PARALLEL EXECUTION OF 0/1 KNAPSACK PROBLEM

CSE4001 – Parallel and Distributed Computing

Submited by

NADIMPALLI L NARASIMHA RAJU – 19BCE2247
K.V.V.M.SAI DIVYESH - 19BCE0841
KANNEBOINA GANESH - 19BCE2582

School of Computer Science and Engineering

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

CONTENTS

1. INTRODUCTION

1.1OBJECTIVE

The main objective of our project is to use parallelism to solve the knapsack problem. This project examines existing and popular techniques for solving the knapsack problem, such as dynamic programming and then proposes a parallel approach to solving the problem byemploying multiple threads of a computer to arrive at a solutior in the shortesttime possible.

The main objective of knapsack problem is to obtain a filling of the knapsack that maximizes the total profit.

0/1 knapsack is nothing but selecting the weights which should not exceed the maximum weight of the knapsack and also able to get the maximum profits for the weights that we can choose For example, knapsack of size=6 with (w1,w2,w3)=(2,3,4) and (p1,p2,p3)=(1,2,5) for this kind of case the best case is selecting w1 and w3 so that we can get the maximum profit of p1+p3=6.

1.2ABSTRACT

In this project we are trying to solve 0/1 KNAPSACK problem using serial and parallel programming with the help of OPENMP.

This project examines existing and popular techniques for solving the knapsack problem, such as dynamic programming and then proposes a parallel approach to solving the problem by employing multiple threads of a computer to arrive at a solution in the shortest time possible.

We are trying to make a comparison between the time taken by solving the problem using dynamic solution and using the parallel solution. Also we are trying to back our idea with statistical data from performing the experiments over data sets of different values.

At the end we planned to draw a graph between input size and execution time for both parallel and serial execution of 0/1 knapsack problem by increasing input size of the problem.

Also calculating the improvement factor to give a rough idea that our parallel execution is quite faster than serial execution.And also another graph between input size and improvement factor we obtained for every input size.

2. LITERATURE SURVEY

(i) Optimal parallel algorithms for the knapsack problems without memory conflicts.

-Li, Ken-Li, Ren-Fa Li, and Qing-Hua Li.

The authors discussed importance of the knapsack problem in the application of crypto system and in theory with the imphasis to find the techniques that could lead to practical algorithms with less run time. The paper prosed a new parallel algorithm for the knapsack problem where the merging algorithm is adopted.

While reviewing various techniques for parallelizing the algorithm using this paper context we can also optimize the algorithm for memory efficiency using openMP.

(ii) OPENMP: An Industry-Standard API for shared memory programming

- L. Dagum

The author present a new way to achieve scalability in parallel software with OpenMP, their portable alternative to message passing. They discuss its capabilities through specific examples and comparisons with other standard parallel programming model. Since we are implementing the parallelization of the 0/1 knapsack problem this paper provide necessary content to ways of applying OpenMP to increase parallelization, efficiency in your code with use of appropriate drivers.

(iii) A parallel algorithm for the knapsack problem using a generation and searching technique.

-Henry Ker Chang Chag, Jonathan Jen Rong Chen, Snyong-Jian-Shyu

The purpose of this paper was to propose a new parallel algorithm for the knapsack problem. The authors developed a generation and searching technique to derive the derived two ordered tests in the preliminary process of general knapsack problem. The proposed parallel algorithm was developed on the basis of SMID machine with shared memory. There algorithm needed 0(2*nu) memory and 0(2*n) processors.

The given paper gives the content of how parallelization can be applied to solve the category of combined problem -0/1 knapsack which are complimenting in our project.

(iv) Computing Partitions with Applications to the Knapsack Problem.

-Horowitz, Ellis, and Sartaj Sahni.

In this paper authors have investigated on all the usual algorithms for this problem in terms of both asymmetric computing time and storage requirements as well as average computing time.

They developed a technique which improved all of the dynamic programming methods by a square root factor. A new branch and bound algorithm which is significantly faster than the green tag and Hegerich algorithm was also presented.

6Since we are also comparing technique before going for the parallel solution, this paper provided the approach for the comparing of algorithm keeping knapsack problem in contest.

3. PROBLEM FORMULATION

The knapsack problem can be better understood with the help of an example.

Lets say we have a knapsack which can hold maximum weight of 10 units. We denote this weight by w. We have n = 4 items to choose from, and the monetary value of these four items are represented as as array int[] val = {10,40,30,50} and weights of these four items are represented by array int[] wt = {5,4,6,3}.

We have to pick items in such a way, that the monetory value of products picked is maximum and the weights of these products should not exceed the maximum weight capacity of the knapsack, W.

