

CPE 642: Solutions To Midterm Exam

Wednesday, February 28, 2001, 7:05 pm – 8:30pm

1 (20%). Consider an array A of elements a_0, a_1, \dots, a_n . Write an algorithm that creates a new array where a_0 will be in position k such that

$$a_i \leq a_k \text{ for all } i < k$$

$$a_k \leq a_j \text{ for all } k < j$$

Note that I am not asking you to sort the list, only that use a_0 as a pivot to create two lists, where the first lists consists of elements smaller than the pivot and the second list consists of elements greater than the pivot..

What is the complexity of the algorithm (**Hint**. Try to use Insertion sort algorithm we discussed, but you need to modify the algorithm.)

Key. First of all, the Insertion sort I referred in the above problem is called as Enumeration sort in the textbook. The idea is to find the rank of a_0 . Once we find the rank, we can start creating a new list with a_0 in position k , and move the other elements appropriately.

The enumeration sort given in the textbook (page 244) is for PRAM CRCW model with “sum” as the result of concurrent write. For our purpose we can use $n-1$ processors, each processor P_i compares one element a_i with a_0 , and writes 1 if $a_i \leq a_0$. Because of the CRCW model assumed, in $O(1)$ time units, we will find how many elements are less than or equal to a_0 , thus defining the position of a_0 ; after this, we create a new list by setting $b_k = a_0$, and copy element a_i to b_j where j is either less than k or greater than k depending whether $a_i \leq a_0$ or $a_0 < a_i$.

```
Find_Rank_of_a0
begin
    rank = 0;
    for each processor  $P_i$  (  $i \geq 0$  ) do
        if  $(a_i < a_0)$  or  $(a_i = a_0)$  then rank = 1;
        else rank = 0;
    end for;
end;

/* create new list
 $b_k = a_0$ ; left =0; right=k;
for i = 1 to n do
    if  $(a_i < a_0)$  or  $(a_i = a_0)$  then
         $b_{left} = a_i$ ; left = left + 1;
    else  $b_{right} = a_i$ ; right = right+ 1;
end for
```

Note that the second part of the algorithm (for copying the original list into a new list) take $O(n)$ and shown here as a sequential algorithm. If we want to consider a parallel version, we could use a PRAM CRCW model with “arbitrary” write to move elements a either to the left or right subtrees as done in PRAM version of the quick sort shown on page 234. This algorithm needs to be modified since we will start with $root = 0$, since we need to construct a tree with a_0 as the pivot element.

Note that in this case, we have $O(\log n)$ time units. But, you also need to account for tree traversal to create the final lists. The tree traversal itself should take $O(\log n)$ time units.

Since we are interested only in the rank of a_0 , we can actually use CREW PRAM model (that is, use mutex variables to increment rank of a_0), and we can also use CREW PRAM model for copying the original list into a new list.

```

Find_Rank_of_a0
begin
    rank = 0;
    for each processor  $P_i$  ( $i > 0$ ) do
        if ( $a_i < a_0$ ) or ( $a_i = a_0$ ) then
            mutex_lock_rank;
            rank = rank + 1;
            mutex_unlock_rank
        end for;
end;

/* create new list

 $b_k = a_0$ ; left = 0; right = k;

for each processor  $P_i$  ( $i > 0$ ) do
    if ( $a_i < a_0$ ) or ( $a_i = a_0$ ) then
        mutex_lock_left;
         $b_{left} = a_i$ ; left = left + 1;
        mutex_unlock_left
    else
        mutex_lock_right;
         $b_{right} = a_i$ ; right = right + 1;
        mutex_unlock_right;
    end for

```

If the left half and right half of the newly created list are equal in size, the new algorithm above should take $O(n/2)$.

2 (35%). Consider a simple version of implementing Bubble sort on Hypercubes. Consider sorting n elements using $n (=2^d)$ processor hypercube. Processor P_0 sends its element to P_1 ; P_1 keeps the larger of the value received and the value contained at P_1 , and returns the smaller value to P_0 . P_0 then communicates with P_2 and receives the smaller of values, and so on. At the end of

this step, P_0 has the smallest value. Then P_1 starts a similar iteration and communicates with processor P_2, P_3, \dots, P_{n-1} ; and at the end keeps the second smallest value at P_1 . We continue in this manner with processors P_2, \dots, P_{n-2} .

What is the complexity of this algorithm?

Key: Let us start with the steps involved with finding the smallest element for P_0 .

P_0 needs to communicate (and receive smaller element) with each of the other processors.

The average distance between any two pair of processors in a d -cube is $d/2$ (see problem 2. 19 which asks you to compute the average distance in a k -ary hypercube, and the solution for this yield $k*d/4$ as the average distance)

So, the communication will cost $(n-1)*2*d/2$ time units (assuming it takes one time unit to send/receive one data item between neighboring nodes).

Also, the computation (to compare 2 elements to find the smaller element) is one time unit

Continuing, P_1 takes $(n-2)*d + 1$, P_2 takes $(n-3)*d+1$ and so on. Note we can stop this at processor P_{n-1} . The total complexity is given by

$$\begin{aligned} &= (n-2) + d * (n-1 + n-2 + \dots + 2) = (n-2) + (\log n) * [(n*(n-1)/2 - 1] \\ &= O(n^2 \log n) \end{aligned}$$

Not very good since the cost of this algorithm = $O(n^3 \log n)$

The main problem with this approach is that most of the time only 2 processors are active while the remaining $n-2$ are idle.

What happens if you change the algorithm as follows.

