Massively Parallel Approximation Algorithms for the Knapsack Problem

Zhenkuang He
Rochester Institute of Technology
Department of Computer Science
zxh3909@g.rit.edu

Committee:
Chair: Prof. Alan Kaminsky
Abstract

An NP problem is a problem can be solved in nondeterministic polynomial time. Because the number of the possible solutions can be exponentially large, parallel computing can play an important role in solving this kind of problem.

Traditionally there are two ways to solve an NP problem: Brute-force algorithm and Non-brute-force algorithm. Brute-force algorithm calculates all the possible solutions, then choose the best one to solve the problem. Non-brute-force algorithm relies on approximation of heuristics. In order to have high level of approximation and take advantage of parallel computing, we are introducing an alternative approach – Massively Parallel Algorithm.

The use of massively parallel stochastic local search (MPSLS) algorithm and massively parallel randomized approximation (MPRA) algorithm are presented in this paper. The project aims to investigate that, if MPSLS or MPRA are suitable algorithms to solve a selected NP problem, how they perform and what advantage they have compare to the original exhaustive search algorithm.

To achieve the goal, we programed all three different algorithms using Java platform, tested the program with multicore computers, collected various data set and performed data comparison and analyzing. The results are presented in this paper.
Contents

1. Introduction ........................................................................................................ 4
2. Description of Knapsack Problem .......................................................................... 5
3. Description of Exhaustive Search Algorithm ......................................................... 5
4. Related work ............................................................................................................. 6
5. Description of Massively Parallel Algorithm
   5.1. Massively Parallel Randomized Approximation (MPRA) Algorithm ................... 9
   5.2. Massively Parallel Stochastic Local Search (MPSLS) Algorithm ..................... 10
6. Design
   6.1. Parallel Program Design for Instance Generation .............................................. 11
   6.2. Parallel Program Design for Exhaustive Search Algorithm ............................ 12
   6.3. Parallel Program Design for MPRA Algorithm ................................................ 13
   6.4. Parallel Program Design for MPSLS Algorithm ............................................... 14
7. Implementation Example
   7.1. Library Used .................................................................................................... 16
   7.2. Exhaustive Search Parallel Program ............................................................... 16
   7.3. MPRA Parallel Program ................................................................................... 16
   7.4. MPSLS Parallel Program .................................................................................. 17
8. Data Collection and Analyze
   8.1. Experiment Equipment ..................................................................................... 18
   8.2. Exhaustive Search Data Analyzing ................................................................... 19
   8.3. MPRA vs. Exhaustive Search Data and Performance Analyzing ....................... 28
   8.4. MPSLS vs. Exhaustive Search Data and Performance Analyzing ..................... 35
   8.5. MPRA vs. MPSLS Data and Performance Analyzing ......................................... 42
   8.6. Extend Research – Hard Knapsack Problem
      8.6.1. Description of Hard Knapsack Problem ....................................................... 47
      8.6.2. Strong Correlated Instance (SCI) .............................................................. 47
      8.6.3. Inverse Strong Correlated Instance (ISCI) .................................................. 47
      8.6.4. Parallel Program Design for Hard Instance Generation ............................ 48
      8.6.5. MPRA vs. Exhaustive Search Data and Performance Analyzing .................. 49
      8.6.6. MPSLS vs. Exhaustive Search Data and Performance Analyzing ................. 56
9. Future Work ........................................................................................................... 63
10. Conclusion .............................................................................................................. 63
11. References ............................................................................................................. 64
Introduction

We are presenting solving a selected NP problem using the technology of parallel computing. Parallel computing is a computation that divides a large problem into smaller ones. By taking advantage of multiple threads running the resulting program concurrently, it can solve a dividable large problem in an efficient way.

Because NP problem can be solved in nondeterministic polynomial time, two different approaches are typically used for finding the solution. One is brute-force algorithm. This kind of approach calculates all the possibilities of solutions the problem may have, choose the best one at the end. The logic is simple but the number of possible solutions could become exponential large with the incensement of the problem size. The other approach for an NP problem is non-brute-force algorithm. This approach will produce approximate solutions. In order to generate good approximation and take good advantage of parallel computing, we are introducing a new algorithm - massively parallel algorithm.

There are two variations of massively parallel algorithm presented in this project: MPSLS and MPRA. MPSLS - Massively Parallel Stochastic Local Search algorithm starts from a random possible solution, then move on to search another better solution based on the initialize point using heuristics algorithm. Each round’s best solution will become the new start point for the next round. At the end there will be a best solution that cannot produce any better result. MPRA - Massively Parallel Randomized Approximation algorithm is based on random choice of solutions. Usually there are large amount of choices and it will pick one with the best result using heuristic algorithm.

In this project, we were given an NP problem. We started by using the existing algorithms (related work) in parallel approach to solve the NP problem. Then we developed the massively parallel algorithm (MPSLS and MPRA). We explored the exact solution of the NP problem with proper problem size by the implementation of exhaustive search algorithm (brute force search) in parallel. We ran both MPSLS and MPRA algorithm with different instances of the NP problem in parallel, measured and compared the speed and the quality of the solutions between exhaustive search and massively parallel search with different problem size. We also investigated and analyzed the performance of the MPSLS and MPRA with large problem size which is not appropriated by brute force search.

Using the Parallel Java 2 library developed by Professor Alan Kaminsky from the Department of Computer Science at the Rochester Institute of Technology, we performed parallel computing algorithm using Java platform and executed experiments with multicore machine for data collecting, comparing and analyzing.
Description of Knapsack Problem

The knapsack problem is an optimum combination problem. Given N items, each item has a value and a weight. There is a knapsack that has a capacity K. Which items should be chosen into the knapsack so that the total value will achieve maximum without exceeding the knapsack’s capacity K is the knapsack problem.

Description of Exhaustive Search Algorithm

How it works
The exhaustive search algorithm builds a 0 - 1 coefficient list for all possible cases. It checks the total $2^N$ possibilities and always maintains the maximum total value without exceeding the capacity of the knapsack at the same time. After looping more than $2^N$ times, it will result in a maximum total value while the total weight is less or equal to the knapsack’s capacity.

Pseudo code

```
Integer N; // Total number of items
Integer C []; // Coefficient array
Integer m; // The number of the optimal configuration
Float sum = 0; // Total weight of configuration
Float value = 0; // Total value of configuration
Float max = 0; // Maximum value
Float W []; // Weight array
Float V []; // Value array
Float K; // Capacity of the knapsack

begin
  for i = 0 to $2^N$ do
    use bit shift to generate all possible configurations and C []
  end for
  for i = 0 to $2^N$ do
    sum = the dot product of C [i] and W [i]
    if (sum <= K) then
      value = the dot product of C [i] and V [i]
      if (value >= max) then
        max = value; // Update the maximum value
        m = i; // Keep track the configuration
      end if
    end if
  end for
  return max and m;
end
```
Related work

Analysis of Paper I


This paper describes a parallel algorithm for the knapsack problem based on the cluster and the two-list algorithm.

1. Problem introduction

   Find a solution for sum of i from 1 to n,
   \[ a_i \cdot b_i = C, \ b_i = 0 \ or \ 1 \]

   Where \( a_i \) (\( i = 1, 2, ..., n \)) and C are positive integer numbers.

2. The two-list algorithm

   This algorithm can be divided into two parts:

   1) Generation stage
   a. Divide A into two equal parts: A1 and A2;
   b. Find all N possibilities for the subset sums of A1 then sorted them in an increasing order and store in array X;
   c. Do the same thing to A2 but store in array Y with decreasing order;

   2) Search stage
   a. \( i=1, \ j=1 \);
   b. If \( x_i + y_i = C \), then stop and a solution is found;
   c. If \( x_i + y_i < C \), then \( i = i +1 \); else \( j = j + 1 \);
   d. If \( i > N \) or \( j > N \), then stop and no solution.
   e. Go to step 2.

   The total time of two-list algorithm should be \( O(N) \).

Compared with this approach, our project implements massively parallel computing algorithm. Other than two-list algorithm, massively parallel algorithm focuses on heuristics approximation.
Analysis of Paper II


This paper explained the use of Genetic algorithms to solve the Knapsack problem. It starts off explaining the GA and then proceeds the approach to use it to solve the Knapsack problem. Although no algorithm is mentioned in the paper, we can have formulated a possible algorithm using GA:

1. Initialize a population using random bits, say 10
2. Calculate the fitness of each knapsack. The fitness function will calculate the weight of the knapsack. This is by taking the dot product of the knapsack array and the value array.
3. If the total weight of the current knapsack is greater than then a random bit is flipped and then checked again.
4. Using any of the available selection techniques, select any 2 knapsack. Using crossover, obtain a child and then mutate it. This ensures that the child is different from the parent.
5. Each such process is called a generation. A maximum number of generations can be specified using a variable and the process can be killed when the value is reached.
6. When more than 90% of the values have the same fitness values, the loop will exit.

Other than using genetic algorithm to solve the Knapsack problem, our project uses a massively parallel algorithm for different approach. Similar with the genetic algorithm, we use a heuristics way for searching a best solution. However massively parallel algorithm should be more efficient because it can generate better approximation.
Analysis of Paper III


This paper analyzed various parallel algorithms for knapsack problem and their time complexity. This paper also proposed an approximation algorithm for the knapsack problem.

1. Application of Dynamic Programing to Knapsack Problem

An intendance of Knapsack (A, c) problem has m objects and the capacity is c. Each object ai has weight wi and profit pi. The problem can be formulated as:

Find integers $x_i \geq 0$ (for 0 <= I <= m) to

Maximize $p_1 \cdot x_1 + p_2 \cdot x_2 + \cdots + p_m \cdot x_m$

Subject to the constraint $w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_m \cdot x_m \leq c$

$f(k, g) = \max \{ p_1 \cdot x_1 + p_2 \cdot x_2 + \cdots + p_k \cdot x_k \mid w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_k \cdot x_k \leq g \}$

Where $x_i$ is positive integer

The dynamic programming iterations for integers 0 <= I <= m and 0 <= g <= c is defined as follows:

$f(k, g) = \max\{ f(k-1, g), p_k + f(k, g-wk) \}$

Where f(-1, g) = 0, f(k, 0) = 0

2. Knapsack Problem Using the Subset Sum Principle

They relate the knapsack solution with subset sum principle. The subset sum objective is to choose the subset which gives a maximum sum of objects. In this algorithm, the authors simplified the problem: the profit for each object is the same as its size ($p_i = s_i$).

This is a sequence of polynomial bounded algorithm sKnapk and the ration of the optimal solution to the algorithm is 1 / 1+k. sKnapk’s time complexity is O(kn^k (k+1)). They presents a theorem: When k > 0, sknapk does O(kn^k) operations; sknap0 does O(n). Hence sknapk in polynomial time for k >= 0.

Other than focusing on the time complexity of algorithm to solve knapsack problem, our project pay more attention to the implementation and data analyze. Moreover, the massively parallel algorithm can provide better approximation than subset sum principle.
Description of Massively Parallel Algorithm

Massively Parallel Randomized Approximation (MPRA) Algorithm

1. We have a coefficient list containing only 0 or 1 that determines which item will be included in the knapsack.

\[ \{C_0, C_1, C_2, C_3, C_4, \ldots \} \]

Thus, the total weight of the knapsack should be

\[ C_0 \cdot W_0 + C_1 \cdot W_1 + C_2 \cdot W_2 + C_3 \cdot W_3 + C_4 \cdot W_4 + \ldots \]

And, the total value of the knapsack should be

\[ C_0 \cdot V_0 + C_1 \cdot V_1 + C_2 \cdot V_2 + C_3 \cdot V_3 + C_4 \cdot V_4 + \ldots \]

Where \( C_i = 0 \) or 1.

2. The capacity of the knapsack is \( K \).

3. The unit of a processing is an iteration, loops are running in each iteration.

4. Each iteration initially include all the items in the knapsack. That means the starting solution should be

\[ C_0 = C_1 = C_2 = C_3 = \ldots = 1 \]

5. Calculate the total weight of the knapsack, if it’s more than \( K \), begin the loop; else store the total value and the iteration is completed.

6. Begin the loop:
   1) Randomly choose one item to take out from the knapsack. Set the coefficient from 1 to 0.
   2) Calculate the total weight of the knapsack.
   3) If the total weight is more than \( K \), continue the loop; else break the loop and store the total value.

7. For parallel purpose each thread (cores) will run certain number of iterations.

8. Each iteration will have a solution.

9. Reduction program will choose the best solution from iterations. (Maximum total value)
Massively Parallel Stochastic Local Search (MPSLS) Algorithm

1. We have a coefficient list containing only 0 or 1 that determines which item will be included in the knapsack.
   \[ \{C_0, C_1, C_2, C_3, C_4, \ldots \} \]

   Thus, the total weight of the knapsack should be
   \[ C_0 \cdot W_0 + C_1 \cdot W_1 + C_2 \cdot W_2 + C_3 \cdot W_3 + C_4 \cdot W_4 + \ldots \]

   And, the total value of the knapsack should be
   \[ C_0 \cdot V_0 + C_1 \cdot V_1 + C_2 \cdot V_2 + C_3 \cdot V_3 + C_4 \cdot V_4 + \ldots \]

   Where \( C_i = 0 \) or 1.

2. The capacity of the knapsack is \( K \).

3. The unit of a processing is an iteration, loops are running in each iteration.

4. Each iteration initially pick a random subset of the possible solution. For example the coefficient list could be
   \[ \{1, 0, 1, 1, 0, \ldots \} \]

5. Calculate the total weight of the knapsack, if it’s more than \( K \), begin the loop; else store the total value as current maximum total value and begin the loop.

6. Begin the loop:
   1) After beginning the coefficient list, flip each item's coefficient, keep checking the total weight after each flipping and keep updating the current maximum total value if the total weight is less than or equal to \( K \).
   2) If the current round cannot produce a better solution (larger total value) than the starting point, take the current maximum total value (starting point) as the best solution of the iteration; else, use the coefficient list that can generate the current maximum total value as the starting base for next loop round.

7. For parallel purpose, each thread (cores) will run certain number of iterations.

8. Each iteration will have a solution.

9. Reduction program will choose the best solution from iterations. (Maximum total value)
**Design**

**Parallel Program Design for Instance Generation**

The instance for the knapsack problem is the items provided for selecting to generate the solution. The total amount of the items would be \( N \). Each item has weight (\( W \)) and value (\( V \)). In this project we randomly select the item’s weight (\( W \)) and value (\( V \)) as a float number between the range \([0, 10]\). Thus we will have two array (list) to store all the items property.

\[
W[i] = \{W_0, W_1, W_2, W_3, ..., W_i\} \\
V[i] = \{V_0, V_1, V_2, V_3, ..., V_i\}
\]

Where \( i = 1, 2, 3, ..., N \).

The number of item (\( N \)) is an important key to the definition of the problem size. When it gets larger and larger, parallel design will make the instance generation more efficient.

The Random object in Java would use the given seed to generate a random sequence of number. It will produce the same sequence with the same seed even the order is random. In parallel computing mechanism, each thread will use the same random seed and run the program at the same time. In order to avoid each thread generating the same sequence of number, we need to have a “skip” algorithm applied to them.

1. Assuming that we have a multicore machine that has \( m \) threads.
2. Divide \( N \) generation mission to each thread. The load of each thread would be \( N/m \).
3. Each thread has a rank different from the other. The rank number begins from 0.
4. Initially they need to skip the number of \( 2 \times \) rank number in the random sequence to pick the number in order to generate the item’s weight (\( W \)) and value (\( V \)).
5. Each time each thread generate two random numbers of weight (\( W \)) and value (\( V \)), it will skip \( 2 \times (m – 1) \) to avoid duplicate pick in the same random sequence.

After the process of instance generation, we will have three different kinds of algorithms to implement in this project.
**Parallel Program Design for Exhaustive Search Algorithm**

1. Get input parameter:
   1) N, the total number of items;
   2) K, the capacity of the knapsack;
   3) Seed, the random seed for randomly initializing the weight and value of each item.

2. Total possibilities needed to go through is \( n = 2^N \).

3. Parallel the initializing jobs to threads. Generate the instance for the knapsack problem.

4. Parallel the calculation jobs to threads. Each thread has its own local parameters:
   1) \( R\_\text{thread} \), to store the solution’s number;
   2) \( \text{SUM}\_\text{thread} \), to store the solution’s total weight;
   3) \( \text{VALUE}\_\text{thread} \), to store the solution’s total value;
   4) \( M\_\text{thread} \), to store the latest largest total value;
   5) BS, the bit set to store the coefficient list

5. Compute the total weight and total value with the coefficient list for all possibilities. Checking if the total weight is less than or equal to K, before getting the result of a possibility. If not, leave it and continue the loop; if so, compare with \( M\_\text{thread} \). If larger than \( M\_\text{thread} \), update the \( R\_\text{thread} \), \( \text{SUM}\_\text{thread} \) and \( \text{VALUE}\_\text{thread} \); if not, leave it and continue the loop.

6. Loop all the possibilities until the end of each thread’s range. The current best solution will be captured to a reduction program, after each loop. The reduction program will keep the best solution till the end of the process.

7. In the end, we will get the best solution from the reduction program and it’s the end of the process.
Parallel Program Design for MPRA Algorithm

1. Get input parameters:
   1) N, the total number of items;
   2) K, the capacity of the knapsack;
   3) I, the total number of iterations the program will run
   4) Seed, the random seed for randomly initialize the weight and value of each item.

2. Parallel the initializing jobs to threads. Generate the instance for the knapsack problem.

3. Parallel the calculation jobs to threads. Each thread has its own local parameters:
   1) SUM_thread, to store the solution’s total weight;
   2) VALUE_thread, to store the solution’s total value;
   3) M_thread, to store the latest largest total value;
   4) BS, the bit set to store the coefficient list.

4. Each iteration initially sets all from 1 to BS, meaning all items are in the knapsack. Compute the total weight at first round, if it’s less than or equal to the capacity of the knapsack (K), then this would be the solution of this iteration, else, begin the loop.

5. MPRA algorithm loop:
   1) Use variable B to count the number of items left. B should be from N to 0.
   2) Randomly choose a bit of BS to set it from 1 to 0, meaning taking one of the items out of the knapsack.
   3) Calculate the total weight, if it’s less than or equal to capacity K, compute the total value which would be the solution of the iteration; else count down 1 for the B (B = B - 1) and begin the next round of the loop.
   4) Each loop will randomly take out an item like step 2) and check the total weight.
   5) Once the program has a solution, that’s the end point, or when B reaches 0 (nothing left in the knapsack).

6. Once the loops and iteration end, and it will store a solution to the local parameter M_thread, then start the next iteration. Each iteration will have a solution to be compared with M_thread, and always keep the M_thread with the best one (largest one).

7. In general, each thread would run the numbers of iterations in parallel and each iteration would run numbers of loops till a solution is generated.

8. Each thread would have a best solution, and the reduction program would compare them in order to generate the best solution of the program, in the end.
Parallel Program Design for MPSLS Algorithm

1. Get input parameters:
   1) N, the total number of items;
   2) K, the capacity of the knapsack;
   3) I, the total number of iterations the program will run
   4) Seed, the random seed for randomly initialize the weight and value of each item.

2. Parallel the initializing jobs to threads. Generate the instance for the knapsack problem.

3. Parallel the calculation jobs to threads. Each thread has its own local parameters:
   1) SUM_thread, to store the solution’s total weight;
   2) VALUE_thread, to store the solution’s total value;
   3) M_thread, to store the latest largest total value;
   4) BS, the bit set to store the coefficient list;
   5) bestSet, the temporary bit set to store the best solution of the round.

4. Each iteration initially selects the subset of the solution of the knapsack problem. That is, for each bit of the BS, randomly set it to 0 or 1. After setting all the BS, the beginning of the algorithm is finished. Compute the total weight at first round, if it’s less than or equal to the capacity of the knapsack (K), then this would be the solution of this iteration, else, begin the loop.

5. MPSLS algorithm loop:
   1) Use Boolean flag (default false) to check if the loop finds the best solution of the iteration.
   2) Dealing with the loop beginning BS, flip each bit of the bit set, which means each round, change one of the bit value to the opposite (0 to 1 or 1 to 0). Calculate the total weight of each round. If the total weight of that round less than or equal to the capacity K, compute the total value; else leave the result.
   3) Once getting a solution, compare the total value to M_thread. That means keeping the best solution value of the loop in the M_thread.
   4) The temporary bit set bestSet will clone the BS which can generate the best solution of the loop.
   5) When each loop ends, it will use the bestSet as the beginning point and start the next loop.
   6) For the next loop, same processes as from 2) to 4), but at the end of the loop the result (total value) needs to be compared with the beginning point. If the new loop can produce a better result than the previous loop, set the flag to true and use that bit set to begin the
next loop; else keep the flag false and take the previous loop’s best solution as the solution of the iteration.

7) Begin the next loop until the flag is false at the end of the current loop.

6. Once the loops and iteration end, and it will store a solution to the local parameter M_thread, then start the next iteration. Each iteration will have a solution to be compared with M_thread, and just always keep the M_thread with the best one (largest one).

7. In general, each thread would run the numbers of iterations in parallel and each iteration would run numbers of loops. Each loop would have numbers of rounds for flipping each bit of the beginning bit set. Among them each thread would generate a solution.

8. Each thread would have a best solution, and the reduction program would compare them in order to generate the best solution of the program, in the end.
Implementation Example

Library Used

Parallel Java 2 Library Develop by Prof. Alan Kaminsky

```
export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
export PATH=/usr/local/dcs/versions/jdk1.7.0_11_x64/bin:$PATH
```

Exhaustive Search Parallel Program

1. Input
   1)  N = 10
   2)  K = 30
   3)  Seed = 135792468

2. Instance generated

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>4.805</td>
<td>6.026</td>
<td>9.446</td>
<td>5.573</td>
<td>0.510</td>
</tr>
<tr>
<td>Value</td>
<td>7.849</td>
<td>2.988</td>
<td>5.645</td>
<td>5.892</td>
<td>7.056</td>
</tr>
<tr>
<td>Weight</td>
<td>6.960</td>
<td>7.166</td>
<td>3.837</td>
<td>1.364</td>
<td>1.386</td>
</tr>
</tbody>
</table>

3. Solution
   Total Weight = 26.028397
   Total Value = 52.87228

MPRA Parallel Program

1. Input
   1)  N = 10
   2)  K = 30
   3)  I = 1000
   4)  Seed = 246813579

2. Instance generated

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>3.715</td>
<td>6.236</td>
<td>9.944</td>
<td>0.815</td>
<td>4.700</td>
</tr>
<tr>
<td>Value</td>
<td>3.445</td>
<td>7.671</td>
<td>7.439</td>
<td>5.442</td>
<td>3.284</td>
</tr>
<tr>
<td>Weight</td>
<td>4.074</td>
<td>5.282</td>
<td>5.425</td>
<td>5.305</td>
<td>2.070</td>
</tr>
<tr>
<td>Value</td>
<td>0.543</td>
<td>3.780</td>
<td>3.103</td>
<td>8.921</td>
<td>7.760</td>
</tr>
</tbody>
</table>

3. Solution
   Total Weight = 27.582565
   Total Value = 33.25381
MPSLS Parallel Program

1. Input
   1) N = 10
   2) K = 30
   3) I = 1000
   4) Seed = 123456789

2. Instance generated

   Weight  |  7.991  |  6.780  |  0.862  |  3.656  |  2.445  |
   Value   |  4.287  |  7.138  |  4.923  |  9.643  |  0.706  |
   Weight  |  3.318  |  5.139  |  6.985  |  5.609  |  0.604  |
   Value   |  8.675  |  9.321  |  0.825  |  6.840  |  4.216  |

3. Solution
   Total Weight = 28.414549
   Total Value = 51.46153
Data Collection and Analyze

Experiment Equipment

This project’s experiment for data collection and analyze use the multicore parallel computer in Computer Science Department of Rochester Institute of Technology

Name of server:

champ.cs.rit.edu

Configuration:

One Intel Xeon E5-2690 processor
Eight dual-hyperthreaded CPU cores
16 threads
2.9 GHz clock
32 GB main memory
One NVIDIA Tesla C2075 GPU accelerator
448 cores
1.15 GHz clock
6 GB global memory
Ubuntu 12.04 LTS 64-bit Linux
Exhaustive Search Data Analyzing

1. Exhaustive Search Program Running I (KnapsackSMP.java)

   1) Parameters for input:
      K = 50, seed = 11111111

   2) Data Collection – Table*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>7.368478</td>
<td>10.95516</td>
<td>11.905649</td>
<td>14.302797</td>
<td>23.739012</td>
<td>25.56031</td>
<td>28.36866</td>
<td>34.22987</td>
<td>44.11256</td>
<td>44.11256</td>
</tr>
<tr>
<td>Value</td>
<td>3.738368</td>
<td>10.95516</td>
<td>11.905649</td>
<td>14.302797</td>
<td>23.739012</td>
<td>25.56031</td>
<td>28.36866</td>
<td>34.22987</td>
<td>44.11256</td>
<td>44.11256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>41.86471</td>
<td>45.24492</td>
<td>45.24492</td>
<td>45.63223</td>
<td>49.79974</td>
<td>43.12968</td>
<td>48.92583</td>
<td>48.92583</td>
<td>48.92583</td>
<td>48.92583</td>
</tr>
<tr>
<td>Value</td>
<td>57.68107</td>
<td>62.65449</td>
<td>62.65449</td>
<td>63.50346</td>
<td>66.28101</td>
<td>66.87293</td>
<td>82.17423</td>
<td>82.17423</td>
<td>82.17423</td>
<td>82.17423</td>
</tr>
<tr>
<td>Time</td>
<td>22</td>
<td>30</td>
<td>33</td>
<td>33</td>
<td>32</td>
<td>38</td>
<td>44</td>
<td>45</td>
<td>47</td>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>49.49104</td>
<td>49.49104</td>
<td>48.87705</td>
<td>48.988388</td>
<td>48.988388</td>
<td>49.64894</td>
<td>49.64894</td>
<td>49.64894</td>
<td>49.64894</td>
<td>49.64894</td>
</tr>
<tr>
<td>Value</td>
<td>83.65367</td>
<td>83.65367</td>
<td>85.27415</td>
<td>89.27049</td>
<td>89.27049</td>
<td>93.49408</td>
<td>93.49408</td>
<td>93.49408</td>
<td>93.49408</td>
<td>93.49408</td>
</tr>
<tr>
<td>Time</td>
<td>69</td>
<td>90</td>
<td>138</td>
<td>229</td>
<td>378</td>
<td>764</td>
<td>1498</td>
<td>2629</td>
<td>5897</td>
<td>12173</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>49.64894</td>
<td>49.88575</td>
<td>49.13837</td>
<td>49.7659</td>
<td>49.87724</td>
<td>49.89674</td>
<td>49.89674</td>
<td>48.92548</td>
<td>49.88158</td>
<td>49.88158</td>
</tr>
<tr>
<td>Value</td>
<td>93.49408</td>
<td>101.7584</td>
<td>100.9148</td>
<td>103.0237</td>
<td>107.0201</td>
<td>116.357</td>
<td>116.357</td>
<td>118.2393</td>
<td>122.9831</td>
<td>122.9831</td>
</tr>
<tr>
<td>Time</td>
<td>24919</td>
<td>49773</td>
<td>102549</td>
<td>207989</td>
<td>414038</td>
<td>859566</td>
<td>1708196</td>
<td>3525224</td>
<td>6808043</td>
<td>14753112</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond
For input items from 1 to 20, program runs using small amount of time. Running time increases but not much.
For input items from 21 to 40, program’s running time increases exponentially. The running time doubles up when the number of items increase 1.
2. Exhaustive Search Program Running II (KnapsackSMP.java)

