CSE-613 Assignment - 1

Akshat Singhal (111496103), Cheuk On Chung(110386696), Iti Inani(111323922)

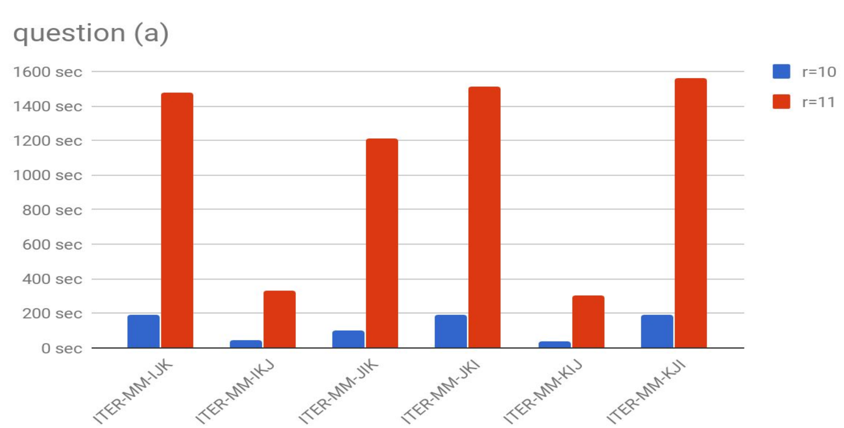
Task# 1

1(a)

Run each implementations in size of 2^{10} and 2^{11} and find out r is 10, which is the largest integer such that none of the implementations takes more than five minutes to perform the multiplication.

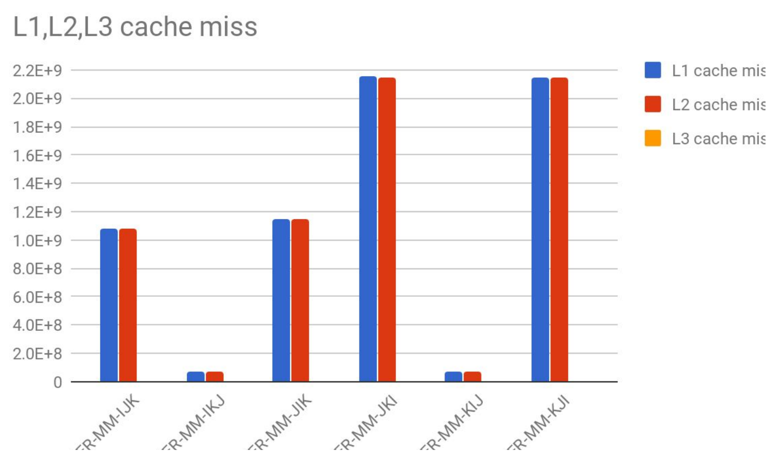
The unit is second. For example, when r is 10 (size is 2^{10} , the time of mmIJK is 189 second.

	ITER-MM-IJK	ITER-MM-IKJ	ITER-MM-JIK	ITER-MM-JKI	ITER-MM-KIJ	ITER-MM-KJI
r=10	189	42	102	193	36	193
r=11	1479	328	1210	1511	302	1561



1(b) We run each algorithm in size of 2^{10} ,as defined in question 1(a) on SKX Node
The L1 L2 L3 cache is listed as below.

	ITER-MM-IJK	ITER-MM-IKJ	ITER-MM-JIK	ITER-MM-JKI	ITER-MM-KIJ	ITER-MM-KJI
L1 cache miss	1075959893	67355599	1143832937	2152046320	69298504	2150252916
L2 cache miss	1074941552	67380072	1142740980	2149406438	71974881	2147550739
L3 cache miss	206224	8746	466746	2216776	2423	145362



1(c) As can be seen from the data above, the running time of ITER-MM-IKJ and ITER-MM-KIJ is far less than the others. The cache misses of ITER-MM-IKJ and ITER-MM-KIJ is also far less than the others. Therefore, when the cache misses increase, the running time increases.

The reason that the two fastest has less cache is because of Locality of Reference[1].
As we can see from the equation: Z[i, j] ← Z[i, j] + X[i, k] × Y [k, j], for the MM-IKJ and MM-KIJ, The read and writes of Z[i,j] are in cache, the reads of Y[k,j] are in cache. And the read of X[i,k] can be factored out of the inner loop. Therefore, they do not have cache miss in the inner loop.

[1] https://en.wikipedia.org/wiki/Locality_of_reference#Matrix_multiplication

1(d) The two fastest implementation is ITER-MM-IKJ and ITER-MM-KIJ
There are 7 types of parallelization for each implementation. Only 6 of them are correct parallelization since one of them causes race condition which slows the program.

For ITER-MM-IKJ,
The first is mmIKJ-1, which only parallelize the loop I.
The second is mmIKJ-2, which only parallelize the loop K.
The third is mmIKJ-3, which only parallelize the loop J.(this is a bad slow one)
The 4th is mmIKJ-4, which parallelize the loop I and loop K.
The 5th is mmIKJ-5, which parallelize the loop I and loop J.
The 6th is mmIKJ-6, which parallelize the loop K and loop J.
The 7th is mmIKJ-7, which parallelize the loop I, look K and loop J.

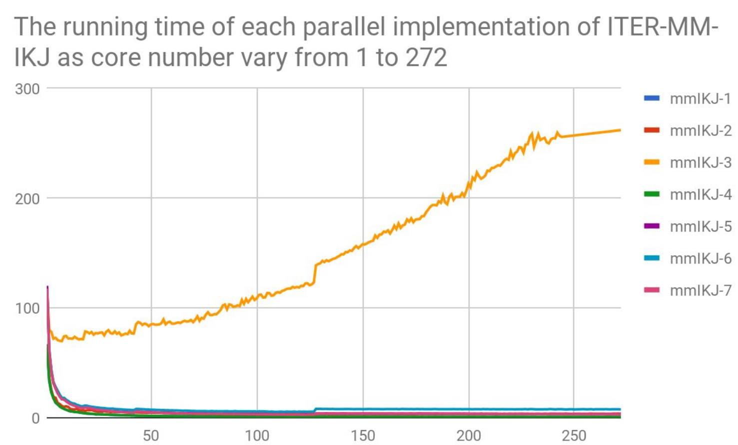
For ITER-MM-KIJ ,
The first is mmKIJ-1, which only parallelize the loop I.
The second is mmKIJ-2, which only parallelize the loop K.
The third is mmKIJ-3, which only parallelize the loop J. (this is a bad slow one)
The 4th is mmKIJ-4, which parallelize the loop I and loop K.
The 5th is mmKIJ-5, which parallelize the loop I and loop J.
The 6th is mmKIJ-6, which parallelize the loop K and loop J.
The 7th is mmKIJ-7, which parallelize the loop I, look K and loop J.

Now run each such parallel implementation on all cores by varying size (n from 2^4 to 2^s) , where s is the largest integer such that none of the parallel implementations takes more than a minute to perform the multiplication.

