QUESTION-1

All the experiments have been done on Stampede2 and Performance measuring library used is PAPI(v5.6.0), Here is the specification of a Compute Node-

Model:	Intel Xeon Phi 7250 ("Knights Landing")
Total cores per KNL node:	68 cores on a single socket
Hardware threads per core:	4
Hardware threads per node:	68 x 4 = 272
Clock rate:	1.4GHz
RAM:	96GB DDR4 plus 16GB high-speed MCDRAM. Configurable in two important ways; see "Programming and Performance: KNL" for more info.
Cache:	32KB L1 data cache per core; 1MB L2 per two-core tile. In default config, MCDRAM operates as 16GB direct-mapped L3.
Local storage:	All but 504 KNL nodes have a 107GB `/tmp` partition on a 200GB Solid State Drive (SSD). The 504 KNLs originally installed as the Stampede1 KNL sub-system each have a 32GB `/tmp` partition on 112GB SSDs. The latter nodes currently make up the `development` and `flat-quadrant` queues. Size of `/tmp` partitions as of 14 Nov 2017.

PAPI version : 5.6.0.0
Operating system : Linux 3.10.0-693.17.1.el7.x86_64
Vendor string and code : GenuineIntel (1, 0x1)
Model string and code : Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz (87, 0x57)
CPU revision : 1.000000
CPUID : Family/Model/Stepping 6/87/1, 0x06/0x57/0x01
CPU Max MHz : 1600
CPU Min MHz : 1000
Total cores : 272
SMT threads per core : 4
Cores per socket : 68
Sockets : 1
Cores per NUMA region : 272
NUMA regions : 1
Running in a VM : no
Number Hardware Counters : 5
Max Multiplex Counters : 384
Fast counter read (rdpmc): no

Question 1(a)

Algorithm	R	Time Taken(seconds)	Time Taken (minutes)
Iter-MM-**ijk**	10	189	3
Iter-MM-**ikj**	10	32	1
Iter-MM-**jik**	10	186	3
Iter-MM-**jki**	10	198	3
Iter-MM-**kij**	10	32	1
Iter-MM-**kji**	10	194	3

Question 1(b)

Algorithm	R	L1	Order of 10	L2	Order of 10
Iter-MM-**ijk**	10	1074153848	10	1074124486	10
Iter-MM-**ikj**	10	34040288	8	34036210	8
Iter-MM-**jik**	10	1076612259	10	1076608904	10
Iter-MM-**jki**	10	1696118704	10	1696048562	10
Iter-MM-**kij**	10	34410231	8	34388883	8
Iter-MM-**kji**	10	1628687365	10	1628677030	10

Question 1(c)

Algorithm Iter-MM-**ikj (2nd) and** Iter-MM-**kij (5th)** perform better than any other permutation of i,j and k because of the fact that matrices are stored internally in row major format i.e. a complete row is stored contiguously in a block and then the next row. This reason is major factor in deciding the number of L1, L2 and L3 cache misses, because if we're able to utilize the whole row fetched in a block for a matrix rather than fetching a new row each time then we can reduce the number of cache misses drastically.

That is what is happening in both of the top 2 algorithms i.e. Iter-MM-**ikj (2nd) and** Iter-MM-**kij (5th).**

Iter-MM-ikj(Z, X, Y, n)

1. for i←1to ndo

2. for k←1to ndo

3.for j←1to ndo

4.Z[i,j] ← Z[i,j] + X[i,k] × Y [k,j]

Here X[i,k] is fixed for all possible j values and further the inner loop is on k i.e. first all the column of the fetched ith row will be used and won't be fetched again afterwards.

For Y[k,j], while it is independent of outer loop i(1), the inner loop(2) is on k and the innermost loop on j allows to access all columns of kth row at the same time. While for a different value of i's, same rows will be fetched but it is still better than an ijk permutation.

Similarly for

Iter-MM-kij(Z, X, Y, n)

1. for k←1to ndo

2. for i←1to ndo

3. for j←1to ndo

4. Z[i,j] ← Z[i,j] + X[i,k] × Y [k,j]

Question 1(d)

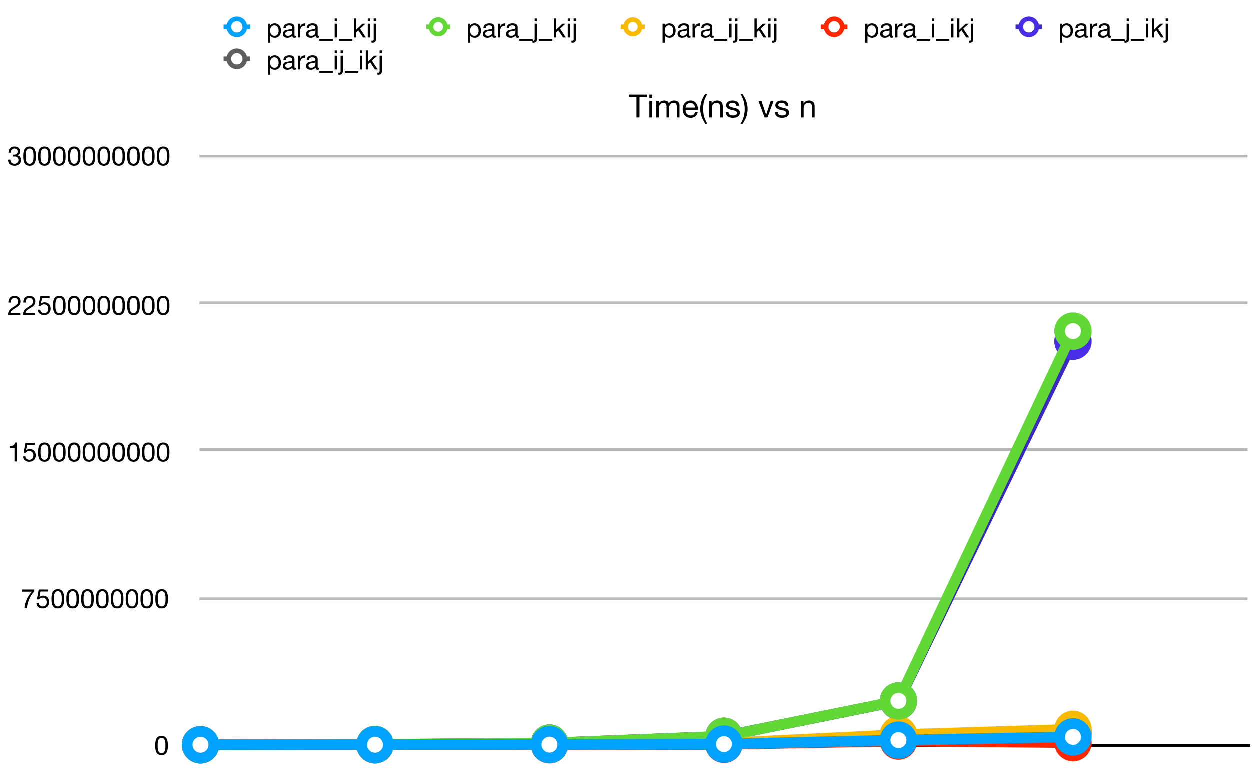
There are only 3 possible ways for each permutation to parallelize the matrix multiplication permutations without compromising with the correctness -

