Subset Sum Problem

Project Report

Alimuddin Khan(aak5031@g.rit.edu)

Karan Jariwala(kkj1811@g.rit.edu)

Rochester Institute of Technology, NY, USA

---

1. Overview of Subset problem

- **Problem statement**: Given a set of positive integers *N* and a sum *S*, does there exist a subset N' ⊆ N such that sum of all the elements in that subset is S.
- **Description**:
- Mathematically, it is denoted as,

$$S = \sum_{n \in N'} n$$
$$N' \in PowerSet(N)$$

- The subset-sum problem is a well-known non-deterministic polynomial-time complete (NP-complete) decision problem and it is also a special case of 0-1 Knapsack problem.

2. Different Approaches to solve subset sum problem

- **Naïve approach**: A naive approach is to solve the subset sum problem by the *brute force* technique where it finds all the possible 2^n subset of n element set. Iterate over all the 2^n subsets and find the sum of each subset and compare it with the target sum. The time complexity for this approach is $O(n2^n)$ which is an exponential time algorithm.
- So, we need a better algorithm to solve a subset sum problem for the larger input size as well as to parallelize such algorithm so that we can utilize all the available resources in order to reduce the running time.
- The fig(1) shows the graph of Number of elements vs Time complexity for different algorithms

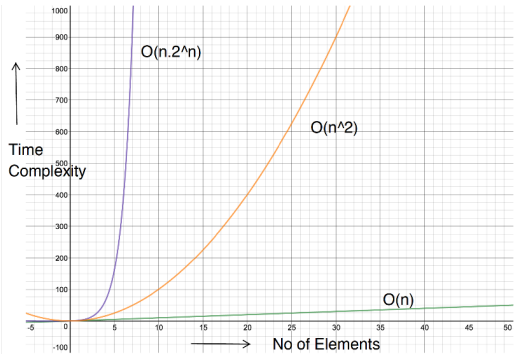


fig (1) Subset Sum algorithm comparison

Below is the data used to generate the above graph.

(1) Exhaustive search(Brute force search algorithm)

- Time-complexity: O($n2^n$)
- Space-complexity: O(2^n)

n	Time Complexity
10	10,240
20	20,971,520
30	32,212,254,720

(2) Dynamic Algorithm

- Time-complexity: O(2^n)
- Space-complexity: O(sum*n)
  (NOTE: Sum is the target sum we are looking for)

n	Time Complexity
10	1024
20	1,048,576
30	1,073,741,824

(3) Two-list Algorithm

- Time-complexity: O($n2^{n/2}$)
- Space-complexity: O($n2^{n/2}$)

n	Time Complexity
10	320
20	20,480
30	983,040

3. Analysis of Research Papers

1. Research Paper 1

a. Title: Parallel solution of the subset-sum problem: an empirical study

b. Problem statement:

The paper addresses how the subset sum problem is parallelized on 3 contemporary but different architectures 128-processor XMT, 16-processor IBM x3755, 240-core NVIDIA FX 5800. They have used two different algorithms to solve the subset sum problem on different machines due to different architecture. They have faced a problem where it was difficult to parallelize for a conventional shared memory machine because of certain limitations of the architecture. Later, they compare the performance on different machines and concluded that the performance varies as the problem sizes changes and changes due to different machine architecture.

c. novel Contribution:

The three machines they have used are:

1. 128-processor Cray Extreme Multithreading (XMT) massively multithreaded machine, 500 MHz, 1-TB. It was easy to parallelize the algorithm on the Cray XMT primarily because of the word-level locking that uses flag bits associated with each 64-bit word. The process ensures that only one thread can access a word at a time and the selected word can be concurrently updated using word-level locking. The algorithm shows very good scaling for large problems and sustained performance as the problem size increases but it shows poor scaling for small problem sizes. It performs best for problem sizes of 10^{12} bits or more.

2. 16-processor IBM x3755 shared memory machine, 2.4-GHz Opterons,16-core, 64-GB. There was no word-level locking feature in this machine, so they need some alternative word algorithm that can implement an efficient solution. The alternating word approach implemented using OpenMP. Two words of row 'i' are modified by executing a bit-wise OR operation with the source word from row 'i-1', suitably shifted. The algorithm performs very well on medium-sized problems that fit within its 64-GB memory but it shows poor scalability as the number of processors increases and degrade in performance as the problem size increases. It can handle a maximum problem sizes of 10^{11} bits.
3. 240-core NVIDIA FX 5800 graphics processing unit (GPU), 1.296 GHz, 240 cores, 4-GB device memory. There was no word-level locking feature in this machine as well, so need some alternative word algorithm that can implement an efficient solution. The machine has a specialized graphical processor unit(GPU) that can support large numbers of threads. They have implemented a word approach in CUDA. It uses two kernel to get the full parallel performance For every row in the table, we then call kernel-1 to copy the row before it and kernel-2 to perform the alternating word approach of setting the bits. GPU performs well for problems whose tables fit within the device memory. It performs best for small problem sizes that have tables of size approximately 10^{10}.

d. Useful for our project:

The useful information from this paper is about the different machine architectures and their parallel approach limitation and how the approaches changes based on different machine architecture.

2. Research Paper 2

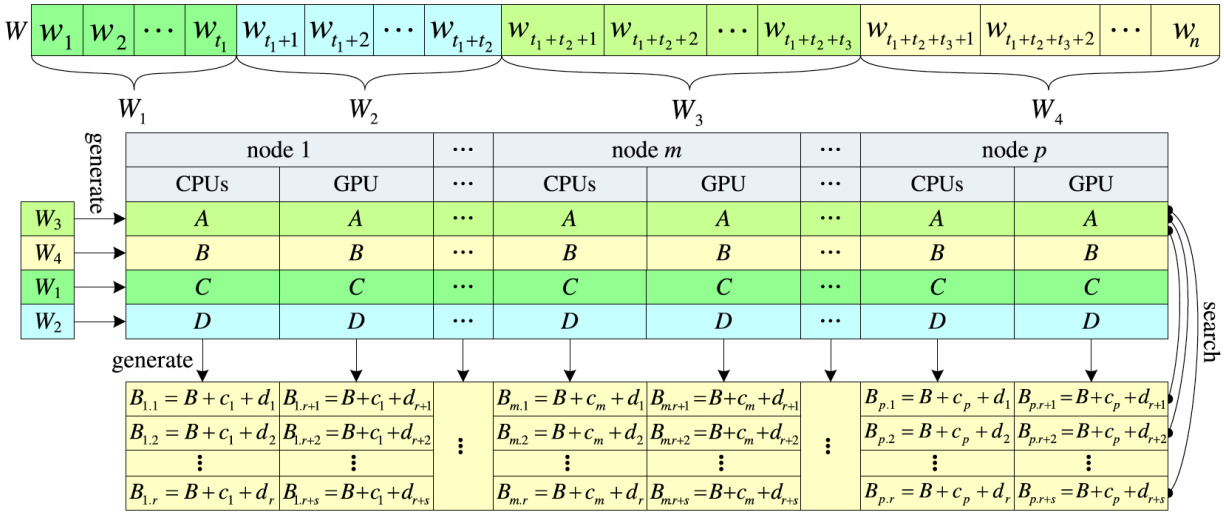
a. Title: A novel cooperative accelerated parallel two-list algorithm for solving the subset-sum problem on a hybrid CPU–GPU cluster

b. Problem Statement:

The paper proposed a novel heterogeneous cooperative computing approach to solve the subset sum problem on a hybrid CPU-GPU cluster, which can take full use of all available computational resources of a cluster. The paper implemented a cost-optimal two-list algorithm. They have face 2 problems while solving the subset sum problem, unbalanced workload distribution and the computational overhead. To solve these problems, they have design a communication-avoiding workload distribution scheme for the parallel two-list algorithm.

c. Novel Contribution:

In the Hybrid MPI-OpenMP-CUDA programming model, they have used MPI for communication operations among cluster nodes and OpenMP and CUDA are used to drive the CPUs and GPU to perform computation.



fig(2) communication avoiding work-load distribution scheme

The figure uses a "communication avoiding work-load distribution scheme" for solving the two mentioned issues.

(1) Solving unbalanced load:

(I) Intra-node load balancing: Partition the load equally between CPU and GPU according to the partition ratio. The partition ratio is given by the formula:

$$R = \frac{1}{1 + \frac{m \times V_{gpu}}{(m-1) \times V_{cpu}}}.$$

(II) Inter-node load balancing: Partition the load evenly between all nodes as per their partition ratio. If all nodes are of same size (Same no-of CPUs and GPUs) and have physically same computational capability, then divide the load equally amongst all nodes.

(2) Solving communication-overhead:

(I) Inter-node Communication: Only the input W needs to be transferred between master and the slave nodes. All nodes compute their results independently and then the result is transferred to the master node at the end. Therefore, it requires very little communication.

(II) Intra-node communication: Only the input set W is copied from host to device and then they compute results independently and device returns result to the host in the end. Hence, very little communication between host and device.

- To analyze the scalability of the improved heterogeneous cooperative implementation, the author has compared the model with best sequential CPU implementation, the single-node CPU-only implementation, the single-node GPU-only implementation, and the hybrid MPI-OpenMP implementation with same cluster configuration. The author has also conducted a strong and weak scaling to evaluate the scalability. From the experiment conducted by the author, they concluded that using hybrid MPI-CUDA implementation, speedup was approximately 30 times more than the best sequential algorithm for the cluster with 32 nodes.

d. Useful for our project:

From this paper, the Proper load balancing scheme (Inter-node and Intra-node) and the minimum communication (Inter-node and Intra node) overhead is very useful to achieve the higher speedup on a cluster and we are planning to handle this two cases for our two-list algorithm.

