INSTITUTE OF MICROELECTRONICS

NSSC - Exercise 3

Group 4

Member:
Christian Gollmann, 01435044
Peter Holzner, 01426733
Alexander Leitner, 01525882

Submission: December 15, 2019

Contents

1 Conjugate Gradients Implementation

For symmetric positive definite matrices A, the A-norm of the error decreases monotonically with increasing iteration number.[1] One can observe this behaviour in our own implementation in Fig.2. In exact arithmetic, the residual should hit it's minimum after approximately n iterations, where n stands for the dimension of the solution vector x which is roughly 5500 in our case. This is as well shown in our plot in Fig.1. After 5500, the residual experiences no significant changes any more. After this point, the minimum is essentially reached. The oscillations that can be observed afterwards are results of the numerical discretization and finite resolution inherent to computer precision. In general, it can be assumed that the minimum point is not exactly represented in the floating point representation of the computer.

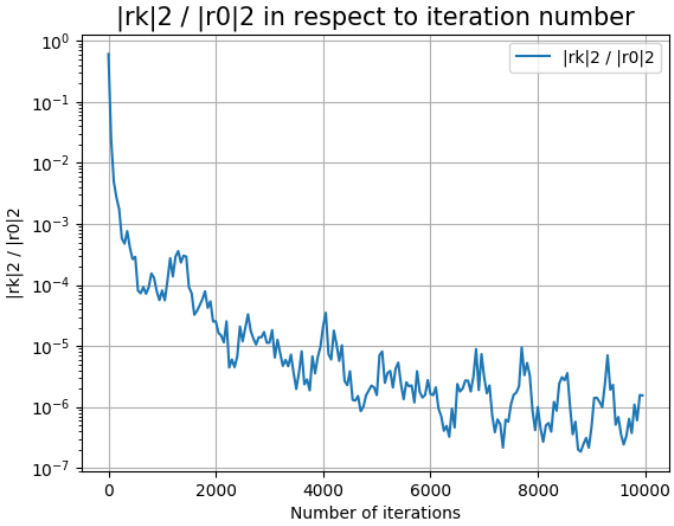

Figure 1: Normed residual of our implementation as a function of number of iterations

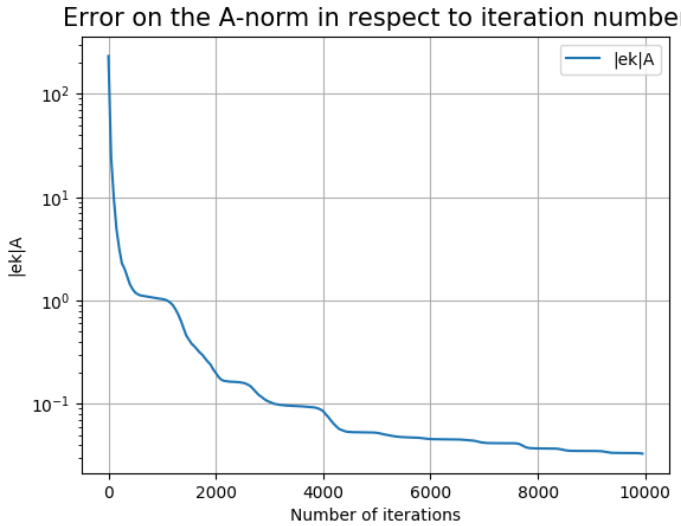

Figure 2: A-norm of the error as a function of number of iterations

Note that for readability reasons, we only plotted every 50^{th} value. Please take into account, that the existing data files will be overwritten once the program is executed and will therefore not deliver the same plots as below when called by the python script. However, a backup containing the data for a run with 10000 iterations is provided in the subfolder *./Abgabe_1/data_10000it/*.

2 Conjugate Gradients in EIGEN

In Task 2, we were quite surprised that the preconditioned methods worked so well since they both don't work with the exact matrix A like we do in our own implementation. The diagonally preconditioned method only takes the diagonal entries into account and the incomplete Cholesky method is "only" an approximation to the Cholesky factorization itself. Nevertheless, both methods gave similar or better results compared to our own implementation. What is quite remarkable as well is the difference in execution time. Our implementation took 4375 seconds to execute 10000 iterations. The diagonally preconditioned Eigen method delivered a similar (slightly better) result in 13 seconds which makes it faster by a factor of 430.

