Homework 30

Brian Knotten, Brett Schreiber, Brian Falkenstein

November 6, 2018

12

In the CREW algorithm, the concurrent read step is $D[i,j] \leftarrow min(D[i,j], D[i,m] + D[m,j])$, since processor $i = 5$ is reading $D[i = 5, m = 4]$ at the same time processor $j = 4$ is reading $D[m = 5, j = 4]$ (These represent the same memory location). To make the algorithm EREW, make n copies of D into a 3D array D' so that each of the n^3 processors $p_{i,j,k}$ is reading from a different cell in D'.

n^3 processors can make n copies of D (which is size n^2) into a 3D array D' in $\log^2 n$ time.

With n^3 processors, the concurrent read step of the CREW algorithm becomes an exclusive read step: $D[i,j,k] \leftarrow min(D[i,j,k], D[i,m,k] + D[m,j,k])$.

Algorithm 1 EREW $O(log^2(n))$ algorithm for APSP

```
Require: A 2D Array $D'$ (With extra space for a third dimension)
  while $c \leq 1$ do
    if $k < c$ then
      $D'[i,j,k+c] \leftarrow D'[i,j,k]$    ▷ With enough available processors, make $n$ copies of $D$ into a 3D array $D'$
    end if
  end while
  $b \leftarrow \lfloor \frac{c}{2} \rfloor$
  while $b > 1$ do
    $D'[i,j,i] \leftarrow min(D'[i,j,i], D[i+b,k+b,i] + D'[k+b,j,i])$    ▷ Use $k$ to perform $\log n$ associative operations, ending up in the first row of $D$.
    $b \leftarrow \lfloor \frac{c}{2} \rfloor$    ▷ Iteratively half $b$ as the minimum gets pushed further to the left, as in the OR algorithm.
  end while
  if $i == 0$ and $j == 0$ and $k == 0$ then
    return $D'[k = 0]$    ▷ One processor outputs the 2D array containing all shortest path distances.
  end if
```

This algorithm runs in $O(log^2(n))$ time, since making n copies of n^2 entries takes $\log^2(n)$ time, and the EREW min algorithm takes $\log(n)$ time (as all associative algorithms do).

13

The algorithm can be modified as follows to return the actual paths. First, when setting up the 2D array D that contains information for all pairs $(v_i, v_j) \in G$, instead of storing the distance between v_i and v_j store a pair: the first element is a list of vertices representing a path from v_i to v_j, and the second element is the distance between v_i and v_j.

When setting up D initially, $D[i,j]$ is initialized as follows: if $i = j$, then $D[i,j] \leftarrow ([v_i], 0)$. Else if there is an edge between v_i and v_j, then $D[i,j] \leftarrow ([v_i, v_j], edge\_weight(v_i, v_j))$. Otherwise, $D[i,j] \leftarrow ([], \infty)$.

When $D[i,j]$ is updated to be $D[i,m] + D[m,j]$, then $D[i,j] \leftarrow (fst(D[i,m]) + +fst(D[m,j]), snd(D[i,m]) + snd(D[m,j]))$, where fst returns the first element of a tuple, snd returns the second element of a tuple, and $a + +b$ represents list a concatenated with list b.

Algorithm 2 EREW $O(log^2(n))$ algorithm for APSP that returns the paths

```
Require: A 2D Array $D'$ (With extra space for a third dimension)
  while $c \leq 1$ do
    if $k < c$ then
      $D'[i, j, k + c] \leftarrow D'[i, j, k]$    ▷ With enough available processors, make $n$ copies of $D$ into a 3D array $D'$
    end if
  end while
  $b \leftarrow \lfloor \frac{c}{2} \rfloor$
  while $b > 1$ do
    $D'[i, j, i] \leftarrow min(D'[i, j, i], ((fstD[i+b, k+b, i]) ++ (fstD'[k+b, j, i]), (sndD'[i+b, k+b, i]) + snd(D'[k+b, j, i])))$
  ▷ See algorithm above for details.
    $b \leftarrow \lfloor \frac{c}{2} \rfloor$    ▷ Iteratively half $b$ as the minimum gets pushed further to the left, as in the OR algorithm.
  end while
  if $i == 0$ and $j == 0$ and $k == 0$ then
    return $D'[k = 0]$    ▷ One processor outputs the 2D array containing all shortest paths and distances.
  end if
```

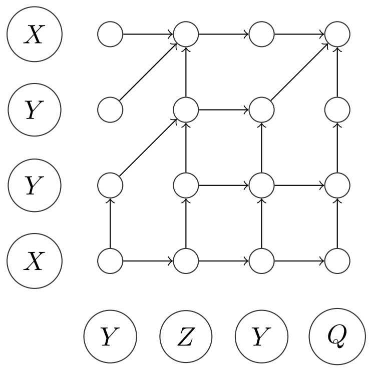
As above, this algorithm runs in $O(\log^2(n))$ time, since the only modifications (concatenating two arrays, and accessig/constructing members of a tuple) require constant time.

14

The main idea of this algorithm is to convert the input of longest common subsequence (referred to as LCS) to the input for all-pairs shortest path (APSP). Because we showed in class a CREW algorithm for APSP that runs in time log^2n with a polynomial number of processors, if we can do the input conversion in log^2n time, we can pass the converted input into our APSP algorithm to get our output in log^2n time.
Because the CREW algorithm fo APSP has already been extensively defined, this algorithm will focus on the input conversion. The constructed graph will have $N * M = n^2$ nodes, where N and M are the lengths of the input strings to LCS, where node n_{ij} corresponds to the $i'th$ letter in A and the $j'th$ letter in B (if the input strings to LCS are A and B). Edges are created according to the following rules:

- If $A[i] = B[j]$, then add an edge from n_{ij} to n_{i+1j+1} with weight -1
- If $A[i]! = B[j]$ then add edges from n_{ij} to n_{i+1j} and n_{ij+1} with weights 1

Thus the minimum weight path from node n_{00} to n_{NM} will have taken the maximum number of 'diagonal' edges (that is, from n_{ij} to n_{i+1j+1}) as these edges have negative weight and will contribute to the min distance.
A simple example of this conversion is provided, where the input strings to LCS are $A = XYYX$ and $B = YZYQ$:



In this picture, the edges with weight -1 are the diagonals, while all horizontal/vertical edges have weight 1. Its important to note that this constructs a graph with no cycles of negative aggregate weight. This is because in any cycle, there will be more positive weighted edges than negatives. With n^2 processors, where each processor is assigned an i, j, an algorithm is given:

Algorithm 3 CRCW O(1) algorithm for converting input from LCS to input for APSP

```
Require: 2 Strings A and B of size $N$ and $M$, a processor $p_{ij}$ and an adjacency matrix Adj of size $NM \times NM$
  if A[i] ==B[j] then
    $Adj[i+1][j+1] \leftarrow -1$    ▷ Negative edge weight, because we found a candidate character for the LCS
  else
    $Adj[i+1][j] \leftarrow 1$    ▷ Positive edge weight, not part of the LCS
    $Adj[i][j+1] \leftarrow 1$
  end if
```

This construction takes constant time, as each processor simply compares 2 characters and makes either 1 or 2 writes. Also note that because i, j are unique for each processor, this algorithm is exclusive write.
Once we have the adjacency matrix constructed, we can simply pass it into the defined algorithm for APSP.
Since the input transformation takes constant time, and time for APSP with $p = O(n)$ is $log^2 n$, our algorithm also runs in time $log^2 n$ with $p = O(n)$ processors.