**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 | | | | | | | | |
| No. P |  | 100000 | 200000 | 500000 | 800000 | 1000000 | 5000000 | 8000000 | 10000000 |
| 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 |
| 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()
   1. A variable called num\_steps keep track of the steps taken in the binomial tree broadcast.
   2. A variable called power\_of\_two is used to keep track of the power of two used in each step.
   3. Inside of a while loop, each process checks if they send or receive vector\_data.
   4. The power\_of\_two is multiplied by 2 on each iteration of the while loop.
   5. The num\_steps are incremented on each iteration of the while loop.
   6. 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 | | | | | | | | |
| No. P |  | 100000 | 200000 | 500000 | 800000 | 1000000 | 5000000 | 8000000 | 10000000 |
| 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 |
| 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.

**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 step-by-step:

1. A variable called step is initialized and set to 1.
2. A variable called found\_index is initialized to -1 (this means no 1 has been found yet)
3. Get into a while loop so long as the step variable is smaller than or equal to the size of the array.
4. Each processor goes over the respective parts of the array and set a variable called local\_index to current index if the value of the element is 1.
5. Each processor performs an exlusive write operation to update the found\_index to the smallest index wherea 1 was found among the processors.
6. The value of the step variable is doubled.
7. If the found\_index is something other than -1 print the index of the first occurrence of 1. If not print something like “no 1 found in the array”.

In each iteration the part to be searched is halved by doubling the value of the step variable. The time complexity of the algorithm is O(log n) because the number of iterations is logarithmic to the input size.

**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.