ASSIGNMENT

REPORT

Topic	TRAPPING RAINWATER PROBLEM	
Course Code & Title	I CS 311 – Parallel and Distributed Computing	
UG Year/Semester	III year – 5th Sem. – 2021 Batch-II	
Team Members	**Roll No.**	**Name**
	2021BCS0036	ABHISHEK HARSH
	2021BCS0064	NIRAJ PORADWAR
Date of final review	31-10-2023	

Signature			
Evaluator-1	**Evaluator-2**	**Evaluator-3**	**Faculty In-charge**
			(Dr. John) Assistant Professor/CSE

Index

Module I - Serial Version

Objective:

The objective of the rainwater trapping problem is to determine how much rainwater can be trapped between a set of vertical bars (or walls) with varying heights when it rains. The problem has asked us for the total volume of rainwater that can be held in the gaps between these bars.

Description:

Imagining a landscape consisting of bars of different heights. When it rains, the bars act as walls that can trap rainwater in the spaces between them. The rainwater trapping problem involves calculating the total volume of rainwater that can be held by these bars. Detailed description:

- Input:
The input to the problem is an array (or a sequence) of integers, where each integer represents the height of a bar. These heights can be thought of as the heights of walls in the landscape.

- Output:
The goal is to determine the total volume of rainwater that can be trapped between these bars.

Method:
To solve the problem, we need to calculate the amount of rainwater that can be trapped at each position between the bars. The trapped rainwater depends on the height of the bar at that position, as well as the highest bar to the left and the highest bar to the right of that position. The minimum of the two highest bars minus the height of the current bar gives the amount of water that can be trapped at that position.

Algorithm:
A common approach to solving the rainwater trapping problem is to use a two-pointer technique, In this algorithm we will iterate through the bars from both the left and right sides, keeping track of the highest bar on each side. As it iterates through the bars, it calculates the trapped water at each position and accumulates the total trapped water.

Explained Algorithm:

Checking if the array height has less than three elements. If it does, it's impossible to trap water, so we will return 0.

Initialize two pointers, left and right, at the beginning and the end of the height array, respectively.

Initialize two variables, left_max and right_max, to keep track of the maximum height on the left and right, initially set to 0.

Initialize a variable water to keep track of the total trapped rainwater, initially set to 0.

Use a while loop to traverse the height array from both ends (from left and right) towards each other.

Within the loop, compare the heights of the bars at left and right.

If the height at the left is less than the height at the right, update the left_max if the current height is greater. Then, calculate and accumulate the trapped water by subtracting the height of the current bar from left_max, and increment the left pointer.

Similarly, If the height at the right is less than or equal to the height at the left, update the right_max if the current height is greater. Then, calculate and accumulate the trapped water by subtracting the height of the current bar from right_max, and decrement the right pointer.

Repeat the process until the left and right pointers meet or cross each other.(stop conditoion)

Then we return the water as the total trapped rainwater.

Solution Description:

let's go through the steps of calculating the trapped rainwater for the input [2, 1, 3, 0, 1, 2, 1, 4, 5]:

Step 1: Initialize variables
- left = 0: The left pointer at the first element (height 2).
- right = 8: The right pointer at the last element (height 5).
- left_max = 0: The maximum height to the left is initially 0.
- right_max = 0: The maximum height to the right is initially 0.
- water = 0: The variable to store the total trapped rainwater is initially 0.

Step 2: Iterate through the array

- Iteration 1 (left = 0, right = 8):
  - The height at left is 2, and the height at right is 5.
  - Since 2 < 5, we update left_max to 2.
  - The trapped water at this position is calculated as left_max - height[left] = 2 - 2 = 0. So, water remains 0.
  - Increment left to 1.

- Iteration 2 (left = 1, right = 8):
  - The height at left is 1, and the height at right is 5.
  - Since 1 < 5, we update left_max to 2.
  - The trapped water at this position is calculated as left_max - height[left] = 2 - 1 = 1. So, water becomes 1.
  - Increment left to 2.

- Iteration 3 (left = 2, right = 8):
  - The height at left is 3, and the height at right is 5.
  - Since 3 < 5, we update left_max to 3.
  - The trapped water at this position is calculated as left_max - height[left] = 3 - 3 = 0. So, water remains 1.
  - Increment left to 3.

- Continue the iterations until the pointers meet or cross each other. Note that when left_max is updated, the trapped water is calculated as the difference between left_max and the current bar's height. When right_max is updated, the same logic applies to the right side.

Step 3: Total trapped rainwater

After all iterations, you sum up the trapped water calculated in each step.

