TECHNISCHE
UNIVERSITÄT
DRESDEN

Computer Science Institute of Systems Architecture, Chair of Systems Engineering

Himeno benchmark

Lab: Concurrent and Distributed Systems

Fritz Louis Wilke

E-Mail: fritz.wilke@tu-dresden.de
Course: INF-MA-PR (graded)
Discipline: M. Sc. Computer Science
Matriculation number: 4536116

May 27, 2021

1 Introduction

This report is part of the *Concurrent and Distributed Systems Lab* which aims to test and improve practical knowledge about scalable, parallel programming.

1.1 The Problem

The *Himeno Benchmark* is - as the name implies - a benchmarking program. It was developed by Dr. Ryutaro Himeno at the RIKEN Institute in 1996, but the source code is still available today. Its goal is to provide a comparable, overall performance benchmark between different machines and architectures. This includes the memory throughput, caching behaviour and CPU performance (FLOPS).

The program requires four inputs. The *rows*, *cols* and *deps* define three of four sizes of the 4D-matrices to work with and *nn* is used to specify the number of *Jacobi* iterations performed. The *Jacobi method* is an iterative algorithm to determine the solutions of a system of linear equations. Usually, the *Jacobi method* is executed until the solution converges, but in this program the number of iterations is defined by *nn*.

The output of the program is calculated in the last iteration of the *Jacobi method* and is called the *Gosa number*. In a nutshell, the *Gosa number* is the sum of the squared average of the direct neighbours of each voxel $v_{ijk} \in M$, where M is the current three dimensional working matrix.

The task of this lab is to find a way to parallelize the described program in a way that it scales well with an increasing number of available CPUs. To achieve this, strategies of concurrent and distributed systems programming are used to plan, implement and evaluate the parallelized program. The provided output must match the one given by the sequential solution, though the algorithm itself may be modified. Note that it is not the goal of the parallelized program to provide a comparable benchmark, as it is intended by the original program.

1.2 Unoptimized Code Base

Ryutaro Himeno's source code is written in C. This code forms the basis for the development of the program. Its basic workflow is summarized as pseudo code in Listing 1. As already mentioned in the introduction, the benchmark works with 4D matrices which are initialized with defined values. Note that the three dimensional size of the matrices is identical, they only differ in the fourth (outermost) dimension.

For each iteration `n`, the `gosa` gets set to zero. After that, the program iterates over every *voxel* to execute its calculations on it. A *voxel* is like a pixel with three dimensions, as it defines a position in a three dimensional matrix. In the next step, the gosa gets increased with the squared result `ss` that was previously calculated. Finally, the `wrk2` matrix is set at the current voxel, with the content of the matrix `p` plus 0.8th of `ss`.

```
input: uint rows, uint cols, uint deps, uint nn
output: float
begin
    // creates and initializes the global 4D matrices: a, b, c, p, bnd, wrk1, wrk2
    initialize_matrices()

    let gosa = 0.0
    for n in 0..nn:
        gosa = 0.0

        // iterate over every voxel (at layer 0)
        for r in 1..rows-1:
            for c in 1..cols-1:
                for d in 1..deps-1:
                    let ss = jacobi_iteration(r, c, d) // can be seen as a blackbox for now
                    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
```

Listing 1: Simplified Pseudo Code of the unoptimized Algorithm

The last step in the iteration `n` is to copy the content of the matrix `wrk2` to the matrix `p`. These contents are then used in the next iteration. Finally, when `nn` iterations have been executed, the *Gosa number* is printed. In this task, the precision used for printing it is six. Seeing the applied loops, this algorithm has the complexity $\mathcal{O}(nn * 2 * (r - 2) * (c - 2) * (d - 2))$, where the variables are the *iteration amount* and *rows*, *cols* and *deps* of the matrices in that order. The computational expense used for the initialization of the matrices is not included in that estimate. The $2*$ comes from the copy operation, since it is implemented by iterating element-wise over the source matrix.

2 Optimization Strategies

To improve the scalability of the shown algorithm, several strategies have been developed and tested. The hard part of developing scalable software (w.r.t. the amount of CPU cores) is to distribute the given workload equally between the CPU cores while still having a correct result in the end. The two most successful strategies are presented in the following sections.

