Subset Sum Problem – Parallel Solution

Project Report

Harshit Shah
hrs8207@rit.edu
Rochester Institute of Technology, NY, USA

1. Overview

Subset sum problem is NP-complete problem which can be solved in pseudo polynomial time. Here, time increases proportional to the number of input elements multiplied by the sum of their values. For larger input size the problem grows exponentially. To address this concern and achieve faster solution, one way is to parallelize the current solution of the problem. In this project we will see that using parallel java 2 (pj2) library [2], we can able to achieve significant speedup.

2. Computational Problem

We are using the variant of the subset sum problem which is discussed in the paper [1]. Input set is consist n elements, A = {a_1, a_2, …, a_n}, where $|A| = n$. Here, each element a_i contains non-negative magnitude or 'weight' s_i. The sum of all weights is given by $S = \sum_{i=1}^{n} Si$. Our problem is to check if there exists a subset $H \in A$ with $|H| = m$ such that $H = \sum_{i=1}^{m} hi = \frac{S}{2}$, where h_i is the weight of the i^{th} object in H. Total sum of all the weights, S must be even so that the dynamic 2-D table could be made.

3. Literature Search

3.1. Research Paper 1:

Title of the paper: *Parallel solution of the subset-sum problem: an empirical study* [3].

3.1.1. Problem Statement:

The paper solves subset sum problem parallelizing it on three different architectures Cray XMT, IBM x3755, NVIDIA FX 5800. It uses two different approaches/algorithms to parallelize these machines. Finally it will show performance comparison of these three machines and prove the hypothesis that performance varies based on different problem size.

3.1.2. Novel Contribution:

Cray XMT (Extremely multithreaded) machine has 128 processor which is suitable for very large sized problems (1012 bits or more). This machine has powerful word-level locking, so when one thread is accessing current word (64-bit word on 64-bit OS), no other thread can access it. The parallel version of subset sum on this machine is implemented using 'Word-oriented approach'. In this approach sum would be stored in an array of words. This approach is similar to naïve dynamic algorithm and parallelize it using world level locking.

Other two machines, IBM x3755 (16-processor shared memory machine) and NVIDIA FX 5800 (240-core graphics processor unit (GPU)) do not have word level locking. So other two algorithms are used named as 'Alternate word approach for conventional shared memory machines' [3] to generate dynamic 2-D table. Important analysis about these two machines is IBM x3755 is suitable for intermediate sized problems (up to 1011 bits) while NVIDIA FX 5800 is suitable for small problems (up to 1010 bits).

3.1.3. Useful for our Project:

The variant of subset sum problem explained in this paper is used as our computational problem. Other major take away is dynamic algorithm implemented for Cray XMT is also used to implement similar algorithm for our sequential and parallel solution.

3.2. Research Paper 2:

Title of the paper: *Efficient Parallelization of a Two-List Algorithm for the Subset-Sum Problem on a Hybrid CPU/GPU Cluster* [4].

3.2.1. Problem Statement:

The paper introduces a hybrid MPI-CUDA dual-level parallelism on the cluster. Two list algorithm for a subset sum problem is parallelized using MPI-CUDA implementation. The main concern with the dual-level parallelization is to allocate most suitable workload to each node. This paper proposes an effective workload distribution such that each node get reasonable amount of workload.

3.2.2. Novel Contribution:

In the hybrid MPI-CUDA Programming model, MPI perform the communication between computing nodes and CUDA computes the tasks on the GPU.

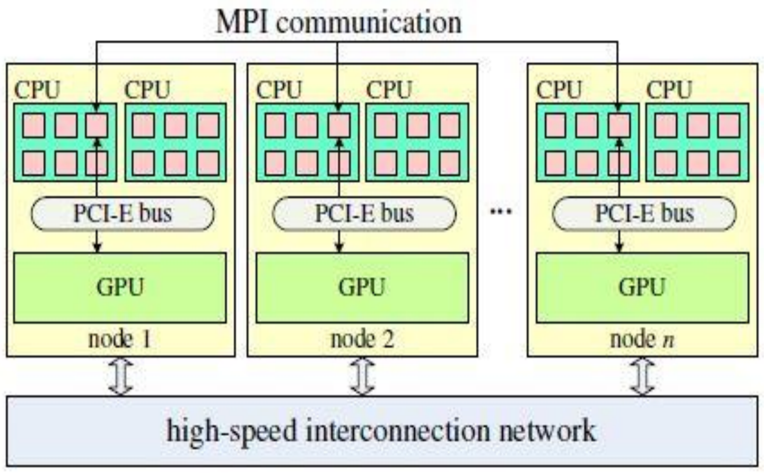


Figure 1: Hybrid MPI-CUDA Parallel Programming Model. [4]

Figure1 depicts the model proposed in the paper. Only one MPI process running on one CPU core for each node. It is used to communicate with GPU through PCI-E bus. It starts transferring data through PCI-E bus, later on invokes CUDA kernel with all GPU threads in parallel. At the end it transfers the output data from GPU to CPU. [4] This model provides dual-level parallelism: coarse-grained data parallelism and task parallelism (MPI), and fine-grained data parallelism and thread parallelism (CUDA).

Parallel two-list high-level algorithm can be described in following three steps.

- The parallel generation stage: Generation of two lists on the GPU
- The parallel pruning stage: Pruning is similar to reduction which is also performed on GPU.
- The parallel search stage: Search is used to find out subset sum problem solution from the two lists.

The proposed model applied with parallel two list algorithm to find out subset sum solution. Later, this model is compared 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 [4]. The result shows that using hybrid MPI-CUDA implementation, speedup was approximate 30 times than the best sequential algorithm for the cluster with 32 nodes.

3.2.3. Useful for our Project:

Load balancing scheme proposed in this paper is important to achieve higher speedup on cluster parallel programming. We looked at this paper to learn about balancing load and reducing communication overhead between CPU and GPU. Moreover, parallel two list algorithm is also a good approach for the solution of subset sum problem on cluster parallel programming.

3.3. Research Paper 3:

Title of the paper: *Parallel Implementation of the Modified Subset Sum Problem in OpenCL* [5].

3.3.1. Problem Statement:

The problem here is modified subset sum problem. The paper explains parallelization of modified subset sum problem with OpenCL. This variant dynamically allocates memory and require less memory to solve the problem.

3.3.2. Novel Contribution:

The modified subset sum problem is a solution to find all vectors with *N* elements where each element from *N* is less than *K* and all elements sums up to the *SUM*. *N, K* and *Sum* are the inputs. For example, consider an input set $A = \{a_1, a_2, \dots, a_n\}$, where $|A| = n$. Here the modified variant tries to find all vector elements a_i, where $i=1 \dots N$ and $a_i \in [0, K]$ and $\sum_{i=1}^{n} ai = Sum$.

The main idea of the proposed approach is to minimize the number of permutations in each thread and also without allocating redundant memory using modified subset sum problem. From the test cases presented in the paper [5], a speedup of up to 10 is achieved and proved that it largely scales with the problem size, and significantly decreases the running time.

3.3.3. Useful for our Project:

We reviewed this paper to develop a memory efficient algorithm for our subset sum problem. Our word oriented approach dividing total sum by 64 is also similar to this memory efficient approach.

4. Sequential Program Design

The sequential algorithm to generate dynamic 2-D table is described as below.

```
1   Read input file of n elements and count sum of all elements as S.
2   Calculate target sum H = S/2, H must be even. Also count total words = (S+1)/64 to represent each
    possible sum.
3   Create 2-D table a[n+1][words+1].
Base case:
4   for i in 0 to n
5       a[i][0] <- 1
Fill the 2-D table:
6   for i in 1 to n
7       pos <- set[i]
8       for j in 0 to words-1
9           a[i][j] <- a[i-1][j]
10      for j <- 0 to H
11          thatPos <- pos + j
12          if (checkBit(a[i-1][j],1) ==1 && thatPos<=H)
13              setBit(a[i-1], thatPos)

14  Check element a[n][words] using checkBit method and if the bit is set to 1 then subset exists.
```

Line number 1-3 in the above algorithm describes to read an input file of n elements, initialize Sum, H (target sum) and words variables. Here words are used to represent each possible sum into binary from for particular row. For example, when total sum S = 26 then target sum H=13, total 64-bit words=1. Suppose row i=4, where weight of the element (4th element in the input file) is 5, then columns j=0,1,2,3,4,5 would be 1. So, a[4][1] = 32 ('11111' binary representation of all columns, 4th row, 1st word).

Line number 4-5 indicate a base case to initialize all rows with first column as 1. This indicates the case when target sum is 0. Line 6-13 represents the logic to fill the 2-D dynamic table. Variable '*pos*' at line 7 indicates 7th element from an input file. Line 9 inside the first inner *for* loop copies the word from previous row to current row. Second inner *for* loop, starting from line 10, set the columns to '1' because of the element a_i from the set. For example, if cell a[i – 1][j] is 1, then the element a[i][j - pos] in the subsequent row will also be 1.

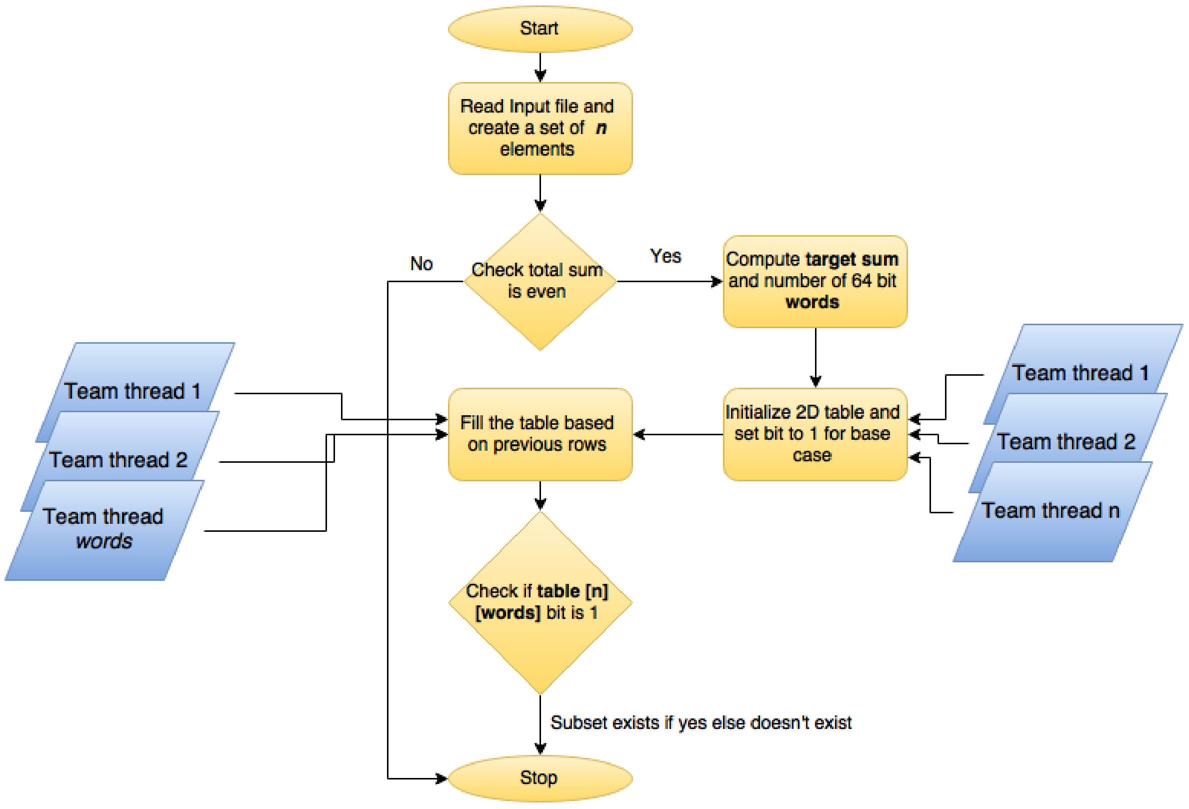
SubsetSumSeq.java file is the sequential version of subset sum problem implemented using parallel java 2 library [2]. It will read input file of n elements and checks if subset is exists or not by generating 2-D table. The program can also print the dynamic 2-D table. Nested *Table* class is created to store column values into 64-bit word format. *setBitInArray* method is used to set bit for particular columns into word format and *checkBitInArray* method is used to check if a bit is set for given index.

5. Parallel Program Design

The parallel program algorithm is similar to the sequential one. The only difference is we will parallelize base case *for* loop and two inner *for* loop using *parallelFor* which is a parallel version of for in parallel java 2 library [2]. Only modified portion of the algorithm is shown below in red text.

```
Base case:
4   parallelFor (0, n)
5       a[i][0] <- 1
Fill the 2-D table:
6   for i in 1 to n
7       pos <- set[i]
8       parallelFor (0, words)
9           a[i][j] <- a[i-1][j]
10      parallelFor (0, H)
11          thatPos <- pos + j
12          if (checkBit(a[i-1][j],1) ==1 && thatPos<=H)
13              setBit(a[i-1], thatPos)
```

Figure 2 displays design and operational flow of the parallel program. When program starts it reads the input files and check if total sum is even or not. If it is even then it start filling 2-D table same as sequential approach.



Start
Read Input file and create a set of n elements
Check total sum is even
No
Yes
Compute target sum and number of 64 bit words
Initialize 2D table and set bit to 1 for base case
Team thread 1
Team thread 2
Team thread n
Fill the table based on previous rows
Team thread 1
Team thread 2
Team thread words
Check if table [n] [words] bit is 1
Subset exists if yes else doesn't exist
Stop

Figure 2: Parallel program design flow.

SubsetSumSmp.java is a parallel version of subset sum problem implemented using parallel java 2 library [2]. As shown in the above algorithm, one base case *for* loop and two inner *for* loops are parallelized. For each core, a team thread will execute and start filling the 2-D table. For all cores, the same number of team threads execute in parallel as shown in Figure 2 and finish the filling of dynamic 2-

D table. For load balancing, after experimenting with different parallel for loop schedules, *guided* schedule is selected as it gives the smallest running time.

6. Developer's manual:

Both sequential and parallel programs have been tested on RIT CS multicore parallel computer *nessie*.

To compile the program first you need to set your Java class path to include PJ2 distribution. To set the java class path to current directory plus the PJ2 jar file follow the commands:

- For the *bash* shell:

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

- For the *csh* shell:

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

Now compile the program on nessie machine as below.

javac SubsetSumSeq.java – To compile sequential program
javac SubsetSumSmp.java – To compile parallel program

7. User's manual:

To run the program on RIT CS multicore parallel computer *nessie* first you need to set your Java class path to include PJ2 distribution and compile the program same as above.

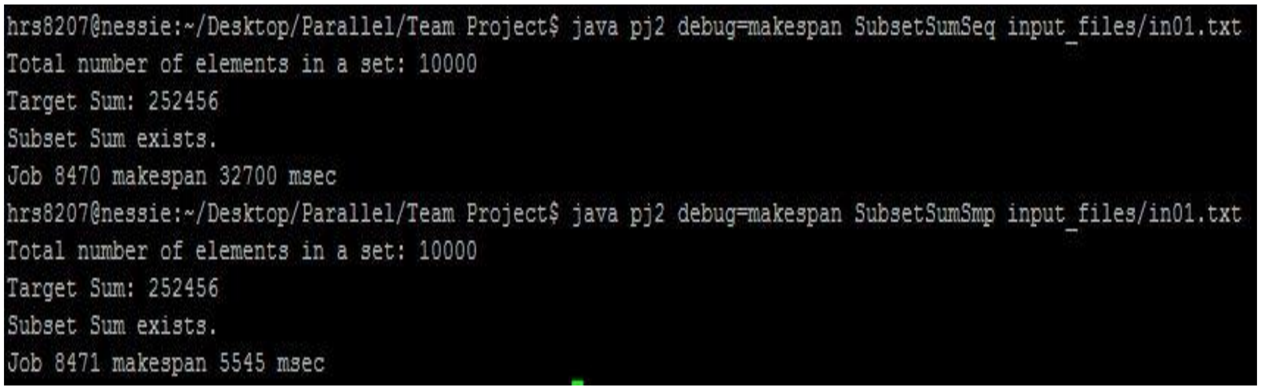
Now run the program as below.

java pj2 SubsetSumSeq <input_file> [p] – To compile sequential program
java pj2 cores=<K> SubsetSumSmp <input_file> [p] – To compile parallel program

Here, <input_file> - file containing n objects for a set.
<K> - Total number of cores.
[p] - optional parameter p to print the dynamic 2D table.

Example Screenshots:

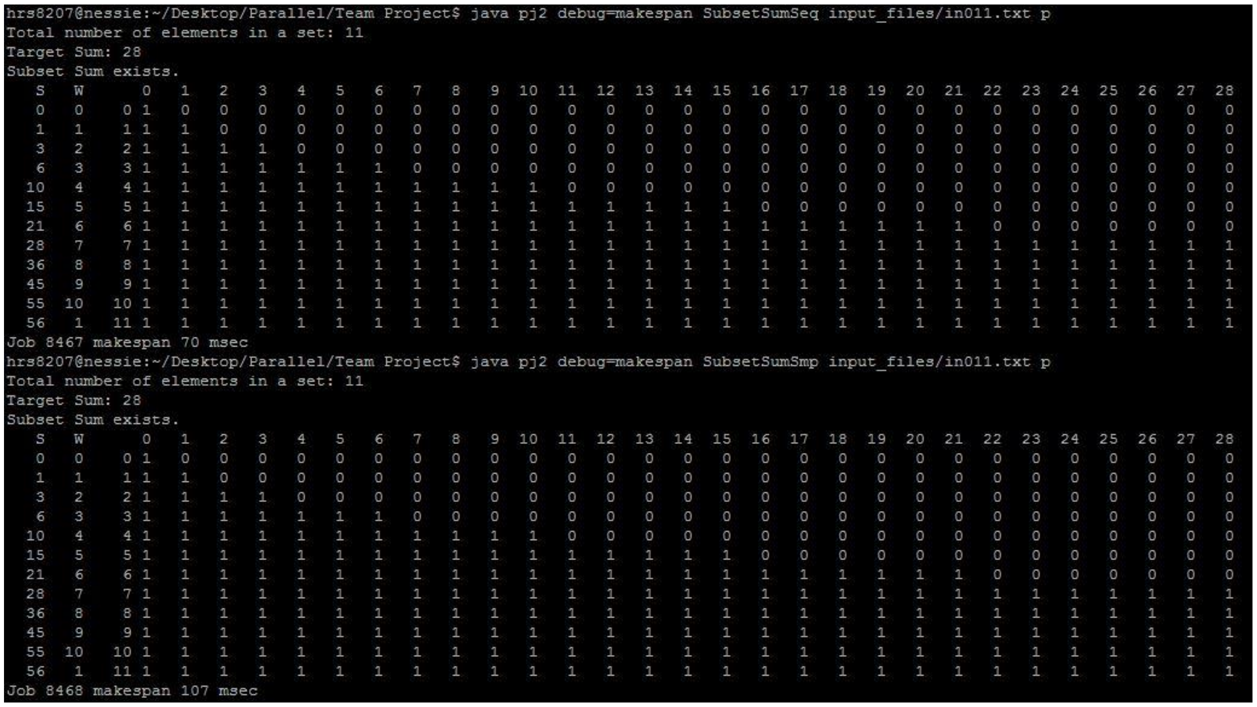
1) Running sequential and Parallel program



```
hrs8207@nessie:~/Desktop/Parallel/Team Project$ java pj2 debug=makespan SubsetSumSeq input_files/in01.txt
Total number of elements in a set: 10000
Target Sum: 252456
Subset Sum exists.
Job 8470 makespan 32700 msec
hrs8207@nessie:~/Desktop/Parallel/Team Project$ java pj2 debug=makespan SubsetSumSmp input_files/in01.txt
Total number of elements in a set: 10000
Target Sum: 252456
Subset Sum exists.
Job 8471 makespan 5545 msec
```

Figure 3: Running Sequential and Parallel program.

2) Running sequential and Parallel program with printing 2-D table option



```
hrs8207@nessie:~/Desktop/Parallel/Team Project$ java pj2 debug=makespan SubsetSumSeq input_files/in011.txt p
Total number of elements in a set: 11
Target Sum: 28
Subset Sum exists.
   S   W       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28
   0   0     0 1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   1   1     1 1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   3   2     2 1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   6   3     3 1   1   1   1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  10   4     4 1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  15   5     5 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0
  21   6     6 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   0   0
  28   7     7 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  36   8     8 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  45   9     9 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  55  10    10 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  56   1    11 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
Job 8467 makespan 70 msec
hrs8207@nessie:~/Desktop/Parallel/Team Project$ java pj2 debug=makespan SubsetSumSmp input_files/in011.txt p
Total number of elements in a set: 11
Target Sum: 28
Subset Sum exists.
   S   W       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28
   0   0     0 1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   1   1     1 1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   3   2     2 1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   6   3     3 1   1   1   1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  10   4     4 1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  15   5     5 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0
  21   6     6 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   0   0
  28   7     7 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  36   8     8 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  45   9     9 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  55  10    10 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  56   1    11 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
Job 8468 makespan 107 msec
```

Figure 4: Running Sequential and Parallel program with printing 2-D table option.

8. Strong Scaling Performance

Table 1 contains strong scaling performance data. '*n*' is total number of elements in the input file and N is a problem size for that input file. N is calculated by (*n**(ceil((*Sum*+1)/64))) where, Sum is total sum of the all input weights. From the table 1 we can see there are total 5 different problems with problem size (N) ranging from 7.89E+07 to 1.64E+09 for the input files consist of elements 10000 to 100000.

Table 1: Strong Scaling Performance

Label (n)	N	K	T (mS)	Speedup	Efficiency
10000	7.89E+07	seq	32654		
		1	23212	1.407	1.407
		2	14441	2.261	1.131
		3	8317	3.926	1.309
		4	7433	4.393	1.098
		5	6911	4.725	0.945
		6	5877	5.556	0.926
		7	5741	5.688	0.813
		8	5387	6.062	0.758
20000	1.59E+08	seq	66872		
		1	49151	1.361	1.361
		2	28423	2.353	1.176
		3	17548	3.811	1.27

		4	15744	4.247	1.062
		5	14547	4.597	0.919
		6	11741	5.696	0.949
		7	11870	5.634	0.805
		8	11129	6.009	0.751
30000	3.61E+08	seq	116190		
		1	97265	1.195	1.195
		2	53189	2.184	1.092
		3	37788	3.075	1.025
		4	31101	3.736	0.934
		5	30698	3.785	0.757
		6	26787	4.338	0.723
		7	25638	4.532	0.647
		8	22948	5.063	0.633
50000	5.08E+08	seq	146797		
		1	146890	0.999	0.999
		2	76151	1.928	0.964
		3	53252	2.757	0.919
		4	45153	3.251	0.813
		5	40771	3.601	0.72
		6	35524	4.132	0.689
		7	36462	4.026	0.575
		8	32171	4.563	0.57
100000	1.64E+09	seq	685964		
		1	457961	1.498	1.498
		2	244101	2.81	1.405
		3	177515	3.864	1.288
		4	150848	4.547	1.137
		5	148621	4.616	0.923
		6	121857	5.629	0.938
		7	109952	6.239	0.891
		8	106364	6.449	0.806

Using above performance data and *ScalePlot* class of PJ2 library [2] following plots are generated. Figure 5 shows running time vs. cores plot, Figure 6 shows speedup vs. cores plot and Figure 7 shows efficiency vs. cores plot.

