Parallel Programming Assignment 1

Akshay Satam (111481679)
Prateek Roy (111481907)

REPORT – ASSIGNMENT 1

Guidelines for running the code:

Question1:

Please use the file matMul.cpp and MatMulRec.cpp to test question 1.
In this file matMul.cpp, we have written different functions for the subparts.
Please find the functions for the subparts:

File matMul.cpp

Question	Function
1a and 1b	Question 1
1d and 1e	Question 2
1i	Question 1i

File MatMulRec.cpp

Question	Function
1g and 1h	Question rec_matrix_mul
1i	Question 1i

You can change the size of the matrix by changing the variable “r”

Actual Report:

1a.

We multiplied the matrices by the given procedures and calculated r to be 10, ie, for r>10, atleast one matrix exceeded 5 minutes. Please find the execution time and the cache misses.

Multiplication	Time (seconds)
Ijk	184.06
Ikj	36.15
Jik	182.40

Jki	187.88
Kij	36.4
Kji	188.24

1b.

Here, the value of r is 10. Please find the L1 and L2 cache misses. We tried to find L3 cache misses but we were unable to find them on Stampede.

Multiplication	**L1 cache miss**	**L2 cache miss**
Ijk	1073345313	1077227238
ikj	34098404	1217484
Jik	1436589870	1078533877
Jki	2745166278	2146832569
Kij	34797879	2126116
Kji	2146162015	2146686890

1c.

In part 1a, we get the following two matrix multiplications to be the fastest – ikj, kij
All the matrix multiplications are serial. Hence, the difference in execution time can be ascribed to the cache mismatches. In these matrix multiplications (ikj and kij) , the total cache mismatches for L1 and L2 are lesser as compared to the other multiplications. Hence, the multiplications are faster.

1d.

We can parallelize the fastest implementations in 3 ways each. Please find the correct parallel algorithms for the two implementations in part 1a:

· ikj 1

```
cilk_for(int  i=0;i<n;i++){
        for(int  k=0;k<n;k++){
        for(int  j=0;j<n;j++){
                        (*z)[i][j]= (*z)[i][j] + (*x)[i][k] * (*y)[k][j];
        }
        }
        }
```

· ikj 2:

```
cilk_for(int  i=0;i<n;i++){
for(int  k=0;k<n;k++){
cilk_for(int  j=0;j<n;j++){
        (*z)[i][j]= (*z)[i][j] + (*x)[i][k] * (*y)[k][j];
}
}
}
```

- ikj 3:

```
for(int  i=0;i<n;i++){
for(int  k=0;k<n;k++){
cilk_for(int  j=0;j<n;j++){
        (*z)[i][j]= (*z)[i][j] + (*x)[i][k] * (*y)[k][j];
}
}
}
```

- Kij 1:

```
for(int  k=0;k<n;k++){
cilk_for(int  i=0;i<n;i++){
cilk_for(int  j=0;j<n;j++){
        (*z)[i][j]= (*z)[i][j] + (*x)[i][k] * (*y)[k][j];
}
}
}
```

- Kij 2:

```
for(int  k=0;k<n;k++){
cilk_for(int  i=0;i<n;i++){
for(int  j=0;j<n;j++){
        (*z)[i][j]= (*z)[i][j] + (*x)[i][k] * (*y)[k][j];
}
}
}
```

- Kij 3:

```
for(int  k=0;k<n;k++){
for(int  i=0;i<n;i++){
cilk_for(int  j=0;j<n;j++){
        (*z)[i][j]= (*z)[i][j] + (*x)[i][k] * (*y)[k][j];
}
}
}
```

Please find the running times of each implementation:

Matrix Multiplication	**Execution Time (Seconds)**	**Execution Time (Seconds)**	**Execution Time (Seconds)**	**Execution Time (Seconds)**	**Execution Time (Seconds)**	**Execution Time (Seconds)**
	S=4	**S=5**	**S=6**	**S=7**	**S=8**	**S=9**
ikj 1	0.000194	0.003	0.001	0.002	0.008	0.054
ikj 2	0.003	0.018	0.006	0.012	0.054	0.387
ikj 3	0.008	0.048	0.314	1.821	10.707	53.468
Kij 1	0.079	0.139	0.362	0.517	0.774	1.259
Kij 2	0.001	0.006	0.014	0.065	0.131	0.325
Kij 3	0.007	0.050	0.306	2.136	10.521	53.199

1e.

