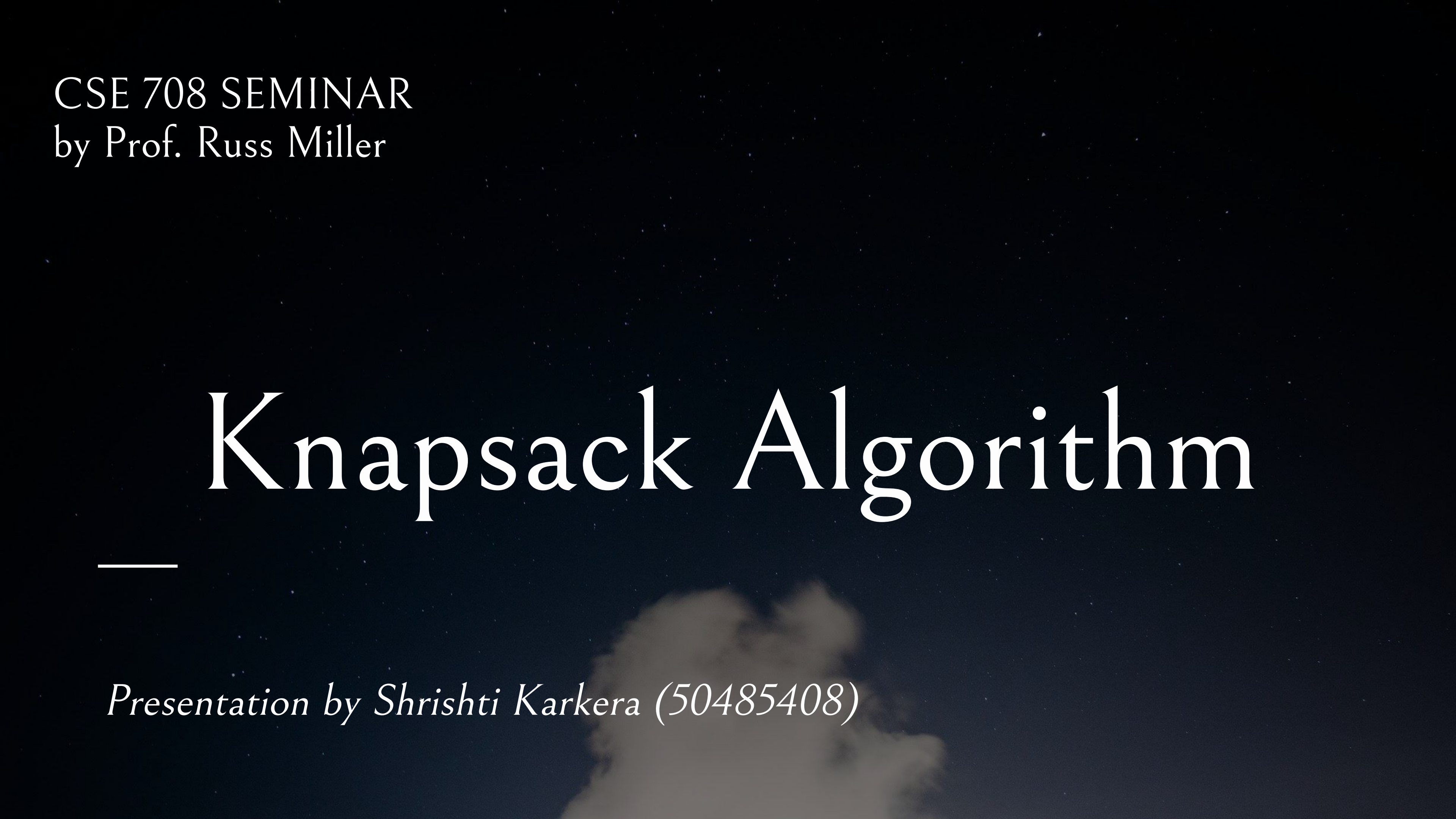
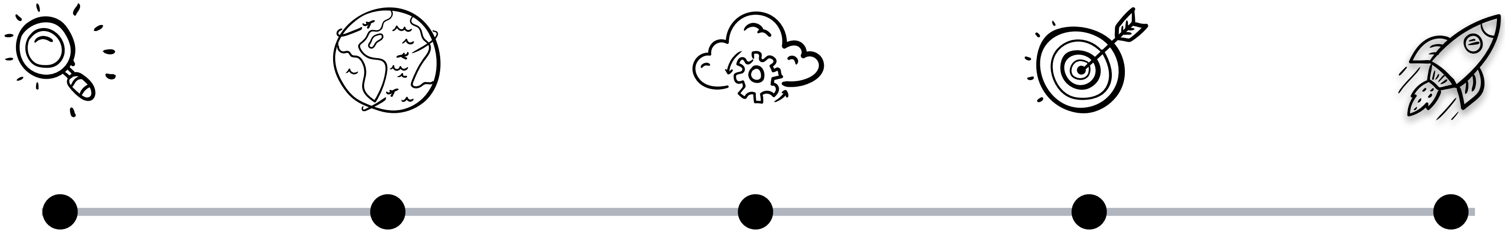
CSE 708 SEMINAR
by Prof. Russ Miller

# Knapsack Algorithm



*Presentation by Shrishti Karkera (50485408)*

# Overview



Problem Definition

- Sequential approach
- Sequential implementation

- Parallel approach
- Parallel implementation

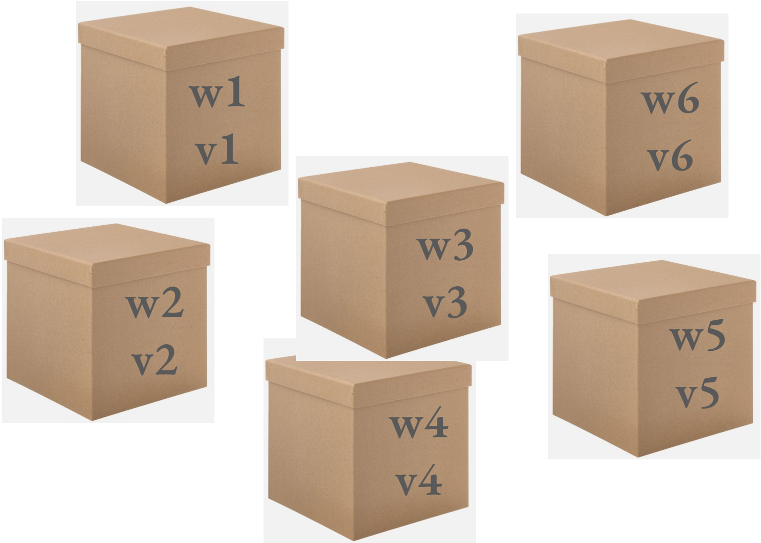
Results

Observation

# 0/1 Knapsack

W <= Total weight

Max Total value



# Recursion

```
                 def knapsack(W, wt, val):

Base Case            if n == 0 or W == 0:
                           return 0

                     not_pick = knapSack(W, wt, val, n-1)
Conditions           pick = -1e9
                     if (wt[n-1] <= W):
                          pick = val[n-1] + knapSack(W-wt[n-1], wt, val, n-1)
                     return max(pick, not_pick)
```

# Recursion with memoization

*dp = 2d array (n+1 x W+1)*

Base Case

```
def knapsack(W, wt, val):
    if n == 0 or W == 0:
        dp[n][W] = 0
        return 0
```

Check if value already present in the table

Conditions

```
    not_pick = knapSack(W, wt, val, n-1)
    pick = -1e9
    if (wt[n-1] <= W):
        pick = val[n-1] + knapSack(W-wt[n-1], wt, val, n-1)
    dp[n][W] = max(pick, not_pick)
```

dp[i][w] = dp(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])

# Tabular DP

weights = [3, 4, 7]

values = [4, 5, 8]

W = 7

w

i

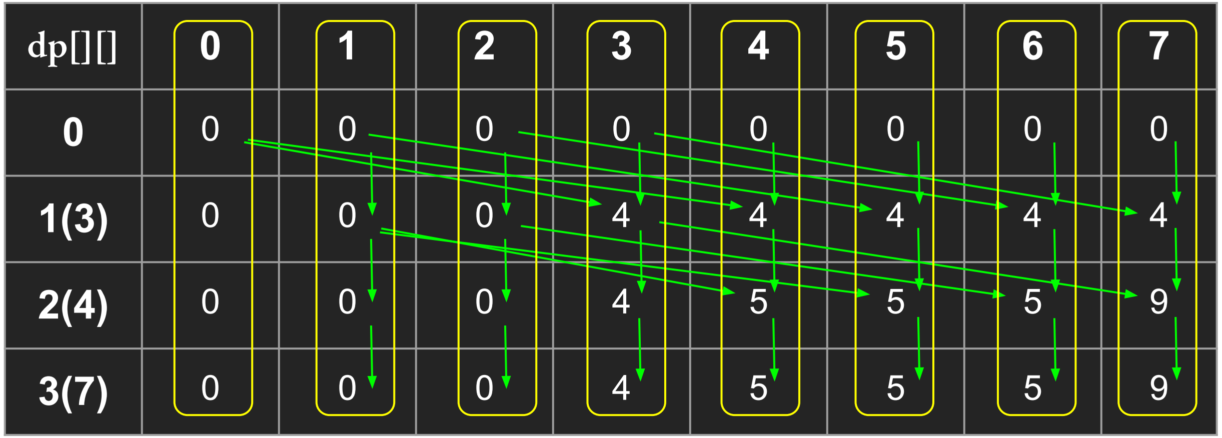
| dp[][] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 |
| **2** | 0 | 0 | 0 | 4 | 5 | 5 | 5 | 9 |
| **3** | 0 | 0 | 0 | 4 | 5 | 5 | 5 | 9 |

Max profit

**dp[i][w] = dp(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])**

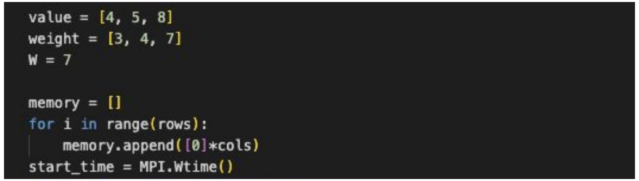
# Approach 1 - 1 column per core

values = [4, 5, 8]

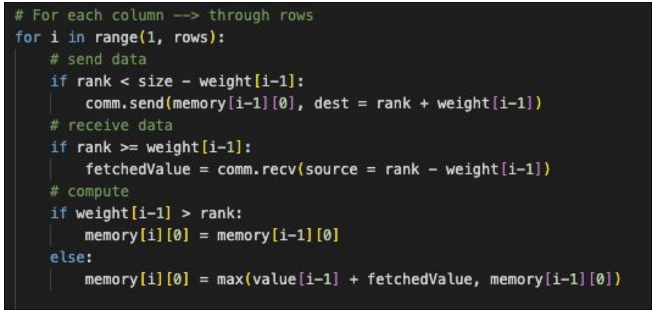


| dp[][] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1(3) | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 |
| 2(4) | 0 | 0 | 0 | 4 | 5 | 5 | 5 | 9 |
| 3(7) | 0 | 0 | 0 | 4 | 5 | 5 | 5 | 9 |

