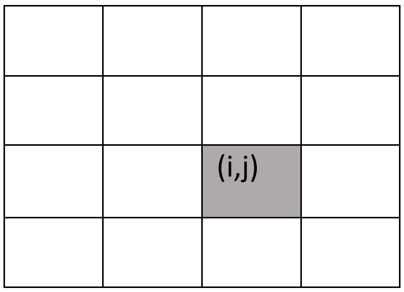
CSE590: Supercomputing, Spring 2016 - Homework #1

Group 18:
Dhanendra Jain – 110369635
Muhammad Ali Ejaz – 110559131
Nirmit Desai – 110351469

a) The nested i and j for-loops are independent and therefore parallelizable as shown below:



The values of k come from a different row and column and hence the k for-loop cannot be parallelized.

Our resultant parallel algorithm would then result in the following:

```
for k ← 1 to n do
     parallel for i ← 1 to n do
          parallel for j ← 1 to n do
               D[i, j] ← min (D[i, j], D[i, k] + D[k, j])
```

The runtime of the serial algorithm is $\Theta(n^3)$. Hence, **work** done when running it on p processors would be

$T_1(n) = n^3/p + (\text{cost required to broadcast})$

$T_1(n) = \Theta(n^3)$

where p is the number of processor, and the cost required to broadcast would be less than n^3

The **span** of the algorithm would be as follows:

$T_\infty(n) = \Theta(n*(\log n + \log n)) = \Theta(n \log n)$

The **parallelism** for the algorithm could thus be computed as shown below:

$T_1(n)/ T_\infty(n) = \Theta(n^2/\log n)$

b)

Figure 2: assuming m=1, $T_1(n) = \Theta(1)$ if n=1 (For A-loop-FW)

Hence, **work**:

$$T_1(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 8T_1\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases} \quad = \theta(n^3)$$

Span:

$$T_\infty(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 6T_\infty\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases} \quad = \theta(n^{\log_2 6}) = \theta(n^{2.59})$$

Parallelism:

$$\frac{T_1(n)}{T_\infty(n)} = \theta\left(n^{3-\log_2 6}\right) = \theta(n^{3-2.59}) = \theta(n^{0.41})$$

Figure 4: assuming m=1, $T_1(n) = \Theta(1)$ if n=1

Hence, **work**:

$$T_{1,B}(n) = T_{1,C}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 8T_1\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases} \quad = \theta(n^3)$$

$$T_{1,A}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 4T_{1,A}\left(\frac{n}{2}\right) + 2(T_{1,B}\left(\frac{n}{2}\right) + T_{1,C}\left(\frac{n}{2}\right)) + \theta(1), & otherwise \end{cases} \quad = \theta(n^3)$$

Span:

$$T_{\infty,B}(n) = T_{\infty,C} = \begin{cases} \theta(1), & if\ n = 1 \\ 4T_\infty\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases} \quad = \theta(n^2)$$

$$T_{\infty,A}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 4T_\infty\left(\frac{n}{2}\right) + 2\max\left\{T_{\infty,B}\left(\frac{n}{2}\right), T_{\infty C}\left(\frac{n}{2}\right)\right\}, & otherwise \end{cases}$$

On substituting $T_{\infty,B/C} = \Theta(n^2)$,

$$T_{\infty,A}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 4T_\infty\left(\frac{n}{2}\right) + \theta(n^2), & otherwise \end{cases} \quad = \quad \theta(n^2 \log n)$$

Parallelism:

$$\frac{T_1(n)}{T_\infty(n)} = \theta(n^3/n^2 \log n) = \theta(n/\log n)$$

Figure 5: assuming m=1, $T_1(n) = \Theta(1)$ if n=1

Hence, **work**:

$$T_{1,B}(n) = T_{1,C}(n) = T_{1,D}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 8T_1\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases} \quad = \theta(n^3)$$

$$T_{1,A}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 2(\left(\frac{n}{2}\right) + T_{1,B}\left(\frac{n}{2}\right) + T_{1,C}\left(\frac{n}{2}\right) + T_{1,D}\left(\frac{n}{2}\right)) + \theta(1), & otherwise \end{cases} \quad = \theta(n^3)$$

Span:

$$T_{\infty,D}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 2T_\infty\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases} \quad = \theta(n)$$

and

$$T_{\infty,B}(n) = T_{\infty,C}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 2T_{\infty,B/C}\left(\frac{n}{2}\right) + 2T_{\infty,D}\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases}$$

On substituting $T_{\infty,D} = \Theta(n)$

$$T_{\infty,B}(n) = T_{\infty,C}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 2T_{\infty,B/C}\left(\frac{n}{2}\right) + \theta(n), & otherwise \end{cases} \quad = \theta(n \log n)$$

Now,

$$T_{\infty,A}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 2T_\infty\left(\frac{n}{2}\right) + 2\max\left\{T_{\infty,B}\left(\frac{n}{2}\right), T_{\infty C}\left(\frac{n}{2}\right), T_{\infty D}\left(\frac{n}{2}\right)\right\}, & otherwise \end{cases}$$

On substituting $\max\{T_{\infty,D} = \Theta(n), T_{\infty,B/C} = \Theta(n \log n)\} = \Theta(n \log n)$

$$T_{\infty,A}(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 2T_{\infty}\left(\frac{n}{2}\right) + 2\,\theta(n\log n), otherwise \end{cases} = \theta(n\,log^2 n)$$

Parallelism:

$$\frac{T_1(n)}{T_{\infty}(n)} = \theta(n^2/log^2 n)$$

c)

The possible values of X, U and V in $A_{loop\text{-}FW}$ comes in same set, that is X≡U≡V. This can be seen in the output of any sample runs of the algorithm; partial output below (please find attached file output.txt for the complete output):

```
--------- in A ------------
xRowStart = 0 and xColStart = 0 and size = 2
uRowStart = 0 and uColStart = 0 and size = 2
vRowStart = 0 and vColStart = 0 and size = 2
```

The possible values of X, U and V in $B_{loop\text{-}FW}$ and $C_{loop\text{-}FW}$ comes in pairs. For the $B_{loop\text{-}FW}$ X and U are disjoint but X and V are possibly same; X≡V. Similarly, for $C_{loop\text{-}FW}$ X and V are disjoint but X and U are possibly same; X≡U. This can be seen in the output of any sample runs of the algorithm; partial output below (please find attached file output.txt for the complete output):

```
--------- in B ------------
xRowStart = 0 and xColStart = 2 and size = 2
uRowStart = 0 and uColStart = 0 and size = 2
vRowStart = 0 and vColStart = 2 and size = 2

--------- in C ------------
xRowStart = 2 and xColStart = 0 and size = 2
uRowStart = 2 and uColStart = 0 and size = 2
vRowStart = 0 and vColStart = 0 and size = 2
```

The possible values of X, U and V in $D_{loop\text{-}FW}$ are all disjoint. This can be seen in the output of any sample runs of the algorithm; partial output below (please find attached file output.txt for the complete output):

```
--------- in iter ------------
xRowStart = 2 and xColStart = 2 and size = 2
uRowStart = 2 and uColStart = 0 and size = 2
vRowStart = 0 and vColStart = 2 and size = 2
```

This means that A_{FW} can spawn only one $A_{loop\text{-}FW}$ instance at a time, while B_{FW} and C_{FW} can spawn two instances of $B_{loop\text{-}FW}$ and $C_{loop\text{-}FW}$ respectively at any given time and hence $B_{loop\text{-}FW}$ and $C_{loop\text{-}FW}$ are more optimizable than $A_{loop\text{-}FW}$. D_{FW} on the other hand can spawn multiple instances of $D_{loop\text{-}FW}$ and hence it is most optimizable among the four.

d)

$A_{loop\text{-}FW}$

```
for k ← 1 to n do
     for i ← 1 to n do
          for j ← 1 to n do
               D[i, j] ← min (D[i, j], D[i, k] + D[k, j])
```

