CS1510 Reduction Problems 21 & 25, Parallel Problems 4 & 5

Rebecca Negley, Sean Myers

November 4, 2011

21. The input to the fixed Hamiltonian path problem is an undirected graph G and two vertices x and y in G. The problem is to determine if there is a simple path between x and y in G that spans all the vertices in G. A path is simple if it doesn't include any vertex more than once. Show that if the fixed Hamiltonian path problem has a polynomial time algorithm then the Hamiltonian cycle problem has a polynomial time algorithm.

    Solution: We must reduce the Hamiltonian cycle problem to to path problem. To do this, we choose the first edge in our graph j. Set x to be the first vertex in j and set y to be the second vertex in j.

    We know that if this problem is a hamiltonian path, then that edge cannot be chosen. It can't be chosen because the Hamiltonian path cannot choose the same vertex over, and must span the entire graph. We know that the path must end at y, and we know it must start at x. Therefore, if the path problem were to choose edge j, it would not be able to choose y again, and if it must end at y, it means that the path that was chosen was wrong, or there are only two vertices in the graph, which means that it is inherently a hamiltonian cycle.

    So we only have to run the Hamiltonian path problem once, and the translating the input takes O(1) time.

    The final algorithm looks something like this in pseudocode:

```
HamCycle(G):
    Edge j = G.getEdge(0)
    Vertex x = j.firstVert
    Vertex y = j.secondVert
    return HamPath(G, x, y)
```

25. We consider a generalization of the fox, goose, and bag of beans puzzle. The input is a graph G and integer k. The vertices of G are objects that the farmer has to transport over the river, there is an edge between two objects if they can not be left alone together on the same side of the river. The goal is to determine if a boat of size k is sufficient to safely transport the objects across the river. The size of the boat is the number of objects that the farmer can haul in the boat. Show that this problem is NP-hard using a reduction from an NP-hard program from class or homework.

    Solution: We will reduce the independent set decision problem to the generalized fox goose beans problem. We are given an input undirected graph G and an integer j in the independent set decision problem. We want to decide if G contains an independent set of size j. First, observe that any set of objects that the farmer leaves on either side of the river must be an independent set. If not, the farmer must be leaving two items with an edge between them that hence cannot be left alone together, which is a contradiction. Construct G' where each vertex v in G is replaced by two vertices v and v' with an edge connecting them in G'. For any two vertices v and w in G with an edge between them, create

four edges in G': (v, w), (v', w), (v, w'), and (v', w'). Then, v, v', w, and w' form a 4-clique, so only one of the vertices may be contained in an independent set.

Observe that G' contains an independent set of size j if and only if G contains an independent set of size j. For any independent set S in G, we can choose an independent set S' of the same size in G' by arbitrarily picking, for each vertex in S, one of the vertices associated with that vertex in G'. Two vertices in G' have an edge between them if and only if their corresponding vertices in G had an edge between them or the two vertices correspond to the same vertex in G. Hence, none of the vertices in S' will be adjacent because all vertices in S were nonadjacent and unique. Also, for any independent set S' in G', we can choose an independent set S of G by picking the associated vertex in G for each vertex in S'. No two vertices in S' can correlate to the same vertex or to adjacent vertices, or else they would be connected. Hence, we have shown that G' contains an independent set of size j if and only if G does. Also, observe that G' contains an independent set of size j if and only if G' contains two unique independent sets (that share no elements) of size j. The if part is trivial, but we must prove the only if. Suppose G' contains an independent set S of size j. Then, we can form another independent set S' of size j by choosing, for each vertex in S', the opposite vertex corresponding to the same vertex in G. Clearly, this will be an independent set because no two vertices can correspond to the same edge in G or to adjacent edges in G, so they cannot be adjacent in G'.

Now we input G' and size $k = 2n - j$ into the generalized fox, goose, bag of beans problem, where n is the number of vertices in G (so $2n$ is the number of vertices in G'). This problem returns true if and only if there is an independent set in G of size j, so we return its output as the output of the problem. To see this, suppose a boat of size $2n-j$ was sufficient for safe transportation. Then, at least j objects were left on the initial side of the river during the farmer's first trip. These j objects must form an independent set in G' as stated above, so there must be an independent set of size j in G. Suppose an independent set of size j exists in G. Then, two distinct independent sets of size j exist in G'. The farmer can leave one of these on the initial shore and transport the rest of the objects across the river. He then leaves the second independent set on the opposite shore and crosses back with the remaining $2n-2j$ objects. He picks up the j objects left on the initial side and returns to the opposite shore, where he now has all of his objects. Hence, the farmer can safely transport the items, so we have shown that the output is correct. We created G' in quadratic time (because number of edges is at most quadratic), so if there exists a polynomial time algorithm for the generealized fox goose beans problem, there exists a polynomial time algorithm for the independent set decision problem. Since the independent set decision problem is NP-hard, the generalized fox goose beans problem must be NP-hard.

4. Design a parallel algorithm for the parallel prefix problem that runs in $O(log\ n)$ with $n/log\ n$ processors on a EREW PRAM.

Solution: We give a divide and conquer algorithm to solve this problem. The first thing we need to do is separate the input integers. We will put them into the two groups based on their position in the input. For example, x_1, the first input will be put into the 'odd' column.

So put all the odd indices for the first $P/2$ processors, and then give the remaining to $P/2$ processors. When going back up, each processor will have to sum the individual elements on both sides, in criss-cross order. So let's say you get elements x_1, x_2, x_3, x_4 back from the previous two splits. To get the correct output we would take individual processors to do: $x_1, x_1 + x_2, x_2 + x_3, x + 3 + x_4$. Each processor would have to do two operations to make sure that there is only exclusive reads, but nonetheless, since these are all runing in parallel, this is O(1) time. The main amount of time is T(n,p)= log(n) + 1. When we input that into the efficiency formula, we see that S(n)/(P*T(n,n/log(n))). That is equivalent to n/(n/log(n)*log(n)) which is equal to 1. So this is an efficient, log(n) algorithm.

5. Give an algorithm that given an integer n computes $n!$ in time $O(log\ n)$ on an EREW PRAM with n processors. Make the unrealistic assumption that a word of memory can store arbitarily large integers.

   Solution: Observe $n! = \prod_{i=1}^{n} i$. We can use a divide and conquer approach to parallelize this algorithm. Half of the processors can be used to calculate $\prod_{i=1}^{n/2} i$, and the other half can be used to calculate $\prod_{i=n/2+1}^{n} i$. Since multiplication is associative, it does not matter in what order things are multiplied together. We then multiply these two halves together to get our final product. We use this method to build a recursive tree. At the bottom, each of the n processors gets assigned to one integer in $\{1, \ldots, n\}$. In the first step, each processor just outputs its input integer. Then, in each following step every other processor from the preceding step will take as input its output from the last step and the output of the next processor from the last step. It will multiply these two integers together and return their product. This takes 1 times step. This continues at each level of the tree ($log\ n$ levels) until the final product is returned. Hence, this algorithm runs in time $O(log\ n)$. Its efficiency is $E(n,n) = \frac{n}{n*log\ n} = \frac{1}{log\ n}$.