We varied the no of processors from 1 to 64. The size of “r” was constant, i.e., 10. Please find the results.

No of cores\Multiplication method	Kij 1	Kij 2	Kij 3	Ijk 1	Ijk 2	Ijk 3
1	105.517	54.055	108.096	50.4	104.018	107.194
2	55.514	28.057	75.997	26.413	55.02	75.162
4	28.79	14.07	69.746	13.414	28.01	68.84
8	15.64	7.18	66.11	6.56	15.20	65.78
16	9.41	4.03	69.73	3.68	8.97	70.28
32	6.02	2.16	79.86	1.84	5.50	78.90
64	4.47	1.18	84.94	0.92	3.56	84.07

1f.

Finding from 1d:

The execution time for every matrix multiplication increases as we increase the size of the matrices. However, the increase is not linear with “s”.

Also, there is a peculiar case with multiplications ikj1 and ikj2. The time for multiplication is more when s = 5 than the time when s = 6.

Finding from 1e:

The multiplication time reduces as we increase the processors. However, the reduction in time is not linear with the number of processors.

1g.

We kept the matrix size to be 1024 * 1024 and varied the M from 2 to 512.
We plotted the timing as follows. We found that the best timing is achieved when M = 7, ie, size of the base-case matrix is 2^7 = 128.
M = 7

2^M	**Time (seconds)**
1	84.79
2	25.53
3	9.79
4	4.51
5	3.06
6	1.99
7	1.66
8	4.00
9	11.9

1h.

We kept the base-case size of the matrices to be 128 * 128 and initial matrix size of 1024 X 1024.
We increased the processors from 1 to 68 and found these results:

No of processors	**Time (seconds)**
1	43.62
2	22.73
4	12.67
8	6.42

16	3.40
32	2.164
64	1.621
68	1.627

Here, we kept on increasing the matrix size and found the execution-times such that the matrix multiplication time did not exceed 1 minute

N	**Time (seconds)**
1024	1.627
2048	7.33
4096	50.023

1i.
We calculated the L1/L2/L3 misses for the fastest algorithm in part 1d (ikj1) and fastest algorithm in part 1g (with bases case m =128):
Part 1d:

Multiplication	**L1 cache misses**	**L2 cache misses**
Ikj1	**371596**	**31343**

Part 1g:

Multiplication	**L1 cache misses**	**L2 cache misses**
PAR-REC-MM with base 128	**3321893**	**320025**

QUESTION 2

2a.

GFlops = (2n*n*n)/(10^9 * RunTime)
The base size taken is 128 which gives the best result.
DRSteal

Size of Matrix (n)	Time (seconds)	GFLOPS
1024	0.078	27.53
2048	0.56	30.67
4096	2.07	66.39

Size greater than 4096 is causing memory limit exceed in stampede 2.

DRShare

Size of Matrix (n)	Time (seconds)	GFLOPS
1024	0.015	143.165
2048	0.09	190.88
4096	1.34	102.56

