CS315A Midterm Solutions

Open Book, Open Notes, Calculator okay — NO computer.
(Total time = 120 minutes)

Name (*please print*):____Solutions____

I agree to abide by the terms of the Honor Code while taking this exam.

Signature:____

Points: 1. (30)___________

2. (20)___________

3. (20)___________

4. (30)___________

Total: (100)___________

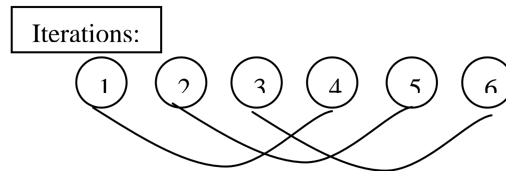
Make sure you state all your assumptions.

SCPD Students: Please attach a routing slip to the midterm.

Question 1. Short Answer (30 pts)

1a. (6 pts)
Give an example of a simple parallel for loop that would execute faster with interleaved static scheduling than by giving each task a single contiguous chunk of iterations. Your example should not have nested loops.

Use a loop where iteration i+p is predicated on iteration i, where p is the number of processors:



Imagine the lines between iterations are dependencies.
If p=3, for the diagram above, and blocking took chunks of three, the algorithm would be serialized (each processor waiting for the others to complete). But, perfect load balancing can be achieved if interleaving is used.

3 points for doing a variable amount of work for each iteration: you still may have a lot of load balancing problems without dynamic scheduling (notice the problem says "static").

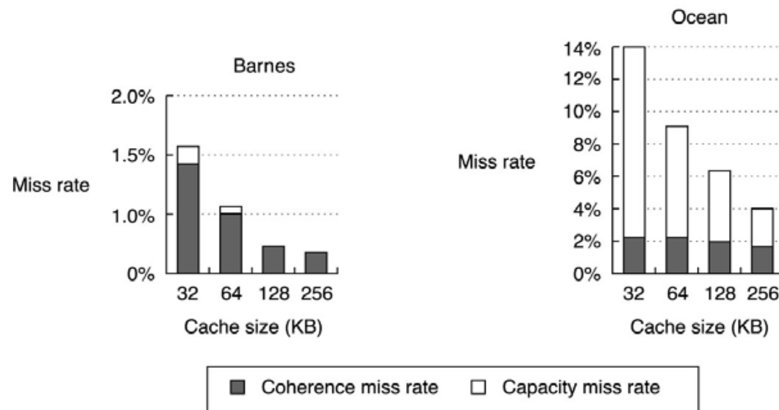
1b. (6 pts)
Consider a parallel make program which compiles and links single library---it computes the dependency graph and performs the compilation of each needed object file on a different processor, finally linking in one sequential step. What kind of data partitioning is this? (output, input, output & input, etc.)

Input. There is only one output (the library) and many inputs, each of which needs to be processed individually.

1c. (6 pts)
How would a shared memory parallel system optimized for TPC-C differ from one optimized for SPLASH-2 applications?

TPC: Little ILP so simple pipelines, FP not important, large L1 caches, Large L2 caches to hold shared meta data, optimize latency of dirty misses.

SPLASH-2: Lots of ILP and FP so complex pipeline, small L1 caches to capture important working sets, L2 size not that important, small amount of sharing so communication misses not that important.



1d. (6 pts)
Why does the coherence miss rate of Barnes, unlike Ocean, decrease with increasing cache size? (Hint: coherence misses are measured as transitions into the M state with the MSI protocol described in class)

The reduction in coherence miss rate happens because these are really capacity misses to non-shared data that masquerade as coherence misses because the MSI protocol counts S -> M transitions as a coherency miss even though the data is not shared. The real coherency miss rate shows up with cache size of 128-256 KB. Ocean does not do much writing of local data so does not exhibit this effect.