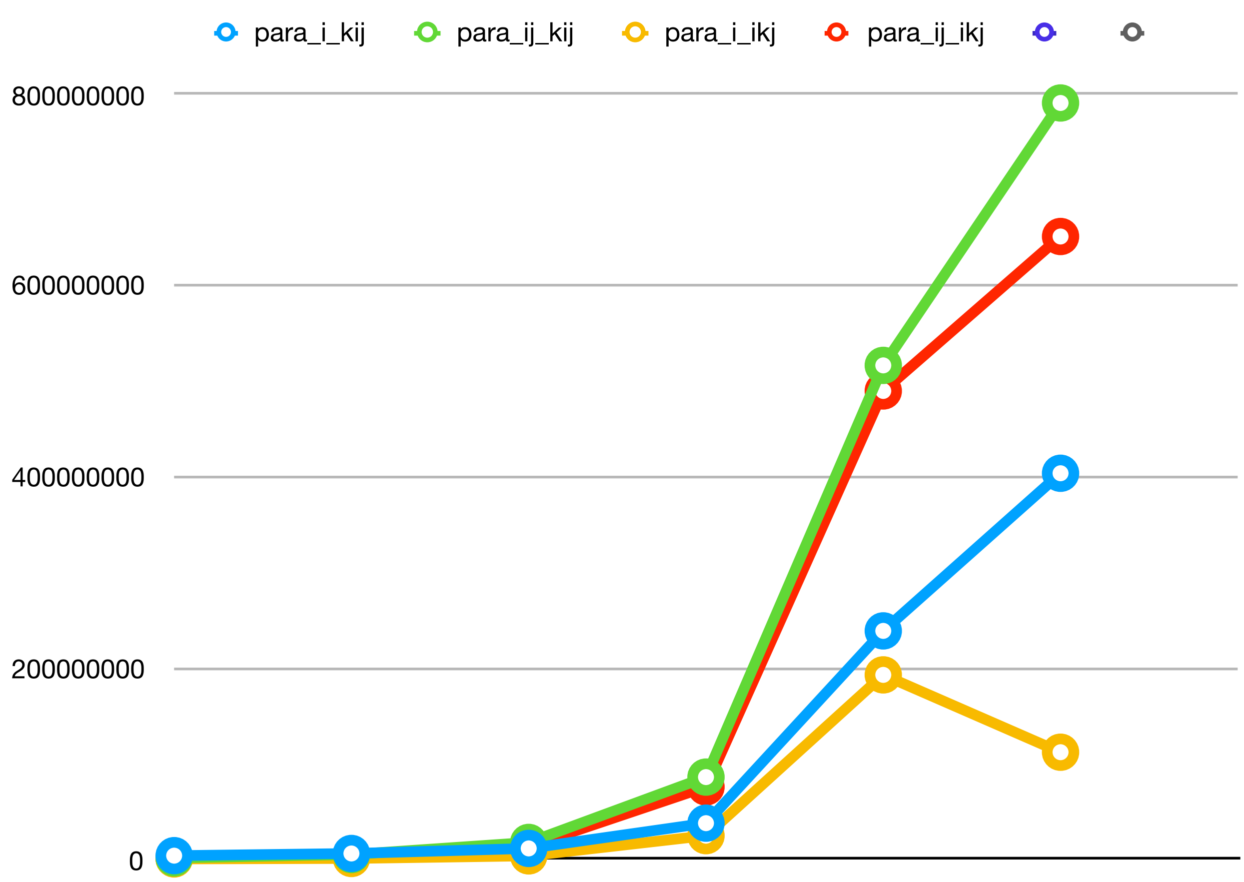
1. Parallelize only i loop
2. Parallelize only j loop
3. Parallelize both i and j

It should be kept in mind that we cannot parallelize on the k loop because that would be an invitation to concurrent accesses and writes and thus assigning garbage values.

Time taken(in nanoseconds) by different parallel implementations for different values of r

Algorithm| R->	4	5	6	7	8	9
para_i_kij	3662085	5936005	11227592	37649445	238844120	403781291
para_j_kij	2534892	19112616	89542474	435432871	2241948762	21098301302
para_ij_kij	1305565	4840515	17501096	85863941	516654586	791221446
para_i_ikj	220800	706160	3415766	24580551	192808845	111894536
para_j_ikj	2815630	19140541	89845773	436665026	2235747496	20583744041
para_ij_ikj	626998	2717344	13104686	75590893	489901900	651515681





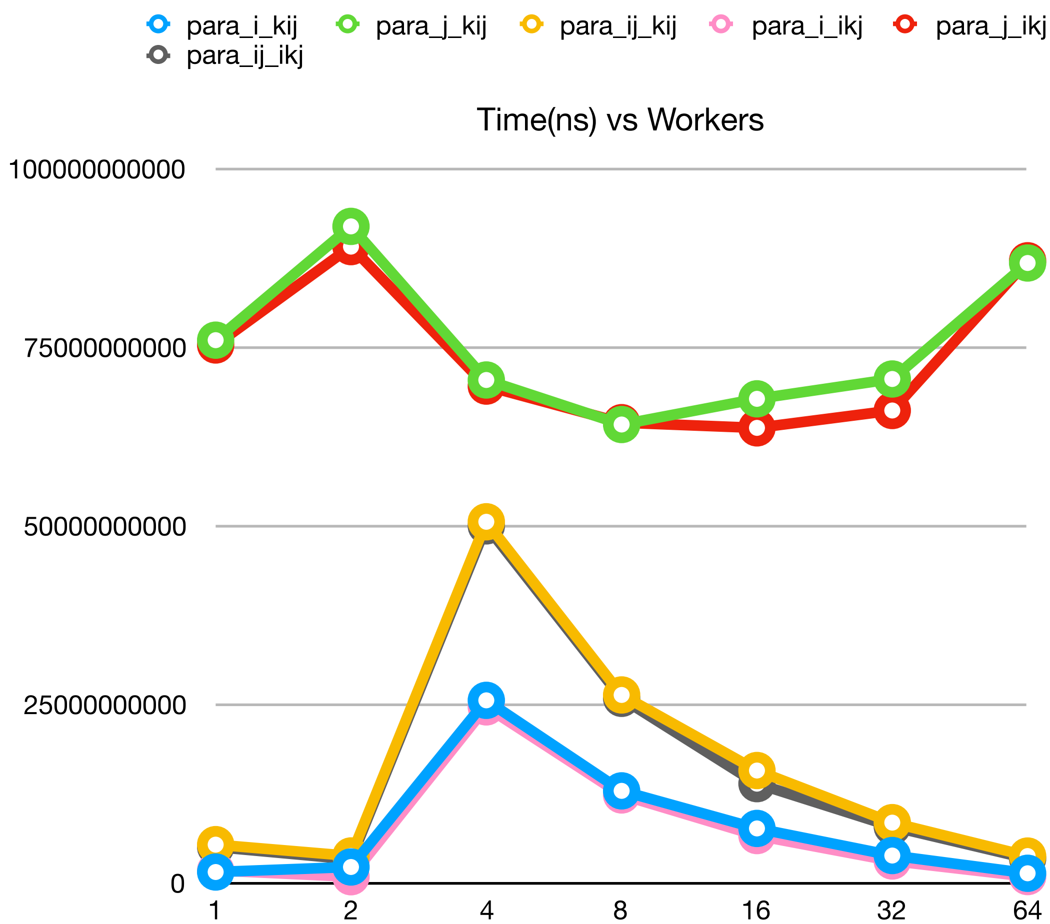
Zoomed version of above graph

Performance-1

para_i_ikj
para_i_kij
para_ij_ikj
para_ij_kij
para_j_ikj
para_j_kij

Question 1(e)

r=10



Question 1(f)

The performance of all 6 algorithms, increase initially due to parallelization overhead and keep decreasing due to the parallel computation, as the number of workers increase the time taken to spawn threads is more than compensated by parallel computations performed.

The best performance is observed in both cases when we parallelize the i loop. This empirical observation can be accounted due

para-i-ikj(Z, X, Y, n)

```
1.cilk_for i←1to ndo
    2. for k←1to ndo
        3.for j←1to ndo
            4.Z[i,j] ← Z[i,j] + X[i,k] × Y [k,j]
```

When we are Parallelizing inner loops i.e. j in case of (j_kij, j_ikj), the parallelization overhead is very high and parallelizing the inner most loop is thus counter productive.

When we parallelize the outer-most loop on i.e. on i for ikj then the overhead of parallelization is more than compensated by the computational speedup as the work is distributed more evenly and the granularity is perfect.

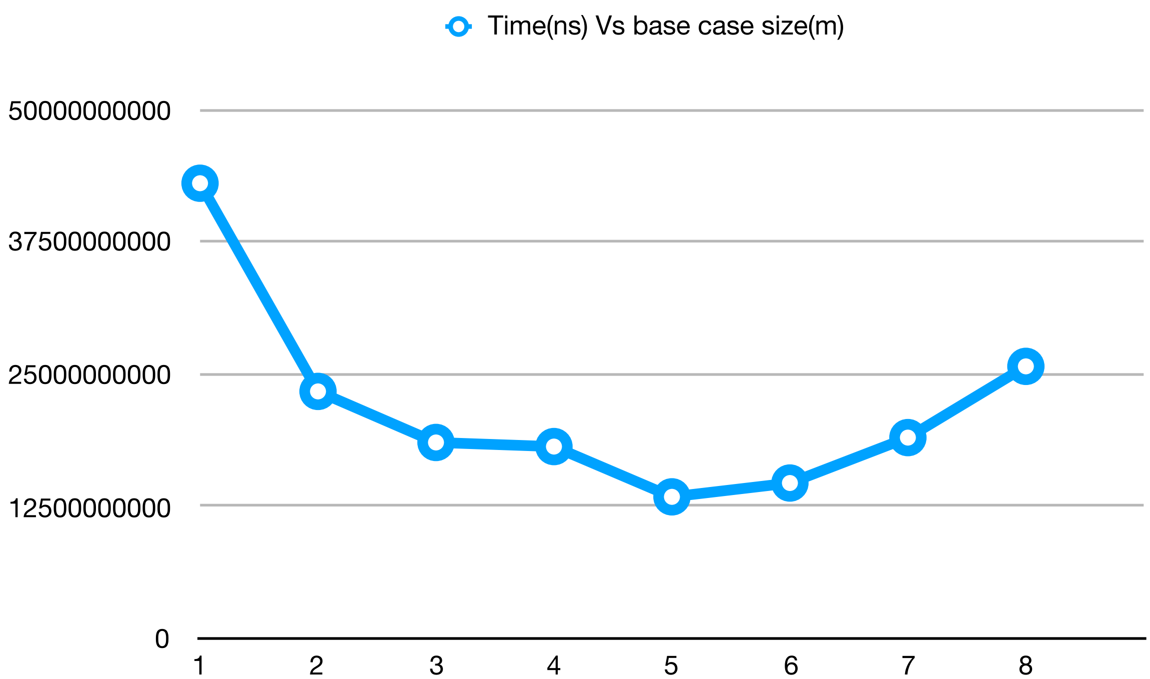
The same applies for the case when we parallelize ij, as parallelizing 2 loops can increase the run-time rather than decreasing it.

Performance

para_i_ikj
para_i_kij
para_ij_ikj
para_ij_kij
para_j_ikj
para_j_kij

Question 1(g)

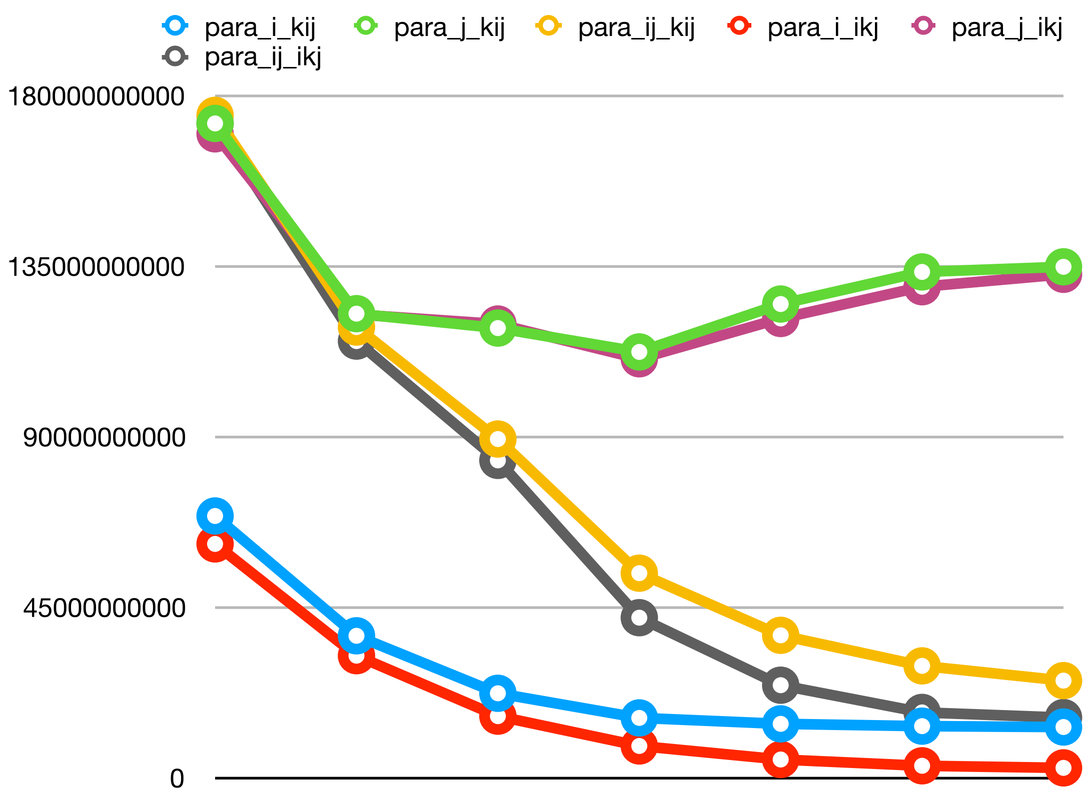
We can see that the optimal base case is when **m=5 (32x32).**



Question 1(h)

Time (ns) vs Number of Cores

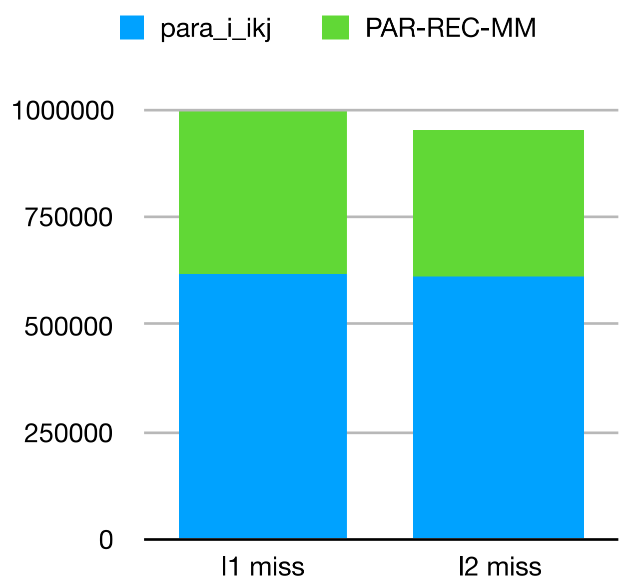
	1	2	4	8	16	32	64
para_i_kij	6926815071	3767048262	2251284241	1602006341	1445152854	1384783767	1361637882
para_j_kij	1727761690	1226174900	1188160399	1125263852	1251544574	1336190158	1349717444
para_ij_kij	1749092352	1190830836	8955564621	5418297867	3779986355	2974070767	2585573375
para_i_ikj	6190016976	3242698353	1648176361	8619543567	5049804625	3390832004	2844535390
para_j_ikj	1700282896	1224069952	1198828261	1107160014	1213409216	1297109214	1329926589
para_ij_ikj	1727835834	1153824524	8392540269	4244772482	2464489885	1746720051	1610499868



If we compare performance with 1(d) and 1(e) we observe that ikj permutation ranked higher than any other permutations. Parallel for i-variable has performed better than any other parallel for permutation because of the fact that in ikj it is the outer loop and that's why it scales well and balances the load well and the overhead of parallelism is more than compensated by the computation that is done in parallel.

Question 1(i)

ALGOTRIHM	L1-miss	L2-miss
para_i_ikj	465344	459129
PAR-REC-MM	379072	343726

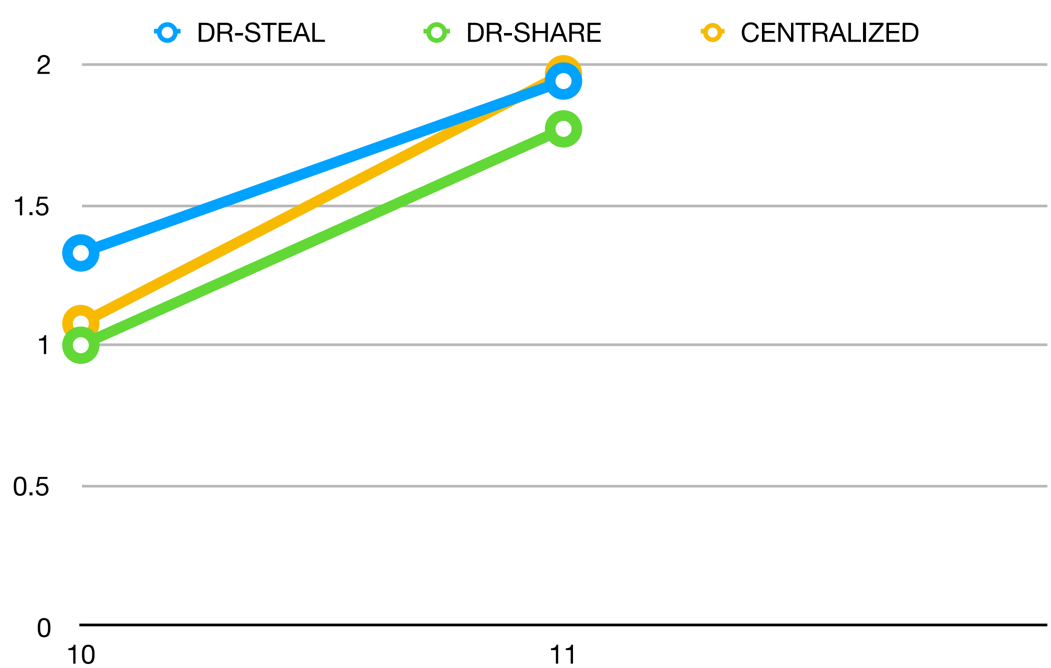


Question 2(a)

We are using 64 cores for this execution.

r->	Time (ns)	GFLOPS r=10	Time (ns)	GFLOPS(r=11)
DR-STEAL	1616163175	1.32875422557503	8849495370	1.94133885218361
DR-SHARE	2147880204	0.999815373315857	9700616759	1.77100792772387
CENTRALIZED	1993737460	1.07711456050989	8731967635	1.9674682616938

GFLOPS vs r



We observed that the following order for schedulers in terms of time taken-

DR-STEAL<CENTRALIZED<DR-SHARE

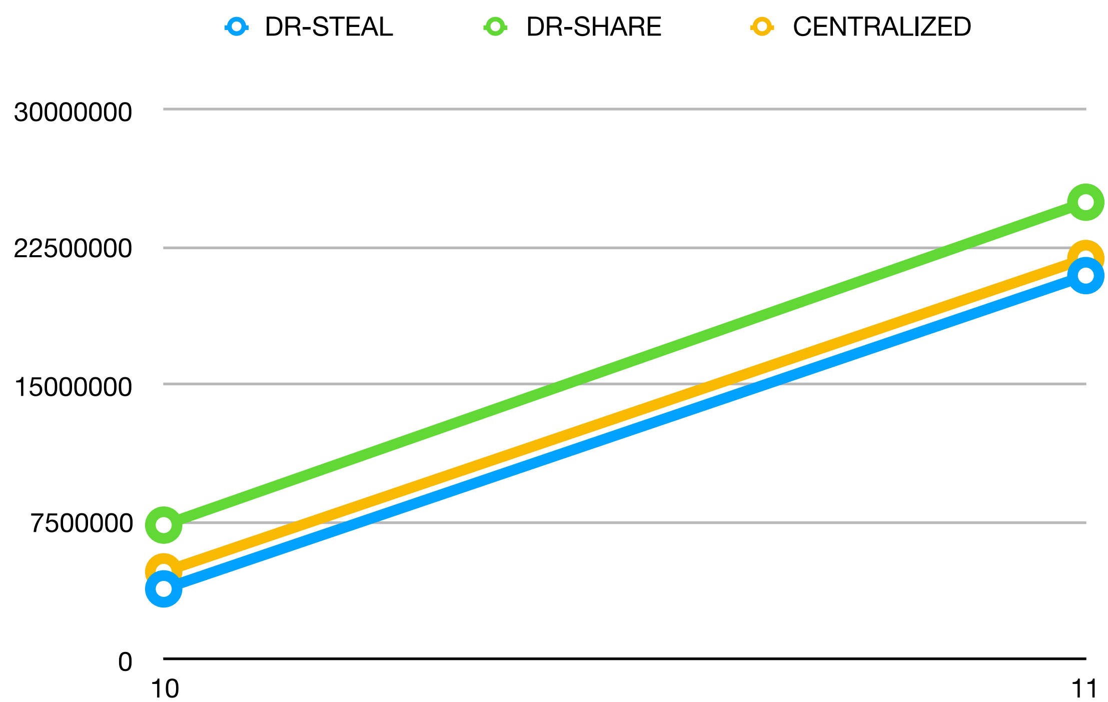
Thus we can say DR-STEAL performs better than other 2 schedulers in terms of time and centralized scheduler may work well for smaller value of n but as the value of n increases, it is not able to scale well.

Theoretically, DR-SHARE should have better performance than Centralized but DR-SHARE is having worse performance because it has to generate random number every time nw process is added. This will add some extra overhead to the system.

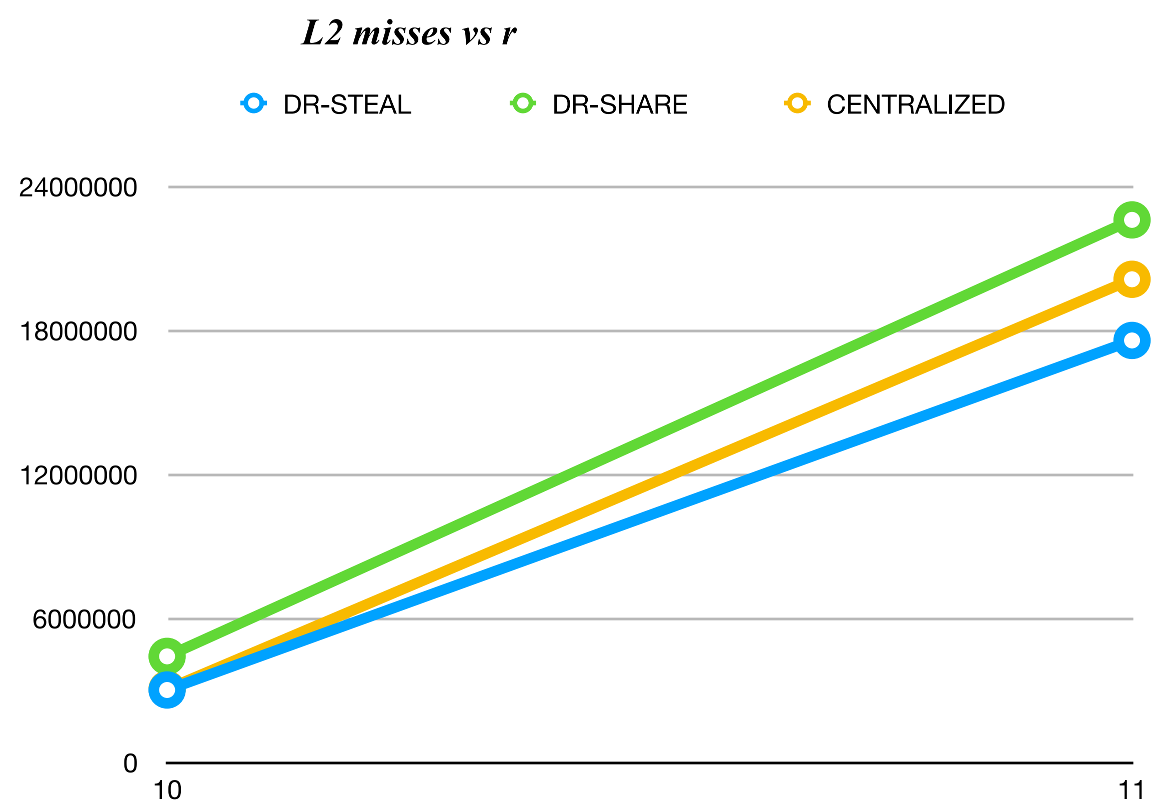
Question 2(b)