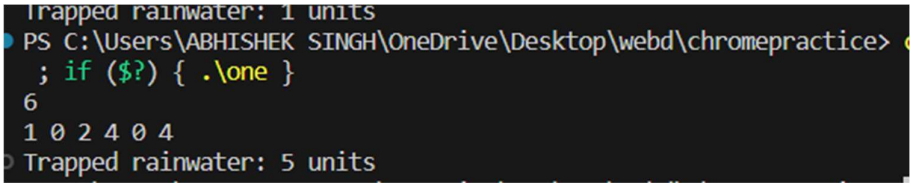
- Total trapped rainwater: 0 + 1 + 0 + 2 + 1 + 0 + 1 + 1 + 0 = 8 units.

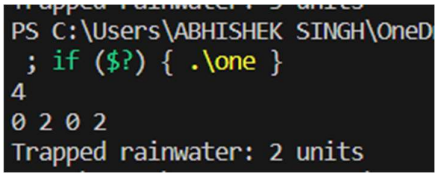
So, for the input [2, 1, 3, 0, 1, 2, 1, 4, 5], the total trapped rainwater is 8 units.

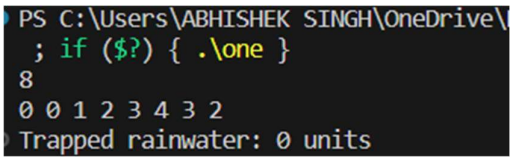
CODE IMPLEMANTATION (serial):

```c
#include <stdio.h>

int trap(int height[], int n) {
    if (n <= 2) {
        return 0;
    }

    int left = 0, right = n - 1;
    int left_max = 0, right_max = 0;
    int water = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] > left_max) {
                left_max = height[left];
            } else {
                water += left_max - height[left];
            }
            left++;
        } else {
            if (height[right] > right_max) {
                right_max = height[right];
            } else {
                water += right_max - height[right];
            }
            right--;
        }
    }

    return water;
}

int main() {
    int n;
    scanf("%d", &n);

    int height[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &height[i]);
    }

    int result = trap(height, n);
    printf("Trapped rainwater: %d units\n", result);

    return 0;
}
```

Three test cases:







TIME COMPLEXITY:

The time complexity of the algorithm used to solve the rainwater trapping problem is O(n), where 'n' represents the number of elements in the input array.

Analysing time complexity:

1. **Initialization of Variables**: Initializing the variables such as `left`, `right`, `left_max`, `right_max`, and `water` takes constant time, denoted as O(1). This step is independent of the input size.
2. **Main Loop**: The primary work occurs within the while loop, traversing the array with two pointers (`left` and `right`). This loop iterates a maximum of 'n' times, where 'n' is the number of elements in the input array. Consequently, the loop contributes O(n) to the time complexity.
3. **Operations Inside the Loop**: Inside the loop, there are simple arithmetic operations, comparisons, and updates to variables. All these operations take constant time for each iteration, represented as O(1).

The time complexity is dominated by the main loop, resulting in an overall time complexity of O(n). This indicates that the time taken to calculate the trapped rainwater is directly proportional to the size of the input array. Therefore, the algorithm's efficiency allows for fast processing, even for significantly large arrays.

Module II - Parallel Version – OpenMP

Identification of Parallelizable Blocks:

The identified parallelizable blocks for the rainwater trapping problem are:

1. *Loop Iteration*: The main loop that traverses the array with two pointers (`left` and `right`) is a potential candidate for parallelization.

2. *Conditional Variable Updates*: The conditional updates of variables like `left_max` and `right_max` within the loop should be carefully managed to prevent data races when parallelizing the loop. Synchronization mechanisms might be needed to handle shared variable updates securely in a parallel context.

Pseudocode for Parallel Version with respect to OpenMP:

function trap_parallel(height):
If the size of the 'height' array is less than 3, return 0.

Set 'left' to 0.
Set 'right' to the size of 'height' minus 1.
Set 'water' to 0.
Using OpenMP's parallel for directive and a reduction clause for 'water'.
Loop from index 1 to size(height) - 2 in parallel:
- For each index 'i', calculate 'left_max' as the maximum height from index 0 to 'i'.
- Calculate 'right_max' as the maximum height from index 'i+1' to the end.
- Calculate trapped water at index 'i' and add it to 'water':
  trapped_water = max(0, min(left_max, right_max) - height[i])
  Increment 'water' by 'trapped_water'.

Return the total accumulated 'water'.

Solution Demonstration with respect to OpenMP:

Explanation:

1. we made the `trap_parallel` function which computes the trapped rainwater using OpenMP parallelization.
2. The `#pragma omp parallel for reduction(+:water)` directive parallelizes the loop. The `reduction(+:water)` clause ensures proper accumulation of the local `water` variables from each thread into the final `water` variable.
3. The loop iterates through each element, calculating the left_max and right_max for each element and computes trapped water at each position.
4. Finally, it returns the total accumulated trapped water.

This is the demonstration of parallel version which divides the work among multiple threads for faster computation. This demonstrates how the rainwater trapping problem can be parallelized using OpenMP directives to enhance performance in a multi-threaded environment.

Iiit kottayam

OpenMP Parallel Code Implementation (C/C++):

Open omp.c ~/finalass Save

```c
#include <stdio.h>
#include <omp.h>

int trap_parallel(int height[], int size) {
    int water = 0;

    #pragma omp parallel for reduction(+:water)
    for (int i = 1; i < size - 1; i++) {
        int left_max = 0;
        int right_max = 0;

        for (int j = 0; j < i; j++) {
            left_max = (height[j] > left_max) ? height[j] : left_max;
        }

        for (int j = i + 1; j < size; j++) {
            right_max = (height[j] > right_max) ? height[j] : right_max;
        }

        int trapped_water = (left_max < right_max) ? (left_max - height[i]) : (right_max - height[i]);
        if (trapped_water > 0) {
            water += trapped_water;
        }
    }

    return water;
}

int main() {
    int num_bars;
    printf("Enter the number of bars: ");
    scanf("%d", &num_bars);

    int height[num_bars];
    printf("Enter the heights of bars:\n");
    for (int i = 0; i < num_bars; i++) {
        scanf("%d", &height[i]);
    }

    int result = trap_parallel(height, num_bars);
    printf("Total trapped rainwater: %d\n", result);

    return 0;
}
```

Sample input output

```
abhi@DESKTOP-70BSB0G:~/finalass$ g++ -o omp -fopenmp omp.c
abhi@DESKTOP-70BSB0G:~/finalass$ ./omp
Enter the number of bars: 4
Enter the heights of bars:
4 0 3 4
Total trapped rainwater: 5
```

Time Analysis for OpenMP Implementation :

The time complexity for the OpenMP implementation of the rainwater trapping problem remains the same as the sequential version: O(n).

In the OpenMP implementation:

- The parallel for-loop parallelizes the computation of trapped water among multiple threads.
- Each thread independently computes trapped water for a range of elements, reducing the final result into a shared variable using the reduction clause.

The overall time complexity is still O(n) due to the **linear relationship** between the input size and the time taken to solve the problem. The parallelization enables concurrent processing of the array, potentially speeding up the computation significantly, especially for larger arrays or on multi-core processors.

Module III - Parallel Version – MPI

Identification of Parallelizable Blocks:

The identified parallelizable blocks for the rainwater trapping problem are:

1. *Loop Iteration*: The main loop that traverses the array with two pointers (`left` and `right`) is a potential candidate for parallelization.

2. *Conditional Variable Updates*: The conditional updates of variables like `left_max` and `right_max` within the loop should be carefully managed to prevent data races when parallelizing the loop. Synchronization mechanisms might be needed to handle shared variable updates securely in a parallel context.

Pseudocode for Parallel Version with respect to MPI :

```
initialize MPI
Get total number of processes (size) and rank of each process

function trap_parallel(height, local_size, rank):
   if local_size < 3:
      return 0

   local_water = 0
   left = calculate_left_boundary(rank, local_size)
   right = calculate_right_boundary(rank, local_size)

   for i from left + 1 to right - 1:
      left_max = max(height[0:i])
      right_max = max(height[i+1:size])

      trapped_water = max(0, min(left_max, right_max) - height[i])
      local_water += trapped_water

   total_water = sum_reduce(local_water)

   if rank == 0:
      print total_water

function calculate_left_boundary(rank, local_size):
   if rank == 0:
      return 0
   else:
      return rank * local_size - 1

function calculate_right_boundary(rank, local_size):
   if rank == num_processes - 1:
      return size - 1
```

```
    else:
        return (rank + 1) * local_size

main():
    num_processes = total number of processes
    rank = rank of current process
    height = array of heights for the entire dataset

    local_size = size / num_processes
    local_height = array to store heights for each process

    scatter_data(height, local_height, local_size)  // Scatter the height array among processes

    trap_parallel(local_height, local_size, rank)  // Call parallel function for each process

    finalize MPI
```

Solution Demonstration with respect to MPI

In an MPI implementation, each MPI process handles a portion of the height array, computes the trapped water for its segment, and then gathers the results to calculate the total trapped rainwater.

Here's an overview:

1. Initialization:
   - Initialize MPI and get the number of processes (`size`) and the rank of each process.
   - Process 0 (rank 0) will hold the complete dataset.

2. Data Distribution:
   - Distribute the heights among the processes using MPI_Scatter. Each process will receive a segment of the height array.

3. Calculation on Segments:
   Each process runs the `trap_parallel` function, which calculates the trapped water for its segment of the height array.

4. Gathering Results:
   - Using MPI_Reduce or MPI_Allreduce to collect the local results calculated by each process and compute the total trapped water.

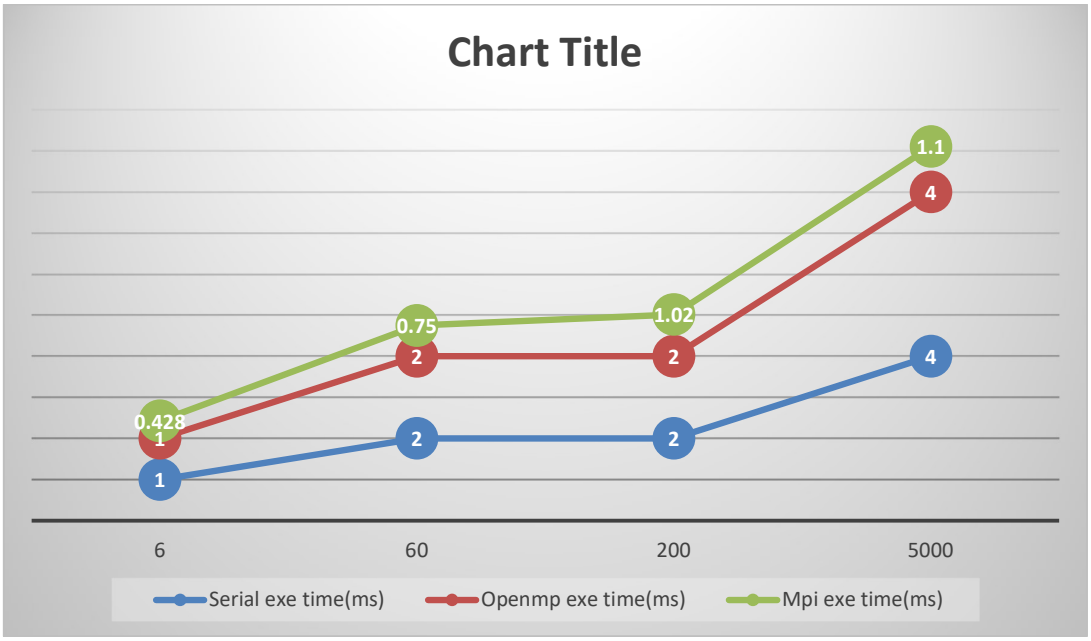
MPI Parallel Code Implementation (C/C++)

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define max(x, y) (((x) > (y)) ? (x) : (y))
#define min(x, y) (((x) < (y)) ? (x) : (y))

int maxWater(int arr[], int local_arr_size) {
   int left_max = 0, right_max = 0;
   int local_result = 0;

   for (int i = 0; i < local_arr_size; i += 4) {
      left_max = max(arr[i], arr[i + 1]);
      right_max = max(arr[i + 2], arr[i + 3]);

      int trapped_water = min(left_max, right_max) - arr[i + 1];
      local_result += max(trapped_water, 0);
   }

   return local_result;
}

int main(int argc, char *argv[]) {
   int rank, size;
   int arr[] = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1}; // Example array
   int n = sizeof(arr) / sizeof(arr[0]);
   int local_arr_size;
   int local_result = 0, total_result = 0;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   local_arr_size = (n / 4) / size * 4;
   int *local_arr = (int*)malloc(local_arr_size * sizeof(int));

   MPI_Scatter(arr, local_arr_size, MPI_INT, local_arr, local_arr_size, MPI_INT, 0,
MPI_COMM_WORLD);
   local_result = maxWater(local_arr, local_arr_size);

   // Use barrier to ensure all processes finish their local computations before reducing
   MPI_Barrier(MPI_COMM_WORLD);

   MPI_Reduce(&local_result, &total_result, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

   if (rank == 0) {
      printf("Total trapped rainwater: %d\n", total_result);
   }
   MPI_Finalize();
   return 0;
}
```

Time Analysis for MPI Implementation

Execution Time Comparison Table and Graph

Number of bars	Serial exe time(ms)	Openmp exe time(ms)	Mpi exe time(ms)
6	1	1	.428
60	2	2	0.75
200	2	2	1.02
5000	4	4	1.1



Iiit kottayam