Name: **Seat Number:** **Sciper:**

CS-302 Midterm Exam

Please do not turn this page until instructed to do so.

Please write your seat number at the top of each page.

You have **150 minutes** in total to answer all questions.

Please write clearly and concisely using a **blue** or **black** pen. Show all work for full credit.

There is an empty sheet at the end that you can use as scratch paper. Please use that first and only ask for extra sheets if you need more.

Total number of pages: **16**

Problem	**Points**
Miscellaneous [29 points]	
Shared Memory [21 points]	
Message Passing [21 points]	
Consistency [29 points]	
Total [100 points]	

Seat Number:

Miscellaneous [29 points]

1. You are building software that consists of two components: a Python codebase and a Large Language Model (LLM). Runtime profiling shows that your software spends 70% of its total execution time on the LLM and the remaining 30% on the Python codebase.

   Currently, your software runs using one GPU (costing 5000 CHF) for the LLM and one CPU (costing 2500 CHF) for the Python codebase. You've received additional funding to speed up the software, and you have two options:

   Option A: Speed up the LLM by paying for more GPUs.
   Option B: Speed up the Python codebase by paying for more CPUs.

   Speedup and cost scale linearly with the number of GPUs (for option A) and CPUs (for option B). Fractional units are allowed. For example, using 1.5 GPUs yield a 1.5x LLM speedup and costs 7500 CHF to run the LLM. Answer the following questions:

   a. To achieve a 20% overall speedup of your software, calculate: [4 points]
      i. The total cost while only speeding up the LLM
      ii. The total cost while only speeding up the Python codebase

      Option A speedup needed: [1 point]

$$1.2 = \frac{1}{\frac{0.7}{x} + 0.3} => x = 1.3$$

      So, cost needed = 1.3 * 5000 + 1 * 2500 = 9000 CHF [1 point]

      Option B speedup needed: [1 point]

$$1.2 = \frac{1}{0.7 + \frac{0.3}{x}} => x = 2.25$$

      So, cost needed = 1 * 5000 + 2.25 * 2500 = 10625 CHF [1 point]

   b. What is the maximum overall speedup that option A and option B can achieve individually assuming an unlimited budget? [2 points]

      Option A maximum speedup = (1 / 0.3) = 3.3x [1 point]
      Option B maximum speedup = (1 / 0.7) = 1.42x (42%) [1 point]

Seat Number:

2. Consider the following snippet of pseudo-assembly code running on a scalar machine: (Assume a word and a register is 32 bits, and memory is byte-addressable)

```
; vecA[] -> r1, vecB[] -> r2, vecC[] -> r3
; result[] -> r4
; &vecA[N] -> r5
loop:
    lw r6, 0(r1)    ; load vecA[i]
    lw r7, 0(r2)    ; load vecB[i]
    lw r8, 0(r3)    ; load vecC[i]
    mul r6, r6, r7  ; vecA[i]*vecB[i]
    add r6, r6, r8  ; vecA[i]*vecB[i] + vecC[i]
    sw  r6, 0 (r4)  ; result[i] = vecA[i]*vecB[i] + vecC[i]
    add r1, r1, 4
    add r2, r2, 4
    add r3, r3, 4
    add r4, r4, 4
    bne r1, r5, loop
```

The arrays **vecA**, **vecB**, **vecC** and **result** contain 8192 elements each. Assume:
(1) All of these arrays are aligned and stored sequentially in memory
(2) The scalar machine contains a 10-entry ROB and a 4-way fully set associative cache with 64-byte cache blocks.

Answer the following questions:

a. Count the total number of instructions executed by the scalar machine. [2 points]

In each iteration, there are 11 instructions. [1 point]
So, total number of instructions = 11 * 8192 = 90112 [1 point]

b. Count the total number of cache misses incurred by the scalar machine. [3 points]

There are four vector elements to be loaded per iteration. [1 point]
One miss per cache block = One miss per 16 elements [1 point]
Number of misses = 4*8192/16 = 2048 [1 point]

c. Assume that a vectorized version of the code snippet runs on a vector machine with 1024-bit vector registers. Count the total number of instructions executed by the vector machine. [2 points]

Number of elements per vector register = 1024/32 = 32 [1 point]
So new number of iterations = 90112/32 = 2816 [1 point]

Seat Number:

3. The following list describes a sequence of memory accesses performed by cores C0 and C1 in a 2-core processor. Each core has a private cache. The private caches are connected to each other and to memory by a bus-based interconnect and the coherence protocol followed is MESI. "ST X" and "LD X" denote stores/loads involving address X. Each character represents a unique address. Assume that (1) all addresses map to different cache sets, so no evictions can happen and (2) no two cache accesses overlap in time.

```
1  C0: LD A
2  C1: LD A
3  C0: ST A
4  C0: ST B
5  C1: LD B
6  C1: ST B
7  C0: LD B
```

Assuming that the private caches of each core are initially empty, answer the following questions:

a. List all remote actions generated during the sequence of memory accesses. [9 points] The set of possible remote actions are: BusRd, BusRdX, BusInv and DataWB.

