THESIS

REVISITING SPARSE DYNAMIC PROGRAMMING FOR THE 0/1 KNAPSACK PROBLEM

Submitted by

Tarequl Islam Sifat

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2019

ABSTRACT

REVISITING SPARSE DYNAMIC PROGRAMMING FOR THE 0/1 KNAPSACK PROBLEM

The 0/1-Knapsack Problem is a classic NP-hard problem. There are two common approaches to obtain the exact solution: branch-and-bound (BB) and dynamic programming (DP). A so-called, "sparse" DP algorithm (SKPDP) that performs fewer operations than the standard algorithm (KPDP) is well known. To the best of our knowledge, there has been no quantitative analysis of the benefits of sparsity. We provide a careful empirical evaluation of SKPDP and observe that for a "large enough" capacity, C, the number of operations performed by SKPDP is invariant with respect to C for many problem instances. This leads to the possibility of an exponential improvement over the conventional KPDP. We experimentally explore SKPDP over a large range of knapsack problem instances and provide a detailed study of the attributes that impact the performance.

DP algorithms have a nice regular structure and are amenable to highly parallel implementations. However, due to the dependence structure, parallelizing SKPDP is challenging. We propose two parallelization strategies (fine-grain and coarse-grain) for SKPDP on modern multi-core processors and demonstrate a scalable improvement in the performance.

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor Professor Sanjay Rajopadhye. The door to Professor Rajopadhye's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to thank my colleagues for their valuable input in my thesis, especially, Nirmal Prajapati and Waruna Ranasinghe.

Finally, I must express my very profound gratitude to my parents and my wife for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The 0/1 knapsack problem (0/1-KP, or just KP in this paper) is a well known, NP-hard combinatorial optimization problem with applications in production planning [1], in risk balancing and assortment optimization [2], and in storage capacity limitation [3]. It seeks to optimally fill a knapsack of a given capacity, C, from a set of N objects. There are two standard algorithmic approaches to solving it exactly: dynamic programming (DP) and branch-and-bound (BB). The execution time of BB may vary with problem instances and this has led to very extensive research on techniques and heuristics that work well in "practice." On the other hand, most standard implementations of DP build (at least conceptually) a table whose size is parameterized by only the number N and C, leading to a complexity of NC, regardless of the specific weight-profit distribution.[1] Most knapsack solvers therefore use DP only to solve small knapsack sub-problems, leaving the "heavy lifting" to BB.

A so called, "sparse" DP algorithm (SKPDP) that performs fewer operations than the standard algorithm (KPDP) is known for a while [4–6] but to the best of our knowledge, there has been no quantitative analysis of its benefits. Moreover, the authors proposed a "wavefront array" (an early form of application-specific hardware accelerator) for this algorithm, but that too, was not actually implemented.

In this paper, we first carefully evaluate the potential benefits of SKPDP (see Chapter 3). We make the rather surprising observation that when C is sufficiently large, the expected execution time (measured by counting the "number of points generated" by SKPDP) becomes invariant with C, leading to an execution time that is only linear in the input size. This means that sparsity provides an exponential gain.

[1]Note that one of the inputs to the KP is the integer C, and its size, i.e., the number of bits needed to represent it, is $\lg C$. This is why an execution time proportional to C is considered "exponential."

Next, we explore two parallelization techniques (see Chapters 4 and 5) for implementing SKPDP on modern multicore CPUs. The first one is a fine-grain technique where processors collaborate to compute the DP table, one row at a time. In the second, “coarse-grain” algorithm, the (virtual) processors operate in a pipelined, “producer-consumer” fashion, each one responsible for computing all the elements in a row. We also do a detailed analysis of the two algorithms identifying (i) the overheads of each scheme, and (ii) situations when they are compute/memory bound. In Chapter 6 we talk about the potential future work. Before providing our experimental results, the very next chapter is dedicated to describing the background knowledge.

Chapter 2

Background

We now describe the background needed to make this paper self contained, and prior related work on the problem.

The 0/1 Knapsack Problem (KP) is formally defined [7] as follows. Given a set of N items and a knapsack with a limited capacity C, where each item i have profit p_i and weight w_i, the objective of the KP is to select items to achieve the maximum total profit without exceeding the capacity. Mathematically,

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{N} p_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{N} w_i x_i < C \\ & x_i \in \{0,1\} \end{aligned} \tag{2.1}$$

There is a significant body of work on algorithms and tools for KP, many of them described in standard textbooks [7–10]. Being an NP hard problem, many authors investigate heuristic and/or approximate algorithms, with or without bounds on the approximations. As for exact solutions, there are two main algorithms: dynamic programming (DP) and branch-and-bound (BB). We focus on DP, defined by the following recurrence.

$$f(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ f(i, j-1), & \text{if } i > 0 \text{ and } j < w_i \\ \max(f(j, i-1), \\ \quad p_i + f(j - w_i, i-1)) & \text{if } i > 0 \text{ and } j \geq w_i \end{cases} \tag{2.2}$$

The function $f(i,j)$ denotes and defines the maximal profit that can be achieved with a knapsack of capacity j, drawing items out of only the first i items. The standard KPDP algorithm computes and stores this table in an order (possibly in parallel) ditated by the dependences of the recurrence. This phase, called the forward pass, does $\mathrm{O}(NC)$ work. Then, an $O(N)$ "backtracking" traversal of the table constructs the actual solution. The total execution time of KPDP is thus $O(NC)$. An example of a table generated by the KPDP algorithm is shown in Table 2.1 which shows the DP table for a KP instance with four items and capacity 10.

Table 2.1: A KPDP table with $N = 4$ and $C = 10$ with weights, $5, 4, 6, 1$ and profits, $7, 8, 9, 4$, respectively. The significance of the boxed entries is explained in Section 2.2.

Items	Capacity										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	$\boxed{7}$	7	7	7	7	7
2	0	0	0	0	$\boxed{8}$	8	8	8	8	$\boxed{15}$	15
3	0	0	0	0	$\boxed{8}$	8	$\boxed{9}$	9	9	$\boxed{15}$	$\boxed{17}$
4	0	$\boxed{4}$	4	4	$\boxed{8}$	$\boxed{12}$	12	$\boxed{13}$	13	$\boxed{15}$	$\boxed{19}$

In this paper, we exploit two known improvements to KPDP. The first one reduces the space complexity to only $O(C)$ so that the whole table does not need not be saved (this comes with a two-fold increase in execution time). The other improvement utilizes the sparsity in the computation of the KPDP table [4–6]. For the sake of completeness, we describe them here.

2.1 Memory efficient KPDP

The dynamic programming algorithm normally requires the entire table to be retained, and this is often unacceptably large for large problems. The memory efficient KPDP is an adaptation of an old trick to improve DP algorithms, that was first proposed by Hirshberg for the longest common subsequence (LCS) problem [11]. The technique was adapted to KP by Kellerer et al. [7, pp 46-50] and by Nibbelink et at. [12] who also propose a hardware implementation on FPGAs. It avoids backtracking by recursively, *directly solving* subproblems of the form $\langle i, j, c\rangle$, that determine the optimal subset of the i-through-j-th (inclusive) objects for a capacity of c. The top level call is thus to $\mathbf{Solve}\langle 1, N, C\rangle$, and $\mathbf{Solve}\langle i, j, c\rangle$ proceeds as follows:

1. **Base case:** if $i = j$, there is only one object, so choose it if and only if it fits (i.e., $w_i \leq c$)

2. **General case:** Let $m = \lfloor \frac{j+i}{2} \rfloor$. Determine (details below) a c^* between 0 and c such that in the (some) solution to the subproblem $\langle i, j, c \rangle$ the combined sum of the weights of the selected object in the "first half" i.e., i-through-m-th objects, is c^*. This is called the "optimal split" of the capacity c. Now,

   (a) Recursively call $\mathbf{Solve}\langle i, m, c^* \rangle$, and

   (b) Recursively call $\mathbf{Solve}\langle m+1, j, (c - c^*) \rangle$

So the main issue is to determine c^*. We first determine for the first "half" of the objects, a vector $X[j]$ for $0 \leq j \leq c$, the optimal profit that can be obtained with capacity j (using only objects in that half). We also construct a similar vector, $X'[j]$ for the other half. This can be done in only $2c$ memory using recurrence (2.2), by maintianing only two rows of the table, and costs $(j-i)c$ work. Using these two arrays, c^* is given by

$$c^* = \mathrm{argmax}_{i=0}^{c} X[i] + X[c-i]$$

The capacity arguments in all the calls at any levels of the recursion tree add up to C, and the number of objects considered halves level by level. Hence, the total work of this algorithm is $2NC$, just a two-fold increase.