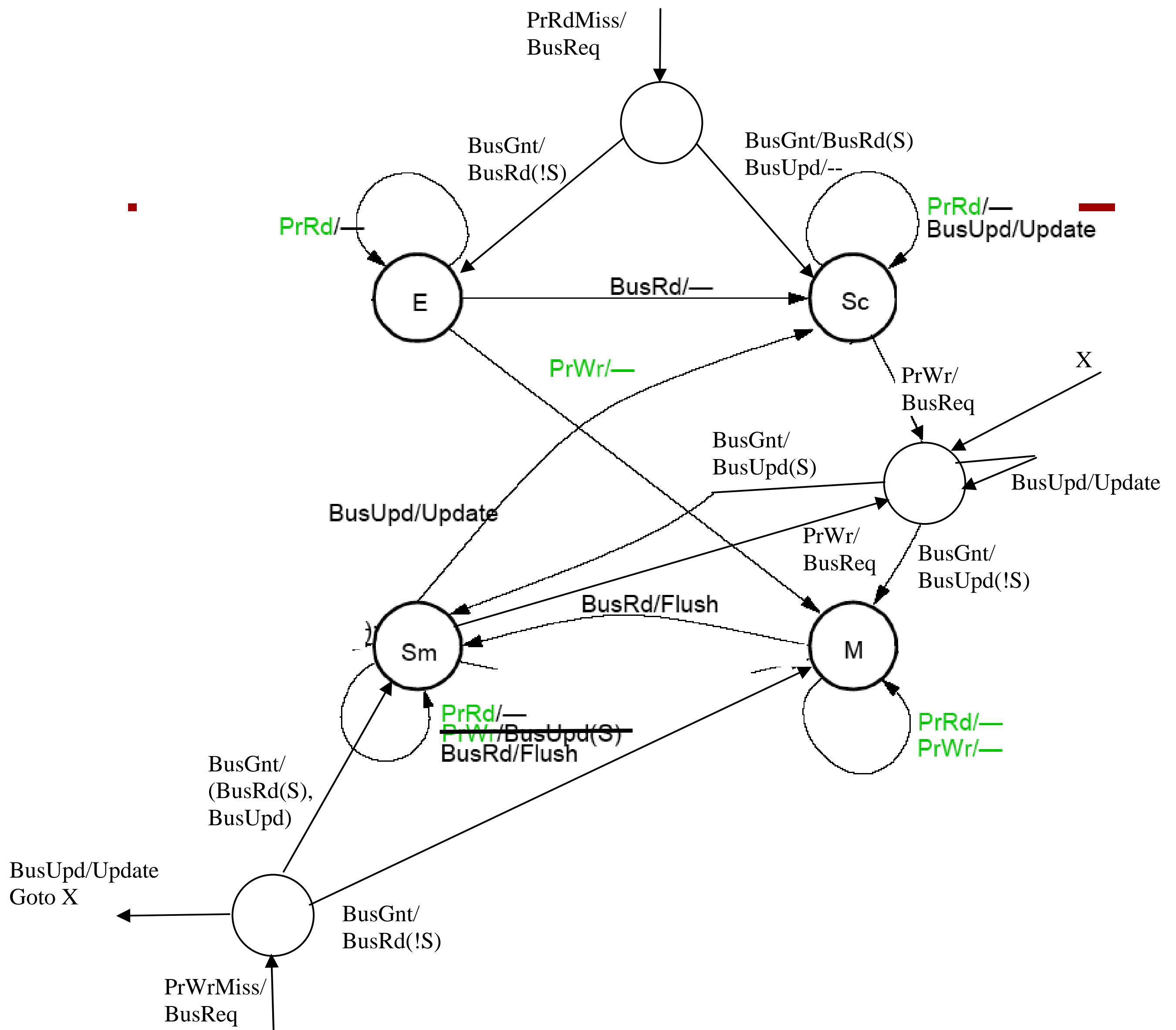
Be careful how you justify your answers; some people said erroneous things.

1e. (6 pts)
The Sun Enterprise 5000 does not use an upgrade transaction. However, it does avoid transferring data on a BusRdX transaction if it turns out that the issuing cache is the only one that has the data. The idea is a simple extension of the way that memory is waved off on a BusRd when one of the caches has the block in M. Using our three wired-OR signals (shared, dirty, inhibit) for reporting status of relevant blocks, explain how to get the desired behavior? (Hint: the issuing cache snoops its own tags during the transaction.)

A BusRdX would be issued by the processor on a write. The line could be in I or S state. If the line is on I state it would get the data form memory or another cache. If the line in S state it would assert the processor would assert inhibit and look to see if the shared line is raised by any other processor. If not it would raise the dirty line to hold of the memory and lower the inhibit line.

Question 2: Cache Coherence Protocols (20 pts)

Add the transient states to the Dragon transition diagram so that the protocol works even though requesting the bus and acquiring the bus is not an atomic operation.

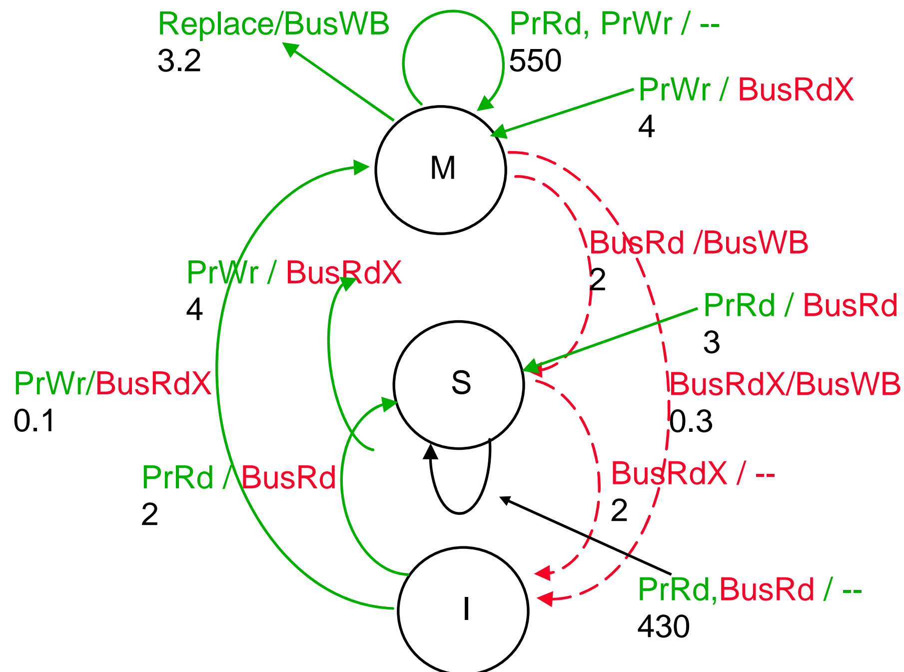


Question 3: Cache Coherence Analysis (20 pts)

3.a (10 pts)

Compute the traffic per CPU using the transition frequencies shown in the figure below. All frequencies are the number of transitions per 1000 data references. Assume the following:

- 1 GIPS processor
- 30% of instructions are loads and stores
- 6 bytes for upgrade/invalidate
- 70 bytes for a cache block read or write



There are no upgrades or invalidates in MSI: all bus transactions read/write an entire line.

Transition	*Number of transitions*	*Cycles*
I->S	*2*	*70*
I->M	*0.1*	*70*
S->S	*430*	*0*
S->M	*4*	*70*
S->I	*2*	*0*
M->M	*550*	*0*
M->S	*2*	*70*
M->I	*0.3*	*70*
Replace/BusWB	*3.2*	*70*
PrWr/BusRdX	*4*	*70*
PrRd/BusRd	*3*	*70*

*Bytes per reference: 70 * (3.2 + 4 + .1 + 2 + 4 + 3 + .3 + 2)/1000 = 1.302*
*References per second: .3 * 1e9*
*Bytes per second: .3 * 1e9 * 1.302 = 372.5 MB/s (390.6 x 1e6 B/s)*

3b. (5 pts)
If the S → M transition which causes a bus read exclusive is replaced with a bus invalidate, what is the bus traffic per processor?

Transition	*Number of transitions*	*Cycles*
I->S	*2*	*70*
I->M	*0.1*	*70*
S->S	*430*	*0*
S->M	*4*	*6*
S->I	*2*	*0*
M->M	*550*	*0*
M->S	*2*	*70*
M->I	*0.3*	*70*
Replace/BusWB	*3.2*	*70*
PrWr/BusRdX	*4*	*70*
PrRd/BusRd	*3*	*70*

*Bytes per reference: 70 * (3.2 + 4 + .1 + 2 + 3 + .3 + 2)/1000 + 6 * 4/1000 = 1.046*
*References per second: .3 * 1e9*
*Bytes per second: .3 * 1e9 * 1.046 = 299.3 MB/s (313.8 x 1e6 B/s)*

3c. (5 pts)
If the MSI protocol of Question 3b is replaced with a MESI protocol and the transition frequency between the E and M states is 2 per 1000 data references what is the bus traffic per processor?

S->M, in a MSI protocol, contains all references S->m and E->M in a MESI protocol. So, since E->M = 2 in MESI, S->M must equal 4-2 = 2 (the old S->M value minus the E->m value).

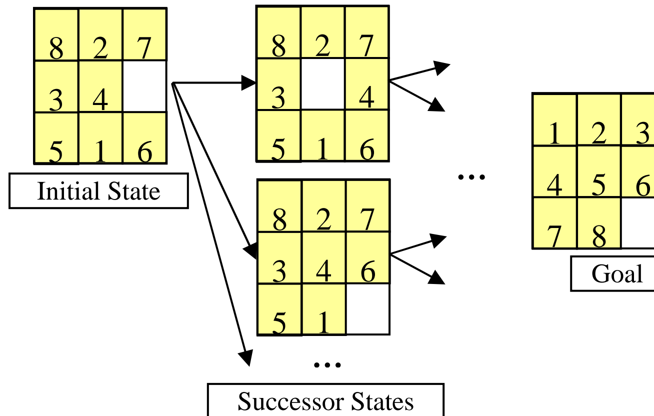
Transition	*Number of transitions*	*Cycles*
I->S	*2*	*70*
I->M	*0.1*	*70*
S->S	*430*	*0*
S->M	*2*	*6*
S->I	*2*	*0*
E->M	*2*	*0*
I->E	*??*	*70*
E->S	*??*	*0*
M->M	*550*	*0*
M->S	*2*	*70*
M->I	*0.3*	*70*
Replace/BusWB	*3.2*	*70*
PrWr/BusRdX	*4*	*70*
PrRd/BusRd	*3*	*70*