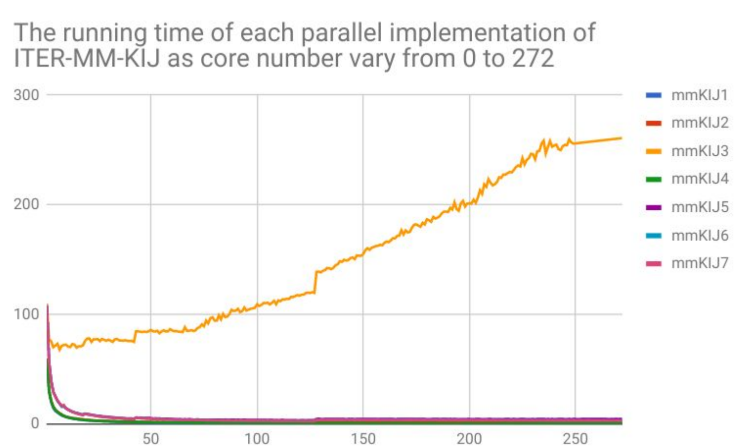
The unit is second. For example, when s=4 (size is 2^4), the running time of mmIKJ-1 is 0.15 second

	s=4	s=5	s=6	s=7	s=8	s=9	s=10
mmIKJ-1	0.15	0.0038	0.0019	0.0043	0.01	0.072	0.57
mmIKJ-2	0.0018	0.0073	0.023	0.056	0.21	0.8	3.33
mmIKJ-3	0.0085	0.056	0.35	1.85	10	56	282
mmIKJ-4	0.0051	0.015	0.0023	0.003	0.011	0.075	0.57
mmIKJ-5	0.0039	0.016	0.0079	0.0148	0.054	0.399	3.12
mmIKJ-6	0.01	0.056	0.04	0.0989	0.37	1.57	7.22
mmIKJ-7	0.0093	0.035	0.00404	0.0091	0.05	0.388	3.06
mmKIJ-1	0.0014	0.00116	0.0023	0.0054	0.013	0.07	0.5
mmKIJ-2	0.0064	0.0058	0.016	0.04	0.11	0.32	1.13
mmKIJ-3	0.012	0.052	0.35	1.92	10.6	55.45	279
mmKIJ-4	0.004	0.0015	0.0017	0.0028	0.01	0.06	0.49
mmKIJ-5	0.011	0.0198	0.0256	0.072	0.22	0.82	4.17
mmKIJ-6	0.0086	0.00527	0.006	0.018	0.059	0.4	3.08
mmKIJ-7	0.018	0.0053	0.00277	0.0086	0.051	0.38	3.005

1(e) The running time (on KNL node) of each parallel implementation of ITER-MM-IKJ as core number vary from 1 to 272 is listed as below:



The running time (on KNL node) of each parallel implementation of ITER-MM-KIJ as core number vary from 1 to 272 is listed as below:



1(f)

From part 1.d, we can see that,
The mm-IKJ-4 (parallelize the loop I and loop K) and mm-KIJ-4 (parallelize the loop I and loop K) are fastest among all the implementation. The mm-IKJ-3(parallelize the loop J) and The mm-KIJ-3(parallelize the loop J) are slowest one.

From part 1.e, we can see that,
The mm-IKJ-3 and The mm-KIJ-3 becomes slower as the core number increases while others become faster as core number increases. For other implementations, the running time become faster as core number increases.

The reason is that cache miss increases as the loop J is being parallelized. Increasing the core only incurs more cache miss when loop J is being parallelized.

Parallelizing the loop I and loop K can reduce running time since it splits the tasks withouting incurring the cache misses among the computation inside. Increasing the cre number can have more tasks splitted can processed in parallel.

1(g)
Here the PAR-REC-MM is computed (on KNL node) with base function of ITER-MM-KIJ in different base case (2,4,8,..,256,512) to find the best base case when r is 10 (size is 2^{10}).

The value of the base case that gives smallest running time is 32.

The unit is second. For example, when r is 10, and the base is 32 , the running time is 0.847 second.

PAR-REC-MM	2	4	8	16	32	64	128	256	512
r=10	1.316	1.0654	0.97784	0.846	0.845	0.904	1.55189	3.619	12.36
r=11	7.04	4.23	3.37	3.36	3.3	3.3	3.5	9.95	26.5
r=12	52	28	24	24	24	23.13	23.4	24.87	69.7

1(h)

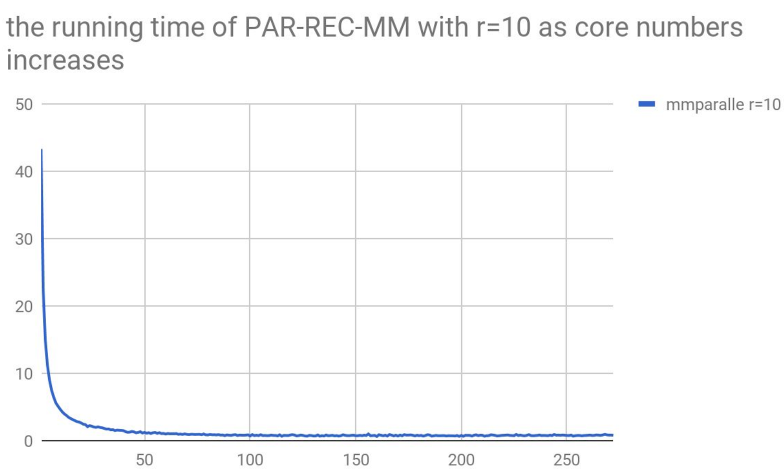
1.The running time (on KNL node) of the PAR-REC-MM with 272 cores, base functions (the serial mm-KIJ and mm-KIJ-4, which parallelize loop K and loop I) is listed as below. The base size is 32.

For example, the running time with size 2^6 and base function of mm-KIJ is 0.02 second

size	Time of mm-KIJ (s)	Time of mm-KIJ-4 (s)
r=6	0.02	0.07
r=7	0.1	0.127744
r=8	0.09	0.301415
r=9	0.16	0.450445
r=10	0.82	0.613645
r=11	2.92	2.81845
r=12	23.2	19.534
r=13	186.6	155.935

The data shows that PAR-REC-MM with base of a parallel mm-KIJ-4 runs faster.

2.The running time of PAR-REC-MM with r=10 and base 32 as core numbers increases is listed as below: the unit is second.



As displayed from data, the running time decreases as core numbers increases.

1(i) Running time and cache miss of ITER-MM-KIJ-4 and Par-Rec-MM on SKX NODE with 192 cores

	ITER-MM-KIJ-4	Par-Rec-MM
r	10	10
base	32	32
L1 cache miss	1285787	853903
L2 cache miss	1319127	122625
L3 cache miss	558640	31535
running time(s)	0.372344	0.202506

The fastest implementation from part 1(d) is ITER-MM-KIJ-4(parallelize loop K and loop I)

We can see that Par-Rec-MM with r=10 and base=32 have less cache miss and faster running time. Therefore the running time becomes faster as cache miss decreases

Task# 2

2(a) Run Par-Rec-MM under DR-Steal, DR-Share and by varying n from 2^{10} to 2^s (consider only powers of 2), where s is the largest integer such that none of the three implementations takes more than 5 minutes to terminate.

Solution: Execution of Par-Rec-MM under each of the three schedulers with given condition resulted in value of s = 14. Although for s = 14, the time is under 2 minutes but for s = 15 the time is well over 12 minutes. Hence, there were 5 different sizes of the matrix for which the algorithm was executed under each scheduler. Below are the findings:

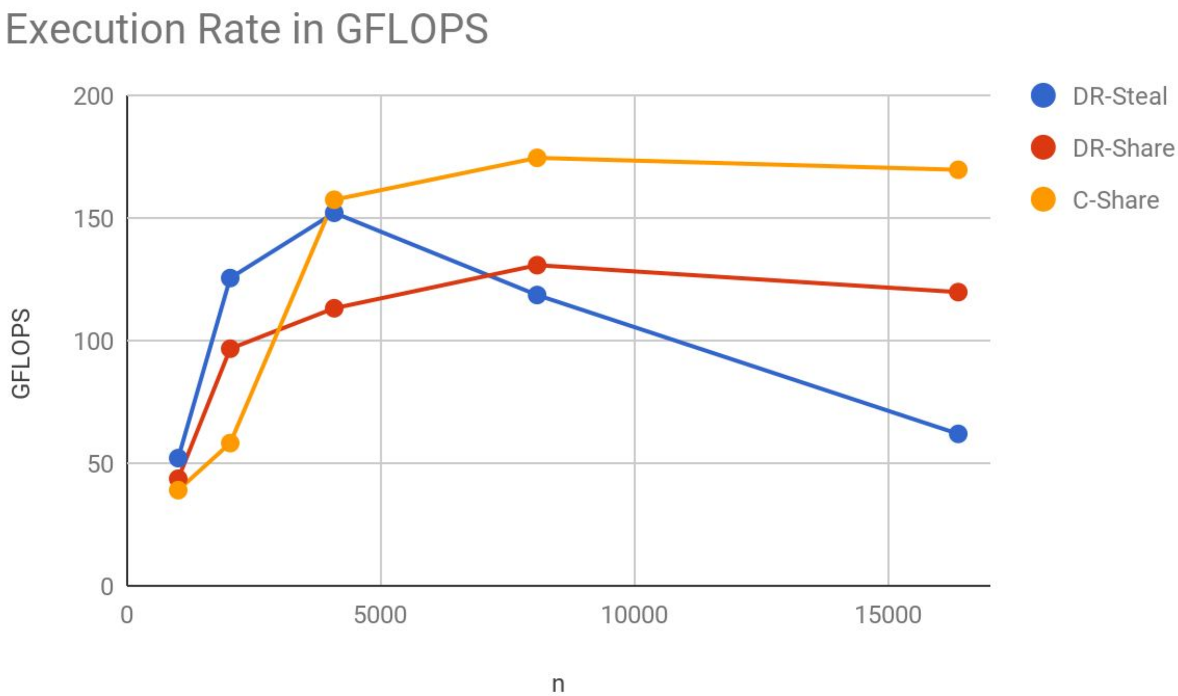
Note: All the runs have been done on SKX nodes and since, for these nodes maximum number of cores is 48, all the executions have been done using 48 threads.
Also, the base size(after which multiplication is done serially) is taken as 256. This is because the size of L2 cache is 1MB for SKX nodes and most optimal value for base size comes out to be 256. L1 cache size was not considered for base size as it would have resulted in base size of 32 which does not give optimal result for larger matrices (details in task 2(b)).

Execution time(in seconds):

n	DR-Steal	DR-Share	C-Share
1024	0.0412512	0.0491964	0.0550689
2048	0.136919	0.177721	0.295238
4096	0.903747	1.21453	0.87317
8092	8.93884	8.10634	6.07356
16384	142.02	73.46	51.858

Rate of Execution in GFLOPS:

n	DR-Steal	DR-Share	C-Share
1024	52.0587	43.65124	38.99631
2048	125.4747	96.66764	58.1899
4096	152.0768	113.1623	157.4023
8092	118.5541	130.7293	174.4835
16384	61.93559	119.7399	169.6188



Explanation of findings:

From the above data, there are 3 parts to be mentioned, one for general trend and two anomalies. THey are as follows:

1. General trend: For any matrix multiplication routine, a typical GFLOPS graph is very similar to the red line graph of DR-Share.
   The reason for this is the size of the matrix. Upto a certain size of a matrix the performance of the routine increases. However, with further increase in the size, execution rate decreases and then, it becomes almost constant. Similar is the case with C-Share and DR-Share. A local maxima can be observed at n = 8192. Further increase in the size decreases the rate of execution. Thus, within the domain of the data set, the highest rates of execution are:
   i) DR-Steal = 152.0768 for n = 4096
   ii) DR-Share = 130.7293 for n = 8192
   iii) C-Share = 174.4835 for n = 8192

2. Anomaly 1: After a steep rise in execution rate, there is a sharp fall in case DR-Steal.
   The reason for such a behavior is non-ideal implementation. An ideal DR-Steal scheduler uses deques to store the tasks. However, we were not able to implement a concurrent deque and so, we had to fall back to concurrent queues. For large matrices, there are more steal operations. Since we are using a queue, all the threads, including the one to which the queue belongs, use the same end of the queue to retrieve a task. Now, for a small matrix size, there are not many steal operations or overheads to affect the performance of the program but as the size increases the cost of overhead operations reduce the performance of the program.

3. Anomaly 2: The peak value of C-Share is curiously high.
   For SKX nodex peak clock rate is 3.7 GHz. FOr all 48 cores, this gives us an rate of 177.4 Ghz. As can be seen from above that the peak value for C-Share is 174.5, which is very close to the maximum possible rate. However, this high rate of execution is possible because of compiler optimizations. We are using O2 optimization in our code and this is the reason for getting such high values.

2(b) Plot cache miss rates for L1, L2 and L3 caches with same condition as in part (a).

Solution: As mentioned in part (a), all the runs have been done on SKX nodes and since, for these nodes the maximum core count is 48, all the executions have been done using 48 threads. Also, there were 5 different sizes of the matrix for which the algorithm was executed under each scheduler.

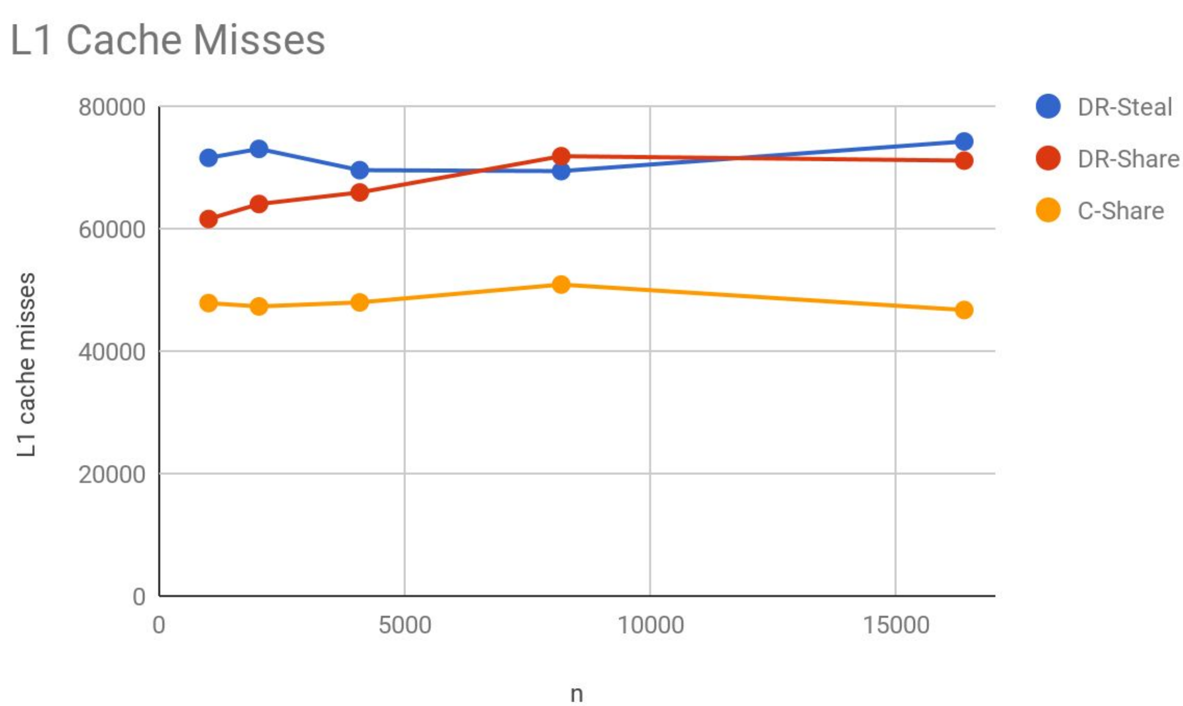
The reason for using SKX nodes is unavailability of L3 cache on KNL nodes.

Temporal Locality - SKX nodes have three types of cache to consider for calculating the block size.
Following are the sizes of cache on SKX nodes:
i) 32KB L1 data cache per core
ii) 1MB L2 per core
iii) 33MB L3 per socket.
Matrices used in the implementation of the PAR-REC-MM algorithm contain “int” values. Since, “int” data type is of 4 bytes, the most optimal base value to fit in a cache can be evaluated using the formula,
b <= $\sqrt{}$(M/3), where M is the effective size of a cache. For different caches we get b as:
i) L1 cache = 32
ii) L2 cache = 256
II) L3 cache = 1024
Since, value of b for L1 cache is too small to produce optimal result and for the given problem b for L3 cache is too large to be considered, value of b for L2 cache was used instead.

On the basis of these interpretations, task 2(a) and 2(b) were executed. Results of cache misses are as follows:

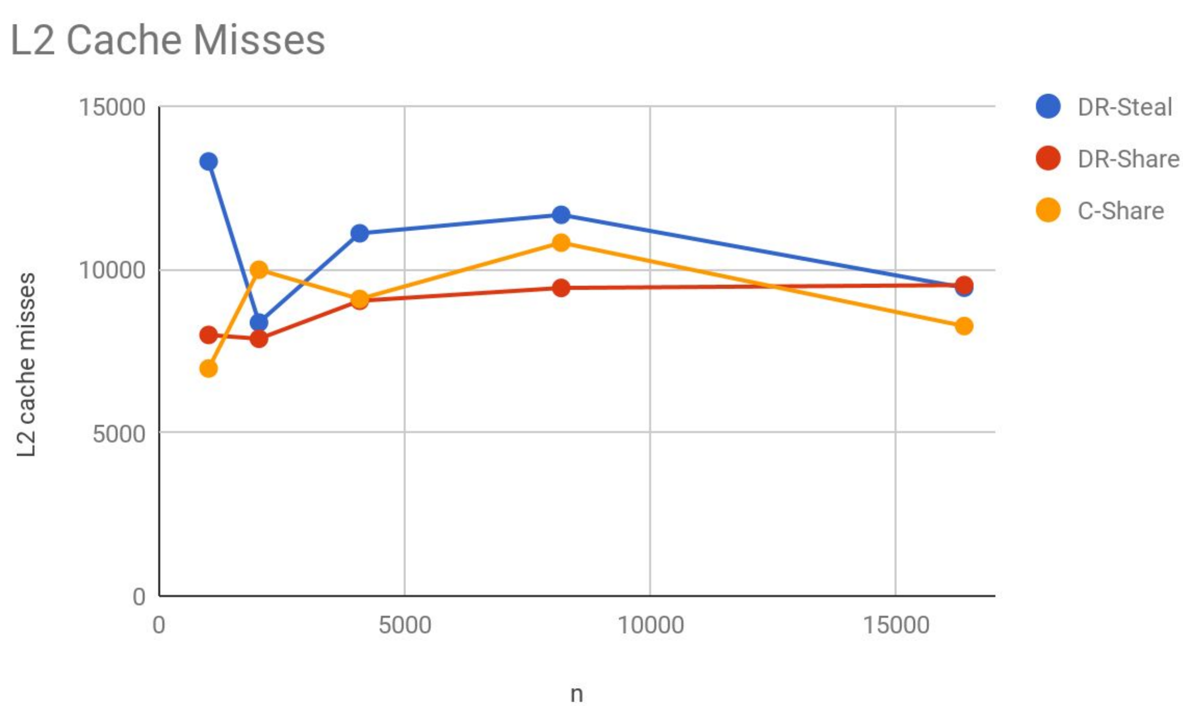
L1 cache misses:

n	DR-Steal	DR-Share	C-Share
1024	71552	61556	47829
2048	73013	64011	47286
4096	69539	65903	47944
8192	69387	71820	50856
16384	74223	71096	46703



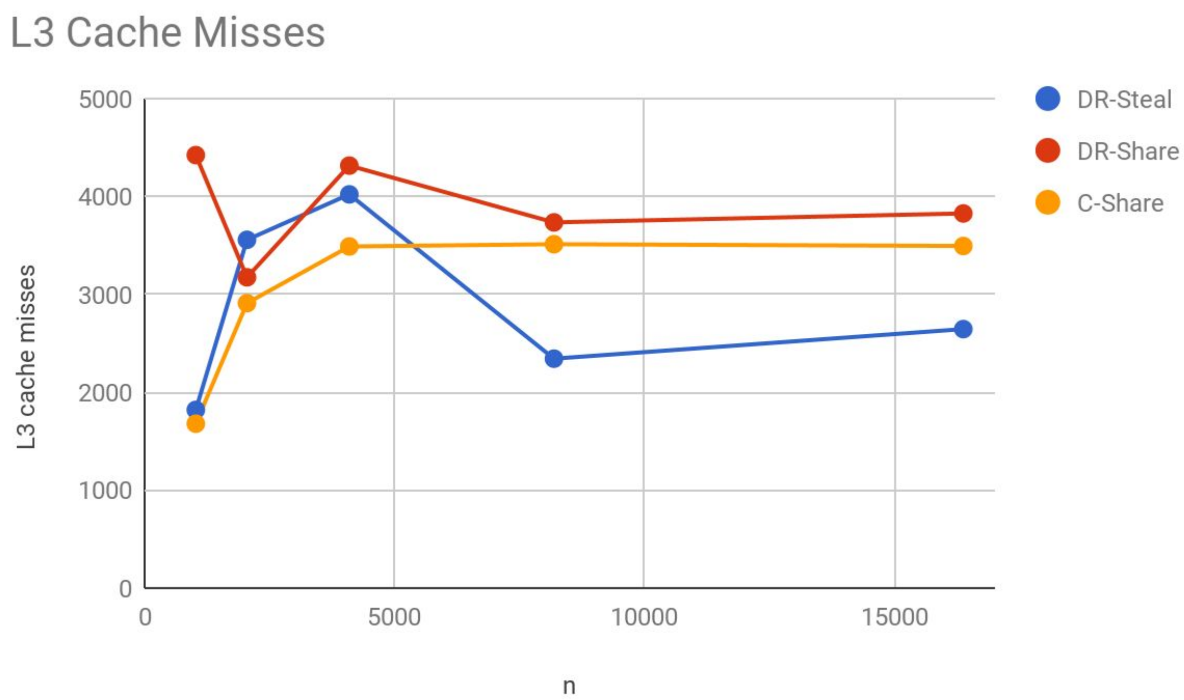
L2 cache misses:

n	DR-Steal	DR-Share	C-Share
1024	13305	7993	6966
2048	8373	7876	9989
4096	11105	9038	9095
8192	11671	9432	10818
16384	9439	9520	8265



L3 cache misses:

n	DR-Steal	DR-Share	C-Share
1024	1820	4422	1679
2048	3557	3174	2910
4096	4021	4315	3489
8192	2343	3735	3512
16384	2645	3826	3494



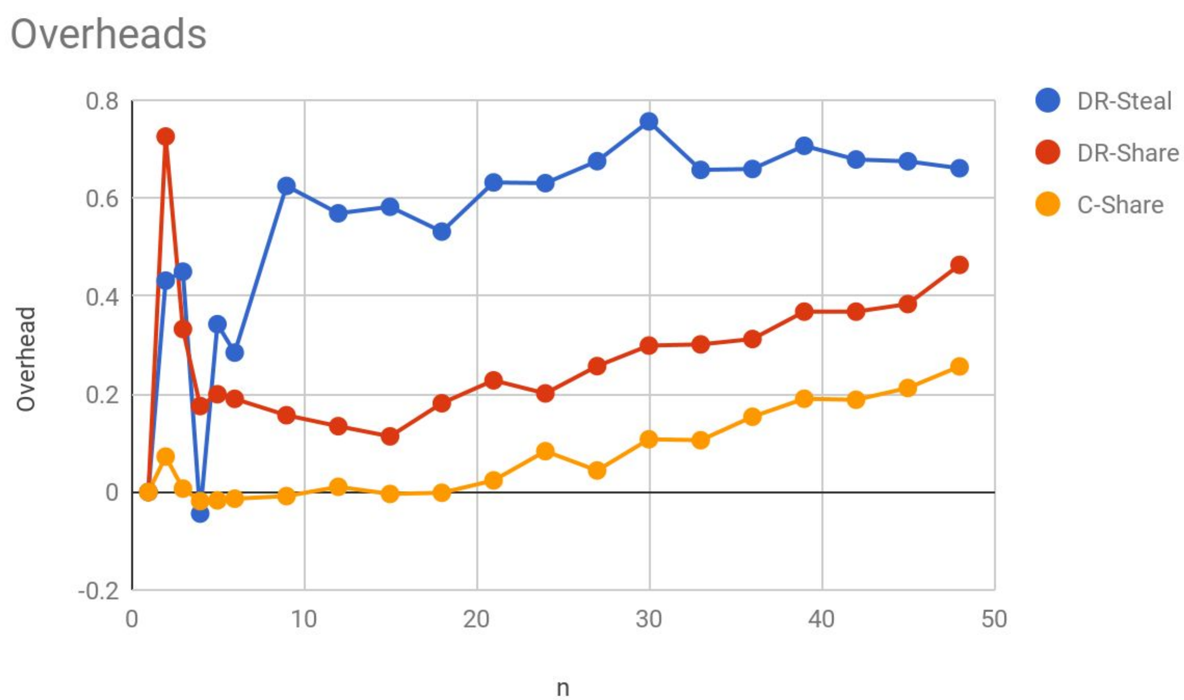
Explanation of findings:
L1 cache incurs the most number of cache misses and L3 the least. The order of number of cache misses is L1 > L2 > L3. This is because the sizes of these caches are in the opposite order, L1 < L2 < L3. Since, L1 cache is the smallest and the fastest we can see that its graph is most closely related to the execution times of all the three programs. The order of L1 cache misses is:
DR-Steal > DR-Share > C-Share.
Whereas, that of the performance of the three programs is:
DR-Steal < DR-Share < C-Share.

2(c) Repeat part 2(a) and vary p from 1 to the maximum number of cores available on the machine. For each run compute its efficiency.

Solution: As mentioned in part 2(a), the maximum value of n is 2^{14}. Since, all the three programs have been executed on SKX nodes, we will show the results from p 1 to 48. Owing to the large number of p, execution time has been calculated at intervals. Below are the results:

Efficiency:

p	DR-Steal	DR-Share	C-Share
1	1	1	1
2	0.568443	0.274319	0.927734
3	0.550167	0.667309	0.993027
4	1.044079	0.824538	1.018608
5	0.65741	0.800471	1.017083
6	0.715421	0.809851	1.013708
9	0.375186	0.84326	1.008349
12	0.431073	0.865656	0.98944
15	0.417746	0.886437	1.003996
18	0.468421	0.818812	1.001537
21	0.368049	0.772105	0.976371
24	0.369537	0.79874	0.916661
27	0.324641	0.743016	0.956108
30	0.243462	0.70108	0.892347
33	0.342439	0.698692	0.894324
36	0.34057	0.687678	0.846363
39	0.293285	0.631778	0.809614
42	0.321216	0.631834	0.811751
45	0.324913	0.616311	0.787671
48	0.339091	0.536393	0.743646



Explanation of findings:
For each of the scheduler, findings are as follows:

1. C-Share - Its program is the most efficient one with the least efficiency being 75%. The efficiency decreases as the number of threads increases. It is because as the number of threads increase, there will be more blocking calls made to the central queue.
   Few of the values are not consistent, i.e., efficiency goes beyond 1. The main reasons for this inconsistency are fluctuation in the clock cycles of the cores and cache optimization performed by the compiler.
2. DR-Steal - The program implementing DR-Steal is the least efficient one with overhead going as high as 75%. Use of queues in instead of deques are the main reason for such high overheads.
3. DR-Share - The program executing DR-Share is moderately efficient with the only anomaly at thread count=2 which is expected as with 2 threads much of the work would go into transferring the extra tasks to the other thread, which then transfers almost the same number of tasks back to the first thread. For higher thread count, the program behaves in an efficient manner and the overhead gradually increases with increase in thread count.

2(d) Repeat parts 2(a) and 2(c) with DR-Steal-Mod and DR-Share-Mod, and compare the results with what you got with DR-Steal and DR-Share, respectively.

Solution: As mentioned in part (a), all the runs have been done on SKX nodes and since, for these nodes maximum thread count is 192, all the executions have been done using 192 cores. Also, there were 5 different sizes of the matrix for which the algorithm was executed under each scheduler.

Below are the tabulated results for DR-Steal-Mod and DR-Share-Mod along compared against DR-Steal and DR-Share.

