Compulsory Assignment 1 in INF236
Spring semester 2024

Due date: Your solution should be bundled as zip file and handed in on mitt.uib.no no later than midnight Sunday 3rd of March.

Your solution should contain one nicely formated pdf report containing all necessary text and figures. This should also list the names and purpose of your program files. Each program should be contained in one separate file. It is advisable to display results using both tables and graphs.

In this assignment you are going to implement and test a parallel algorithm for radix sort using numbers of the type unsigned long long. Use the genrand64_int64() function in the file mt19937-64.c to generate your data. You get access to this routine by storing the file mt19937-64.c in the same directory as your program and adding a line #include "mt19937-64.c" at the top of your program. It is possible, but not necessary, to seed the random generator so that you generate different data for each run.

You should neither include initial time spent on memory allocation nor time spent on generating data in your experiments.

All algorithms should be implemented in C or C++ and parallelized using OpenMP. The final running of the programs should be done on brake.ii.uib.no. Compile using the -O3 -fopenmp flags (and -lm if you need math routines). Do not use more than 80 threads for any run. Perform each run at least three times and only use the best timing.

Individual work: You may talk and discuss the assignment with you fellow students, but you *must* do the programming yourself. If you copy code from someone both you and the person you copy from will automatically fail the course. The same is also true if you copy from the Internet.

Problem 1

Write an efficient sequential program that uses radix-sort to sort an array of integers (of type unsigned long long, 64 bits). 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 if the program works when b is a power of 2. The program should verify that the result is sorted. The program should output the time used to perform the sorting not including the time used for allocation of memory and initialization.

To avoid dynamic memory allocation you should use one array of size n to hold all the buckets. Thus, for each digit first count the number of elements in each bucket before calculating where each bucket starts. Following this you can place each element in its appropriate position.

Your report should explain how the algorithm has been implemented.

Problem 2

Perform and document experiments using the program from Problem 1 where you vary n and b to determine the largest amount of numbers you can sort in about 10 seconds. Your experiments should also include timings where you try all possible values of b for this maximal n value. Discuss and explain your results.

Create an expression that is as exact as possible for the running time for the algorithm. Your expression should be a function of n and b and should also include constants. To do this you might first want to consider the theoretical running time of the algorithm. To measure the constants you could perform timed experiments of the individual parts of the code.

How well does your expression compare with the actual measurements?

Problem 3

Develop an efficient parallel version of your code using OpenMP. Document the design choices you make. When faced with design choices you should base your decisions on timed experiments.

Your report should explain how the algorithm has been implemented. It should be possible to change the number of threads without having to recompile the program, thus this number should not be hard coded into the program.

Problem 4

Perform both strong and weak scaling experiments on your code. For the strong scaling experiments you should use the maximal value of n found in Problem 2. Compute and plot the speedup of the program compared to the sequential program developed in Problem 1 and also to the program itself when

run on one thread.

Based on your experiments, create an as accurate as possible expression for the parallel running time (as in Problem 2) and comment on how well this fits with the actual running times.

Perform the weak scaling experiments up to the largest possible value of n that your program can handle.