3. Research Paper 3:

a. Title: GPU implementation of a parallel two-list algorithm for the subset-sum problem

b. Problem Statement:

The paper proposed an efficient Two-List algorithm for solving the subset-sum problem on Graphics Processing Unit(GPU) using Compute Unified Device Architecture(CUDA). The algorithm composed of 3 stages which are a generation stage, a pruning stage, and a searching stage. One of the important factor is the task distribution between CPU and GPU, efficiently utilizing GPU memory, and low CPU-GPU communications. The paper addresses 2 problems while solving a subset sum problem on a GPU. First is Recursive divide and conquer strategy used in generation stage which is not well supported by current GPU and the second is Redundancies in pruning and search stages.

c. Novel Contribution:

The author has proposed a sequential and parallel two-list algorithm on a Cluster parallel machine as well as on a hybrid Cluster parallel machine. The sequential two-list algorithm is divided into parts: (1) The generation stage where the input list has been feed to the algorithm and it produce the sorted two sub list. (2) The search stage which is to find the output or a solution from the combinations of two sorted sub lists. The parallel two-list

algorithm is divided into 3 stages. The extra stage is a pruning stage which is designed to reduce the thread overhead of each thread. The other two stages are similar but here they are executed on many threads parallelly.

The paper address 2 problem which was discussed above. The author has solved both the issues. The first problem is solved by replacing the recursion issue on GPU with the iterative approach. And the second problem which is a redundancy issue is solved by presenting the modified and efficient generation stage, pruning stage, and search stage. The modified generation stage worked as while generating lists say A and B, it discards those subset sums which are larger than target sum M. The modified pruning stage and the search stage has been improved by using different lemma's. While analyzing the execution time of sequential and the modified parallel program, the parallel generation stage and the search stage obtains better efficiency.

d.Useful for our project:

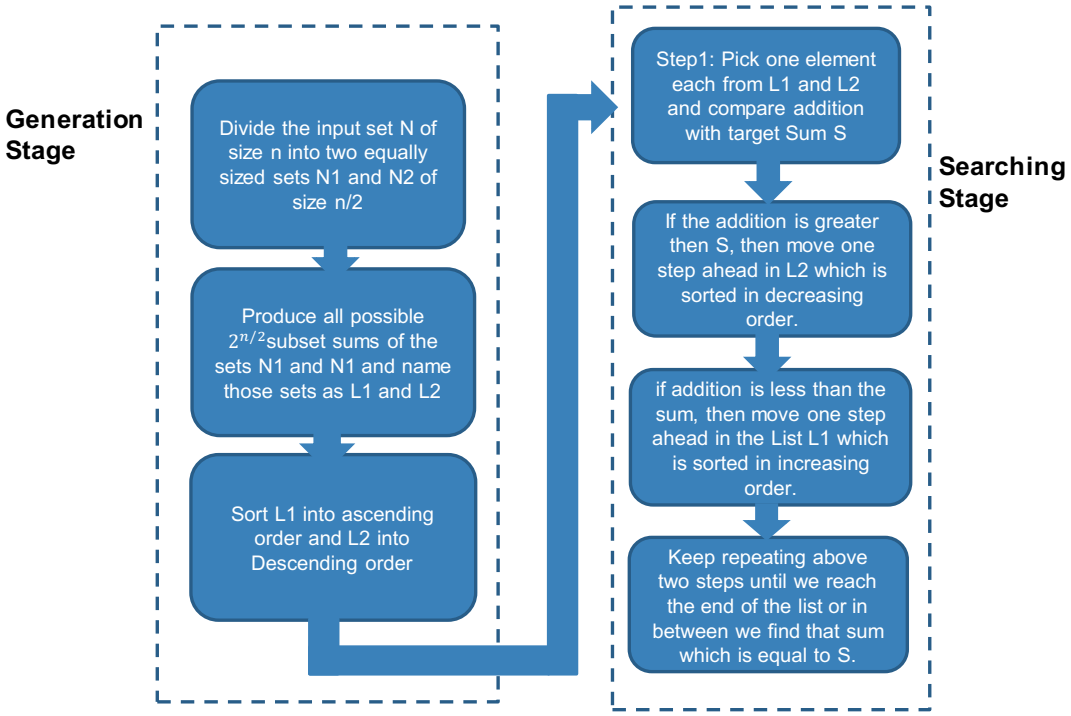
The author has proposed an efficient parallel two-list algorithm. We are planning to implement the same with some modification in the data structures to obtain the speedup and the efficiency.

4. Sequential Program Design Approach:

The sequential two-list algorithm consist of two stages:

1. Generation Stage
2. Search Stage.

Below is the block diagram of the sequential approach.



fig(3) Sequential Approach

The sequential algorithm is as follows:

A} **Data generation stage:**

def **generateSeqSubsets**(Set S):

```
Create a set A = {}
Add 0 to set A
// Now set A has A ={0}
Create and empty set A' = {}
For each element in set S:
      for each subsetsum in A:
            add(subsetsum + element) in A'
      Merge A' in A
      clear A'
return A
```

Explanation: The above data generation can be well understood using following example;

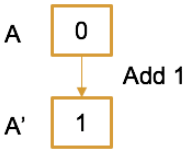
Lets say we have set S = {1,5, 8, 11}

Step 1 : Initialize A

Step 2 : Initialize A' = {}

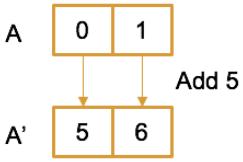
Step 3 : Repetitively Pick one element from S

3(a) Pick 1:

