Praktikum im SS 2022
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Vincent Bode, MSc., Minh Chung, MSc., Dennis-Florian Herr, MSc., Bengisu Elis, MSc., Dr. Karl Fürlinger, Amir Raoofy, MSc., Dr. Josef Weidendorfer

Assignment 6 - Project MG – Due: June 30th, 2022

1. Background

This assignment is the first of two projects, making up the 2nd part of the lab. You will on a specific architecture assigned to your group, and we expect you to make use of the knowledge you gained during the 1st part of this lab. We provide you with a source code in sequential version, and we expect you to parallelize and optimize this code (e.g. through tiling). See the `src/` folder in the repository.

The code is about a typical HPC problem: solving a linear system of equations ($\mathbf{A}x = b$) via iterative approximation. Such solvers are at the heart of many number-crunching problems, and thus they are representative for a large fraction of time spent on HPC systems.

The concrete problem we look at is to calculate the static heat distribution on a 2d quadratic metal plate (unit square), with given fixed temperatures on the 1d plate sides as boundary condition. The problem is stated as partial differential equation, which can be solved via Finite Difference discretization on a regular grid (see Figure 1) over the problem domain, resulting in the mentioned linear system of equations. The temperature at each grid point is an unknown in this system.

While there are various approaches, we consider the following solvers:

- Jacobi smoother[1]
- Multigrid scheme[2]

Jacobi smoother

The *Jacobi* smoother is an iterative solver for linear equations. It starts with an initial guess for the unknowns, and it updates the solution iteratively. Two 2d arrays are used: one contains the approximation of the unknowns from the previous iteration $i-1$, and results for iteration i are written to the other array. In the next step, the arrays exchange their roles. To calculate the new approximation of an unknown at each inner grid point, the solver takes the values of direct neighbors into account. This is called a stencil update, with the stencil specifying which

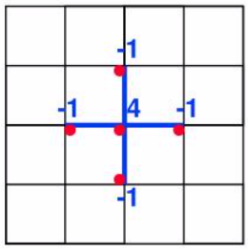

Figure 1: Illustration of a Jacobi update step with a 5-point stencil for a grid point.

neighbor values and coefficients should be used for an update. For our concrete problem, we use a so-called 5-point stencil (see Figure 1).

Multigrid scheme

While a Jacobi smoother on the final grid resolution does converge to the solution, the convergence is very slow. To speed up the process, a hierarchical scheme can be used with a series of grids with different resolutions, going up and down, together with a number of smoothing passes (e.g., using the Jacobi smoother). This scheme is called Multigrid, and it can be proven to have much better convergence than the Jacobi smoother. On coarser levels, we do not solve the original system of equations, but we solve for a residual equation ($r = b - \mathbf{A}x$), trying to find the right error correction of the current result to become a better approximation. Going to coarser level is called *Restriction*, resulting in one value every second row and column with $1/4$ of the number of grid points. Going to finer resolution is called *Prolongation*, which does an interpolation of found error corrections added to the previous values on the finer grid. There are different strategies to visit the different grid resolutions (V vs. W cycle). On each coarsening/refining step, we do a number of Jacobi smoother passes.

See Figure 2 for a graphical illustration of the different steps involved. Usually, it is assumed that the solution is good enough when the sum of error corrections found for all values on the finest grids is below a given threshold. However, in this project we only look at the time spent to run a given number of Multigrid "V" cycles using a given finest resolution.

Running the code

```
./multigrid preSmoothingSteps postSmoothingSteps numberGridLevels
   numberMultigridIterations
```

Listing 1: Parameters of the multigrid executable.

The code has the problem hard-coded; the following command line parameters are available:

- *preSmoothingSteps* : number of Jacobi passes done before going to coarser grid resolution;
- *postSmoothingSteps* : number of Jacobi passes done before going to finer grid resolution;
- *numberGridLevels* : log_2 of the number of grid points in each dimension; increment by 1 results in 4 times more grid points used in the discretization;
- *numberMultigridIterations* : number of V cycles done.

After each iteration of a Multigrid cycle, the code prints out the error sum.

[1] https://www5.in.tum.de/lehre/vorlesungen/sci_compII/ss18/smoothing.pdf
[2] https://www5.in.tum.de/lehre/vorlesungen/sci_compII/ss18/multigrid.pdf