1) Parameters for input:
   \( K = 60, \text{ seed } = 135792468 \)

2) Data Collection – Table*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>4.804572</td>
<td>10.83061</td>
<td>20.27613</td>
<td>25.849377</td>
<td>26.359592</td>
<td>33.31989</td>
<td>40.48587</td>
<td>44.32333</td>
<td>45.68705</td>
<td>47.0732</td>
</tr>
<tr>
<td>Time</td>
<td>20</td>
<td>23</td>
<td>21</td>
<td>32</td>
<td>25</td>
<td>32</td>
<td>22</td>
<td>24</td>
<td>24</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>50.12465</td>
<td>52.02697</td>
<td>54.68321</td>
<td>54.70162</td>
<td>54.70162</td>
<td>55.53915</td>
<td>58.08517</td>
<td>58.08517</td>
<td>59.47132</td>
<td>59.29977</td>
</tr>
<tr>
<td>Value</td>
<td>71.17979</td>
<td>77.15424</td>
<td>81.79742</td>
<td>83.65089</td>
<td>83.65089</td>
<td>88.60431</td>
<td>84.43575</td>
<td>84.43575</td>
<td>90.01587</td>
<td>92.61505</td>
</tr>
<tr>
<td>Time</td>
<td>35</td>
<td>30</td>
<td>36</td>
<td>34</td>
<td>35</td>
<td>41</td>
<td>41</td>
<td>56</td>
<td>62</td>
<td>66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>59.93846</td>
<td>59.93846</td>
<td>58.227505</td>
<td>58.227505</td>
<td>59.199448</td>
<td>59.19945</td>
<td>59.21786</td>
<td>59.21786</td>
<td>59.21786</td>
<td>59.21786</td>
</tr>
<tr>
<td>Value</td>
<td>93.78578</td>
<td>93.78578</td>
<td>99.38318</td>
<td>99.38318</td>
<td>101.65599</td>
<td>101.656</td>
<td>103.5095</td>
<td>103.5095</td>
<td>103.5095</td>
<td>103.5095</td>
</tr>
<tr>
<td>Time</td>
<td>83</td>
<td>88</td>
<td>133</td>
<td>218</td>
<td>392</td>
<td>687</td>
<td>1431</td>
<td>2882</td>
<td>5834</td>
<td>11998</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>58.68824</td>
<td>58.68824</td>
<td>59.89938</td>
<td>59.37521</td>
<td>59.37521</td>
<td>59.37521</td>
<td>59.72221</td>
<td>59.72221</td>
<td>59.9391</td>
<td>59.9391</td>
</tr>
<tr>
<td>Value</td>
<td>107.0513</td>
<td>107.0513</td>
<td>126.3307</td>
<td>128.9811</td>
<td>128.9811</td>
<td>128.9811</td>
<td>130.796</td>
<td>130.796</td>
<td>133.7524</td>
<td>135.6059</td>
</tr>
<tr>
<td>Time</td>
<td>22141</td>
<td>49726</td>
<td>101454</td>
<td>218</td>
<td>392</td>
<td>687</td>
<td>1431</td>
<td>2882</td>
<td>5834</td>
<td>11998</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond
For input items from 1 to 20, program runs using small amount of time. Running time increases but not much.
For input items from 21 to 40, program's running time increases exponentially. The running time doubles up when the number of items increase 1.
3. Exhaustive Search Program Running III (KnapsackSMP.java)

1) Parameters for input:
   K = 30, seed = 246813579

2) Data Collection – Table*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weight</strong></td>
<td>4.804572</td>
<td>10.83061</td>
<td>20.27613</td>
<td>25.849377</td>
<td>26.359592</td>
<td>33.31989</td>
<td>40.48587</td>
<td>44.32333</td>
<td>45.68705</td>
<td>47.0732</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>20</td>
<td>23</td>
<td>21</td>
<td>32</td>
<td>25</td>
<td>32</td>
<td>22</td>
<td>24</td>
<td>24</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weight</strong></td>
<td>50.12465</td>
<td>52.02697</td>
<td>54.68321</td>
<td>55.70162</td>
<td>54.70162</td>
<td>55.53915</td>
<td>58.08517</td>
<td>58.08517</td>
<td>59.547132</td>
<td>59.29977</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>71.17979</td>
<td>77.15424</td>
<td>81.79742</td>
<td>83.65089</td>
<td>83.65089</td>
<td>88.60431</td>
<td>84.43575</td>
<td>84.43575</td>
<td>90.01587</td>
<td>92.61505</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>35</td>
<td>30</td>
<td>36</td>
<td>34</td>
<td>35</td>
<td>41</td>
<td>41</td>
<td>56</td>
<td>62</td>
<td>66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weight</strong></td>
<td>59.93846</td>
<td>59.93846</td>
<td>58.227505</td>
<td>58.227505</td>
<td>59.199448</td>
<td>59.19945</td>
<td>59.21786</td>
<td>59.21786</td>
<td>59.21786</td>
<td>59.21786</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>93.78578</td>
<td>93.78578</td>
<td>99.38318</td>
<td>101.65599</td>
<td>101.656</td>
<td>103.5095</td>
<td>103.5095</td>
<td>103.5095</td>
<td>103.5095</td>
<td>103.5095</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>83</td>
<td>88</td>
<td>133</td>
<td>218</td>
<td>392</td>
<td>687</td>
<td>1431</td>
<td>2882</td>
<td>5834</td>
<td>11998</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weight</strong></td>
<td>58.68824</td>
<td>58.68824</td>
<td>59.89938</td>
<td>59.37521</td>
<td>59.37521</td>
<td>59.37521</td>
<td>59.72221</td>
<td>59.72221</td>
<td>59.9391</td>
<td>59.95751</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>107.0513</td>
<td>107.0513</td>
<td>126.3307</td>
<td>128.9811</td>
<td>128.9811</td>
<td>128.9811</td>
<td>130.796</td>
<td>130.796</td>
<td>133.7524</td>
<td>135.6059</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>22141</td>
<td>49726</td>
<td>101454</td>
<td>218</td>
<td>392</td>
<td>687</td>
<td>1431</td>
<td>2882</td>
<td>5834</td>
<td>11998</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond
For input items from 1 to 20, program runs using small amount of time. Running time increases but not much.
For input items from 21 to 40, program’s running time increases exponentially. The running time doubles up when the number of items increase 1.

4. Analysis

With three different input instance experiments, we can see the incensement of the problem size would be double up when the number of items increase 1, the Exhaustive Search algorithm needs the program running till the end, so the execute time would be doubled. Moreover, with the advantage of the parallel computing, we can divide the mission load to 16 threads and run simultaneously.
MPRA vs. Exhaustive Search Data and Performance Analyzing

1. MPRA program running I (KnapsackMPRASMP.java)

   1) Parameters for input:
   N = 30, K = 50, seed = 11111111

   2) Exhaustive Search algorithm result:
   Best total value = 93.49408
   Running time = 12173

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
<th>1,000,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>69.54002</td>
<td>79.36294</td>
<td>81.95788</td>
<td>83.39362</td>
<td>88.49409</td>
<td>90.34671</td>
<td>93.49408</td>
<td>93.49408</td>
</tr>
<tr>
<td>Value%</td>
<td>74.38%</td>
<td>84.89%</td>
<td>87.66%</td>
<td>89.20%</td>
<td>94.65%</td>
<td>96.63%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>39</td>
<td>55</td>
<td>62</td>
<td>139</td>
<td>532</td>
<td>4562</td>
<td>44868</td>
<td>455683</td>
</tr>
<tr>
<td>Time%</td>
<td>0.32%</td>
<td>0.45%</td>
<td>0.51%</td>
<td>1.14%</td>
<td>4.37%</td>
<td>37.48%</td>
<td>368.59%</td>
<td>3743.39%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPRA 30 items Time% vs. Value% Chart](chart.png)
2. MPRA program running II (KnapsackMPRASMP.java)

