UNIVERSITÄT SIEGEN

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

Exercise Sheet 4

(Deadline: 01.02.2018)

Parallel Processing
Winter Term 2017/18

Preparation: Download the program codes for this exercise sheet from the Internet:
`http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/ws1718/pv/u4Files.zip`

Exercise 1: Numerical integration using MPI

Parallelize the code in `integrate.cpp` (identical to 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.

Note: The MPICH2 (Hydra) process manager, which is the default in the lab room H-A 4111, does not forward the standard input correctly! To start your program, use the command `mpiexec.gforker`. However, it will start all processes on the local computer. Thanks to the multicore CPUs, you will still get a speedup.

Exercise 2: 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).

b) Extend your program so that after the computation, the matrix is written correctly to the file `Matrix.txt` using `Write_Matrix()`. Avoid storing the **complete** matrix on one node!

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, perform a more accurate performance analysis with Scalasca and/or Jumpshot and try to optimize your program as much as possible, e.g., by using non-blocking receive operations.

Exercise 3 (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 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, perform a more accurate performance analysis with Scalasca and/or Jumpshot and try to optimize your program as much as possible, e.g., by using non-blocking receive operations.