# Assignment3 (Resit): Five-Point-Stencil

Given is a C program for implementing the approximation of a simple heat diffusion scheme in 2D (five-point stencil).

It iteratively computes matrices of size $n*n$ from an initial $n*n$-matrix That matrix has all values set to `0.0` but the value in the middle of the first row, ie., the value at index $[0, n/2]$. That value is initialised by a heat value which is provided as an argument to the program. The iteration step recomputes all non-boundary elements by computing a weighted sum of the current values of the element itself and its four immediate neighbouring elements.

NB: The C code provided takes two arguments, the heat value and the number of rows/columns of the matrix. As a first try, you may want to run `./a.out 40 15` which leads to a matrix of 15 by 15 elements and an initial heat value of `40.0`.

This assignment explores how this algorithm can be executed on a multicore system using MPI. Your tasks are to:

- Measure the performance of the sequential version
- Develop an MPI version
- Measure the performance of your parallel version

This assignment needs to be done individually.
Make sure that you:

- specify exactly what hardware is being used (CPU version, clock frequency, memory, etc.)
- specify exactly what software is being used (MPI version, compiler version, compiler flags, etc.)
- specify exactly which parameters (HEAT and EPS) you are using

## 1 Task 1: Sequential Evaluation

### 1.1 Runtime

Evaluate the performance of the sequential code. For that purpose, comment out all terminal outputs within the main iteration. Insert time measurements just before and after the main loop. Use the highest level of compiler optimisation on your machine. Typically, this is -O3 but you should look into the man pages of your compiler. Present the wallclock time as a function of the number of elements in the matrix ($n^2$). Make sure that you vary the size of your matrix from sizes of a few kB to something as large

as your machine permits with reasonable runtimes. You may have to adjust the HEAT parameter for your machine to obtain a reasonable range of runtimes. Higher HEAT values lead to longer runtimes.

### 1.2 FLOPS

Compute the FLOPS (floating point operations per second) your program has achieved for the different input sizes and produce a bar graph with the number of elements on the x-axis and FLOPS on the y-axis. As an approximation of the total floating point operations done in a program run that iterates $it$ times over a matrix of size $n * n$ you can assume that you perform $it * 9 * n * n$ floating point operations. This ignores the stability test but slightly over-approximates the relaxation step wrt. the boundary elements.

## 2 Task 2: MPI

### 2.1 Implemention

Rewrite the program to execute the iteration in parallel using MPI. To do so, you must not materialise the full matrix on any single rank. Instead, you need to come up with a partitioning of the matrix between your ranks. Bear in mind that you need to have an overlap in your partitioning as you need to be able to access all the neighboring elements for all non-boundary elements of the (virtual) global matrix! Bear also in mind that the standard MPI calls `MPI_Send` and `MPI_Recv` are synchronous: If all ranks send at the same time and there is no rank performing a receive, your program deadlocks!

### 2.2 Description

Provide a short description on how you parallelised the code. How do you partition the matrices? How do you communicate? How do you perform the stability test? Relate your description to the source code.

## 3 Task 3: Performance

Repeat the exercise from Task1 for the MPI version. Measure the runtimes (no runtime graphs needed) and then provide graphs for the FLOPS achieved on different problem sizes. Using the results from Task 1 as sequential base line, compute the speedup and efficiency achieved for different numbers of processors/cores used.