Since this is a 0/1 Knapsack problem, we can either include the item in our knapsack or exclude it, but not include a fraction of it, or include
it multiple times.

Step 1:

First, we create a 2-dimensional array (i.e. a table) of n + 1 rows and w+1 columns.

A row number i represents the set of all the items from rows 1— i. For instance, the values in row 3 assumes that we only have items 1, 2, and 3.
A column number j represents the weight capacity of our knapsack. Therefore, the values in column 5, for example, assumes that our knapsack can hold 5 weight units.

Putting everything together, an entry in row i, column j represents the maximum value that can be obtained with items 1, 2, 3 … i, in a knapsack that can hold j weight units.

Let’s call our table mat for matrix. Therefore, int[][] mat = new int[n + 1][w+1].

Step 2:

We can immediately begin filling some entries in our table: the base cases, for which the solution is trivial. For instance, at row 0, when we have no items to pick from, the maximum value that can be stored in any knapsack must be 0.

Similarly, at column 0, for a knapsack which can hold 0 weight units, the maximum value that can be stored in it is 0. (We're assuming that there are no massless, valuable items.)

We can do this with 2 for loops:

```
// Populate base cases
int[][] mat = new int[n + 1][w + 1];
for (int r = 0; r < w + 1; r++) {
mat[0][r] = 0;
}
for (int c = 0; c < n + 1; c++) {
mat[c][0] = 0;
}
```

Step 3:

Now, we want to begin populating our table. As with all dynamic programming solutions, at each step, we will make use of our solutions to previous sub-problems.

We'll first describe the logic, before showing a concrete example.
The relationship between the value at row i, column j and the values to the previous sub-problems is as follows:

Recall that at row i and column j, we are tackling a sub-problem consisting of items 1, 2, 3 … i with a knapsack of j capacity. There are 2 options at this point: we can either include item i or not. Therefore, we need to compare the maximum value that we can obtain with and without item i.

The maximum value that we can obtain without item i can be found at row i-1, column j. This part's easy. The reasoning is straightforward: whatever maximum value we can obtain with items 1, 2, 3 … i must obviously be the same maximum value we can obtain with items 1, 2, 3 … i - 1, if we choose not to include item i.

To calculate the maximum value that we can obtain with item i, we first need to compare the weight of item i with the knapsack's weight capacity.

Obviously, if item i weighs more than what the knapsack can hold, we can't include it, so it does not make sense to perform the calculation. In that case, the solution to this problem is simply the maximum value that we can obtain without item i (i.e. the value in the row above, at the same column).

However, suppose that item i weighs less than the knapsack's capacity. We thus have the option to include it, if it potentially increases the maximum obtainable value. The maximum obtainable value by including item i is thus = the value of item i itself + the maximum value that can be obtained with the remaining capacity of the knapsack. We obviously want to make full use of the capacity of our knapsack, and not let any remaining capacity go to waste.

Therefore, at row i and column j (which represents the maximum value we can obtain there), we would pick either the maximum value that we can obtain without item i, or the maximum value that we can obtain with item i, whichever is larger.

Step 4:

Once the table has been populated, the final solution can be found at the last row in the last column, which represents the maximum value obtainable with all the items and the full capacity of the knapsack.
return mat[n][w];

