CS6560: Parallel Computer Architecture-Assignment#5

Abhishek Yadav, CS12B032

April 12, 2016

Q1

Sufficient conditions for sequential consistency are as described in the reference book:

- All core insert their **loads** and **stores** in the memory order, i.e. <m order, respecting their program order(<p), regardless of whether they are to the same or different addresses which is what is said in part **1.a** of the problem statement.
- Every load gets its value from the last store before it (in global memory order) to the same address. This condition applied inductively for all processes, conforms to the statement in **1.c**
- After a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation which conforms with the condition in **1.b**

Since, these conditions conform to the conditions sufficient for SC hence using these conditions we can get guaranteed SC.

Q2

First thing to note here is that barrier function call in each process will have a local copy of the barrier variable(**B**) unless passed by reference and the effect of fetch-and-add() on B will not be observed in other processes hence should be declared global/shared since not specified explicitly. So modified code will be:

```
globalVar B=0;
BARRIER(N:integer){
    if(fetch-and-add(B,1)==N-1)
        B=0;
    else
        while(B!=0) do {};
}
```

Q3

(a) order that will violate write atomicity:
P1 writes to A
P2 reads from A and writes to u
P4 writes to A which will, through magic writing capability, essentially write to the cached copy in P2's private cache.
Next, if **P2** or **P3** will try to read the value of A into v or w respectively, they will get the stale copy of A from the homenode which violates the write atomicity.

(b) One of the sequential consistency conditions is: "Every load gets its value from the last store before it(in global memory order) to the same address." which is violated when P2 or P3 read stale value of A(=1) in v or w although the address was written by P4 before these loads.

(c) **Coherence definition**: For any given memory location, at any moment in time, there is either a single core that may write it(and that may also read it) or some number of cores that may read it., i.e. there is never a time when a given memory location may be written by a core and simultaneously either read or written by other cores. Using the above definition we can see that coherence is violated when P1 and P4 try to write to the same location simultaneously. As discussed in class, we can employ several techniques to solve this problem such as using locks, or tokens or queues to store the requests in the order they arrive and so on..But the tokens method is most efficient so we can use it where each requesting processor gets a token number and snoops on the shared bus for the same number and a processor after completing execution increments its token number and transmits it on the bus and so on.

(d) In invalidation-based protocol, write-atomicity doesn't get violated because the requesting processor sends an invalidation signal on the shared bus and all other cores having the cached copy, including the home node, of the location to be written, invalidate themselves. Now, a core requesting to write at location A will send a request to A's home-node but since home node doesn't have the valid copy, it will send the owner core's id which wrote to the location most recently and hence write-atomicity is preserved. If cores will insert their request in the memory order as in update-based protocol, there will be no SC violation. But coherence violation may take place in this case as well and can be solved the same way.

(e) No, can't construct the above mentioned situations for update protocols on a bus because then home-node will be able to update it's local copy of A whenever an update to the location A will be snooped on the bus and hence always have the most recently written value to any location, here A, and others, when trying to read from A, will get the latest written value.