TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Colloquium

Lab: Concurrent and Distributed Systems
Himeno Benchmark and Mandelbrot Set

Fritz Louis Wilke
fritz.wilke@tu-dresden.de
4536116
M. Sc. Computer Science

Agenda

The **Mandelbrot Set**

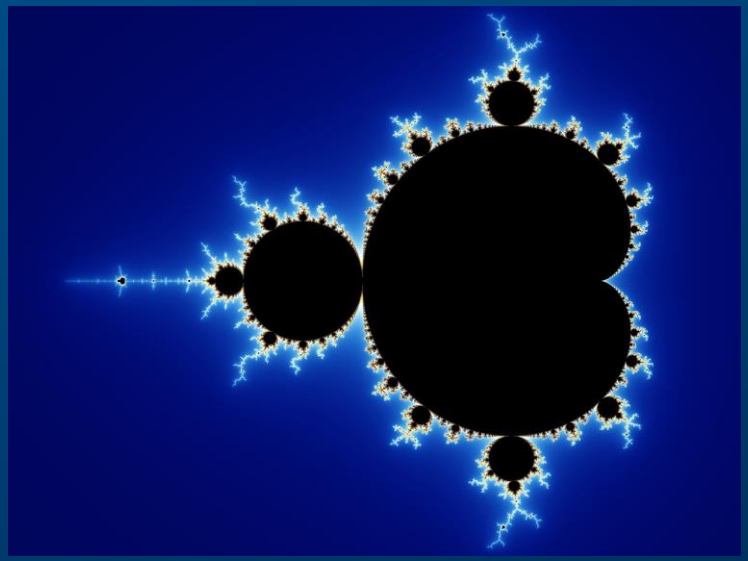
- **Problem**
- (Unoptimized) **Algorithm**
- **Optimizations**
- **Evaluation**

The **Himeno Benchmark**

- **Problem**
- (Unoptimized) **Algorithm**
- **Optimizations**
- **Evaluation**

Summary

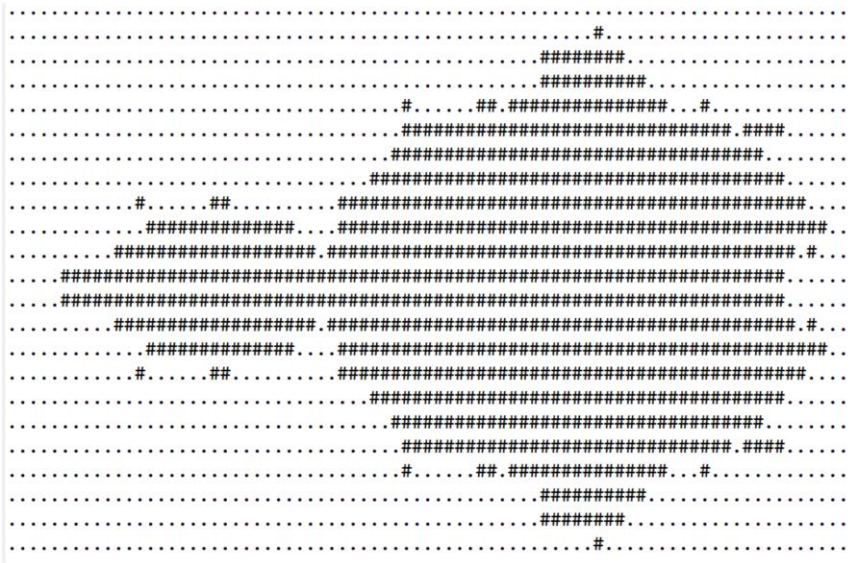
The **Mandelbrot Set**



https://commons.wikimedia.org/wiki/File:Mandel_zoom_00_mandelbrot_set.jpg

Problem
The Mandelbrot Set

- Set of complex numbers
- $z_0 := 0$
- $z_{n+1} = z_n^2 + \boldsymbol{c}$
- $M = \{z_n \in \mathbb{C} \mid |z_n| \leq 2 \; \forall n \in \mathbb{N}\}$
- c depends on pixel coordinate in image



Unoptimized Algorithm
The Mandelbrot Set

```
input: uint rows, uint cols, uint nn
output: char[]
begin
    let img = char[rows][cols]
    for r in 0..rows:
        for c in 0..cols:
            let z = complex(0, 0)
            let n = 0

            while |z| < 2.0 and n < nn:
                z = z*z + complex( c*2 / cols - 1.5, r*2 / rows - 1.0 )
                n = n + 1
            img[r][c] = (n == nn) ? ´#´ : ´.´

    for r in 0..rows:
        for c in 0..cols: print( img[r][c] )
        print( ´\n´ )
end
```

Optimizations
The Mandelbrot Set

General:

- **Simplifying** calculations
- Working directly with **floats** instead of `std::complex<float>`
- Storing and **reusing** interim results
- Working on **continuous** memory (as image)

Parallelization:

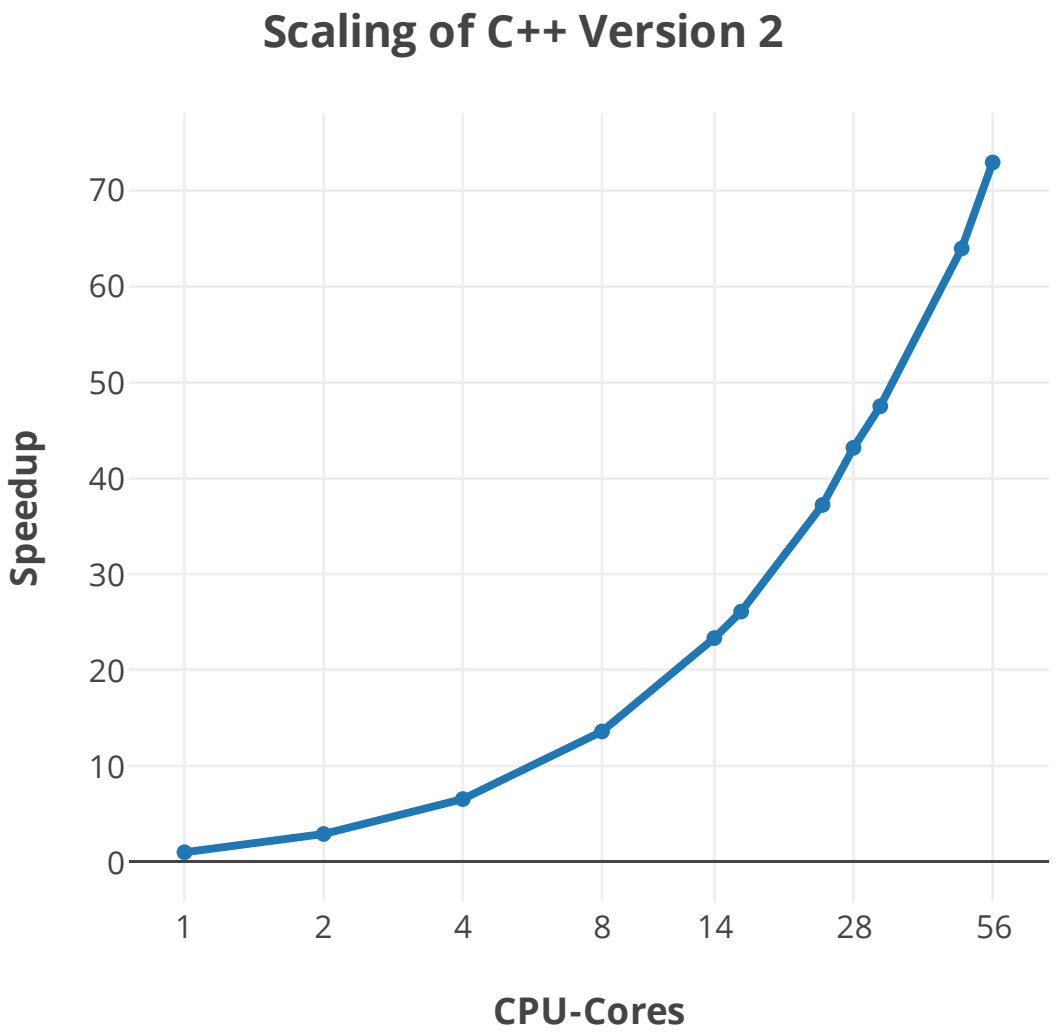
- **Parallelize** the pixel calculation (inner for loop)
- Provide and collect work **dynamically** via **queues**
- Aligning data structures to fit **cache lines**
- **Each** worker thread has one **input-** and **output queue**

