2202-COL380 Minor 1

Chinmay Mittal

TOTAL POINTS

37.5 / 50

QUESTION 1

11 pts

1.1 **2 / 2**

✓ **+ 0.5 pts** *correct answer : 1*

✓ **+ 1.5 pts** *Since the critical section is anonymous, at most one thread can execute it simultaneously*

- 2 pts incorrect

- 1.5 pts correct answer but incorrect explanation

1.2 **2 / 2**

✓ **+ 0.5 pts** *correct answer = no. of threads available*

✓ **+ 1.5 pts** *each thread can execute its own named critical section*

- 2 pts incorrect

- 1.5 pts correct snswer but incorrect explanation

1.3 **2 / 2**

✓ **+ 1 pts** *Correct answer (Only if the writer of x and y are different threads)*

✓ **+ 1 pts** *Correct justification (To be processor consistent, if there are two different variables, X and Y, from two different threads, then the order between them need not be seen consistently by a different thread.)*

+ 0 pts Incorrect

1.4 **0 / 3**

+ 1 pts There can be "many" L1 cache misses

+ 2 pts Correct Justification: Misses during Virtual Address translation

✓ **+ 0 pts** *Incorrect*

1.5 **1 / 2**

+ 0 pts Answer: 0. Correct but no/incorrect explanation

✓ **+ 1 pts** *There may be zero misses because the integer might already be in the cache*

+ 1 pts Memory might not even be accessed and the integer might be loaded from a register

+ 0 pts Incorrect

+ 1 pts 1 miss on first access and then 0 subsequently

QUESTION 2

2 **2 / 2**

✓ **+ 1 pts** *'a' might never be read from memory in non-master threads, and even though the master thread flushes the updated value of a to memory, the other threads could still keep seeing 0. Hence that would result in an infinite busy wait and no output*

✓ **+ 1 pts** *'a' may implicitly be flushed for other threads. But a++ is not an atomic operation and hence other threads can print any value of a (>1).*

+ 0 pts Incorrect/No Explanation

QUESTION 3

3 **1 / 2**

✓ **+ 0.5 pts** *Correct example. The example needs to invoke the notion of the duration for each operation. For example, a read in thread B started after a write completed in thread A must see the effects of that write in a Linearizable system. They may be reordered (in the same way in every thread's view, of course) and still maintain sequential consistency.*

+ 0.5 pts Correct explanation of sequential consistency

✓ **+ 0.5 pts** *Correct explanation of non-linearizability.*

+ 1.5 pts Correct explanation of both sequential consistency and non-linearizability for the example.

+ 0 pts Unattempted/Incorrect answer.

1 can't say that the history is sequential consistency without the knowledge of which operation is read and which is write

QUESTION 4

4 **3 / 3**

✓ **+ 1 pts** *Correct example*

+ 1 pts Partially Correct explanation of why linearizability is desirable

✓ **+ 2 pts** *Correct explanation of why linearizability is desirable in the example. (Linearizability is relevant when there is a notion of time visible to the user. Consider, for example, an ATM deposit into some account just before a withdrawal. Because there is not overlap, the user expects that the deposit will credit before the withdrawal posts. Sequential consistency can reorder the two operations.*

+ 0 pts Unattempted/Incorrect answer.

QUESTION 5

5 **3 / 3**

+ 0 pts (The compiler and architecture can each try to optimize sequential code.) Compiler reorders operations if the results are independent of each other. It may also simplify certain operations (e.g., by assigning variables to registers).
The execution pipelines can have different latencies. (e.g., cache-hit memory ops can complete before an earlier cache-miss one.) The architecture can issue multiple (independent) sequential instructions in the same clock. It can also issue later (dependent) instructions speculatively before the previous instruction has completed.

✓ **+ 1 pts** *Reason 1*

✓ **+ 1 pts** *Reason 2*

✓ **+ 1 pts** *Reason 3*

QUESTION 6

6 **2 / 3**

+ 1 pts Replacing static scheduling with block scheduling in the code.

