15-418/618: Spring 2022 Exam #1

- You will prepare your answers using the answer spaces provided in this exam. Write your answers in the spaces provided; you may print and scan the exam or add your answers digitally.
- You will be given 48 hours to complete the exam, i.e., until 11:59pm EST on Saturday 2/26. Submit your exam to Gradescope.
- If there is some reason you do not believe you will be able to make this deadline, you must inform us by Monday 2/21 and get an explicit waiver from us.
- You may access any course materials but, unless explicitly told otherwise, *do not access any other materials.*
- All requests for clarifications must be made via *private* Piazza posts, and all clarifications will be responded to via a *public* FAQ within the first 24 hrs. Do *not* ask for clarifications in public.
- The exam will be designed so that someone with complete preparation can complete it in two hours. We are allowing 48 hours only to provide more flexibility, and to account for the fact that students may be operating in different time zones.

Problem 1	24 points
Problem 2	14 points
Problem 3	23 points
Problem 4	18 points
Problem 5	21 points
Total	**100 points**

Sign the following pledge:

> I do hereby swear that, in generating my answers to this exam, I made use of only course resources or others with explicit permission. I did not have any communication of any form with anyone other than the course instructors and teaching assistants about the contents of this exam.
>
> _____
>
> Your signature

Problem 1: Multiple choice [2pts each]

Be sure to read each of the following questions & answers carefully. Multiple answers may be correct.

L1 Why parallelism?

Which of the following is true about efficiency and speedup of a parallel programming model?

a. Speedup is limited by the parallelizable sections of code

b. Efficiency is always directly proportional to speedup with a fixed number of hardware resources

c. Assuming the same overall speedup is achieved, efficiency increases as number of processors decrease

d. Ideal speedup (i.e., N× speedup with N processors) is always possible

L2 ILP

Which of the following are true about CPU pipelining?

A. Pipelining improves the throughput of the CPU.
B. Pipelining reduces the latency of a single instruction's execution.
C. Data forwarding reduces the number of flushes in pipeline execution.
D. Pipelining speedups are not affected by the dependencies in code.

L3 Multicore

Which of the following statements about multicores are correct?

A. Adding local caches to each core in a multi-core architecture can reduce contention on the shared global main memory
B. Superscalar typically does better than multicore in hiding long memory latencies
C. A multicore with simpler cores is always better than a "single but fancy"-core given the same amount of transistors
D. It is worthwhile to buy a larger multicore with more cores to speed up an an I/O-intensive program

L4 Parallel programming models

Which of the following statements is/are correct for different communication models?

A. Shared address space model is more scalable on a NUMA machine than on a uniform memory access machine
B. Shared variables must always be tightly synchronized among accessing threads to ensure correctness in shared address space model
C. ISPC program is an example of the data-parallel program.
D. In the message passing model, hardware support is required to organize and pass message

L5 Parallel programming basics

A program calls a function N times, each time requiring 5 units of work. If the code is parallelized, it also requires 3 units of work per function call to be run sequentially at the end of the program. What is the maximum number of processors, P, where the sequential code completes faster than the parallel code (assuming no additional overheads)?

(A) 1
(B) 2
(C) 3
(D) 4

L6 Work distribution and scheduling

What of the followings are true about workload assignments:

(A) For the grid solver example, block assignment is better than interleaved assignment when assigning tasks to multiple worker threads.

(B) Static assignments are not applicable when the task sizes are different.

(C) It is better to prioritize larger tasks during the dynamic assignment of multiple tasks across workers.

(D) Small task granularity is always preferred in dynamic workload assignments.

L7 GPUs & CUDA

Which of the following are true about CUDA and GPUs:

A. __syncthreads() is used as an barrier for all threads within the grid
B. Threads within the block can share data via thread-local memory
C. The size of warp is hardware specific by default but can be set by user
D. There can be data-race problem for threads within block as well as threads across blocks

L8 Locality, communication, and contention

Select all that are true about *Inherent* and *Artifactual* communication

A) In a parallel program, inherent communication arises from sending and receiving data that must be shared between multiple processors
B) Artifactual communication can be reduced by increasing the cache line size.
C) In a message-passing model, there is zero artifactual communication because we can always send the exact data required to another processor
D) Artifactual communication scales proportionally with granularity of communication.

