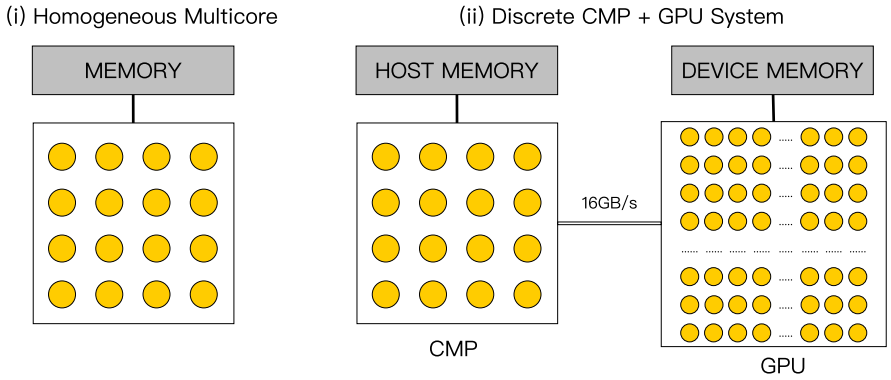
Solutions for the Exam in EDA284 (Chalmers) and DIT430 (GU) Parallel Computer Architecture, Saturday, March 21st, 2020, 8:30h - 12:30h

Problem 1 (12 points)

A team of researchers is considering purchasing a new computing system to run a scientific application in the fastest possible time. The team has analyzed the application and observed that it consists of two phases: (a) a serial phase, which runs for 0.5 second on a single core, and (b) a parallel phase, which consists of 16G ($16\times1024\times1024\times1024$) floating point (FP) operations. All FP operations can be executed in parallel. The parallel phase has a streaming behavior and there is no reuse of data, so caches have no impact. The arithmetic intensity of phase (b) is 0.4 Flops/Byte according to the DRAM Roofline model. The team has two architecture choices: (i) a multicore (CMP) architecture with 16 single-threaded cores, and (ii) a two-chip system with the same homogeneous multicore CMP with 16 cores, a discrete GPU with 128 SMs (SIMT Core Clusters), and an interconnect bus between CMP and GPU with a fixed bandwidth of 16GB/s. See the figures below for diagrams of these two architectures.



The peak performance throughput for each core on the CMP is 0.5GFlops/s, and for each SM on the GPU it is 1GFlops/s. The memory bandwidth is 16GB/s for the CMP and 128GB/s for the GPU. The processed dataset must be present in the host memory after the computation has finished. When using the interconnect bus, you may consider that other connection latencies such as link setup or message assembly are negligible. Overlapping of communication and computation is not possible, i.e. these two phases must happen at different times. Finally, note that the serial phase can only be executed on a CMP core, never on the GPU. In order for the team to select the best architecture and the best configuration (i.e. number of cores or SMs), your task is to find the case that leads to the minimum total execution time:

(a) What will be the execution times of architecture (i) when using 4, 8 or 16 cores? Show the steps to reach the results.

(b) Next consider architecture (ii): What will be the execution times if the number of used SMs on the GPU is 32, 64 or 128? Given the results from (a) and (b), which configuration should be ultimately chosen? Discuss the impact considering both execution time and energy consumption.

(c) The memory on the CMP can be upgraded to a memory bandwidth of 32GB/s. If so, what will be the execution times of part (a)? Similarly, if a more advanced memory technology is used on the GPU, delivering 256GB/s, what will be the execution times of part (b)? Does this impact the team's choice of architecture and configuration?

(d) Considering the above results, what is the main bottleneck that needs to be addressed in order to improve the execution time in each architecture?

Answer to Problem 1:

(a) AI $\times$ CMP DRAM Bandwidth = 0.4 $\times$ 16 GB/s = 6.4 GFlops/s

When using 4 CMP cores:
Peak = 4 $\times$ 0.5 = 2 GFlops/s
Attainable GFlop/s = min(2, 6.4) = 2 GFlops/s => Compute-bound
Parallel part execution time = 16 GFlops / 2 GFlops/s = 8 s
Total execution time = 0.5 + 8 = 8.5 s

When using 8 CMP cores:
Peak = 8 $\times$ 0.5 = 4 GFlops/s
Attainable GFlop/s = min(4, 6.4) = 4 GFlops/s => Compute-bound
Parallel part execution time = 16 GFlops / 4 GFlops/s = 4 s
Total execution time = 0.5 + 4 = 4.5 s

When using 16 CMP cores:
Peak = 16 $\times$ 0.5 = 8 GFlops/s
Attainable GFlop/s = min(8, 6.4) = 6.4 GFlops/s => Memory-bound
Parallel part execution time = 16 GFlops / 6.4 GFlops/s = 2.5 s
Total execution time = 0.5 + 2.5 = 3 s

Conclusion: The best configuration is 16 cores.