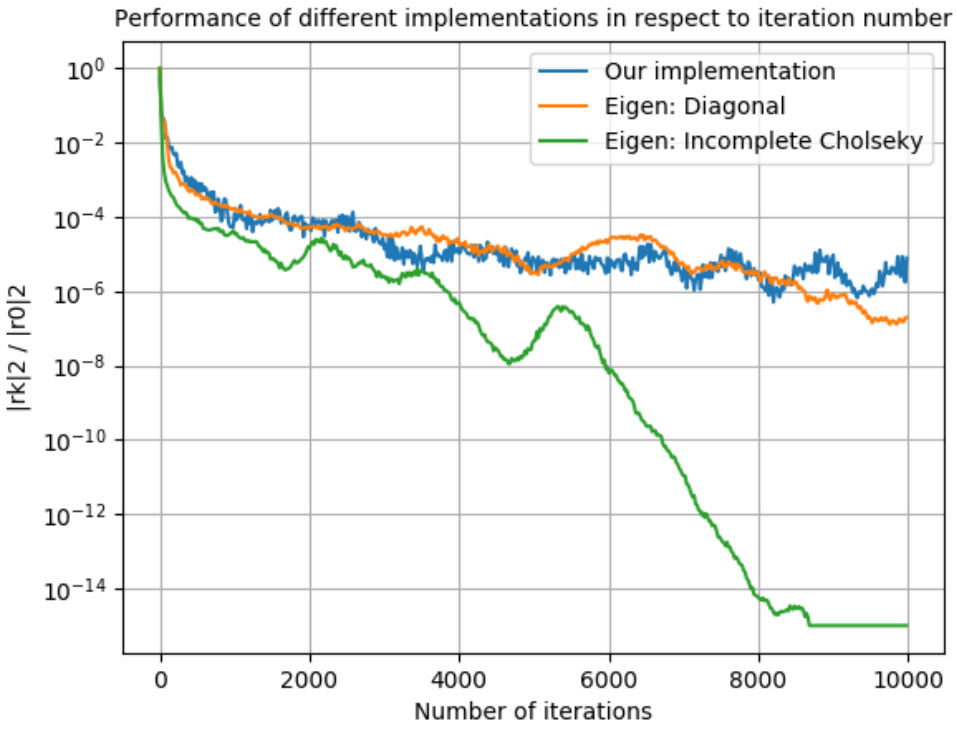

Figure 3: Comparison of three different versions of the Conjugate Gradients method

Note that for readability reasons, we only plotted every 20^{th} value. The program itself is also designed to only give back the value of every 20^{th} iteration. It can be called similar to our program in Task 1. Please consider, that the existing data files will be overwritten once the program is executed and will therefore not deliver the same plots as below when called by the python script. However, backups containing the data files for runs with 10000 iterations is provided in the subfolder *./Abgabe_2/backup_10000it/*.

3 Monte Carlo Integration

The main task in this exercise is to implement a numerical integration for a one-dimensional function $f(x) \to y$ using a Monte Carlo technique.

$$I = \int_a^b f(x)\,dx \;\approx\; \frac{1}{N}\sum_{i=1}^{N} f(\epsilon_i) \tag{1}$$

Where ϵ_i $i = 1, 2...N$ consist of random numbers in the range (a, b). An easier way to implement the random numbers is to transform the boundaries of the integral from (a, b) to $(0, 1)$.

