UNIVERSITÄT SIEGEN

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

Exercise Sheet 4

(Deadline: 30.01.2024)

Parallel Processing
Winter Term 2023/24

Preparation: Download the program codes for this exercise sheet from the Internet:
`https://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u4Files.zip`

Exercise 1: Point to point communication with MPI

Modify the code in `point2point.cpp`, such that at the indicated position

- Process 1 sends the contents of variable `myval` to process 3, that receives it in `myval`,
- Process 2 sends the contents of variable `myval` to process 0, that receives it in `myval`.

Thus, with the given initialization, the value printed by process 3 at the end of the program should be 1, the value printed by process 0 should be 2.

Note that this exercise checks for MPI skills in a way comparable to an exercise in the exam!

Exercise 2: Parallelization of a map operation using MPI

Parallelize the code in `map.cpp`! Note that only process 0 should initialize the array `x`, and only process 0 should check the result at the end. The application of the function `complex_fct()` on the input array `x` should be distributed to all processes. Note that `complex_fct()` is a pure function that does not have any side effects.

For simplicity, you can assume that the array size can be evenly divided by the number of processes (with the given size of 10000 elements, this is the case for 1, 2, 4, 8, and 16 processes). Do not modify any code in `fcts.cpp`!

Note that this exercise checks for MPI skills in a way comparable to an exercise in the exam!

Exercise 3 (Compulsory Exercise for 6 LP): Parallelization of a simple optimization code with MPI

In this exercise, we consider the optimization code of exercise sheet 2 again.

a) Parallelize the code in `optimize1.cpp`. Your program should always use four MPI processes, where each process computes one configuration. At the end, you should compute the final minimum using `MPI_Reduce()`.

b) Parallelize the code in `optimize2.cpp`. Since the execution time of `computeCost()` shows a high variation, use the manager/worker model (see Section 1.8.2 of the lecture slides).

Process 0 should be the manager, that sends tasks descriptions to the workers and receives their result. In this code, the task description simply is the number of the configuration to be evaluated (i.e., the argument passed to `computeCost()`). Once a result has been received from a worker, the next task description is sent to that worker. In case the manager is running out of tasks (i.e., all 100 tasks have already been sent), this task description should contain a special value (e.g., a number ≥ 100) that instructs the worker to terminate.

In both files `optimize1.cpp` and `optimize2.cpp`, the code to initialize MPI is already given. Do not modify any code in `functions.cpp`!

Exercise 4: Numerical integration using MPI

Parallelize the code in `integrate.cpp` (identical to that of exercise sheet 2) with MPI using a reduction (`MPI_Reduce()`). The initialization of MPI is already given. Measure the speedup with different values for the number of intervals and different numbers of processes. Interpret your results.

Exercise 5 (Compulsory Exercise for 5 CP and 6 CP): Parallelization of the Jacobi method using MPI

The sequential code in the files `heat.cpp` and `solver-jacobi.cpp` shall be parallelized using MPI step by step.

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 section 5.5 of the lecture slides) or in both dimensions (see last slide of chapter 3.1 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 section 5.5 of the lecture slides and the example code `vecmult3.cpp`[1]).

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

In this exercise, you shall parallelize the sequential code for the Gauss/Seidel method, as provided in the file `solver-gauss.cpp` using MPI.

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 section 5.5 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.

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.

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

Exercise 7: 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**[2] (select the exam named "Demo-Prüfung Parallelverarbeitung") and the screen casts provided in the moodle course[3]! 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[4]. There is also an interactive simulation[5], which, however, differes slightly from the laptop keyboards.

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