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\} \quad \text{Targets Sum} = 6

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

\textbf{Subset Sum Count} = 3

Constraints: All elements of the array are whole numbers.
2) Sequential Approach

\[ T(n) = O(\text{sum} \times \text{array size}) \]

if (array[i] > j):
\[ \text{DP}[i][j] = \text{DP}[i-1][j] \]
else:
\[ \text{DP}[i][j] = \text{DP}[i-1][j] + \text{DP}[i-1][j-\text{array}[i]] \]
3) Sequential Implementation

```
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 ) :  \text{DP}[i][j] = \text{DP}[i-1][j] \\
else :  \text{DP}[i][j] = \text{DP}[i-1][j] + \text{DP}[i-1][j-\text{array}[i]]
4) Parallel Approach

If (array[i] > j): \( DP[i][j] = DP[i-1][j] \)
Else: \( DP[i][j] = DP[i-1][j] + DP[i-1][j-array[i]] \)
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

**Issue 1: Infinite Wait Time**

Receiver has to wait indefinitely 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.
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.

### Issues

**Issue 1**: Infinite Wait Time

**Issue 2**: Can’t send requests

**Issue 3**: Store & process multiple requests

### Fixes

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

**Fix 3**: Synchronize processors at each layer of computation to reduce the quantity of information in the memory.
6) Parallel Implementation

```
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

```c
// 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],
                 1, 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]);
}
```

```c
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]]
```
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]]
If you want to reserve all the cores on a node for your own exclusive use during your job, you can use the \texttt{--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 "\texttt{--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.
Total Nodes Requested

Number of Cores/Processors per Node

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

1 Processor per Node

Multiple Processors per Node

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.

Sequential Code Execution

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 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)

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

<table>
<thead>
<tr>
<th>PEs</th>
<th>Data/PE (Col/PE)</th>
<th>Input Size (Total Col)</th>
<th>Nodes</th>
<th>Cores per Node</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>14</td>
<td>2000</td>
<td>15</td>
<td>10</td>
<td>0.198445</td>
</tr>
<tr>
<td>180</td>
<td>12</td>
<td>2000</td>
<td>18</td>
<td>10</td>
<td>0.263768</td>
</tr>
<tr>
<td>230</td>
<td>9</td>
<td>2000</td>
<td>23</td>
<td>10</td>
<td>0.29483</td>
</tr>
<tr>
<td>250</td>
<td>9</td>
<td>2000</td>
<td>25</td>
<td>10</td>
<td>0.365401</td>
</tr>
<tr>
<td>300</td>
<td>7</td>
<td>2000</td>
<td>30</td>
<td>10</td>
<td>0.45483</td>
</tr>
<tr>
<td>350</td>
<td>6</td>
<td>2000</td>
<td>35</td>
<td>10</td>
<td>0.494543</td>
</tr>
</tbody>
</table>
## 8) Results

### B) Scaled Execution (Gustafson's Law)

<table>
<thead>
<tr>
<th>PEs</th>
<th>Data/PE (Col/PE)</th>
<th>Input Size (Total Col)</th>
<th>Nodes</th>
<th>Cores per Node</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>200</td>
<td>20</td>
<td>1</td>
<td>0.033918</td>
</tr>
<tr>
<td>40</td>
<td>10</td>
<td>400</td>
<td>40</td>
<td>1</td>
<td>0.020783</td>
</tr>
<tr>
<td>70</td>
<td>10</td>
<td>700</td>
<td>70</td>
<td>1</td>
<td>0.020312</td>
</tr>
<tr>
<td>85</td>
<td>10</td>
<td>850</td>
<td>85</td>
<td>1</td>
<td>0.068161</td>
</tr>
<tr>
<td>121</td>
<td>10</td>
<td>1210</td>
<td>121</td>
<td>1</td>
<td>0.194891</td>
</tr>
<tr>
<td>140</td>
<td>10</td>
<td>1410</td>
<td>140</td>
<td>1</td>
<td>0.41438</td>
</tr>
<tr>
<td>180</td>
<td>10</td>
<td>1800</td>
<td>18</td>
<td>10</td>
<td>0.109324</td>
</tr>
<tr>
<td>230</td>
<td>10</td>
<td>2300</td>
<td>23</td>
<td>10</td>
<td>0.136648</td>
</tr>
<tr>
<td>250</td>
<td>10</td>
<td>2500</td>
<td>25</td>
<td>10</td>
<td>0.144504</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>3000</td>
<td>30</td>
<td>10</td>
<td>0.22354</td>
</tr>
<tr>
<td>320</td>
<td>10</td>
<td>3200</td>
<td>32</td>
<td>10</td>
<td>0.21438</td>
</tr>
<tr>
<td>350</td>
<td>10</td>
<td>3500</td>
<td>35</td>
<td>10</td>
<td>0.240661</td>
</tr>
</tbody>
</table>

![Graph showing the relationship between time in seconds and number of processors]

![Graph showing the relationship between time in seconds and number of processors]
## 8) Results

### C) Sequential Execution

<table>
<thead>
<tr>
<th>PEs</th>
<th>Data/PE (Col/PE)</th>
<th>Input Size (Total Col)</th>
<th>Nodes</th>
<th>Cores per Node</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
<td>200</td>
<td>1</td>
<td>1</td>
<td>0.005372</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>400</td>
<td>1</td>
<td>1</td>
<td>0.016355</td>
</tr>
<tr>
<td>1</td>
<td>700</td>
<td>700</td>
<td>1</td>
<td>1</td>
<td>0.020413</td>
</tr>
<tr>
<td>1</td>
<td>850</td>
<td>850</td>
<td>1</td>
<td>1</td>
<td>0.16985</td>
</tr>
<tr>
<td>1</td>
<td>1210</td>
<td>1210</td>
<td>1</td>
<td>1</td>
<td>0.273971</td>
</tr>
<tr>
<td>1</td>
<td>1400</td>
<td>1400</td>
<td>1</td>
<td>1</td>
<td>0.497114</td>
</tr>
</tbody>
</table>

![Graph showing the relationship between input size and time in seconds](image)
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

### Speedup

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

Formula: \[ \text{Speedup} = \frac{T_{seq}}{T_p} \]

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

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

(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/#
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/#/