3.1 Transformation

The first integral is defined by:

$$I_1 = \int_a^b \sin(x)\,dx \tag{2}$$

with a simple substitution by $u = \frac{x-a}{b-a}$ and $x = u(b-a) + a$ $dx = du(b-a)$we could rewrite (2) into a different boundary.

$$I_1 = (b-a)\int_0^1 \sin(x(b-a)+a)\;dx \approx \frac{b-a}{N}\sum_{i=1}^{N}\sin(\epsilon_i(b-a)+a) \tag{3}$$

The second integral is defined by:

$$I_2 = \int_a^b \sin^2\left(\frac{1}{x}\right)dx \tag{4}$$

and now we could rewrite (4) as described above.

$$I_2 = (b-a)\int_0^1 \sin^2\left(\frac{1}{x(b-a)+a}\right)\;dx \approx \frac{b-a}{N}\sum_{i=1}^{N}\sin^2\left(\frac{1}{\epsilon_i(b-a)+a}\right) \tag{5}$$

The third integral is defined by:

$$I_3 = \int_a^b x^3\,dx \tag{6}$$

and now we could rewrite (4) as described above.

$$I_3 = (b-a)\int_0^1 \Big(x(b-a)+a\Big)^3\;dx \approx \frac{b-a}{N}\sum_{i=1}^{N}\Big(\epsilon_i(b-a)+a\Big)^3 \tag{7}$$

3.2 Random number generator

The sequence to evaluate our random numbers ϵ_i is implemented by creating a MASTER RNG with a fixed seed. This MASTER RNG creates the seeds for the RNGs for the other threads when the program is parallelized. Those RNGs create $N = sampl$ random numbers.

