PDC Summer School - Report

Magnus B. Johansen

September 2023

1 Introduction

In this report, we consider the parallelization of an ”energy storms” code in the C programming languages using multiple strategies. First, we give some general remarks about the structure of the code, focusing on regions that require special attention for efficient parallelization. Second, we go through the changes made to the serial code for all three approaches and give a performance analysis for the parallel code, including some scaling plots. We also discuss further optimizations that could not be performed within the given time frame. Finally, we discuss further parallelization strategies one could have utilized to speed up the code.

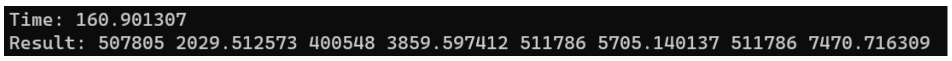
2 Remarks about the structure of the code

In the code, we first allocate and initialize the arrays containing the local energy maxima and their positions for each storm. Unless we are debugging, these are the arrays which are printed to the terminals at the end. As such, these are the only arrays which must be collected in a single place and not distributed across several processes or on a device. Conversely, the rest of the data can be distributed in this way. Secondly, 1-D arrays for two copies of the layer are allocated and initialized. Next a loop is started, in which the description of a storm is read from a file and applied to the entire array. Since the maximum is determined after each storm with the layer not being cleared, this must be done serially. This can be verified by running the serial version with the storms in different orders. Within each loop iteration, a loop over specific impacts is performed, with each of them potentially impacting the entire array. Since each impact necessitates accessing the entire array, this very quickly leads to data races for the shared memory approaches. As such, we also generally treat each impact serially. Finally, each element of the array is updated independently based on the impact, leading to a prime opportunity for parallelization. Next, the energy relaxation is handled. For the copying and stencil operation, there is no data dependency, meaning that they are readily carried out in parallel. For the distributed memory approach, however, halo cells are needed for this step. Finally, the maximum value and its position is determined. This last part constitutes a classic reduction with the caveat that the position must be tracked as well.

From this it is clear that we should process each impact separately to avoid data races while delegating the act of accessing the array. Furthermore, special attention should be given to the stencil as well as the reduction. All of the parallel implementations should handle these problems.

3 Sequential baseline

In order to have a significant workload for the parallelization to have an impact, the largest set of unique test cases were run. This consisted of all four test files with the pattern `test_07_a1M_p5k_w[1-4]` with a layer size of 6000000. The output for the maxima and their positions was exactly the same for all implementations, showing that the implementations are indeed equivalent. Running the sequential code resulted in the following output:



```
Time: 160.901307
Result: 507805 2029.512573 400548 3859.597412 511786 5705.140137 511786 7470.716309
```

Thus, the rest of the results should be compared with the sequential baseline of **160.9** seconds. Furthermore both this code and the three others presented were compiled using the `-O3` and `-ffast-math` flags, both of which yielded significant performance increases while only yielding slightly different results from the unoptimized versions.

4 OpenMP Multicore

4.1 Short description of the implementation

For this approach, the approach was simply to add compiler directives of the type `#pragma omp parallel` to the loops in question. As the loop iterations were simply spread out across the threads, there was no need to consider most of the loops in more detail. However, in order to optimize the reduction one had to go beyond the simple operations. If one were only interested in the maximum, a simple `reduction` clause would be enough. Since the position was also needed, more care has to be taken. An approach may be to use atomics, but this would impact the performance quite a bit. Since we are only interested in carrying out the reduction on floats but also want to have an index, the best approach would be to declare a structure consisting of both and then defining a new reduction. This can be done using the following pragma:

```
#pragma omp declare reduction(max_idx : struct maxn : omp_out = omp_in.val >
                    omp_out.val ? omp_in : omp_out)
```

where the maximum is determined based on the `val` member variable. With this, we can simply use the standard `reduction` and get the maximum and index from the final structure. Finally, since the creation of parallel regions result in some extra overhead, multiple loops were combined in single parallel regions.

