Compulsory Assignment 1 in INF236
Spring semester 2026

Due date: Your solution should be bundled as zip file (no other format is allowed) and handed in on mitt.uib.no no later than midnight March Sunday 8.

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. Your zip file should in total consist of three files in the same directory: a pdf-file containing the report, and one program file each for problems 1 and 4. Do not include any other files!

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. It is advisable to display timing results using both tables and graphs.

All algorithms should be implemented in C or C++ and parallelized using OpenMP. The final running of the programs must be done on the university computer.

Compile using the -O3 -fopenmp flags (and -lm if you need math routines). Do not use more than 32 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. This is strictly enforced!

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 final 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 or for checking the final result.

You should use bitwise operations to get access to the current "digit" of a particular number. In particular, `a << b` and `a >> b` will shift the content of `a` respectively `b` places to the left and `b` places to the right. The command `a & b` will perform a bitwise AND operation between `a` and `b`. Thus the commands

```
x = ((unsigned long long) 1 << b) - 1;  // Calculate 2^b - 1
y = a[i] & x;       // Do a bitwise AND operation with a[i] and x
```

will give you the rightmost `b` bits of `a[i]` stored as an integer in `y`. To get the next `b` bits you can execute

```
y = (a[i] >> b) & x;  // Shift a[i] b bits to the right before AND
```

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 the position where each bucket starts. Following this you can place each element in its appropriate position.

Explain how the algorithm has been implemented together with pseudo-code. Also, give the running time in O-notation as a function of n and b.

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. Note that you might not be able to run your program for very large values of b.

For the final n and b values you should also report the accumulated time you spend in the different stages of the algorithm. This includes

1. Counting the number of elements in each bucket.
2. Performing the prefix sum to determine the position of each bucket.
3. Placing the elements in their correct bucket.

Discuss and explain your results.

Problem 3

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

Your report should explain how the algorithm has been implemented together with pseudo-code. Also, derive the running time in O-notation as a function of n, b and p.

Problem 4

Perform both strong and weak scaling experiments on your code using up to 32 threads.

For the strong scaling experiments 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 compared to the program itself when run on one thread. Also report and comment on how the different stages of the algorithm scales when changing the number of threads.

For the weak scaling experiments start from $p = 1$, and with n and b set to the numbers used in the strong scaling. Then increase both the number of threads and the size of the problem while attempting to keep the amount of work for each thread constant. Plot the number of threads versus the time spent.

Discuss and explain your results.