

# Aerial Robotics Kharagpur Documentation Template

Ashwary Anand, Other Author/ Contributor Names in Order of their contribution to the project \*

graphics

**Abstract**—Should be half column long. Should be clear enough to explain your whole documentation. Similar to a TL;DR

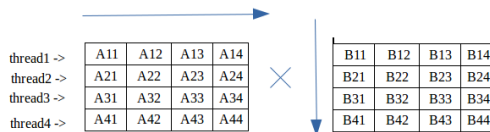
## I. INTRODUCTION

The First problem statement asks us to use optimisation techniques to reduce the running time of the code . A bash script to time your code has been provided 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 does not work for larger values of n.

## II. PROBLEM STATEMENT

**This should cover one full column of page.**

Explain in details what your problem statement was with all the necessary images and equations required.



## III. RELATED WORK

I have made changes in the original code and developed 8 new codes with the optimisation for different parts.

### • ArkOptimisation.cpp

Original cpp file. Time execution of almost 9 seconds

```
g++ -std=c++11 -O3 ArkOptimisation.cpp -o ArkOptimisation.exe
g++ -std=c++11 -O3 ArkOptimisation.exe -F C:\Users\anand\Documents\timeit\timeit.bat
Execution Time (seconds): 8.7147622
Done
```

### • arkoptimisation\_strassen.cpp

- Used Instantaneous storage of matrix values for part 1.
- Used Strassen Algorithm for part 2.
- Used normal summation of all elements and then subtracted 1 from those which had value 0.

\*Write anyone who might have helped you accomplish this eg any senior or someone

### • arkoptimisation.cpp

Using Loop Nest Optimisation for part 2 of the problem.

### • arkoptimisationpart2\_try2.cpp

Using pthread for part 2 of the problem

### • dijkstra\_2.cpp

Using pthread for part 2 of the problem and dijkstras for part 1 of the problem

### • dijkstra\_4.cpp

Using pthread for part 2 of the problem and dijkstras for part 1 of the problem and a variant of dijkstras for Maxcostmat

### • next\_1\_ark.cpp

Using blocking for part 2 of the problem and simultaneous storage for part 1

### • next\_2\_ark.cpp

Using blocking for part 2 of the problem and dijkstras for part 1 of the problem and a variant of dijkstras for Maxcostmat

### • ARK\_PART1.cpp

Using loop swapping for part 2 of the problem and simultaneous storage for part 1

## IV. INITIAL ATTEMPTS

### PART-1

- storing the values of the MincostMat and Maxcost-Mat while calling the recursive function so that the same value does not get computed more than once in the recursion.
- used dijkstras algorithm to compute the shortest distance of each cell from the last cell.

– Description of Algorithm The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

– Time-Complexity The complexity bound depends mainly on the data structure used to represent the set Q.

\* data structure used is array or linked-list  
 $\Theta(|E| + |V|^2) = \Theta(|V|^2)$

\* data structure used is adjacency lists with a self-balancing binary search tree.  
 $\Theta((|E| + |V|)\log(|V|))$

### PART-2

## STRASSEN ALGORITHM:

### DESCRIPTION

Time complexity of matrix multiplication is  $O(n^3)$  using normal matrix multiplication. And Strassen algorithm improves it and its time complexity is  $O(n^{2.8074})$

- It uses divide and conquer strategy, and thus, divides the square matrix of size  $n$  to  $n/2$ .
- It reduces the 8 recursive calls to 7
  - $p=(a_{11}+a_{22})*(b_{11}+b_{22});$
  - $q=(a_{21}+a_{22})*b_{11};$
  - $r=a_{11}*(b_{12}-b_{22});$
  - $s=a_{22}*(b_{21}-b_{11});$
  - $t=(a_{11}+a_{12})*b_{22};$
  - $u=(a_{11}-a_{21})*(b_{11}+b_{12});$
  - $v=(a_{12}-a_{22})*(b_{21}+b_{22});$



Execution time for  $\text{size} = 64$  : 0.1291577

Filename : arkoptimisation\_strassen.cpp

### LOOP NEST OPTIMISATION AND LOOP TILING :

#### DESCRIPTION

Loop nest optimization (LNO) is an optimization technique that applies a set of loop transformations for the purpose of locality optimization or parallelization or another loop overhead reduction of the loop nests. One classical usage is to reduce memory access latency or the cache bandwidth necessary due to cache reuse for some common linear algebra algorithms. Loop tiling partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused. The partitioning of loop iteration space leads to partitioning of a large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements.

#### IMPLEMENTATION

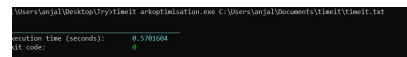
```
for (i = 0; i < N; i += 2)
{
    for (j = 0; j < N; j += 2)
    {
        acc00 = acc01 = acc10 = acc11 = 0;
        for (k = 0; k < N; k++)
        {
            acc00 += B[k][j + 0] * A[i + 0][k];
            acc01 += B[k][j + 1] * A[i + 0][k];
            acc10 += B[k][j + 0] * A[i + 1][k];
            acc11 += B[k][j + 1] * A[i + 1][k];
        }
        C[i + 0][j + 0] = acc00;
```

```
C[i + 0][j + 1] = acc01;
C[i + 1][j + 0] = acc10;
C[i + 1][j + 1] = acc11;
}
```

For part 2, while matrix multiplication, values loaded must be reused at least twice. This code has had both the  $i$  and  $j$  iterations blocked by a factor of two and had both the resulting two-iteration inner loops completely unrolled.

#### DRAWBACK

The code does not use the cache very well. During the calculation of a horizontal stripe of product matrix results, one horizontal stripe of `MincostMat` is loaded, and the entire matrix `MaxcostMat` is loaded. For the entire calculation, `productmat` is stored once (that's good), `MincostMat` is loaded into the cache once (assuming a stripe of `MincostMat` fits in the cache with a stripe of `MaxcostMat`), but `MaxcostMat` is loaded  $N/\text{ib}$  times



### SWAPPING THE LOOP ,MISS RATE ANALYSIS

The three loops in iterative matrix multiplication can be arbitrarily swapped with each other without an effect on correctness or asymptotic running time. However, the order can have a considerable impact on practical performance due to the memory access patterns and cache use of the algorithm. which order is best also depends on whether the matrices are stored in row-major order, column-major order, or a mix of both. For example:

- Stepping through columns in one row:

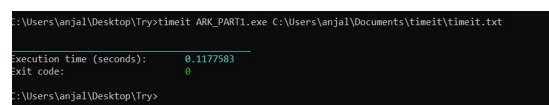
```
for(int i=0;i<n;i++)
sum+=a[0][i]
```

  - accesses successive elements
  - if block size (B)  $\geq 4$  bytes, exploit spatial locality
  - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:

```
for(int i=0;i<n;i++)
sum+=a[i][0]
```

  - accesses distant elements
  - no spatial locality!
  - compulsory miss rate = 1 (i.e. 100)

Time execution for  $\text{size} = 4$  for `kij` loop.



The first execution in the image below shows the time execution for `ijk` loop whereas the next one shows the time execution for `kij` loop for  $\text{size} = 500$ . Misses per

inner-loop iteration is 0.25 for Matrix A ,0.25 for Matrix B and 1 for Matrix c in case of ijk loop. Misses per inner-loop iteration is 0.0 for Matrix A ,0.25 for Matrix B and 0.25 for Matrix c in case of kij loop.

```
C:\Users\anjali\Desktop\Try>timeit ARK_PART1.exe C:\Users\anjali\Documents\timeit\timeit.txt
Execution time (seconds): 0.6354209
Exit code: 0

C:\Users\anjali\Desktop\Try>g++ ARK_PART1.cpp -o ARK_PART1.exe

C:\Users\anjali\Desktop\Try>timeit ARK_PART1.exe C:\Users\anjali\Documents\timeit\timeit.txt
Execution time (seconds): 0.5907490
Exit code: 0
```

## BLOCKED MATRIX MULTIPLY

A sub-block within the matrix of size `bsize` is taken. Sub-blocks (i.e.,  $A_{xy}$ ) can be treated just like scalars.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

```
int jj, kk, bsize=4;
long long sum;
for (jj = 0; jj < sizen; jj += bsize)
{
    for (i = 0; i < sizen; i++)
        for (j = jj; j < min(jj + bsize, sizen); j++)
            productMat[i][j] = 0;
    for (kk = 0; kk < sizen; kk += bsize)
    {
        for (i = 0; i < sizen; i++)
        {
            for (j = jj; j < min(jj + bsize, sizen); j++)
            {
                sum = 0;
                for (k = kk; k < min(kk + bsize, sizen); k++)
                {
                    sum += MincostMat[i][k] * MaxcostMat[k][j];
                }
                productMat[i][j] += sum;
            }
        }
    }
}
```

```
C:\Users\anjali\Desktop\Try>timeit next_1_ark.exe C:\Users\anjali\Documents\timeit\timeit.txt
Execution time (seconds): 0.6464092
Exit code: 0

C:\Users\anjali\Desktop\Try>g++ next_1_ark.cpp -o next_1_ark.exe

C:\Users\anjali\Desktop\Try>timeit next_1_ark.exe C:\Users\anjali\Documents\timeit\timeit.txt
Execution time (seconds): 0.8569422
Exit code: 0
```

In the above terminal results the first time execution shows that for block size =25 which is approximately 0.6 seconds for `sizen=500` and for block size=4 which is approximately 0.8 seconds for the same `sizen` value.

## ANALYSIS

- Innermost loop pair multiplies a 1 X `bsize` sliver of A by a `bsize` X `bsize` block of B and accumulates into 1 X `bsize` sliver of C.
- Loop over i steps through n row slivers of A and C, using same B.
- Use a block size smaller than the size of the CPU cache.

- The row sliver and the block in B are reused many times in a row.
- They are in cache after the first time they are used.
- Then go on to another small block, get it in the cache.
- If you do it in the right order, you multiply all the horizontal slivers in A times one block in B, before going on to another block in B.

## V. FINAL APPROACH

### MULTI-THREADING OR PARALLEL PROGRAMMING

Multi-threading can be done to improve the performance even more. In multi-threading, instead of utilizing a single core of your processor, we utilize all or more core to solve the problem. We create different threads, each thread evaluating some part of matrix multiplication. Depending upon the number of cores your processor has, you can create the number of threads required. Although one can create as many threads as needed, a better way is to create each thread for one core. Using

`pthread_exit()`

we return computed value from each thread which is collected by

`pthread_join()`

Inbuilt library used : `pthread.h`

time of execution for `n=500` : 0.1871937

filename : `arkoptimisationpart2.try2.cpp`

```
C:\Users\anjali\Desktop\Try>timeit arkoptimisationpart2_try2.exe C:\Users\anjali\Documents\timeit\timeit.txt
Execution time (seconds): 0.1871937
Exit code: 0
```

## 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

Write about the problems in your algorithm / approach and limitations in testing (if any due to hardware or otherwise) and how to tackle them and any future work which can be done to improve the results further.

## CONCLUSION

### Concluding Observations:

- Programmer can optimize for cache performance.
  - through the organisation of data structures
  - increasing the accessibility of data items
  - \* Nested loop structure

**\* Blocking**

- **All systems favor “cache friendly code”**
  - **Use small strides (spatial locality)**
  - **small temporal locality**

**REFERENCES**

- [1] **Geeks for Geeks,Wikipedia,Portland State University lectures**
- [2] **Studymite.com**
- [3] **Author Names, "Paper Name", Conference / Journal where the paper was published , Year of Publication**