2.2 Sparse KPDP (SKPDP)

The KPDP algorithm generally retains a table which contains the maximum profits achievable by any value of capacity between 0 and C and considering the items incrementally. To clarify, in the table 2.1, the cell indexed by the 3^{rd} row and the 8^{th} column contains the maximum profit value considering only the first 3 items and a capacity value of 8. Andonov and Rajopadhye [4, 5] and Dupont de Dinechin et al. [6] showed how, for the KPDP and the UKP (the unbounded knapsack problem), the sequences representing the i-th row could be computed from the one representing the $(i-1)$-st row. The work is similar to early ideas on a list based algorithm [13] that reduced the complexity of the subset-sum problem to $O(\sqrt{2^N})$, but uses *streams* of *index-value pairs*. The main idea behind the SKPDP is the notion of monotonically: it is clear that for any fixed i, the values of $f(i,j)$ are monotonically non-decreasing with respect to j. This is because, in any subproblem, we can do no worse than we are currently doing by increasing capacity. This is illustrated in Table 2.1, where the boxed entries are the first occurrence of a value in a given row. Because of sparsity, one may enviage a "sparse" representation, where all the table entries are not computed, rather only the "points of inflection" are explicitly evaluated. To do this, we use a representation for the table in the form of "index-value" pairs: the entire i-th row is represented by a list, of the form, $\langle j, f(i,j)\rangle$, as shown in Table 2.2. We refer to these pairs, representing the boxed entries of the Table 2.1, as "critical points".

Table 2.2: The SKPDP representation of Table 2.1.

Items	
1	$\langle 0,0\rangle$, $\langle 5,7\rangle$
2	$\langle 0,0\rangle$, $\langle 4,8\rangle$, $\langle 9,15\rangle$
3	$\langle 0,0\rangle$, $\langle 4,8\rangle$, $\langle 6,9\rangle$, $\langle 9,15\rangle$, $\langle 10,17\rangle$
4	$\langle 0,0\rangle$, $\langle 1,4\rangle$, $\langle 4,8\rangle$, $\langle 5,12\rangle$, $\langle 7,13\rangle$, $\langle 9,15\rangle$, $\langle 10,19\rangle$

2.3 Add-Merge-Kill

Andonov and Rajopadhye [14] and Dupont de Dinechin et al. [6] proposed to implement the SKPDP algorithm on dedicated application specific hardware as a WAP (Wavefront Array Processor [15, 16]). However, because the memory efficient technique had not been discovered at that time the total number of data transfers that the accelerator needed to perform was the same order as the number of computations.

Regardless of whether it is implemented in hardware or in software, and whether the final algorithm is memory-efficient or not, the heart of the Andonov-Rajopadhye sparse algorithm is a technique called Add-Merge-Kill, by which any row of the sparse table can be constructed from the previous one. To understand it, consider the 3rd row of the Table (2.2),

$$\langle 0,0\rangle, \langle 4,8\rangle, \langle 6,9\rangle, \langle 9,15\rangle, \langle 10,17\rangle \tag{2.3}$$

To generate the 4th row we first add the weight and profit of the 4th element $\langle 1,4\rangle$ to every element of the list (2.3). We get,

$$\langle 1,4\rangle, \langle 5,12\rangle, \langle 7,13\rangle, \langle 10,19\rangle, \langle 11,21\rangle \tag{2.4}$$

Next, we merge the two lists (keeping them in increasing order of the first component of each tuple, producing the list:

$$\langle 0,0\rangle, \langle 1,4\rangle, \langle 4,8\rangle, \langle 5,12\rangle, \langle 6,9\rangle, \langle 7,13\rangle, \langle 9,15\rangle, \langle 10,19\rangle, \langle 10,17\rangle, \langle 11,21\rangle \tag{2.5}$$

Then, we delete (kill) all the dominated elements, i.e., those whose second component is not (strictly) greater than a preceding element's second component, producing the list,

$$\langle 0,0\rangle, \langle 1,4\rangle, \langle 4,8\rangle, \langle 5,12\rangle, \langle 7,13\rangle, \langle 9,15\rangle, \langle 10,19\rangle \tag{2.6}$$

which is exactly row 4 of Table 2.2. Notice that $\langle 11,21\rangle$ has been discarded because 11 is greater than the maximum capacity 10. Algorithm 1 gives a pseudo-code of our implementation of the Add-Merge-Kill algorithm.

Figure 2.1 shows a graphical depiction of the algorithm 1. The solid blue line represents the 3^{rd} sequence S_3. The dotted orange line represents the sequence we get after adding $\langle 1,4\rangle$ to each pairs of S_3. The solid green line represents the 4^{th} sequence which we get by taking the maximum profit values at each critical points of the previous two sequences.

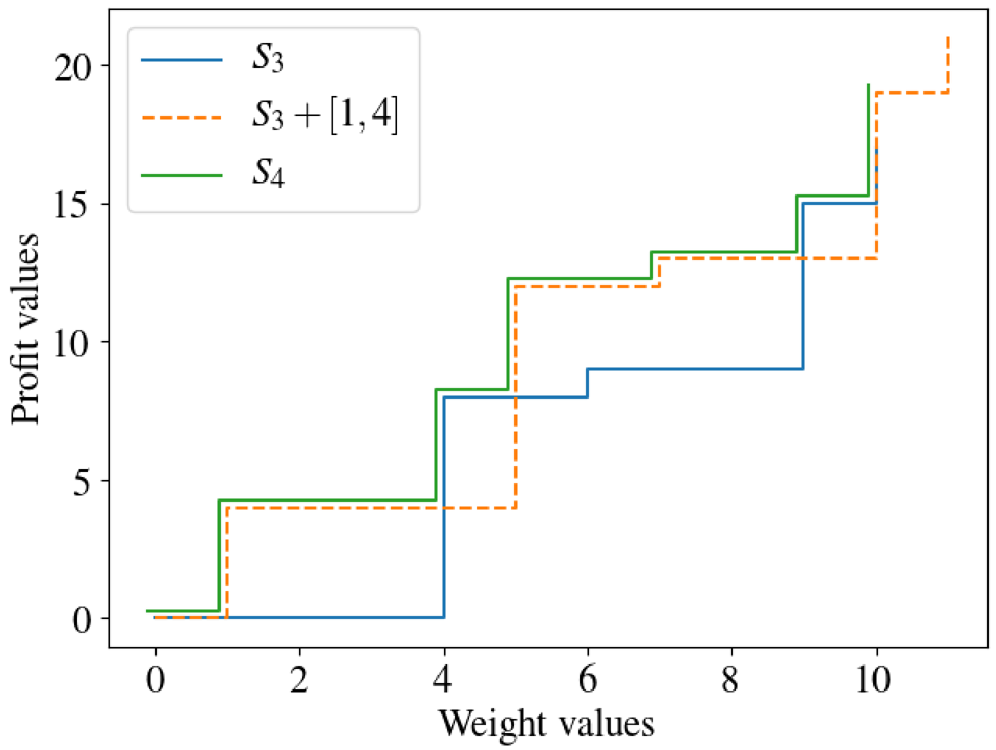


Figure 2.1: Add-Merge-Kill algorithm; the solid blue line represents the 3^{rd} sequence S_3; the dotted orange line represents the sequence we get after adding $\langle 1, 4 \rangle$ to each pairs of S_3; the solid green line represents the 4^{th} sequence which we get by taking the maximum profit values at each critical points of the previous two sequences.

Algorithm 1: Add-Merge-Kill

```
Input  : $S_i$ : i-th row
         $\langle w_{i+1}, p_{i+1} \rangle$: weight and profit of the $(i+1)$-th item
Output: $S_{i+1}$ : (i+1)-th row
$S_i \leftarrow$ current row, $S_{i+1} \leftarrow$ empty;
$j, k, p \leftarrow 0, 0, 0$;
/*Add step*/
$S_i' \leftarrow$ replace each element $\langle x, y \rangle$ in $S_i$ by $\langle x + w_{i+1}, y + p_{i+1} \rangle$ ;
/*Merge-Kill step*/
while the end of $S_i$ not reached and $S_i'[k].weight \leq C$ do
    if $S_i[j].weight < S_i'[k].weight$ then
        if $S_i[j].profit < S_i'[k].profit$ then
            $S_{i+1}[p] \leftarrow S_i[j]$;
            $p \leftarrow p + 1, \quad j \leftarrow j + 1$;
        else
            $k \leftarrow k + 1$;
        end
    else if $S_i[j].weight > S_i'[k].weight$ then
        if $S_i[j].profit > S_i'[k].profit$ then
            $S_{i+1}[p] \leftarrow S_i'[k]$;
            $p \leftarrow p + 1, \quad k \leftarrow k + 1$;
        else
            $j \leftarrow j + 1$;
        end
    else
        if $S_i[j].profit > S_i'[k].profit$ then
            $k \leftarrow k + 1$;
        else
            $j \leftarrow j + 1$;
        end
    end
end
/*Append the remaining items*/
while the end of $S_i$ not reached do
    $S_{i+1}[p] \leftarrow S_i[j]$;
    $p \leftarrow p + 1, \quad j \leftarrow j + 1$;
end
while the end of $S_i'$ not reached and $S_i'[k].weight \leq C$ do
    $S_{i+1}[p] \leftarrow S_i'[k]$;
    $p \leftarrow p + 1, \quad k \leftarrow k + 1$;
end
```

2.4 Symbols

If not stated otherwise the meaning of the symbols generally used in this paper is defined in the table 2.3.

