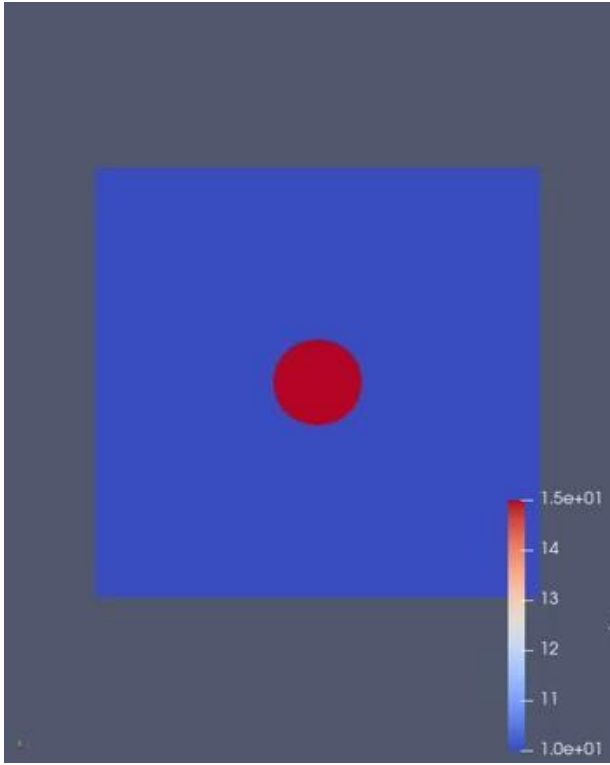
SWE Final Project

Group 1

Project Description

What SWE Does

- Simulate the flow of water
  - Domain decomposition
  - Numerical simulation
- Reproduce water height change in reality
- Application fields:
  - Hydraulic Engineering for canals and dams
  - Oceanography for tsunamis and storm modeling
  - Atmosphere modeling

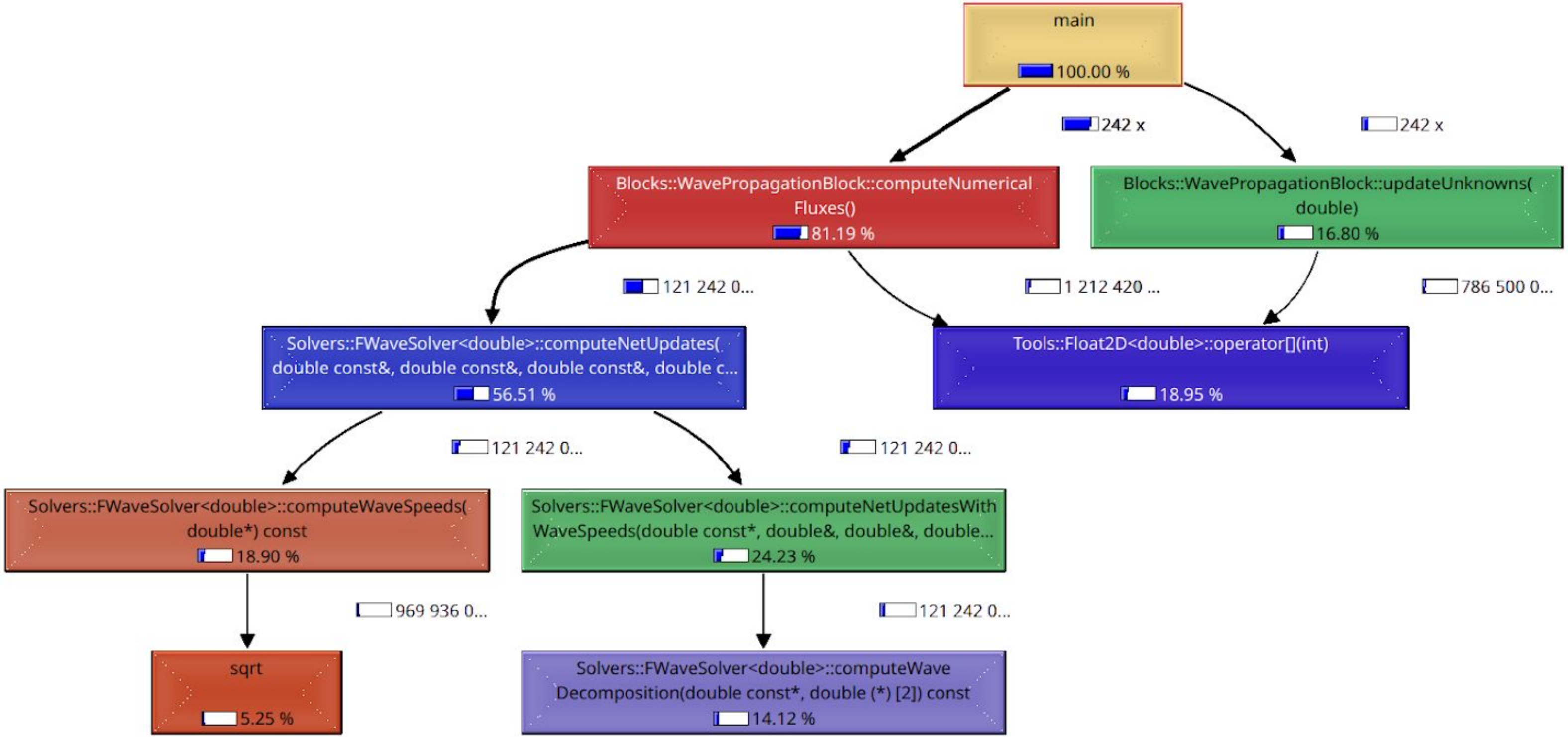


Idea of Optimization

1. **Vectorizing solvers to enable SIMD processing**
    a. Intrinsics
    b. OpenMP SIMD pragmas
2. **Multithreading with OpenMP**
    a. Parallelizing for-loops
    b. Tuning scheduling scheme to resolve load imbalance
3. **Restructuring the code for better efficiency**
    a. Precomputing variables to decrease computing overhead
    b. rearranging loop configuration for better cache locality

Profiling

Callgrind Profiling



VTune Profiling

Computer information:

```
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
```

Execution command: vtune -collect hotspots -result-dir SWE mpiex
-np 10 ./SWE-MPI-Runner -x 1000 -y 1000

VTune Profiling – Baseline

Elapsed Time: 18.374s

CPU Time:	159.609s
Total Thread Count:	43
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
PMPI_Allreduce	libmpi.so.40	43.067s	27.0%
Solvers::FWaveSolver<double>::computeWaveSpeeds	SWE-MPI-Runner	20.041s	12.6%
Solvers::FWaveSolver<double>::computeWaveDecomposition	SWE-MPI-Runner	18.012s	11.3%
Tools::Float2D<double>::operator[]	SWE-MPI-Runner	12.143s	7.6%
Blocks::WavePropagationBlock::updateUnknowns	SWE-MPI-Runner	10.734s	6.7%
[Others]	N/A*	55.612s	34.8%

*N/A is applied to non-summable metrics.

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Module
PMPI_Allreduce	43.067s	libmpi.so.40
Solvers::FWaveSolver<double>::computeWaveSpeeds	20.041s	SWE-MPI-Runne
Solvers::FWaveSolver<double>::computeWaveDecomposition	18.012s	SWE-MPI-Runne
Tools::Float2D<double>::operator[]	12.143s	SWE-MPI-Runne
Blocks::WavePropagationBlock::updateUnknowns	10.734s	SWE-MPI-Runne
__ieee754_sqrt	7.731s	libm.so.6
Blocks::WavePropagationBlock::computeNumericalFluxes	7.662s	SWE-MPI-Runne
MPI_Sendrecv	6.672s	libmpi.so.40
Solvers::FWaveSolver<double>::computeNetUpdatesWithWaveSpeed	6.491s	SWE-MPI-Runne
Solvers::FWaveSolver<double>::determineWetDryState	6.272s	SWE-MPI-Runne
Solvers::FWaveSolver<double>::computeNetUpdates	5.060s	SWE-MPI-Runne
Solvers::WavePropagationSolver<double>::storeParameters	3.191s	SWE-MPI-Runne
std::max<double>	2.650s	SWE-MPI-Runne
func@0xc3b0	2.042s	SWE-MPI-Runne
MPI_Init	1.924s	libmpi.so.40
func@0x2c520	1.480s	libnetcdf.so.19

Building setup:

ENABLE_OPENMP=OFF
ENABLE_VECTORIZATION=OFF
Solver: FWaveSolver

Resolving Load imbalances
From wetting/ drying

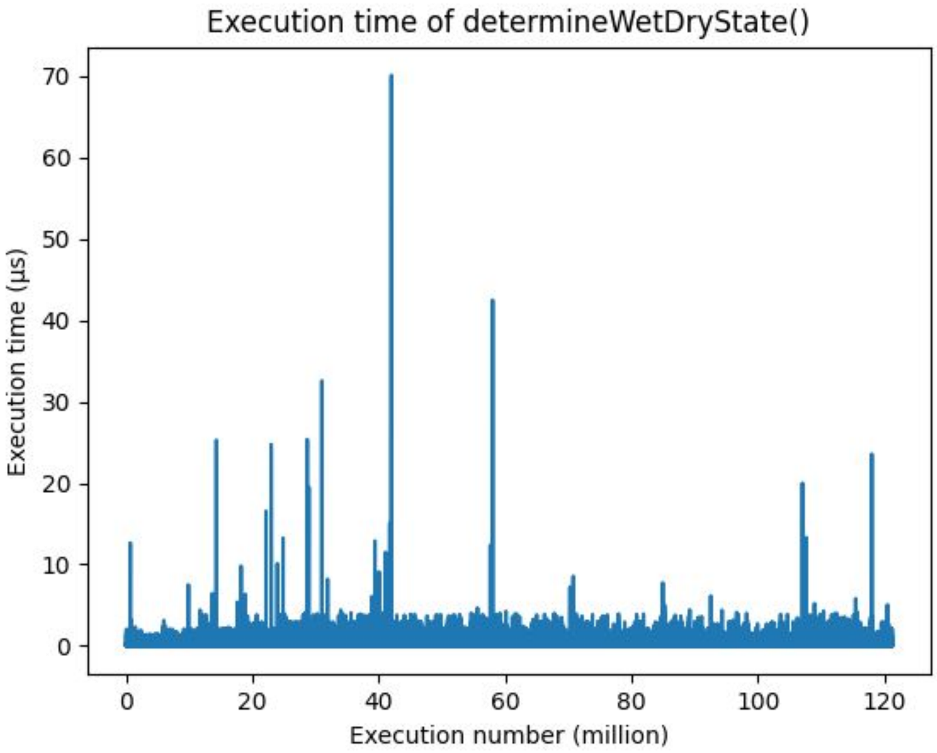
Timing the function

Started by timing the calls

Some spikes

Does not show load imbalance

Can lead to one

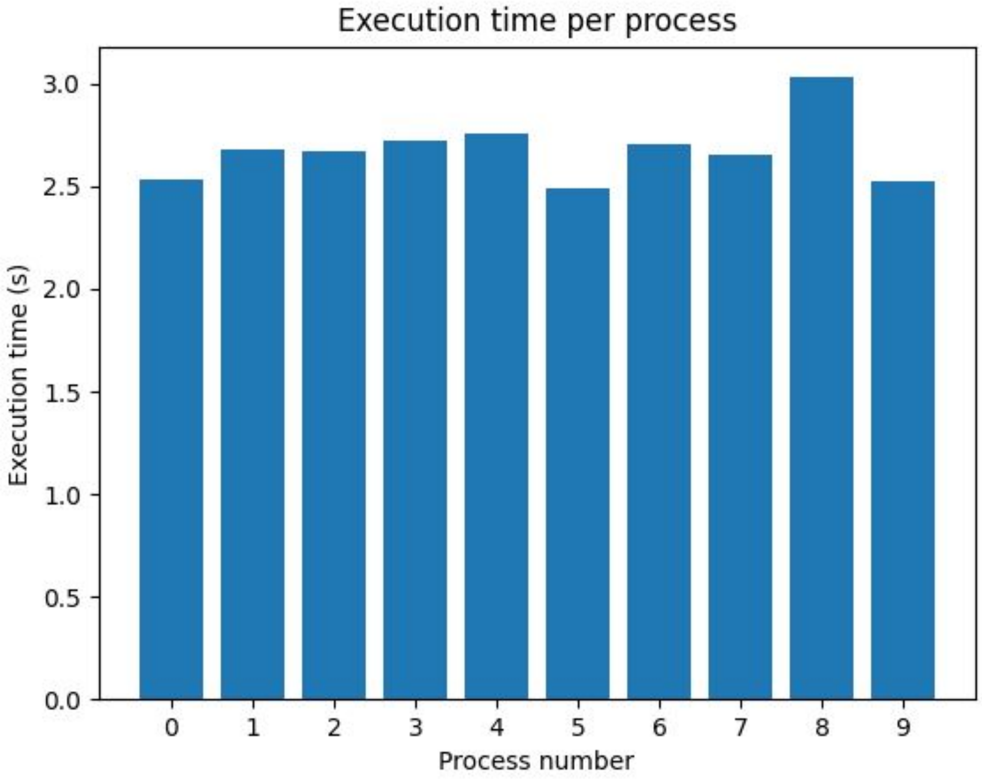


Profiling the code

Tried many profilers

Ended up manually timing

Attaching PID to find imbalance



Not that much of a load imbalance, larges difference 0.5s

Code

determineWetDryState() called once in computeNetUpdates()

computeNetUpdated called in nested for loops

Idea: add dynamic scheduling to these for loops

```
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);
  }
}
```

Using dynamic scheduling

Using schedule(dynamic) to distribute load

Critical section

```
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);
   }
 }
```

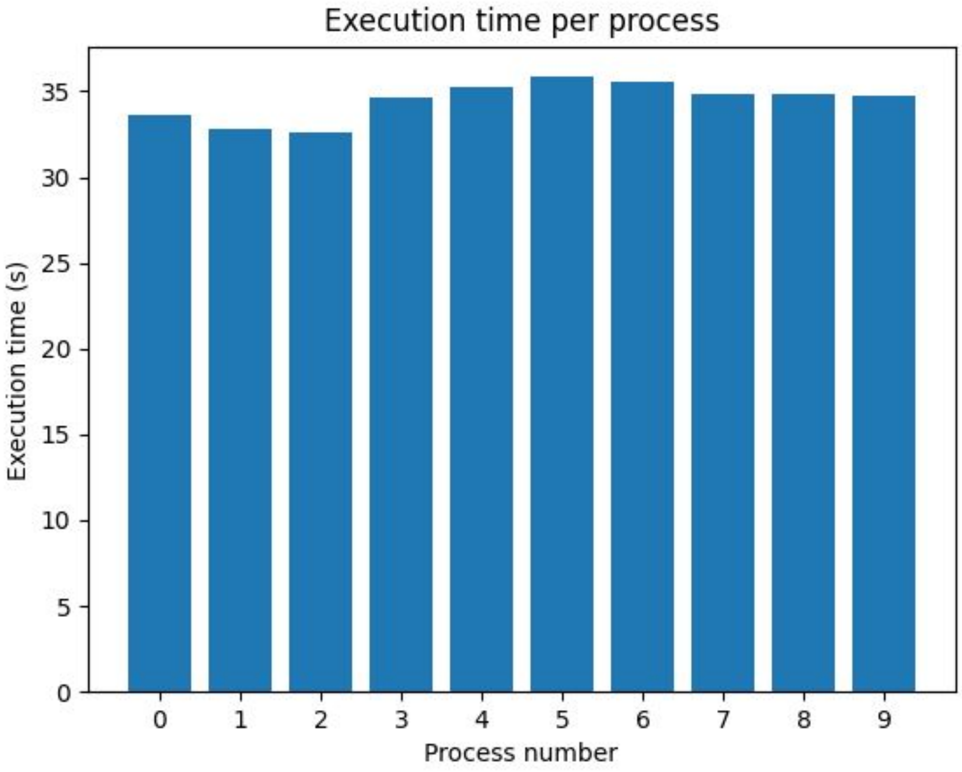
Profiling after optimization

Better balanced

Far slower,

probably critical section

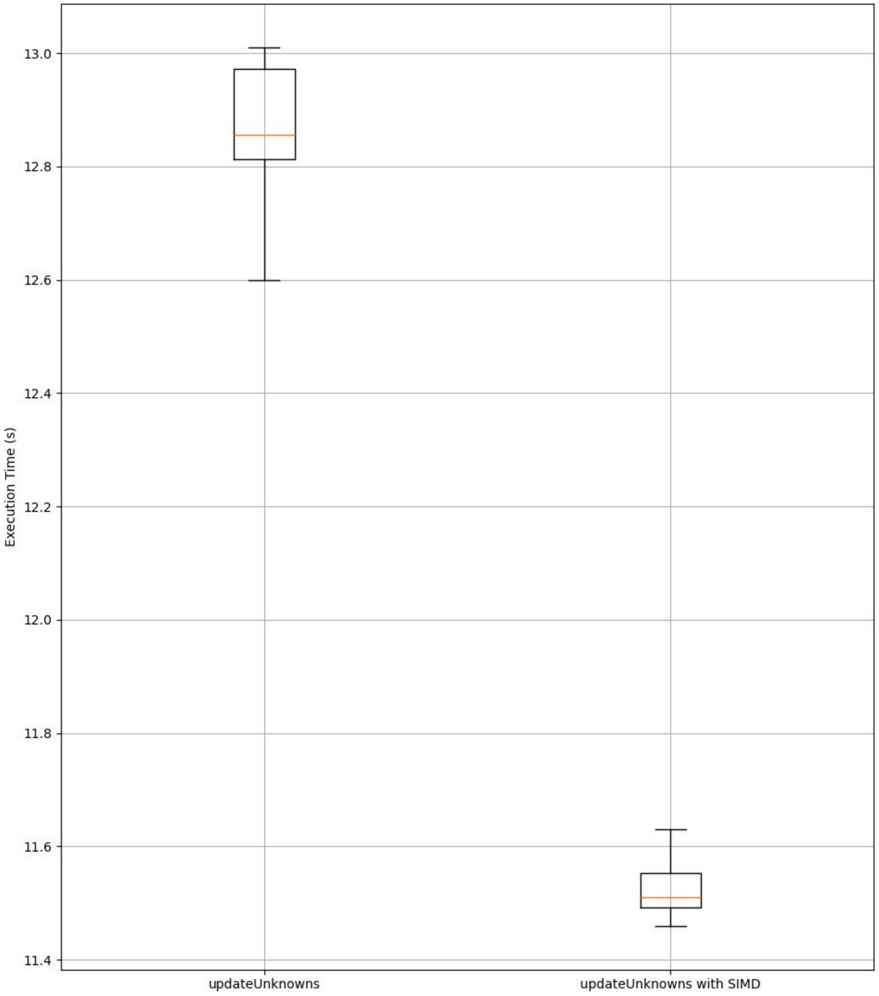
Tried reduction but got errors we were unable to fix



