Research Paper abstract

This is a research paper published by D. S Hirschberg from Rice University, situated in Houston Texas. The paper proposes a new fast bucket sorting algorithm techniques. The problem is presented that the conventional bucket sort makes use of O(log n) and utilizes n processors. The new proposed algorithm uses a technique that will make use of more space that the product of the time and processors available. The model provides no memory contention as a realistic model is used as a base. Additionally another procedure is also presented that can sort the numbers in a time complexity of O (k log n) while making use of n^(1+1/k) processors where k is an arbitrary integer. The model for the computation of the procedure is such that the parallel fetches from the same memory block are allowed.

Problem Statement

The paper describes that there is generally a problem with the time and space tradeoff in a serial algorithm. So to solve a problem within a certain time frame, we require a specific or minimal space since these two are inversely proportional in a sequential running time algorithm. So there a recent developments in work while the paper was being written and these were the evaluation of polynomials, matrix and graph theoretic problems and in sorting. When we solve these problems by a parallel algorithm, there is a new dependency. This dependency is the relation of time with the number of processors that are being used. So now to solve a problem using a bounded number of processors, there is a minimal threshold amount needed. This time can be reduced by using more processors which is again an inverse.

Problem solution

This paper proposes a solution for the problem of dependencies. The paper proposes an algorithm Design technique that can illustrate this three way trade-off of memory, time and number of processors. It can be applied on serial algorithms. Also, this algorithm will reduce the number of processors and the time complexity on the expense of space as compared to previous solutions.

Solution Background

The paper addresses the paper of Muller and Preparata who were the first ones to demonstrate a network which could sort n numbers in a tome complexity of O (log n). They had O(n^2) elements processing.

Algorithm 1

Assumptions

This part of the algorithm will sort the numbers while using an n number of parallel processors in a time complexity of O (log n) while assuming that processors have access to a common memory and their own as well. These processors are all synchronized and the model is a SIMD.

How the algorithm works?

The algorithm will sort n numbers using n parallel processors in a time complexity of O (log n) where numbers will be in the ascending order and that <u>any duplicate numbers are discarded</u> to remove the issue of memory conflict from the same numbers. This is a parallel version of the bucket sort.

M areas of memory are present for each bucket. Each will have an area of size n which is the number of inputs to be sorted. For area j, if processors search for the presence and find each other the lower index one will continue and the higher one will deactivate. Each processor will check if its buddy is in its area or not. If buddy is active then no shift else if buddy is of higher rank or inactive then the shift is completed if the index is lower. After k iterations there will be a mark on all locations who have a k bit of 0 and those bits that coincide with the processor. Now we have each number that is sorted and in the next algorithm we will now find their actual rankings assuming that the duplicates will be kept in a manner like done previously.

Algorithm 2.2

Assumptions

This algorithm gives the ranking or order of the input array where equal numbers will be kept in order but with different ranks with the input being for a small predefined set. The same pattern of algorithm 1 will be followed.

How the algorithm works?

In contrast of algorithm 1, instead of a flag, a count will run of how many processors were active in the block of indices for 2^k making this a new algorithm. If an active buddy processor is encountered, then the lower buddy remains active. There processors will add their counts of processors, that had index greater than I. Active one keeps the count in the head of their block. Finally, one will be the most active processor per area, and it will have the number of different I's. Inactive processors, will keep count of smaller indexes of other processors. This algorithm presents a count of the total numbers of count that are greater than equal to it. Each duplicate has a count that is equal to higher indexes. The rank is the difference of these two duplicates.

Algorithm 2.3

Assumptions

In case of persistence that no more than 1 processor may access the memory even for fetches, we used the reverse procedure of 2.1 to make 2.3. In the sequence of algorithm 2 which includes all three variants the space needed is O (m x n), the time complexity is O(log n + log m) while making use of n processors.

How the algorithm works?

