University of St Andrews

MAY 2021

48 HOUR ASSESSMENT

SCHOOL OF COMPUTER SCIENCE

MODULE CODE: **CS4204**

MODULE TITLE: Concurrency and Multicore Programming

TIME TO HAND IN: 48 hours

EXAM INSTRUCTIONS:
a. Answer both questions.
b. Each question indicates the number of marks it carries. The paper carries a total of 60 marks.

This assessment consists of exam-style questions and you should answer as you would in an exam. As such, citations of sources are not expected, but your answers should be from your own memory and understanding and significant stretches of text should not be taken verbatim from sources. Any illustrations or diagrams you include should be original (hand or computer drawn). You may word-process your answers, or hand-write and scan them. In either case, please return your answers as a single PDF. If you handwrite, please make sure the pages are legible, the right way up and in the right order. Your submission should be your own unaided work. While you are encouraged to work with your peers to understand the content of the course while revising, once you have seen the questions you should avoid any further discussion until you have submitted your results. You must submit your completed assessment on MMS within 48 hours of it being sent to you. Assuming you have revised the module contents beforehand, answering the questions should take no more than three hours.

In questions where you have to write code, minor syntax problems will not be penalised, as long as it is clear what you mean. It is a written exam, we do not expect to run your code.

1. (a) State Amdahl's law and show how it is derived.

[2 marks]

(b) Suppose that the speedup obtained over the sequential case by using 2 processors instead of 1 is S_2. Assuming the conditions of Amdahl's law hold, what is the expected speedup S_4 obtained by using 4 similar processors over the sequential case? Express your answer in terms of S_2.

[4 marks]

(c) Consider the following program, on three threads, operating on two shared-memory locations `X` and `Y`. (The location `r` is thread-local, which you can consider to be a register if you wish):

Initially: X = 0; Y = 0;		
Thread 0	Thread 1	Thread 2

`X = 1;`	`while (X != 1) {/* SPIN */}`
	
	`Y = 1;`
		`while (Y != 1) {/* SPIN */}`
		
		`r = X;`

What observations of values are possible in the thread-local location `r`, under Sequential Consistency? What observations are possible under X86 memory consistency model (X86-TSO)? Justify your answers in each case.

[6 marks]

(d) What observations are possible under the ARM memory consistency model, for the same program as in Question 1(c)? Justify in each case.

[2 marks]

(e) Using the ARM memory consistency model, how would you forbid all the observations of the program in Question 1(c) that are impossible under Sequential Consistency?

[2 marks]

(f) We wish to implement an account data structure, which holds a value (assume integer). This should support the following three operations:

- `getvalue(Account a)` – returns the current value in the account `a`;
- `deposit(Account a, int d)` – should add `d` to the value held in account `a`;
- `withdraw(Account a,int w)` – should subtract `w` from the value held in account `a`, provided the account value was greater than or equal to `w` to begin with.

Provide an implementation using a mutex or similar construct.

[6 marks]

(g) Prove that your implementation from Part (f) is linearisable.

[4 marks]

(h) Provide a lock-free implementation of the account data structure, assuming the availability of a compare-and-swap or similar operation. Give a justification for why your implementation is lock-free.

[4 marks]

[Total marks 30]

2. (a) List the benefits of using a parallel pattern approach over using threads directly.

[2 marks]

(b) OpenMP and TBB differ in that OpenMP is pragma based and TBB is library based. What are the advantages and disadvantages of each?

[2 marks]

(c) Sketch a functional semantics for the reduce pattern, showing the functional operation, the types of operations and data, and how the parallelism arises.

[3 marks]

(d) For the following code, and using the cost models from the lectures, investigate which pattern instantiation gives the best speedups. Show your working. You can assume both `filter` and `geoRef` are pure functions that do not emit side-effects. You can assume a copy overhead of $2ms$ per task, `filter` takes on average $50ms$ and `geoRef` takes on average $5ms$. You can assume an 8-core machine.

```
for (int i = 0 ; i < 100 ; i++)
        filter(geoRef(Image[i]));
```

[4 marks]

(e) For a GPU, explain each of the following CUDA terms:

- `threadIDx.x`
- `blockIDx.x`
- `blockDim.x`

[3 marks]

(f) For the following code snippet, use OpenMP directives to parallelise `histogram()`. Use the following OpenMP directives:

- omp for
- omp task

```
//A a vector that we'll compute the histogram of.
//Assume that each value A[i] is in the range [0, 1].
//N length of A. H histogram output
//B number of bins in histogram H. (initialized to zeros)
void histogram(float* A, int N, float* H, int B) {
   for (int i = 0; i < N; i++)
      for (int j = 0; j < B; j++) {
         float bin_begin = float(j)/B;
         if (A[i] <= bin_begin)
            H[j] = H[j] + A[i];
      }
}
```

[4 marks]

(g) Given the following code example, parallelise the example using OpenMP, removing the dependency in the process.

```
for(i=0;i< N-1; i++) {
        x = b[i] + c[i];
        a[i] = a[i+1] + x;
}
```

[4 marks]

(h) Given the following code example, rewrite it so that it uses a single parallel pipeline pattern with four stages using GrPPI instead of pthreads. In your answer you may wish to think of how you will remove the pthreading code, how you will modify the code so that `Worker` behaves like a pipeline stage, how you will introduce a single point that calls a GrPPI pipeline and how you might need to replicate `Worker` for the multiple stages of the pipeline.

```
...
#define SHARED 1

void *Worker(void *);
void InitializeData();
void Barrier();

pthread_mutex_t barrier;  /* mutex semaphore for the barrier */
pthread_cond_t go;        /* condition variable for leaving */
int numArrived = 0;       /* count of the number who have arrived */

int dataSize, numWorkers;

int data[100];
```

// (continued on next page...)

```
int main(int argc, char *argv[]) {
   /* thread ids and attributes */

   pthread_t workerid[4];
   pthread_attr_t attr;
   long i;
   FILE *results;

   /* set global thread attributes */
   pthread_attr_init(&attr);
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* initialize mutex and condition variable */
   pthread_mutex_init(&barrier, NULL);
   pthread_cond_init(&go, NULL);

   /* read command line and initialize data */
   dataSize   = 100;
   numWorkers = 4;
   InitializeData();

   /* create the workers, then wait for them to finish */
   for (i = 0; i < numWorkers; i++)
      pthread_create(&workerid[i], &attr, Worker, (void *) i);
   for (i = 0; i < numWorkers; i++)
      pthread_join(workerid[i], NULL);

   for (i = 0; i < dataSize; i++) {
      fprintf(results, "%d ", data[i]);
   }
   fprintf(results, "\n");
}
```

// (continued on next page...)

```
void *Worker(void *arg) {
   long myid = (long) arg;
   int i;
   int mystate = myid;

   for (i=0; i<myid; i++) {
      Barrier();
   }

   // process items one by one
   for (i=0; i<dataSize; i++) {
      mystate += data[i]+1;
      data[i] = data[i]+mystate;
      Barrier();
   }

   for (i=0; i<numWorkers-myid-1; i++) {
      Barrier();
   }
}

void InitializeData() {
   int i;
   for (i = 0; i < dataSize; i++) {
      data[i] = 0;
   }
}

void Barrier() {
   pthread_mutex_lock(&barrier);
   numArrived++;
   if (numArrived == numWorkers) {
      numArrived = 0;
      pthread_cond_broadcast(&go);
   } else
      pthread_cond_wait(&go, &barrier);
   pthread_mutex_unlock(&barrier);
}
```

[8 marks]

[Total marks 30]

***** END OF PAPER *****