Table 2.3: Meaning of the symbols used in this paper

Symbols	**Meaning**
N	Number of items in a knapsack problem instance
C	Capacity of a knapsack problem instance
w_i	Weight of the i-th item of a knapsack instance
p_i	Profit of the i-th item of a knapsack instance
P	Number of processors/threads
P_k	Symbolizes the k_{th} processor/thread
W_{avg}	Average weight of the items of a knapsack problem instance
λ	Input parameter to SKPDP that is used to determine the fraction of the number of the items that fit into the knapsack on average; fraction of the objects that fit = $\frac{1}{\lambda}$ when $W_{\text{avg}} = \frac{\lambda C}{N}$
σ	Input parameter to SKPDP that that controls the variance in profitability (ratio of p_i and w_i) among different items and it is also referred as the noise value

The next three chapters describe our main results: empirical study of the benefits of sparsity, a fine-grain parallelization, and a coarse-grain parallelization that provides scalable speedup on shared-memory platforms.

Chapter 3

Empirical Analysis of Complexity

To investigate the potential gains of the sparse implementation over the dense version, we implemented SKPDP and measured the total number of iterations for many instances of knapsack problem. We define "iteration count" as the total number of Merge-Kill (MK) operations executed by the sparse algorithm. We use "iteration count" as a surrogate for this total number of computations in the sparse version. Note that it is trivial to compute the total number of iterations and the total operation count is a multiple of the number of iterations. This is a constant multiple, and moreover, we observed a near perfect correlation between "iteration count" and the actual execution time. The reduction in execution time can be exponential when $N \ll C$. The goal of this empirical study is to identify the different types of problem instances where SKPDP can outperform conventional KPDP. Given that we are interested in knapsack problem instances with significantly large values of C, we must use the memory efficient version of knapsack that uses divide-and-conquer strategy so that we don't run out of available physical memory. Even though the memory efficient version does about twice as much computation compared to the full table version, it enables us to experiment with knapsack problem instances with very large values of C.

3.1 Generation of Problem Instances

Researchers have been synthesizing different types of knapsack problem instances. Pisinger [17] surveyed the performance of several popular algorithms on a suite of knapsack problem instances, and showed that problem instances with strongly-correlated weights and profits are hardest to solve. Since we are interested in DP, we generate instances where the capacity, C, and correspondingly, the weights and profits can be arbitrarily (i.e., exponentially) increased. In our exploration, we seek to study the impact of a certain set of parameters, as discussed below.

Furthermore, it is important to make sure that the knapsack problem instances are not contrived.

3.1.1 The expected number of objects that fit

We control the fraction (of the N items) the fit into the knapsack. For this, we introduce a parameter, λ, (where $\lambda \geq 2$) and set the average weight of all the items to $W_{\text{avg}} = \frac{\lambda C}{N}$ so that the fraction of items that fit into the knapsack is $\frac{1}{\lambda}$.

3.1.2 Distribution of the weights

Next, we design two types of weight distribution schemes for the average weight given by W_{avg} above. In the first scheme, we use a normal distribution with a mean of W_{avg} and a standard deviation of $0.3W_{\text{avg}}$. For the second type, we use a log-normal distribution with a mean of W_{avg} while making sure that the largest weight (given by $W_{\max}$) is close to C. This helps us analyze the impact of very large weights on SKPDP.

3.1.3 The variance in profitability

It is well known that for every item (i) in the knapsack problem, when the ratio of the profit (p_i) and weight (w_i) is a constant, we end up with the hardest problem instances, essentially variants of the subset-sum problem. We would like to observe the impact of variance in profitability (p_i/w_i) of the knapsack items on SKPDP. Therefore, we generate problem instances with different levels of correlation between weights and profits. For this, we first set the profits to be a constant multiple of the weights, and then introduce a small random noise value. The noise is used to adjust the variance in profitability of the knapsack items.

$$p_i = c_0 w_i + r_i \tag{3.1}$$

where, c_0 as a constant, and r_i is in the range $[-\sigma W_{\text{avg}}, \sigma W_{\text{avg}}]$. According to Pisinger [17], if $c_0 = 1$ and $r_i = \alpha W_{\max} \quad [0 < \alpha < 1]$ equation (3.1) generates *Strongly Correlated* problem instances, otherwise *Weakly Correlated* problem instances are generated. While *Strongly Correlated* problem instance are the second hardest after the Subset Sum problem instances, *Weakly Correlated* problem instance can have variable levels of hardness depending on the value of σ. Note that our approach of generating the weights and profits is a bit different from that of Pisinger [17]. Instead of choosing the number of items, N and the range of weights, R, we pick the number of items and the capacity, C first. Also, instead of randomly choosing weights within the given range, we take a normal distribution of the weights in order to better control the average weight. This is useful for evaluating SKPDP because the fraction of the number of items that fit into the knapsack is an important attribute to be explored.

We also make sure that our problem instances are not artificially easy. When generating a problem set (varying N and C) we make sure to maintain the same values of the three attributes we discussed earlier. So, a problem instance with a larger capacity also has a weight distribution where the average weight, W_{avg} is proportionally larger.

3.2 Gain

We measure the total work done by SKPDP in terms of the number of iterations executed. For SKPDP to be beneficial, the number of operations performed by the sparse algorithm must be significantly less than that of the dense version. The number of iterations required to generate one row of the sparse table is no more than twice the size (number of pairs) of the previous row. So, the total number of operations needed to generate the whole sparse table is a constant factor of the total number of pairs/critical points in the generated sparse table. With the increase in sparsity of the table, we expect improvement in the performance. In the worst case, a problem instance will hardly have any sparsity and, therefore, most of the rows in the sparse table will be of length C. To generate each row we perform approximately $2C$ Merge-Kill operations. So, in the worst case the upper bound on the computational complexity of SKPDP will be $O(NC)$. Because we are using the memory efficient divide-and-conquer strategy to find the exact solution, the constant factor is 4 (the sparse algorithm manipulates index-value *pairs*, two-fold inefficient than simply the profit value computed by the dense KPDP). We define potential performance gain as,

$$\begin{aligned} \text{gain} &= 1 - \frac{\text{no. of iterations for SKPDP}}{\text{no. of iterations for dense KPDP}} \\ &\approx 1 - \frac{\text{no. of iterations for SKPDP}}{2NC} \end{aligned} \tag{3.2}$$

The denominator is $2NC$ because we are using the memory efficient implementation of KPDP. From 3.2, we can deduce that the value of gain is in the range $[-1, 1]$. A positive gain implies that SKPDP performs better than conventional KPDP and a negative value of gain means that SKPDP performs worse than conventional KPDP. We use the number of iterations (of Merge-Kill operations) needed to compute gain, instead of the sum of the lengths of the rows in the sparse table, because the iteration count is more representative of the actual operation count and the execution time. Also note that when the gain is 1, the number of perations performed by SKPDP is negligible compared to KPDP (ther fractional part tends to zero).

In Figure 3.1, we plot the computational gain of SKPDP against problem instances with the total knapsack capacity C ranging from 2^{12} to 2^{49} and number of items N ranging from 2^8 to 2^{12}. To generate this plot, we used 190 different knapsack problem instances with 38 different values of C and 5 different values of N, each with a normal distribution of the weight values, where approximately $1/8$ of the items fit into the capacity (i.e., average weight is $8C/N$) and *Weakly Correlated* weight and profit values with $\sigma = 0.1\%$. For smaller values of C the gain is negative, meaning that the overhead of calculating the sparse data structure is high compared to the reduction in data point in the table. As we consider problem instances with larger values of C, the value of gain approaches 1, which indicates that the number of iterations needed for SKPDP is significantly less than conventional KPDP. This prompts us to do further investigations on the benefits of SKPDP.