1) Parameters for input:
   \[ N = 32, K = 50, \text{seed} = 11111111, \]

2) Exhaustive Search algorithm result:
   \[ \text{Best total value} = 101.75842 \]
   \[ \text{Running time} = 49773 \]

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
<th>1,000,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
<td>70.4277</td>
<td>77.98408</td>
<td>79.46177</td>
<td>85.12373</td>
<td>87.18032</td>
<td>91.4786</td>
<td>91.96709</td>
<td>93.49408</td>
</tr>
<tr>
<td><strong>Value%</strong></td>
<td>69.21%</td>
<td>76.64%</td>
<td>78.09%</td>
<td>83.65%</td>
<td>85.67%</td>
<td>89.90%</td>
<td>90.38%</td>
<td>91.88%</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>37</td>
<td>52</td>
<td>57</td>
<td>126</td>
<td>555</td>
<td>5471</td>
<td>53741</td>
<td>545955</td>
</tr>
<tr>
<td><strong>Time%</strong></td>
<td>0.07%</td>
<td>0.10%</td>
<td>0.11%</td>
<td>0.25%</td>
<td>1.12%</td>
<td>10.99%</td>
<td>107.97%</td>
<td>1096.89%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![Graph showing Time% vs. Value% for MPRA 32 items](image-url)
3. MPRA program running III (KnapsackMPRASMP.java)

1) Parameters for input:
   N = 34, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 103.02371
   Running time = 207989

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
<th>1,000,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>69.71226</td>
<td>79.6081</td>
<td>89.31035</td>
<td>94.17329</td>
<td>95.781746</td>
<td>99.0196</td>
<td>99.92379</td>
<td>100.91475</td>
</tr>
<tr>
<td>Value%</td>
<td>67.67%</td>
<td>77.27%</td>
<td>86.69%</td>
<td>91.41%</td>
<td>92.97%</td>
<td>96.11%</td>
<td>96.99%</td>
<td>97.95%</td>
</tr>
<tr>
<td>Time</td>
<td>36</td>
<td>53</td>
<td>61</td>
<td>125</td>
<td>666</td>
<td>6201</td>
<td>59177</td>
<td>598044</td>
</tr>
<tr>
<td>Time%</td>
<td>0.02%</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.06%</td>
<td>0.32%</td>
<td>2.98%</td>
<td>28.45%</td>
<td>287.54%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPRA 34 items Time% vs. Value%](image-url)
4. MPRA program running IV (KnapsackMPRASMP.java)

1) Parameters for input:
   N = 36, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 116.35696
   Running time = 859566

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
<th>1,000,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>71.406494</td>
<td>79.68108</td>
<td>81.67065</td>
<td>96.25172</td>
<td>97.421295</td>
<td>99.84628</td>
<td>102.535126</td>
<td>104.42444</td>
</tr>
<tr>
<td>Value%</td>
<td>61.37%</td>
<td>68.48%</td>
<td>70.19%</td>
<td>82.72%</td>
<td>83.73%</td>
<td>85.81%</td>
<td>88.12%</td>
<td>89.74%</td>
</tr>
<tr>
<td>Time</td>
<td>46</td>
<td>74</td>
<td>73</td>
<td>158</td>
<td>764</td>
<td>5773</td>
<td>66234</td>
<td>669868</td>
</tr>
<tr>
<td>Time%</td>
<td>0.005%</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.02%</td>
<td>0.09%</td>
<td>0.67%</td>
<td>7.71%</td>
<td>77.93%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond
*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm
*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart
5. MPRA program running V (KnapsackMPRASMP.java)

1) Parameters for input:
   N = 38, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 118.23929
   Running time = 3525224

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
<th>1,000,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
<td>74.785126</td>
<td>84.72325</td>
<td>93.77962</td>
<td>100.1357</td>
<td>108.09961</td>
<td>112.48563</td>
<td>112.48563</td>
<td>114.46181</td>
</tr>
<tr>
<td><strong>Value%</strong></td>
<td>63.25%</td>
<td>71.65%</td>
<td>79.31%</td>
<td>84.69%</td>
<td>91.42%</td>
<td>95.13%</td>
<td>95.13%</td>
<td>96.81%</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>44</td>
<td>58</td>
<td>72</td>
<td>161</td>
<td>720</td>
<td>7455</td>
<td>73584</td>
<td>745330</td>
</tr>
<tr>
<td><strong>Time%</strong></td>
<td>0.001%</td>
<td>0.002%</td>
<td>0.002%</td>
<td>0.00%</td>
<td>0.02%</td>
<td>0.21%</td>
<td>2.09%</td>
<td>21.14%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPRA 38 items Time% vs. Value% chart](image-url)
6. MPRA program running VI (KnapsackMPRASMP.java)

1) Parameters for input:
   N = 40, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 122.98312
   Running time = 14753112

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
<th>1,000,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>78.75547</td>
<td>84.51789</td>
<td>91.92534</td>
<td>102.7504</td>
<td>105.75942</td>
<td>111.10464</td>
<td>113.91586</td>
<td>117.792816</td>
</tr>
<tr>
<td>Value%</td>
<td>64.04%</td>
<td>68.72%</td>
<td>74.75%</td>
<td>83.55%</td>
<td>86.00%</td>
<td>90.34%</td>
<td>92.63%</td>
<td>95.78%</td>
</tr>
<tr>
<td>Time</td>
<td>36</td>
<td>60</td>
<td>69</td>
<td>159</td>
<td>887</td>
<td>8529</td>
<td>79559</td>
<td>825398</td>
</tr>
<tr>
<td>Time%</td>
<td>0.0002%</td>
<td>0.0004%</td>
<td>0.0005%</td>
<td>0.001%</td>
<td>0.006%</td>
<td>0.06%</td>
<td>0.54%</td>
<td>5.59%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPRA 40 items Time% vs. Value%](image-url)
7. Analysis

After experimenting with items from 30 to 40, we compared Exhaustive Search algorithm and MPRA, the MPRA has a solid performance. MPRA takes less running time than exhaustive search algorithm and produces a better result (pretty close to 100%). In addition, the advantage of MPRA becomes obviously when the number of items increases. When 40 items are set, MPRA uses 6% less running time than exhaustive search algorithm while generating the solution of the problem closed to 100%. When MPRA runs more than 10 million iterations, the result is acceptably good.
MPSLS vs. Exhaustive Search Data and Performance Analyzing

1. MPSLS program running I (KnapsackMPSLSSMP.java)

   1) Parameters for input:
      \( N = 30, K = 50, \text{seed} = 11111111 \)

   2) Exhaustive Search algorithm result:
      Best total value = 93.49408
      Running time = 12173

   3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>62.178448</td>
<td>79.16324</td>
<td>88.82101</td>
<td>88.82101</td>
<td>90.810074</td>
<td>92.9364</td>
<td>93.49408</td>
</tr>
<tr>
<td>Value%</td>
<td>66.51%</td>
<td>84.67%</td>
<td>95.00%</td>
<td>95.00%</td>
<td>97.13%</td>
<td>99.40%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>41</td>
<td>57</td>
<td>61</td>
<td>139</td>
<td>662</td>
<td>5890</td>
<td>59059</td>
</tr>
<tr>
<td>Time%</td>
<td>0.34%</td>
<td>0.47%</td>
<td>0.50%</td>
<td>1.14%</td>
<td>5.44%</td>
<td>48.39%</td>
<td>485.16%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS 30 items Time% vs. Value%](image-url)
2. MPSLS program running II (KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 32, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 101.75842
   Running time = 49773

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>77.22777</td>
<td>91.595856</td>
<td>91.59586</td>
<td>99.4209</td>
<td>99.4209</td>
<td>101.75842</td>
<td>101.75842</td>
</tr>
<tr>
<td>Value%</td>
<td>75.89%</td>
<td>90.01%</td>
<td>90.01%</td>
<td>97.70%</td>
<td>97.70%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>36</td>
<td>72</td>
<td>65</td>
<td>173</td>
<td>725</td>
<td>6426</td>
<td>63109</td>
</tr>
<tr>
<td>Time%</td>
<td>0.07%</td>
<td>0.14%</td>
<td>0.13%</td>
<td>0.35%</td>
<td>1.46%</td>
<td>12.91%</td>
<td>126.79%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS 32 items Time% vs. Value% chart](image)
3. MPSLS program running III (KnapsackMPSLSSMP.java)

1) Parameters for input:
   \( N = 34, K = 50, \text{seed} = 11111111, \)

2) Exhaustive Search algorithm result:
   Best total value = 103.02371
   Running time = 207989

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>71.0024</td>
<td>86.761696</td>
<td>98.35107</td>
<td>98.35107</td>
<td>98.351074</td>
<td>100.69786</td>
<td>103.02371</td>
</tr>
<tr>
<td>Value%</td>
<td>68.92%</td>
<td>84.22%</td>
<td>95.46%</td>
<td>95.46%</td>
<td>95.46%</td>
<td>97.74%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>40</td>
<td>75</td>
<td>64</td>
<td>150</td>
<td>794</td>
<td>7213</td>
<td>72301</td>
</tr>
<tr>
<td>Time%</td>
<td>0.02%</td>
<td>0.04%</td>
<td>0.03%</td>
<td>0.07%</td>
<td>0.38%</td>
<td>3.47%</td>
<td>34.76%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS 34 items Time% vs. Value%](image)
4. MPSLS program running IV (KnapsackMPSLSSMP.java)

1) Parameters for input:
   \( N = 36, K = 50, \text{seed} = 11111111, \)

2) Exhaustive Search algorithm result:
   Best total value = 116.35696
   Running time = 859566

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>95.61475</td>
<td>102.3567</td>
<td>102.3567</td>
<td>110.36608</td>
<td>113.978226</td>
<td>116.35696</td>
</tr>
<tr>
<td>Value%</td>
<td>0.00%</td>
<td>82.17%</td>
<td>87.97%</td>
<td>87.97%</td>
<td>94.85%</td>
<td>97.96%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>36</td>
<td>60</td>
<td>65</td>
<td>169</td>
<td>850</td>
<td>7915</td>
<td>83575</td>
</tr>
<tr>
<td>Time%</td>
<td>0.004%</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.02%</td>
<td>0.10%</td>
<td>0.92%</td>
<td>9.72%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart
5. MPSLS program running V (KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 38, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 118.23929
   Running time = 3525224

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>94.81702</td>
<td>101.5004</td>
<td>105.9547</td>
<td>111.20085</td>
<td>113.14429</td>
<td>117.07292</td>
</tr>
<tr>
<td>Value%</td>
<td>0.00%</td>
<td>80.19%</td>
<td>85.84%</td>
<td>89.61%</td>
<td>94.05%</td>
<td>95.69%</td>
<td>99.01%</td>
</tr>
<tr>
<td>Time</td>
<td>39</td>
<td>66</td>
<td>88</td>
<td>155</td>
<td>891</td>
<td>7810</td>
<td>78188</td>
</tr>
<tr>
<td>Time%</td>
<td>0.001%</td>
<td>0.002%</td>
<td>0.002%</td>
<td>0.00%</td>
<td>0.03%</td>
<td>0.22%</td>
<td>2.22%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond
*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm
*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS 38 items Time% vs. Value%](chart.png)
6. MPSLS program running VI (KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 40, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 122.98312
   Running time = 14753112

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>87.197395</td>
<td>108.2081</td>
<td>112.0404</td>
<td>117.86047</td>
<td>117.860466</td>
<td>122.98312</td>
<td></td>
</tr>
<tr>
<td>Value%</td>
<td>70.90%</td>
<td>87.99%</td>
<td>87.99%</td>
<td>91.10%</td>
<td>95.83%</td>
<td>95.83%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>49</td>
<td>79</td>
<td>77</td>
<td>241</td>
<td>1020</td>
<td>9463</td>
<td>86975</td>
</tr>
<tr>
<td>Time%</td>
<td>0.0003%</td>
<td>0.0005%</td>
<td>0.0005%</td>
<td>0.002%</td>
<td>0.007%</td>
<td>0.06%</td>
<td>0.59%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS 40 items Time% vs. Value%](chart.png)
7. Analysis

After the experimenting with items from 30 to 40, we compared Exhaustive Search algorithm and MPSLS, the MPSLS has a solid performance. MPSLS Takes less running time than exhaustive search algorithm and produces a better result (pretty close to 100%). In addition, the advantage of MPSLS becomes obviously when the number of items increases. When 40 items are set, MPSLS uses 1% less running time than exhaustive search algorithm while generating the solution of the problem closed to 100%. When MPSLS runs more than 10 million iterations, the result is acceptably good. On the other hand, for some situations, MPSLS generates no solution at the beginning of the experiment. For example, if 100 iterations are taken, MPSLS couldn’t generate a solution due to total weight is more than the capacity, so 100 iterations could be too small in randomly selecting a subset for solutions. However, the performance becomes much better if more than 100 iterations are taken.
MPRA vs. MPSLS Data and Performance Analyzing

1. MPRS and MPSLS program running I (KnapsackMPRASMP.java, KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 50, K = 100, seed = 123456789

2) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPRA's Value</td>
<td>136.7569</td>
<td>138.0036</td>
<td>150.0331</td>
<td>157.5208</td>
<td>161.6373</td>
<td>166.00835</td>
<td>169.44061</td>
</tr>
<tr>
<td>MPSLS's Value</td>
<td>154.545</td>
<td>154.545</td>
<td>158.4081</td>
<td>170.935</td>
<td>172.90108</td>
<td>172.90108</td>
<td>182.94771</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

3) Data Collection – Chart
2. MPRS and MPSLS program running II (KnapsackMPRASMP.java, KnapsackMPSSLSSMP.java)

1) Parameters for input:
   \(N = 75, K = 150, \text{seed} = 987654321\)

2) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPRA's Value</td>
<td>186.5419</td>
<td>186.5419</td>
<td>205.3417</td>
<td>214.3574</td>
<td>217.81424</td>
<td>231.71165</td>
<td>231.71165</td>
</tr>
<tr>
<td>MPSLS's Value</td>
<td>198.0729</td>
<td>214.2605</td>
<td>214.2605</td>
<td>230.5996</td>
<td>230.59956</td>
<td>231.16258</td>
<td>240.84283</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

3) Data Collection – Chart

![MPRA vs. MPSLS Chart](image)
3. MPRS and MPSLS program running III (KnapsackMPRASMP.java, KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 100, K = 200, seed = 111222333

2) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>MPRA’s Value</th>
<th>MPSLS’s Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>227.95</td>
<td>230.111</td>
</tr>
<tr>
<td>1,000</td>
<td>230.4318</td>
<td>230.111</td>
</tr>
<tr>
<td>10,000</td>
<td>235.6709</td>
<td>239.1435</td>
</tr>
<tr>
<td>100,000</td>
<td>249.3268</td>
<td>259.473</td>
</tr>
<tr>
<td>1,000,000</td>
<td>260.2875</td>
<td>279.98267</td>
</tr>
<tr>
<td>10,000,000</td>
<td>260.2875</td>
<td>279.98267</td>
</tr>
<tr>
<td>100,000,000</td>
<td>269.16138</td>
<td>279.98267</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

3) Data Collection – Chart
4. MPRS and MPSLS program running IV (KnapsackMPRASMP.java, KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 150, K = 300, seed = 444555666

2) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPRA's Value</td>
<td>342.1534</td>
<td>361.1877</td>
<td>366.5721</td>
<td>404.8529</td>
<td>404.85287</td>
<td>410.46948</td>
<td>419.26617</td>
</tr>
<tr>
<td>MPSLS's Value</td>
<td>386.8752</td>
<td>409.214</td>
<td>409.214</td>
<td>412.8548</td>
<td>441.56464</td>
<td>441.56464</td>
<td>441.56464</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

3) Data Collection – Chart

![MPRA vs. MPSLS Chart](image_url)
5. MPRS and MPSLS program running V (KnapsackMPRASMP.java, KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 200, K = 500, seed = 777888999

2) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPRA's Value</td>
<td>530.5212</td>
<td>554.0493</td>
<td>567.0156</td>
<td>579.1552</td>
<td>612.3609</td>
<td>612.4827</td>
<td>616.0567</td>
</tr>
<tr>
<td>MPSLS's Value</td>
<td>581.8215</td>
<td>593.5054</td>
<td>596.6514</td>
<td>633.7401</td>
<td>633.74005</td>
<td>633.74005</td>
<td>633.74005</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

3) Data Collection – Chart

6. Analysis
   MPRA and MPSLS can run with large number of items while exhaustive search algorithm cannot. On one hand, from developer’s perspective, MPRA is easier for programming. On the other hand MPSLS can generate better result overall. In addition, Massively Parallel program performs better when the large number of items are selected.
Extend Research – Hard Knapsack Problem

Description of Hard Knapsack Problem

One of the important aspect to affect the complicity of the Knapsack problem is: the relation between the item weight and value of the input instance.

The random instance has been used for the experiment of this project. There are no relation between each item’s weight and value. This kind of the input instance is regard that can be solve easily. Because it might have some possibilities for the algorithm that can find the true solution with no need to search till to the end. However if each item’s weight and value has some kind of relationship, it will reduce the possibilities of the easy finding solution.

In this extend research, we will introduce two kind of hard instance for the knapsack problem: the Strong Correlated Instance (SCI) and the Inverse Strong Correlated Instance (ISCI).

