Knapsack OpenMP Project

Patrick Williams and Sarah Parsons

October 3, 2019

1 Introduction

For this assignment, we worked to optimize cache locality and to parallelize different solutions of the Knapsack problem for improved performance. For each of the three original sequential algorithms, we have analyzed the complexities and compared the speedups for the parallel updates that we added. Throughout our analysis, we determine the work and span for p processors when parallelizing the three algorithms and compare the speedups to what should be theoretically achievable.

2 Sequential Algorithm

With the original sequential code, running the three knapsack algorithms with parameters set to a capacity of 8000 and num_items at 50,000, the total computation time required around 23 seconds and 9 seconds for the memory inefficient and memory efficient algorithms, respectively. When swapping the capacity and numitems, the times changed only slightly.

After reducing the number of items to half of the original amount, the computation times also decreased by half of the original times. Similarly, after halving the original capacity, the times decreased by half.

We also experimented with unbalanced and balanced sets of parameters. We discovered that weighing the numitems parameter to be significantly larger than the capacity led to slower times for the memory inefficient algorithm, but faster times for the memory efficient algorithms.

Our findings related to the parameters are showin in Table 1. From this, we concluded that the memory efficient algorithm to find the value requires about half as much time as the memory inefficient algorithm. However, the memory efficient algorithm to determine the solution (including backtracking) requires a little less time than the memory inefficient algorithm.

3 Cache-Locality Optimization

In order to optimize cache locality, we focused on the spatial locality of the table that is constructed for the memory inefficient algorithm. With the linear indexing function li, configured for column-major indexing, we noticed that the build_table function traverses the table in row-major order. Upon identifying this performance bug, we swapped the for loops in the build_table function so that the loop that iterates over

Table 1: Computation times for each sequential algorithm for various capacities and number of items

Mem Ineff			Mem Eff Val			Mem Eff Sol		
CAPACITY	NUMITEMS	TIME (s)	CAPACITY	NUMITEMS	TIME (s)	CAPACITY	NUMITEMS	TIME (s)
8000	50000	22.72	8000	50000	8.79	8000	50000	19.97
50000	8000	25.02	50000	8000	9.26	50000	8000	21.76
8000	25000	11.22	8000	25000	4.40	8000	25000	9.99
4000	50000	10.95	4000	50000	4.24	4000	50000	9.43
50	3000000	10.20	50	3000000	2.61	50	3000000	7.25
50000	3000	9.54	50000	3000	3.52	50000	3000	8.28

the rows is first before the loop that iterates over the columns. Thus, the build-table function now performs in column-major order, which is consistent with the linear indexing function.

As we improved the cache-locality, we know that the cache complexity would also be impacted. In order to fully evaluate the effect of our changes, we would need to know the number of cache lines that are available on a node on the DEAC cluster. For L cache lines, W capacity, and j items, the cache complexity of the original row-major algorithm was *O(jW)*. However, when indexing in column-major and building the table in column-major, the expected cache complexity of the inefficient algorithm is *O(jW/L)* for computing the final result and *O(j)* for backtracking to find the path. For the efficient version, the cache complexity for filling in the table and solving for the value is expected to be *O(W/L)*, assuming that two columns fit in the cache. Otherwise, the cache complexity would be *O(jW/L)*. For finding the path, the cache complexity would be the same as filling in the table, so long as four columns of data can fit in the cache.

Table 2 shows the times of the memory inefficient algorithm for the selected sets of parameters for both the original and cache-optimized algorithms, including the ratios of the two times. The table shows that three of the four sets of parameters saw a roughly 2x speedup, while the unbalanced parameters favoring numitems saw little improvement.

Table 2: Computation times and comparisons for memory inefficient algorithm with and without cache optimization for various capacities and number of items

Cache-Locality Optimized				No Cache-Locality Optimization			
Mem Ineff				Mem Ineff			
CAPACITY	NUMITEMS	TIME (s)		CAPACITY	NUMITEMS	TIME (s)	No_Cache/Cache Time
8000	50000	11.03		8000	50000	22.72	2.06
8000	25000	5.47		8000	25000	11.22	2.05
4000	50000	5.33		4000	50000	10.95	2.06
50	3000000	10.15		50	3000000	10.20	1.01

In conclusion, it can be shown that the cache-locality optimization for the memory inefficient algorithm makes a relatively significant impact on the computation times. Given our theoretical analysis, we would expect the improved times to decrease by a factor of L from *O(jW)* to *O(jW/L)*. Our times consistently decreased by a factor of 2, and despite not knowing the number of cache lines, we can argue that we are now leveraging the cache. However, knowing that the theoretical complexity for cache-locality assumes constants, we understand that there are most likely more than two cache lines on a node. We did not consider cache locality for the backtrack function of the memory inefficient algorithm due to the various dependencies of the function and the overhead needed to account for such dependencies.

4 Parallelization

Using OpenMP, we parallelized the memory inefficient algorithm by improving the build_table function with a '#pragma omp parallel for' clause in the inner for loop iterating over the rows of the table. By doing so, we parallelized the process of filling in each column of the knapsack table. For the memory efficient value algorithm, we parallelized the inner for loop of the compute_table function iterating over the rows of the table. Finally, for the memory efficient solution algorithm, we parallelized the inner for loop iterating over rows in the findk function used in the backtrack_implicit function.

To test our parallelization efforts, we recorded strong scaling results for a capacity of 8,000 and numitems of 50,000 for an increasing number of threads, and then did the same after swapping the parameter values. As shown in the tables and figures below, the computation time for almost all algorithms saw significant decreases as the number of threads increased. The exception was the memory efficient solution algorithm, which saw a significant slowdown after 16 threads only for capacity=8000 and numitems=50,000, and not the inverse set of parameters. The speedups of these algorithms are also displayed in Tables 4 and 6.

Figure 1: Plot of computation times for increasing number of threads used on parallelized algorithms for a capacity of 8,000 and 50,000 items

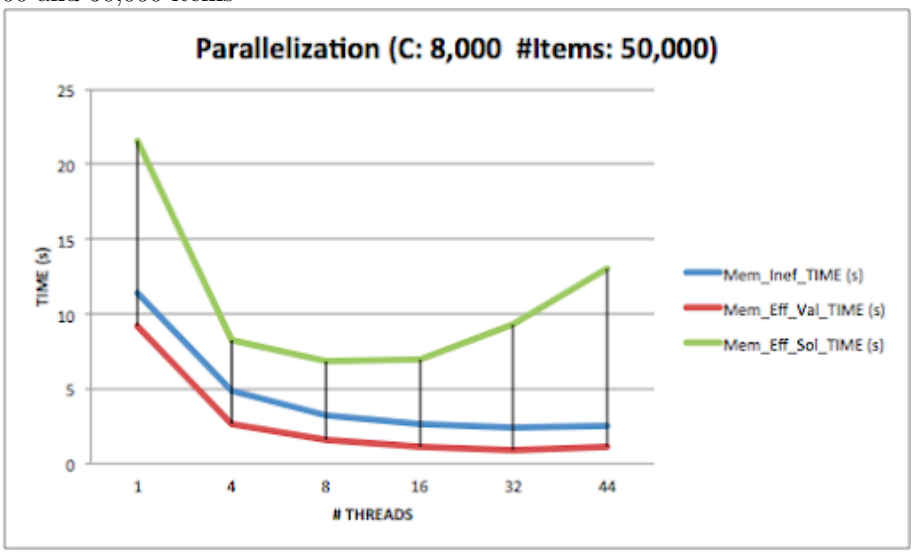

Table 3: Computation times for increasing number of threads used on parallelized algorithms for a capacity of 8,000 and 50,000 items

			Mem Inef	Mem Eff Val	Mem Eff Sol
CAPACITY	NUMITEMS	# THREADS	TIME (s)	TIME (s)	TIME (s)
8,000	50,000	1	11.42	9.23	21.6
		4	4.85	2.6	8.28
		8	3.2	1.6	6.85
		16	2.62	1.09	6.95
		32	2.4	0.9	9.31
		44	2.54	1.1	13

Table 4: Speedup and efficiency calculations for increasing number of threads used with parallelized algorithms for a capacity of 8,000 and 50,000 items

			Mem Inef		Mem Eff Val		Mem Eff Sol	
CAPACITY	NUMITEMS	# THREADS	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
8,000	50,000	1	--	--	--	--	--	--
		4	2.35	0.59	3.55	0.89	2.61	0.65
		8	3.57	0.45	5.77	0.72	3.15	0.39
		16	4.36	0.27	8.47	0.53	3.11	0.19
		32	4.76	0.15	10.26	0.32	2.32	0.07
		44	4.50	0.10	8.39	0.19	1.66	0.04

Figure 2: Plot of computation times for increasing number of threads used on parallelized algorithms for a capacity of 50,000 and 8,000 items

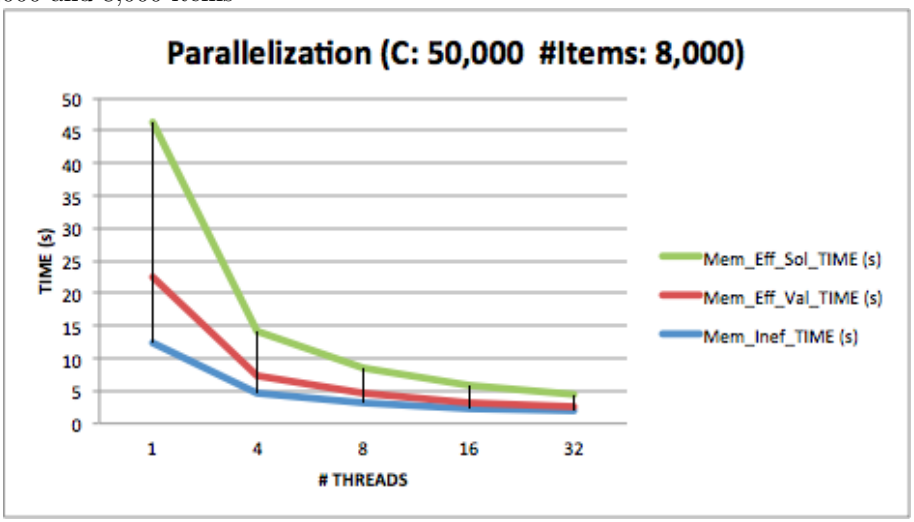

Table 5: Computation times for increasing number of threads used on parallelized algorithms for a capacity of 50,000 and 8,000 items

			Mem Inef	Mem Eff Val	Mem Eff Sol
CAPACITY	NUMITEMS	# THREADS	TIME (s)	TIME (s)	TIME (s)
50,000	8,000	1	12.3	10.07	23.91
		4	4.68	2.75	6.62
		8	3.08	1.44	3.88
		16	2.35	0.86	2.58
		32	1.96	0.49	2.02

Table 6: Speedup calculations for increasing number of threads used with parallelized algorithms for a capacity of 50,000 and 8,000 items

			Mem Inef		Mem Eff Val		Mem Eff Sol	
CAPACITY	NUMITEMS	# THREADS	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
50,000	8,000	1	--	--	--	--	--	--
		4	2.63	0.66	3.66	0.92	3.61	0.90
		8	3.99	0.50	6.99	0.87	6.16	0.77
		16	5.23	0.33	11.71	0.73	9.27	0.58
		32	6.28	0.20	20.55	0.64	11.84	0.37

5 Discussion

As evidenced in the declining computation times, the complexity of each algorithm did improve by our parallelization efforts. We can analyze the complexities of work and span as we would for a PRAM model. Our parallel changes to the build_table function in the memory inefficient algorithm should have theoretically improved the work from $O(jW)$, for j items and W capacity, to time $T1/p = O(jW/p)$, for p processors. It is important to note that the span for this algorithm is $O(jlog(W))$, which is performed when the threads are spawned for each column in the parallel for loop. Since this must be done for each item, this cost is not trivial. Thus, $O(Tp) = O(jW/p) + O(jlog(W))$. Based on our results, we see that the speedup was only around half of the number of threads for thread counts of 4 and 8. As we increased the thread count, the ratio of thread count to speedup began to decline. Thus, we were not able to achieve linear speedup. This

behavior is expected due to the sequential processes that make up the memory inefficient algorithm (e.g. syncing threads upon completing parallel computations for each item, completing the backtrack function sequentially, and initializing the first column of the table sequentially). This decrease in speedup is expected however, as syncing produces a large cost, especially when the number of items increases and when the number of threads increases.

