18 April 2011

CSE P 506 – Assignment 2

1. **[5 pts] Mutual Exclusion with Dekker's Lock**
   In lecture you have seen Dekker's Lock (mutual exclusion protocol) for two processors. Dekker's Lock provides some good properties for concurrent programs such as mutual exclusion, freedom from deadlock and freedom from starvation. You can see the algorithm explained with pseudocode at http://en.wikipedia.org/wiki/Dekker%27s_algorithm.
   In this question you are asked to improve this algorithm to work with three processors.
   a. **[2 pts]** Give the algorithm in pseudocode (as in the example) for each processor.
   b. **[1 pts]** Prove that your algorithm provides mutual exclusion
   c. **[1 pts]** Prove that your algorithm provides freedom from deadlock
   d. **[1 pts]** Prove that your algorithm provides freedom from starvation.
   (Hint: don't use the same text in Wikipedia ☺)

2. **[10 pts] Parallel Knapsack**
   Knapsack problem is a combinatorial optimization problem. Given a knapsack with capacity W and n items (numbered 0…n-1), each with weight w_i and profit v_i, the goal is to fill the knapsack with a subset of items that maximizes profit without exceeding the capacity. The naïve approach of enumerating all subsets of items is exponential and obviously not acceptable. Luckily, this problem can be solved using a technique called *dynamic programming.*

   The dynamic programming strategy is applicable if the problem under question has the *optimal substructure* property: the optimal solution to a problem can be obtained by extending optimal solutions to sub-problems. In many cases, the sub-problems are overlapping requiring one to solve the same sub-problem repeatedly. As such, dynamic programming involves remembering the solutions to sub-problems in a table so that each sub-problem is solve at most once.

   Now we will show that the knapsack problem has the optimal substructure property. Let us create a table with W+1 columns and n rows with the following invariant:
   "Table[i, w]'s value is either unset (not evaluated yet) or it represents the maximum profit that can be collected when filling a knapsack of weight w while only using items 0 to i-1."
   This invariant guarantees that the bottom row represents the optimal solution for weight = w (since all items are considered) and Table[n-1,W] is the solution we seek.

The problem of determining the Table[i+1,w] can be phrased in terms of sub-problems as follows. There are two cases for the optimal solution of fitting items 0…i into the knapsack of weight w – the solution either contains item i or it does not. In the former case, the solution involves optimally filling the knapsack of weight w-w_i with items 0…i-1. In the latter case, the solution involves optimally filling the knapsack of weight w with items 0…i-1. Together with edge conditions, the table can be filled with the following recurrence relation:

- Table[0, w] = 0 (if you take no item, for all weights you gain nothing).
- Table[i, 0] = 0 (if you cannot carry an item, for all items you gain nothing.)
- Table[i+1, w] = Table[i, w], if $w_i > w$ (item i weights more than the current limit, you cannot include it to the solution).
- Table[i+1, w] = max(Table[i,w], v_i + Table[i, w- w_i]), if $w_i <= w$

More information about knapsack and dynamic programming is available at
http://en.wikipedia.org/wiki/Knapsack_problem.
http://en.wikipedia.org/wiki/Dynamic_programming.

Your task in this question is to parallelize the knapsack solution described above. You need to first implement the sequential version so that you can report your speedup. You can use a version from the Internet for sequential implementation as long as it uses the dynamic programming mentioned above and you mention this in your report.

a. **[3 pts]** Describe the parallel algorithm. Argue its correctness and efficiency.
b. **[0 pts]** (Prerequisite for 'c') Implement the sequential solution in C#.
c. **[7 pts]** Implement your parallel algorithm in C# and demonstrate speedup over the sequential version for a knapsack of capacity 10000 and 1000 items (table size of ~10 M). Each item should have a randomized weight and profit in the range [5..50].
d. **[3 pts, bonus]** Longest common subsequence (LCS) problem is similar (a little harder to parallelize) to Knapsack. It finds the longest common subsequence between two strings. More information is located at: http://en.wikipedia.org/wiki/Longest_common_subsequence_problem. Implement both a sequential and parallel version of this algorithm in C#, report your speedup with 95% confidence level for random experiments and show that you can get speedup. (For sequential version, again you can use an already existing solution as long as you mention it)

Some important information

- Normally the solution for Knapsack reports both the maximum gain and the set of items that would guarantee this gain. Here, we only ask you for the gain alone.
- Try to use the same data structures in both the sequential and parallel versions as much as possible. For example, using an ArrayList to store

items in sequential version and an Array in the parallel version might affect your speedup. At the least, you should reuse the Table and Item data structures in both versions.

- For each experiment the random data should be generated once and feed to both sequential and parallel algorithm (as in Assignment 1), so that they will do the same computation and a comparison would be logical.
- It might be a good idea to compare the result coming from the sequential algorithm with the result coming from parallel algorithm to find some possible bugs.