1 Introduction

The purpose of this assignment is to practice parallel programming in the distributed memory paradigm using the Message Passing Interface (MPI). In order to do so, you will implement a given sorting algorithm along with code to distribute the input vector, collect the output vector, and verify the correctness of the parallel sorting algorithm.

```c
//
// Sorts input[] into output[] in ascending order.
//
void count_sort ( int n, int input[], int output[] )
{
    // For each element in the input
    for( int a = 0; a < n; ++a ) {
        // Count the number of preceding elements in the output
        int count = 0;
        for( int b = 0; b < n; ++b ) {
            if( input[ b ] < input[ a ] ) {
                ++count;
            } else if( input[ b ] == input[ a ] ) {
                if( b < a ) {
                    ++count;
                }
            }
        }
        // Place the element in its correct position in the output
        output[ count ] = input[ a ];
    }
}
```

Figure 1: The sequential count sort algorithm implemented in C.

The topic is sorting or more precisely an (inefficient) sorting algorithm that we will call count sort (not to be confused the efficient algorithm that goes by a similar name). A C implementation of the sequential count sort algorithm is presented in Figure 1. The basic idea of count sort is straightforward: For each input element, count the number of other inputs elements that will precede it in the output. For simplicity, let us from now on assume that the input is an array of int:s of length n > 0 and that we are sorting these numbers in ascending (increasing) order.

To illustrate the count sort algorithm, as implemented in Figure 1, let us consider the steps of the algorithm when given the following input array of length 10:

```c
int input[] = {3, 8, 5, 6, 3, 5, 2, 8, 1, 4};
```

Each iteration of the outer a-loop produces a value for the count variable and writes one element of the output array, as specified by the following table.
Parallelization for distributed memory architectures

Count sort can be parallelized for distributed memory architectures in more than one way. In this assignment, we will use one of the simpler approaches that doesn’t require duplication of the entire input array. For simplicity, we assume that the number of processes, $p$, evenly divides the length of the input, $n$. We assume that the input is distributed using a block distribution with block size $b = n/p$. That is, process $q \in \{0, 1, \ldots, p-1\}$ stores the sub-array $\text{input}[q*b:q*b+b-1]$ of length $b$.

Consider the steps of the algorithm in Figure 1 that can be performed locally on process $q$ without communicating with the other processors. Since process $q$ holds only a sub-array of $\text{input}$ and each iteration of the inner $b$-loop requires access to the entire $\text{input}$ array, it is immediately clear that process $q$ cannot complete any iteration of the outer $a$-loop on its own. However, it can perform some of the inner iterations for some of the outer iterations.

More specifically, it can perform the inner iterations $b = q*b, q*b+1, \ldots, q*b+b-1$ for the outer iteration $a = q*b, q*b+1, \ldots, q*b+b-1$.

After these purely local steps, each process has performed $b$ partial outer iterations. Let us make the reasonable decision that each process should be responsible to finish the $b$ outer iterations it has started. To proceed further, each process needs another chunk of $b$ elements from the $\text{input}$ array. Since the $\text{input}$ array is distributed using a (uniform) block distribution, we can shift the distribution one block in either direction. After this communication step, we can again perform $b$ inner iterations for each of the $b$ outer iterations assigned to each process. Repeatedly shifting and performing inner iterations eventually completes the outer iterations.

At this point, the $\text{count}$ is known for each input element, and what remains is to effect the permutation from the $\text{input}$ array to the $\text{output}$ array using the $\text{count}$s. In general, each process needs to scatter its elements of $\text{input}$ to all other processes. Each process always receives exactly $b$ elements for its portion of $\text{output}$, but the number of elements that a process sends out is in general different for each destination. One of the main difficulties is to figure out an effective way of applying the permutation of elements. The simplest approach, but perhaps not the most efficient, is to scan the local $\text{count}$s $p$ times and pack the input elements and the associated $\text{count}$s in a contiguous block for each destination. After receiving elements from the others, a process can inspect the incoming $\text{count}$s and place the incoming elements in their intended position in the $\text{output}$ array.