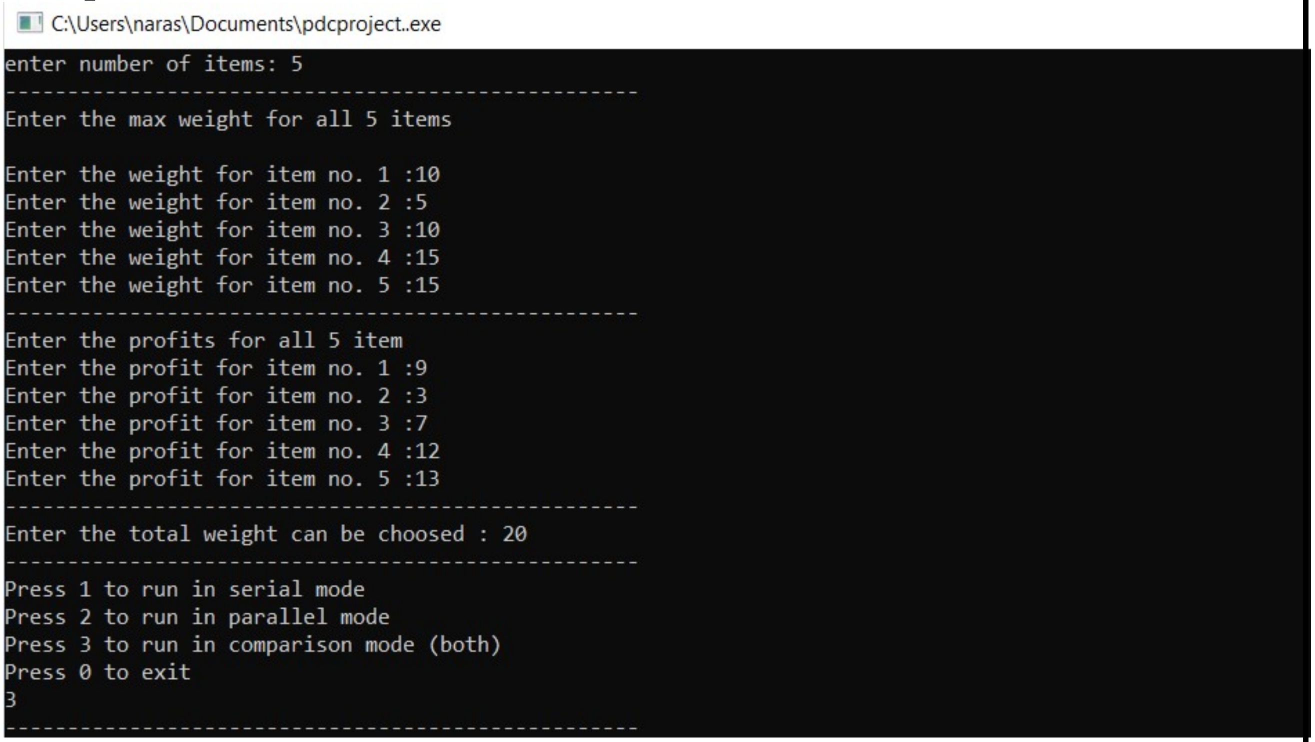
4. ALGORITHM

```
for (int item = 1; item <= n; item++) {

        for (int capacity = 1; capacity <= w; capacity++) {

                int maxValWithoutCurr = mat[item - 1][capacity];

                int maxValWithCurr = 0;

                int weightOfCurr = wt[item - 1];

                if (capacity >= weightOfCurr) {

                        maxValWithCurr = val[item - 1];

                        int remainingCapacity = capacity - weightOfCurr;

                        maxValWithCurr += mat[item - 1][remainingCapacity];

                }
                mat[item][capacity] = Math.max(maxValWithoutCurr,
                                                maxValWithCurr);

        }
```

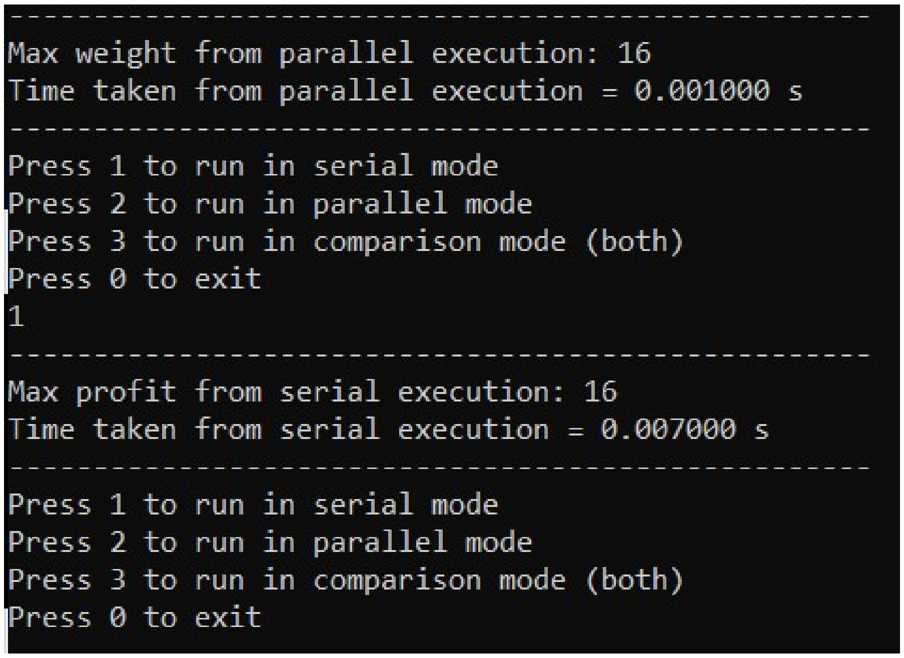
5. CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <malloc.h>
#include <time.h>
double serialExecution(int W, int wt[], int val[], int n){
int getMax(int x, int y) {
 if(x > y) {
 return x;
 } else {
 return y;
 }
}
void knapSack(int W, int wt[], int val[], int n) {
 int i, w;
 int V[n+1][W+1];
 for(w = 0; w <= W; w++) {
 V[0][w] = 0;
 }
 for(i = 0; i <= n; i++) {
 V[i][0] = 0;
 }
 for(i = 1; i <= n; i++) {
 for(w = 1; w <= W; w++) {
 if(wt[i] <= w) {
 V[i][w] = getMax(V[i-1][w], val[i] + V[i-1][w - wt[i]]);
 } else {
 V[i][w] = V[i-1][w];
 }
 }
 }
 printf("Max profit from serial execution: %d\n", V[n][W]);
}
 //int n;

