CMPT 431 Project Report

Gonzalez, Juan Antonio   Kong, Raymond   Marin Estrada, Adrian

December 4th, 2024

1 Introduction

The development of parallel algorithms is inefficient and time-consuming. The performance on large datasets is often poor and requires a substantial amount of resources that end up making them expensive and not necessary. With the advancement of technology, speedup and optimization can be achieved by utilizing multiple threads and processors. The problem our group is solving is the one-dimensional 0/1 Knapsack Problem. It is a combinatorial optimization problem, which is a type of optimization problem that involves finding the best solution from a finite set of possible solutions. This problem has many real-world applications, which include space management, scheduling, and resource allocation.

2 Background

The Knapsack Problem can be defined as follows. Given a capacity C and a set S of n objects, each object i has weight w_i and profit p_i. The objective is to select a subset s of S such that the total profit is maximized and the total weight does not exceed C. Given a solution vector v = [v_1, v_2, . . . , v_n], v_i = 0 if object i is in the solution and v_i = 1 if it is in the solution. This ensures that objects can only be used once or not at all. In short, we wish to find a subset s of n items such that their summed weight is less than or equal to C, and that their combined value is maximized with no partial items being included.

$$\sum_{i=1}^{n} p_i v_i \to max$$

$$\sum_{i=1}^{n} w_i v_i \leq C$$

$$x_i \in \{0, 1\},$$

$$1 \leq i \leq n.$$

Picture on 12.2.2[2]

In layman's terms: each iteration of calculations takes a single item x of weight w_i and value p_i. For each weight value from 0 to capacity, we check if the previous iteration's max profit at that weight is higher or lower than if we added item x by adding p_i and p_{w-i} of the last iteration. We can then acquire a maximum profit for all weights and find the max profit we can hold considering all items do not exceed the capacity.

With the program being well-known as a NP-hard problem, only a few algorithms are able to solve this problem optimally[3]. It should be noted, using the approach above requires only the resulting value of the previous iteration of calculations and takes up O(n x W) Time and Space. This is known as Bellman's Principle of optimality and is said to be the greatest way to solve the Knapsack problem[5]. This way of solving makes the problem known as a serial monadic formulation. This is a lot better than the brute force approach by considering all 2^n possible solutions and finding the max of that.

Before Bellman's principle of optimality, another solution to the Knapsack problem was to use the Two-List algorithm presented by Horowitz and Sahni [4]. This algorithm divided capacity W into W_1 and W_2 equally. It would then create N subset sums for W_1 and W_2 respectively and find two sums that added to W. This is a lot slower as finding as many sums for W_1 and W_2 would take 2^w time complexity.

3 Implementation Details

3.1 Serial Implementation

We found a variety of serial implementations of the knapsack program using recursive and dynamic approaches. The dynamic approach we chose from GeeksforGeeks[1] improves upon standard bottom-up approach in dynamic programming by turning the 2D array used to store the max profits at capacity c such that $1 \leq c \leq C$ into a 1D array of size C. This was done by taking advantage of the fact that each iteration checking whether object v_i should be added to capacity c only uses the previous round as reference, allowing for a more space efficient method while maintaining the time complexity of $O(N * C)$.

We further improved the serial algorithm by eliminating the last-round calculations for all values of c such that $1 \leq c < C$, as only C needs to be calculated to acquire the final value.

3.2 Parallel Implementation

3.2.1 Using multiple matrices

Unlike the serial implementation, the parallel implementation requires of one additional array. One array would hold values being calculated using the current item being looked at, and the other one holds the maximum values from the previous item iteration. This is done because while one thread is trying to get values from the current array, another one could be editing that same value. Thus, creating a race condition and producing errors in the calculation.

Nevertheless, using one array for the previous and another for the current would require swapping the arrays at the end of each iteration. To achieve this, two barriers had to be used, one to prevent swapping while other threads were still working, and another one to prevent threads from continuing the next iteration while swapping was still in progress. These barriers significantly increased overhead, so we came up with a solution to only require of one. Instead of two arrays, we used three. For every item iteration, a thread overwrites a different array in round-robin fashion and, to calculate its values, it uses the array used in the last item iteration and blocks progress until all threads are done with that iteration. For example, we have arrays a_1, a_2, a_3. The first item iteration writes on a_2 while referring to values a_1 to reference the previous iteration. The second writes on a_3 while referring to a_2. The third writes on a_1 while referring to a_3. This process is repeated in future operations and makes it so we continue to overwrite an array without worrying about interfering with other threads progress.

3.2.2 Work Division

To divide the work between threads, each thread works on a specific weight range. For example, if the number of threads t is 2 and the capacity c is 10, the first thread would work in weights 0 through 5 and the second one would work in weights 6 through 10. At the end of every item iteration, the previous item iteration array is swapped with the current one by the root thread (thread with id 0) while the other threads wait at a barrier to synchronize their work. Moreover, the aforementioned weight range can be fixed (i.e., splitting the capacity c into t equal parts), or it can be tuned with granularities. By this we mean: instead of dividing the weight ranges into t ranges, every thread t_i takes on a weight range of granularity g items; whenever t_i is done working on its current weight range, it takes the next available g items to work on. Building up on the previous example, with a granularity g of 3, the first thread would first work with weights 0 through 2 while the second thread works in items 3 through 5. The first thread is a lot faster than the second one, so it finished with its work. It then takes items 6 through 8. Because it is so fast, it finished with this range before the second thread is done with its initial range, so it takes the following items 9 through 10.

This was done with the purpose of minimizing the amount of time that threads are waiting for other threads to finish their work. Thus, keeping them busy. Moreover, to achieve this, we used an atomic variable with the initial index of the next weight range which is updated using a compare-and-set (CAS) operation.

3.3 Distributed Implementation

3.3.1 General Idea

Given access to P processes, we distributed the work of the final solution vector evenly between the P processes such that each processes is in charge of at least $\lfloor \frac{capacity+1}{P} \rfloor$ indexes of the solution vector; capacity being incremented by 1 to account for the 0th index. Should $\lfloor (capacity + 1) \bmod P \rfloor \neq 0$, the remainder will be evenly split among the processes.

Similarly to the memory efficient serial version of the Knapsack problem we are parallelizing we can take advantage of the relation between the calculations of each iteration of the solution vector using only its preceding calculations so that we may work with two vectors. One vector will contain the results of the previous iteration, and the other will be used to calculate the current iteration. With each process aware of their start & end indexes in the solution vector, only one P^{th} the work to do. At the end of each round, processes will have the opportunity to synchronize.

Considering the distributed implementation of the knapsack problem, each process will store the weights and profits of all objects in its respective columns. In the j^{th} iteration, getting $F[j-1, w]$ can be done locally but $F[j-1, w-w_i]$ must be fetched from another process. In IPC[2], circular shift is used to fetch this process which uses global communication operations that provide simultaneous, one-to-one data redistribution. This gave us some insight into how to solve the problem with MPI and how we could use available functions to communicate between processes effectively.

3.3.2 Initial Point-to-Point Communication

The initial implementation aimed at minimizing data being sent between processes. Every time a value v_i was changed in the solution vector, both the index of the change and the new value would be noted in an array *changes_to_make*. At the end of a round, the synchronization process would begin. Note how no single v_i of the solution vector would ever be dependent on any other individual solution v_j where $i < j$. Therefore, it is unnecessary for any process P_j to send their changes to another process P_i (given $i < j$).

Thus, we synchronized starting with the root processes: P_i would send the size of *changes_to_make* to P_{i+1}, and P_{i+1} would receive the vector as *changes_to_process*. P_{i+1} would in turn send the changes made in P_i & P_{i+1} to P_{i+2}, P_{i+2} would send the changes its received as well as the changes its made to P_{i+3}, and so for up until P_{world_size-1}.

After a process P_i had received the previous process's (P_{i-1}) changes in their *changes_to_process* vector, P_i would iterate through *changes_to_process* and apply the changes to their vector solution. This process was done using Point to Point communication in MPI and would repeat for every round up until the very last line, where only the final capacity was calculated.

This however became problematic due to the serial nature of the synchronization process. While the work itself was parallelized we were bottle-necked by by the synchronization being linear resulting in slower times, as well as dead-locking due to the difficulty in coordinating the point to point communication

3.3.3 Move to Many-to-Many Communication

By removing the processes from point-to-point communication to many-to-many communication, we were able to ensure correctness of our program. Using MPI_Gatherallv(), each process was able to ensure that all other processes were aware of the work being done. However our first attempt at using many-to-many communication was still flawed. While the program was correct and consistent, due to the overhead of each process communicating the entirety of their allocated section in the solution array to other processes, our distributed implementation averaged from 2 to 10 times slower than the original serial implementation.

3.3.4 Final Implementation

To improve upon our implementation, we combined the idea of using many-to-many communication along with the original usage of providing each process with a sparse array of the changes to be made from the other processes. During the synchronization phase, each process would communicate the size of their sparse array *changes_to_make* to all other processes using MPI_Gatherall(), and receive other processes' response in *num_changes_per_process*. Using the new information, each process can allocate the space needed to receive the sparse arrays from all other processes, as well as calculate the displacement array needed for $MPI\_Gatherallv$, allowing for us to greatly improve our throughput.

4 Evaluation

4.1 Methodology

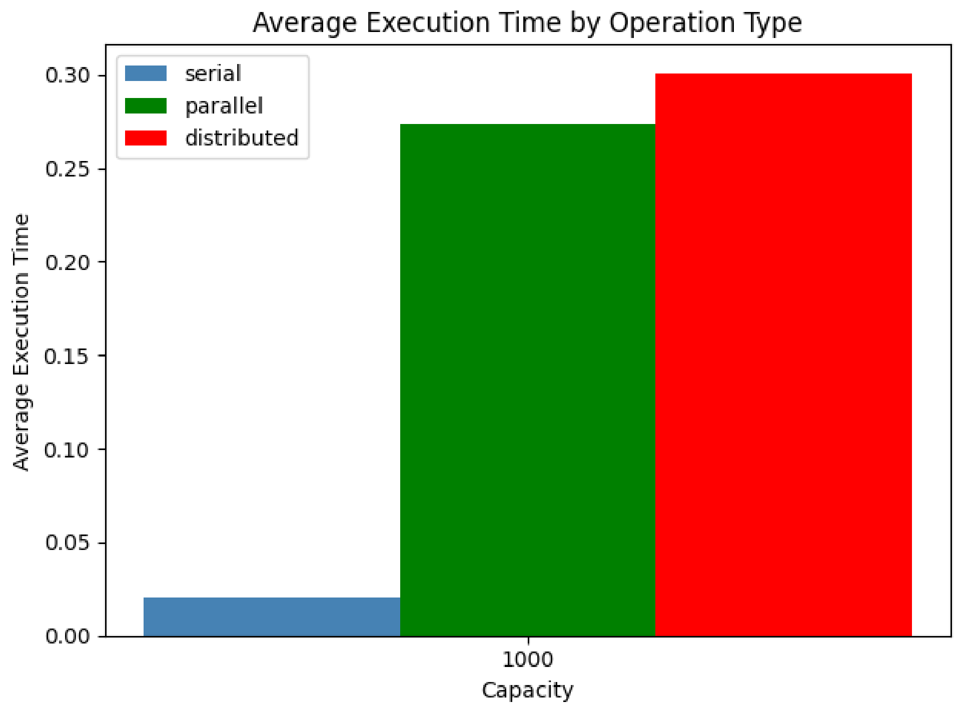
To test our thread & MPI parallelizations, we tested each of our available implementations 3 times on a variety of different criteria on SFU's Slurm Cluster, and took the average of each set of 3 runs with each test using the file *hundred_thousand_input.txt*. The file consists of one hundred thousand items of with weights ranging from one thousand units to ten thousand units, and values ranging from 1 unit to ten thousand units.