Parallel test program

Your first task is to implement the parallel count sort algorithm described in Section 2 along with a parallel test program capable of evaluating both the correctness and the performance of the parallel algorithm. The main steps of the test program are outlined below. Process 0 is referred to as the root process and is responsible for setting up the input array, verifying the correctness, and input/output.
1. Obtain the length, $n$, of the input array from the command line.

2. Allocate an `int` array of length $n$ on the root and populate it with random integers using the `rand` standard C library function. This is the input array.

3. Redistribute the input array from the root to all $p$ processes. The target is a block distribution with block size $b = n/p$.

4. Call the parallel count sort algorithm, producing an output array distributed identically to the input array.

5. Redistribute the output array from all $p$ processes to the root process.

6. On the root process, sort the input array using the `qsort` standard C library function and verify that the outputs agree. Output the length of the array, the number of processes, the time for Step 3, the time for Step 4, the time for Step 5, and whether or not the output was correct.

Note that in order to measure the time for Steps 3–5, you need to insert barrier synchronizations at appropriate locations and Steps 3–5 should be repeated multiple times ($\approx 10$) in order to gather statistics on the execution times and obtain some degree of confidence in the results. When presenting the final results, it is advisable to choose the minimum measured execution time for each configuration of $n$ and $p$.

Your test program should have the name `mpi-count-sort` and take the length of the input array as its only command line argument. For example, to run the program on 16 processes with $n = 1000$ one should issue the command

```
mpirun -np 16 ./mpi-count-sort 1000
```

or, if running OpenMPI via the Abisko batch system, the command

```
srun -n 16 ./mpi-count-sort 1000
```

You are allowed to (but not forced to) rely on the assumption that $p$ evenly divides $n$. However, any $n$ should be accepted as input to the program and, if necessary, silently rounded down to the nearest multiple of $p$. In the example above, the program should accept $n = 1000$ as the input but silently round it down to $n = 992 = 62p$. Be prepared to answer the following question:

- How can the program be modified to support arbitrary $n$?

4 Performance evaluation

Alongside your parallel count sort implementation, also implement the sequential count sort algorithm (copy/paste from Section 1). Measure both the sequential execution time (again, perform repeated experiments and select the best execution time) and the parallel execution time on varying array lengths. For example, test the lengths

$$n \in \{100, 200, 300, 500, 700, 1000, 1500, 2000, 4000, 8000\}$$
on one node of Abisko using

\[ p \in \{6, 12, 18, 24, 32, 40, 48\}. \]

Compute the speedup, \( S \), by dividing the sequential execution time with the parallel execution time. Produce a figure with \( n \) on the \( x \)-axis, \( S \) on the \( y \)-axis and one graph for each \( p \). For each \( p \), how large must \( n \) be such that \( S > 1 \)?

5 Examination

This assignment should be completed either individually or in pairs (we recommend that you work in pairs). You may only discuss the assignment with others if the discussion is held on a high level of abstraction and, in particular, all the source code and all the documentation must have been produced by the group member(s).

The examination will be performed orally with the aid of printed copies of the source code and the results of experiments (you are expected to print and bring these to the examination). You should be prepared to answer detailed questions about the source code, the parallelization scheme(s), and the results of your experiments. In particular, you should have prepared answers to the questions raised in this document.

Electronic copies of the source code and the results should be packaged and sent by email to both lecturers no later than the due date. Put the assignment number and the user names of the group member(s) in the subject line. Place the files in a directory named after the CS usernames of the group member(s) in the following form: “user[-user]?” For example: “dv09xyz-ens12abc”. Include instructions on how to build the programs from the sources, a list of required modules, the raw data from the experiments, and the SLURM batch scripts required to reproduce the experiments. Compress and package the directory in a gzipped tarball (file ending .tar.gz) using the same naming scheme as the directory itself (e.g., “dv09xyz-ens12abc.tar.gz”).