 clock_t start, end;
 start = clock();
 knapSack(W, wt, val, n);
 end = clock();
 double time_taken = (double)(end - start) / (double)(CLOCKS_PER_SEC);
 printf("Time taken from serial execution = %1f s\n",(double)(time_taken));
 return (double)time_taken;
}
double parallelExecution(int W, int wt[], int val[], int n){
 int* weights;
 int* values;
 clock_t start, end;
 int i,j;
 int max(int a, int b)
 {
 return (a>b?a:b);
 }
 weights = (int*)malloc((n+1)*sizeof(int));
 values = (int*)malloc((n+1)*sizeof(int));
 for(int i=1; i<=n; i++){
 weights[i] = wt[i];
 values[i] = val[i];
 }
 int F[n+1][W+1];
 for(i=0;i<(W+1);i++){
 F[0][i] = 0;
 }
 for(i=0;i<n+1;i++){
 F[i][0] = 0;
 }
 for(i=1;i<=n;i++)
 {
 for(j=1;j<=W;j++){
 F[i][j] = -1;
 }
 }
 int parallelKnapsack(int i, int j, int F[n+1][W+1])
 {
 int value;
 if(F[i][j] < 0)
 {
 if(j < weights[i]) //If weight of item is more than current capacity
 value = parallelKnapsack(i-1, j, F); //Value of previous item
 else
 {
 int a;
 int b;
 #pragma omp parallel sections
 {
 //Two sections that can run independently and simultaneously
 #pragma omp section
 {
 a = parallelKnapsack(i-1, j, F);
 }
 #pragma omp section
 {
 b = (values[i] + parallelKnapsack(i-1, j -
 weights[i], F));
 }
 }
 value = max(a,b);
 }
 F[i][j] = value;
 }
 return F[i][j];
 }
 start = clock();
 int res;
 #pragma omp parallel
 {
 #pragma omp single nowait
 {
 res = parallelKnapsack(n, W, F);
 }
 }
 end = clock();
 double time_taken = (double)(end - start) / (double)(CLOCKS_PER_SEC);
 printf("Max weight from parallel execution: %d\n", res);
 printf("Time taken from parallel execution = %1f s\n",(double)(time_taken));
 free(weights);
 free(values);
 return (double)time_taken;
}
int main(){
 int i,n;
 printf("enter number of items: ");
 scanf("%d", &n);
 printf("--------------------------------------------------");
