CSE 708 Seminar

Subset Sum Count (Using OpenMP in C)

Name: Kunal Chand
UB Email: kchand@buffalo.edu
Person Number: 50465175
Instructor: Professor Russ Miller

Content

1) Problem Statement

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

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

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

Subset Sum Count = 3

Constraint: All array elements are whole numbers.

2) Sequential Solution

		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	-	-	-	-	-	-	-

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]]

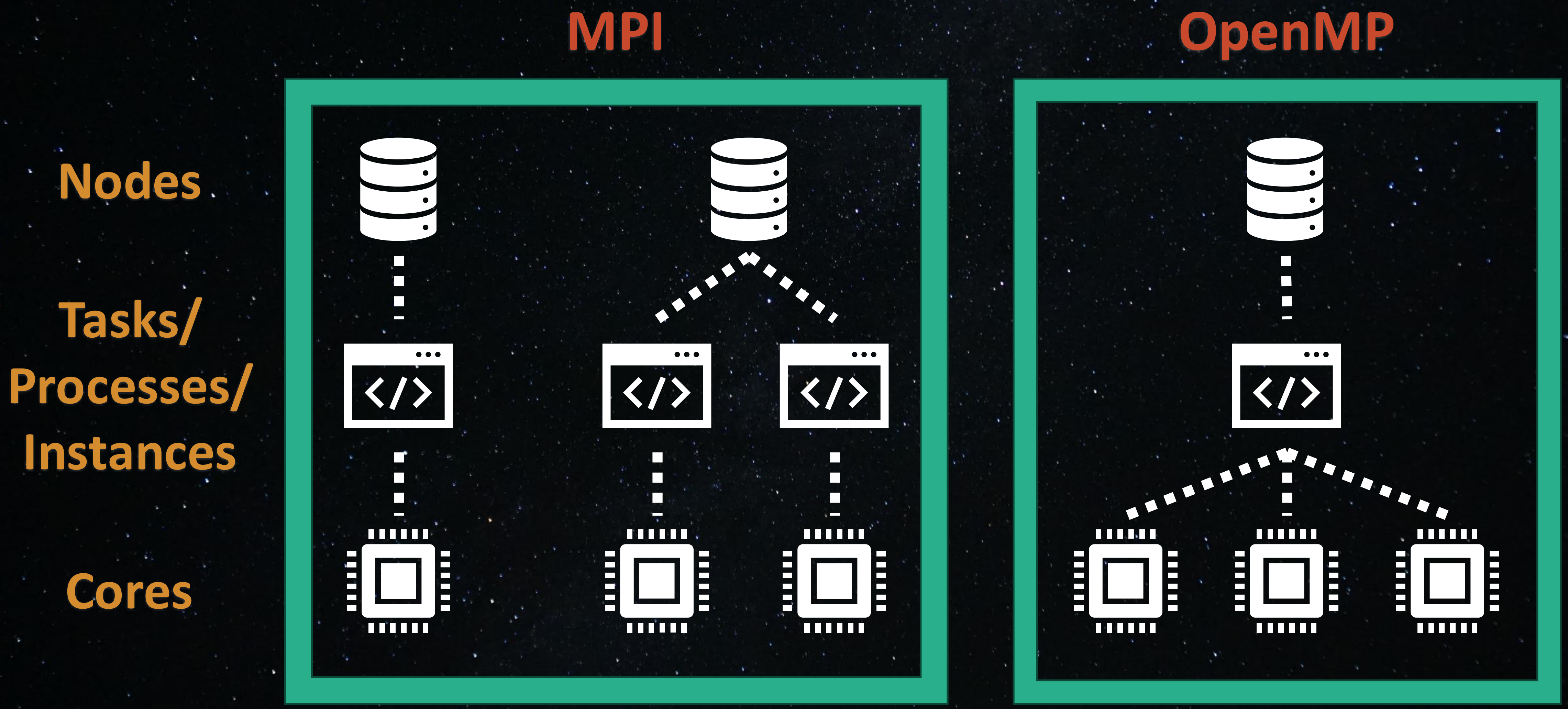
2) Sequential Solution

		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

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]]

```java
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];
}
```

3) MPI vs OpenMP



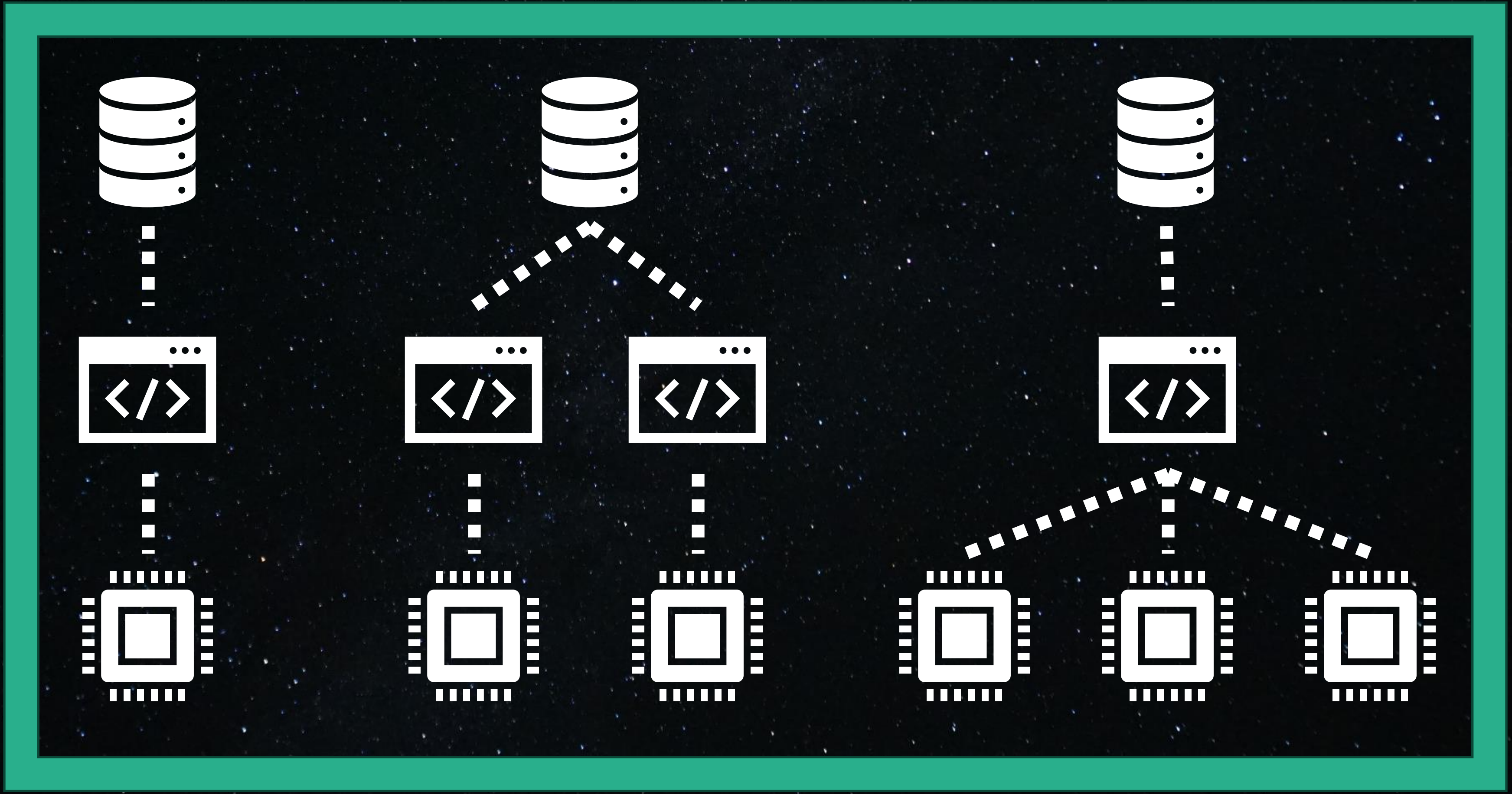
3) MPI vs OpenMP

Hybrid = MPI + OpenMP

Nodes

Tasks/
Processes/
Instances

Cores



3) MPI vs OpenMP

MPI

```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=1
```

OpenMP

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=3
```

Hybrid

```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=3
```

3) MPI vs OpenMP

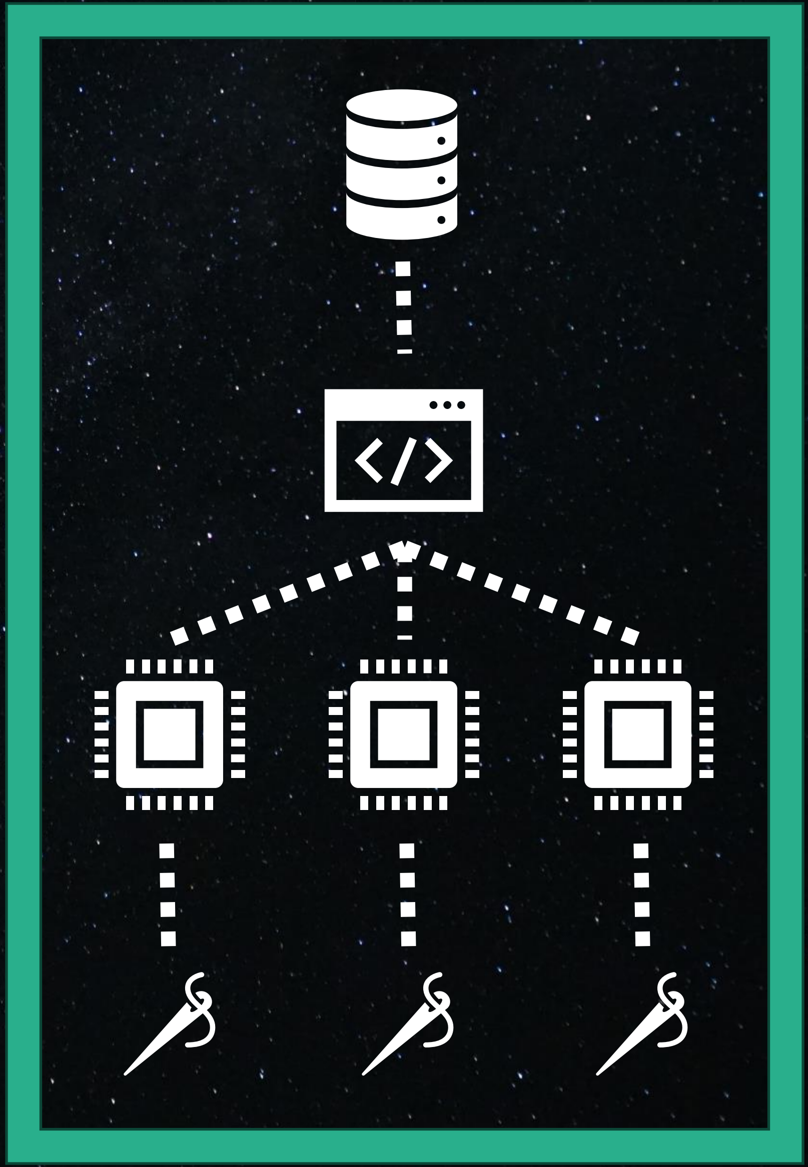
Nodes

Tasks / Processes / Instances

Cores (Hardware)

Threads (Software)

OpenMP



4) Slurm Script

Set # Nodes to 1

```
#SBATCH --nodes=1
```

Set # Instance to 1

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

Set # CPU Cores

```
#SBATCH --cpus-per-task=4
```

Set # threads = # CPU cores
Hence 1 thread per core

```
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
```

4) Slurm Script

OpenMP Configuration

Reserve all nodes & cores (Pizza Box) for exclusive use

Set # threads = # CPU cores
Hence 1 thread per core

Enables support for OpenMP pragmas and directives

slurm.sh ×

708 > open_mp_c_execution > parallel > slurm.sh

```bash
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=64

#SBATCH --time=00:10:00

#SBATCH --partition=general-compute
#SBATCH --qos=general-compute

#SBATCH --cluster=ub-hpc
#SBATCH --reservation=ubhpc-future

#SBATCH --job-name="ssc_64_threads"
#SBATCH --output=output_openmp-64.txt

#SBATCH --exclusive

module load intel

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK


gcc -fopenmp -o compiled_file ssc_openmp.c

./compiled_file

```

5) Parallel Solution

Sequential Code Snippet

```
for (int i = 1; i <= n; i++){
    for (int j = 0; j <= sum; j++){
        if (a[i - 1] > j){
            dp[i][j] = dp[i - 1][j];
        }
        else{
            dp[i][j] = dp[i - 1][j] + dp[i - 1][j - a[i - 1]];
        }
    }
}
```

Parallel OpenMP Code Snippet

```
for (int i = 1; i <= n; i++) {
    #pragma omp parallel for
    for (int j = 0; j <= sum; j++) {
        if (a[i - 1] > j){
            #pragma omp atomic write
            dp[i][j] = dp[i - 1][j];
        }
        else {
            #pragma omp atomic write
            dp[i][j] = dp[i - 1][j] + dp[i - 1][j - a[i - 1]];
        }
    }
}
```

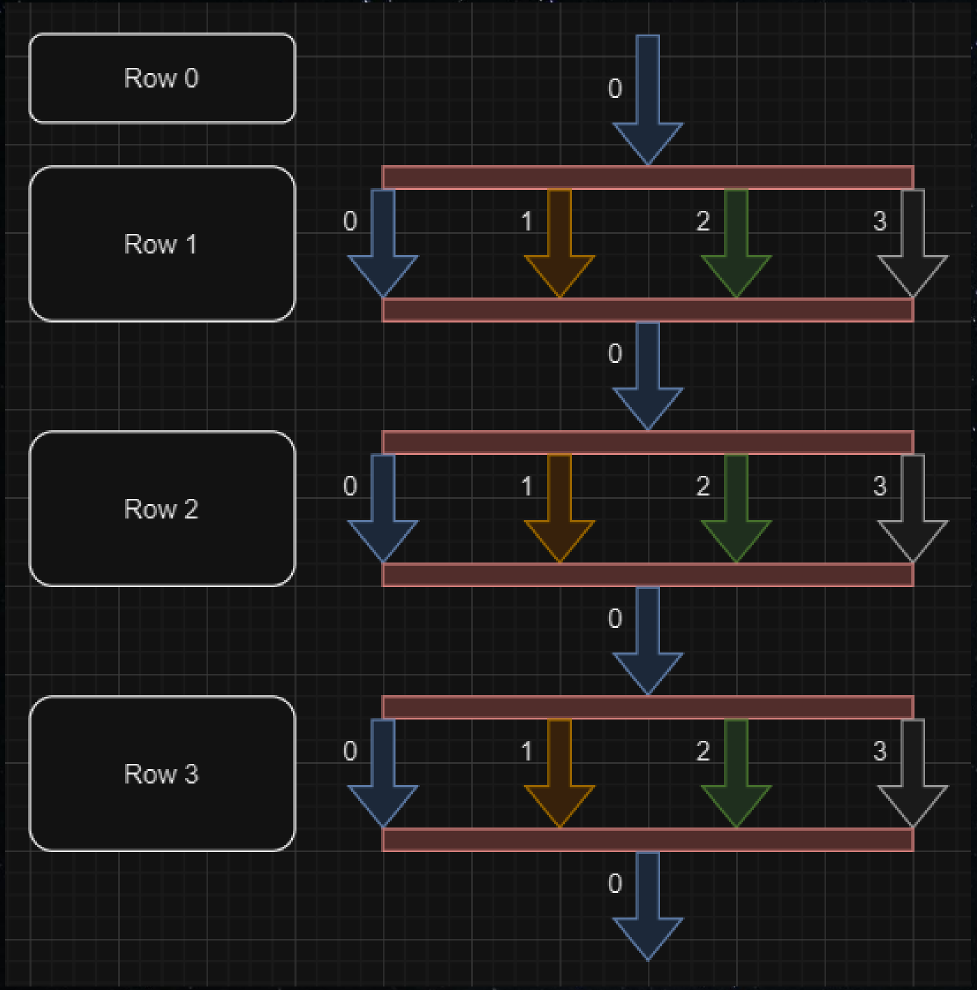
5) Parallel Solution

```
// Row Wise Iteration
for (int i = 1; i <= n; i++) {
    #pragma omp parallel for
    for (int j = 0; j <= sum; j++) {
        // Parallel column execution for a given row i
    }
    // Resumes sequential execution
}
```

```
for (int i = 1; i <= n; i++) {
    #pragma omp parallel for
    for (int j = 0; j <= sum; j++) {

        int thread_id = omp_get_thread_num();
        thread_table[i][j] = thread_id;

        if (a[i - 1] > j){
            #pragma omp atomic write
            dp[i][j] = dp[i - 1][j];
        }
        else {
            #pragma omp atomic write
            dp[i][j] = dp[i - 1][j] + dp[i - 1][j - a[i - 1]];
        }
    }
}
```

```
Max Threads = 4

Input Array = [3, 1, 2, 4, 5]
Target Sum = 6

Thread Table:
Row 0: 0 0 0 0 0 0 0
Row 1: 0 0 1 1 2 2 3
Row 2: 0 0 1 1 2 2 3
Row 3: 0 0 1 1 2 2 3
Row 4: 0 0 1 1 2 2 3
Row 5: 0 0 1 1 2 2 3

DP Table:
Row 0: 1 0 0 0 0 0 0
Row 1: 1 0 0 1 0 0 0
Row 2: 1 1 0 1 1 0 0
Row 3: 1 1 1 2 1 1 1
Row 4: 1 1 1 2 2 2 2
Row 5: 1 1 1 2 2 3 3

Subset Sum Count = 3
```

5) Parallel Solution




```
Max Threads = 4

Input Array = [3, 1, 2, 4, 5]
Target Sum = 6

Thread Table:
Row 0: 0 0 0 0 0 0 0
Row 1: 0 0 1 1 2 2 3
Row 2: 0 0 1 1 2 2 3
Row 3: 0 0 1 1 2 2 3
Row 4: 0 0 1 1 2 2 3
Row 5: 0 0 1 1 2 2 3

DP Table:
Row 0: 1 0 0 0 0 0 0
Row 1: 1 0 0 1 0 0 0
Row 2: 1 1 0 1 1 0 0
Row 3: 1 1 1 2 1 1 1
Row 4: 1 1 1 2 2 2 2
Row 5: 1 1 1 2 2 3 3

Subset Sum Count = 3
```

6) Execution Result

A) Standard Execution (Amdahl's Law):

- Total size of input data remains same
- Increase the number of cores
- With more cores, each core operates on lesser data

B) Scaled Execution (Gustafson's Law):

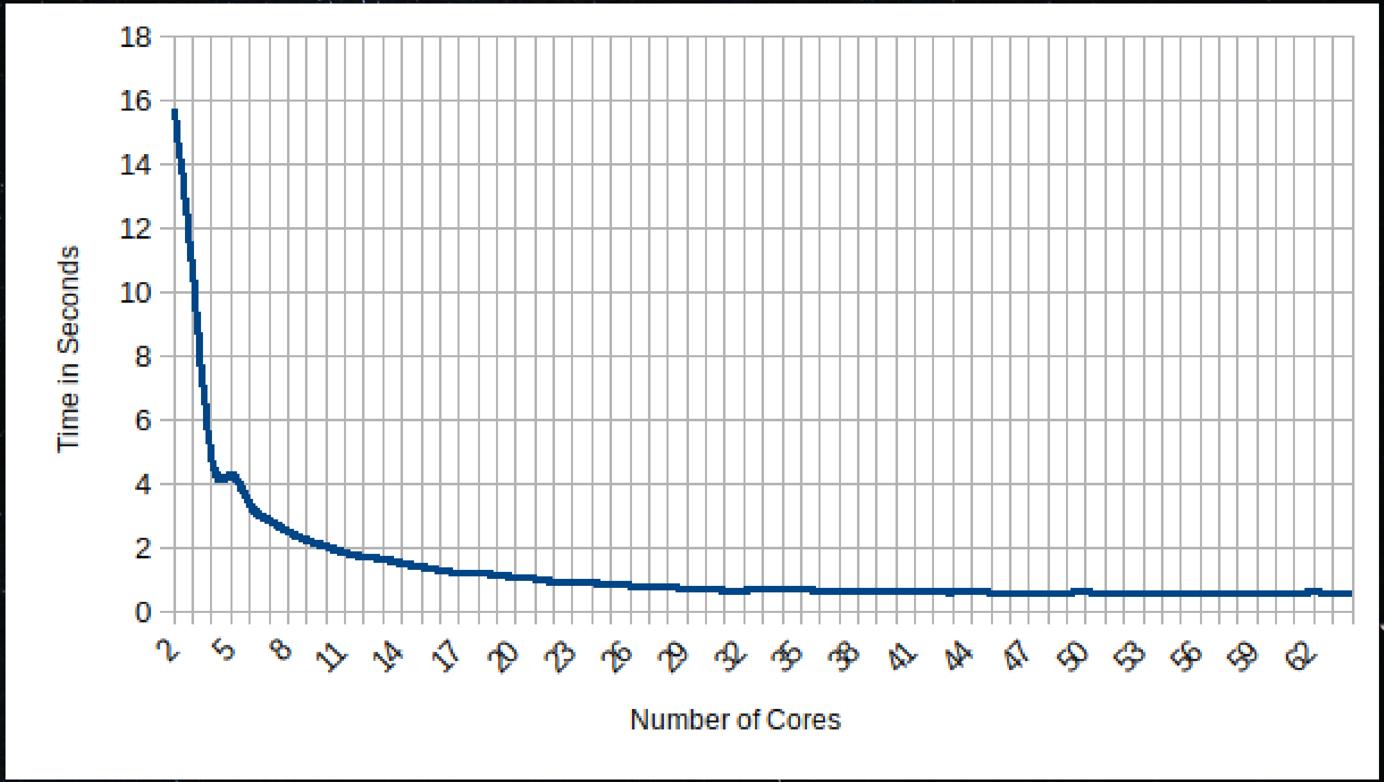
- Fix amount of data that each core operates on
- Increase the number of cores
- With more cores, the total size of the input data should get increased, because data per core is constant

C) Sequential Execution

6) Execution Result

A) Standard Execution (Amdahl's Law):

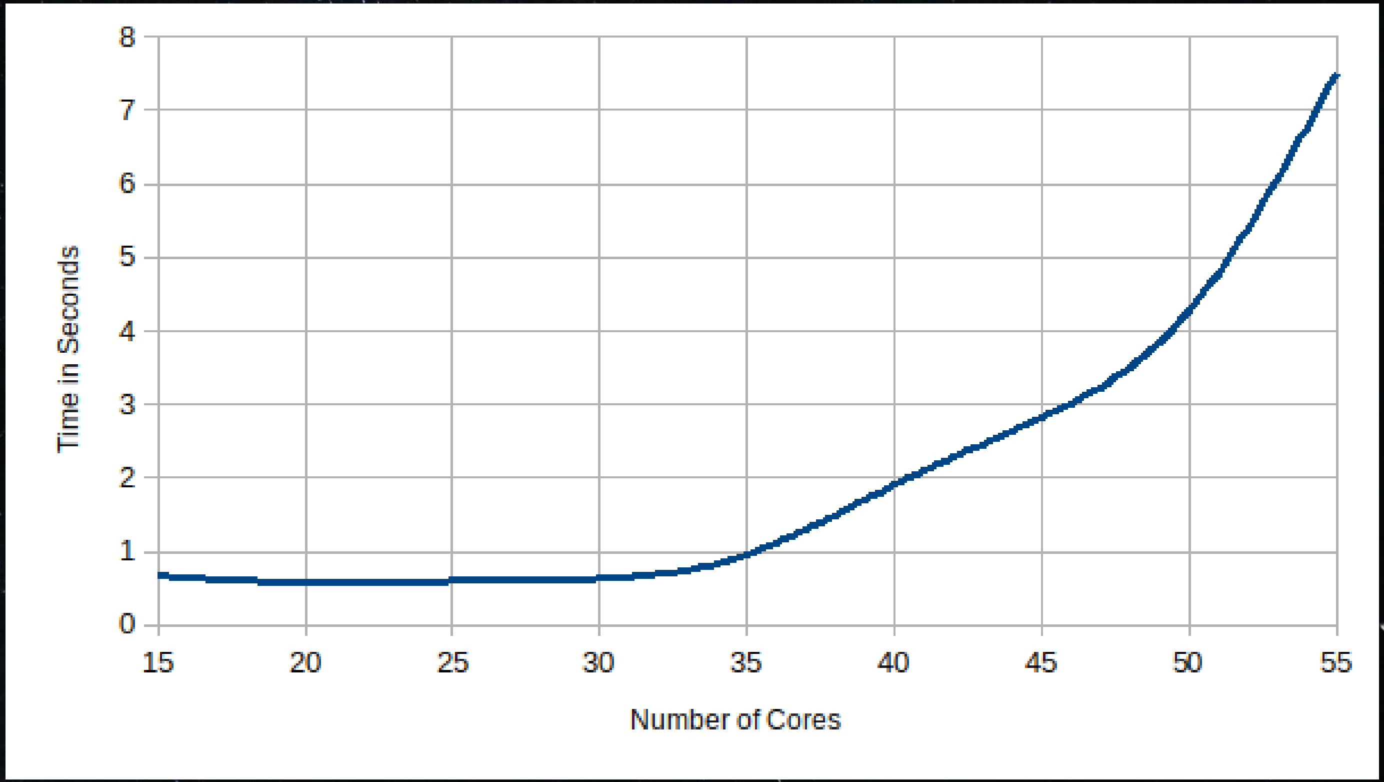
Cores	Data/Core (Col/Core)	Input Size (Total Col)	Time (in seconds)
1	100 M	100 M	31.46356
2	50 M	100 M	15.75797
3	33.3 M	100 M	10.53631
4	25 M	100 M	4.781726
7	14.3 M	100 M	2.885004
15	6.67 M	100 M	1.430131
25	4 M	100 M	0.890899
35	2.85 M	100 M	0.716509
45	2.2M	100 M	0.617453
64	1.56 M	100 M	0.589691



6) Execution Result

B) Scaled Execution (Gustafson's Law):

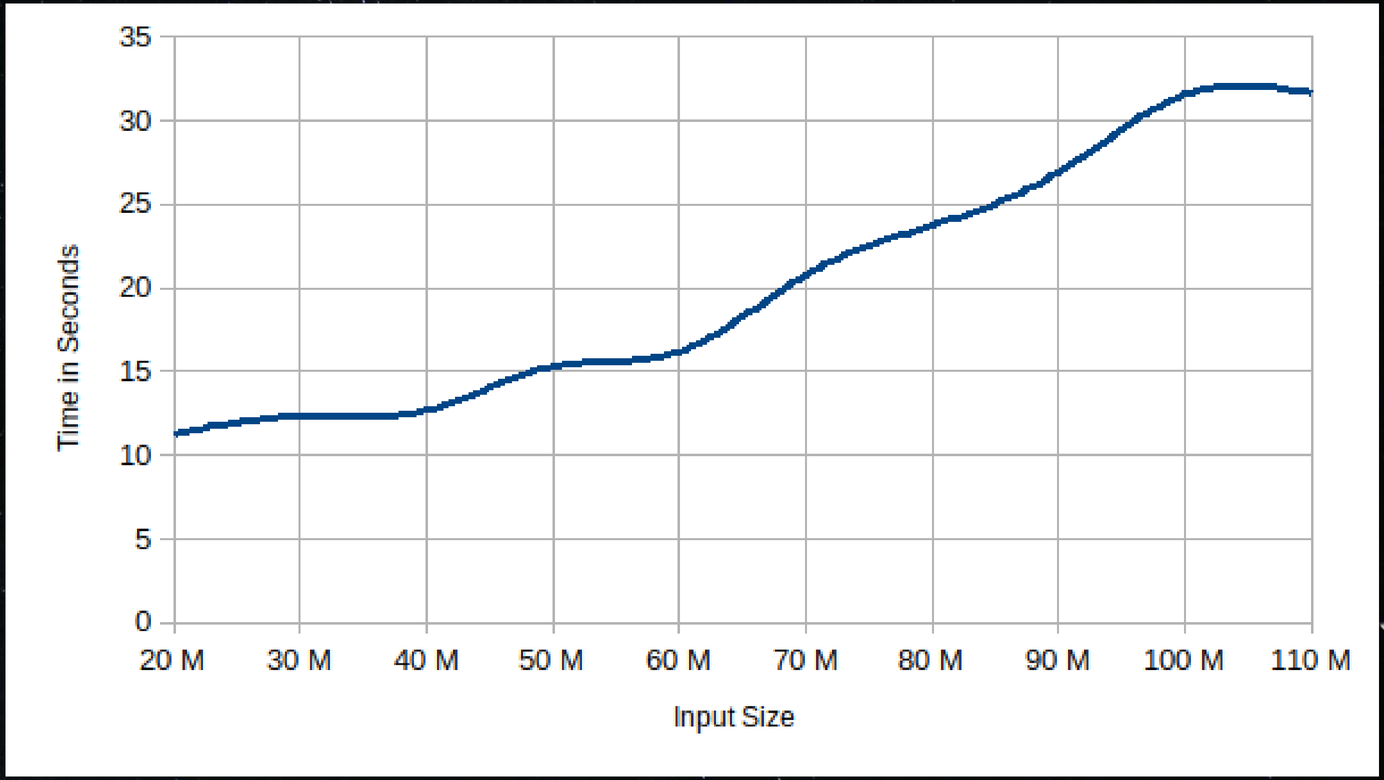
Cores	Data/Core (Col/Core)	Input Size (Total Col)	Time (in seconds)
10	2 M	20 M	0.624523
15	2 M	30 M	0.675528
20	2 M	40 M	0.586507
25	2 M	50 M	0.607871
30	2 M	60 M	0.6315
35	2 M	70 M	0.96475
40	2 M	80 M	1.916973
45	2 M	90 M	2.823243
50	2 M	100 M	4.27448
55	2 M	110 M	7.51923



6) Execution Result

C) Sequential Execution

Cores	Data/Core (Col/Core)	Input Size (Total Col)	Time (in seconds)
1	20 M	20 M	11.30215
1	30 M	30 M	12.34482
1	40 M	40 M	12.6843
1	50 M	50 M	15.32529
1	60 M	60 M	16.19791
1	70 M	70 M	20.78737
1	80 M	80 M	23.74192
1	90 M	90 M	26.91424
1	100 M	100 M	31.57325
1	110 M	110 M	31.6582



6) Execution Result

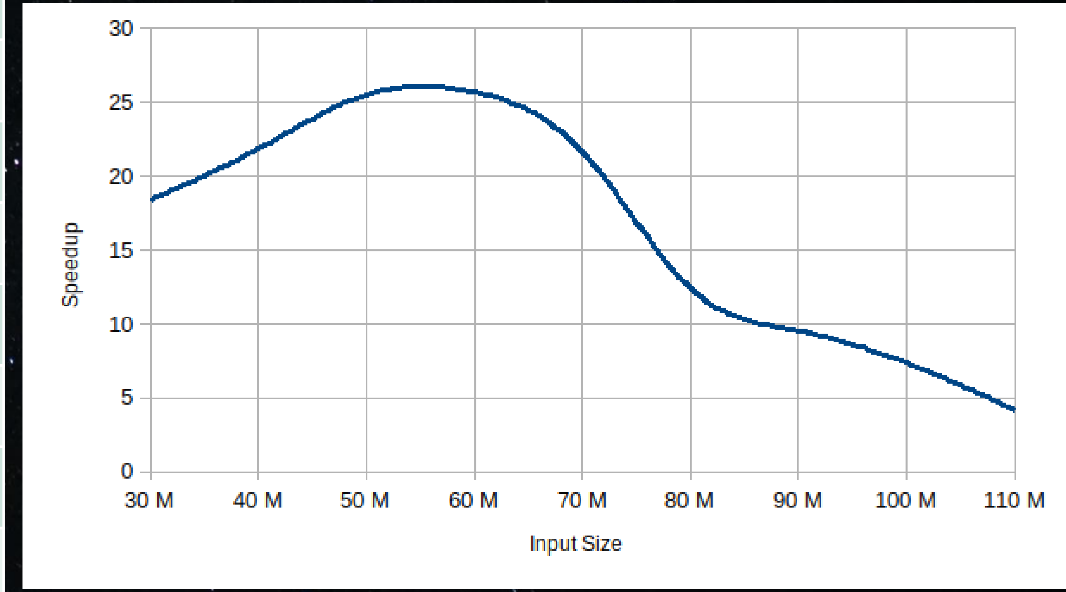
Speedup

Formula: Speedup = T_{seq}/T_p

T_{seq} is the execution time of sequential algorithm

T_p is the execution time of parallel algorithm with p processors

Input Size (Total Col)	seq	T_{seq}	Cores	Data/Core (Col/Core)	T_p	Speedup
20 M	1	11.3	10	2 M	0.62	18.2258
30 M	1	12.34	15	2 M	0.67	18.4179
40 M	1	12.68	20	2 M	0.58	21.8620
50 M	1	15.32	25	2 M	0.6	25.5334
60 M	1	16.19	30	2 M	0.63	25.6984
70 M	1	20.78	35	2 M	0.96	21.6458
80 M	1	23.74	40	2 M	1.91	12.4293
90 M	1	26.91	45	2 M	2.82	9.54255
100 M	1	31.57	50	2 M	4.27	7.3944
110 M	1	31.65	55	2 M	7.51	4.2143



7) Reference

1) GFG: https://www.geeksforgeeks.org/count-of-subsets-with-sum-equal-to-x/
2) Princeton University BootCamp: https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf
3) Dr. Jones Lectures on OpenMP