3.3 Experimental Results

We did all experimentation using C programming language on linux machines with *Intel(R) Xeon(R) CPU E5-2650 v2* processors running at 2.60GHz. The processor has 8 cores with hyper-threading (16 threads). The machines are equiped with 32GB of RAM. The compiler used is *Intel*

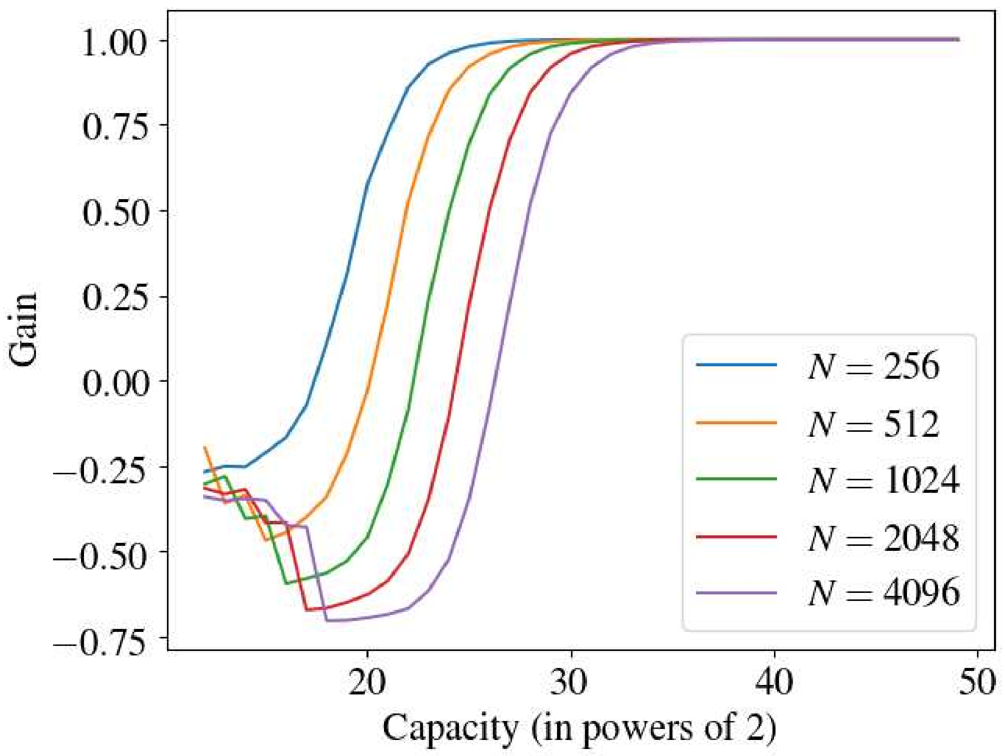


Figure 3.1: Gain due to sparsity (on a log-log plot) as a function of C, on 190 different knapsack problem instances each with items with (i) *Weakly correlated* weight and profit values with $\sigma = 0.1\%$, (ii) normal distribution of the weight/profit values, and (iii) $W_{\text{avg}} = 8C/N$.

C++ Compiler version 19.0.0.117. We used *OpenMP* [**?**] for implementing prallelization on CPU. We used Intel VTune 2019 performance analysis tool to analyze our programs.

Figure 3.2 shows a log-log plot of the execution time vs the capacity using the same problem instances that are used to generate figure 3.1. It confirms our hypothesis that SKPDP performs significantly better for problem instances where $N \ll C$. Note that the problem instances in Figure 3.2 have the same attributes as those in the Figure 3.1. For problem instances with larger capacity, the weight values are also proportionally larger which ensures that the average number of elements that fit into the capacity remains N/λ (where $\lambda \geq 2$) for all problem instances. This further guarantees that the problems are not easily solvable.

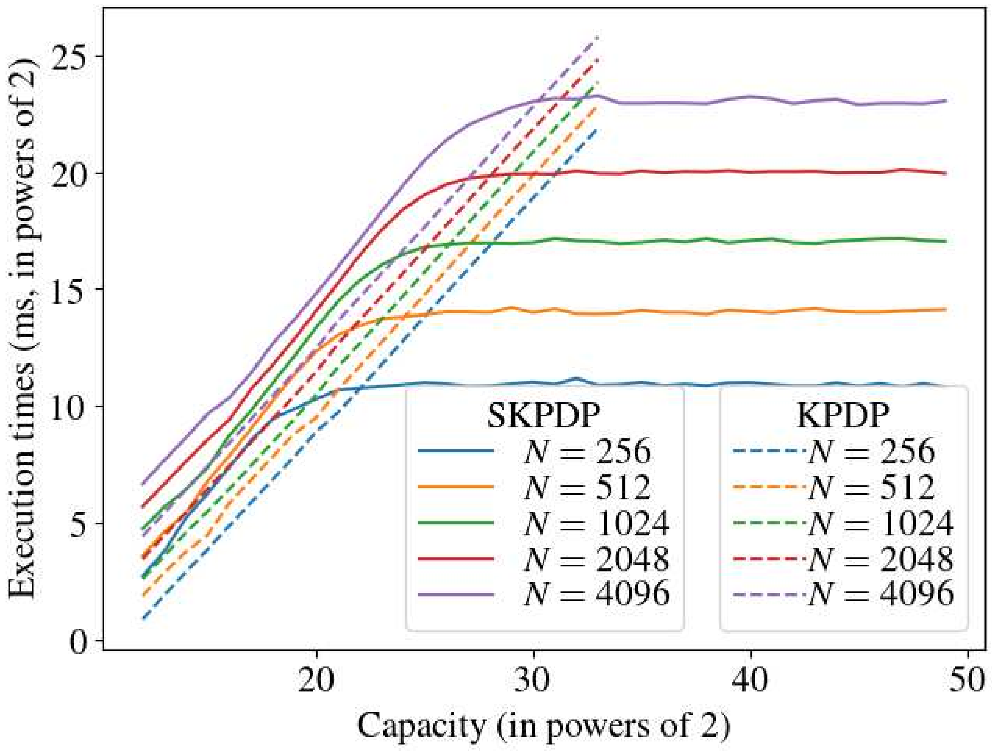


Figure 3.2: Execution times of SKPDP for different instances of knapsack problem with varying values of N and C. For all problem instances $W_{\text{avg}} = \frac{8C}{N}$. The solid lines represent the execution times of SKPDP and the dashed lines represent the execution times of conventional KPDP.

In Figure 3.2, we observe that while the execution times of KPDP(dashed lines) stays linear with capacity C (exponential with input size $\lg(C)$), the execution times of SKPDP(solid lines) become invariant with capacity. So, for problem instances where $N \ll C$, the gain in performance can be exponential with SKPDP compared to KPDP. On the other hand, when C is not large enough SKPDP is only a constant factor worse than KPDP. For the problem instances that we explored, in the worst case SKPDP is approximately 7 times slower than conventional KPDP.

From Figure 3.3, we observe the correlation between the iteration count and execution time for the SKPDP implementation. It is clear that for SKPDP the iteration count is proportional to the execution time.

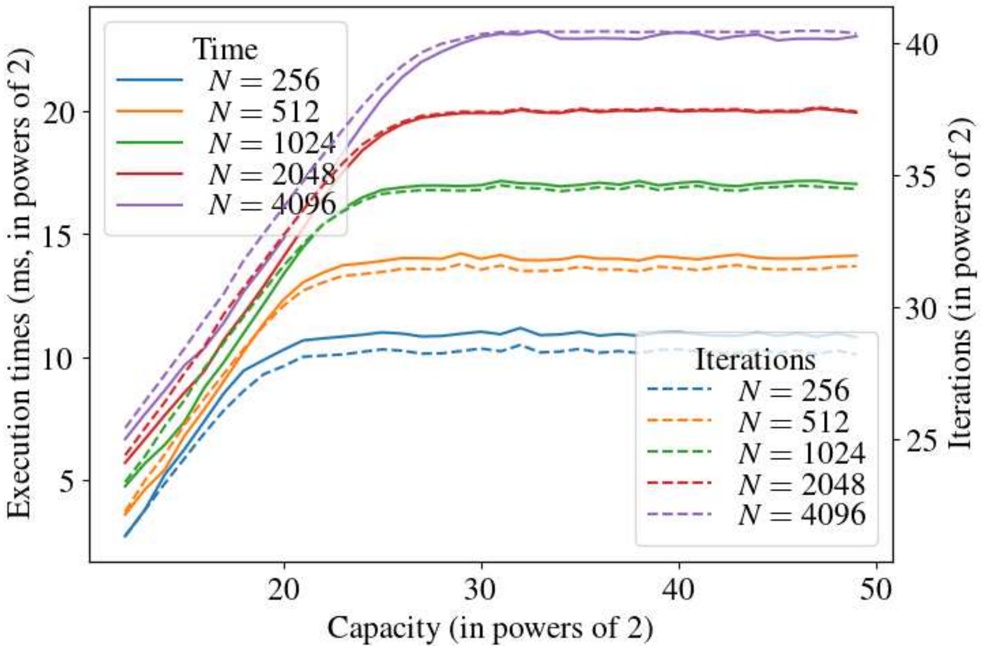


Figure 3.3: Relation between execution time and the iteration count of SKPDP across instances of the knapsack problem. The correlation of the iteration count and the execution time suggests that the iteration count is directly proportional to the execution time.

The challenge is to identify those problem instances that have large enough value of C such that SKPDP proves to be beneficial over conventional KPDP. Such instances depend on the number items, N, the correlation between the weights and profits(i.e., profitability) and the total number of items that fit into the capacity(guided by λ).

3.3.1 Weakly Correlated Problem Instances

For *Weakly Correlated* problem instances the variance in profitability is can be controlled using the value of σ in equation (3.1). Figure 3.4 shows that this is also true for SKPDP. When $\sigma = 0$, the SKPDP has exponential behavior. As we increase the noise(σ) value, thus increasing the variance in profitability among the items, we see much better performance by SKPDP. Even with a fairly low variance in profitability we can see an exponential performance improvement from SKPDP. On the other hand, the variance in profitability does not impact the complexity of conventional KPDP.