✓ **+ 2 pts** *Explaining that block scheduling removes false sharing that was taking place in the static version.*

+ 0.5 pts [Partial marks] Instead of block scheduling, any other optimization is incorporated in the code, e.g., dynamic scheduling.

+ 1 pts [Partial marks] Explaining an optimization other than block scheduling.

+ 0 pts Incorrect/Not attempted

How will the "omp parallel for" change?

QUESTION 7

7 **0.5 / 4**

+ 1 pts Cache coherence is insufficient, Understand Processor consistency (it requires that writes from one thread appear to all other threads in that thread's order, which coherence does not guarantee pre se).

+ 2 pts Writes from each processor must complete in the order issued. Later writes must not return (even on cache hit) until all earlier writes have (even if they had cache miss).

+ 1 pts Disallow Compiler re-ordering or removal of writes.

+ 0 pts No answer

+ 0.5 *Point adjustment*

QUESTION 8

8 **4 / 4**

✓ **+ 4 pts** *Correct*

+ 2 pts Partially correct explanation

+ 0 pts Incorrect/Not attempted.

+ 0 pts # Explanation

Yes, two threads may access the critical section parallelly.
- FIFO consistency does not guarantee that writes to a given variable in two threads are seen in the same order by all threads.
- For two threads to enter the critical section, they must both have none set to true, for which neither flag must have been 1 (when checked at the if-statement in the loop).
- Now, both threads do set their flag to true before reaching the for-loop. However, it may happen due to FIFO consistency that a thread does not see the other thread’s write to flag until much later. Similarly the other thread may see this thread's true flag setting late (much after its own setting of flag). Thus, both see their own flag writes first, both find the other's flag false, and both skip the setting of none to false.
- Therefore, both threads may enter the critical section together.

QUESTION 9

9 **6 / 6**

✓ **+ 6 pts** *Correct*

+ 3 pts Draw Dependency Graph

+ 1 pts Critical Path length is correct

+ 1 pts Average concurrency is correct

+ 1 pts Maximum concurrency is correct

+ 0 pts Incorrect / Unattempted

QUESTION 10

10 **4 / 6**

✓ **+ 1.5 pts** *Parameters to consider (# links, #ports, Diameter, Bisection Width, Blockingness, Routing, Layout)*

✓ **- 0.5 pts** *Incomplete set of considerations (deducted from #1)*

✓ **+ 3 pts** *Justification based on values of parameters (At least 3 separate items, up to 1 mark each. Extra credit for more than 3 items.)*

- 1 pts Insufficient justification (deducted from #3)

- 2 pts Improper justification (deducted from #3)

✓ **+ 1.5 pts** *In comparison to others..*

- 0.5 pts Insufficient comparison (deducted from #6)

✓ **- 1 pts** *Comparison not useful (deducted from #1)*

+ 0 pts No/Incomprehensible answer

- 0.5 *Point adjustment*

QUESTION 11

11 **5 / 6**

✓ **+ 1 pts** *Indicate what is needed overall*

✓ **+ 1 pts** *How to signal arriving at barrier*

✓ **+ 1 pts** *Detect "everyone's" arrival*

✓ **+ 0.5 pts** *Proceed after detecting all arrival updates (syncing their views)*

✓ **+ 2.5 pts** *Arrival condition reset for next use*

✓ **- 2 pts** *Not synchronized reset (after all have checked it)*

+ 0 pts No acceptable answer

+ 1 *Point adjustment*

Right approach for re-entry. Need to carry it through.

2 Could this not disrupt the count < NUMT check of another thread?

Entry No: 2020CS10336 **Name:** CHINMAY MITTAL

Fill these first on all sheets

COL 380 MINOR EXAM

SEMESTER II 2022-2023

1 hour, 50 marks

Maximum marks are listed in [] for each question. Justify your answers: most marks are designated for correct justification. Use the provided space. Follow Course's Academic Integrity Code.

1. [11] Fill in the blanks. (Not all need actual numbers, you may use words to best describe the quantity.)

a) [2] At most 1. thread(s) may execute its/their anonymous critical section simultaneously.

Explain: Only one thread can execute a critical section at a time the others have to wait to acquire the lock once the thread finishes the critical section it releases the corresponding lock.

b) [2] At most N thread(s) may execute its/their named critical section (not necessarily the same name) simultaneously.

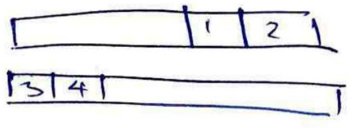
Explain: if the names are different all threads can execute simultaneously

c) [2] To be Processor Consistent, updates to variables *X* and *Y* may be observed to have occurred in mutually opposite orders by threads *i* and *j* respectively, only if $X \neq Y$ and updates don't occur in the same thread.

Explain: Processor Consistency requires all updates to a single variable to be sequentially consistent and also writes in the same thread have to follow program order.

d) [3] There can be at most 2 L1 cache misses in reading (e.g., moving to a register) an `int` variable in a C program. (Do not assume any compiler alignment.)

Explain: The integer might be split across multiple cache lines. if integers are 4 bytes and cache lines are 64 bytes then the int can be split across at most 2 cache lines and 2 misses might occur.



e) [2] There must be at least 0 L1 cache misses in reading an `int` variable in a C program. (You may assume any compiler optimization.)

Explain: if the compiler optimizes to fit the entire integer onto a single cache line. Then at most 1 cache miss might occur. The variable might be in L1 cache itself and 0 cache misses might occur.

2. [2] Explain what non-master threads print in the following code-fragment?

```
int a = 0;
#pragma omp parallel shared(a)
{
   #pragma  omp master
   { // Only master thread executes this
        a++;
      #pragma omp flush
    }
   while(!a); a++;
   printf("%d\n", a);
}
```

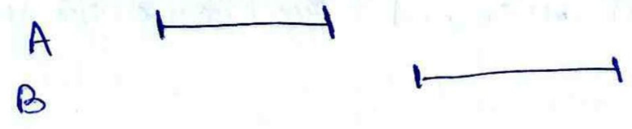
It might happen that non master thread does not print anything (because for it a always remains 0) (no flush in non master threads)

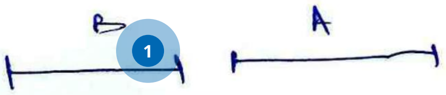
Otherwise a becomes 1 for all threads. and they write to it concurrently (multiple writes may occur) So the result produced can be anything 2, 3 . . . - - -

no other thread updates before us → 2; another thread updates before us → 3 etc.

3. [2] Explain one example of a Sequential Consistent history that is not Linearizable.

eg

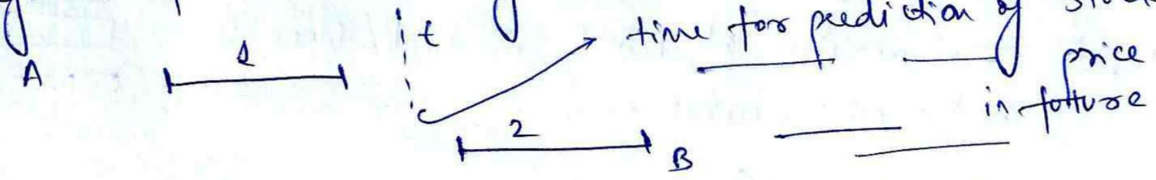




A sequentially consistent history is but this history is not linearizable time stamp of A is less than time stamp of B.

4. [3] Give one example of an application where Sequential Consistency of operations is insufficient and Linearizability is desirable.

Consider a multithreaded program in the stock market, where one thread sends out order to the exchange and other reads acknowledgments from the exchange.



Sending the orders must happen before the prediction time and acknowleding the respone must happen after the prediction time. Hence linearizable is required. Sequential consistency might have orderd 2 before 1 which is not desirable.

Entry No: 2020CS10336 Name: CHINMAY MITTAL

5. [3] List three separate reasons why two consecutive instructions in a sequential execution context may complete out of order.

1) Compiler May decide to do generate the assembly out of order if it beleives that such reordering does not affect correctness of the program.

