COMP90025 Project 1B: Knapsack Filling

Xinyan Shan (1047188) — Spartan username: sxy0322
Saier Ding (1011802) — Spartan username: Saier

Introduction

MPI (Message Passing Interface) is an application programming interface that is designed to transfer messages among processes in order to perform a task (Pacheco, 2011). It has been widely employed for parallel applications.

The knapsack problem is a classic NP-hard resource allocation problem. There are many solutions to solve this problem such as exhaustive method, recursive method, backtracking method and dynamic programming method. In this report, the experiment will adopt MPI to parallelize dynamic programming method and exhaustive method.

Hypothesis and Method

In fact, there are many different solutions, which possess various algorithm complexity, to the knapsack problem. In the case where the capacity of the knapsack is C and the number of items is n. The complexity of harnessing the exhaustive algorithm is 2^n, while the complexity of the dynamic programming method is $n * C$. Usually, the knapsack problem is classified into two categories. The first category is to employ the exhaustive method when the capacity of the knapsack (C) is large and the number of items (n) is relatively small. In the other class, the method of dynamic programming is adopted when the capacity of the knapsack is relatively small and the number of items is large (Martello, Pisinger, & Toth, 1999). Furthermore, both classifications would employ MPI functions to parallelize the mentioned algorithms to gain the best complexity and performance.

Results and Analysis

In the dynamic programming part, "*MPI_Send*" and "*MPI_Recv*" functions are invoked to achieve the parallelism by distributing the weight of the knapsack (wt) to multiple processes, and the value corresponding to the current wt is transferred by multiple processes. Through this way, the program can implement message passing between processes to increase efficiency and reduce time complexity. Besides, because of the time complexity of dynamic programming is $n * C$, so an assumption would be made that the capacity and items would have an influence on the performance. In order to prove this assumption, three experiments are conducted in this paper. The time of these experiments are presented in Figure 1, 2 and 3separately.

Items \ Nodes	2	4	6	8	12
200	71.3	58.0	38.0	37.0	29.4
400	159.8	105.6	94.5	76.5	65.6
600	306.1	208.6	151.6	123.8	97.3
800	343.7	268.1	203.4	165.6	130.5
1000	446.0	323.5	258.1	217.0	180.2
1200	479.2	405.3	342.5	255.7	217.4

Figure 1. The time (ms) of capacity=100 based on the different number of nodes and items

By observing the data in figure 1, it is straightforward to draw a conclusion that with the same capacity and the same number of items, as the number of nodes increases, the elapsed time decreases. Moreover, when the capacity and the number of nodes are the same, the running time increases with the number of items. This phenomenon indicates that the number of items and the algorithm complexity might be positively correlated. Furthermore, it also suggests that the more processes are in parallel, the higher the execution efficiency is.

Items \ Nodes	2	4	6	8	12
200	105.7	141.7	103.2	94.5	87.9
400	245.3	316.1	269.3	223.9	196.5
600	758.9	476.8	436.5	348.5	300.6
800	588.2	707.3	624.1	511.1	400.8
1000	1324.3	933.9	759.0	655.7	475.1
1200	1192.5	1129.1	952.3	791.1	604.8

Figure 2. The time (ms) of compacity=400 based on the different number of nodes and items

Comparing the data in figure 2 with figure 1, a regular pattern could be observed that as the capacity increases, the elapsed time also increases, which indicates that the capacity and the algorithm complexity are also positively correlated. However, when the capacity is particularly large and the number of items is small, the program will be relatively simple to solve knapsack problem dynamically, and the time spent would be less. Nontheless, as the number of items increases, the time

spent will be more than the program with smaller capacity. This could explain the trend of the data in figure 3.

Items \ Nodes	2	4	6	8	12
200	99.4	196.2	171.9	150.3	130.1
400	892.6	673.0	577.7	406.3	310.2
600	1641.4	1266.3	837.3	681.7	554.4
800	2704.7	1659.5	1193.0	1085.8	793.3
1000	3053.3	2165.6	1469.4	1307.7	1020.9
1200	3790.1	2828.5	1970.0	1603.7	1244.0

Figure 3. The time (ms) of compacity=1000 based on the different number of nodes and items

Conclusion

In conclusion, adopting MPI, which is characterised by its internal message passing mechanism, is an effective strategy to improve the algorithm efficiency by processing massive data in parallel. Finally, by designing and tuning algorithm for the specific knapsack issue, the knowledge of several calculation methods for solving this problem has been acquired.

Reference List

Martello, S., Pisinger, D., & Toth, P. (1999). Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science, 45*(3), 414-424. doi:10.1287/mnsc.45.3.414

Pacheco, P. (2011). An Introduction to Parallel Programming. Retrieved from https://www.sciencedirect.com