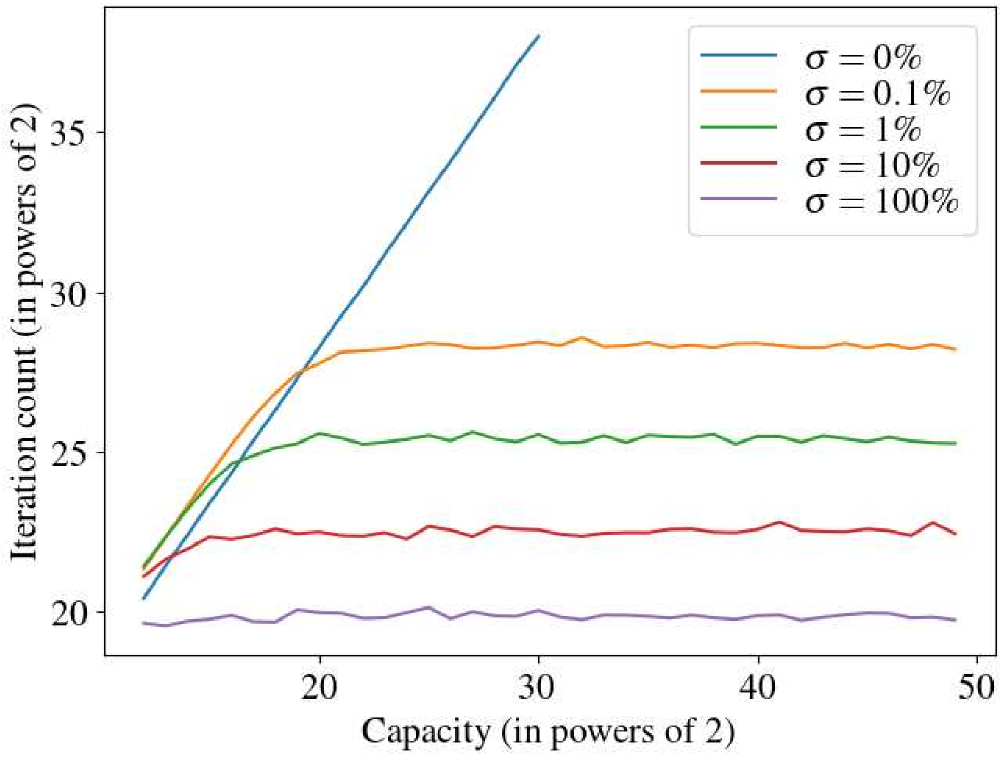


Figure 3.4: Iteration count of SKPDP for different levels of variance in profitability; For all problem instances $N = 256$ and $W_{\text{avg}} = \frac{8C}{N}$

3.3.2 Impact of the fraction of the items that fit into the capacity

Figure 3.5 compares the iteration count of knapsack problem instances with 4 different average weights of the items. When we want *half* of the items to fit into the capacity (on average), we make sure that $W_{\text{avg}} = \frac{2C}{N}$; when we want *1/4-th* of items to fit into the capacity, we make sure that $W_{\text{avg}} = \frac{4C}{N}$ and so on. From figure 3.5, we see that for any values of N and C, the problem instances with higher values of W_{avg} have more sparsity. Again, this attribute does not impact the complexity of conventional KPDP.

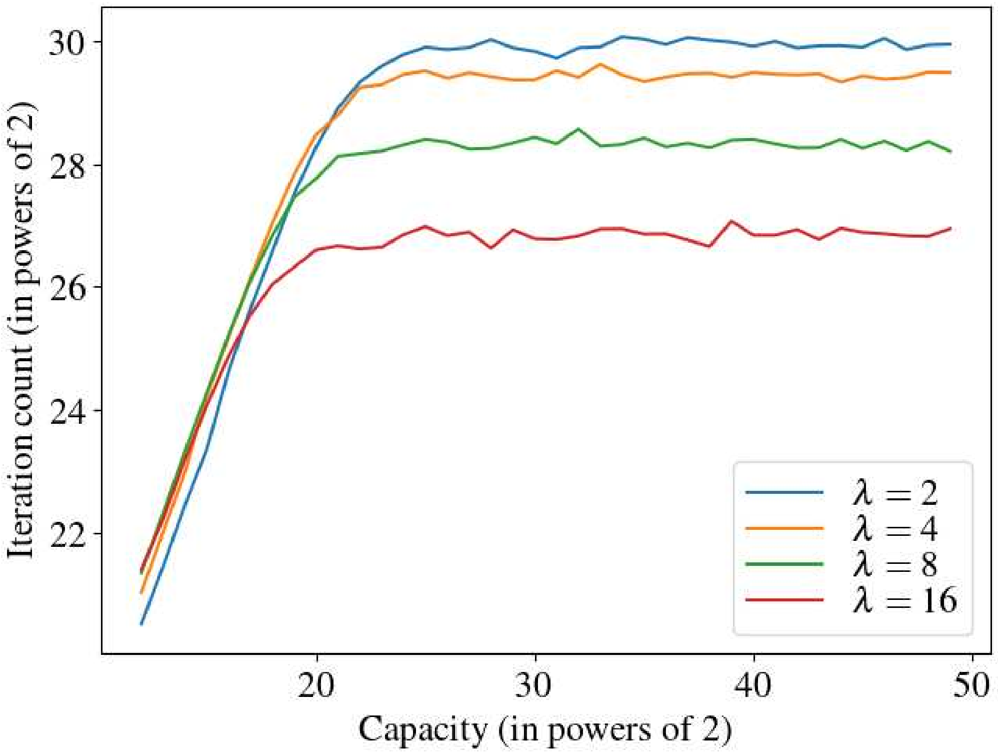


Figure 3.5: Iteration count of SKPDP for different number of the items fitting into the capacity; $N = 256$, and the noise is $\sigma = 0.1\%$

As we can see from the figures 3.4 and 3.5, for large values of C, the iteration count reaches a peak and (almost) flattens out. The peak values vary for different values of correlations between the weights and profits and also the fraction of the number of elements that fit into the knapsack. Figure 3.6 confirms this fact. A block in the figure 3.6 represents the peak iteration count achieved by knapsack problem instances with λ values ranging from 2 to 16 and σ values ranging from 0.1% to 100%.

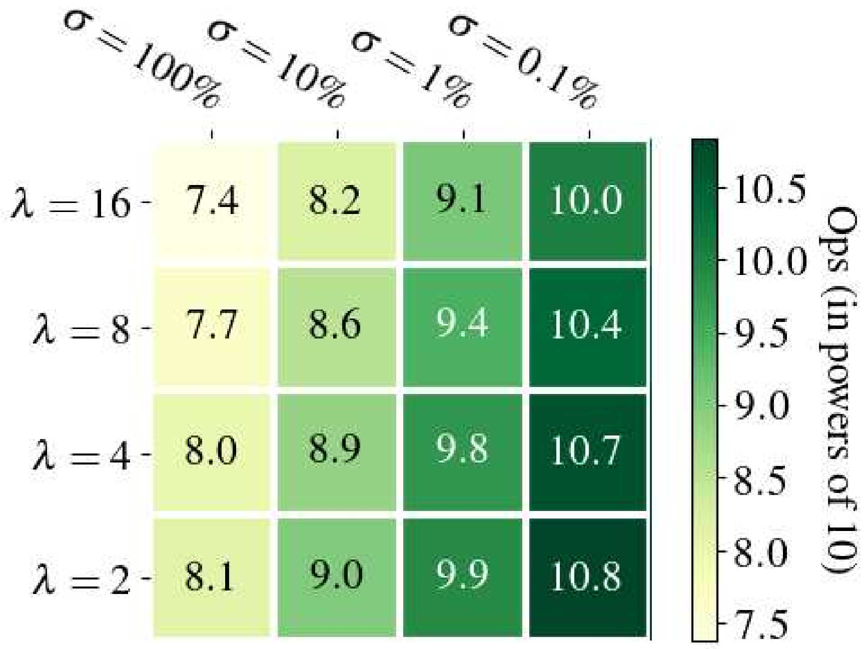


Figure 3.6: The impact of λ and σ on the sparsity; For all problem instances $N = 2^{10}$ and $C < 2^{50}$; Each block shows the peak iteration count for values of λ and σ.

