NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

CS3211 – Parallel and Concurrent Programming

(Semester 2: AY2021/22)

Final Assessment Paper – Marking Scheme

Time Allowed: 2 hours

INSTRUCTIONS TO STUDENTS

1. Write your Student Number only. Do not write your name.
2. This assessment paper contains **FOUR** questions and comprises **FOURTEEN** printed pages. Additionally, there is an appendix comprising **TWO** pages.
3. Students are required to answer **ALL** questions within the space in this booklet.
4. This is an OPEN BOOK assessment. Calculators are allowed.
5. Feel free to use pencil for answering the questions.
6. Write your Student Number below.

STUDENT NO: _____

This portion is for examiner's use only

Question	Marks	Remarks
Q1	/26	
Q2	/10	
Q3	/32	
Q4	/12	
Total	/80	

Q1. [26 marks] General Questions

a. [4 marks] Relaxed consistency models at the programming language level address the concern that software complexity is increased by the need for unnecessary synchronization.
Is this statement true or false? Justify your answer.

Answer:

[4 marks]

- False, relaxed consistency models are harder to reason about
- False, relaxed actually needs more synchro by programmer
- False, relaxed consistency helps performance

[3 marks]

- True, relaxed consistency helps performance (misunderstanding software complexity)
- True, can prevent data races

[2 marks]

- False, relaxed consistency does not make software less complex
- False, irrelevant justification

b. [4 marks] The modern C++ memory model exhibits a different behavior when it is running on a weak-consistent architecture (such as ARM). Is this statement true or false? Justify your answer.

Answer:

[4 marks]

- False, C++ memory model provides the same guarantees / "contract" across architectures
- False, C++ memory model has/provides the same behavior across architectures
- False, C++ memory model designed to be architecture-agnostic
- False, C++ memory model / compiler abstracts away differences in architecture

[3 marks]

- True, different compiler / different implementation of atomics / different performance due to extra synchro

[2 marks]

- False, architecture must adhere to the memory model
- False, irrelevant / tautological / nonsense justification

c. [4 marks] Programs using C++ atomics might suffer of deadlocks. Assume that the program does not use any other synchronization mechanism (such as mutex or semaphores). Is this statement true or false? Justify your answer.

Answer:

[4 marks]

- (expected answer) True, deadlock can happen due to busywaiting / mutex / spinlock using atomics
- False, at least one thread is guaranteed to make progress
- False, atomics cannot result in the 4 conditions for deadlock

[2 marks]

- False, concurrent access without explicit synchronization for atomics is defined
- False, irrelevant / tautological / nonsense justification

d. [4 marks] Explain one advantage and one disadvantage of using channels (message passing) instead of synchronization primitives and shared memory for a program running on one computer. You may refer to Go channels in your explanation.

Answer:

[+ 1 mark]

- +: No race conditions (but explained like data races)
- +: Channels do synchro. and data transfer at the same time
- +: No additional data structures for synchro (but the channel itself is a data structure...)
- +: Avoids contention over shared memory (channel buffer is still ultimately shared memory)
- +: Channels are buffered i.e. no blocking to send an update, vs a shared memory data structure with a mutex (but you could just design your shared memory such that it behaves similarly..)
- +: Channels are guaranteed FIFO (you could design shared memory like that too)
- -: Shared memory is more intuitive for shared objects (very subjective)

[+2 marks]

- +: Allows for easy distributed systems
- +: Data is simply protected by confinement (rather than requiring manual synchro)
- +: Channels are easier to reason about than shared memory / higher level / CSP
- +: Channels result in clear ownership / separation of concerns
- +: No data races / safe(r)
- -: No RAII i.e. need to explicitly send message to unlock something etc
- -: More tedious to do some operations (e.g. updating a complex data structure)
- -: Lower performance / lose locality
- -: No way for multiple goroutines / threads to concurrently access data over a channel (serialized by channel / owner goroutine)

e. [4 marks] In (safe) Rust, data races are not possible. This means that your Rust code will not have race conditions. Is this statement true or false? Justify your answer.

Answer:

[4 marks]

- False, timing / ordering of events can still affect correctness

[3 marks]

- False, *other types* of race conditions may occur / there can still be races due to shared memory (without explicit reference to ordering of events)
- False, race conditions can still happen with *incorrect synchro*

[2 marks]

- False, irrelevant / tautological / nonsense justification

f. [6 marks] In this world of multiple cores, cloud computing, web scale, and problems that may or may not be parallelizable, we find the modern developer a bit overwhelmed. In 2005, Herb Sutter authored an article titled "The free lunch is over: A fundamental turn toward concurrency in software". Toward the end, Sutter states: "We desperately need a higher-level programming model for concurrency than languages offer today."

Based on the concepts studied in CS3211, answer and justify your answer for the following two questions:

- What is one of benefits that use of concurrency brings nowadays? (2 marks)
- Do you think that the models for concurrency as implemented by various programming languages have progressed enough since 2005 to enable the modern developer to take advantage benefits of concurrency? (4 marks)

Answer:

[+3/4 marks]

- Answer "did the models progress enough?" with good justification - Any answer was accepted:
  - Yes, the progress is enough because now we have memory models and ownership/borrowing to help us predict the behavior of a program
  - Yes, good scaling on hardware
  - No, because the models are still complex; deadlocks, livelocks, races, etc.

[+2 marks]

- Benefits of concurrency are explained.
  - Concurrency helps us take advantage of the hardware

Q2. [10 marks] C++ Memory Model

a. [5 marks] Consider 2 threads in C++ pseudo-code. There are 2 `std::atomic` variables, `x` and `y` initialized to `0` before the threads have been created.

Thread 1	**Thread 2**
`x.store(1, memory_order_release);` `y.store(2, memory_order_relaxed);`	`while (y.load(memory_order_relaxed)!=2);` `cout << x.load(memory_order_acquire);`

Are the following statements TRUE or FALSE? Optionally, you may justify your answers.

Answer:

Statement	True/False
1) The code will always print 1.	F
2) The code might run into an infinite loop.	F
3) The code will never print 0.	F
4) The code will print 1 or a value that is assigned to `x` after 1.	F (T was accepted if 5 was T)
5) The code will print 0, 1, or a value that is assigned to `x` after 1.	T

Justification (optional):

a. [5 marks] Consider 3 threads in C++ pseudo-code. There are 3 `std::atomic` variables, `x`, `y`, and `z` initialized to `0` before the threads have been created.

Thread 1	**Thread 2**

`x.store(3, memory_order_relaxed);`
`z.store(4, memory_order_release);`
`x.store(5, memory_order_relaxed);`
	`while (y.load(memory_order_acquire)!=2);`
	`while (z.load(memory_order_acquire)!=4);`
	`cout << x.load(memory_order_relaxed);`

Thread 3	

`while (z.load(memory_order_acquire)!=4);`
`x.store(1, memory_order_relaxed);`
`y.store(2, memory_order_release);`	

Are the following statements TRUE or FALSE? Optionally, you may justify your answers.

Statement	True/False
1) The code will print 1, 5, or a value that is assigned to `x` after 1 or 5.	T
2) The code will never print 1.	F
3) The code will never print 3.	T
4) The code will never print 5.	F
5) The code might run into an infinite loop.	F

Justification (optional):

Q3. [32 marks] Concurrent Programming Paradigms

In answering points a.-e. use the problem description of the **N-body simulation from Appendix.**

a. [6 marks] Identify at least **two** parts of the **N-body simulation** presented in the Appendix (page 15) that could be implemented concurrently.
Focus on finding parts that would benefit from a concurrent implementation. For each part that you would implement concurrently, briefly explain:
    - What is the benefit, if any, of this new concurrent implementation?
    - How would you quantify the potential benefits of these new implementations?
    - What are the potential disadvantages, if any, of the new implementation?

Answer:

Advantages:

- [2 marks] Run in parallel

Disadvantages:

- [2 marks] Coding complexity
- [2 marks] Context switching
- [2 marks] Increased memory usage

[2 marks] Quantify by measuring the execution time

b. [8 marks] You are required to implement concurrently, in **modern C++**, the N-body simulation presented in the Appendix (page 15). Your new implementation should aim to **bring the highest benefits from concurrency**.
Sketch your implementation in modern C++ and explain the following points:
    - Which part(s) of the N-body simulation are you implementing concurrently? You may refer to point a. and choose one of the parts presented there.
    - What shared data structures are you using (if any)?
    - How do you ensure that access to these data structures would not produce race conditions or any other types of synchronization problem?
    - What would be the maximum level of concurrency that you expect in your implementation? (i.e. number of threads that could potentially run in parallel)

    You can use the line numbers to refer to the lines of pseudo-code in Appendix. Do not copy over the lines that stay the same. Minor mistakes in C++ syntax do not bring marks deductions.
    In this answer, you are not allowed to use the quad-tree structure presented in Q4.

Answer:

[+6 marks]

- One thread to handle a chunk of bodies.
- Use a thread pool.
- One thread for each iteration for one of the loops: #3, #18, or #25.
- Pipeline

[-3 marks]

- No C++ pseudo-code, and implementation is not clear

[-2 marks]

- Threads creation is not shown (explained)
- Implementation contains data race
- Forgot major detail in algorithm

[-1 mark]

- Implementation deadlocks
- unnecessary locking
- Clearly not familiar with threading API
- Approach does not bring the highest benefit from concurrency.

[+1 mark]

- Maximum level of concurrency is well explained.
- Shared data structures are explained and protected.

c. [4 marks] Assume your concurrent implementation of the N-body simulation (Appendix) from point b. does not run as expected (it blocks, crashes, etc.). What the steps that you would take to identify the issues and fix your implementation?
Make your answer specific to your concurrent N-body simulation. General answers will not receive full credit.

Answer:

[+2 marks]

- Run both sequential and concurrent implementations, compare results
- Incrementally remove concurrency
- Reduce problem size to ease debugging
- Run single-threaded first
- Use Valgrind/TSan/Asan, gdb
- uses a "possible bug"-centric methodology

d. [8 marks] You are required to implement concurrently, in **Go**, the N-body simulation presented in the Appendix (page 15). Your new implementation should aim to **bring the highest benefits from concurrency by using the message passing paradigm**. As such, your Go implementation should use goroutines, channels, and channel peripherals like `select`. You may use read-only shared data structures, but you are not allowed to use any utilities from `sync` package in Go (mutex, condition variable, pools are not allowed).

Sketch your implementation in Go and explain the following points:

- Which part(s) of the N-body simulation are you implementing concurrently? You may refer to point a. and choose one of the parts presented there.
- Briefly describe the jobs (tasks) done concurrently by the goroutines and their interaction using channels.
- What would be the maximum level of concurrency that you expect in your implementation? (i.e. number of tasks that could potentially run in parallel)

You can use the line numbers to refer to the lines of pseudo-code in Appendix. Do not copy over the lines that stay the same. Minor mistakes in Go syntax do not bring marks deductions.

In this answer, you are not allowed to use the quad-tree structure presented in Q4.

Answer:

[8 marks]

- One goroutine for each body at line #18 or #25 or #20 or #3 (or a combination)

[-2 marks]

- Concurrency level not explained (correctly)
- Inefficient use of concurrency.

[-1 mark]

- No Go pseudo-code, just explanation is used
- Usage of sync package

e. [6 marks] You are required to design your **Rust** concurrent implementation of the N-body simulation presented in the Appendix (page 15). Your new implementation should aim to **bring the highest benefits from concurrency**.
Choose one of the concurrency programming paradigms discussed for Rust (in CS3211), sketch your implementation in Rust pseudo-code, and explain the following points:
    - Which paradigm are you using in this new implementation?
    - Which part(s) of the N-body simulation are you implementing concurrently? You may refer to point a. and choose one of the parts presented there.
    - Briefly describe the jobs (tasks) done concurrently.
    - What would be the maximum level of concurrency that you expect in your implementation? (i.e., number of threads that could potentially run in parallel)

    You can use the line numbers to refer to the lines of pseudo-code in Appendix. Do not copy over the lines that stay the same.
    In this answer, you are not allowed to use the quad-tree structure presented in Q4.

Answer:

[+4 marks]

- Rayon would be the best match
- Other solutions accepted, but the #threads should increase with #bodies

[-1 mark]

- No concurrency benefits
- Incorrect level of concurrency
- Unclear how the implementation would work.

[-2 marks]

- No pseudo-code was used and the solution is unclear.

[+1 mark]

- Paradigm is mentioned.
- The maximum level of concurrency is explained.

Q4. [12 marks] Concurrency in Action

In answering points a.-b. use the problem description of the **N-body simulation from Appendix.** Suppose you are working on a concurrent implementation of the N-body simulation.

In this question, we improve on the efficiency of finding the number of nearby bodies to a specific body *b* (line 21) for the N-body simulation from Appendix (page 15). We create a tree structure to quickly determine which bodies are potentially in the radius of *b*. We will use a tree structure called a quad-tree.

A quad-tree is a tree data structure in which each node has exactly four children. For this N-body simulation, the 2D space is partitioned into squares, such that each leaf of the quad-tree contains at most *L* bodies. Given a list of bodies and their positions, this quad-tree is built by recursively bisecting the space containing those bodies into nodes.
Once built, a node in the quad-tree has the following information:

- `isLeaf`: shows if this node is a leaf
- `children`: array of 4 node pointers representing the children of this node
- `bodies`: list of bodies at this node

Figure 1 shows an example of quad-tree.

