**FAST PARALLEL SORTING ALGORITHM SUMMARY**

In this paper D. S. Hirschberg presents a parallel bucket-sort algorithm whose time complexity is O (log n) and it uses n processors. The approach it uses consumes more space as compared to the product of processor and time combined. Frequently there are space and time compromises in algorithms running serially. So, to compute it within a certain time, a minimal amount of space is required. The space requirement can be minimized if more time is given to the process. In recent years more development work is being done on parallel algorithms with major problem areas such as sorting, computing polynomials and arithmetic expressions and graph theories. In order to minimize the time in parallel algorithms more processors are used and it has a direct proportionality with time. An algorithm is designed to sort n numbers in O (log n) time that requires less processors. First the above-mentioned algorithm is used at the compromise of greater space requirements but lesser no of processors and time. A SIMD model is used. It will sort the numbers in O (log n) using n processors in time O (log n) under the assumption that there are no duplicate numbers. The result then goes to the second algorithm A parallel version of bucket sort. Implementation of the parallel bucket sort would be for each processor to be assigned to cz; the ith number being sorted. A memory issue could come up resulting in simultaneous attempts of several processors to store different values of I into the same bucket.

The solution is to eliminate duplicate copies. Then among the numbers being sorted, allow only one processor to be active when we place it in a bucket. There will be m areas of memory, one for each bucket and each area will be of size n, the number of input numbers to be sorted.

One by one, each processor will check whether or not it is "buddy", a word used in the Buddy System for DMA. If yes, then the processor with higher rank will deactivate. If not active or is active but not in the higher rank, then the processor will continue to shift until it marks the location of buddy. At the last iteration, a sign will be present at each location determining the bits of address of active processors in the area it is present. It is seen that this algorithm requires space S = O(mn), time T -- O (log n), and the use of n Processors. There is another bucket-sort that will give the actual ranking of the input numbers, equal numbers being kept in the same order but assigned different ranks. The difference between it and the first algorithm is that it keeps a running count of how many processors were originally active in each block of indices of size 2k. Processors in the active state keep their count at the head of the largest block that they have. As a result, there will be at most one active processor per area and A [j, 0] will be the number of different fs such that ci = j. A processor in an inactive or idle state, will keep its count at the head of the largest block which had no other processors of index smaller than I.

At this instance we have each number that is among the numbers to be sorted and we have a count of how many times each number occurs. After this we total the count of all numbers that are greater than ci in order to know the actual ranking of the numbers. This is done in the same manner used above. Algorithm 2 requires space S = O(mn), time T = O (log n + log m), and the use of n processors. Then there is an algorithm (3) that will sort n arbitrary numbers in time O (log n) and will use n 3/2 processors. A modified version of Algorithm 3 using the same time and require only n 4/3 processors is algorithm 4. So, we can make an algorithm to sort

n numbers in O (k log n) time that uses n 1+1/k processors. Although these algorithms avoid memory-store issues, they don't avoid memory-fetch issues. That is why, allowing more than one processor to simultaneously access the same memory location overcomes this.