3.3 Runtime analysis

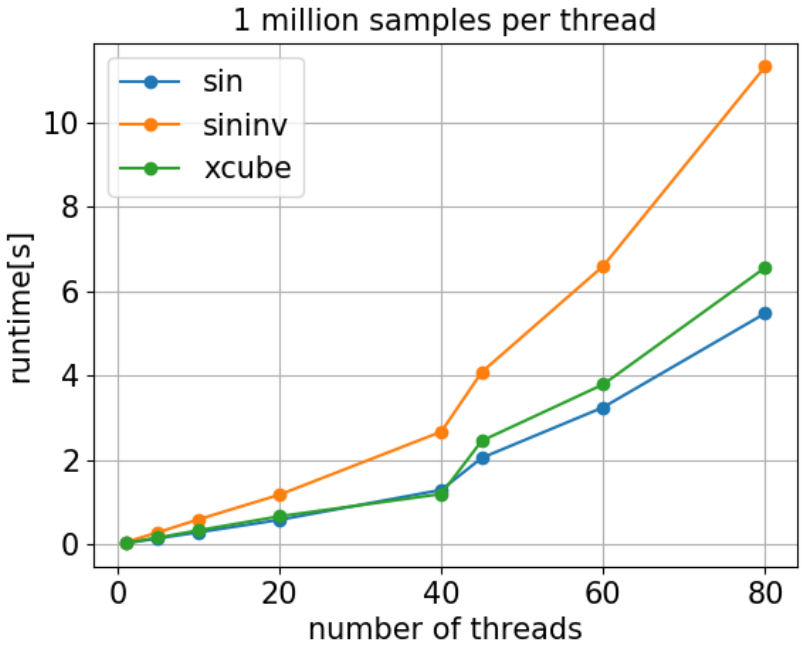

Figure 4: Different runtimes for 1 million samples per thread

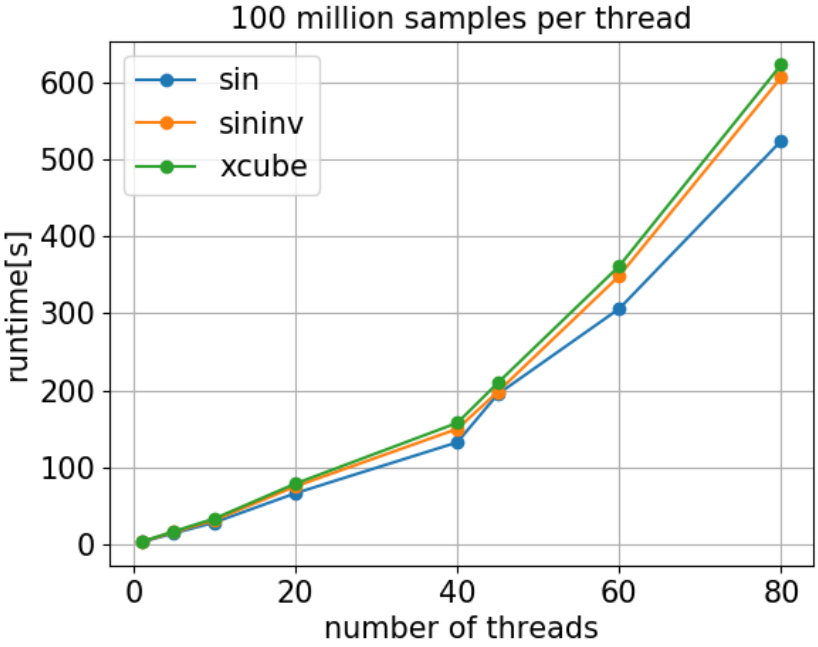

Figure 5: Different runtimes for 100 million samples per thread

The increased runtime per thread is due to the overhead of spawning the threads and the reduction at the end. The sightly increased slope in the runtime when using more than 40 threads is due to

the architecture of the cluster. Every node consists of $2 \times 20 = 40$ physical cores with potential hyper-threading. From thread 1 to 20 we might work on one CPU and then from thread 21 to 40 we probably start using the second CPU as well. After thread 40 we have to make use of hyper-threading. This is the reason why the slope of the runtime increases slightly. To picture this behaviour better we also simulate it for 45 and 60 threads. This behaviour means we only reached a poor weak scaling.

4 Parallel Jacobi Solver

4.1 About our implementation

Goal: Parallelize your Jacobi solver developed in Exercise 2 using OpenMP. In particular, you should identify the for loops eligible for OpenMP parallelization and discuss your choice and adopt the implementation of Exercise 2 so that not a fixed number of iterations is performed, but instead the program stops the iterations once the runtime exceeds 10 seconds make your program callable by:

```
./stencilomp resolution numthreads e.g.
./stencilomp 64 4
```

where *resolution* defines the grid spacing (same as in Exercise 2) and *numthreads* referring to the number of OpenMP threads to be used and print the number of performed iterations, the total runtime of the iteration loop, and the average runtime per iteration to the terminal.

The loop that is eligible for parallelization is the innermost loop which calculates the next iteration of the grid (vector), here called *xkp*1. In the following, we present the relevant snippet of our code which can be found in full in the file *stenciljacobi.cpp*.

```
// Begin code snippet
// chunk size is set in main and passed to the function
chunk = int(double(vec_size)/threads);
...
int jacobiMethod(...){ // Implementation of the jacobi method as a function
...
    for (size_t iteration = 0; iteration < maxIter; iteration++)
    {
        #pragma omp parallel for schedule(static, chunk)
        shared(b,xk,xkp1)
        firstprivate(innerLen,aij,daii,vec_size)
        for (size_t i = 0; i < vec_size; i++) // i is index of vector
        {
            double temp = 0.0;
            if (i>innerLen-1)
            {
                temp = temp - xk[i-innerLen];
            }
            if (i>0 && i%innerLen!=0)
            {
                temp = temp - xk[i-1]; //left of diag
            }
            if (i<(vec_size-1) && (i+1)%innerLen!=0)
            {
                temp = temp - xk[i+1]; //right of diag
            }
            if (i<vec_size-(innerLen))
            {
                temp = temp - xk[i+innerLen];
            }
            xkp1[i] = (temp*aij + b[i])*daii;
        }
        xk.swap(xkp1);
```

```
            runtime = omp_get_wtime() - t0;
            stop = runtime > maxTime_s;
            if (stop)
            {
                iterationsDone = iteration+1;
                break;
            }
        }
    }
// End code snippet
```

The chunk size for the schedule(static) and schedule(dynamic) was manually set to the above value. When selecting the static schedule type, it should default to the above value, but we encountered issues with this during testing and therefore chose to set it manually to the value it should use in default. The default behaviour for schedule(dynamic) should be $chunksize = 1$, but we encountered a bizarre behaviour in this case as well. Without manually setting the chunk size, our program did not scale at all and it even performed worse when run multithreaded compared to single threaded. We therefore chose to set it manually here as well.

