Title:-**Parallel Implementation of Bucket Sort Algorithm in MPI and CUDA**

**Research Domain**:-Sorting Algorithms

**Abstract:-Various methods, such as address-calculation sorts, distribution counting sorts, radix sorts, and bucket sorts, use the values of the numbers being sorted to increase efficiency but do so at the expense of requiring additional storage space. In this implementation of bucket sort presented, the primary advantanges are that:-**

**(i) linear average-time performance is achieved with an additional amount of storage equal to any fraction of the number of elements being sorted.**

**(ii) no linked-list data structures are used (all sorting is done with arrays).**

**Analytical and empirical results show the trade-off between the additional storage space used and the improved computational efficiency obtained.**

**Computer simulations show that for lists containing 1,000 to 30,000 uniformly distributed positive integers, the sort developed here is faster than both Quicksort and a standard implementation of bucket sort. Furthermore, the running time increases with size at a slower rate.**

**ALGORITHM USED:**-

The proposed bucket sort, called *Groupsort*, breaks the range of values, say, u to v, intosubranges of equal length, also referred to as *intervals*, numbered 0 through K − 1. (It is the flexibility in the choice of K that allows the user to control the amount of additional storage space that is needed subsequently.) Hereafter, the set of elements of the array being sorted that fall within each subrange is referred to as a *group* .

As mentioned previously, linked lists are used to store the numbers in the groups because it is not known, apriori, how many numbers are in each group. This problem is overcome here by using one additional pass through the array to count the number of elements that fall in each group. These totals are stored in an additional array Count of K elements. Knowing the number of elements in each group allows one to partition the original array x into K subarrays (of varying lengths) that will eventually hold the numbers in each of the K groups, thus avoiding the need for linked lists.

To illustrate this feature, suppose there are 12 keys, stored in x[1],...,x[12], whose values range from u = 1 to v = 199, as shown in Fig. 1a. This range is then sub-divided into, say, K = 4 equal intervals. In this case, the 4 intervals are:

Range of Interval values

0 1–49

1 50–99

2  100 – 149

3  150 – 199

One additional pass through the array in Fig. 1a provides the number of elements (3, 3, 2, and 4, respectively) falling in each interval. This information results in the partition of the original array x into the 4 subarrays shown in Fig. 1b, in which the pointer (arrow) numbered k indicates the starting position for the numbers that will eventually be put in group k.

Having determined the starting location of each subarray, one more pass through the array is used to insert each element into the next available position of the group to which the element belongs. More specifically, for each i = 1, . . . , n, x[i] is exchanged with the element in the next available position of the subarray in x to which x[i] belongs. In this example, the first key, whose value is 75, belongs to group 1 and is therefore exchanged with the next available element in subarray 1 of the array x, namely, x[4] = 98, as seen in Fig.1b. This exchange process eventually results in all numbers being placed in their appropriate group, as illustrated for this example in Fig. 1c. In the implementation that follows, it is shown how these steps are performed using one additional array of K elements, thus allowing the user to control the amount of additional storage space.

As with other bucket sorts, once the elements of x are moved to their proper group, the numbers in each group must be sorted to obtain the final list. Because in Groupsort the numbers in each group are stored in consecutive positions of the array, it is possible to use Quicksort to do so rather than the less efficient insertion sort.

*Implementation*

On the basis of the foregoing discussion, an efficient implementation of the proposed algorithm follows (in which it is assumed that the values of the n keys are stored as positive integers in the array elements x[1], . . . , x[n] and K groups are used):

*An efficient implementation of Groupsort*  
*Step 1:* Find the minimum and maximum values of the keys, say, u and v, respectively,

and move those elements to x[1] and x[n]. It remains to sort only x[2], . . . , x[n − 1].

*Step 2:* For each j = 0, . . . , K − 1, store into the array element Count[j], the number of elements of x whose key has a value in interval j of the range u to v. That is, after initializing each Count[j] to 0, compute, for each i = 2, . . . , n − 1,

􏰃 J:=[K ∗ (x[i] − u) ]􏰄 v−u+1 ]

(The computation of j in this manner is the same as the corresponding computation in interpolation search.) In so doing, move any element belonging to group 0 or group K − 1 to the next available position in those two subarrays (which is not possible for elements belonging to other groups). After this step, only those elements belonging to groups 1,...,K − 3 must be moved. (The elements in group K − 2 will then automatically be in their correct positions.)

*Step 3:* Now reuse the array Count to indicate the starting location of each subarray by setting Count[K − 1] to n + 1 − Count[K − 1] and then computing, for each j = K − 2, . . . , 1,

Count[j] := Count[j + 1] − Count[j].

When a number is moved to its correct group k, the value of Count[k] is updated to keep track of the next available position in subarray k. Because the values of Count are changing continuously, the position where one group ends and the next begins is lost. One way to maintain this information is to use a new array of K elements to mark the ending position of each group. However, it is possible to accomplish the same objective without using an additional array. This is done by converting the value in the last postition of each subarray into a sentinel value. When all keys are positive integers, as is assumed here, a sentinel value is the negative of the last element in each group. These elements are made positive when they are moved. (To apply this technique to keys of other values, either an additional bit for each element or an additional array of K elements is needed.) In this case, therefore, mark the end of each subarray by replacing the element in the last position of each subarray with its negative.

Move each element x[i], other than those in group 0 and group K − 1, to the next available position in the subarray to which x[i] belongs. This is done by moving sequentially all elements currently in subarray j (j = 1, . . . , K − 3) to their proper groups, until all elements in subarray j belong to group j. This, in turn, is accomplished by exchanging the element x[i] in the next available position of subarray j with the element in the next available position of the subarray to which x[i] belongs, until the element in x[i] belongs to group j. The values of the array Count that point to the next available position in each subarray are updated continuously. Negative values of the array elements indicate when one group is completed and the next group begins. These values are made positive when they are moved or when a group is completed. For further efficiency, before moving an element in the array, check to see if that element already is located in its correct group; if so, then skip that element and increase the pointer in that subarray by 1.

*Step 4:* Sort the numbers in each of the individual groups using Quicksort.

GroupSort is designed under the assumption that the values of the keys are distributed uniformly throughout the range. If this is not the case, the number of elements in a group can be large. When the number of elements in a group exceeds some critical value, the use of GroupSort, rather than QuickSort, to sort the numbers in that group may be more efficient. The determination of this critical value is given in the next section, together with an analysis of the average running time of the algorithm and a space/time trade-off.

Team Members:-

name:-Anubhav Tikku Reg no-160905256 rn-59

name:-Divyansh Verma Reg no-160905256 rn-49

name:-Saurav Panda Reg no-160905256 rn-37