INF236 PARALLEL ALGORITHMS - ASSIGNMENT 3 2014

General rules

- The assignment is due on April 30 at 6 p.m.
- Your answer is to be uploaded at the course page at MiSide.
- Your answer should be a zip-file containing program files and a brief and concise report on how the problem was solved.
- Grading system: 0–100 points. Passing the assignment with at least 40 points is a prerequisite for admission to the exam. The total score on all three compulsory assignments counts 50% of the final grade.

Problem

a) Write an efficient sequential program that uses *Radixsort* to sort an array of integers (4 bytes = 32 bit `int`). The program should operate on the binary representation of the numbers. It should take two input parameters, the number of elements to be sorted (n) and the number of bits (b) that should be interpreted as one digit. It is sufficient that the program works when b divides 32. If, for instance, $b = 4$, *Radixsort* needs 8 iterations, and will put array entries in $2^4 = 16$ different buckets in each iteration. Your implementation should first count the number of elements in each bucket before it calculates where each bucket starts. Next, you put each element in its proper position. The program should output the time used to do the sorting (excluding the time used for e.g. allocation of memory). Your report should explain briefly how the algorithm has been implemented.

b) Make experiments with the program from question a), where you vary n and b to determine how large arrays you can sort in about 10 seconds. Your experiments should also include timings where you try all values of $b = 1, 2, 4, 8, 16$ for this maximal n-value. Document and discuss your results briefly and concisely. Give the the theoretical running time of the algorithm, and analyze whether the observed running times agree with the theoretical one.

c) Develop an efficient parallel version of your code using OpenMP. Your report should briefly and concisely explain how the algorithm has been implemented. It should be possible to change the number of threads without having to recompile the program, and therefore this number should not be hard-coded in the program.

d) Recall that in *strong scaling* experiments, we keep the instance size fixed, and investigate how much the running time decreases when the number of threads increases. Perfect strong scaling means that the running time is proprtional to the inverse of the number of threads. In *weak scaling* experiments, we study the running time as a function of the number of threads when the *ratio between instance size and number of threads* is fixed. Perfect weak scaling means that this function is a constant.

Make strong and weak scaling experiments with your code. For the strong scaling experiments, you should use the maximal value of n found in question b). Compute and plot the speedup of the program compared to the sequential program developed in question a), and also to the program itself when run on a single thread. Also, compare the running time of your algorithm with some parallel implementation of *Mergesort* (see Exercise 6).

Make the weak scaling experiments with n varying up to the largest value that your program can handle. Plot the results and comment briefly.