Full Name: _____

Andrew Id: _____

15-418/618, Spring 2019

Exam 2 SOLUTIONS

April 15, 2019

Instructions:

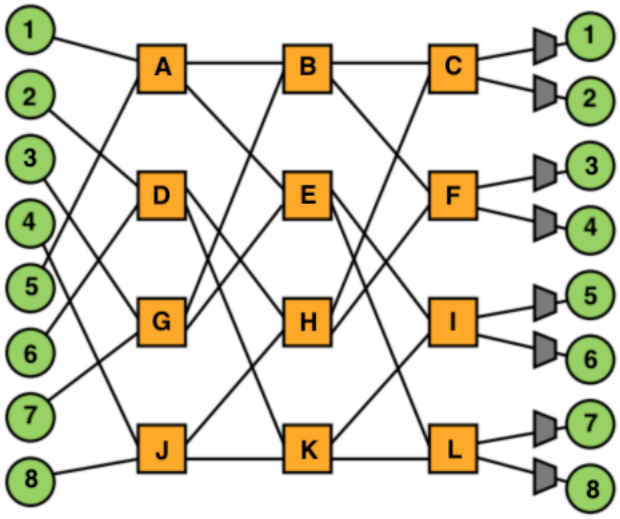
- Write your answers in the space provided for the problem. If your work gets messy, please clearly indicate your final answer.
- The exam has a maximum score of **60** points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is CLOSED BOOK, CLOSED NOTES (with the exception of your one sheet of notes.)

Problem	Your Score	Possible Points
1		9
2		9
3		8
4		10
5		12
6		12
Total		60

Interconnection Networks

Problem 1. (9 points):

You are building a **packet switched logarithmic network** for an eight-core processor. The logarithmic network is pictured below with the processors numbered 1 to 8 and the switches labeled A to L. Assume that a packet is 64 bytes.



A. (3 pts) Is this network blocking? If it is, list two source-destination pairs that would block each other.

Solution: It is blocking. Consider simultaneous messages from 1 - 7 and 4 - 8

B. (3 pts) With store and forward routing, what is the minimum latency for sending a single packet from one processor to another? Assume a link can transmit 4 bytes per cycle.

Solution: With store-and-forward flow control, a packet must be completely transmitted to the next switch on its route before that switch can begin transmitting it. Since it takes 16 cycles to transfer a packet and the each path is 4 hops, the minimum latency is 64 cycles.

C. (3 pts) If the network is designed to use cut-through routing, what it is the minimum latency for sending a single packet from one processor to another? Assume a link can transmit 4 bytes per cycle.

Solution: 18 cycles total. It takes 4 cycles for the header to reach its destination, and from there the next 4 bytes are right behind it. It takes 15 additional cycles for the rest of the packet to arrive. Bringing us to a total of 19 cycles.

Heterogeneous Parallelism

Problem 2. (9 points):

You are part of a team that is designing new family of single-chip parallel processors. Your colleagues have already designed the following two CPUs, which will be the building blocks of your system design:

CPU-Lean: this CPU was designed for area efficiency;

CPU-Fast: this CPU was designed for speed. It is *twice as fast* as the **CPU-Lean** design, but it also takes up four times as much area.

Your team is considering several different machine designs, having an overall area equivalent to that of N **CPU-Lean** cores. That is, it will have P_L lean cores and P_F fat cores, such that $4P_F + P_L = N$.

Assume the following:

- The key **benchmark** that your team cares about takes **200** seconds to run sequentially on a single **CPU-Lean** core, and **100** seconds to run sequentially on a single **CPU-Fast** core.
- The benchmark's computation consists of *parallel* and *sequential* parts, where the fraction of the original sequential time that is *parallel* is f. These two cannot overlap: while the sequential portion is executing on one core, the others remain idle.
- The parallel portion of the benchmark will experience linear speedup when it runs on multiple CPUs (i.e., there are no inefficiencies in running in parallel).

A. (1pt) First consider a **lean-only** machine, where $P_F = 0$. Write an equation for the total execution time of the benchmark, as a function of f and P_L, making optimal use of the processing elements.

Solution:

$$T = 200\left(\frac{1-f}{1} + \frac{f}{P_L}\right)$$

B. (2pts) Now consider the case where $P_F > 0$ Write an equation for the total execution time of the benchmark, as a function of f, P_L, and P_F, making optimal use of the processing elements.

Solution:

$$T = 200\left(\frac{1-f}{2} + \frac{f}{2P_F + P_L}\right)$$

C. (6pts) Using your equations above, calculate the **execution time (in seconds)** for the benchmark with the following machine configurations (all with $N = 20$), assuming $f = 0.90$.

As an aid, separately list the time spent for the sequential portion T_{seq}, the time spent for the parallel portion T_{par}, and the overall time $T_{\text{tot}} = T_{\text{seq}} + T_{\text{par}}$.

P_L	P_F	T_{seq}	T_{par}	T_{tot}
20	0			
0	5			
4	4			

Solution:

P_L	P_F	T_{seq}	T_{par}	T_{tot}
20	*0*	*20.0*	*9.0*	*29.0*
0	*5*	*10.0*	*18.0*	*28.0*
4	*4*	*10.0*	*15.0*	*25.0*

Problem 3. (8 points):

Consider the following version of compare-and-swap:

```
bool CAS(int *addr, int check, int new) {
  atomic {
    int old = *addr;     // Read
    if (old == check) {  // Compare
      *addr = new;       // Write
      return true;
    }
    return false;
  }
}
```

You are given the following sequential code implementing a bounded stack of integers using an array and a counter indicating the number of elements in the stack.

```
#define MAXLEN 1000
int stack[MAXLEN];
int count = 0;

void push(int x) {
    int ccount = count;
    if (ccount >= MAXLEN)
      return;    // Silently fail if stack is full
    stack[ccount] = x;
    count = ccount + 1;
}

void pop(int *val) {
    int ccount = count;
    if (ccount == 0)
      return;   // Silently fail if stack is empty
    *val = stack[ccount-1];
    count = ccount - 1;
}
```

Here are attempts at lock-free implementations of `push` and `pop`:

```
void push(int x) {
  while (1) {
    int ccount = count;
    if (ccount >= MAXLEN)
      return;    // Silently fail if stack is full
    if (CAS(&count, ccount, ccount+1)) {
      stack[ccount] = x;
      return;

    }
  }
}

void pop(int *val) {
  while (1) {
    int ccount = count;
    if (ccount == 0)
      return;   // Silently fail if stack is empty
    if (CAS(&count, ccount, ccount-1) {
      *val = stack[ccount-1];
      return;
    }
  }
}
```

A. (4 pts) Identify a problem with the lock-free versions of `push` and `pop`.

Solution: There's a race. Suppose one thread starts to push, successfully executing `CAS`. A second thread could start to pop, executing `CAS` and reading from the top of the stack before the first thread has a chance to store its value there. Trickier versions involving multiple pushes and pops are also possible.

B. (2 pts) Explain briefly why it is not possible to do lock-free implementations of these operations using CAS.

Solution: Push operation is doing two write operations, one on `count` and the other one by writing the value to the stack. Using a CAS we cannot guarantee that both operations happen together atomically.

C. (2 pts) Suppose you have a double-word CAS with the following prototype:

```
// Atomic compare-and-swap two integers simultaneously
// Both locations are updated if and only if both existing
// values match their check values.
bool DCAS(int *addr1, int check1, int new1,
          int *addr2, int check2, int new2);
```

Explain (without writing code) how you could implement the push operation using DCAS.

Solution: Using a DCAS, we can atomically increment the count and write the top-of-stack value. This will prevent any other thread from interrupting our update.

Memory Consistency

Problem 4. (10 points):

Assume that global variable `data` has initial value 0, and `ready` has initial value `false`. Consider the following code snippets being executed concurrently by two threads:

Thread 1:

```
1: *data = 1;
2: *ready = true;
3: *ready = false;
4: *data = 2;
5: *ready = true;
```

Thread 2:

```
A: while (!*ready) { /* nothing */ }
B: printf("Data = %d\n", *data);
```

A. (6 pts) For **sequentially consistent** execution, indicate which of the following outputs is possible. For each, give an ordering of the 7 steps (1–5 for Thread 1 and A–B for Thread 2) that would lead to this outcome. For the sequentially consistent cases, the ordering must be sequentially consistent. For the ones that are not sequentially consistent, give an ordering that minimizes the number of inconsistencies.

Output	Possible (Y/N)	Step ordering
Data = 0		
Data = 1		
Data = 2		

Solution:

Output	*Possible (Y/N)*	*Step ordering*
Data = 0	*N*	*2, A, B, 1, 3, 4, 5*
Data = 1	*Y*	*1, 2, A, B, 3, 4, 5*
Data = 2	*Y*	*1, 2, 3, 4, 5, A, B*

B. (2 pts) Now suppose this code runs on a processor with a weak consistency model, where loads and stores to different memory locations by one thread can appear to occur to other threads as if they did not occur in program order. Would this change what the program could print? Explain your answer.

Solution: Yes. Output "Data = 0" now becomes possible, because Thread 2 could observe the store of step 2 occuring before the store of step 1.

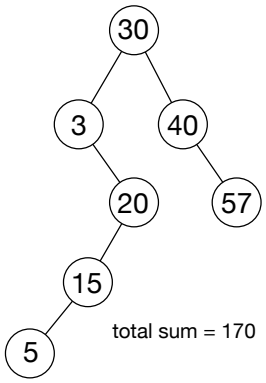
C. (2 pts) Where could you place a minimum set of fences to guarantee that only sequentially consistent outputs would occur with this program?

Solution: One fence is required between steps 1 and 2, and a second fence is required between steps A and B.

Transactions on Trees

Problem 5. (12 points):

Consider the binary search tree illustrated below.



The operations `insert` (insert value into tree, assuming no duplicates) and `sum` (return the sum of all elements in the tree) are implemented as transactional operations on the tree as shown below:

```
struct Node {
   Node *left, *right;
   int value;
};
Node* root;  // root of tree, assume non-null

void insertNode(Node* n, int value) {
  if (value < n->value) {
    if (n->left == NULL)
       n->left = createNode(value);
    else
       insertNode(n->left, value);
  } else {
    if (n->right == NULL)
       n->right = createNode(value);
    else
       insertNode(n->right, value);
  }
}

int sumNode(Node* n) {
   if (n == null) return 0;
   int total = n->value;
   total += sumNode(n->left);
   total += sumNode(n->right);
   return total;
}
```

```
void insert(int value) {
   bool done = false;
   while (!done) {
     xbegin();
     insertNode(root, value);
     done = xend();
}

int sum() {
   int rval = 0;
   bool done = false;
   while (!done) {
     xbegin();
     rval = sumNode(root);
     done = xend();
   }
   return rval;
}
```

Consider when the following four operations are executed by different threads, starting with the original tree.

```
T1: insert(10);
T2: insert(25);
T3: insert(24);
T4: printf("Sum = %d\n", sum());
```

A. (4 pts) Consider the different orders in which these operations could be executed. Draw all possible trees that could result. (**Note:** you can draw just the subtrees rooted at node 20, since that is the only part of the tree that is affected.)

Solution: There are only two. 20 –> 25 –> 24 or 20 –> 24 –> 25

B. (2 pts) How many different values could thread T4 print? Explain. (You need not list them.)

Solution: The original sum, adding any combination of 10, 25, and 24, for a total of 8 possibilities.

C. (2 pts) Do your answers to parts A or B change depending on whether the implementation of transactions uses optimistic or pessimistic conflict detection? Why or why not?

*Solution: Definitely not! The choice of how to **implement** a transaction cannot change the **semantics** of the transactional abstraction.*

D. (2 pts) Consider an implementation using **lazy** data versioning and **optimistic** conflict detection that manages transactions at the granularity of tree nodes (the read and writes sets are lists of nodes). Assume that the transaction for `insert(10)` commits when those for `insert(24)` and `insert(25)` are at node 20, and for `sum()` is at node 40. Which of the four transactions (if any) are aborted? **Please describe why.**

Solution: Only `sum` is aborted since the write set of the committing transaction (which is node 5) conflicts with the read set of `sum`. Note that there is no conflict with the other insertions since they read no data written by `insert(10)`.

E. (2 pts) Now consider a version that uses **optimistic** conflict detection for reads and **pessimistic** conflict detection for writes. Does transactional memory in this case offer any performance benefit for `sum()` compared to a fine-grained locking approach? Explain.

Solution: Probably not and in fact it might be worse, because the read set of the `sum()` transaction will almost always be changed by the insertions causing the `sum()` transaction to abort on commit wasting potentially a sizeable amount of work.

A Simple Image Processing Pipeline

Problem 6. (12 points):

Consider the following code to perform a vertical convolution on an input image.

```
float input[H+3][W];
float output[H][W];

void convolve(float output[H][W], float input[H][W]) {
  for (int j=0; j<H; j++) {
    for (int i=0; i<W; i++) {
      float accum = 0.f;
      for (int jj=0; jj<4; jj++) {
        // count as two floating-point operations
        accum += 0.25 * input[j+jj][i];
      }
      output[j][i] = accum;
    }
  }
}

convolve(output, input);
```

We consider execution under the following conditions:

- $H = W = 4096$.
- The cache is fully associate and uses write-back plus write-allocate policies. It has a capacity of 16,384 bytes and a block size of 32 bytes.
- Both arrays begin on cache boundaries.

A. (2 pts) What is the arithmetic intensity of this program, defined as the number of floating-point operations divided by the number of load and store operations.

Solution: It is $8/5 = 1.60$*. For each pixel, there are 4 loads, 1 store, and 8 arithmetic operations.*

B. (3 pts) Under the conditions described, what would be the cache hit rate for load operations?

Solution: It is 7/8. Each cache block is loaded once and then used for another seven values of `i`*. By the time the next iteration of* `j` *occurs, it would have been evicted.*

C. (3 pts) A colleague suggests switching the outer two loops, as follows:

```
void convolve(float output[H][W], float input[H][W]) {
  for (int i=0; i<W; i++) {
    for (int j=0; j<H; j++) {
      float accum = 0.f;
      for (int jj=0; jj<4; jj++) {
        // count as two floating-point operations
        accum += 0.25 * input[j+jj][i];
      }
      output[j][i] = accum;
    }
  }
}
```

What would be the hit rate for load operations in this case?

Solution: It is 3/4. Each block would be loaded for one value of `j` and then used used 3 more times. But, by the time the next iteration of `i` occurs, it would have been evicted.

D. (4 pts) Describe (in words; no code is necessary) how you could modify the second version of the program to achieve the maximum possible hit rate on loads, while having the same arithmetic intensity? What would that hit rate be?

Solution: Restructure the outer two loops so that the image is processed in vertical stripes that are 32/4 = 8 pixels wide. The hit rate would be 31/32.