Final Project

Pull request: https://github.com/TUM-I5/SWE/pull/12

How to build/run our project

Add environments variables
export NETCDF_LIBRARIES=$(nc-config --libs)
export NETCDF_INCLUDES=$(nc-config --includedir)

Load necessary modules
module load cmake
module load gcc/11.3.0
module unload openmpi
module load intel-mpi/2021.9.0
module load netcdf-hdf5-all/4.7_hdf5-1.10-gcc12-impi

Initial Profiling

local machine CPU info: …

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  20
  On-line CPU(s) list:   0-19
Vendor ID:               GenuineIntel
  Model name:            12th Gen Intel(R) Core(TM) i7-12700H
    CPU family:          6
    Model:               154
    Thread(s) per core:  2
    Core(s) per socket:  14
    Socket(s):           1
    Stepping:            3
    CPU(s) scaling MHz:  15%
    CPU max MHz:         4700.0000
    CPU min MHz:         400.0000
    BogoMIPS:            5376.00
```

Profiling on Intel Vtune

The original code is profiled with Intel Vtune on a local machine with the parameters mentioned above. The number of MPI processes is set to 10. The profiling analysis is performed in “Hotspots” mode. The number of grids on x and y dimensions are set

to 1000 each. This profiling is named as the baseline for following analysis.

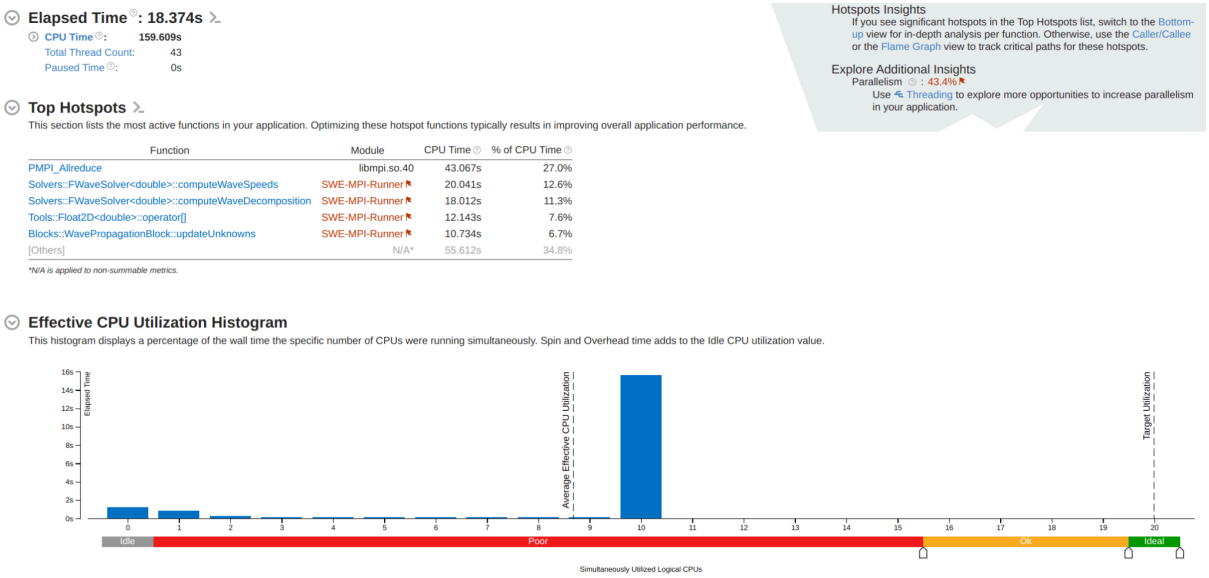


Figure 1: Baseline profiling result in V-tune with 10 MPI processes.

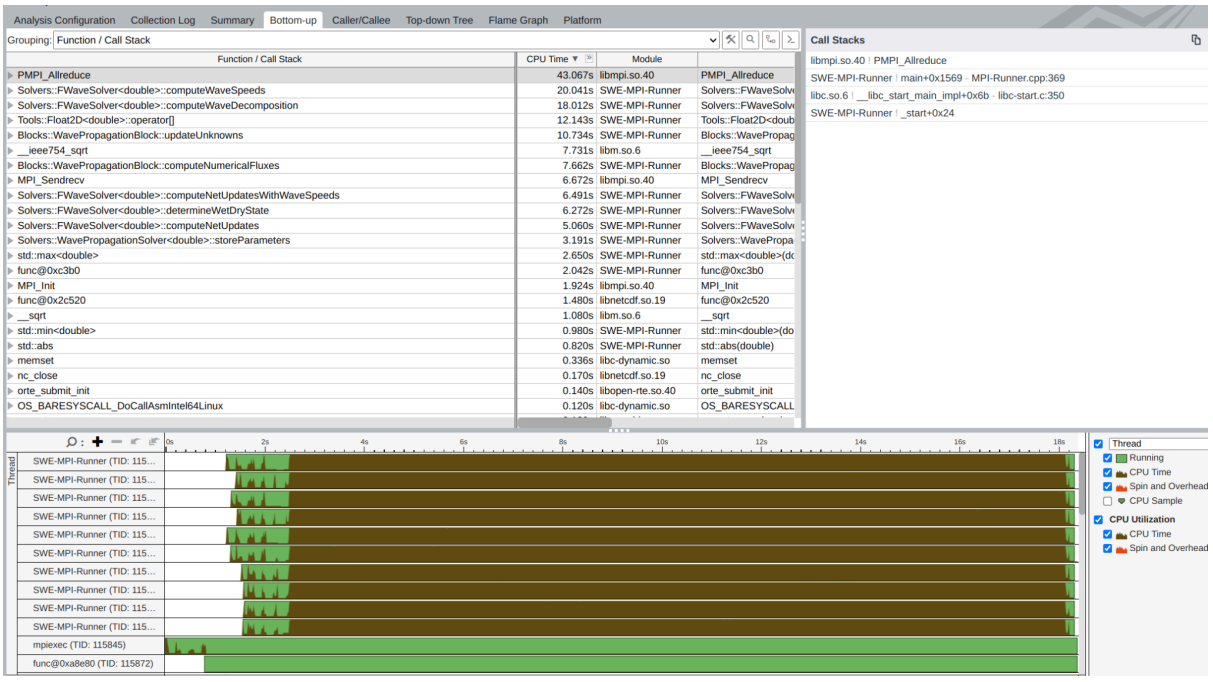


Figure 2: Rank of hotspots in the original code.

Callgrind

Callgrind is a profiling tool in the Valgrind suite that uses instrumentation to analyze program performance. It focuses on function calls and execution paths and helps identify critical sections of the code.

Sequential case

This tool was executed only locally on Linux with CPU because it was not possible to load it on the remote computation node.

Below is a call graph of the process `**./build/SWE-MPI-Runner -x 500 -y 500**` produced by the visualization tool Kcachegrind. This code was **executed in a serial manner**.

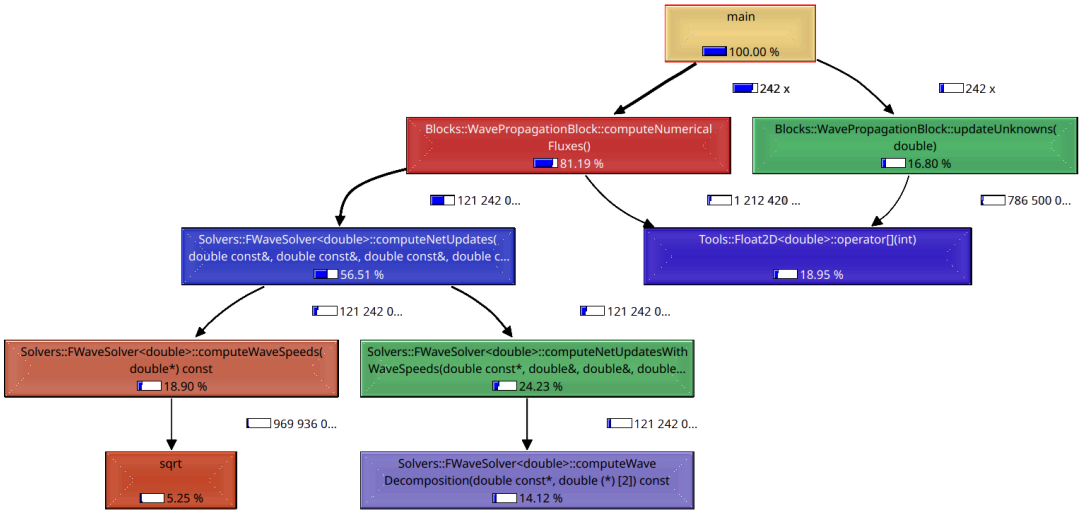


Figure 3. Relative Call graph of the SWE project

Incl.	Self	Called	Function	Location
99.98	0.00	1	main	SWE-MPI-Runner: MPI-Runner.cpp, new_allocator.h
81.17	12.11	242	Blocks::WavePropagationBlock::computeNumericalFluxes()	SWE-MPI-Runner: WavePropagationBlock.cpp
56.50	7.22	121 242 000	Solvers::FWaveSolver<double>::computeNetUpdates(double const&, double...	SWE-MPI-Runner: FWaveSolver.hpp