Each set for our parallelizations consisted of permutations of capacities available in the knapsack = {1e3, 1e4, 1e5, 1e6} and the number of cores being used {2, 4, 8}. In addition to this, our parallelized version also included permutations of granularities being used = {no_granularity, fine_grain_granularity, course_grain_granularity}. We defined no-granularity as defining a constant weight range for threads to work from thread initialization. We defined fine-grain-granularity as 10 for capacity 1e3, 100 for capacity 1e4 and 1e5, and 10000 for capacity 1e6. We defined course-grain-granularity as 100 for capacity 1e3, 1000 for capacity 1e4, 10000 for capacity 1e5, and 100000 for capacity 1e6. This results in 12 runs for the serial version, 108 runs for our parallel implementation, and 36 runs for our distributed implementation. In total, we conducted 156 tests.

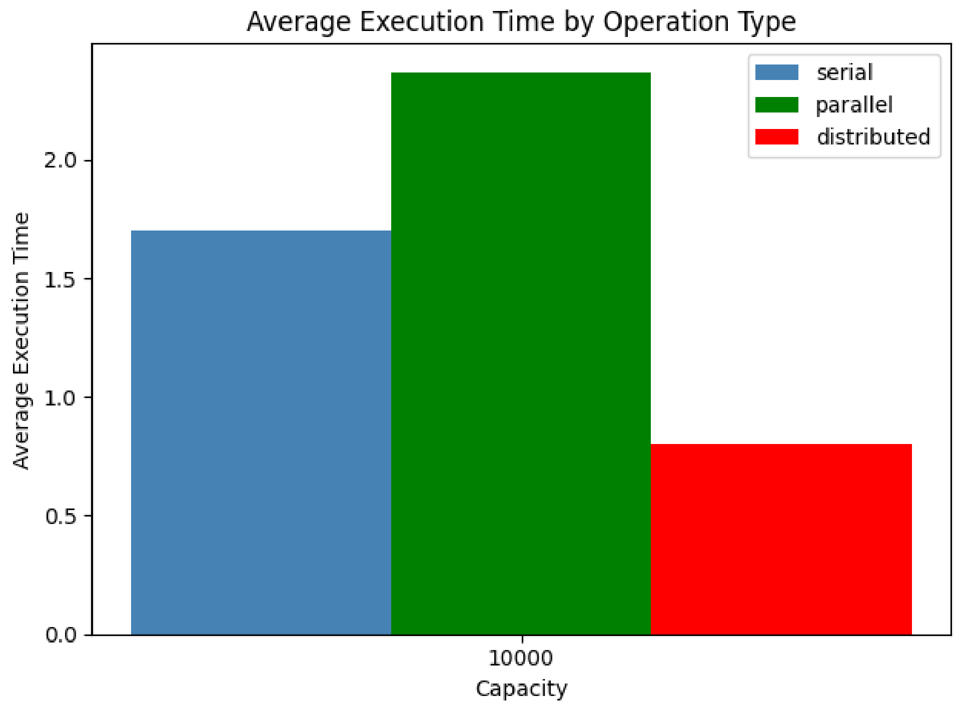
4.2 Results

We calculated our speedup using the equation of $\frac{Execution\ time\ on\ 1-processor\ System}{Execution\ time\ on\ p-processor\ system}$ and three graphs with the average execution time for each of the three capacities are presented below. For parallel and distributed executions, the fastest average execution time was included out of the different number of threads.

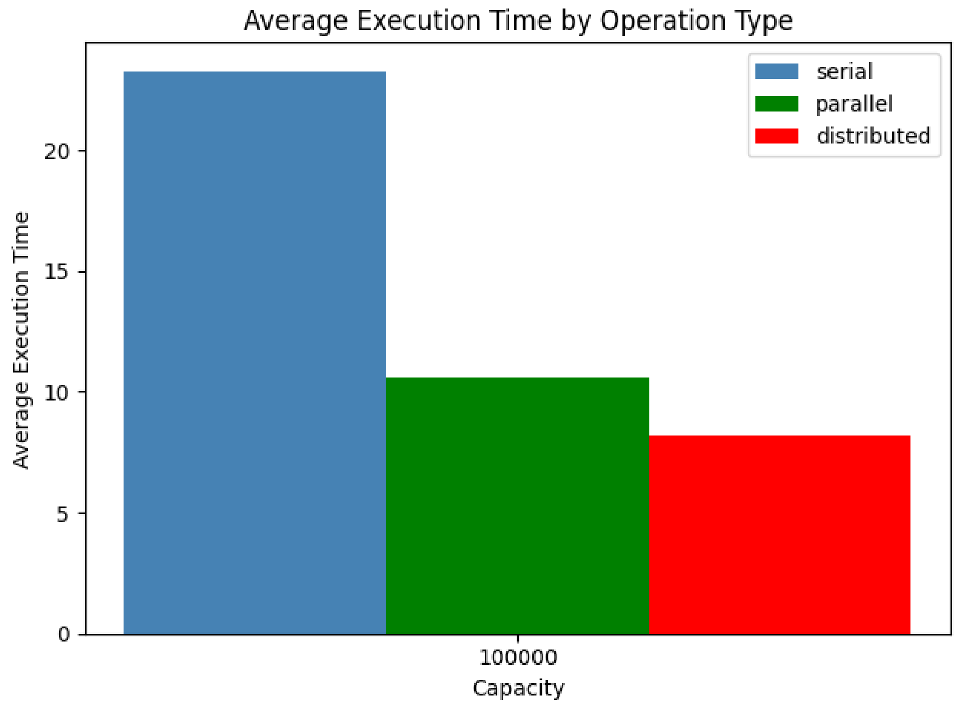
When using capacity 1e3:



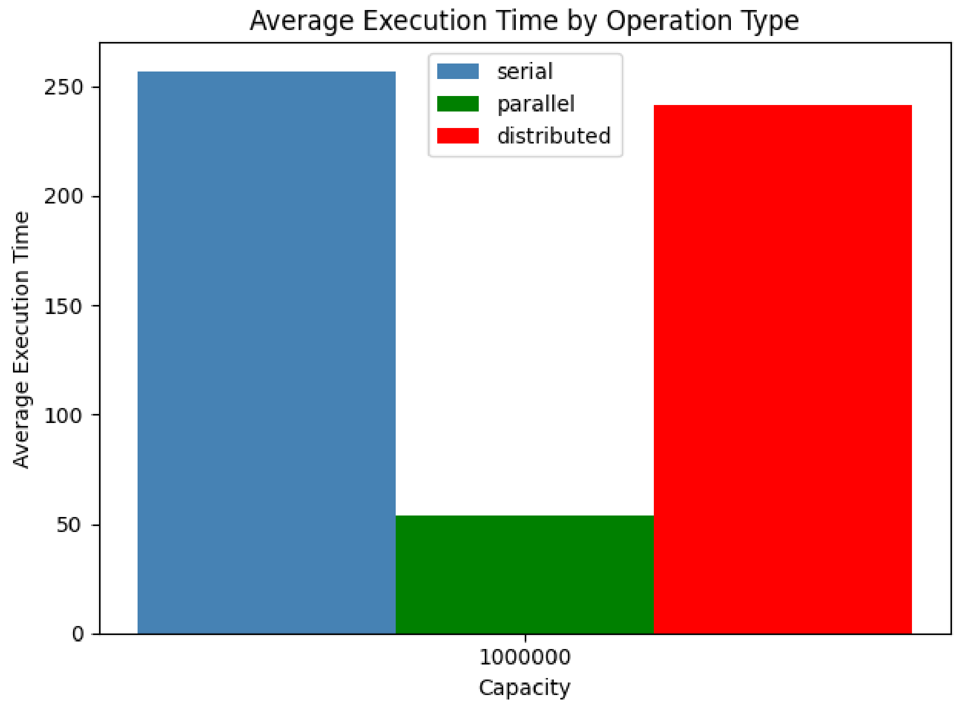
When using capacity 1e4:



When using capacity 1e5:



When using capacity 1e6:



Firstly, our tests show that our parallelized implementation has no speedup for capacities 1e3 using 2 threads and 1e4 using 4 threads. Moreover, it has a speedup of 2.19 when using 4 threads in a capacity of 1e5, and a speedup of 4.78 when using 8 threads in a capacity of 1e6. This was achieved when using the no-granularity option, which always gave consistent results. By this we mean, the other granularities would often present a higher speedup in one iteration but a huge slowdown in the next. This suggests that, on average, the work done per thread per portion of the solution vector it was assigned to for no-granularity was roughly the same. Thus, the CAS operations caused by using granularity incurred extra unnecessary overhead. However, it could also mean that we need to better adjust the granularities as some threads were left waiting at the barrier for longer than expected.

Secondly, our distributed implementation had its highest speedups when using 8 processes. The distributed implementation had speedup of: 2.1085 on a capacity of 1e4, 2.8511 on a capacity of 1e5, and 1.0644 on a capacity of 1e6. The speedup not having a one-to-one ratio with the number of processors used indicates that our parallelization of the serial knapsack algorithm is not embarrassingly parallel. This is due to the synchronization required by each thread/process in between rounds. The overhead in communication comes at a slight cost time, but still allows for greater throughput.

It is also interesting to note that using the largest capacity of 1e6 actually lowered the speedup of the distributed implementation. This is an indication of how on larger problems, there will be scenarios where every round will require multiple changes to the solution vector. By synchronizing using a sparse vector, each processor will be sending data close to twice as large as what is has been assigned as it needs to send both the index of the change and the actual change. This can potentially be improved upon by instead of sending the indexes, sending a bit vector providing the location of changes to be made alongside the actual changes.

It should be noted that for lower capacities our speedup was negligible or nonexistent. This was expected; the added overhead of communication between threads/processes in non-embarrassingly parallel implementations is larger than the speedup provided by splitting the work among P threads/processes. It is only when testing on larger capacities that the division of work surpasses the cost of the overhead. This highlights while parallelization is important, it should be used when the situation calls for it.

5 Conclusion

We were able to successfully parallelize the efficient knapsack solution we found. Our multi-threaded implementation showed an increasing speedup as the capacity increased reaching a speedup of 4.78 when using capacity of 1e6. Our distributed implementation had its greatest speedup being 2.8511 when using 8 processes and the knapsack having a capacity of 1e5. From our parallelization process, we learned that parallelization is not the one all be all, and should only be used when the amount of work to be split up outweighs the overhead of communication involved in splitting up the work.

References

[1] GeeksforGeeks: 0/1 knapsack problem - dp-10 (2024), `https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10` accessed: 2024-12-03

[2] Grama, A.: Introduction to parallel computing / Ananth Grama [and others]. Addison-Wesley, Harlow, England ; New York, 2nd ed. edn. (2003)

[3] Li, K., Liu, J., Wan, L., Yin, S., Li, K.: A cost-optimal parallel algorithm for the 0–1 knapsack problem and its performance on multicore cpu and gpu implementations. Parallel Computing **43**, 27–42 (2015). https://doi.org/https://doi.org/10.1016/j.parco.2015.01.004, `https://www.sciencedirect.com/science/article/pii/S0167819115000113`

[4] Qiao, S., Wang, S., Lin, Y., Zhao, L.: A distributed algorithm for 0-1 knapsack problem based on mobile agent. In: 2008 Eighth International Conference on Intelligent Systems Design and Applications. vol. 2, pp. 208–212 (2008). https://doi.org/10.1109/ISDA.2008.110

[5] Thant Sin, S.T.: The parallel processing approach to the dynamic programming algorithm of knapsack problem. In: 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus). pp. 2252–2256 (2021). https://doi.org/10.1109/ElConRus51938.2021.9396489