3.4 Strongly correlated problem instances

When we set the noise $\sigma = 0$, we end up with a variation of the Subset-sum problem. According to Pisinger [17], the hardest knapsack problem instances, other than the Subset-sum problem, are the ones with the following attributes

$$p_i = w_i + \alpha W_{\max} \tag{3.3}$$

For the problem instances, that adhere to the equation (3.3), the Pearson correlation coefficient between the weights and profits is 1, but unlike the Subset-sum problem different items have different values of profitability. This variance in profitability is dependent on the distribution of the weight (or profit) values and the value of α. In equation 3.3, when $\alpha = 0$ (which is same as setting the noise $\sigma = 0$ in equation 3.1) the variance in profitability is zero. Given that $\alpha > 0$, if the weight distribution is random then the variance in profitability is also random and the execution time of SKPDP is unpredictable. Knapsack problem instances that have a normal weight distribution and adhere to the equation (3.3) are hard for SKPDP. Figure 3.7 shows the iteration count for SKPDP on a knapsack problem instances with no variance in profitability (similar to the Subset-sum problem) and also on a *Strongly Correlated* problem instance. For reference, we have included the iteration count of KPDP for different values of N and C (which is $N * C$). Iteration counts of SKPDP for the problem instances with no variance in profitability is very similar to KPDP. For *Strongly correlated* problem instances SKPDP also shows asymptotically exponential complexity but we do observe an exponential improvement (no matter how small of an exponent it may be).

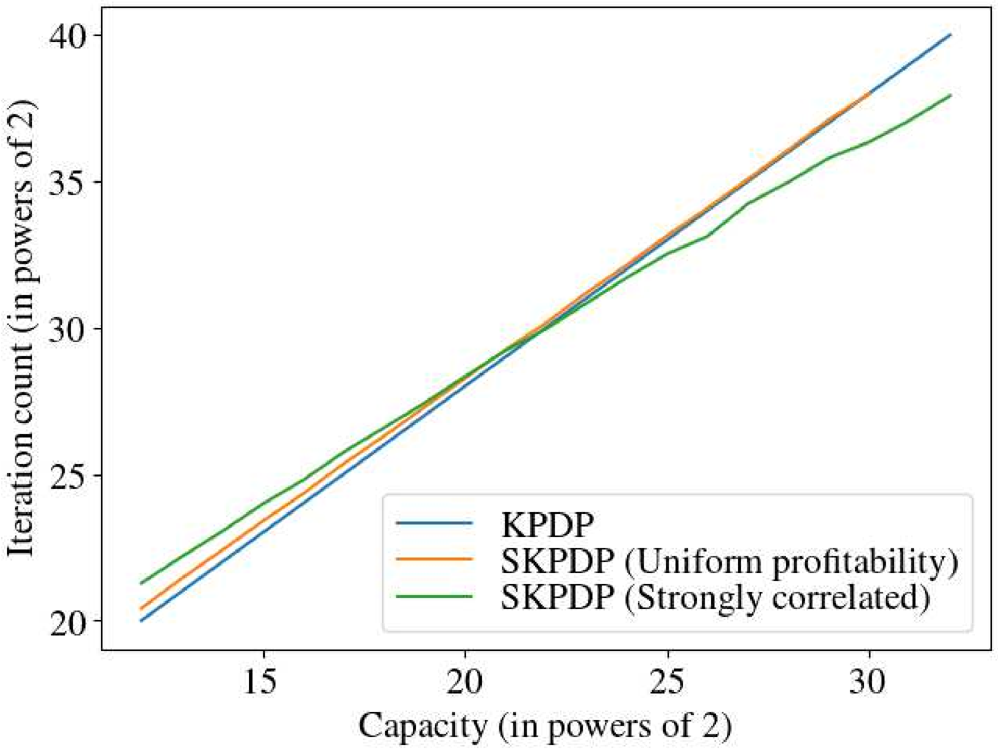


Figure 3.7: Comparision of iteration counts of KPDP, SKPDP (on Perfectly correlated Knapsack problem instance), and SKPDP (on Strongly Correlated problem instance); where $N = 256, \sigma = 0.001, \lambda = 8$.

Chapter 4

Fine-grain Parallelization of SKPDP

The conventional KPDP table is filled out from top to bottom and left to right. Because each row depends on entries from only the only previous one, the left to right computation can be trivially parallelized. But in SKPDP, the left to right computation, Add-Merge-Kill (AMK), is inherently sequential. In this chapter we explain how this can nevertheless be parallelized, for the price of producing some additional (potentially incorrect) points which we fix later in a cleanup stage. This is similar to, and yet independent of the notion of *rank convergence* proposed by Maleki et al. [18] who also break dependencies to parallelize DP, possibly incorrectly, and subsequently fix errors (due to broken dependency) in a later stage. They showed that a property called *rank convergence* can significantly reduce the overhead of the cleanup stage. However, the KPDP is not amenable to this because its rank can be proved to remain non-convergent (the matrices involved remain full-rank). Nevertheless, we show how the basic idea is adaptable.

Figure 4.1 illustrates the parallelization. If the size of the row is small, the row is computed by a single thread (the yellow rows), otherwise the row is split into P contiguous sections (red row), with the k_{th} processor computing the k_{th} section of the row. The size of each section is bound by C/P.

Since we represent a sparse row (see Table 2.2) as a list of "index-value" pairs of the form $\langle j, f(i,j)\rangle$, we now have P such sublists of index-value pairs. All threads independently apply AMK to their respective sublists, producing outputs $O_1 \ldots O_P$. However, if we simply concatenate the outputs, we will not get a legal sequence of pairs that represent S_{j+1}: some of the index-value pairs in O_k may be dominated by the elements in $O_{k'}$ for $k' < k$, and none of them have been compared with each other.

Define x such that the last x elements of O_k have a value of j no less than the first element of O_{k+1}. Similarly, define y such that the first y elements of O_{k+1} have no higher value of j than the last element of O_k. Then, we apply AMK to the last x elements of O_k and the first y elements of

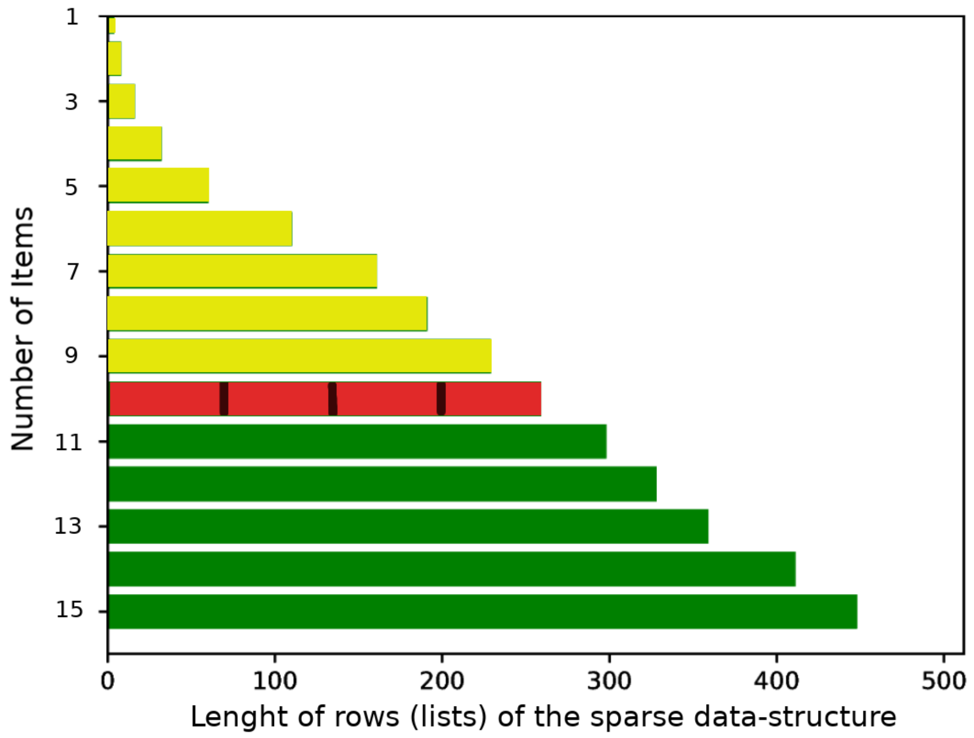


Figure 4.1: Fine-Grain Parallelization of SKPDP; $N = 16, C = 512, \sigma = 0.001, \lambda = 2$

O_{k+1}. This "stitching" step does $x + y$ extra iterations of AMK. This too can be done in parallel, all threads (in parallel) applying the AMK on their suffix and the prefix of the next section.

4.1 Overhead Analysis

In the worst case $x = y = w_i$. So, every stitching for a row will require at most $2(P-1)w_i$ extra AMK operations. In order to parallelize on P processors, the size of the input row must be greater than $\alpha w_i P$ (here, $\alpha \geq 2$ is a tunable constant). This ensures that there is no overlap between O_{k-1} and O_{k+1} while stitching. In the worst case, $2w_i$ elements will need to be merged for every section for every row. Thus a total of $2P\sum_i w_i$ extra computations must be performed for stitching. The upper bound on this is C, the capacity. Every row may do as much as C extra work. This implies a factor of two overhead. Because of this, and because our preliminary implementation confirmed the high overhead, we did not pursue this parallelization further, even though the algorithmic strategy was intriguing.