Memory Access	Remote Action(s)
`C0: LD A`	BusRd
`C1: LD A`	BusRd
`C0: ST A`	BusInv
`C0: ST B`	BusRdX
`C1: LD B`	DataWB, BusRd
`C1: ST B`	BusInv
`C0: LD B`	DataWB, BusRd

b. List the state of the cache blocks corresponding to addresses A and B in each of the private caches after each memory access. [7 points]
(Note: Block A refers to the cache block corresponding to address A and Block B refers to the cache block corresponding to address B)

Memory Access	Private cache of C0		Private cache of C1	
	Block A	Block B	Block A	Block B
`C0: LD A`	E	-	-	-
`C1: LD A`	S	-	S	-
`C0: ST A`	M	-	I	-
`C0: ST B`	M	M	I	-
`C1: LD B`	M	S	I	S
`C1: ST B`	M	I	I	M
`C0: LD B`	M	S	I	S

Seat Number:

Shared Memory [21 points]

4. Consider a matrix multiplication **`matC = matA * matB`** where,
   - **`matA`** is of dimension M x K
   - **`matB`** is of dimension K x N
   - **`matC`** is of dimension M x N

   All three matrices contain 32-bit integer elements and are stored in row-major format. Below, there are two possible code snippets to implement the matrix multiplication: (assume all elements in **`matC`** are initialized to zero)

   Snippet 1:

```
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++) {
         for(int k = 0; k < K; k++)
              matC[i][j] += matA[i][k] * matB[k][j];
    }
}
```

   Snippet 2:

```
for(int k = 0; k < K; k++) {
    for(int j = 0; j < N; j++) {
         for(int i = 0; i < M; i++)
              matC[i][j] += matA[i][k] * matB[k][j];
    }
}
```

   Assuming the code snippets are running on a machine with a 256KB private cache with 64B cache blocks for each core, answer the following questions:

   a. For a single-threaded implementation, which of the two code snippets would you expect to finish faster and explain why for the following two cases. [4 points]

      Case 1: K >> M, N and M, N < 200
      Case 2: M >> N, K and N, K < 200

      Case 1: When K >> M, N then snippet 2 [1 point] offers better data locality [1 point] in the cache leading to better performance.

      Case 2: When M >> N, K, then snippet 1 [1 point] offers better data locality [1 point] in the cache leading to better performance.

b. Assume you want to parallelize the matrix multiplication using OpenMP. Therefore, you simply add only a **`#pragma omp parallel for`** before the outermost loop in the two code snippets. Answer the following questions:

   i. Is there a possibility of functionally incorrect program behavior in the parallelized version of the two code snippets? Explain your answer. [3 points]

   Snippet 1 does not have any risks. [1 point]

   Snippet 2 has a risk [1 point]. It will have data races because threads will try to update the same output matrix element. [1 point]

   ii. What modification (if any) would be needed to make the parallelized version of the two code snippets correct? List modifications (if any) for each code snippet separately. [2 points]

   Snippet 1 requires no modification [1 point]

   Snippet 2 requires a #pragma omp atomic for the update OR a collapse(2) clause [1 point]

   iii. After applying the necessary modifications (if any) for correctness, do either of the two parallelized code snippets suffer from true sharing or false sharing? If so, identify which one(s), specify the type of sharing, and explain your reasoning. [4 points]

   Snippet 1 suffers from false sharing [1 point] because threads can update elements in the same row that may belong to the same cache block [1 point]

   Snippet 2 suffers from true sharing and false sharing [1 point] because different threads are trying to access the same variable or neighboring elements. This is a problem of the algorithm. [1 point]
   OR
   [If answer to ii was collapse(2)] Then Snippet 2 suffers from false sharing [1 point] because threads can update elements in the same row [1 point]

5. Assume you have two matrices **`matA`** and **`matB`**, each of size N x N. All elements are 32-bit integers, and both matrices are properly initialized and stored in row-major format. Consider the following code snippet which transposes **`matA`** and stores it in **`matB`**.

```
#pragma omp parallel for
for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
            matB[j][i] = matA[i][j];
    }
}
```

Assuming the code snippet runs on a machine with 64-byte cache blocks and N >> 64, answer the following questions:

a. Identify and explain the two main performance bottlenecks in the code snippet. [4 points]

Bottleneck 1: Poor cache locality for matB [1 point]
The same thread has to access elements in different rows which will likely lie in different cache lines. [1 point]

