15-418/618 Exam 2

Jiaqiang Ruan

TOTAL POINTS

59 / 81

QUESTION 1

Multiple Choice 15 pts

1.1 1A **0 / 1**

+ **1 pts** Correct

✓ **+ 0 pts Incorrect**

1.2 1B **0 / 1**

+ **1 pts** Correct

✓ **+ 0 pts Incorrect**

1.3 1C **0 / 1**

+ **1 pts** Correct

✓ **+ 0 pts Incorrect**

1.4 1D **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.5 1E **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.6 1F **0 / 1**

+ **1 pts** Correct

✓ **+ 0 pts Incorrect**

1.7 1G **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.8 1H **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.9 1I **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.10 1J **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.11 1K **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.12 1L **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.13 1M **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.14 1N **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

1.15 1O **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

QUESTION 2

Locks and Memory Consistency 18 pts

2.1 2A.1 Lock Properties **1 / 1**

+ **0 pts** Incorrect

✓ **+ 1 pts Correct**

2.2 2A.2 Lock Properties **1 / 1**

+ **0 pts** Incorrect

✓ + 1 pts **Correct**

2.3 2A.3 Lock Properties 1 / 1

+ 0 pts Incorrect

✓ + 1 pts **Correct**

2.4 2A.4 Lock Properties 0 / 1

✓ + 0 pts **Incorrect**

+ 1 pts Correct

Fetch and add actually is implemented as a CAS/load linked-store conditional. But, for this problem, the key insight is seeing that the busy loop of a ticket lock just consists of no RdX requests. We have a problem with the CAS loop since there is a very high chance of the CAS failing under high contention, requiring a restart of the CAS. This results in a lot of repeated invalidation, and bus transactions. The delay is meant to solve this, but such a problem doesn't occur in the spin loop of a ticket lock. All we see is initially Rd's, then reading from each processor's cache, then a single Rdx (by the thread unlocking), and the cycle repeats.

2.5 2A.5 Lock Properties 2 / 2

+ 0 pts Incorrect

✓ + 2 pts **Correct**

2.6 2A.6 Lock Properties 0 / 2

✓ + 0 pts **Incorrect**

+ 2 pts Correct

The only elements from the ticket lock struct that are in each thread's caches is next_ticket, and serving. Also, all three elements of the struct are on separate cache lines (padding is on its own cache line because it has to be cache aligned) For threads 0-7, serving will be in a shared state. We know that thread 0 is in the critical section, and we haven't updated serving yet until we leave the critical section. For threads 0-6, next_ticket will be invalidated, but will be modified in thread 7. This is because it is the last thread to modify it, and no other thread has attempted to read or write it yet.

2.7 2A.7.a Lock Properties 2 / 2

✓ + 2 pts **Correct**

+ 1.5 pts Partial

+ 1 pts Partial

+ 0 pts Incorrect

2.8 2A.7.b Lock Properties 1 / 2

+ 2 pts Correct

✓ + 1 pts **Partial**

+ 0 pts Incorrect

When re-entering the busy-wait loop, Thread 0 would then issue a bus read request for ticket->serving and get a shared copy.

2.9 2B.1 Memory consistency 1 / 1

✓ + 1 pts **Correct**

+ 0 pts Incorrect.

2.10 2B.2 Memory consistency 1 / 1

✓ + 1 pts **Correct**

+ 0 pts Incorrect.

