CS701 - PPA - 1
Open Multiprocessing (OpenMP)

Team Members
Keerti Chaudhary (181CO226) Trivedi Naman Manishbhai(181CO156)

1)

```
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q1$ g++ -o hello q1.cpp -fopenmp
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q1$ ./hello
Hello: 0
World: 0
Hello: 1
World: 1
Hello: 3
World: 3
Hello: 2
World: 2
Hello: 4
World: 4
```

The number of Threads = 5. Hence “Hello” and “World” are printed 5 times.

2)

```
ments/OpenMP/Q2$ ./a.out
Hello World! 3
Hello World! 2
Hello World! 0
Hello World! 1
```

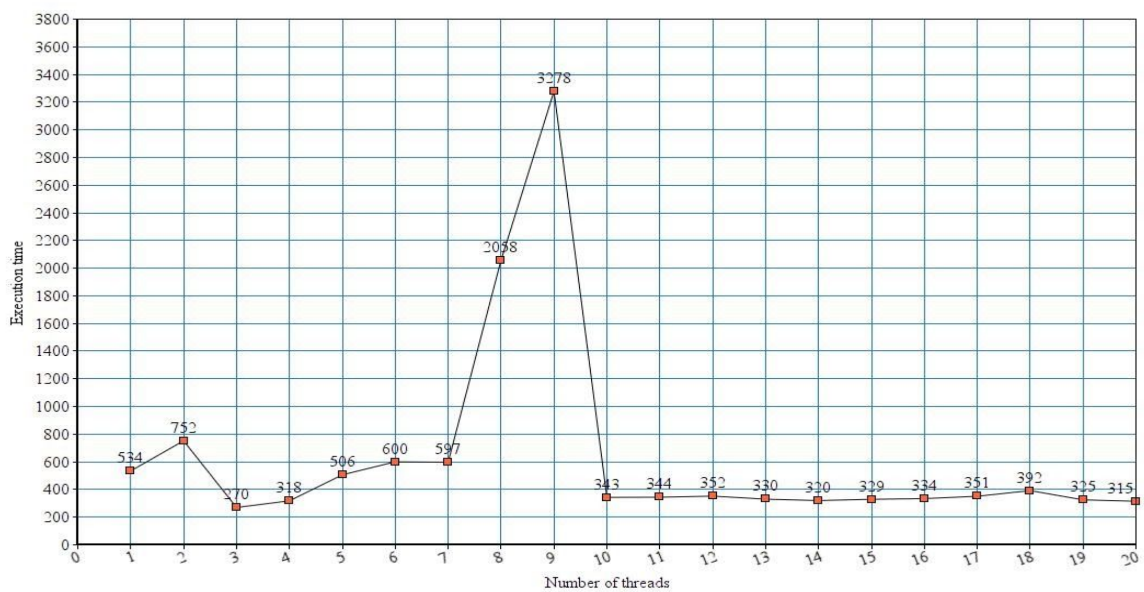
The number of threads is 4. Hence thread ID ranges from 0 to 3.

3)

```
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q3$ g++ -o daxpy q3.cpp -fopenmp
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q3$ ./daxpy
Num_threads      Exec Time (micro-s)
1               534
2               752
3               270
4               318
5               506
6               600
7               597
8               2058
9               3278
10              343
11              344
12              352
13              330
14              320
15              329
16              334
17              351
18              392
19              325
20              315
```

When the number of threads is increased further, runtime increases since the number of threads are greater than the number of cores in the system. Hence time is consumed in scheduling the excess threads by operating system among the cores.

Exec time vs no of threads

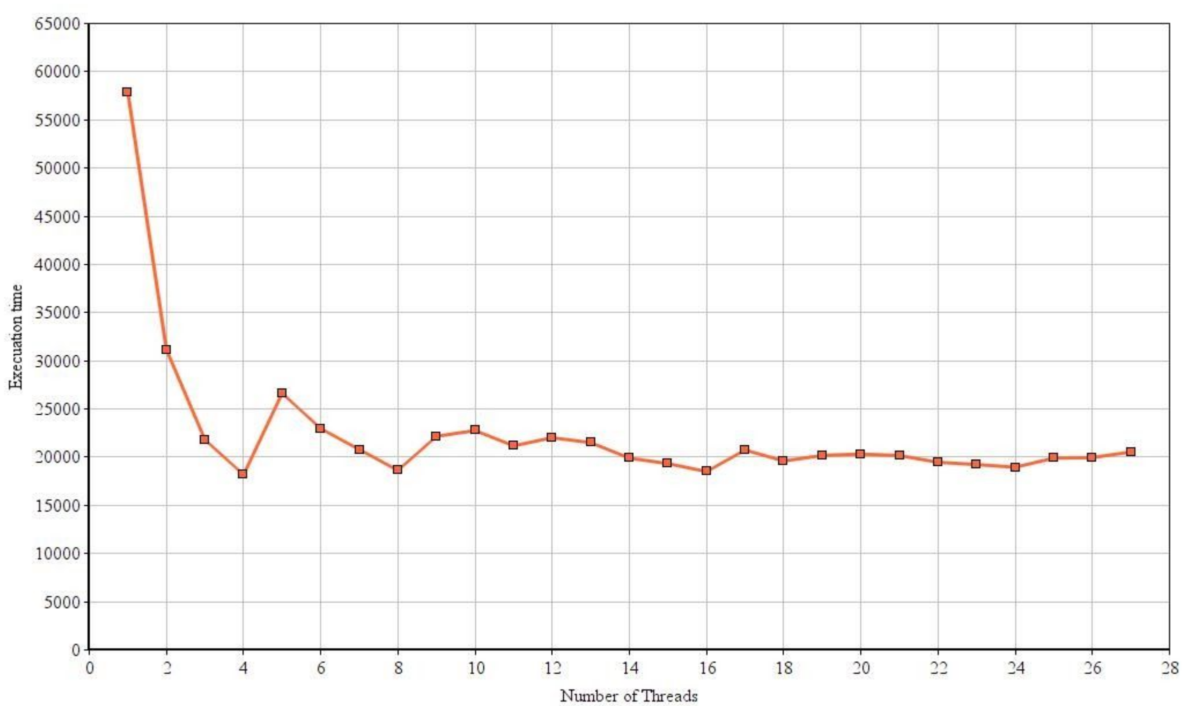


Speedup = T1/Tn, hence maximum speedup will be when Tn is smaller, therefore maximum speedup will be when the number of threads = 3

4)

```
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q4$ g++ -o matrix q4.cpp -fopenmp
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q4$ ./matrix
Num_threads      Exec Time (micro-s)
1                57881
2                31154
3                21797
4                18183
5                26631
6                22963
7                20790
8                18631
9                22161
10               22762
11               21175
12               22022
13               21504
14               19899
15               19327
16               18505
17               20743
18               19598
19               20181
20               20322
21               20164
22               19467
23               19235
24               18956
25               19902
26               19952
27               20538
```

Matrix multiplication done using parallel calculations speeds up the process as seen in the output. The matrix multiplication algorithm 3 loops nested with a complexity of O(n^3). We fork the master thread at each level of the loop to obtain the best speedup. The innermost loop sums up the product of the corresponding row and column. This is done using the reduction operation provided by OpenMP.



5)

```
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q5$ ./calculate
Value of Pi by Sequential Calculation  : 3.141593
Parallel Calculation
Pi :  3.141593   Speedup: 0.489552      Threads : 2
Pi :  3.141593   Speedup: 0.332804      Threads : 3
Pi :  3.141593   Speedup: 0.271645      Threads : 4
Pi :  3.141593   Speedup: 0.239640      Threads : 5
Pi :  3.141593   Speedup: 0.207211      Threads : 6
Pi :  3.141593   Speedup: 0.181080      Threads : 7
Pi :  3.141593   Speedup: 0.221869      Threads : 8
Pi :  3.141593   Speedup: 0.255792      Threads : 9
Pi :  3.141593   Speedup: 0.232992      Threads : 10
Pi :  3.141593   Speedup: 0.214680      Threads : 11
Pi :  3.141593   Speedup: 0.206145      Threads : 12
Pi :  3.141593   Speedup: 0.185534      Threads : 13
Pi :  3.141593   Speedup: 0.192922      Threads : 14
Pi :  3.141593   Speedup: 0.167830      Threads : 15
Pi :  3.141593   Speedup: 0.186283      Threads : 16
Pi :  3.141593   Speedup: 0.202870      Threads : 17
Pi :  3.141593   Speedup: 0.200251      Threads : 18
Pi :  3.141593   Speedup: 0.197134      Threads : 19
Pi :  3.141593   Speedup: 0.182535      Threads : 20
```

Shared variables are

1. sum: which stores the area under the curve

Private variables are

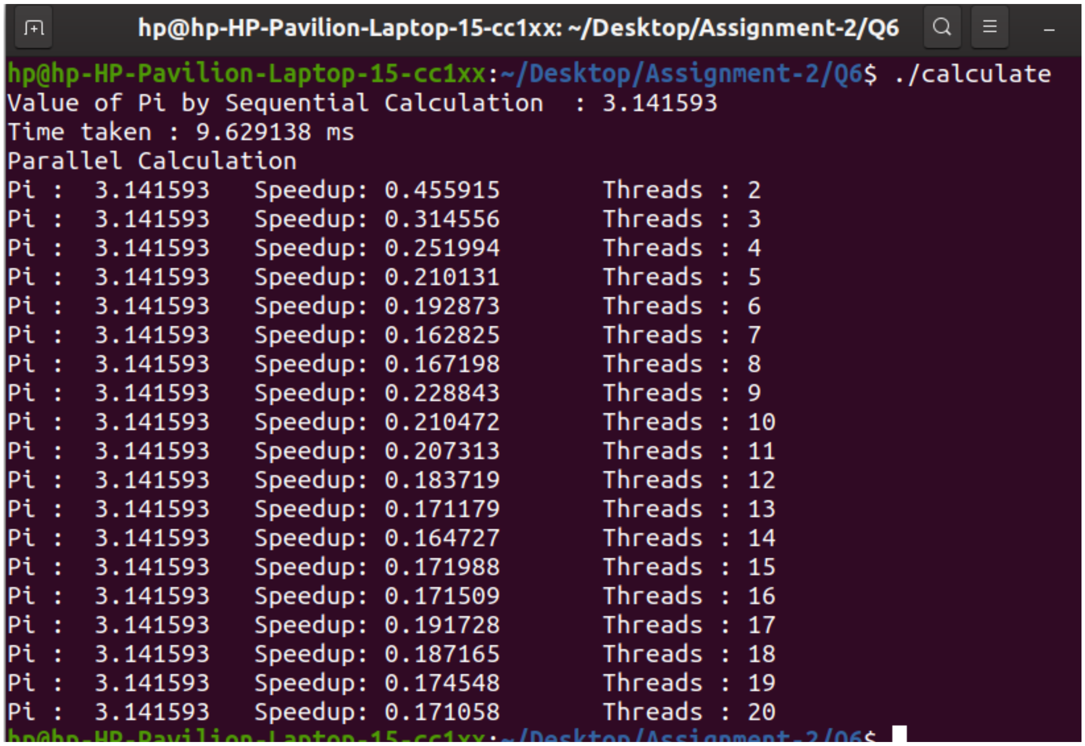
1. ID: thread ID

2. increment: loop increment to split the loop iteration among the threads

3. t_sum: to calculate the sum for each thread

Since every thread spawned calculates its sum, the total sum has to be summed up. This sum variable is shared but cannot be simultaneously updated by multiple threads. Hence its updating is given inside a “critical” construct which ensures that only one thread updates its value at a time.