L9 Application case studies

What is the main purpose of semi-static work assignment in Barnes-Hut?

(a) To keep data in GPU shared memory.
(b) To balance work among threads inexpensively.
(c) To avoid false sharing.
(d) To improve the accuracy of results.

L10 Workload-driven performance evaluation

Circle the statements that apply to trace-driven simulation:

A. Trace-driven simulation can be used to evaluate chip design features on proposed chips that have not been manufactured yet.
B. Producing the simulation data could take orders of magnitude longer to run than the actual workload that is being simulated.
C. Trace instrumentation can change patterns of contention and communication in the simulated workload.
D. The results of trace-driven simulation can be replayed to test multiple proposed architectures on the same execution trace.

L11 Snooping coherence

Which of the following is true regarding the E state in the MESI cache coherence protocol compared to the original MSI protocol?

A. A cache line in E state exclusively services read misses.
B. It reduces interconnect traffic when only one processor intends to read-modify-write an address of which no other cache has a valid copy.
C. It eliminates the need for flush operations.
D. It reduces the number of state transitions in most programs.

L12 Snooping implementation

Which of the following techniques help prevent deadlocks in the snooping-based multiprocessor implementation?

(a) Processors must be allowed to serve incoming transactions while waiting to issue requests
(b) A write that obtains exclusive ownership must be allowed to complete before exclusive ownership is relinquished
(c) Use separate queues for request and response transactions between L1 and L2 cache
(d) Use FIFO arbitration for bus access contention

Problem 2: SIMD / Multicore [14pts]

The following is a simple code snippet to calculate the value of {x^(-y)} for every (x, y) pairs in the input arrays X and Y, with a very naive sequential approach (there are certainly faster algorithms, but this one is chosen for simplicity). Assume no overflow/underflow.

```
for (int i = 0; i < N; i++) {
      float x = X[i];
      int y = Y[i];
      float z = 1.f / x;
      while (y > 1) {
            z /= x;
            y -= 1;
      }
      output[i] = z;
}
```

An ISPC SIMD implementation (without tasks) of the previous program is shown below:

```
foreach (i = 0 .. N) {
      float x = X[i];
      int y = Y[i];
      float z = 1.f / x;
      while (y > 1) {
            z /= x;
            y -= 1;
      }
      output[i] = z;
}
```

This SIMD program can achieve close to 8× speed up compared to the sequential program when all elements in array X are identical and all elements in array Y are identical (x, y are constant).

Problem 2.1: Data-dependent performance

Suppose we initialized array X with random values, but array Y still has identical values. Do you expect speedup to significantly change? Explain your answer. **[2pts]**

Suppose we randomly initialized array Y with random values, and array X still has identical values. Do you expect speedup to significantly change? Explain your answer. **[2pts]**

Problem 2.2: Multicore scaling

Contrast the ISPC version of the program with a multithreaded version running on a multicore without SIMD. We partition X and Y arrays, assigning (x, y) pairs to different threads.

First suppose that both X and Y are randomly initialized. What speedup do you expect with 8 cores? Contrast this speedup with the above SIMD approach. Is it better or worse? Why? **[4pts]**

Second, suppose that X and Y are randomly initialized but then sorted in increasing order. We find that our multithreaded implementation has a much lower speedup than in the first case. What kind of workload assignment do you think that we are using, block or interleaved? Please explain. **[2pts]**

*(In block assignment with P threads, thread i is responsible for elements i * N / P to (i + 1) * N / P - 1 of the X and Y arrays. In interleaved assignment, thread i is responsible for elements i + j * N / P for j = 0, 1, 2…)*

Problem 2.3: Work assignment

Now suppose each core of this multicore machine has its own L1 cache with a line size of 64 bytes and again both X and Y are randomly initialized.

