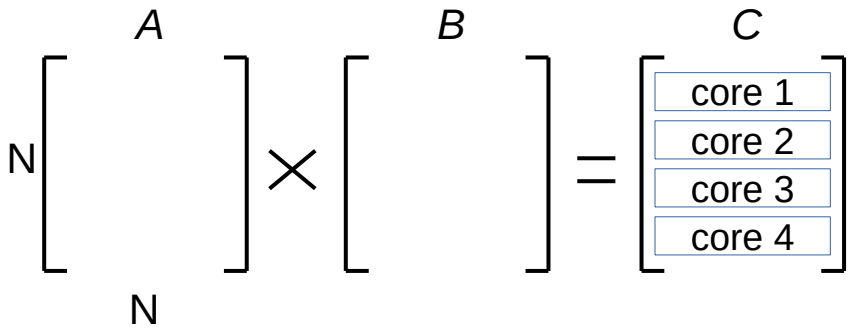
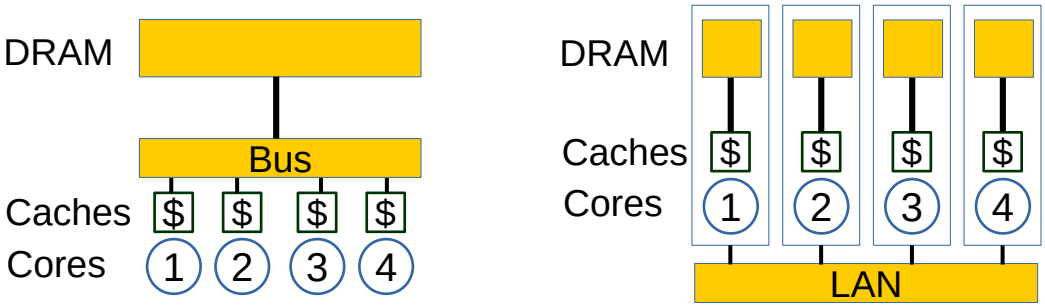
Problem 1 2018 re-exam

The two main programming paradigms for parallel computers are shared memory and message passing. In the course, the parallelization of a matrix multiplication ($A \cdot B = C$) was used to exemplify both paradigms. In this problem we want to analyze the performance of both approaches.

Assume that the matrices are square and each consists of N rows and columns. A typical way to decompose (parallelize) the problem over four cores is shown in the following figure:



This strategy can be used to parallelize the algorithm for both (a) shared memory and (b) message passing systems. The figure below shows two such systems.



(a) shared memory

(b) message passing

The task is to discuss whether the five following statements are correct in the context of the two paradigms and the two shown systems. You should write 1-2 sentences for each statement and paradigm (i.e., total 10 answers). Just stating *true* or *false* will not be considered sufficient.

Unless otherwise specified, the following assumptions are to be considered:

- In both systems the DRAM, Bus and LAN support up to 32 GB/s of bandwidth.
- Initially the matrices A and B are stored in the DRAM memory connected to core 1. The algorithm finalizes when matrix C is stored back in the memory of core 1.
- The matrix multiplication is parallelized into multiple threads, each consuming a constant 4 GB/s of DRAM bandwidth on each core.
- The DRAM capacity is not a limiting factor

- The cache coherence protocol used in the shared memory system (a) is MSI-invalidate

Statement A “In order to function correctly, it is necessary to explicitly copy both matrices A and B into the DRAM memory connected to each core.”

St. B “As long as the (Bus / LAN) interconnect bandwidth is larger than 16 GB/s, the execution time is independent of the speed of the interconnect (Bus / LAN)”

St. C “The execution time does not depend on the DRAM bandwidth, as long as the bandwidth exceeds 8 GB/s”

St. D “As the matrix size N increases, the speed-up compared to a single core approaches 4x”

St. E “Neither of the two parallelization paradigms requires Operating System support.”

Please justify your answers. Simply answering *Correct* or *False* will not be considered not enough.