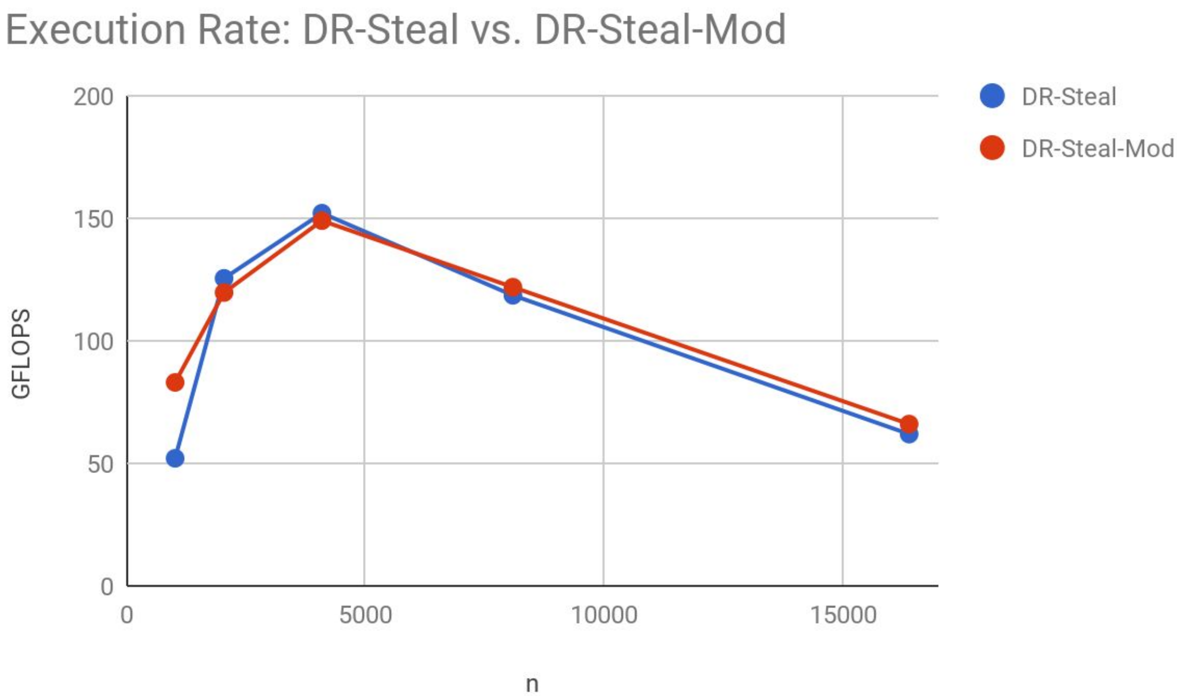
Execution time:

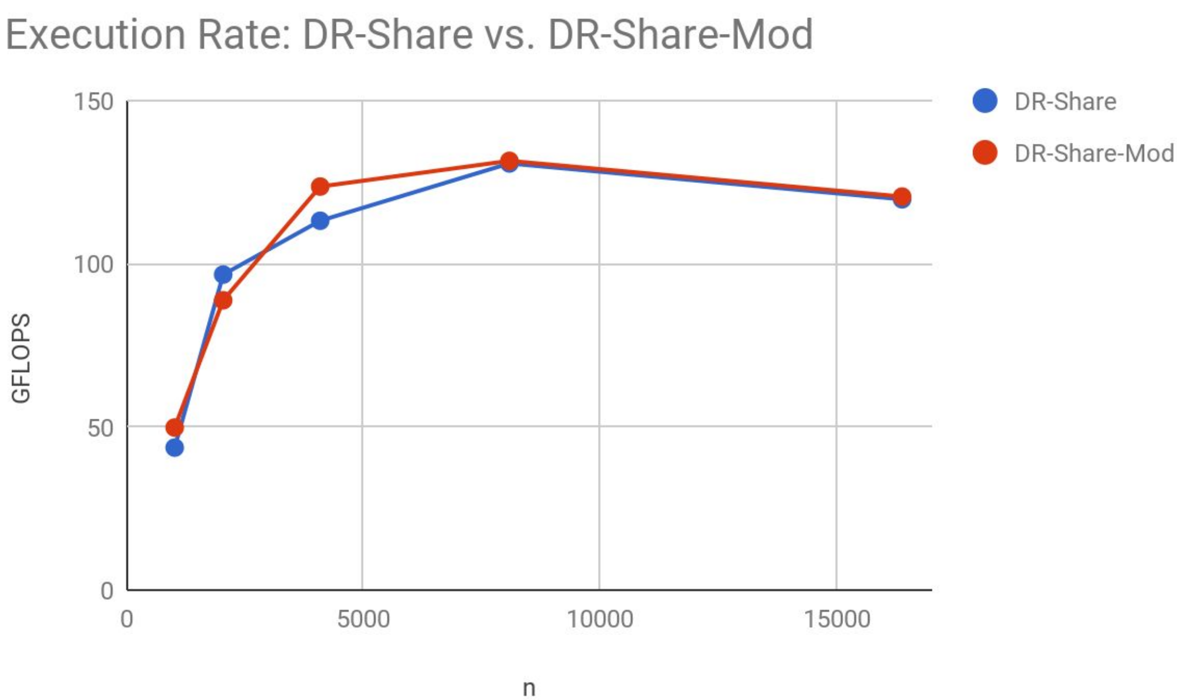
n	DR-Steal	DR-Steal-Mod	DR-Share	DR-Share-Mod
1024	0.0412512	0.0258696	0.0491964	0.0431204
2048	0.136919	0.143542	0.177721	0.193485
4096	0.903747	0.922331	1.21453	1.11157
8092	8.93884	8.69512	8.10634	8.05818
16384	142.02	133.22	73.46	72.9578

GFLOPS:

n	DR-Steal	DR-Steal-Mod	DR-Share	DR-Share-Mod
1024	52.0587	83.01186	43.65124	49.80203
2048	125.4747	119.6853	96.66764	88.79174
4096	152.0768	149.0126	113.1623	123.644
8092	118.5541	121.8771	130.7293	131.5106
16384	61.93559	66.02682	119.7399	120.5641

Below are two graphs, one comparing DR-Steal with DR-Steal-Mod and the other, DR-Share with DR-Share-Mod.


Execution Rate: DR-Steal vs. DR-Steal-Mod
DR-Steal
DR-Steal-Mod
GFLOPS
200
150
100
50
0
0
5000
10000
15000
n


Execution Rate: DR-Share vs. DR-Share-Mod
DR-Share
DR-Share-Mod
GFLOPS
150
100
50
0
0
5000
10000
15000
n

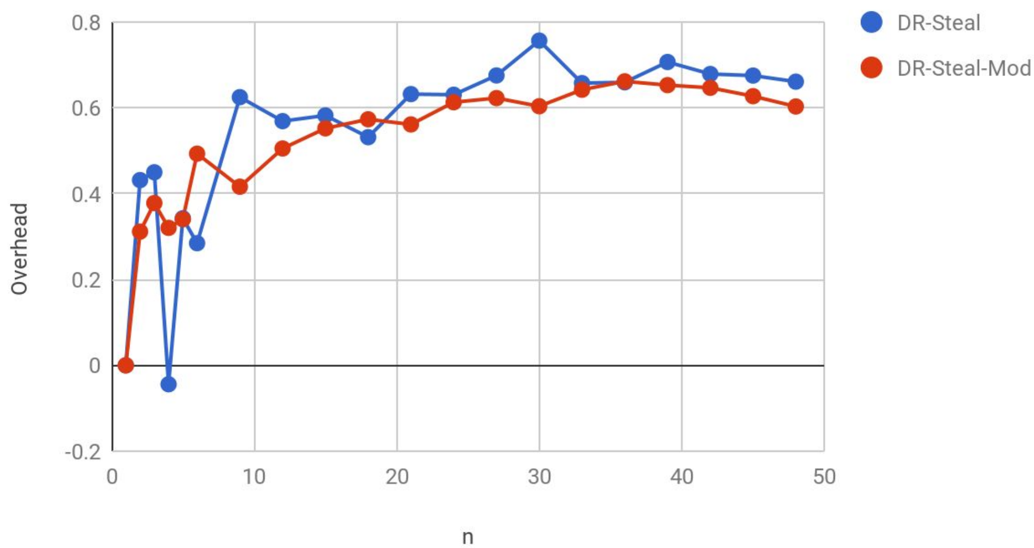
Efficiency:

Below is the table for efficiency of DR-Steal-Mod and DR-Share-Mod, along with that of DR-Steal and DR-Share.

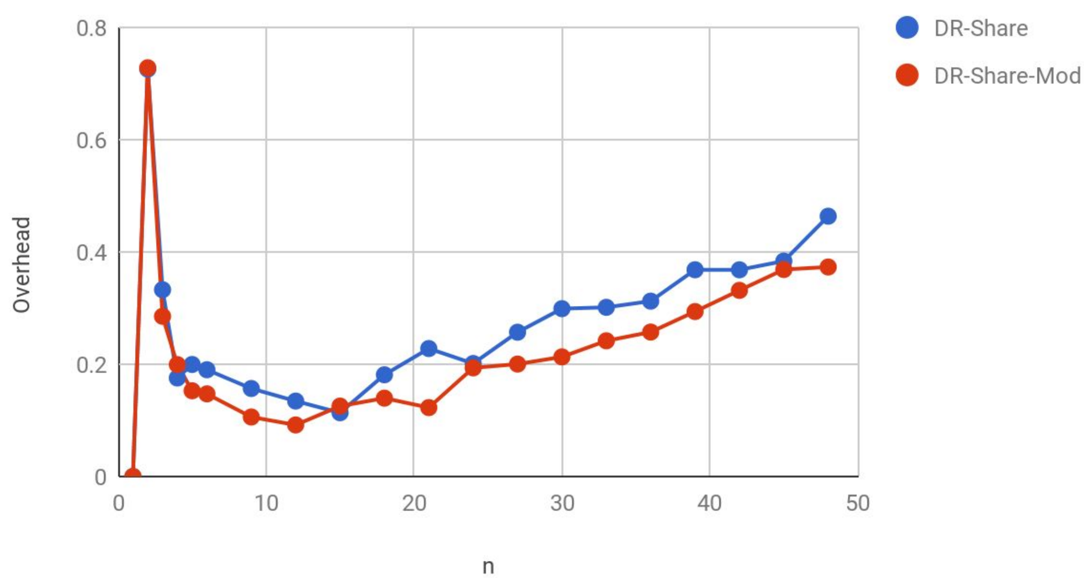
n	DR-Steal	DR-Steal-Mod	DR-Share	DR-Share-Mod
1	1	1	1	1
2	0.568443	0.688432	0.274319	0.272
3	0.550167	0.621951	0.667309	0.714698
4	1.044079	0.679721	0.824538	0.800694
5	0.65741	0.65955	0.800471	0.8473
6	0.715421	0.506899	0.809851	0.852839
9	0.375186	0.583812	0.84326	0.894023
12	0.431073	0.494767	0.865656	0.9082
15	0.417746	0.447896	0.886437	0.87459
18	0.468421	0.426744	0.818812	0.860458
21	0.368049	0.438744	0.772105	0.877267
24	0.369537	0.387009	0.79874	0.80652
27	0.324641	0.377345	0.743016	0.799773
30	0.243462	0.396348	0.70108	0.786984
33	0.342439	0.357889	0.698692	0.758317
36	0.34057	0.338314	0.687678	0.742746
39	0.293285	0.347144	0.631778	0.706219
42	0.321216	0.353356	0.631834	0.668482
45	0.324913	0.372923	0.616311	0.631371
48	0.339091	0.334447	0.536393	0.613343

Below the graphs comparing overheads for DR-Steal vs DR-Steal-Mod and DR-Share vs DR-Share-Mod.

Overheads: DR-Steal vs DR-Steal-Mod



Overheads: DR-Share vs DR-Share-Mod



Explanation of findings:

Both parameters, i.e., rate of execution and efficiency, show that DR-Steal-Mod and DR-Share-Mod have slightly better performance than DR-Steal and DR-Share respectively.

Both the algorithms have a very minor difference from their counterparts which is the reason for the difference in their performance. Selecting a deque(or queue in our case) with more(or, less) work enhances the performance.

Task# 3

3)a)

To prove- During 2p ln p failed steal attempts, the thread has checked all deques in the system w.h.p. in p (and found each of them empty).

We will find out expected number of failed steal attempts such that thread has checked all deques in the system and found each of them empty.

This can be related to Coupon Collector's Problem as given below-

https://en.wikipedia.org/wiki/Coupon_collector%27s_problem

Let *T* be the time to collect all *n* coupons, and let t_i be the time to collect the *i*-th coupon after $i - 1$ coupons have been collected. Think of *T* and t_i as random variables. Observe that the probability of collecting a **new** coupon is $p_i = (n - (i - 1))/n$. Therefore, t_i has geometric distribution with expectation $1/p_i$. By the linearity of expectations we have:

$$\begin{aligned} \mathrm{E}(T) &= \mathrm{E}(t_1) + \mathrm{E}(t_2) + \cdots + \mathrm{E}(t_n) = \frac{1}{p_1} + \frac{1}{p_2} + \cdots + \frac{1}{p_n} \\ &= \frac{n}{n} + \frac{n}{n-1} + \cdots + \frac{n}{1} \\ &= n \cdot \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}\right) \\ &= n \cdot H_n. \end{aligned}$$

Let T be the number of steal attempts such that all deques has been checked and found to be empty .

Let t_i be the ith attempt such that i-1 deques has been checked.

Probability of checking a new deque $pr_i = \left(\frac{p-(i-1)}{p}\right)$.

By linearity of expectations we have :

$$\begin{aligned} \mathrm{E}(T) = \mathrm{E}(t_1) + \mathrm{E}(t_2) + \mathrm{E}(t_3) + \ldots\ldots\ldots\ldots + \mathrm{E}(t_n) &= \frac{1}{pr_1} + \frac{1}{pr_2} + \ldots\ldots\ldots + \frac{1}{pr_n} \\ &= \frac{p}{p} + \frac{p}{p-1} + \ldots\ldots\ldots + \frac{p}{1} \\ &= p.\left(\frac{1}{1} + \frac{1}{2} \ldots\ldots\ldots + \frac{1}{n}\right) \\ &= p.H_p \end{aligned}$$

H_p is pth harmonic number hence asymptotic bound on H_p is

$H_p = \Theta(logp)$

Which implies that $p.H_p = \Theta(plogp)$.

Since $lnp = 2.303logp$

$p.H_p \le plnp$

Hence expected number of steal attempts such that thread has checked all deques and found them empty are $plnp$.

Thus,

After $2plnp$ failed steal attempts thread has checked all the deques in the system and found each of them empty w.h.p. In p.

3)b)
If a thread finds every deque in the system empty in consecutive failed steal attempts that does not guarantee that the entire system has run out of work.
We will assume Thread t1 is trying to steal from deques of other threads and the steal attempts are failing.
1. At the same time there can be a thread t2 which is working on a task and which will spawn new tasks. The deque of Thread t2 is empty hence t1's steal attempt is failed. Still t2 is working and it might spawn new tasks and hence entire system hasn't run out of work.
2. It might happen that while t1 is trying to steal from deque of t2, there was another thread t3 which was successful in stealing from t2's deque and started working on it and hence now t2's deque is empty. T1 steal attempt would fail. But system has not run out of work since t3 is working on a task.

3)c) Even if threads follow the strategy in part 3(a) to terminate prematurely (as suggested by part 3(b)), all work in the system will still be completed.
1. Though threads terminate after 2plnp failed steal attempts and system has not run out of work then there are still threads that are working on tasks which will complete their work and then only terminate. All of the threads have not terminated.
Since few threads terminated, so it will take longer time to complete the work , but threads that are currently working on tasks will make sure the work is completed. As soon as they finish their tasks they will steal work from other deques and will terminate after 2plnp failed steal attempts.

3)d) (Collaborated with Nikitha Kandru)
To Prove - During any sequence of p consecutive enqueues each deque undergoes O (ln p/ ln ln p) enqueue operations w.h.p. in p.
Each deque should undergo atmost ln p/ln ln p enqueue operations
Relating it to balls and bins problem where balls are enqueue operations and bins are deques.

We need to prove – When p balls are thrown then all bins will have at most ln p/ln ln p balls with high probability in p.

Suppose we take a subset of balls of size m from n -
Then
Pr[m balls fall in bin i] = $\frac{1}{p^m}$

Total subsets of size m chosen from p - $\binom{p}{m}$

Using Boole's inequality (https://en.wikipedia.org/wiki/Boole%27s_inequality), we take union bound of the probability for all the subsets of size m -

Pr[bin i has at least m balls] $\le \binom{p}{m} * \frac{1}{p^m}$
According to standard inequality on Binomial coefficients http://page.mi.fu-berlin.de/shagnik/notes/binomials.pdf) -

$\forall$ 1 ≤ k ≤ n : $\binom{n}{k} \le \left(\frac{ne}{k}\right)^k$
Hence , $Pr[bin\ i\ has\ at\ aleast\ m\ balls] \le \binom{p}{m} * \frac{1}{p^m} \le \left(\frac{e}{m}\right)^m$

We need to prove that w.h.p each bins will have at most m balls
Let's assume m = $\frac{3ln\,p}{lnlnp}$

⇒ Pr[bin i has at least m balls] $\le (\frac{e}{m})^m = \left(\frac{e\,lnlnp}{3lnp}\right)^{\frac{3lnp}{lnlnp}}$
Simplifying the equation further by taking log and converting it to exponential form we get
Pr[bin i has at least m balls] $\le\ exp\left(\frac{3lnp}{lnlnp}(lnlnlnp\ - lnlnp)\right)$
$= \ exp\left(-3lnp\ +\ \frac{lnp.lnlnlnp}{lnlnp})\right)$

When p is very large-

Pr[bin i has at least m balls] $\le\ -2lnp\ =\ \frac{1}{p^2}$
Using union bound -
Pr[there exists a bin with at least m balls] $\le p.\frac{1}{p^2} = \frac{1}{p}$

Thus, Pr[all bins have at most m balls]$\ge 1\ -\frac{1}{p}$ which is high probability in p.

Hence proved that all bins will have O($\frac{lnp}{lnlnp}$) balls w.h.p in p.
Relating to the dequeues and enqueue operations - all dequeues will undergo O($\frac{lnp}{lnlnp}$) enqueue operations w.h.p in p.

3)e)
i)
f_i = fraction of p deques which have i tasks.
f_{i+1} = fraction of p deques which have i+1 tasks.

To enqueue (i+1)th task ,algorithm will select two deques which have at least i tasks.

Probability to choose deques which contains i tasks = f_i (since $p * f_i$ deques can be selected out of p deques)
Probability to choose deques which contains i+1 tasks $\le\ f_i * f_i$ ----------(1)
Since at least two deques need to have i tasks so that those two can be chosen by DR-SHARE-MOD.
Also we know that
Probability to choose deques which contains i+1 tasks = f_{i+1} -----------(2)

From (1) and (2) we get $f_{i+1} \le f_i * f_i\ =\ f_i^2$, Hence Proved.

ii)
Since f_2 = fraction of deques which have at least 2 tasks. If we multiply it by 2 tasks it should be less than equal to 1.
We know that f_2 *2 $\le\ 1$
Which implies that $f_2 \le\ \frac{1}{2}$
From (i) we proved that $f_{i+1} \le\ f_i^2$
Hence $f_3 \le\ (f_2)^2\ =\ \left(\frac{1}{2}\right)^2 = \frac{1}{2^2}$
Similarly, $f_4 \le\ \left(\frac{1}{2^2}\right)^2 = \left(\frac{1}{2^{2^2}}\right)$

$$f_5 \le \left(\frac{1}{2^{2^2}}\right)^2 = \frac{1}{2^{2^3}}$$

$$f_6 \le \left(\frac{1}{2^{2^3}}\right)^2 = \frac{1}{2^{2^4}}$$

Hence by mathematical induction we prove that $f_i \le \left(\frac{1}{2^{2^{i-3}}}\right)^2 = \frac{1}{2^{2^{i-2}}}$

Thus, $f_i \le \frac{1}{2^{2^{i-2}}}$

iii)

To prove - No task is likely to have a rank larger than log log p during those p consecutive enqueue attempts.

We will take the proof from (ii)

$f_i \le \frac{1}{2^{2^{i-2}}}$

Taking ln on both the sides -

$ln\,(2^{2^{i-2}}) \le ln\left(\frac{1}{f_i}\right)$

Taking ln again on both the sides

$lnln\,(2^{2^{i-2}}) \le lnln\left(\frac{1}{f_i}\right)$

$\Rightarrow ln\ 2^{i-2} \le lnln\left(\frac{1}{f_i}\right)$

$\Rightarrow i-2 \le lnln\left(\frac{1}{f_i}\right)$

Also there can be at most 1 deque which will have p tasks(highest rank of tasks, so fraction of deques for the highest i is 1/p.

Hence $f_i = \frac{1}{p}$

Thus $i-2 \le lnln\,(p)$

$\Rightarrow i \le lnln\,(p)+2$

Also ln(p) = 2.303(logp)

Hence upper bound on i (i.e. rank of any task) is O(loglogp).