# 360.242 **Numerical Simulation and Scientific Computing I**, 2024W

Exercise #3: Task 4

Group 05

TU Wien —January 9, 2025


Team: Kablanbek Abdyrakhmanov (12347305)
Rodrigo Brito Interiano (12347308)
Alice de Carolis (12402529)
Leonie Theresa Greber (11801674)


# 4 Parallel Jacobi Solver

## 4.1 OpenMP Parallelization

The following section provides an argument for why we chose to parallelize particular for-loops.

### 4.1.1 Code given with Task description

- Initialization Loop: Initializes the default values and west and east boundary. Could be parallelized because each cell is updated independently, therefore the for-loops are independent of each other. Collapse(2) is used to parallelize both dimensions i and j.
- Jacobi Calculation Step Loop: Updates each point independently, enabling parallelization similar to the initialization loop.
- South Boundary Loop: Parallelizable due to point independence. Utilizes Collapse(2) for parallel execution over dimensions i and j.
- North Boundary Loop: Similar to the south boundary loop.
- Norm-2 Calculation Loop: Because each point contributes independently, we can parallelize the for-loop. Requires a reduction (+:sum) to combine partial results from all threads in the end.

- Infinity Norm Calculation Loop: Again each point is compared independently but as with the Norm-2 Loop we need a reduction: reduction(max:max) so that it is ensured that the infinity norm find the global maximum and to prevent race conditions.

#### 4.1.2 Code from Ex2

- Build Loop: Initialize each point independently, and by using collapse(2) we can parallelize over k and j.
- Solve Loop: Updates each point of the jacobi iteration independently, but only for the two inner loops. To ensure that the swapping of the old and new value is executed by a single thread '#pargma omp single' is used to prevent race conditions from occurring.
- Resid Loop: Due to the independence of the residual point calculations, the loop can be parallelized over both dimensions.
- Norm-2 Loop: As with the given implementation, we can parallelize the calculation of the sum by ensuring the correct combination with reduction(+:sum).
- NormInf Loop: Checks each point independently and by utilizing reduction(max:norm) it is ensured that the results of the threads are combined correctly.

### 4.2 Results

**Remark:** Due to some inconsistencies in the cluster calculations, minor adjustments were made to the graphs. Note that the runtime calculations did produce some outliers. We tested different slurm scripts for reliability, but it was not possible to find a configuration that convinced us. We decided to use three distinct Slurm scripts (one for each policy), each reserving 40 CPUs for calculations, to maintain consistency across all thread configurations.

#### 4.2.1 Code Given

We obtained the following plot for the speedup of the total runtime for 1, 2, 4, 8, 10, 20 and 40 threads for three different OpenMP scheduling policies:

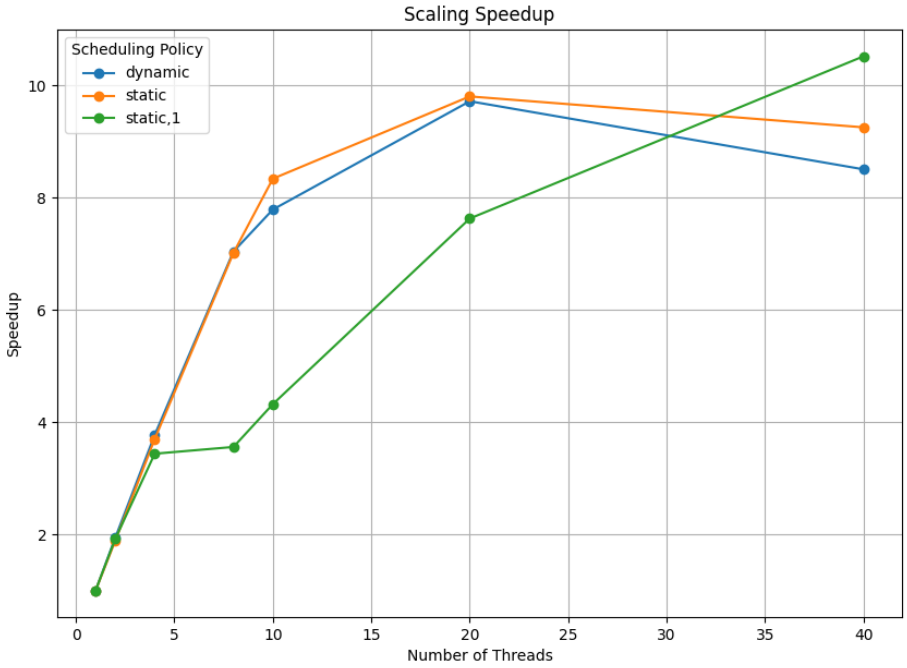


Figure 1: Speedup

The plot demonstrates that the speedup closely approaches the ideal for the 'dynamic' and 'static' scheduling policies when using up to 10 threads. However, the 'static' policy with 1 chunk exhibits a flattening of the speedup curve even earlier than the other two policies.

Therefore, we observe strong scaling behavior for the first 8-10 threads. Beyond this point, factors such as data traffic between nodes and network latency may be limiting further performance gains.

Inspecting the data, the performance of the 'static',1 policy with 40 threads might be an outlier, as discussed previously. Alternatively, this configuration could represent the most effective approach if a high number of threads is available.

Overall, the 'static' approach generally has a slightly better performance than the 'dynamic' approach, although the difference between the two policies is not substantial.

Following plots show the runtime of each approach in seconds vs. the thread count and the efficiency.

