# MA513

# Parallel Computing

# Assignment-1 Report

Paranjay Bagga
160101050

---

## *Problem Statement :-*

To understand the effect of cache optimizations on the time taken by various sequential programs for Matrix Multiplication.

## *System Specifications :-*

All the experiments have been performed on **Dell Inspiron 5559** laptop with **Intel i5-6200U dual core processor** and **8 GB RAM**. Each core has **2 threads** with the following specifications :-

1. **Clock Speed** :
    a. Max : 2800 MHz
    b. Min : 400 MHz
2. **Memory Size** : 8054440 KB = 7865 MB
3. **Cache Size** :
    a. L1d : 32 KB
    b. L1i : 32 KB
    c. L2 : 256 KB
    d. L3 : 3072 KB

**LineSize** of each cache is **64**.

## ***Theoretical Computations involving Matrix Multiplication :-***

*Assumptions* :

- Both the matrices are of size NxN.
- Clock cycles required for addition of 2 integers : **1**
- Clock cycles required for multiplication of 2 integers : **3**

*Calculations* :

- For computing each element in the resulting NxN matrix, total multiplications = **N** & total additions = **N-1**.
- Thus, the total no. of clock cycles required for computing each resultant matrix element = **(N)*(3) + (N-1)(1) = 4*N - 1**.
- Hence, total clock cycles req. for computing whole resultant matrix = **$(N*N)*(4*N - 1) = 4*N^3 - N^2$**
- Therefore, theoretically, total time req. for matrix multiplication = **$(4*N^3 - N^2)$ / (Clock Speed Freq.)**
  **= *$(4*N^3 - N^2) / (2.8 * 10^9)$ sec***
  (taking max CPU clock speed)

*Theoretical Estimation of time req. for various '**N**' :*

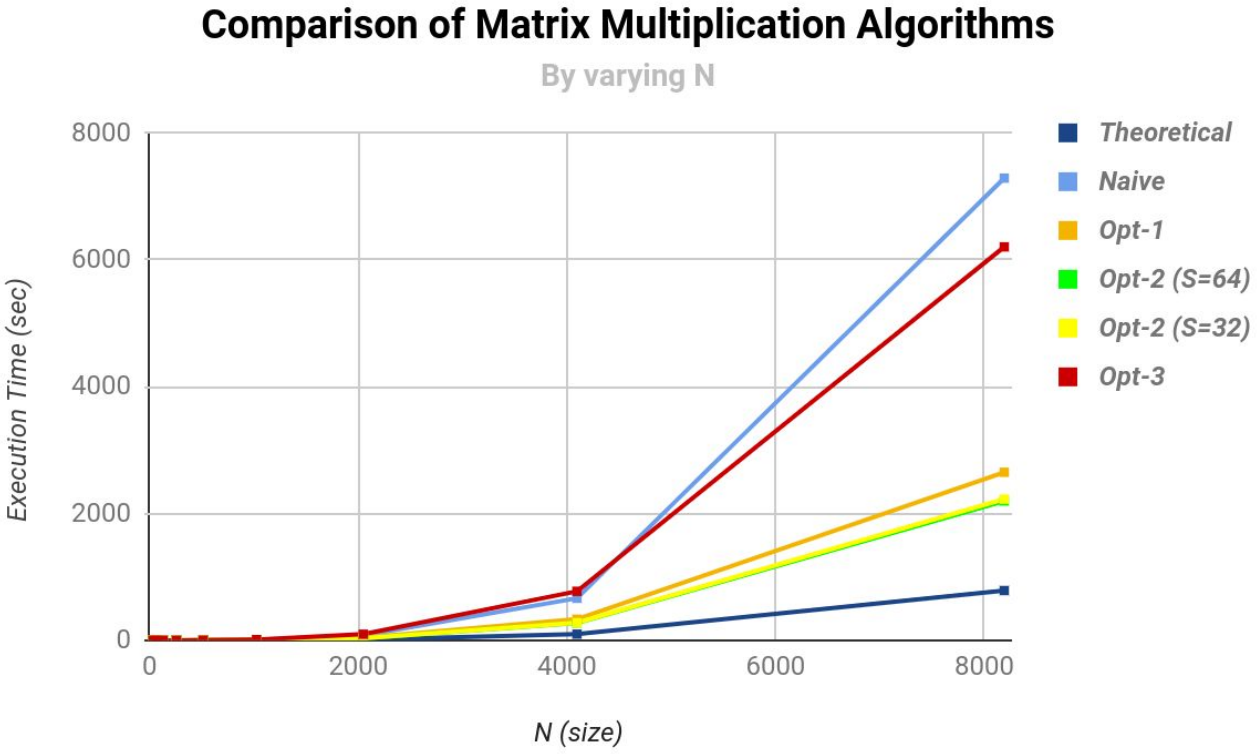
| **N** | **Theoretical Time Estimate (sec)** |
|---|---|
| **32** | 0.0000464 |
| **64** | 0.000373029 |
| **128** | 0.00299008 |
| **256** | 0.023944 |
| **512** | 0.191646 |
| **1024** | 1.53354 |
| **2048** | 12.2698 |
| **4096** | 98.1647 |
| **8192** | 785.341 |

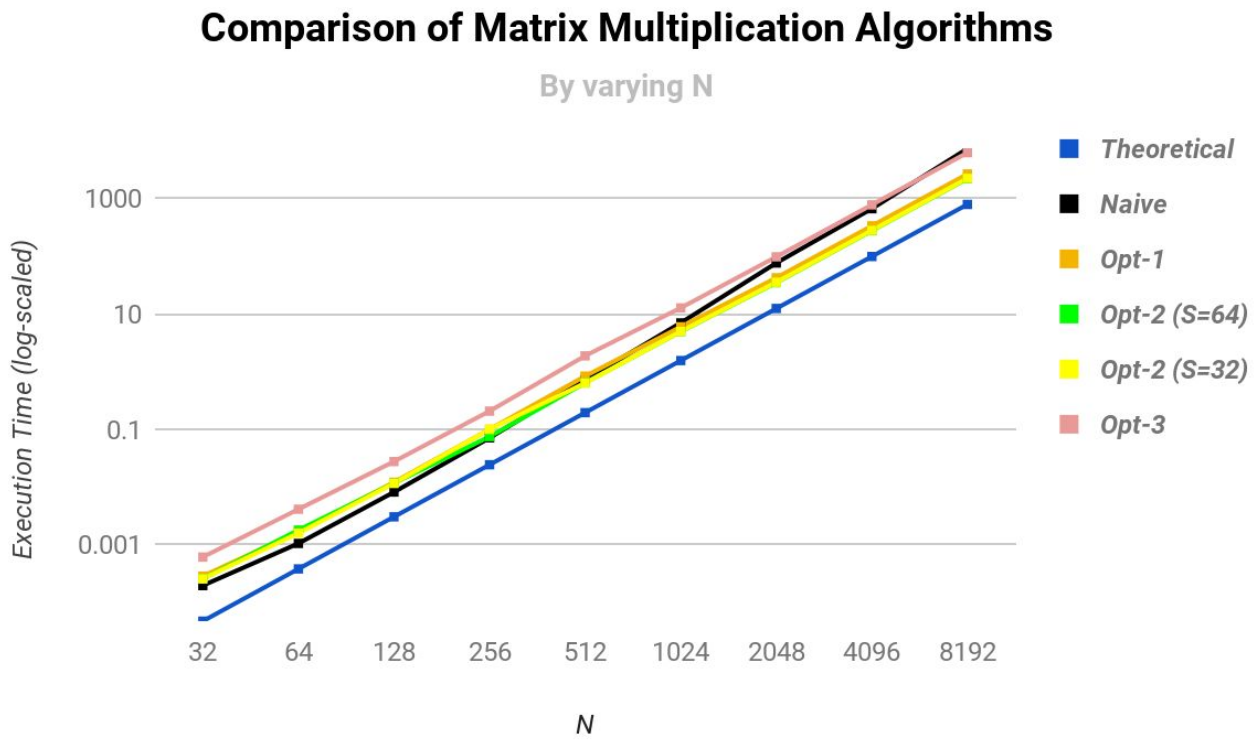
## *Theoretical Estimation of 'S' for Optimization-2 :-*

- $s^2$ = O(Memory Size)
- Thus s = O(√M)
- Cache Memory Size (L1d) = 32 KB = 32 * 1024 Bytes = 32768 Bytes
- Assuming 1 integer takes 4 Bytes space, total integers that could be accommodated in L1d cache = sqrt(32768/4) = sqrt(8192) = 90.5
- Thus, keeping '**s**' a power of 2 for easy calculations, therefore s = **64**.

***Experimental Values (in seconds) for various Matrix Multiplication algorithms :-***

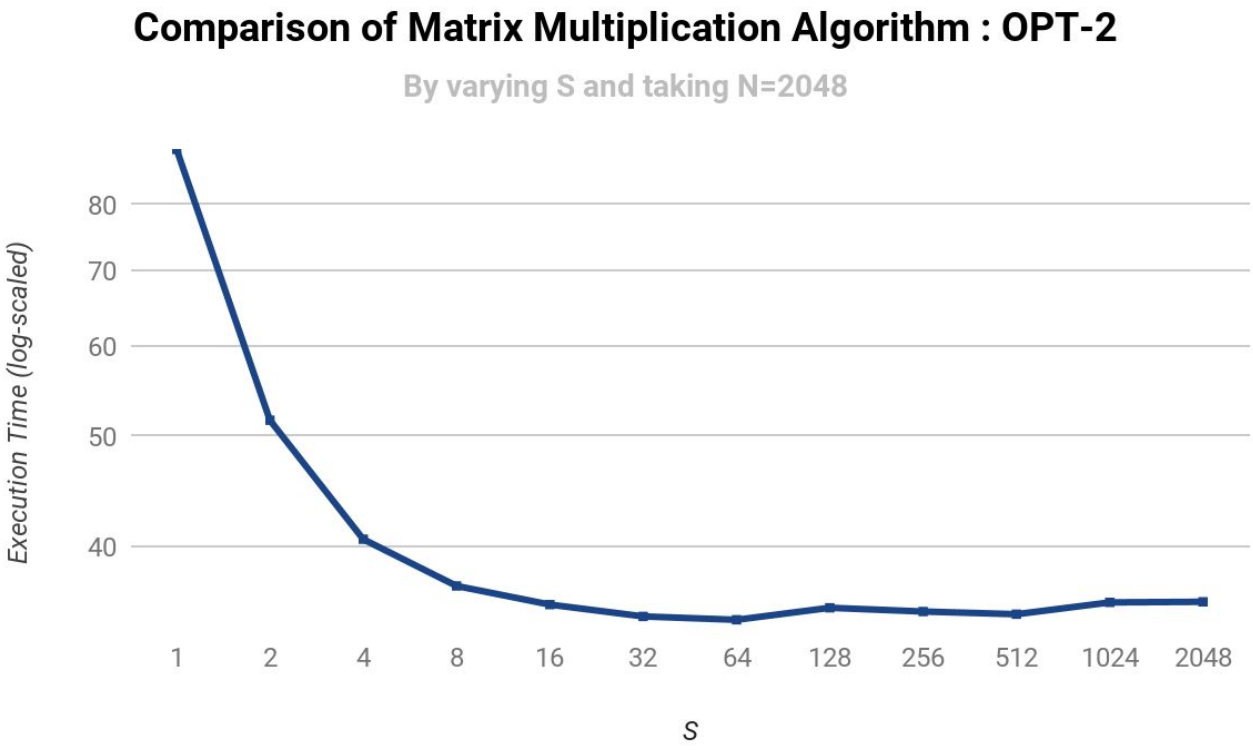
| N | Theo-retical | Naive | Opt-1 | Opt-2 (S=64) | Opt-2 (S=32) | Opt-3 |
|---|---|---|---|---|---|---|
| **32** | 0.000046 | .0001921 | .0002803 | .0002510 | .0002510 | .0005996 |
| **64** | 0.000373 | .0010323 | .0016472 | .0017624 | .0015604 | .0040426 |
| **128** | 0.002990 | .0080263 | .0119194 | .011438 | .0113923 | .0272039 |
| **256** | 0.023944 | .0698858 | .0983857 | .0736368 | .0985188 | .2039024 |
| **512** | 0.191646 | .7141663 | .8242451 | .629611 | .6251896 | 1.8558794 |
| **1024** | 1.53354 | 6.921182 | 5.9103039 | 4.8722665 | 4.91027 | 12.5831873 |
| **2048** | 12.2698 | 75.702026 | 41.8808392 | 34.4951215 | 35.0368268 | 96.8575197 |
| **4096** | 98.1647 | 663.09131 | 333.830818 | 274.682657 | 278.038222 | 773.566261 |
| **8192** | 785.341 | 7286.1370 | 2649.59360 | 2197.65395 | 2226.72454 | 6202.51643 |


Comparison of Matrix Multiplication Algorithms
By varying N
Execution Time (sec)
8000
6000
4000
2000
0
0
2000
4000
6000
8000
N (size)
Theoretical
Naive
Opt-1
Opt-2 (S=64)
Opt-2 (S=32)
Opt-3


Comparison of Matrix Multiplication Algorithms
By varying N
Execution Time (log-scaled)
1000
10
0.1
0.001
32
64
128
256
512
1024
2048
4096
8192
N
Theoretical
Naive
Opt-1
Opt-2 (S=64)
Opt-2 (S=32)
Opt-3

***Experimental Values (in seconds) for OPT-2 Matrix Multiplication algorithm by varying only S and taking N=2048 :-***

| S | Time |
|---|---|
| **1** | 89.4218571 |
| **2** | 51.6479762 |
| **4** | 40.5802632 |
| **8** | 36.915138 |
| **16** | 35.54625567 |
| **32** | 34.70508967 |
| **64** | 34.4672745 |
| **128** | 35.31786067 |
| **256** | 35.05077167 |
| **512** | 34.85963067 |
| **1024** | 35.70448367 |
| **2048** | 35.74804767 |



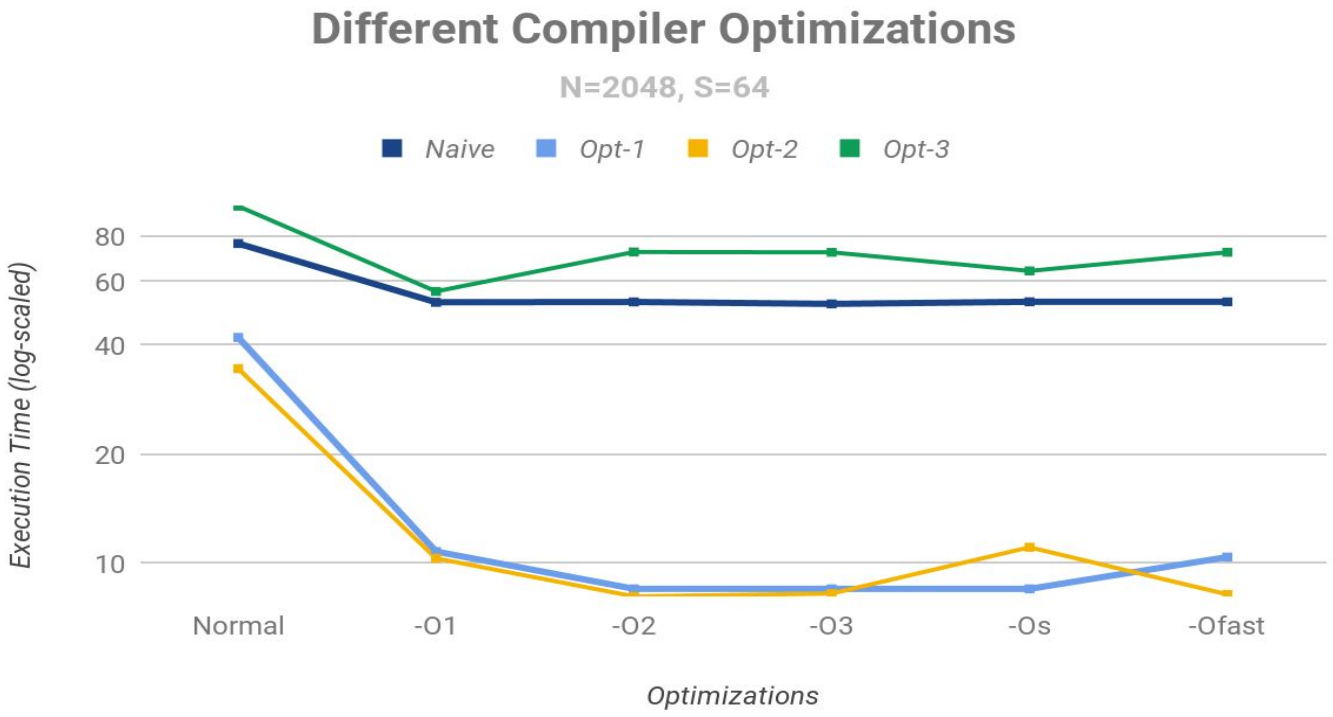

## ***Algorithmic Implementation Optimizations :-***

- In the initial (Basic) implementation of all the algorithms, many repetitive multiplications, additions, variable recalculations and same array element accesses were present, which were removed by replacing the repetitions with temporary variables storing one time variable calculations.
- These above mentioned optimizations reduced the runtime by a great factor.
- Multiplications were replaced by additions and when necessary, by shift operator too, wherever possible, which saved processor's clock cycles and hence, as a result, reduced the program runtime.

## ***Compiler Optimizations :-***

For further optimisations, compiler's -O1, -O2, -O3, -Os & -Ofast options were used. The below values are taken for N=2048 and S=64.

| **Optimization** | **Naive** | **Opt-1** | **Opt-2** | **Opt-3** |
|---|---|---|---|---|
| **Normal** | 76.36342 | 42.06009267 | 34.482128 | 96.82368134 |
| **-O1** | 52.58312534 | 10.788714 | 10.33126167 | 56.333432 |
| **-O2** | 52.67909634 | 8.51202267 | 8.11296234 | 72.36356434 |
| **-O3** | 52.0379967 | 8.50784767 | 8.2583234 | 72.22068934 |
| **-Os** | 52.74261567 | 8.507501 | 11.08171867 | 64.15913834 |
| **-Ofast** | 52.720721 | 10.41360034 | 8.214731 | 72.2287867 |



## ***Observations and Conclusions :-***

- Algorithms Opt-1 and Opt-2 perform better than naive and Opt-3 ones as expected since they include loop interchange optimization which increases the hit ratio of the cache and hence reduces the runtime.
- Opt-2 is expected to perform better than Opt-1 as both have loop interchange optimization, and in addition, Opt-2 does the matrix multiplications in small blocks which further increases the hit ratio evenmore, and the same result is seen experimentally too.
- Naive algorithm performs better than Opt-3 for small values of N, which is expected behaviour as Opt-3 involves recursive calls which adds some extra runtime cost. Though for large values of N, this extra runtime cost becomes irrelevant and hence Opt-3 starts performing better than Naive as N increases beyond a sufficiently large value.
- Opt-2 keeps on improving performance better till some optimal value of “s” after which if ‘s’ is increased, the block of the matrix does not fit into the cache which in turn increases the miss ratio. However, further increasing the value of ‘s’ doesn’t have much effect on the elapsed execution time.
- Between s=32 and s=64, it was observed that for small values of n, s=32 performed slightly better than s=64, but for large values of n, we saw that s=64 performed sufficiently better than s=32.
- Compiler Optimizations -O1, -O2, -O3, -Os & -Ofast  reduce the runtime of the Naive, Opt-1 and Opt-2 algorithms to a great extent. However, the recursive version in Opt-3 does not get much benefitted from the compiler optimizations -O2, -O3 & -Ofast since very less scope of vector optimizations is involved in Opt-3 whereas -O3 (& hence -Ofast) mainly performs vector optimizations by switching on *-ftree-loop-vectorize* flag.
- Also, the reason why -Os provides less time for Opt-3 than the above mentioned 3 optimizations is maybe because -Os enables all -O2

optimizations except those that often increase code size and also switches on the flag : *-fprefetch-loop-arrays*.

- For Opt-3 recursive code, compiler optimisation -O1 performed the best of all.