2.11 2B.3 Memory consistency 1 / 1

✓ + 1 pts **Correct**

+ 0 pts Incorrect

2.12 2B.4 Memory consistency 0 / 1

+ 1 pts Correct

✓ + 0 pts **Incorrect**

2.13 2B.5 Memory consistency 1 / 1

✓ + 1 pts **Correct**

+ 0 pts Incorrect

2.14 2B.6 Memory consistency 0 / 1

+ 1 pts Correct

✓ + 0 pts **Incorrect**

QUESTION 3

Managing Concurrency 10 pts

3.1 3A 2 / 2

✓ **+ 2 pts** **Deadlock, explanation**

+ 1 pts Deadlock

+ 0 pts Incorrect

3.2 3B 2 / 2

✓ **+ 2 pts** **Yes, explanation**

+ 1 pts Yes

+ 0 pts Incorrect

3.3 3C 2 / 2

✓ **+ 2 pts** **Aborts, delete_from_head**

+ 1 pts Aborts

+ 0 pts Incorrect

3.4 3D 2 / 2

✓ **+ 2 pts** **No aborts**

+ 0 pts Incorrect

3.5 3E 2 / 2

✓ **+ 2 pts** **No aborts**

+ 0 pts Incorrect

QUESTION 4

Heterogeneous vs. Homogeneous Parallelism 20 pts

4.1 4A.1 Homogeneous sequential 1 / 1

✓ **+ 1 pts** **Correct**

+ 0 pts Incorrect

+ 0.5 pts Almost correct, has an extra f factor

4.2 4A.2 Homogeneous parallel 1 / 1

✓ **+ 1 pts** **Correct**

+ 0 pts Incorrect

+ 0.5 pts Extra f term

4.3 4A.3 Homogeneous end-to-end 2 / 2

✓ **+ 1 pts** **OOO core correct**

✓ **+ 1 pts** **IO core**

+ 1 pts Effort only

+ 0 pts Incorrect/blank

4.4 4A.4 Homogeneous best design f=0.9 1 / 1

✓ **+ 1 pts** **Correct**

+ 0.5 pts Correct but incompletely answered

+ 0 pts Incorrect

4.5 4A.5 Homogeneous best design f=0.8 1 / 1

✓ **+ 1 pts** **Correct**

+ 0.5 pts Correct but incomplete

+ 0 pts Incorrect

4.6 4A.6 Homogeneous qualitative 1 / 1

✓ **+ 1 pts** **Correct**

+ 0.5 pts Close

+ 0 pts Incorrect

4.7 4B.1 Heterogeneous sequential 1 / 1

✓ **+ 1 pts** **Correct**

+ 0 pts Incorrect.

+ 0.5 pts Incorrect inclusion of (1-f).

4.8 4B.2 Heterogeneous parallel 1 / 1

✓ **+ 1 pts** **Correct**

+ 0 pts Incorrect.

- 0.5 pts Unnecessary f term.

- 0.5 pts Missed the OOO core in the parallel region.

- 0.5 pts Didn't account for area of OOO core.

4.9 4B.3 Heterogeneous end-to-end 1 / 1

✓ **+ 1 pts** **Correct**

+ 1 pts Correct (based on previous answers)

+ 0 pts Incorrect

- 0.5 pts Missed OOO core in parallel region.

- 0.5 pts Inverse of correct answer.

✓ **+ 0 pts** **Weird formula. Possibly equivalent to the right answer, but hard to tell.**

4.10 4B.4 Heterogeneous best r_l = 1 2 / 2

✓ **+ 2 pts Correct**

+ 2 pts Close enough.

+ 1 pts Pretty close.

+ 0 pts Incorrect.

4.11 4B.5 Heterogeneous best r_l varies 2 / 2

✓ **+ 2 pts Correct**

+ 2 pts Close enough.

+ 1 pts Wrong speedup, but correct trend wrt core size.

+ 0 pts Incorrect.

4.12 4B.6 Heterogeneous qualitative 0.5 / 1

✓ **+ 1 pts Correct**

✓ **- 0.5 pts Doesn't emphasize the importance of having BOTH a big CPU and a GPU to achieve optimal performance.**

- 0.5 pts Doesn't identify analogy to GPUs.

+ 0 pts Was looking for more insightful discussion.

+ 0 pts Incorrect.

4.13 4C.1 Scaling N=4 2 / 2

✓ **+ 2 pts Correct**

+ 1 pts Wrong answer, but correct trend (homogeneous vs heterogeneous).

+ 0.5 pts Heterogenous speedup is wrong, leading to wrong conclusion.

+ 0.5 pts Homogeneous speedup is wrong, leading to wrong conclusion.

+ 0 pts Incorrect.

4.14 4C.2 Scaling N=16 2 / 2

✓ **+ 2 pts Correct**

+ 2 pts Close enough.

+ 1 pts Wrong answer, but correct overall trends.

+ 1 pts Incomplete answer.

+ 0 pts Incorrect.

4.15 4C.3 Scaling qualitative 1 / 1

✓ **+ 1 pts Correct**

+ 0 pts Incorrect.

+ 0 pts Answer does not address the scaling of heterogeneous vs homogeneous designs.

QUESTION 5

Memory Performance during Convolution Computation 18 pts

5.1 5A.A(1) Sequential Performance 0.5 / 0.5

✓ **+ 0.5 pts Correct: 1.0**

+ 0 pts Incorrect

5.2 5A.A(2) Sequential Performance 0 / 0.5

+ 0.5 pts Correct: 3.0

✓ **+ 0 pts Incorrect**

5.3 5A.A(3) Sequential Performance 0 / 0.5

+ 0.5 pts Correct: 13.0

✓ **+ 0 pts Incorrect**

5.4 5A.B(1) Sequential Performance 0.5 / 0.5

✓ **+ 0.25 pts 3,843,072**

✓ **+ 0.25 pts Main Memory**

+ 0 pts Incorrect

5.5 5A.B(2) Sequential Performance 0.25 / 0.5

+ 0.25 pts 3,264

✓ **+ 0.25 pts L1 Cache**

+ 0 pts Incorrect

5.6 5A.B(3) Sequential Performance 0 / 0.5

+ 0.5 pts Correct: 192

✓ **+ 0 pts Incorrect**

5.7 5A.C(1) Sequential Performance 0.5 / 0.5

✓ **+ 0.25 pts 626,688**

✓ **+ 0.25 pts Main Memory**

+ 0 pts Incorrect

5.8 5A.C(2) Sequential Performance 0.5 / 0.5

✓ **+ 0.25 pts 626,688**

✓ **+ 0.25 pts Main Memory**

+ **0 pts** Incorrect

5.9 5A.C(3) Sequential Performance **0 / 0.5**

+ **0.5 pts** Correct: 1

✓ **+ 0 pts Incorrect**

5.10 5A.D(1) Sequential Performance **0 / 1**

+ **1 pts** Correct

✓ **+ 0 pts Incorrect**

5.11 5A.D(2) Sequential Performance **0 / 0.5**

+ **0.5 pts** Correct: ~1.0

✓ **+ 0 pts Incorrect**

5.12 5A.D(3) Sequential Performance **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

5.13 5A.D(4) Sequential Performance **0 / 0.5**

+ **0.5 pts** Correct: 13.0

✓ **+ 0 pts Incorrect**

5.14 5A.D(5) Sequential Performance **0.25 / 0.5**

+ **0.5 pts** Correct: 14.0

✓ **+ 0.25 pts Incorrect but consistent**

+ **0 pts** Incorrect

Interesting that you got the correct final numerical value

5.15 5B.A(1) Parallel Performance **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

5.16 5B.A(2) Parallel Performance **0.5 / 0.5**

✓ **+ 0.5 pts Correct**

+ **0 pts** Incorrect. Based on an invalid analysis

+ **0 pts** Incorrect. No work shown

5.17 5B.A(3) Parallel Performance **1 / 1**

✓ **+ 1 pts Correct**

+ **0 pts** Incorrect

5.18 5B.A(4) Parallel Performance **0 / 0.5**

+ **0.5 pts** Correct

✓ **+ 0 pts Incorrect. Faulty reasoning**

+ **0.25 pts** Incorrect. Minor math error

5.19 5B.A(5) Parallel Performance **0.25 / 0.5**

+ **0.5 pts** Correct

+ **0 pts** Incorrect

✓ **+ 0.25 pts Incorrect, but consistent**

+ **0.25 pts** Minor math error

5.20 5B.A(6) Parallel Performance **0.25 / 0.5**

+ **0.5 pts** Correct

+ **0 pts** Incorrect

✓ **+ 0.25 pts Incorrect, but consistent**

5.21 5B.B Parallel Performacne **0 / 3**

+ **3 pts** Correct

+ **1 pts** Does not follow specified structure

+ **2 pts** Performs extra work

✓ **+ 0 pts Invalid code**

+ **2 pts** Need to stagger offsets by 4

- **1 pts** No explanation given

+ **1 pts** Valid strategy

5.22 5B.C Parallel Performance **0 / 1.5**

+ **1.5 pts** Correct

+ **0.5 pts** Ignores cost of loading additional values

✓ **+ 0 pts Incorrect**

+ **0 pts** Incorrect. No explanation given

+ **0.75 pts** Incorrect but consistent

+ **1.25 pts** Minor math error

+ **1 pts** Major math error

5.23 5B.D Parallel Performance **0 / 1.5**

+ **1.5 pts** Correct

+ **0.5 pts** Based on invalid analysis

✓ **+ 0 pts Incorrect**

+ **0.75 pts** Incorrect, but consistent

+ **1.25 pts** 1.25 Minor math error

Full Name: Jiaqiang Ruan
Andrew Id: jruan

15-418/618
Exam 2
Answer Sheet

Problem 1: Multiple Choice (15 points)

Problems that state "list all that apply" may have any number of correct answers (including zero). If none are correct, list "none." The other problems have unique answers.

A. Which of the following memory consistency policies allow the following behavior? Assume x and y start with value zero. (List all that apply.)

```
Thread 0        Thread 1
--------        --------
x <- 1          y <- 1
print(y)        print(x)

Output:
0
0
```

(a) Non-coherent shared memory.
(b) Coherent shared memory.
(c) Total-store-order (TSO) consistency.
(d) Sequential consistency.

Answer: ac

B. What is the key optimization enabled by total-store-order (TSO) consistency vs. sequential consistency (SC)? (List all that apply.)

(a) MESI coherence.
(b) Prefetching.

(c) Atomic memory operations.

(d) Store buffers.

Answer: b

C. Which of the following statements are true about implementations of MPI (list all that apply)

(a) MPI assumes the program is running on a machine with a cache-coherent shared address space.

(b) The MPI library allocates the needed space for all buffers.

(c) Synchronous communication creates a synchronization barrier for the sender and receiver.

(d) A sender can modify the send buffer as soon as the call to `MPI_Send` returns.

Answer: d

D. Which of the following statements are true about implementations of OpenMP (list all that apply)

(a) OpenMP assumes the program is running on a machine with a cache-coherent shared address space.

(b) OpenMP can only exploit parallelism by splitting up the index ranges in `for` loops.

(c) OpenMP implements all atomic operations via locks.

(d) OpenMP reduction assumes the reduction operation is associative.

Answer: ad

E. Assume that in a computer system, all operations continually abort and retry. Which one of the following could describe this scenario?

(a) Deadlock.

(b) Livelock.

(c) Starvation.

(d) None of the above.

Answer: b

F. To implement a lock-free single reader, single writer unbounded queue, how many additional pointers does the program need beyond those needed for a sequential implementation?

(a) 0

(b) 1

(c) 2

(d) 3

Answer: c

G. Which of the following statements is/are true? List all that apply.

(a) Test & Test and Set locks guarantee fairness of locks, as opposed to Test and Set locks

(b) It is better to spinlock than block a thread if scheduling overhead is larger than expected wait time

(c) You can replace all uses of locks with atomic transaction regions

(d) Pessimistic conflict detection detects conflicts earlier than optimistic detection

Answer: bd

H. Which of the following are domain-specific languages? List all that apply.

(a) Ruby.

(b) Liszt.

(c) Halide.

(d) C++.

Answer: bc

I. You are tasked with implementing a hardware transactional memory system in a new multicore processor with a local cache for each processor. Which of the following pieces of information are necessary for you to proceed with your implementation? (List all that apply)

(a) The cache coherence protocol

(b) The clock rate of each core

(c) The number of cores

(d) The versioning policy

Answer: ad

J. When training neural networks on a cluster and using a parameter server, what can we always expect?

(a) Parameters are always fresh

(b) The neural network is running in some model-parallel manner

(c) Training will always converge

(d) Updates to the server happen asynchronously

Answer: d

K. Which of the following statements is/are true? (list all that apply)

(a) ASICs have longer development workflows than FPGAs

(b) An ASIC circuit is more reconfigurable than a FPGA circuit

(c) ASICs developed for a given task are faster than FPGAs at that task

(d) ASICs developed for a given task use more power than FPGAs for the same task

Answer: ac

L.
```
lock:     ts  R3, memory[address]
          bnz R3, lock
unlock :  st  memory[address], #0
```

Assume the above assembly code follows a RISC ISA, with **ts** representing the test and set instruction. Assume that four threads running on four different processors are trying to acquire the lock. Which of the following is a/are plausible scenarios due to given lock mechanism?

(a) Starvation

(b) Deadlock

(c) Above lock doesn't ensure mutual exclusion

(d) All of the above

Answer: a

M. Which of the following performance evaluation tools has the least overhead?

(a) valgrind

(b) gprof

(c) top

(d) pin

Answer: c

N. What is the worst case latency of an n-node mesh network?

(a) $O(\sqrt{n})$

(b) $O(\log n)$

(c) $O(n)$

(d) $O(n \log n)$

Answer: a

O. Which of the following interconnect networks is non-blocking?

(a) Mesh

(b) Fat Tree

(c) Tree

(d) Multi-stage Logarithmic

Answer: b

Problem 2: Locks and memory consistency (18 points)

2A: Lock properties (12 points)

Provide brief answers to the following questions:

(1) The C keyword `volatile` indicates to the compiler that changes to a value may occur between different accesses, even if it does not appear to be modified. Why must the field `serving` in `ticket_lock_t` be declared as volatile to guarantee correct compilation?

Answer: The program uses volatile to ensure `serving` stored in memory/cache that is visible to all processors instead of register or private buffer in some processor. Then all the access to `serving` will be referred to this unique copy.

(2) Does this lock guarantee fairness (i.e., any request will eventually be granted)? Explain.

Answer: Yes. Because the fetch-and-add operation is atomic and always completes. Each thread will have its unique ticket number and will be executed in order.

(3) Why can the incrementing of `next->serving` in function `ticket_lock_unlock` be done with ordinary addition rather than atomic fetch-and-add?

Answer: Because there is only one thread owning and releasing the lock at a time. There is no contention.

(4) We saw in Exercise 4 that we could improve the performance of a lock based on atomic compare-and-swap by inserting a random delay in the busy-waiting loop, with the amount of the delay following an exponential distribution. In similar experiments for a ticket lock, this scheme leads to worse performance. Explain why that could be the case.

Answer: Trying compare-and-swap in a loop results in lots of interconnect traffic, whereas fetch-and-add only needs interconnect traffic that is linear to the number of callers.