When one compares the (single thread) performance of our code to the reference version, it becomes immediately obvious that our version is comparatively slow. The slower performance is explained by our approach to the storage of the grid. We only store the inner grid points of the domain in a std::vector and therefore have to be careful when the stencil would include elements from the boundary of the domain. This is done via four if-checks in the inner most for loop. The boundary terms are taken of by including them in the right hand side vector of the equation. For a more in depth explanation, see our submission for exercise 2.

A better performing variant is realized in the reference implementation where the whole grid (including the boundary points) are stored in one std::vector. The inner most loop(s) for calculating the next iteration of the Jacobi method then only loops over the inner points and leaves the boundary points as they are. The main advantage of this method are that it does not require the four if-checks and thus needs $4 * (resolution - 2)^2$ operations (roughly speaking) less per iteration. Additionally, we believe that the if-checks might also (partially) break data locality compared to the reference implementation. However, our implementation will also use $2 * 4 * (resolution - 1)$ elements of *double* less memory. The first 2 represents the grid and right hand side vector, the $4 * (resolution - 1)$ represents the number of elements needed to store the boundary.

Keeping the non-optimized state of our implementation in mind, we will now move onto the parallelization and scalability analysis.

4.2 Parallelization and scalability analysis

Goal: Investigate the scalability and the parallelization by plotting the speedup of the average iteration runtime for 1, 2, 4, 8, 10, 20 and 40 threads on the cluster for three different OpenMP scheduling policies for the parallelized loop: schedule(static), schedule(static,1), schedule(dynamic) Plot and discuss the results for resolutions 256, 512, 1024, and 2048.

In Figure 6, the plot for the average runtime per iteration is shown. As mentioned above, the chunk size was set manually for both the *dynamic* and *static* schedule type, so that every thread receives one chunk of equal size. $Static, 1$ refers to static schedule with $chunksize = 1$. $dynamic, innerLen$ uses a chunk size that was set to the "length" of the inner grid points on one axis ($innerLen = resolution - 2$).

Every simulation uses the same resolution ($resolution = 256$), but different numbers of threads and different scheduling policies. One can clearly see that the $static, 1$-policy scales by far the worst and actually plateaus after a certain point (4 threads). A chunk size of 1 is obviously bad

because it breaks any chance we have in keeping spatial and temporal locality (within the threads). It was actually surprising to us how well this policy kept pace for thread counts below 4. The static schedule with optimized chunk size performed the best but the dynamic schedule with optimized chunk size performed almost as well. The additional overhead caused by the dynamic schedule was really only noticeable for 20 threads and more. We will talk about the differences between the dynamic schedules later.

