Bader Houran

202205005

Lebanese American University- LAU

**1)**

**Evaluation of the Trapezoidal Method**

#include <mpi.h>

#include <stdio.h>

#include <math.h>

// Function to evaluate the curve (y = f(x))

float f(float x) {

return x \* x ; // Example: y = x^2

}

// Function to compute the area of a trapezoid

float trapezoid\_area(float a, float b, float d) {

float area = 0;

for (float x = a; x < b; x+=d) {

area += f(x) + f(x+d);

}

return area \* d / 2.0f;

}

int main(int argc, char\*\* argv) {

int rank, size;

float a = 0.0f, b = 1.0f; // Limits of integration

int n;

float start, end, local\_area, total\_area;

double start\_time, end\_time, elapsed\_time;

MPI\_Init(&argc, &argv); // Initialize MPI

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Get rank of the process

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Get number of processes

if (rank == 0) {

// Get the number of intervals from the user

printf("Enter the number of intervals: ");

scanf("%d", &n);

}

// Broadcast the number of intervals to all processes

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

start\_time = MPI\_Wtime();

// Calculate the interval size for each process

float d = (b - a) / n; // delta

float region = (b - a)/ size;

// Calculate local bounds for each process

start = a + rank \* region;

end = start + region;

// Each process calculates the area of its subinterval

local\_area = trapezoid\_area(start, end, d);

// Reduce all local areas to the total area on the root process

MPI\_Reduce(&local\_area, &total\_area, 1, MPI\_FLOAT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

end\_time = MPI\_Wtime();

if (rank == 0) {

printf("The total area under the curve is: %f\n", total\_area);

execution\_time = end\_time - start\_time;

printf("Execution time: %f seconds\n", execution\_time);

}

MPI\_Finalize(); // Finalize MPI

return 0;

}

1)Execute the previous modified code:

S(p)=Execution time using one processor/ Execution time using multiple processors=0.000193/0.000190(for 2 processors) =1.015789 seconds

Efficiency=Speed-Up/Number of Processors=1.015789 /2=0.5078945 seconds

2) Graph:

import matplotlib.pyplot as plt

# Example data

processors = [1,2,3, 4]  # Number of processors

times = [0.000193,0.000190,0.000179, 0.000107]  # Updated execution times (in seconds)

# Calculate speed-up

serial\_time = times[0]

speed\_up = [serial\_time / t for t in times]

# Calculate efficiency

efficiency = [speed\_up[i] / processors[i] for i in range(len(processors))]

# Plot speed-up

plt.figure()

plt.plot(processors, speed\_up, marker='o')

plt.title('Speed-Up vs Number of Processors')

plt.xlabel('Number of Processors')

plt.ylabel('Speed-Up')

plt.grid(True)

plt.show()

# Plot efficiency

plt.figure()

plt.plot(processors, efficiency, marker='o')

plt.title('Efficiency vs Number of Processors')

plt.xlabel('Number of Processors')

plt.ylabel('Efficiency')

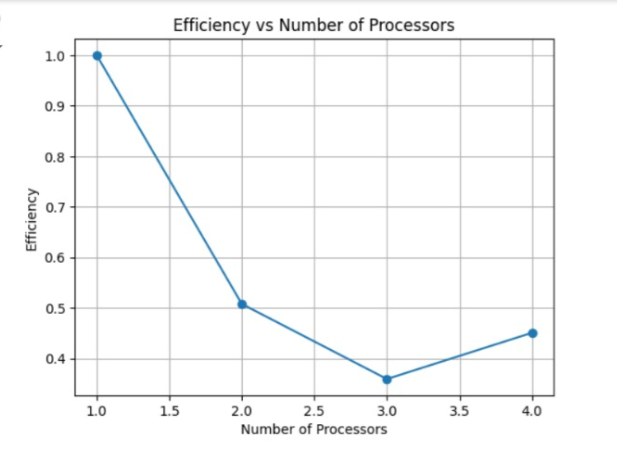
plt.grid(True)

plt.show()

A graph with a line and a point

Description automatically generated

3) Plot the efficiency of the program in addition to the speed-up.



**2)**

The pseudocode of the Bucketsort Algorithm:

Int n // array size

Int p // number of processors

Int array[] = new int[n]

Int portionSize = n /p //number of elements assigned to each processor

MPI\_Scatter(array, &portionSize, Pgroup, root=master);

Int m = p;

SmallBuckets[p] //an array of p minibuckets

For i =0 to p-1{

Minibuckets = m // m is the number of buckets where m=p

SmallBuckets[i] = new int[m]

}

For each element in local portion of array {

Identify target minibucket based on element's value range

Place element into corresponding minibucket within SmallBuckets[rank]

}

initialize Largebuckets[p]

MPI\_AllToAll (smallbuckets, Largebuckets) //each small buckets in all the processors with specific range will go to the large bucket with same range

For each bucket in LargeBuckets {

QuickSort(bucket) // Sort each bucket individually

}

MPI\_Gather(LargeBuckets, root = master);

If rank==master{

Merge

}