(5) Suppose the definition of type `ticket_value_t` were changed from `unsigned long` to `unsigned char`. Would this impose any additional limits on the values of `niters` or `nthreads` in the main program? Explain.

Answer: The range of unsigned char is from 0 to 255. Increasing 255 results in 0. Therefore the maximum limit for the threads that are waiting or holding the lock is 256. nthreads should be less or equal to 256. There is no limit on niters, because each thread iterates sequentially.

(6) Suppose you run the main program on an 8-core processor with `nthreads` set to 8. The machine uses a bus-based cache coherence system following an MSI protocol. Assume at some point in the main program execution that Thread 0 is in its critical section and all other threads are in their busy-wait loops, attempting to acquire the lock, with Thread 1 having the smallest ticket value and Thread 7 having the largest. For all of the caches, describe which field(s) of the lock would be held in each cache and what their state(s) would be.

Answer: All eight threads have local cache for `serving` with shared state. Only thread 7 has local cache for `next_ticket` with modified state.

(7) Describe all of the bus traffic and cache transitions that would occur due to the following sequence. Describe only the traffic relevant to the lock. Assume there are no conflict misses. For each of the actions listed, describe any resulting actions that would occur by other threads or caches until a steady state is reached.

(a) Thread 0 exits its critical section and releases the lock, and Thread 1 then acquires the lock and enters its critical section.

Answer: Thread 0 first increases `serving` and releases the lock. This issues BusRdX and turn `serving` into modified state. All other threads invalidate `serving`. Other threads read and test `serving`. This issues BusRd and get `serving` in shared state. Thread 0 flushes `serving` and turn it into shared state. Thread 1 enters critical section.

(b) Thread 0 attempts to acquire the lock and re-enters the busy-wait loop.

Answer: Thread 0 increases `next_ticket`. This issues BusRdX and will get `next_ticket` into modified state. The last thread that increased `next_ticket` would hold it in modified state. After receiving BusRdX from thread 0, it flush and invalidate `next_ticket`. Thread 0 enters busy-wait loop.

2B: Memory consistency (6 points)

For each of the six possible fence locations either: 1) justify why it is required by describing a scenario in which incorrect behavior could arise without it, or 2) justify why it is not required.

Possible fence #1

Answer: Required. The thread spawns before the initialization is finished and acquires the lock because of the wrong initialization.

Possible fence #2

Answer: Not required. Previous fetch-and-add makes sure that serving is read only after my-ticket is added.

Possible fence #3

Answer: Not required. The order of reading serving and testing my-ticket doesn't matter. Because the reading and testing are in the loop. The latest value of serving will be finally read and the test will break.

Possible fence #4

Answer: Not required. My ticket is a local variable and the variable serving in the lock is only read but not write. The order of these operations will not have effect on other threads.

Possible fence #5

Answer: Required. The thread spawns before the initialization is finished and acquires the lock because of the wrong initialization.

Possible fence #6

Answer: Required. Before serving is incremented, there comes up with a thread having next ticket number as the old serving number and acquire the lock (eg. getting 0 by increasing 255 in the unsigned char case).

Problem 3: Managing Concurrency (10 points)

Provide brief answers to the following questions:

A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

- Test 1: `insert_from_head(1)`, `delete_from_head(9)`
- Test 2: `insert_from_head(8)`, `delete_from_head(2)`
- Test 3: `insert_from_head(8)`, `insert_from_tail(1)`

The first two unit tests complete without error, but the third test goes badly and it does not terminate with the right answer. Describe what behavior is observed and why the problem occurs. (All unit tests start with the list in the state shown above.)

Answer: There is dead lock. Insert-from-head with 8 want to acquire lock on the element of 6 and 9 by HOH starting from 0. At the same time, insert from tail with 1 want to acquire lock on the element of 0 and 2 starting from the tail. They both wait for the lock acquired by the other thread to proceed somewhere from the element of 2 to 6.

B. Imagine you removed all locks and implemented `insert_from_head`, `delete_from_head`, and `insert_from_tail` by placing the entire body of these functions in an atomic block for execution on a system **supporting lazy optimistic transactional memory with aggressive conflict resolution**. Does this fix the correctness problem described in part A? Why or why not?

Answer: Yes. The first thread that finished the transaction will commit and force the other transaction to abort and restart because of the aggressive conflict resolution. Then the other transaction will restart and commit.

C. Consider two transactions simultaneously performing `insert_from_head(3)` and `delete_from_head(9)`. Assume both transactions start at the same time on different cores and the transaction for `insert_from_head(3)` proceeds to commit while the `delete_from_head(9)` transaction has just iterated to the node with value 6. Must either of the two transactions abort in this situation? Why? **(Remember this is an lazy optimistic transactional memory system!)**

Answer: Yes. Lazy optimistic transaction system will store the modification in buffer and only check conflict when it is about to commit. Since delete-from-head(9) has proceeded to the node of 6, it didn't see the inserted 3 when it is proceeding the list. So either delete-from-head aborts when he found out the inserted node caused by the commit of insert-from-head. Or insert-from-head aborts when it is about to commit and detects the conflict in the other transaction.

D. Must either transaction abort if the transaction for `delete_from_head(9)` proceeds to commit before the transaction for `insert_from_head(3)` does? Why? **Please assume that at the time of the attempted commit, `insert_from_head(3)` has iterated to node 2, but has not begun to modify the list.**

Answer: No. When delete-from-head commit, it didn't detect conflict because the meory is not changed and other transaction didn't make any modification. So it commits. As for insert-from-head, it doesn't reply on the node after node 3. So when it commits, it won't detect conflict because of the inserted node 9. So it will commit too.

E. Must either transaction abort if the situation in part D is changed so that `delete_from_head(9)` attempts to commit first, but by this time `insert_from_head(3)` has made updates to the list (although not yet initiated its commit)? Why?

Answer: It depends on what granularity the conflict detection is implemented on. If the conflict detection is implemented on the list itself. Then since the list is modified, there will be conflict and abortion. If the granularity is implemented on node level, delete-from-head(9) only modifies node 6 and insert-from-head(3) modifies node 2 and 5. There won't be conflict and abortion.

Problem 4: Homogeneous vs. Heterogeneous Parallelism (20 points)

This problem concerns the costs and benefits of heterogeneous parallelism.

We will begin by considering multicore CPU processors, comparing the performance of symmetric (homogeneous) and asymmetric (heterogeneous) designs.

Imagine you are a hardware architect designing a new multicore CPU. Suppose that the CPU can take at most N resources (e.g., resources could be area of at most 200 mm^2 or power of at most 100 W).

You can design CPU cores with the following characteristics:

- **Out-of-order ("OOO"):** The OOO core achieves sequential speedup (vs. some baseline core) as a function of resources r
$$\text{speedup}_O(r) = \sqrt{r+1} - 1 \tag{1}$$
- **In-order ("IO"):** The IO core achieves sequential speedup (vs. some baseline core) as a function of resources r:
$$\text{speedup}_I(r) = 1 - \frac{1}{2r+1} \tag{2}$$

For this problem, all speedups are normalized to a core with sequential performance $= 1$. Feel free to give approximate numerical answers (e.g., within 0.1 of optimal), and we recommend that you write a simple program or use spreadsheet software to get numerical answers.

4A: Homogeneous multicore (7 points): Suppose an application spends a fraction f of its execution running in parallel with ideal speedup, and $1 - f$ executing sequentially.

(1) What is the speedup **in the sequential region** of the program for a single core of size r? (Give an expression involving N, f, and r.)

OOO Core. *Answer*:
$$\text{speedup}_O(r) = \sqrt{r+1} - 1 \tag{3}$$

IO Core. *Answer*:
$$\text{speedup}_I(r) = 1 - \frac{1}{2r+1} \tag{4}$$

(2) What is the speedup **in the parallel region** of the program for a homogeneous multicore? (Give an expression involving N, f, and r.)

OOO Core. *Answer*:

$$\frac{N}{r} * \text{speedup}_O(r) = \frac{N}{r} * (\sqrt{r+1} - 1) \tag{5}$$

IO Core. *Answer*:

$$\frac{N}{r} * \text{speedup}_I(r) = \frac{N}{r} * (1 - \frac{1}{2r+1}) \tag{6}$$

(3) What is the **end-to-end speedup** for a homogeneous multicore? (Give an expression involving N, f, and r.)

OOO Core. *Answer*:

$$\frac{1}{\frac{f}{\frac{N}{r} * \text{speedup}_O(r)} + \frac{1-f}{\text{speedup}_O(r)}} = \frac{N}{rf + N(1-f)} * (\sqrt{r+1} - 1) \tag{7}$$

IO Core. *Answer*:

$$\frac{1}{\frac{f}{\frac{N}{r}*\text{speedup}_I(r)} + \frac{1-f}{\text{speedup}_I(r)}} = \frac{N}{rf + N(1-f)} * (1 - \frac{1}{2r+1}) \tag{8}$$

(4) Suppose that $f = 0.9$ and $N = 16$. What speedup is achieved by the **best** homogeneous OOO and IO designs? Which performs better? How many cores does the best design have?

Answer: The best homogeneous OOO is r = 4, speedup = 3.803, it has 4 cores. The best homogeneous IO is r = 1, speedup = 4.267, it has 16 cores. The IO is better.

(5) Now suppose that $f = 0.8$ and $N = 16$. What speedup is achieved by the **best** homogeneous OOO and IO designs? Which performs better? How many cores does the best design have?