In step 1, processor P_0 broadcasts its value to processors P_1, \dots, P_{n-1} ; each processor returns the smaller of the values (comparing the value received from P_0 and the local value). For this purpose we can use one-to-all broadcast (P_0 sends its value to all other processors) and all-to-one receive (P_0 receives smaller values from all other processors). P_0 will then finds the minimum of all values received locally (reduction operation) and keeps only the smallest value. Thus, at the end of this step, P_0 has the smallest value. We move to P_1 , repeat the one-to-all broadcast and all-to-one receive so that the (next) smallest element will be kept at P_1 , and so on.

Will this work? If it does, what is the complexity of this algorithm? If not why not?

Key. This approach will not work. Consider what happens when P_0 has the largest element. During the first one-to-all broadcast, each of the remaining processor will keep the largest value sent by P_0 , and send their smaller values. P_0 will then keep the smallest value and discards all the

other values. At the end of this step has P_0 the smallest value but all the remaining processors contain the same largest value originally contained at P_0 -- all the other values are lost!

3 (20%). Looking at the costs for all-to-all broadcast on 2-D mesh and a Hypercube (with equal number of processors) show which of the two interconnection networks are better for the following values.

$$t_s = 100, t_w = 1; p = 256$$

$$t_s = 10, t_w = 10; p = 1024.$$

You can use either Store and Forward or Cut Through routing. Do you think that 2-D mesh will ever outperform a Hypercube? If not why? If yes, under what conditions?

Key. For all-to-all broadcast, there is no difference in cost between Store and Forward and Cut-Through routing methods

$$a). t_s = 100, t_w = 1; p = 256$$

$$\text{Hypercube} = t_s \log p + t_w m(p-1) = 100 \cdot 8 + 1 \cdot m \cdot 255 = 255m + 800$$

$$\text{2-D Mesh} = 2t_s (p^{1/2} - 1) + t_w m(p-1) = 2 \cdot 100 \cdot 15 + 1 \cdot m \cdot 255 = 255m + 3000$$

$$b), t_s = 10, t_w = 10; p = 1024.$$

$$\text{Hypercube} = t_s \log p + t_w m(p-1) = 10 \cdot 10 + 10 \cdot m \cdot 1023 = 10230m + 100$$

$$\text{2-D Mesh} = 2t_s (p^{1/2} - 1) + t_w m(p-1) = 2 \cdot 10 \cdot 31 + 1 \cdot m \cdot 1023 = 10230m + 620$$

Thus, for given values, Hypercube is better than 2-D mesh.

If m is large then the difference between Hypercube and 2-D mesh will become negligible (since the effect of t_s is insignificant).

In general Hypercube always has a better performance since, a hypercube has d connections for each processor while 2-D mesh has only 4 connections per processor. If we assume that the cost of building a network is proportional to the number of communication links, Hypercube will be much more expensive. For the two cases above, for (a) with 256 processors, Hypercube will cost ($256 \cdot 8 = 2048$) while 2-D mesh will cost ($4 \cdot 256 = 1024$), and for (b) Hypercube will cost ($10 \cdot 1024 = 10240$) while 2-D mesh will cost ($94 \cdot 1024 = 4096$).

4 (25%). Consider a special case of matrix multiplication where the two $n \times n$ matrices (A and B) are in Lower Triangular form – as shown below (only the elements along the diagonal and below the diagonal are non-zero).

a_{00}					b_{00}				
a_{10}	a_{11}				b_{10}	b_{11}			
a_{20}	a_{21}	a_{22}			b_{20}	b_{21}	b_{22}		
a_{n-10}	a_{n-11}	a_{n-12}		a_{n-1n1}	b_{n-10}	b_{n-11}	b_{n-12}		b_{n-1n1}

Consider distributing the computation on a parallel processing system with p processors ($p < n$) using

- row or column striping,
- cyclic striping (of rows or columns), and
- checkerboard partitioning

Which of these would you think achieves better performance than the others? Present reasons for your answer based on issues like communication cost, and balanced computation among the processors.

Key.

Note that the result matrix will also be a lower-triangular in shape.

For one thing, you should immediately rule out the checkerboard partitioning since, some partitions will have no elements and those processors will have no work to do (thus not efficient use of the available processors).

In order to compare the other two alternatives, let us first start with the result matrix C .

a) row portioning. Consider that each processor computes n/p rows of C . Let us consider assigning n/p rows of A and n/p rows of B to each processor.

Initially examine the case where $n=p$; each processor is assigned a single row.

Consider computing row-0 of the result matrix by processor P_0 . We need to compute only one element and the computation does not require any communication. For P_1 , we need to compute two elements and P_1 must acquire column data from P_0 to compute the second element.

Thus, higher numbered processor perform more computation and require more communication.

Consider a slight variation. We use row striping to distribute A and column striping to distribute B (and row striping of C). Once the load is imbalanced since lower numbered processors need to compute fewer element and higher numbered processors compute more results. For each result we involve increasingly more communication. The only difference is that all the elements of a column needed to compute results reside in a single processor (unlike the case when we used row striping for B matrix resulting in all-to-all communication).

b). Consider using cyclic partitioning. This way we will have more balanced load since processor P_0 computes the elements of row-0, row- p , row- $2p$ etc. Note that the amount of communication is proportional to the amount of computation.

An even better way of balancing load would be a variation of the cyclic striping idea. We alternate between assigning rows in the order of processor numbers and in the reverse order of processor numbers. Thus, we allocate rows $0, 1, \dots, p-1$ to processors P_0, P_1, \dots, P_{p-1} ; allocate rows $p, p+1, \dots, 2p-1$ to processor $P_{p-1}, P_{p-2}, \dots, P_1, P_0$, rows $2p, 2p+1, \dots, 3p-1$ to processors P_0, P_1, \dots, P_{p-1} and so on.