0/1 Knapsack Parallel Implementation

Shubhanshu Yadav
School of Engineering and Applied Sciences
Columbia University
sy2511@columbia.edu

Vinti
School of Engineering and Applied Sciences
Columbia University
vv2236@columbia.edu

Abstract
The goal of this project was to implement a parallel multiplace 0/1 Knapsack algorithm and compare it with a sequential dynamic programming algorithm and a single place multithreaded dynamic programming implementation. All the algorithms were programmed in X10 and then tested with large data sets. The results gathered were used for analysis and comparison of the three implementations and drawing conclusions concerning the performance of the three implementations.

1. Introduction
The 0/1 knapsack problem is an optimization problem. Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less that or equal to a given limit and the total value is as large as possible. This problem often arises in resource allocation.

There are several existing algorithms to solve this problem. The most common algorithms are Brute Force, Backtracking, Best-First Branch and Bound and Dynamic Programming. In this report the focus will be on dynamic programming and its parallel implementation. This work is inspired by the paper of Jianhua Lin and James A. Storer[1] who presented a parallel multiplace algorithm for this problem. This time consumed is of order O(mc/n +c.logn) where m is the number of objects, c is the knapsack capacity and n is the number of places. Multiplace implementation is effective when m is large and c is relatively small. Also, due to communication cost, the factor of logn slows down. This multiplace algorithm was compared with other multiplace algorithms (Lee, Shragowitz and Sahni[2] and Chen, Chern and Jang[3].

Section 2 describes the 0/1 knapsack problem and the sequential dynamic programming approach to solve it. Section 3 reviews Lee's algorithm and Section 4 reviews Chen's algorithm and their performance. Section 5 presents the parallel algorithm we implemented. Section 6 discusses the results and performance of our parallel implementation on different data sets.

2. Knapsack problem and dynamic programming
As stated in the introduction, the formal definition of knapsack problem is as follows:

If the knapsack capacity is c, and the total number of elements is m, then given a set A of these m objects, each with profit p_i and weight w_i , we wish to determine the subset T of A such that

$$\text{Maximize } \Sigma p_i x_i$$

$$\text{Subject to } \Sigma w_i x_i \leq c$$

Where x_i is either 0 or 1. Using dynamic programming, the entire knapsack problem is split into subproblems and each subproblem is solved separately. Larger subproblems can then be solved using these subproblems.

Decomposing the problem into smaller subproblems: We can use the following function to find the solution.

$$0 \leq j \leq c \text{ and } 0 \leq k \leq m$$

$$f_k(j) = \max\{f_{k-1}(j), f_{k-1}(j-w_k) + p_k\}$$

In the above equation, $f_k(j)$ is the maximum profit of the knapsack at capacity 0 if the first j objects are included. The value of $f_k(0)$ will be 0 for all k, since at capacity 0 the knapsack can not hold any objects. The profits possible when the k^{th} object is calculated is called the profit vector, and equivalent to f_k .

The optimal solution to the knapsack problem can be obtained by calculating $f_0, f_1, f_2, \ldots, f_m$ until $f_m(c)$is computed for the correct answer. This allows us to build up a table of values to be accessed in each successive stage of computation. In other words, for each possible object, we consider whether to include it at each possible capacity (from 0 to the maximum capacity). We store the maximum profit we could reach for that object at that capacity in a table. The calculation continues on to determine whether including the next item is more profitable than not including it. The columns in the table are the profit vectors for the objects. Algorithm1 is the algorithm for dynamic programming.

Algorithm 1: The sequential dynamic programming algorithm

KNAP (p,w,m,c)

```
{   for (w in 0 to c) F0(w)=0;
    for (i in 1 to m)
        for (w in 0 to c)
        if (wi ≤ w)  Fi(w) = max{Fk-1(w),Fk-1(w-wk)+ pk}
        else Fi(w)= Fk-1(w);
    return Fm(c)
}
```

3. Review of Lee's Parallel Algorithm
This algorithm is designed for a hypercube network. In addition, it utilizes concepts for solving the knapsack such as merging lists, dividing problem into sub-problems, and using group of solvers and levels of calculations to combine the results.

The algorithm has five main steps to calculate the results of a 0-1 Knapsack. Assume n solvers are available and we have m objects. Our knapsack has a maximum capacity of c.

1. The original problem is divided into n sub-problems, and the sub-problems are assigned to the solvers.
2. Each solver creates a profit vector for capacity c, given the objects in the assigned sub-problems.
3. The solvers distribute their profit vectors and combine them to get profit and history vectors for the whole problem.

4. The *n* solvers trace back the combining history to get a value, which is the optimal solution to the original problem.
5. Each processor traces back its dynamic programming history to get an optimal solution vector.

The difficulty here is how to combine the solutions of the sub-problems.

Let KNAP(X,c) and KNAP(Y,c) be sub problems of the KNAP(A,c) where XUY = A and X∩Y = ϕ. Let $F_X(c)$ and $F_y(c)$ be the optimal profit vectors for the two problems respectively. Then, $F_A(c)$ is the optimal profit of KNAP(XUY,c) where

$$F_A(w) = max(F_x(j)+ F_y(w-j)) \text{ , for } 0 \le w \le c \text{ and } 0 \le j \le w$$

The above combination process requires c^2 computations. Hence when the knapsack size c is much large compared to m, then this algorithm becomes very slow. On implementing this algorithm, the running time was found to be worse than the sequential dynamic programming algorithm even when the number of places *n* was increased.

4. Review of Chen's Parallel algorithm

The parallel solution proposed in Pipeline Architectures for Dynamic Programming Algorithms (Chen, Chern, & Jang, 1990) for the 0-1 Knapsack Problem also takes a dynamic programming approach. However, instead of considering a hypercube of processors, this paper presents a solution that is computed in a pipeline-like fashion.

In this algorithm, the processors pass data to their neighbor to the right, and receive data from their left neighbor. Computed data flows through the ring of processors like oil a pipeline. In the sequential dynamic programming algorithm, data dependency occurs only within adjacent stages and the same stage. Hence the calculation for $F_k(w)$ depends on itself and $F_{k-1}(w)$.
Creating a linked list representing a ring network of processors can do the implementation of this algorithm, in which a processors left neighbor is represented by the previous processor in the list and the right neighbor is the next processor in the list. By connecting the first and last processor, a ring is formed.

When a knapsack problem arrives, the server sends the knapsack definition to each processor. The first processor initiates calculation. In pipeline implementation, the first solver would calculate the profits for the first object and send this information to its right neighbor and weight for new data from its left neighbor. The right neighbor would begin the calculation of data for the next object. When it finishes, it would wait for more data from its left neighbor and pass on its calculated data to the right neighbor. In this way, the processors would calculate the values for each object, pass the result to the next processor until the final object has been calculated. When the final object is calculated, the processor runs the backtracking algorithm and returns the results.

This algorithm can allow multiple different knapsacks to be calculated in parallel. For example, once the second processor has begun processing the data for the first knapsack, the first processor could receive data on a new problem and begin processing that data. Hence the processor would be solving different knapsacks at the same time.

However, we are focused on improving the parallel processing of a single knapsack and in this algorithm, if only one knapsack is being calculated then no work is done in parallel as each processor will wait until the previous processor is finished to begin its calculations. A distributed implementation does not have any benefits over a single processor implementation for solving a single knapsack.

5. Review of Lin and Storer's Algorithm

The algorithm that we have used in our final implementation is the one proposed by Lin and Storer[1]. This algorithm runs in the time O(*mlogc*) on a hypercube of *c* processors. For *n* places implementation, it would take O(*mlogn.n/c*). It has a speedup of O(c^2/(*logn.n)*). Hence to get significant speedup, it is important to have a large *n* that is comparable to the value of *c*. Hence when *n=c*, even if c is very large, the speedup and efficiency will be very good. The main idea of the algorithm is to compute successively the optimal profit vector $F_j(c)$ in parallel. The elements of $F_j(c)$ depends only on those of $F_{j-1}(c)$ and more importantly, the computation of each element is independent of one another, which permits efficient parallelism.

