[20250415] INFOB3CC - Concurrency - 2(H) - USP

Course: BETA-INFOB3CC Concurrency (INFOB3CC)

Duration:	2 hours
Number of questions:	6
Generated on:	Jun 25, 2025

Contents: Pages:

[20250415] INFOB3CC - Concurrency - 2(H) - USP

Course: Concurrency (INFOB3CC)

- The exam is a closed book exam.
- The exam must be made alone. No communication with others is allowed.
- Provide brief and concise answers. Overly verbose responses or nonsense added to otherwise good answers can deduct from your grade.
- When asked to explain your choice on a multiple choice question, your reasoning should explain why your chosen answer is correct and why the others are not correct.
- If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
- You may assume that threads do not crash.
- You have two hours to complete the exam. You can go back to previous questions.
- *Good luck! (:*

Number of questions: 6

You can score a total of 39 points for this exam, you need 22.28 points to pass the exam.

1 Besides atomic compare-and-swap, many processors also feature an atomic fetch-or function. Given a pointer (memory location or IORef) and a value, it atomically loads the current value in that location, computes *bitwise or* of that value and the given value, writes that computed value to the location, and returns the old value. It is thus an atomic version of the following function:

```
int fetch_or(int *variable, int arg) {
  int old_value = *variable; // Read the old value
  *variable = old_value | arg; // Write the new value to the variable
  return old_value;
}
```

Alternatively, it is the atomic equivalent of the following Haskell code:

```
fetchOr :: IORef Int -> Int -> IO Int
fetchOr ref arg = do
  old <- readIORef ref
  writeIORef ref (old .|. arg)
  return old
```

Bitwise or computes the logical or per bit of a number. For example, 0b1100 | 0b1010 = 0b1110.

Recall that atomic compare-and-swap (*atomic_compare_exchange*, or *casIORef* in Haskell) has the following signature:

```
struct Result { bool success; int original; }
Result atomic_compare_exchange(int *variable, int expected, int
replacement);
```

It is possible to implement a lock using *atomic fetch-or* (without atomic compare-and-swap). Similar to the CAS lock, this only requires a single variable as shared state. We call this variable *state*.

In the following questions, you should give an implementation of this lock.

1.5 pt. **a.** What value(s) of *state* should denote that the lock is taken, and which should denote that the lock is free?

Choose the values such that the lock can be implemented using atomic fetch-or.

2 pt. **b.** Implement a function to acquire the lock. You may give code either in C or in Haskell. Use *atomic_fetch_or* to denote the atomic fetch-or operation.

1.5 pt. **c.** Implement a function to release the lock. You may give code either in C or in Haskell.

3 pt. **d.** While many processors support *atomic_fetch_or* natively, there exist some processors that do not. Fortunately, *atomic_fetch_or* can be implemented as a function in terms of *atomic_compare_exchange*.

Implement *atomic_fetch_or* in terms of *atomic_compare_exchange (or casIORef).*

2 In this function, we consider various functions in Haskell to change the value of a mutable variable.

First, consider the following function to change the value of an IORef:

```
changeIORef :: IORef Int -> IO ()
changeIORef ref = do
  currentValue <- readIORef ref
  let newValue = currentValue * 2 + 1
  writeIORef ref newValue
```

Assume that the IORef is only accessed via changeIORef. It is accessed from multiple threads.

2 pt. **a.** Which properties hold for this implementation?

- **a.** It performs the change atomically
- **b.** It is wait-free
- **c.** It is deadlock-free
- **d.** It is starvation-free

Now consider an implementation using MVars:

```
changeMVar :: MVar Int -> IO ()
changeMVar var = do
  currentValue <- takeMVar var
  let newValue = currentValue * 2 + 1
  putMVar var newValue
```

Again, you may assume that the MVar is only accessed via this function and that it is filled at the beginning of the program. It is accessed from multiple threads.

2 pt. **b.** Which properties hold for this implementation?

- **a.** It performs the change atomically
- **b.** It is wait-free
- **c.** It is deadlock-free
- **d.** It is starvation-free

Finally, consider an implementation using TVars:

```
changeTVar :: TVar Int -> IO ()
changeTVar var = atomically $ do
  currentValue <- readTVar var
  let newValue = currentValue * 2 + 1
  writeTVar var newValue
```

You may again assume that the TVar is only accessed via this function, from multiple threads.

2 pt. **c.** Which properties hold for this implementation?

a. It performs the change atomically

b. It is wait-free

c. It is deadlock-free

d. It is starvation-free

3

2 pt. **a.** Give the thread hierarchy on a GPU, i.e. the relations between threads, thread blocks, grids and warps on a GPU.

2 pt. **b.** Why is it important to consider this thread hierarchy when implementing performant algorithms (for instance a scan) on a GPU?

3 pt. **c.** How do threads, SIMD and simultaneous multithreading on a CPU relate to the thread hierarchy on a GPU?

4 A scan can be executed sequentially with one (memory) read and one (memory) write per element, in linear time.

1 pt. **a.** Can a three-phase parallel scan be implemented with only one read and one write per element?

a. Yes

b. No

1 pt. **b.** Is reduce-then-scan (a three-phase parallel scan) *efficient*?

a. Yes

b. No

1 pt. **c.** Is reduce-then-scan (a three-phase parallel scan) *optimal*?

a. Yes

b. No

2 pt. **d.** What property (or properties) for a givenoperator $\oplus$ of a scan is (or are) absolutely required to evaluate it efficiently in parallel?

a. $\forall x\ y.\ x \oplus y = y \oplus x$

b. $\forall x.\ x \oplus x = x$

c. $\forall x\ y.\ x \oplus y = (x \oplus y) \oplus y$

d. $\forall x\ y\ z.\ x \oplus (y \oplus z) = (x \oplus y) \oplus z$

1 pt. **e.** The input of a scanl, with operator +, can be reconstructed from its output. Which parallel pattern should be used to convert the output of a scanl to its corresponding input?

a. map

b. fold

c. scanl

d. scanr

e. stencil

f. permute

g. backpermute

5 Consider the following algorithm, which performs a *parallel* map over the input array and three recursive calls of arrays of half the size:

```
procedure erumpent(ps) {
  n = length(ps)

  if n == 0 { return 0 }
  if n == 1 { return ps[0] }

  qs = map (\p -> p * n + 1) ps

  a = erumpent(slice(qs, 0, n * 0.5))
  b = erumpent(slice(qs, n * 0.25, n * 0.75))
  c = erumpent(slice(qs, n * 0.5, n * 1))

  return a + b + c
}
```

The three recursive calls are executed in parallel.

The array length and slice (view the subarray between two indices, without copying) functions can be executed in constant time.

2 pt. **a.** What is the asymptotic span of this algorithm? Show how you computed the span.

2 pt. **b.** What is the asymptotic work of this algorithm? Show how you computed the work.

6 In this question, we model the growth of a new fungus-type organism. This organism turns energy into body mass efficiently and thus grows rapidly. This growth is modeled on a 2-dimensional grid and simulated with small time steps. If some cell is occupied, then all 8 adjacent cells will also be occupied in the next timestep. The organism also stays present in the current cells.

Time t=1	Time t=2	Time t=3
□□□□□□□ □□□□□□□ □□■□□■□□ □□□■□□□□ □□□□□□□ □□□□□□□	□□□□□□□ □■■■■■□ □■■■■■□ □■■■■■□ □■■■■■□ □□■■■□□ □□□□□□□	■■■■■■■ ■■■■■■■ ■■■■■■■ ■■■■■■■ ■■■■■■■ □■■■■■□

The grid is stored as a two-dimensional array of booleans.

You are asked to choose the most performant option to implement several operations for this problem using data parallel patterns / combinators.

1 pt. **a.** With which parallel pattern (or combinator) can a single time step be simulated?

a. map

b. stencil

c. fold

d. scan

e. permute

f. backpermute

g. zip

2 pt. **b.** After simulation a certain number of time steps, we need to know the number of populated cells. This can be computed in two steps. Which parallel patterns should be used in those two steps?

		map	stencil	fold	scan	permute	backpermute	zip
		A	**B**	**C**	**D**	**E**	**F**	**G**
Step 1	**1**	○	○	○	○	○	○	○
Step 2	**2**	○	○	○	○	○	○	○

After some iterations, the shape of the organism will be convex. For this question, it is enough to think of this as a rectangle with only some cells at the borders missing. The size of the organism in later steps of the simulation can now be computed directly: each step will grow the rectangle by 1 on all sides, with the same number of cells missing in each step. For instance, in this example you can see that there always 6 cells missing from the rectangle:

Time t = 1	Time t = 2	Time t = 3
■□□ □■□ □□■	■■■□□ ■■■■□ ■■■■■ □■■■■ □□■■■	■■■■■□□ ■■■■■■□ ■■■■■■■ ■■■■■■■ ■■■■■■■ □■■■■■■ □□■■■■■

2 pt. **c.** We must now detect whether the shape is already convex. This can be done by counting for each row and column, the number of transitions from empty to filled (and filled to empty) cells.

Given a boolean array corresponding to a *single row* of the grid, how can we compute the number of transitions from empty to full?

		stencil	fold	scan	permute	backpermute	zip
		A	**B**	**C**	**D**	**E**	**F**
Step 1	**1**	○	○	○	○	○	○
Step 2	**2**	○	○	○	○	○	○

3 pt. **d.** Besides the boolean grid, we are now also given a matrix of temperatures. We want to perform some analysis over the temperatures of the filled cells. Hence we need to convert the matrix of temperatures to a vector (1-dimensional array) containing only the temperature values of filled cells.

How can that vector be constructed?

		map	stencil	fold	scan	permute	backpermute	zip
		A	**B**	**C**	**D**	**E**	**F**	**G**
Step 1	**1**	○	○	○	○	○	○	○
Step 2	**2**	○	○	○	○	○	○	○
Step 3	**3**	○	○	○	○	○	○	○

Name:				Signature:
Date:	/ /	Date of birth:	/ /	

Course: BETA-INFOB3CC Concurrency (INFOB3CC) - Questions: [20250415] INFOB3CC - Concurrency - 2(H) - USP

- The exam is a closed book exam.
- The exam must be made alone. No communication with others is allowed.
- Provide brief and concise answers. Overly verbose responses or nonsense added to otherwise good answers can deduct from your grade.
- When asked to explain your choice on a multiple choice question, your reasoning should explain why your chosen answer is correct and why the others are not correct.
- If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
- You may assume that threads do not crash.
- You have two hours to complete the exam. You can go back to previous questions.
- *Good luck! (:*

1
8 pt.

a.

Answer:

b.

Answer:

c.

Answer:

d. Answer:

2
6 pt.

a. A ☐ B ☐ C ☐ D ☐

There are multiple answers possible

b. A ☐ B ☐ C ☐ D ☐

There are multiple answers possible

c. A ☐ B ☐ C ☐ D ☐

There are multiple answers possible

3
7 pt.

a.

Answer:

b.

Answer:

c.

Answer:

4
6 pt.

a. A ○ B ○

b. A ○ B ○

c. A ○ B ○

d. A ☐ B ☐ C ☐ D ☐

There are multiple answers possible

e. A ○ B ○ C ○ D ○ E ○ F ○ G ○

5
4 pt.

a.

Answer:

2

4

6

8

10

b.

Answer:

2

4

6

8

10

6
8 pt.

a. A ○ B ○ C ○ D ○ E ○ F ○ G ○

b.

	A	B	C	D	E	F	G
1:	○	○	○	○	○	○	○
2:	○	○	○	○	○	○	○

Choose one option per row

c.

	A	B	C	D	E	F
1:	○	○	○	○	○	○
2:	○	○	○	○	○	○

Choose one option per row

d.

	A	B	C	D	E	F	G
1:	○	○	○	○	○	○	○
2:	○	○	○	○	○	○	○
3:	○	○	○	○	○	○	○

Choose one option per row

Elaboration of the answer

1.
8 pt.

a.

Correction criterion	Points
1 = locked, 0 = unlocked	0 to 1.5 points
Total points:	*1.5 points*

b.

Correction criterion	Points
while (	0 to 1 points
atomic_fetch_or(&state, 1) == 1) { /* spin */ }	0 to 1 points
Total points:	*2 points*

c.

Correction criterion	Points
state = 0	0 to 1.5 points
Total points:	*1.5 points*

d.

Correction criterion	Points
current = *variable OR ticket <- readForCAS ref	0 to 1 points
atomic_compare_exchange(variable, current, current | new_value) OR casIORef ref ticket (peekTicket ticket .|. new)	0 to 1 points
Entire code is in a loop. if (success) { return current; } OR if success then return (peekTicket ticket) else atomicSwap ref new (or with a local function that loops, which takes a ticket as argument)	0 to 1 points
Total points:	*3 points*

2. 6 pt.

a. A

1 pt. B

1 pt. C

1 pt. D

Bonus: 0 pt.

b. 1 pt. A

B

1 pt. C

1 pt. D

Bonus: 0 pt.

c. 1 pt. A

B

1 pt. C

D

Bonus: 0 pt.

3.
7 pt.

a.

Correction criterion	Points
Grid is the entire execution of a kernel	0 to 0.5 points
Grid contains multiple thread blocks (that may execute independently)	0 to 0.5 points
Thread blocks contain warps	0 to 0.5 points
Warps contain threads	0 to 0.5 points
Total points:	*2 points*

b.

Correction criterion	Points
For communication: communication in a warp is cheapest, within a thread block somewhat cheap (via shared memory), and within a grid is most expensive (via global memory). OR: For synchronization: threads in a warp are synchronized via lock step execution, threads within a thread block can manually synchronize, threads within a grid typically cannot synchronize.	0 to 2 points
Total points:	*2 points*

c.

Correction criterion	Points
CPU thread ~ GPU warp	0 to 1 points
GPU warp executes SIMD instructions. OR: CPU SIMD lane ~ GPU thread	0 to 1 points
Simultaneous multithread on a CPU is similar to how a GPU switches between warps (when a warp has a latency)	0 to 1 points
Total points:	*3 points*

4.
6 pt.

a. 1 pt. B

b. 1 pt. A

c. 1 pt. A

d. A
B
C
1 pt. D
Bonus: 0 pt.

e. 1 pt. E

5. 4 pt.

a.

Correction criterion	**Points**
f(n) = 1 (map is parallel)	0 to 0.5 points
T(n) = T(n/2) + 1	0 to 0.5 points
Case 2 OR: f and recursive calls have similar contribution	0 to 0.5 points
Span is T(n) = O(log n)	0 to 0.5 points
Total points:	*2 points*

b.

Correction criterion	**Points**
f(n) = n	0 to 0.5 points
T(n) = 3T(n/2) + n	0 to 0.5 points
Case 1 OR: recursive calls dominate over f	0 to 0.5 points
Span is T(n) = O(n^(log_2(3)) ~ O(n^1.585)	0 to 0.5 points
Total points:	*2 points*

6. 8 pt.

a. 1 pt. B

b.
1. 1 pt. A
2. 1 pt. C

c.
1. 1 pt. A
2. 1 pt. B

d.
1. 1 pt. A
2. 1 pt. D
3. 1 pt. E

Caesura

Applied guessing score: 5.567 pt

Points scored	Grade
39	10
38	9.73
37	9.46
36	9.19
35	8.92
34	8.65
33	8.38
32	8.12
31	7.85
30	7.58
29	7.31
28	7.04
27	6.77
26	6.50
25	6.23
24	5.96
23	5.69
22	5.42
21	5.15
20	4.89
19	4.62
18	4.35
17	4.08
16	3.81
15	3.54
14	3.27
13	3.00
12	2.73
11	2.46
10	2.19
9	1.92
8	1.65
7	1.39
6	1.12
5	1.00
4	1.00
3	1.00
2	1.00
1	1.00
0	1.00

Question identifiers

These identifiers can be used to track the exact origin of the question. Use these identifiers together with the identifier of this document when sending in comments about the questions, so that your comment can be connected precisely with the question you are referring to.

Document identifier: 68313-116294

Question number	Question identifier	Version identifier
1	655659	7cbb1656-6a37-4acc-ab06-9db73b5b7087
2	627520	8e8089e6-6270-42ed-8494-e2545dd77005
3	656078	996ed5fe-c158-4621-a1ec-71fdfa95be06
4	521784	eba79ada-e0c9-46e7-9f47-e8929310c22f
5	520484	3f1bc78e-8f30-4697-9bcb-3b366e734a8f
6	656082	eb3ba6cf-0277-4f21-bb89-aeda107c7ae4