Answers to Problem 1

Statement (A)
(a): For the Shared memory system no copies are necessary as all cores have access
(b): Copies are necessary in this case, but the matrix A does not need to be copied in full, only the rows belonging to each core

Statement (B)
(a) This is correct, as the MxM kernel needs 4x 4GB/s to operate without contention
(b) Given that matrices need to be copied, the execution time will always depend on the LAN bandwidth

Statement (C)
(a) this is not correct, as the maximum memory bandwidth required is 4x 4GB/s = 16 GB/s
(b) this is correct, as only 4 GB/s are required for each node (each node executes a single MxM kernel)

Statement (D)
(a) true, since the 4 cores are not able to saturate the system’s bandwidth
(b) true, for the same reason and (a), and also because the relative amount of communication required for the MxM (proportional to N^2) becomes negligible compared to computation (proportional to N^3) when N becomes large

Statement (E)
(a) incorrect, as there is always a need to create threads, which requires OS support (e.g. pthread_create)
(b) incorrect. In addition to creation of processes it is necessary to communicate data, which involves the OS

Problem 1 2018 exam

The goal of this exercise is to reason about three locking schemes designed to protect a shared data structures in three different scenarios. The underlying parallel computer is a sequentially consistent shared memory bus-based multiprocessor that uses the MSI-invalidate protocol to manage coherence across private caches.

The three scenarios under consideration are:

- Scenario #1: the shared data structure is accessed by a single thread.
- Scenario #2: the lock is accessed by multiple threads but lock contention is very low.
- Scenario #3: the lock is accessed by many threads, and lock contention is high.

The following codes show the implementation of the three locking schemes. In the following assume that R0=0, and that the lock is taken when its value is 1. The variable holding the lock is `_lock`.

Action/Label	**Lock #1**	**Lock #2**	**Lock #3**

`Lock:`	`T&S R1, _lock`
	`BNEZ R1, Lock`
		`LW R1, _lock`
		`BNEZ R1, Lock`
		`T&S R1, _lock`
		`BNEZ R1, Lock`
			`ADDI R1,R0,1`
			`LL R2,_lock`
			`SC R1,_lock`
			`BEQZ R1, Lock`
			`BNEZ R2, Lock`

`Unlock:`	`SW R0, _lock`	`SW R0, _lock`	`SW R0, _lock`

That task is to describe, for each of the three locking schemes:
(a) How well do the locking schemes apply to the three scenarios shown above?
(b) What synchronization hardware needs to be added to the basic bus-based multiprocessor (as described in Lecture 4) to support the locking schemes?
(c) Assume now that the memory consistency model is a synchronization-based memory consistency model such as weak ordering. Will the locking schemes still work correctly?

Solution to the Problem

The three locks correspond to a test-and-set based spinlock, a test-and-test-and-set spinlock, and a LL-SC spinlock that implements the same functionality as the test-and-set spinlock (the first four lines emulate a T&S and the fifth line checks if the lock was successfully taken).

(a) Performance and Efficiency

Scenario 1:
Since the data is only accessed by a single thread it is not necessary to actually implement mutual exclusion in the access. No lock should be used for this case. However, even if a lock is used, the overhead will be low, since the lock will always reside in the local cache and it will always be taken in the first try.

Scenario 2:

In scenario 2 multiple threads access the data structure. As long as contention is low Locks 1 and 3 are a slightly better solution as it takes only one atomic read-write to take the lock, while Lock 2 requires at least 1 load and 1 atomic read-write operation.

Scenario 3:
Given that the lock is highly contented, we assume that most threads do not immediately get access to the lock. Given that the coherence protocol is MSI, using Lock 1 will result in the lock migrating across all spinning threads. Each T&S generates an invalidation and a migration. Lock 2 is a better solution in this case as it exploits the 'S' state that allows multiple threads to share the value. All threads keep the lock locally in the 'S' state, and only when the lock is released by the holding thread is a invalidation generated. Hence there is only one invalidation each time a lock is release, instead of continuously by all the spinning threads (Lock 1). Since Lock #3 is an emulation of Lock #1, the behavior is similar to Lock #1.

(b) Hardware support

Locks 1 and 2 require HW necessary to implement atomic read-modify-write instructions:

- invalidation-based coherence protocol
- cache controller acquires cache line in ‘M’ state and performs T&S atomically

Lock 3 uses LL-SC which is typically found in load/store architectures (e.g. RISC pipelines). LL-SC support requires:

- LL-bit, set when LL is executed
- method to detect writes to cache line with lock (resets LL-bit)
- SC fails if LL-bit has been reset

(c) Memory Consistency

In weak ordering, accesses to synchronization data are treated differently by the hardware
from accesses to other shared and private data. Such operations act as memory barriers on all accesses, hence the behavior for programs using locks is equivalent to sequential consistency. Hence all three lock implementations are still correct under weak ordering.

