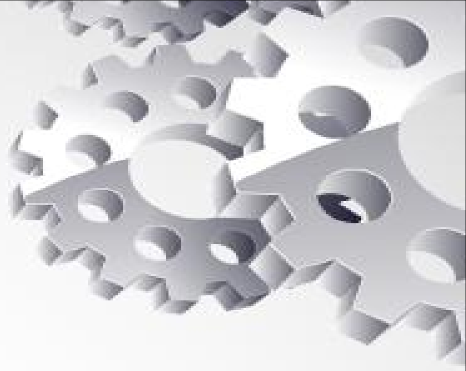
CSE 633: Parallel Algorithms

Subset Sum Count (0-1 Knapsack Variant)

Name: Kunal Chand
UB IT Name: kchand
Person Number: 50465175
Instructors: Professor Russ Miller & Dr. Matthew D. Jones

Contents

1) Problem Statement
2) Sequential Approach
3) Sequential Implementation
4) Parallel Approach
5) Parallel Implementation Issues
6) Parallel Implementation
7) Setup
8) Results
9) Future Work
10) References



1) Problem Statement

Determine the count of subsets within an array whose sum equals a given target sum.

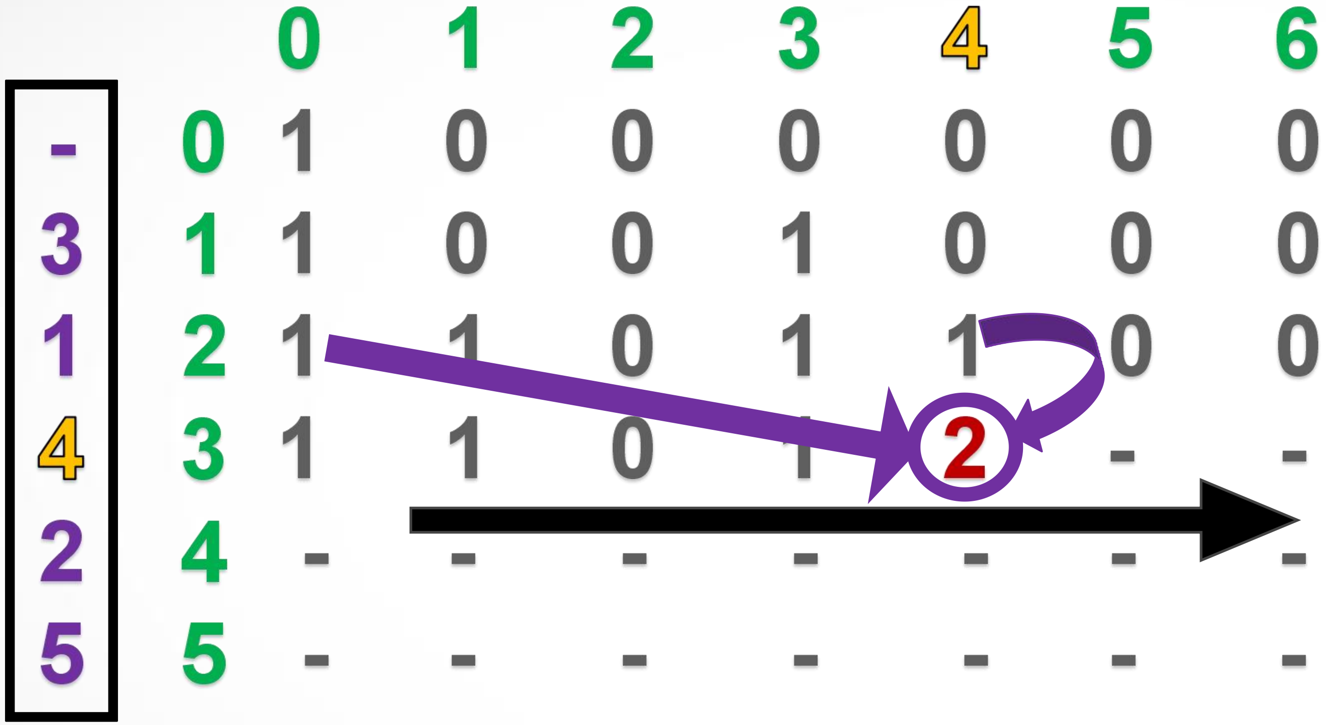
array[] = {3, 1, 4, 2, 5}   Targets Sum = 6

subsets = {3, 1, 2}, {4, 2}, {1,5}

Subset Sum Count = 3

Constraints: All elements of the array are whole numbers.

2) Sequential Approach



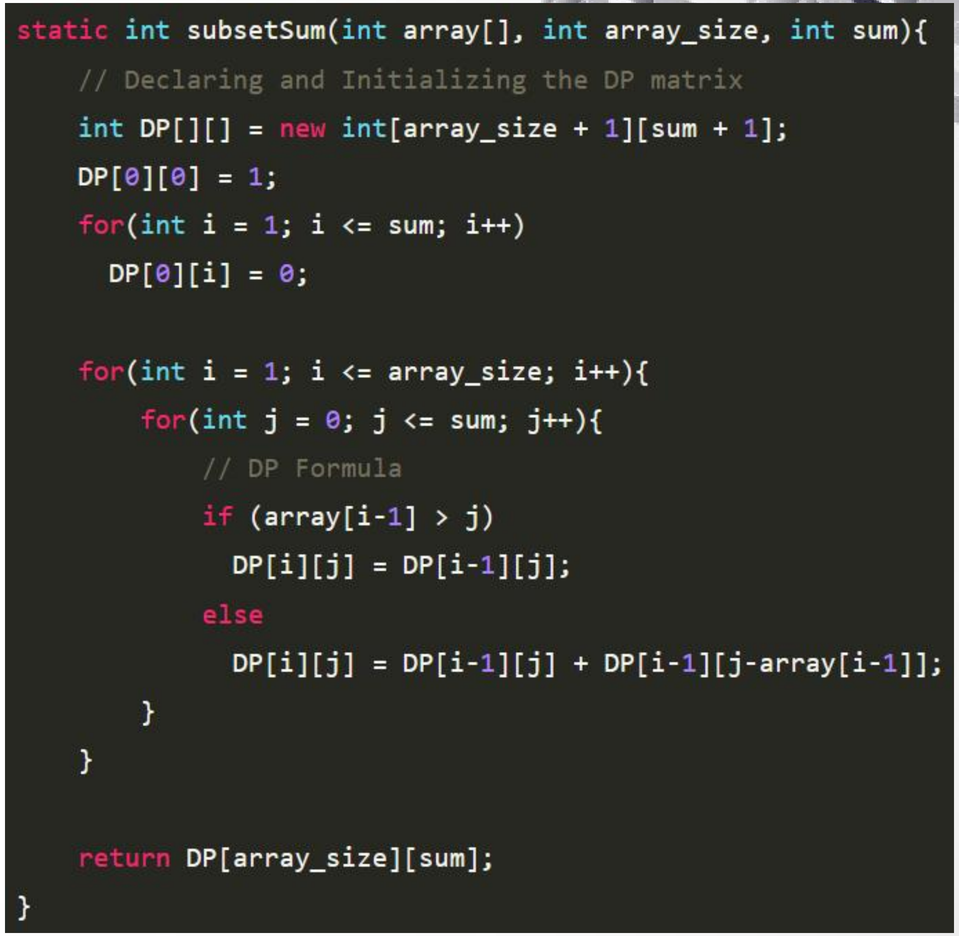
		0	1	2	3	4	5	6
-	0	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0
1	2	1	1	0	1	1	0	0
4	3	1	1	0	1	2	-	-
2	4	-	-	-	-	-	-	-
5	5	-	-	-	-	-	-	-

T(n) =
O(sum * array size)

if (array[i] > j): DP[i][j] = DP[i-1][j]
else: DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i]]

3) Sequential Implementation

		0	1	2	3	4	5	6
-	0	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0
1	2	1	1	0	1	1	0	0
4	3	1	1	0	1	2	1	0
2	4	1	1	1	2	2	2	2
5	5	1	1	1	2	2	3	3

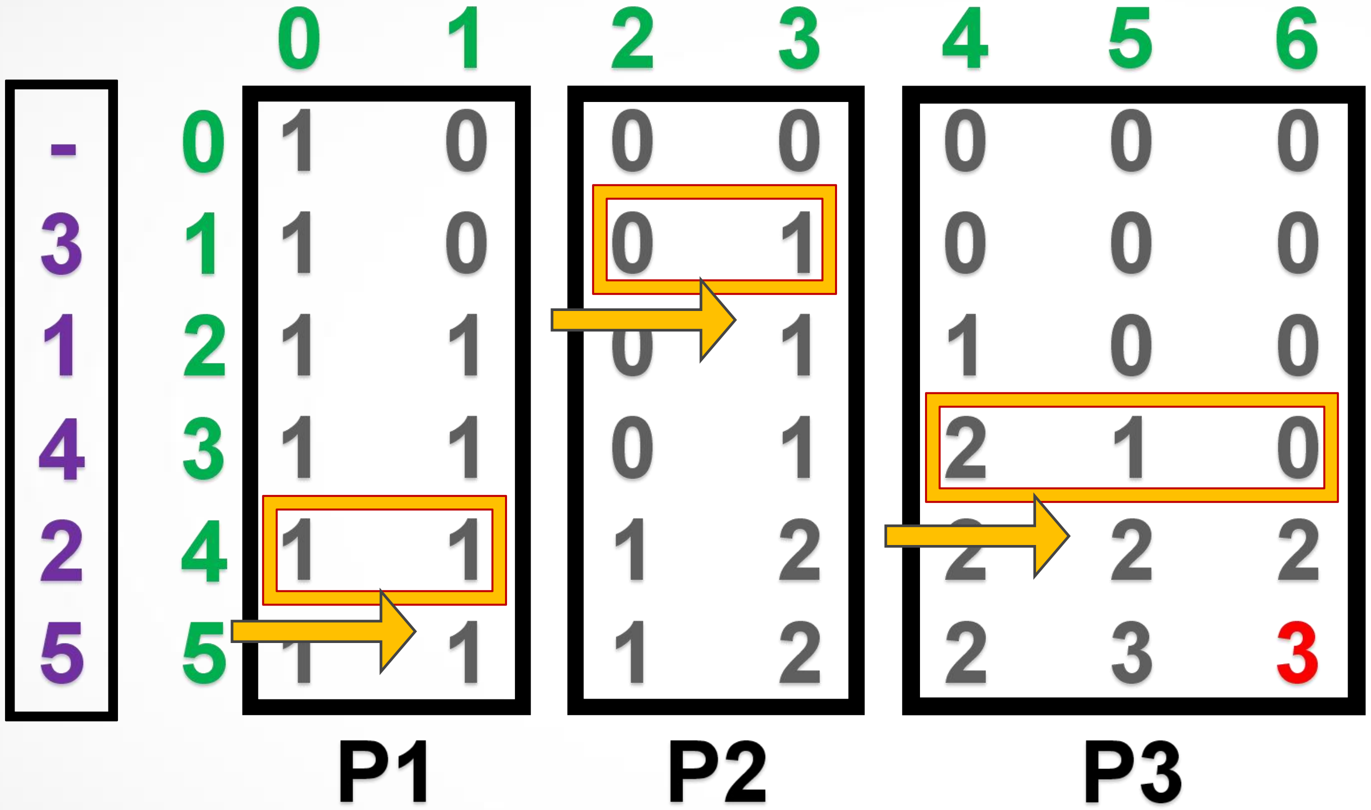


```
static int subsetSum(int array[], int array_size, int sum){
    // Declaring and Initializing the DP matrix
    int DP[][] = new int[array_size + 1][sum + 1];
    DP[0][0] = 1;
    for(int i = 1; i <= sum; i++)
      DP[0][i] = 0;

    for(int i = 1; i <= array_size; i++){
        for(int j = 0; j <= sum; j++){
            // DP Formula
            if (array[i-1] > j)
              DP[i][j] = DP[i-1][j];
            else
              DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i-1]];
        }
    }

    return DP[array_size][sum];
}
```

if (array[i] > j): DP[i][j] = DP[i-1][j]

else: DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i]]