24.22	9.03	121 242 000	Solvers::FWaveSolver<double>::computeNetUpdatesWithWaveSpeeds(doub...	SWE-MPI-Runner: FWaveSolver.hpp
18.99	18.99	2 003 334 870	Tools::Float2D<double>::operator[](int)	SWE-MPI-Runner: Float2D.hpp
18.90	11.49	121 242 000	Solvers::FWaveSolver<double>::computeWaveSpeeds(double*) const	SWE-MPI-Runner: FWaveSolver.hpp
16.79	9.34	242	Blocks::WavePropagationBlock::updateUnknowns(double)	SWE-MPI-Runner: WavePropagationBlock.cpp
14.12	13.46	121 242 000	Solvers::FWaveSolver<double>::computeWaveDecomposition(double const*...	SWE-MPI-Runner: FWaveSolver.hpp

Figure 4. Table of the functions from the Call graph

The profiling results are not exactly the same because of different processors and architectures. However, they provide valuable information about the candidates for optimization.

- FWaveSolver
  - computeNetUpdates
  - computeNetUpdatesWithWaveSpeed
  - computeWaveSpeeds
  - computeWaveDecomposition
- WavePropagationBlock
  - computeNumericalFluxes
  - updateUnknows

Optimization 1 - resolving load imbalances from wetting/drying with openMP

The function handling wetting and drying is located in build/_deps/swe-solvers-src/Source/FWaveSolver.hpp
The function, determineWetDryState(), determines the state of a single edge between two cells of the simulation. From the VTune output in figure 2, we see that this function has a CPU time of 6.272s. The whole program has a CPU time of 159s, so this function constitutes 4% of the total cpu time.

To get more insight we timed the execution of the function manually with "std::chrono::high_resolution_clock". The result of this timing can be seen in figure 5. It is apparent from the spikes in the graph, that some calls to determineWetDryState() take a lot longer to execute than most other. Importantly, since we are timing the execution of one function call on its own, this does not show a load imbalance, as that depends on how the work is distributed among the processes. However, if one thread had to handle more spikes than the others that could cause a load imbalance.

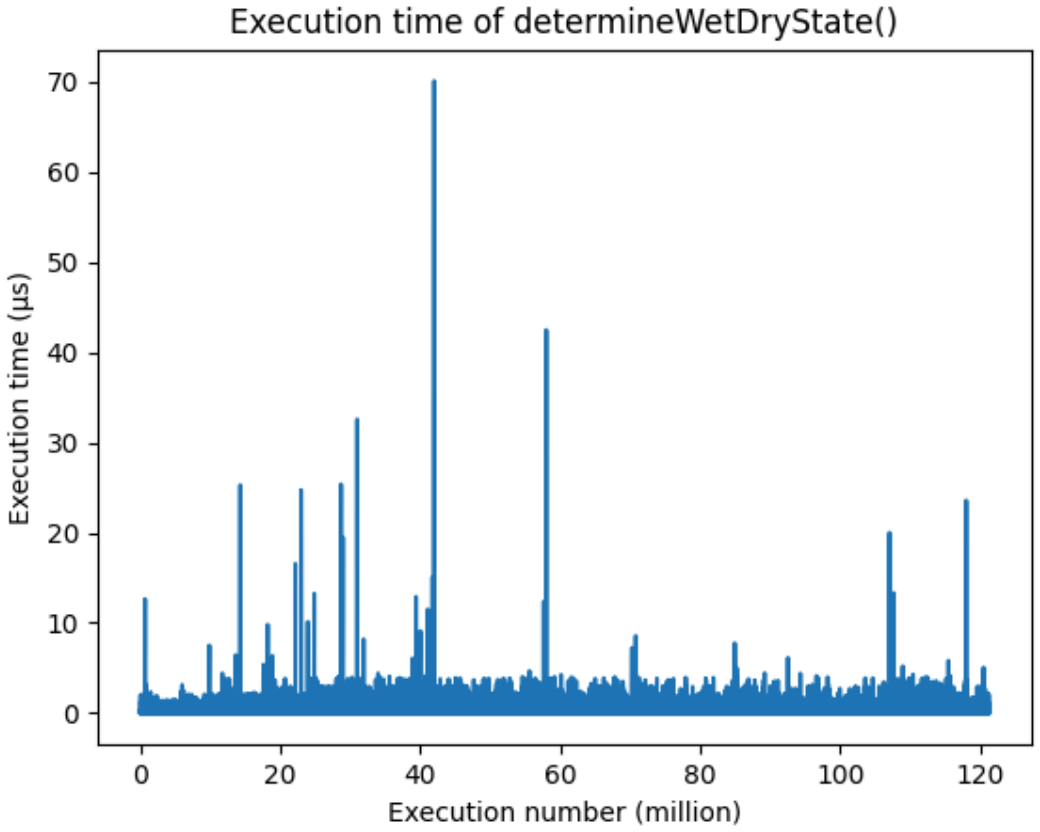


Figure 5: the execution time of all calls to determineWetDryState(). The specific call to the function is on the x axis, and spikes indicate that some calls take much longer than the others.