4.2 Performance analysis

In evaluating the results of the optimization, it is quite important to have the correct reference. For multicore approaches, it is often natural to compare the results obtained using multiple cores to that obtained using one core using the same binary. However, due to changes in the algorithm or added parallel overhead, the time to solution for the binary optimized for multicore run using one core is often higher than for the optimized serial code. Here we show both comparisons. In order to analyze the performance, we have performed a strong scaling analysis with an increasing number of threads. The results compared to the parallel version are shown in Fig. 1.

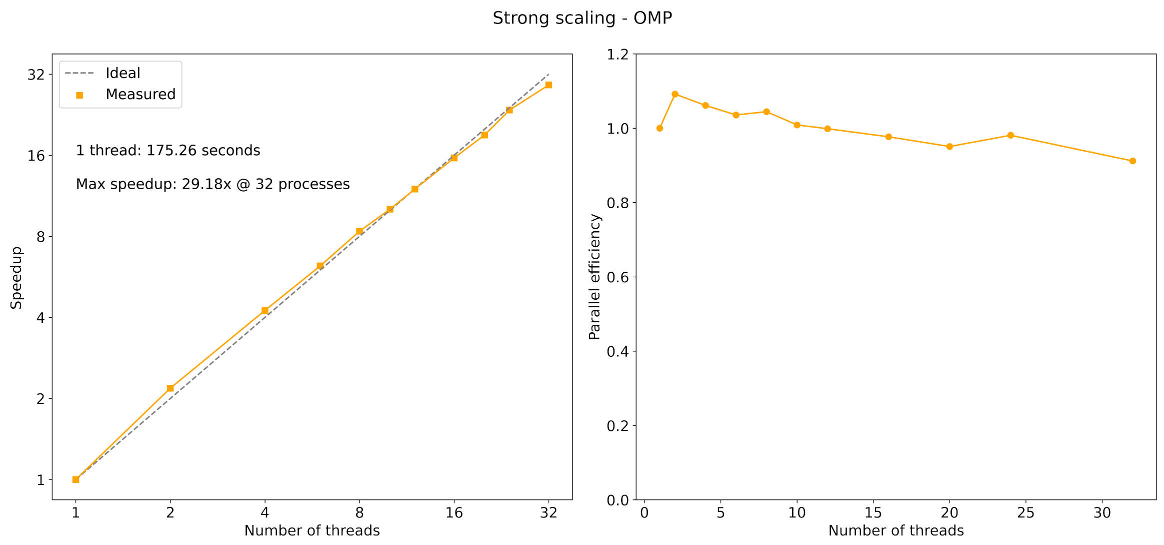


Figure 1: Strong scaling results for the OMP implementation. Note that the speedup is compared to the OMP version running on a single thread.

We see superlinear speedup for the lower thread numbers, which gradually gives way to dropping parallel efficiency due to the larger amount of overhead, e.g. cache consistency that inevitably comes from shared memory parallelism. As mentioned, this apparent superlinear scaling is probably the result of more overhead for the 1 thread run compared to the serial. Comparing instead against the sequential version gives results shown in Fig. 2.