 int W,weights[n],values[n];
 printf("\nEnter the max weight for all %d items \n \n",n);
 for(i=1;i<=n;i++)
 {
 printf("Enter the weight for item no. %d :",i);
 scanf("%d", &weights[i]);
 }
 printf("--------------------------------------------------");
 printf("\nEnter the profits for all %d item \n",n);
 for(i=1;i<=n;i++)
 {
 printf("Enter the profit for item no. %d :",i);
 scanf("%d", &values[i]);
 }
 printf("--------------------------------------------------");
 //Read Max. Weight Capacity of Knapsack
 printf("\nEnter the max weight capacity can be choosed : ");
 scanf("%d", &W);
 void compare(){
 double serialTime = serialExecution(W,weights,values,n);
 double parallelTime = parallelExecution(W,weights,values,n);
 double factor = serialTime/parallelTime;
 printf("\nParallel execution is %lf times faster than serial execution in this
example.",factor);
 printf("\n--------------------------------------------------");
 }
 while(1){
 int choice;
 printf("--------------------------------------------------\n");
 printf("Press 1 to run in serial mode\n");
 printf("Press 2 to run in parallel mode\n");
 printf("Press 3 to run in comparison mode (both)\n");
 printf("Press 0 to exit\n");
 scanf("%d",&choice);
 printf("--------------------------------------------------\n");
 if(choice == 1){
 serialExecution(W,weights,values,n);
 } else if(choice == 2){
 parallelExecution(W,weights,values,n);
 } else if(choice == 3){
 compare();
 } else if(choice == 0){
 break;
 } else{
 printf("Invalid choice\n");
 }
 }
}
```

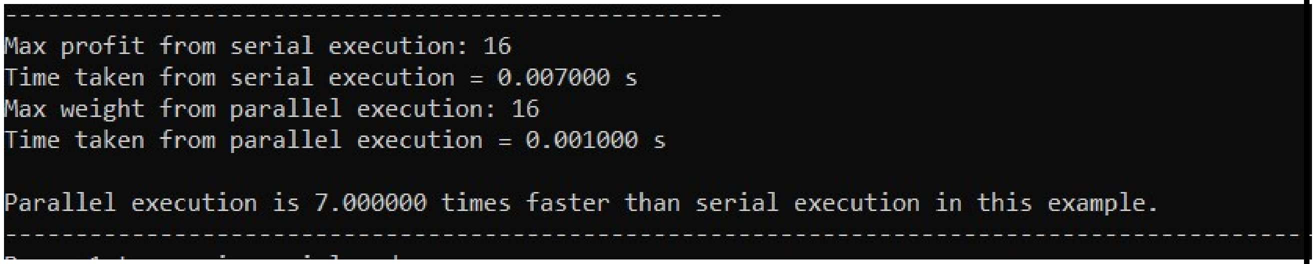
Output Screenshots:-



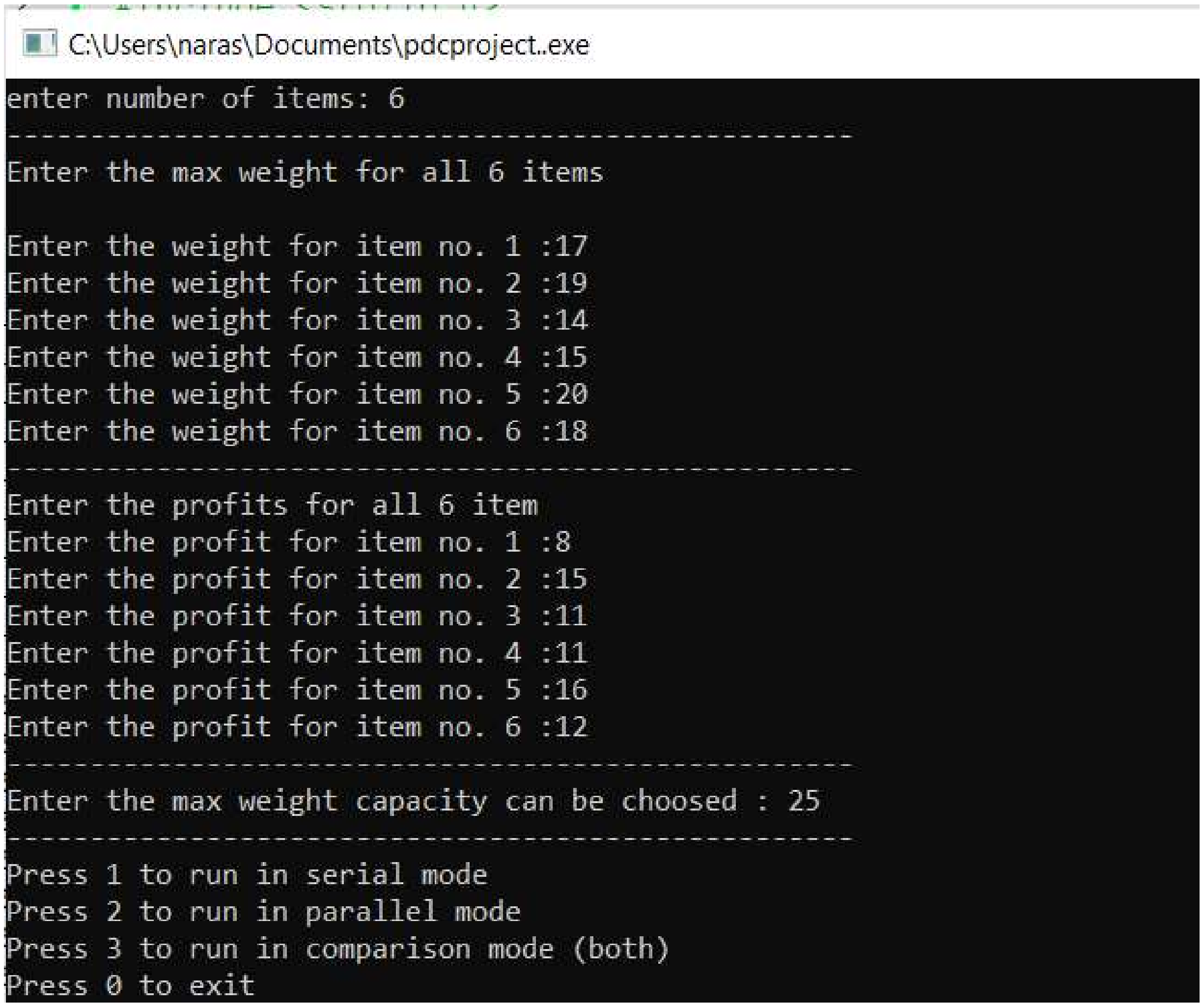
```
C:\Users\naras\Documents\pdcproject..exe
enter number of items: 5
--------------------------------------------------
Enter the max weight for all 5 items