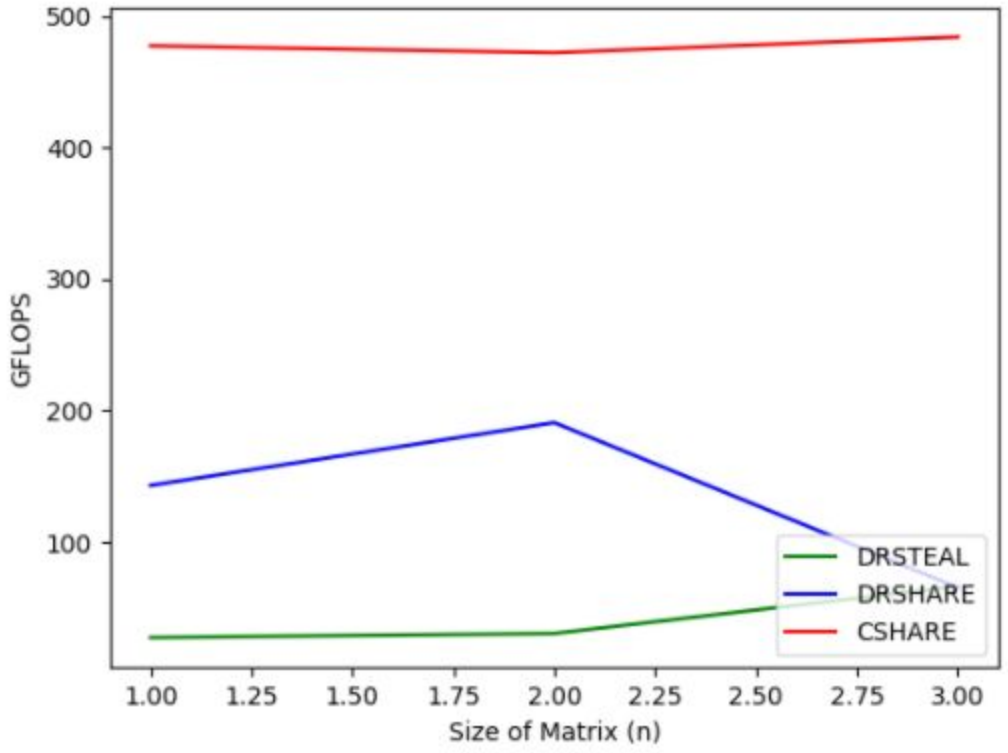
Size greater than 4096 is causing memory limit exceed in stampede 2.

CShare

Size of Matrix (n)	Time (seconds)	GFLOPS
1024	0.0045	477.218
2048	0.0364	471.97
4096	0.2840	483.93

Size greater than 4096 is causing memory limit exceed in stampede 2.

Findings : Here we observe that CShare has the maximum GFlops. This is well evident from the fact that CShare runs most efficiently with least runtime than other two algorithms. The DRSteal takes more time because the random number generator generates a thread queue which has no task in it queue. So there are more fail steal attempts and even these steal attempts takes locks on the threads queue. So DRSteal takes more time than CShare and DRShare.



2b.

GFlops Cache Misses = (3n*n*n)/(TotalCacheMiss)

DRSteal

Size of Matrix (n)	L1 Cache Miss	L2 Cache Miss	L3 Cache Miss	Total Cache Miss	GFlops Cache Miss
1024	5070667	5316719	1366279	11753665	274.06
2048	19916520	59781230	32179182	111876932	230.34
4096	54895625	98478451	75439852	160913928	1281.17

Size greater than 4096 is causing memory limit exceed in stampede 2.

DRShare

Size of Matrix (n)	L1 Cache Miss	L2 Cache Miss	L3 Cache Miss	Total Cache Miss	GFlops Cache Miss

1024	5106	3891	2347	11344	283958.52
2048	48968	17729	12898	79595	323761.59
4096	85412	34469	113588	233469	883022.71

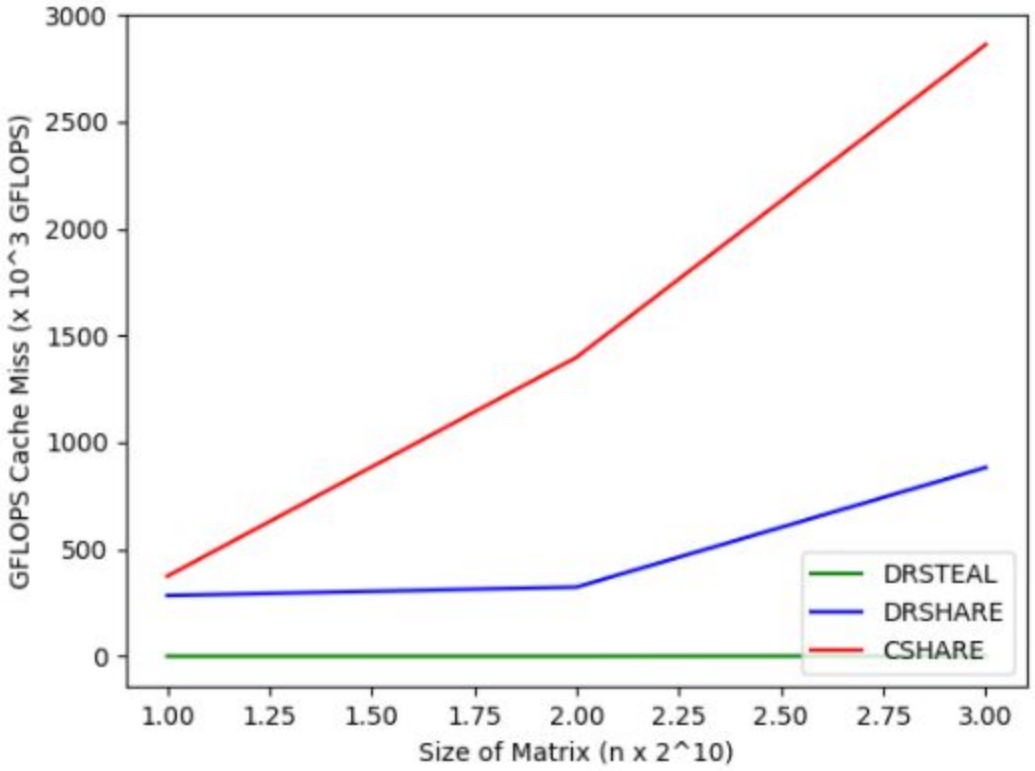
Size greater than 4096 is causing memory limit exceed in stampede 2.

CShare

Size of Matrix (n)	**L1 Cache Miss**	**L2 Cache Miss**	**L3 Cache Miss**	**Total Cache Miss**	**GFlops Cache Miss**
1024	6601	1335	673	8609	374169.52
2048	11502	4401	2529	18432	1398101.33
4096	30867	25649	15538	72054	2861165.65

Size greater than 4096 is causing memory limit exceed in stampede 2.

Findings: DRSteal has the most cache misses than other two algorithms. This is also evident from the fact that DRSteal is least efficient and takes maximum runtime. The cache miss in case of CShare is less because, the subtasks are put in near neighborhood of each other in the global queue, and the individual threads executing these tasks pick the task from the queue in FIFO order. So task are executed in kind of sequential manner and so they are less cache misses.
DRSteal has most cache misses because each threads tries to steal from the top(recently added task) and even some thread run tasks from end of their queue(task added first). So at a single instance of time, the data access are random according to the thread scheduling, and thus the cache misses are high.
DRShare has still moderate cache misses because, each thread executes tasks from its own queue from bottom i.e in a fixed order, thus there is sequential data access. But still there are multiple threads running at a single instance and each tries to put task in others queue, so it may happen that some thread have to jump to a new task(data which is not sequential) and execute it. Thus it will incur some cache misses but better than DRSteal where all the execution of tasks are random.



2c)

Efficiency = T1 / (p * Tp)
T1 = Time taken when there are 1 core = 2.98 seconds

DRSteal:

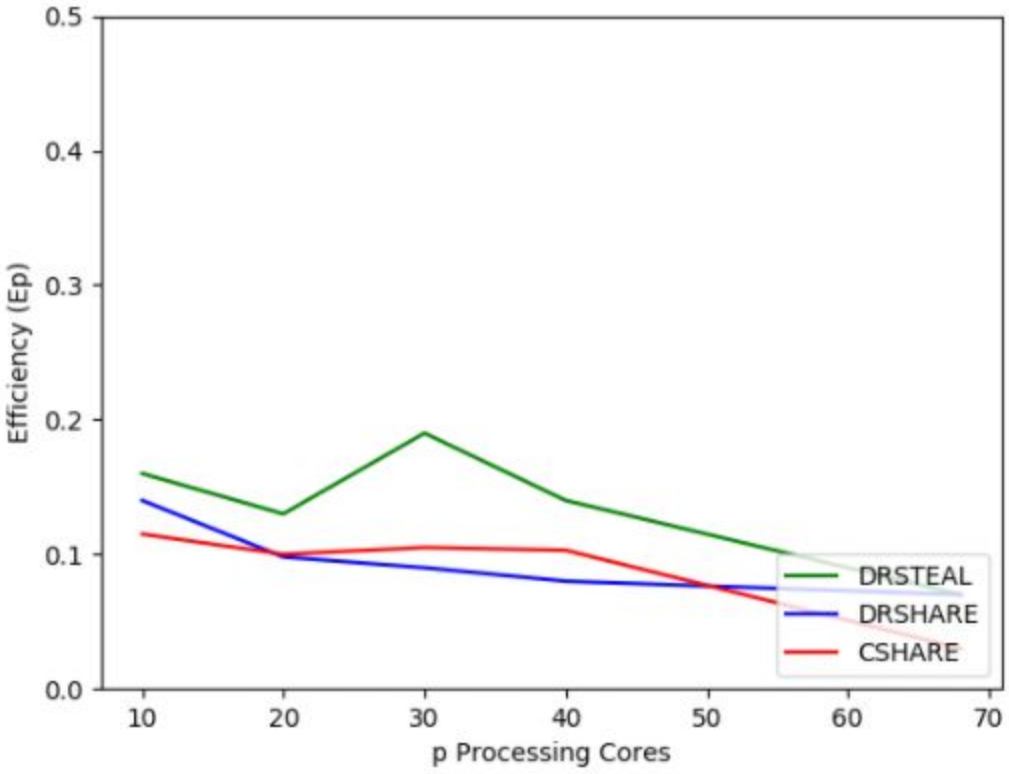
Size of Matrix (n)	**No of Cores (p)**	**Time (seconds)**	**Efficiency**	**1-Efficiency**
2048	10	1.79	0.16	0.84
2048	20	1.08	0.13	0.87
2048	30	0.511	0.19	0.81
2048	40	0.52	0.14	0.86
2048	68	0.56	0.07	0.93

DRShare:

Size of Matrix (n)	No of Cores (p)	Time (seconds)	Efficiency	1-Efficiency
2048	10	2.09	0.14	0.86
2048	20	1.52	0.098	0.902
2048	30	1.09	0.09	0.91
2048	40	0.84	0.08	0.92
2048	68	0.62	0.07	0.93

CSHARE:

Size of Matrix (n)	No of Cores (p)	Time (seconds)	Efficiency	1-Efficiency
2048	10	2.58	0.115	0.885
2048	20	1.49	0.1	0.9
2048	30	0.94	0.105	0.895
2048	40	0.72	0.103	0.897
2048	68	0.28	0.03	0.97



Findings: Efficiency of all the algorithms initially increase with increase in cores. But after 30 cores it decreases a bit. The cause of this behaviour is whenever the number of cores increases, that means there are more number of threads involved and thus more locks for some work queue. Thus due to many locks trying to steal/share work in job queue, the efficiency decreases. Basically many threads are simultaneously trying to enter the critical section and thus many threads go to sleep and hence the efficiency decreases.

2d)

DRSteal-MOD

Size of Matrix (n)	**Time (seconds)**	**GFLOPS**
1024	0.17	12.63
2048	0.67	25.64
4096	1.98	69.41

Size of Matrix (n)	L1 Cache Miss	L2 Cache Miss	GFlops Cache Miss
1024	2625304	5837165	380.64
2048	19723794	44653875	400.3
4096	205125465	565412358	267.55

DRShare-MOD

Size of Matrix (n)	Time (seconds)	GFLOPS
1024	0.011	195.225
2048	0.075	229.06
4096	1.51	91.15

Size of Matrix (n)	L1 Cache Miss	L2 Cache Miss	GFlops Cache Miss
1024	9723	14016	135710.5
2048	37816	19037	453270.78
4096	458795	25648	425557.66

In DRShare-MOD, we put the task in the queue of a thread who has minimum task. In DRSteal-MOD, we steal task from the queue of a thread who has least task in its queue. The performance is similar to the original DRShare and DRSteal as in 2a.
Findings: DRSteal-MOD has the maximum Cache misses as well as the runtime as compared to DRShare-MOD. But the performance of DRSteal-MOD and DRShare-MOD has improved slightly from DRSteal and DRShare because now we make decision based on where workload is less before stealing or sharing work, and thus evenly distributing the tasks. But we can see the cache misses has increased from the basic versions of these algorithms because again the work is distributed based on even distribution and not based on temporal locality of the tasks thus increasing the cache misses.

