Aerial Robotics Kharagpur Task Round Documentation
Task 1 : Optimize Me

Nikhil Giri*

***Abstract*— This Documentation gives a summary of how I approached the tasks 1, given to me during ARK Software Selection Round**

From the time when I was first given these tasks to the time i am writing this documentation I have learned and explored a lot of things. I learnt a lots of algorithms and also tried implementing them. These algorithms are essential for drones to judge and correct their path on basis of algorithms they encounter. .

In this task, We had to perform some optimization techniques on a recursive program that can reduce its run time . Though during programming we consider recursive algorithms as optimal and best possible solutions, a simple recursion implementation in cases of large data set and calculation intensive functions can increase the Run-time exponentially. In this paper we discuss some of the methods and implementations that can make programs run efficiently, which are a must for a drone as it needs to take swift decisions, so writing efficient codes are compulsion. Some of these methods and implementation are Storing values in different Data structures, Removing redundant recursive calls, Sequential Memory Access and Parallel programming

I. INTRODUCTION

The Task : Optimize me , challenges us to optimize the code to amazing levels. Basically, we have been given an base code which has some mistakes and is not efficient. Along with it we have been given a text file with hints and using these we have to give an optimized solution. Using these we have to start by Applying basic optimization algorithms such as memorization to advanced solutions like parallel programming.

II. PROBLEM STATEMENT

In this task, An unoptimized C++ code was given to us with a Markdown file stating how the code ideally is expected to work. On basis of this Markdown file we have to make sure that the originally given algorithms are correct and if not then we have to correct the original algorithm. Once that is done the massive and the main task is to optimize the given algorithms to execute entire operation as fast as possible even when the dimension of matrices used increases to 1000x1000 and more.

Now Taking about the Task details and the Original Code:

costMatrixA and *costMatrixB* are two ‘n‘ x ‘n‘ matrices given to us. Each cell in these matrices contains the cost you have to pay on landing on that cell.

You have two functions *FindMinCostA()* and *FindMaxCostB()*.

- *FindMinCostA()*‘ which returns the minimum cost of going from cell ‘i,j‘ to cell ‘n,n‘ in ‘costMatrixA‘. This cost is the sum of the costs of the cells of ‘costMatrixA‘ which will be on your path from ‘i,j‘ to ‘n,n‘.This path will be the minimum cost path out of all possible paths.
- *FindMaxCostB()* which returns the maximum cost of going from cell ‘i,j‘ to cell ‘n,n‘ in ‘costMatrixB‘.This cost is the sum of the costs of the cells of ‘costMatrixB‘ which will be on your path from ‘i,j‘ to ‘n,n‘.This path will be the miximum cost path out of all possible paths

HINT 1: For this part, you can use additional storage and think about ways to avoid repetition in recursive calls.

For the next part, we have a product matrix prductMat. prductMat[i][j] stores the value of ‘FindMinCostA(i,0)*FindMaxCostB(0,j,n) + FindMinCostA(i,1,n)*FindMaxCostB(1,j,n).... + FindMinCostA(i,n-1,n)*FindMaxCostB(n-1,j,n)‘

HINT 2: We have used the common matrix multiplication code to do this. However, this can be further optimised to maximize the chances of sequential element access.

After obtaining the product matrix, we apply a basic a filter on it. If the filter’s dimension is ‘c‘ x ‘n‘, then we replace the dot product of this filter corresponding ‘c‘ rows of ‘prductMat‘ with a single element in a new matrix whose dimension is ‘(n/c)‘ x ‘1‘.The dimension of the filter given to you will be ‘4‘ x ‘n‘

HINT 3: In this part, the same operation (dot product) is going to be repeated several times. Also, the operation here can easily be vectorised. Try to think of parallel computation based optimisations to optimise this portion.

```
1
2  #include <bits/stdc++.h>
3  #include <iostream>
4
5  using namespace std;
6
7  const int sizen = 20;
8
9  long long costMatrixA[sizen][sizen];
10 long long costMatrixB[sizen][sizen];
11
12 long long productMat[sizen][sizen];
13
14 //Simple recursion  which returns the minimum cost
       of going from i,j to n,n
15 long long FindMinCostA(int i, int j, int n)
16 {
17     //going out of bounds
18     if (i >= n)
19         return 0;
20     //going out of bounds
21     if (j >= n)
22         return 0;
23     //reaching the last cell
24     if (i == n - 1 && j == n - 1)
25         return costMatrixA[i][j];
```

```
    //going down or right
    return costMatrixA[i][j] + min(FindMinCostA(i + 1, j, n), FindMinCostA(i, j + 1, n));
}
//Simple recursion which returns the maximum cost of going from i,j to n,n
long long FindMaxCostB(int i, int j, int n)
{
    //going out of bounds
    if (i >= n)
        return 0;
    //going out of bounds
    if (j >= n)
        return 0;
    //reaching the last cell
    if (i == n - 1 && j == n - 1)
        return costMatrixB[i][j];
    //going down or right
    return costMatrixB[i][j] + max(FindMaxCostB(i + 1, j, n), FindMaxCostB(i, j + 1, n));
}

int main()
{
    int i, j, k;
    srand(time(0));
    // initialisation
    for (i = 0; i < sizen; i++)
    {
        for (j = 0; j < sizen; j++)
        {
            costMatrixA[i][j] = 1 + rand() % 10;
            costMatrixB[i][j] = 1 + rand() % 10;
            productMat[i][j] = 0;
        }
    }
    //creating productMat as explained in the beginning
    for (i = 0; i < sizen; i++)
    {
        for (j = 0; j < sizen; j++)
        {
            for (k = 0; k < sizen; k++)
                productMat[i][j] += FindMinCostA(i, k, sizen) * FindMaxCostB(k, j, sizen);
        }
    }
    //filter of size 4 x n
    long long filterArray[4][sizen];
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < sizen; j++)
            filterArray[i][j] = rand() % 2;
    }
    // matrix of dimension (sizen/c) x 1 where c = 4
    long long finalMat[sizen / 4];
    // applying the filter
    for (i = 0; i < sizen - 4; i += 4)
    {
        long long sum = 0;
        // dot product of 4xn portion of productMat
        for (j = 0; j < sizen; j++)
        {
            for (int filterRow = 0; filterRow < 4; filterRow++)
            {
                sum += productMat[i + filterRow][j];
            }
        }
        finalMat[i / 4] = sum;
    }

    return 0;
}
```

Listing 1. Original Code

For sake of simplicity we split the original code into three sections i.e.

Part 1 : Initialization of the 2 cost matrices as well as creation of functions FindMinCostA and FindMaxCostB

Part 2 : Initialization of the productMat

Part 3 : Creating and executing Filter on product Matrix

The Task is to optimize all three part and achieve as low execution time as possible. To do so we will start with basic code optimization and slowly try to implement advanced methods such as parallel computation based optimisations.

III. Related Work

During this task I learnt about Multithreading and its implementation via Pthread. I also Read many things about how memory executes a code and how sequential memory access makes a program run faster.

IV. INITIAL ATTEMPTS

First I took some time to understand the code and focused towards the recursion. Initially, I thought instead of returning values back the function why not return as soon as the function reaches(n-1,n-1) but seeing that instead of calling one function our function calls two different function and takes min or max of it i failed to think of a way to do so . Next day while searching google I found that this is already a famous recursion optimization technique called Tail recursion which uses an accumulator. But even after a lot of brainstorming i could not find a way to implement it in the given code code. So This attempt made no progress towards the goal but gave a confidence to me as i thought a famous code optimization technique all by myself. After that, based on my understanding of code and hints given in the Markdown file i thought up of my final approach.

V. Final Approach

Part1 : On careful observation of recursion i found that On calling the recursive function at (0,0) the function at some point calculate the min/max for all (i,j). So instead of just returning those values once and losing them, I stored them into two additional matrix called Minmat where minmat[i][j] represents the min cost path from (i,j) to (n-1,n-1) and the other matrix Maxmat where it similarly stores the max cost path. After that I called both the functions, FindMinCostA and FindMaxCostB at (0,0) and stored up the two matrix. Then for calculation of productmat(Part 2) instead of calling a recursive function each time i used the stored up value. This optimized the code to great extent and shaved loads of additional calculations. Furthermore, keeping in mind that the limit of the cost matrix is from 1-10 , even for n=1000 the maximum values to be handled is well within the range of int. So using long long int uses additional space and loads up the memory so I changed all of then to int and this also optimized the code to a surprising extent as it saved a lot of space in memory. Here is the final Part 1 code :

```
int costMatrixA[sizen][sizen];
int costMatrixB[sizen][sizen];
int productMat[sizen][sizen];
int minmat[sizen][sizen];
int maxmat[sizen][sizen];
```

Listing 2. Global declarations for Part 1

```
int FindMinCostA(int i, int j, int n)
{
    //going out of bounds
    if (i >= n)
        return 1000000;
    //going out of bounds
    if (j >= n)
        return 1000000;
    //reaching the last cell
    if (i == n - 1 && j == n - 1){
    int x = costMatrixA[i][j];
    minmat[i][j] = x;
    return x;
    }
    //going down or right
   int a,b;
    if(minmat[i+1][j] ==0){
    a = FindMinCostA(i + 1, j, n);
    }else{
    a = minmat[i+1][j];
    }
    if(minmat[i][j+1] ==0){
    b = FindMinCostA(i, j + 1, n);
    }else{
    b = minmat[i][j+1] ;
    }
    int x = costMatrixA[i][j] + min(a,b);
    minmat[i][j] = x;
    return x;
}


//Simple recursion which returns the maximum cost
    of going from i,j to n,n
int FindMaxCostB(int i, int j, int n)
{
    //going out of bounds
    if (i >= n)
        return 0;
    //going out of bounds
    if (j >= n)
        return 0;
    //reaching the last cell
    if (i == n - 1 && j == n - 1){
        int y= costMatrixB[i][j] ;
        maxmat[i][j] = y;
        return y;
    }

    //going down or right
    int a,b;
    if(maxmat[i+1][j] ==0){
    a = FindMaxCostB(i + 1, j, n);
    }else{
    a = maxmat[i+1][j];
    }
    if(maxmat[i][j+1] ==0){
    b = FindMaxCostB(i, j + 1, n);
    }else{
    b = maxmat[i][j+1] ;
    }
    int y= costMatrixB[i][j] + max(a,b);
    maxmat[i][j] = y;
    return y;
}
```

Listing 3. Optimized FindMinCostA and FindMaxCostB functions

Here first of all we declare all matrices in listing-2 with size of a variable named sizen which we can change and see the effect of increase or decrease in dimension on performance.

In Listing-3 we start optimizing the 2 recursive function. Lets see FindMinCostA function line by line.

In line 4-8 : we are setting value if i or j go beyond bounds we return an absurdly large value so that the min function will reject them . In the case of FindMaxCostB function we return zero so that max function will not consider them as all values are greater than 0.

In line 10-14 : we are setting up the end condition where if a path reaches the last cell i.e. (n-1,n-1) we start returning the values.

in line 16-29 : we first check the minmat which is the matrix where we save the minimum path values. If it is uninitialized then we call the recursion finction on that cell and the save the returned value of the recursive function in minmat. If it has already been found then we just return the value stored without any operations which in turn saves a lot of redundant calculations.

Similarly, we do the same steps in the FindMaxCostB function.

Part 2 : In the original code we had a nested loops for matrix multiplication where we were constantly calling the recursive function. Now the storage of the min and max values in minmat and maxmat respectively will help us in optimization. Instead of calling recursive function again and again we will hard code it to call both functions at (0,0) this will initialize all the other cells as explained in part 1. Now we will can the productmat on these two matrices and save a huge amount of resources. Further we will now use a advance optimization technique known as sequential memory access where we will transpose the matrix we transverse column wise. Though it uses some resources to transpose but with a big amount of data such as 1000x1000 it is well worth doing so. Now to further optimize it I used Parallel Programming using "PTHREAD" to under take calculation of many rows in matrix multiplication at once.

Here is the final Part 2 code :

ETA

```
int maxmat_1[sizen][sizen];                 //
    transpose of maxmat
int thread_num =1;
int task_per_thread = sizen/maxThread;

void* funct(void* arg){
    int start  = (task_per_thread-1)*thread_num;

   for (int i = start; i < (start+task_per_thread);
     i++) {
            for (int j = 0; j < sizen; j++) {
                for (int k = 0; k < sizen; k++)
                {
                    productMat[i][j] += minmat[i][k
    ] * maxmat_1[j][k];
                }
        }
   }
   thread_num += 1;

   return arg;
```

```

}
```

Listing 4. Global declarations for Part 2 and Function to execute threads

```
int main()
{
    int i, j, k;
    srand(time(0));
    // initialisation
    for (i = 0; i < sizen; i++)
    {
        for (j = 0; j < sizen; j++)
        {
            costMatrixA[i][j] = 1 + rand() % 10;
            costMatrixB[i][j] = 1 + rand() % 10;
            productMat[i][j] = 0;

        }
    }
    //creating productMat as explained in the
    beginning
    FindMinCostA(0,0,sizen);
    FindMaxCostB(0,0,sizen);

    for (i = 0; i < sizen; i++)
    {
        for (j = 0; j < sizen; j++)
        {
            maxmat_1[j][i] = maxmat[i][j];
        }
    }

    pthread_t threads[maxThread];

    // Creating some threads, each evaluating its
    own part
    for (int i = 0; i < maxThread; i++) {
        int* p;
        pthread_create(&threads[i], NULL, funct, (
    void*)(p));
    }

    // joining and waiting for all threads to
    complete
    for (int i = 0; i < maxThread; i++)
        pthread_join(threads[i], NULL);

//*************PART 3 HERE****************
}
```

Listing 5. Initialization of all matrices and optimized Part 2

Now, In listing 4 we set some global variables which we can change depending on the system and declare the transpose matrix maxmat-1 as we transverse it column wise and define a function called funct which takes a thread and and executes its section of matrix multiplication. The number of row a single thread handles is defined by the global variables.

In Listing-5 we start optimizing the the matrix multiplication. Lets see the code line by line.

In line 6-15 : we are using rand() function to generate a number between (1,10) and then initailizing the costMatrixA and costMatrixB using these values. Also we are creating an empty (all cell equal to 0) Productmat.

In line 17- 26 : we are calling the recursive functions at (0,0) and storing the values in minmat and maxmat as discussed in part 1. Then for sequential element access we are using two nested loops to transpose the maxmat into maxmat-1

In line 28-38 : This code is a basic pthread application where we create a number of threads determined by global variable. Then using pthread-create we call the funct function on each thread. Then using pthread-join we wait for all the thread to complete their tasks.

Part 3 : In part 3, Markdown file directs us to create a we apply a basic a filter on it. If the filter's dimension is 'c' x 'n', then we replace the dot product of this filter corresponding 'c' rows of 'productMat' with a single element in a new matrix whose dimension is '(n/c)' x '1'. To do this we create a filter Array and initialize the elements randomly with 1 or 2. Then we take repeated dot product with the productmat and store the values. Now, from optimization perspective I tried to add pthread parallel programming to this part by using new threads via creating new function separately for dot product and even tried to merge the two function but neither of them gave significant reduction in time. Rather i saw that the ones with parallel dot product took more time than the one where i have simple implementation. Since i have to submit the code which works the best here is just a simple implementation of the part 3.

Here is the final Part 2 code :

```
int main()
{
    //***************PART 2 HERE******************
    int filterArray[4][sizen];

    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < sizen; j++)
            filterArray[i][j] = rand() % 2;
    }

    // matrix of dimension (sizen/c) x 1 where c =
    4
   int finalMat[sizen / 4];
    // applying the filter
    for (i = 0; i < sizen - 4; i += 4)
    {
        int sum = 0;
        // dot product of 4xn portion of productMat
        for (j = 0; j < sizen; j++)
        {
            for (int filterRow = 0; filterRow < 4;
    filterRow++)
            {
                sum += productMat[i + filterRow][j]
     * filterArray[filterRow][j];
            }
        }
        finalMat[i / 4] = sum;

    }
}
```

Listing 6. Simple Basic Filter Application for Part 3

In listing 6, Lets see the code line by line:

In line 4-10: We declare the filterMatrix by the name filterArray and then initialize each element randomly as 1 or 2.

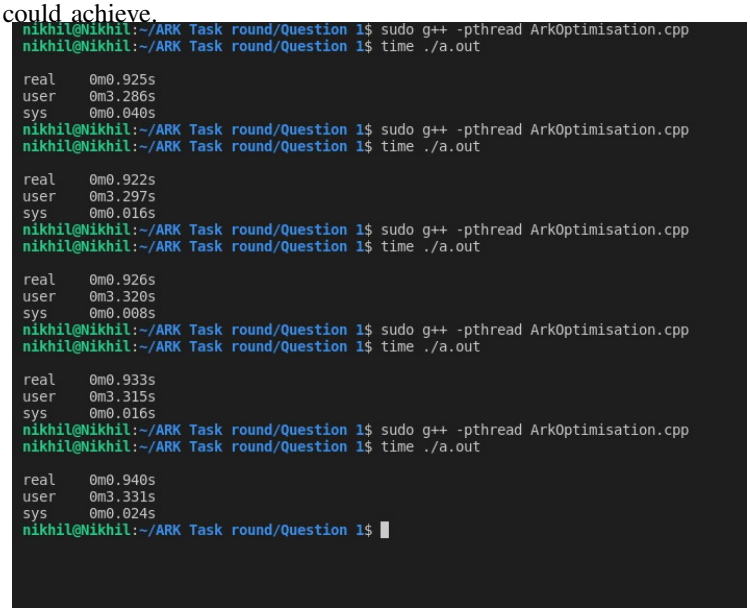
In line 13-26: We are taking repeated dot product on productMat sections of 4 rows and then storing the values in finalMat.

And Thats it.This final code for Question 1

VI. Results and Observation

After running the code multiple times i found that the result varies system to system and also state of the system used. I ran the codes on virtual machine which is limited only to half of the RAM available and on top of that if application such as Google Chrome, Microsoft Teams run in the background further slows down the compiler.

So i ran the program after restarting the system and clearing all the cache and this was the most optimized result i could achieve.



So when there is no unnecessary background application running the average time taken for n(i.e. sizen) = 1000 is around 0.92 second whereas in normal condition this average increases to 1.1 seconds .But again these where the output in my Laptop and virtual machine. On different system the program might run faster or slower.

VII. Future Work

As I said That i implemented many different variations and decided the best one via just few test runs. But Later I would like to tabulate the data and then find the best code. I would also try implement cache Hit and Miss concept and some other advanced Optimizing solutions.

CONCLUSION

The average time for code execution is 0.92 seconds. This task makes it clear that by taking precautions and care during coding a recursive function as well as implementing certain algorithms and technique, can optimize the code and reduce the run-time in a way that seems like impossible at first.

References

[1] Ulrich Drepper, , ”What Every Programmer Should Know About Memory”, 2007.

[2] Hongqing Liu, Stacy Weng and Wei sun,”Introduction of Cache Memory”,2001

[3] Geeks For Geeks,”Multiplication of Matrix using threads”,2020