Enter the weight for item no. 1 :10
Enter the weight for item no. 2 :5
Enter the weight for item no. 3 :10
Enter the weight for item no. 4 :15
Enter the weight for item no. 5 :15
--------------------------------------------------
Enter the profits for all 5 item
Enter the profit for item no. 1 :9
Enter the profit for item no. 2 :3
Enter the profit for item no. 3 :7
Enter the profit for item no. 4 :12
Enter the profit for item no. 5 :13
--------------------------------------------------
Enter the total weight can be choosed : 20
--------------------------------------------------
Press 1 to run in serial mode
Press 2 to run in parallel mode
Press 3 to run in comparison mode (both)
Press 0 to exit
3
--------------------------------------------------
```



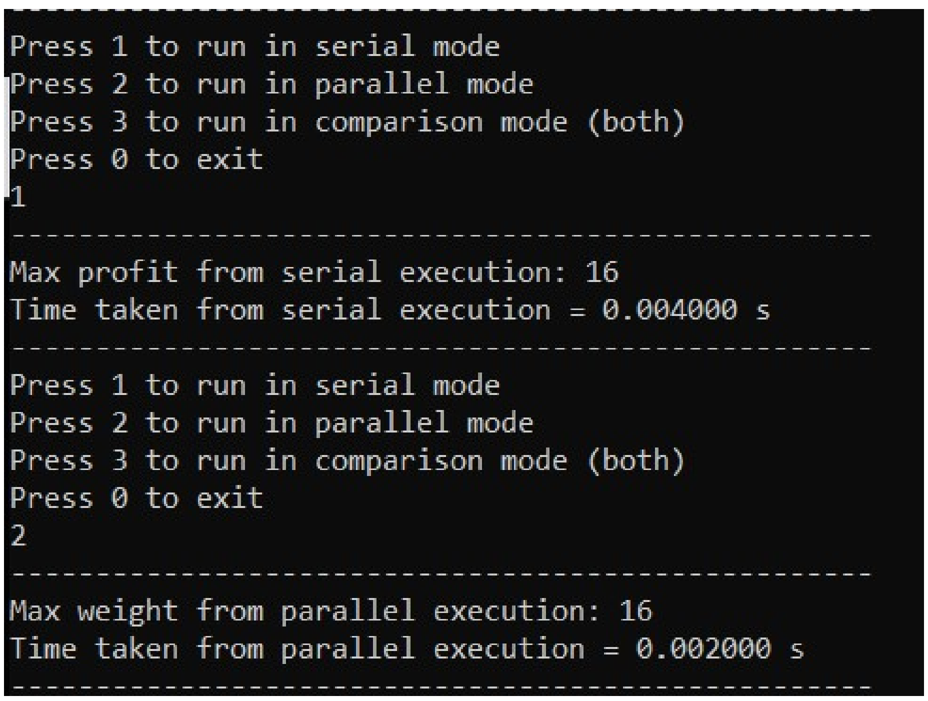
```
---------------------------------------------------
Max weight from parallel execution: 16
Time taken from parallel execution = 0.001000 s
---------------------------------------------------
Press 1 to run in serial mode
Press 2 to run in parallel mode
Press 3 to run in comparison mode (both)
Press 0 to exit
1
---------------------------------------------------
Max profit from serial execution: 16
Time taken from serial execution = 0.007000 s
---------------------------------------------------
Press 1 to run in serial mode
Press 2 to run in parallel mode
Press 3 to run in comparison mode (both)
Press 0 to exit
```



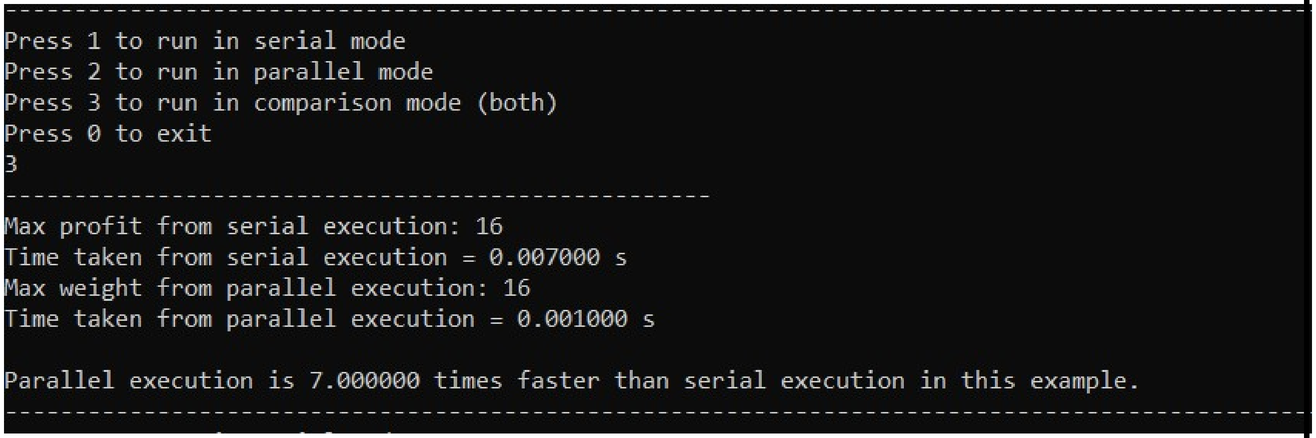
```
---------------------------------------------------
Max profit from serial execution: 16
Time taken from serial execution = 0.007000 s
Max weight from parallel execution: 16
Time taken from parallel execution = 0.001000 s