(b) AI $\times$ GPU DRAM Bandwidth = 0.4 $\times$ 128GB/s = 51.2GFlops/s
Data copy time = 2 $\times$ 16GFlops / (0.4 $\times$ 16GB/s) = 5s

When using 32 SMs on GPU:
Peak = 32 $\times$ 1GFlops/s = 32GFlops/s
Attainable GFlop/s = min(32, 51.2) = 32GFlops/s => Compute-bound
Parallel part execution time = 16GFlops / 32GFlops/s = 0.5s
Total execution time = 0.5 + 0.5 + 5 = 6s

When using 64 SMs on GPU:
Peak = 64 $\times$ 1GFlops/s = 64GFlops/s
Attainable GFlop/s = min(64, 51.2) = 51.2GFlops/s => Memory-bound
Parallel part execution time = 16GFlops / 51.2GFlops/s = 0.3125s
Total execution time = 0.5 + 0.3125 + 5 = 5.8125s

When using 128 SMs on GPU:
Peak = 128 $\times$ 1GFlops/s = 128GFlops/s
Attainable GFlop/s = min(128, 51.2) = 51.2GFlops/s => Memory-bound
Parallel part execution time = 16GFlops / 51.2GFlops/s = 0.3125s
Total execution time = 0.5 + 0.3125 + 5 = 5.8125s

Conclusion: If taking the energy consideration into account, then we should choose 64 SMs as the power consumption will be lower than 128SM, but performance will be the same.

(c) CMP:

If the CMP memory bandwidth increases to 32GB/s, then AI $\times$ CMP DRAM Bandwidth = 0.4 $\times$ 32 GB/s = 12.8 GFlops/s

When using 4 CMP cores:
Peak = 4 $\times$ 0.5 = 2 GFlops/s
Attainable GFlop/s = min(2, 12.8) = 2GFlops/s => Compute-bound
Parallel part execution time = 16GFlops / 2GFlops/s = 8s
Total execution time = 0.5 + 8 = 8.5s

When using 8 CMP cores:
Peak = 8 $\times$ 0.5 = 4 GFlops/s
Attainable GFlop/s = min(4, 12.8) = 4 GFlops/s => Compute-bound
Parallel part execution time = 16GFlops / 4GFlops/s = 4s
Total execution time = 0.5 + 4 = 4.5s

When using 16 CMP cores:
Peak = 16 $\times$ 0.5 = 8 GFlops/s
Attainable GFlop/s = min(8, 12.8) = 8 GFlops/s => Compute-bound
Parallel part execution time = 16GFlops / 8GFlops/s = 2s
Total execution time = 0.5 + 2 = 2.5s

Conclusion: The best configuration is 16 cores.

GPU:
AI $\times$ GPU DRAM Bandwidth = 0.4 $\times$ 256GB/s = 102.4GFlops/s
Data copy time = 2 $\times$ 16GFlops / (0.4 $\times$16GB/s) = 5s

When using 32 SMs on GPU:
Peak = 32 $\times$ 1GFlops/s = 32GFlops/s
Attainable GFlop/s = min(32, 102.4) = 32GFlops/s => Compute-bound
Parallel part execution time = 16GFlops / 32GFlops/s = 0.5s
Total execution time = 0.5 + 0.5 + 5 = 6s

When using 64 SMs on GPU:
Peak = 64 $\times$ 1GFlops/s = 64GFlops/s
Attainable GFlop/s = min(64, 102.4) = 64 GFlops/s => Compute-bound
Parallel part execution time = 16GFlops / 64GFlops/s = 0.25s
Total execution time = 0.5 + 0.25 + 5 = 5.75s

When using 128 SMs on GPU:
Peak = 128 $\times$ 1GFlops/s = 128GFlops/s
Attainable GFlop/s = min(128, 102.4) = 102.4GFlops/s => Memory-bound
Parallel part execution time = 16GFlops / 102.4GFlops/s = 0.15625s
Total execution time = 0.5 + 0.15625 + 5 = 5.65625s

Conclusion: The best configuration is 128 SMs.