Parallel Workflow
The Mandelbrot Set

Evaluation
The Mandelbrot Set

- Unoptimized code as **reference**
- **C++** and **C#**
- Scaled very well
- No decline after **14** or **28** cores due to local queues
- Parallel code amount over 100% (not realistic)

Lang.	V	Optimization	Time	Abs.	Rel.
C++	0	Unmodified	207050	1x	1x
	1	I/O Threads	2182	86x	83x
	2	+ Simplified	**263**	**716x**	73x
C#	0	Unoptimized	63332	1x	1x
	2	Both	452	146x	**1798x**

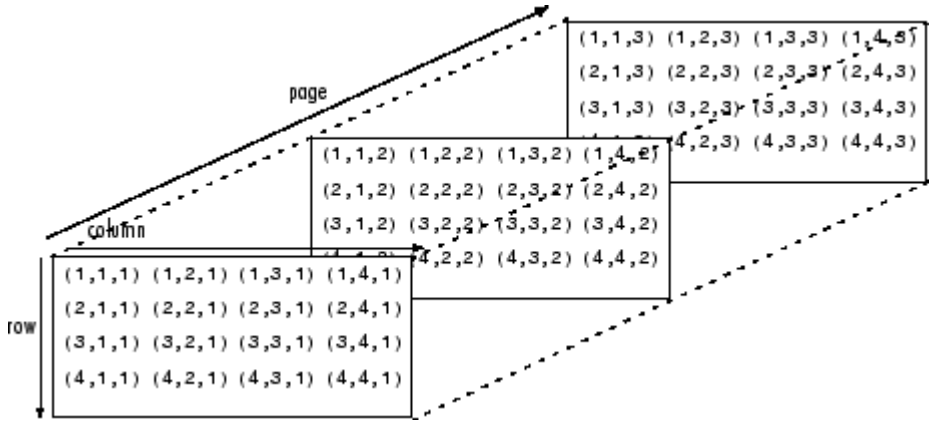


The **Himeno Benchmark**

http://www.isum2017.udg.mx/es/contenido/dr-ryutaro-himeno

Problem
The Himeno Benchmark

- A **benchmarking** program
- Extremely **memory intensive**
- Uses the **Jacobi method** for calculations
- Works on several 4D-Matrices, where first 3 dimensions are same size



http://www.isum2017.udg.mx/es/contenido/dr-ryutaro-himeno

Unoptimized Algorithm
The Himeno Benchmark

```
input: uint rows, uint cols, uint deps, uint nn
output: float
begin
    initialize_matrices()
    let gosa

    for n in 0..nn:
        gosa = 0.0
        for r in 1..rows-1: // iterate over every voxel
            for c in 1..cols-1:
                for d in 1..deps-1:
                    let ss = jacobi_iteration(r, c, d)
                    gosa += ss*ss
                    wrk2[0][r][c][d] = p[0][r][c][d] + 0.8*ss
        wrk2.copy_to(p) // copy content of p to wrk2

    print( gosa ) // print result (with a precision of 6)
end
```

Optimizations
The Himeno Benchmark

General:

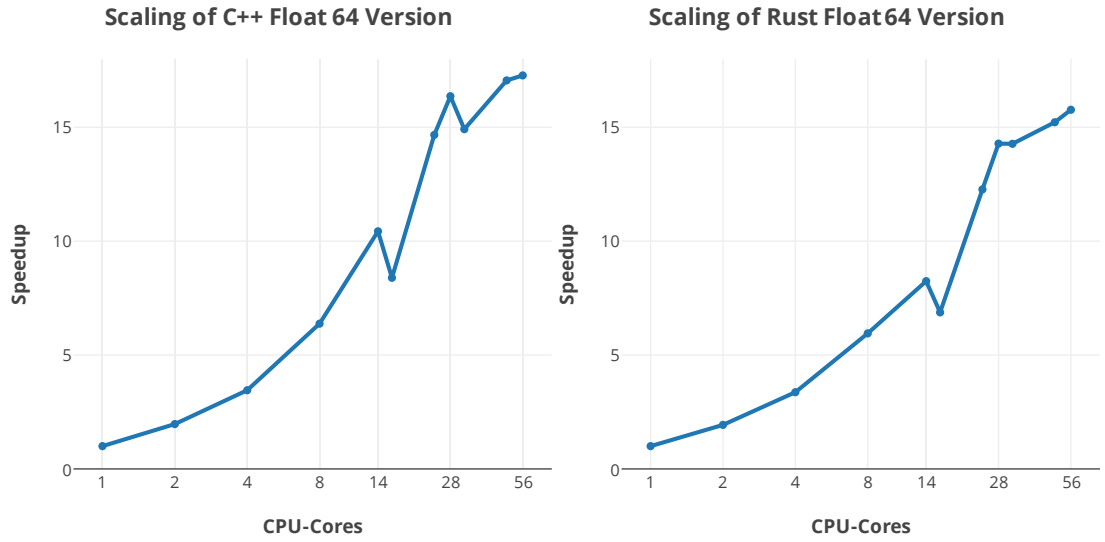
- Drastically **reducing memory usage**:
  - **Substitute** 5 of 7 matrices by constants
  - From 33 to 9 **matrix reads/writes (by 73%)**
  - **Pointer swapping** instead of matrix copying (reduces $(rows - 2) * (cols - 2) * (deps - 2)$ to 1) per it.
  - **Reduce** matrix sizes by 2 each dimension
- Only calculate the gosa in **last iteration**

Parallelization:

- Parallelize on the **rows loop** (outermost, parallelizable loop)
- Threads get work from a **shared atomic variable** containing the next row
- Calculate gosa **sub-sums** in parallel (only F64)