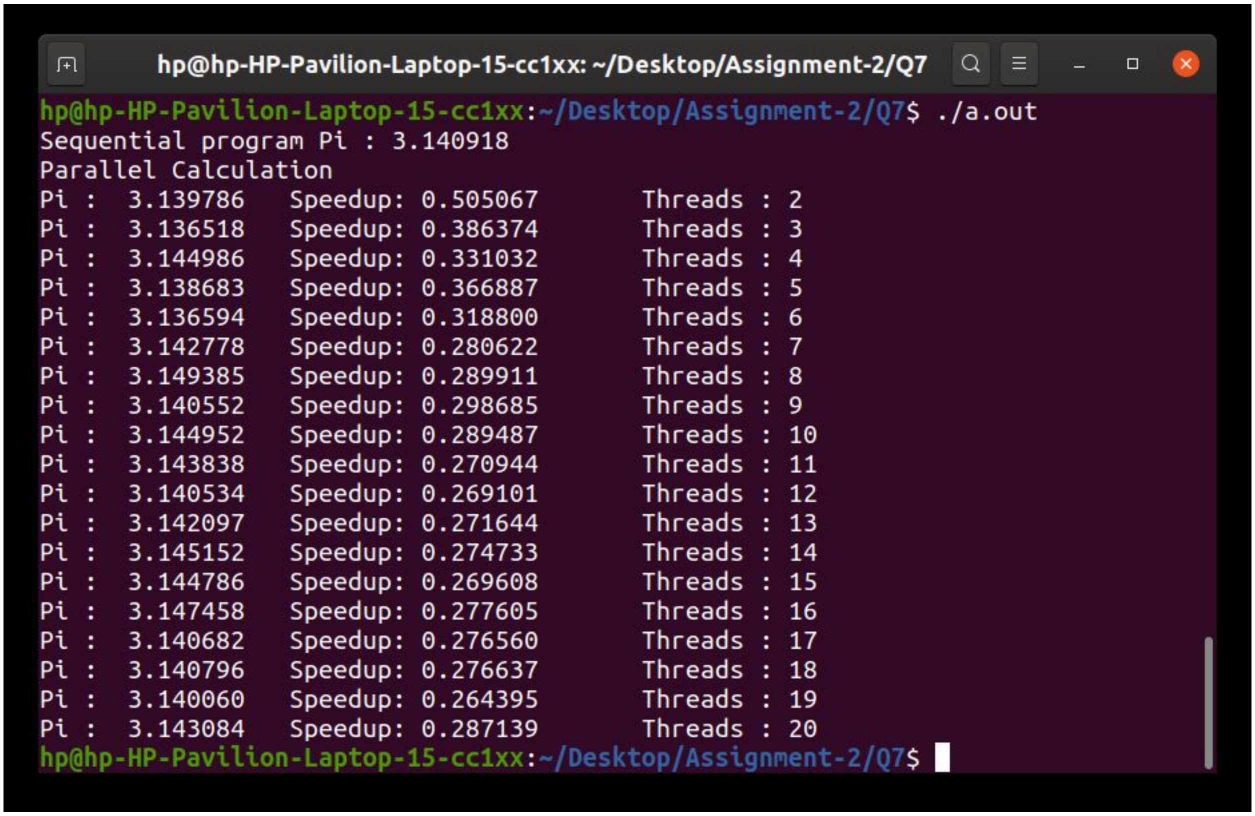
6)



```
hp@hp-HP-Pavilion-Laptop-15-cc1xx: ~/Desktop/Assignment-2/Q6
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q6$ ./calculate
Value of Pi by Sequential Calculation  : 3.141593
Time taken : 9.629138 ms
Parallel Calculation
Pi :  3.141593   Speedup: 0.455915       Threads : 2
Pi :  3.141593   Speedup: 0.314556       Threads : 3
Pi :  3.141593   Speedup: 0.251994       Threads : 4
Pi :  3.141593   Speedup: 0.210131       Threads : 5
Pi :  3.141593   Speedup: 0.192873       Threads : 6
Pi :  3.141593   Speedup: 0.162825       Threads : 7
Pi :  3.141593   Speedup: 0.167198       Threads : 8
Pi :  3.141593   Speedup: 0.228843       Threads : 9
Pi :  3.141593   Speedup: 0.210472       Threads : 10
Pi :  3.141593   Speedup: 0.207313       Threads : 11
Pi :  3.141593   Speedup: 0.183719       Threads : 12
Pi :  3.141593   Speedup: 0.171179       Threads : 13
Pi :  3.141593   Speedup: 0.164727       Threads : 14
Pi :  3.141593   Speedup: 0.171988       Threads : 15
Pi :  3.141593   Speedup: 0.171509       Threads : 16
Pi :  3.141593   Speedup: 0.191728       Threads : 17
Pi :  3.141593   Speedup: 0.187165       Threads : 18
Pi :  3.141593   Speedup: 0.174548       Threads : 19
Pi :  3.141593   Speedup: 0.171058       Threads : 20
```

The reduction clause is used to sum the area under the curve. Every thread creates its local copy of the sum and then the reduction is done using the operator specified in *reduction (op: list)*.

