# NAME : SATYA AKHIL CHOWDARY
# CLASS NO. : Parallel Data Processing SEC-01
# HW1

**Output for Program (MIN, MAX, AVG) :**

MAX of SEQ Run : 6013
MIN of SEQ Run : 2307
AVG of SEQ Run : 4052.0
MAX of SEQFib Run : 10937
MIN of SEQFib Run : 7334
AVG of SEQFib Run : 8867.0
MAX of NOLOCK Run :3952
MIN of NOLOCK Run : 881
AVG of NOLOCK Run : 1997.0
MAX of NOLOCKFib Run : 4722
MIN of NOLOCKFib Run : 2150
AVG of NOLOCKFib Run : 3308.0
MAX of FINELOCK Run : 3152
MIN of FINELOCK Run : 795
AVG of FINELOCK Run : 1697.0
MAX of FINELOCKFib Run : 4576
MIN of FINELOCKFib Run : 2108
AVG of FINELOCKFib Run : 2972.0
MAX of COARSELOCK Run : 3527
MIN of COARSELOCK Run : 899
AVG of COARSELOCK Run : 1900.0
MAX of COARSELOCKFib Run : 7980
MIN of COARSELOCKFib Run : 5412
AVG of COARSELOCKFib Run : 6087.0
MAX of NOSHARING Run : 4086
MIN of NOSHARING Run : 823
AVG of NOSHARING Run : 1931.0
MAX of NOSHARINGFib Run : 4754
MIN of NOSHARINGFib Run : 2163
AVG of NOSHARINGFib Run : 3393.0

**Worker Threads Used :** 8

**SPEED UPS :**

**1) NOLOCK :**4052/1997 = 2.02904
**2) FINELOCK :** 4052/1697 = 2.38774
**3) COARSELOCK :** 4052/1900 = 2.13263
**4) NOSHARING :** 4052/1931 = 2.09839

1. **Which program version (SEQ, NO-LOCK, COARSE-LOCK, FINE-LOCK, NO-SHARING) would you normally expect to finish fastest and why? Do the experiments confirm your expectation? If not, try to explain the reasons.**

**A)** I would expect FineLock to run fastest of all. The reason behind this being, fine lock holds lock over single (k,v) pair instead of whole data structure while allowing other threads to process the data on other (k1,v) pairs in parallel. Being a mulit-threading program, it can even pre-process the data in parallel before even working on shared data structure. Yes, the results confirm to my understanding. Avg run time of Fine Lock is 1697ms, which is fastest of all.

2. **Which program version (SEQ, NO-LOCK, COARSE-LOCK, FINE-LOCK,NO SHARING ) would you normally expect to finish slowest and why? Do the experiments confirm your expectation? If not, try to explain the reasons.**

**A)** I would expect SEQ to run slowest of all. The reason behind this is, the program runs sequentially while processing each input and during some pre-processing over it before adding to the shared data structure. Since the data is being processed in general, it will take more time than multi-threading versions. Even if we consider, updating the shared data structure to be of negligible time, the pre-processing has to be done in sequential, so it will be the slowest of all. Yes, my results confirm to this. Avg run time of SEQ is 4052ms.

3. **Compare the temperature averages returned by each program version. Report if any of them is incorrect.**

**A)** The average times of SEQ, FINELOCK, COARSELOCK, NOSHARING are same. However, NOLOCK results are different, since the there are no locks in the program, one thread can write the value while other thread is reading the value.

4. **Compare the running times of SEQ and COARSE-LOCK. Try to explain why one is slower than the other. (Make sure to consider the results of both B and C—this might support or refute a possible hypothesis.)**

**A)** SEQ runs slower than CoarseLock. The reason behind this is the pre-processing done on input data before updating the accumulation data structure. In our program, we are splitting the string and checking if the string part equals ‘TMAX’.

In Coarse, this can happen in parallel in 8 threads, where as in SEQ it has to be done sequentially. Since, the SEQ and COARSE doesn't have any difference in terms of computation apart from pre-processing, I believe even if we increase the computation before updating accumulation data structure, the deviation will stay close to a constant. Since the same computation will also be executed by SEQ as well, however this may not be the case if we are do to complex pre-processing on each thread.

**5 How does the higher computation cost in part C (additional Fibonacci computation) affect the difference between COARSE-LOCK and FINE-LOCK? Try to explain the reason.**

A) With Higher computation, we can see much more deviation between Fine Lock and Coarse Lock. The reason behind this, Coarse Lock holds the lock over whole data structure while computation is being done. If the computation is complex, the data structure is locked for more time and hence delaying other threads to access the data structure. Fine Lock on other hand, only holds lock over single (key, value) entry, therefore even if the computation is complex other threads can access other entries. So, the higher computation goes more the difference between run times of Fine Lock and Coarse Lock.

**Explanation :**

**FINELOCK.java Line:29**

```
if(tempobj != null) {
                updateobj(parts[3],tempobj);
        }
        else {
                tempobj = FINELOCK.tempmap.putIfAbsent(parts[0],new
ValObj(Double.parseDouble(parts[3]), 1));
                if(tempobj!= null) {
                        updateobj(parts[3],tempobj);
                }
        }
```

The flow will be, the first thread t1, checks if hashmap.get(k) is not null i.e basically straight update after locking the object. Let's assume two threads t1, t2 trying to put some value on same key for the first time in the hashmap. Now, the first if condition fails, so t1 goes to else, where it tried to do putifabsent. Putifabsent returns null, if its putting the value for the first time while getting a lock, if the key is already present, that value will be returned. In the mean time, the thread t2 can't go forward due to the lock from t1. So, once t1 releases lock, t2 will go forward with processing.

The reasons behind picking Concurrent Hashmap for Fine Lock is it locks over single entry unlike Synchronizedmap and also disallows use to null keys and values. It provides putifabsent method which allows to do contest between multiple threads.