Description of the algorithm:

Given a knapsack problem KNAP(A,*c*), let $F_j(c)=(f_j(0), f_j(1),., f_j(c))$ be the optimal profit vector for KNAP(A,*c*) where A = $\{a_1, a_2, a_3,.., a_j\}$.

$$f_j(y)= f_{j-1}(y), \qquad 0 \le y \le w$$

$$f_j(y)=\max\{f_{j-1}(y), f_{j-1}(y-w_j)+ p_j\} \ , \qquad w_j \le y \le c$$

$f_j(y)$ depends on at most two values of F_{j-1}. Thus, if we assign one place for computing each $f_j(y)$ and all the places do it in parallel, the computation of $F_j(c)$ from $F_{j-1}(c)$ takes constant number of operations and the total time for computing $F_m(c)$ is proportional to *m,* the number of objects. Algorithm 2 is the algorithm for calculating $F_1(c), F_2(c)\ldots., F_m(c)$.

Algorithm 2: Computing $F_1(c), F_2(c),\ldots., F_m(c)$

All places initialize $f_0(y)=0$
for *i* in 1 to *m* **do** (in parallel at all places) **begin**
places $(0 \le y \le w)$ **do** $f_j(y)= f_{j-1}(y)$
places $(w_i \le y \le c)$ **do**
$f_j(y)=\max\{f_{j-1}(y), f_{j-1}(y-w_j)+ p_j\}$
end

After $F_m(c)$ is obtained, we need to backtrack to find the solution vector. The algorithm for this is described below:

Algorithm 3: Tracing back to find the solution

All places initialize W to c and P to $f_m(c)$
for *i* in *m* **downto** 1 **do** (in parallel)
if $f_{i-1}(W) = P$
then $z_i = 0$
else
$z_i = 1;$
$P = P - p_i;$
$W = W - w_i;$

Complexity and Analysis of the new algorithm:
The above parallel algorithm is relatively simple and straight forward compared to Lee's algorithm and more effective that Chan's algorithm. Also with number of places equal to *c* , the speedup and the efficiency that can be achieved is O(*c/logc*) and O(*1/logc*) respectively. Hence this algorithm would show best performance if implemented on a massive multiprocessor system or if implemented in a network of significantly large number of places. The time complexity for this procedure is *O(mlogc)*.
In backtracking part, one of the difficulties is that in each step there is only one active place. But the value of W and P is known

by all the places. This reduces the amount of communication required among the places.

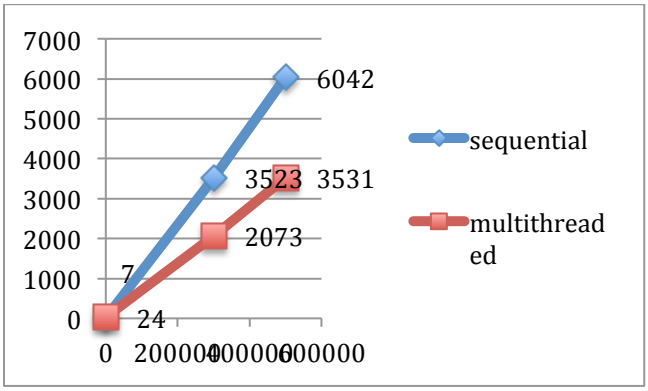
The slow down of a logarithmic factor in this parallel implementation can be due to two costs, one is the data communication between the n places and the other is when there are very few places compared to capacity of knapsack c.

6. Results and discussions

The above-mentioned parallel algorithm was implemented using multithreaded as well as multi places design. Test cases with different number of inputs and capacities will ran and a comparison was made between the runtime for sequential, multithreaded and multi places implementation. We did not see much difference between the run time between sequential and multiplaces implementation probably due to communication overhead and less number of places, therefore we have not included these results here but the code for multiplaces implementation has been given. For the multithreaded implementation, we did observe a scale up which increased as we increased the capacity of the knapsack.

