NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

CS3210 – Parallel Computing

(Semester 1: AY2021/22)

Final Assessment Paper - Answers

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.
3. Students are required to answer **ALL** questions within the space in this booklet.
4. This is a CLOSED BOOK assessment. An A4 reference sheet and 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	/28	
Q3	/10	
Q4	/16	
Total	/80	

Q1. [26 marks] Heterogeneity and Performance

A. [8 marks] You are working on a game implementation where you need to check and compute new positions and state for each form of life (`Entity`). The position and the state of each `Entity` changes at every step. You parallelize the computation using one thread to compute and update the `Entity`'s positions, and another thread to simulate the `Entity`'s state. The global information of the game's N entities is stored in the global array `entities` in the code snippets below.

Line Code snippets

```
struct Entity {
  float position; // assume 1D position for simplicity
  float state;
};
Entity entities[N];

void* update_positions(void* arg) {
   for (int i=0; i<N; i++) {
       entities[i].position = compute_new_position(i);
   }
}

void* update_state(void* arg) {
   for (int i=0; i<N; i++) {
       entities[i].state = compute_new_state(i);
   }
}

// .. initialize entities here
pthread_t t0, t1;
pthread_create(&t0, NULL, &update_positions, NULL);
pthread_create(&t1, NULL, &update_state, NULL);
pthread_join(t0, NULL);
pthread_join(t1, NULL);
```

Figure 1: Code snippet for A.

(Assume the code compiles and runs successfully.)

Answer points i. and ii. according to the above description and code snippet.

i. Since there is no synchronization between thread 0 and thread 1 in the code snippet from A. (Figure 1), you expect near a perfect 2× speedup when running on two-core processor that implements (invalidation-based) cache coherence. You are shocked when you don't obtain it. Why is this the case? Explain.
   Assume that there is sufficient bandwidth to keep two cores busy. (4 marks)

Answer:

This is a classic false-sharing situation. Assuming the threads iterate through the array at equal rates (that is, they are on the same loop iteration at about the same time), both threads will be writing to elements on the same cache line at about the same time. The cache line will bounce back and forth between the caches of the two processors. In the worst case, every write is a miss.

Grading Scheme

- [1 mark] Identifying the access pattern to shared lines as **false sharing** between both threads
  - Credit is awarded if false sharing is implicitly described by the explanation, e.g. by specifying both threads access (logically disjoint) data in the same cache lime
- [1 mark] Assumption about **similar processing speeds** or **concurrent access** or **symmetric nature of the access**
  - False sharing results in a performance degradation only if the shared cache lines are accessed concurrently; if the cache line exists in only one of the private caches, no invalidation of the (shared) line occurs!
  - No credit for other assumptions, e.g. cache configuration, memory hierarchy
- [2 marks] Explaining **with reference to the provided code snippet** how false sharing degrades performance by generating a large number of invalidations from writes to shared lines
  - Partial credit awarded if justification is incomplete, i.e. only makes reference to either the provided code snippet or the invalidation-based coherence protocol

If false sharing is not mentioned or described, then the total grade awarded for this part is capped to 2m.

Common Mistakes

Students attributed the performance degradation to a large variety of causes, most of which were not relevant, including:

- Vague descriptions of contention for shared memory
- Overhead of thread creation and management
- Requiring memory consistency instead of coherence – note consistency model is the same for the entire system
- "Overhead" of cache coherence protocol – note cache coherence protocols are *essential* to ensuring correct execution on multi-processor systems with private caches!
- Differences in complexity of `compute_new_position` and `compute_new_state`
- Cache thrashing due to large working set – will also affect execution on a single thread
- Loading of lines from one thread overwriting lines from the other thread – not an issue with private caches, which require cache coherence

ii. Modify the code snippet from A. (Figure A) to correct the performance problem. You are allowed to modify the code and data structures as you wish, but you are not allowed to change what computations are performed by each thread and your solution should not substantially increase the amount of memory used by the program. You may use pseudo-code to explain your solution. (4 marks)

Answer:
A simple solution is to change the data structure from an array of Student structures to two arrays, one for each field. As a result, each thread works on its own array and scans over it contiguously.

float position[N]; float state[N];

Some students mentioned that an alternative solution was to offset the position of the threads in the arrays to ensure that, at any one moment, each thread operating in distant parts of the array. One example was to have one thread iterate from i=0 to N, and the other iterate backwards from N-1 to 0. This solution eliminates the false sharing and was given full credit. It should be noted that the spatial locality of data access is not as good (by a factor of 2) in this scenario than for the solution described above since each thread only makes use of 1/2 of the data in each cache line it loads.

<u>**Grading Scheme**</u>

- [2 marks] **CORRECTNESS**: solution eliminates *most* of the false sharing in the original code
  - o Both solutions (data reordering, execution offset) satisfy this property
- [1 mark] **OPTIMALITY**: solution uses the exact same amount of memory as the original code <u>AND</u> does not modify the computations performed by each thread
  - o This point is only awarded if the **CORRECTNESS** property is satisfied
- [1 mark] **EXPLANATION**: explains how false sharing is eliminated by removing **concurrent accesses to the same cache line** from both threads
  - o No credit awarded for incorrect or incomplete justification

If false sharing is not mentioned or described in part Q1A (i), credit will still be awarded for the modification. However, the EXPLANATION grade will not be awarded. If computations performed by each of the two threads are modified, a 1m penalty is imposed.

<u>**Common Mistakes**</u>

- No explanation provided for modified code snippet
- Introducing synchronisation variables between the two threads – only further degrades performance since the synchronisation variables are shared!
- Re-arranging the computations performed by each thread – not allowed as explicitly specified in the question
- Increasing the number of threads during execution
- Introducing additional parallel programming frameworks whilst not being explicitly allowed to, e.g. OpenMP and CUDA

B. [12 marks] You are part of a team that is designing new family of single-chip parallel processors. Your colleagues have already designed the following two CPU cores, which will be the building blocks of your system:

- **CPU-wimpy core**: this CPU core was designed for power efficiency.
- **CPU-brawny core**: this CPU core was designed for speed. It is **twice as fast as the CPU-wimpy** core, but it also **uses four times more power**.

Your team is considering three chip designs:

a. **Wimpy only chip**: all CPU-wimpy cores on the same chip,
b. **Brawny only chip**: all CPU-brawny cores on the same chip, and
c. **Heterogeneous chip**: N_{wimpy} CPU-wimpy cores and N_{brawny} CPU-brawny cores on the same chip.

Assume the following:

- The key benchmark that your team cares about takes 200 seconds to run sequentially on a single CPU-wimpy core, and 100 seconds to run sequentially on a single CPU-brawny core.
- The benchmark's computation consists of parallel and sequential parts, where the **parallel portion is f**. These two cannot overlap: while the sequential portion is executing on one core, the others remain idle.
- The parallel portion of the benchmark will experience linear speedup when it runs on multiple CPUs (i.e., there are no inefficiencies in running in parallel).

Additional note:

- Power is measured in Watts
- Energy is measured in Watts-hour (or Joules)
- Total energy consumption of a program = Power Used x Total Execution Time

Answer points i.-iii. based on the description found above at B.:

i. [4 marks] First consider a wimpy-only chip (design a.), where N_{brawny} = 0. Write an equation for the total execution time of the benchmark, as a function of f and N_{wimpy}, making optimal use of the cores. Briefly explain your answer.

Answer:
Given the benchmark takes 200 seconds to run sequentially on a single CPU-wimpy core, then:

- The sequential portion takes $t_{seq} = 200(1-f)$ seconds on a single CPU-wimpy core
- The parallel portion takes $t_{par} = 200f$ seconds on a single CPU-wimpy core

With N_{wimpy} CPU-wimpy cores, the workload of the parallel portion can be evenly divided to complete in only

$$t'_{par} = \frac{200f}{N_{wimpy}} \text{ seconds}$$

The total execution time is the sum of t_{seq} and t'_{par}, i.e.

$$\begin{aligned} T &= t_{seq} + t'_{par} \\ &= 200(1-f) + \frac{200f}{N_{wimpy}} \\ &= 200\left((1-f) + \frac{f}{N_{wimpy}}\right) \end{aligned}$$

<u>Grading Scheme</u>

- [1m] Correct expression for the execution time t_{seq} of the sequential portion
- [2m] Correct (optimal) expression for the execution time t'_{par} of the parallel portion with N_{wimpy} wimpy cores
    - Partial credit awarded if the expression for t_{par} is correct but not t'_{par}
- [1m] Correct expression for the total execution time
    - No credit awarded if no working or explanation is provided

If f is taken as the sequential fraction instead, credit for the above marking points will still be granted, but a 1m penalty will be imposed.

Common Mistakes

- Not reading the question and treating f as the **sequential** fraction instead
- Using Amdahl's law and computing the speedup instead of the total execution time
- Substituting $T_{0,brawny} = 100$ instead of $T_{0,wimpy} = 200$, or N_{brawny} instead of N_{wimpy} in the expressions for execution time

ii. [2 marks] Now consider a heterogeneous chip that contains both CPU-wimpy and CPU-brawny (design c.). Write an equation for the total execution time of the benchmark, as a function of f, N$_{wimpy}$, and N$_{brawny}$, making optimal use of the cores. Briefly explain your answer.

Answer:
To make optimal use of the cores, we want to maximise utilisation of all cores where possible, which minimises the total execution time.

Since the heterogeneous chip has both types of CPU cores, we minimise the execution time by executing the sequential portion on the faster CPU-brawny cores:

- The sequential portion takes $t_{seq} = 100(1-f)$ seconds on a single CPU-brawny core

Now, observe that the one CPU-brawny core is *twice* as fast as one CPU-wimpy core. Thus, to minimise the total execution time, we want all CPU-brawny and CPU-wimpy cores to complete their assigned workloads of the parallel portion in the same duration. This implies each CPU-brawny core should be assigned twice the workload of a CPU-wimpy core; in effect, each CPU-brawny core *performs* the work of two CPU-wimpy cores.

As the parallel portion takes $t_{par} = 200f$ seconds on a single CPU-wimpy core, with N_{wimpy} CPU-wimpy and N_{brawny} CPU-brawny cores, then the parallel portion takes

$$t'_{par} = \frac{200f}{2N_{brawny} + N_{wimpy}} = \frac{100f}{N_{brawny} + \frac{N_{wimpy}}{2}}$$

The total execution time is the sum of t_{seq} and t'_{par}, i.e.

$$\begin{aligned} T &= t_{seq} + t'_{par} \\ &= 100(1-f) + \frac{200f}{2N_{brawny} + N_{wimpy}} \end{aligned}$$

Grading Scheme

- [1m] Correct (optimal) expression for the execution time t_{seq} of the sequential portion

  - No credit awarded if the sequential portion is executed on a CPU-wimpy core
- [1m] Correct (optimal) expression for the execution time t'_{par} of the parallel portion
  - No credit awarded if the parallel portion is not optimally executed across all cores
  - No credit awarded for assigning 2/3 of the work to CPU-brawny cores and 1/3 of the work to CPU-wimpy cores; this is optimal ONLY when $N_{brawny} = N_{wimpy}$

If f is taken as the sequential fraction instead, credit for the above marking points will be granted with no penalty, as long as ONLY f and $(1-f)$ are interchanged.

Common Mistakes

- Not reading the question and treating f as the **sequential** fraction instead
- Using Amdahl's law and computing the speedup instead of the total execution time
- Incorrect substitution of variables in the final expression; common ones include
  - Substituting $T_{0,brawny} = 100$ instead of $T_{0,wimpy} = 200$ (or vice versa)
  - Substituting N_{brawny} instead of N_{wimpy} (or vice versa)
- Double-counting the execution time of the parallel portion by including one term for each type of CPU core
- Suboptimally executing the sequential portion
  - Executing the sequential portion entirely on a CPU-wimpy core or a mix of core types
- Suboptimally executing the parallel portion by not assigning CPU-brawny cores twice the work of each CPU-wimpy core
  - Executing 2/3 of the work on CPU-brawny cores and 1/3 of the work on CPU-wimpy cores; will result in one type of core idling by finishing earlier than the other

iii. [6 marks] Assume that a CPU-wimpy uses 1 Watt, and f = 90%. When cores don't execute instructions, their power usage can be approximated with 0.
Consider the three designs mentioned above:
a. **Wimpy only chip with 20 cores**
b. **Brawny only chip with 5 cores**
c. **Heterogeneous chip with 4 CPU-wimpy cores and 4 CPU-brawny cores**
Which of the three designs (a., b. or c.) is the most energy efficient for the given benchmark? (4 marks)
Justify (explain) one observation that you make about the energy efficiency of these designs. (2 marks)

Answer:

Consider a chip with N_{wimpy} CPU-wimpy cores and N_{brawny} CPU-brawny cores. From Q1A (ii), the execution of the sequential and parallel portions are:

$$t_{seq} = \begin{cases} 100(1-f), & N_{brawny} > 0 \\ 200(1-f), & N_{brawny} = 0 \end{cases}$$

$$t_{par} = \frac{200f}{2N_{brawny} + N_{wimpy}}$$

Energy efficiency is the *amount of computation performed* per *unit energy consumed* (giving rise to units of *performance per watt*), which differs from power efficiency. Since the amount of computation performed ("performance") is the same for all three designs, the most energy-efficient design expends the *least* energy executing the entire benchmark.

Since a CPU-brawny core uses four times as much power as a CPU-wimpy core (1 W), each CPU-brawny core uses 4 W of power. The energy consumed by the single core with power P during the sequential portion of the benchmark is

$$E_{seq} = t_{seq}P = \begin{cases} 400(1-f), & N_{brawny} > 0 \\ 200(1-f), & N_{brawny} = 0 \end{cases}$$

The energy consumed by all cores during the parallel portion is

$$E_{par} = t_{par}\left(4N_{brawny} + N_{wimpy}\right) = \frac{200f\left(4N_{brawny} + N_{wimpy}\right)}{2N_{brawny} + N_{wimpy}}$$

Thus, the total energy consumed to execute the benchmark is simply

$$E = E_{seq} + E_{par}$$

Hence, the final values for the three designs with $f = 0.9$ are:

a. $E = 200(1-0.9) + 200(0.9)(4\cdot 0 + 20)/(2\cdot 0 + 20) = 200\text{ J}$
b. $E = 400(1-0.9) + 200(0.9)(4\cdot 5 + 0)/(2\cdot 5 + 0) = 400\text{ J}$
c. $E = 400(1-0.9) + 200(0.9)(4\cdot 4 + 4)/(2\cdot 4 + 4) = 340\text{ J}$

We observe that the energy efficiency of a design decreases as we increase the proportion of CPU-brawny cores used in the execution of the benchmark. In general, the design with all CPU-wimpy cores achieves *twice* the energy efficiency of the design with all CPU-brawny cores, since it executes the same workload at half the speed (doubling the execution time) but with four times less power consumed.

We also observe that the heterogeneous design achieves a balance between energy efficiency and performance between the two extremes of an all CPU-wimpy (best efficiency) and all CPU-brawny (best performance) design; the total execution times for these three designs are

a. $T = 200(0.1 + 0.9/20) = 29\text{ seconds}$
b. $T = 100(0.1 + 0.9/5) = 28\text{ seconds}$
c. $T = 100(0.1) + 200(0.9)/(2\cdot 4 + 4) = 25\text{ seconds}$

Grading Scheme

- [1m] Identifying the **wimpy-only chip** as the most **energy**-efficient
  - Credit is awarded even if no or contradictory working is shown
- [1m x 3] Correct working for energy consumed by each of the three designs
  - 1m penalty imposed if power efficiency or another quantitative metric is computed
  - 1m penalty imposed if energy consumed for sequential and parallel portions are not computed separately for all three designs
  - Max 1m penalty for incorrect substitution of variables (e.g. per-core power)
  - If energy consumed is not explicitly computed, marks are awarded with this rubric:
    - [3m] Sound quantitative justification why the *wimpy-only chip* is most efficient

    - [2m] Sound quantitative justification why the chosen design is most efficient OR qualitative justification why the *wimpy-only chip* is most efficient
    - [1m] Qualitative justification why the chosen design is most efficient OR an an incomplete justification why the *wimpy-only chip* is most efficient
- [2m] Sound observation about energy efficiency **with qualitative justification**
    - Partial credit awarded if observation is not supported by working or an explanation

No marks were deducted for arithmetic errors; however, up to 1m penalty was imposed out of the 3m of working for incorrect variables in the expressions, e.g. $\boldsymbol{P_{brawny} = 1\ \mathrm{W}}$.

Common Mistakes

- Not reading the question and treating f as the **sequential** fraction instead (ECF awarded)
- Not stating assumptions when executing the benchmark on the heterogeneous chip to prioritise efficiency over execution time
- Incorrect unit conversions (no penalty)
- Computing the total power of each chip (the same) and concluding that all chips were equally efficient
    - Each chip uses a different amount of energy based on the composition of its cores!
- Computing the total execution time and concluding the chip that completes the fastest is the most *energy* efficient
    - Execution time does not tell the full story – a significant proportion of the execution time is spent in the sequential portion, where all but one core idles!
- Computing total energy as the product of *peak power* and execution time, instead of computing the energy consumed in each portion separately and taking the sum
    - This significantly overestimates the total energy consumed in the sequential portion, where most cores actually idle!
- Computing other metrics that do not have dimensions of energy, or energy efficiency (performance per unit energy)
    - Only metrics granted full credit are: energy used, average power or energy efficiency
- Incorrect substitution of variables in the final expression; common ones include
    - Substituting $T_{0,brawny} = 100$ instead of $T_{0,wimpy} = 200$ (or vice versa)
    - Substituting N_{brawny} instead of N_{wimpy} (or vice versa)
    - Substituting $P_{brawny} = 4$ W instead of $P_{wimpy} = 1$ W (or vice versa)
- Not stating an observation about energy efficiency or performance or their tradeoff

C. [6 marks] Answer points i. – iii. based on the following task dependence graph in Figure 2:

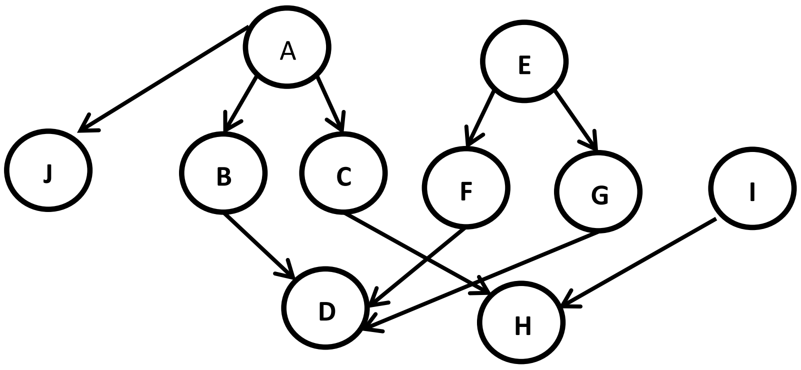

Figure 2: Task dependency graph

Assume tasks are of different sizes and they take the following units of time to execute:

Task	A	B	C	D	E	F	G	H	I	J
Time units	5	1	2	2	5	1	10	2	1	5

i. Plot the degree of concurrency (or degree of task parallelism) over time for the graph in Figure 2. (2 marks)

Answer:

ii. What is fastest execution time for the graph in Figure 2? (2 marks)

Answer:

Critical path: E->G->D. The execution time is 17.

iii. What is the maximum achievable speedup given infinite resources (Figure 2)? (2 marks)

Answer:

The maximum achievable speedup is the total execution of all the tasks divided by the critical path: Speedup = 34/17= 2.

Grading scheme:

- [6 marks] Many students had correct answers for all points (Q1C)
- [-2 marks] Some students did not plot the degree of concurrency over time. Instead they plotted how the tasks execute one after another over time, and the degree of concurrency is not clear.

Problem description for Q2: Distributed Task Scheduling

Large computing systems (e.g. datacenters or supercomputers) employ a distributed memory or distributed-shared memory organization, with reliance on a fast network of interconnects for inter-processor communication and memory access. Such systems need complicated schedulers that can maximize usage of the available nodes (computers) available in the computing system. Many times, multiple tasks (programs) need to be scheduled to run in parallel different nodes.

The set of tasks are seeded from an initial pool and are dynamically updated during execution to include new tasks generated when tasks are completed. There are several types of tasks, but they are either computationally intensive tasks or memory intensive tasks, and they might take different time to execute. Once a task becomes available, the execution of that task can be completed independently of any other tasks. The tasks produce new tasks that need to be scheduled and executed.

Assume you are working on a scheduler that assigns the tasks to run on the resources available in a large computing system. The goal is to maximize both the task throughput and average utilization of resources.

Q2. [28 marks] Distributed Memory

For point a. – g., you are solving the distributed task scheduling problem for a large distributed memory system. Assume the system has P cores (each one with its memory) and the total number of tasks that need to be executed is N. Assume P << N.

a. [6 marks] Explain how would you design a **master-worker** parallel algorithm to solve the distributed task scheduling problem? You may use MPI-like pseudo-code to show your algorithm for solving the distributed task scheduling problem. Remember that you need to schedule and execute the tasks in a parallel fashion. New tasks are generated as some tasks complete. Avoid mixing elements of other parallel programming patterns in your solution.

   In your solution, focus on identifying the problem's parallelism and on explaining the steps of your solution. Do not focus on optimizing your solution. Differentiate between the master and the worker operations (pseudo-code).

Answer:

Master, in a loop:

- All available tasks are sent for execution to ALL workers (scatter)
- Receive all new tasks from ALL workers (gather)
- Wait for all workers to complete before moving on to the next iteration

Worker, in a loop:

- Receive tasks (a task) from master
- Compute
- Send the new tasks to master

A design where the master sends a task and waits for a new task (result) from ANY worker before sending the next task will not take advantage of the parallelism, but it is accepted. Instead, the master should make sure that, if available, tasks are sent to all available workers before receiving.

It is very easy to introduce task pool elements in this solution. No penalty was applied for such answers, but the student had difficulty in answering point c.

[-1 mark] the master sends a task and immediately calls a blocking receive waiting to receive the new tasks produced by that specific worker.

[-1 marks] The implementation has minor logic mistakes

[2 marks] for using master-worker

[2 marks] first tasks can be executed

[1 mark] the newly created tasks are sent back to the master, but it is not clear how the will be executed.

[1 mark] the newly created tasks can be executed

b. [2 marks] What type of parallelism (data or task) does the distributed task scheduling problem entail for your solution from point a.? Explain your choice.

Answer:

[2 marks] Task parallelism.

[0 marks] Data parallelism was difficult to justify.

c. [5 marks] Propose a better programming pattern to solve the distributed task scheduling problem. Explain why your chosen pattern would be better than the master-worker programming pattern (as applied in your solution for point a.)? As part of your answer, briefly explain:
   - how your chosen pattern would be applied. (2 marks)
   - what metric are you using to conclude that the new pattern is better. (1 marks)
   - why the chosen pattern is better than master-worker. (2 marks)

Answer:

Task pool would have been the best choice. Tasks are added into a pool as they are created. The processes use a producer-consumer pattern to synchronize in getting tasks, running them and adding the newly created task back to the task pool. However, students that used a task pool for 2a. answer had difficulties in answering this question.

[2 marks] for explaining how to apply the pattern

[2 marks] Good comparison with master-worker

[1 mark] Metrics, such as utilization of the processes, resource utilization, idle time, etc, were accepted. Speedup was not accepted as a metric.

d. [2 marks] What type of interconnection network would you use to connect your P processors to make your implementation in MPI from point a. faster? Justify your choice with an example of how your chosen interconnect would improve the performance.

Answer:

Since 2.a. requires to use master-worker, a star interconnect would work best because the workers need to communicate only with the master.

[1 mark] Complete interconnect was not accepted as a fully correct answer. If the answer in 2a uses some other pattern, complete interconnect might be needed, and it would receive full marks. Task pool would not need a complete interconnect either.

e. [4 marks] As mentioned in the distributed task scheduling problem, the tasks that need to be executed are not of the same type. They might be either computationally intensive tasks or memory intensive tasks, and they might take different times to execute. Explain how your solution from point a. would handle this issue. Would there be any inefficiencies due to this characteristic of the problem?

Answer:

[4 marks] All workers get a task to execute, and they need to wait for all workers to finish before they can get another task. The execution time of different tasks differs. As such, many times workers have to wait for other tasks to finish. Marks were allocated for such points made in the answer, even if other slightly incorrect points were included.

[-1 mark] The problem is not clearly explained, or it is not clear how different tasks execute.

Students explain that tasks are assigned based on the hardware abilities of different workers. Such an approach seems difficult to achieve and did not receive full marks unless it was explained with details. Such a solution would also suffer of inefficiencies when the hardware does not match the types of tasks.

[-3 marks] Points made in the answer are incorrect, or the points in the question are not addressed.

f. [5 marks] You are now required to use a ring topology to solve the distributed task scheduling problem. All the processes are virtually placed in a ring topology such that each process will receive tasks from the process before it and will send tasks to the process after it (the last process will send tasks to the first process). Each process can communicate only with the neighbors.
   Explain your solution for the distributed task scheduling problem. Use MPI-like pseudo-code if needed. For each process, explain:
   - How the communication is done with the neighbors (1 mark),
   - What type of communication is used and why: blocking/non-blocking, synchronous/asynchronous (2 mark),
   - How deadlocks are avoided (1 mark),
   - What information is sent and received from the neighbors (1 mark).

Answer:

[6 marks] Communication is done via point-to-point message passing in a clockwise manner. Upon receiving a list of tasks from the worker to the left, a node will take one or several tasks, and pass remaining task list clockwise. When passing the tasks clockwise, the node adds more tasks (if they were created by the computation).
The communication can be blocking asynchronous communication as sends are needed to be paired with receives. To avoid deadlock, we can use buffers or use an odd-even send-receive pattern in communication.
Alternatively, non-blocking communication should not create deadlock, but can be more difficult to synchronize.

[-1 mark] – if the answer does not mention what tasks are sent among neighbors, and how the new tasks are added to the process

[-1 mark] – if the description does not mention that messages are sent in one direction only (clockwise or anti-clockwise)

[-1 mark] – if the deadlock avoidance is not mentioned explicitly

[-1 mark] – if the type of communication is mentioned, but not explained why it was chosen.

g. [4 marks] Assume that both your implementations for the distributed task scheduling problem from points a. and f. work according to requirements. The execution of this scheduling algorithm must end when there are no more tasks generated (that need scheduling). Explain:
   - ~~How~~ **When and how** would the implementation from point a. terminate? (2 mark)
   - Can you apply the same approach to terminate the implementation from point f.? Explain your answer and try to explain a potential termination approach for the implementation in point f. (2 marks)

Answer:

Termination for master-worker: 2 conditions must be met:

- No more tasks in the task pool [1 mark]
- All workers are idle [1 mark]
  Master sends a termination message to the workers

Termination for ring: several methods were accepted as long it makes sense. Partial marks were awarded as well, based on the explanation provided.

No marks allocated for approaches that assign a node to query all the other nodes, without mentioning how that would happen in a ring.

Termination approach:

- When each process is done processing (no more tasks to complete or generate), the process sends a Completion message to its right neighbor along with the completion messages it receives from the left neighbors.
- When the process sees again its initial message it means that all processes have ended, and no more tasks are in the system. The process sends a Finalize message. Rank 0 (by convention) calls MPI_Finalize once N finalize messages are received.
- If a process still has tasks to execute when a completion message reaches, that process will not forward the completion messages.

Q3. [10 marks] Memory Model

Assume the following code snippets are executed on four cores of a shared-memory machine. Initially before execution, all variables are equal to 0.

Table 1: Code snippets for Q3

P1	**P2**	**P3**	**P4**

(1) while (X == 0);
(2) print A
	(3) A = 1
	(4) while (T == 0);
	(5) C = A;
	(6) print B
		(7) while (A == 0);
		(8) T = 1
		(9) B = A
			(10) while (A == 0);
			(11) X = 1
			(12) print C

a. [2 marks] The code in Table 1 should ensures that "print A" and "print B" give the same value. Give an execution scenario by specifying the instruction interleaving to show how the code fails even under sequential memory consistency.

Answer: 3-7-8-10-11-1-2-4-5-6-9-12

b. [2 marks] Modify the code in Table 1 "print A" and "print B" give the same value (use line numbers). Explain why your new code will not fail under sequential memory consistency. You are not allowed to add new statements to the code (but you can swap the existing ones for each core).

Answer: swap 8 and 9.

c. [4 marks] Assume you are using the code modified for point b. that ensure the code will not fail under sequential memory consistency ("print A" and "print B" give the same value under sequential consistency).
This new code from point b. should ensures that "print A" and "print B" give the same value. Briefly explain if and how this new code from point b. would fail under relaxed consistency models:
i. TSO (1 mark):

Answer:

Code would not fail

ii. PC (2 mark):

Answer:

Code would fail: 9-6-7-1 (but P1 did not see the change in B)-2 (B is 0) – 3-5 -4-9-10 (A is 1)

iii. PSO (1 mark):

Answer:

Code would fail: It is possible to reorder 8 and 9.

Table 2: Copy of Table 1: Code snippets for Q3

P1	**P2**	**P3**	**P4**

(1) while (X == 0);
(2) print A
	(3) A = 1
	(4) while (T == 0);
	(5) C = A;
	(6) print B
		(7) while (A == 0);
		(8) T = 1
		(9) B = A
			(10) while (A == 0);
			(11) X = 1
			(12) print C

d. [2 marks] Assume you are using the code from Table 1 (copied over in Table 2). The code in Table 2 should ensure that all print statements (A, B, C) give the same value under total store order (TSO) model. How would you change the code from Table 1 to achieve this?

You are not allowed to add new statements to the code (but you can swap the existing ones for each core).

Answer:
A few answers were received:

- [2 marks] Impossible
- [1 mark] Moving the print statements as first statements in in P1, P2 and P4
- [0 marks] moving statements from one processor to another

Q4. [16 marks] General Questions

a. [3 marks] Relaxed consistency models address the concern that software complexity is increased by the need for unnecessary synchronization.
Is this statement true or false? Justify your answer.

Answer:

[1 mark] False.

[2 marks] Relaxed consistency allows for increased performance during execution. Synchronization is needed with or without relaxed consistency.

b. [3 marks] Cache coherence is the property that multiple caches contain the same set of items. Is this statement true or false? Justify your answer.

Answer:
[1 mark] False.

[2 marks] Cache coherence ensures that a value (of a memory location) is the same when present in multiple caches. The caches may contain any items (memory locations)

c. [4 marks] When developing a parallel solution, one should first develop and benchmark a serial solution. This is done
   - to provide the fastest development of a baseline against which correctness of more complex solutions can be measured, and
   - to provide a baseline, against which to measure performance gains.

   Is this statement true or false? Justify your answer.

Answer:

[2 marks] True.

[2 marks] Both points are correct.

Partial marks were awarded if students said that only one point is correct, while the other is false.

d. [6 marks] Ten years ago, IEEE's Technical Committee on Parallel Processing (TCPP), supported by funding from the National Science Foundation (NSF) and Intel and IBM Corporations, has been developing a Parallel and Distributed Computing Curriculum that seeks to embed parallel and distributed computing throughout all courses in an undergraduate Computer Science (CS) curriculum. The authors of this proposal write:

"In the past, it was possible to delegate issues regarding parallelism and locality to advanced courses that treat subjects such as operating systems, databases, and high-performance computing: the issues could safely be ignored in the first years of a computing curriculum. But it is clear that changes in architecture are driving advances in languages that necessitate new problem-solving skills and knowledge of parallel and distributed processing algorithms at even the earlier stages of an undergraduate career."

Discuss this claim! Do you agree or disagree? Give at least two arguments and justify your opinion with at least two examples.

Note: This question will be graded both for content/correctness of your technical points. Your answer should be well-written, organized, and clear. Use the information covered in CS3210 to argument your answer and give examples.

Answer:

Most of the students agreed with the claim.

Any good explanations showing two DIFFERENT arguments and explaining them were accepted. Most of the students that attempted this question managed to get full or almost-full marks.

END OF PAPER