Figure 6 and Figure 7 plots show that up to 4 cores, speedup and efficiency is almost ideal. Note that sequential version takes more time than parallel version running on a single core. So, until 3-4 cores, speedup and efficiency is achieved more than ideal case.

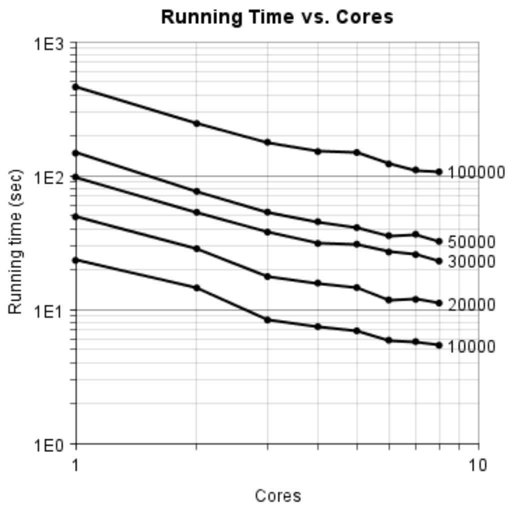


Figure 5: Running Time vs. Cores[2]

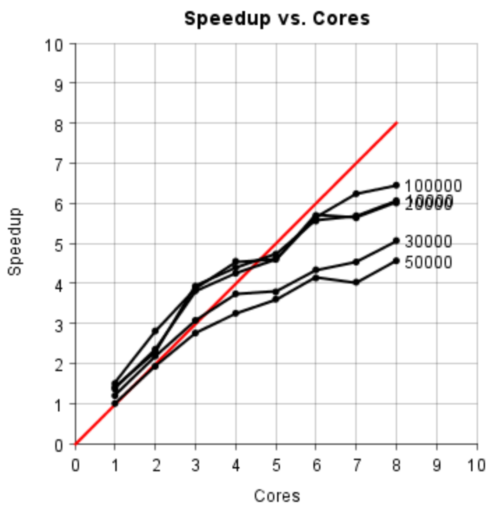


Figure 6: Speedup vs. Cores[2]

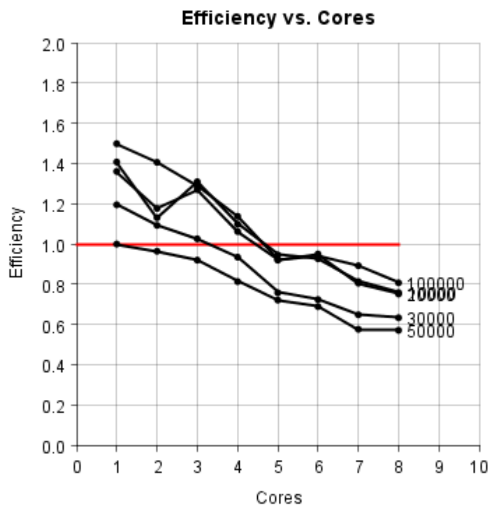


Figure 7: Efficiency vs. Cores[2]

Figure 7 also depicts that for larger problem size efficiency achieved is more with increasing the number of cores. In our case, for n=100000 the problem size is 1.64E+09 and efficiency is 0.806. In the paper [3] published by S. Bokhari proved that similar algorithm will work better for the problem size of 10^{12} bits or more. Thus, strong scaling performance is not ideal in our case is because of the sequential version take significantly larger time for the same problem size on parallel program with 1 core. Another reason could be the small problem size as our maximum problem size is 1.64E+09.

9. Weak Scaling Performance

Table 2 contains weak scaling performance data. '*n*' is total number of elements in the input file, H is the target sum and N is a problem size for that input file. N is calculated by (*n**(ceil((*Sum*+1)/64))) where, *Sum* is twice the target sum H. From the table 2 we can see there are total 5 different problems with problem size (N) ranging from 3.09E+07 to 4.08E+09 for the input files consist of elements 2000 to 50000. Using this data and ScalePlot class of PJ2 library [2] following plots are generated. Figure 8 shows running time vs. cores plot, Figure 9 shows sizeup vs. cores plot and Figure 10 shows efficiency vs. cores plot.

Table 2: Weak Scaling Performance

Label	n	H	N	K	T (mS)	Sizeup	Efficiency
2000	2000	494796	3.09E+07	seq	12719		
	2000	494796	3.09E+07	1	14594	0.872	0.872
	2700	732498	6.18E+07	2	12368	2.055	1.028
	3100	949863	9.20E+07	3	11842	3.196	1.065
	3500	1127567	1.23E+08	4	11466	4.424	1.106
	3600	1365816	1.54E+08	5	11314	5.585	1.117
	3700	1596649	1.85E+08	6	11975	6.34	1.057
	3800	1829415	2.17E+08	7	11975	7.461	1.066
	4000	1977588	2.47E+08	8	13812	7.361	0.92
5000	5000	315377	4.93E+07	seq	20366		
	5000	315377	4.93E+07	1	14002	1.455	1.455
	6500	487285	9.90E+07	2	14433	2.834	1.417
	7200	662619	1.49E+08	3	15573	3.957	1.319
	8000	780347	1.95E+08	4	14811	5.444	1.361
	8400	938518	2.46E+08	5	15699	6.485	1.297
	8800	1083967	2.98E+08	6	18105	6.804	1.134
	9200	1202380	3.46E+08	7	18481	7.73	1.104
	10000	1253381	3.92E+08	8	22255	7.274	0.909
10000	10000	252456	7.89E+07	seq	32654		
	10000	252456	7.89E+07	1	23212	1.407	1.407
	12000	423793	1.59E+08	2	23208	2.834	1.417
	13500	559982	2.36E+08	3	24297	4.024	1.341
	15000	664256	3.11E+08	4	25792	4.997	1.249
	16000	788289	3.94E+08	5	26500	6.156	1.231
	17200	881838	4.74E+08	6	28060	6.991	1.165
	18500	962972	5.57E+08	7	30821	7.476	1.068
	20000	1000421	6.25E+08	8	36141	7.16	0.895
20000	20000	180420	1.13E+08	seq	47556		
	20000	180420	1.13E+08	1	33811	1.407	1.407
	25000	286755	2.24E+08	2	33502	2.82	1.41
	28000	389878	3.41E+08	3	36393	3.953	1.318
	32000	450314	4.50E+08	4	38064	4.989	1.247
	34000	536466	5.70E+08	5	39607	6.069	1.214
	36000	594327	6.69E+08	6	41356	6.817	1.136
	38000	654771	7.78E+08	7	48837	6.714	0.959
	40000	711741	8.90E+08	8	53903	6.96	0.87
50000	50000	325320	5.08E+08	seq	146797		
	50000	325320	5.08E+08	1	146890	0.999	0.999

	58000	564679	1.02E+09	2	149651	1.975	0.988
	66000	739787	1.53E+09	3	171393	2.571	0.857
	75000	865348	2.03E+09	4	165332	3.543	0.886
	82000	983087	2.52E+09	5	171047	4.253	0.851
	88000	1098516	3.02E+09	6	187407	4.655	0.776
	94000	1196678	3.52E+09	7	190591	5.326	0.761
	100000	1305102	4.08E+09	8	200441	5.876	0.734

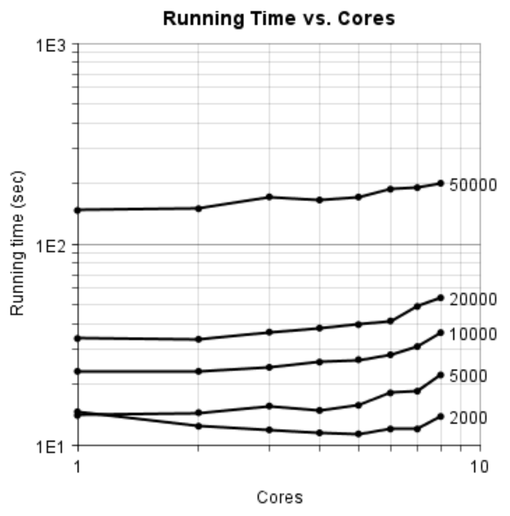


Figure 8: Running Time vs. Cores[2]

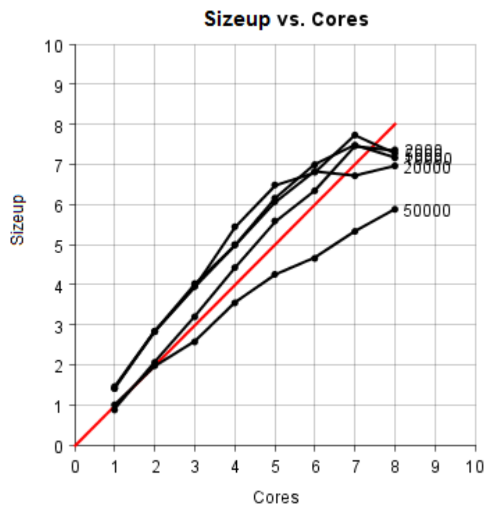


Figure 9: Sizeup vs. Cores[2]

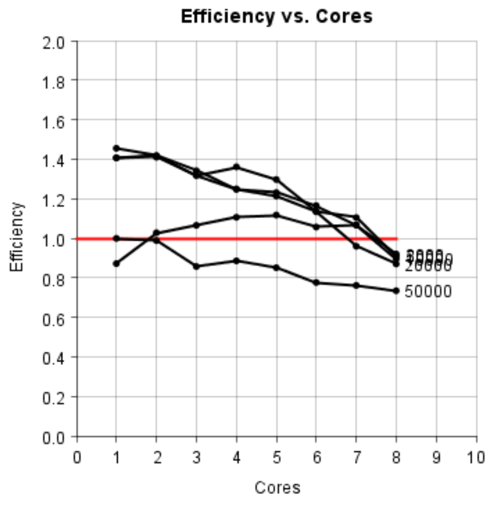


Figure 10: Efficiency vs. Cores[2]

Figure 9 and Figure 10 plots show that up to 7 cores, size up and efficiency is almost ideal and good as expected. One important observation is efficiency we are getting is more than 1 up to 7 cores for all problems except the last one. This is because of sequential program takes more time than the parallel program with same problem size for single core. For the last problem with n=5000, we are not achieving ideal sizeup and efficiency compared to other problems.

Figure 9, sizeup vs cores, shows that for the last problem with n = 50000 which has larger problem size doesn't give ideal scaling. Similarly earlier problems give more than 1 accuracy. The reasons behind this type of non-ideal weak scaling are as below.

- Sequential program takes more time than the parallel program with 1 core for the same problem size.
- Another reason is that program has sequential dependency. It has to wait until previous row is completely filled for the current row processing.
- The program depends on the input weights/values. For example, program with same problem size and different input weights takes different running time to process. The range of the inputs may affect significantly on the running time. I have generated all the files with different range of input values according to problem size.
- Use of arrays for different problem sizes results in more/less memory allocation-deallocation overhead.

10. Future work

Goal of our project is to find best parallel solution to subset sum problem. Generally dynamic algorithm runs faster than any naïve approach to solve subset sum problem. Thinking so we decided to solve subset sum problem by filling 2-D dynamic table. The parallel two list algorithm approach is widely used to solve subset sum problems. Even the parallel two list algorithm works best with GPU/CPU hybrid programming. Future work is to implement parallel two list algorithm to solve subset sum problem and measure the same strong scaling and weak scaling performance. Compare running time, speedup and efficiency with our approach.

The sequential and parallel program both use arrays to store total sum for particular weight. Instead implement a class like BitSet64 which can store maximum possible sums even though element is greater than 64. So we can work with very large problem sizes and also achieve similar speedup.

11. What we learned

First, we learned different implementation of dynamic algorithms to solve subset sum problem in parallel. Another important part is to handle sequential dependencies while programming for parallel computers. Second, widely used parallel two list algorithm to solve subset sum problem on GPU/CPU dual programming.

While programming for 64-bit word oriented approach extensively explored BitSet64 class from PJ2 library and BitSet class provided with java. I also learned how to use ScalePlot class of PJ2 library to represent running time, speedup and efficiency plots. At last I understood that the use of arrays to store 64-bit words may restrict you for larger problem size because of heap size limit. Therefore, it is better to avoid arrays.

12. Team Contribution

Picking up a research topic and literature search was the same amount of contribution from both of us, Ajith Paul and me. Preparing all presentation slides and later on, including sequential and parallel program design, their implementation, strong and weak scaling performance, different test cases, preparing input and output files, project report including all team deliverables was my contribution.

13. References

1. Garey MR, Johnson DS. *Computers and Intractability*. W. H. Freeman: New York, 1979.
2. A. Kaminsky. *Parallel Java 2 Library*. http://www.cs.rit.edu/~ark/pj2.shtml.
3. Bokhari, S. S. (2012), “Parallel solution of the subset-sum problem: an empirical study”. *Concurrency Computat.: Pract*. Exper., 24: 2241–2254.
4. Letian Kang; Lanjun Wan; Kenli Li, "Efficient Parallelization of a Two-List Algorithm for the Subset-Sum Problem on a Hybrid CPU/GPU Cluster," in *Parallel Architectures, Algorithms and Programming (PAAP), 2014 Sixth International Symposium on* , vol., no., pp.93-98, 13-15 July 2014.
5. Petkovski Dushan; Mishkovski Igor, “Parallel Implementation of the Modified Subset Sum Problem in OpenCL”, *ICT Innovations 2015, Web Proceedings ISSN*, pp.144-153, 2015.