Work:

The runtime of the serial algorithm is $\Theta(n^3)$. Hence, running it on p processors would result in

$$T_1(n) = n^3/p + (\text{cost required to broadcast}) = \Theta(n^3)$$

where p is the number of processor, and the cost required to broadcast would be less than n^3

Span:

$$T_\infty(n) = \Theta(n*n*n) = \Theta(n^3)$$

Parallelism:

$$T_1(n)/ T_\infty(n) = \Theta(n^3/n^3) = \Theta(1)$$

$B_{loop\text{-}FW}$

```
for k ← 1 to n do
     for i ← 1 to n do
          parallel for j ← 1 to n do
               D[i, j] ← min (D[i, j], D[i, k] + D[k, j])
```

Work:

The runtime of the serial algorithm is $\Theta(n^3)$. Hence, running it on p processors would result in

$$T_1(n) = n^3/p + (\text{cost required to broadcast}) = \Theta(n^3)$$

where p is the number of processor, and the cost required to broadcast would be less than n^3

Span:

$$T_\infty(n) = \Theta(n * n * \log n) = \Theta(n^2 \log n)$$

Parallelism:

$$T_1(n)/ T_\infty(n) = \Theta(n^3 / n^2 \log n) = \Theta(n/\log n)$$

$C_{loop\text{-}FW}$

```
for k ← 1 to n do
     parallel for i ← 1 to n do
          for j ← 1 to n do
               D[i, j] ← min (D[i, j], D[i, k] + D[k, j])
```

Work:

The runtime of the serial algorithm is $\Theta(n^3)$. Hence, running it on p processors would result in

$$T_1(n) = n^3/p + (\text{cost required to broadcast}) = \Theta(n^3)$$

where p is the number of processor, and the cost required to broadcast would be less than n^3

Span:

$$T_\infty(n) = \Theta(n * \log n * n) = \Theta(n^2 \log n)$$

Parallelism:

$$T_1(n) / T_\infty(n) = \Theta(n^3 / n^2 \log n) = \Theta(n/\log n)$$

$D_{loop\text{-}FW}$

```
parallel for k ← 1 to n do
     parallel for i ← 1 to n do
          parallel for j ← 1 to n do
               D[i, j] ← min (D[i, j], D[i, k] + D[k, j])
```

Work:

The runtime of the serial algorithm is $\Theta(n^3)$. Hence, running it on p processors would result in

$$T_1(n) = n^3/p + (\text{cost required to broadcast}) = \Theta(n^3)$$

where p is the number of processor, and the cost required to broadcast would be less than n^3

Span:

$$T_\infty(n) = \Theta(\log n + \log n + \log n) = \Theta(\log n)$$

Parallelism:

$$T_1(n) / T_\infty(n) = \Theta(n^3/\log n)$$

e)

Please find attached all the three implementations of Floyd-Warshall's APSP using Cilk.

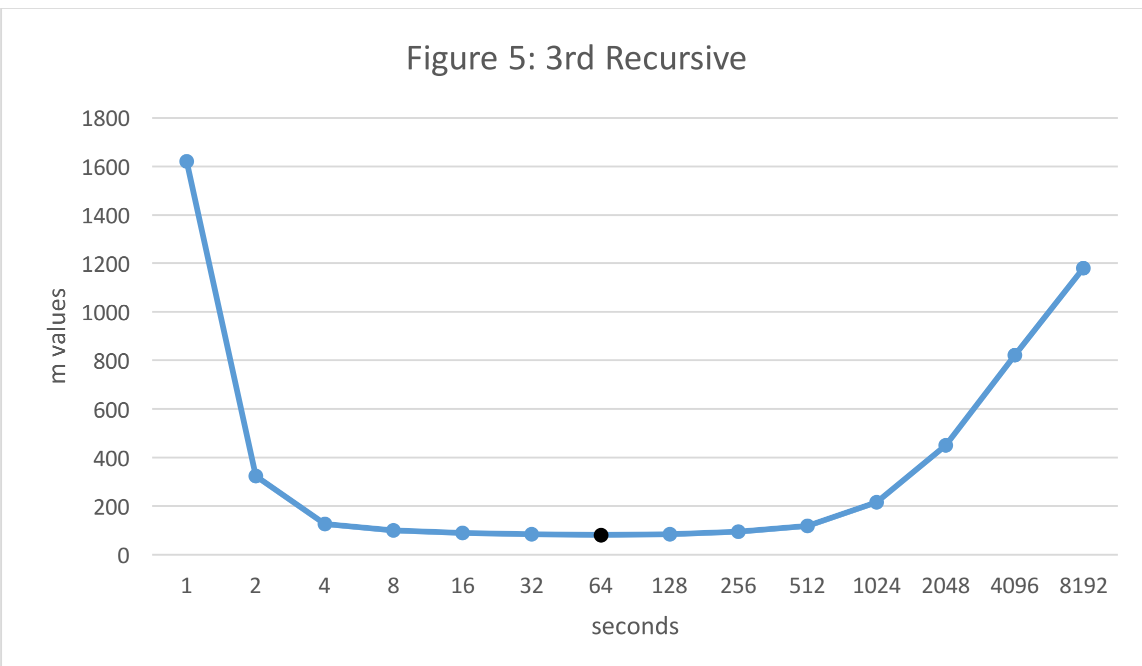
The iterative parallel algorithm when ran for **n=2^{13}** (m value doesn't matter here) gave the running time as **137.574005 seconds**.

The below tables and line plots show the variation in running time for m between 2^0 to 2^{13} keeping n=2^{13} fixed.

For figure 5: 3rd recursive implementation:

m	Seconds
1	1620.678955
2	323.994995
4	126.117996
8	99.333
16	89.788002
32	85.543999
64	**80.837997**
128	85.287003
256	94.606003
512	119.241997
1024	217.371002
2048	449.697998
4096	822.14502
8192	1181.396973

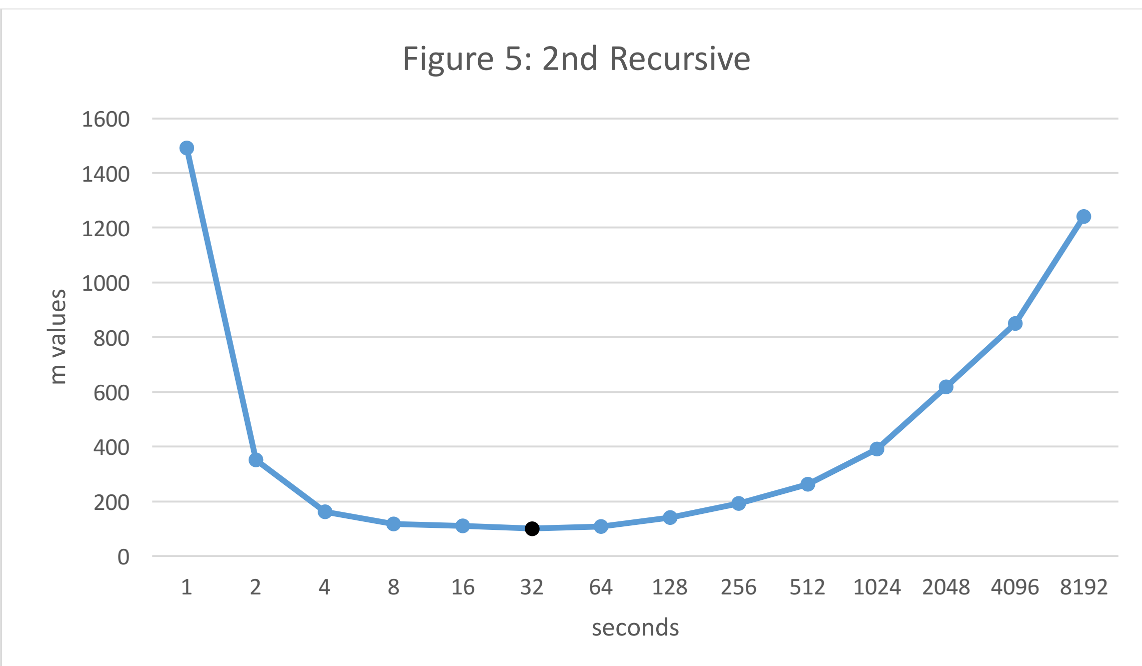
Best running time is for m = 64



For figure 4: 2nd recursive implementation:

m	Seconds
1	1492.016968
2	350.242004
4	160.955002
8	116.786003
16	109.93
32	**101.619003**
64	107.775002
128	140.231003
256	191.934006
512	262.880005
1024	391.71701
2048	618.911011
4096	849.536011
8192	1240.140015