7)



```
hp@hp-HP-Pavilion-Laptop-15-cc1xx: ~/Desktop/Assignment-2/Q7
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q7$ ./a.out
Sequential program Pi : 3.140918
Parallel Calculation
Pi :  3.139786   Speedup: 0.505067      Threads : 2
Pi :  3.136518   Speedup: 0.386374      Threads : 3
Pi :  3.144986   Speedup: 0.331032      Threads : 4
Pi :  3.138683   Speedup: 0.366887      Threads : 5
Pi :  3.136594   Speedup: 0.318800      Threads : 6
Pi :  3.142778   Speedup: 0.280622      Threads : 7
Pi :  3.149385   Speedup: 0.289911      Threads : 8
Pi :  3.140552   Speedup: 0.298685      Threads : 9
Pi :  3.144952   Speedup: 0.289487      Threads : 10
Pi :  3.143838   Speedup: 0.270944      Threads : 11
Pi :  3.140534   Speedup: 0.269101      Threads : 12
Pi :  3.142097   Speedup: 0.271644      Threads : 13
Pi :  3.145152   Speedup: 0.274733      Threads : 14
Pi :  3.144786   Speedup: 0.269608      Threads : 15
Pi :  3.147458   Speedup: 0.277605      Threads : 16
Pi :  3.140682   Speedup: 0.276560      Threads : 17
Pi :  3.140796   Speedup: 0.276637      Threads : 18
Pi :  3.140060   Speedup: 0.264395      Threads : 19
Pi :  3.143084   Speedup: 0.287139      Threads : 20
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q7$
```

Random numbers are generated using the **Linear Congruential Generator** defined by

X(n+1) = (a*X(n) + c) mod m

Bonus part

1. Random number generator was made threadsafe using the “#pragma omp thread private(seed)” construct in OpenMP. This allows each thread to have its private seed.

2. The random number generator was made numerically correct, ie with non-overlapping sequences of pseudo-random numbers by using the leapfrog algorithm.

This can be done for linear congruential generators and involves replacing the multiplier a and the additive constant c by new values a^N and c(a^N – 1)/(a-1), where N is the number of threads.

8)

```
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q8$ gcc -o pc q8.c -fopenmp
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q8$ ./pc
Producer populated data
Consumer calculated Array sum
In 0.002724 seconds, Sequential code gives sum : 100000.000000
Producer populated data
Consumer calculated Array sum
In 0.000460 seconds, Parallel code gives sum : 100000.000000
Speed up : 0.168758
```

Two threads, one for producer and one for the consumer was created. The sections clause was used to reserve each section for one thread.

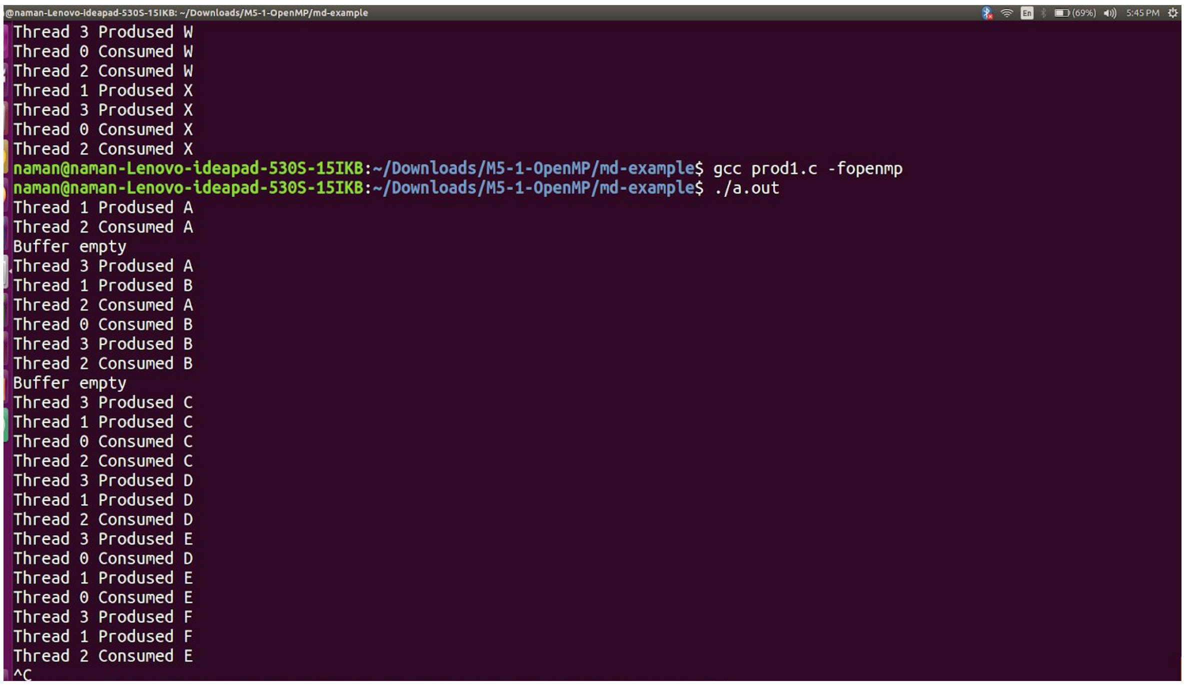
The consumer knows the producer has completed execution by getting notified by a global *flag.*