How should we assign work amongst threads in the multithreaded implementation. Do you expect block or interleaved assignment to perform better? Explain your answer. **[2pts]**

Now consider how to assign work among SIMD lanes. Do you expect block or interleaved assignment to perform better? Explain your answer. **[2pts]**

(Assignments defined the same as above, except consider lane i instead of thread i. You may assume that N is much larger than the L1 cache size.)

Problem 3: Parallel programming [23pts]

Last week, one of the TAs told a 418/618 student about a wonderful parallel-computing joke that they wanted to tell the other course staff. They forgot who they told and they don't want to tell the joke to the other course staff if they have already heard the joke.

Problem setup

They decide the best way to decide whether to tell the joke is to run a large simulation that tracks how the joke spreads over time depending on which student they told. In this simulation, the TA represents the class as a graph where each person is represented by a node and an edge represents whether two people talk to each other.

At each timestep, every node communicates with its neighbors in the graph to determine whether the joke spreads (ie. work scales with the number of edges). Since this is very compute-intensive, the 418/618 TA decides that the only solution is to parallelize their simulation! They focus on how to decompose their simulation's work through partitioning the graph.

Parallel machines

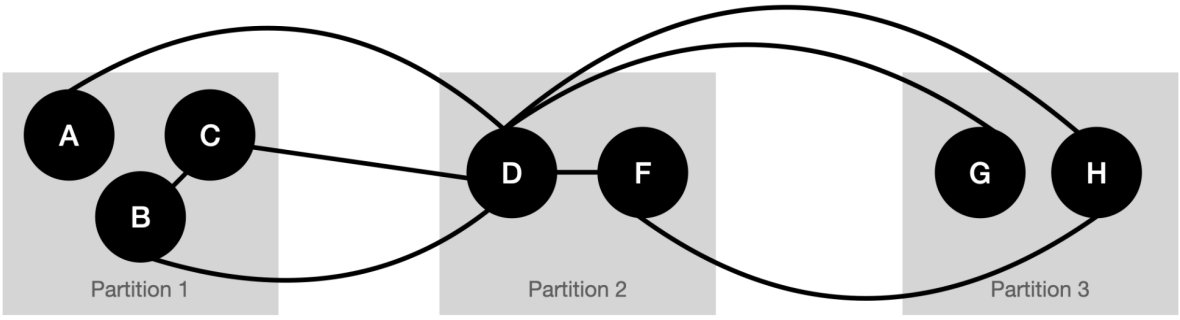
In addition, the TA has two machines that use different parallel programming paradigms and the TA decides to optimize their code for both machines.

The *Message-Passing Machine (MPM)* uses message passing between cores to share data. Each core is asked to simulate a small, connected partition of the graph. At each timestep, for every edge that connects nodes on two ***different*** cores, two messages have to be sent (one in each direction) which are expensive operations.

The *Data-Parallel Machine (DPM)* allows the programmer to spawn many tasks, which are each defined by a set of nodes that can be run in parallel. (A task is the same as a graph partition.) If an edge connects two nodes in the ***same*** task, this machine has a high conflict overhead because there is an internal dependency in the task and the machine has to communicate this. Assume that communication between tasks is relatively cheap on the DPM.

Problem 3.1: Naïve decomposition

First, the TA decides to statically partition the graph nodes into tasks evenly and randomly. To test out this decomposition, they look at communication cost for simulating one timestep of this subgraph:



How many messages will be passed in this subgraph on the Message-Passing Machine? **[1pt]**

How many conflicts will occur in this subgraph on the Data-Parallel Machine if each partition is a task? **[1pt]**

Problem 3.2: Fully-connected graphs

The TA next decides to test out their scheme on all of the course staff, who all know each other and, therefore, form a fully-connected graph (in a fully-connected graph, each node has an edge to every other node).

How many messages would need to be passed in a fully-connected graph with N nodes on the Message-Passing Machine? You may assume that the graph's nodes are divided evenly and randomly across all cores, and that the machine has C cores where N is a multiple of C. **[3pts]**
Hint: Consider how many edges there are in total and how many will end up within a partition.

How many conflicts would occur in a fully-connected graph with N nodes on the Data-Parallel Machine? You may assume that the graph's nodes are divided evenly and randomly across all cores, and that the machine has C cores where N is a multiple of C. **[1pt]**

Assume that each core must have the same number of nodes, is there a way to distribute nodes among cores to cause fewer messages? If so, explain one way to accomplish this and, if not, explain why not. **[1pt]**

Problem 3.3: Communication in more graphs

Satisfied with their experiments among the course staff, the TA now decides to test out their scheme on their 418/618 graph, which has a much less regular structure. However, they realize that randomly assigning nodes to a partition is probably not the best choice.

What should the TA optimize for when partitioning the graph to reduce communication in MPM? In DPM? **[2pts]**

If the TA focuses only on minimizing communication, what are two other performance bottlenecks could they run into for MPM? **[2pts]**

How about for DPM? **[2pts]**

Problem 3.4: Simulation scaling

The TA starts wondering how their simulation scales on the MPM. For all questions in this problem (unless specified otherwise), assume that the graph has *N* nodes, *e* edges per node, and that the TA partitions the graph randomly into *P* partitions such that *P* is also the number of cores in the machine. *Hint: Due to this division, you can assume* $\frac{e(P-1)}{P}$ *edges per node between cores.*

The TA thinks that the problem's scaling is problem-constrained. How does the communication scale if the TA decides to use *kP* cores instead of *P* cores? How does the arithmetic intensity scale in the same situation? **[4pts]**

The TA next realizes that the problem may be time-constrained if they start considering how the joke spreads outside of 418/618, but they still only have 1 week to run their simulations. In a time-constrained situation, how does the size of the graph (ie. the number of nodes that can be computed) scale as the number of cores increases to *kP* if *e* is held constant? Instead, if *e* is scales linearly with *N* (ie. *e* = *N/x* for some constant *x*), how does the graph size scale? How does the arithmetic intensity scale in both situations? **[6pts]**

Problem 4: GPUs [18 points]

Recall the render we implemented in the Assignment 2: Given a lot of circles, we want to render a 2D image where the pixel is blended by the colors of circles that cover it. A clever TA comes up with a new problem for the next semester. In this problem, the pixel value is just an integer (instead of a float vector of length 4), and each circle has an integer value as well. When it comes to the render, we want to fill the pixel value by taking the maximum value. For example, if a pixel is covered by circles with values (0, 1, 2, 3), then the pixel should be filled with value 3.

To help students get started, the TA provides the following starting pseudocode. It first creates a global array to store the best (higher) color of each pixel in the image. Then it assigns each thread to a circle. For each pixel inside the circle, it continuously updates the global array and re-shades the pixel with the maximum color.

The TA also provides 3 black-box helper functions listed below, similar to Assignment 2:

Listing 1: Initial pseudocode.

```
// returns true if the pixel lies in the current circle
__global__ bool isInCircle(float3 position, int pixelX, int pixelY);
// returns true if color1 is better than color2 (better ~= higher pixel value)
__global__ bool isBetter(int color1, int color2);

__global__ float maxColor[imageWidth * imageHeight];

__global__ void renderKernel() {

    // part1: get the circle indices within the block
    int circleIdx =  threadIdx.y * blockDim.x + threadIdx.x;
    int numThreads = blockDim.x * blockDim.y;

    if (circleIdx >= Params.numCircles) {
        return;
    }

    // Read image info and circle info
    short imageWidth = Params.imageWidth;
    short imageHeight = Params.imageHeight;
    short rad = Params.radius[circleIndex];
    float3 p = Params.position[circleIdx*3];
    float4 color = Params.color[circleIndex];

    // compute the bounding box of the current block
    short minX = max(min(static_cast<short>(imageWidth * (p.x - rad)),
                                            imageWidth),
                     0);
    short maxX = max(min(static_cast<short>(imageWidth * (p.x + rad)) + 1,
```

```
                                            imageWidth),
                    0);
    short minY = max(min(static_cast<short>(imageHeight * (p.y - rad)),
                                            imageHeight),
                    0);
    short maxY = max(min(static_cast<short>(imageHeight * (p.y + rad)) + 1,
                                            imageHeight),
                    0);

    // part2: render each pixel in the bounding box
    for (int pixelY = minY; pixelY < maxY; pixelY++) {
        for (int pixelX = minX; pixelX < maxX; pixelX++) {

            // check whether the pixel is in the circle
            if (!isInCircle(p, pixelX, pixelY))
                continue;
            // Q4.2: Is this correct?
            float targetColor = maxColor[pixelX, pixelY];
            if (isBetter(color, targetColor)) {
                ATOMIC { // low-level implementation elided.
                    maxColor[pixelX, pixelY] = color;
                }
            }
        }
    }

}

void render() {
    dim3 blockDim(BLOCK_X, BLOCK_Y, 1);
    dim3 gridDim(_____________________,
                          _____________________); // Q4.1
    renderKernel<<<gridDim, blockDim>>>();
}
```

Problem 4.1: Cuda basics

Before we can run the kernel, we need to find the appropriate number of threads per block and the number of thread blocks. In the method `render()`, fill in the gridDim here. (Number of circles is `Param.numCircles`) **[2pts]**

Problem 4.2: Correctness

Is the current implementation correct? (See comment in pseudocode.) Explain why or why not. **[2pts]**

A clever student finds that the pixel loop can be further parallelized (still in the same kernel). Each thread block is assigned an image block. In part 2, it computes, in parallel, over each pixel to perform the rendering. Rather than looping over all circles (i.e. in part 1), we precompute the potential circle indices by evaluating the circle intersection against the image block in parallel. (The intuition behind part 1 is that when the number of the circles is large, we can eliminate a lot unnecessary checks in part 2)

Here is the proposed pseudocode.

Listing 2: Modified pseudocode.

```
__global__ void renderKernelBetter() {
      // part1: get the indices and count for the circles has
      // intersection within the block
      int circleIdx =  threadIdx.y * blockDim.x + threadIdx.x;
      // BLOCK_SIZE: size of threadBlock but also image block working on
      int circleIndices[BLOCK_SIZE];
      int circleNums; // the number of intersected circles,
                      // which needs to be computed

      // get the circleIndices efficiently with exclusive scan
      ... // implementation emitted

      // part2:
      int pixelX = blockIdx.x * blockDim.x + threadIdx.x;
      int pixelY = blockIdx.y * blockDim.y + threadIdx.y;

      for (int i=0; i<circleNums; i++) {
            int circleIdx = circleIndices[i];
            Get circle information by circleIdx
            Check if the pixel is inside the circle
            Render the pixel value // TODO: is ATOMIC render still needed?
      }
}
```

Answer the following questions based on this implementation.

Problem 4.2: Performance

Is the ATOMIC inside the inner loop still needed? Why or why not? Is there any other synchronization that needs to be specified **in this kernel**? **[2pts]**

Which are the following changes may help to optimize the code with regards to run time performance. **[2pts]**

A. Use thread-local memory to restore the circleIndices.
B. Copy the image pixel values (to be rendered) into shared memory.
C. Swap the thread mapping of pixelX and pixelY, i.e. pixelX = blockIdx.y * blockDim.y + threadIdx.y. (Assume pixelX accesses image with stride of 1)
D. None of the above.

