Homework 34

Nicholas Amoscato
naa46@pitt.edu

Josh Frey
jtf15@pitt.edu

November 22, 2013
CS 1510 Algorithm Design

1. **Parallel Problem 17:**

   First, we recall the MIN problem presented in class that outputs the smallest x_i from a list of integers $x_1, \ldots, x_n$. We can easily change this algorithm to find the maximum x_i by simply writing 1 when $x_i \geq x_j$ and 0 otherwise (as opposed to writing 1 when $x_i \leq x_j$). This algorithm runs in constant time with n^2 processors.

   We realize that as long as we have at least n^2 processors, where n is the number of integers we are trying to MAX, this problem can be solved in constant time. This follows from the algorithm in which an $n \times n$ table is constructed.

   Applying this realization to this problem, we claim that we can find the MAX of $n^{1/2}$ numbers in constant time with $n^{1/2} \times n^{1/2} = n$ processors. Therefore, we divide our input of n integers into $n^{1/2}$ segments of size $n^{1/2}$. Assuming that we can find the maximum of each $n^{1/2}$ segment in $T(n^{1/2}, n^{1/2})$, we know that we can find the maximum of these resulting $n^{1/2}$ integers in constant time. Thus, our recurrence relation is defined as follows:

$$T(n,n) \leq T(n^{1/2^1}, n^{1/2^1}) + 1 \quad (1)$$
$$T(n^{1/2^1}, n^{1/2^1}) \leq T(n^{1/2^2}, n^{1/2^2}) + 1 \quad (2)$$
$$\cdots \quad (3)$$
$$T(n^{1/2^{d-1}}, n^{1/2^{d-1}}) \leq T(n^{1/2^d}, n^{1/2^d}) + 1 \quad (4)$$

   Given that we can find the maximum of two integers in constant time, this process continues until there are two integers left to compare. That is,

$$n^{1/2^d} = 2 \quad (5)$$

where d is the number of steps of the recurrence relation illustrated above. Solving for d, we get

$$\log\left(n^{1/2^d}\right) = \log 2 \tag{6}$$

$$\frac{1}{2^d}\log n = 1 \tag{7}$$

$$\log n = 2^d \tag{8}$$

$$\log\log n = d \tag{9}$$

Thus, the recurrence relation will terminate in $\log\log n$ steps. Given that finding the maximum of these recursive results happens in constant time, this algorithm runs in time $O(\log\log n)$.

2. **Parallel Problem 18:**

   We obtain a parallel algorithm that finds the maximum number in a sequence $x_1, \ldots, x_n$ of integers in the range $[1, n]$ in constant time on a CRCW-Priority PRAM with n processors $p_1, \ldots, p_n$ where p_1 is the processor with the lowest identifier (and highest priority). We assume that the integers are stored in an array X of size n.

   Knowing that the integers in X fall within the range $[1, n]$, we create a temporary array T of size n that will keep track of whether or not an integer exists in X. Specifically, if the integer x does exist, $T[x] = 1$; otherwise $T[x] = 0$. We initialize T by assigning each processor an arbitrary index in T and have them write a 0 to their respective index.

   Next, we assign each processor p_i to an array location $X[n-i+1]$. That is, the first processor p_1 is assigned $X[n-1+1] = X[n]$, the second processor p_2 is assigned $X[n-2+1] = X[n-1]$ and so on. The assignment of all of the processors can happen in constant time.

   Then we have each processor p_i write a 1 to $T[X[i]]$. As described above, this designates that the integer $X[i]$ does exist in X. This step can also happen in constant time.

   Finally, for each processor p_i, if $T[i] = 1$ (in other words, if there is an integer $i \in X$), we output i. By the nature of assigning the highest priority processor to the highest index of X, we ensure that only the highest index i where $T[i] = 1$ will be output.

3. **Parallel Problem 19:**

   We obtain a parallel algorithm that finds the maximum number in a sequence $x_1, \ldots, x_n$ of integers in the range $[1, n]$ in constant time on a CRCW-Common PRAM with n processors $p_1, \ldots, p_n$ by following a similar approach as in problem 18. That is, we create a temporary array T of size n that is initialized to 0. Again, assuming that the integers are stored in an array X of size n, we assign each processor p_i to an arbitrary index of X and have them write a 1 to $T[X[i]]$.

As before, we must output the highest index i in which $T[i] = 1$. However, we can not simply rely on the priority PRAM as in problem 18. Instead, we describe an alternative way to find the maximum index on a CRCW-Common PRAM.

We recall a realization that we made in problem 17. That is, we can find the maximum of n numbers in constant time with n^2 processors. However, given that we only have n processors, we can only find the maximum of at most $n^{1/2}$ numbers in constant time. Thus, it seems that we must break this problem into $n^{1/2}$ subproblems.

We segment T into $n^{1/2}$ chunks of $n^{1/2}$ numbers. We realize that if none of the values $T[i \times n^{1/2}], \ldots, T[(i+1) \times n^{1/2} - 1]$ in a given chunk c_j are 1 (if they are all 0), the maximum index is definitely not in c_j. We can "summarize" c_j with a 0. Otherwise, the maximum index might be in this chunk, and we set $c_j = 1$.

This is in fact an informal definition of binary OR. Given that we have $n^{1/2}$ different chunks, we create an array $C = [c_1, \ldots, c_{n^{1/2}}]$ of size $n^{1/2}$ that will store the results of these ORs. Specifically, c_1 will coorespond to the first $n^{1/2}$ numbers of X, c_2 will coorespond to the second $n^{1/2}$ numbers of X, and so on. We assign $n^{1/2}$ processors to each chunk and compute the OR of $n^{1/2}$ numbers in constant time using the algorithm that was presented in class. Each processor p_j writes the result of its OR to its respective index $C[j]$.

When this step has finished, C will contain $n^{1/2}$ ones or zeroes that effectively summarize the $n^{1/2}$ chunks of X. We can now find the maximum index of C in which $c_j = 1$ in constant time with n processors using the MAX algorithm described in problem 17. The resulting index j will tell us which segment the maximum number resides.

Knowing that there are $n^{1/2}$ numbers in each segment, we can run this MAX algorithm once more on $T[j \times n^{1/2}], \ldots, T[(j+1) \times n^{1/2} - 1]$ to determine the maximum index in constant time.