Furthermore, the speedup for the memory efficient value algorithm has a higher improvement rate than the other algorithms as there is no sequential backtracking necessary (recursively or iteratively). Thus, the bulk of the work in the compute_table function can be performed in parallel with little sequential processing required. For the memory efficient value algorithm, the compute_table function was implemented with a parallel for loop on W, which reduced the work from *O(jW)* to *O(jW/p)*, similar to the build_table function. The span remained the same as the memory inefficient algorithm, *O(jlog(W))*, since the outer for loop was not parallelized and all threads must sync upon the computation of each column. Therefore, *O(Tp) = O(jW/p) + O(jlog(W))* for the memory efficient value algorithm. Our results show that the speedup and efficiency for this algorithm is the greatest of the three algorithms for each thread count set. As a result of the use of the backtrack function in the memory inefficient algorithm, we did not expect to achieve as large of a speedup as the memory efficient value algorithm, where there is no backtracking process and where the compute_table function has been parallelized. The memory efficient solution algorithm performs backtracking with the backtrack_implicit function, which consists of recursive calls, and thus, requires more processing.

Finally, for the memory efficient solution algorithm, the findk function was parallelized in the inner for loop iterating over rows of the table. Consequently, the work needed to traverse the table was theoretically reduced from *O(jW)* to *O(jW/p)* (multiplied by a constant since the problem was divided recursively). Since the findk function dominates the time complexity of the backtrack_implicit function, our parallel changes made a noticeable difference in computation times for the third algorithm. Unlike the other algorithms, the span for the memory efficient solution algorithm is more complex because of the recursive calls to findk in the backtrack_implicit function. With this recursive algorithm, we can compute the span using a binary tree with a depth of *log(j)* and a sum of *jlog(W)* across each level. Therefore, the span is *O(jlog(j)log(W))*, which is a factor of *log(j)* greater than the memory inefficient and memory efficient value algorithms. So, *O(Tp) = O(jW/p) + O(jlog(j)log(W))*. The greater span for this solution is reflected in the total times recorded for each thread count when comparing the three algorithms. Although slightly slower in total time for the same capacities and numitems, the memory efficient solution algorithm had greater speedups and efficiencies than the memory inefficient algorithm. We observed that this was due to the fact that the memory efficient algorithm is based on the backtrack_implicit function while the memory inefficient algorithm does not take advantage of the same midpoint efficiencies, and also uses a sequential backtrack function. As with the other algorithms, we did not achieve linear speedup for the memory efficient solution algorithm. This was primarily due to the thread syncs required in the findk function and the unparallelized recursive nature of the backtrack_implicit function.

Based on our results for both sets of tested parameters, we determined that when the capacity is larger than numitems, speedup continues to improve as more threads are added. However, when the capacity is smaller than the numitems, there is a point at which the speedup begins to decline. We argue that the reason for this behavior is due to the fact that only the inner for loops in each algorithm is parallelized. After processing each column, all threads must sync before moving to the next column. With greater volumes of items, this causes a delay in the performance given the time needed to sync all threads for each additional item. Consequently, the time required to sync begins to build up as more threads are added to the problem. Eventually, the sync time supersedes the time for parallel work for each column and the speedup decreases. On the contrary, when there are fewer columns than rows, and thus fewer syncs than parallel regions, the time to parallelize outweighs the time to sync and the speedup continues to improve as more threads are used.

The backtrack and convert_path_to_used functions were not changed, and thus, were expected to maintain a work complexity of *O(j)* for j items, as they traverse each column in the table sequentially. With linear time, these processes were considered trivial in our analysis of the complexity and so were not included in our discussion.

6 Future Work

There are many ways for us to continue work on this project, as we did not achieve perfect parallel speedup. To improve speedup, we could attempt to parallelize the columns and the rows by separating the table into blocks, and letting each thread work on a different block of the table at a time instead. To do this, we would need to implement a pipeline approach, where each thread would be waiting on adjacent blocks to be computed before continuing, but this would still give more efficiency than syncing before each column. This would contribute to solving the issue of waiting for threads to sync before continuing with calculations.

For the memory efficient (value and solution) algorithm, we parallelized the findk function but did not attempt to parallelize the recursive aspect of the backtrack_implicit function. To do this, we discovered that we would need to make use of OpenMP concepts that we were not familiar with, such as task and taskwait. Doing further research into utilizing OpenMP more would open more possibilities for making the algorithms more efficient.