Problem 4.3: Workload Distribution

For the modified code in Listing 2, is there any workload imbalance amongst the threads in the same block in part 1? **[2pts]**

What about threads across blocks? **[2pts]**

How about cross block thread workload balance for part 2? **[2pts]**

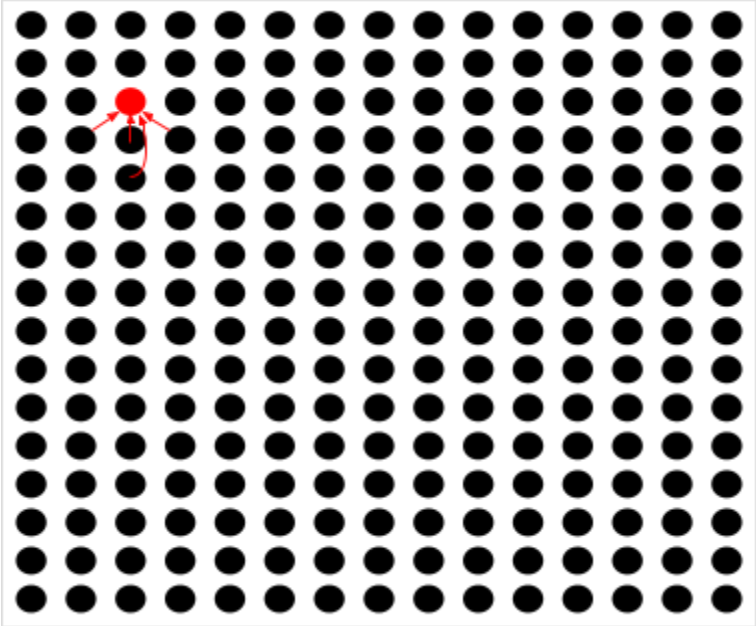
Problem 4.4: Exclusive Scan

A student proposed that it is more efficient to sort the circles by their values before calling the renderKernelBetter. Briefly describe how to sort the circles in parallel using cuda based on its color value **with exclusive scan**. *Hints: Assume atomic in-place arithmetic is supported in cuda (e.g., +=, -=), and assume most circles have different colors and that minColor and maxColor are given.* **[4pts]**

Problem 5: Communication and coherence [21pts]

This problem asks you to consider the grid computation depicted below. The computation iterates updating the grid until a termination condition is reached. Updating each cell of the grid requires reading from three neighbors on the row below, as well as the cell two rows directly below (as shown by the red arrows in the figure).

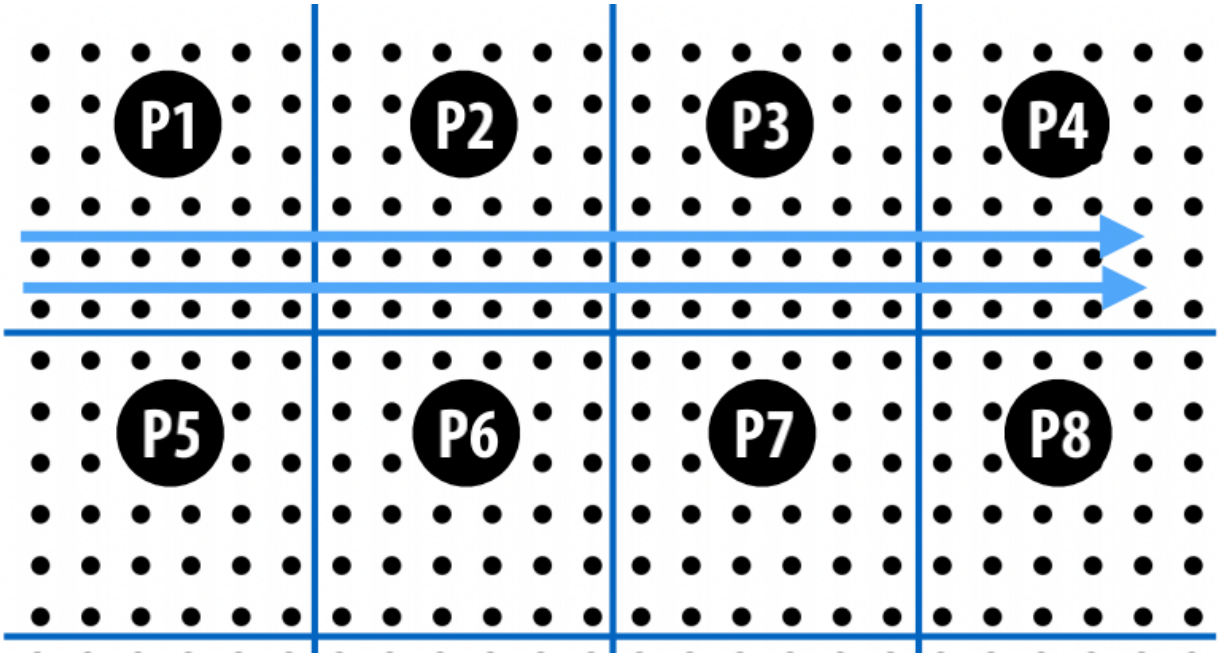
Suppose the grid has n rows and m columns.



