COL 380
Homework 2 Solutions

Author : Aman Bhatia
Dated : 25 March 2016

Sequential Consistency, Linearizability

Ans(i) For both (a) and (b), histories are linearizable and sequentially consistent. Consider the following histories,

```
<r.wr(1), B>                  <r.wr(2), C>
<r.rd(1), A>                  <r.wr(1), B>
<r.wr(2), C>                  <r.rd(1), A>
<r.rd(2), B>                  <r.rd(1), B>

Part(a)                       Part(b)
```

Ans(ii)

- In Linearizability, we have constraint that method call appear to take place instantaneously between method invocation and response events. It allows only the reordering of overlapping method calls. While in SC, we only need to preserve the program order, i.e., we have no constraint over the real time ordering of method calls. Hence, we can conclude that ***W(SC, Linearizability)*** holds true.

- Strict Consistency also impose the condition on the possible ordering among the overlapping method calls. Strict consistency uses the concept of absolute time where as linearizability uses the concept of global time. Hence, we can conclude that ***W(Linearizability, Strict Consistency)*** holds true.

- From above two, using the transitivity of relation W, ***W(SC, Strict Consistency)*** also holds true.

Ans(iii) In the Lamport Model, we maintain a single clock, which can not identify concurrent events. Where as in vector clock, for each processor in the distributed system, we have a separate time stamp.

At each event, associated vector clocks for SC history of part(a) is as follows,

```
<r.wr(1), B> , (0,1,0)
<r.rd(1), A> , (1,1,0)
<r.wr(2), C> , (1,1,1)
<r.rd(2), B> , (1,2,1)
```

Problem 2: OpenMP

(Ques) Consider the loop:

```
a[0] = 0;
for ( i = 1; i < n ; i++)
     a[i] = a[i - 1] + i ;
```

Is the loop parallelizable (with or with loop transformation)? If so, then write snippet of OpenMP code for the parallel version of this loop.

(Ans) The given loop is not parallelizable because of loop-carried dependencies, i.e. the i^{th} element of the loop depends on the value of the $i-1^{th}$ element of the loop. However, the formula for a[i] is easily derivable,

```
a[i] - a[i - 1] = i
a[i-1] - a[i-2] = i-1
.
.
.
a[1] - a[0]     = 1
```

On adding, ________________________________

$$a[i] - a[0] = \Sigma(i) = \frac{i(i+1)}{2}$$

Hence, we get the following loop,

```
a[0] = 0;
#pragma omp parallel for
for ( i = 1; i < n ; i++)
     a[i] = i*(i+1)/2 ;
```

Problem 3: OpenMP Debug

(Ques) Consider the following program. Identify correctness issues in the program and specify the solutions (SHOW ONLY RELEVANT CHANGES): the for loop must be parallelized on 2 threads collectively calling foo() 10 times and then each thread prints "Hello World" in an ordered fashion. (6 marks)

```
#pragma omp parallel
{
     omp_set_num_threads(2);
     #pragma omp parallel for
     for ( int i = 0; i < 10; i++)
     {
          foo();
     }
     printf("hello world from %d \n" , omp_get_thread_num()) ;
}
```

(Ans) There are corretness issues with the code because "omp parallel for" is called within "omp parallel", which will create more threads then needed. Also as we want the "hello world" to be ordered, so we will print it only when thread id is according to the order. Hence, the correct code snippet is as follows,

```
int i;
omp_set_num_threads(2);
#pragma omp parallel for
for ( i = 0; i < 10; i++)
{
     foo();
}
// implicit barrier here

int thread_id = 0;
#pragma omp parallel
{
     while(thread_id != omp_get_thread_num());
     printf("hello world from %d \n" , omp_get_thread_num());
     thread_id++;
}
```

Problem 4: OpenMP Performance

(Ques) Find all performance issues with the program listed below and provide an alternate implementation addressing all the issues: (4 marks)

```
#pragma omp parallel for
for (i=0; i<N ; ++i){
    #pragma omp critical
    {
        if (arr[i] > max) max = arr[i];
    }
}
```

(Ans) In the given loop, the loop iteration is in the critical section. Hence, we will not be able to get any performance advantage. The efficient implementation will be to use reduction with the parallel for. Hence, the correct and efficient implementation is as follows,

```
int i, max=0;
#pragma omp parallel for reduction(max:max)
for (i=0; i<N ; ++i){
    if (arr[i] > max) max = arr[i];
}
```

--------------------END OF FILE--------------------