UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Excercise Sheet 8

(To be processed until 28.01.)

Lecture Parallel Processing Winter Term 2024/25

Exercise 1: Parallelization of the Jacobi method using MPI (Compulsory Exercise, Weight 3! Submit until Tuesday, January 28th, 10:00 via moodle)

The sequential code in the files `heat.cpp` and `solver-jacobi.cpp` (in the archive `u08eFiles.zip`[1] on the lecture's web page) shall be parallelized using MPI step by step (the code is identical to that of Exercise 1 on Exercise Sheet 4).

a) Parallelize only the `main()` function and the `solver()` function. For now, please ignore the file output of the matrix (function `Write_Matrix()`), i.e., comment out the call in `main()`. However, make sure that the control values are output correctly at the end of `main()`. In the case of correct parallelization, the output values must correspond exactly to those of the sequential version!

You can distribute the matrix in one dimension only (strip-wise, i.e., contiguous blocks of rows, see Sect. 4.9 of the lecture slides) or in both dimensions (see last slide of Sect. 4.1 of the lecture). The strip-wise distribution is (much) simpler, but the block-wise one (with larger process numbers) possibly more efficiently. Ideally, your partitioning should work for all matrix sizes and process numbers (see Sect. 4.9 of the lecture slides and the example code `vecmult3.cpp`[2]).

b) Extend your program such that after the computation, the matrix is written correctly to the file `Matrix.txt` using `Write_Matrix()`. Since you will not have a parallel file system, do not use any of the MPI I/O routines, but rather let each process send its part of the matrix to process 0, which appends it to the file. Since the matrix is large, avoid storing the **complete** matrix in process 0!

Note that you may have to extend the interfaces of the functions `solver()` and `Write_Matrix()` with additional parameters.

Measure how much time your program needs. Try different values for the matrix size (reference values: 500, 2000 and 6000) and measure the speedup with different (2 to 16, possibly even more) processes.

If necessary, do a more detailed performance analysis (if possible, using Scalasca) and try to optimize your program as much as possible, e.g., by using non-blocking receive operations.

Exercise 2: Parallelization of the Gauss/Seidel method using MPI (For Motivated Students)

In this exercise, you shall parallelize the code for the Gauss/Seidel method, as provided in the file `solver-gauss.cpp` (in the archive `u08eFiles.zip`[3] on the lecture's web page) using MPI (the code is identical to that of Exercise 3 on Exercise Sheet 4).

For simplicity, the function `solver()` in this version performs a fixed number of iterations, calculated in advance from the precision parameter. This allows pipelined parallelization (where in contrast to the OpenMP parallelization using diagonal traversal, the `i` and `j` loops are not rewritten, see Sect. 4.9 of the lecture slides). For example, process 0 sends its last row to process 1 after each iteration, and then waits for the first row of process 1. Process 1 can (and must) send this row immediately after its calculation.

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u08eFiles.zip`
[2] `https://www.bs.informatik.uni-siegen.de/web/wismueller/vl/gen/pv/03Code.zip`
[3] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u08eFiles.zip`

Before you start programming, first consider **exactly** which communications are necessary and how the sequence of the calculations and communications should look like! Also note that you may need to add additional parameters to the interface of the `solver()` function.

Measure how much time your program needs. Try different values for the matrix size (reference values: 500, 2000 and 6000) and measure the speedup with different (2 to 16, possibly even more) processes.

If necessary, do a more detailed performance analysis (if possible, using Scalasca) and try to optimize your program as much as possible, e.g., by using non-blocking receive operations.

Exercise 3: Acquaint yourself with the electronic exam system

The exam for this module will be conducted **electronically** using the Q-Exam system. The **computers (laptops) will be provided by the university**.

To get acquainted with the system, please **have a look at the provided demo exam**[4] (select the exam named "Demo-Prüfung Parallelverarbeitung") and the screen casts provided in the moodle course[5]! If you have questions or problems, please contact the e-assessment support team (e-klausuren@uni-siegen.de). You can also ask questions immediately before the exam starts.

Please notice that the laptops provide a **German keyboard**. The '#' key is located left of the 'Enter' key, '{', '}', '[', and ']' can be entered by pressing the 'AltGr' key (right of the 'Space' key) together with '7', '8', '9', or '0'. The 'Ctrl' key is named 'Strg'. Please acquaint yoursef with that layout! An accurate image can be found on wikipedia[6]. There is also an interactive simulation[7], which, however, differes slightly from the laptop keyboards.

[4] `https://uni-siegen.q-examiner.com/`
[5] `https://moodle.uni-siegen.de/course/view.php?id=23366#section-6`
[6] `https://en.wikipedia.org/wiki/German_keyboard_layout`
[7] `https://www.branah.com/german`