Parallel execution is 7.000000 times faster than serial execution in this example.
-----------------------------------------------------------------------------------------
```



```
C:\Users\naras\Documents\pdcproject..exe
enter number of items: 6
---------------------------------------------------
Enter the max weight for all 6 items

Enter the weight for item no. 1 :17
Enter the weight for item no. 2 :19
Enter the weight for item no. 3 :14
Enter the weight for item no. 4 :15
Enter the weight for item no. 5 :20
Enter the weight for item no. 6 :18
---------------------------------------------------
Enter the profits for all 6 item
Enter the profit for item no. 1 :8
Enter the profit for item no. 2 :15
Enter the profit for item no. 3 :11
Enter the profit for item no. 4 :11
Enter the profit for item no. 5 :16
Enter the profit for item no. 6 :12
---------------------------------------------------
Enter the max weight capacity can be choosed : 25
---------------------------------------------------
Press 1 to run in serial mode
Press 2 to run in parallel mode
Press 3 to run in comparison mode (both)
Press 0 to exit
```



```
---------------------------------------------------
Press 1 to run in serial mode
Press 2 to run in parallel mode
Press 3 to run in comparison mode (both)
Press 0 to exit
1
---------------------------------------------------
Max profit from serial execution: 16
Time taken from serial execution = 0.004000 s
---------------------------------------------------
Press 1 to run in serial mode
Press 2 to run in parallel mode
Press 3 to run in comparison mode (both)
Press 0 to exit
2
---------------------------------------------------
Max weight from parallel execution: 16
Time taken from parallel execution = 0.002000 s
---------------------------------------------------
```

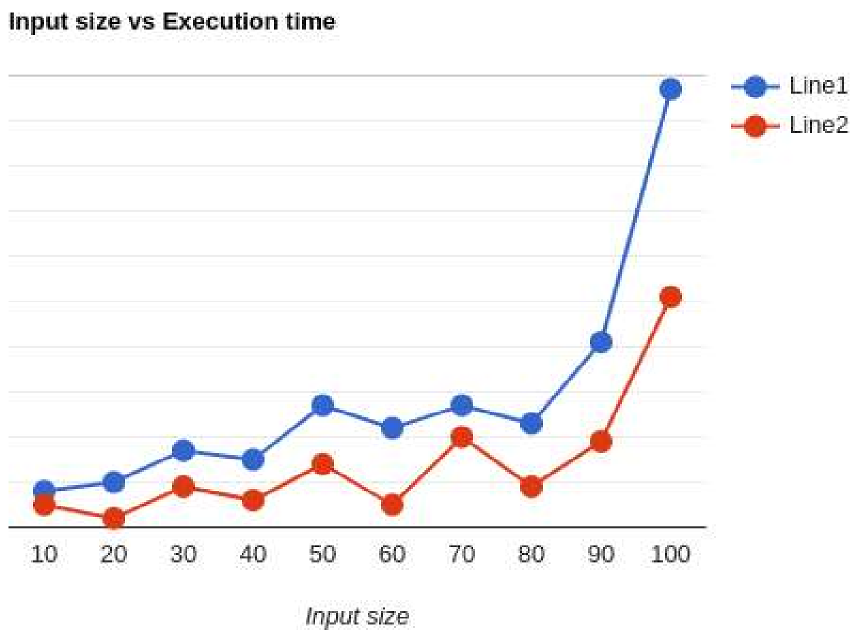


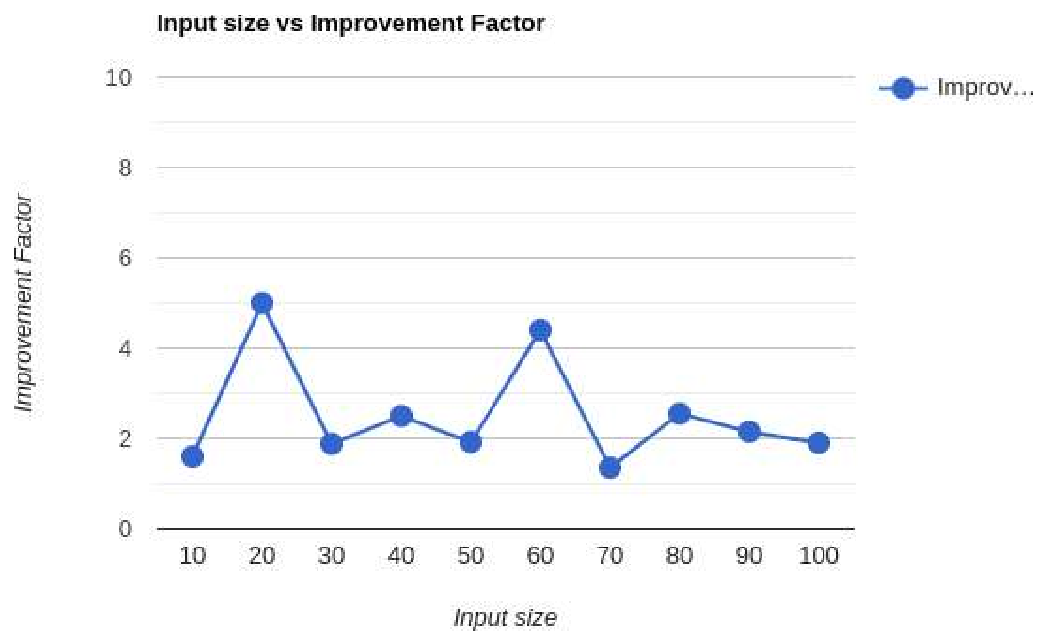
```
-----------------------------------------------------------------------------------------------
