CS6868 Final

Dept of CSE, IIT Madras

Total marks = 40, Time = 180 min

04 May 2018

Read the instructions and questions carefully. You may make any reasonably assumptions that you think are necessary; but state them clearly. You will get an answer sheet with 12 pages. Leave the first page empty. Start each question on a new page. **For any question (including all its sub-parts), the answer should NOT cross two pages.** The spill over text will be strictly ignored. If you scratch/cross some part of the answer, you may use space from the next page.

1. [8] **OpenMP**: (a) What are the differences between the `single`, `master`, and `critical` sections [1.5].

   (b) What are the uses of and differences between `private`, `firstprivate`, and `lastprivate` clauses? [1.5]. Show an example code to use the `lastprivate` clause in a meaningful manner? [1]

   (c) A student wrote the code shown on the right, when asked to write an **efficient parallel** code to perform the following integration:

   $$\int_0^1 \frac{4}{(1+x^2)} dx$$

   Does the code compute the expected value? If yes, prove it. If No, state why and suggest a fix to do the computation correctly+efficiently in parallel. [3]

```
static long num_steps = 100000;
double step;
double computePi (){
   int i;   double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;
#pragma omp parallel for
   for (i=0;i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   pi = sum;
   return pi; } /* end computePi */
```

   (d) Consider the code in the file `test.c`, shown on the right. How many threads are created during the execution of `./test`, as shown below [1].

```
$ export OMP_NUM_THREADS=4
$ gcc -fopenmp test.c -o test
$ ./test
```

```
$ cat test.c
void main(){
omp_set_num_threads(6);
#pragma omp parallel for num_threads(8)
   for (int i=0;i<2;++i){
      S; // some code not shown.
   }
}
```

2. [8] **Mutual Exclusion**: a) Briefly explain the three properties (mutual exclusion, deadlock freedom, and starvation freedom) expected from an implementation of locks [3].

   b) Among the above three properties which ones are satisfied and which ones are not satisfied by the B2Lock algorithm, shown on the right. Assume strict consistency. [2]

```
class B2Lock {
   boolean[] flag;
  public B2Lock (int n) {
    flag  = new boolean[n];
    for (int i=0; i<n; i++) {
       flag[i] = false;
    }
  } /* end constructor */
public void lock() {
  int i = my_id(); // computes thread id.
  flag[i]  = true;
  while (for some k != i flag[k] is true
          && i > k);
 } /* end lock */
public void unlock(){
  int i = my_id(); // computes thread id.
  flag[i] = false; } } /* end class */
```

   c) Among the above three properties which ones are satisfied and which ones are not satisfied by the Lock2 algorithm, shown on the right. Assume: only two threads with ids 0 and 1 and strict consistency. [2]

```
public class Lock2{
 private int v;
 public void lock() {
  int i = my_id(); // computes thread id.
  v = i;
  while (v == i) {};
 }
 public void unlock() {}
}
```

   d) We define an interval as the time between two events of a thread. Which of the choices on the right is/are true on the precedence relation between intervals of different threads? [1] We use A, B, C to denote intervals and $\rightarrow$ to denote the precedence relation.

   i. $A \rightarrow A$

   ii. $(A \rightarrow B) \Rightarrow (B \rightarrow A)$

   iii. $A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C$

   iv. It is possible that both $A \rightarrow B$ and $B \rightarrow A$ are false.

3. [8] **Concurrency I**

(a) A conman had Rs 10 in his bank. But he needed Rs 15 to gamble. He studied the code of the banking software (shown on the right) and used two different computers and clicked on withdraw button on the two computers (by specifying ten rupees in one and five rupees in the other, as the amount of withdrawal) at the same time. He claimed that two events happened: (1) he succeeded in withdrawing the complete Rs 15 and (2) his balance at the end was Rs 5. State if event 1 is possible [1.5]. State if event 2 is possible [1.5]. Briefly justify your answers.

```
void deposit(int amt) {
   lock(m); //m: a shared lock
   balance = balance+amt;
   unlock(m); } // end deposit
int read_balance() {
   int t;
   lock(m);
   t = balance;
   unlock(m);
   return t; } //end read_balance
```

```
int withdraw(int amt){
   int t = read_balance();
   lock(m);
   if (t <= amt) {
      balance = 0;
   } else {
      balance = balance-amt;
      t = amt;
   }
   unlock(m);
   return t; } //end withdraw
```

(b) Briefly contrast lock-free and wait free data-structures? [1].

(c) State if Linearizability and Sequential consistency lead to composable histories? [0.5+0.5].

(d) Consider an implementation of stack, that supports `push` and `pop` operations. Say, we have two stacks `p` and `q` and a sequence of operations shown on the right. We use the notation `<op>` to denote the interval of the operation `op`. Is the sequence sequentially consistent? [1.5] Linearizable? [1.5]. Briefly justify.

```
<p.push(y)>  <q.push(y)> <p.pop(y)>
      <q.push(x)> <p.push(x)> <q.pop(x)>


------------- time ------------------------>
```

---

4. [8] **Concurrency II** (a) Which of the following is true with respect to a consensus protocol? [1]

   i. all threads decide on the same value.
   ii. the value chosen by a thread is the input of some thread.
   iii. threads agree on the chunks of iterations to be executed.
   iv. threads identify the thread ids of other threads.

(b) State true or false [8 × 0.5]. Note: not all the answers are same.

   i. The consensus numbers of compare-and-set and fetch-and-increment match.
   ii. The consensus numbers of atomic read/write registers and atomic memory cells match.
   iii. The consensus numbers of two-assignment objects matches that of FIFO queue.
   iv. Sequential consistency is commonly supported in the hardware.
   v. Compare-and-set along with read/write registers can be used implement a FIFO queue.
   vi. In Java, if all the shared fields are volatile then they need not be accessed (read/written) in synchronized blocks.
   vii. In Java, accesses (reads/writes) to shared volatile fields must always be done inside synchronized blocks.
   viii. The OpenMP `flush` pragma helps speed up memory writes.

(c) Consider the implementation of CASReg shown below:

```
public class CASReg {
  private long value;
  public boolean cas(long ex, long new) {
    if (this.value==ex) {
      this.value = new;
      return true;
    }
    return false; } /* end cas */ } // class CASReg
```

This code is used along with the code shown in the right to realize compare-and-swap operations. Overall, there are some issues in the code. Fix them [3].

```
class myThread extends Thread {
  CASReg r = new CASReg();
  public void run(){
    long tid = Thread.currentThread().getId();
    if (r.cas(0, tid)){
      // critical section only one thread
      // should enter.
      r.cas(tid,0);
    }
  } /* end run */ } // class myThread
...
new myThread().start(); //  creates
new myThread().start(); // two threads.
```

---

5. [8] **MPI**: Mark true or false [4 × 0.5]. Note: not all the answers are same.

   i. `MPI_Win_lock` is used to lock a shared location in a window.
   ii. In a `MPI_Put` call, the target process can use `MPI_Post` to expose the window.
   iii. In a `MPI_Put` call, the target process can use `MPI_Wait` to unexpose the window.
   iv. `MPI_Fence` can be used to implement `MPI_Post` and `MPI_Wait`.

(b) Consider the snippet of the `MPI` code shown on the right. List the issues with the code and the fixes thereof [3].

```
MPI_Win_start(togroup, 0, win);
MPI_Win_post(fromgroup, 0, win);
MPI_Put(&from,n,MPI_Int,neighborRank,n,MPI_Int, win);
MPI_Win_wait(win);
MPI_Win_complete(win);
```

(c) How can you use `MPI_Get_Accumulate` and `MPI_Accumulate` to implement `MPI_Get` and `MPI_Put`, respectively? [2] Its okay, if you don't remember the complete signature of the accumulate functions.

(d) Mark the statements true/false [1]. Note: not all the answers are same.

   i. Task parallelism is suitable for MPI and OpenMP.
   ii. Loop parallelism is suitable for OpenMP and Java.
   iii. Compared to Java, fork-join pattern fits better with MPI.
   iv. Master/worker pattern fits well with divide-conquer type of algorithms.