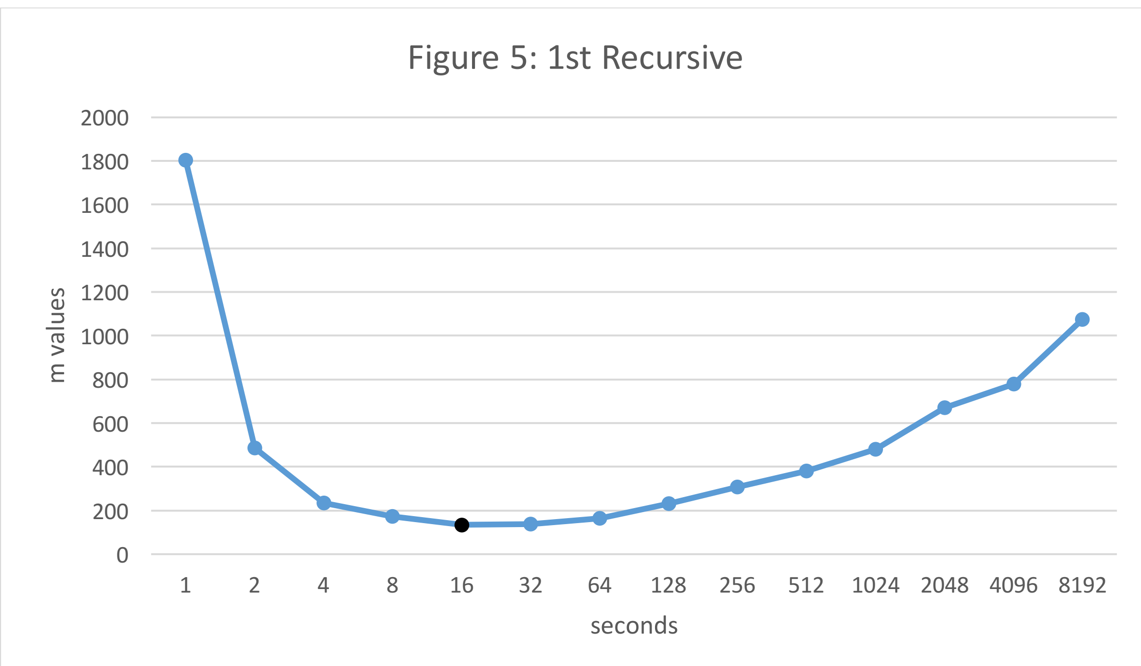
Best running time is for m = 32



For figure 2: 1st recursive implementation:

m	Seconds
1	1803.753052
2	485.403015
4	234.712997
8	171.934006
16	**135.057999**
32	137.313995
64	163.203003
128	231.447998
256	306.871002
512	380.311005
1024	479.204987
2048	669.250977
4096	777.583984
8192	1073.828979

Best running time is for m = 16

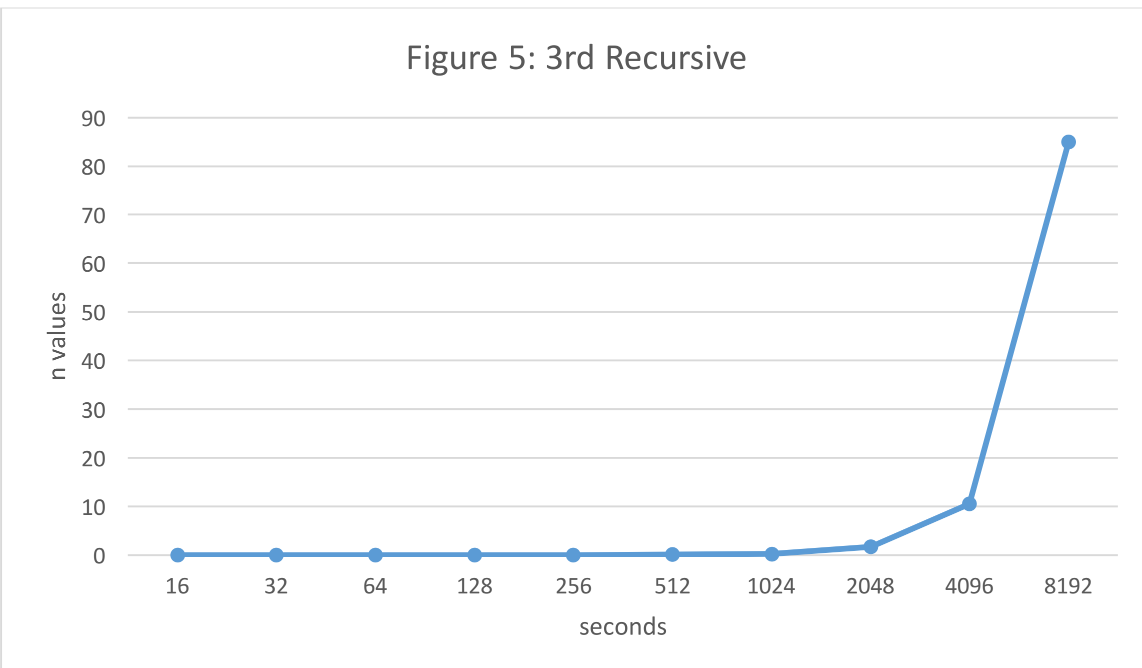


f)

We ran our implementation on Stampede which has 16 processors. Hence our implementation was also on 16 parallel processors. Please find below the plots of the running time of each implementation as we varied n from 2^4 to 2^{13} (with our optimal value of m obtained in part (e)).

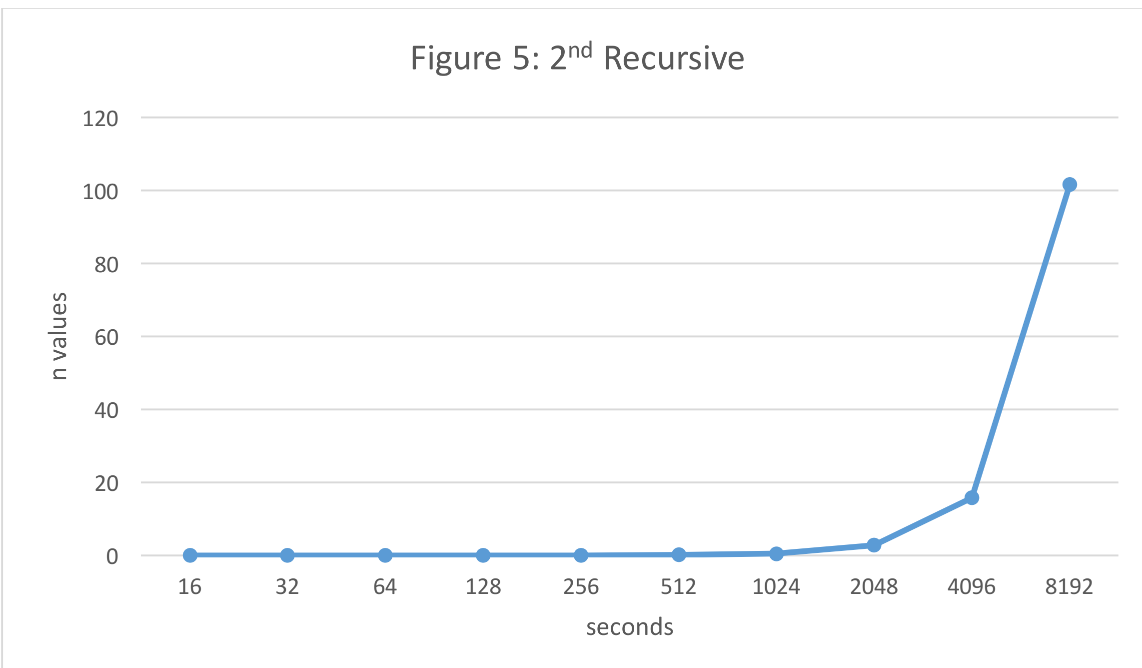
For figure 5: 3rd recursive implementation (For m = 64):

n	Seconds
16	0
32	0
64	0
128	0.003
256	0.017
512	0.075
1024	0.312
2048	1.682
4096	10.575
8192	84.992996



