[Template for Applied Session submission]

Applied Session Week {Week 10}

Student ID	Student Name	Student Email
32837933	Chua Sheng Xin	schu0077@student.monash .edu

Tasks

{Answers
Question 1
A) The differences between GPGPU/GPU and CPU in terms of the number of cores, core complexities, memory bandwidth and instruction level parallelism (ILP) exploitation are as follows. CPU typically has a small number of complex cores while GPGPU/GPU has thousands of simpler specialized cores designed for parallel processing. Moreover, CPU cores are more complex and optimized for high instruction level parallelism (ILP) with sophisticated control logic, branch prediction and large cache memories. In contrast, GPGPU/GPU Cores are simpler with the focus on throughput and massive data level parallelism but less emphasis on control logic. CPU in general has lower memory bandwidth but with lower latency and larger caches while GPGPU/GPU has significantly higher memory bandwidth optimized for high throughput in parallel operations but with higher latency. Additionally, CPU is designed to maximize ILP with techniques like superscalar execution, out of order execution and speculative execution while GPGPU/GPU focuses more on data parallelism where the same kernel function can be repeatedly applied into the records in an input stream with limited ILP within each individual core.
B) Diagram:

```
Central Processing Unit -----PCI Express(PCIe)-----Graphics Processing Unit
            |                                                      |
            |                                                      |
     DDR5 SDRAM                                                 GDDR5
```

The data transfer between the memory of a machine for example, DDR5 SDRAM and the memory of a GPU for example, GDDR5 can be facilitated through a PCI Express (PCIe) interface. When DDR5 SDRAM is connected to the CPU, GDDR5 is connected to the GPU and both CPU and GPU are connected via a PCI Express (PCIe), the data can be transferred. The CPU can issue memory transfer request via PCIe that enables the data to be moved from DDR5 SDRAM to GDDR5 through the PCIe bus. After receiving the data, the GPU processes it in parallel. When the processes are finished, the processed data is then transferred back to DDR5 via PCIe.
C) The simple processing and data flow of a CUDA programming model can be explained by an example of application in matrix multiplication which involves heavy computation and is ideal for GPU parallelization. This task involves multiplying two matrices and this can be done in parallel since each element of the resulting matrix can be computed independently. The flow in this case starts when the CPU allocates memory and initializes the matrices. The CPU then transfers the matrices to the GPU. In the GPU, the CUDA kernel is launched to execute matrix multiplication on the GPU by distributing the task among the many GPU cores. The GPU performs parallel computations by handling multiple matrix

cells concurrently. Once the computation is complete, the results are transferred back to DDR5 via PCIe. This approach significantly speeds up computation for large matrices and demonstrates the efficiency of using a GPU for parallel tasks.

Question 2

A) The three reasons distributed systems need to synchronize time are consistency in data operations, coordination of events and fault tolerance. In a distributed system, multiple nodes may be accessing and updating shared data at the same time. If the nodes have different perceptions of time, it can lead to inconsistencies in data operations. For example, a banking application has multiple branches that update a customer's balance. If one branch updates the balance based on an outdated timestamp, it may overwrite a more recent transaction from another branch. Moreover, many distributed applications rely on the knowledge on the order of events to function correctly. Synchronizing time makes sure that events are executed in the correct sequence across different nodes. In a distributed transaction system, certain operations must occur in a specific order to maintain data integrity. A deadlock might happen if one node believes an event occurred before another due to a time mismatch. Additionally, time synchronization plays a crucial role in enabling fault tolerance mechanisms and recovery processes. A consistent time reference helps the system to identify and recover from failures. In a distributed system using leader election protocols, nodes are required to determine which node should become the leader based on timestamps. If the nodes do not have synchronized clocks, it may lead to scenarios where two nodes believe they are the leader.

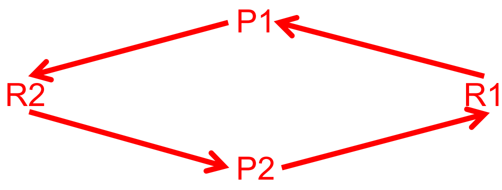
B) The difference between logical clock and real clock synchronization is as follows. Real clock synchronization is the alignment of physical clocks across distributed systems. This involves synchronizing the actual time as measured by clocks to assure that all nodes in a distributed system have a consistent timing. Protocols such as the Network Time Protocol (NTP) and Precision Time Protocol (PTP) are commonly used for this purpose. The purpose of real clock synchronization is to maintain the fact that all nodes in a distributed system agree on the actual time. This is vital for applications that depend on accuracy such as logging events and transaction timestamps. Real clock synchronization typically involves exchanging time messages between nodes. For example, NTP servers provide time information that clients can use to adjust their clocks. The synchronization process takes in account of network delays and other factors to minimize gaps in the measured time. Logical clocks on the other hand provide a way to order events in a distributed system without relying on physical time. They assign a timestamp to events based on a defined logical sequence rather than actual time. Lamport timestamps are one of the most known methods for implementing logical clocks that allows systems to maintain a partial ordering of events. In Lamport's algorithm, each process maintains a counter that is incremented with each local event and included in messages sent to other processes. Upon receiving a message, a process updates its counter to be greater than both its current value and the value in the received message. The goal of logical clocks is to provide a mechanism for ordering events in a distributed environment where there is no need for a real time clock across nodes. Logical clocks help in establishing a "happens-before" relationship among events in a distributed system that informs systems about causality. The references are included below.

C) Diagram:

$$\begin{tikzcd}
E10(1) \arrow[r, dash] \arrow[drrrr] & E20(2) \arrow[r, dash] & E30(3) \arrow[r, dash] & E40(4) \arrow[r, dash] & E50(5) \arrow[r] & P1 \\
E15(1) \arrow[r, dash] & E25(2) \arrow[r, dash] \arrow[urr] & E35(3) \arrow[r, dash] & E45(4) \arrow[r, dash] & E55(5) \arrow[r] & P2
\end{tikzcd}$$

The diagram contains two processes (P1, P2) with events occurring within each process. Each event will have a logical timestamp assigned inside the brackets. For example, event E10 in Process P1 has timestamp of 1. An arrow from one event to another indicates a message being sent and received or some causal influence between those events. Events within the same process follow a natural progression from earlier to later based on their logical clocks. Arrows between different processes indicate that one process's event influences another. The arrows infer causal ordering and if there is an arrow from A → B, A must happen before B in real time, regardless of their logical clocks. For example, there is an arrow from event E10 to E55 so E10 must happen before E55. The problem lies in the fact the logical clock in each process advances independently or through messages, but there is no record of where the advances comes from. In the diagram, E25 → E40 implies C(E25) < C(E40) but we cannot assume that the next events E35 → E50. Hence even though the arrows form a partial order which is the ordering for causally related events, it might not be consistent between consecutive sets of events. In events that the processes are concurrent, meaning no causal relationship and as such, arrows, exists between them, Lamport's Clock Algorithm also cannot be used to deduce the ordering of events. In summary, Lamport's Clock Algorithm can only be used to deduce partial ordering at best between processes.

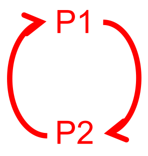
Question 3
A) Resource Allocation Graph:



In the above graph there are two processes P1, P2 and two resources R1, R2. R1 is assigned to P1 but P1 still requires R2 to complete its operations. In the same way, R2 is assigned to P2 but P2 needs a further R1 to complete its operations. Thus a deadlock occurs because neither processes can complete their operations and release resources for the other process and it can be observed that there are certain conditions to be met for a deadlock to occur. The four necessary conditions for deadlock to occur are:

1. Mutual exclusion: Each resource can be held by at most one process.
2. Hold and wait: Processes holding resources can request more resources.
3. Non-preemption: Resources cannot be forcibly taken from a process once granted.
4. Circular wait: There is a cycle of two or more processes are waiting for resources held by one of the other processes.

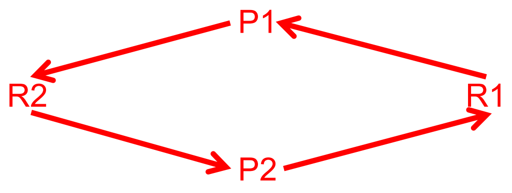
B) Wait For Graph:



The Chandy-Misra-Haas algorithm is a distributed deadlock detection algorithm that works on wait for graph and uses probe messages to detect cycles that are deadlocks. Each process sends a probe message when it has a need to wait for a resource held by another process. The probe message contains the initiator's ID, the current process ID, and the process being waited on. For example, the process P1 is waiting for P2, it sends a probe message along the edge. When the process P2 receives a probe, it checks if it is waiting

for another process. If so, it forwards the probe message, adding its own ID to the chain. If the probe message returns to the initiator P1, that implies that a cycle is detected and there is a deadlock among those processes. The relationship between the Chandy-Misra-Haas algorithm and the wait for graph is that the algorithm effectively traverses the wait for graph using probe messages to detect deadlocks.
C) Resource Allocation Graph:

$$\begin{tikzcd}
 & P1 \arrow[dl] & \\
R2 \arrow[dr] & & R1 \arrow[ul] \\
 & P2 \arrow[ur] &
\end{tikzcd}$$



False deadlocks could happen with the centralized algorithm, based on the above resource allocation graph due to delayed messages or asynchronous communication. A centralized deadlock detection algorithm collects information about the system's state in the form of the resource allocation graph from various nodes to detect cycles. However, we should consider a distributed system in the above graph that shows that the Process P1 is holding Resource R1 and is about to release it. Process P2 requests R1 but has not yet received information that P1 released it due to network delay. The centralized algorithm based on outdated information may assume P2 is still waiting indefinitely for R1 and detects a false deadlock. However, P1 may have in reality already released R1 and the system is no longer in deadlock. The false deadlock happens because the central controller incorrectly detects a deadlock due to outdated state information.
}

{Screenshots}

{References
1. Lamport, L. (1978). “Time, Clocks, and the Ordering of Events in a Distributed System.” Communications of the ACM, 21(7), 558-565.
2. Mills, D. L. (1991). “Network Time Protocol (Version 1) Specification and Implementation.” RFC 1059.
}