UNIVERSITÄT SIEGEN

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

Excercise Sheet 4

(To be processed until 03.12.)

Lecture Parallel Processing
Winter Term 2024/25

Exercise 1: Parallelization of the Jacobi method using OpenMP (Compulsory Exercise, Weight 2! Submit until Tuesday, December 03rd, 10:00 via moodle)

In this exercise we consider the Jacobi method for the iterative solution of a boundary value problem. In Jacobi iteration, a new matrix of values is computed from a given one by assigning to each inner element of the new matrix the mean value of the four neighboring elements of the old matrix. The iteration is repeated until the maximum change of an element is below a predefined limit.

Have a look at the code in the archive `u04eFiles.zip`[1] on the lecture's web page. Parallelize the sequential code for the Jacobi iteration in the file `solver-jacobi.cpp` using OpenMP directives. Use an OpenMP reduction to compute the maximum change in the array (variable `diff`).[2]

Test the program with different matrix sizes and thread counts. At the end, the program prints the values of several matrix elements. Note that in your parallel version these results must be **exactly** the same (that is, in **all** decimal places) as in the sequential program!

For further verification, you can also view the output (in the file `Matrix.txt`) graphically, using the Java program `ViewMatrix`. This output is, however, only created with a matrix size up to 1000.

Conduct a performance analysis (if possible, using Scalasca) and try to optimize your program as far as possible, e.g., by eliminating barriers. Determine the achievable speedup for different numbers of threads.

Exercise 2: Parallelization of the Gauss/Seidel method using OpenMP

In this exercise, the Gauss/Seidel method is to be parallelized. In contrast to the Jacobi method, the Gauss/Seidel method uses only one matrix, in which each element is replaced by the mean value of its neighbor elements. The value of the left and the upper neighbor already originates from the current iteration, so that data dependences result.

In this exercise you should parallelize the method by restructuring the loops, such that the matrix is traversed diagonally (see Sect. 3.3 of the lecture slides).

Have a look at the code in the archive `u04eFiles.zip`[3] on the lecture's web page. Parallelize the sequential code of the Gauss/Seidel relaxation in the file `solver-gauss.cpp` using OpenMP directives. The code given for this exercise differs from Exercise 1 only by the different implementation of the function `solver()`. For simplicity, this solver does not iterate until the maximum change is below a threshold, but instead estimates the necessary number of iterations in advance.

For the parallelization, also pay attention to the notes in Exercise 1.

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u04eFiles.zip`

[2] Reductions of this form are directly supported by OpenMP only as of version 4.0. Therefore, if necessary, use the method shown in the lecture slides at the end of Sect. 3.4.3.

[3] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u04eFiles.zip`

Exercise 3: Pipelined parallelization of the Gauss/Seidel method using OpenMP (For Motivated Students)

If the number of iterations is known in advance, the Gauss/Seidel method can also be parallelized in a pipeline-like manner (see Sect. 3.3 of the lecture slides). For the parallel execution of the `i` loop, a synchronization must be provided which ensures the following conditions (induced by the data dependences): before the `i`-th row is computed in iteration `k`,

1. the (`i-1`)-th row in iteration `k` and
2. the (`i+1`)-th row in iteration `k-1`

must have been computed. Since OpenMP version 4.5, the `ordered` directive supports a `depend` clause which enables to specify a synchronisation safeguarding the specified dependencies (see the note at the end of Section 3.4.4 of the lecture slides).

Parallelize the Gauss/Seidel method (see the code in the archive `u04eFiles.zip`[4] on the lecture's web page) using the above synchronization. Compare the achievable speedup with the parallelization using a diagonal traversal of the matrix from Exercise 2. Conduct a performance analysis (if possible, using Scalasca) and try to optimize your program further.

[4] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u04eFiles.zip`