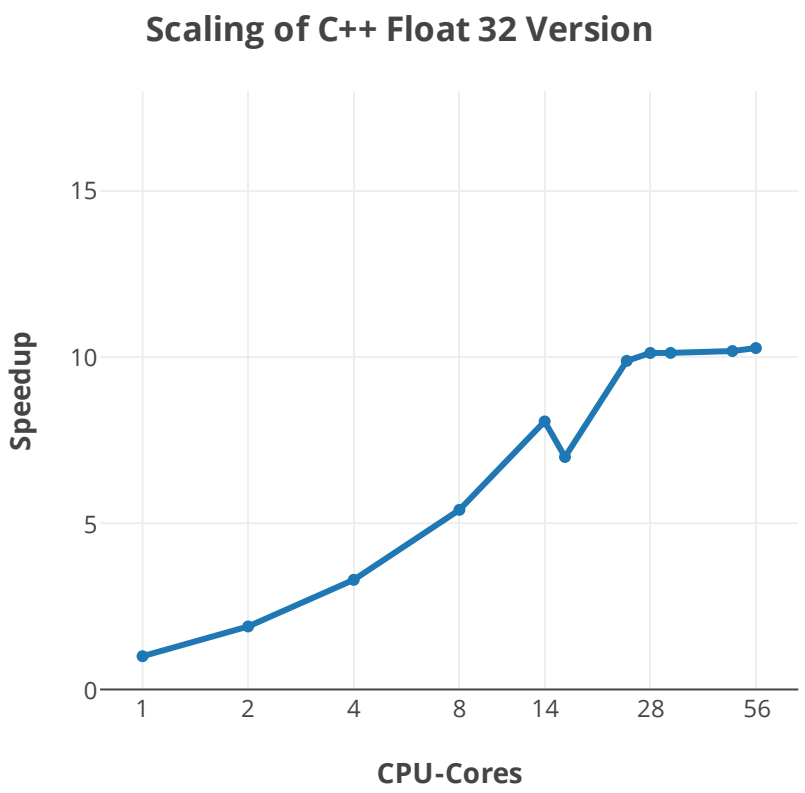
Evaluation
The Himeno Benchmark

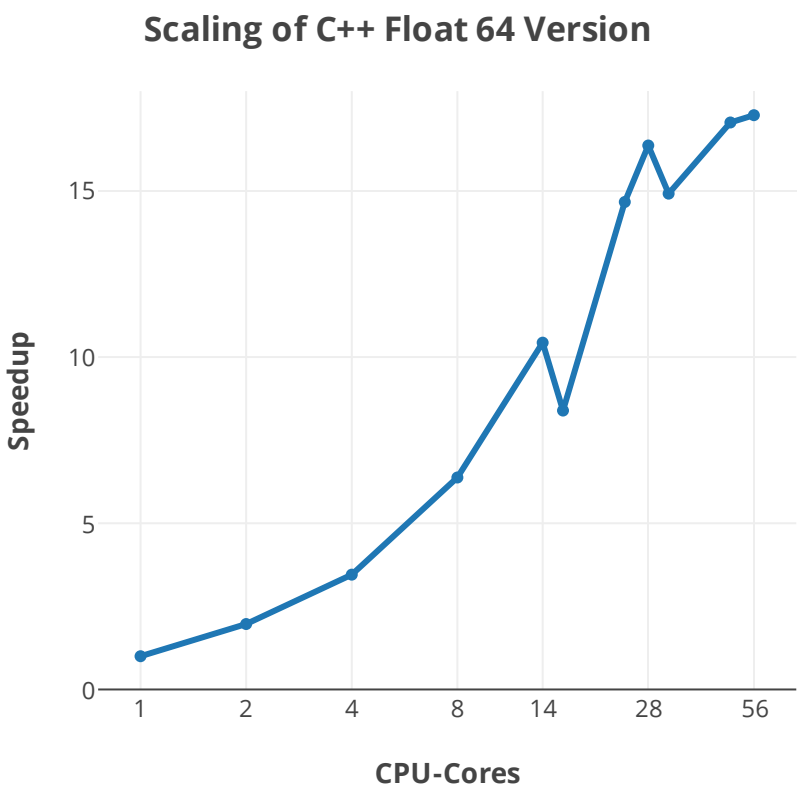
- Unoptimized code as **reference**
- **C++** and **Rust**
- Scaled ok (w.r.t. the high memory usage)
- Strong decline at **16** and **32** cores
- Parallel code up to **97.37%**