The number of transitions from I->E is a subset of those transitions from I->S. We can't find the exact number, but it doesn't matter since I->E and I->S take the same number of bus cycles.

*Bytes per reference: 70 * (3.2 + 4 + .1 + 2 + 3 + .3 + 2)/1000 + 6 * 2/1000 = 1.034*
*References per second: .3 * 1e9*
*Bytes per second: .3 * 1e9 * 1.034 = 295.8 MB/s (310 x 1e6 B/s)*

Question 4: Parallel Programming (30 pts)

An incarnation of the 8-puzzle problem is displayed below: there are eight numbered tiles and the solution is a set of steps transforming the initial state to the goal state. A step consists of sliding one of the tiles into the empty slot. One simple way to find the solution with the fewest steps is by performing a breadth-first search on the tree of successors from the initial state.



4a. (25 pts)

Write a parallel algorithm for *p* processors that finds the first goal state in the successor tree. Compute the speedup for your algorithm using the timing information below.

Assume the solution is at depth *d* and each state has an average of *s* successors. Do not worry about repeated states. Assume any reasonable shared-memory parallel programming model. In other words, you don't have to show accurate calls to pthreads/OpenMP; just use pseudo code so we know when threads begin/end and any synchronization (locks, barriers, etc.). For timing analysis, assume perfect caches. Use the following functions with corresponding timing information.

Function	**Time**
`BARRIER()`	t_B*
`LOCK(Lock l)/UNLOCK(Lock l)`	t_L
`void enqueue(Queue q, State/States s)`	t_E**
`State dequeue(Queue q)`	t_D**
`States successors(State s)` (Returns all successors of s)	t_S
Comparing two states (use ==)	t_C

* Time that the last arriving processor is delayed.
** Queue access is not atomic; may require locks.

Make sure that when a thread finds the solution, it terminates the search across the whole system.

This code ended up being a little harder than I originally thought because I didn't think through the "find first solution" completely. To correctly answer the problem, you would need to prevent one processor from getting ahead of the others. Here is a correct solution, notice the extension of enqueue/dequeue to carry an additional int. You could do it in other ways, as people demonstrated. Basically, all processors must reach a barrier once they dequeue a state which is in the next level of the tree---this ensures that each level gets completely searched before moving on to the next level. There is some additional code you would need to deal with the first few iterations, since one barrier would not be enough. 15pts for code, 10pts for analysis.

```
void findSolution(State init, State goal) {
     done = false
     enqueue(q, successors(init), 0);

  parallel {
     int lastLevel = 0;
     int thisLevel;
     while(!done) {
          lock(ql)
          (next, thisLevel) = dequeue(q);
          enqueue(q, successors(next), thisLevel+1);
          unlock(ql)
          if (thisLevel != lastLevel) {
               BARRIER(); // wait until everyone
                          // reaches thisLevel.
               lastLevel = thisLevel;
          }
          if(next == goal) {
               done = true;
               break;
          }
     }
  }
}
```

Analysis:
There are $O(s^d)$ nodes which must be examined before the solution will be found.
Non parallelizable part: $t_s + t_e$
Sequential runtime: $t_s + t_e + O(s^d)*(t_d + t_e + t_s + t_c)$
That's just the non-parallelizable part plus, for each iteration of a simple loop, we dequeue the next state, enqueue its successors, and compare it with the goal.
Parallel runtime: $t_s + t_e + O(s^d)/p*(2*t_l + t_d + t_e + t_s + t_c) + d*t_b$
So, for each parallel iteration, there is a lock/unlock pair, dequeue, enqueue of successors, and compare. Then there's one barrier per level in the tree until you find the solution.

So, the speedup is just the sequential runtime over the parallel runtime.

4.b (5 pts) Short answer
In the 8-puzzle program, what would constitute blocking/chunking? (Hint: think about how you choose what work to do next).

Blocking would be dequeing more than one state per iteration, thus amortizing the cost of dequeing/enqueing successors.