Problem 3 2017 exam

A traditional mutual exclusion algorithm that does not require explicit synchronization is Dekker's algorithm. Dekker's algorithm for two threads (T1 and T2) can be described as follows:

```
INIT A=B=0

T1                              T2
...                             ...
A=1                             B=1
while(B==1);                    while(A==1);
<critical section>              <critical section>
A=0                             B=0
```

We describe loads and stores using the notations:
$L^{x}(A)Y$: Load by thread `x` to address `A` returning value `Y`
$S^{x}(A)Y$: Store by thread `x` to address `A` writing value `Y`

Considering only the first two lines of the algorithm, there are four possible outcomes as follows

Outcome 1

T1	T2
$S^{1}(A)1$	$S^{2}(B)1$
$L^{1}(B)0$	$L^{2}(A)0$

Outcome 2

T1	T2
$S^{1}(A)1$	$S^{2}(B)1$
$L^{1}(B)0$	$L^{2}(A)1$

Outcome 3

T1	T2
$S^{1}(A)1$	$S^{2}(B)1$
$L^{1}(B)1$	$L^{2}(A)0$

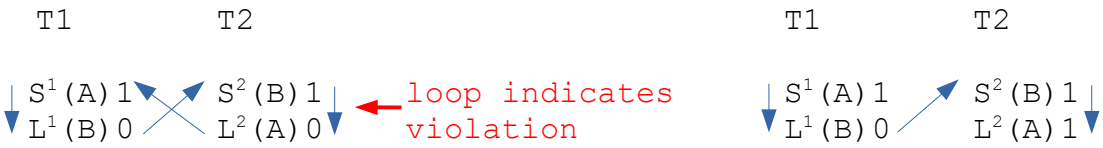
Outcome 4

T1	T2
$S^{1}(A)1$	$S^{2}(B)1$
$L^{1}(B)1$	$L^{2}(A)1$

Which of these outcomes are possible under Sequential consistency?
Which of these outcomes are possible under Total Store Order (ie, a relaxed memory model with Store-Load relaxation and Forwarding Store Buffers)?
Which of these outcomes are possible under Relaxed Memory Ordering?

Solution to Problem 3:

Sequential consistency:
To understand which of these outcomes are acceptable we need to construct a order that generates them, while respecting intra-thread dependencies, and respecting the orders dictated by the consistency model. Sequential consistency is the strongest model, as it forces to respect all orders: load-load, load-store, store-load, store-store. Out of the four outcomes, the only outcome that is not possible under SC is Outcome 1. The observed outcome can only happen if a loop occurs, which is a violation of consistency. For all other outcomes it is possible to find such an order



Total Store Order (TSO):
TSO relaxes the Store-to-load intra-thread dependency. Under this condition, the store and the load within the same thread are no longer ordered. As a consequence, all four outcomes are possible.

Relaxed Memory Ordering (RMO):
The RMO model is even more relaxed than TSO. But TSO already accepts all the outcomes, hence the case of RMO is identical to TSO for the case of Dekker's algorithm.

Message Passing Problem:

Synchronous version:

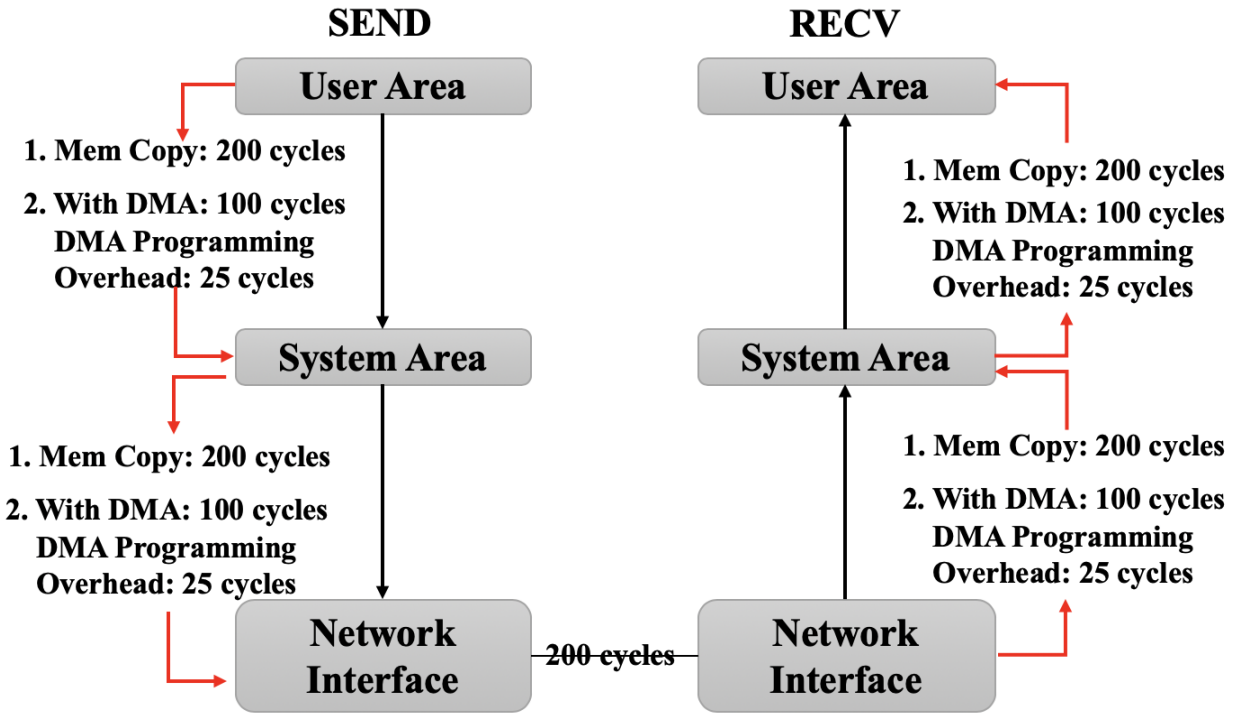
CODE FOR THREAD T0:
```
SEND(&B[1],sizeof(int),T1,SEND_B1);
RECV(&B[0],sizeof(int),T1,SEND_A1);
 <Unrelated computation;>
```