4) Parallel Approach



		0	1	2	3	4	5	6
-	0	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0
1	2	1	1	0	1	1	0	0
4	3	1	1	0	1	2	1	0
2	4	1	1	1	2	2	2	2
5	5	1	1	1	2	2	3	3
		P1		P2		P3		

if (array[i] > j): DP[i][j] = DP[i-1][j]

else: DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i]]

5) Parallel Implementation Issues

> *MPI is not a remote procedure call, i.e., there are no ways to notify a process that some message has come in (e.g., by raising a signal).* ***Messages are sent but if the receiving process doesn't actually go look for them, then nothing will happen***
>
> *~Stack Exchange*

> ***You can't access data from another process****, you can't expect a signal to be raised if an incoming message arrives; everything only ever happens if you actively send a message or look whether one came in*
>
> *~Stack Exchange*

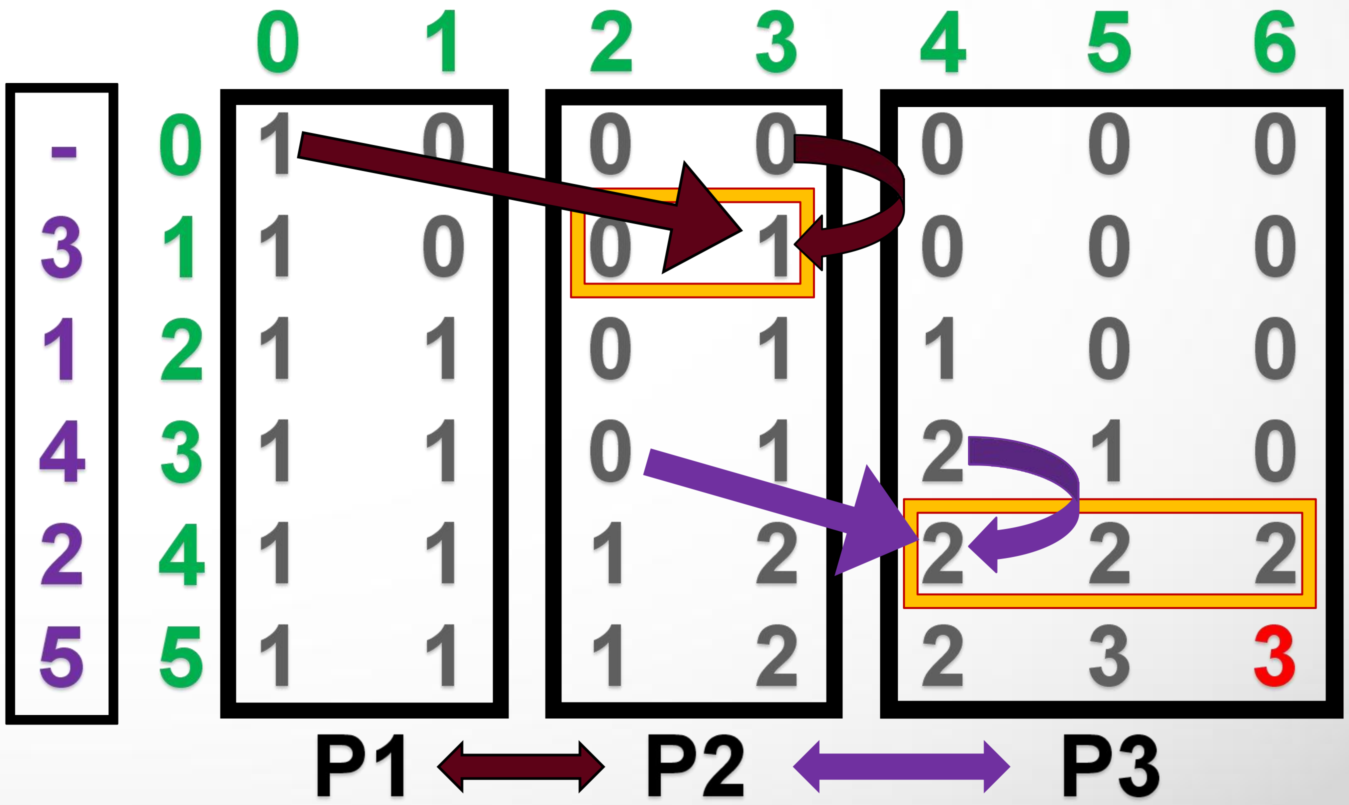
Issue 1: Infinite Wait Time

Receiver has to wait indefinetly for any future incoming messages. Possibility of a Deadlock. How long should a processor should wait?

Idea: We can develop a 3-way handshake protocol (similar to TCP/IP protocol) to establish a connection and communicate efficiently by sending requests.

Issue 2: Can't send requests

In MPI, we can only send data. There is no concept of sending requests like a web browser requesting for a webpage from server and getting an HTML page in response.

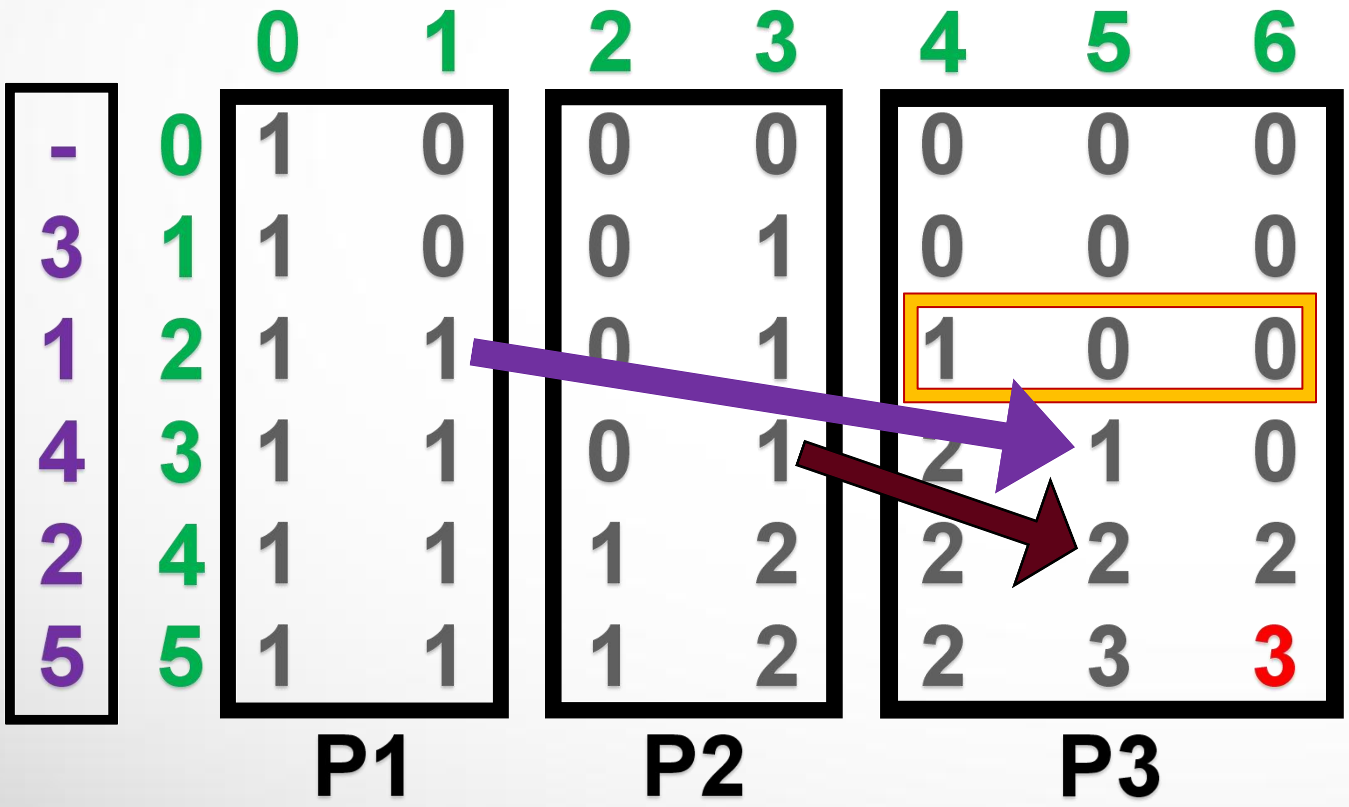


		0	1	2	3	4	5	6
-	0	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0
1	2	1	1	0	1	1	0	0
4	3	1	1	0	1	2	1	0
2	4	1	1	1	2	2	2	2
5	5	1	1	1	2	2	3	3

5) Parallel Implementation Issues

Issue 3: Store & process multiple requests

Every processor working independently will have to store the incoming information and process it be a request or a data. And then act on it accordingly. Storing such information will require a complex data structure and can also consume memory and time for just handing info related to communication.



		0	1	2	3	4	5	6
-	0	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0
1	2	1	1	0	1	1	0	0
4	3	1	1	0	1	2	1	0
2	4	1	1	1	2	2	2	2
5	5	1	1	1	2	2	3	3

Fixes

Issue 1: Infinite Wait Time

Issue 2: Can't send requests

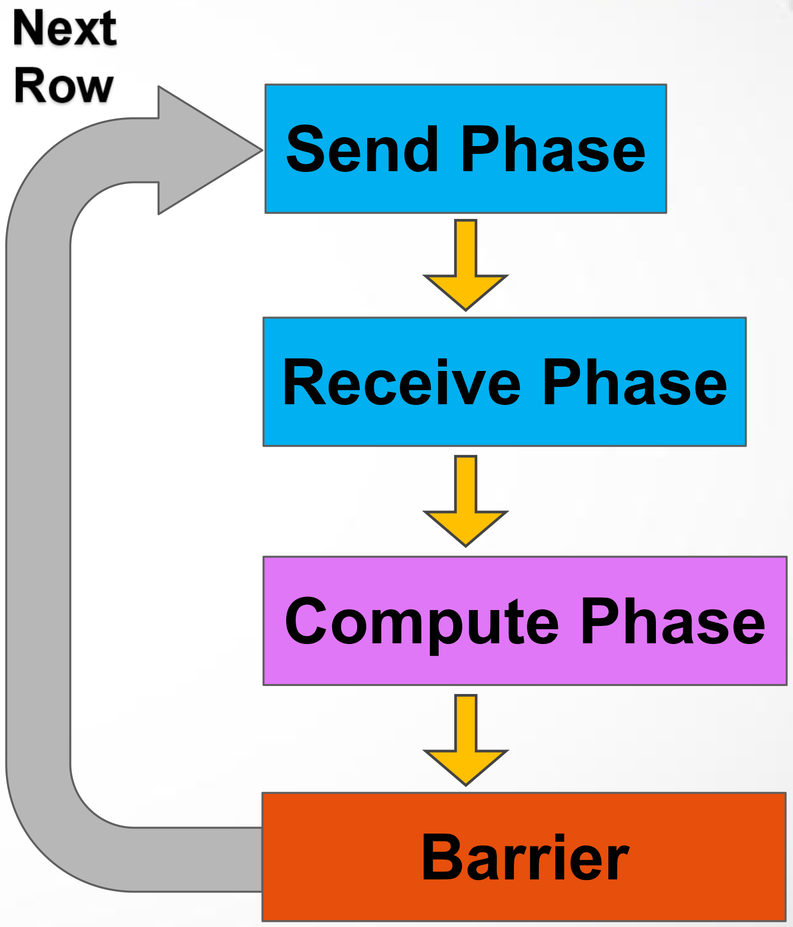
Fix 1&2: With use of Blocking mode of communication, exchancge of data in MPI is a possible if Source, Destination & the Message are known.

Issue 3: Store & process multiple requests

Fix 3: Syncronize processors at each layer of computation to reduce the quantity of information in the memory

6) Parallel Implementation

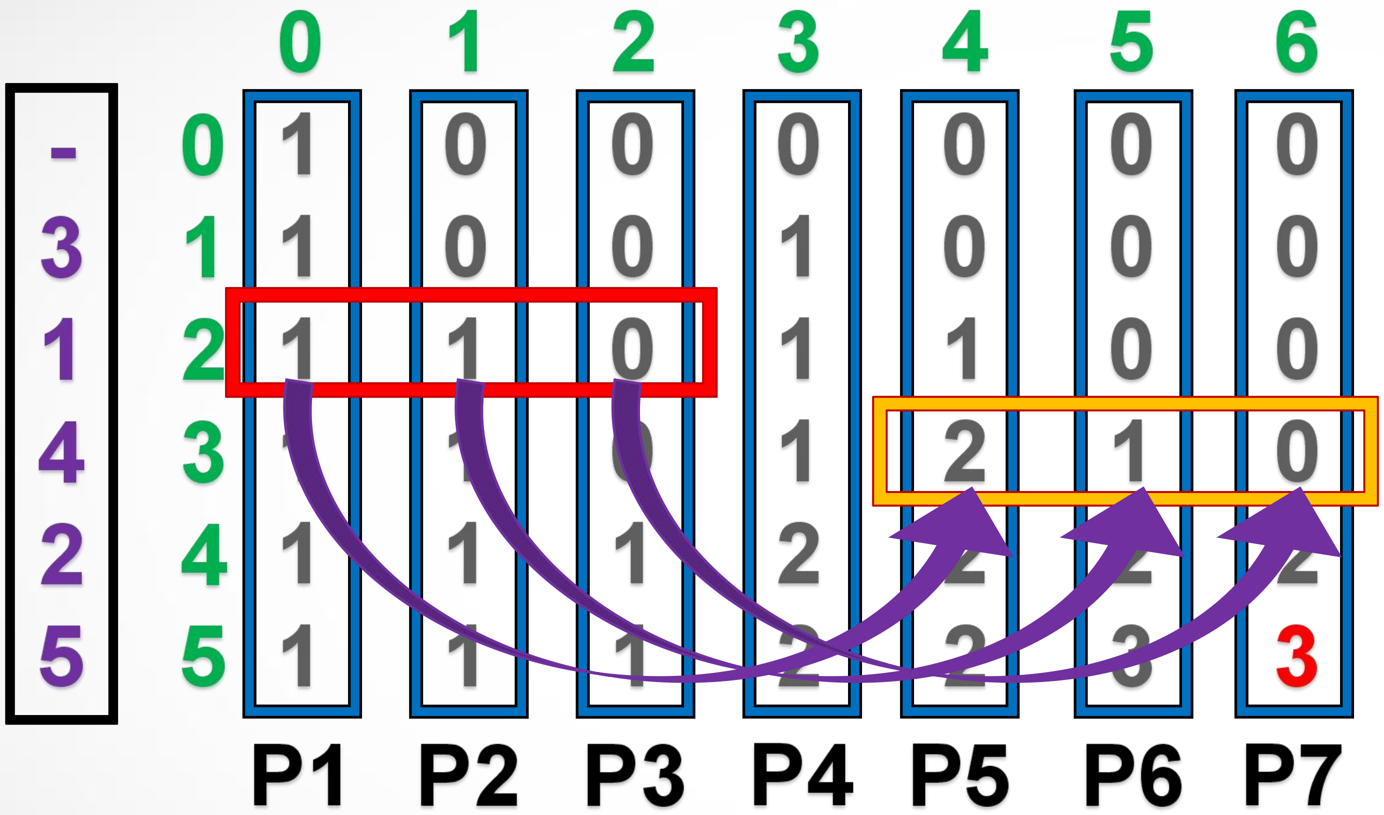
		0	1	2	3	4	5	6
-	0	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0
1	2	1	1	0	1	1	0	0
4	3	1	1	0	1	2	1	0
2	4	1	1	1	2	2	2	2
5	5	1	1	1	2	2	3	3
		P1	P2	P3	P4	P5	P6	P7



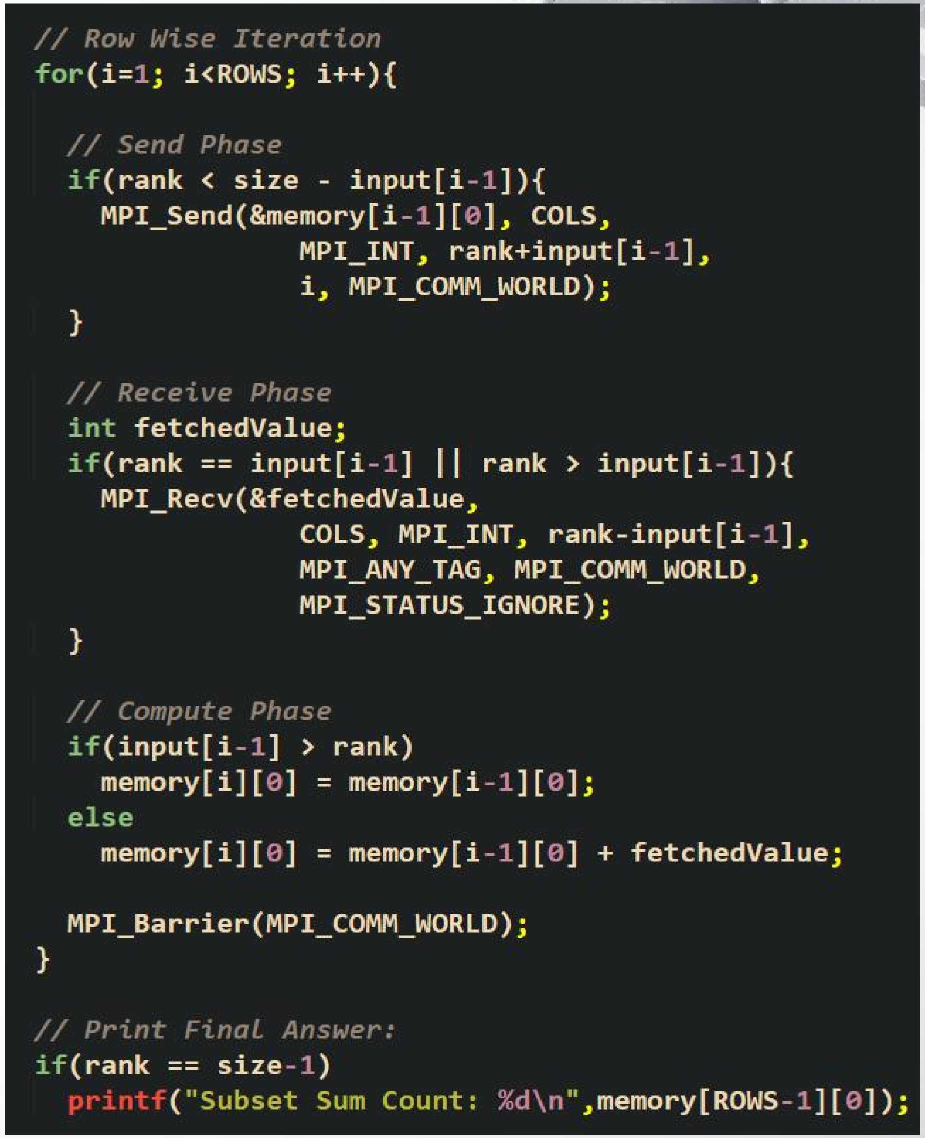
if (array[i] > j): DP[i][j] = DP[i-1][j]

else: DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i]]

6) Parallel Implementation



		0	1	2	3	4	5	6
-	0	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0
1	2	1	1	0	1	1	0	0
4	3	1	1	0	1	2	1	0
2	4	1	1	1	2	2	2	2
5	5	1	1	1	2	2	3	3
		P1	P2	P3	P4	P5	P6	P7