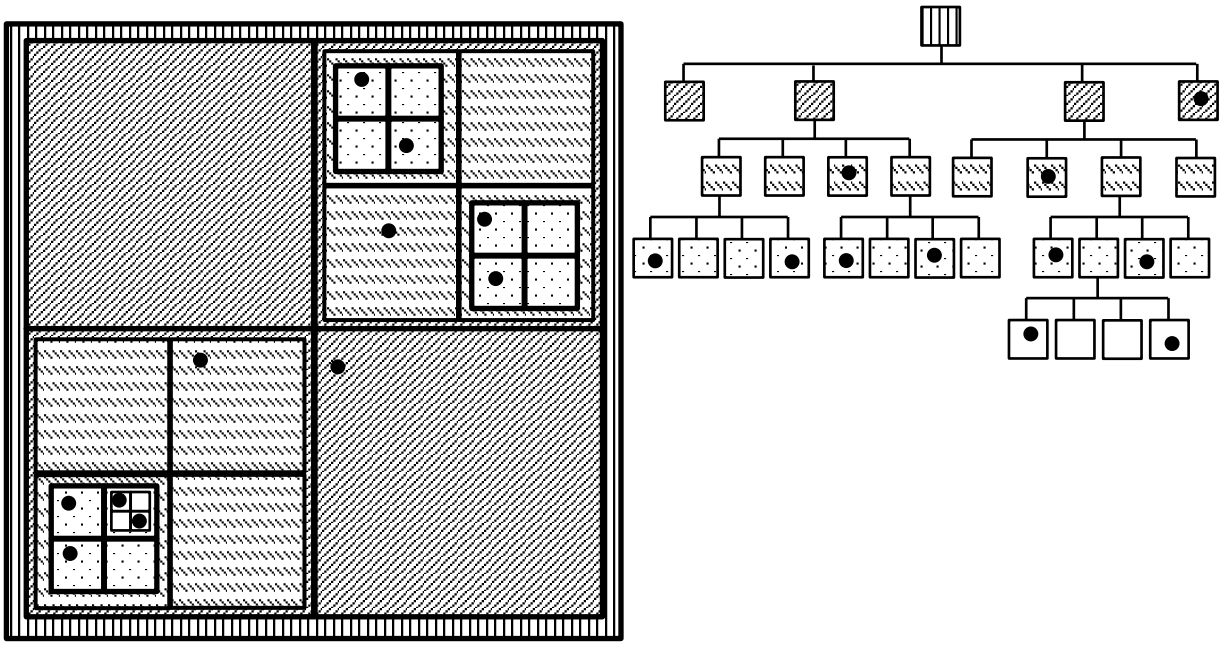


Figure 1: Quad-tree example with at most 1 body in a leaf.

a. [8 marks] Assume that you can build the quad-tree structure efficiently. Using this quad-tree, sketch a concurrent implementation for the N-body simulation from the Appendix and explain the following points:
    - Which part(s) of the N-body simulation are you implementing concurrently?
    - Briefly describe the jobs (tasks) done concurrently.
    - What would be the maximum level of concurrency that you expect in your implementation? (i.e., number of tasks that could potentially run in parallel)

Note the following:

- You may choose to present your answer in any programming language studied in CS3211 (modern C++, Go, or Rust).
- You should not focus on building the tree structure. Instead, you should use this structure for a concurrent implementation of finding nearby bodies.
- In this answer, you should not repeat the implementations presented in Q3 points b., d., or e.

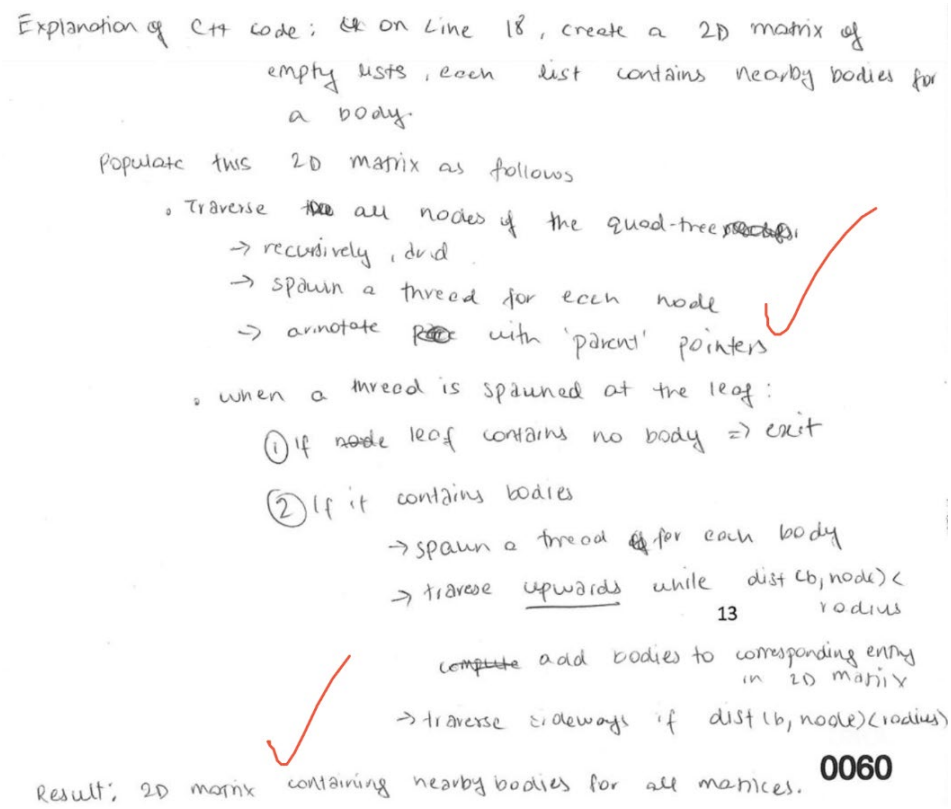
Answer:

[+1 mark]

- Paradigm is mentioned.
- The maximum level of concurrency is explained.