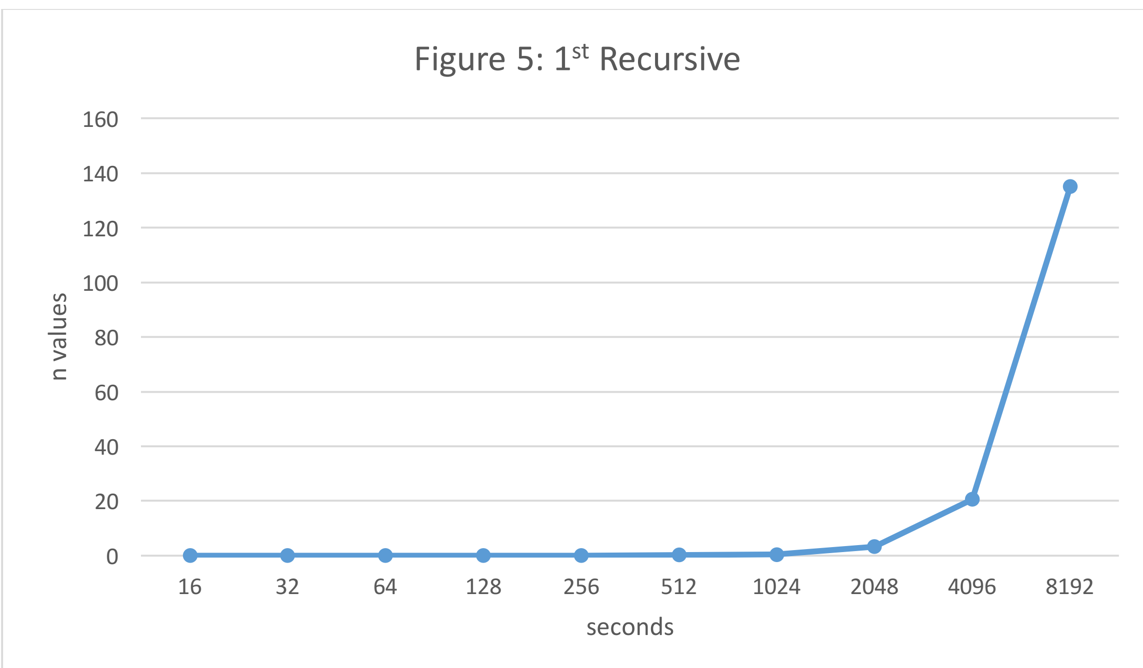
For figure 4: 2^{nd} recursive implementation (For m = 32):

n	Seconds
16	0
32	0
64	0
128	0.003
256	0.018
512	0.097
1024	0.509
2048	2.699
4096	15.738
8192	101.619003



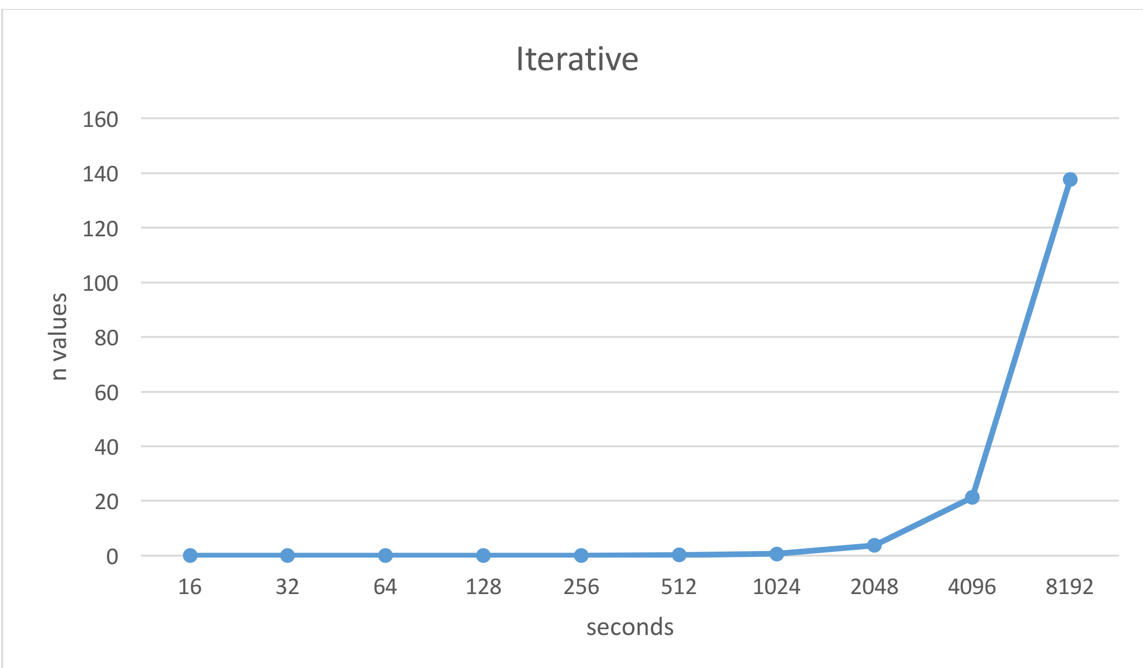
For figure 2: 1st recursive implementation (for m = 16):

n	Seconds
16	0
32	0
64	0
128	0.002
256	0.015
512	0.09
1024	0.539
2048	3.275
4096	20.545
8192	135.057999