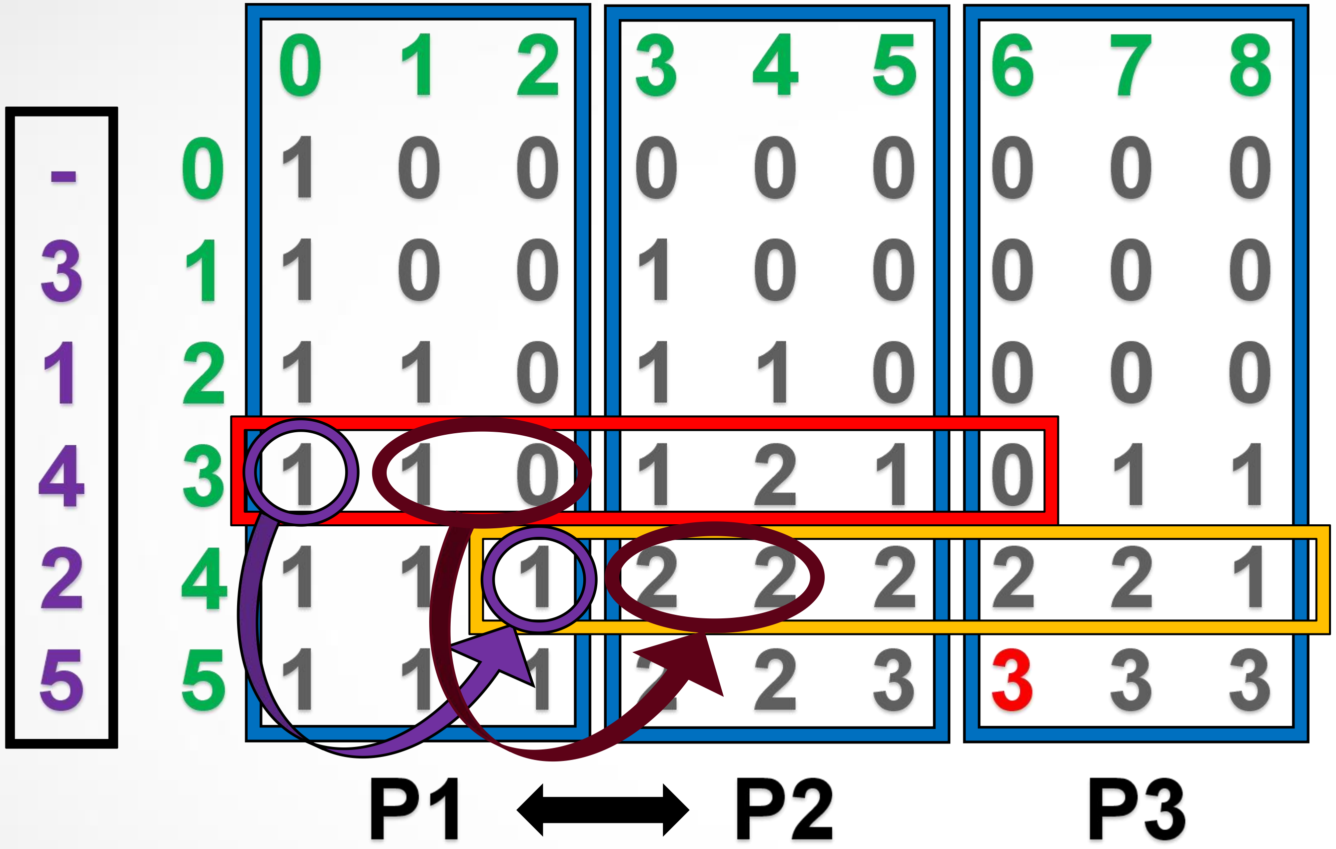


```
// Row Wise Iteration
for(i=1; i<ROWS; i++){

  // Send Phase
  if(rank < size - input[i-1]){
    MPI_Send(&memory[i-1][0], COLS,
                MPI_INT, rank+input[i-1],
                i, MPI_COMM_WORLD);
  }

  // Receive Phase
  int fetchedValue;
  if(rank == input[i-1] || rank > input[i-1]){
    MPI_Recv(&fetchedValue,
                COLS, MPI_INT, rank-input[i-1],
                MPI_ANY_TAG, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
  }

  // Compute Phase
  if(input[i-1] > rank)
    memory[i][0] = memory[i-1][0];
  else
    memory[i][0] = memory[i-1][0] + fetchedValue;

  MPI_Barrier(MPI_COMM_WORLD);
}

// Print Final Answer:
if(rank == size-1)
  printf("Subset Sum Count: %d\n",memory[ROWS-1][0]);
```

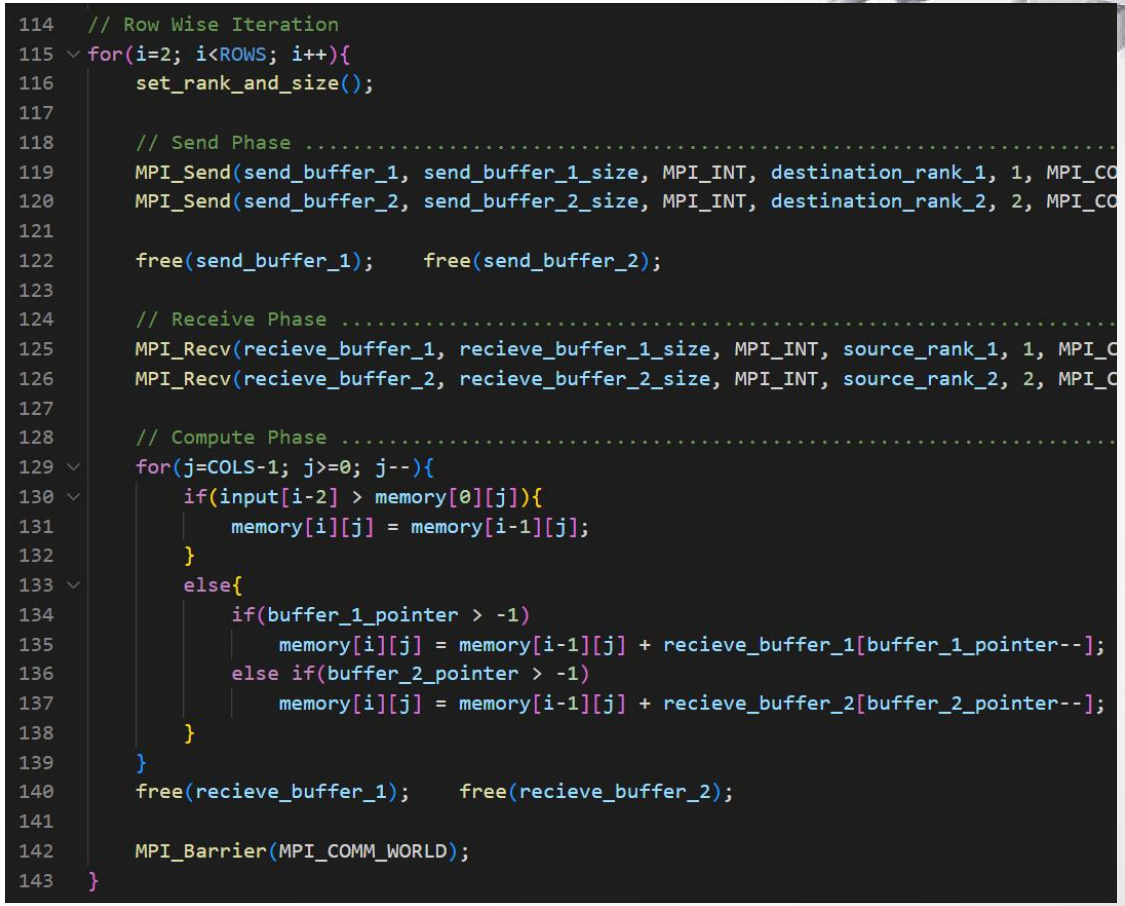
if (array[i] > j): DP[i][j] = DP[i-1][j]

else: DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i]]

6) Parallel Implementation



		0	1	2	3	4	5	6	7	8
-	0	1	0	0	0	0	0	0	0	0
3	1	1	0	0	1	0	0	0	0	0
1	2	1	1	0	1	1	0	0	0	0
4	3	1	1	0	1	2	1	0	1	1
2	4	1	1	1	2	2	2	2	2	1
5	5	1	1	1	2	2	3	3	3	3



```
// Row Wise Iteration
for(i=2; i<ROWS; i++){
    set_rank_and_size();

    // Send Phase ..........................................................................
    MPI_Send(send_buffer_1, send_buffer_1_size, MPI_INT, destination_rank_1, 1, MPI_CO
    MPI_Send(send_buffer_2, send_buffer_2_size, MPI_INT, destination_rank_2, 2, MPI_CO

    free(send_buffer_1);    free(send_buffer_2);

    // Receive Phase .......................................................................
    MPI_Recv(recieve_buffer_1, recieve_buffer_1_size, MPI_INT, source_rank_1, 1, MPI_C
    MPI_Recv(recieve_buffer_2, recieve_buffer_2_size, MPI_INT, source_rank_2, 2, MPI_C

    // Compute Phase .......................................................................
    for(j=COLS-1; j>=0; j--){
        if(input[i-2] > memory[0][j]){
            memory[i][j] = memory[i-1][j];
        }
        else{
            if(buffer_1_pointer > -1)
                memory[i][j] = memory[i-1][j] + recieve_buffer_1[buffer_1_pointer--];
            else if(buffer_2_pointer > -1)
                memory[i][j] = memory[i-1][j] + recieve_buffer_2[buffer_2_pointer--];
        }
    }
    free(recieve_buffer_1);    free(recieve_buffer_2);

    MPI_Barrier(MPI_COMM_WORLD);
}
```

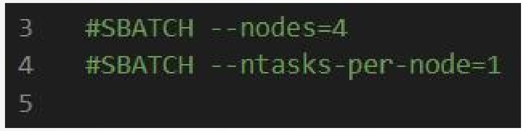
if (array[i] > j): DP[i][j] = DP[i-1][j]

else: DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i]]

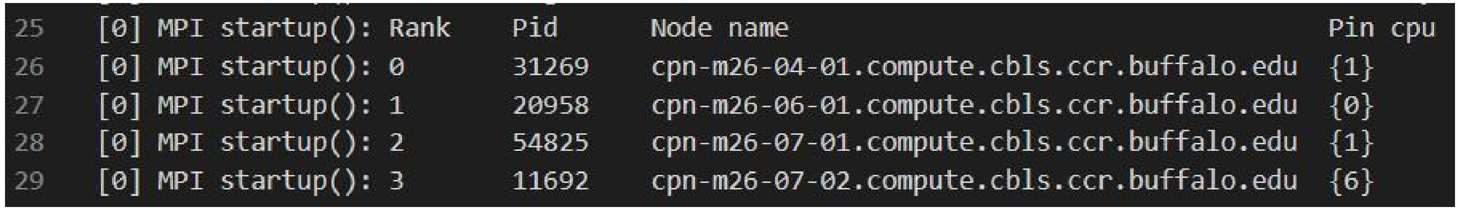
7) Setup

slurm script

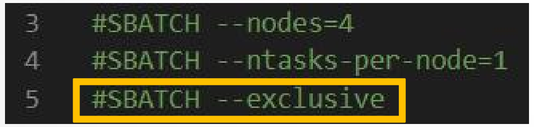
Debug Output Log



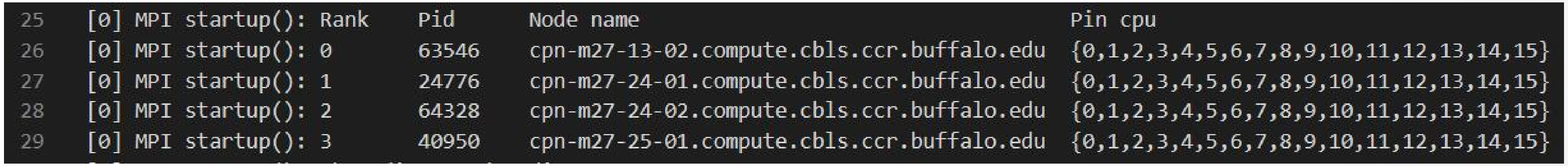
```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1

```



```
[0] MPI startup(): Rank    Pid      Node name                              Pin cpu
[0] MPI startup(): 0       31269    cpn-m26-04-01.compute.cbls.ccr.buffalo.edu  {1}
[0] MPI startup(): 1       20958    cpn-m26-06-01.compute.cbls.ccr.buffalo.edu  {0}
[0] MPI startup(): 2       54825    cpn-m26-07-01.compute.cbls.ccr.buffalo.edu  {1}
[0] MPI startup(): 3       11692    cpn-m26-07-02.compute.cbls.ccr.buffalo.edu  {6}
```



```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
#SBATCH --exclusive
```

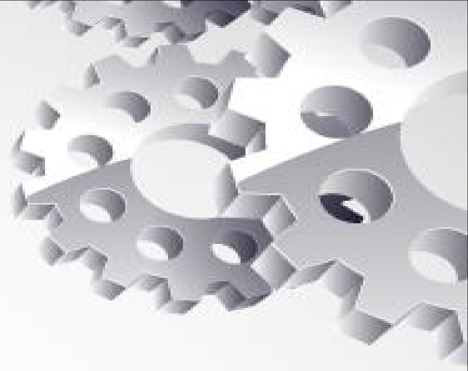


```
[0] MPI startup(): Rank    Pid      Node name                                   Pin cpu
[0] MPI startup(): 0       63546    cpn-m27-13-02.compute.cbls.ccr.buffalo.edu  {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
[0] MPI startup(): 1       24776    cpn-m27-24-01.compute.cbls.ccr.buffalo.edu  {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
[0] MPI startup(): 2       64328    cpn-m27-24-02.compute.cbls.ccr.buffalo.edu  {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
[0] MPI startup(): 3       40950    cpn-m27-25-01.compute.cbls.ccr.buffalo.edu  {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
```

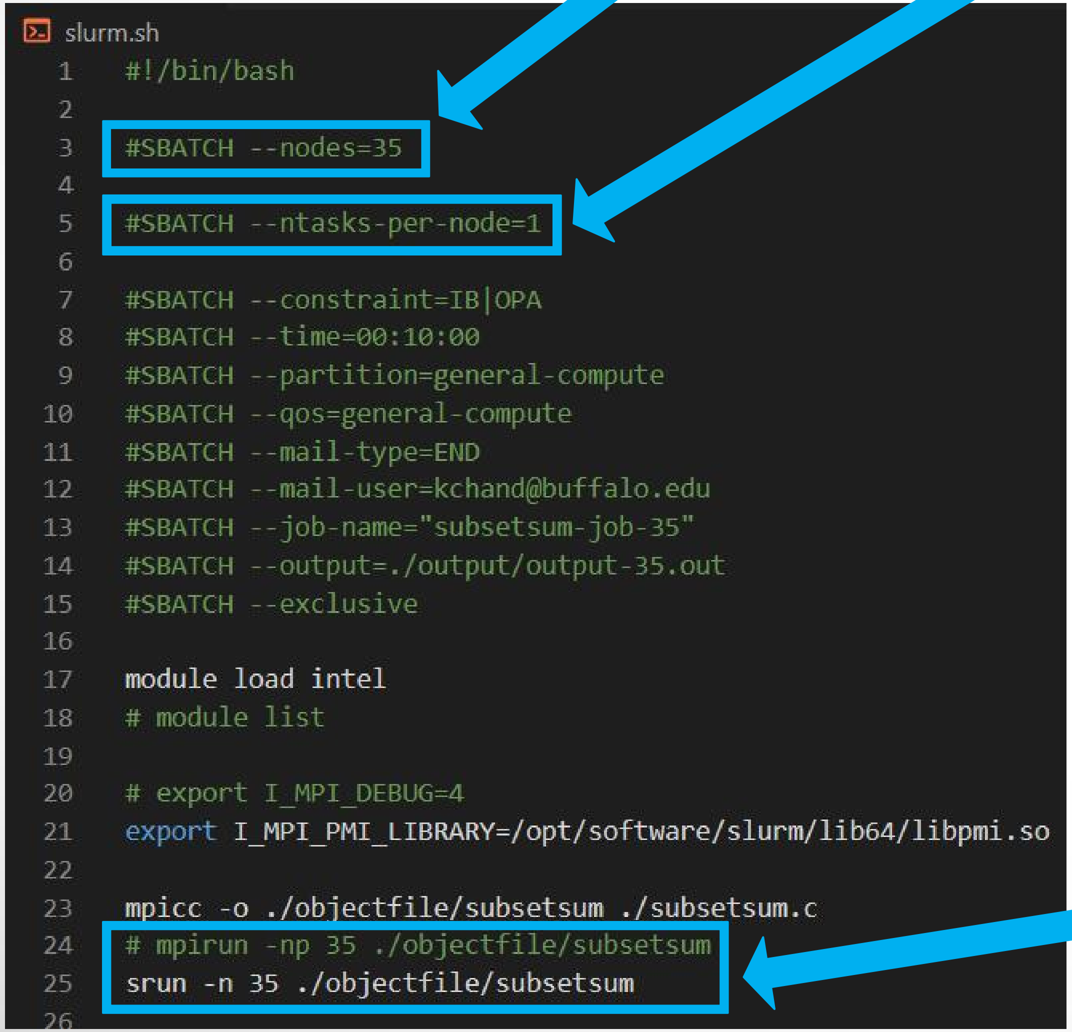
If you want to reserve all the cores on a node for your own exclusive use during your job, you can use the **--exclusive** option. It tells Slurm to reserve all the CPU cores on each node that you request, so that no other job can run on those cores while your job is running.

When the "**--exclusive**" flag is used, it indicates that the job requires exclusive access to the nodes, which means that no other jobs will be running on those nodes. This allows the MPI (Message Passing Interface) library to pin each MPI process to a specific set of CPUs on each node.

7) Setup



slurm script



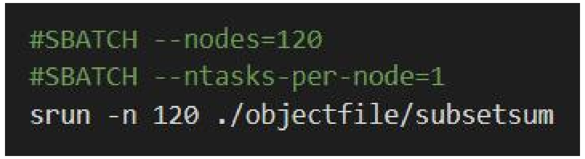
slurm.sh

```
#!/bin/bash

#SBATCH --nodes=35

#SBATCH --ntasks-per-node=1

#SBATCH --constraint=IB|OPA
#SBATCH --time=00:10:00
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute
#SBATCH --mail-type=END
#SBATCH --mail-user=kchand@buffalo.edu
#SBATCH --job-name="subsetsum-job-35"
#SBATCH --output=./output/output-35.out
#SBATCH --exclusive

module load intel
# module list

# export I_MPI_DEBUG=4
export I_MPI_PMI_LIBRARY=/opt/software/slurm/lib64/libpmi.so

mpicc -o ./objectfile/subsetsum ./subsetsum.c
# mpirun -np 35 ./objectfile/subsetsum
srun -n 35 ./objectfile/subsetsum
```

Total Nodes Reqested

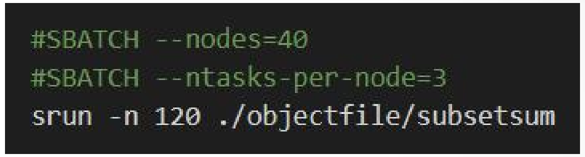
Number of Cores/Processors per Node

Number of Processors Requested = (Total Nodes Requested) * (Number of Cores/Processors per Node)



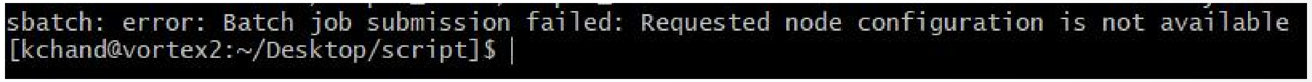
```
#SBATCH --nodes=120
#SBATCH --ntasks-per-node=1
srun -n 120 ./objectfile/subsetsum
```

1 Processor per Node

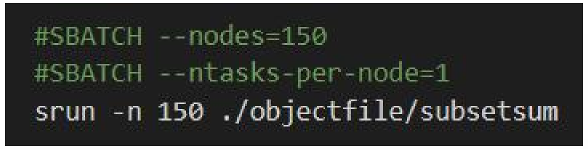


```
#SBATCH --nodes=40
#SBATCH --ntasks-per-node=3
srun -n 120 ./objectfile/subsetsum
```

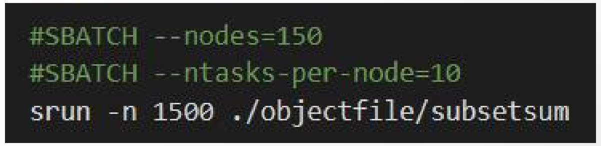
Multiple Processors per Node



```
sbatch: error: Batch job submission failed: Requested node configuration is not available
[kchand@vortex2:~/Desktop/script]$ |
```



```
#SBATCH --nodes=150
#SBATCH --ntasks-per-node=1
srun -n 150 ./objectfile/subsetsum
```

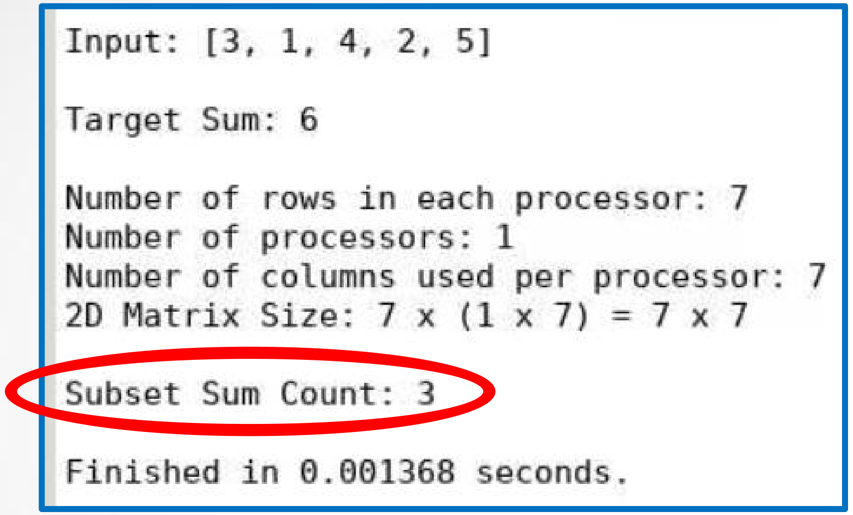


```
#SBATCH --nodes=150
#SBATCH --ntasks-per-node=10
srun -n 1500 ./objectfile/subsetsum
```

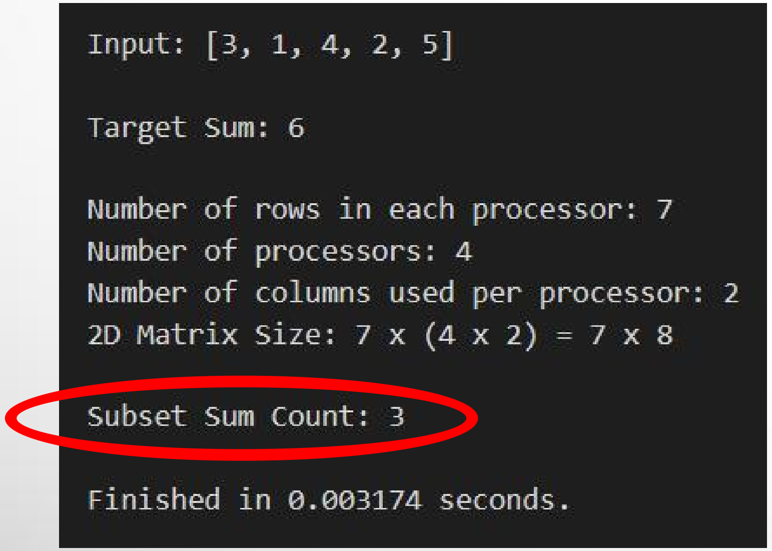
Number of Processors Requested

8) Results

Parallel Code Execution

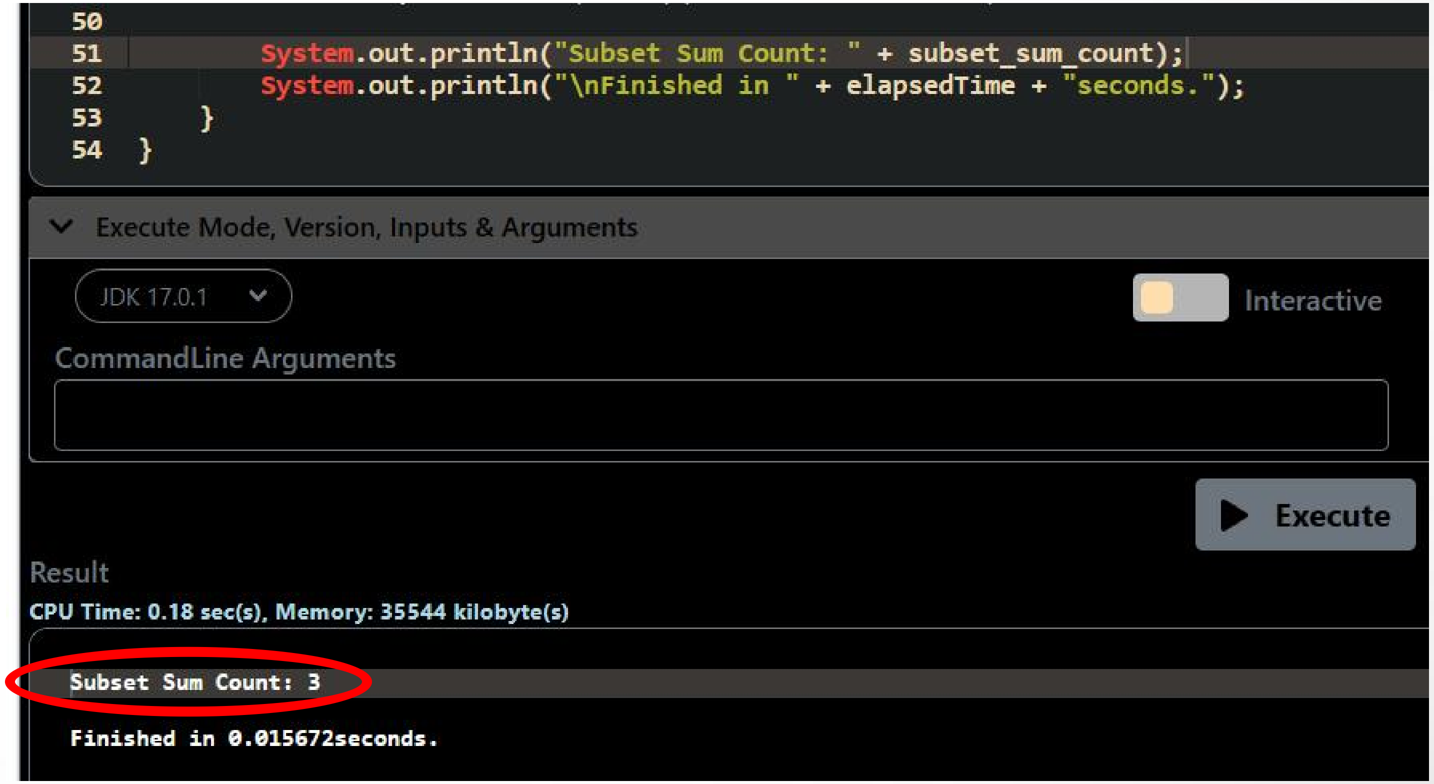


```
Input: [3, 1, 4, 2, 5]

Target Sum: 6

Number of rows in each processor: 7
Number of processors: 1
Number of columns used per processor: 7
2D Matrix Size: 7 x (1 x 7) = 7 x 7

Subset Sum Count: 3

Finished in 0.001368 seconds.
```



```
Input: [3, 1, 4, 2, 5]

Target Sum: 6

Number of rows in each processor: 7
Number of processors: 4
Number of columns used per processor: 2
2D Matrix Size: 7 x (4 x 2) = 7 x 8

Subset Sum Count: 3

Finished in 0.003174 seconds.
```

Sequential Code Execution



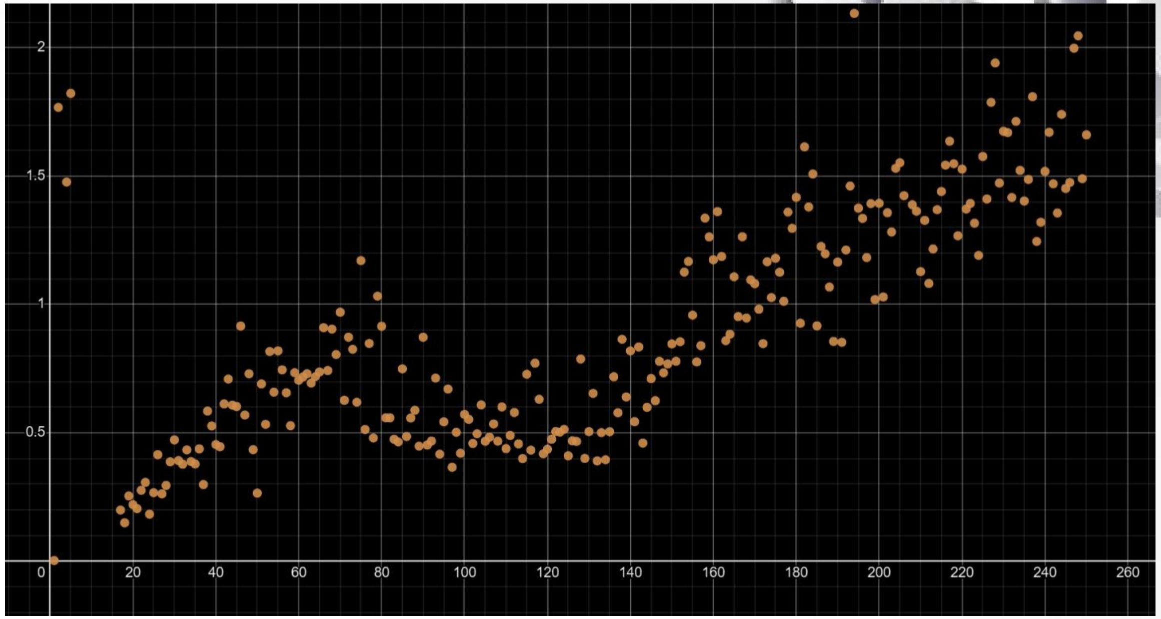
```

        System.out.println("Subset Sum Count: " + subset_sum_count);
        System.out.println("\nFinished in " + elapsedTime + "seconds.");
    }
}
```

```
CPU Time: 0.18 sec(s), Memory: 35544 kilobyte(s)

Subset Sum Count: 3

Finished in 0.015672seconds.
```

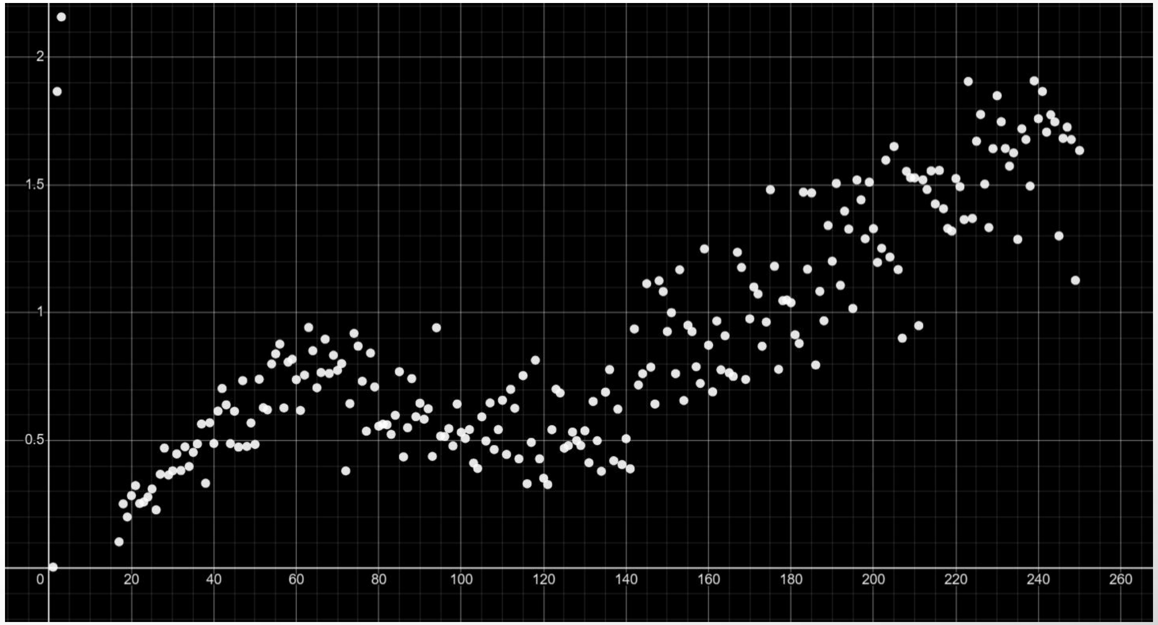
(Sequential Code Executable Link)

8) Results

(Mid Term Presentation Recap)



# of Nodes = 1 (relatively small)
Processors per Node = 1 to 250
Target Sum = 1000
Size of 2D Matrix = [64 x 1000]



# of Nodes = 1 (relatively small)
Processors per Node = 1 to 250
Target Sum = 2000
Size of 2D Matrix = [64 x 2000]

8) Results

A) Standard Execution (Amdahl's Law)

- Total size of input data remains same
- Increase the number of processors
- With more processors, each processor has lesser data

B) Scaled Execution (Gustafson's Law)