Strong Correlated Instance (SCI)

We define the Strong Correlated Instance (SCI) with the condition of the input instance below:

\[ W_i \text{ is randomly chosen in } [0, R]; \]
\[ V_i = W_i + \frac{R}{100} [2] \]

Inverse Strong Correlated Instance (ISCI)

We define the Inverse Strong Correlated Instance (ISCI) with the condition of the input instance below:

\[ V_i \text{ is randomly chosen in } [0, R]; \]
\[ W_i = V_i + \frac{R}{100} [2] \]
Parallel Program Design for Hard Instance Generation

The instance for the knapsack problem would be the items provided for choosing to generate the solution. The total amount of the items would be N. Each item has weight (W) and value (V). In this extend research we are randomly choosing the item weight (W) or value (V) as a float number between the range [0, 10]. Thus we will have two array (list) to store all the items property.

\[ W[i] = \{W_0, W_1, W_2, W_3, \ldots, W_i\} \]
\[ V[i] = \{V_0, V_1, V_2, V_3, \ldots, V_i\} \]

Where i = 1, 2, 3, ..., N.

The number of item (N) is an important key for the definition of the problem size. When it became larger and larger, parallel design should take part in to make the instance generation more efficient.

We will use the same “skip” mechanism for the instance generation like we done before in this project. But we will have different algorithm for different correlated instance generation.

*Strong Correlated Instance (SCI)*

1. Randomly choosing the item weight (W) as a float number between the range [0, 10]
2. Calculate value (V) with the condition
   \[ V_i = W_i + \frac{R}{100} \] [2]

*Inverse Strong Correlated Instance (ISCI)*

1. Randomly choosing the item value (V) as a float number between the range [0, 10]
2. Calculate weight (W) with the condition
   \[ W_i = V_i + \frac{R}{100} \] [2]
MPRA vs. Exhaustive Search Data and Performance Analyzing

Strong Correlated Instance (SCI)

1. MPRA program running I (KnapsackMPRASMP.java)

1) Parameters for input:
   \( N = 30, K = 50, \text{seed} = 11111111 \)

2) Exhaustive Search algorithm result:
   Best total value = 51.496536
   Running time = 12192

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>51.04474</td>
<td>51.19659</td>
<td>51.19659</td>
<td>51.29918</td>
<td>51.381294</td>
<td>51.39779</td>
<td>51.481625</td>
</tr>
<tr>
<td>Value%</td>
<td>99.12%</td>
<td>99.42%</td>
<td>99.42%</td>
<td>99.62%</td>
<td>99.78%</td>
<td>99.81%</td>
<td>99.97%</td>
</tr>
<tr>
<td>Time</td>
<td>33</td>
<td>53</td>
<td>60</td>
<td>112</td>
<td>513</td>
<td>4755</td>
<td>44748</td>
</tr>
<tr>
<td>Time%</td>
<td>0.27%</td>
<td>0.43%</td>
<td>0.49%</td>
<td>0.92%</td>
<td>4.21%</td>
<td>39.00%</td>
<td>367.03%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPRA with SCI 30 Items Chart]
2. MPRA program running II (KnapsackMPRASMP.java)

1) Parameters for input:
   \[ N = 32, \ K = 50, \ \text{seed} = 11111111, \]

2) Exhaustive Search algorithm result:
   \[ \text{Best total value} = 51.588562 \]
   \[ \text{Running time} = 50542 \]

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>51.05223</td>
<td>51.07239</td>
<td>51.300003</td>
<td>51.3</td>
<td>51.360302</td>
<td>51.399105</td>
<td>51.496536</td>
</tr>
<tr>
<td>Value%</td>
<td>98.96%</td>
<td>99.00%</td>
<td>99.44%</td>
<td>99.44%</td>
<td>99.56%</td>
<td>99.63%</td>
<td>99.82%</td>
</tr>
<tr>
<td>Time</td>
<td>39</td>
<td>55</td>
<td>58</td>
<td>121</td>
<td>587</td>
<td>5241</td>
<td>51548</td>
</tr>
<tr>
<td>Time%</td>
<td>0.08%</td>
<td>0.11%</td>
<td>0.11%</td>
<td>0.24%</td>
<td>1.16%</td>
<td>10.37%</td>
<td>101.99%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = \frac{\text{Total Value from MPRA}}{\text{Total Value from Exhaustive Search algorithm}}

*Time% = \frac{\text{Running time of MPRA}}{\text{Running time of Exhaustive Search algorithm}}

4) Data Collection – Chart

![MPRA with SCI 32 Items](image-url)
3. MPRA program running III (KnapsackMPRASMP.java)

5) Parameters for input:
   N = 34, K = 50, seed = 11111111,

6) Exhaustive Search algorithm result:
   Best total value = 51.797703
   Running time = 205125

7) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>51.03737</td>
<td>51.280716</td>
<td>51.483505</td>
<td>51.56761</td>
<td>51.574104</td>
<td>51.677383</td>
<td>51.696213</td>
</tr>
<tr>
<td>Value%</td>
<td>98.53%</td>
<td>99.00%</td>
<td>99.39%</td>
<td>99.56%</td>
<td>99.57%</td>
<td>99.77%</td>
<td>99.80%</td>
</tr>
<tr>
<td>Time</td>
<td>35</td>
<td>73</td>
<td>69</td>
<td>136</td>
<td>645</td>
<td>5806</td>
<td>57831</td>
</tr>
<tr>
<td>Time%</td>
<td>0.02%</td>
<td>0.04%</td>
<td>0.03%</td>
<td>0.07%</td>
<td>0.31%</td>
<td>2.83%</td>
<td>28.19%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

8) Data Collection – Chart
**Inverse Strong Correlated Instance (ISCI)**

1. MPRA program running I (KnapsackMPRASMP.java)
   
   1) Parameters for input:
      
      N = 30, K = 50, seed = 11111111

   2) Exhaustive Search algorithm result:
      
      Best total value = 49.39854
      Running time = 12247

   3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>49.08127</td>
<td>49.285385</td>
<td>49.357117</td>
<td>49.386074</td>
<td>49.39854</td>
<td>49.39854</td>
<td>49.39854</td>
</tr>
<tr>
<td>Value%</td>
<td>99.36%</td>
<td>99.77%</td>
<td>99.92%</td>
<td>99.97%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>35</td>
<td>57</td>
<td>62</td>
<td>113</td>
<td>531</td>
<td>4570</td>
<td>44744</td>
</tr>
<tr>
<td>Time%</td>
<td>0.29%</td>
<td>0.47%</td>
<td>0.51%</td>
<td>0.92%</td>
<td>4.34%</td>
<td>37.32%</td>
<td>365.35%</td>
</tr>
</tbody>
</table>

   *Time unit: Millisecond

   *Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

   *Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPRA with ISCI Chart](image_url)
2. MPRA program running II (KnapsackMPRASMP.java)

1) Parameters for input:
   \[ N = 32, \ K = 50, \ seed = 11111111, \]

2) Exhaustive Search algorithm result:
   Best total value = 49.39995
   Running time = 50531

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>49.09907</td>
<td>49.289696</td>
<td>49.365112</td>
<td>49.39474</td>
<td>49.39995</td>
<td>49.39995</td>
<td>49.39995</td>
</tr>
<tr>
<td>Value%</td>
<td>99.39%</td>
<td>99.78%</td>
<td>99.93%</td>
<td>99.99%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>42</td>
<td>51</td>
<td>63</td>
<td>122</td>
<td>601</td>
<td>5257</td>
<td>45013</td>
</tr>
<tr>
<td>Time%</td>
<td>0.08%</td>
<td>0.10%</td>
<td>0.12%</td>
<td>0.24%</td>
<td>1.19%</td>
<td>10.40%</td>
<td>89.08%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond
*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm
*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPRA with ISCI 32 Items Chart](image)
3. MPRA program running III (KnapsackMPRASMP.java)

1) Parameters for input:
   N = 34, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 49.39859
   Running time = 208291

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>49.07338</td>
<td>49.28042</td>
<td>49.28042</td>
<td>49.375618</td>
<td>49.39859</td>
<td>49.39859</td>
<td>49.39859</td>
</tr>
<tr>
<td>Value%</td>
<td>99.34%</td>
<td>99.76%</td>
<td>99.76%</td>
<td>99.95%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>40</td>
<td>52</td>
<td>66</td>
<td>125</td>
<td>648</td>
<td>6249</td>
<td>48134</td>
</tr>
<tr>
<td>Time%</td>
<td>0.02%</td>
<td>0.02%</td>
<td>0.03%</td>
<td>0.06%</td>
<td>0.31%</td>
<td>3.00%</td>
<td>23.11%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPRA / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPRA / Running time of Exhaustive Search algorithm

4) Data Collection – Chart
Analysis

From the experiment above, we can see that MPRA could present a solid performance even in solving the hard knapsack problem. This kind of the knapsack problem uses a hard instance for input, and it might reduce the possibility of finding an easy solution, but MPRA uses a heuristics algorithm as the approach so it can still generate a good result after the first 100 iterations (beginning point), then became normal and generate good result after that. MPRA can generate acceptably good result with 10 million iterations or more, even there is SCI or ISCI.
MPSLS vs. Exhaustive Search Data and Performance Analyzing

Strong Correlated Instance (SCI)