Q3a.

Probability of stealing from some deque (say deque d) = 1/p

Probability of not stealing from deque d = 1 - 1/p

Probability of not stealing from deque d after pk times = $(1 - 1/p)^{pk}$

We know that $(1-1/p)^{p} = 1/e$

Hence, Probability of not stealing from deque d after pk times = $(1/e)^{k}$

Let, k = c * log p

Hence, Probability of not stealing from deque d after pk times = $(1/e)^{c^* \log p}$

Hence, Probability of not stealing from deque d after pk times = $1/p^{c}$

Now, c = k/ log p

Hence, as we increase k , that it, the number of attempts, the Probability of not stealing from deque d is very low. Hence, the probability of not missing any processor is very high.

Now, let us calculate the value of the number of attempts, which is pk.

Hence, pk = p * c * log p

Hence, number of attempts = c * p log p

Putting c =2, we get number of attempts = 2 * p log p

Thus , we have proved that if we do 2 p log p attempts, we will not miss any processor with high probability in p.

Q3b.

Even if a thread finds all the deques empty, there may be other threads who are still working on some task. These threads may then put some tasks on their deque. Hence, the thread which previously checked for these deques will think they are empty, even though there is work in the system.

Q3c.

Lets say, all threads check for other deques as suggested in part 3a. This will ensure that they check all the deques and will pick up tasks, if found on some deque.

But as proved in part 3b, even if the thread misses work on some deque due to checking earlier than when the work was actually put, the thread which put the work in it's deque will definitely pick it up after it is done with the current task.

Hence, it is guaranteed that all the work in the system will be completed.

Q3d.

Referred from: https://www.corelab.ntua.gr/courses/rand-alg/slides/balls-bins.pdf

We have to find a sequence of values b_i such that the number of enqueues to any queue with i tasks is bounded by b_i with high probability in n. Now, let us say we know b_i. Then we will have to calculate b_{i+1}.

A queue has at least i+1 tasks if there are d attempts to enqueue a queue with i tasks.

Therefore, the probability that a queue has atleast i+1 tasks is at most $(b_i / n)^{d}$.

Using Chernoff bounds, it follows that $b_{i+1} <= cn (b_i /n)^{d}$ for constant c, so by selecting j = O(ln ln n) we are done.

Hence, it is proved that if there is a sequence of p consecutive enqueues each deque undergoes O(ln p / ln ln p) enqueue operations w.h.p. in p.

Q3e.

i)
F_i is the fraction of deque that has received i tasks during the p enqueue attempts. As there are p cores, that means there are p dequeues. So if f_i is the fraction of deques, that means only $p*f_i$ Deques have i tasks in them and their rank is i.
So $f_i = p*f_i$.
Now to add 1 task in each of the dequeue with i tasks, and assuming no two enqueues are occuring at same time,
$f_{i+1} = (1/p)^p*f_i$ as Probability to put a task in a dequeue is 1/p. So total probability to add 1 task to i dequeues is $p*f_i$.

As we can see $(1/p)^p*f <= p*f_i$ p
Hence $f_{i+1} <= f_i$ (Answer)

ii)
We know that if there is a sequence of p consecutive enqueues each deque undergoes O(ln p / ln ln p) enqueue operations w.h.p. in p from 3d.
That means $f_i <= (½^(2^(i-1)))$.

iii) As each deque undergoes O(ln p / ln ln p) work, then no task can have rank lnp * (ln ln p)/lnp = ln ln p (Answer)

For question 2 , we are running on Login Node and not Compute Node.