Problem 5.1: Memory layout & Artifactual communication

Suppose a 12×24 grid is loaded into an 8-core shared memory system using a 2D row-major memory layout. Each processor has its own local cache and is assigned a 6×6 square block shown below. Assume that the cache line size is 4 grid elements. For convenience, each element in the grid is identified by a tuple that contains its row index and column index (e.g., the top-left element is identified by (1,1) and the right-bottom element (12, 24)).

List all elements that will be communicated between cores for P1 to compute (6,4). Which of them are inherent communication? Which of them are artificial communication? Explain your answer. **[1 pts]**



Now assume that the system uses an invalidation-based cache coherence protocol.
What is the **absolute worst-case** communication to update cell (6,4) in this problem? **[4 pts]**

What kind of cache misses are experienced in this worst case? (Assume it is not the first iteration of the grid solver.) **[2 pts]**

Is the worst case better or worse in a message-passing system? (In terms of number of bytes communicated.) Explain your answer. **[2 pts]**

Problem 5.2: Modifying the coherence protocol

To test whether iteration should continue, this grid solver computes whether the sum of the cells in the grid has converged to zero. (Suppose you know this will happen eventually because math.) Specifically, the program implements this operation via the following loop, which is run by all threads in parallel:

```
  std::atomic<int> gridTotal; // shared by all threads
  int grid[n][m]; // shared by all threads
  bool converged; // shared by all threads
  pthread_barrier_t barrier; // shared by all threads

  // …

  while (!converged) {

      // perform one iteration over the grid… (not shown)

      // now test convergence!
      // sum up all cells owned by this thread into global gridTotal
1     for (int r = myRowStart; r < myRowEnd; r++) {
2         for (int c = myColStart; c < myColEnd; c++) {
3             gridTotal += grid[r][c]
4         }
5     }
6
7     pthread_barrier(&barrier);
8
9     // thread 0 tests if the total is ~zero
10    if (tid == 0) {
11        if (gridTotal == 0) {
12            converged = true;
13        }
14    }
15    pthread_barrier(&barrier);
  }
```

You notice that this code performs poorly on your multicore CPU, and you identify the machine's MSI coherence protocol as the problem.