[+6 marks] Find bodies implemented concurrently.

1. Async/await: for each of the tree node, spawn a future that finds nearby bodies within the radius of b. Then await these futures bottom up. We need recursive futures for this → introduce an indirection using Box, or use recursive futures.
2. Use threads: for each body, spawn 4 threads to find nearby bodies in each child cell. Threads must be spawned recursively, until the section has one body or less.
3. Find all nearby bodies of all bodies, together, at the beginning of each step. We need to add parent pointers on the quad tree.



b. [4 marks] Suppose that you have a 2D square where the distribution of bodies is not uniform within the space. Explain two advantages of using this quad-tree structure in your concurrent implementation. You might compare with one of the implementations provided in Q3 under points b., d., or e.

Answer:

[+2 marks]

- Find nearby bodies that can be implemented concurrently with the quadtree
- The distribution of the bodies is not uniform; an appropriate paradigm such as async/await can help to balance tasks among threads
- The quad tree is cache friendly
- Less synchronization is needed because each body has its own quad tree.

END of PAPER

Do not write your answers on this sheet. This sheet should not be submitted with your paper.

Appendix: Problem scenario for Questions 3 and 4

N-body simulation

You are tasked to implement a program that simulates the movement of bodies. The bodies have varying masses and initial positions. Each body is defined using an id, and it has a mass, initial velocity, and initial position. Bodies are affected by gravitational forces from nearby bodies. Assume that bodies exist in a 2D square, and they interact with other bodies using gravitational forces in the 2D space*.
The simulation is done in steps. At each step, the resulting force (or acceleration) of each body is computed by summing up the interaction forces with all other bodies.

Assume that the bodies interact only within a specified radius. Consider the following sequential implementation of the **N-body simulation**.

Line#	Algorithm sketch
1	**find_nearby_bodies(Body b):**
2	nearby_bodies = list()
3	for each attractor in all_bodies:
4	if dist(b, attractor) < radius:
5	nearby_bodies.add(attractor)
6	return nearby_bodies
7	
8	**computeForce(Body target, Body attractor,**
9	**float radius):**
10	dist = dist(target, attractor)
11	dir = direction_from_target_to_attractor_body()
12	force = dir * target mass * attractor mass *
13	(G / (dist * dist));
14	return force
15	
16	**Main simulation:**
17	for each step:
18	for each Body b in all_bodies:
19	// find nearby bodies whose distance to this
20	// body is less than 'radius'
21	nearby_bodies = find_nearby_bodies(b)
22	// compute the sum of gravitational forces
23	// that need to be applied to this body
24	force = 0;
25	for each attractor in nearby_bodies:
26	force += computeForce(b, attractor);
27	// update the position and velocity of
28	// body b using the computed force
29	update(b)

Figure 2: Sequential implementation of N-body simulation

*Note (It is not necessary to use this note in solving the paper):
The gravitational force is computed as follows: $F_{ij} = Gm_i \cdot \frac{m_j(q_j - q_i)}{\|q_j - q_i\|^3}$
The total force F_i on body *i*, due to its interactions with the other N - 1 bodies, is obtained by summing all interactions: $F_i = \sum_{\substack{j=1 \\ i \neq i}}^{j=N} F_{ij} = Gm_i \cdot \sum_{\substack{j=1 \\ i \neq i}}^{j=N} \frac{m_j d_{ij}}{\|d_{ij}\|^3}$

BLANK PAGE

Do not write your answers on this sheet. This sheet should not be submitted with your paper.