9)

```
hp@hp-HP-Pavilion-Laptop-15-cc1xx: ~/Desktop/Assignment-2/Q9
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q9$ gcc -o pc q9.c -fopenmp
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q9$ ./pc
Buffer empty
Thread 1 Prodused A
Thread 0 Consumed A
Thread 1 Prodused B
Thread 1 Prodused C
Thread 0 Consumed B
Thread 0 Consumed C
Thread 1 Prodused D
Thread 1 Prodused E
Thread 0 Consumed D
Thread 1 Prodused F
Thread 0 Consumed E
Thread 0 Consumed F
Thread 1 Prodused G
Thread 0 Consumed G
Thread 1 Prodused H
Thread 1 Prodused I
Thread 0 Consumed H
Thread 1 Prodused J
Thread 0 Consumed I
```

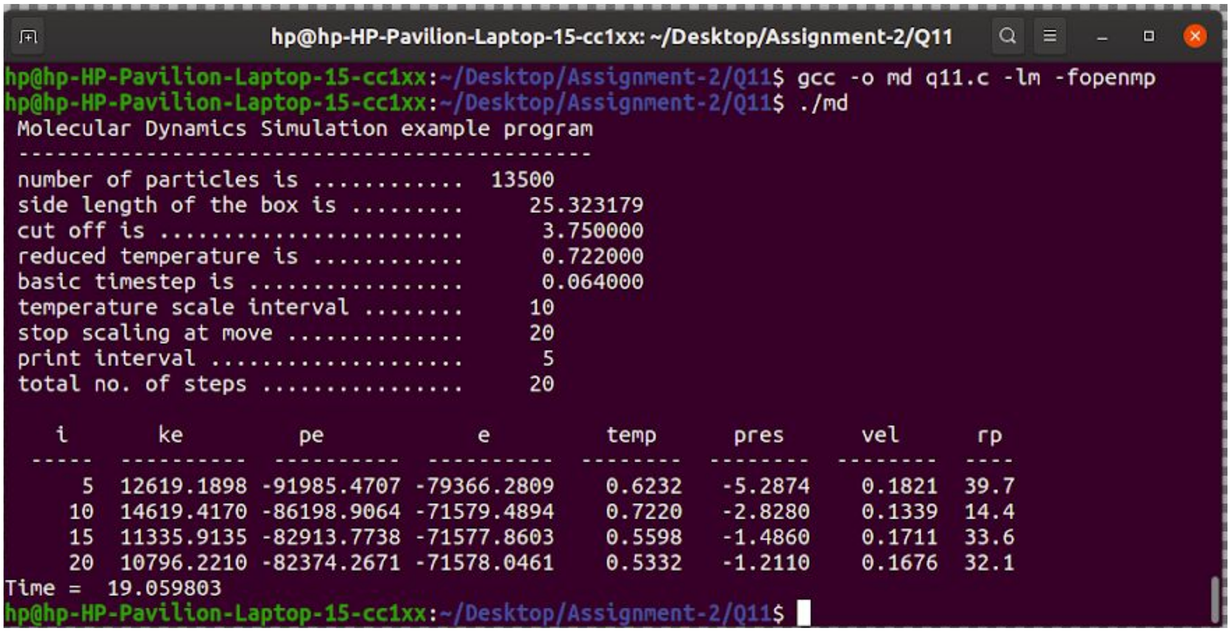
Here we have to introduce the sleep statement to make sure that that none of them are running multiple times without the other getting a chance.
As for the race condition, the critical section problem may occur due to both updating the same values. So we kept it under the critical section. OpenMP gives APIs who implement locks for us to ensure the atomic behavior for that section.