Bottleneck 2: False sharing between threads [1 point]
Various threads will try to update elements in the same row which likely lie in the same cache line leading to false sharing [1 point]

b. Propose a single solution to overcome the two bottlenecks. Explain how the solution overcomes the bottlenecks and how would you choose any parameters (if applicable) for your solution. [4 points]

The solution is to use tiling [1 point]. Separate the matrices A and B into tiles and transpose each tile separately [1 point]. Allocate one tile per thread [1 point]. The total size of the tile (number of elements * size of elements) should be roughly equal to the private cache size of each core. [1 point]

Seat Number:

Message Passing [21 points]

6. Answer the following questions in the context of multiprocessors.

   a. What execution model does MPI implement? List three advantages of message passing over shared memory. [4 points]

   MPI implements the Single Program Multiple Data (SPMD) model [1 point].

   Three advantages: [1 point each]
   * Multi-node scalability
   * Performance transparency (more explicit control over data movement)
   * Fault tolerance

   b. Explain the tradeoffs involved in using non-blocking calls over blocking calls. [2 points]

   * Higher performance because no need to wait [1 point]
   * More programmer effort needed to ensure correctness [1 point]

   c. What are the advantages of MPI collectives over point-to-point communication? Are there any use cases where point to point communication is optimal to use? [3 points]

   Advantages: [1 point each]
   * MPI collectives are highly optimized
   * MPI collectives reduce programmer effort

   Use point to point communication over collectives when data transfer patterns are irregular and unpredictable [1 point]

Seat Number:

7. Consider the following code snippet. A master process (rank 0) sends data to two workers (rank 1 and rank 2). Worker 1 performs addition and Worker 2 performs multiplication and then each of them send the result back to the master process. Assume the variable **data** is properly initialized for each process.

```
#define CMD_WORK 100
#define CMD_DONE 200
...
if (rank == 0) {
   int work1 = 42, work2 = 24;
   MPI_Send(&work1, 1, MPI_INT, 1, CMD_WORK, MPI_COMM_WORLD);
   MPI_Send(&work2, 1, MPI_INT, 2, CMD_WORK, MPI_COMM_WORLD);

   for (int i = 0; i < 2; ++i) {
       MPI_Recv(&data, 1, MPI_INT, 1, CMD_DONE, MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
    }
}
else if (rank == 1) {
   MPI_Recv(&data, 1, MPI_INT, 0, CMD_WORK, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
    data += rank;
   MPI_Send(&data, 1, MPI_INT, 1, CMD_DONE, MPI_COMM_WORLD);
}
else if (rank == 2) {
   MPI_Recv(&data, 1, MPI_DOUBLE, 0, CMD_DONE, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
   data *= rank;
    MPI_Send(&data, 2, MPI_INT, 2, CMD_DONE, MPI_COMM_WORLD);
}
```

There are six bugs in the code snippet. List all of the bugs and propose the corrections needed to make it a correct program. [12 points]

Bug 1: master process's Recv has wrong receiver ID
Fix 1: MPI_Recv(&data, …, i+1, …, MPI_STATUS_IGNORE);

Bug 2: worker 1's Send has wrong sender ID
Fix 2: MPI_Send(&data, …, 0, …, MPI_STATUS_IGNORE);

Bug 3: worker 2's Recv has wrong sender ID
Fix 3: MPI_Send(&data, …, 0, …, MPI_STATUS_IGNORE);

Bug 4: worker 2 needs to send one data item not two
Fix 4: MPI_Send(&data, 1, …)

Bug 5: worker 2 is expecting wrong tag
Fix 5: MPI_Recv(&data, …, CMD_WORK, …);

Bug 6: worker 2 is expecting to receive double datatype but should be int
Fix 6: MPI_Recv(&data, …, MPI_INT, …);

Seat Number:

Consistency [29 points]

8. Answer the following questions in the context of consistency models:

   a. What does it mean for a program to be data-race-free? Does the following snippet of code contain a data race? Explain your answer. [4 points]

```
// Thread 0                  // Thread 1
// x = 0, y = 0;             // x = 0, y = 0

if (x == y) y++;             if (x != y) x++;
```

   A data race is when two conflicting operations from different threads occur simultaneously [1 point]. Two operations conflict if they access the same location, and at least one is a write [1 point].

