UNIVERSITÄT SIEGEN

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

Exercise Sheet 2

(Deadline: 17.12.2022)

Parallel Processing
Winter Term 2021/22

Preparation: Download the program codes for this exercise sheet from the Internet:
`https://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u2Files.zip`

Exercise 1: Loop parallelization

In `loops.cpp`, you'll find some loops that process arrays. Analyze these loops and decide whether they can be parallelized (without synchronization). If necessary, you can also reorganize the loops to eliminate dependences.

Test your analysis by parallelizing the appropriate loops using the OpenMP directive `parallel for` (do not use any other directives). If you compile the program (using `make`) and run it, the code in `main.cpp` checks if your parallel code still calculates exactly the same result, and prints an error message if not. The code in `main.cpp` must not be changed!

Exercise 2 (Compulsory Exercise for 6 LP): Parallelization of a numerical integration using OpenMP

The code in `integrate.cpp` calculates the integral of a given function $f(x)$ between 0 and 1 using the center rule. The function $f(x)$ computes the expression $4/(1 + x^2)$, so the integral has exactly the value π. The file contains the integration code four times: one version (implemented in the function `Serial_Integration()` should remain sequential for comparison, the other three (in the functions `Parallel_Integration1()` to `Parallel_Integration3()`) you should parallelize as follows:

1. using an OpenMP reduction,
2. using the OpenMP `ordered` directive,
3. by splitting the loop into a parallel and a sequential part.

The `main()` routine calls all four functions and prints their result, the error, the runtime, and the speedup, if applicable. The `makefile` contains the necessary commands to compile and link the program (you only have to invoke the command `make`).

Measure how much time each of your functions requires. Try different values for the number of threads and intervals (recommended starting value: 100,000) and compare the times for the sequential and parallel implementations. Interpret your results.

In the second variant (with `ordered` directive), consider how you can improve the runtime by a suitable scheduling of the loop!

The first parallel version (with the OpenMP reduction) does not compute exactly the same result as the sequential code. Why not? For versions 2 and 3, make sure that the result matches exactly with the sequential version!

Exercise 3 (Compulsory Exercise for 5 LP and 6 LP): Parallelization of the Jacobi method using OpenMP

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.

Parallelize the sequential code for the Jacobi iteration in the file `solver-jacobi.cpp` using OpenMP directives. The required computation of the maximum change is a reduction, which in this form is directly supported by OpenMP only as of version 4.0. Therefore, use the method shown in the lecture slides at the end of Section 2.6 ("Synchronization") for the parallel computation.

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 4 (Compulsory Exercise for 5 LP and 6 LP): 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 Section 2.5 of the lecture slides).

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 3 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 3.

Exercise 5 (For Motivated Students): Pipelined parallelization of the Gauss/Seidel method using OpenMP

If the number of iterations is known in advance, the Gauss/Seidel method can also be parallelized in a pipeline-like manner (see Section 2.5 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. Because OpenMP does not provide "point-to-point" synchronization, e.g., via condition variables, a self-implemented synchronization construct must be used. This is given as class `Cond` in the file `cond.h`:

- `Cond(int n)`: Constructor. The parameter `n` specifies the matrix size of the Gauss/Seidel method.
- `void signal(int k, int i)`: Signals that the computation of row `i` in iteration `k` has been completed.
- `void wait(int k, int i)`: Waits until the computation of row `i` in iteration `k` has been completed (using busy waiting).

Parallelize the Gauss/Seidel method using the above synchronization. Compare the achievable speedup with the parallelization from Exercise 4. Conduct a performance analysis (if possible, using Scalasca) and try to optimize your program further.