CS1510 Parallel Problems 12, 13, 14, & 16

Rebecca Negley, Sean Myers

November 11, 2011

12. Explain how to modify the all-pairs shortest path algorithmm for a CREW PRAM that was given in class so that it runs in time $O(log^2n)$ on a EREW PRAM with n^3 processors.

    Solution: We need to recreate the D array so that it can handle Exclusive reads. To do this, we create an edge buffer and in parallel, put in i and j, but then in non-parallel, put in m from the previous one. So it will start with all DBuff[i,0,j] being filled. Then we can fill DBuff[i,1,j], then with both 0 and 1 being filled we can fill 2 and 3 in parallel. This makes it log time to re-fill the array (which is important to keep this in loglog time).

    We also need to create a sum buffer to hold the sums so that we don't have to calculate both in parallel, which seems to be an issue. Do the same kind of thing where m is not going to be run in complete parallel.

    function recreateBuffers()
    par for i = 0 to n
    par for j = 0 to n
    par-kinda-for m = 0 to n //Explained above
    DBuff[i,m,j] = D[i,j]
    Sum[i,m,j] = D[i,m]
    Sum[i,m,j] += D[m,j]
    return

    Repeat log n times
    For all i,j,m in parallel recreateBuffers() T[i,m,j] =minDBuff[i,m,j], Sum[i,m,j] D[i,j] = min T[1,1,j] ... T[1,n,j]

    This still runs in log*log time because recreateBuffers takes log time to run, since m

13. Explain how to modify the all-pairs shortest path algorithm for a CREW PRAM that was given in class so that it actually returns the shortest paths (not just their lengths) in time $O(log^2n)$ on a EREW PRAM with n^3 processors.

    Solution: Create an array of paths path[n,n] before the algorithm runs. These paths each hold a list. It is a list of vertices in order to get to the end node. So if to get from vertex 1 to 8 is going from 1 to 2, then 2 to 8, path[1,8] holds a list [1,2,8]. Anyways, when min is sum, this means that the graph needs to be updated. So if(DBuff[i,m,j] ¡ Sum[i,m,j]) replace path[n,n] with path[i,m] + path[m,j]. The paths are first filled with just a list of To do it in parallel, we can use the same strategy as we did above, where we place path[n,n,n] except the last column is a copy (do the recalculation in the recreateBuffers)

14. Explain how to solve the longest common subsequence problem in time $O(log^2n)$ using at most a polynomial number of processors on a CREW PRAM.

Solution: We construct this algorithm in a similar way to the one constructed in the notes for the shortest path algorithm. We have two strings $A_1A_2\cdots A_m$ and $B_1B_2\cdots B_n$ (WLOG $n \geq m$). Let $\mathrm{S}(i_1,i_2,j_1,j_2)$, $1 \leq i_1 \leq i_2 \leq m, 1 \leq j_1 \leq j_2 \leq n$, denote the longest common subsequence of $A_{i_1}\cdots A_{i_2}$ and $B_{j_1}\cdots B_{j_2}$. If the inequalities are not obeyed, define S to be 0. Then, in the end, we want to return $S(1,m,1,n)$. Observe that $\mathrm{S}(i_1,i_2,j_1,j_2)=\max_{i_1\leq i_3\leq i_2, j_1\leq j_3\leq j_2}(\mathrm{S}(i_1,i_3,j_1,j_3)+\mathrm{S}(i_3+1,i_2,j_3+1,j_2))$. We will initially assign all $\mathrm{S}(i_1,i_2,j_1,j_2)$ to 1 if $i_1=i_2$, $j_1=j_2$, and $A_{i_1}=B_{j_1}$ and 0 otherwise. Then, for fixed i_1,i_2,j_1,j_2, we assign n^2 processors to each pair (i_3,j_3). If $i_1 \leq i_3 \leq i_2$ and $j_1 \leq j_2 \leq j_3$, we let $\mathrm{T}(i_1,i_3,i_2,j_1,j_2,j_3)=\max(\mathrm{S}(i_1,i_2,j_1,j_2), \mathrm{S}(i_1,i_3,j_1,j_3)+\mathrm{S}(i_3+1,i_2,j_3+1,j_2))$. Then, we compute our new $\mathrm{S}(i_1,i_2,j_1,j_2)$ by taking $\max_{i_1\leq i_3\leq i_2, j_1\leq j_3\leq j_2}(\mathrm{T}(i_1,i_3,i_2,j_1,j_3,j_2))$. We can calculate this max in $O(log\ n)$ time by first assigning n processors to each fixed i_k and using simple parallel recursion in $log\ n$ time. Then, we take then max of the n values we get using another simple parallel recursion, which again takes $log\ n$ time. We use this method in parallel for each $(i_1,i_2,j_1,j_2) \in [1,m]\times[1,m]\times[1,n]\times[1,n]$. Now, we have an accurate measure for every set of strings A_pA_{p+1} and B_q and every set of strings A_p and B_qB_{q+1} in $O(log\ n)$ time. Doing this for another iteration also gives us an accurate measure for every set of strings where one is up to length 4 and the other is up to length 2 and every set of strings where both are up to length 3. Continuing on, we double our sum of string lengths every time, so we reach our final state in $log(n+m) = O(log\ n)$ iterations. Hence, the whole algorithm takes time $O(log^2 n)$.

16. Design a parallel algorithm that merges two sorted arrays into one sorted array in time $O(1)$ using a polynomial number of processors on a CRCW PRAM.

Solution: [NOTE:Unless otherwise noted, my arrays are indexed from 1 for simplicity.] We seek to merge two sorted arrays $[x_1,\ldots,x_n]$ and $[y_1,\ldots,y_n]$ into $[z_1,\ldots,z_{2n}]$. First create a temporary n-sized array to keep track of which x_i each y_j shoud follow. Consider some fixed y_j. Using $n+1$ processors, we check whether $x_i < y_j \leq x_{i+1}$ for all $0 \leq i \leq n$. Consider $x_0 = -\infty$ and $x_{n+1} = +\infty$. If the inequality is true, that processor writes i to TEMP[j]. Only one processor will write to TEMP[j], specifically the one representing the first location where we can place y_j. We can fill in each TEMP[j] location in parallel if we have $(n+1)*n$ processors. Now create a second n-sized temperory array TEMP2 and a third n-sized array TEMP3. TEMP2 will mark how many y_js come before each x_i, and TEMP3 will mark the first y_j after each x_i. TEMP3 is indexed from 0. Again consider some fixed j. If TEMP[j] $= i$ and TEMP[$j+1$] $= k \neq i$, then we know that $y_1,\ldots,y_j$ all will precede $x_{i+1},\ldots,x_k$ in our final list and y_{j+1} will be the first y value to come after x_k. In parallel, we write j to TEMP2[$i+1$],...,TEMP2[k] to denote that each of these x values must be shifted over j places and write $j+1$ to TEMP3[k]. Each j will then correspond to at most n writes. Also, we must write TEMP[1] to TEMP3[TEMP[1]]. Then,we can fill in TEMP2 and TEMP3 with n^2 processors in time $O(1)$. We did not necessarily assign a value to each location in TEMP3, but we did assign a value to each location that we will use (each x that will directly precede a y). Now, we have a sort of blueprint for merging our lists. All that remains is to merge them according to this plan in time $O(1)$.

Assign one processor to each x_i. Have that processor write x_i to z[i+TEMP2[i]]. Also assign one processor to each y_j. First, that processor should look at TEMP[j] to determine what x value to follow. Then, that processor should look at TEMP2[TEMP[j]] to determine how far that x value was shifted in the new array. Finally, that processor should look at TEMP3[TEMP[j]] to determine the first y value that followed the appropriate x value. The processor finally writes y_j to z[TEMP[j]+TEMP2[TEMP[j]]+1+($j-$TEMP3[TEMP[j]])]. Since this takes constant time, we assign one processor to each y_j and complete this step in constant time. Hence, using a total of $(n+1)*n$ processors (the maximum used at any step), we merge two sorted arrays in constant time. In pseudocode:

```
MERGE(X, Y, n)
    par for j from 1 to n do:
        par for i from 0 to n do:
            if x_i < y_j ≤ x_{i+1} do:
                TEMP[j]=i
    par for j from 1 to n do:
        p =TEMP[j]
        q =TEMP[j + 1]
        if p ≠ q do:
            par for m from p to q do:
                TEMP2[m]=j
            TEMP3[q]=j + 1
    TEMP3[TEMP[1]]=TEMP[1]
    par for i from 1 to n do:
        z[i+TEMP2[i]]=x_i
        z[TEMP[j]+TEMP2[TEMP[j]] + 1 + (j−TEMP3[TEMP[j]])]=y_i
```