(d) In the case of architecture (ii), although the GPU is very fast, it is severely limited by the slow interconnect (16GB/s). Hence, to improve performance the focus should be

on obtaining a faster interconnect. Note that the CPU memory subsystem needs to support at least the same BW as the interconnect. For the case of architecture (i), both peak core performance and memory bandwidth need to improve to lower the execution time.

Problem 2 (10p)

The invalidate-based MESI cache coherence protocol state diagram was introduced in Lecture 4 - Slide 22. You are given the following machine code

```
1 (1) LW z1, 0(z4) // LW: Load Word
2 (2) LW z2, 0(z5)
3 (3) SW z3, 0(z5) // SW: Store Word
4 (4) SW z2, 0(z4)
5 (5) LW z1, 0(z5)
```

Assume that the addresses in z4 and z5 map to different cache lines. The code is executed by two cores, namely C0 and C1. Each has a private L1 cache. Following is the instruction sequence in real time:
C0(1), C0(2), C1(1), C1(2), C0(3), C1(3), C1(4), C0(4), C0(5), C1(5)
Here, C0(1) means: C0 executes instruction 1.

Core(Instruction)	State z4 - C0	State z5 - C0	State z4 - C1	State z5 - C1
C0(1)				
C0(2)				
C1(1)				
C1(2)				
C0(3)				
C1(3)				
C1(4)				
C0(4)				
C0(5)				
C1(5)				

(a) For the table above, fill in the MESI state of the L1 cache lines for z4 and z5 at C0 and C1 after each call in the sequence.

(b) For this code, and assuming the system is implemented as a Chip Multiprocessor (CMP), which protocol is preferred: MOESI or MESI? Why?

(c) Assume a cache line size of 32 bytes. This time, memory locations mapped by z4 and z5 are accessed in offsets of the processor numbers (C0.offset = 0 x 2 = 0. C1.offset = 1 x 2 = 2).
Hence, instruction (3) becomes:
`SW z3, 0(z5)` on C0.
`SW z3, 2(z5)` on C1.
Do you see a potential problem? Explain in 1 line, and modify either instruction to fix the problem.

Answer to Problem 2:

(a)

Core(Instruction)	State z4 - C0	State z5 - C0	State z4 - C1	State z5 - C1
C0(1)	E	I	I	I
C0(2)	E	E	I	I
C1(1)	S	E	S	I
C1(2)	S	S	S	S
C0(3)	S	M	S	I
C1(3)	S	I	S	M
C1(4)	I	I	M	M
C0(4)	M	I	I	M
C0(5)	M	S	I	S
C1(5)	M	S	I	S

(b) MOESI is better in this case due to the fact that the load in C1(5) can be served by C0 cache that is in Owned state. In MESI, C0 has to update the main memory before sharing the modified value. If there are more stores, a write to a cache line owned by another core causes the owning cache to write back to main memory and invalidates all copies, before the requesting cache modifies the value.

(c) False sharing, as C0 and C1 access the memory location z5 within 32 byte on the same cache line causing unnecessary invalidation. One way to fix is allocating on different cache lines, which should result, for example, in instruction (3) being `SW z3, 32(z5)` for C1.

Problem 3 (10p)

In the design exploration of a new chip-multiprocessor system that is designed to have 512 cores, the architecture team decides to use a clustered (hierarchical) organization in which each cluster consists of 64 cores, each core has its own private L1 cache and the 64 cores share a L2 cache. The cache block size is 64 bytes.

(a) The architecture team first decides to use a presence-flag-based directory cache protocol inside each cluster and across clusters. Calculate the overhead of maintaining cache coherence in each L2 cache block for the intra-cluster protocol, and in each memory block for the inter-cluster protocol.

(b) Next the architecture team changes to use a limited-pointer directory cache protocol with four pointers inside each cluster, and a coarse-vector directory cache protocol that partitions the clusters into groups of four clusters. Calculate the overhead of maintaining cache coherence in each cache block at the L2 cache level (intra-cluster) and at the memory block level (inter-cluster).

For the intra-cluster interconnect the team is considering four choices of on-chip interconnection networks: a 1D linear array network (LA), a uni-directional ring (UR), a bi-directional ring (BR), and a N×N mesh network (NM), as exemplified in Figure 1 below for a smaller number of cores. The team has decided to use a directory-based (DB) protocol between the L1 caches as the intra-cluster coherence protocol. The latency to communicate between two cores connected directly by a link is 1 cycle. The router latency is one cycle per router that the packet goes through. The access time to the directory is a fixed 8 cycles. Assume that there is no contention in any link or the directory. In this exercise, latency is considered to be the time it takes to reach the destination cores, not including any acknowledgments.

