MONASH University

[Template for Applied Session submission]

Applied Session Week {Week 12}

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

Tasks

{Answers
Question 1
A) For lock based concurrency control, parallel and distributed algorithms can be improved in performance by distinguishing read locks from write locks because read locks allow multiple transactions to read the data concurrently while write locks prevent other transactions from reading or writing the data. This distinction enables parallelism as multiple transactions can hold read locks simultaneously without causing conflicts while write locks guarantee the consistency of data by blocking conflicting operations. For example, if two transactions need to read the same data, both can proceed in parallel if they hold read locks. However, if one of the transactions requires a write operation, it must acquire an write lock that forces other transactions to wait. Consider two transactions, T1 and T2 with operations on a shared data item X.
Diagram:

```
        Read(X)       Write(X)
T1|--------|-----------------|----------------|
Begin                                  End
        Read(X)                        Write(X)
T2|--------|-----------------------------------|----------------|
Begin                                                        End
```

Without distinguishing between read and write locks, T1 and T2 may block each other unnecessarily when they both attempt to place a read lock on X. By distinguishing between read locks and write locks, T1 and T2 can both perform the read operation at the same time. Even though for write operation, they have to wait for one another to finish the operation. The above scenario illustrates that distinguishing read locks from write locks allows more transactions to run in parallel.
B) Lock granularity would affect the parallel performance of a parallel / distributed algorithm due to the size of the data item being locked. Finer granularity enables smaller lock units like rows and allows for higher concurrency as locks are placed on smaller subsets of data. There is also more overhead for lock management. Coarser granularity enables larger lock units like entire tables and reduces the number of locks but decreases concurrency as more transactions are blocked. For coarse grained locking, there is lower concurrency but less overhead. Consider two transactions, T1 and T2 that are working on rows in the same table.
Diagram:
Coarse grained locking (table level lock):

Time	Transaction T1	Transaction T2
t1	Lock Table 1	
t2	Update Table 1	
t3	Unlock Table 1	
t4		Lock Table 1

t5		Update Table 1
t6		Unlock Table 1

Fine grained locking (row level lock):

Time	Transaction T1	Transaction T2
t1	Lock Row 1	Lock Row 2
t2	Update Row 1	Update Row 2
t3	Unlock Row 1	Unlock Row 2

C) Clock synchronization algorithms are needed for timestamp based concurrency control because in timestamp based concurrency control, each transaction is assigned a unique timestamp when it starts. The system ensures that conflicting operations are executed in timestamp order and preserves the serializability of transactions. This technique requires that the timestamps reflect the actual order in which transactions occur. If clocks are not synchronized, the system is not able to correctly determine which transaction occurred first. Moreover, timestamp based concurrency control uses timestamps to resolve conflicts such as write-write conflict or read-write conflict. If two machines have unsynchronized clocks, one machine might process an operation based on an incorrect assumption about the transaction order. Additionally, transactions with newer timestamps must not be allowed to read or write data that was modified by older transactions. The transaction needs to be known or it will lead to unnecessary rollbacks or errors.

Question 2

A) In row-wise block striped decomposition, the matrix is divided row by row into horizontal strips where each processor or node is responsible for a certain number of rows. For a matrix A of size N x N and another matrix B of size N x N, the multiplication C=A×B involves multiplying rows of A by columns of B. In the row wise decomposition, each processor gets a block of rows of A while the entire Matrix B is shared across all processors. For example, there are Matrix A and B, both of size 4×4 computed by 2 processors. The row-wise block-striped decomposition would divide matrix A into two horizontal strips, so processor 1 gets the first 2 rows of A, and processor 2 gets the last 2 rows of A. Each processor needs the entire matrix B to compute its part of the result matrix C. As such, both processors must receive a copy of matrix B and this results in communication overhead. In this scheme, the entire matrix B must be broadcast to all processors regardless of how the matrix A is partitioned. Since the entire matrix B needs to be transmitted to each processor, this can result in higher communication overhead for large matrices. For tile segmentation partitioning, the matrix is divided into smaller tiles or sub-matrices to improve load balancing and reduce communication overhead since tiles are smaller and can be computed more independently. For matrices A and B of size 4×4 and 2 processors, we divide each matrix into 2 x 2 tiles. Processor 1 could work on the top left and bottom left blocks of A and B while Processor 2 could work on the remaining blocks. Each processor would only need a portion of the matrix B corresponding to the tile they are working on which significantly reduces the amount of communication needed. In this scheme, processors only need to exchange the tiles that they require for their computations. Instead of transferring entire matrices, smaller tiles are communicated between processors and because less data is communicated, this scheme is more communication efficient for large matrices.