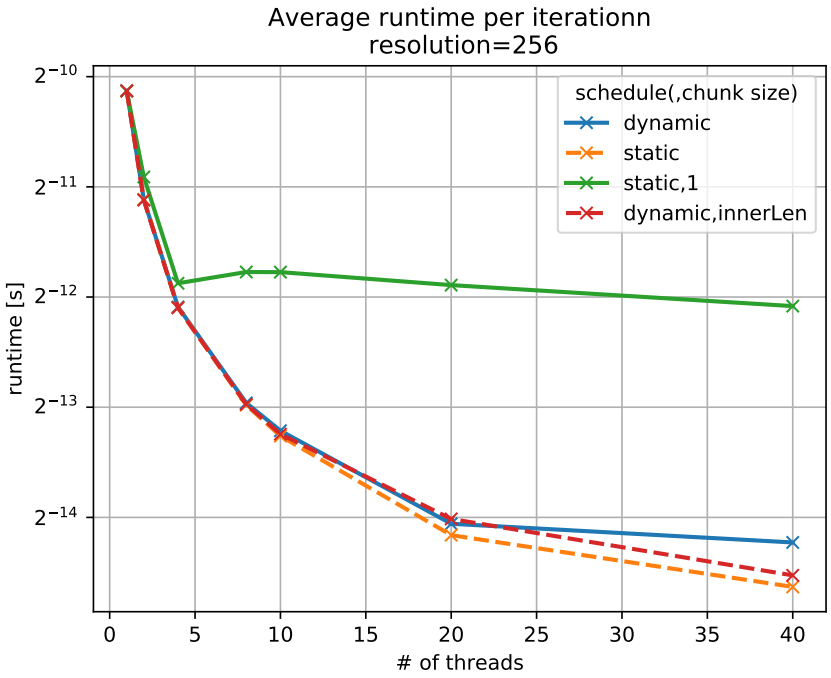

Figure 6: Plot of the average runtime per iteration with $resolution = 256$. The y-axis (runtime) uses a logarithmic scale in base 2.

The plots for the remaining resolutions $(512, 1024, 2048)$, Figure 7 - Figure 9, show very similar behaviour except for one aspect. As the resolution is increased, the dynamic schedule with the same chunk size catches up to the static schedule for high thread counts. In the last plot, Figure 9, we can see that both schedules perform almost identical.

We believe this can be explained in the following way. As the resolution increases, the length of the grid vector increases as well ($vector\_size = \mathcal{O}(resolution^2)$) and therefore the length of the for-loop needed to calculate the next iteration of the Jacobi method. We manually set the chunk size in a way, that divides the parallelized for-loop into equal chunks so that one thread receives only one of these chunks from the scheduler. As the size of the for-loop increases, the overhead caused by the scheduler will become less and less noticeable since the calculation of one iteration takes much longer in the first place.

The dynamic schedule type performs better for the smaller chunk size ($chunk\_size = innerLen$) because it can dynamically react to threads that finish faster. The load is thus (approximately) balanced between the threads even if some threads execute slower. Slower executing cores/threads might be caused by various factors such as certain chunks of the for-loop being more time intensive to calculate (should not be a big issue in our case) or even slight differences between the physical cores.

Overall, for this particular task, the static schedule still seems to work the best since every iteration of the parallelized for-loop should take almost equal time to resolve.

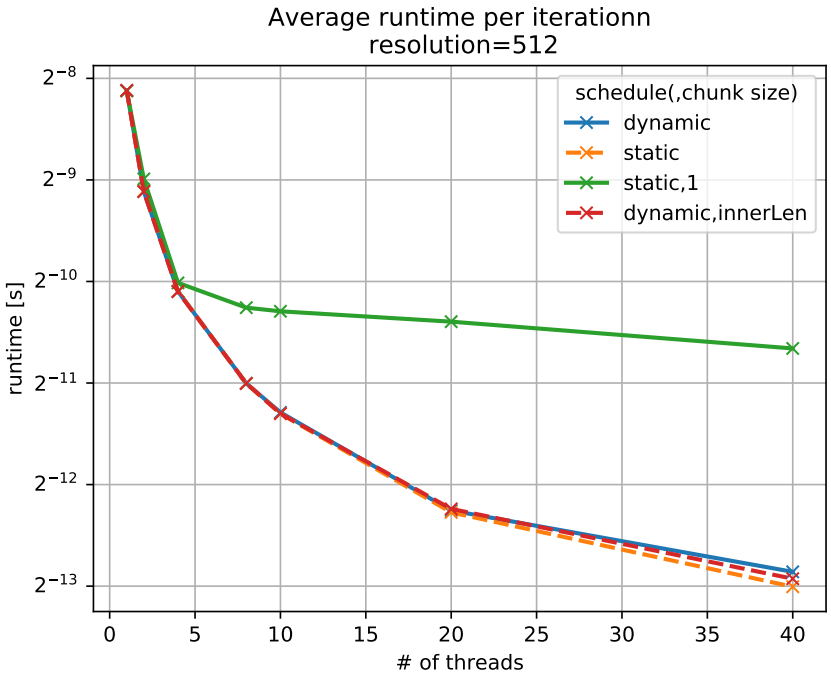

Figure 7: Plot of the average runtime per iteration with *resolution* = 512. The y-axis (runtime) uses a logarithmic scale in base 2.

