# Lab Course: Efficient Programming of Multicore Processors and Supercomputers

Group 1: Jonas Mayer, Paul Preißner, Konrad Pröll

July 2, 2017

## 5. Parallelization of heat with MPI

We at first noticed that the program consists of four different parts: The parsing of the input, the initialization of the values, the actual relaxation and the writing of the output.

### Input Parsing

When parsing the input, we quickly found two different concepts: Parsing in one process and broadcasting the data or parse on all different processes. First, we thought that parsing in one process and broadcasting the data would be the smarter choice, as we would not have trouble managing the IO, however, we noticed that broadcasting the heatsources, an array of structs, was not as trivial as we thought, so we changed to all processes reading the data. This is in our opinion acceptable, beacause the input file in this particular application is pretty small. In general, you do not want to parse large files in every process.

### Initialization

We found multiple difficulties in this section. First, the resolution might not be divisible by the number of processes in one direction. In general this should not be as much of a problem, because you usually only run an mpi-parallelization for a very large inputsize, but as this was not mentioned in the task, we had to deal with it. We decided that we then would want to pad the array with '0', so that every process has a same size array. Also, initializing the heatsources at first seemed like it was difficult, however, because we had a cartesian topology, we could simply use MPI_Cart_coords to determine the coordinates of a certain process and therefore determine a) whether it sits at what would be a border in the global array and b) how much offset it has from 0,0, so that we can calculate the distance to the sources correctly.

## Relaxation

The next step was to synchronize the data between the neighboring processes. To simplify this, we made use of the MPI topologies (MPI_Cart_shift). Therefore, we had each process sent the outermost calculated gridpoints to the neighboring processes according to the cartesian grid, and receive the data from the neighboring processes into the outermost gridpoints, which where not recalculated due to missing neighbors. In order to do this communication, we used selfdefined datatypes to describe the different datalayouts for sending the values up/down and left/right: Up and Down are continous, while left and right are vectors, in which each entry has a distance of linesize to the last one.

In the beginning, we did this routine right in heat.c after the call of the relaxation, but optimized it later so that we swapped the data nonblockingly at the beginning of the calculation, calculated the gridpoints which do not rely on data from other processes, which are located at coordinates 2 to np-2. Afterward, we synchronize the communication and calculate the last layer.

Also, you can use blocking communication with MPI_sendrecv. However, this is slightly slower than nonblocking communication.

## Gather the Output

After every process has computed the relaxations on its tiles, the output has to be gathered at the root process. For this we again had two differnt concepts.

The first concept was that every single process would coarsen its local array into a uvis of with global visres at the corresponding offset. Afterwards the processes would simply sum their local uvis to a global array at the root process. This approach promised an exact result and a fairly easy implementation due to the minor modification to the original code. Since uvis was supposed to fit into memory anyway it seemed adequate to compute at full resolution at every process. This approach was ommitted due the failure to find a bug causing errors in the right bottom image corner.

The second approach was to coarsen the individual arrays to arrays of a local size proportional to the tilesize. Then the processes gather their local uvis to the root process which prints everything to the image file. In order to do this correctly, a subarray type needed to be created. At this point, we again faced the problem with a arraysize that is not divisible by the number of processes, an we did that at two different spots: For uvis, we ust decided that we would increase the resolution so far, that the length and broadth, respectively, are divisible by the number of processes. This is in our eyes acceptable, because this does just increase the resolution at most by the number of processes in this dimension minus -1.

Second, we had to deal with our padded arrays. We again used the assumption that you only would use MPI with very high resolutions, so we decided to use coarsen to stretch padded arrays in a way, that their coarsened version would only use their real gridpoints, but coarse them to the same size as non-padded arrays. This is very problematic for very small arraysizes, but as already

mentioned, you would generally not use an MPI-parallelization for those. For reasonable resolutions, this yields pretty good results.

## Current State

Right now, our program does parse the data as it should, initialize the local array (incl heatsrc), transfer the ghost cells between neighbors and calculate the residua. After that the grids are coarsened and gathered to the root process before being printed. After that we started to work on an MPI-OMP Hybrid solution, when the clock reached 9am on June $26^{th}$ and we were surprised by supoermuc going into maintenance mode. Thus, at this time, we can only provide speedup for square topologies.

## Speedup

To be completed when supermuc is available again.

| Measurement | Floprate | Speedup |
|---|---:|---|
| Sequential (Baseline) | 3,896.2 MFlop/s | 1 |
| Auto parallelization 8 threads | 13,208.1 MFlop/s | 3.39 |
| OMP 16 Threads | 22,247.3 MFlop/s | 5.71 |
| MPI 64 processes, square, blocking | 74,064.5 MFlop/s | 19.01 |
| MPI 64 processes, line, blocking | 76,077.4 MFlop/s | 19.52 |
| MPI 64 processes, col, blocking | 72,963.2 MFlop/s | 18.73 |
| MPI 64 processes, square, nonblocking | 74,478.7 MFlop/s | 19.12 |
| MPI 64 processes, line, nonblocking | 80,607.0 MFlop/s | 20.69 |
| MPI 64 processes, column, nonblocking | 71,746.0 MFlop/s | 18.41 |

In this context, line means that processes only communicate lines, while col means that processes only communicate columns. Earlier, we had a Floprate of nearly 80 GFlop/s on square topologies, but after we fixed the issues regarding non-square topologies, the performance got worse. However, for smaller grids (and for cases where only lines are transferred), we still achieve this speedup.