Answer: The best homogeneous OOO is r = 8, speedup = 3.333, it has 2 cores. The best homogeneous IO is r = 1, speedup = 2.667, it has 16 cores. The OOO is better.

(6) Compare your last two answers for $f = 0.9$ to $f = 0.8$. What do they say about how applications affect the best architecture? About the resilience of different architectural choices to different applications?

Answer: The proportion of sequential and parallel of the code will affect the best choice of the architecture. If there is large amount of sequential work, it might prefer to have higher performance for each core instead of having more cores. The OOO architecture in this question is more stable to the change of f instead of IO architecture. The speedup of IO architecture declines quickly as the f is getting smaller.

4B: Heterogeneous multicore (8 points): Now suppose that you are able to combine different kinds of cores in your multicore design. Let's consider the performance of a heterogeneous multicore CPU using a mix of OOO and IO cores, written as a function of f, N, r_O (out-of-order core size) and r_I (in-order core size).

Note: The optimal heterogeneous design will consist of a single OOO core plus many IO cores. The sequential region runs on the OOO core.

(1) What is the speedup **in the sequential region** of the program? (Give an expression involving N, f, r_O, and r_I.)

Answer:

$$\text{speedup}_O(r_O) = \sqrt{r_O + 1} - 1 \tag{9}$$

(2) What is the speedup **in the parallel region** of the program for a heterogeneous multicore? (Give an expression involving N, f, r_O, and r_I.)

Answer:

$$\frac{N - r_O}{r_I} * \text{speedup}_I(r_I) + \text{speedup}_O(r_O) = \frac{N - r_O}{r_I} * (1 - \frac{1}{2r_I + 1}) + \sqrt{r_O + 1} - 1 \tag{10}$$

(3) What is the **end-to-end speedup** for a heterogeneous multicore? (Give an expression involving N, f, r_O, and r_I.)

Answer:

$$\frac{1}{\frac{f}{\frac{N-r_O}{r_I} * \text{speedup}_I(r_I) + \text{speedup}_O(r_O)} + \frac{1-f}{\text{speedup}_O(r_O)}} = \frac{(\frac{N-r_O}{r_I} * S_I + S_O) * S_O}{S_O + (1-f) * \frac{N-r_O}{r_I} * S_I} \tag{11}$$

$$= \frac{(\frac{N-r_O}{r_I} * (1 - \frac{1}{2r_I+1}) + \sqrt{r_O + 1} - 1) * (\sqrt{r_O + 1} - 1)}{(\sqrt{r_O + 1} - 1) + (1-f) * \frac{N-r_O}{r_I} * (1 - \frac{1}{2r_I+1})} \tag{12}$$

(4) Suppose that $f = 0.9$ and $N = 16$ and, for now, fix in-order cores at size $r_I = 1$. What speedup does the **best** heterogeneous design achieve?

Answer: The best speedup is 5.916 with $r_O = 6$

(5) What is the **best** overall speedup when the **in-order core size** r_I **varies** from 0.1, 0.5, 1.0, 2.0, to 4.0?

Answer: when r_I = 0.1, the best speedup is 9.250, r$_O = 7.3$.
$when$r$_I$ = 0.5, the best speedup is 7.268, r$_O = 6.5$.
$when$r$_I$ = 1.0, the best speedup is 5.916, r$_O = 6$.
$when$r$_I$ = 2.0, the best speedup is 4.542, r$_O = 6$.
$when$r$_I$ = 4.0, the best speedup is 3.469, r$_O = 8$.

(6) Qualitatively, what does this say about the optimal heterogeneous architecture? (Does it remind you of any designs you've used this semester?)

Answer: The optimal heterogeneous architecture is higher than the homogeneous one. And as for application that involves lots of parallel work, it is better to have more simple cores to finish the work instead of a few good performance cores. This remind me of GPU design. It uses lots of core to finish the parallel work.

4C: Scaling (5 points): Finally, let's use the model you've already developed to see how things change with different resource budgets. For the rest of this problem, fix $f = 0.9$ and $r_I = 0.1$ in the heterogeneous design.

(1) What are the best **heterogeneous** and **homogeneous** designs when $N = 4$?

Answer: Best homogeneous designs is 8 IO cores with r = 0.5, speedup is 2.353. Best heterogeneous designs is $r_O = 1.8$, speedup is 2.810.

(2) What are the best **heterogeneous** and **homogeneous** designs when $N = 64$?

Answer: Best homogeneous designs is 32 IO cores with r = 2, speedup is 6.244. Best heterogeneous designs is $r_O = 31.2$, speedup is 27.353.

(3) What does this say about the benefits of heterogeneity as resource budgets increase over time?

Answer: When the resource budget is larger, heterogeneous design has a even better performance than homogeneous design.

Problem 5: Memory Performance during Convolution Computation (18 points)

5A: Sequential performance (8 points)

A. Consider the following code:

```
#define LEN 5932
float data[LEN];
float sum = 0.0;
for (int i = 0; i < LEN; i++)
   sum += data[i];
```

Calculate the CPIS for this code assuming array `data` is initially in the following memory levels. Remember: report these numbers to the first decimal place.

L1 cache. *Answer*: 1 clock cycle

L2 cache. *Answer*: 9 clock cycles

Main memory. *Answer*: 49 clock cycles

B. Determine the following regarding function `convolve`:

(1) The total number of input data elements accessed during the computation, and the highest memory level in which this would fit (where L1 $>$ L2 $>$ Memory.)
Answer: (20000+16)*192 = 3843072, this only fits in the main memory

(2) The number of input data elements accessed for one value of outer loop index `i`, and the highest memory level in which this would fit.
Answer: 17*192 = 1564, this can fit in L1.

(3) The number of times each input data element is accessed for one value of outer loop index `i`.
Answer: 17*192 = 1564.

C. Determine the following regarding function `convolve`:

(1) The total number of weight values accessed during the computation, and the highest memory level in which this would fit.
Answer: 192*192*17=626688, this can only fit in main memory.

(2) The number of weight values accessed for one value of outer loop index `i`, and the highest memory level in which this would fit.
Answer: 192*192*17=626688, this can only fit in main memory.

(3) The number of times each weight value is accessed for one value of outer loop index `i`.
Answer: 20000

D. Determine the single-core performance of function `convolve`:

(1) Describe the access pattern for the input data within the innermost two loops, in terms of which levels of memory the data would reside in.
Answer: For ii from 0 to csize-2, input[i+ii][din] is already cached in L1 because of previous calculation, they are loaded from L1. For ii equals to csize-1, input data is loaded from main memory. Every 4 floats number is load into a cache block.

(2) What would be the resulting average number of cycles per load operation for the input data?
Answer: (16*192*1+192*49/4)/(17*192) = 1.7

(3) Describe the access pattern for the weights within the innermost two loops, in terms of which levels of memory the weights would reside in.
Answer: For each ii, every weight need to be loaded from main memory because both L1 and L2 cannot store the whole weight and old cache for weight will be invalidated because of the limit. Every 4 float number is load together into the cache block.

(4) What would be the resulting average number of cycles per load operation for the weights?
Answer: 49/4 = 12.3

(5) What is the CPIS for the function?
Answer: 12.3+1.7=14

5B: Parallel performance (10 points)

A. Determine the multicore performance of function `convolve_par` when running with T threads for $T \leq 8$.

(1) For a given thread, describe the access pattern for the input data within the innermost two loops, in terms of which levels of memory the data would reside in.

Answer: inputd data will be cached to L1 from main memory for the first iteration on dout. For the rest dout, input date will load from L1 cache.

(2) For a given thread, what would be the resulting average number of cycles per load operation for the input data?

Answer: (17*192/4*49+191*17*192*1)/(192*17*192)=1.1

(3) For all threads, describe the access pattern for the weights within the innermost two loops, in terms of which levels of memory the weights would reside in.

Answer: Each time weight need to be loaded from main memory. Each 4 four floats are cached together to L1.

(4) For a given thread, what would be the resulting average number of cycles per load operation for the weights?

Answer: 49/4 = 12.3

(5) What is the CPIS per thread for the function?

Answer: 12.3+1.1 = 13.4

(6) What is the speedup, relative to sequential execution, as a function of T?

Answer: 14*T/13.4 = 1.045T

B. Describe how you could restructure the order in which elements are accessed within the inner loop of the code to give the program superlinear speedup. You can assume that $T \in \{1, 2, 4, 8\}$. You may wish to make use of the values `thread_count` and `thread_id` that are computed within the parallel block. You should write code for the restructured loop.

Answer:

```
// inner loop
if (dout % thread_count == thread_id) {
    for (int tempi=0;tempi<dsize;tempi++){
        for (int din=0; din<input_depth;din++) {
            output[tempi][dout] = += weights[dout][ii][din]*input[tempi+ii][din];
        }
    }
}
```

C. What would be the resulting CPIS per thread for $T \in \{2, 4, 8\}$? Explain.

Answer:

D. What would be the speedup, relative to the sequential code for $T \in \{2, 4, 8\}$?

Answer:

Pledge

On the answer sheet, you must (electronically) sign the following pledge:

> I do hereby swear that, in generating my answers to this exam, I made use of only the resources explicitly listed in the exam document. I did not have any communication of any form with anyone other than the course instructors and teaching assistants about the contents of this exam.
>
> Signed
> Jiaqiang Ruan