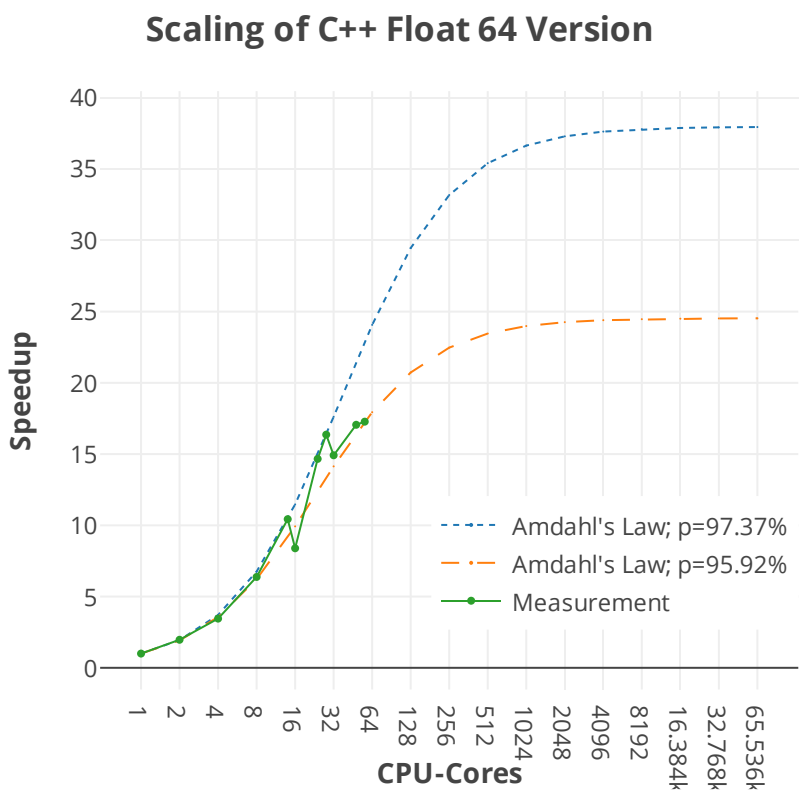


Lang.	FXX	Optimization	Time	Abs.	Rel.	p @ 28	p @ 56
C++	32	Unmodified	171978	1x	1x	0%	0%
		Parallel	34346	5x	10x	93.46%	91.91%
	64	Unmodified	207050	1x	1x	0%	0%
		Parallel	19294	10x	**17x**	**97.37%**	**95.92%**
Rust	64	Unoptimized	289943	1x	1x	0%	0%
		Parallel	**7108**	**43x**	16x	96.44%	95.36%