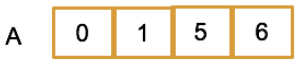


3(b) Merge A and A' int A

Step 3(b) : Pick 5



3(b) Merge A and A' int A



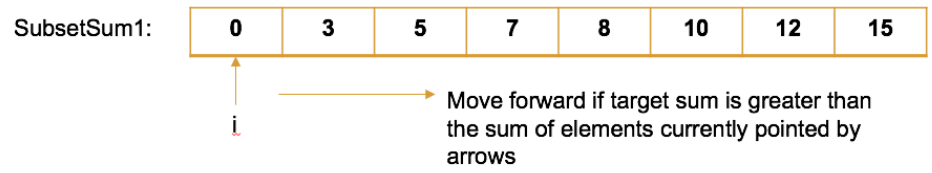
Continue these steps for all elements present in S and we will get all the possible subset sums

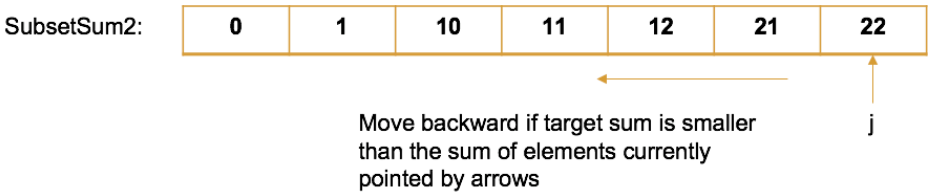
B} **Search Stage:**

In this stage we give sorted sublist sums L1, L2 and target sum as the input and programs returns true or false depending on the presence of the target sum in the given subset sums.

```
def search(L1, L2, targetSum):
      i = start index of L1 i.e. 0
      j = last index of L2 i.e. L2.length – 1
      While( i < L1.length() && j >= 0){
            if(L1(i) + L2(j) == targetSum )
                  return true;
            else if(L1(i) + L2(j) < targetSum ) {
                  // move ahead in lists 1
                  i++;
            }else{
                  //move behind in list 2
                  j--;
            }
      }
      return false;
```

Explanation: Consider we have two subset sums and we are looking for target sum 15.



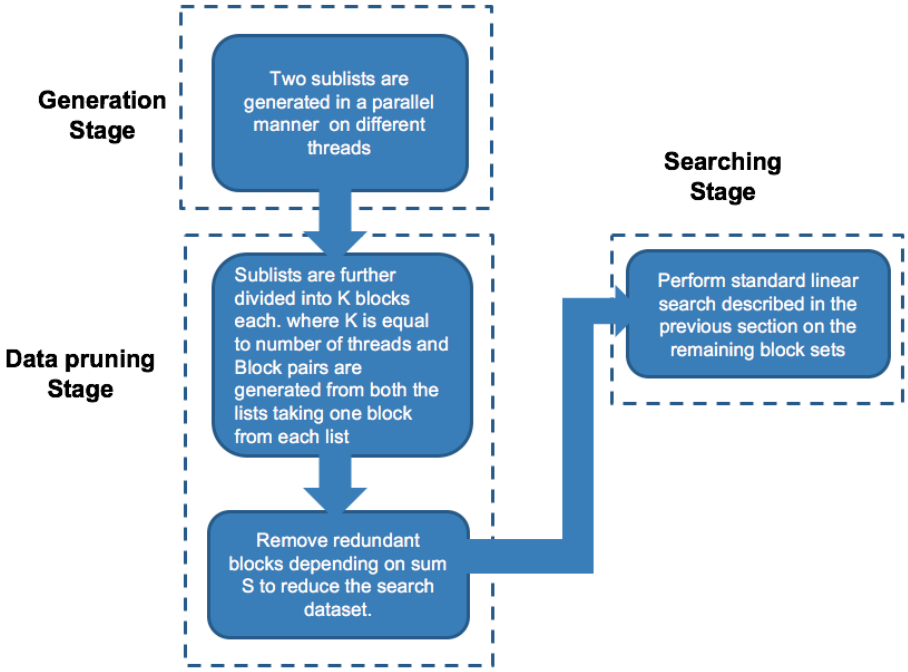


If we run the above algorithm then we stop by 3 and 12 in lists L1 and L2 respectively as 12 +3 = 15. On reaching this position we will return true.

5. Parallel Program Design Approach

The parallel design approach is similar to the sequential approach but the two-list generation stage will now happen in parallel and the same sequential approach.

Below is the block diagram of the parallel approach.



fig(4) Parallel approach

The Parallel algorithm is as follows:

Similar to sequential approach, it also ha two stages,

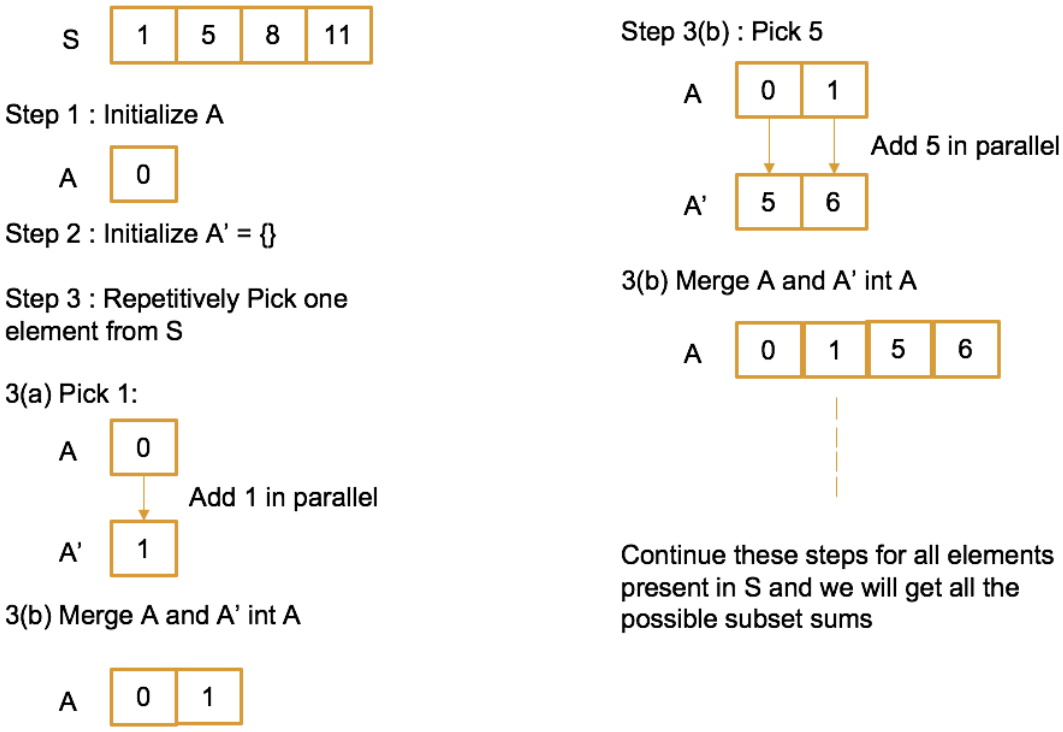
A} Data Generation Stage:

The algorithm for parallel data geration stage is as follows;

```
def generateParallelSubsets(Set S):
   Create a set A = {}
   Add 0 to set A  Now set A has A ={0}
   Create and empty set A' = {}
   For each element in set S:
        // this loop will be done in parallel
        parallelFor each subsetsum in A:
             add(subsetsum + element) in A'
        Merge A' in A
        clear A'

   return A
```

Explanation: Given set S = {1, 5, 8, 11}. If we run the above algorithm it goes through following stages;



As you can see addition of element can be done independently in parallel. This increases the speed of data generation stage significantly. Ideally we get the speedup of K where K is the number of cores.

B} Data Search Stage: Data search stage is same as mentioned in the sequential approach

6. Developer's manual

The subset sum sequential and parallel program has been compiled and run on RIT CS machines. We have used Nessie and Kraken to test our code. The Nessie has 16 cores and kraken has 80 cores. First, to compile the code, you need to add the java classpath to include PJ2 distribution. The PJ2 library is installed almost on all the RIT CS machines that includes Nessie as well as kraken.

Please follow the below steps:

I. Set the java classpath:

Bash shell

```
$ export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
```

csh shell:

```
$ setenv CLASSPATH .:/var/tmp/parajava/pj2/pj2.jar
```

II. Compile the java files as:

```
Javac *.java
```

7. User's manual

To run the sequential version of the program, enter the following command:

```
Java pj2 SubsetSumSeq <N> <T> <seed> <R>
```

To run the parallel version of the program, enter the following command:

```
Java pj2 cores=<K> SubsetSumSmp <N> <T> <seed> <R>
```

```
Here,
<K> = Number of cores
<N> = Number of elements in an input set
<T> = Target sum
<seed> = Random seed
<R> = Range value of the input elements
```

Example:

```
aak5031@nessie:~/courses/fall_2016/parallel/researchProject/finalSubmission$ java pj
2 SubsetSumSeqDemo 10 123 123456 100
Input Set: {   96   97   67   83    4   85   38   23    7   47 }

First Half list: {   96   97   67   83    4 }
Second Half list:
{   85   38   23    7   47 }

Possible subset sum from list1: {0, 4, 67, 71, 83, 87, 96, 97, 100, 101, 150, 154, 1
63, 164, 167, 168, 179, 180, 183, 184, 193, 197, 246, 247, 250, 251, 260, 264, 276,
280, 343, 347}
Possible subset sum from list2: {0, 7, 23, 30, 38, 45, 47, 54, 61, 68, 70, 77, 85, 9
2, 108, 115, 123, 130, 132, 139, 146, 153, 155, 162, 170, 177, 193, 200}

Target sum: 123
Is Target sum found? true
```

fig(5) Demo program

Note: The output fo generated list and all possible subset sums is done only in SubsetSumSeqDemo.java file. Remaining files only print the true or false depending on the availability of the target sum.

8. Strong scaling performance

- We ran our program for four different sizes of input.
- For each input size we ran our program for 1, 2, 4, 8 cores.
- This is the data that we obtained for strong scaling.

