Researchers offer a parallel bucket-sort technique that uses n processors and runs in time O(log n). The algorithm employs a method that calls for more than the sum of the processors' time and space. No memory conflict is allowed in the realistic model that is employed. Additionally, a method is provided to sort n integers utilizing n^1+1/k processors in O (k log n) time, where k is any arbitrary integer. This procedure's computing paradigm allows for several concurrent fetches from the same memory address.

Serial algorithms frequently have to trade off time and space. A minimum quantity of space is needed to accomplish an issue within a set length of time. If we provide more time for the procedure, this space demand may be decreased. There will be a minimum amount of space needed in the limit.

We shall present an algorithm design technique which dramatically exemplifies the three-way tradeoff between space, time, and processors. It is hoped that this technique can also be applied to strictly serial algorithms.

There is frequently a trade-off between time and space, meaning that in order to solve a task within specific time constraints, a minimal amount of space is required. This is true for the parallel version of the bucket sorting algorithm as well. In order to solve a problem in a given number of processors, a minimum amount of time is needed. This time will be lowered with more processors on hand. This trade-off between time and processors has received a lot of attention in the parallel processing community. Now presenting an algorithm that balances time, space, and processors in three different ways. Muller and preparation presented a network with 0 (n2) processing components that can sort n numbers in O (log n) time.

In technique 1, duplicate numbers are eliminated and n numbers are sorted over n parallel processors in O (log n) time. For implementation, each processor must put the value of I in a bucket designated as ci. This presents an issue, therefore we develop a solution by getting rid of duplicates of the same number. When we put I in bucket ci, only one processor (the one with the least index) will be active for each number that appears among the numbers being sorted. Then, each processor checks to see if its "friend" is active in the same region, and the processor with the higher rank and greater index 0 deactivates, if your buddy isn't online or has a better rank, the processor will keep going. It should be mentioned that this bucket-sort method needs n processors, time T O (log n), and space O (mn).

The difference between Algorithm 2 and Algorithm 1 is that Algorithm 2 provides the actual ranks of the input numbers. We will keep track of the number of processors that were active during each block, and if a processor notices an active buddy, only the lower buddy will be active following the presence of just one representative of each number in the numbers that need to be sorted. Each duplicate now contains a count of the CTs that are identical to it but have a higher index. The sum of these differences plus one is Rank (the D value). Space S = O (mn), time T = O (log n + log m), and n processors are used in algorithm 2.

In algorithm 2.3, it is assumed that memory area A has been initialized to zero. The issue is that many entries that are viewed simultaneously will have multiple contents referring to the same place. Algorithm 3 is available to fix this issue.

With Algorithm 3, we divide the n number into n^1/2 groups using n^3/2 processors and a time complexity of O (log n). We then apply bucket sort to the elements using count[j] as the key for the first element in the group, which is equivalent to enumeration sort. All elements then perform a binary search on the n^1/2 groups before a bucket sort using j as the key is performed on all of the elements. Algorithm 4 is created by slightly modifying algorithm 3.

Using n^4/3 processors, Algorithm 4 first divides the input number into groups, each having n^1/3 elements. Within each group, we calculate the count for each element, and then we apply bucket sort to the count to put the elements in rank order. Next, we divide the n^2/3 groups into n^1/2 sectors, each with n^1/3 groups, and perform binary search for each element within each sector. Finally, we calculate the value of count (joke). Then calculate count[j] as the product of count[j, k] over k, and then do a bucket sort on all n items; this has an O (k log n) time complexity and requires n^1+1/k processors.

Although we were able to eliminate memory-store conflicts, all of these techniques had memory-fetch conflicts.