Fig1.a

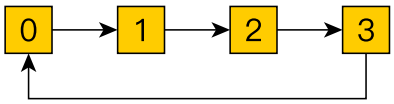


Fig1.b

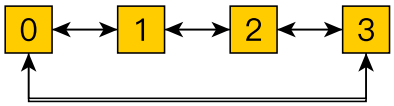


Fig1.c

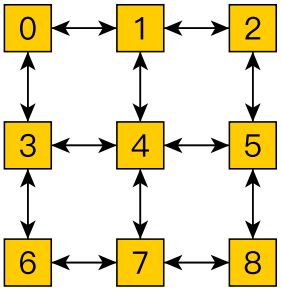


Fig1.d

(c) What are the worst-case and average-case latencies for a coherence message to reach all possible destinations (inside the cluster) for each of the four interconnect topologies? Assume that the requesting core is located in the position of core 0 in the above figure.

Answer to Problem 3:

(a) Intra-cluster protocol: there are 64 cores inside of each cluster, each presence-flag vector in L2 cache consists of 64 bits.
Inter-cluster protocol: there are 512 / 64 = 8 clusters, each directory entry at the memory contains 8 bits.
Thus, the overhead at the L2 cache is is 64 / (64$\times$8) = 12.5%, and the overhead at the memory is 8 / (64$\times$ 8) = 1.5%.

(b) Intra-cluster protocol: With 64 processors per cluster, each pointer contains 3 bits, the total is 3$\times$4 = 12 bits.
Inter-cluster protocol: We group the 8 clusters into 2 groups, each presence flag bit in the coarse vector represents a group. Hence there will be 2 bits in the presence flag vector.
Hence the intra-cluster protocol overhead is 12 / (64$\times$8)= 2.3%.
The overhead for the inter-cluster protocol is 2 / (64$\times$8) = 0.4%.

(c) LA-worst case: 8 + 63 cycles, LA-average case: 8 + 32 cycles
UR-worst case: 8 + 63 cycles, UR-average case: 8 + 32 cycles
BR-worst case: 8 + 32 cycles, BR-average case: 8 + 16 cycles
NM-worst case: 8 + 14 cycles, NM-average case: 8 + 7 cycles

Problem 4 (10p)

Modern parallel systems are built from multiple CMPs connected over a network, each having a complex cache hierarchy with multiple levels of private and shared caches. In this exercise, we ask you to propose and discuss several CMP-based system architectures tailored for several application scenarios. The only two design parameters to be considered are (1) the cache hierarchy (i.e. levels of private and shared caches), and (2) the hardware support for message passing.

The list of application scenarios is as follows:

(a) The workload consists of multithreaded SPMD programs (e.g. OpenMP), such that there is a high degree of inter-core data sharing but threads only update their own local data.

(b) The workload consists of multithreaded SPMD programs (e.g. OpenMP), such that there is high inter-core data sharing and threads update each other's data.

(c) The workload consists of message passing SPMD programs (e.g. MPI), such that each MPI process consists of a single thread and there is heavy communication between MPI processes.

(d) The workload consists of message passing SPMD programs (e.g. MPI), such that each MPI process is multithreaded and communication between MPI processes is light and infrequent.

(e) The workload consists of applications from unrelated users. Assume that each application uses only a single core and that each application has a different working set size.

Your task is to, for each of the five scenarios (a)-(e):

1. Propose a system architecture that can execute the workload targeting high performance and low energy (you may draw a diagram if you want), and
2. Briefly discuss why the selected cache hierarchy and message passing support are appropriate for each scenario

Answer to Problem 4:

In this question you are asked to propose CMP-based system architectures for various application scenarios. There is no single answer, the goal is to provide reasonable architectures and to be able to explain why it works. Below are sample architectures and answers that would be valid for this problem.

(a) For this scenario, a CMP architecture with a first level of private caches and a second level of shared caches would do the job. The private level would cache the mutable data and allow its access at low latency. As the mutable data is only local there would be no coherence overheads associated to its management (data will remain in 'M' state). Since the read-only data is shared by multiple threads, it is convenient to provide a shared L2 cache that will avoid the problem of replication. Note that even if no shared L2 is available, the architecture would still be reasonably good, since read-only data can be replicated without high coherence costs (exploiting 'S' state). No message passing hardware is required in this case.

(b) In this case in which data is accesses and update by all cores, it could be good to have only a shared level of caching. A single shared level will have higher latency, but it will avoid continuous invalidations and cache line migrations which would be expected according to the described access pattern. Again, no message passing hardware is needed.

(c) For this scenario we could consider clustered systems with a single core and a large cache, interconnected by message passing hardware using techniques such as DMA or message processors to accelerate the MPI transfers. Since the MPI processes are single threaded, there is no need for multiple cores.

(d) In this case a multicore processor with at least one level of private caches and one level of shared caches would be reasonable. The behavior of the multithreaded MPI processes is not described, so a general multicore architecture makes sense. Since MPI is infrequent, there is no need for expensive MPI hardware, and using a cheap interconnect such as Ethernet would suffice.

(e) This is a fairly common cloud usage scenario. There are many ways to execute this. One option would be a multicore with private-only caches to support the working set of each individual processor. However, because of the varying working set sizes this could lead to low cache utilitzation and high miss rates. Hence adding a large shared cache which dynamically assigns space to different cores would be a better options. This is in fact the common architecture that you will find in modern server platforms.

Problem 5 (10p)

The following code approximates an integral using middle Riemann sum.

```
1  typedef float* arr_t;
2  float integrate(arr_t x, arr_t fx, int bound1, int bound2){
3      float integration = 0;
4      for (unsigned int i = bound1; i < bound2 - 1; i++)
5          integration += ((fx[i] + fx[i + 1]) * 0.5) * ( x[i + 1] - x[i]);
6      return integration;
7  }
8  void read_heavy_io_from_file(arr_t& arr, int& N) {
9  // file processing
10 }
11 void display_graph(arr_t x, arr_t fx, int N, float integral) {
12 // image rendering
13 }
14 void main() {
15     arr_t x, fx; int N;                 // declare variables
16     read_heavy_io_from_file(x, N);      // read x values
17     read_heavy_io_from_file(fx, N);     // read fx values
18     float integral = integrate(x, fx, 0, N); // compute integration
19     display_graph(x, fx, N, integral); // display histogram
20 }
```

Below are your profiling results for each single call to the functions for $N = N_H$ on a single CMP:

Function Name	Time
`read_heavy_io_from_file`	1
`display_graph`	1
`integrate`	6

(a) You decide to parallelize the function `integrate` on $P = 8$ CMPs to speedup the solution for $N = N_H$. 1) What is your expected speedup (write the formula and substitute the terms). 2) What is the maximum speedup that can be achieved for $P = \infty$?

(b) You decide to run P such problems on P CMPs, i.e., the overall problem size solved on P CMPs is $N_P = P \times N_H$. What is the expected speedup achieved by running the Riemann sum of size N_P on P CMPs (as a function of P)? Note that the execution times shown by the table linearly scale with input size. Describe one advantage and one limitation of such parallelization technique.

(c) You are asked to enhance the ISA to support vector arithmetic instructions. Name at least 3 vector floating point arithmetic instructions needed to vectorize this code and describe their functionality.

(d) For the `integrate` function, assume a scalar code time $T_S = 8$. Having SIMD_LENGTH=128 bits, assume that the code is vectorized by the compiler and is fully scalable to the vector width. What is your expected vector code time T_V given that a `float` is 4 bytes.

Answer to Problem 5:

(a) Expected Speedup (for $N = N_H$) = $S_8 = \frac{T(1)}{T(8)} = \frac{2+1+6}{2+1+6/8} = \frac{9}{3.75} = 2.4$
Maximum Speedup (at $P = \infty$) = $\frac{1}{1/3} = 3$

(b) $S_p = 1 + F(P-1)$
Parallel Fraction= $F = \frac{6}{9} = \frac{2}{3}$.
$S_p = 1 + \frac{2}{3}(P-1)$
Alternative Solution:
$S_p = s + P(1-s)$
Serial Fraction= $s = \frac{3}{9} = \frac{1}{3}$.
$S_p = \frac{1}{3} + P\frac{2}{3} = \frac{1+2P}{3}$
Alternative Solution:
$S_p = P - s(P-1)$
Serial Fraction= $s = \frac{3}{9} = \frac{1}{3}$.
$S_p = P - \frac{1}{3}(P-1)$

Advantage: overcomes the weakness of Amdahl's law by scaling the problem size with the number of resources.
Disadvantage: such as increased overhead of communication, large memory requirements.