CODE FOR THREAD T1:
```
RECV(&A[0], sizeof(int),T0,SEND_B1);
SEND(&A[1],sizeof(int),T0,SEND_A1);
 <Unrelated computation;>
```

Asynchronous version:

CODE FOR THREAD T0:
```
ASEND(&B[1],sizeof(int),T1,SEND_B1);
 <Unrelated computation;>
ARECV(&B[0],sizeof(int),T1,SEND_A1);
```

CODE FOR THREAD T1:
```
        ASEND(&A[1],sizeof(int),T0,SEND_A1);
 <Unrelated computation;>
        ARECV(&A[0],sizeof(int),T0,SEND_B1);
```



Assume that the unrelated computation takes 500 cycles. The context switches between user-level and operating system level costs 100 cycles. Consider the following four scenarios:

(a). No special hardware support. How long does it take to execute the program with synchronous and asynchronous primitives, respectively?

(b). DMA programmed by O/S without support for user messages. What will be the new execution time of synchronous and asynchronous version respectively? Give the percentage of performance improvement with respect to problem(a).

(c). User level messages with O/S support and DMA. In this scenario, the message could be delivered directly to the user area from the network interface. However, incoming messages are taken care of by users instead of OS level, which costs context switches. What will be the new

execution time of synchronous and asynchronous version respectively? Give the percentage of performance improvement with respect to problem(a).

(d). User level messages with a dedicated message processor. This dedicated message processor is in the NIC hardware. Hence, it comes for free and takes care of the message delivery. What will be the new execution time of synchronous and asynchronous version respectively? Give the percentage of performance improvement with respect to problem(a).

Solution to Message Passing Problem

Solution to (a):

Synchronous: execution time = 1000(SEND+RECV) + 1000(SEND+RECV) + 500(unrelated) = 2500 cycles
Asynchronous: execution time = 1000(SEND+RECV) cycles

Solution to (b):

Synchronous: execution time = 600(SEND+RECV) + 600(SEND+RECV) + 500(unrelated) + 4 x 25(DMA) = 1800 cycles

Performance improvement: (2500 - 1800) / 2500 = 28%

Asynchronous: execution time = 600(SEND+RECV) + 4 x 25(DMA) = 700 cycles

Performance improvement: (1000 - 700) / 1000 = 30%

Solution to (c):

Synchronous: execution time = 100(context switch) + 400(SEND+RECV) + 100(context switch) + 400 (SEND+RECV) + 500(unrelated) + 2 x 25 (DMA) = 1550 cycles

Performance Improvement: (2500 - 1550) / 2500 = 38%

Asynchronous: execution time = 100(context switch) + 400(SEND+RECV) + 2 x 25(DMA) = 550 cycles

Performance Improvement: (1000 - 550) / 1000 = 45%

Solution to (d):

Synchronous: execution time = 100(context switch) + 400(SEND+RECV) + 100(context switch) + 400 (SEND+RECV) + 500(unrelated) = 1500 cycles

Performance Improvement: (2500 - 1500) / 2500 = 40%

Asynchronous: execution time = 100(context switch) + 400(SEND+RECV) = 500 cycles

Performance Improvement: (1000 - 500) / 1000 = 50%