Report: Exercises 2

Sevde Yanik & Sima Esmaeili

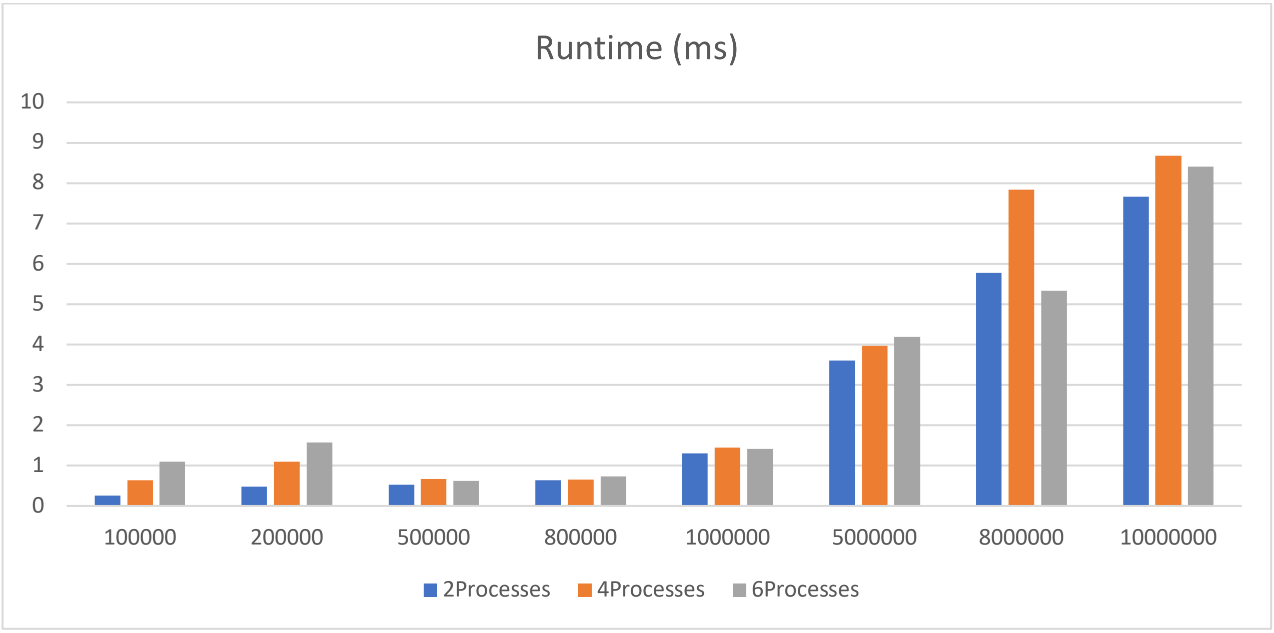
Problem 1

Using MPI, a broadcast operation is implemented which broadcasts the whole content of a vector from the source process to all the other processes.

1. In the source process a large vector with random integers is generated.
2. Timer is started and current time is retrieved as start_time.
3. Broadcast operation takes place. (The source process sends the whole content of the vector to all the other processes)
4. Timer stops and retrieves current time as end_time.
5. The sum of the 3 least significant bits is calculated for each process and they are printed together with the ranks of the processes. (To make sure the broadcast operation run correctly)
6. The source process calculates the total time for the broadcast operation by substracting end_time and start_time, then it prints the total_time out.

The program is run with different vector sizes and different number of processes, the runtimes are noted below. (time in ms)

		Vector size							
N		100000	200000	500000	800000	1000000	5000000	8000000	10000000
o	2	0,26	0,48	0,52	0,64	1,31	3,61	5,78	7,66
.	4	0,64	1,09	0,67	0,65	1,45	3,97	7,83	8,68
P	6	1,09	1,57	0,62	0,73	1,41	4,19	5,33	8,4



From the results obtained we can see that increasing the vector size also increases the runtime. This is pretty much expected as larger vectors require more time to transmit. Also, in most cases having higher number of processes didn't improve the runtime. This might be due to an overhead being introduced during the communication.

Problem 2

The program is modified, it still uses MPI but this time the broadcast operation is a binomial tree broadcast. The content of the vector is sent to some of the processes, and they pass them down to other processes.

1. In the source process a vector with random integers is generated.
2. Timer is started and current time is retrieved as start_time.
3. Binomial tree broadcast operation takes place, using MPI_Send() and MPI_Recv()
    a. A variable called num_steps keep track of the steps taken in the binomial tree broadcast.
    b. A variable called power_of_two is used to keep track of the power of two used in each step.
    c. Inside of a while loop, each process checks if they send or receive vector_data.
    d. The power_of_two is multiplied by 2 on each iteration of the while loop.
    e. The num_steps are incremented on each iteration of the while loop.
    f. The while loop ends when power_of_two is greater than the number of processes.
4. Timer stops and retrieves current time as end_time.
5. The sum of the 3 least significant bits is calculated for each process and they are printed together with the ranks of the processes.
6. The source process calculates the total time for the broadcast operation by substracting end_time and start_time, then it prints the total_time out.

The program is run with different vector sizes and different number of processes the runtimes are noted below. (time in ms)

	Vector size								
N		100000	200000	500000	800000	1000000	5000000	8000000	10000000
o	2	0,25	0,47	0,57	0,65	1,42	2,99	6,02	6,38
.	4	0,57	0,95	1,24	1,41	1,31	10,24	9,29	12,51
P	6	0,73	1,47	1,7	1,97	3,1	13,6	15,8	17,08

Comparing the binomial tree broadcast to the normal broadcast we unfortunately can't see an improvement, this might be the result of a few factors:

- The binomial tree broadcast involves more communication steps compared to the normal tree broadcast, there might be overheads introduced in these additional communication steps.
- There may be a load imbalance between the processes, some processes may have to handle more communication.
- There may be synchronization delays between processes to ensure the correct ordering of the message.

Problem 3

The program that uses parallelism to compute prefix-sums is implemented in C++ using MPI. Our program computes the terms xi for i = 1, ..., n from a linear recurrence xi = ai * xi-1 + bi where x0 = a0 for given sequences ai and bi.
Breaking down the program step-by-step:

1. Three arrays a, b and x are initialized, a has values from 1 to 10 and b is initialized with 1 and x with 0.
2. The prefix sums of sequence a are calculated using MPI_Scan. The results are stored in prefix_sums_a array. The MPI_Scan performs a reduction operation on the input array.
3. The terms for x[i] are computed using the linear recurrence. The first term x[0] is set to a[0] and the rest of the terms are computed using xi-1, ai and bi elements.
4. The results of the prefix sums are printed out.

The distribution technique which is implemented in the code leverages MPI to evenly distribute the workload of computing the xi among multiple processes. By dividing the total number of terms equally among the processes, load balancing is achieved, ensuring that each process receives a comparable amount of computation.

Problem 4

The O(n/log n)-processor EREW PRAM algorithm that finds the first one in a boolean array of size n in O(log n) time using MPI is explained step-by-step:

1. The function called find_first_one function takes a reference to a vector<bool> called A as its parameter. This function is responsible for finding the index of the first occurrence of one in the array.
2. The function retrieves the total number of processes (num_procs) and the rank of the current process (rank) using MPI functions MPI_Comm_size and MPI_Comm_rank, respectively.
3. It calculates the local size of the array (local_size) based on the total number of processes and the size of the input array. That way, each process will be responsible for searching a portion of the array. (n/p)
4. The local start index (local_start) and the local end index (local_end) are calculated based on the rank of the current process and the local size.
5. Each process initializes the local variable local_first_one to -1. This variable will store the index of the first occurrence of true found in the local portion of the array. If no true value is found, the variable will remain as -1.
6. Each process searches for the first occurrence of true in its assigned portion of the array by iterating from local_start to local_end. If a true value is found, the index is stored in the local_first_one variable, and the loop is broken.
7. The MPI_Reduce function is used to find the minimum value of local_first_one across all processes and store it in the global_first_one variable on the root process (rank 0). This effectively determines the minimum index of the first true value across all portions of the array.

8. If the current process is the root process (rank 0), it checks the value of global_first_one. If it is not equal to -1, it means a true value was found in the array. The root process then prints the index of the first true value. If global_first_one is -1, it means no true value was found, and the root process prints a corresponding message.
9. The main function initializes MPI using MPI_Init.
10. A vector<bool> called A is created and populated with the input values. In this example, A represents the array [0, 0, 0, 1, 0, 0, 1, 1].
11. The find_first_one function is called with the array called A.
12. MPI is finalized using MPI_Finalize.

Output: First one found at index: 3

Performance Analysis:
The program finds the first one in an array in O(n/p) time and since p = O(n/log n) the runtime complexity of O(log n) is achieved.

Problem 5
The algorithm computes and returns an array B that contains the non-zero elements of A in ascending order, let's consider :
A = {false, true, false, true, true, false, false, true, true, true}.
the output is B = {2, 4, 5, 8, 9}.

Steps:

1. Compute counts of non-zero elements in parallel and store them in count_values.
2. Perform parallel prefix sum computation on count_values.
3. Scatter non-zero elements from A to their corresponding positions in B based on the computed counts.
4. Sort array B in ascending order.
5. Delete zeros from B to obtain the result.

Performance Analysis
The algorithm achieves a running time of O(log n) due to the parallel prefix sum computation and O(n/log n) processors utilized for counting non-zero elements.