1 Question 1

1.1 Task a)

The program was implemented as instructed and can be found attached to this submission.

1.1.1 Task a.1

An implementation based on `#pragma omp for nowait reduction(+: sum)` was chosen. In a parallel section before the loop, each thread is assigned its own random number generator with thread-dependent seed.

1.1.2 Task a.2

An implementation around `#pragma omp for nowait` was chosen. The random number generators were dealt with as in the task above. The individual threads' results were written into an unpadded vector `std::vector<double>`.

1.1.3 Task a.3

An approach identical to task a.2 above was chosen, except for padding the vector with the number of `double` vector elements necessary to obtain a separate 64-bit cache line for each thread.

1.2 Task b)

The results of running the compiled program on Euler and my own machine (a 2018 Macbook Pro) are plotted in figures 1 and 2. The impact on performance of false sharing is clearly evident in both plots. Also, it is in line with expectations that the performance of the implementation using padded vectors almost matches what the built in reduction method achieves.

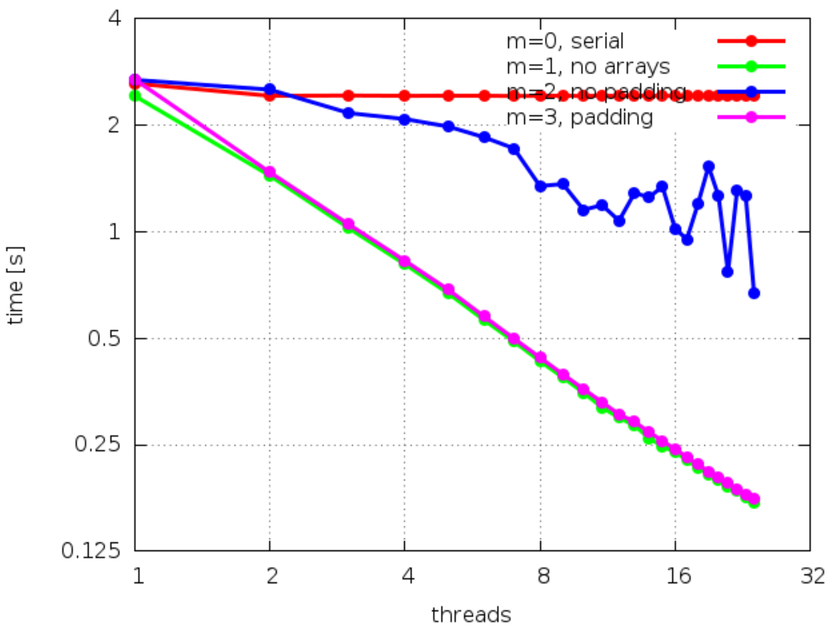

Figure 1: Parallel Monte Carlo integration on Euler compute cluster.

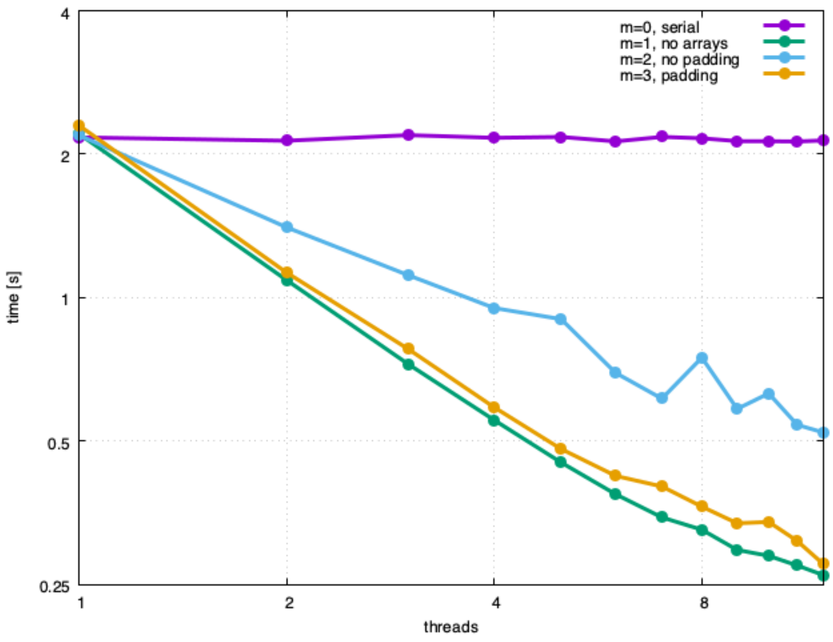

Figure 2: Parallel Monte Carlo integration on 2018 Macbook Pro.

1.3 Task c

- Yes, for the amount of tasks N significantly larger than the number of threads, the workload is distributed evenly among the threads. For smaller N this balance might not been exactly achievable.
- No, there is no perfect scaling as there is unparallelized overhead, but the modes `m=1` and `m=3` are not far off.
- A serial program and a single thread OpenMP program will get the same results provided the random generator seed remains unchanged. A multi-threaded OpenMP program on the other hand may show very small differences in the order of magnitude of machine precision when compared to the serial/single-thread results. This is due to the fact that each thread's random number generator is initialized with a different seed and also due to small numerical floating point rounding errors when adding the individual thread results.

2 Question 2

There is a problem with the global variable `pos` in the `if` statement starting on line 13: In order to write to the array `good_members`, the value of `pos` must be read in line 14. This read is not protected from race conditions with the atomic write on line 17. As `#pragma omp atomic` cannot be extended to several statements, a possible (but still ugly) solution would be to replace the `for` loop block from lines 12 to 19 with snippet 1.

Listing 1: Proposed modification for correctness in task 2.

```
#pragma omp parallel for
for (int i= 0; i < N; i++)
{
    if (is_good(i)) // No change until here
    {
        int thread_pos{0}; // Define thread specific position variable
        #pragma omp critical // This block replaces the #pragma omp atomic section
        {
            thread_pos= pos; // Protected read
            pos++; // Protected write
        }
        good_members[thread_pos]= i; // Use thread specific variable
    }
}
```