UNIVERSITÄT SIEGEN

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

Excercise Sheet 3

(To be processed until 26.11.)

Lecture Parallel Processing
Winter Term 2024/25

Exercise 1: Dependence analysis

Consider the following program fragment:

```
double b[10], a[10], c[10];
for (int i=1; i<10; i++) {
  a[i] = (b[i-1] + b[i]) / 2; // S1
  b[i] = a[7] * c[i];         // S2
  c[i-1] = 2 * b[i];          // S3
}
```

Which of the following statements are correct and which are incorrect?

a) There is a **true** dependence between **different** loop iterations from S1 to S2.
b) There is an **anti** dependence within a **single** loop iteration from S1 to S2.
c) There is a **true** dependence within a **single** loop iteration from S2 to S1.
d) There is a **true** dependence between **different** loop iterations from S2 to S1.
e) There is an **anti** dependence between **different** loop iterations from S2 to S1.
f) There is a dependence from S1 to S3.
g) There is an **anti** dependence between **different** loop iterations from S2 to S3.
h) There is an **output** dependence within a **single** loop iteration from S2 to S3.
i) There is a **true** dependence between **different** loop iterations from S3 to S2.
j) The loop does not contain any **output** dependencies.

***Note**: this is a question similar to what you can expect in the exam.*
***Hint**: six statements are correct, four are incorrect.*

Exercise 2: Loop parallelization

Look at the code in the archive `u03eFiles.zip`[1] on the lecture's web page. 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 3: Parallelization of a simple optimization code with OpenMP (Compulsory Exercise! Submit until Tuesday, November 26th, 10:00 via moodle)

In this exercise, you have to parallelize a simple optimization code. We assume that there is a task (e.g., assembling a car) that can be solved in several different ways (configurations). For each configuration, we compute the cost (e.g., by

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u03eFiles.zip`

simulating the assembly process), and then calculate the minimum cost of all examined configurations. For simplicity, the code does not determine which configuration actually leads to the minimal cost.

Obviously, the computation of the cost can be carried out independently (and thus, concurrently) for each configuration, so the problem is easy to parallelize.

Have a look at the code in the archive `u03eFiles.zip`[2] on the lecture's web page. In `optimize.cpp`, a loop is used to find the minimum cost for 100 different configurations. Parallelize this loop and determine the speed-up.

Note that the execution time of the function `computeCost()` varies for different configurations (i.e., it depends on the function's parameter), so use a suitable strategy for scheduling the loop iterations to threads (see Sect. 3.2.1 of the lecture).

Do not modify any code in the file `functions.cpp`!

Exercise 4: Parallelization of a numerical integration using OpenMP (Compulsory Exercise! Submit until Tuesday, November 26th, 10:00 via moodle)

Have a look at the code in the archive `u03eFiles.zip`[3] on the lecture's web page. 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!

[2] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u03eFiles.zip`
[3] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u03eFiles.zip`