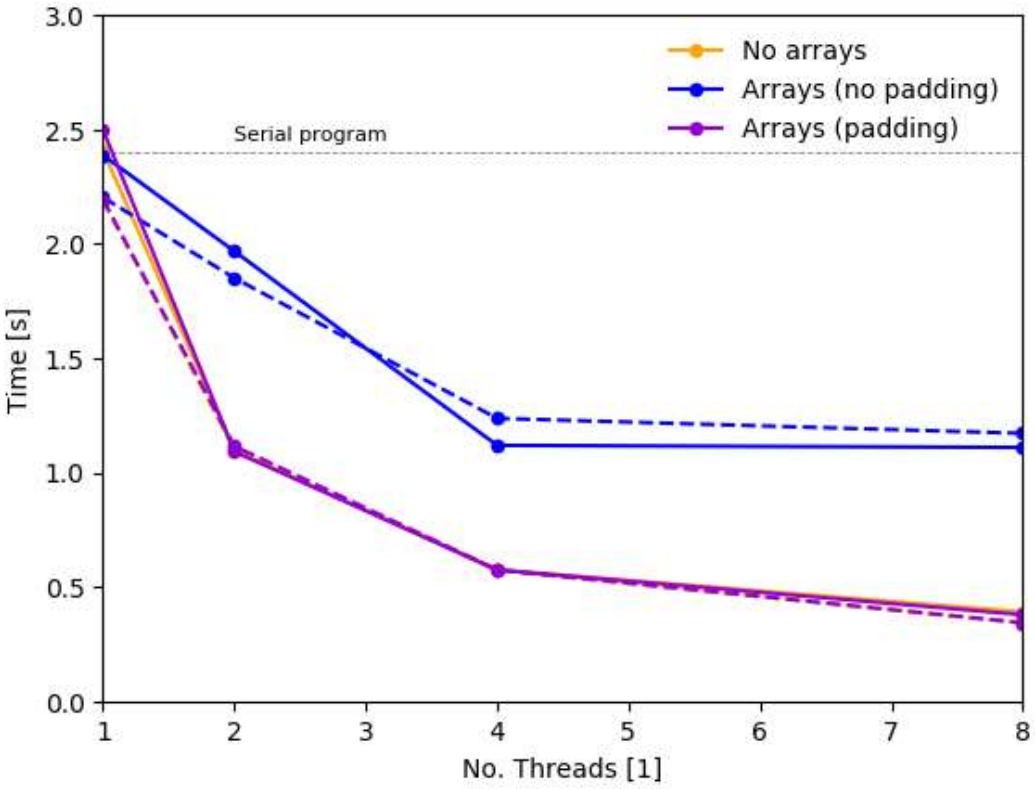

Figure 6: Results of the computation time of the Monte Carlo simulation for a serial implementation and 3 different parallel implementations. The computation time for reach piece of code is mapped against the used number of threads. Each parallel code was tested twice (continuous and dashed lines) to check for performance differences. The number of samples used is n = 1e8 (provided by the TAs).

Question 2: OpenMP Bug Hunting I

The problem of the given code is in line 14. Although the incrementing of the variable pos is done correctly using the atomic keyword in order to avoid a race condition, the assignment of “good members” could potentially cause errors.

To illustrate the issue in line 14 consider the following procedure. Four threads are spawned as thread team to loop over the 1,000 array values and each thread is assigned 250 indexes to loop over (Thread 1: 0-254, Thread 2: 255-499, Thread 3: 500-749, Thread 4: 750-999). Imagine that each thread encounters a “member” during the first for loop iteration, i.e. the array indexes 0, 250, 500, and 750 are all evaluated to be “good members”. What could happen in this case is that all four threads assign their good member index i to the good_members array at the same position, which is position 0. Therefore, not all good member variables would be stored in the array good_members as this first iteration already skipped 3 good members due to overwriting at the same array index (position 0).

In order to avoid this problem, the whole code section from line 14 to line 17 needs to be encapsulated in a critical section, i.e. a section that is only worked on by one thread of the thread team at a time, so that no more than one thread can store at the same array index (pos). Line 16, #pragma omp atomic, becomes unnecessary as the whole region is already marked as critical section so that the atomization of the pos increment is no longer required.

bug_hunting_2_solution.cpp > ...

```
1  /*
2   * The following code resolves the inefficiencies of the bug hunting
3   * problem of question 3b.
4   * Note: This code is not fully functional as the definiation of
5   * some of the given variables is missing.
6   */
7
8  #include <omp.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 #define N 1000
13
14 // Array of structs, defined elsewhere.
15 extern struct data member[N];
16
17 // Returns 1 if member[i] is a "good" member, 0 otherwise, defined elsewhere.
18 extern int is_good(int i);
19
20 int good_members(N);
21 int pos = 0;
22
23 void find_good_members()
24 {
25     #pragma omp parallel for
26     for (int i=0; i < N; i++)
27     {
28         if (is_good(i)) {
29             #pragma omp critical
30             {
31                 good_members[pos] = i;
32                 pos++;
33             }
34         }
35     }
36 }
```

Figure 7: Proposed solution for the bug hunting problem of question 2 of exercise 2.

Question 3: OpenMP Bug Hunting II

Question 3a)

Note: The assumptions for the correction of the code of this assignment sections were made according to the TAs explanations that we may assume that the code works correctly without any of the included omp library functions.

One important point that needs to be mentioned with regards to the serial version of the code, i.e. code without any omp library functions, is that the variable sum is updated to a new value with every for loop iteration. Therefore, sum holds a new value for every iteration. Furthermore, it is to say that the value of the variable t directly depends on the value of the iteration variable step. The function 'do_work(t, sum)' on line 31 on the assignment sheet clearly reveals that the combination of the specific values of sum and t throughout every iteration needs to be preserved. Every version of sum is provided with t as input to the 'do_work' function, where t depends on the step value of the previous iteration. The solution below preserves this input combination.

The first two for loops inside the 'step for loop' can be parallelized, but need not be parallelized to achieve a correct result. Whether or not these two for loops should be parallelized depends on the size of the iteration variables n and m. In case these are relatively small, a parallelization is probably not the ideal choice, as the overhead of spawning the nested thread team could take more time than executing the for loop in a serial manner. However, for big values of n and m a nested parallelization could be beneficial in terms of performance. Moreover, the number of available cores on the used machine needs to be taken into consideration. If the number of spawned threads in the nested thread team is greater than the number of available cores oversubscription is caused. Oversubscription can sometimes lead to a performance gain, as discussed in the lecture. The impact of oversubscription on the code performance should be tested on the used machine to gain an insight if its use results in a performance gain or decrease.

The code section that sums the values of the array z is included in a critical section. Due to the variable sum being globally defined, each thread has access to the same variable, i.e. same address in memory. In order to achieve the same result as the serial version of the code it is necessary to add to sum with every iteration and use it as argument for the 'do_work' function. If all threads have access to sum at any given time it cannot be guaranteed that the particular values of sum, after every iteration in the serial code, are used to do 'do_work()'. The 'do_work' function is thus encapsulated in the critical sectio. After the for loop is done computing sum for a thread 'x', no other thread is allowed to add to sum before thread x executes the do_work function. Otherwise, a false value of sum could be passed to 'do_work'.

In order to mimic the behaviour of updating the value of t a new variable cnt is introduced. As the serial code relies on specific combinations of sum and t being passed to 'do_work' it is not possible that a thread of any number executes do work after a particular step. For instance, assume that the variable t passed to 'do_work' was created by passing step=1 to 'new_value'. The next usage of 'do_work' is then forced to use the value of t generated with step=2. Thus, only step=2 is allowed to be used as input argument for the computation of the next t. As cnt is also updated in the critical section no issues can be caused concerning its incrementing.

It could be a performance boost to parallelize the loop that adds all values stored in z to sum. This may or may not be a useful implementation (refer to the discussion above if a nested parallelization is a reasonable choice in this case).

bug_hunting_3a_solution.cpp > ...

```cpp
/*
 * The following code resolves the bugs of the bug hunting problem
 * of question 3a.
 * Note: This code is not fully functional as the definiation of
 * some of the given variables is missing.
 */

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

void do_work(const float a, const float sum);
double new_value(int i);

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

    #pragma omp parallel for num_threads(<reasonable_num>)
    for (int step=0; step < 100; step++)
    {

        #pragma omp parallel for num_threads(<reasonable_num>)
        for (int i=1; i < n; i++)
        {
            b[i - 1] = (a[i] + a[i - 1]) / 2;
            c[i - 1] += a[i];
        }

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

        #pragma omp critical
        {
            #pragma omp parallel num_threads(<reasonable_num>)
            #pragma omp for reduction (+:sum)
            for (int i=0; i < m; i++)
                sum = sum + z[i];

            do_work(t, sum);

            t = new_value(cnt);
            cnt++;
        }
    }
}
```

Figure 8: Proposed solution for the bug hunting problem of question 3a of exercise 2.

Question 3b)

The code of question 3b is correct, but has some performance issues that can be resolved. The figure below illustrates two versions how to improve on the given code.

The first variant collapses the 'omp parallel' and 'omp for' statements into one 'omp parallel for' statement. This allows reducing the synchronization barriers of the threads. Specifically, the 4 synchronization barriers of the code are reduced to 2. Furthermore, the number of lines is reduced and makes the code more concise.

As nested parallelism should generally be avoided, although it can boost the performance at times, a second variant is proposed in the figure below. The second variant makes use of the concept of loop collapsing. As the two given for loops in the 'nesting' function are perfectly nested the precondition for loop collapsing is given. This variant makes the code more concise, reduces the number of synchronization barriers from 4 to 2, and avoids nested parallelism.

bug_hunting_3b_solution.cpp > ...

```cpp
/*
 * The following code resolves the inefficiencies of the bug hunting
 *  problem of question 3b.
 * Note: This code is not fully functional as the definiation of
 * some of the given variables is missing.
 */

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

/* Variant 1. Nested parallism. Reduction of sync. barriers. */
void work(int i, int j);

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


/* Variant 2. Loop collapsing. */
void work(int i, int j);

void nesting(int n)
{
    int i, j;
    #pragma omp parallel
    {
        // Collapse the two perfectly nested for loops.
        #pragma omp for collapse(2)
        for (i=0; i < n; i++)
        {
            for (j=0; j < n; j++)
                work(i, j);
        }
    }
}

```

Figure 9: Proposed solution for the bug hunting problem of question 3b of exercise 2.