Basically for algorithm 2.1 we will reverse the loop and start from log n to 0 as compared to vice versa for algorithm 2.1. Similarly wait till r is greater than k. Likewise, begin only when x is not equal to y. This algorithm also ensures that the area of memory at the en dis initialized to 0. This is because when several instances of the algorithm run, the memory will be cleared after the termination of each run. There are methods that make it unnecessary to initialize the area. For the serial programs, a location pointer can be included to a back pointer on the stack. Every time an entry is accessed, verification can be made by checking that the pointer in entry points are pointing to active regions on the stack and the back pointers to the entry points. This will be valid for parallel processes until a restriction of multiple access for the same location is prohibited even for memory fetch instructions. This is done to prevent any memory conflict events. This is a problem faced in algorithm 2 so now algorithm 3 is proposed as a solution.

Algorithm 3.0.

Assumptions

This will require n ^ 3/2 processors and the input will be from 0 to n-1 where all will be integers. These will be sorted in a stable manner in an ascending order.

How the algorithm will work?

1. First of all we will partition n input numbers into n ^ ½ groups where each will have n ^ ½ elements.
2. For each and every partition, determine the count which will be achieved in the O (log n) time by making use of n ^ ½ professors per each element. The processors will then be assigned one to each element in the group of I and j and be compared with ci and cj. Then summing these can be done again in O (log n) time.
3. Now again in each group perform a bucket sort using the count of j as the key for the jth element in the group. C count (j) will store cj where count j is the offset from the start for each group. Now we eliminated memory conflicts as all count j are distinct within a single group. Step 2 and 3 have effectively sorted elements. Now we use an Enumeration sort to put them in an array.
4. Now for all elements do a binary search for the n ^ ½ groups. For each element in a group, the n ^ ½ groups assigned. The binary search will be used to search for elements in the group to determine if for all k the value for count j,k. if k<g then smaller than or equal to elements, if k-g then j elements and if k>g then no of elements of ci will be less than cj where ci is ith element of group k and cj is fixed.
5. For the elements in j, evaluate the count (j) = sum over k of the count (j , k). This will be done by n^1/2 processors in time O (log n) per element for a total of the n ^ 3/2 processors.
6. Now a bucket sort on all n elements will be done using count (j) as the key for jth elements.
7. End the algorithm 3.

Algorithm 4.0.

Assumptions

This parallel sort algorithm will make use of n ^ 4/3 processors. Since algorithm 3 requires O (log n) and uses n^ 3/2 processors. A simple fictation of algorithm 3 will also use a same order and the magnitude with the change in processors to n ^ 4/3. So now we will implement the parallel sort of bucket.

How the algorithm works?

1. The n input numbers will be partitioned into groups where each will have n ^ 1/3 elements.
2. In each group, for each element j, the count (j) will be determined with a condition that the count(j) is such a number that the ci is less than the cj and that the number of I is less than j such that ci = cj.

3. For each group we will do the bucket sort for the count (j) obtained in the second step. This will then be rearranged into elements of rank order for each group.
4. Now divide the n ^ 2/3 groups into the n ^ 1/3 sectors where each one will consist of n ^ 1/3 groups.
5. For each sector we will now find for each element of j in group g, do a binary search that will search each n ^ 1/3 group in sector of j to determine for all of the k the value of count (j,k) and that if it is equal to

- If k<g, the number in I of group k such that ci is less than equal to cj.
- If k=g, then j.
- If k is greater than g, the number of I in group k such that the ci is less than cj.

Then for each element j we will evaluate the count (j) to be equal to the number of I in the j sector such that the ci is less than cj plus number of I less than equal to j sector such that ci is equal to cj. This number is then a simple sum over k of the count (j,k).

6. Inside each sector, a bucket sort of all elements using the count (j) as a key for the elements in j. This will sort elements in the rank order within each sector.
7. For the elements j in a sector t, a binary search will be done for each n ^ 1/3 sector to determine for all of the k the value such that the count (j,k) will be such that this is equal to if

- K is less than t, then the number of I in the sector k will be such that ci is less than equal to cj.
- If k is equal to t, then j.
- If k is greater than t, the number of I in the sector k will be such that ci is less than cj.

Then the count(j) will be compared to the sum over k of the count (j,k) will come under evaluation.

8. A bucket sort of all n elements will be done.
9. This is the end of algorithm 4.