L1-miss->	R=10	R=11
DR-STEAL	3876366	20964191
DR-SHARE	7355445	24967520
CENTRALIZED	4806081	21897237

L1 misses vs r



L2-miss->	R=10	R=11
DR-STEAL	3043236	17576177
DR-SHARE	4441099	22588972
CENTRALIZED	3075357	20119335



L1 and L2 misses are in line with the time required for the schedulers to finish. Empirically we can see that steal has the lowest running time among all the schedulers and thus it must have incurred lower cache misses. Lower cache misses in DR-STEAL algorithm can be accounted due to the fact that the height of a task signifies how big it is. Since towards the top of a dequeue the task are larger and has the higher probability of being stolen, the grain size is such that it does not incur cache misses because all the data required is already in the memory.

While centralized may seem to perform nearly as well as DR-share but scalability is an issue with it. As we increase size of the matrix and the number of cores it performs begins to drop.

DR-Share takes more time than DR-steal because of the fact that load balancing is a problem in DR-share. In DR-steal, processors who are looking for work steal while in DR-share, processors randomly puts work in another processors dequeue who might be busy executing several other task.

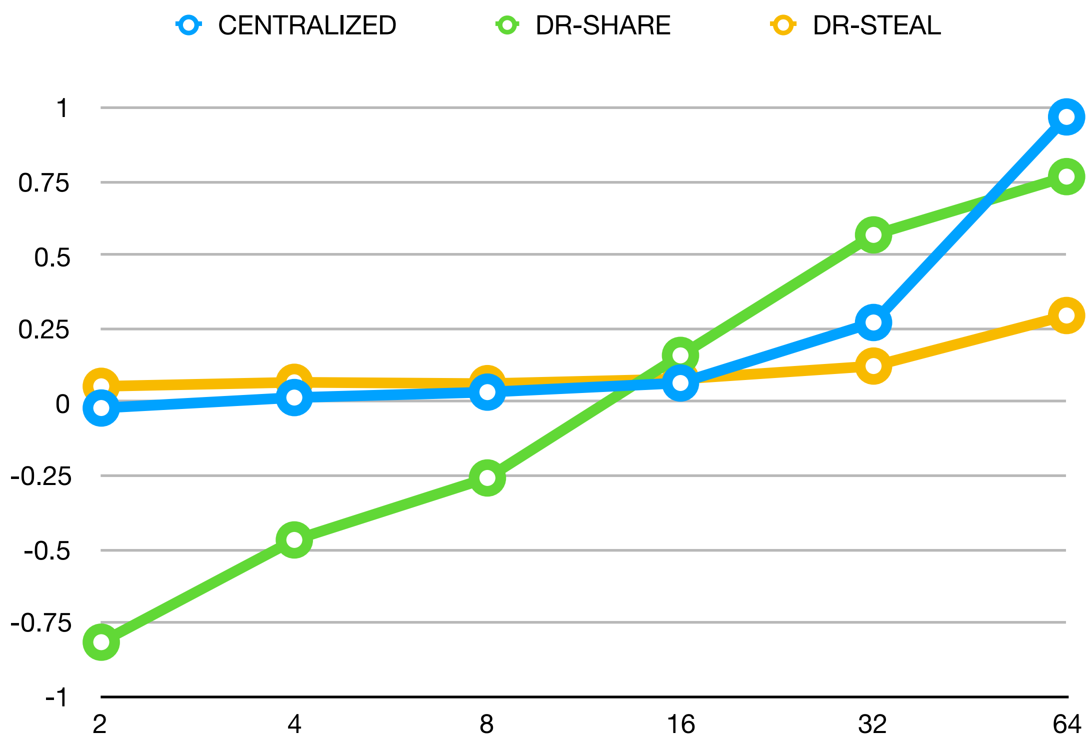
Question 2(c)

Intuitively, as the number of cores increase Centralized scheduler should not be able to scale well while DR-Steal should be able to scale well and DR-share's performance should also drop due to load imbalance.

Time in ns

	CENTRALIZED	DR-SHARE	DR-STEAL
1	48028880110	399472626359	394080948449
2	23545074042	109947595849	208279243109
4	12197958215	67994376362	105659302130
8	6215805435	39697145610	52573080702
16	3211240141	29691978078	26737760839
32	2059057707	28969952027	14040115427
64	25483310086	26829715893	8731967635

cores|1-Efficiency->	CENTRALIZED	DR-SHARE	DR-STEAL
1	0	0	0
2	-0.019934786026272	-0.816650119879057	0.0539601004725101
4	0.0156368946456503	-0.468770829782377	0.0675668386392171
8	0.0341380410743182	-0.257875787479698	0.0630163213119251
16	0.065219393420942	0.159131834132109	0.0788286488771035
32	0.271073123237386	0.569087323234636	0.122868347980338
64	0.970551264761675	0.767356471020706	0.294830778479446



Plot of (1-efficiency) for the 3 algorithms

From empirical results we can see that while that as the number of cores are increased, efficiency decreases almost linearly for DR-Share and decreases quickly for centralized scheduler thus 1-efficiency i.e. overhead of scaling increases for both of them. While this overhead does not seem to decrease in case of DR-steal scheduler, as the number of Cores increase it remains **constant**.

