Report: Exercises 3

Sevde Yanik & Sima Esmaeili

Problem 1

Using MPI, write a C++ program that implements the parallel version of the quicksort algorithm. Number of processes is equal to number of elements. Test with different input sizes and verify correctness.

This program implements a parallel quicksort algorithm using the Message Passing Interface (MPI) for distributed computing. The input is provided as an array of integers, where the example input array `data[] = {4, 2, 8, 1, 5, 7, 3, 6, 9, 0}` consists of ten elements. The program divides the input array into smaller chunks and distributes them among multiple processes using MPI. Each process performs quicksort on its assigned chunk of data. The sorted chunks are then merged iteratively until a single sorted array is obtained. In this case, the outcome of running the code would be the sorted array `0 1 2 3 4 5 6 7 8 9`, which represents the elements of the input array in ascending order. This approach demonstrates the efficiency and scalability of parallel sorting with quicksort, leveraging the power of distributed computing using MPI.

Problem 2

This code showcases the application of the fast inefficient ranking algorithm to partially sort a 4x4 character array. The input is represented by the `arr` variable, which holds the initial array. The program starts by displaying the initial array as the "Initial Array" output. Subsequently, the fast inefficient ranking algorithm is executed, involving multiple passes over the array to compare and swap adjacent elements within each row. The step-by-step process illustrates the comparisons and swaps performed during each pass. Finally, the "Sorted Array" is printed, presenting the partially sorted array as the resulting output. It is important to note that the fast inefficient ranking algorithm employed here is not an efficient sorting method, but rather serves as a demonstrative example to highlight sorting principles rather than emphasizing performance optimization.
Input:
{'I', 'H', 'Y', 'P'},
{'S', 'B', 'R', 'P'},
{'X', 'T', 'W', 'J'},
{'Q', 'D', 'A', 'B'}

the partially sorted array as the resulting output:
{'H', 'I', 'P', 'Y'},
{'B', 'P', 'R', 'S'},
{'J', 'T', 'W', 'X'},
{'A', 'B', 'D', 'Q'}

Problem 3

1. **For the two parallel loops in BucketSortPRAM, state for both arrays A and B whether there is concurrent or exclusive read/write access to their elements.**

For the first parallel loop where insert function is called:

- Each processor reads the value A[i] to determine its index and the bucket it corresponds to B[index(A[i])]. -> Concurrent read
- Each processor writes the value A[i] into the corresponding bucket B[index(A[i])], writing into the same bucket concurrently is not allowed as it may lead to inconsistency. -> Exclusive write

For the second parallel loop where the BubbleSort function is called:

- Each processor reads the elements of the bucket it is assigned B[i]. -> Concurrent read
- Each processor writes the sorted elements back to the bucket it is assigned B[i], this there is no concurrent writing to the same bucket. -> Exclusive write

2. **Implement a parallel version of copy in an EREW PRAM model.**

```
parallel_copy(A: Array[1,…,n], B: Array[1,…,nb]):

    pfill(A, 0) // Initialize array A with 0 values using n/2 processors

    prefix_sum = pPrefAdd(B) // Compute the prefix sum of array B using n/2 processors

    for i = 1 to len(A) in parallel do

        dest = prefix_sum[i] // Destination index for the element in array B

        A[dest] = B[i] // Copy element from array B to array A at the appropriate index
```

What is the parallel complexity (depending on nb and l, the maximum length of any array in B), and how many processors can your algorithm use?

The parallel complexity of this algorithm depends on the values of nb (number of buckets) and l (maximum length of any array in B). Assuming n is the length of the input array A:

The pfill operation takes O(log n) time and uses n/2 processors. The pPrefAdd operation takes O(log n) time and uses n/2 processors. The copying loop iterates lenA times, which is equal to n. Each iteration takes constant time, so the loop has a parallel complexity of O(1). The overall parallel complexity is O(log n) + O(log n) + O(1) = O(log n). The algorithm can use a total of n processors, as both pfill and pPrefAdd use n/2 processors each, and the copying loop can use the remaining n/2 processors.

Problem 4

This code demonstrates the pancake sorting algorithm, which is used to sort an array of numbers. The input array contains the values {54, 85, 52, 25, 98, 75, 25, 11, 68}. The pancakeSort function applies the sorting algorithm, which involves finding the largest number in the array and flipping the array multiple times to move the largest number to its correct position. This process is repeated until the entire array is sorted. The code uses OpenMP, a parallel programming framework, to speed up the sorting process by executing some operations concurrently. Finally, the sorted array is printed to the console.

Problem 5

You are given two sorted arrays A = [a1,...,an] and B = [b1,...,bn], each of size n. The goal is to merge them into one sorted array C = [c1, . . . , c2n] of length 2n.

(a) We first consider the following subproblem. Given an index i ∈ {1, . . . , n}, we want to find the final position j ∈ {1, . . . , 2n} of the value ai in the array C. Provide a fast sequential algorithm to compute j. What is the running time of your algorithm?

We can use a binary search algorithm to find the final position (j) of the value ai in the merged array C.

1. Initialize variable called start as 0 and end as 2n - 1.
2. While start <= end, do the following:
3. Compute a middle index as (start + end) / 2.
4. If ai is equal to the element at middle index in array C, set j as middle index + 1 (since j = 1,…,n) and break the loop.
5. If ai is less than the element at middle index in array C, update end as middle index - 1.
6. If ai is greater than the element at middle index in array C, update start as middle index + 1.
7. Return j.

Since we do a binary search on the merged array, the running time of this algorithm is O(log n).

(b) Use the above algorithm to construct a parallel merging algorithm. The work T1 of your algorithm should be at most O(n log n), and the span T∞ should be as small as possible. What is the span T∞ of your algorithm?

To construct a parallel merging algorithm, we can utilize the sequential algorithm from part (a) and divide the work among multiple processes. Here's a parallel merging algorithm:

1. Let’s say the number of processes available is P.
2. Each process p receives a subset of the array A and B such that process p receives n/P elements from both arrays. This ensures that the workload is evenly distributed among the processes.

3. Each process independently performs the sequential algorithm from part (a) on its subset of elements to find the corresponding positions in the merged array. Thus, work is divided among the processes.
4. Each process p communicates the positions it computed to process 0. This allows process 0 to collect the positions from all processes.
5. Process 0 collects the positions received from all processes and merges them into a single sorted array C. This step combines the computed positions to obtain the final merged array.
6. Process 0 prints out the final merged array C.

The work T1 of this algorithm is O(n log n) since each process performs a binary search, and there are log n levels of binary search. The span T∞ is O(log n) since the binary search is inherently sequential, and each level of binary search depends on the previous level's results.

(c) We now want to solve the merging problem in constant time in parallel. Show that by using O(n) processes, the subproblem considered in (a) can be solved in O(1) time. Use this to derive a constant-time parallel algorithm to merge the two sorted arrays. How many processors do you need to achieve a constant-time algorithm?

To solve the merging problem in constant time in parallel, we can use O(n) processes. Here's the algorithm:

1. P is the number of processes available, where P >= n.
2. Divide both arrays A and B into P equal-sized subarrays and distribute them among the processes.
3. Each process p performs the sequential algorithm from part (a) independently on its subarrays to find the corresponding positions in the merged array.
4. Each process communicates its computed positions to the root process (e.g., process 0) using an MPI gather operation.
5. The root process receives all the computed positions from other processes and merges them into a single sorted array.
6. The root process outputs the final merged array.
7. This algorithm achieves constant time because each process only performs a constant number of operations, and the communication step takes O(1) time with O(n) processes.
8. To achieve a constant-time algorithm, you need O(n) processors, where n is the size of the input arrays A and B.

P.S. In our program the size for the arrays A and B is 4, so we run with mpirun -np 4.