Single Core Optimizations

updateUnknowns

- responsible for updating the cell averages of the simulation grid based on the computed updates
- loops over all updated values -> possible to use SIMD operation
- we used __m256d represents a 256-bit SIMD register that can store and operate on four double-precision (64-bit) floating-point values simultaneously



load/store: _mm256_loadu_pd, _mm256_storeu_pd

arithmetic operations: _mm256_add_pd, _mm256_mul_pd, _mm256_sub_pd

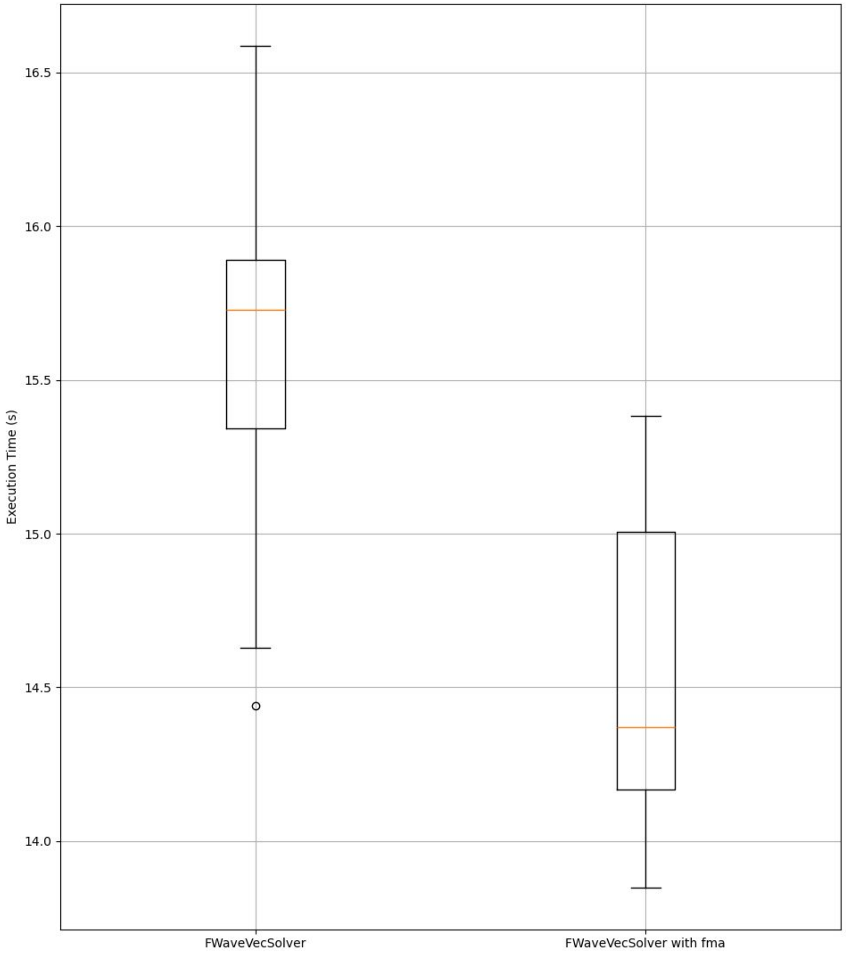
handling of conditions:

_mm256_cmp_pd: compare to generate a mask

_mm256_blendv_pd: apply mask to update values selectively:

computeNetUpdate

- former solver has lot of possibilities to optimization
- vectorized solver addresses these issues
  - **SIMD Directives:** uses #pragma omp declare simd
  - **Reduced Redundant Computations:**
    - calculated values are stored as member variables and accessed multiple times
    - reduces the number of operations and square root computations
  - **Minimal Use of Arrays:** store everything in variable
  - **Inline of Wetting/Drying**



- Square root caching
- Branching reduction
- Prefetching values
- Fused multiply-add operations
- Additional inlining

Float2D

- module is a convenience helper class that handles 2D float arrays
- **<double>operator[](int)** from report, takes around 19% of of all instructions
- module is already very lightweight -> limited options for optimizing
- improvements
  - precomputation of access patterns
  - store **data_ + (rows_ * i)**
  - precomputation in parallel
  - function call inlining
- initial 18.95% was lowered to 17.01% instructions

Loop Fusion, OMP and SIMD in Wave Propagation

Wave Propagation

Original code:

- Loops through all cells in a block
- Calls the solver inside the loop
- Independent caller for each cell
- Two nested for-loops for vertical and horizontal cell edges

Our idea:

- Loop fusion - merge two nested loops into one
- OMP - parallel for the outer for loop
- SIMD - switch to vectorized solver and vectorize inner loop

Loop Fusion in Wave Propagation

Original code:

```
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) {
     RealType maxEdgeSpeed = RealType(0.0);
     solver(...);
     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);
     solver(...);
     maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
   }
 }
…
```

Optimized:

```
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);
     solver(...);
     maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);

// Updates for horizontal edges
     RealType maxEdgeSpeed = RealType(0.0);
     solver(...);
     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++) {...}
…
```

OMP and SIMD in Wave Propagation

```
Blocks::WavePropagationBlock::computeNumericalFluxes() {
 RealType maxWaveSpeed = RealType(0.0);
#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
       solver(...);
       // Update the thread-local maximum wave speed
       maxWaveSpeedLocal_u = std::max(maxWaveSpeed, maxEdgeSpeed);

       // Compute the net-updates for the horizontal edges
       solver(...);
       // 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) {...}
// Dealing with the horizontal edges for cells on the ny_+1 boundary
#pragma omp for
   for (int i = 1; i < nx_ + 1; ++i) {...}
#pragma omp critical
   { maxWaveSpeed = std::max(std::max(maxWaveSpeedLocal_u, maxWaveSpeedLocal_v), maxWaveSpeed); }
 }
```

Starting the parallel region

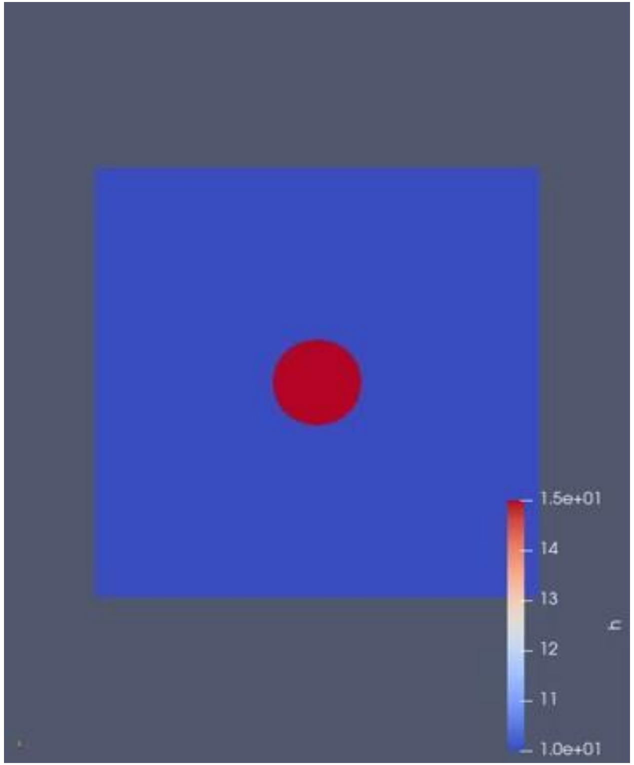
Parallelizing the outer loop

Vectorizing inner loop

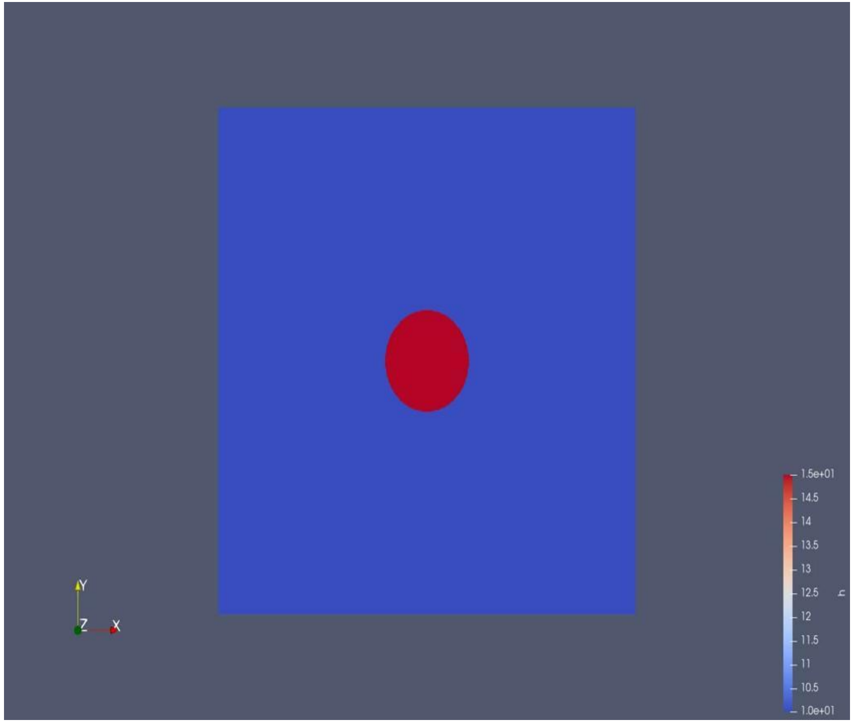
Critical section for global maximum wave spee

Combined results

Simulation Results (Dam-break scenario)