2.1 Strategy I: Equal Work Distribution

The goal of this strategy is to find a way to split the workload equally, **before** the actual computation starts. As the amount of calculations only depends on the given parameters and does not vary at runtime, the workload can be seen as fixed. Accordingly, it is a reasonable approach to simply split the workload into equal parts, one for each *worker thread*. It is expected that each calculation takes the same amount of CPU cycles, or else this concept will not succeed. If one *worker thread* finishes earlier than another, then it will go idle and it wastes parallel computation power.

Finding such a way to split workload is not trivial, especially having only the source code by hand. The simplified pseudo code in listing 1 makes this already a bit easier. Usually, it is the best solution to parallelize the *outermost* loop. Unfortunately, the iterations from zero to nn (line 8) depend on each other and can thus not be parallelized directly. It can be seen that the matrix $wrk2$ depends on p (line 17), whereas p depends on $wrk2$ (line 19) in the same iteration n later on. Thus, p ultimately depends on itself of the previous iteration $n - 1$. Having said this, the next loop in the hierarchy would be a feasible target, which is the *rows* loop in line 12. There is no dependence of rows **inside** an iteration n. If the amount of concurrent threads of the runtime environment is t, then each *worker thread* receives $\omega = (rows - 2)/t$ rows to calculate the new `wrk2[0][r][c][d]` values of. The threads get created and are joined for each iteration n, thus creating $nn * t$ threads throughout the runtime of the program. Since the creation of threads takes only a few microseconds it should not hurt the performance too much, respecting that $rows * cols * deps$ is much greater than nn.

This strategy should improve the performance due to the following reasons:

- No synchronization overhead, only some overhead due to thread creation and joining
- No threads needed to distribute workload which causes inter-thread or even inter-socket communication
- Parallelizing the most work intensive calculations, namely the lines 15-17 of listing 1

One should keep in mind that there will still be a lot cache invalidation since the *Himeno Benchmark* is extremely memory intensive. This can be seen in the source code to which line 15 in listing 1 relates. These widely distributed matrix accesses will almost always result in a cache miss, since only voxel neighbours on the *deps* axis reside in continuous memory. It is not possible to prevent this behaviour.

2.2 Strategy II: Atomic Variable

This strategy uses a shared atomic variable v_{shared} to distribute the workload. The shared variable contains the next work portion for a *worker thread* and can be accessed and modified in a **thread safe** manner. The *worker thread* then calculates its assigned work and gets more once it is done. An atomic variable can be seen as a shared variable with a mutex that is used to prevent simultaneous variable access from multiple threads. This strategy does not necessarily expect that each *worker thread* calculates as fast as the others. It can thus cope with unforeseen scheduling behaviour that results in threads being delayed. The balancing act is to find a loop that is fine enough (inner loops) to make equal distribution possible but also coarse enough (outer loops) to prevent frequent, concurrent access of the shared variable. Since memory access is comparably slow to cache access, it should be avoided to access a shared variable too often.

Having said this, the *rows* or *cols* loop (line 12 and 13 in listing 1) are justified candidates for the shared variable. If the *rows* loop is used, the content of the shared variable is trivial: simply put the next, not yet calculated row in it and increase by one when reading it. When working more fine grained and using the *cols* loop, things get more tricky since the variable must also contain the information of the current *row*. A simple solution would be to not only store the current col of the current row ($v_{shared} \in [1, rows - 1]$) which would need two shared variables, but also **contain** the current row **implicitly** ($v_{shared} \in [0, (rows - 2) * (cols - 2)]$). The current row and col would be derived like this:

$$\begin{aligned} row &= 1 + \lfloor v_{shared} \div (cols - 2) \rfloor \\ col &= 1 + v_{shared} \bmod (cols - 2) \end{aligned}$$

This strategy should improve the performance due to the following reasons:

- Parallelizing the most work intensive calculations, namely the lines 15-17 of listing 1
- The ability to cope with unforeseen delays of threads (eg. due to scheduling)
- Not having too frequent accesses on the shared variable v_{shared}

It can be seen that this strategy provides advantages similar to strategy I. However, this strategy can cope with different CPU times of the *worker threads*, whereas strategy I cannot. If the CPU time gets distributed evenly throughout the threads though, using and accessing a shared variable would be an unnecessary slow down.

3 Implementation

This section presents the most important implementation details of the optimized program. At first, general improvements on the code are elaborated, which also affect the single thread performance. The latter subsection focuses on the improvements that have been made specifically for multithreading.