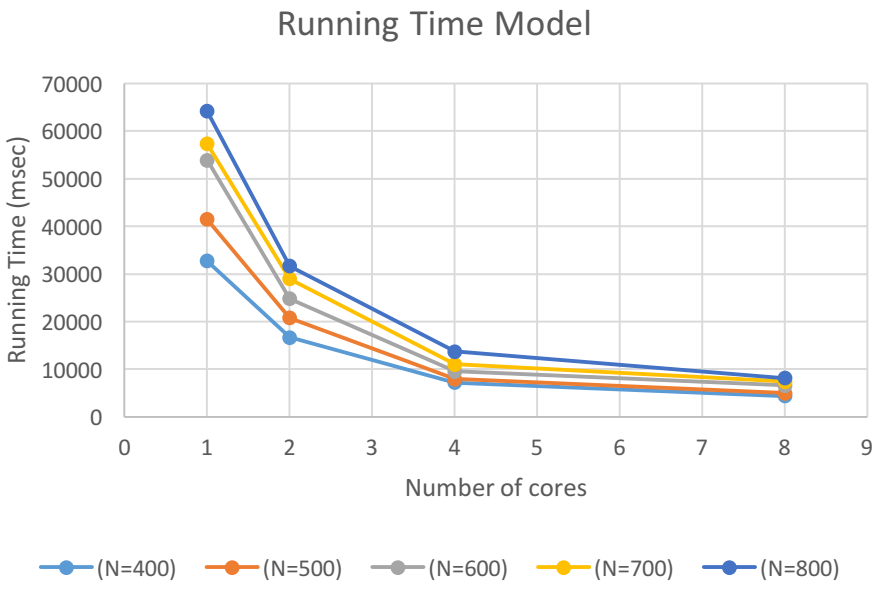
(NOTE: For all of these problems Range selected was 1000 i.e. R = 1000

N	cores	Running Time	SpeedUp	Efficiency
400	1	32732	1	1
	2	16708	1.96	0.98
	4	7191	4.55	1.13
	8	4409	7.42	0.92
500	1	41474	1	1
	2	20781	1.99	0.99
	4	7976	5.23	1.3
	8	5018	8.26	1.03
600	1	53904	1	1
	2	24784	2.17	1.08
	4	9544	5.6	1.4
	8	6620	8.14	1.02
700	1	57357	1	1
	2	28970	1.98	0.99
	4	11038	5.19	1.29
	8	7432	7.72	0.96
800	1	64183	1	1
	2	31692	2.02	1.01
	4	13718	4.67	1.16
	8	8158	7.86	0.98

table(1) Strong scaling Performance

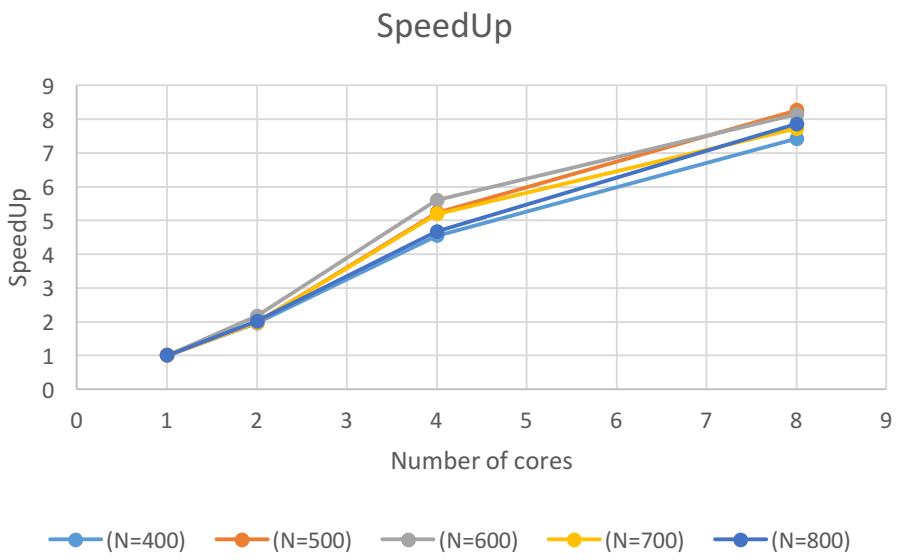
Graphical Representaion of above data;

a. Running time



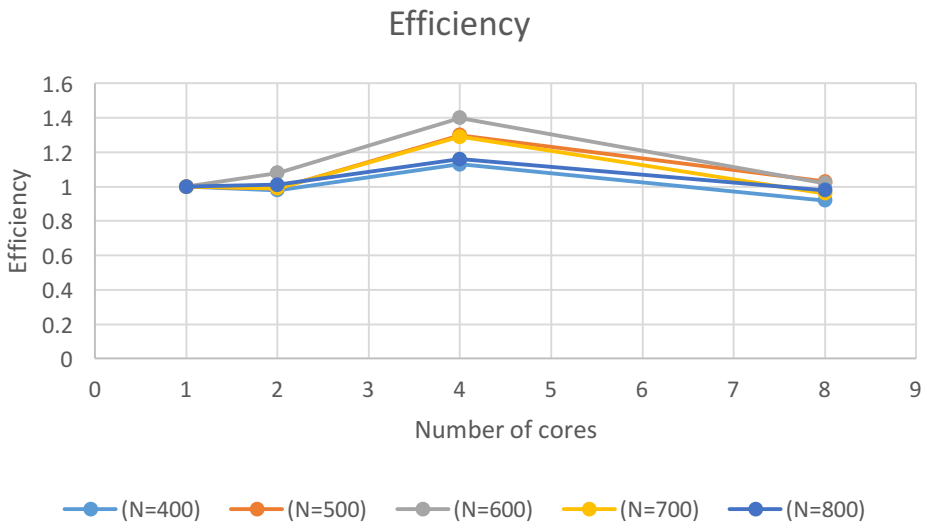
fig(6) Strong scaling running time model

b. speedup



fig(7) String scaling speed up

c. Efficiency



fig(8) Strong scaling Efficiency

Observation: We can see that we are getting efficiency equal to or more that 1. Sometimes we get efficiency more than one because we are dealing with BitSet data structure and with increase in number of cores, number of bits being handled in a single CPU cycle increases exponentially and not linearly.

9. Weak scaling performance

Taking weak scaling performance was a bit difficult as our data was not increasing linearly with increase in umber of elements in the data set. Instead it was increasing 2^n times for the given input of size N. Similarly it also had dependency on the range in which all elements lie.

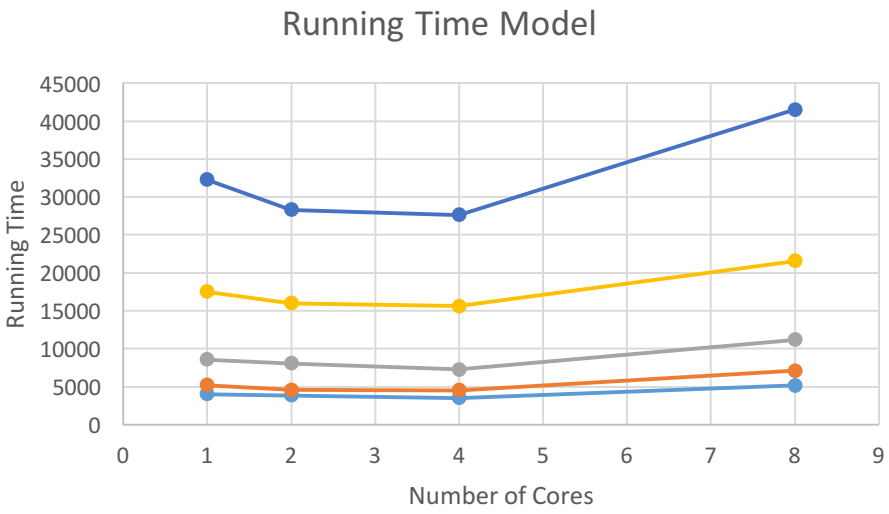
For a given input size we increased the number of elements by a small factor(as it was increasing program size exponentially) and range of inputs in a quadratic fashion to match with the number of cores increment. Following is the data that we obtained after running program;

N	Range	cores	Running Time	Sizeup	Efficiency
400	500	1	4022	1	1
420	625	2	3834	2.09	1.04
430	782	4	3514	4.57	1.14
435	976	8	5132	6.3	0.79
500	600	1	5182	1	1
520	750	2	4578	2.26	1.13
530	938	4	4515	4.59	1.14
535	1172	8	7106	5.83	0.73
600	700	1	8553	1	1
620	875	2	8072	2.12	1.06
630	1094	4	7247	4.72	1.18
635	1367	8	11196	6.11	0.77
700	800	1	17513	1	1
720	1000	2	15991	2.19	1.09
730	1250	4	15609	4.48	1.12
735	1563	8	21568	6.5	0.81
800	900	1	32272	1	1
820	1125	2	28320	2.28	1.14
830	1406	4	27640	4.67	1.16
835	1758	8	41524	6.2	0.78

table(2) Weak scaling performance

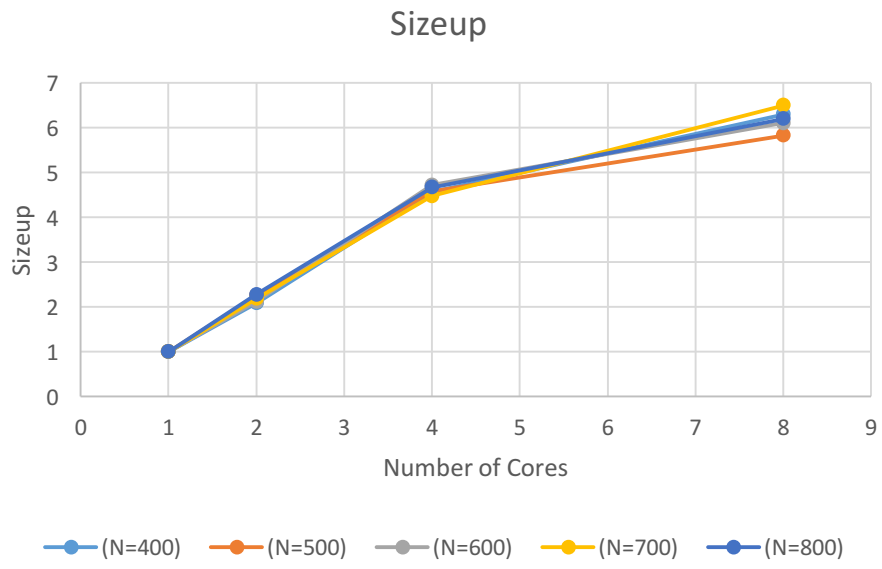
Grpahical representation of the above data looks like as follows;

a. Running Time



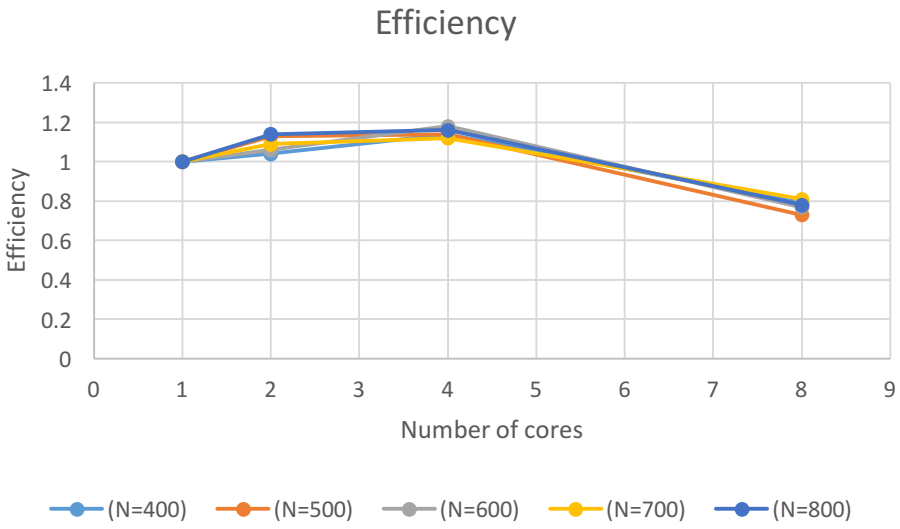
fig(8) Weak scaling running time model

b. Sizeup



fig(9) Weak scaling sizeup

c. Efficiency:



fig(10) Weak scaling efficiency

Observation: As we can see efficiency is really good upto 4 cores. When we increase the number of cores to 8 we get the efficiency around 0.8. The reason for this reduced efficiency is because of the overhead required to generate the parallel for loops and also the sequential dependency in the program.

10.Future work

The main aim of our project was to implement the efficient parallel algorithm to solve the subset sum problem which gives the efficient time and space complexity. The parallel two-list algorithm works better than the dynamic approach as well as the naive brute force technique. The future work is to implement the parallel two-list algorithm on CPU-GPU hybrid cluster machine and try to achieve near perfect speedup and efficiency. Also, we would like to modify our search algorithm, where we are currently searching the element linearly, to logarithmic algorithm which will further reduce the running time and enhance the speedup and efficiency.

11.Things we learned from the project

We learned different algorithms to solve the subset problem and which algorithm gives the higher speedup and efficiency when run on cluster parallel machine. From the research, we concluded that two-list algorithm is by far the best-known algorithm for solving subset sum problem.

We used many data structures such as Integer[], ArrayList<Integer>, ArrayList<BitSet64>, Set<Integer>, HashSet, BitSet<Integer>. Many of them have their limitations. For 'n' element set, the algorithm generates approximately 2^n number of possible subset sums and as 'n' value increases, the number of possible subset sum elements grows exponentially. We can't use

Integer[] because it is impossible to allocate a huge amount of contiguous memory(2^n) for very high 'n' number. The size of ArrayList<Integer> and ArrayList<BitSet64> increases during runtime but when we tried to implement the parallel approach where multiple threads try to add elements at end, we faced a concurrent modification issue. We can't implement our own vbl because it will also result in same error while reduction process. We also tried to use Set, HashSet and TreeSet and we were not facing the same issue as before. But, here we were saving the space complexity but were compromising with the time complexity because during the two-list generation stage, we needed the element access time complexity as $O(1)$ but here we were getting an $O(n)$ and the amount of time it took to get the output was quite high. The only option we were left with in our scenario is use BitSet<Integer> and by using this data structure, we were getting the excellent speedup and efficiency.

12.Team Contribution

- We started researching on the topic together. We read 3-4 research papers each and then finally agreed on Two list algorithm to solve the subset sum problem.
- Alimuddin's contribution:
  - Design and implementation of sequential approach
- Karan Jariwala's contribution:
  - Design and implementation of parallel approach
- We worked together for all the four presentations and final report.

13.REFERENCES

[1] A. Kaminsky. Parallel Java 2 Library. http://www.cs.rit.edu/~ark/pj2.shtml.

[2] Saniyah S. Bokhari (2012), "Parallel solution of the subset-sum problem: an empirical study", Concurrency Computat.: Pract. Exper. 2012, 24: 2241–2254.

[3] Lanjun Wan, Kenli Li, Keqin Li(2016), "A novel cooperative accelerated parallel two-list algorithm for solving the subset-sum problem on a hybrid CPU–GPU cluster", Journal of Parallel and Distributed Computing Volume 97, July 2016, Pages 112–123

[4] Lanjun Wan, Kenli Li, Jing Liu and Keqin Li(2014), "GPU implementation of a parallel two-list algorithm for the subset-sum problem", Concurrency Computat.: Pract. Exper. 2015; 27:119–145.