Code



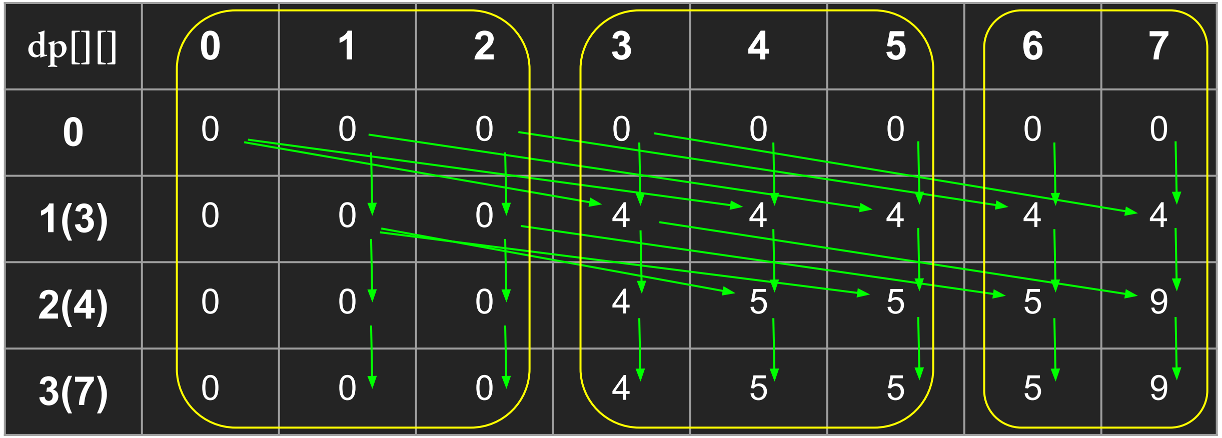
```
value = [4, 5, 8]
weight = [3, 4, 7]
W = 7

memory = []
for i in range(rows):
    memory.append([0]*cols)
start_time = MPI.Wtime()
```



```
# For each column --> through rows
for i in range(1, rows):
    # send data
    if rank < size - weight[i-1]:
        comm.send(memory[i-1][0], dest = rank + weight[i-1])
    # receive data
    if rank >= weight[i-1]:
        fetchedValue = comm.recv(source = rank - weight[i-1])
    # compute
    if weight[i-1] > rank:
        memory[i][0] = memory[i-1][0]
    else:
        memory[i][0] = max(value[i-1] + fetchedValue, memory[i-1][0])
```

# Approach 2 - multiple columns per core

values = [4, 5, 8]

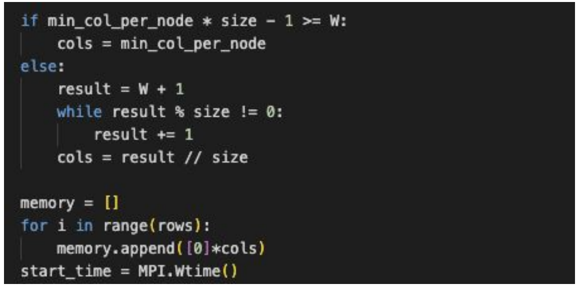


| dp[][] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1(3) | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 |
| 2(4) | 0 | 0 | 0 | 4 | 5 | 5 | 5 | 9 |
| 3(7) | 0 | 0 | 0 | 4 | 5 | 5 | 5 | 9 |

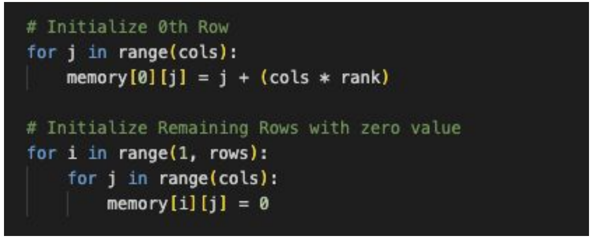
Iterate

1. Send data
2. Receive data
3. Calculate for the current cell

Code

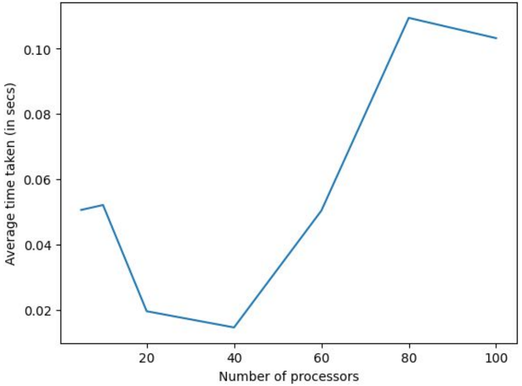


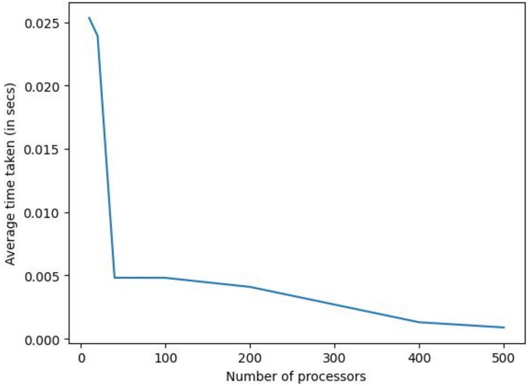
```
if min_col_per_node * size - 1 >= W:
    cols = min_col_per_node
else:
    result = W + 1
    while result % size != 0:
        result += 1
    cols = result // size

memory = []
for i in range(rows):
    memory.append([0]*cols)
start_time = MPI.Wtime()
```

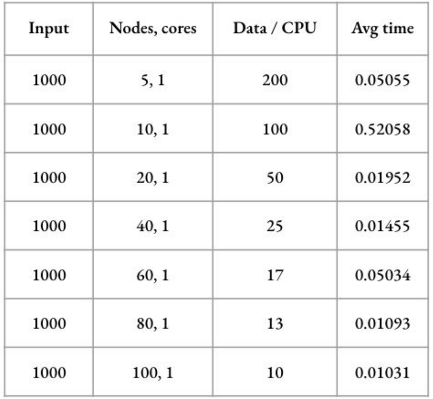


```
# Initialize 0th Row
for j in range(cols):
    memory[0][j] = j + (cols * rank)

# Initialize Remaining Rows with zero value
for i in range(1, rows):
    for j in range(cols):
        memory[i][j] = 0
```

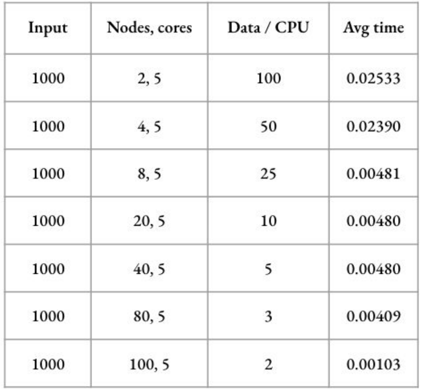
# Standard execution







| Input | Nodes, cores | Data / CPU | Avg time |
|---|---|---|---|
| 1000 | 5, 1 | 200 | 0.05055 |
| 1000 | 10, 1 | 100 | 0.52058 |
| 1000 | 20, 1 | 50 | 0.01952 |
| 1000 | 40, 1 | 25 | 0.01455 |
| 1000 | 60, 1 | 17 | 0.05034 |
| 1000 | 80, 1 | 13 | 0.01093 |
| 1000 | 100, 1 | 10 | 0.01031 |



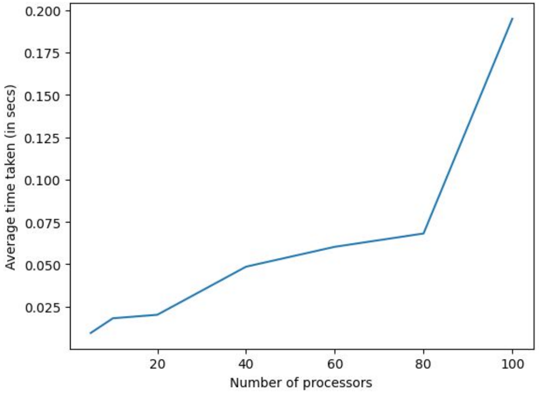
| Input | Nodes, cores | Data / CPU | Avg time |
|---|---|---|---|
| 1000 | 2, 5 | 100 | 0.02533 |
| 1000 | 4, 5 | 50 | 0.02390 |
| 1000 | 8, 5 | 25 | 0.00481 |
| 1000 | 20, 5 | 10 | 0.00480 |
| 1000 | 40, 5 | 5 | 0.00480 |
| 1000 | 80, 5 | 3 | 0.00409 |
| 1000 | 100, 5 | 2 | 0.00103 |

## Amdahl's Law

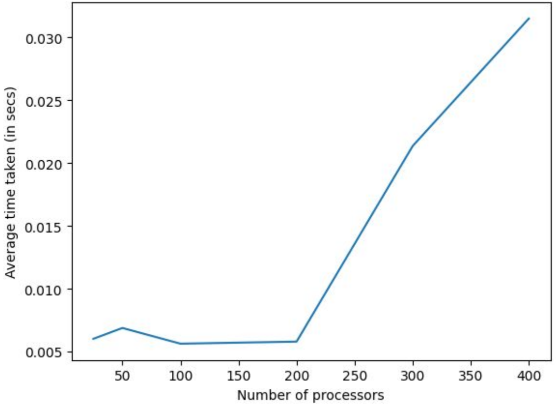
$$S_p \leq \frac{1}{(1-f)+\frac{f}{p}}$$

f is the fraction of the program that must be executed serially (i.e., cannot be parallelized) and p is the number of processors.

# Scaled execution



| Input | Nodes, cores | Data / CPU | Avg time |
|---|---|---|---|
| 100 | 5, 1 | 20 | 0.09533 |
| 200 | 10, 1 | 20 | 0.01390 |
| 400 | 20, 1 | 20 | 0.02481 |
| 800 | 40, 1 | 20 | 0.04480 |
| 1200 | 60, 1 | 20 | 0.06480 |
| 1600 | 80, 1 | 20 | 0.06809 |
| 2000 | 100, 1 | 20 | 0.19103 |



| Input | Nodes, cores | Data / CPU | Avg time |
|---|---|---|---|
| 250 | 5, 5 | 10 | 0.00633 |
| 500 | 10, 5 | 10 | 0.00690 |
| 1000 | 20, 5 | 10 | 0.00561 |
| 2000 | 40, 5 | 10 | 0.00570 |
| 3000 | 60, 5 | 10 | 0.02180 |
| 4000 | 80, 5 | 10 | 0.03109 |

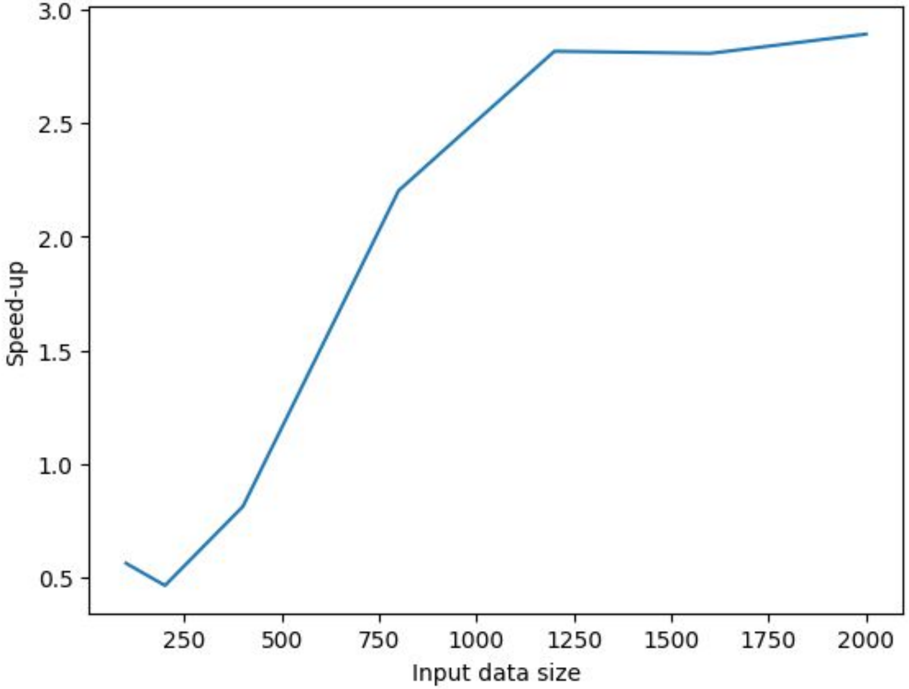
## Gustafson's Law

$$S_p = p - (p - 1) * f$$

where
f is the fraction of the program that is inherently serial and p is the number of processors

# Speedup

$$\textbf{Speed-up} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$$



# References

https://mpi4py.readthedocs.io/en/stable/tutorial.html

https://rabernat.github.io/research_computing/parallel-programming-with-mpi-for-python.html

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

https://www.educative.io/answers/difference-between-amdahls-and-gustafsons-laws

https://www.stolaf.edu/people/rab/pdc/text/alg.htm#:~:text=to%20be%20avoided.-,Speedup,we%20have%20n%2Dfold%20speedup.

# Thank you!

Feel free to ask questions