Homework 31

Brian Knotten, Brett Schreiber, Brian Falkenstein

November 8, 2018

2

To give evidence that finding a fast efficient parallel algorithm (i.e. a $O(log^k(n))$ algorithm for some k with a poly number of processors) for N is at least as hard as finding a fast efficient parallel algorithm for the BFV problem, you could present a parallel algorithm for the BFV problem that is $O(log^k(n))$ for some k with a poly number of processors that is only missing code for N.
The algorithm should convert the input from the BFV problem to the input to the N problem in at most poly-log time, run the missing code for N using the converted input, and convert the output of the N problem to the output of the BFV in at most poly-log time.
Further, the missing code for N should give a correct output given the input iff there is a correct output for the BFV problem given the input.
Given this parallel algorithm for the BFVmissing the code for N, if you were to find parallel code for N that runs in poly-log time with a poly number of processors then you would immediately have a poly-log algorithm for the BFV problem with a poly number of processors by "plugging in" the code for N into the parallel code for BFV.
Because the code we are plugging in is poly-log with a poly number of processors, our input/output transformations are also poly-log with a poly number of processors, and the code gives a correct output for BFV iff the code for N gives a correct output, the newly-constructed algorithm correctly solves the BFV problem in poly-log time using a poly number of processors.
Therefore finding a fast efficient parallel algorithm for problem N is at least as hard of a problem as finding a fast efficient parallel algorithm for the Boolean Formula Value problem.

16

Given two lists of increasing items $a_0 ... a_{n-1}$ and $b_1 ... b_{n-1}$, the following must be true for the merged increasing list M of size $2n$:

1. The first element of M must be the minimum of a_0 and b_0
2. If the first element of M is a_0, then the second element must be the minimum of a_1 and b_0.
   If the first element of M is b_0, then the second element must be the minimum of a_0 and b_1.
3. If the second element of M is a_0, then the first element must have been b_0, and so the third element must be the minimum of a_1 and b_1.
   If the second element of M is b_0, then the first element must have been a_0, and so the third element must be the minimum of a_1 and b_1 as above.
   If the second element of M is a_1, then the first element must have been a_0, and so the third element must be the minimum of a_2 and b_0.
   If the second element of M is b_1, then the first element must have been b_0, and so the third element must be the minimum of a_0 and b_2.

The third row could be any of $a_0, a_1, a_2, b_0, b_1, b_2$ in the general case. But the reasoning above shows that there are only three possibilities for what the third entry could be. In fact, the other two minimums will occupy the first two entries of M. So it can be said that the first entry of M is the maximum of $min(a_0, b_0)$, the second entry of M is the maximum of $min(a_1, b_0), min(a_0, b_1)$, and the third entry of M is the maximum of $min(a_2, b_0), min(a_1, b_1), min(a_0, b_2)$. So in general, the ith entry of M is the maximum of $min(a_{i-j}, b_j)$for all j from 1 to i.

The outline for the algorithm is as follows: use n^2 processors to calculate all pairwise minimums in constant time, and store the results in a table A such that $A[i][j] = min(a_{i-j}, b_j)$ (unless $j > i$, in which case $A[i][j] = -\infty$, which will never be the maximum).

Note also that when i or j exceeds n, this implies there are either no more as or no more bs to consider, and so the rest of M must be filled with either bs or as respectively. And so in these cases no minimum is calculated; the entry in the other list is returned. Consider when $i = 2n$. That implies that the last row is the maximum of $2n$ different entries: $a_0, a_1, ...a_{n-1}, b_0, b_1, ...b_{n-1}$. This makes sense intuitively, since the last entry in M must have the maximum value of all of a and b.

After A gets populated in constant time by n^2 processors, $M[i]$ must be filled in with the maximum entry $A[i][j]$ for all j. Each row can do this calculation in constant time with n^2 processors as discussed in class. Since n rows need to do this, n^3 total processors are needed. After a constant number of steps, M will be fully populated with the merged increasing elements of a and b, and so the algorithm is finished.

Algorithm 1 CRCW Merge two sorted length n lists in $O(1)$

```
Require: Increasing lists $a_1...a_n$, $b_1...b_n$
  if $j < i$ then        ▷ First, populate each row $i$ of $A$ with the candidate entries that could be in position $i$ of the merged list.
      if $i > n \wedge j > n$ then
          $A[i][j] \leftarrow a_{i-j}$
      else if $i > n \wedge j < n$ then
          $A[i][j] \leftarrow b_j$
      else
          $A[i][j] \leftarrow min(a_{i-j}, b_j)$
      end if
  else
      $A[i][j] \leftarrow -\infty$
  end if
```

17

Break the input into $\sqrt{n}$ chunks of size $\sqrt{n}$ and recursively solve the problem in parallel with $\sqrt{n}$ processors assigned to each chunk. We have n processors, so the $\sqrt{n}$ max values returned by the chunks can be "maxed" in constant time using our n processors (see hint).

The recurrence relation for this algorithm is $T(n, n) = T(\sqrt{n}, \sqrt{n}) + 1$.
Without loss of generality, let $n = 2^k$ for some k. At each step we are breaking the input into $\sqrt{n}$ size chunks i.e. we are halving the exponent k. Therefore we can only take the square root $O(log(k))$ times before k drops to 1 or lower (and n drops to 2 or lower). $n = 2^k$, so $k = log(n)$ and the number of square roots taken is $O(log(k) = O(log(log(n)))$. Each of the $\sqrt{n}$ chunks returns a single max value and we have n processors, so we have the required $\sqrt{n}^2 = n$ processors to compute the max of those $\sqrt{n}$ numbers in constant time. Therefore the runtime of this algorithm is $O(log(log(n))$ with n processors on a CRCW Common PRAM.