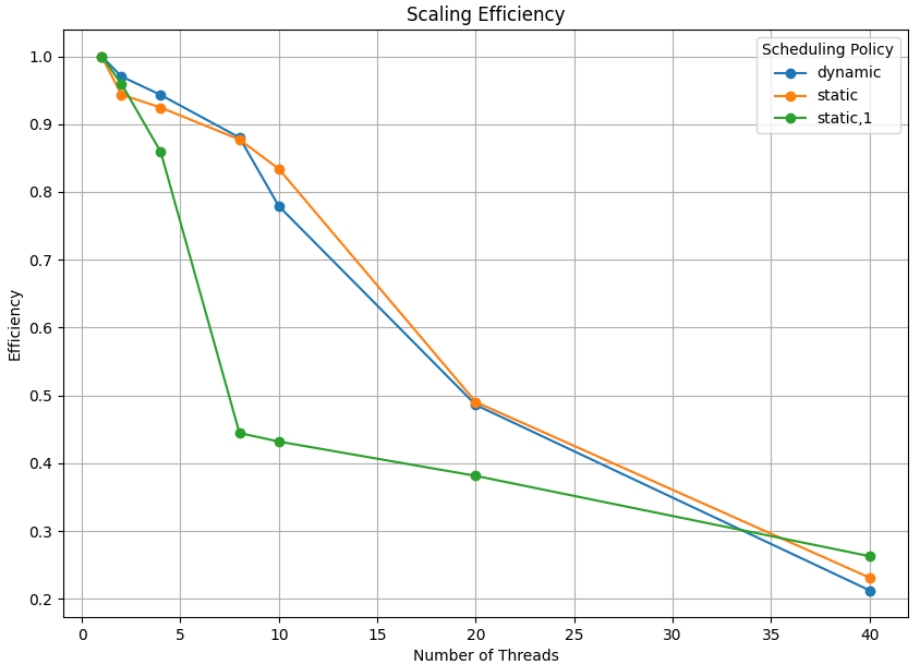


Figure 2: Efficiency

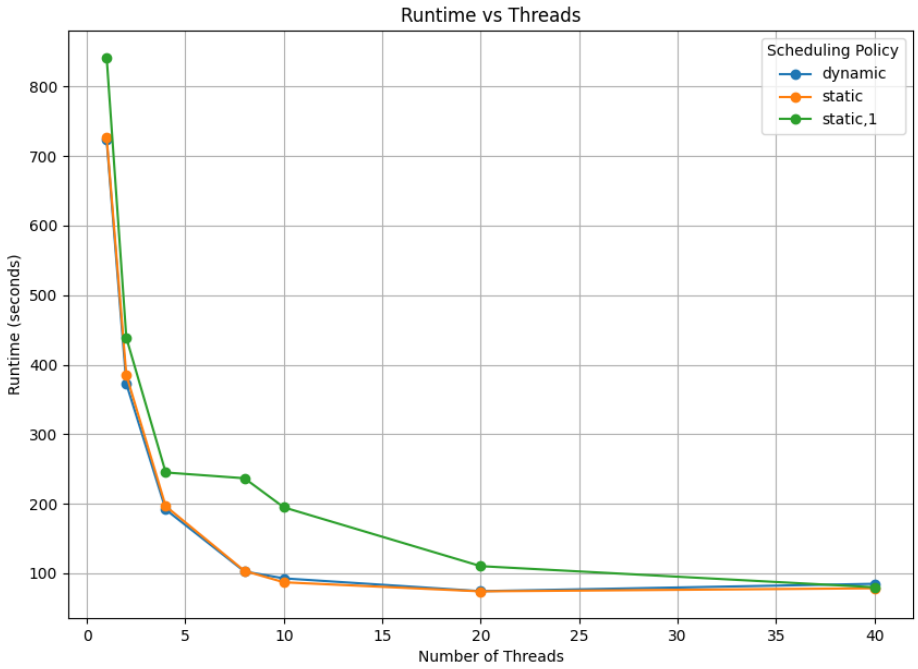


Figure 3: Runtime

### 4.2.2 Code from Ex2

We also performed the processing for the code from Exercise 2. However, the speedup plot for this exercise reveals that the execution time remains the same across all scheduling policies.

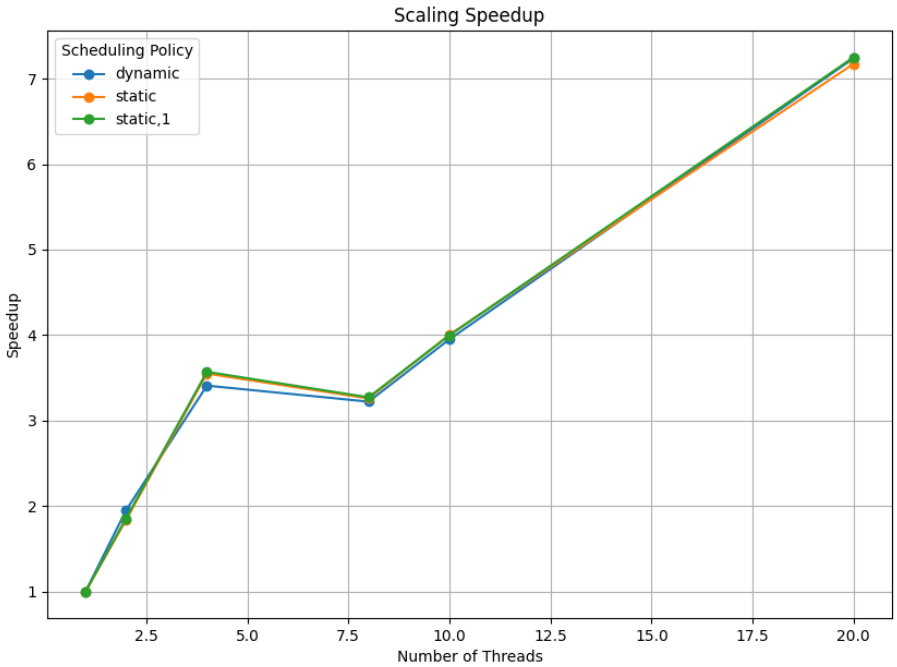


Figure 4: Speedup

This behaviour might be attributed to a highly uniform workload distribution. However, further investigation would be required to determine the exact cause, which exceeds the scope of this assignment. Additionally, the solution for the given code has already been discussed above.