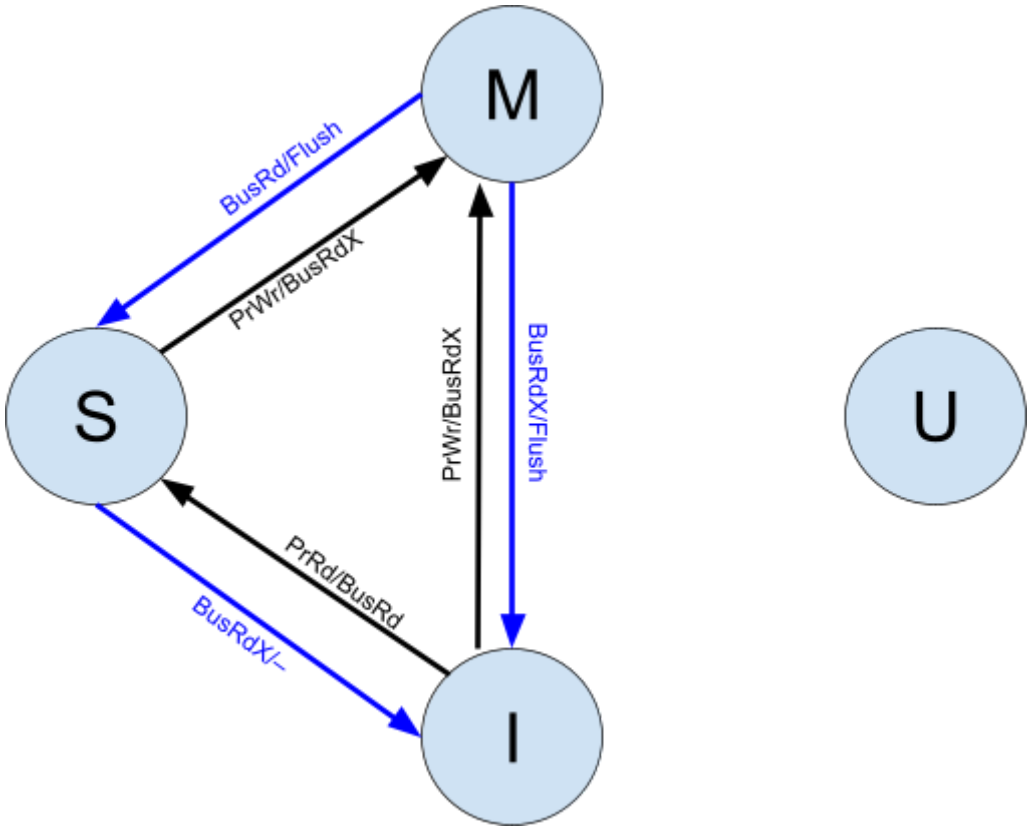
Which of the lines in the above code is the source of poor performance in MSI? Why? **[1pts]**

To solve this performance problem, you decide to design a better coherence protocol. You modify the system as follows:

- Your coherence protocol adds a new state called “U” (for ‘update’) in which the *only* operation a core can perform is integer addition.
- Your system adds a new bus operation called “BusReduce”, which causes other caches to flush their copies of the line. After completing a “BusReduce”, the originating cache controller can reduce (i.e., add up) the values sent by other controllers to get a global sum across all caches.
- You modify the processor to support a new “PrAdd” operation. PrAdd performs integer addition on a value in the cache, and it does not return any value to the processor. You also modify the compiler to generate PrAdd operations where appropriate.

Fill in the missing state transitions in the tables on the next page to implement the ‘MUSI’ protocol correctly so that the above program code runs efficiently **without any modification**. *(Hint: Multiple caches can have the same address in state U at the same time!)* Mark unreachable state/action pairs with an X. **[1pt each]**

Note: Unlike lecture slides, the diagram below does not show events where no action or state transition occurs. You may use the diagram to reason through your answer, but you must fill in the tables on the next page to get points on this problem. ***We will not grade this diagram.***



Current state	Event	Next state	Bus Action
M	PrRd	M	–
M	PrWr	M	–
M	PrAdd		
M	BusRd	S	Flush
M	BusRdX	I	Flush
M	BusReduce		
U	PrRd		
U	PrWr		
U	PrAdd		
U	BusRd		
U	BusRdX		
U	BusReduce		
S	PrRd	S	–
S	PrWr	M	BusRdX
S	PrAdd		
S	BusRd	S	–
S	BusRdX	I	–
S	BusReduce		
I	PrRd	S	BusRd
I	PrWr	W	BusRdX
I	PrAdd		
I	BusRd	I	–
I	BusRdX	I	–
I	BusReduce		