1. MPSLS program running I (KnapsackMPSLSSMP.java)

   1) Parameters for input:
      N = 30, K = 50, seed = 11111111

   2) Exhaustive Search algorithm result:
      Best total value = 51.496536
      Running time = 12192

   3) Data Collection – Table*


<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>51.142197</td>
<td>51.142197</td>
<td>51.2895</td>
<td>51.39609</td>
<td>51.494225</td>
<td>51.494225</td>
<td>51.496536</td>
</tr>
<tr>
<td>Value%</td>
<td>99.31%</td>
<td>99.31%</td>
<td>99.60%</td>
<td>99.80%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>40</td>
<td>51</td>
<td>77</td>
<td>126</td>
<td>657</td>
<td>5790</td>
<td>54416</td>
</tr>
<tr>
<td>Time%</td>
<td>0.33%</td>
<td>0.42%</td>
<td>0.63%</td>
<td>1.03%</td>
<td>5.39%</td>
<td>47.49%</td>
<td>446.33%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS with SCI 30 Items Chart](chart.png)
2. MPSLS program running II (KnapsackMPSLSSMP.java)

1) Parameters for input:
   \( N = 32, K = 50, \) seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 51.588562
   Running time = 50542

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>50.92084</td>
<td>51.199173</td>
<td>51.395596</td>
<td>51.40648</td>
<td>51.468353</td>
<td>51.493683</td>
<td>51.49995</td>
</tr>
<tr>
<td>Value%</td>
<td>98.71%</td>
<td>99.25%</td>
<td>99.63%</td>
<td>99.65%</td>
<td>99.77%</td>
<td>99.82%</td>
<td>99.83%</td>
</tr>
<tr>
<td>Time</td>
<td>38</td>
<td>58</td>
<td>65</td>
<td>207</td>
<td>738</td>
<td>6448</td>
<td>64641</td>
</tr>
<tr>
<td>Time%</td>
<td>0.08%</td>
<td>0.11%</td>
<td>0.13%</td>
<td>0.41%</td>
<td>1.46%</td>
<td>12.76%</td>
<td>127.90%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS with SCI 32 Items](image)
3. MPSLS program running III (KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 34, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 51.797703
   Running time = 205125

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>51.02038</td>
<td>51.25256</td>
<td>51.468826</td>
<td>51.49228</td>
<td>51.655746</td>
<td>51.699436</td>
<td>51.797703</td>
</tr>
<tr>
<td>Value%</td>
<td>98.50%</td>
<td>98.95%</td>
<td>99.37%</td>
<td>99.41%</td>
<td>99.73%</td>
<td>99.81%</td>
<td><strong>100.00%</strong></td>
</tr>
<tr>
<td>Time</td>
<td>42</td>
<td>66</td>
<td>75</td>
<td>141</td>
<td>811</td>
<td>7026</td>
<td>75518</td>
</tr>
<tr>
<td>Time%</td>
<td>0.02%</td>
<td>0.03%</td>
<td>0.04%</td>
<td>0.07%</td>
<td>0.40%</td>
<td>3.43%</td>
<td>36.82%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS with SCI 34 Items](chart)
Inverse Strong Correlated Instance (ISCI)

1. MPSLS program running I (KnapsackMPSLSSMP.java)
   1) Parameters for input:
      \( N = 30, \, K = 50, \, \text{seed} = 11111111 \)
   2) Exhaustive Search algorithm result:
      Best total value = 49.39854
      Running time = 12247
   3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>48.78592</td>
<td>49.279034</td>
<td>49.279034</td>
<td>49.364822</td>
<td>49.394012</td>
<td>49.39474</td>
<td>49.39854</td>
</tr>
<tr>
<td>Value%</td>
<td>98.76%</td>
<td>99.76%</td>
<td>99.76%</td>
<td>99.93%</td>
<td>99.99%</td>
<td>99.99%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>36</td>
<td>51</td>
<td>63</td>
<td>132</td>
<td>653</td>
<td>5893</td>
<td>57998</td>
</tr>
<tr>
<td>Time%</td>
<td>0.29%</td>
<td>0.42%</td>
<td>0.51%</td>
<td>1.08%</td>
<td>5.33%</td>
<td>48.12%</td>
<td>473.57%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

![MPSLS with ISCI 30 Items Chart](chart.png)
2. MPSLS program running II (KnapsackMPSLSSMP.java)

1) Parameters for input:
   \[N = 32, \ K = 50, \ \text{seed} = 11111111,\]

2) Exhaustive Search algorithm result:
   \[\text{Best total value} = 49.39995\]
   \[\text{Running time} = 50531\]

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>48.75013</td>
<td>48.987312</td>
<td>49.1364</td>
<td>49.22453</td>
<td>49.372295</td>
<td>49.39538</td>
<td>49.39854</td>
</tr>
<tr>
<td>Value%</td>
<td>98.68%</td>
<td>99.16%</td>
<td>99.47%</td>
<td>99.64%</td>
<td>99.94%</td>
<td>99.99%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Time</td>
<td>36</td>
<td>54</td>
<td>73</td>
<td>137</td>
<td>730</td>
<td>6547</td>
<td>65556</td>
</tr>
<tr>
<td>Time%</td>
<td>0.07%</td>
<td>0.11%</td>
<td>0.14%</td>
<td>0.27%</td>
<td>1.44%</td>
<td>12.96%</td>
<td>129.73%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond

*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm

*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

MPSLS with ISCI
32 Items
3. MPSLS program running III (KnapsackMPSLSSMP.java)

1) Parameters for input:
   N = 34, K = 50, seed = 11111111,

2) Exhaustive Search algorithm result:
   Best total value = 49.39859
   Running time = 208291

3) Data Collection – Table*

<table>
<thead>
<tr>
<th>Iterations</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>100,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>48.85489</td>
<td>49.03219</td>
<td>49.189106</td>
<td>49.19948</td>
<td>49.28668</td>
<td>49.298733</td>
<td>49.38171</td>
</tr>
<tr>
<td>Value%</td>
<td>98.90%</td>
<td>99.26%</td>
<td>99.58%</td>
<td>99.60%</td>
<td>99.77%</td>
<td>99.80%</td>
<td>99.97%</td>
</tr>
<tr>
<td>Time</td>
<td>38</td>
<td>53</td>
<td>68</td>
<td>140</td>
<td>732</td>
<td>6636</td>
<td>70144</td>
</tr>
<tr>
<td>Time%</td>
<td>0.02%</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.07%</td>
<td>0.35%</td>
<td>3.19%</td>
<td>33.68%</td>
</tr>
</tbody>
</table>

*Time unit: Millisecond
*Value% = Total Value from MPSLS / Total Value from Exhaustive Search algorithm
*Time% = Running time of MPSLS / Running time of Exhaustive Search algorithm

4) Data Collection – Chart

MPSLS with ISCI
34 Items

Value%
Analysis

From the experiment above, like MPRA, we can see that MPSLS could present a solid performance even in solving the hard knapsack problem. This kind of the knapsack problem uses a hard instance for input, and it might reduce the possibility of finding an easy solution, but MPSLS uses a heuristics algorithm as the approach so it can still generate a good result after the first 100 iterations (beginning point). MPSLS can generate acceptably good result with 10 million iterations or more, even there is SCI or ISCI.
**Future Work**

In this project, we investigated and experimented with two approaches of Massively Parallel algorithm - MPRA and MPSLS. We found that they have better performance than the original Exhaustive Search algorithm. Moreover, the MPSLS could have a better result than MPRA.

We also discovered that the capacity K affects the performance of all the algorithms. We can only put a few items in the knapsack if K is too small, or we can put almost all the items in the knapsack if K is too large. For MPRA, if K is too small, the run time of each iteration will increase. For MPSLS, if K is too small, it may fail to find suitable solutions in many situations. We are also not sure about what could happen if K is too large.

In the future we can run more experiments by only changing the capacity K of the knapsack. This allows us to further investigate the impact of the capacity K on the performance of the algorithms.

**Conclusion**

Through this project, we have introduced a new algorithm to solve the Knapsack problem – Massively Parallel algorithm. There are two different approaches of this algorithm, one is Massively Parallel Randomized Approximation (MPRA) algorithm, and the other is Massively Parallel Stochastic Local Search (MPSLS) algorithm. Compare to the original Exhaustive Search algorithm (brute force search), the MPRA and MPSLS try to generate a good approximation of the solution using heuristics search. MPRA uses a randomly picking algorithm to try different possibilities in an efficient way. MPSLS uses a local search algorithm that starts with a random point, then search the solution with a heuristic logic.

Both of the Massively Parallel algorithms have better performance than the original Exhaustive Search algorithm. MPRA and MPSLS are faster and produce better result in the experiments compare to the Exhaustive Search algorithm. Overall, MPSLS generates better result than MPRA in our experiments. Moreover, we can solve a Knapsack problem with big problem size using Massively Parallel that the original algorithm cannot. This is an important advantage of the Massively Parallel algorithm we developed.
References


