## 1. Parallel Monte Carlo using OpenMP

b) We benchmark the four implementations on our local computer and on the Euler cluster with up to 24 threads on each. We repeat the measurements once for the compiler flag $-O0$ and once for $-O3$ on both systems. The results can be seen in Figures 1 to 4. The behaviour on both our local machine and on the Euler cluster are very similar and look qualitatively roughly the same. Two distinct things to notice are that in the case of $-O0$ we can observe false sharing as the unpadded array implementation in both cases performs significantly worse than the other implementations. On the local machine the unpadded array implementation even performs worse than the serial implementation, up to 6 threads. Another thing to notice is that the implementation without array is performing better than the other implementations on our local machine, but no distinct improvement is noticeable on the Euler cluster.

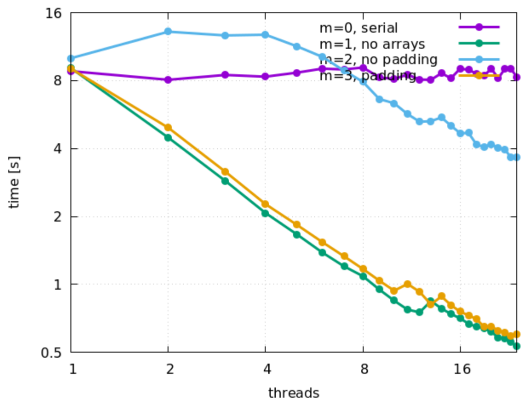

**Figure 1:** Scaling behavior on local computer for 4 different implementations. Parameter $-O0$.

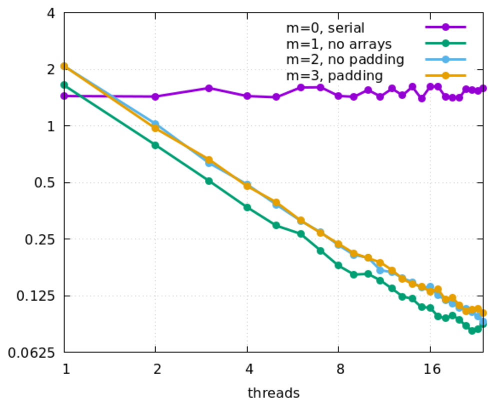

**Figure 2:** Scaling behavior on local computer for 4 different implementations. Parameter $-O3$.

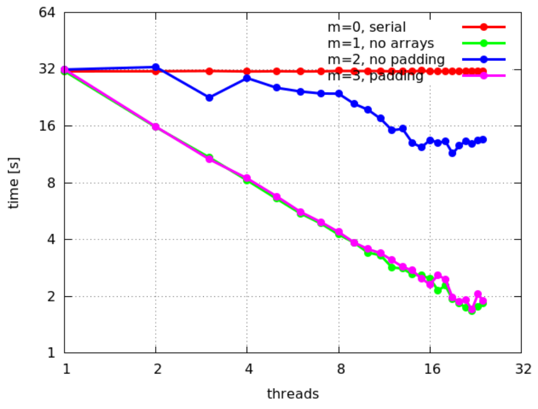

**Figure 3:** Scaling behavior on Euler cluster for 4 different implementations. Parameter $-O0$.

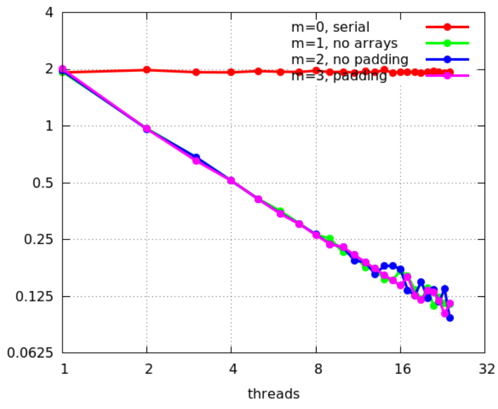

**Figure 4:** Scaling behavior on Euler cluster for 4 different implementations. Parameter $-O3$.

b) The work $N$ is equal among all threads $n$ as $N$ tends to infinity. We divide the number of times we draw our random numbers by all threads. We assume that every evaluation of the PRNG takes the same time and therefore every loop iteration as well. This means that the difference of the workload per thread gets negligible for large $N$, even when $N$ is not a multiple of $n$. The maximum load-imbalance is one loop iteration. The code is very close to achieving perfect scaling, but falls off a bit for higher thread numbers. We assume this is because of the overhead of spawning the different processes compared to the relatively small workload of $1e8$. On $24$ threads this leaves a merely $4$ million iterations per thread. Additionally, the spawning of the random number generator and setting the seeds in the implementation where only 'omp parallel for reduction' is allowed is done in the OpenMP master, which is serial. The serial fraction will become smaller as our workload $N$ increases, leading to a better scaling behavior. The numerical results we get are the same every time we run it on the serial code and with just one OpenMP thread. However, when running with more than one OpenMP thread, we have slightly different results every time. We assume this is because the order of summation in the reduction is different for different executions, leading to different numerical rounding and therefore also result.

## 2. OpenMP Bug Hunting I

We have a race condition with the variable `pos`, as the read access on line $14$ is not together with the increase of the variable on line $17$. A possible scenario with two threads $\mathcal{A}$ and $\mathcal{B}$ demonstrates the issue:

1. Thread $\mathcal{A}$ finds a good member $i$ at location $1$.
2. Thread $\mathcal{A}$ assigns 1 to `good_members[0]`.
3. Thread $\mathcal{B}$ finds a good member at location $512$.
4. Thread $\mathcal{A}$ assigns 512 to `good_members[0]`.
5. Thread $\mathcal{B}$ updates `pos` to $1$.
6. Thread $\mathcal{A}$ updates `pos` to $2$.

One possible solution would be to wrap the read and write of the `pos` variable in a critical section, like so:

```
if(is_good(i){
    #pragma omp critical
    {
        good_members[pos] = i;
        pos++;
    }
}
```

## 3. OpenMP Bug Hunting II

a) Analyzing the code in Question $3a$ we find the following bugs:

1 The arrays `a`, `b`, `c` and `z` are all neither declared nor initialized. This can be fixed by declaring and initializing the four arrays.

2 Every thread will execute the for-loop on line $13$ completely, which means that we'll have nested parallelism (I assume this is not intended). The nested parallelism can be removed by removing the `parallel` on line $15$.

3 There is a data race for the arrays `b` and `c` because of the `nowait` on line $15$ and the access in line $23$. Removing the `nowait` clause will fix this issue.

b) Analyzing the code in Question $3a$ we propose the following improvements: the `#pragma omp parallel` and `#pragma omp for` can be grouped into one OpenMP statement, namely `#pragma omp parallel for`. The best improvement is achieved when grouping all four OpenMP statements, thereby removing the nested parallelism. This can be done as each loop consists of only one statement. The resulting code would then read 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);
        }
    }
}
```