Evaluation
The Himeno Benchmark



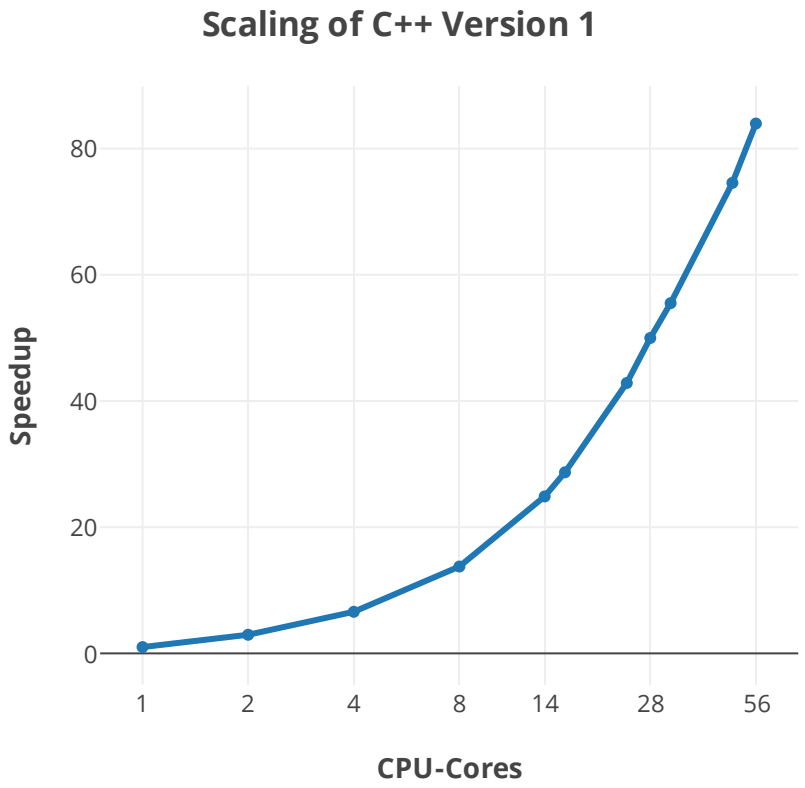


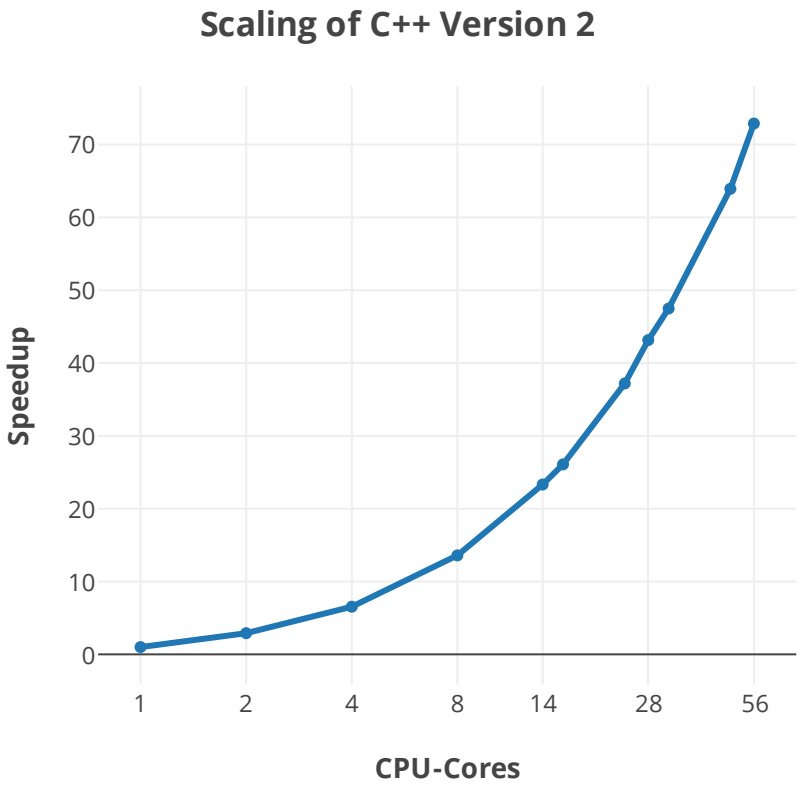


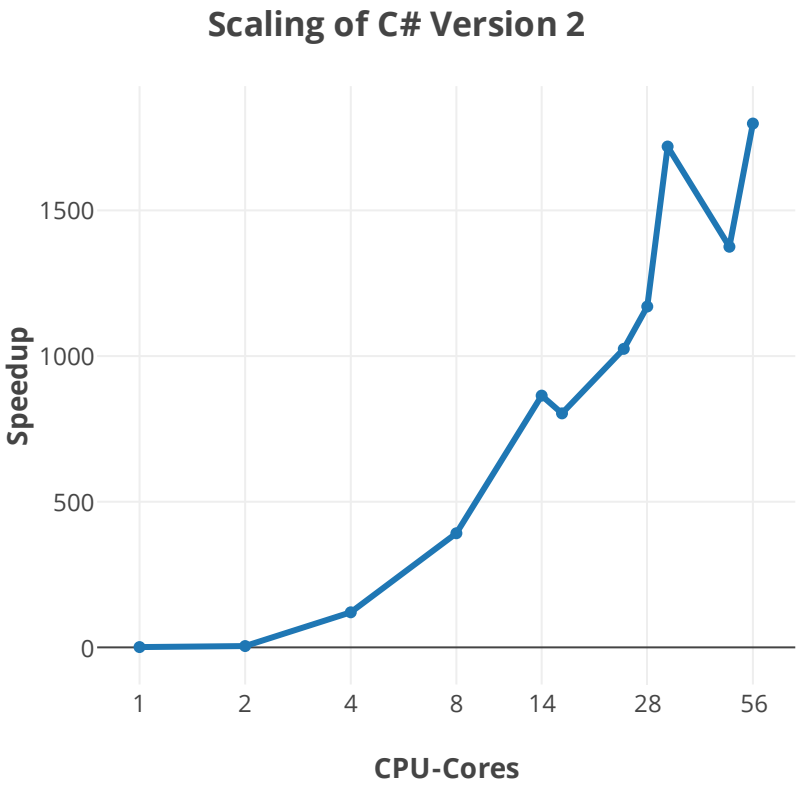
Summary

- Solved workload distribution with **queues** and **atomic variables**
- **Optimized** and **simplified** the algorithms
- Speedup of **1798** (716 abs.) for the **Mandelbrot Set**, optimal goal was **51**
- Speedup **of 17** (43 abs.) for the **Himeno Benchmark**, optimal goal was **13**
- Learned about **memory throughput** and **cache invalidation** as limiting factor

Scaling of all Versions
The Mandelbrot Set







Scaling with Amdahl's Law of all Versions
The Himeno Benchmark