3.1 General Improvements

Having a closer look into Himeno's source code, one may observe that there are a lot memory accesses which are also widely scattered w.r.t. locality. This is a killer for multithreaded performance since it makes the usage of CPU caches almost forfeited. As long as *cache thrashing* occurs, it is hard to parallelize the program efficiently because the speed of memory accesses is the limiting factor. An important objective would therefore be to **reduce memory usage** as far as possible.

Inspecting the matrix accesses, it can be seen that the matrices a, b, c, bnd and $wrk1$ are never modified. Theses matrices can be replaced by constants which reduces the memory usage drastically. The constants for the matrices at are:

$$\begin{aligned} a[0..2] &:= 1; a[3] := 1/6 \\ b &:= 0 \\ c &:= 1 \\ bnd &:= 1 \\ wrk1 &:= 0 \end{aligned}$$

Multiplications with these constants can be optimized further on by **removing** products of zero multiplications (matrix *b* and *c*) and simplifying multiplications with one (matrix *a*[0..2], *c* and *bnd*). The only matrices that are still in use after these modifications are matrix *a* at layer 3 ($a[3] := 1/6$) as constant and the mutable matrices *p* and *wrk*2.

Another, more obvious optimization target is the copy operation from line 19 of listing 1. This can be replaced with a simple variable swap. If *wrk*2 and *p* are both **pointers** to matrices, then the following code reduces the complexity of the copy operation from $\mathcal{O}((r-2)*(c-2)*(d-2))$ (exponential) to $\mathcal{O}(1)$ (constant): `std::swap(p, wrk2)`.

Furthermore, the matrix sizes can be reduced. An inspection of the code reveals that the outermost *rows*, *cols* and *deps* are read but **never written**. Therefore the matrices can be reduced by two in every dimension. To prevent reads that are out of bound, a new function is implemented that calculates these read-only values on-the-fly when accessing out of bounds. Else, the actual matrix values are returned.

The last big optimization is the calculation of the *Gosa number*. Since it is set to zero at the beginning of each iteration *n* (see line 9 in listing 1), it is sufficient to only calculate it in the last iteration. This saves the execution of line 16 in listing 1 for every iteration, excluding the last. Note that for the sake of readability of the source code, it was ported from C to C++, so that an own class can be created for the matrices and other quality of life features are available.

3.2 Parallel Implementation

Despite the general improvements mentioned in the previous section, not much code had to be changed. We assume that strategy II provides better scalability than the first one since one can always expect variable calculation times of the same work when using several concurrent threads, especially when they're not bound to specific CPUs. The calculation is parallelized on the *rows* loop, namely line 12 in listing 1. This should reduce the simultaneous access on the shared variable enough to prevent unwanted side effects.

The workflow of the parallel program is summarized as follows:

1. Read the parameters from `stdin`
2. Create and initialize the two matrices `p` and `wrk`
3. For each iteration $n \in [1, nn]$ do:
    a) Create $t - 1$ *worker threads*, matching the hardware concurrency
    b) Let the threads (including the main thread) calculate until there is no work left. The work is received by *reading and incrementing* the shared variable.
    c) If it is the last iteration: modify the *Gosa* variable (in a thread safe way)
    d) Join the *worker threads*
    e) Swap the pointers of the matrices `p` and `wrk`
4. Print the *Gosa number* that was calculated in the last iteration

For the given task it is possible to implement and evaluate two different versions: one using 32 bit floating point variables (single precision or *F32*) and one using 64 bit floating point variables (double precision or *F64*). Due to the lack of precision using *F32* variables, it is not possible to

Lang.	FXX	Measurement	Time@56 [ms]	Speedup@56 abs	Speedup@56 rel	Git Hash	Evaluation Log
C++	32	Unmodified	171978	1x	1x	`c4e204b7`	30.04.2021 20:17:30
		Parallel	34346	5x	10x	`43c3df53`	09.05.2021 16:17:19
	64	Unmodified	207050	1x	1x	`c4e204b7`	30.04.2021 20:17:30
		Parallel	19294	10x	**17x**	`43c3df53`	09.05.2021 16:17:19
Rust	64	Unoptimized	289943	1x	1x	`3773b864`	12.05.2021 16:17:25
		Parallel	**7108**	**43x**	16x	`3fadcfc3`	26.05.2021 20:17:11

Table 1: Evaluation Results of the Implementations

use partial sums of the *Gosa number* for each thread. Therefore, the *Gosa* variable must be shared throughout the *worker threads* when using *F32* variables to prevent an invalid result. As a consequence, the *Gosa* variable is protected by a mutex in the *F32* version.

The *F64* versions (implemented in C++ and Rust) are not tied to this restriction. Each individual *worker thread* uses its own *F64* variable to sum up the *Gosa number* using the voxels it worked with. Once every thread terminates, the partial sums of the threads merge to a correct result. Therefore, no locking is needed to calculate the *Gosa number* in the *F64* versions.

Rust is chosen as an alternative programming language to test the performance of another modern, hardwarenear language and also to learn a new programming language. Rust provides similar programming paradigms as C++, allowing it to compare the measurement results to the other implementation. The unoptimized Rust program, which is used for the absolute speedup calculation, is developed as close as possible to the original C code from Himeno. Note that only the *F64* version is implemented in Rust.

4 Evaluation

The previously described parallel implementations are evaluated in this chapter. All important measurement results can be seen in table 1, categorized by their programming language and the used floating point precision. The shown **absolute speedup** is the speedup of the evaluated program using 56 concurrent threads with respect to the runtime of the unoptimized program (using 56 concurrent threads). The **relative speedup** refers to the runtime of the same program and evaluation run using only a single thread. This speedup is also shown in detail in figure 1. The evaluated source code and commit is available using the Git hash which includes a hyperlink to the source file.

It can be seen in table 1 that the best overall runtime results from the multithreaded Rust implementation, which took around seven seconds. It is followed by the *F64* C++ implementation with 19 seconds and *F32* C++ program that ran 34 seconds. The **absolute speedups** are 43, 10 and five respectively. Remind that these speedups refer to the unoptimized programs. The best relative speedup is achieved by the *F64* C++ implementation. This program reached a

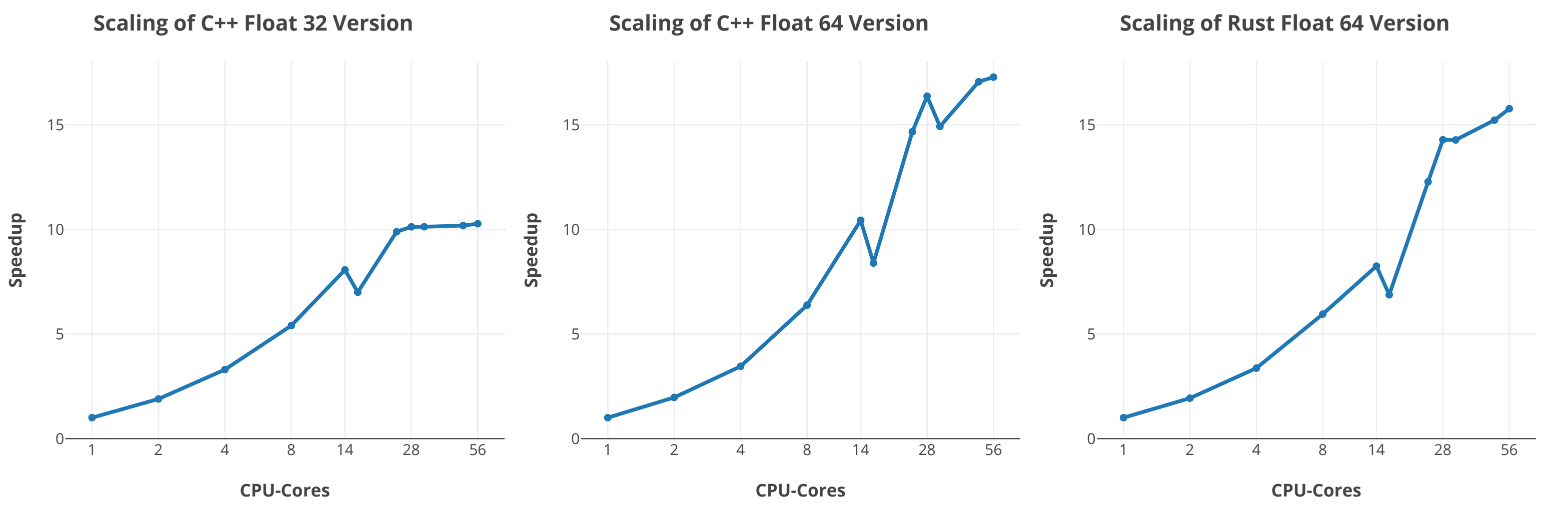

Figure 1: Scaling of the C++ Versions Float 32 (left), Float 64 (middle) and Rust Float 64 (right)

relative speedup of 17 using 56 concurrent threads.