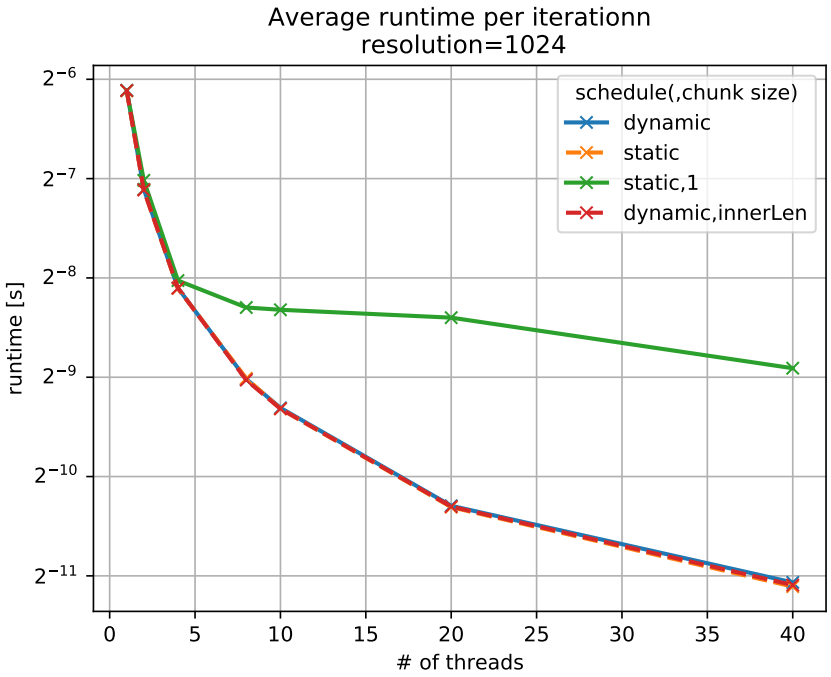

Figure 8: Plot of the average runtime per iteration with *resolution* = 1024. The y-axis (runtime) uses a logarithmic scale in base 2.

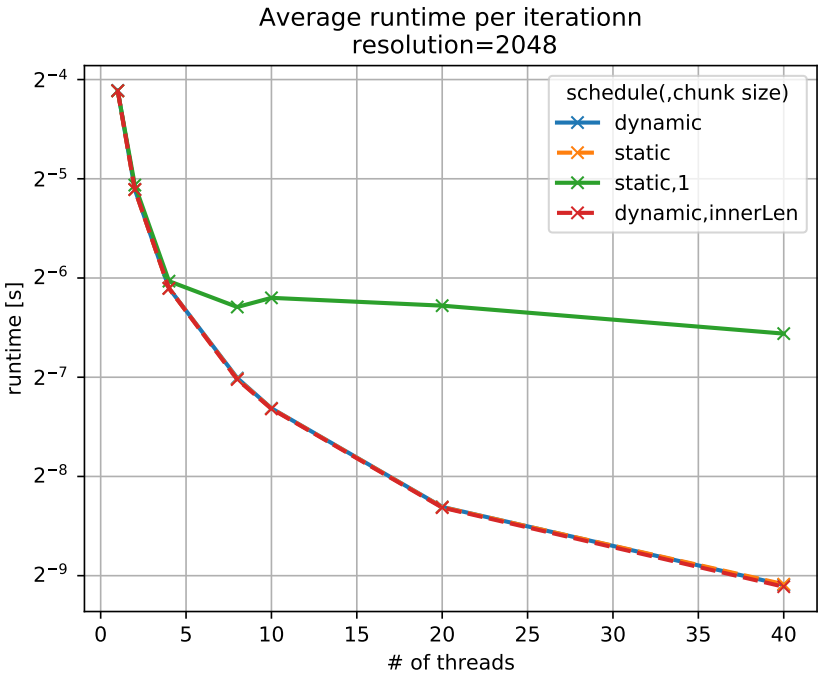

Figure 9: Plot of the average runtime per iteration with *resolution* = 2048. The y-axis (runtime) uses a logarithmic scale in base 2.

References

[1] `https://www.researchgate.net/publication/228862503_On_the_Behavior_of_the_Residual_in_Conjugate_Gradient_Method`