To profile the function and get information about the load imbalance we first tried to use likwid with the performance groups FLOPS_DP and FLOPS_SP. However, after running the command “likwid-perfctr -g FLOPS_DP -g FLOPS_SP mpiexec -np 30 ./SWE-MPI-Runner-reference -x 500 -y 500“ we got the error “Unsupported Processor”. When referencing the supported architectures list [1] it does not contain the microarchitecture golden clove used by the saphire rapids Intel Xeon Platinum 8480+ Processor that is used on CoolMUC4.

This result made us next try perf, using the “seconds time elapsed” to time every thread and observe if there was large discrepancies. However, on the cluster we got “perf: command not found” even after loading “likwid/5.2.2-gcc12-perf”. We were not able to run it locally either, getting the error “access to performance monitoring is limited”.

Therefore we next tried to use vallgrind to achieve something similar, getting the instruction count for each thread to see if there is a load imbalance. The results are shown in figure 6. The results from vallgrind was that every thread spent the exact same amount of instructions on determineWetDryState(), 3811500 for a grid of 250x250. The chance that every thread is running the exact same number of instructions are very slim, so we therefore probably have done something wrong in our profiling.

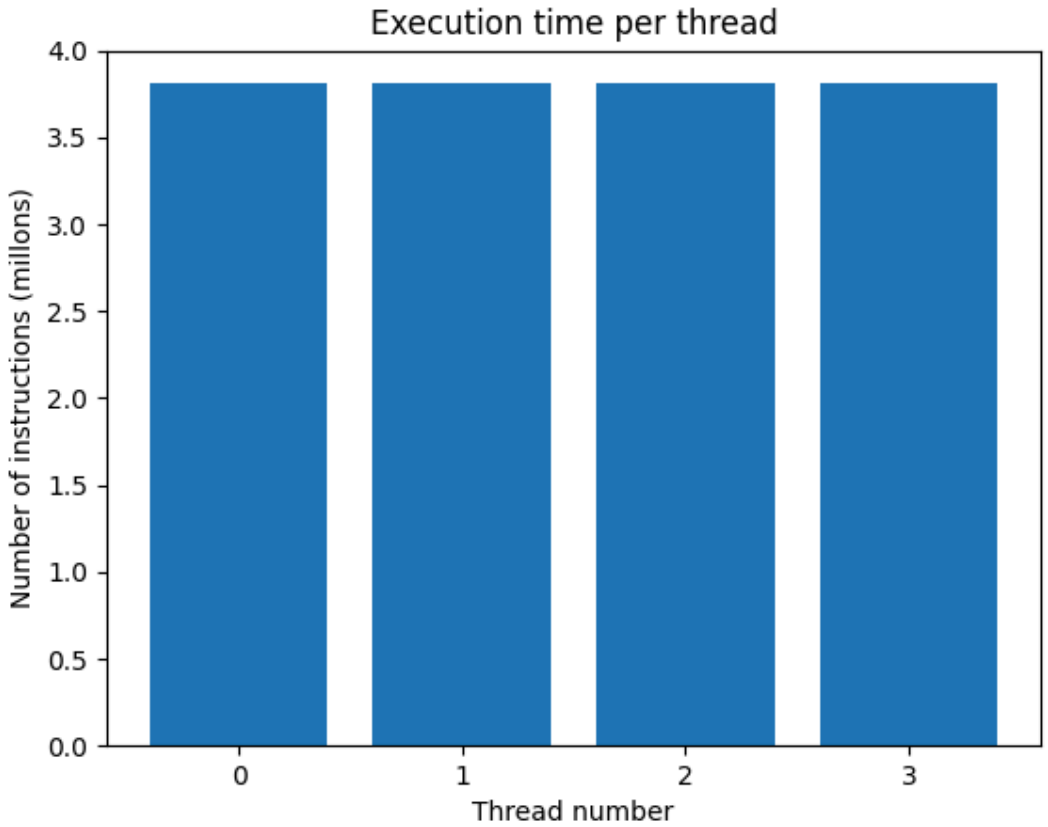


Figure 6: Number of instructions per thread

Due to these result we opted to use a manual method where we would time the function, much like in figure V, but include the PID so we can sum up the work per process. The work per process can be seen in figure 7. We ran this for x 1000, y 1000.

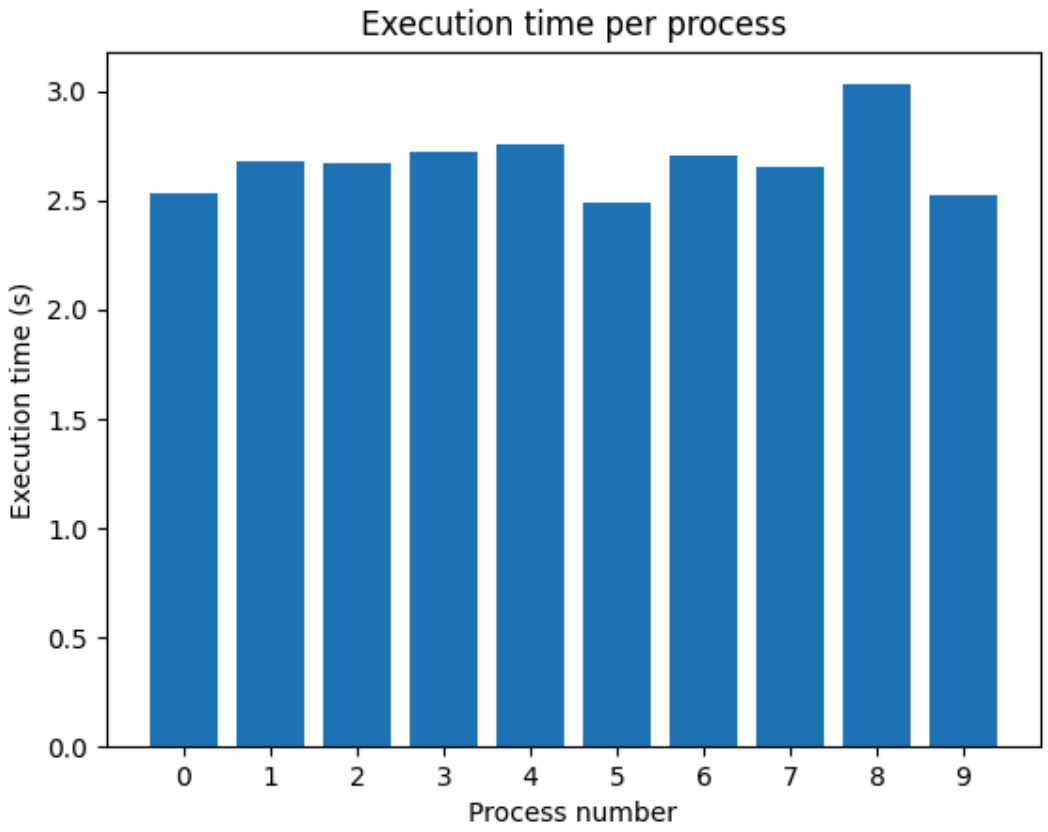


Figure 7: execution time per process, measured in CPU time.

The reason for this not adding up to the CPU time seen in Vtune, is that our code to measure the function time slows down the program quiet a lot. However, this should be the same for all threads, and the load imbalances should still be visible. The larges difference in load was 0.54s between thread 5 and 0.

For our load imbalance code we wanted to add openMP dynamic scheduling to the for loop calling determineWetDryState(). determineWetDryState is called once by computeNetUpdates() in build/_deps/swe-solvers-src/Source/FWaveSolver.hpp, which is the function responsible for computing the updates for one cell.
computeNetUpdates() is called in a nested for loop in Source/Blocks/WavePropagationBlock.cpp