Chapter 5

Coarse-grain Parallelization

In the previous chapter, we discussed parallelizing the computation of a single row in the SKPDP table. In this chapter, we explore the potential of computing multiple rows at the same time.

The idea of using the coarse-grain parallelization technique stems from the realization that a whole row of the sparse table does not have to be computed before starting the calculation of the next row. A similar technique is used to parallelize the Unbounded KPDP [19] which has dependencies between rows. As a row of the sparse table is being computed, the values can be used immediately for calculating the next row. A thread is responsible for the computation of a row and each thread acts as a consumer of the values produced by the thread computing the previous row. In hardware, this can be implemented as streaming input to processing elements of a hardware accelerator [4], but in software, element-wise synchronization between threads can be very costly. In order to minimize the synchronization cost between threads, the data between two consecutive threads can be shared in blocks or bursts instead. Here, the block or burst size becomes a very important parameter to be tuned because huge block sizes can hinder the parallelism and tiny block sizes can introduce a lot of costly synchronization between threads.

One obvious way of computing multiple rows at a time is to spawn as many threads as the number of rows in the table. In this approach, the computation of each row is assigned to a thread. In the beginning, all the threads except for the first one will be blocked because only the first thread has available input data to proceed. As soon as the first thread produces a block of output data, it passes the data along to the second thread to start computing the first part of the second row. Similarly, the third row starts computing when data is made available by the second row and so on. Notice that this approach can spawn many threads because the number of threads spawned is $O(N)$. Interestingly, this approach does not cause deadlock despite such a large number of threads because when a thread is waiting on the adjacent threads it releases the physical core for other threads and there are no circular dependencies.

While the previous approach works, it may have a lot of overhead due to spawning $O(N)$ threads at once. Instead of spawning $O(N)$ threads at once, we could work with P rows at a time spawning only P threads. Here P is a parameter of the algorithm. With this approach, we calculate the sparse table of SKPDP in $\lceil N/P \rceil$ separate passes.

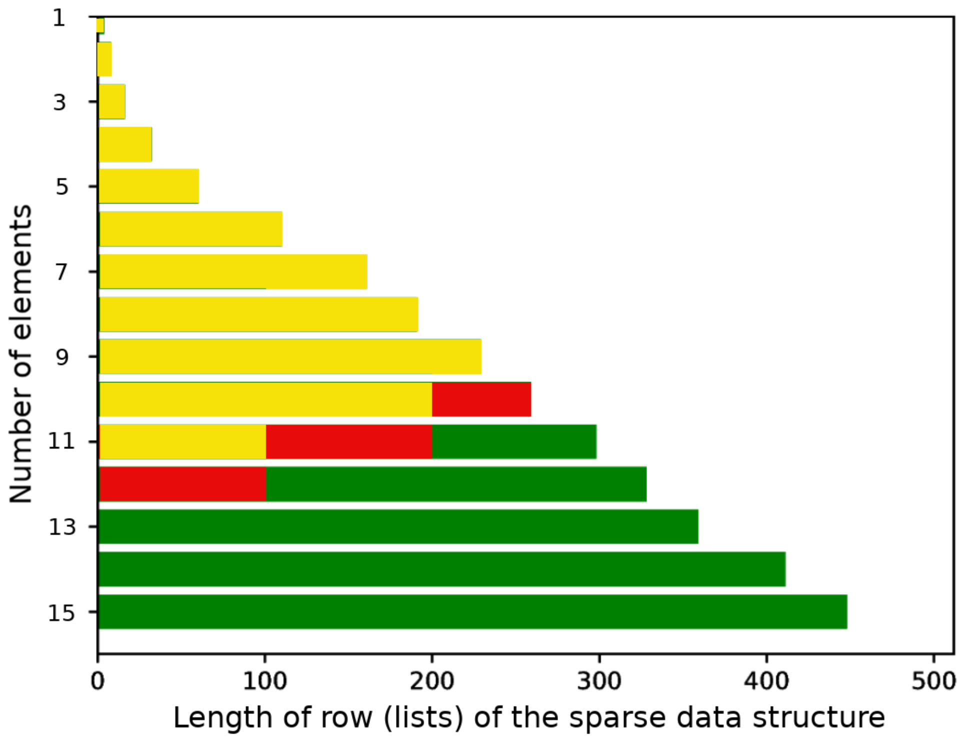


Figure 5.1: Coarse-grained parallelization of SKPDP; $N = 16, C = 512, \sigma = 0.001, \lambda = 2$.

Figure 5.1 illustrates the idea of pipelining in the context of SKPDP for a small knapsack problem instance with $N = 16$ and $C = 512$. The horizontal bars represent the size of each row of the sparse table. The yellow colored parts are already calculated; the red blocks are being calculated in parallel and the green parts are yet to be calculated. Each thread P_k is a producer for the thread P_{k+1} and consumer of the thread P_{k-1} where $0 < k < P$. P threads calculate P rows at a time. For every pass of P threads, the first thread reads an entire input row that is saved in the memory and the last thread writes an entire output row. In other words, every P_{th} row of the sparse table is stored in the slow memory. The rows in-between are computed in small chunks. Every thread in-between maintains an input buffer and an output buffer to facilitate the producer-consumer relation between the preceding and the succeeding threads. Every thread also maintains a local buffer of size w_i. The local buffer is necessary because in the Add-Merge-Kill algorithm the two elements that we are comparing can be w_i apart. The local buffer is implemented as a FIFO stack using a rotating array. Two *OpenMP* locks are used for each thread to maintain synchronization between consecutive threads, called the input lock and the output lock. Each thread uses the input lock to make sure that it does not attempt to read from an empty input buffer and uses the output lock to make sure that it doesn't write to a full output buffer. The use of the locks for synchronization is as follows. Once the output buffer is full, a thread P_k cannot write to its output buffer anymore. The thread P_k tries to acquire the input lock of the next thread P_{k+1} and dumps its output buffer into the input buffer of the P_{k+1}. Notice that here P_k acquires the input lock of P_{k+1} to ensure that P_{k+1} is not trying to read from its input thread while P_k is writing. In addition to the two locks per thread, we use flags to ensure correctness in the code.

5.0.1 Overhead Analysis

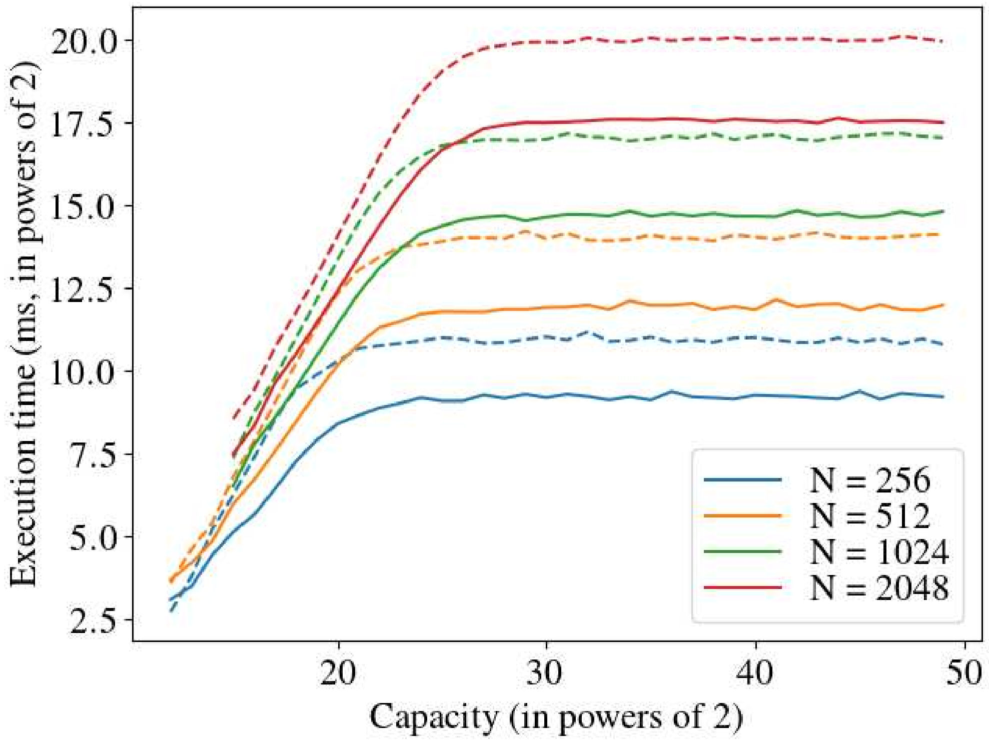
The overheads of this technique are the block-wise synchronizations and the additional memory storage requirements. If P threads maintain a buffer of w_i, then the worst-case space-complexity of this algorithm can be calculated as $O(2NC + W_{max}P)$. This means that large values of w_i can hurt the performance. Also, very large values of P will require a large memory footprint, whereas small values of P will reduce the scope of parallelization.

5.0.2 Implementation Details

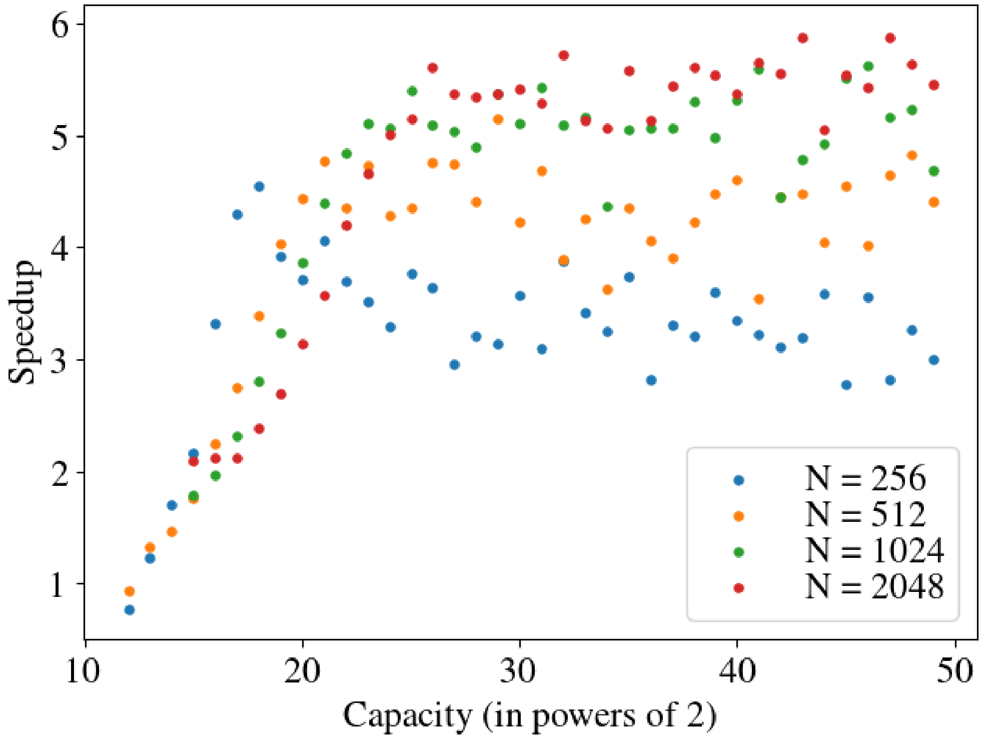
Using this technique we store the every P-th row of the sparse table into slow memory. The $(P - 1)$-th thread is responsible for storing an entire sparse row, whereas the threads 0 to $P - 2$ maintain local buffers of size w_i. Two very important parameters for this implementation are the block-size and the thread count (P). Depending on the problem instance different values of block-size and thread count performs the best. We run a grid search on these two parameters and pick the minimal execution time in our plots. Threads are created using the *OpenMP* environment variable. We use *OpenMP* locks for synchronization of threads.

5.0.3 Results

Coarse-grain parallelization performs very well on SKPDP. Figure 5.2a compares the execution times of SKPDP and it's coarse-grained parallelization. Figure 5.2b shows the performance improvement of the coarse-grained parallelization over the sequential implementation. For large problem instances, we observe more than $5x$ speedup using the coarse-grained parallelization approach on an eight-core machine.



(a) Execution times



(b) Improvement by parallelization

Figure 5.2: Comparison of the execution times of Coarse-grained Pipelined vs Sequential SKPDP(in log scale); dotted lines represent the Sequential SKPDP and solid lines represent the Coarse-grained SKPDP; for all problem instances $\sigma = 0.1\%$ and $\lambda = 8$.

Chapter 6

Future work

6.1 A model to predict the performance gain by SKPDP

At this point we know that as long as a knapsack problem instance have variance in profitability and $N \ll C$ we will get performance gain by using SKPDP. Figure 3.6 shows a pattern of how different attributes of the knapsack problem instance contribute to the sparsity. But, we don't have a model to predict whether for a given problem instance SKPDP will be better than conventional KPDP. Ideally a linear prediction model can tell us how much gain we can expect from SKPDP when used on a particular knapsack problem instance. We could potentially use Machine Learning tools to build a prediction model for this purpose.

6.2 Ordering the items by weight values to maximize sparsity

We observed that re-ordering the items of a knapsack problem instance ends up producing slightly different sparsity (i.e. different iteration counts). More experimentation is required to figure out the optimal ordering of the items to maximize sparsity of SKPDP.

6.3 Comparing SKPDP with BB algorithm

We did not compare the SKPDP with Branch-and-Bound(BB) algorithm. It would be interesting to see if BB algorithm can also perform as well as SKPDP for similar problem instances. Run time of BB algorithm is unpredictable compared to SKPDP.

6.4 Impact of other weight distributions on SKPDP

We did not explore the impact of other types of weight distribution on SKPDP. The only two distributions that we used are normal and log-normal distributions. Normal distribution is convenient to fine tune the average weight W_{avg} of the items of knapsack problem. If we want to come up with a performance model, analyzing the impact of different weight distributions is important.

6.5 Choosing the best input parameters for the coarse-grained parallelization of SKPDP

The performance of coarse-grained parallelization is dependent on the input parameters, namely buffer size and thread count. It is imperative to come up with an heuristic that can automatically pick the best input parameters depending on the knapsack problem instance. Machine Learning could be used to build a fairly accurate predictive model as long as we can collect a large amount of empirical data.

Chapter 7

Conclusion

A large subset of knapsack problem instances can benefit from utilizing the inherent sparsity of a KPDP. Especially the problem instances where the maximum capacity is much larger than the number of items, exhibit exponential sparsity as long as there is some variation in the profitability among the items. The fact that for many realistic problem instances SKPDP is exponentially better than conventional KPDP, is a significant discovery. We also show the different attributes of a knapsack problem instance that can effect the sparsity of SKPDP. We propose two parallelization techniques and present initial findings of those parallel implementations. While the proposed fine-grained parallelization technique proved to have significant overhead due to additional memory accesses, speedup achieved by the coarse-grained parallelization is exceptional. The next big step towards making SKPDP more useful is to find a model to analytically predict whether using SKPDP on a given knapsack problem instance is advantageous or not.

Bibliography

[1] Victor C.B. Camargo, Leandro Mattiolli, and Franklina M.B. Toledo. A knapsack problem as a tool to solve the production planning problem in small foundries. *Computers and Operations Research*, 39(1):86–92, 2012. Special Issue on Knapsack Problems and Applications.

[2] Robert P. Rooderkerk and Harald J. van Heerde. Robust optimization of the 0-1 knapsack problem: Balancing risk and return in assortment optimization. *European Journal of Operational Research*, 250(3):842–854, 2016.

[3] Fariborz Jolai, M.J. Rezaee, M. Rabbani, J. Razmi, and Pariviz Fattahi. Exact algorithm for bi-objective 0-1 knapsack problem. *Applied Mathematics and Computation*, 194(2):544–551, 2007.

[4] R. Andonov and S. V. Rajopadhye. A sparse knapsack algo-tech-cuit and its synthesis. In *International Conference on Application-Specific Array Processors (ASAP-94)*, pages 302–313, San Francisco, August 1994. IEEE.

[5] R. Andonov and S. Rajopadhye. Knapsack on VLSI: from algorithm to optimal circuit. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):545–561, June 1997.

[6] F. de Dinechin, D. Wilde, S. Rajopadhye, and R. Andonov. A regular vlsi array for an irregular algorithm. In *Irregular 96: Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 195–200, Santa Barbara, CA, August 1996. Springer Verlag.

[7] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Heidelberg, 2004.

[8] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementation*. John Wiley and Sons, 1990.

[9] T. C. Hu. *Integer Programming and Network Flows*. Addison Wesley, 1969.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[11] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[12] K. Nibbelink, S. Rajopadhye, and R. McConnell. 0/1 knapsack on hardware: A complete solution. In *ASAP 2007: 18th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Montréal, Québec, Canada, 2007.

[13] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *J. ACM*, 21(2):277–292, April 1974.

[14] P. Quinton, S. V. Rajopadhye, and T. Risset. Extension of the ALPHA language to recurrences on sparse periodic domains. In *IEEE Conference on Application-specific Systems, Architectures and Processors*, Chicago, IL, Aug 1996.

[15] S. Y. Kung. *VLSI Array Processors*. Prentice Hall, 1988.

[16] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. B. Rao. Wavefront array processor: Language, architecture and applications. *IEEE Transactions on Computers*, C-31:1054–1066, 1982.

[17] David Pisinger. Where are the hard knapsack problems? *Comput. Oper. Res.*, 32(9):2271–2284, September 2005.

[18] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Efficient parallelization using rank convergence in dynamic programming algorithms. *Commun. ACM*, 59(10):85–92, September 2016.

[19] Hammad Rashid, Clara Novoa, and Apan Qasem. An evaluation of parallel knapsack algorithms on multicore architectures. In *CSC*, 2010.