CSC726 Assignment 1

Dylan King and Gabriel Marcus

October 3, 2019

For this assignment, a dynamic programming algorithm was used to find solutions to the knapsack problem with a given capacity and number of items. Here, we assume that the reader is familiar with the dynamic programming solution method using $K[i, j]$, where $K[i, j]$ is the value of the optimal solution using the first j items and a maximum weight of i. In the first part of this exercise, a sequential algorithm was run to find solutions using two different methods: a memory inefficient approach that constructs the entire array K, and an efficient approach that does step by step column calculations. For reference, using the sequential algorithm, a capacity of 30000 and number of items equal to 5000, the running times for memory inefficient solution, memory efficient value and memory efficient solution were 9.2, 3.5 and 8.1 seconds, respectively.

The first modification made to the program was an attempt at cache optimization. Specifically, the build_table function was edited so that the for loop indexed by j, which traverses the columns of the matrix, was set as the outer loop. The inner loop, iterating over each row in a fixed column, could then be computed in a column major fashion and exploit the spatial locality in the cache. That is, since elements in the same column are adjacent in memory, loading values within one column reduces the number of required memory transfers. Implementing this change led to improved performance for the memory inefficient solution computation with a completion time of 4.5 seconds. However, very slight increases in completion times were observed for the memory efficient value calculation (3.7 seconds) and memory efficient solution calculation (8.9 seconds) simply due to fluctuations in the cluster itself.

To make the program run even faster, OpenMP was used to parallelize the cache-optimized code. In three of the functions, build_table, compute_table and findk, a pragma was added to run the code in parallel. In each case, the pragma was applied to a for loop so that the work involved could be divided up among multiple processors. In all three cases, the operation parallelized is when, for a fixed column j, we update the array at each row. This process always references smaller values of j (specifically $j - 1$; this is the same reason that the memory efficient algorithm exists) and so we may parallelize each of these entries. In the memory inefficient algorithm, this is done as we go across the entire table. In the memory efficient algorithms, this is done in the same manner as we traverse our implicit 2D table.

Results of parallelizing with a range of processor numbers are illustrated in

Figure 1, where we measure the time taken for a 30000×5000 test. From the data, one can see that all three versions of the algorithm follow a similar trend regardless of memory efficiency or the final result obtained. With a small number of processors (≤ 10), a significant speedup in computation time is clearly visible as processor number is increased. However, as the number of CPUs increases past 10 the speedup becomes negligible. Also, it appears that the memory efficient solution algorithm consistently takes the longest time regardless of processor number but nearly matches the time required for the memory inefficient solution when the processor number becomes greater than 10. On the other hand, the memory efficient value algorithm is noticeably the fastest over all processor numbers.

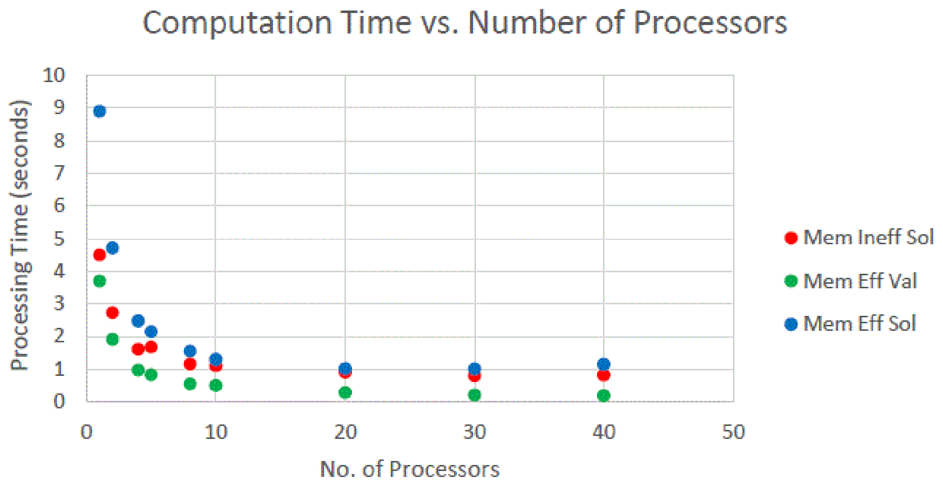

Figure 1: A plot illustrating speedup with processor number for the memory inefficient solution, memory efficient value and memory efficient solution algorithms using a 30000×5000 test size.

Based on the speedup values obtained, it appears (as theory suggests) that parallelization is most effective when the number of processors is relatively small. In fact, all three data plots seem to loosely indicate the $\frac{\text{work}}{\#\text{ processors}} + \text{span}$ relationship we expect. As mentioned above, the memory efficient algorithm which finds the solution does not parallelize well. We believe that this is because the recursive call used to find the 'midpoint' of the table is not parallelizable; if it were parallelizable, each thread would require its own pair of vectors to scan the implicit table; effectively removing the memory efficiency of this algorithm.

Below we record some additional data on both the parallel and non-parallel algorithms on a larger 30000×300000 test size, where the scaling (and decay to span) can clearly be observed.

Algorithm	time
Seq. Std.	43.67
Seq. Cache	22.92
Mem. Val	19.7
Mem Sol.	46.11

# of processors	Seq. P	Mem. Val P.	Mem Sol. P
1	25.27	21.43	51.06
2	15.66	10.91	27.33
3	10.57	7.36	19.002
4	8.90	5.72	15.22
5	7.78	4.84	13.09
10	5.50	2.71	8.42
15	4.78	2.02	6.98
20	4.51	1.75	6.82
25	4.23	1.48	6.49
30	4.06	1.25	7.15
35	3.90	1.13	6.53
40	3.88	1.07	6.84