CSE 613: Parallel Programming

Homework-1, 28 March

Spring 2019

Sagar Jeevan (111861945)
Sirisha Mandali (111862254)

---

1. (A)

R	IJK	IKJ	JIK	JKI	KIJ	KJI
3	0.00222	8.3152e-05	8.1238e-05	8.0456e-05	7.7619e-05	8.0124e-05
4	0.00054	0.00049	0.00054	0.00051	0.00051	0.00052
5	0.00393	0.00373	0.00390	0.00381	0.00370	0.00380
6	0.02984	0.02806	0.02967	0.02949	0.02928	0.03144
7	0.24851	0.22544	0.23701	0.23778	0.22521	0.23669
8	1.93862	1.81009	1.88143	1.87327	1.79166	1.87071
9	15.1385	14.5477	14.8091	15.8504	14.4198	15.0042
10	19.1556	15.2591	17.7869	20.1427	15.2629	18.0387
11	170.606	127.477	162.406	162.958	134.993	154.187

1.(B)

A. IJK

R	L1 TCM	L2 TCM	L3 TCM
3	346	179	56
4	497	324	151

5	728	473	254
6	370	291	153
7	315	180	68
8	343	188	94
9	223429305	3993892	41229
10	1831314445	67554737	368829
11	13749300310	539044324	3406600

B. IKJ

R	L1 TCM	L2 TCM	L3 TCM
3	160	155	43
4	257	182	46
5	531	390	187
6	129	112	48
7	128	110	39
8	121	75	22
9	8606868	4246579	36774
10	68249252	68224156	236423
11	549970383	544524633	2530047

C. JIK

R	L1 TCM	L2 TCM	L3 TCM
3	116	103	23
4	257	182	46
5	452	350	173

6	103	92	34
7	99	96	25
8	97	64	19
9	220817792	8158403	135869
10	1538583900	71632688	425497
11	13732430614	555448096	4069095

D. JKI

R	L1 TCM	L2 TCM	L3 TCM
3	115	91	20
4	232	173	48
5	525	356	186
6	92	110	45
7	92	70	17
8	122	99	41
9	429393085	5315472	32359
10	3205280214	69508104	229883
11	25044655215	546556039	3507910

E. KIJ

R	L1 TCM	L2 TCM	L3 TCM
3	154	102	28
4	232	176	57
5	520	379	185
6	106	93	41

7	107	76	22
8	110	64	27
9	8952984	4799136	99144
10	69697507	71016194	553563
11	556503780	555265629	7228325

F. KJI

R	**L1 TCM**	**L2 TCM**	**L3 TCM**
3	139	125	24
4	229	193	63
5	489	370	177
6	102	99	39
7	113	66	20
8	97	78	30
9	429666490	3907505	27109
10	3304456337	67703840	230886
11	24958637662	539045268	3808320

1.(C)

From 1 (A) & 1 (B), we can see that IKJ take less runtime as they have less number of total cache misses when compared with the other loops.

```
void iter_mm_jik(int n)
     for(int j=0;j<n;j++)
          for(int i=0;i<n;i++)
               for(int k=0;k<n;k++)
                    Z[i][j] = Z[i][j] + X[i][k] * Y[k][j]
```

Consider the IKJ code above. The IKJ is faster because of two main reasons.

1. ***In line 5 of above code***, X[i][j] remains constant until j has finished all of its iterations. So, it's more likely that X[i][k] will always be in cache.
2. Also, as j iterates, Y[K][J] is more likely to be found in same block because they are contiguous. These two cases is however not true for any other ordering of for loops.

---

1. (D)

IKJ, and KJI are the two fastest implementation. We have replaced one or more for loops with parallel for loops. Out of those many ways, the below ones **(7)** are correct. Table below shows the run times of the correctly parallelized ones.