Press 1 to run in serial mode
Press 2 to run in parallel mode
Press 3 to run in comparison mode (both)
Press 0 to exit
3
---------------------------------------------------
Max profit from serial execution: 16
Time taken from serial execution = 0.007000 s
Max weight from parallel execution: 16
Time taken from parallel execution = 0.001000 s

Parallel execution is 7.000000 times faster than serial execution in this example.
-----------------------------------------------------------------------------------------------
```

6. RESULTS AND CONCLUSIONS

The following results were obtained after multiple executions of programs on different input sizes. The following data may vary with every execution as execution times are also affected by other processes running on operating system. But it gives a rough idea that our parallel execution is quite faster than serial execution.

S.no	Input size(n)	Serial Time (s)	Parallel Time (s)	Improvement factor
1	10	0.000008	0.000005	1.60
2	20	0.000010	0.000002	5.00
3	30	0.000017	0.000009	1.88
4	40	0.000015	0.000006	2.50
5	50	0.000027	0.000014	1.92
6	60	0.000022	0.000005	4.40
7	70	0.000027	0.000020	1.35
8	80	0.000023	0.000009	2.55
9	90	0.000041	0.000019	2.15
10	100	0.000097	0.000051	1.90




Input size vs Improvement Factor
Improvement Factor
Input size
Improv…
0
2
4
6
8
10
10
20
30
40
50
60
70
80
90
100

7. REFERENCES

[1] A parallel algorithm for the knapsack problem using a generation and searching technique. -Henry Ker Chang Chag, Jonathan Jen Rong Chen, Snyong-Jian Shyu

[2] Computing Partitions with Applications to the Knapsack Problem. -Horowitz, Ellis, and Sartaj Sahni.

[3] Optimal parallel algorithms for the knapsack problems without memory conflicts. - Li, Ken-Li, Ren-Fa Li, and Qing-Hua Li.

[4] OPENMP: An Industry-Standard API for shared memory programming

[5] Concurrency and Parallelism -SASH GOLOSHIFIN, DIMA ZORBALEV, JOO FLATOW