UNIVERSITÄT SIEGEN

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

Exercise Sheet 1

(Deadline: 03.12.2020)

Parallel Processing
Winter Term 2020/21

Preparation: Download the program codes for this exercise sheet from the Internet:
`https://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/u1Files.zip`.
In the directories `Exercise1` to `Exercise3` you will find the given code for the respective exercises.

Exercise 1: Data Dependences, Programming with C++ Threads

Look at the code in the file `compute.cpp`. First, analyse the data dependences between the different function calls. The functions `f1` to `f7` do not have any side effects, thus they, e.g., do not access global variables.

Based on the data dependences, determine which calls of the functions `f1` to `f7` can be executed in parallel. Try to remove the occurring anti dependences and output dependences (if any) by renaming one or several variables.

Then, draw a task graph (see section 1.8.7 of the lecture) representing the dependences between the different tasks, where in this case a task corresponds to a function call. In the example, all functions have an execution time of $1s$. What is the execution time of the parallel program?

Parallelize the program using C++ Threads. If neccessary, implement a proper synchronization using condition variables. Pay attention to ensure that the printed result is *exactly* the same as with the sequential program.

Exercise 2: Run-Time Analysis of a Sequential Program

Before a program (which typically is not known in all its details) is parallelized, it is common practice to use performance analysis tools in order to identify the most compute intensive parts of the program code.

A simple, sampling-based tool for this purpose is the program `gprof`, which measures for each procedure (or method) P in the program how often it is called and how much time is required to process P. In doing so, the tool differentiates between

- the *inclusive time* of P, which also includes the processing time of the procedures called by P (*children*), and
- the *exclusive time* (or *self time*) of P, which does not include the processing time of these calls.

When, e.g., the procedure

```
void funcA() {
   for (int i=0; i<N; i++) { ... }
   funcB();
}
```

is invoked, the *exclusive time* of `funcA` is only the runtime of the `for` loop, while the *inclusive time* also includes the runtime of `funcB`.

In this exercise, you should familiarize yourself with `gprof`. First, compile the program code in `example.cpp`[1] with instrumentation for `gprof`:

```
g++ -pg -g -o example example.cpp
```

Start the program with `./example`. During the execution, the gathered profiling data is written to a file `gmon.out`. A readable form of this data is then obtained using the command `gprof ./example`. Thoroughly look at the output, in particular the included explanations, and answer the following questions:

a) Which function requires the most computing time (*exclusive*) and would therefore be the first candidate for parallelization?

[1] By the way, this code is a nice example of *obfuscation*, to prevent you from inspecting the (rather short) source code.

(i) Which speedup and which efficiency would you expect in the best case, if only this function is parallelized?

(ii) How often is the function called?

(iii) What other functions does the function call?

b) Which function is most frequently called and from which other functions?

c) How is the computation time of the function `func5` composed?

Exercise 3 (Main Exercise): Parallelization of Quicksort using C++ Threads

In this exercise, the Quicksort algorithm is to be parallelized. The basic idea of Quicksort is as follows:

- From a given array, an element is selected as a pivot or a reference element, e.g., the element from the center.
- A larger element is sought from below and a smaller one from above.
- If such items are found, they are placed incorrectly and are swapped.
- This search and swap will continue until 'below' and 'above' meet. After that, there are only smaller (or equal) elements below, and only larger (or equal) elements above.
- These two areas must now be sorted in turn, which translates to two recursive calls to quicksort.

The procedure

```
void quicksort(int *a, int lo, int hi)
```

in `qsort.cpp` sorts the portion of the array starting at index value `lo` up to and including index value `hi` in ascending order. In addition, some auxiliary functions are implemented:

- `void initialize(int *a, int n)` initializes an array `a` of length `n` with random numbers.
- `int checkSorted(int *a, int n)` checks if an array `a` of length `n` is sorted.
- `void printArray(int *a, int n)` can be used as necessary to print a (small!) array.

a) Compile the program and run it. Test the code for different array lengths (recommended starting value: 10,000,000 elements). Record the time that the program requires for sorting the array in each case.

b) Rewrite the program that one of the two recursive invocations of `quicksort` will execute in a new thread, but only if the array length for this invocation is greater than a constant `MINSIZE`. Try different values for `MINSIZE` (recommended starting value: 1,000,000).

Compare the performance of your implementation with the previous measurements, using a table. Calculate the speedup and interpret your results.

For motivated students: The C library offers a function `qsort` (documentation: `man qsort`):

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *))
```

Compare the performance of your implementation against the function `qsort` in the C library.

Links

- C++ threads tutorial:
  https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial