COMP 35112

Two hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Chip Multiprocessors

Date: Friday 2nd June 2017

Time: 14:00 - 16:00

Please answer any THREE Questions from the FOUR Questions provided

---

This is a CLOSED book examination

The use of electronic calculators is permitted provided they are not programmable and do not store text

[PTO]

Note: Where a question asks for instruction-level code, a format similar to ARM assembler is expected. However, marks will not be lost if the format is incorrect, as long as the meaning of each instruction is clear (from an accompanying explanation and/or comments).

Question 1.

a) Explain the key distinctions between parallel programming models that are based on **data-sharing** and those based on **message-passing**.

(3 marks)

b) The recent advent of GP-GPU hardware architectures has led to a **hybrid** style of parallel programming model in which elements of both data-sharing and message-passing are evident. Describe, using appropriate diagrams, the nature of the underlying hardware architecture of GP-GPU computers and explain in detail how this has influenced the programming constructs available in languages such as CUDA or OpenCL.

(6 marks)

c) Consider the Java code fragment below, in which the work done in any call to method `do_work(i)` is known to be independent of the work done for a call using any other value of `i` (i.e. there are no loop carried dependences and all iterations of the loop could, in principle, be executed in parallel).

```
for (i=0 ; i < N ; i++)    // N is very large
    { do_work(i); }
```

(i) Given that the amount of work done per iteration of the loop is **constant**, describe the set of appropriate options for **scheduling** the iterations of the loop to threads in a parallel implementation looking to improve the performance of the code fragment.

(3 marks)

(ii) Now suppose that the amount of work done per iteration of the loop **varies** substantially, and that the amount of work done for any particular value of `i` cannot be predicted in advance of execution. Describe the new set of appropriate options for scheduling the iterations of the loop to threads in a parallel implementation looking to improve the performance of the code fragment. Explain why these new options are more expensive to use (in terms of execution time) than those in your answer to part c)(i).

(5 marks)

d) Why is 'pure' functional programming attractive as a means of programming chip multiprocessors? In what ways is it similar to or distinct from data-sharing or message-passing?

(3 marks)

Question 2.

a) Explain why cache coherence is a key feature in chip multiprocessor design.
(4 marks)

b) Assume a 'snooping bus' MSI cache coherence protocol in which a cache line can be in one of the three states: M (modified), S (shared), I (invalid). Explain briefly what each of these states means in terms of the collective state of the multiple caches.
(3 marks)

c) In the context of your answer to part b), explain briefly why it is additionally helpful to differentiate the following two states: (i) a cache line that would be in state S in exactly one core with every other core in the system holding that cache line in state I; and (ii) a modified cache line that has been copied by another core (which now holds this cache line in state S) without having been written back to memory. (2 marks)

d) Suppose the states described in part c) are denoted state E (exclusive) and state O (owner); the resulting protocol is termed MOESI. In a MOESI system with **three** cores, each core is executing the simple program shown below, which reads the shared variable x from memory, adds one to the value, then writes the new value back to the shared variable in memory. Assume that the execution is interleaved amongst the cores as shown with respect to any snooping bus transactions that may occur.

	core 1	**core 2**	**core 3**
	ldr r1, x		
		ldr r1, x	
	add r1, r1, #1		
		add r1, r1, #1	
	str r1, x		
			ldr r1, x
		str r1, x	
			add r1, r1, #1
			str r1, x

Assuming that the value of x is not cached in any of the cores at the start of the execution, list the cache states in each core for the cache line holding the value of x after each instruction has completed execution. Explain why these states exist.
(6 marks)

e) What are the main impediments to the scalability of a "snoopy bus"-based coherence protocol and how would using a directory-based protocol affect both the performance and scalability of cache coherence? For an MSI-like protocol, compare the details of the bus-based and directory-based protocols for a state change required by a write-miss in the cache of a core for a cache line currently in the shared state.
(5 marks)

[PTO]

Question 3.

a) Where multiple hardware units are available, it is possible to use them for **speculation**, thereby speeding up the execution of a sequential program. Explain the general principles of speculation, the constraints under which it may be applied, and the costs of using it.

(4 marks)

b) Explain what is meant by **thread level speculation** (TLS). Describe **two** ways in which threads for TLS may be generated automatically from a sequential program. Use pseudocode examples to illustrate your answer.

(6 marks)

c) What additional data structure resources (compared to those needed for sequential execution) are required to implement fully general **loop-based TLS**? How are these additional resources utilised at runtime?

(7 marks)

d) Briefly describe how multicore hardware might be modified so as to provide support that improves the performance of any activity identified in your answer to part c).

(3 marks)

Question 4

a) For an Unbounded Queue implemented in Java as a linked-list with a dummy "sentinel" node, describe, using suitable pseudocode and diagrams, the expected correct behaviour of an enqueue method which adds a new message at the tail of a queue (`enQ(msg)`) and a dequeue method which returns a message from the head of the queue (`msg□deQ()`). How many references does each method have to update?

(2 marks)

b) Give an example, with the aid of diagrams and/or pseudocode, of the problems that might be encountered when two threads attempt to enqueue (enQ()) to the same unbounded queue "at the same time" if no synchronisation is implemented in the enqueue method.

(3 marks)

c) Give example code for the enQ() method using Java locks and explain briefly how the problem described in your answer to part b) is avoided.

(3 marks)

d) Give a brief explanation of what is meant by a "lock-free" data structure and, in terms of expected performance and code complexity, briefly compare Java classes implemented as lock-free data structures with equivalent classes using Java synchronization techniques (i.e. synchronized methods/blocks or Java locks).

(3 marks)

e) The following code is an implementation of a lock-free unbounded queue using Java AtomicReferences and the compareAndSet instruction. For the problem of two threads attempting to enqueue at the same time, introduced in part b) of this question, explain how the problem described in your answer to part b) is overcome in this lock-free implementation. Use diagrams and pseudocode to support your explanation, as necessary.

```
class Node {
   private Object value ;
   private AtomicReference<Node> next ;

   public Node(Object newValue) {
      value = newValue ;
      next = new AtomicReference<Node>(null) ;
   }
}
```

(code continued on next page)

```
class UbQueue {
   private AtomicReference<Node> head, tail ;

   public UbQueue() {
      head.set(new Node(null)) ; // sentinel
      tail.set(head) ;
   }

   public enQ(Object item) {
      Node n = new Node(item) ;

      while (true) {
         Node last = tail.get() ;
         Node afterLast = last.next.get();
         if (last == tail.get()) {
            if (afterLast == null) {
               if (last.next.compareAndSet(null, n){
                  tail.compareAndSet(last, n) ;
                  return ;
               }
            } else { // afterLast wasn't null!
               tail.compareAndSet(last, afterLast);
            }
         } // no else if last not equal to tail
      } // end of while loop

   } // end of enQ
```

(code continued on next page)

```
   public Object deQ() throws EmptyException {

      while(true) {
         Node first = head.get() ;
         Node last = tail.get() ;
         Node second = first.next.get() ;

         if (first == head.get()) {
            if (first == last) {
               if (second == null) {
                  throw new EmptyException() ; // return on
          empty
               }

               tail.compareAndSet(last, second);
            } else {
               Object ans = second.value ;
               if (head.compareAndSet(first,second))
                  return value;
            }
         }
      } // end of while loop

   } // end of edQ

} // end of UbQueue
```

(7 marks)

f) It is possible for another thread to attempt a dequeue operation on the queue while the two enqueue operations are still in progress. Explain how this dequeue operation might help all threads to make progress.

(2 marks)

END OF EXAMINATION