2) The Hardware may schedule these instructions parallely / out of order on seperate cores. if it beleives correctness is not dependent on order.

3) The ~~programming platform might cause the~~ first operation might be slower eg memory read and might take several cycles, whereas the second operation might complete before the first one. (eg in pipelined architectures

6. [3] How would you improve the running time of the following OpenMP code-fragment? (Be sure to explain what you are improving and why your change causes the improvement.)

```
struct {
   float x, y, d;
} points[N];
#pragma omp parallel for schedule(static, 1) shared(x0, y0)
   for(int i=0; i<N; i++) {
      points[i].d = pow(points[i].x-x0),2) + pow(points[i].y-y0),2)
   }
```

There might be false sharing if the threads process interleaved chunks in the points array. (Because of writes to points[i].d)

Hence we can ensure that threads processes continuous chunks to avoid false sharing

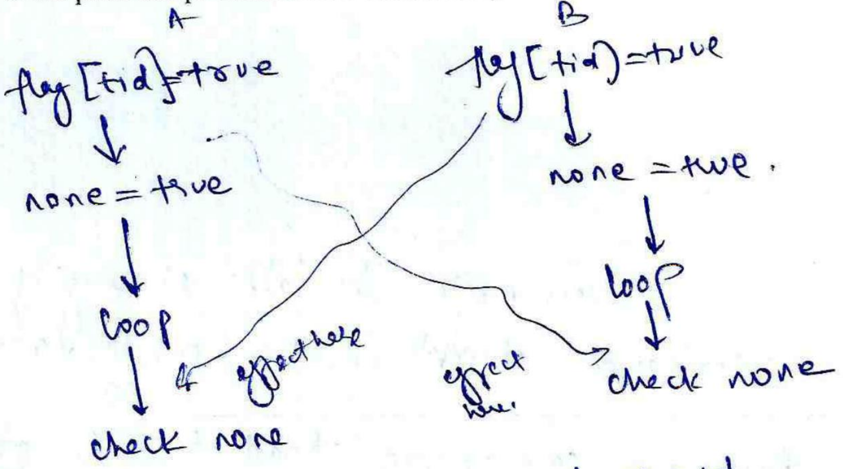
```
tid = omp_get_thread_id()
left = tid * chunk_size;
right = (tid+1)* chunk_size;
```

7. [4] Is cache coherence sufficient to guarantee Processor Consistency on X86-like architecture? If it is, prove. If it is not, indicate what platform support may be needed to guarantee Processor Consistency.

Yes cache coherence is sufficient to guarantee processor consistency.
Consider any variable. All processor caches have the same view of the variable. Any write to this variable will appear to all thread's caches together. (No two threads can have a different view of the same variable. Hence the order in which the caches get updated by writes from different threads is the consistent global order of write(x) as seen by all threads. and hence processor consistency is guaranteed.

8. [4] In executing the following code-fragment, is it possible for two (or more) threads to execute `criticalSection` in parallel (i.e., simultaneously) if the platform provides FIFO consistency? Explain.