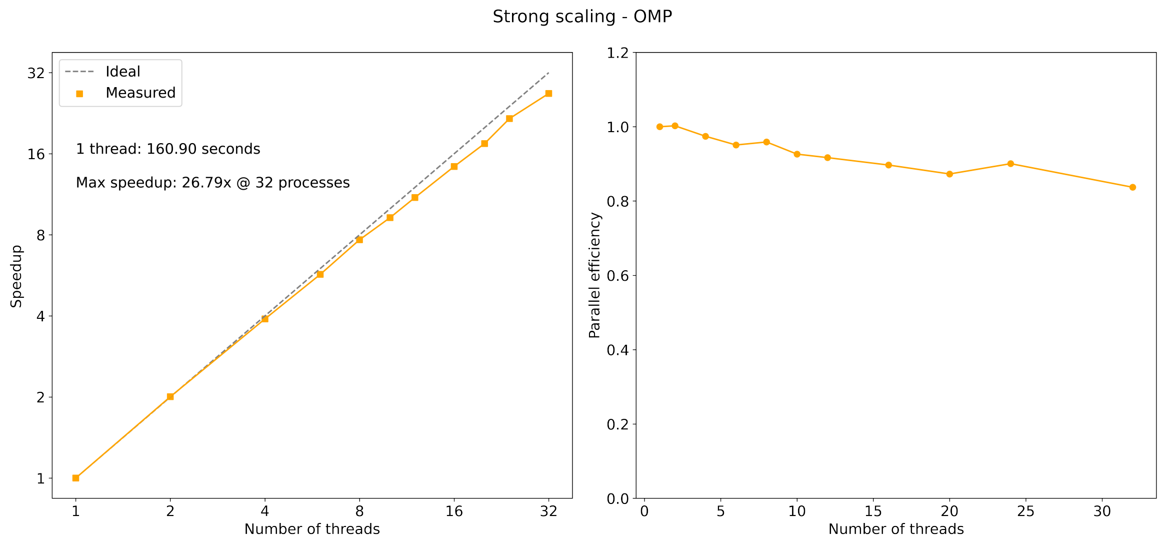


Figure 2: Strong scaling results for the OMP implementation. Note that the speedup is compared to the sequential version.

Here we see no superlinear speedup, only gradual degradation due to mounting overhead. While we are still able to get great performance from increasing the amount of threads, we are facing rapidly diminishing returns, indicating that this version should not be run on many more threads.

5 MPI

5.1 Short description of the implementation

For the MPI parallelization, a domain decomposition was chosen. Accordingly, both the `layer` and `layer_copy` arrays are broken into chunks based on the rank in the simplest possible manner. If the size of the array is not divisible by the number of processes, the last rank is assigned the remaining elements. When processing the impact of a particle, the global position is translated into a local position to preserve the distance. Just before the stencil operation, the halo cells are communicated. First, every rank other than the first initiates a non-blocking receive from the rank below followed by a send from every rank other than the last, effectively transmitting the right halo cell. Afterwards, a similar process is performed to transmit the left halo cell. Finally, for the reduction, an approach similar to that used for OMP is used. MPI natively supports the use of a structure consisting of a float and an integer, `MPI_FLOAT_INT`. Using this together with a `MPI_REDUCE` call with the operation `MPI_MAXLOC` achieves the same results as that used for OMP.

5.2 Performance analysis

In comparing the performance of the sequential and MPI code, we must again be clear on the basis for comparison. Running the MPI code with one process results in close to the same (± 2s) performance as the serial code, making a straightforward comparison very natural. Like for OMP, we have chosen to do strong scaling to look at the performance improvement. Here, we chose to do scaling for up to 256 processes on four seperate nodes. The results are analyzed both with regard to speedup and parallel efficiency. The results are shown in Fig. 3.
As is evident from the figures, the parallel implementation is quite efficient, achieving superlinear

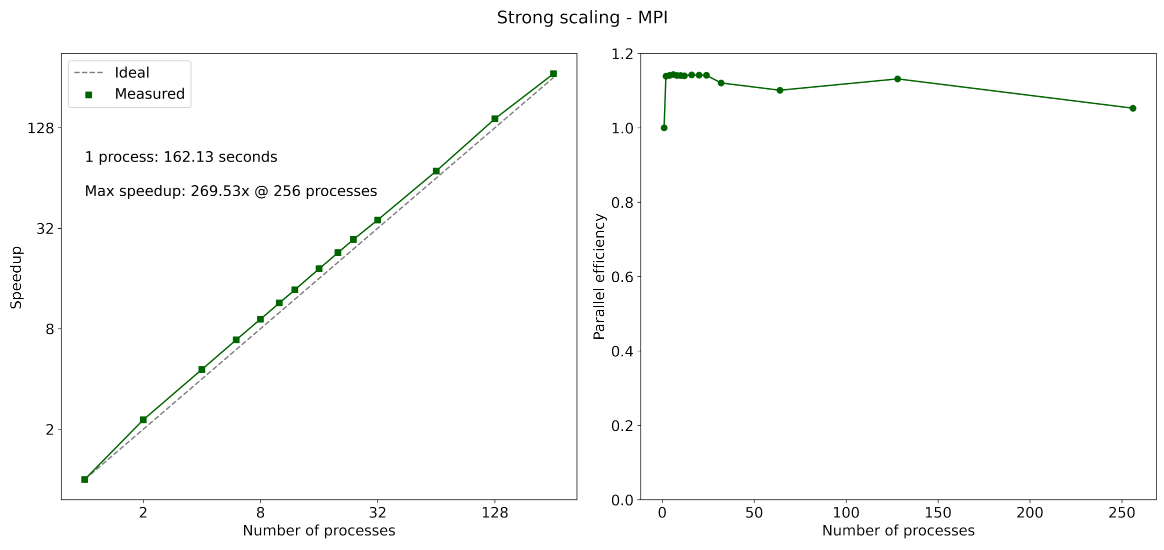


Figure 3: Strong scaling for the MPI code.

speedup across a large amount of ranks and even nodes. This is a reflection of the almost embarrassingly parallel nature of the underlying problem. The only MPI calls are to send/receive two floats and perform a parallel reduce. Furthermore, the static domain decomposition seems to be adequate for load balancing purposes as the extra terms in the final rank do not seem to impact the scaling much. The eventual drop in parallel efficiency is most likely due to the increased latency when communicating across the network as well as the increased communication overhead from the many processes. One way to avoid this using pure MPI might be to create multiple communicators, handling different sections of the array. This would lower the amount of synchronization across all processes.

6 GPU programming using HIP

6.1 Short description of the implementation

Since only the `maximum` and `positions` arrays are used for output, it is much more efficient to allocate and initialize the layer arrays on the GPU rather than copying them in from the host. Then, every operation involving the layers needs to be defined as kernels launched from the host using the `__global__` identifier. Each kernel launches a number of threads designed for the input length, working in parallel by using the thread and block IDs. Due to the seperate layer arrays, there are no problems with data races for most of the loops. However, similar to the problems with the other methods, a parallel reduction for the maxima is tricky to manage. For this report, a version using atomics was implemented. However, this resulted in bugs when running some of the tests. Since the serial version also turned out to be faster than the parallel approach, it was decided to stick with the serial version.

6.2 Performance analysis

Unlike OMP and MPI, there is no straightforward way to do scaling for the HIP problem. As such, we simply report the result obtained from running the test case used for all the others. The result was **8.41 s** which represents a speedup of **19.1x**. One way to check whether the reduction implementation represents a bottleneck is to run `test_08`. Doing so reveals that the HIP implementation is much slower than even the sequential version. Thus to improve the performance, this aspect should be a focus. The optimal approach would be to store intermediate results in shared memory and iterating until only one result is left. While this is most straightforward if only the max is needed, it should be possible (like for OMP and MPI) to make a structure which also keeps track of the position. However, this could not be completed in time for submission.

7 Comparison of the different approaches

Here we compare the results of the three parallelism approaches. Looking at the performance, we see that while the GPU speedup is good at ∼19x, both the other approaches are able to reach that point with enough computational resources and even go further beyond. With both the MPI and OMP implementations achieving superlinear scaling, it is clear that these two methods are to be preferred. The most performant strategy for the particular problem may be a hybrid approach, where the array is broken into chunks running on different processes and all loops are distributed over multiple threads. This minimizes the amount of parallel overhead for both MPI and OMP, allowing a proper balance for larger systems. This would involve some tweaking for optimal performance, but that is also the case for many other approaches such as GPUs. To improve on the GPU performance it is crucial to optimize the reduction as this is the major bottleneck for the current implementation.

8 Further work

Of the other parallelization techniques not described within this report, the technique of GPU offloading is the most prominent. Similar in scope and application to OMP, it allows for the offloading of specific loops and operations onto an accelerator. However, as we have seen, the optimal data movement and handling of threads required quite a bit of oversight. Since the offloading does not give this level of fine-grained control, it is not too promising for increasing the performance at present. It also remains to check whether the performance observed here is portable to different platforms. It would especially be interesting to translate the HIP version to CUDA to see whether the bottlenecks observed here would persist.