CS1510 Reduction Problems 17, 18, & 22, Parallel Problem 1

Rebecca Negley, Sean Myers

October 31, 2011

19. Show by reduction that if the decision version of the SAT-CNF problem has a polynomial time algorithm
then the decision version of the 3-coloring problem has a polynomial time algorithm

    Solution:

20. In the dominating set problem the input is an undirected graph G, the problem is to find the smallest dominating set in G. A dominating set is a collection S of vertices with the property that every vertex v in G is either in S, or there is an edge between a vertex in S and v. Show that the dominating set problem is NP -hard using a reduction from the vertex cover problem

    Solution:

23. In the disjoint paths problem the input is a directed graph G and pairs $(s_1, t_1), ..., (s_k, t_k)$ of vertices. The problem is to determine if there exist a collection of vertex disjoint paths between the pairs of vertices (from each s_i to each t_i). Show that this problem is NP -hard by a reduction from the 3SAT problem. Note that this problem is not easy.

    Solution: In order to do this, we must take formula F from 3SAT and convert it into an input for disjoint paths in polynomial time and where the output of 3SAT is 1 if and only if the output to disjoint paths is 1.

    To start off, we create x starting vertices, one for each variable in formula F. We will then add 2 vertices to each starting vertex, representing the negated form and the affirmative form of the variable. Each variable and its negation will equal a collection of vertices that it is currently at (will explain more on this later), which will be at most 2, and at least 1. For now, each starts at 1.

    For each clause, a "gadget" will be created. A gadget is just a clause broken down into path form in such a way that the properties of the clause are reflected in the gadget, so that way it forms a 1 to 1 relational output among the two algorithms. These gadgets will not be made in parallel, but in pipeline. For each variable that is input to the gadget, any later clause that needs to connect that clause, will connect it from the gadget, not the original vertex. At the end of all the clauses, the last pipe will be the end node t_i which will connect the original variable and its negation from the last gadgets that use those variables. Refer to this figure 1 for a pictoral demonstration.

    How do we create a gadget from a clause? Well, we must first see what a clause is made of. The clause is made up of three literals, all or'ed together. What the OR means is that 1 or 2 or three of those literals can equate to true, and the clause will be true. An equivalent way of saying this, is that if 2 or less of the negated literals of the clause are chosen, then the clause will equate to true. So if we are given a clause of (x v y v z), we would create two vertices to connect (x, y, z) to both of these.

What does this mean for the disjoint paths problem? It means that there can only be two paths can be created, and it needs to choose another variable path to traverse. Just like ins 3-SAT, the only way to satisfy the clause is to choose two or less variables, the same goes for the path problem, it can only choose two paths in a given gadget(clause). Back to the example, the next time we see an x, y or z, we take the two vertices that were produced in the previous gadget made, and feed those as inputs into the new gadget. So at each Gadget, we need to update the collection of vertices that input variables are at, and use those as inputs into the next gadget. This process continues until we run out of clauses, and then we create the last vertices t_i for each literal of a variable, and then connect all the updated variable instances into their respective t_i. This path will then output the correct ouput to the 3SAT problem.

An algorithm would look something like this:

```
3SAT(formula F):
    Graph G
    List¡SomeDataStructureWithANameAndACollection¿ vars
    for each(letter in F):
        G.add(new Vertex(letter+"start"))
        G.addVertex(new Vertex(letter)).addEdgeFrom(letter+"start")
        G.addVertex(new Vertex("negated" letter)).addEdgeFrom(letter+"start")
        vars.add(new DC(letter, G.getVertex(letter))
        vars.add(new DC("negated"+letter, G.getVertex("negated+letter))

    for each(clause in F):
        G.add(new Gadget(clause))
        foreach(literal in clause):
            vars.getNegatedform.updateCollection(g.getGadget(clause))

    for each(letter in F):
        G.add(new Vertex("endNode"+letter)
        G.addEdges(vars.get(Letter),"endNode"+letter)
        G.addEdges(vars.get("negated"+letter),"endNode"+letter)
    return path(G)
```

2. You know that lots of famous computer scientists have tried to find a fast efficient parallel algorithm for the following Boolean Formula Value Problem:
INPUT: A Boolean formula F and a truth assignment A of the variables in F .
OUTPUT: 1 if A makes F true, and 0 otherwise.
Moreover, most computer scientists believe that there is no fast efficient parallel algorithm for the Boolean Value Problem. You want to find a fast efficient parallel algorithm for some new problem N. After much effort you can not find a fast efficient parallel algorithm for N, nor a proof that N does not have a fast efficient parallel algorithm. How could you give evidence that finding a fast efficient parallel algorithm for N is at least as hard of a problem as finding a fast efficient parallel algorithm for Boolean Formula Value problem? Be as specific as possible, and explain how convincing the evidence is.

Solution: In order to show that is just as hard to find a fast efficient parallel algorithm for N, we must reduce Boolean Formula Value Problem to N and have the conversion take less than or equal to poly-log time $O(log_k(n))$. A reduction algorithm would look something like this:

```
boolValue(n,p):
    Do some kind of conversion to make it so that the output of Boolean formula is 1 if and only if the output of $N$ i
```

Can use the p processors to do this in parallel time if it is more efficient to do so)
return N(newInput, p);

3. Consider the problem of taking as input an integer n and an integer x, and creating an array A of n integers, where each entry of A is equal to x.

   - Give an algorithm runs in time O(log n) on a EREW PRAM using n processors. What is the efficiency of this algorithm?
   - Give an algorithm that runs in time O(log n) on a EREW PRAM using n= log n processors. What
     is the efficiency of this algorithm?
   - Give an algorithm that runs in time O(1) on a CRCW PRAM using n processors. What is the efficiency of this algorithm

   Solution: