Parallel computing Exercise 4

2.2.2018

Submit the X1 task by Fri 2.2. 09:00 using a separate www-form as instructed in a separate email. Submit each of the other solutions separately to Moodle by 2.2. 09:00 (1 hour before exercises). Take skeletons from course www-page.

16. Parallelize the standard Quicksort using OpenMP. It is enough to parallelize the two recursive calls. You can either use `#sections` or make a parallel loop of two iterations.
17. Make the duplicates finding (task 12) faster by first parallel sorting the input (task 16, and then finding all the duplicates much faster as they are in consecutive locations.
18. Refine the prefix-sum algorithm (handouts p. 38, slides p. 126) to a work optimal one by using blocking. Try to make indexing work for any input size. We'll implement this in OpenMP next week.

The following X1 exercise replaces standard exercises 19 and 20.

The answers to X-exercises have to be unique for every student. No copies of the same answer are allowed. The answer has to be sent via email by Friday 09:00. You will receive an acknowledgment upon successful processing. Answers will be graded. The answer must also contain a short self-evaluation in which you describe whether the program works, nearly works, or does not probably work; how efficient it is, what speedups you got, etc. A correct and proper self-evaluation is worth one point (in case of a proper answer). As the answer is a C program, the self-evaluation must be included in the comments of the program.

Send your solution using a www-form, address and credentials of which you got by email. The solution should be a compilable C source code file If you do not receive an acknowledgment, or you receive compilation errors in the acknowledgement, send it again correctly.

Take a skeleton from course www-page. Do not change the header (name, parameters) of the X-task method(s). Please make sure that the program is compilable as such, i.e., have the whole answer in the same file.

X1. Write a parallel OpenMP program to find prime numbers. The input is a pair of `long` integers A and B and the result is all primes of the interval $A..B$. The parallel function finds the primes and stores them in ascending order into an array. Take the sequential program from the www-page. For max 4 points, it is sufficient to parallelize the *findPrimes*() function. For full 6 points, you need to parallelize also the helping function *smallPrimes*() (even if the *smallPrimes*() is quite fast for small n).