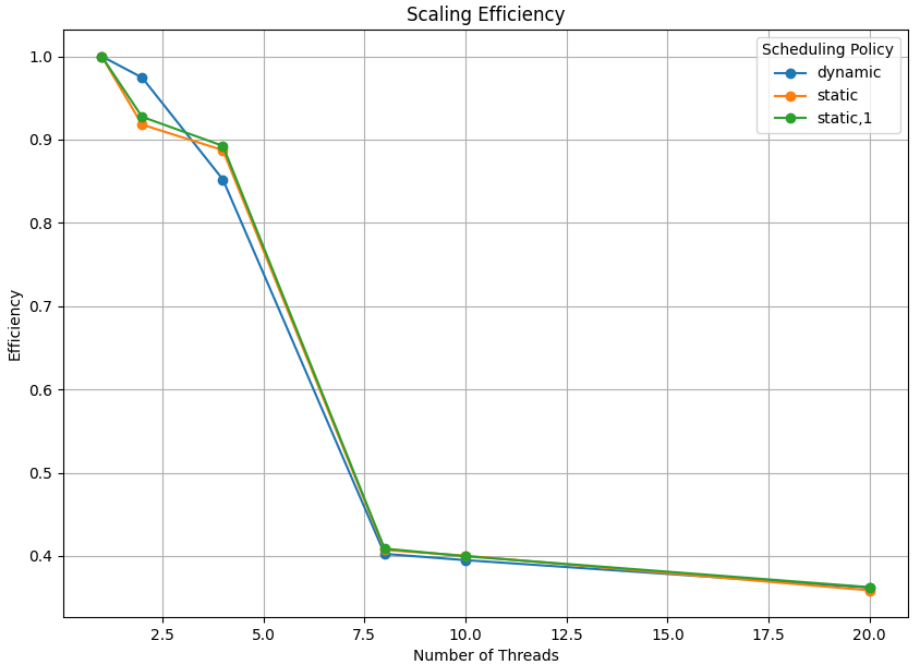


Figure 5: Efficiency

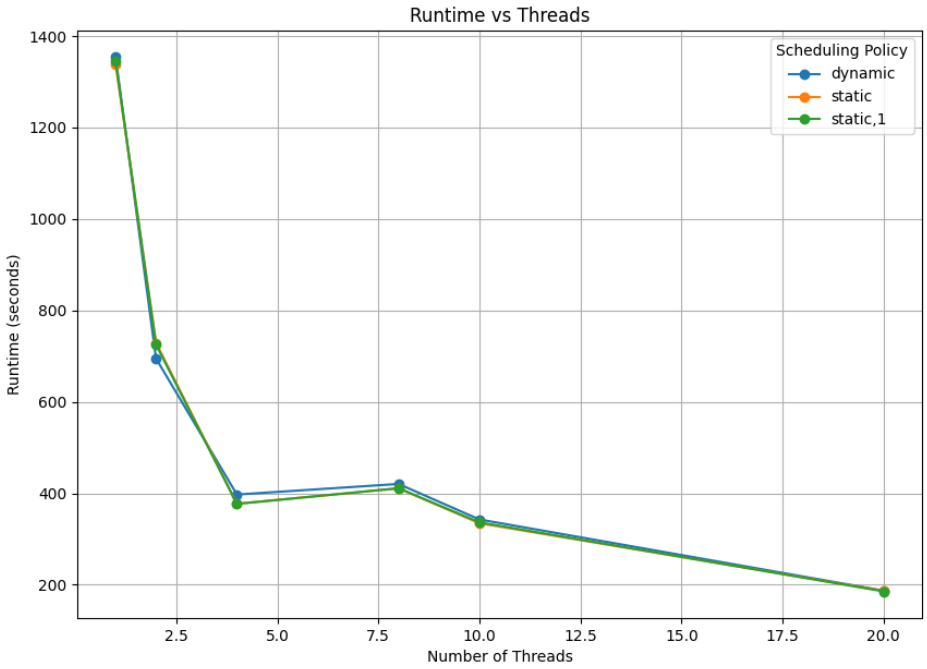


Figure 6: Runtime

## Build and Run

1. **make clean**: cleans up generated files
2. **make all**: compiles programme defined in `main.cpp`
3. **make run**: runs programme with resolution = 2048 and 100000 iterations
4. **make run_small**: runs programme with resolution = 128 and 1000 iterations
5. **make run_test**: runs programme with resolution = 128 and 1000 iterations and it is possible to manually set the number of threads and scheduling policy that should be used