   Yes, the program contains a data race. [1 point]
   One thread is trying to modify a variable that another thread is trying to read without using any atomics [1 point]

   b. In C++, does declaring a variable as **`volatile`** ensure sequentially consistent (SC) execution on an ARMv8 machine? If yes, explain how. If no, explain how to achieve SC execution on an ARMv8 machine. [2 points]

   No, simply using volatile in C++ does not help as it only prevents the compiler from reordering/ordering instructions, the hardware can still reorder instructions. [1 point]

   The variable needs to be declared as atomic so that a fence is inserted into the pipeline. [1 point]

   c. How can you detect data races in Java using Happens Before relationships? How does the **`volatile`** keyword help prevent data races in Java? [3 points]

   If you can find two actions Ax so that Ax does not $\rightarrow$ Ay [1 point] , and Ay does not $\rightarrow$ Ax [1 point], you have found a data race

   Volatile in Java guarantees a $\rightarrow$ between the write and read to a variable [1 point]

Seat Number:

9. Assume the given program shown below runs on a single-core processor that implements two different memory consistency models:
   1. Modified sequential consistency model that does not implement cache peeking but uses a Store Buffer (SB).
   2. Weak consistency model that implements cache peeking and also uses the Store Buffer (SB).

   The specifications of the processor are as follows:
   1. 2-entry store buffer (SB) and 2-entry load/store queue (LSQ)
   2. 1-cycle execution latency for a cache hit
   3. 100-cycle execution latency for a cache miss
   4. Memory fence operations take 1-cycle to resolve
   5. At most only one instruction can enter, and at most only one instruction can exit each of the LSQ and the SB in a given cycle.
   6. A single instruction cannot enter and leave the same structure in the same cycle.
   7. Instructions are removed from the LSQ and SB based on the memory ordering specifications, and the ROB can accommodate all instructions shown in the program.

   Show the changes in the content of the LSQ and SB for the given program. **Indicate the cycle number at which each change occurs,** knowing that the first instruction enters the hardware structures at cycle 0. Please denote peeking operations when they happen by using '`(P)`' in your diagrams, for example, **`LD D (P)`**. The content of the LSQ and SB in cycle 0 is shown as a reference.

   Please note that "**`ST X`**" and "**`LD X`**" denote stores/loads involving address **`X`**, and "**`A`**", "**`B`**", "**`C`**", etc. each refer to distinct memory addresses. The actual value loaded/stored is irrelevant. You may have more space than the number of changes in the hardware structures.

   Executed program:
```
1  LD A
2  ST B
3  LD B
4  ST C
5  LD D
6  ST E
7  LD F
8  ST G
9  FENCE
10 LD G
11 LD H
```

   Initially, the cache only contains valid blocks corresponding to addresses **`A`**, **`C`**, and **`F`**. Assume that all addresses map to different cache sets, so no evictions can happen.

Seat Number:

a. **Modified Sequential Consistency [10 points]**

Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle101	Cycle102
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
			ST C	ST C	LD D
LD A	ST B	LD B	LD B	LD B	ST C
SB	**SB**	**SB**	**SB**	**SB**	**SB**
		ST B	ST B		

Cycle103	Cycle104	Cycle203	Cycle204	Cycle303	Cycle304
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
ST E	ST E	LD F	ST G	ST G	FENCE
LD D	LD D	ST E	LD F	LD F	ST G
SB	**SB**	**SB**	**SB**	**SB**	**SB**
ST C			ST E		

Cycle305	Cycle404	Cycle405	Cycle406	Cycle506	Cycle
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
LD G	LD G	LD H			
FENCE	FENCE	LD G	LD H		
SB	**SB**	**SB**	**SB**	**SB**	**SB**
ST G					

Cycle	Cycle	Cycle	Cycle	Cycle	Cycle
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
SB	**SB**	**SB**	**SB**	**SB**	**SB**

-5 points for a major mistake (e.g., bypassing inside LSQ)
-3 points for consistency model mistakes
-2 points for peeking/not peeking correctly and cache access mistakes
-1 points for a minor mistake

Seat Number:

b. Weak Consistency [10 points]

Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
					ST E
LD A (P)	ST B	LD B	ST C	LD D	LD D
SB	**SB**	**SB**	**SB**	**SB**	**SB**
				ST C	
		ST B	ST B	ST B	ST B

Cycle101	Cycle104	Cycle105	Cycle106	Cycle107	Cycle205
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
ST E	LD F	ST G	FENCE	LD G	LD G
LD D	ST E	LD F	ST G	FENCE	FENCE
SB	**SB**	**SB**	**SB**	**SB**	**SB**
		ST E		ST G	

Cycle206	Cycle207	Cycle306	Cycle	Cycle	Cycle
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
LD H					
LD G	LD H				
SB	**SB**	**SB**	**SB**	**SB**	**SB**

Cycle	Cycle	Cycle	Cycle	Cycle	Cycle
LSQ	**LSQ**	**LSQ**	**LSQ**	**LSQ**	**LSQ**
SB	**SB**	**SB**	**SB**	**SB**	**SB**

Seat Number:

Seat Number: