Optimize Me

Anuj Kakde (20CS30005)

Abstract— **The objective of this task is basically to optimize the given program by following the instructions given in the "instructions.md" file.**

I tried to optimize the recursive functions by using the memoization technique in which we store and use the already found out values by the recursive functions. But then I found out that during multiplication of product matrix, we were calling the recursive function a lot of times.

So in first part I replaced the recursive functions FindMinCostA and FindMaxCostB with two for loops which calculates the cost to go from i,j to n,n in the i,j cell in the matrix. This will save the time because before the recursive functions were being called many times during matrix multiplication, whereas now we can directly access what we need from the calculated matrix.

Next we have to multiply the new found matrices costMatrixA and costMatrixB. I have used two different techniques for this, Normal Optimized Algorithm Multi-threading Algorithm.

We can observe that cache friendly approach of normal matrix multiplication program runs faster than the naive implementation. Also the wall clock time (real time) required by the multi-threading algorithm is significantly less than the normal method.

Next we have a filter array of size 4 x n each element of which is either 0 or 1. To apply the filter on the product matrix,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 bottleneck of this program is the matrix multiplication part. So any optimizations done to the filter part will not improve the overall performance of the program.

Optimizations are an important part of Ariel robotics as the time required by the drone to process anything should be as less as possible.

I. INTRODUCTION

In this task we are given a incomplete code which we have complete and optimize according to the given instructions. Initially we are given two matrices which are basically just nodes containing given weights and we need to optimise the function to find the shortest path to reach from any given cell to the (n,n) cell. Then we have to compute the product matrix. Finally we are supposed to apply a filter to the product matrix.

II. PROBLEM STATEMENT

The objective of this task is basically to optimize the given program by following the instructions given in the "instructions.md" file. Initially the provided code will not work for larger values of n. The code should run efficiently for values of n up to 1000. Code working on large values of n in less time will fetch maximum points. n will always be divisible by 4.

Optimisation techniques can range from basic time complexity reduction to more advanced optimisations using parallel programming, locality of references and the like.

We have two functions FindMinCostA() and FindMaxCostB() given to us.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.

Initially these functions will work for small values of n like 3 or 4. However for higher values of n like 100 they will fail. We need to start by optimising these functions to work for n ¡= 1000.

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)
```

Initailly the code has 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.

Marks will be given the basis of the extent to which we succeed at optimising the code. We can use the time command to measure the running time of our code in Linux/MacOS. We are also given instructions for getting the program's running time on Windows

III. RELATED WORK

For this task I learnt the concepts like memoization which could be implemented to improve the recursive algorithm.

For the second part of the task I had to understand the concept of parallel computations and cache memory. In particular I have used the technique of multi-threading to improve the performance of the code. Also writing a "cache-friendly" code boosts the performance of the program.

IV. Initial Attempts

Initially, after reading the problem statement I thought of using the Djkstra's algorithm of path finding to find the shortest path based on costs, but after reading the problem again and confirming with the mentor I got to know that we just have to move down or left. The Djkstra's algorithm would just complicate things here.

Then I tried to optimize the recursive functions by using the memoization technique in which we store and use the already found out values by the recursive functions. But then I found out that during multiplication of product matrix, we were calling the recursive function a lot of times. So I decided to change the approach and instead first calculate a matrix which will store the cost of going from i,j to n,n in the i,j cell.

V. Final Approach

In the main function, we first check if sizen is divisible by 4 and if not then we return and terminate the program. Then we initialize the matrices costMatrixA and costMatrixB. The elements of both these matrices are between 1 to 10 inclusive. This is achieved by using the rand() function.

As mentioned in the previous section, I have replaced the recursive functions FindMinCostA and FindMaxCostB with two for loops which calculates the cost to go from i,j to n,n in the i,j cell in the matrix. This will save the time as before, the recursive functions were being called many times during matrix multiplication, whereas now we can directly access what we need from the calculated matrix. I have stored the said matrix directly in the costMatrixA and costMatrixB instead of declaring new matrices as after this we don't need the individual cost of cells.

In the first for loop, we calculate result only for last row and last column as it can be directly found out by taking the sum of individual costs from that cell to the last cell. This was possible as for this case we have only one possible path.

In the next for loop we find the result for the remaining cells. As we know that we can go only right or down from any cell (say i,j), if we know the minimum/maximum cost of going from (i+1,j) to (n,n) and (i,j+1) to (n,n) then we can also find the minimum and maximum cost to go from (i,j) to (n,n).

Next we have to multiply the new found matrices costMatrixA and costMatrixB. I have used two different techniques for this.

A. *Normal Optimized Algorithm*

In this method I have used three simple nested for loops to calculate the product matrix. The optimization I did here was that instead of the loops being in order (i,j,k) I rearranged the loops to (i,k,j). After doing this the code became faster than what it was before and gave results with same accuracy.

This is due to the way in which cache memory is handled by the CPU. The cache memory is limited but is faster. Thus if we are able to write a code in which the number of times the contents of cache memory needs to change is less, it would automatically speed up the program.

The code used is-

```
for (i = 0; i < sizen; i++)
{
  for (k = 0; k < sizen; k++)
  {
    for (j = 0; j < sizen; j++)
      productMat[i][j] +=
      costMatrixA[i][k]*costMatrixB[k][j];
  }
}
```

B. *Multi-threading Algorithm*

This part uses multi-threading along with the normal cache-friendly multiplication algorithm. Multi-threading is a CPU feature that allows two or more instruction threads to execute independently while sharing the same process resources. This means multiple concurrent tasks can be performed within a single process. A thread is a path of execution within a process. This enables effective utilization of multiprocessor system since more than one processors are working at a given instant of time.

I have used 4 threads at once since that is the standard number of cores a normal computer today has. A single processor computer can also run this program but the speed boost will not be the same. A part of code is given below-

```
/* declaring four threads*/
pthread_t threads[MAX_THREAD];

/* Creating four threads,
each evaluating its own part*/
for (int i = 0; i < MAX_THREAD; i++)
{
    pthread_create(&threads[i],
        NULL, multi, NULL);
}

/* joining and waiting for all
threads to complete*/
for (int i = 0; i < MAX_THREAD; i++)
{
pthread_join(threads[i], NULL);
}
```

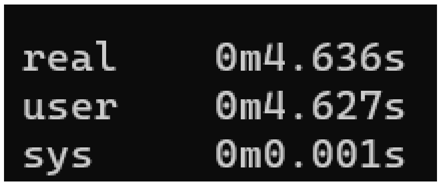
In the code I have declared an array of threads and created them using a for loop. Each thread executes one-fourth part of matrix multiplication by running the multi function. Then by using the pthread_join function I combined all the threads to the main thread.

Next we have a filter array of size 4 x n each element of which is either 0 or 1. To apply the filter on the product matrix,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'.

I have implemented the naive version of applying the filter due to two factors. First is that even after applying vectorization technique, the running time did not change. This can be due to the fact that the new compilers automatically apply this technique to improve performance. Second thing is that after playing around with the code I found the the matrix multiplication part is the bottleneck of this code and thus any optimizations to the process of applying filter would not affect the overall performance of the code.

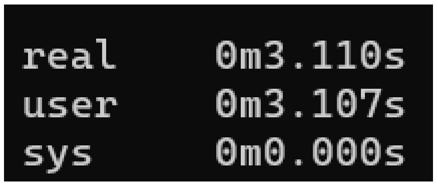
VI. Results and Observation

I have made two versions of the solution for this task. The only change in both the version is how the matrix are being multiplied, After running the codes and collecting the data the following observations are made.



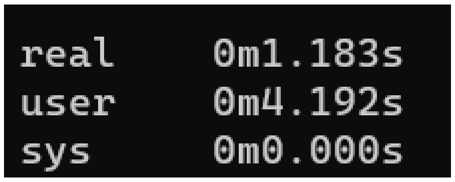
```
real    0m4.636s
user    0m4.627s
sys     0m0.001s
```

Fig. 1. Time taken by Normal Matrix Multiplication in order of ijk



```
real    0m3.110s
user    0m3.107s
sys     0m0.000s
```

Fig. 2. Time taken by Normal Matrix Multiplication in order of ikj



```
real    0m1.183s
user    0m4.192s
sys     0m0.000s
```

Fig. 3. Time taken by Matrix Multiplication using Multi-threading

Thus we can see that cache friendly normal matrix multiplication program runs faster than the naive implementation. Also the wall clock time (real time) required by the multi-threading algorithm is significantly less than the normal method.

VII. Future Work

The bottleneck of this program is the matrix multiplication part. So any optimizations done to it will improve the overall performance of the program. We can apply some more complex algorithms like Strassen's algorithm of matrix multiplication to speed up the process further.

Conclusion

This task enabled me to learn some good optimization techniques like memoization, parallel computation, vectorization etc. It was a nice experience to be able to implement the techniques myself.

Optimizations are an important part of Ariel robotics as the time required by the drone to process anything should be as less as possible. Any delay in processing information while flying can lead to disaster.

References

[1] https://cs.brown.edu/courses/cs033/lecture/18cacheX.pdf
[2] https://www.geeksforgeeks.org/multiplication-of-matrix-using-threads/
[3] https://chryswoods.com/vector_c%2B%2B/