B) Based on the theory of Bernstein's conditions, Instruction 1 (y = y + z + 1):
Read set: {y, z}
Write set: {y}
Instruction 2 (x = z):
Read set: {z}
Write set: {x}

For Bernstein's conditions, there must be no intersection between the read set of one instruction and the write set of another. There is no intersection between Instruction 1's write set and Instruction 2's read set: {y} and {z}. Moreover, there is no intersection between Instruction 1's read set and Instruction 2's write set: {y, z} and {x}. Moreover, there is no intersection between Instruction 1's write set and Instruction 2's write set: {y} and {z}. Thus, the two instructions are independent and can be executed in parallel.
C) In Cannon's Algorithm, matrix data is partitioned into submatrices or tiles. The algorithm involves shifting rows and columns of submatrices between processors in a systematic way to perform matrix multiplication in parallel. Each processor holds a tile of the matrix, and the communication happens in a circular manner where each processor sends and receives tiles from its neighbors. Each processor initially holds a block of matrices A and B. However, for efficient matrix multiplication, the blocks of matrices need to be aligned such that relevant data is available for computation without excessive communication. Each block of matrix A in processor (i,j) is cyclically shifted left by i positions. This ensures that every processor gets the right initial block of matrix A for the first multiplication. Similarly, each block of matrix B in processor (i,j) is cyclically shifted up by j positions. After the initial alignment, each processor computes the product of the blocks it holds: the submatrices of A and B and adds the result to its local portion of the output matrix C. After each multiplication step, the blocks of matrix A are shifted left by one position cyclically, and the blocks of matrix B are shifted up by one position cyclically. This cyclic shifting makes sure that each processor eventually computes the product of all pairs of submatrices it is responsible for. These shifts and computations repeat for the square root of the number of processors so that every processor has completed its portion of the computation for the final matrix C.

Question 3
A) Dennard scaling describes how as transistors get smaller, their power density remains constant. The power density stays constant when transistors get smaller is important for exponential growth in computational performance because it allowed manufacturers to increase the number of transistors without increasing the overall power consumption. More transistors meant higher performance as more transistors allow for greater parallelism, better processing capabilities and more complex designs. Moreover, smaller transistors allowed for the inclusion of more features within the same chip, improving efficiency and overall performance. There is also the reason of sustained growth in performance where the combination of increasing transistor density with manageable power consumption enabled exponential growth in computational performance by following Moore's Law for decades. The consistency of the power density of transistors is important for maintaining exponential growth in computational performance because it means that as we add more transistors, we can increase the clock speed without increasing power consumption. Without this power scaling, increasing transistor density would have led to unmanageable heat dissipation and power consumption and limited performance improvements.
B) The reasons why algorithms/tasks may not have significant performance improvements despite more cores or higher frequencies are limited parallelizability and memory and I/O bound tasks. Moore's Law suggests that the number of transistors doubles approximately every two years, which typically results in either more cores being added to processors or higher clock frequencies. However, despite this increase in hardware capabilities, Amdahl's Law explains why not all algorithms benefit significantly from more cores or higher frequencies. Amdahl's Law defines the potential speedup of a task when part of the task can be parallelized. It is expressed as $S=1/((1-P)+P/N)$ where S is the speedup, P is the portion of the task that can be parallelized and N is the number of processors/cores. As such, if a portion of the algorithm remains sequential, increasing the number of cores

will not significantly improve performance. For example, if 90% of an algorithm can be parallelized, increasing the number of cores beyond a certain point will result in diminishing returns, because the serial part becomes the bottleneck. Another reason is memory and I/O bound tasks. Some algorithms are memory-bound or I/O-bound which limiting factor is not the computation, but how fast data can be fetched from memory or how fast I/O operations like reading or writing from disk can be performed. Increasing core counts or clock frequencies in these cases does not help much, as the system waits for data from memory or storage. For example, if a large dataset needs to be read from disk or fetched from memory, adding more cores would not improve performance if the bottleneck is the speed of memory access.
C) Keck's Bandwidth Law would affect the performance of a distributed application in the future in terms of the relationship between computational power and communication bandwidth in distributed systems. The law emphasizes the growing gap between processor speeds which are increasing rapidly and network bandwidth which grows at a slower rate. This mismatch creates a bottleneck for distributed applications that require frequent communication between nodes. Keck's Bandwidth Law can be summarized as Ttotal=Tcomputation+Tcommunication where Tcomputation is the time spent on local computation and Tcommunication is the time spent communicating data between distributed nodes. As processors become faster, the time spent on local computation decreases. However, network bandwidth increases at a slower pace, so the time spent on communication does not decrease proportionally. In a distributed application, as more nodes or processors are added to the system, the amount of communication required between nodes increases. If the bandwidth is limited, communication delays will dominate overall performance. Many distributed applications such as large scale simulations or machine learning workloads require frequent communication between nodes. If the network bandwidth does not scale with computational power, the application becomes network bound rather than compute bound, limiting the potential performance improvements that could be achieved by adding more processors. For example: In a distributed machine learning task, models and gradient updates need to be communicated between nodes frequently. If network bandwidth is insufficient, this communication will slow down the entire process regardless of how fast each node can compute. In the future, as we continue to add more cores to distributed systems, Keck's Bandwidth Law suggests that unless network bandwidth can keep up with the growth in computational power, performance improvements for distributed applications will be limited by communication delays.

}

{Screenshots}

{References}