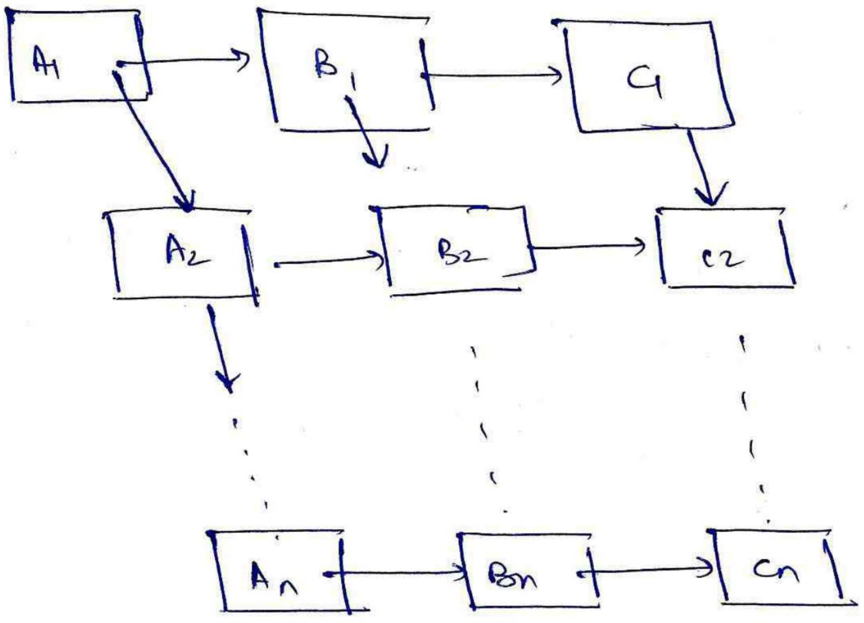
```
// Initially flag is universally false
#pragma omp parallel shared(flag)
{
  int tid = omp_get_thread_num();
  int count = omp_get_num_threads();
  flag[tid] = true;
  bool none = true;
  for(int i=0; i<count; i++)
    if(i != tid && flag[i]){
        none = false; break;
    }
  if(none)
    criticalSection();
  flag[tid] = false;
}
```



in FIFO consistency only the writes in one thread should appear to have effect in order to the other thread. (Reads need to be ordered. Both of these threads have a consistent view where the write to flag of the other thread appear to have effect after the loop (and thus their own none remains true). Both of them can thus enter the critical section.

Entry No: 2020CS10336 **Name:** CHINMAY MITTAL

9. [6] Depict a dependency task graph that demonstrates a 3-stage pipeline, where each stage processes a sequence of data items in a stream of n items. What is the critical path length of this graph, the average concurrency, and maximum concurrency (Recall that in a fully expanded task graph, a task begins only after its predecessor tasks complete.)



Maximum concurrency $\longrightarrow 3$ (where all 3 stages are running simultaneously (eg A_3, B_2, C_1)

two A_i, A_j or B_i, B_j or C_i, C_j cannot run simultaneously

critical path length

$\longrightarrow (n-1) + 2 = n+1$

$A_1 \longrightarrow A_2 \longrightarrow A_3 \cdots A_n \longrightarrow B_n \longrightarrow C_n$

Average concurrency $= \dfrac{\text{\# of tasks}}{\text{critical path length}}$

$= \dfrac{3 * n}{n+1}$

10. [6] You have to design a network comprising 128 nodes. Choose a network topology for it, and explain your choice.

A hyper cube network is a good choice for such a network here we have 2^7 nodes, Hence we can easily create a 7-dimensional hyper cube. Each node will have 7 connections which is not much and can be supported easily, this provides a good Bisection bandwidth of 64, allowing a lot of concurrent data transfers. The number of connections is also of similar size to the number of nodes $(64 * 7)$ and hence this network is cost effective also.

Entry No: 2020CS10336 **Name:** CHINMAY MITTAL

false initially.

11. [6] Implement the OpenMP pragma *barrier* (meaning your code will run in place of barrier). Rough code is OK. You may use Locks (or any other in-built synchronization mechanism). Note that barriers can be used multiple times in a parallel region.

```
bool reached_barrier [NUMT] ;     bool all_inside = false;
                                  int count_reached = 0;
func barrier ( threadfid )        omp_lock* lock;
{                                 omp_lock_init (lock);
    ~~if (not reached_barrier [fid]) {~~

        ~~while (not all_inside~~

    ~~false {~~   if (not reached_barrier [fid]) {  // waiting to enter barrier

                    reached_barrier [fid] = true;

                    acquire lock
                       count_reached ++
                    release lock
                      # pragma omp ~~flush~~ (count_reached)
                    while ( count_reached < NUMT ) {
                          # pragma omp flush (count_reached)
                    }
                    return ;
                } else {  ?
                  // waiting to exit barrier

                    reached_barrier [fid] = false;
                    acquire lock
                       count_reached --
                    release lock
                    # pragma omp flush (count_reached)
                    while ( count_reached > 0 ) {
                          # pragma omp flush (count_reached)
                    }
                    return ; }.
}
```

2