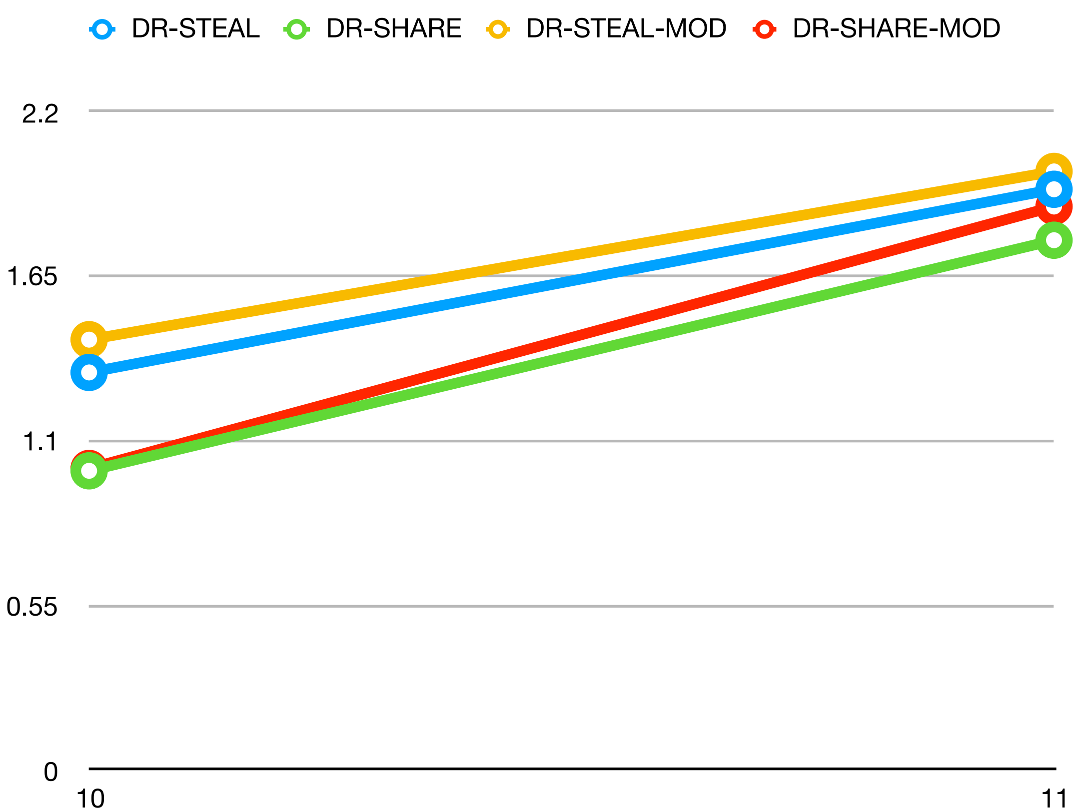
Thus our results confirm with the general intuition that DR-steal scheduler scales well and centralized does not scale well as the number of cores increase.

Question 2(d)

DR-STEAL-MOD is performing slightly better than DR-STEAL and DR-SHARE-MOD is also performing slightly better than DR-SHARE.

Since we are taking two random numbers, we can choose better queue to add or steal task. Theoretically known as the power of 2, taking 2 dequeues reduces the upper bound on load per bin from O(lnp/lnlnp) to O(lnlnp/2) which ensures that no bin has more than O(lnlnp/2) tasks. Thus ensuring better load balancing.

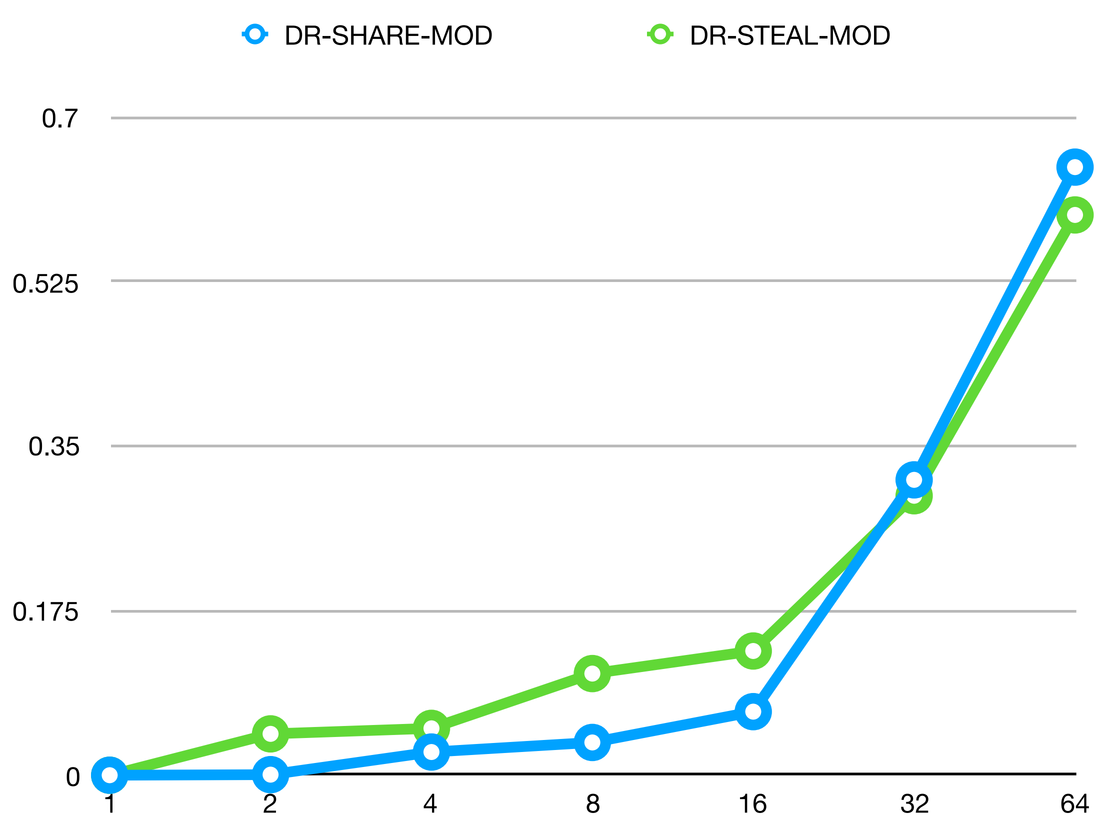
r->	Time (ns)	GFLOPS r=10	Time (ns)	GFLOPS(r=11)
DR-STEAL-MOD	1492918298	1.43844686670188	8585441260	2.00104673292005
DR-SHARE-MOD	2130984009	1.00774273243268	9120370385	1.88368108517338



N-workers vs time (ns)

	DR-SHARE-MOD	DR-STEAL-MOD
1	47506127240	46036607599
2	23767178725	24076828877
4	12176691820	12112682334
8	6152561329	6451868350
16	3184341560	3315193225
32	2164359169	2046916338
64	2100638432	1779573045

cores|1-Efficiency->	DR-SHARE-MOD	DR-STEAL-MOD
1	0	0
2	0.000593890640673833	0.0439644723525523
4	0.0246503742097662	0.0498263239807673
8	0.0348302784061529	0.108076042829516
16	0.0675833931269609	0.132090415352034
32	0.314084973735706	0.297165223237986
64	0.64663921843119	0.595789565505374



Empirically we observe that DR-steal-mod performs better than DR-share-mod and we also observed both modified versions performed better than DR-steal and DR-share correspondingly, due to better load balancing the runtime went down.

