Name: _____ Uniqname: _____

EECS 570 Winter 2022 Midterm Exam

Total points: 100
Total time: 80 minutes
Closed book, closed notes.
Calculators are allowed, but no PDAs, Portables, Cell phones, etc.
For calculation questions, show your work and clearly indicate your final answers.
Use pen or a dark pencil.
Do not spend too much time on any one question.
State any reasonable assumptions that you need to make.
Please try to write all your answers in the assigned spaces.
If you really need extra space, you may use the backs of pages (but please **clearly** indicate that you are doing so).

Please sign the Honor Code:

I have neither given nor received unauthorized aid on this examination, nor have I concealed any violations of the Honor Code.

_____

Q1	/10
Q2	/15
Q3	/20
Q4	/20
Q5	/15
Q6	/20
Total	/100

1 Amdahl's Law (10 points)

Consider a program where 20% of the computation must run serially, while the remaining 80% of the computation is parallelisable.

(a) (5 points) What is the speedup gained by running this program on a processor with 8 cores?

(b) (5 points) If we assume that the parallel portion of the program is "embarrassingly parallel" (i.e., its performance scales linearly in the number of cores for an arbitrary number of cores), what is the maximum possible speedup we can gain by parallelising this program?

2 Shared Memory and Message Passing (15 points)

(a) (8 points) Is it possible to implement a message passing system on a shared memory architecture? Why or why not? Briefly explain your answer.

(b) (7 points) Is it possible to implement a shared memory abstraction on top of a message passing architecture? Why or why not? (Assume that processes in the message passing system do not share any address space.) Briefly explain your answer.

3 GPUs, Costs, and a Dash of Datacenters (20 points)

Answer the following questions. **Be sure to briefly explain your answers.**

(a) (7 points) Why can branches in GPU code be problematic?

(b) (6 points) Under what circumstances will an n-processor parallel system be more **cost-effective** than using n uniprocessor systems?

(c) (7 points) Is there a single optimal architecture for datacenter hardware, or are different architectures optimal for different performance (QoS) requirements and different request distributions? ("Request distributions" here refers to the fraction of requests that are "big"/"small" requests, etc.)

4 Cache Coherence (20 points)

(a) (10 points) Consider a MESI snooping coherence protocol where each core has its own private cache. Assume that the memory controller responds to read requests by either providing the line to the requestor with exclusive permissions (E state) or without exclusive permissions (i.e., S state), depending on its knowledge of the current state of the system. In other words, it is the responsibility of the memory controller to decide whether the requestor gets the line in E state or S state. The protocol does not have clean eviction notifications for lines in S.

Is it possible to have an execution where at some point, no private caches have the line for a given address `x`, a private cache then requests the line, and the line is provided to it in S state (as opposed to E)? If yes, clearly describe how this could happen in an execution. If no, clearly explain why such a scenario is impossible.

(b) (5 points) Give one difference between typical snooping coherence protocols and typical directory coherence protocols.

(c) (5 points) Give an example of a coherence protocol deadlock that could occur in a naively designed coherence protocol due to network and/or buffering issues. The protocol you use in your example must be an MSI protocol.

5 Transactional Memory (15 points)

Consider the following lock-based program where each thread runs the code below. Assume that the function `threadId()` returns the thread number of the thread that calls it. So it will return 0 for thread 0, 1 for thread 1, so on and so forth. `a1` and `a2` are arrays of integers. `a1Len` contains the length of array `a1`, and `a2Len` contains the length of array `a2`. The function `process(..)` both reads and writes each of its arguments. The threads enforce mutual exclusion using lock `l1`. **Please read the code carefully!**

```
01:   lock(l1);
02:   for (i = threadId(); i < a1Len - 1; i += 8) {
03:     process(a1[i], a1[i+1]);
04:   }
05:
06:   for (i = threadId(); i < a2Len; i += 8) {
07:     a2[i] = a1[i];
08:   }
09:   unlock(l1);
```

(a) (7 points) Convert this lock-based code to code that uses only transactions. Your transaction-based code should only be able to generate exactly the same possible results that could have been generated by the original lock-based code. In other words, no new results should become possible, and no previously allowed results should become forbidden.

(b) (8 points) Assume that your transactional code runs on a hardware transactional memory system where each core has its own private cache which has 64-byte cache lines. Assume that integers are 64 bits (8 bytes) in size, and that arrays `a1` and `a2` can fit in a given core's private cache at the same time. The hardware transactional memory uses the coherence protocol (as is typical) to detect conflicts between transactions. In such a system, will the transactional memory code be faster, roughly as fast as, or a lot slower than the lock-based code? Explain your answer.

6 Synchronization (20 points)

(20 points) You are tasked with implementing a highly concurrent queue data structure that can be accessed in parallel by multiple threads **without locks**. The queue stores its entries as a singly-linked list with a head and tail pointer. Entries are added (enqueued) to the tail of the list and removed (dequeued) from the head of the list. The structure of the linked list nodes and the list's enqueue function are as follows. (In the code below, `fetch_and_store(A, B)` stores the value of B at the memory location A, and returns the old value at memory location A. So for instance, `fetch_and_store(tail, I)` makes `tail` point to `I` and returns the old value of `tail`.)

```
01:   struct node {
02:     node *next;
03:     void *data;
04:   }
05:
06:   node *head = NULL;
07:   node *tail = NULL;
08:
09:   void enqueue(node *I) {
10:     if (I == NULL)
11:       return;
12:
13:     I->next = NULL;
14:     pred = fetch_and_store(tail, I);
15:     if (pred == NULL) {
16:       head = I;
17:     } else {
18:       pred->next = I;
19:     }
20:   }
```

Implement the list's dequeue function **on the next page**. The dequeue function should remove the head of the list and return a pointer to it, updating the list's head and tail pointers appropriately. If the queue is currently empty, the dequeue function should spin until the queue becomes non-empty. **You may assume that at most one instance of the dequeue function is ever running at any time (similar to how at most one thread can ever be releasing an MCS lock at any time).** However, multiple enqueuers may be running the enqueue function at the same time, and multiple enqueuers may overlap execution with an instance of your dequeue function. The code must run correctly in all such cases.

Remember, **you may not use locks in your dequeue function**. However, you can use a compare-and-swap (CAS) instruction, whose semantics are as follows:

`CAS(A, B, C)` compares the value at address A with the value of B. If they match, it stores C at address A and returns true. Otherwise it returns false. So for example, `CAS(tail, head, NULL)` checks if `tail` and `head` point to the same address, and if they do, it sets `tail` to NULL and returns true. If `tail` and `head` point to different addresses, it returns false. The CAS instruction executes atomically.

```
node *dequeue() {

}
```