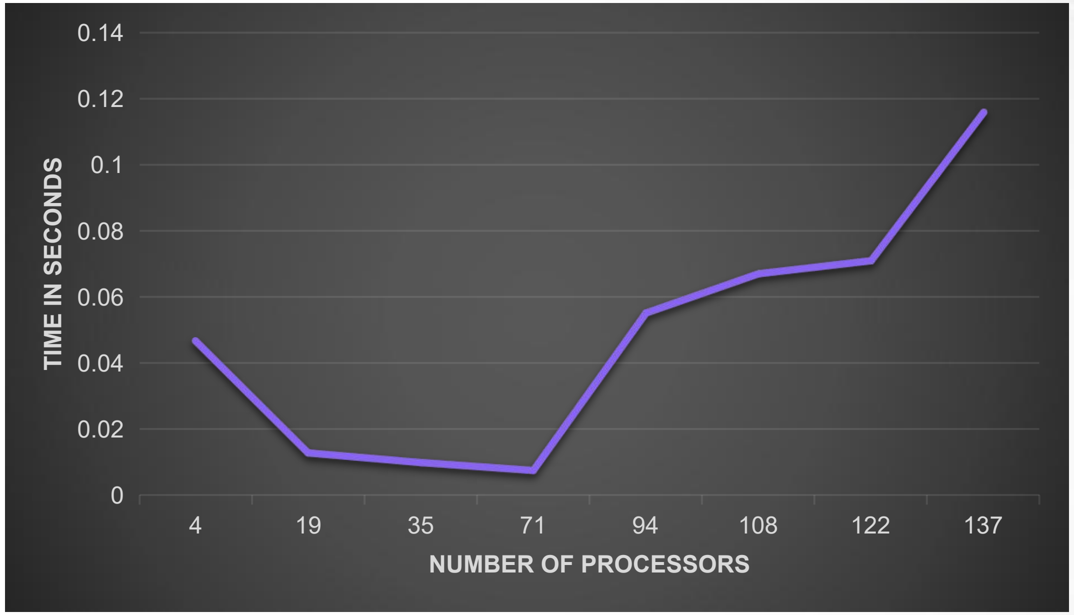
- Fix amount of data in each processor
- Increase the number of processors
- With more processors, the total size of the input data should also be increased. Because the data per processor remains constant as we increase the total input size.

C) Sequential Execution

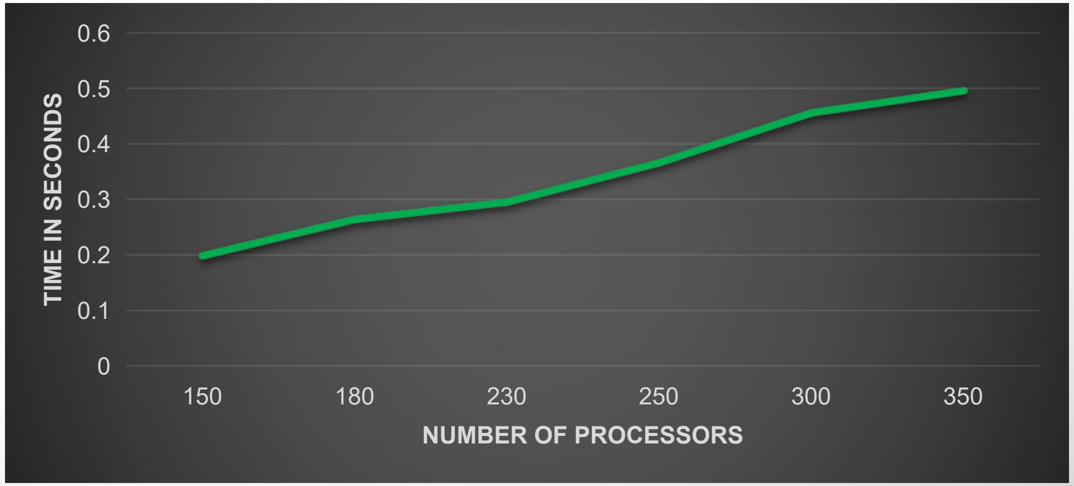
8) Results

A) Standard Execution (Amdahl's Law)

PEs	Data/PE (Col/PE)	Input Size (Total Col)	Nodes	Cores per Node	Time (in seconds)
4	501	2000	4	1	0.046854
19	106	2000	19	1	0.013044
35	58	2000	35	1	0.010136
71	29	2000	71	1	0.007746
94	22	2000	94	1	0.055246
108	19	2000	108	1	0.067106
122	17	2000	122	1	0.071037
137	15	2000	137	1	0.115846



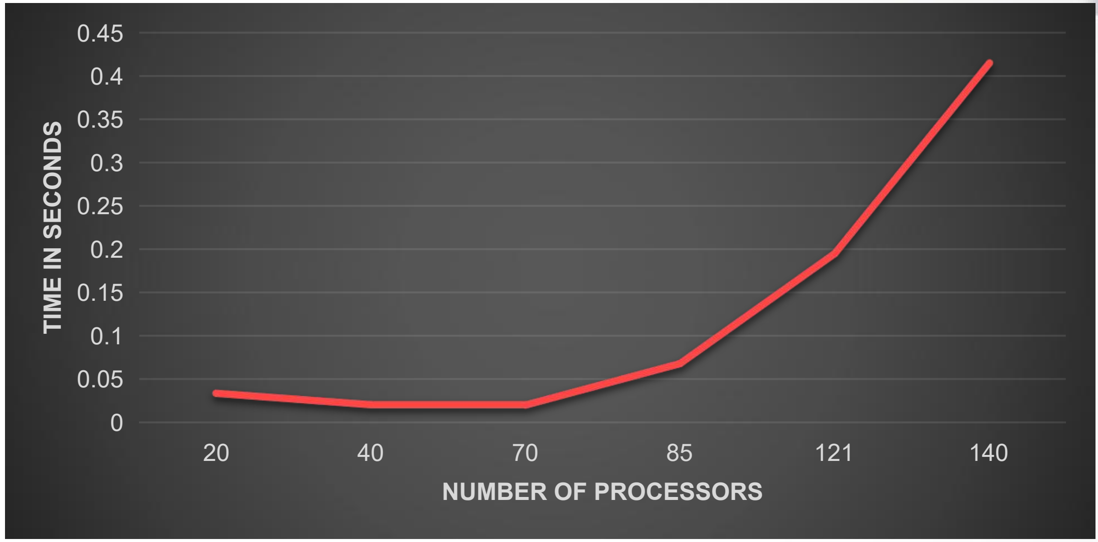
PEs	Data/PE (Col/PE)	Input Size (Total Col)	Nodes	Cores per Node	Time (in seconds)
150	14	2000	15	10	0.198445
180	12	2000	18	10	0.263768
230	9	2000	23	10	0.29483
250	9	2000	25	10	0.365401
300	7	2000	30	10	0.45483
350	6	2000	35	10	0.494543



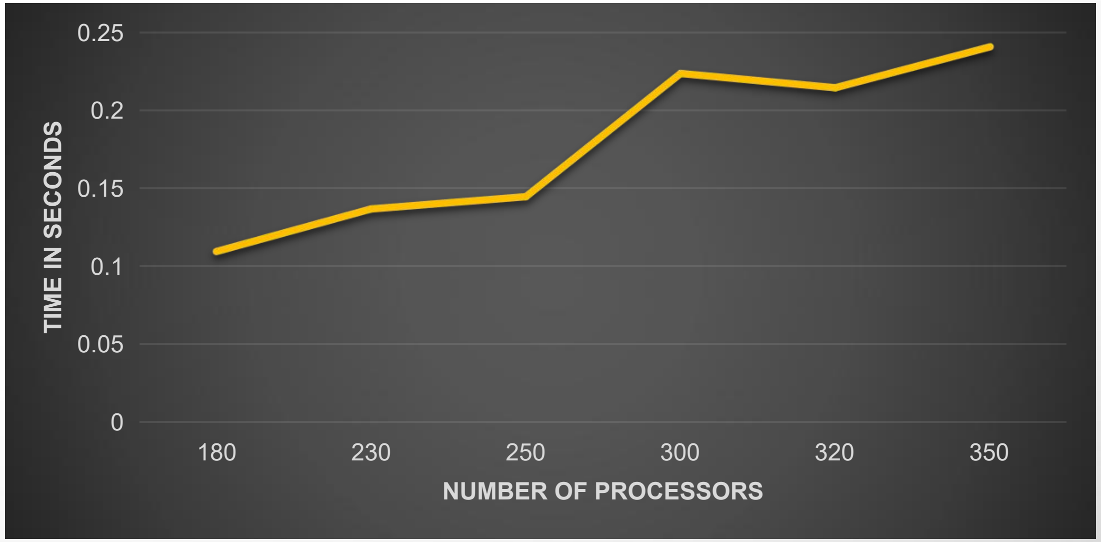
8) Results

B) Scaled Execution (Gustafson's Law)

PEs	Data/PE (Col/PE)	Input Size (Total Col)	Nodes	Cores per Node	Time (in seconds)
20	10	200	20	1	0.033918
40	10	400	40	1	0.020783
70	10	700	70	1	0.020312
85	10	850	85	1	0.068161
121	10	1210	121	1	0.194891
140	10	1410	140	1	0.41438



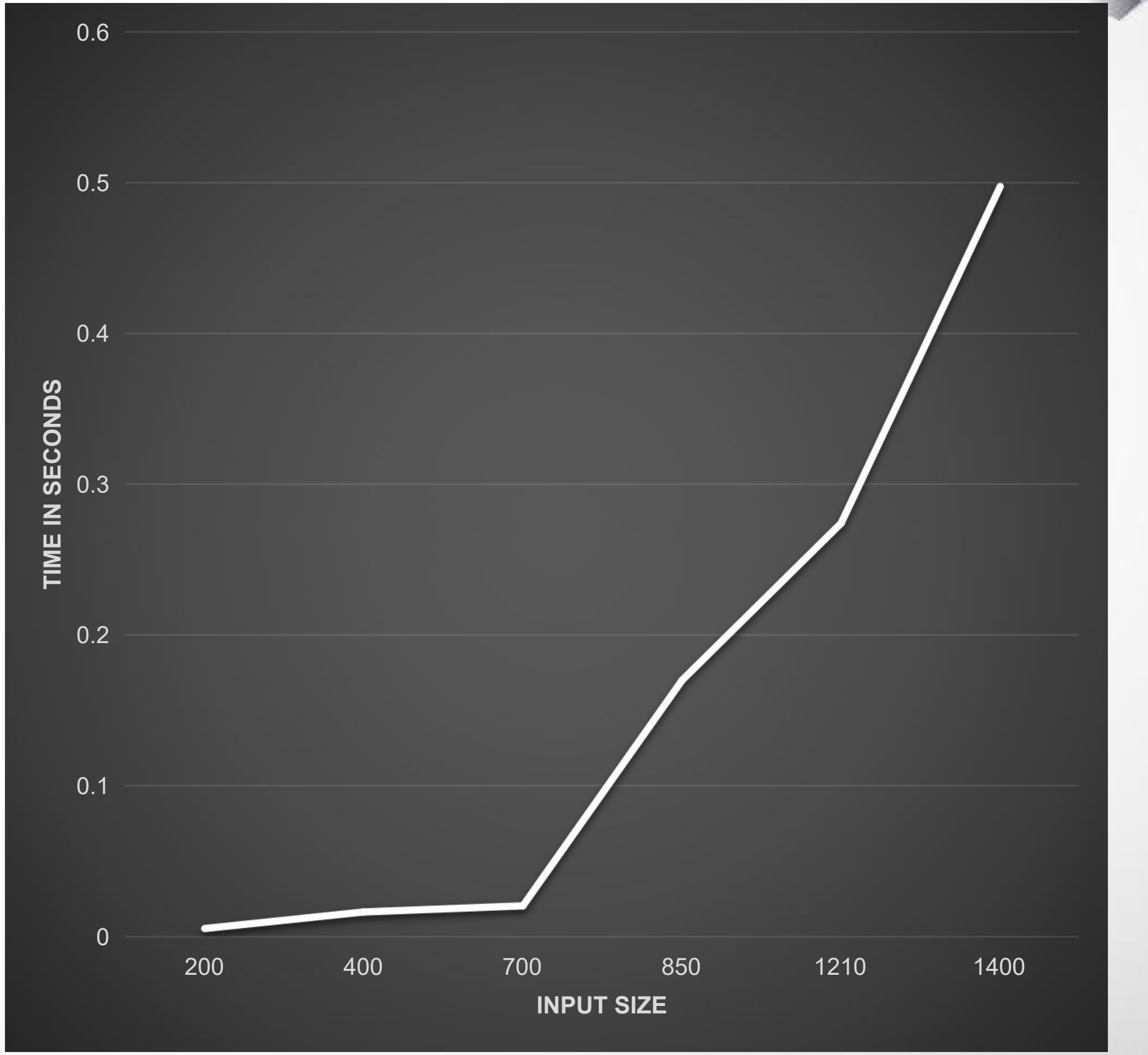
PEs	Data/PE (Col/PE)	Input Size (Total Col)	Nodes	Cores per Node	Time (in seconds)
180	10	1800	18	10	0.109324
230	10	2300	23	10	0.136648
250	10	2500	25	10	0.144504
300	10	3000	30	10	0.22354
320	10	3200	32	10	0.21438
350	10	3500	35	10	0.240661



8) Results

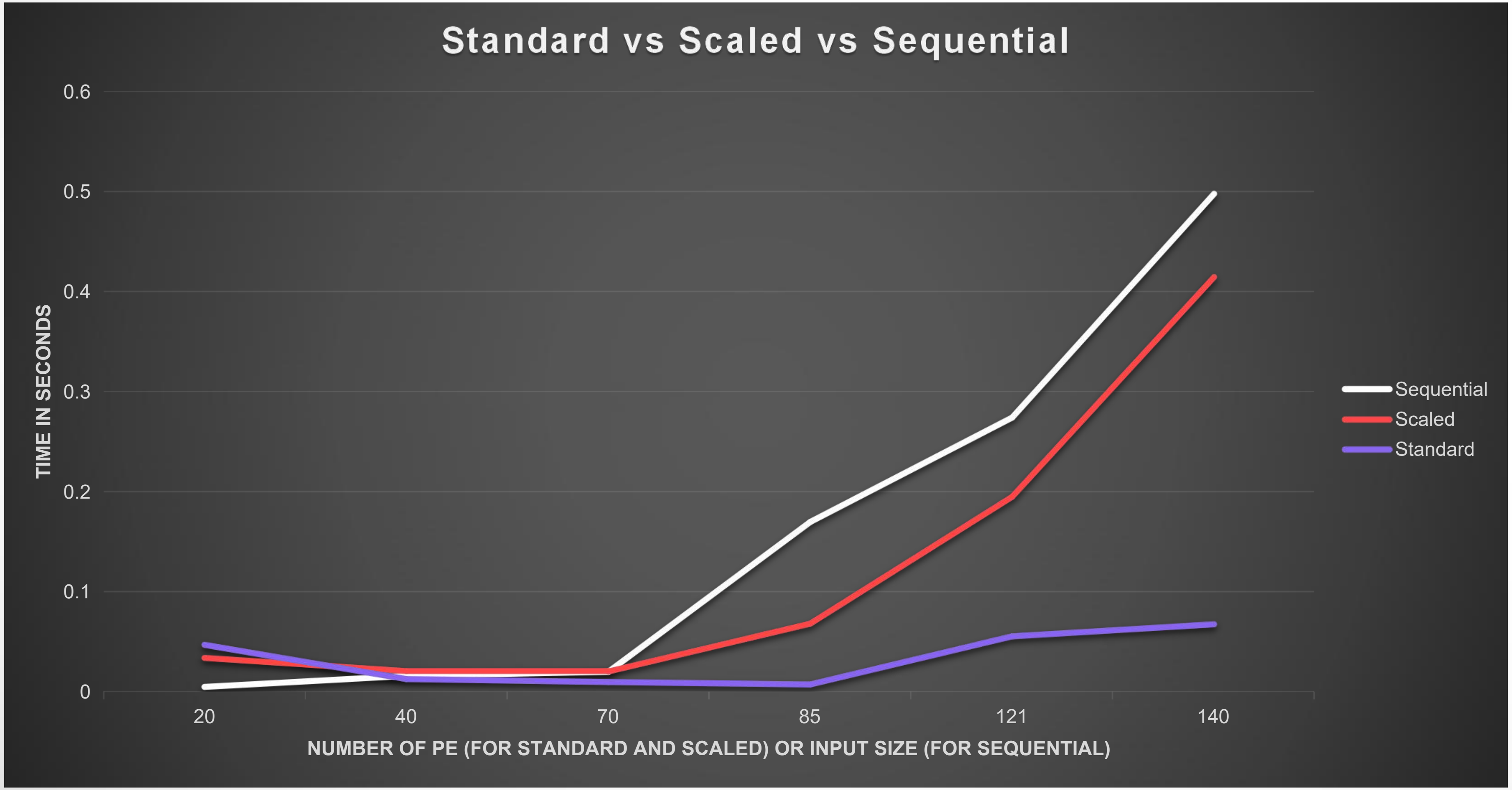
C) Sequential Execution

PEs	Data/PE (Col/PE)	Input Size (Total Col)	Nodes	Cores per Node	Time (in seconds)
1	200	200	1	1	0.005372
1	400	400	1	1	0.016355
1	700	700	1	1	0.020413
1	850	850	1	1	0.16985
1	1210	1210	1	1	0.273971
1	1400	1400	1	1	0.497114



8) Results

Comparasion



Conclusion

> The scaled execution curve is steeper than the standard execution curve, indicating that the benefits of parallelism are more pronounced when the problem size is adjusted appropriately for the number of processors used.

>Scaled execution curve has slope less than the slope of the sequential execution curve. This indicates that as the workload increases, the system's performance improves due to parallelism.

8) Results

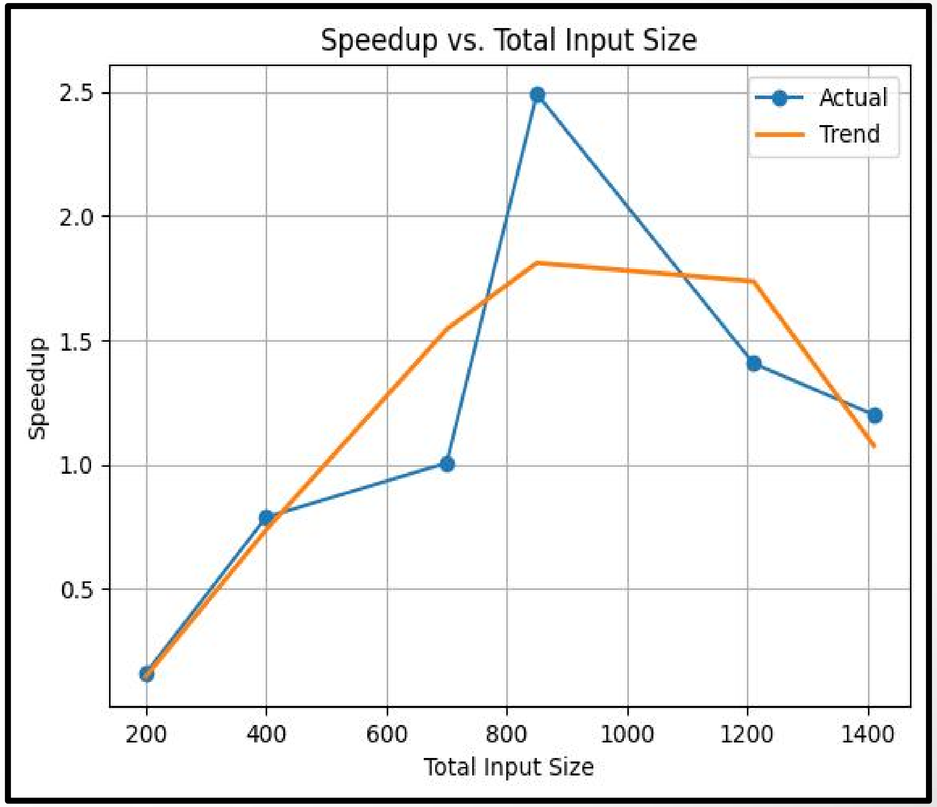
Formula: Speedup = T_{seq} / T_p

T_{seq} is the execution time of sequential algorithm.

T_p is the execution time of the parallel algorithm with p processors.

Speedup

Input Size (Total Col)	seq	T_{seq}	p	Nodes	Cores per Node	Data/PE (Col/PE)	T_p	Actual Speedup	Trend Speedup
200	1	0.005372	20	20	1	10	0.033918	0.15838198	0.14515
400	1	0.016355	40	40	1	10	0.020783	0.78694125	0.735993
700	1	0.020413	70	70	1	10	0.020312	1.00497243	1.54382
850	1	0.16985	85	85	1	10	0.068161	2.491894192	1.811017
1210	1	0.273971	121	121	1	10	0.194891	1.405765274	1.736656
1410	1	0.497114	140	140	1	10	0.41438	1.199657319	1.074976



(Plot Trend Line Code Link)

9) Future Work

1) Access nodes greater than 143 nodes with 1 core per node.

2) Explore strategies which avoid barrier synchronization and gather results with the optimal approach.

3) Implement my parallel approach using OpenMPI or Hybrid of MPI and OpenMPI.

10) References

1) GFG: https://www.geeksforgeeks.org/count-of-subsets-with-sum-equal-to-x/#

2) Coding Ninja: https://www.codingninjas.com/codestudio/library/count-number-of-subsets-with-given-sum

3) Aditya Verma's Video: https://www.youtube.com/watch?v=F7wqWbqYn9g

4) Dr. Jones Lectures on MPI

5) MPI Tutorial: https://mpitutorial.com/tutorials/

6) Desmos Graphing Calculator: https://www.desmos.com/calculator

7) Plotly Chart Studio: https://chart-studio.plotly.com/create/#/