Results and Observations:

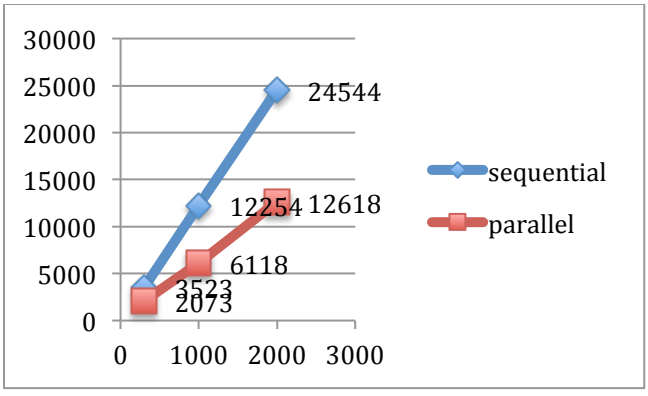
The given plot compares the run time for sequential and multithreaded implementation at a fixed number of items (*m=300*) and variable knapsack size.



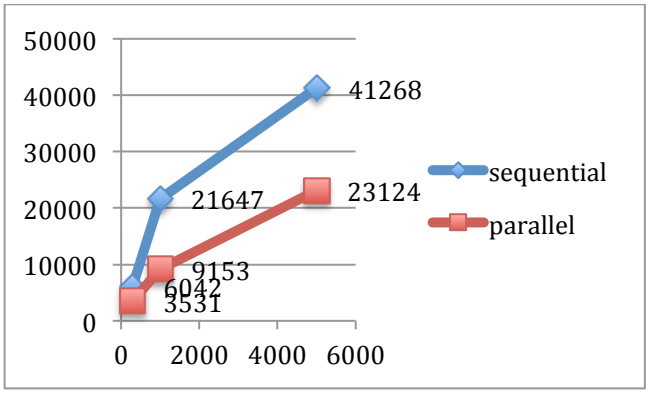
We observe in the plot that as the capacity of the knapsack increases, the parallel implementation gives better runtime as compared to sequential implementation. This does not hold true for lower values of knapsack.

The following plots compare the runtime for sequential and parallel implementation for fixed capacity and varying number of items (*m=300, m=1000, m=2000*).

Capacity of knapsack = 300000:



Capacity of knapsack = 500000:



Conclusion:

As we said earlier, the multi places algorithm gave slower run time mostly due to very few places as compared to the capacity of knapsack *c* and also due to communication overhead. The multithreaded implementation, which used the same parallel algorithm as multiplaces showed a scaling up of runtime as the capacity of the knapsack and the number of items increased. Hence in order to use the parallel implementation of 0/1 knapsack problem, it is important to consider the knapsack size and number of items. For smaller knapsack capacity and less number of users, the sequential implementation gives a faster run time as compared to multithreaded. For multiplaces algorithm, the implementation would give a good scale up only if there is sufficient number of machines/places. Due to limited resources, it was hard to find out what number of places can give an optimal performance.

7. References

[1] Lin, Jianhua, Debray, and Storer, James A. A new Parallel Algorithm for the Knapsack Problem and its Implementation on a Hypercube, *Frontiers of Massively Parallel Computation, 1990*

[2] Lee, J., Shragowitz, E., and Sahni, S., "A Hypercube Algorithm for the 0/1 Knapsack Problem," *Journal of Parallel and Distributed Computing, vol. 5, pp.438-456, 1988*

[3] Chen, G., Chern, M. and Jang, J., Pipeline Architecture for Dynamic Programming Algorithms, *Parallel Computing 13 pp 111-117, 1990*

[4] Andonov, R., Raimbault, F., Quinton, P., Dynamic Programming Parallel Implementations for the Knapsack Problem, *submitted for review to the Journal of Parallel and Distributed Computers, 1993*

[5] Hunt, R., Comparison of Parallel Algorithms for the 0-1 Knapsack Problem on Networked Computers

[6] Dzikowski, J., 0/1 Knapsack algorithm comparison, *Illinois Institute od Technology*