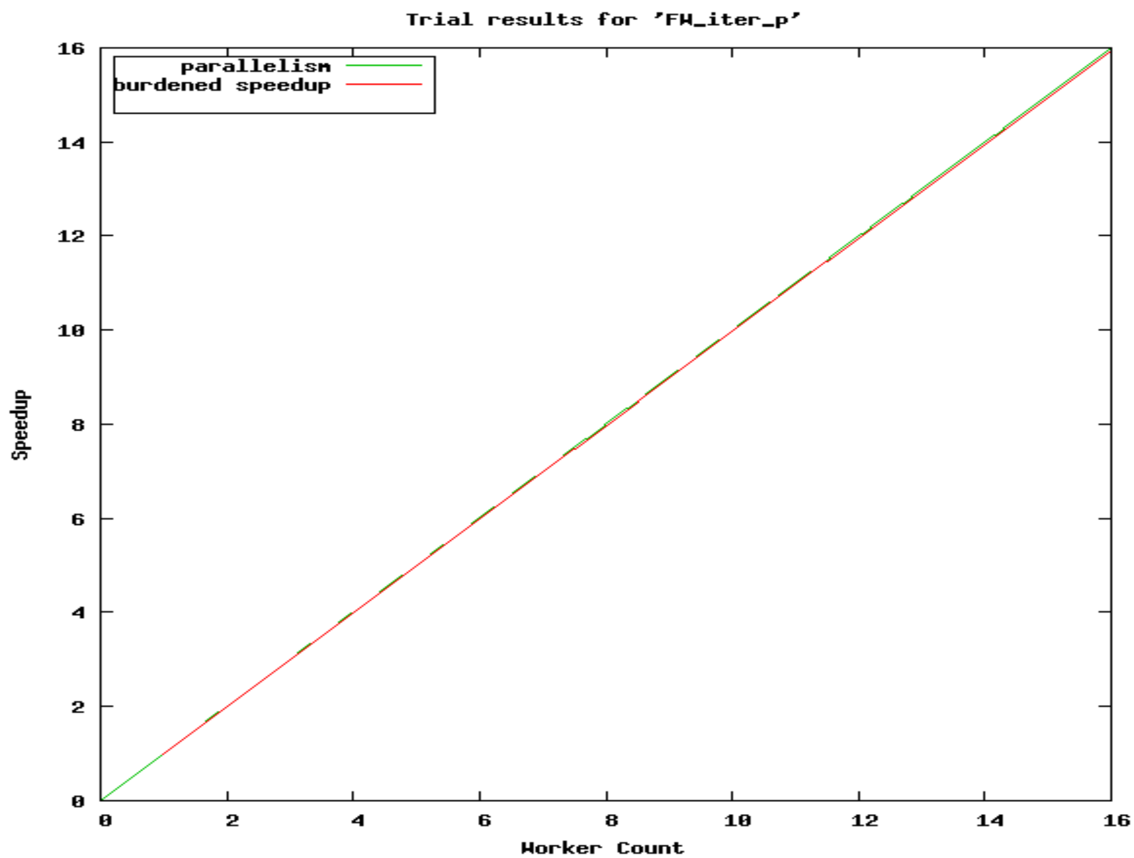
For iterative implementation:

n	Seconds
16	0
32	0.001
64	0.003
128	0.009
256	0.043
512	0.166
1024	0.712
2048	3.592
4096	21.204
8192	137.574005

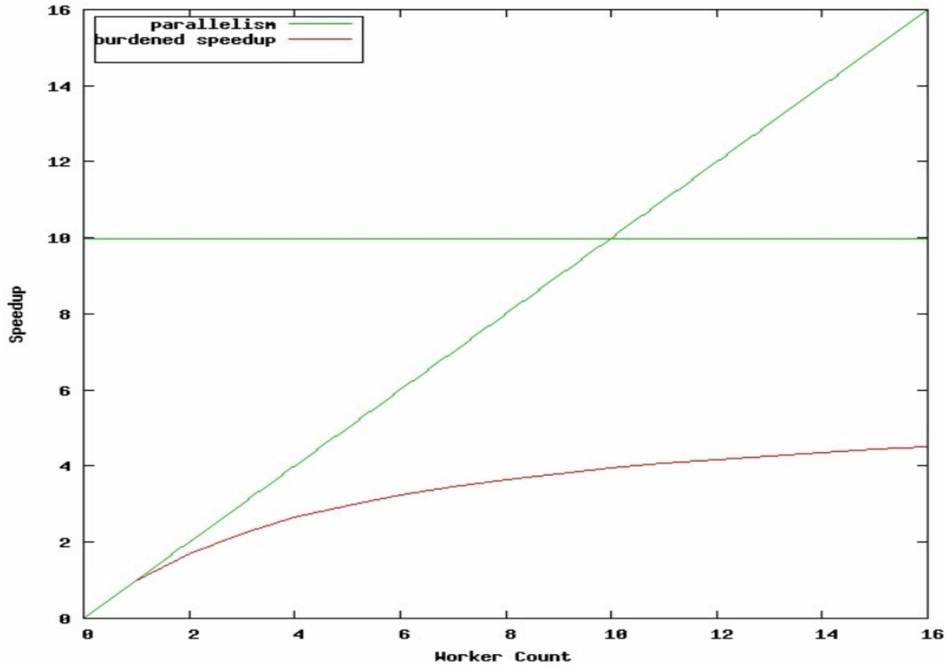


g)
We tried running cilkview with –trials all option for running on all threads but it took lot of time and the job got timed out in Stampede. So, we only ran cilkview without trial options, which takes default number of threads as 16. Below is the output obtained:

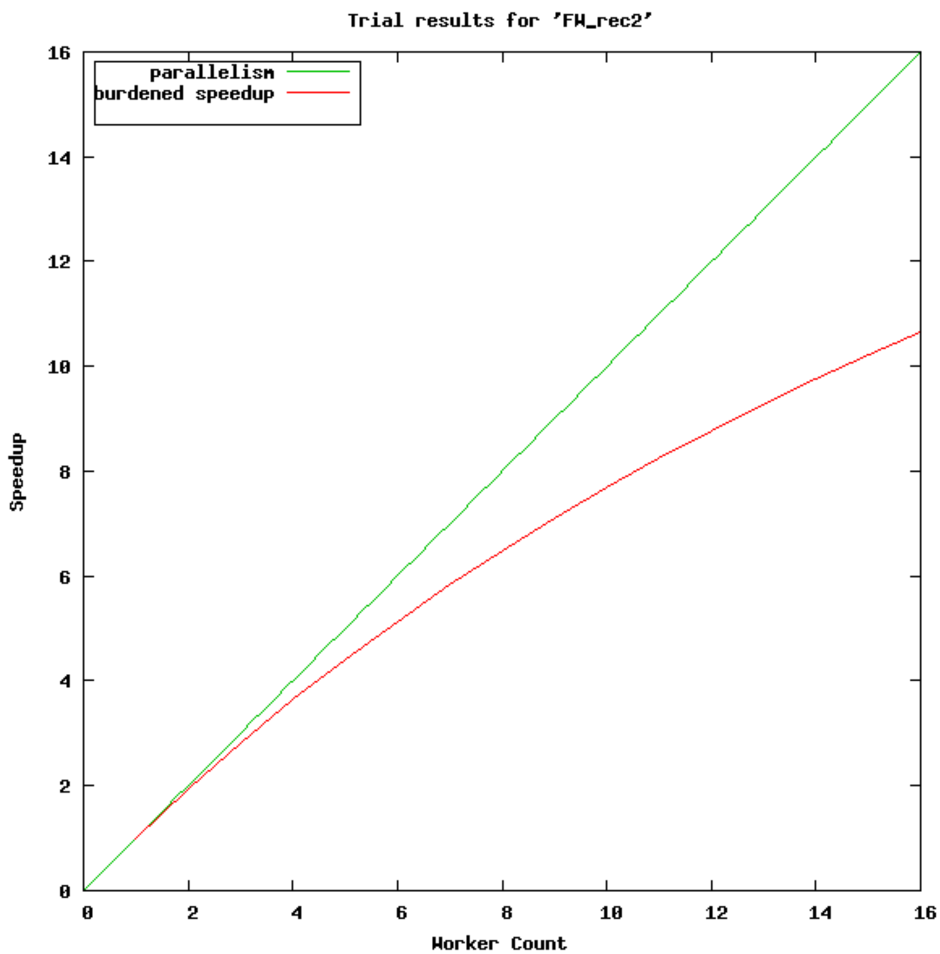
For iterative:



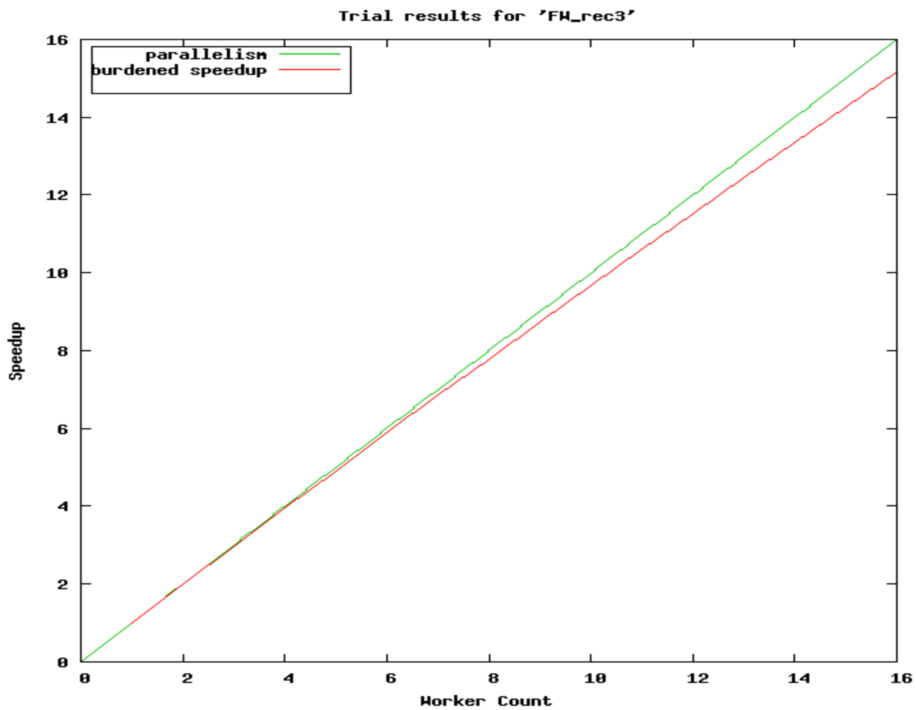
For figure 2:



For figure 4:



For figure 5:



h)

Please find attached all the three implementations of Floyd-Warshall's APSP using OpenMP.

The iterative parallel algorithm when ran for $n=2^{13}$ (m value doesn't matter here) gave the running time as 131.488 seconds.

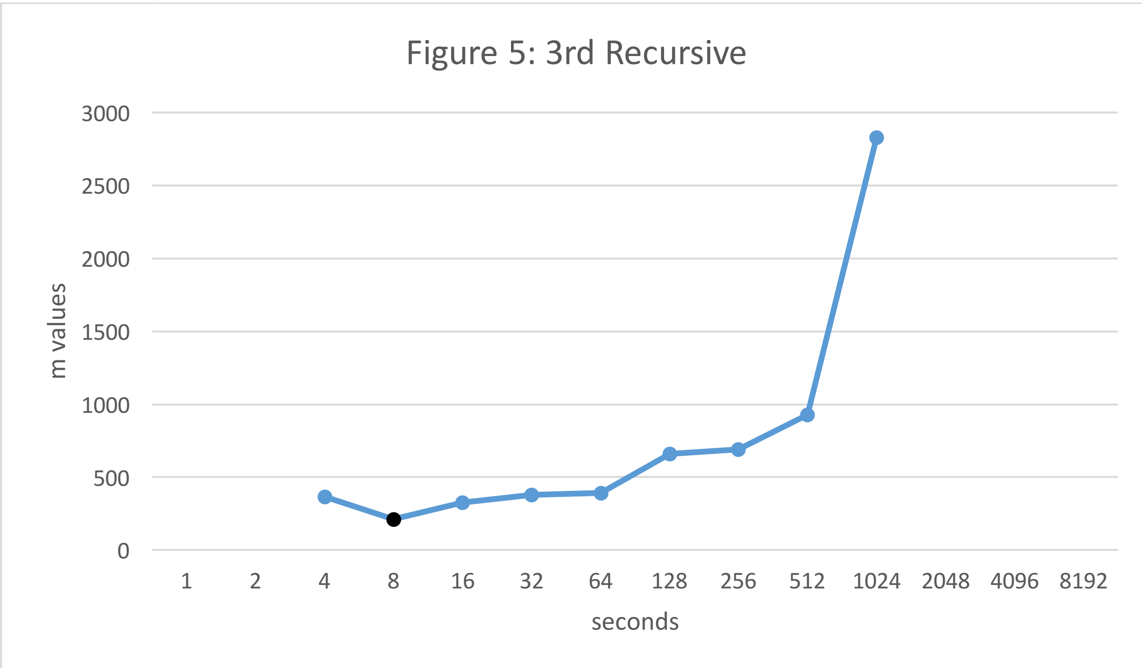
The below tables and line plots show the variation in running time for m between 2^0 to 2^{13} keeping $n = 2^{13}$ fixed.

For figure 5: 3rd recursive implementation:

m	Seconds
1	
2	
4	367.155
8	**213.133**
16	324.934
32	378.818
64	390.673
128	660.698
256	689.046
512	926.31
1024	2829.93
2048	
4096	

8192	

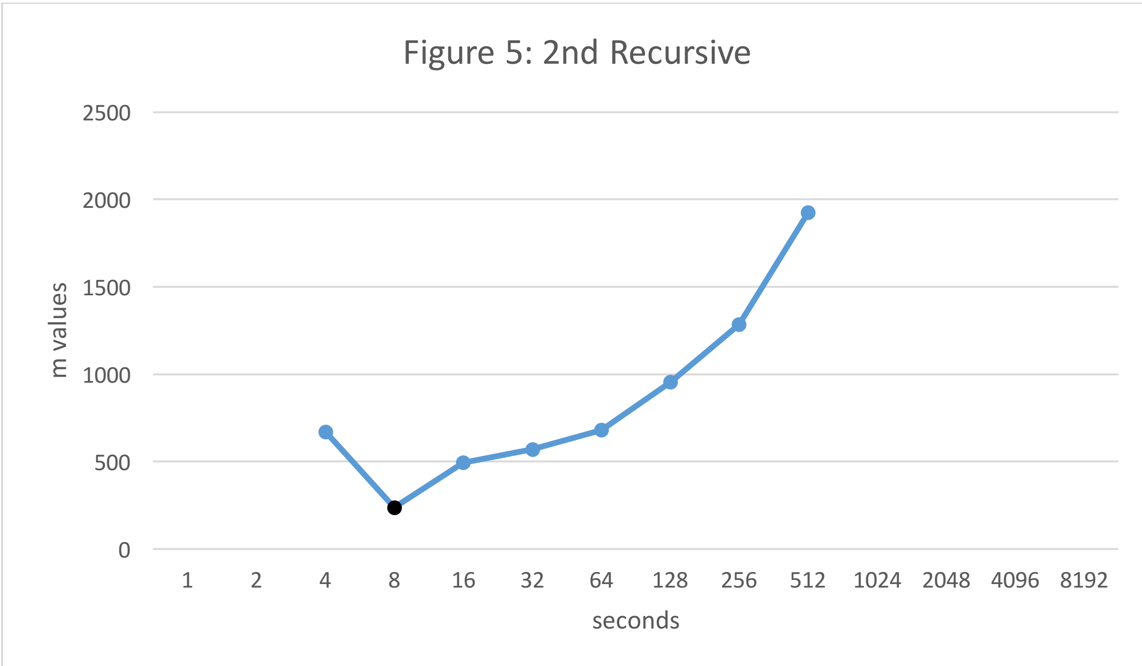
Best running time is for m = 8



For figure 4: 2nd recursive implementation:

m	Seconds
1	
2	
4	668.293
8	**238.829**
16	493.073
32	571.683
64	682.683
128	953.752
256	1284.02
512	1923.71
1024	
2048	
4096	
8192	

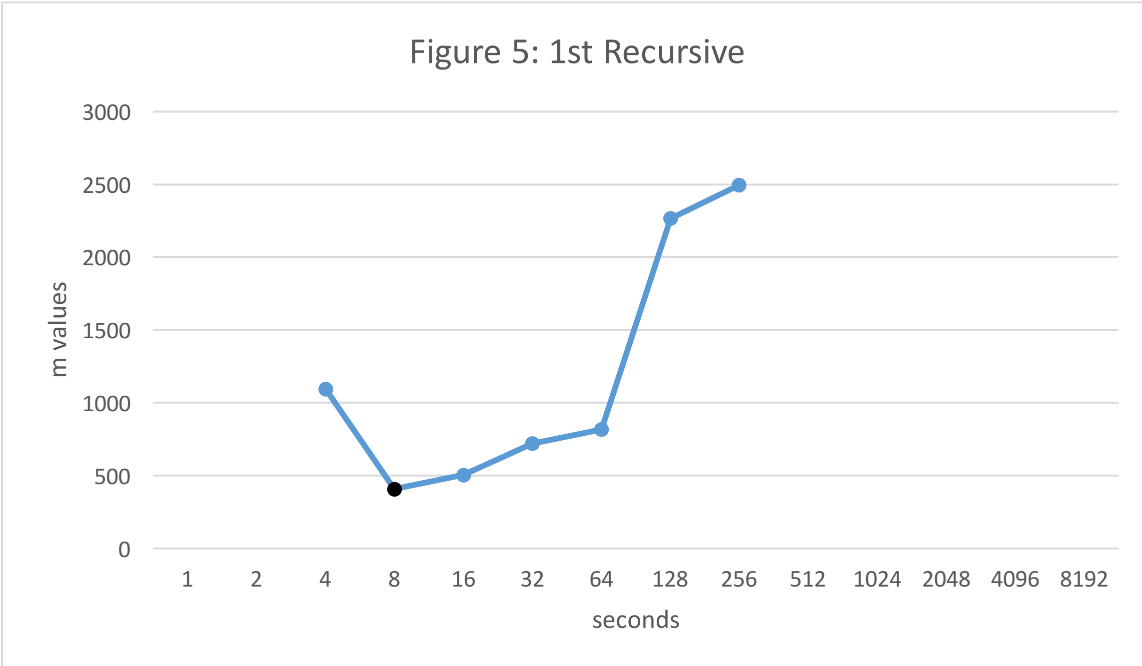
Best running time is for m = 8



For figure 2: 1st recursive implementation:

m	Seconds
1	
2	
4	1092.54
8	**409.597**
16	505.58
32	721.738
64	817.825
128	2267.08
256	2494.17
512	
1024	
2048	
4096	
8192	

Best running time is for m = 8

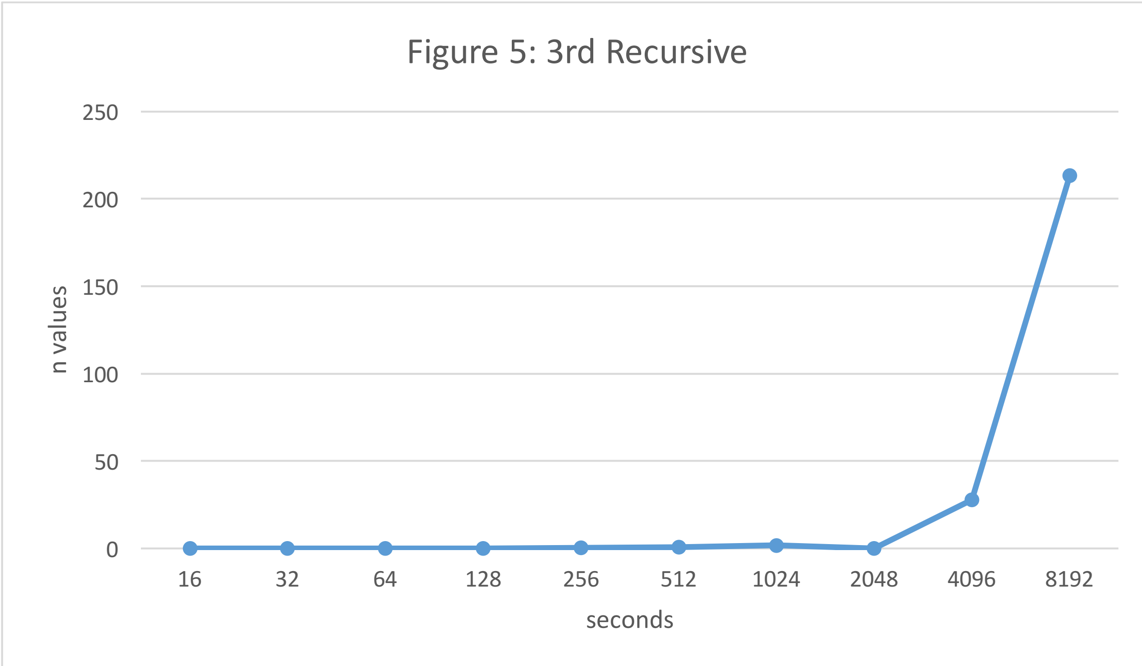


Part II

We ran our implementation on Stampede which has 16 processors. Hence our implementation was also on 16 parallel processors. Please find below the plots of the running time of each implementation as we varied n from 2^4 to 2^{13} (with our optimal value of m obtained in part (e)).

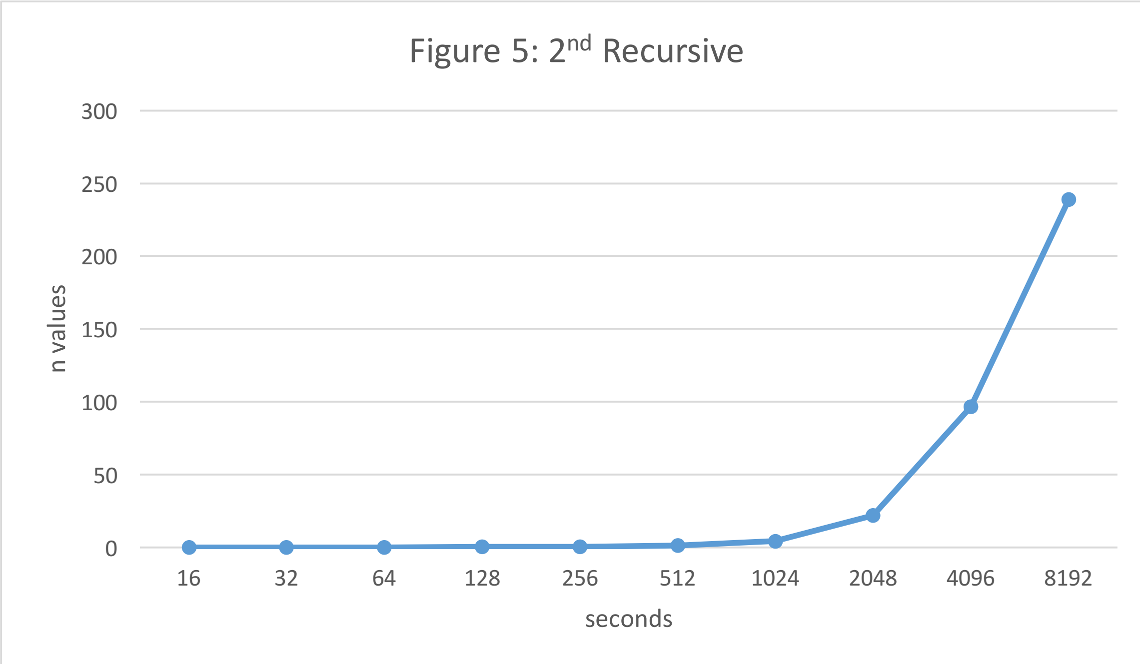
For figure 5: 3rd recursive implementation (for m = 8):

n	Seconds
16	0.0334589
32	0.0444312
64	0.0965519
128	0.118872
256	0.257943
512	0.714743
1024	1.95797
2048	6.79686
4096	27.7258
8192	213.133



