**Summary**

**MEMBERS**

**k201639 aman nadeem**

**k201068 syed abu bakr**

However, a lot of work has lately been done to design algorithms for parallel processing. Serial techniques have a space-time tradeoff. It takes very little time, but if we use additional processors for the computation, this time might be even shorter. A third method of algorithm creation offers a trade-off between the number of processors, available space, and available time. In this article, these algorithms are covered.

***First Parallel Bucket Sort Algorithm*,** this algorithm will sort n numbers using n parallel processors in O(log n) time by following an approach that a processor Pi is assigned to Ci to place the value of i in bucket Ci but there is a problem that if for different indexes of I if the value Ci is identical than there may occur a memory conflict, to resolve this issue all the duplicates are to be removed by deactivating Pi temporarily if there is Pj whose index j is smaller than i and Ci=Cj. There will be m areas of memory of size n one for each bucket, in each area j the processor Pi having ci=j will leave a mark then like a binary-tree marks of other active processors will be searched. If two processors discover each other’s presence than higher ranking one will be deactivated. Each processor Pi can make mark, iteratively each processor then determines whether its buddy is active or not if yes then no shift will occur. Unless the buddy's location has a lower index than the one being used currently, the processor will continue to move its mark there. When the kth iteration is complete, a mark will be present at each location whose last k bits are zeros and whose other (log n) - k bits match the corresponding bits of an address of a processor that is active in that area. Each location will then be indicated if any of the 2k processors were previously operating there. If any of the n processors that were initially active in that area, i.e. if any of the n numbers to be sorted was j, the area bucket number, were present after log n iterations, the first location in that area will be marked.

***Second Parallel Bucket Sort Algorithm*,** in this algorithm we will keep a running count of processors which are active in each block of size 2k,in any case Pi will add to their running count of no. of processors having indices greater than i. At the end there will be one active processor per area. After obtaining a count of no. of times each number occurs we will now take accumulative count of numbers greater than ci. Each number has count of total numbers that are greater than it plus the number of ci’s equal to it. Each of the duplicates has a count that are equal too it but of higher index. The D-value, i.e. rank, is just the difference of these two quantities plus one. It can be calculated by: for all I do{if floag=0 then D[i}<-A{ci,0}-A{ci,y}+1}. If we want that only one processor simultaneously access a location, not even for fetches, then the *D*-values of duplicates can be evaluated using the reverse of previously discussed algo. The algorithm requires S=O(mn), T=O(log n + log m), and ‘n’ processors. The algorithm assume that an area of memory is initialized to zero. But there are methods in which it’s unnecessary to initialize the area. For serial programs, one can include at each location a pointer to a backpointer on stack. This method is valid for parallel programs also but with a restriction that at each step of the process, each active processor can initialize the location its buddy is working on (to zero), then reinitialize the contents of location it’s working on (latest value). A location will thus be initialized if either of the two processes to which it might be relevant is active.

***(Parallel sort using n3/2 processors)*** partition n input numbers into n1/2 groups, each having n1/2 elements. For each element, j, determine count[j]=(!= of I such that ci<cj)+(!= of i<=j such that ci=cj). This can be done in O(log n) by n1/2 processors per element(total n processors per group or n3/2 processors in total). Within each group bucket sort using count[j] as key for jth element in the group. Ccount[j]<-cj. There’ll be no memory conflicts since the count[j] within a group are all distinct. In the next step: count[j,k]= if k<g, != of elements I such that ci<=cj, if k=g,j, if k>g, != of elements I such that ci<cj. ci referes ith element in group k and cj is fixed. For all elements, j, evaluate count[j]=sum(over k) of count[j,k]. this will be done in time O(log n) and requires n1/2 processors for a total of n3/2processors. For a bucket sort on all n elements using count[j] as the key for jth element. There’ll be no memory conflicts since count[j] will be rank of jth element.

***(Parallel sort using n4/3 processors)*** : partition n numbers into n3/2 groups having n1/3 elements. In each group for each element, j, determine count[j]=(!= of i such that ci<cj)+(!= of i<=j such that ci=cj). in each group do bucket sort count[j]. This will rearrange elements in rank order within each group. Divide n2/3 groups into n1/3 sectors, each having n1/3 groups. In each sector for each element j in group g, do binary search of each n1/3 groups in j’s sector. Count[j,k]= if k<g,!= of I in the group k such that ci<=cj, if k=g,j, if k>g, != of i in group k such that ci<cj. Do bucket sort in each sector using count[j] as the key for element j. It’ll rearrange elements in rank order within each sector. Now in each sector do binary search of n1/3 sectors to determine count[j.k]=if k<t, != of I in sector k such that ci<=cj, if k=t,j, if k>t, != of I in sector k such that ci<cj. evaluate count[j]=sum (over k) of count[j,k]. Do bucket sort of all n elements.