Question-3a.

Probability of stealing from i^{th} deque = $\frac{1}{p}$

Probability of not stealing from i^{th} deque = $1 - \frac{1}{p}$

Probability of not checking from i^{th} deque k time = $(1 - 1/p)^k$

Probability of not checking from i^{th} deque pk time = $(1 - 1/p)^{pk}$

Probability of not checking from i^{th} deque pk time (using approximation $(1-1/p)^p$) = $(1 - 1/e)^k$

$k = c^* \ln p + d$

We know that when p is large,

$= (1 / e)^{c^* \ln p + d}$

$= 1/(e^{c \ln p} e^{d})$

$= 1/(p^c e^d)$

Few then take a union bound of these probabilities over all subsets ${}^{P}C_1$,

$= {}^{P}C_1 / p^c e^d$

$= p / p^c e^d$

$= 1 / p^{c-1} e^d <= 1 / p^c$

So, number of steal attempts required to check each sequence with high probability = p*k

$= p * (c * \ln p)$

$= O(p * \ln p)$

If we take c >= 2,
As $1 - (1/p^2)$ is a high probability constrain.

Question-3b.

In the part 3a we calculated the number of steal attempts required to check every deque in the system with high probability. High probability bound ensures that if we try O(plnp) time, in most cases the processor would have checked all other deques but there is a very small probability that one or more deque(s) are not selected even once while looking for work. That's why O(plnp) failed steal attempts cannot guarantee that the entire system has run out of work.

Question-3c.

If we use a premature termination criteria i.e. a processor tries O(plnp) times before terminating and terminates if it does not find work in 2plnp consecutive attempts even though there might be some work left on an unchecked processor.

Even if all the idle processors terminate like this prematurely, the processor(s) which have twork will still complete it before terminating. Thus it is ensured that some processor will complete the work and terminate only after its deque is empty, therefore ensuring that all work in the system will still be completed

Question-3d.

We are given p consecutive ensues in the systems and total number of queues are also p.
Then, we know that probability of selecting an i^{th} dequeue $= \frac{1}{p}$

probability of not selecting an i^{th} dequeue $= 1 - \frac{1}{p}$

So, Probability of that the i^{th} dequeue has k enqueues out of p,

$C_k^p(\frac{1}{p})^k(1-\frac{1}{p})^{p-k}$

Pr(Bi) = Pr(ith dequeue has at least k enqueues) = $\sum_{i=k}^{p} C_i^p(\frac{1}{p})^i(1-\frac{1}{p})^{p-i}$

We take union bound over all bins,

Pr(any dequeue has at least k enqueues) <= $\sum_{i=1}^{p} Pr(B_i)$

For this we first need an upper bound on Pr(Bi)

By simple approximation, for i<=n
$(\frac{n}{i})^i <= C_i^n <= (\frac{ne}{i})^i$

$$Pr(B_i) \leq \sum_{i=k}^{n} (\frac{n*e}{i})^i (\frac{1}{n})^i$$
$$Pr(B_i) = (\frac{e}{k})^k * (1 + \frac{e}{k} + (\frac{e}{k})^2 + \ldots)$$
$$Pr(B_i) = (\frac{e}{k})^k * \frac{1}{1 - e/k}$$

Pr(ith dequeue has at least k enqueues) <= $(\frac{e}{k})^k$

Let k = 3 * $\frac{lnp}{lnlnp}$

Pr(ith dequeue has at least k enqueues) <= $(\frac{e}{k})^k$

$$<= (\frac{e}{\frac{3lnp}{lnlnp}})^{3*\frac{lnp}{lnlnp}}$$

$$<= (\frac{e}{\frac{3lnp}{lnlnp}})^{3*\frac{lnp}{lnlnp}}$$

$$< = (\frac{elnlnp}{3lnp})^{3*\frac{lnp}{lnlnp}}$$
$$< = exp(\frac{-3lnp + (3*(lnlnlnp)*lnlnp)}{lnlnp}$$

When value of p is large then with high probability we can say that,

Pr(ith dequeue has at least k enqueues) <= $(1/p)^2$

Therefore, the probability that no bin contains more than k balls (union bound on all the bins) =

Pr(no dequeue has more than $\frac{3lnp}{lnlnp}$ enqueues) >=

1-Pr(any dequeue has at least k enqueues) = 1- $\sum_{i=1}^{p} Pr(B_i)$

=> $1-p(1/p)^2$

=> $1-1/p$

Therefore each deque undergoes atmost O($\frac{lnp}{lnlnp}$) enqueues.

Question-3(e)i.

Let ni be the total number of deques with i-tasks, then the
Pr(selecting deque with i-tasks) = $f_i = n_i/p$

Since these i.i.d events, the probability of selecting another deque with i-tasks is again the same.Therefore,
Pr (selecting 2 deques with i-tasks) = f_i^2

If we take union bound over all p enques, then
Expected number of queues with i+1 tasks-

$$f_{i+1} = p * f_i^2$$

$$f_{i+1} <= f_i^2$$

Question-3(e)ii.

$$\frac{n_{i+1}}{p} <= f_i$$

$$f_{i+1} <= (\frac{n_i}{p})^2$$

$$f_2 <= (f_1)^2$$

$$f_3 <= (f_1)^{2^2}$$

Therefore,

$$f_i <= (f_1)^{2^{i-1}}$$

$$f_i <= (\frac{n_1}{p})^{2^{i-1}}$$

$$f_i <= (\frac{n_1}{p})^{2^{i-1}}$$

$$f_i <= (\frac{p/2}{p})^{2^{i-1}}$$

$$f_i <= (\frac{p/2}{p})^{2^{i-1}}$$

$$f_i <= (\frac{1}{2})^{2^{i-1}}$$

$$f_i <= \frac{1}{2^{2^{i-1}}}$$

Question-3(e)iii

$$f_i <= \frac{1}{2^{2^{i-1}}}$$

when i = loglogn

$$f_{loglogn} <= \frac{1}{2^{2^{(loglogn)-1}}} = \frac{1}{2^{logn}} = \frac{1}{n}$$

Thus we can say that having rank greater than or equal to loglogn is a low probability event. Therefore, no task is likely to have a rank larger than loglogn during p consecutive enqueue attempts.