(c) 1) single precision vector add. 2) single precision vector subtract. 3) single precision vector multiplication. 4) single precision vector reduction (integration is a reduction variable).

(d) Ideal vector speed up in `integrate` = $\frac{128bits}{32bits} = 4$. Hence, $T(vector) = 8/4 = 2$

Problem 6 (8p)

Consider the following two barrier implementations written in C++:

Barrier #1:

```
1  // global variables
2  int counter = threadnum;  // threadnum is the number of threads in barrier
3
4  // per-thread function to call barrier
5  void barrier(){
6    counter--;            // decrease pending threads counter
7    while(counter){ };    // wait until all threads have arrived
8  }
```

Barrier #2:

```
1  // global variables
2  struct{
3     int f;  // local flag to indicate that a thread has reached the barrier
4  }flags [threadnum] __attribute__((aligned(64)));
5                     // initialized to {0, 0, 0, ...}, no false-sharing
6  bool done = false; // indicates if threads can be released from barrier
7
8  // per-thread function to call barrier
9  void barrier(){
10   int l_counter;   // counter used by thread 0 to count waiting threads
11   int thread_id = get_thread_id(); // thread_id is the calling thread's
12                                    // identifier = {0, ..., threadnum-1}
13   flags[thread_id].f = 1;  // announce #thread_id has reached barrier
14   if(thread_id == 0   // thread 0 checks the state of the barrier
15     while(!done){
16       l_counter = 0;
17       for(int i = 0; i < threadnum; i++)
18          l_counter += flags[i].f; // increment l_counter if flag is set
19       if(l_counter == threadnum) done = true; // release the barrier
20     }
21   else                // wait until all other threads have arrived
22     while(!done){ };  // and the barrier is released
23 }
```

These two barriers are to be used on a bus-based multiprocessor with private L1/L2 caches. Coherence is managed via a snoop-based protocol based on MSI-invalidate. Your task is to discuss the relative merits of these two implementations:

(a) Assume first that each line is executed atomically and behaves according to sequential consistency. How well does each of these barriers scale when additional cores are added to the system? Discuss the scalability of each barrier in terms of the cache coherence protocol.

(b) Now relax the assumption of atomic execution, as is the case with real hardware. Will the above codes still execute correctly? Are there any instructions that need special treatment from the hardware? If so, please identify those instructions and describe how the hardware should behave when encountering these special instructions.

(c) Finally, relax also the assumption of sequential consistency. Will barrier #2 operate correctly under relaxed memory consistency models?

Answer to Problem 6:

(a) **Barrier 1:** While the threads are looping, each will get a copy of the cache line in 'S' state. Under MSI-invalidate, when the next thread decreases the counter this will invalidate the contents of all the caches. The higher the number of cores the more frequent the number of invalidations, and also the larger the number of cores.
Barrier 2: Worst case: thread '0' arrives first at the barrier. It will then get all the cache lines in the 'S' state, and each thread arrival to the barrier will triggers the invalidation of this cache line, which will be subsequently reloaded. Since only one cache line is invalidated, this scheme is less costly, even in this worst case scenario.
Best case scenario: thread '0' is the last to arrive. In this case there is no invalidation to determine if all threads have arrived at the barrier, and only one invalidation broadcast when the variable 'done' is set to true.

(b) Without atomic execution the first barrier will not execute correctly, since it relies on the fact that `counter--` is an indivisible operation. To avoid this issue the `counter--` operation should be an atomic read-modify-write operation. When the hardware encounters this operation the cache controller invalidates all other copies and executes the decrement on the cache line while blocking all other requests to the cache line.
Barrier #2 does not require atomic instructions and will operate correctly under sequential consistency.

(c) Without sequential consistency, then some of the goals of using barriers would not be broken. For example, it would be possible that threads that have already been released from the barrier still do not observe writes from other threads that were performed before they reached the barrier. To avoid this, one option is to insert an ordering operation when the barrier is released. This could be for example a memory barrier after the `while(!done)` statement, or by marking the reads of variable 'done' as atomic. Also, to ensure that all the local loads and writes are globally performed before entering the waiting loop, the update of the flag (`flags[thread_id].f = 1`) should flush the store buffers and load queues. This can be achieved via an atomic write, or also by inserting a memory barrier.