Fast Parallel Sorting Algorithms

Summary

In serial algorithms we have space-time tradeoff which means that to solve a problem in a certain time a minimal amount of space is required while in parallel algorithms there is tradeoff between time and number of processors which means that to solve a problem with a specific number of processors minimal time is required.There is another way of designing algorithm which provides a three way tradeoff between space,time and number of processors. This article is about these algorithms. The respective article has given two ways of parallel bucket sort algorithm whose explanation is given below.

Explanation for first Parallel Bucket Sort Algorithm

This algorithm sorts n numbers using n (parallel) processors in time O(log n) with the condition that all duplicate numbers are to be discarded this is because if each processor p_i places the value i bucket c_i then there may be several values of i with identical numbers c_i and a memory conflict can occur. For this, processor p_i will be deactivated if there is another processor p_i whose index, j, is smaller than i, and $c_j = c_i$. For the implementation of this algorithm we have to have m areas of memory. Each area will be of size n i.e. the array size. Within each area, j, the processors (p_i) having $c_i = j$ will leave marks indicating their presence. By using a binary tree the processor will search for the presence of another processor if found the lower ranking one will continue while the higher ranking processor will be deactivated. Now each processor can make its mark at location i in area c_i. Each processor then determines whether or not its *buddy* is active within the same area. If the buddy is active, a higher rank processor will be deactivated, else the processor will continue shifting its mark to the location of the buddy if that location is of lower index than the one currently in use. After log n iterations, the first location in an area will be marked if and only if any of the n processors originally were active in that area, i.e. if any of the n numbers to be sorted was j, the area bucket number.

Explanation for second Parallel Bucket Sort Algorithm

In this algorithm, we keep the count of how many processors were originally active in each block of indices of size 2^k.Active processors keep their count at the head of the largest block that they have investigated (which will be of size 2^k). At the end there will be at most one active processor per area. An inactive processor, p_i, will keep its count at the head of the largest block which had no other processors of index smaller than i. Now one representative of each number is isolated that appears among the numbers to be sorted and the count of how many times each number occurs is obtained. The counts are accumulated of all numbers that are greater than c_i in order to know the actual ranking of the numbers, assuming that duplicate numbers will be kept. The representative of each number that appears among the (ci} has a count of the total number of c_t's that are greater than it plus the number of c_t's that are equal to it. Each of the duplicates has a count of the number of c_t's that are equal to it but of higher index. The D-value, i.e. rank, is just the difference of these two quantities plus one. D-values of the duplicates can be calculated by *for all i do if flag = 0 then D[i]* $\leftarrow$ *A* $[c_i, 0] - A [c_i, y] + 1.$ If we want the processors not access a location simultaneously even for fetches then for the evaluation of D-values the procedure of the above explained algorithm should be reversed. This algorithm for bucket sort requires space O(mn) and tine taken is O(log n + log m) and the use of n processors. The algorithm discussed assume that the area has been initialized to zero. There is a method to make it unnecessary to initialize the area. One can include each location pointer to a backpointer on a stack but this can result in a memory fetch conflict. To cater this issue each active processor can initialize the location its buddy is working on (to zero), then reinitialize the contents of the location it is working on (to its latest value). A location will thus be initialized if either of the two processes to which it might be relevant is active.

Explanation for parallel sort using $n^{3/2}$ processors

In this algorithm the n arbitrary numbers will sort in O(log n). The n input numbers in will be divided into $n^{1/2}$ groups each having $n^{1/2}$ elements. Within each group determine count[j] = (# i such that $c_i < c_j$) + (# of i <= j such that $c_i = c_j$).This is done in O(log n) using $n^{1/2}$ processors per element or $n^{3/2}$ in total. Now sort the elements by bucket sort by using count[j] as the key for j^{th} element in the group. There will be no memory conflict in this case. All elements will now do a binary search of $n^{1/2}$ groups by using $n^{1/2}$ processors which is assigned to each group. Now for all elements, j, count[j] is evaluated as count[j]= sum(over k) of count[j,k]. This will also take O(log n). In the end the n elements will be sorted by bucket sort using count[j] as the key.

Explanation for parallel sort using $n^{4/3}$ processors

In this algorithm the n arbitrary numbers will sort in O(log n). The n input numbers in will be divided into $n^{2/3}$ groups each having $n^{1/3}$ elements. Within each group determine count[j] = (# i such that $c_i < c_j$) + (# of i <= j such that $c_i = c_j$).Now count[j]'s obtained is sorted by using bucket sort to rearrange the elements in rank order in each group. Now $n^{2/3}$ groups is divided into $n^{1/3}$ sectors, each sector consisting of $n^{1/3}$ groups. In each sector sum(over k) of count[j,k] is calculated. Now each sector is sorted by using bucket sort with count[j] as the key element of j. Now for all elements (j) in sector t do a binary search of $n^{1/3}$ sectors to determine, for all k, the value of count [j,k] if k<t, # of i in sector k, k=t, j , and k>t# of i in sector k such that $c_i < c_j$ for all. Evaluation of count[j] = the sum (over k) of count[j,k] is done and in the end the n elements are sorted by bucket sort in O(log n).

Note that these algorithms avoid memory-store conflict but do have memory fetch conflicts