# EX3: Task 4

[January, 2024]


### Group Members:

- Marcus Konrath, 01300688
- Uğur Cem Erdem, 12346958
- Katharina Perl, 01426575


## CPP Code

We started with the code provided for the Jacobi solver, as our implementation from Exercise 2 was not functioning as expected. To streamline the task and avoid unnecessary outputs we modified the program by adapting the output formatting and disabling the call to the write function that generated CSV files. We made this adjustment because the CSV files were not required for this task and were cluttering the output directory.

To parallelize the Jacobi solver we identified the most computationally intensive part of the code: the nested loop that updates the interior points of the grid. We chose to parallelize the outer loop over the grid rows (for (size_t j = 1; j ¡ N - 1; ++j)). The computation for each grid point in the Jacobi method only depends on neighboring values from the previous iteration. This independence allowed us to distribute the workload across threads without introducing data dependencies. We used OpenMP #pragma omp parallel for.

## Makefile

One of the challenges we had was testing three different scheduling modes: static, static,1, and dynamic. At first, it seemed like we'd need to create and manage three separate .cpp files, each with the scheduling mode hardcoded. To avoid this redundancy and simplyfy the process we came up with a better solution using the Makefile. Our Makefile dynamically generates three separate executables, each with a different scheduling mode. It uses preprocessor macros, which we defined at the beginning of the .cpp file (SCHEDULE_MODE and OUTPUT_MODE).

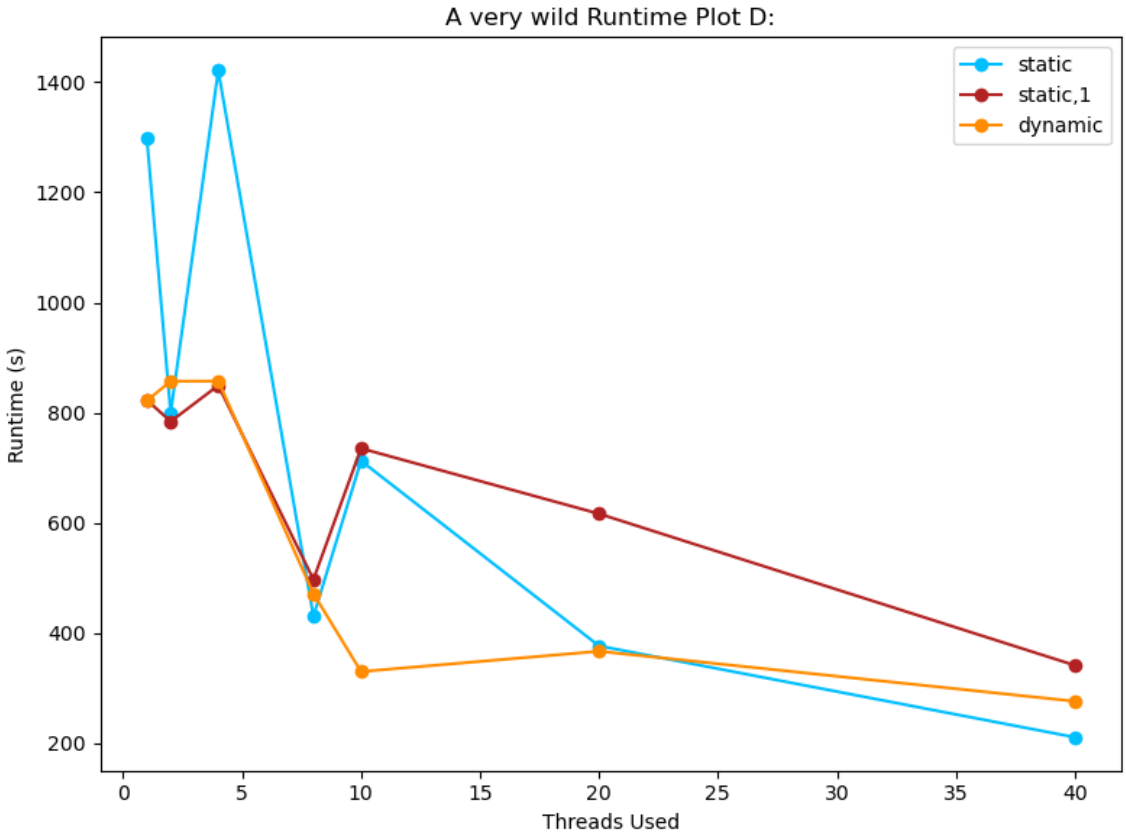
# Batch File

In Task 3, we had to run our program multiple times with different combinations of threads and cores. Doing it manually (editing the batch file on the cluster, saving it, running it for each configuration...) was slow and a bit frustrating. To make this process more efficient we developed a script that automates these steps. The script loops through preset arrays of thread and core counts, creates a job script for each combination and submits it automatically. This saved us a lot of time, reduced mistakes and made sure all the jobs were set up consistently.

Selection of core numbers: see report of task 3; we used the formula explained there

# Output

When we first ran our code on the HPC cluster, we were surprised by the runtimes. Specifically, with a lower number of threads, the runtimes did not behave as we had expected. As the thread count increased, the runtimes sometimes even slowed down instead of improving, contrary to what we anticipated. Our very first output looked like this:




After fine-tuning and making some little adjustments to the code, the runtimes improved. For the best-performing run we encountered the following runtimes:

| Threads | Runtime |
|---|---|
| 1 | 1607.12 |
| 2 | 1383.61 |
| 4 | 1410.47 |
| 8 | 796.474 |
| 10 | 574.231 |
| 20 | 223.734 |
| 40 | 173.804 |

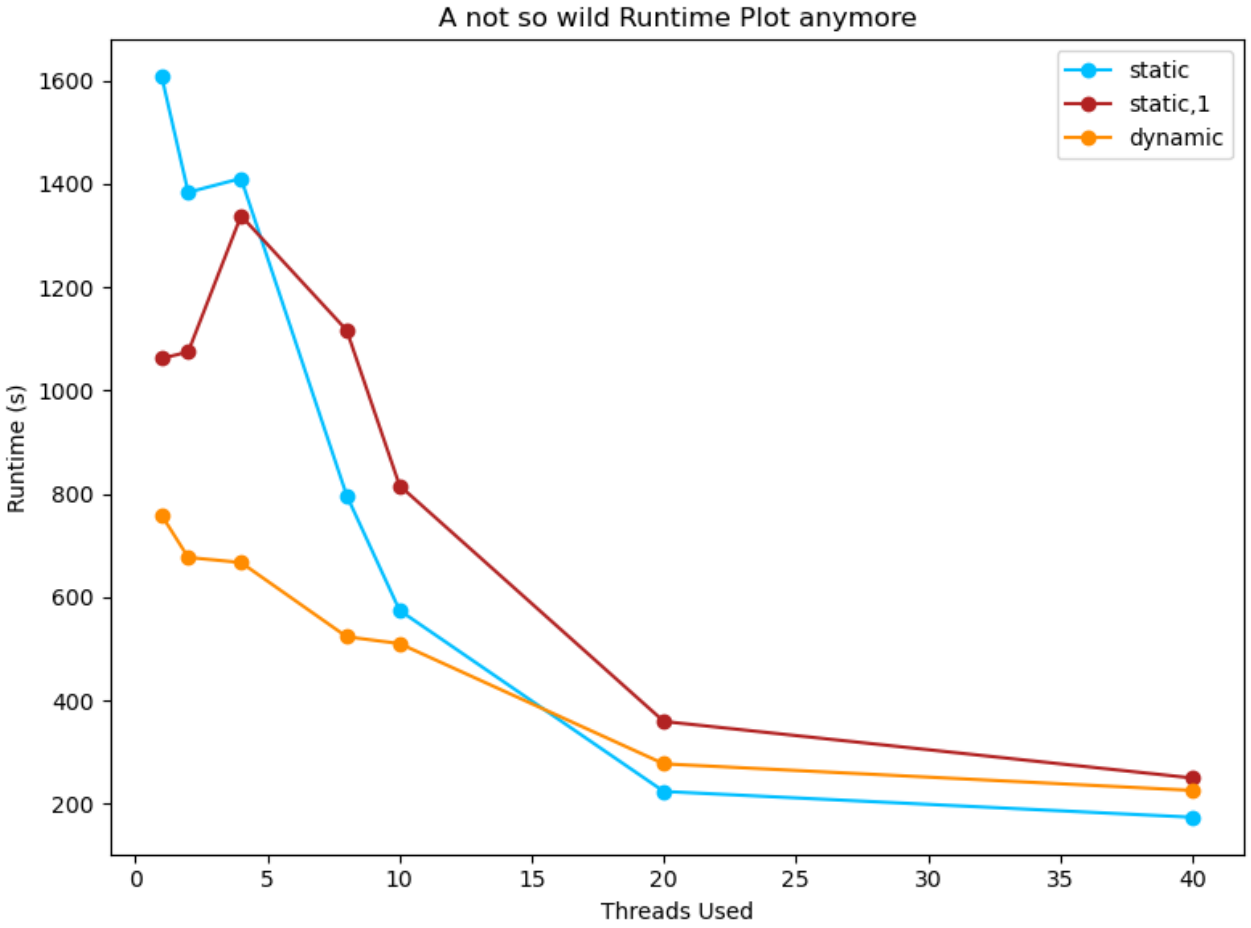
Table 1: * static

| Threads | Runtime |
|---|---|
| 1 | 1062.05 |
| 2 | 1074.59 |
| 4 | 1338.26 |
| 8 | 1116.32 |
| 10 | 815.824 |
| 20 | 359.266 |
| 40 | 249.827 |

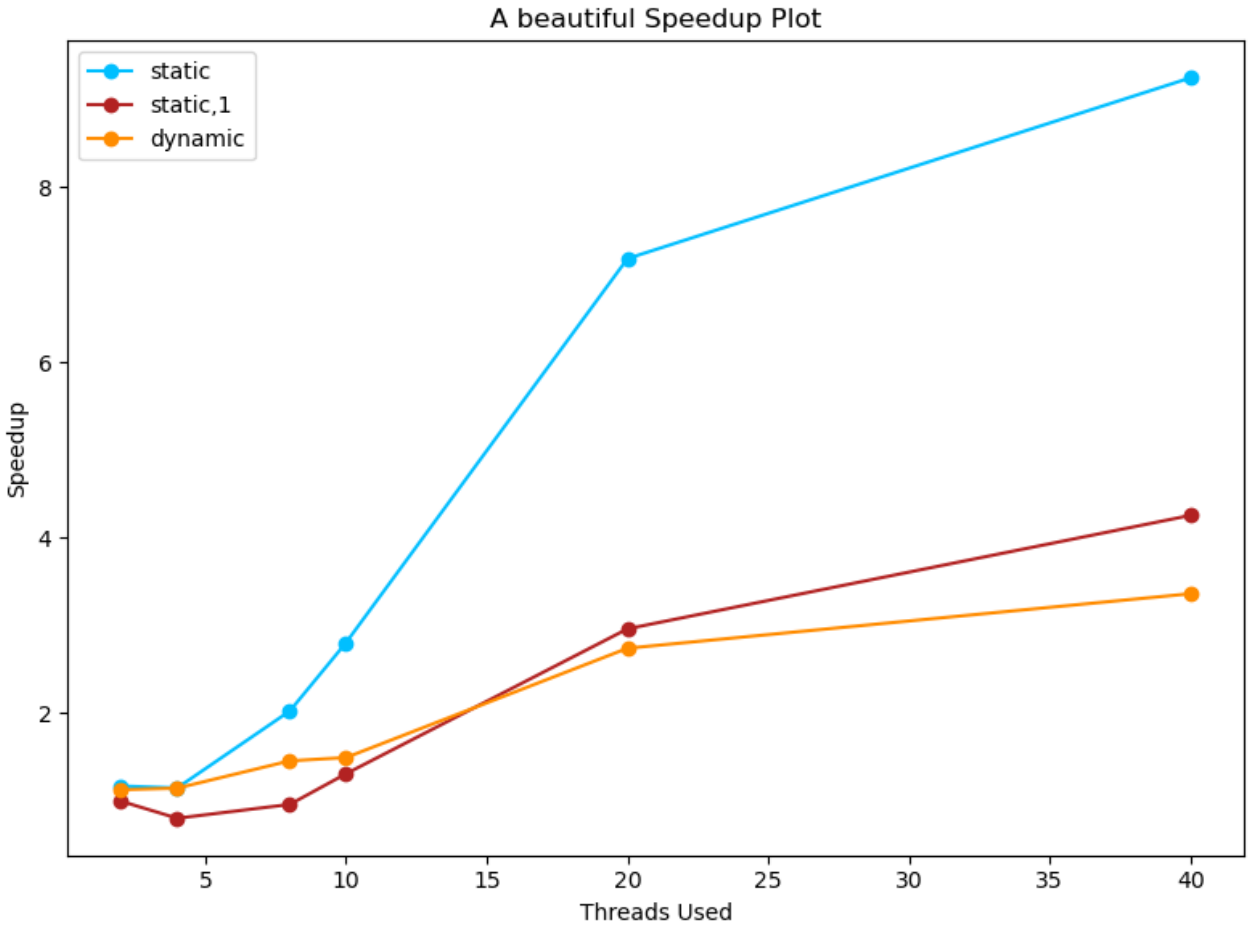
Table 2: * static,1

| Threads | Runtime |
|---|---|
| 1 | 758.044 |
| 2 | 676.84 |
| 4 | 666.89 |
| 8 | 523.176 |
| 10 | 510.006 |
| 20 | 277.205 |
| 40 | 225.815 |

Table 3: * dynamic




We also had a look at the speedup: see next page




As shown in the best-runtime plot, the runtimes now behave more as we expected. With fewer threads, the runtimes are generally higher, but they start decreasing noticeably as the thread count increases. There are still a few points where the runtimes don't decrease smoothly, but the overall trend is pretty much what we expected. We thought that 'dynamic' scheduling would give the best runtimes because it prevents threads from waiting. Unlike 'static' scheduling, where the work is evenly split among threads right at the start, 'dynamic' scheduling assigns tasks to threads as they finish, which helps balance the load and reduces the chances of some threads sitting idle. As we can see in the plot, 'dynamic' does indeed have lower runtimes, but with higher thread number, static seems to be better. Also in the speedup plot, 'static' received the best results.

**Conclusio:** In our case, the 'static' scheduling mode appears to be the most effective choice when working with higher thread numbers. On the other hand, for lower thread numbers, 'dynamic' scheduling proves to be the better option.