The same code accomplished a relative speedup of ten using *F32* variables. The lower speedup can be concluded from the shared *Gosa* variable mentioned in the previous chapter. It seems that this is the limiting factor which prevents the program from scaling higher than ten. The graph in figure 1 (left) shows this limit very well. There is no significant performance increase from using 28 concurrent threads or more. This graph also shows a strong performance decrease when using 16 threads. This is because after 15 or more threads, the second CPU socket is used, forcing the hardware to communicate via the slow CPU BUS.

This performance collapse can also be discovered in the other two graphs in figure 1. However, there is no sharp scaling limit on the *F64* implementations. The C++ version even resembles the curve of *Amdahl's law*. Using Amdahl's law, the parallel code amount for the C++ *F64* implementation is 97.37% at 28 threads and 95.92% using 56 threads. The parallel code amount for the *F32* is 93.46% and 91.91% for 28 and 56 threads respectively. The Rust version results in 96.44% and 95.36% parallel code. These results seem realistic, but due to the unstable scaling curve, a precise upper scaling limit can not be predicted using Amdahl's law. The Appendix contains graphs showing the curve of Amdahl's law with these parallel code parts.

5 Conclusion

The desired optimal speedup of 13 is surpassed by both, the *F64* C++ and Rust implementation. Due to the memory intensive nature of the *Himeno Benchmark*, the scaling behaviours of the programs are neither smooth nor spectacularly high, but they show a satisfying efficiency in comparison to the unoptimized programs. It is shown that it suffices to implement basic multithreading strategies in combination with general code optimizations to reach this speedup.

6 Appendix

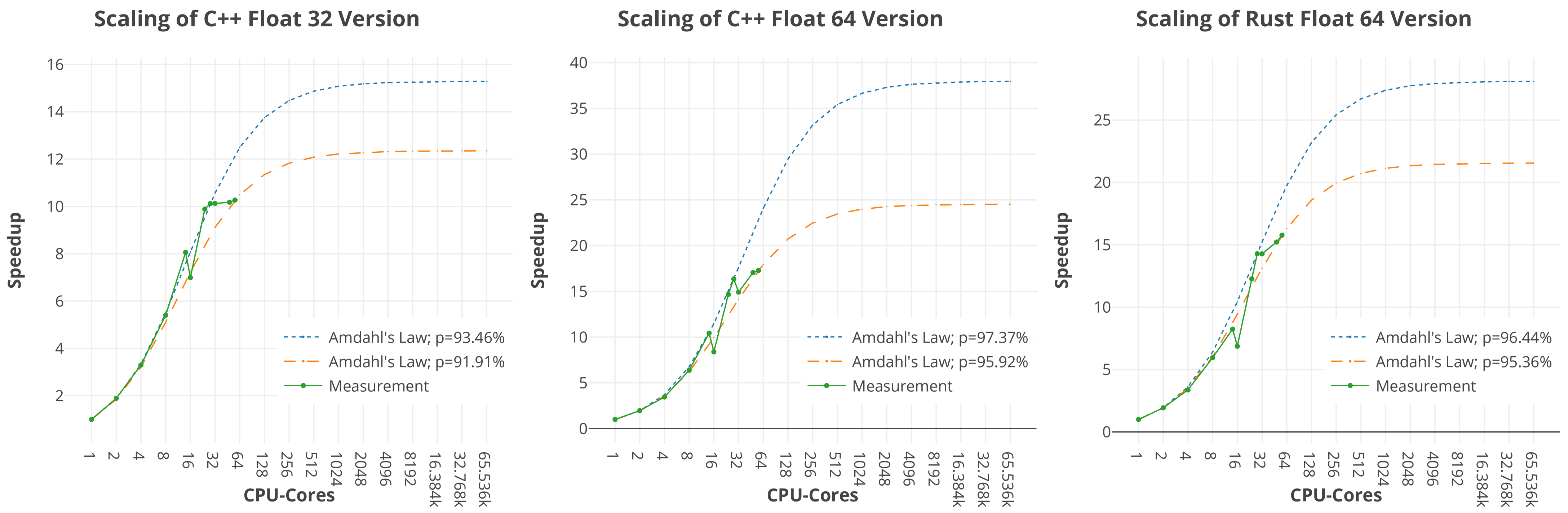

Figure 2: Scaling of the C++ Versions Float 32 (left), Float 64 (middle) and Rust Float 64 (right) enhanced with Amdahl’s law