R	**IKJ1 (PFF)**	**IKJ3 (FFP)**	**IKJ4 (PPF)**	**IKJ5 (PFP)**	**KJI2 (FPF)**	**KJI3 (FFP)**	**KJI6 (FPP)**
4	0.06736	0.00094	0.00074	0.00091	0.00075	0.000957	0.00094
5	0.07039	0.00577	0.00503	0.00561	0.00504	0.00602	0.00578
6	0.09529	0.04220	0.03795	0.04088	0.03911	0.042707	0.04236
7	0.35605	0.31779	0.30217	0.31500	0.31178	0.329035	0.32497
8	0.16112	0.52953	0.06259	0.06428	0.12606	0.452049	0.13027
9	0.52536	2.47765	0.45024	0.45758	0.91980	2.05731	0.93512
10	3.64184	10.5894	3.53923	3.59351	5.05736	9.79535	5.10994
11	27.7839	57.5502	27.6204	27.8473	29.8441	53.7329	30.1798

---

1. (E)
IKJ4, for r = 10

P	Time
1	124.542
5	28.3768
10	14.5571
15	9.44513
20	7.17134
25	5.65187
30	4.77488
35	4.30542
40	3.80605
45	3.34799
48	3.19777

KJI2, for r = 10

P	Time
1	134.684
5	31.5285
10	16.6637
15	11.8357
20	9.39525
25	8.22015
30	7.40278
35	6.91511
40	6.74256
45	6.54085
48	6.27217

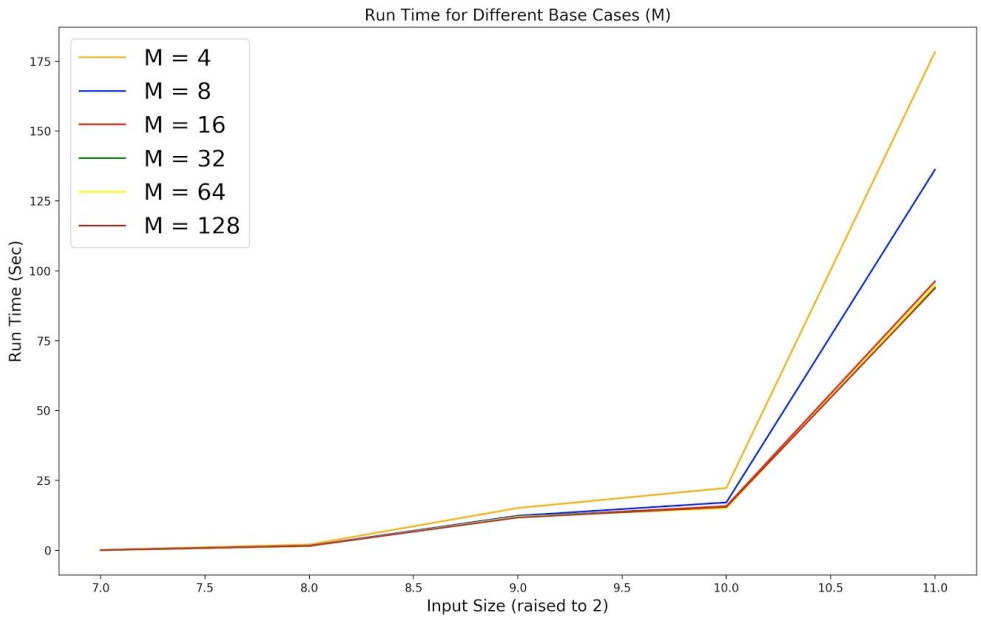
1.(F)

As we can observe that the runtime for r = 10 in IKJ4 is 3.53923 and for KJI2 is 5.0536 and as we increase the number of cores, the run time reduces as it is an iterative parallel for loop.

1. (G)

N	M = 4	M = 8	M = 16	M = 32	M = 64	M = 128
5	0.00385	0.00304	0.00297	0.00305	-	-
6	0.02934	0.02610	0.02367	0.02325	0.02427	-
7	0.14193	0.04617	0.035617	0.03205	0.03173	0.02886
8	2.03238	1.58794	1.50127	1.46738	1.48553	1.56052
9	15.1056	12.3481	11.9192	12.0379	11.9391	11.7056
10	22.2891	17.0967	15.7817	15.3304	15.0136	15.4494
11	178.2173	136.1678	96.1831	94.03223	94.8186	93.6775

Plot below shows the variation of runtime for different base cases.
M = 32, 64, and 128 works equally good.
We chose m value as 64.



1. (H)

IKJ, for r = 10

P	Time
1	96.1276
5	119.103
10	120.843
15	123.975
20	125.757
25	128.565
30	130.925
35	133.992
40	136.587
45	139.138
48	140.398

KJI, for r = 10

P	Time
1	96.74823
5	232.881
10	264.83
15	279.711
20	288.523
25	296.305
30	298.991

35	308.008
40	311.703
45	315.574
48	313.555

---

1 (i)

PAR-REC-MM Implementation (from 1 (G)) BASE CASE M = 64

Size (powers of 2)	**L1 TCM**	**L2 TCM**	**L3 TCM**
7	13969	1870	346
8	168495	3044	793
9	1172422	4679	1554
10	21822605	5269	3032
11	135275618	18709	11088

Fastest Implementation of 1(D)

Size (powers of 2)	**L1 TCM**	**L2 TCM**	**L3 TCM**
7	147870	7299	1382
8	1112900	30824	5678
9	8658731	4007373	35536
10	68496603	68462021	244589
11	550331433	545452083	4909497

As per the observations from 1 (D), and 1 (G), the run time for PAR-REC-MM is more than fastest implementation of 1 (D). But, PAR-REC-MM has lower cache misses.

This is because we are using IKJ for base which has good cache efficiency. Whereas, in fastest implementation of 1 (D) we are parallelizing two for loops (IKJ PPF). So, the access to X[I][K], and Y[K][J] will differ. Run time of PAR-REC-MM is more because of the overhead of the recursive calls.

---

2. a)

As expected, DR-STEAL performs much better than the other two implementations because it steals from other deques when its own deque is empty. Next, DR-SHARED performs better than C-SHARE.

C-SHARE Scheduler

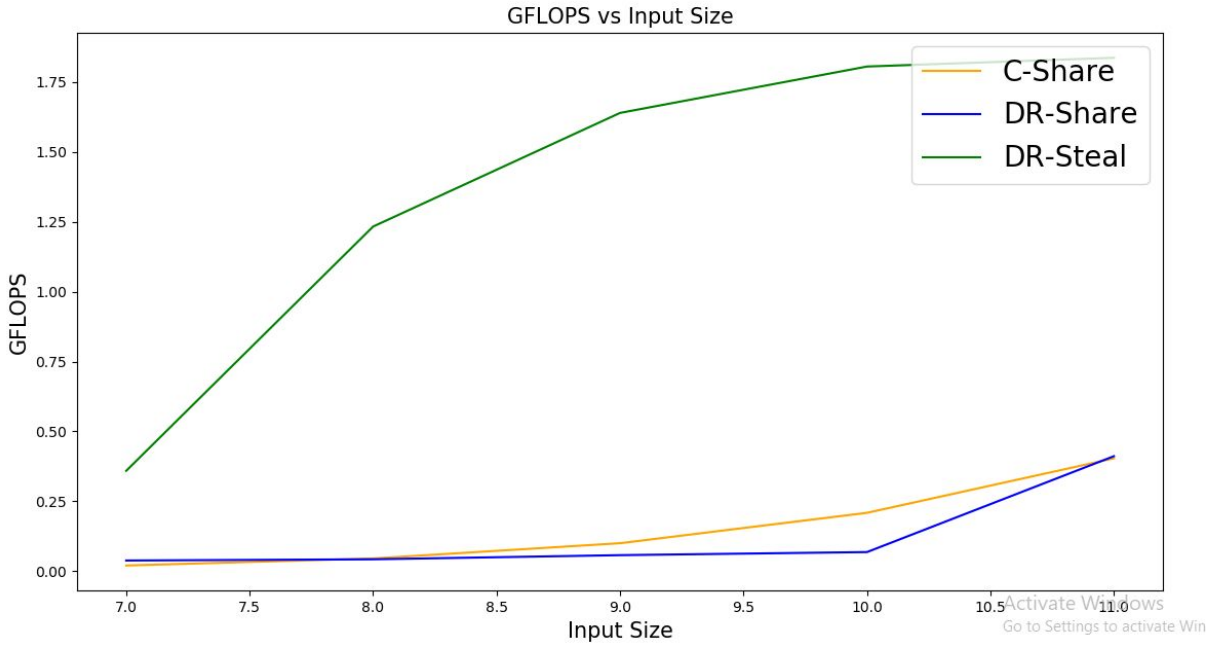
Size (raised to 2)	**Run Time (Sec)**	**Rate of Execution (GFLOPS)**
7	0.22794	0.02011
8	0.75265	0.04525
9	2.69788	0.09989
10	10.2847	0.20858
11	42.5759	0.40351

DR-SHARE Scheduler

Size (raised to 2)	**Run Time (Sec)**	**Rate of Execution (GFLOPS)**
7	0.02198	0.19082
8	0.09123	0.36777
9	0.60034	0.44713
10	4.66673	0.46019
11	35.5697	0.48299

DR-STEAL Scheduler

Size (raised to 2)	Run Time (Sec)	Rate of Execution (GFLOPS)
7	0.01168	0.358924
8	0.02720	1.23323
9	0.16369	1.63989
10	1.18916	1.80588
11	9.35123	1.83718
12	80.3127	1.7113



2 (b)

In all of the cases, L1 cache miss rate is above 100% indicating that only a few data are stored and thus it's more likely to be not found. In central, L1 cache miss rate is around 100% and is not too bad. DR-SHARE scheduler has the worst of all because the jobs are put into random deques. Thus, it is more likely that different random blocks are being used during execution. The L2 cache miss rate is not too bad as Share more likely because the tasks are put into the thread's own deque.

C-SHARE Scheduler

Size (raised to 2)	L1 TCM	L1 TCA	L1 Cache Miss Rate (%)
7	6512032	6511464	100.009
8	12848544	12847976	100.004
9	13888928	13888360	100.004
10	28671328	28670760	100.002
11	30420320	30419752	100.002

Size (raised to 2)	L2 TCM	L2 TCA	L2 Cache Miss Rate (%)
7	6511520	4223790	1.54163
8	12848032	4223790	3.04183
9	13888416	4223790	3.28814
10	28670816	4223790	6.78794
11	30419808	4223790	7.20202

Size (raised to 2)	L3 TCM	L3 TCA	L3 Cache Miss Rate (%)
7	6336840	47510159429065	1.33379e-05
8	6336840	47922281139657	1.32232e-05
9	6336840	47596412382665	1.33137e-05
10	6336840	47240298614217	1.34141e-05
11	6336840	47418645118409	1.33636e-05

DR-SHARE Scheduler

Size (raised to 2)	L1 TCM	L1 TCA	L1 Cache Miss Rate (%)

7	30655184	4222984	725.913
8	37573328	4222984	889.734
9	34460368	4222984	816.019
10	11141840	4222984	263.838
11	13783760	4222984	326.399

Size (raised to 2)	L2 TCM	L2 TCA	L2 Cache Miss Rate (%)
7	6336832	47535211421129	1.33308e-07
8	6336832	47401602407881	1.33684e-07
9	6336832	47890241879497	1.3232e-07
10	6336832	46947067215305	1.34978e-07
11	6336832	47867433536969	1.32383e-07

Size (raised to 2)	L3 TCM	L3 TCA	L3 Cache Miss Rate (%)
7	0	140730934146352	0
8	0	140728538769792	0
9	0	140734310149968	0
10	0	140720709897616	0
11	0	140728540712464	0

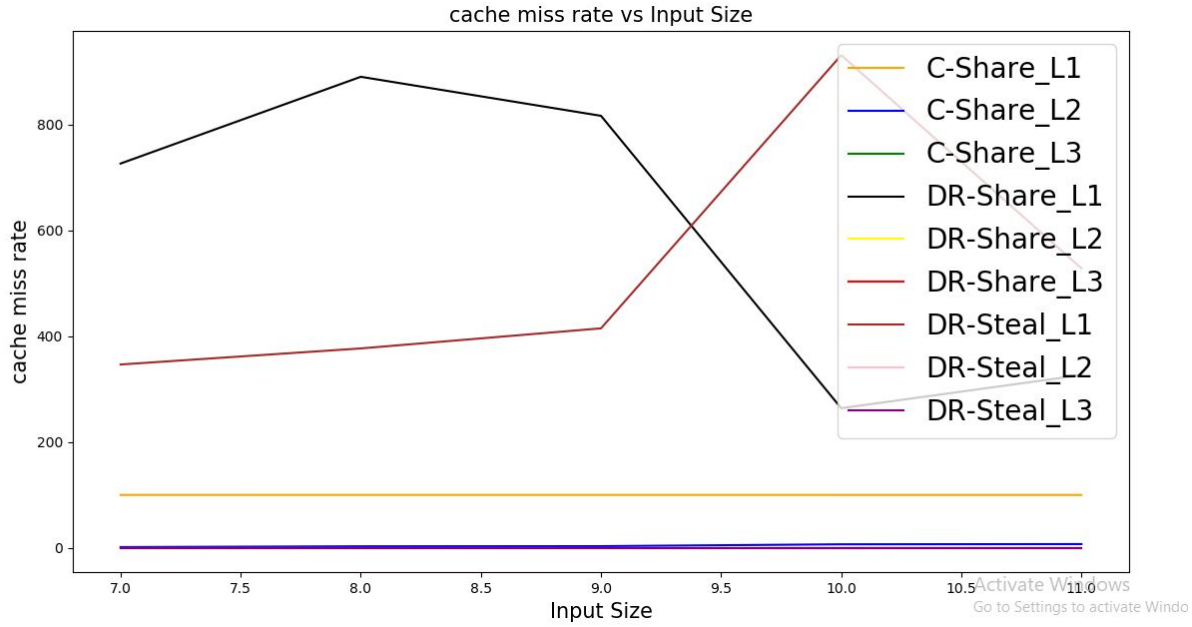
DR-STEAL Scheduler

Size (raised to 2)	L1 TCM	L1 TCA	L1 Cache Miss Rate (%)
7	14625616	4222794	346.349
8	15907664	4222794	376.709

9	17509200	4222794	414.635
10	39295824	4222794	930.565
11	22289232	4222794	527.831

Size (raised to 2)	L2 TCM	L2 TCA	L2 Cache Miss Rate (%)
7	6336824	47665366363593	1.32944e-07
8	6336824	47973266775497	1.32091e-07
9	6336824	47840913751497	1.32456e-07
10	6336824	47634264669641	1.33031e-07
11	6336824	47929067589065	1.32213e-07

Size (raised to 2)	L3 TCM	L3 TCA	L3 Cache Miss Rate (%)
7	0	140724912976496	0
8	0	140736400999056	0
9	0	140734647498752	0
10	0	140724500037040	0
11	0	140720491528272	0



2. (C)

The efficiency of all the schedulers follow the same pattern. It starts of with 1 and drops gradually indicating the increase in the overhead. The tasks may not need the maximum available cores for it to process. All of the experiments have been performed with size n = 10, power of 2.

C-SHARE Scheduler

Number of Cores	**Run Time (sec)**	**Efficiency**
1	43.3853	1
5	5.63079	0.0802606
10	6.58836	0.0685952
15	5.96365	0.0757808
20	6.5173	0.0693431
25	6.78728	0.0665849
30	6.71765	0.0672751

35	6.89815	0.0655147
40	6.30904	0.0716322
48	6.1728	0.0732132

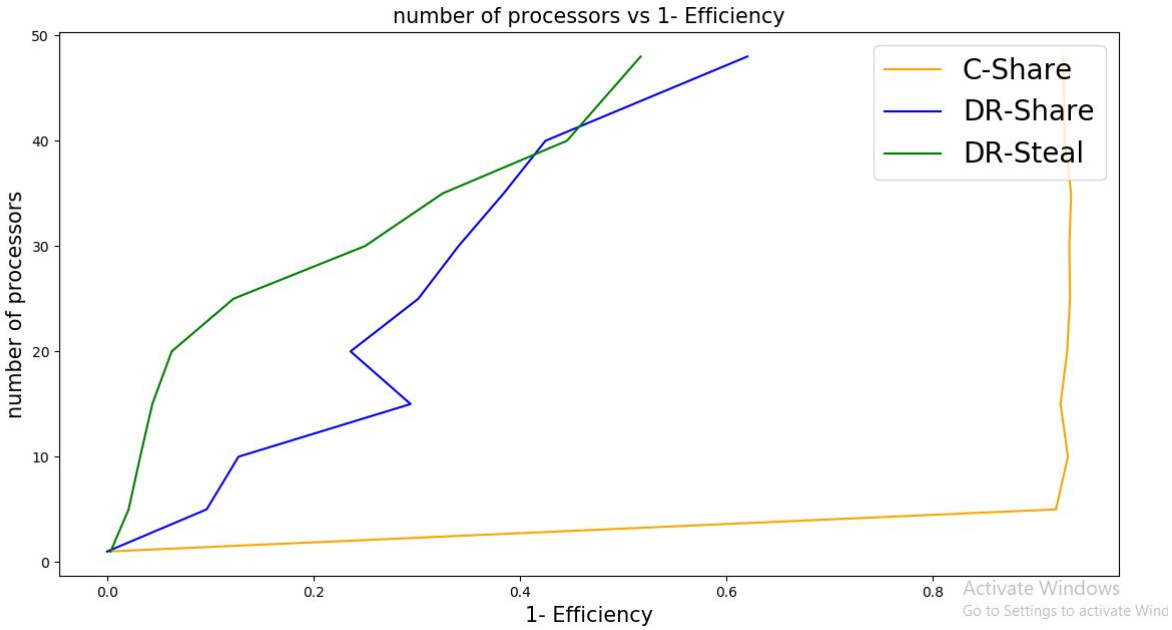
DR-SHARE

Number of Cores	**Run Time (sec)**	**Efficiency**
1	17.052	1
5	3.77447	0.903544
10	1.95392	0.872706
15	1.61052	0.705859
20	1.92449	0.764066
25	0.976636	0.698397
30	0.861987	0.659407
35	0.791394	0.615622
40	0.741303	0.575068
45	0.662154	0.572274
48	0.936516	0.379331

STEAL

Number of Cores	**Run Time (sec)**	**Efficiency**
1	17.3543	1
5	3.54471	0.979167
10	1.79284	0.96798
15	1.20993	0.956216

20	0.925649	0.937412
25	0.791059	0.877522
30	0.771224	0.750076
35	0.734803	0.674789
40	0.782792	0.554244
45	0.764732	0.504296
48	0.748874	0.482789



2 (d)

DR-STEAL MOD performs better compared to DR-SHARE MOD. The GFLOPS are higher in DR-STEAL MOD because because threads steal from the deque that has more load thus balancing the load. Where as share just executes tasks from its own deque thus there is longer wait times for tasks to be executed.

DR-SHARE MOD Scheduler

Size (raised to 2)	**Run Time (Sec)**	**Rate of Execution (GFLOPS)**
7	0.25532	0.02133

8	0.46564	0.87654
9	1.08213	0.39407
10	3.68742	0.58588
11	22.1813	0.77451

DR-STEAL MOD Scheduler

Size (raised to 2)	**Run Time (Sec)**	**Rate of Execution (GFLOPS)**
7	0.11528	0.03873
8	0.22144	0.13207
9	0.64824	0.41967
10	2.95522	0.72754
11	21.4604	0.80238