```
void Blocks::WavePropagationBlock::computeNumericalFluxes() {
RealType maxWaveSpeed = RealType(0.0);

for (int i = 1; i < nx_ + 2; i++) {
  for (int j = 1; j < ny_ + 1; ++j) {
    RealType maxEdgeSpeed = RealType(0.0);

    wavePropagationSolver_.computeNetUpdates(
      …
    );
    maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
  }
}
for (int i = 1; i < nx_ + 1; i++) {
  for (int j = 1; j < ny_ + 2; j++) {
    RealType maxEdgeSpeed = RealType(0.0);

    wavePropagationSolver_.computeNetUpdates(
    …
    );
    maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
  }
}
…
```

Content of computeNumericalFluxes. Arguments for compute net updates as well as the end of the function are not shown to keep it somewhat brief.

We used openMP pragmas to collapse the for-loops and add dynamic scheduling. Changes are shown in bold.

```
void Blocks::WavePropagationBlock::computeNumericalFluxes() {
 RealType maxWaveSpeed = RealType(0.0);

 #pragma omp parallel
 {
   RealType maxWaveSpeedLocal = RealType(0.0);
   #pragma omp for schedule(dynamic) reduction(max:maxWaveSpeedLocal)
   for (int i = 1; i < nx_ + 2; i++) {
     for (int j = 1; j < ny_ + 1; ++j) {
       RealType maxEdgeSpeed = RealType(0.0);

       wavePropagationSolver_.computeNetUpdates(
…
       );
       maxWaveSpeedLocal = std::max(maxWaveSpeedLocal, maxEdgeSpeed);
     }
   }
   #pragma omp for schedule(dynamic) reduction(max:maxWaveSpeedLocal)
   for (int i = 1; i < nx_ + 1; i++) {
     for (int j = 1; j < ny_ + 2; j++) {
       RealType maxEdgeSpeed = RealType(0.0);
       wavePropagationSolver_.computeNetUpdates(
…
       );
       maxWaveSpeedLocal = std::max(maxWaveSpeedLocal, maxEdgeSpeed);
     }
   }
   #pragma omp critical
   {
     maxWaveSpeed = std::max(maxWaveSpeed, maxWaveSpeedLocal);
   }
 }
…
```

After doing this optimization we once again ran our manual load imbalance program. The results of our manual run can be seen in figure 8.

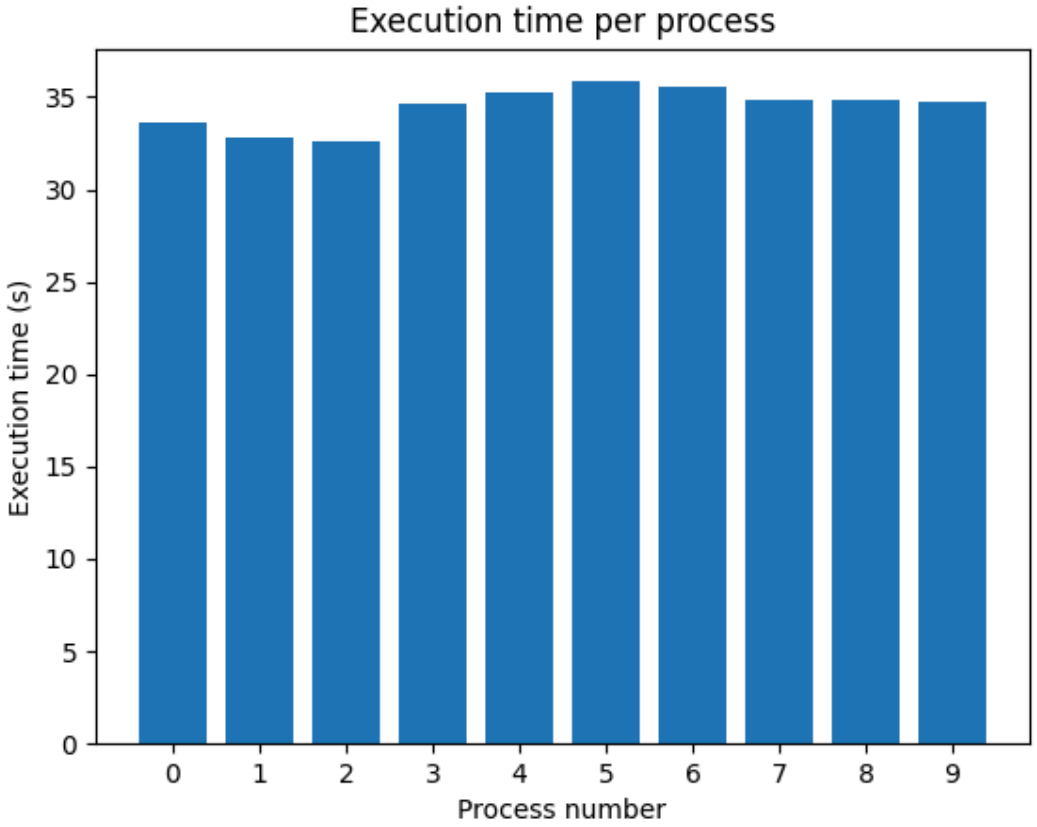


Figure 8: Execution time per thread after dynamic scheduling

While the load this time is much more balanced, the execution time per thread is also much higher (~10x). We think this might be because of the critical section in the dynamically scheduled code, which we have seen in assignment 2 to overall slow down code for many processes. However, when we try to run the code with a reduction instead we get some floating point exceptions that we are unable to mitigate.

computeNumericalFluxes

```
cmake -DENABLE_OPENMP=ON ..
export OMP_NUM_THREADS=4
```

```
Mon Jan 20 11:23:14 2025    Simulation finished. Printing statistics for each process.
---------------------------------------------------------------
Mon Jan 20 11:23:14 2025    Process 0 - CPU Time: 135.085 seconds
```

Optimization 2 - Single-Core Optimizations

The tests were conducted sequentially because, in this part, we are focusing on **single-core.**

updateUnknowns

Location: *Source/Blocks/WavePropagationBlock.cpp*

Function Description

The function is part of the simulation solver and is responsible for updating the cell averages of the simulation grid based on the computed updates. Specifically, it updates the height h, horizontal momentum hu, and vertical momentum hv of each cell in the grid. Using `_mm256_cmp_pd` and `_mm256_blendv_pd` eliminates the need for conditional branches and improves execution efficiency.

Optimization description

__m256d is a data type provided by Intel's AVX (Advanced Vector Extensions) intrinsics. It represents a **256-bit SIMD register** that can store and operate on four double-precision (64-bit) floating-point values simultaneously. These registers allow vectorized computations, meaning a single instruction can process multiple data elements in parallel.

We have rewritten this method using the instructions:

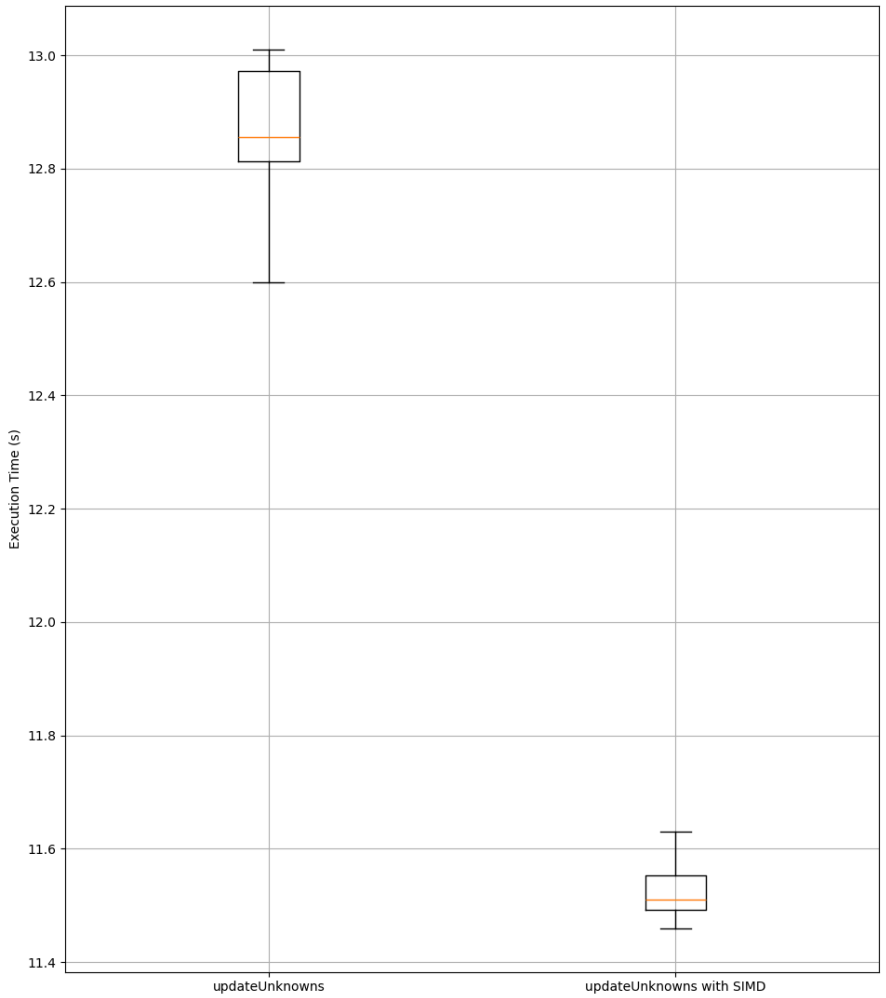
- load/store: _mm256_loadu_pd, _mm256_storeu_pd
- arithmetic operations: _mm256_add_pd, _mm256_mul_pd, _mm256_sub_pd
- handling of conditions:
  - _mm256_cmp_pd: compare all four elements in the h vector against a value to generate a mask
  - _mm256_blendv_pd: the mask generated by the previous function is then used to update values selectively:

Results

Testing command: ./build/SWE-MPI-Runner -x 500 -y 500
Machine specs: AMD Ryzen 7 5800H

baseline	12.60	12.64	13.87	12.86	12.81	12.82	14.60	12.85	12.86	13.01
simd	11.56	11.49	11.47	11.53	11.46	11.51	11.51	11.75	11.50	11.63

`FWaveVecSolver` has an average execution time of **13.09 seconds**
The optimized solver has an average execution time of **11.54 seconds**



computeNetUpdates

File Location: *swe-solvers/Source/FWaveVecSolver.hpp*

This piece of code is a lot, as seen in the initial profiling result. Combined with the function calls to `fWaveComputeWaveSpeeds` and `fWaveComputeWaveDecomposition` it takes around 52% of the computation. However, this code is not part of the `SWE` project. We have made a fork on the `SWE-Solvers` to be able to make changes. The main branch of the `SWE` uses `FWaveSolver, but it also uses` optimized `FWaveVecSolver.` Below, we compare the differences between them.

Description of the existing optimizations

SIMD Directives: `FWaveVecSolver` uses `#pragma omp declare simd` to indicate that the functions `computeNetUpdates`, `fWaveComputeWaveSpeeds`, and

`fWaveComputeWaveDecomposition` should be vectorized, allowing the compiler to generate SIMD instructions

Reduced Redundant Computations: `FWaveVecSolver` avoids the direct computation of values. Instead, the calculated values are stored as member variables and accessed multiple times, which reduces the number of floating point operations and square root computations. A good example of such an approach is the computation of `roeSpeed0` and `roeSpeed1`.

Minimal Use of Arrays: `FWaveSolver` stores results in arrays, while the `FWaveVecSolver` uses individual variables. This can help the compiler generate more efficient SIMD.

Inline of Wetting/Drying: `FWaveVecSolver` directly inlines the logic for determining the wet/dry state.

Attempts at optimization

Square root caching: Computing square roots is the most expensive operation in this module. In our attempt, we stored the computed values of the square root using a hashmap. This significantly reduced the number of calls to compute the square root; however, the hashmap was too heavy, leading to a performance decrease.

Branching reduction: branching of the program can be reduced by applying the operation used in the optimization of `updateUnknowns` by applying _mm256_cmp_pd with a combination of _mm256_blendv_pd. However, SIMD mask instructions are applied to multiple data points in parallel (4 or 8). Since the solver processes single values, we did not use the remaining values of the operations, which achieved branchless execution but did not bring performance benefits. The number of instructions should still be the same, but the branching, if only if-else, is relatively straightforward, so the compiler might not have a problem handling this.

Prefetching values: We tried prefetching values (e.g., hLeft, hRight, huLeft, huRight, bLeft, bRight). Performance did not benefit from this, which is most likely because only a few values fit into the CPU cache without explicit prefetching. If the data is already in the cache or the memory access pattern is predictable, the overhead of prefetching can outweigh the benefits.

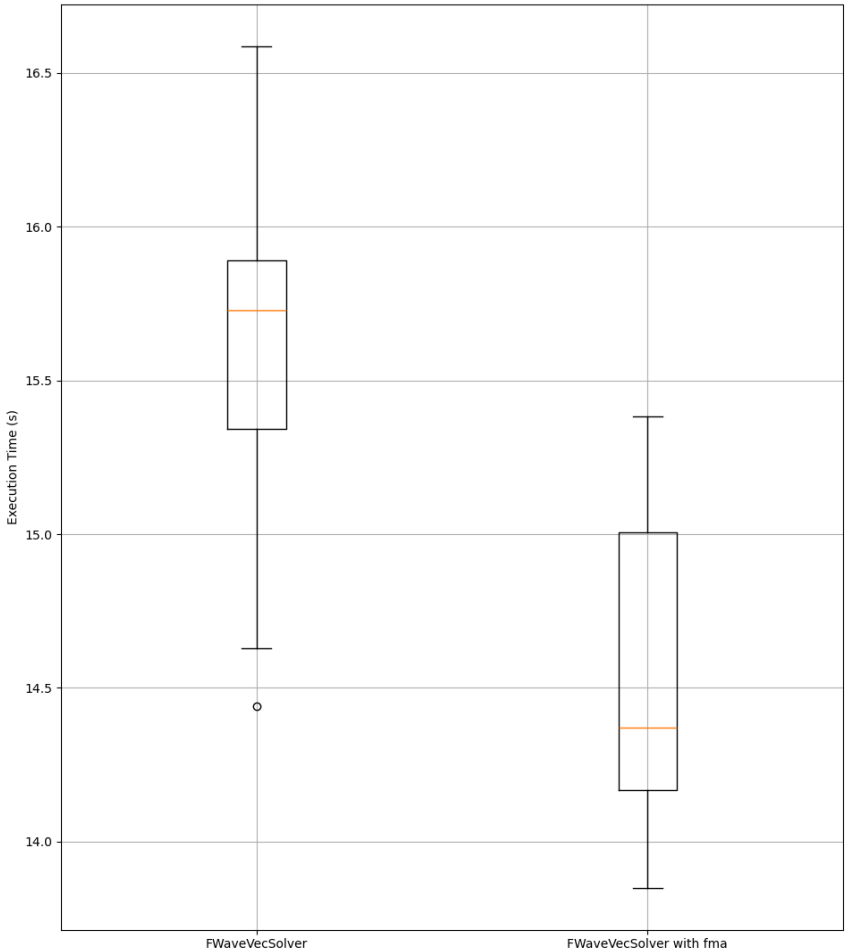
Fused multiply-add operations: Using instructions std::fma, we can combine multiplication and addition into a single operation in numerous module places. This effectively reduces the number of instructions, and we have seen performance increase for this one.

Results

Testing command: ./build/SWE-MPI-Runner -x 500 -y 500
CPU: AMD Ryzen 7 5800H

In the outcome the only modification that made sense was introduction of `fma` operations and inlining of functions `fWaveComputeWaveSpeeds`, `fWaveComputeWaveDecomposition`. We compared the implementation by averaging the average CPU time of the simulation.

nofma	15.79	14.62	15.78	16.58	15.29	15.92	15.67	16.01	15.49	14.43
fma	15.06	14.31	14.43	14.14	14.25	15.38	13.99	14.85	15.05	13.84



`FWaveVecSolver` has an average execution time of **15.56 seconds**
The optimized solver has an average execution time of **14.53 seconds**

This modification speeds up the code.

Tools::Float2D

File Location: *Source/Tools/Float2D.hpp*

Function Description

The Float2D module is a convenience helper class that handles 2D float arrays. It provides a simple way to access and manipulate elements in a 2D grid. However, it is called many times in the simulation, so making it as efficient as possible might be reasonable. Specifically <double>operator[](int) from the Valgrind profiling report, this operation takes around 19% of compute time.

Optimization Description

Since the code is already very lightweight, there are limited options for optimizing it. We inlined the function calls and precomputed the pointers. Instead of recalculating `data_ + (rows_ * i)` every time we store the result in `columnPointers_[i],` which is computed in parallel to minimize the overhead.

Results

When measuring and comparing the execution time, the difference was at the level of statistical error. For this reason, we ran the callgrind again to estimate the number of instructions. The result is that the initial 18.95% was lowered to 17.01%, which makes it not the most effective but definitely suitable optimization.

Optimization 4 - Loop fusion in wave propagation

According to the profiling results, a significant hotspot of the program is calling the solver for updating the values on the network. The solver is called by the for-loops over each grid point in the block and through the whole domain. In an individual loop, the current value of wave height h, wave discharge hu and hv, and bathymetry are passed into the solver to compute the update for next time step. As these quantities are processed individually in each loop of calling the solver, the for-loop can be parallelized by OMP directives and vectorization instructions.

To speed up the solver, the first thing to do is to switch to the `FWaveVecSolver` where the functions are already declared for vectorization using #pragma omp declare simd. This directive enables the compiler to generate vectorized functions. Compared to `FWaveSolver, FWaveVecSolver` precomputes some quantities to avoid redundant calculation of square root, which is time consuming according to the profiling results. These features make `FWaveVecSolver` the optimal solver in our project.

Loop Fusion: loop fusion refers to the technique that merges separated loops that loop through the same domain in order to improve cache locality and parallelism. The original code set two nested for loops to calculate the updates on the vertical and horizontal edges separately. Our code merges these two loop blocks into one. Two extra loops are required for the remaining column and row, because the loop indices for both edges are adapted to the interval [1,nx_+1] and [1,ny_+1].

OpenMP provides powerful possibilities in parallelizing for loops. Our plan is to distribute the solving of cells to multiple threads. According to the solver, the time step width calculation requires maximum wave speed in that domain block. A local variable is created on each thread to hold the maximum speed value on that thread, and the local values are reduced at the end to evaluate the global maximum wave speed. As previously mentioned, the solver is iteratively called in nested for-loops through all cells in a domain block. The outer loops are parallelized through `#pragma omp for`. The inner loops are vectorized with OpenMP vectorization directive `#pragma omp simd reduction(max:maxWaveSpeedLocal)`. A simple `#pragma omp for` is used for the threading of residual loops. The reduction clause reduces multiple values on the vector register to a single maximum value. At the end, a critical section is generated to allow each thread to write their local maximas and compare them for the global maxima.

Original:

```
void Blocks::WavePropagationBlock::computeNumericalFluxes() {
 RealType maxWaveSpeed = RealType(0.0);
// Compute updates on vertical edges
 for (int i = 1; i < nx_ + 2; i++) {
   for (int j = 1; j < ny_ + 1; ++j) {
     ...
     wavePropagationSolver_.computeNetUpdates(
       …
     );
     maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
   }
 }
// Compute updates on horizontal edges
 for (int i = 1; i < nx_ + 1; i++) {
   for (int j = 1; j < ny_ + 2; j++) {
     RealType maxEdgeSpeed = RealType(0.0);

     wavePropagationSolver_.computeNetUpdates(
     …
     );
     maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
   }
 }
…
```

After loop fusion

```
void Blocks::WavePropagationBlock::computeNumericalFluxes() {
 RealType maxWaveSpeed = RealType(0.0);

 for (int i = 1; i < nx_ + 1; i++) {
   for (int j = 1; j < ny_ + 1; ++j) {
// Updates for vertical edges
     RealType maxEdgeSpeed = RealType(0.0);

     wavePropagationSolver_.computeNetUpdates(
       …
     );
     maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);

// Updates for horizontal edges
     RealType maxEdgeSpeed = RealType(0.0);

     wavePropagationSolver_.computeNetUpdates(
     …
     );
     maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
   }

 }
// Loops for the last row (i=nx_+1) and last column (j=ny+1)
 for (int j = 1; j < ny_ + 1; j++) {...}
 for (int i = 1; j < nx_ + 2; i++) {...}
…
```

After OMP and SIMD

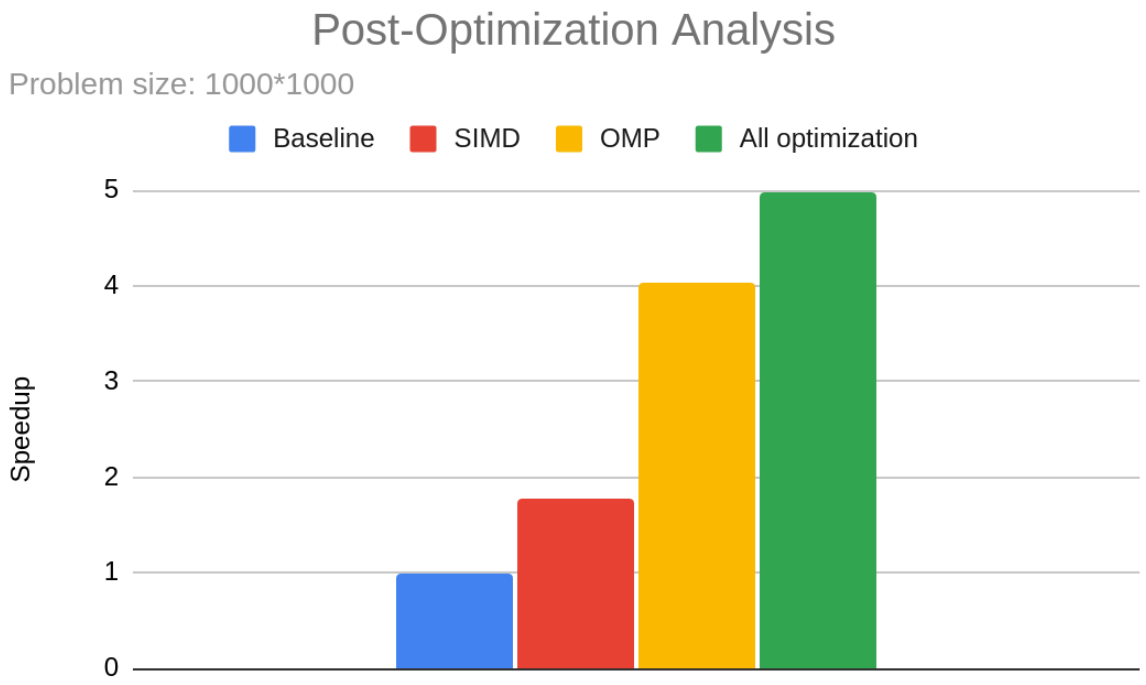
```
#pragma omp parallel
 {
   // Initialize two local variables for vertical and horizontal edges
   RealType maxWaveSpeedLocal_u = RealType(0.0);
   RealType maxWaveSpeedLocal_v = RealType(0.0);
```

```
#pragma omp for
   for (int i = 1; i < nx_ + 1; i++) {
#pragma omp simd reduction(max : maxWaveSpeedLocal_u) reduction(max :
maxWaveSpeedLocal_v)
     for (int j = 1; j < ny_ + 1; ++j) {
       // Compute the net-updates for the vertical edges
       …
       // Update the thread-local maximum wave speed
       maxWaveSpeedLocal_u = std::max(maxWaveSpeed, maxEdgeSpeed);

       // Compute the net-updates for the horizontal edges
       …
       // Update the thread-local maximum wave speed
       maxWaveSpeedLocal_v = std::max(maxWaveSpeed, maxEdgeSpeed);
     }
   }

// Dealing with the vertical edges for cells on the nx_+1 boundary
#pragma omp for
   for (int j = 1; j < ny_ + 1; ++j) {
     …
     maxWaveSpeedLocal_u = std::max(maxWaveSpeed, maxEdgeSpeed);
   }

// Dealing with the horizontal edges for cells on the ny_+1 boundary
#pragma omp for
   for (int i = 1; i < nx_ + 1; ++i) {
     …
     maxWaveSpeedLocal_v = std::max(maxWaveSpeed, maxEdgeSpeed);
   }

#pragma omp critical
   { maxWaveSpeed = std::max(std::max(maxWaveSpeedLocal_u, maxWaveSpeedLocal_v),
maxWaveSpeed); }
 }
…q
```

Profiling of the resulting code

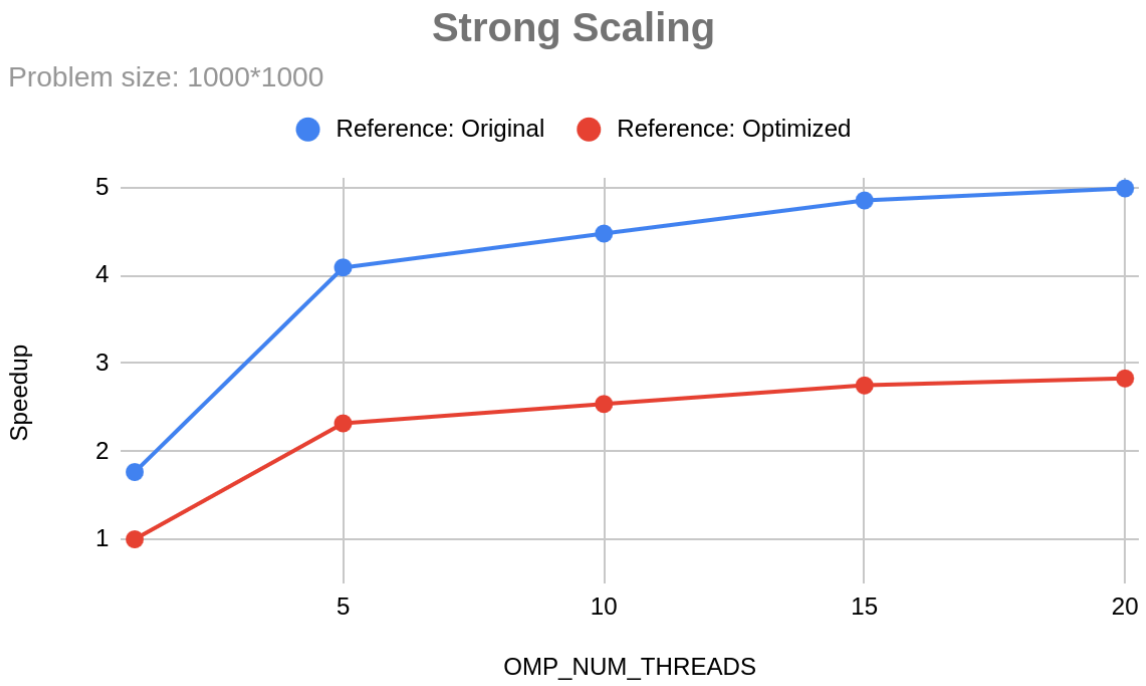
Comparison to baseline

The figure below compares the effect of optimizations with the naive serial implementation. The execution gets around 1.7 speedup after vectorization. OpenMP improves the performance alone by a factor of 4. All the optimizations done in the project contributes to 5-time-faster execution in total.



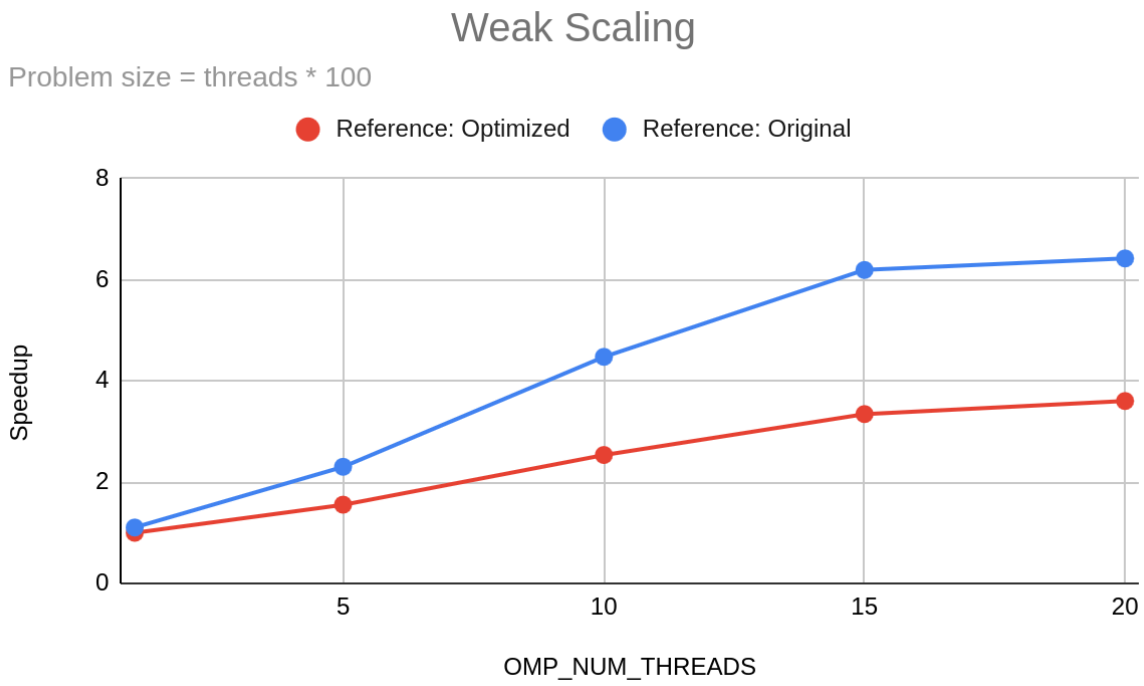
Scaling study

To understand the performance of our optimization, we perform scaling studies. The strong scaling is conducted at fixed problem size (1000 grid points on each x and y direction). The number of threads used is increased from single to 20 (max number of threads at local). The vectorization is enabled and MPI is switched off. Speedups are calculated with respect to raw code without using any acceleration technique, and optimized code run with a single thread.



The results of strong scaling indicate a significant improvement when switching from single thread to 5-thread implementation. However, the increase in speedup gets slower with further increasing number of threads. Only a minor speedup increase is observed when using all possible threads at local (20 threads). In our opinion, the hyper-threading in CPUs causes two threads competing for resources on a physical core and the communication overhead increases as parallelism. These factors depresses the improvement of speedup at high-number-threading executions.

For weak scaling, the problem size at x- and y-dimension is set to 100 times the thread number. Other settings in the strong scaling remain unchanged. As the result shows, the optimized code exhibits quasi-linear scaling before the 15-thread threshold. The increase slows down after it due to the reason mentioned above.



Testing

The program writes the resulting heights of water into an output binary file of type .nc. The Network Common Data Form is a file format commonly used for storing scientific data. For testing, we wrote a simple diff script using the library netCDF4 that allowed us to compare the resulting files. Specifically, we tested grid sizes 250x250 and 500x500. All optimization mentioned in this report passed this test.

References

[1] Likwid Supported Architectures.
https://rrze-hpc.github.io/likwid/Doxygen/index.html#Architectures
Accessed 21.01.2025