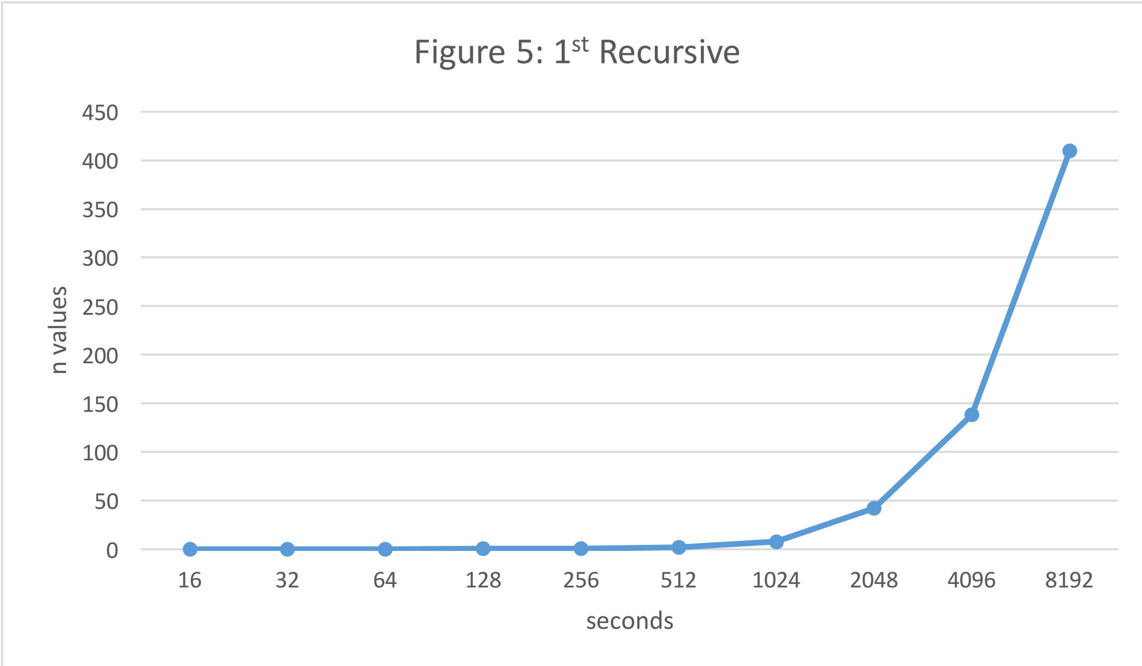
For figure 4: 2nd recursive implementation (For m = 8):

n	Seconds
16	0.0759211
32	0.0815771
64	0.102882
128	0.252649
256	0.296818
512	1.28552
1024	4.53154
2048	22.1703
4096	96.6611
8192	238.829



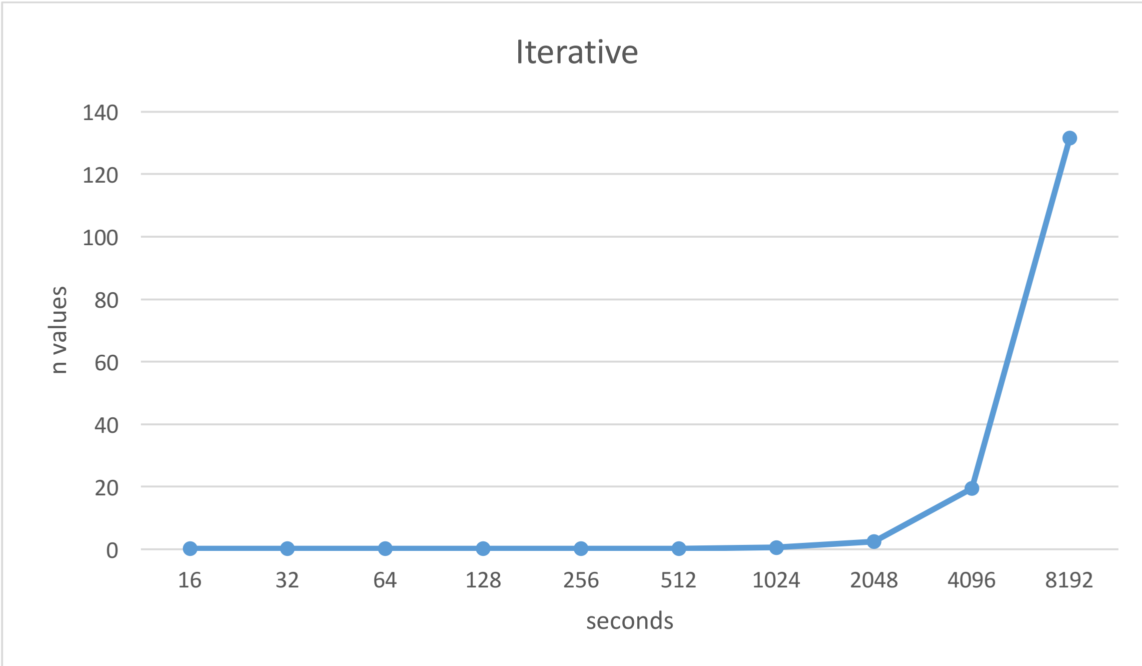
For figure 2: 1st recursive implementation (For m = 8):

n	Seconds
16	0.0905531
32	0.1228729
64	0.274769
128	0.4739841
256	0.921766
512	2.283593
1024	7.9083
2048	42.2005
4096	138.0546
8192	409.597



For iterative implementation:

n	Seconds
16	0.18602
32	0.168832
64	0.163706
128	0.199946
256	0.224751
512	0.311379
1024	0.728141
2048	2.54009
4096	19.5146
8192	131.488



Scalability plot for OMP:
n = 2048 (because n = 8192 was taking a long time)

No. of threads	rec 3	rec2
1	56.46308	67.78456
2	33.70528	31.6793
3	31.37058	29.3446
4	30.24488	28.2189
5	19.43668	17.4107
6	17.13428	15.1083
7	15.48588	13.4599
8	13.53478	11.5088
9	13.20998	11.184
10	12.09008	10.0641
11	12.09778	10.0718
12	10.50049	8.47451
13	10.69944	8.67346
14	10.81969	8.79371
15	8.82371	10.378
16	6.79686	8.82284

We couldn’t do for rec 1 due to long execution time on Stampede.

i)

Yes, we can improvise the parallelism even further by using extra space to store intermediate values as shown below:

$X_{ij} = \min\{X_{ij}, U_{i1}+V_{1j}, U_{i2}+V_{2j}\}$ for $i,j \in (1,2)$

Since both the parts $U_{i1}+V_{1j}$ and $U_{i2}+V_{2j}$ try to update X_{ij}, they cannot be parallelized. As such, if we introduce some extra space to hold intermediate values of one set, we can further parallelize our algorithm at the cost of this extra space, that is, we can evaluate $D_{FW}(X_{11},U_{11},V_{11})$ and $D_{FW}(X_{11},U_{12},V_{21})$ simultaneously as $D_{FW}(X_{11},U_{11},V_{11})$ and $D_{FW}(T_{11},U_{11},V_{11})$ and then at little extra cost we can update the final value at X_{11} by finding the appropriate minimum value from the current X_{11} and T_{11}. Thus, for instance, our new $D_{FW}(X,U,V)$ will look as shown below:

```
D_FW(X, U, V)

  if X is an m × m matrix then D_loop-FW(X, U, V)

  else

      parallel:D_FW(X_11,U_11,V_11),D_FW(X_12,U_11,V_12),D_FW(X_21,U_21,V_11),
D_FW(X_22,U_21,V_12),   D_FW(T_11,U_12,V_21),   D_FW(T_12,U_12,V_22),   D_FW(T_21,U_22,V_21),
D_FW(T_22,U_22,V_22)

  parallel for i ← 1 to n do

        parallel for j ← 1 to n do

             X_ij = min{X_ij, T_ij}
```

The **span** of the new implementation for the $D_{loop\text{-}FW}$ will be then as below:

$$T_\infty(n) = \begin{cases} \theta(1), & if\ n = 1 \\ T_\infty\left(\frac{n}{2}\right) + \theta(\log n), & otherwise \end{cases} = \theta(log^2 n)$$

Thus, the **parallelism** would be:

$$\frac{T_1(n)}{T_\infty(n)} = \theta(n^3/log^2 n)$$

while the additional space would be:

$$S_\infty(n) = \begin{cases} \theta(1), & if\ n = 1 \\ 8S_\infty\left(\frac{n}{2}\right) + \theta(n^2), & otherwise \end{cases} = \theta(n^3)$$