10)



Here to avoid a producer or consumer entering while any other producer or consumer is in the critical section we define a globe critical section. Synchronous behaviour can not be achieved as there may be too many processes in the ready queue of some a processor executing a particular thread thereby delaying its turn. Here using a critical section helps avoids us from ending into an infinite loop.

11)



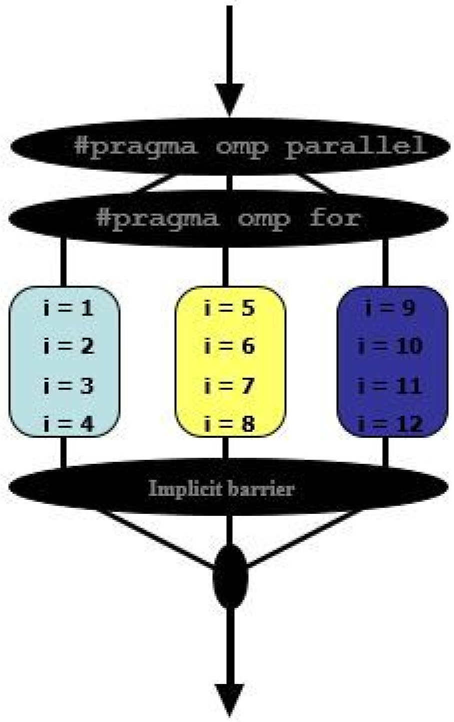
```
hp@hp-HP-Pavilion-Laptop-15-cc1xx: ~/Desktop/Assignment-2/Q11
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q11$ gcc -o md q11.c -lm -fopenmp
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q11$ ./md
 Molecular Dynamics Simulation example program
 ---------------------------------------------
 number of particles is ............  13500
 side length of the box is .........     25.323179
 cut off is ........................      3.750000
 reduced temperature is ............      0.722000
 basic timestep is .................      0.064000
 temperature scale interval ........     10
 stop scaling at move ..............     20
 print interval ....................      5
 total no. of steps ................     20

    i        ke          pe           e         temp      pres      vel      rp
  -----  ----------  ----------  ----------   --------  --------  --------  ----
      5  12619.1898 -91985.4707 -79366.2809    0.6232   -5.2874    0.1821  39.7
     10  14619.4170 -86198.9064 -71579.4894    0.7220   -2.8280    0.1339  14.4
     15  11335.9135 -82913.7738 -71577.8603    0.5598   -1.4860    0.1711  33.6
     20  10796.2210 -82374.2671 -71578.0461    0.5332   -1.2110    0.1676  32.1
Time =  19.059803
hp@hp-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/Assignment-2/Q11$
```

The key things to note in the parallelized code:-

- Here loop iterations are independent so the order of their execution doesn't matter and thus they can be parallelized
- Here we have parallelized the outer food loop
- Variable j needs to be private as every loop might be executing different iterations
- We don't need to bother with the local variables of the section
- Variables epot and vir are being accumulated in every iteration hence they will be our reduction variables
- There are sections in which f[i] and f[j] are updated, we do not need to worry about f[i] as it has been taken care of by omp for, but the f[j] update needs to be handled as it should not be updated simultaneously by all the threads, values j might be same for many, thus it is being declared under the critical section.

Ans1. How does the data-sharing change?

This is the general functioning of omp. The task is distributed equally for iterations amongst all the threads and in the end, partial-sums of each thread are summed together to get the final result.



Ans2. Here as there was a race condition in the update of f[j] thus we had to introduce omp atomic/critical for resolving it. So if we use different arrays for each thread then there won't occur any race condition, this will save us from bottleneck occurring due to atomic operations. In the end, they are going to executed serially after this for a loop as one thread, the master thread thus the value in the master thread should be correct and accumulation of all.