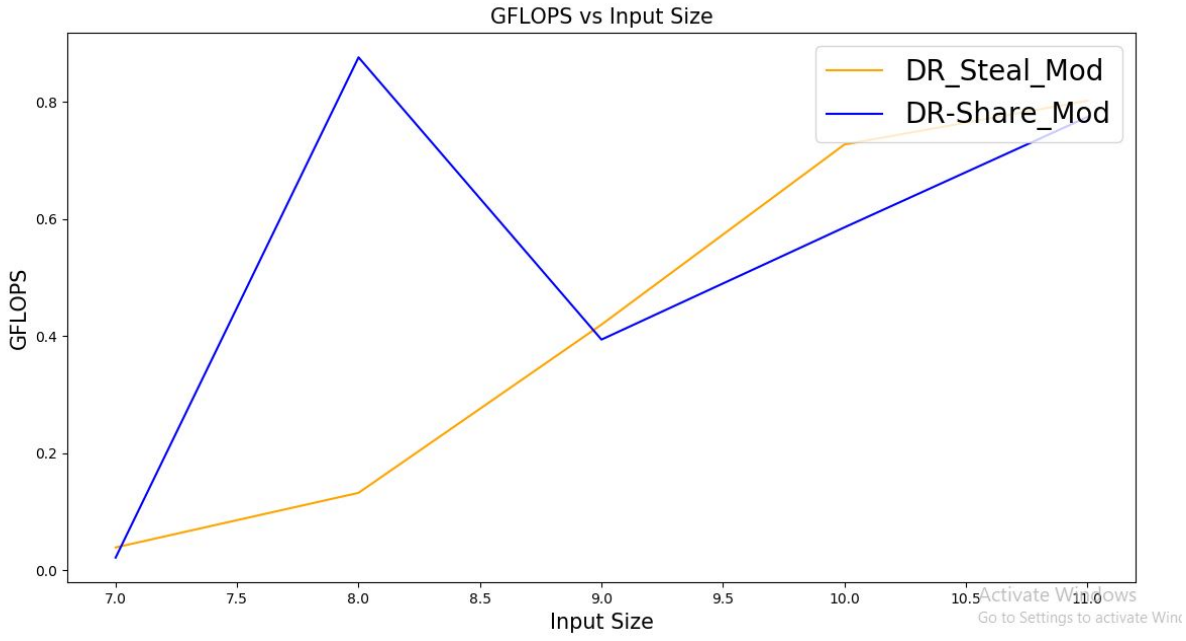
DR-SharedMOD

Number of Cores	**RunTime (sec)**	**Efficiency**
1	18.9247	1
5	3.7252	0.992501
10	1.86471	0.991377
15	1.21012	1.01843
20	0.91603	1.00904
25	0.774783	0.954398
30	0.767742	0.802626
35	0.640757	0.824306
40	0.582761	0.793048

45	0.530037	0.775052
48	0.583001	0.660601

DR-STEALMOD

Number of Cores	RunTime (sec)	Efficiency
1	14.5837	1
5	2.94601	0.99006
10	1.64132	0.88885
20	0.93231	0.78212
30	0.714387	0.69047
40	0.552948	0.65937
48	0.523001	0.660601



---

3. (A)

Given: P threads

This is similar to Coupon Collector problem. There are P deques, and the problem statement is that, we need to check how many attempts each thread should make in order to check all the deques, if it's empty.

In DR-STEAL, each thread has its own deque. Suppose a thread has attempted to steal from n-1 deques, the probability of the thread attempting to steal from one of the unvisited deque is (P - (n-1))/P. Given this scenario, the number of steal attempts needed for a thread to find a non empty deque can be thought of as a geometric distribution (the expectation is 1/p). The number of steal attempts therefore is P/(P - (n-1)). Therefore, for the entire system, the expected number of attempts is,

$$E = P/(P - (P-1)) + P/(P - (P-2)) + \ldots + P/P$$
$$E = P\ [1 + \tfrac{1}{2} + \tfrac{1}{4} + \ldots 1/p]$$
$$E = P \log P$$

Hence, after 2P log P attempts, the thread has checked all the deques in the system.

---

3. (B)

Assume initial condition of a system where there is only one task, and that P - 1 deques are empty.

For example, assume there are P = 20 deques, and the big main task is in deque 3. Assume thread 1 is trying to steal from deques and that it randomly chose deque 2. At the same time, thread 2 found its own deque empty and tried to steal from deque 3. Having found deque 2 empty, thread 1 might steal from deque 3 but only to see it empty because thread 2 might have already stolen from it and pushed sub tasks in its own deque. Likewise thread 1 will find all of the deuques empty and quit.

This doesn't guaranteed that the system has run out of work.

---

3. (C)

Since in DR-STEAL, each thread will check it’s own deque first, and pushes tasks into its own deque, all the work in the system will still be completed.
