Task 1: Optimise Me

Ashwani Kumar Kamal

***Abstract*— The purpose of this document is to point out some basic optimization techniques that can make a recursive program reduce its run time massively. Though in many algorithms, recursion seems an obvious solution, it can however make a program much memory intensive and expensive in terms of run time. To counter this complication, we can use additional data structures (making the program space expensive) and reducing redundant recursive calls, thereby improving the efficiency of the program by a lot. Very similar is the case with the well known matrix multiplication. Exploiting how the processor works during execution of a program and taking advantage of cache built the algorithmic efficiency of the same program improves many fold.**

I. INTRODUCTION

The basic outline of the problem was to optimise the given .cpp file and bring down the run time. Some hints were given to start off the work. Based upon the hints I implemented some techniques namely, memoization, vectorization, cache hit and miss (locality of references), implementing parallelism on independent loop segments (in vector and matrix multiplication). I even tried implementing a recursive approach to matrix multiplication (Stressan's algorithm), however it added some latency due to stock recursive calls, hence had to reject it from the final solution.

II. PROBLEM STATEMENT

In the first task you have to use optimisation techniques to reduce the running time of the code provided to you. First download the zip file here. Before proceeding further, read the Instructions.md file which provides a lengthier explanation of the problem statement along with hints. Markdown(md) files can be viewed on Typora or on your code editor. A bash script to time your code has also been provided to you along with instructions on how to run it. Optimisation techniques can range from basic time complexity reduction to more advanced optimisations using parallel programming, locality of references and the like. Initially the code provided to you will not work for larger values of n, we will slide n from 4 to 1000 to test your code. Code working on large values of n in less time will fetch maximum points. Start with basic time complexity reduction and ease into more advanced methods to see how your running time reduces and have fun trying to achieve the minimum running time possible!

$costMatrixA$ and $costMatrixB$ are two $n * n$ matrices given to you. 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 minimum cost path out of all possible paths. n will be always be divisible by 4.
These functions will work for small values of n like 3 or 4. However for higher values of n like 100 they will fail. Start by optimising these functions to work for n ¡= 1000.
For the next part, we have a product matrix $prductMat$. Where $productMat[i][j]$ stores the value of,

$$\sum_{k=0}^{n-1} FindMinCostA(i, k, n) * FindMaxCostB(k, j, n)$$

After obtaining the product matrix, we apply a basic a filter on it. If the filter's dimension is $c * n$, 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 $(n/c) * 1$.
For example:

The product matrix is [[1,2,3,4],[2,3,4,5]]
The filter is [[1,2,1,2],[2,1,2,1]]
Then, after applying the filter, the 1 x 1 matrix will be
[1.1 + 2.2 + 3.1 + 4.2 + 2.2 + 3.1 + 4.2 + 5.1] = [36]
The dimension of the filter given to you will be $4 * n$.

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

III. RELATED WORK

- Memoization [1]
- Cache hit and misses [2]
- Parallel computing and OpenMP[3]

IV. INITIAL ATTEMPTS

After going through a dry run of the $FindMinCostA()$ function, I could see there were many redundant calls being made. For example $FindMinCostA(i, i, n)$ where $i > 0$ gets called many times. Seeing this pattern and the paradigm that we can move only right and bottom, it is easy to deduce

that we will require the value of $FindMinCostA(i, j, n)$ multiple times where $i > 0$ and $j > 0$. So, an obvious solution that pops in mind is we backtrack our way from cell n, n to i, j and in the process store the values somewhere for calculating for cell x, y where $x < i$ or $y < j$.

V. Final Approach

A. Fixing the original code

1) Function boundary conditions: The cost matrices can contain integers ranging from 1 till 10. Checking how $FindMinCostA()$ behaves on the right and bottom boundary of the matrix ($n-1, i$ and $i, n-1$ cells) reveals that function doesn't return correct results. For cell $(i, n-1)$ it may be noted that value of $FindMinCostA(i, n)$ (rightwards outside the matrix) must be higher as compared to $FinMinCost(i+1, n-1)$ for it to avoid the matrix boundary. Considering the worst case scenerio that the cells $(i+1, n-1)$, $(i+2, n-1)$, ... all conatin the largest cost possible (10) in costMatrixA, we need to set $FindMinCostA(i, n)$ as greater than the largest cost possible for cells $(i+1, n-1)$, $(i+2, n-1)$, ... summed up (as we are tracking the total cost). One such upper bound can be $10 * n + 1$.
So return $10 * n + 1$ for $FindMinCostA(i, j, n)$ where $j >= n$.
Similar is the case with bottom boundary. $FindMaxCostB()$ is perfect already as ir returns 0 for boundary (lower than the lowest possible value of $costMatrixB[i][j]$).

2) Applying Filter on productMat: The original code doesn't seem to use the $filterMat$ at all. It only iterates through the cells in vector form.

B. Memoization

The basic approach towards memoization is as follows:

```
#define MAX 100

int memo[MAX] = {0};
int memoisedRecur(params)
{
    if(baseCase) return finalResult
    if(memo[params] == 0)
    {
        // Actual computation part
        memo[params] = memoisedRecur(nextParams);
        return memo[params];
    }
    return memo[params];
}
```

Implementing this simple data structure (or a lookup table) allows us to avoid all those unnecessary recursive calls. Once a recursive call returns, the value is stored in memo for all sorts of future calls. Similar to this, I've added two data structures viz. $resultA$ and $resultB$, dimensions being same as cost matrices. The idea is that $resultA[i][j]$ will contain the value of $FindMinCostA(i, j, n)$. Moreover the iterating has been done in reverse order (starting from n, n going till $0, 0$) for storing these values. Reason being that the paradigm only allows us to go either right or bottom, so for any cell i, j we will require the values of $i+1, j$ and $i, j+1$ before us to compare and return as required. Iterating backwards allows this to happen efficiently.
This technique alone produces massive improvements in run time of the program.
Example: for n = 8 the original .cpp takes around 0.1 seconds, for n = 12 it takes 1.6 seconds and barely runs for n = 16.
After memoization for n = 8 takes 0.08 seconds, n = 12 takes 0.08 seconds, n = 16 takes 0.08 seconds, n = 1000 takes 6 seconds (already a big victory)

C. Cache hit and misses

Programs tend to use data and instructions with addresses near or equal to those they have used recently
Consider the conventional matrix multiplication:

```
void naiveMultiplication(int A[MAX][MAX], int B[MAX
    ][MAX], int C[MAX][MAX], int n)
{
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
          for (int k = 0; k < n; k++)
              C[i][j] = C[i][j] + A[i][k]*B[k][j];
}

void cacheMultiplication(int A[MAX][MAX], int B[MAX
    ][MAX], int C[MAX][MAX], int n)
{
    for (int i = 0; i < n; i++)
      for (int k = 0; j < n; j++)
          for (int j = 0; k < n; k++)
              C[i][j] = C[i][j] + A[i][k]*B[k][j];
}
```

For the first function on each iteration, the value of k is changing increasingly. This means that when running the innermost loop, each iteration of the loop is likely to have a cache miss when loading the value of B[k][j]. The reason for this is that because the matrix is stored in row-major order, each time you increment k, you're skipping over an entire row of the matrix and jumping much further into memory, possibly far past the values you've cached. However for the second variant, there seems to be no cache misses at all. And since the product C does not depend on the order of loops, the second variant is comparatively faster than first.

D. Vectorization and parallel computing

Conventionally the program runs only in a single thread. However since we know that some parts of the program (in a nested loop) are completely independent of each other, we can make them run on separate threads and exploit the fact of having multiple processing cores. The method does not provide an upper hand algorithmic-ally but does speeds up the process 4 times (if there are 4 cores). So at the same time 4 different sections of the loop can execute simultaneously saving a lot of time. I used Open-MP library (omp.h header) to achieve the same in matrix multiplication and filter application part.

VI. Results and Observation

Compare your results with all the available algorithms which you may have used to tackle the PS. If possible, present your and their results in tabular / graphical format.

Explain the trend of results in details. Mention the drawbacks (if any) of your algo compared to other algo and the reason of picking up the approach over the other if you have implemented any algo over the other.

VII. Future Work

Parallel algorithm for dense matrix multiplication, Cannon's algorithm for matrix multiplication

CONCLUSION

The final time averages around 1.1 seconds for n = 1000

References

[1] Striver "Memoization (1D, 2D and 3D)", Geeksforgeeks, 2021

[2] Laura Toma, "Intro to Parallel Programming with OpenMP", Bowdoin, 2017

[3] Akbar B, "OpenMP — Hello World program", GeeksforGeeks, 2019

[4] StackOverflow thread "Why does the order of loops in a matrix multiply algorithm affect performance", 2011