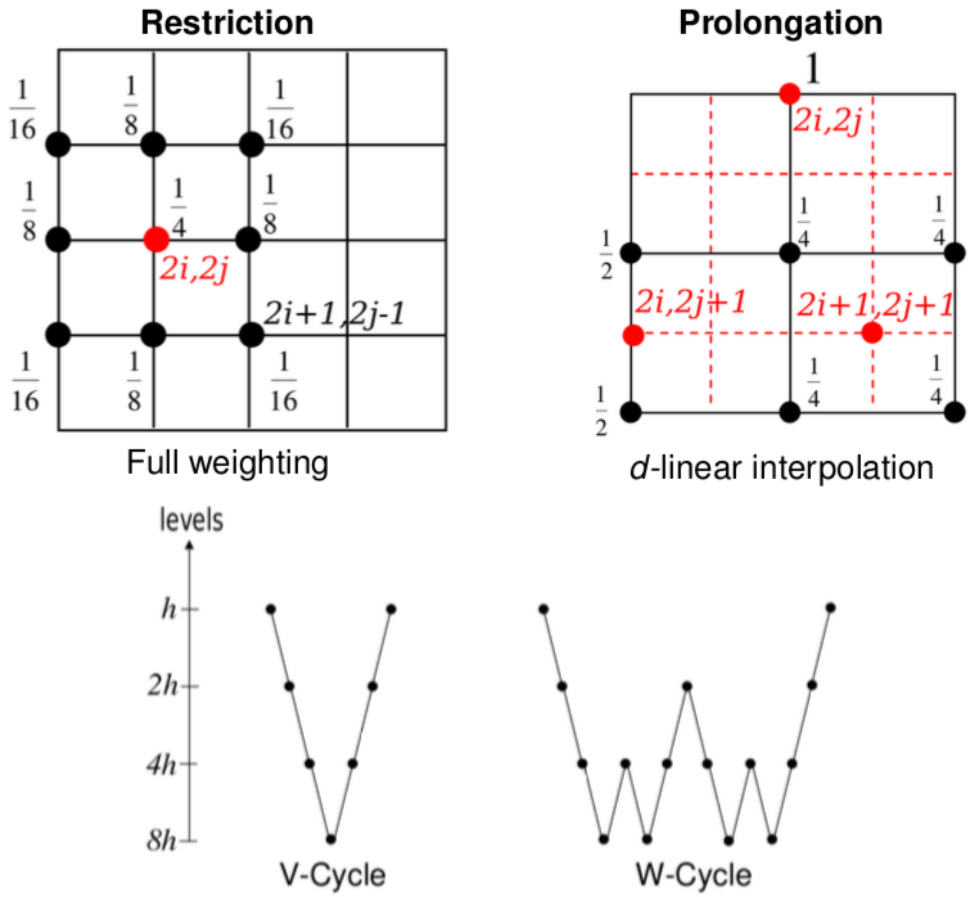

Figure 2: Illustration of the actions done in a Restriction and Prolongation, as well as typical V and W cycles of a Multigrid scheme.

Parameters to use for comparison of runtime in the report (and among student groups)

Use the following parameters for time measurements:

```
./multigrid 4 6 14 8
```

Listing 2: Parameters of the multigrid executable.

Note that for this configuration, the code allocates around 8 GB of memory.

2. **Hints and Ideas for Optimization**

   (a) Preparation for profiling: With high optimization, all functions may be merged together due to inlining. Explicitely tell the compiler to not inline selected functions to get a better profile. Make sure this does not create overhead. Furthermore, it is instructive to see the time spent on the different coarseness levels of the multigrid scheme in the profile. E.g. you can add the coarseness level as template parameter by making corresponding functions into template functions. Describe your steps in the report and show the improved profile results.

   (b) Parallelization: Use a profiler to check where most of the time is spent. Which functions should be a target for OpenMP parallelization? For each function, which loop is the best to add an OpenMP directive? On coarse grid levels with small grid size, it is contraproductive to parallelize. Try to find the right level beyound which parallelization is

useful. In your report, motivate improvements steps by showing profiling results, and provide performance numbers for each improvement step you do.

(c) Cache optimization: a good strategy for reuse of data from cache is to do multiple Jacobi passes interweaved in one sweep over the data. For this, do a profile run checking for cache misses triggered in the code. For optimization, you can do an update over a few grid lines or over a 2d block of the grid, and directly reuse the new updated values for a partial 2nd Jacobi pass over the same region (line-wise is called ”wave propagation”, block-wise is so-called ”diamond tiling”). E.g. if you have done updates for row 1 and row 2, you can start the next Jacobi pass already on row 1. This is extended to block-wise by only updating sections of rows (to allow reuse to happen even with small L1 caches). Be careful about data dependences. Also, think about how to interweave more than 2 Jacobi iterations.

3. **Tasks for CPU Architectures**

   (a) SIMD - Check if the compiler was able to auto-vectorize the various inner loops, especially in the Jacobi solver. Try to make use of intrinsics.

   (b) Cache-optimization in Jacobi solver/smoother.

   (c) Parallelization with OpenMP. Start with parallelization without cache optimization. As 2nd step, try to parallelize your cache-optimized code.

4. **Tasks for GPU Architectures**

   (a) OpenMP: use OpenMP offloading to run on GPUs. Hint: keep data on the GPU as long as possible.

   (b) Multi-GPU: Utilize multiple GPUs for both versions of your code.

   (c) Make use of a Low-level vendor specific programming model (CUDA/HIP).

   (d) Shared Memory: On GPUs Shared Memory is faster than Global Memory. Make sure your data is used from shared memory. Tip : Adjust your block sizes so that data for each block fits into dedicated shared memory.