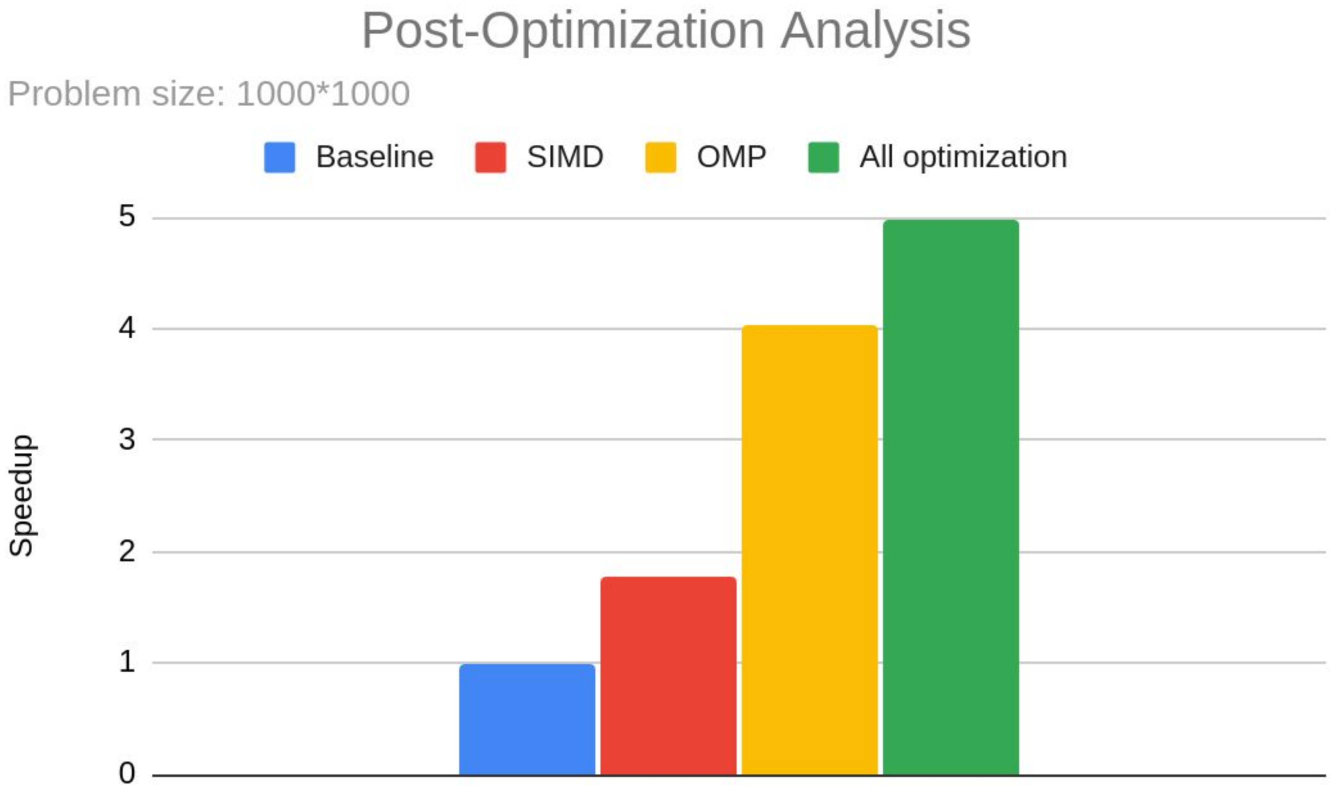


Baseline



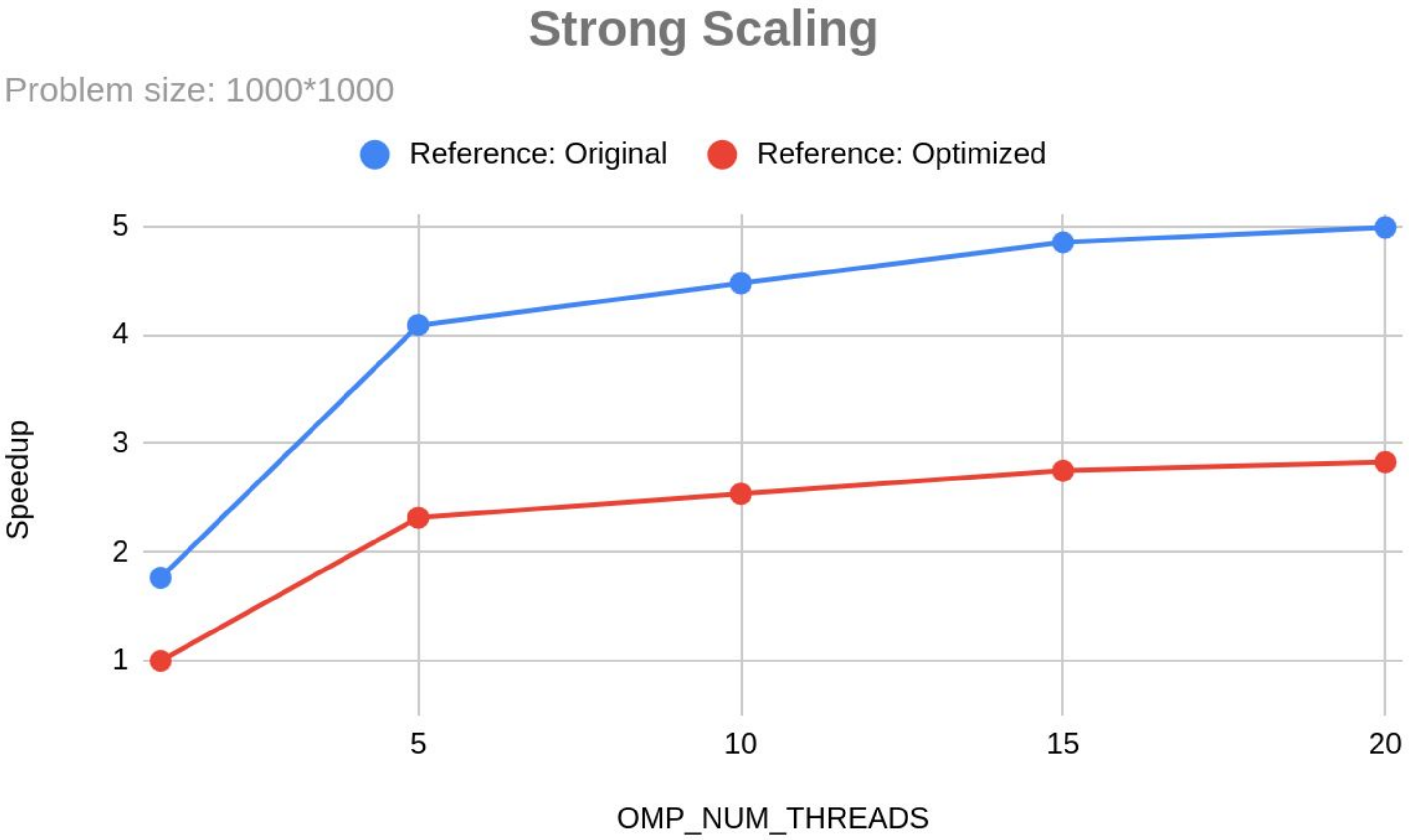
Post-optimization

Performance Comparison



Scaling Study

Strong Scaling



Weak Scaling