15-418/618: Spring 2021 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 4pm EST on Friday 3/12. 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 Tuesday 3/9 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 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	22 points
Problem 2	16 points
Problem 3	15 points
Problem 4	20 points
Problem 5	12 points
Problem 6	15 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 & short answer

2 points each. *Circle all correct answers.*

1. What is true about pipelining? ***A***
   - A. Pipelining improves the throughput of the CPU.
   - B. Pipelining reduces the latency of instruction execution.
   - C. Dependent instructions in the pipeline should wait for previous instructions to commit before they commit.
   - D. Pipelining is unaffected by branch misprediction.

2. Which of the following are true of latency and throughput bounds on CPU performance. ***D***
   - A. We should consider how to schedule dependent instructions one after the other when computing the *throughput bound*.
   - B. The issue rate of an execution unit is equal to 1 if the unit is fully pipelined.
   - C. Out-of-order execution means we could get rid of the dataflow graph because executions would be interleaved.
   - D. The slowest of either throughput bound or latency bound could be used as an approximation of long-run average performance.

3. Which of the following are true of multicore CPU processors? ***C***
   - A. Multi-threading could improve the performance of a single-threaded program.
   - B. A CPU spends its most chip area on the execution unit.
   - C. Hyper-threading allows multiple instruction streams to be executed on the same core.
   - D. Superscalar means running threads on multiple cores.

4. Which of the following techniques can reduce memory access stalls in a program? ***C,D***
   - A. Out of order execution.
   - B. Pipelined functional units.
   - C. A cache hierarchy.
   - D. Prefetching.

5. For each of the following assertions regarding CUDA, circle whether this is true of the threads within a block, or the blocks within a kernel, or both.

A. Can quickly & safely synchronize with others:	[✓ threads]	blocks	both
B. Can share data:	threads	blocks	[✓ both]
C. Have a native barrier operation:	[✓ threads]	blocks	both
D. May deadlock if one spins-waits on another:	threads	blocks	[✓ both]
E. May have data races:	threads	blocks	[✓ both]

6. You have a 4-stage superscalar processor pipeline. This pipeline has a 3-wide fetch stage, 2-wide decode stage, 4-wide execution stage, and a 3-wide commit stage. What is the maximum throughput of this pipeline?

Decode stage is the bottleneck. 2 instruction/cycle.

7. For Barnes-Hut, how does the performance of semi-dynamic work assignment change when the moving speed of stars increases dramatically, e.g., by several orders of magnitude?

The semi-dynamic assignment performs worse when the speed increases. The communication cost will always remain high when the speed is high, and the semi-dynamic assignment wouldn't reduce communication since the pattern changes very quickly.

8. You are running a parallel circle renderer on a 4 core CPU. You noticed that rendering a 1 million circle scene of size 40x30px takes 8 seconds. However, when switching to a 16 core CPU, you noticed that the renderer now takes < 1s to render the same scene. Is this expected? Briefly explain your answer.

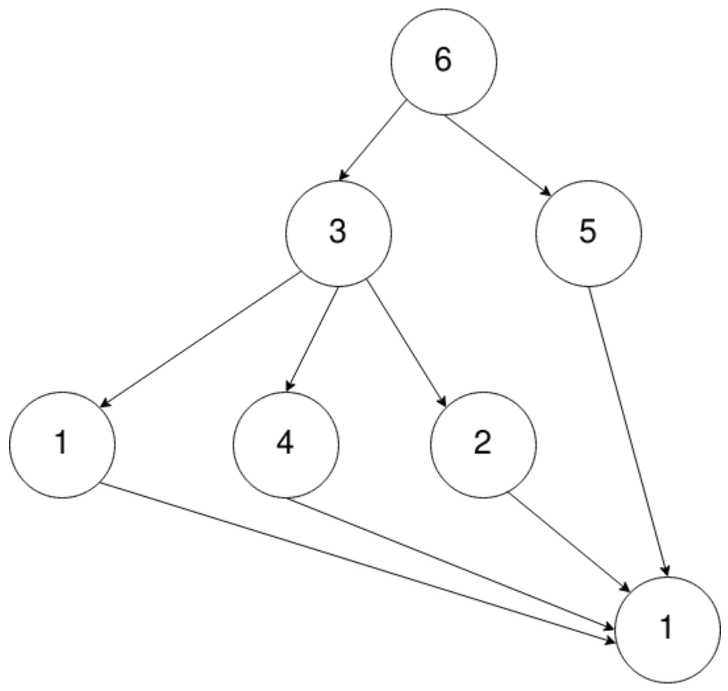
Yes. Superlinear speedup.

9. In a sentence or two, give an advantage of Cilk's fork-join parallelism over pthreads.

Clik uses a pool of worker threads and they steal work from a queue.. One thread per execution unit vs exactly as many threads as execution contexts.

10. Consider the task dependency graph below. Each circle is a sequential task, and the number inside is its execution time (in cycles) on a single core. What is this application's minimum execution time when we have 2 cores? (Ignore all other overhead.)
15 cycles

What about its minimum execution time with 3 cores?
14 cycles



11. What is the communication model for pthreads?
Shared Address Space Model

What is the communication model for MPI?
Message passing model

Give one advantage for each communication model vs. the other.
Reasonable judgement will be okay.

Problem 2: ILP & SIMD

16 points. Consider the following C program:

```
#include <stdlib.h>
#define N 100000

void initialize(float *array, int length) {
     // initialize the array with random values...
}

void generate_output(float* input, float *output,
                     float *kernel, int i) {
     float a1 = output[i-1];
     float a2 = input[i];

     float b1 = a1 * kernel[0];
     float b2 = a2 * kernel[1];
     float b3 = b1 + b2;
     output[i] = b3;
}

float* func() {
     float *input= (float*)malloc((N+2) * sizeof(float));
     float *output = (float*)malloc((N+2) * sizeof(float));
     float kernel[2] = {1, 0.5};
     initialize(input, N + 2);

     // 0 padding on both ends
     input[0] = 0;
     input[N+1] = 0;
     output[0] = 1.0;

     for (int i = 1; i <= N; i++) {
          generate_output(input, output, kernel, i);
     }
     return output;
}
```

You should assume the following:

- There are no cache misses.
- The overhead of updating the loop index i is negligible.
- The load/store units perform any necessary address arithmetic.
- The overhead due to procedure calls, as well as starting and ending loops, is negligible.

Suppose you have a processor with one core that has:

- 2 load/store units that execute in a single cycle.
- 2 arithmetic units that execute in 3 cycles (regardless of operation).
- All execution units are fully pipelined.

2.1) Dataflow graph (4 points)

Draw the data flow graph for the loop and identify the critical path.

Critical path: Load output[i-1] => Multiplication => Addition => Store output[i], 8 cycles

2.2) Execution time (4 points)

What is the execution time of this program (as a function of N)?

8N. We are bounded by two load instructions and two arithmetic operations.

2.3) Processor ILP upgrade? (2 points)

You see an advertisement that is selling a new processor that has 4 arithmetic units, but is otherwise identical. Will you purchase it for this program? Why or why not?

No. We are limited by the latency bound.

2.4) Processor SIMD upgrade? (2 points)

Now you see another advertisement for a processor with 8-wide SIMD instructions. Will you purchase it for this program? Why or why not?

No. TheCalculation for each output element is not independent.

2.5) Changing the kernel (4 points)

Now consider another way of generating output:

```
void generate_output(float* input, float *output,
                     float *kernel, int i) {
     float a1 = input[i-1];    // note: change here!
     float a2 = input[i];

     float b1 = a1 * kernel[0];
     float b2 = a2 * kernel[1];
     float b3 = b1 + b2;
     output[i] = b3;
}
```

How does this modification change your answer to 2.3?

Now that we are bound by load/store operations, increasing the number of arithmetic units isn't going to help. So the answer to 2.3 won't change.

How does it change your answer to 2.4?

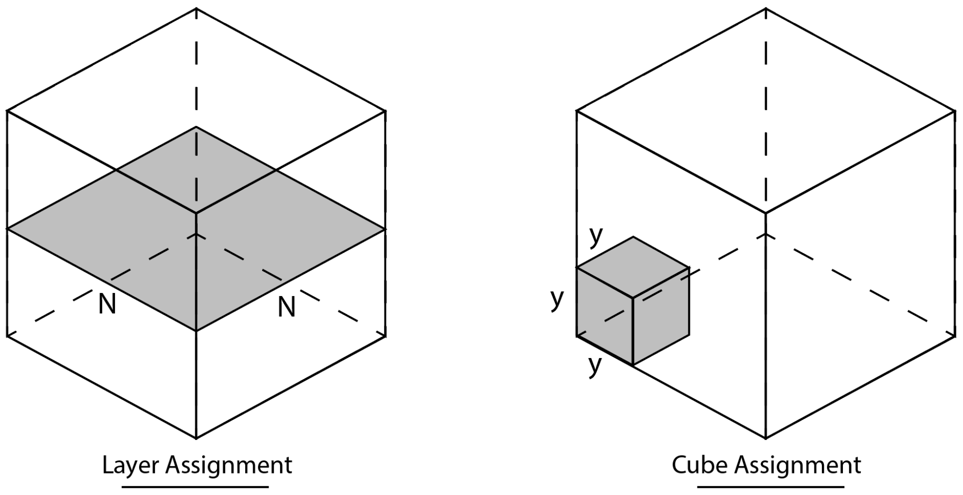
Now that each calculation is independent, we could benefit from SIMD executions.

Problem 3: Parallel programming models

15 points. Suppose you are trying to run a NxNxN 3-D grid solver. In this solver, each computation on a cell requires reading data from its 6 neighboring cells (front, back, above, below, top, bottom) and writing the computation result to the cell. You have P processors that can execute in parallel.

Assume that all numbers (e.g., N, P, etc) divide evenly and that P << N. Also assume that each cell is given its own cache line and all processors are assigned the same amount of work.

Answer the following questions in terms of big O notation.



3.1) Layer assignment (5 points)

You first assign work by partitioning the layers (each itself an NxN 2D array) among the processors. That is, processor 0 works on layers 0 to *x-1*, processor 1 works on layers *x* to 2*x-1*, etc. Under this assignment, what is the communication cost for each processor per iteration?

Each processor needs to communicate with the layer above and below, so the communication cost is O(N^2).

What is the arithmetic intensity per iteration?

The computation of each processor is O(N^3/P). So the arithmetic intensity is O(N^3/P)/O(N^2) = O(N/P)

3.2) Cube assignment (5 points)

Now you assign work in cubes. For example, processor 0 works on cells from (0,0,0) to (*y, y, y*). Assuming you assign work to minimize communication cost, what is the communication cost for each processor?

$y = N/\sqrt[3]{P}$**, hence communication cost is** $6y^2 = \frac{6N^2}{P^{\frac{2}{3}}}$, **O(N^2/P^(⅔))**

What is the arithmetic intensity?

Computation is O(N^3/P). Arithmetic intensity is O(N^3/P)/O(N^2/P^(⅔)) = O(N/P^(⅓))

3.3) Linear data layout (5 points)

Nanxi thinks it is wasteful that each cell needs its own cache line. She wants to design some cache line layout schemes to improve performance. Each cache line can hold at most N cells.

For continuous NxN-layer assignment (from 3.1), Nanxi thinks the most intuitive scheme is to have an entire row of N cells share the same cache line. How will communication cost and arithmetic intensity change with different orientations of the grid cells? (There are 3 orientations to consider, along the x-, y-, and z-dimensions of the grid.) Explain your answer.

Without loss of generality, assume the layers are parallel to the xy-plane.

If the cache lines are laid out parallel to x or y axis, the communication cost and arithmetic intensity will be similar to what we calculated in 3.1 since processors are only reading 2 layers of adjacent cache lines.

If the cache lines are laid out parallel to z axis, the communication cost will increase significantly because all processors are sharing the same cache line. The false sharing will result in great write contention on the cache lines. Arithmetic intensity will decrease significantly because of the increase in communication cost.

Problem 4: CUDA

20 points. In Assignment 2, we had you implement a Cuda rasterizer for circles as quickly as you could using primitive colors. Most modern day images ditch these primitive colors for more advanced textures, requiring expensive memory look-ups for these texture colors in favor of more realistic visual results. In this problem, we will be extending the Cuda rasterizer to work with texture look-ups while still aiming for fast parallel performance.

One assumption we will be making here is that *all primitives are fully opaque*. Thus, you do not have to worry about blending colors across circles (like in Assignment 2), rather you only need to focus on rendering the primitive closest to the camera (i.e., the triangle with the smallest z value). We would like to write a rasterization pipeline in Cuda that takes advantage of this assumption to make the pixel shading as fast as possible. Below we have provided you with starter code for our rasterizer.

We have provided you with 3 black-box helper functions listed below, similar to Assignment 2:

```
// returns true if pixelNorm lies in the triangle with vertices (a,b,c)
__global__ bool in_primitive(float3 a, float3 b, float3 c,
                             float2 pixelNorm);

// rasterizes the triangle using pInfo->texture
__global__ void rasterize(float2 pixelNorm, primitiveInfo* pInfo);

// returns interpolated depth of the triangle at pixelNorm
__global__ float depth_at_point(float3 a, float3 b, float3 c,
                                float2 pixelNorm);
```

On the next page is an implementation of our rasterizer that works with texture look-ups. You may use our implementation as starter code for any code that you are asked to write in this problem.

```
__global__ void kernelRenderPrimitives() {

  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int numThreads = blockDim.x * numBlocks.x;
  int width = cuConstRendererParams.screenWidth;
  int height = cuConstRendererParams.screenHeight;

  float invWidth = 1.f / cuConstRendererParams.screenWidth;
  float invHeight = 1.f / cuConstRendererParams.screenHeight;

  // this is a global variable
  float bestDepth[height, width];

  int numPrimitives = cuConstRendererParams.numPrimitives;
  for (int i = index; i < numPrimitives; i += numThreads) {

    // get primitive vertices
    float3 a = *(float3*)(&cuConstRendererParams.position[3 * (3 * i + 0)]);
    float3 b = *(float3*)(&cuConstRendererParams.position[3 * (3 * i + 1)]);
    float3 c = *(float3*)(&cuConstRendererParams.position[3 * (3 * i + 2)]);

    // pinfo->texture stores the texture as we as other primitive data
    pInfo *pinfo = (pInfo*)(&cuConstRendererParams.primitiveInfo[i]);

    // defines primitive bounding-box in 2D using orthographic projection
    // clip into [0,0] - [wth,hgt] range
    int minX = max(min(min(a.x, b.x), c.x), 0);
    int maxX = min(max(max(a.x, b.x), c.x), width);
    int minY = max(min(min(a.y, b.y), c.y), 0);
    int maxY = min(max(max(a.y, b.y), c.y), height);


    for (int pixelX = minX; pixelX <= maxX; pixelX++) {
      for (int pixelY = minY; pixelY <= maxY; pixelY++) {
        // compute normalized pixel coordinates
        float2 pixelNorm = make_float2(invWidth * (float(pixelX) + 0.5f),
                                       invHeight * (float(pixelY) + 0.5f));
        if ( in_primitive(a, b, c, pixelNorm) ) {
          // rasterizes primitive using the texture.
          // TODO: is this correct?
          float d = depth_at_point(a, b, c, pixelNorm);
          if(d < bestDepth[pixelY, pixelX]) {
             Lock { // low-level mutex code elided ...
               rasterize(pixelNorm, pinfo);
               bestDepth[pixelY, pixelX] = d;
             }
          }
        }
      }
    }
  }
}
```

4.1) Correctness (5 points)

Your 418 partner isn't sure as to whether a specific part of the code is correct and marks it with a **`TODO`** comment. You run the results and indeed, there is a bug with your code affecting the visual output. What is the bug? How would you go about fixing the problem?

Thread A and B may have closer depths, with A.z > B.z, but B.z may get the lock first and have its results overwritten by A, causing an incorrect ordering. Instead, the lock should incorporate the depth read/comparison.

4.2) Performance (5 points)

You find the above code to be very slow, so you try a different avenue of parallelism where you parallelize over pixels instead (again, similar to Assignment 2). For each pixel, you iterate through all the primitives and hold onto a reference of the closest primitive before rasterizing it in the end, minimizing the amount of calls to rasterize to just one per thread.

```
for each pixel: ← (parallel over threads)
    for each primitive: //we are looping through all primitives
        if depth(pixel, primitive) < bestDepth:
            bestPrimitive = primitive;
            bestDepth = depth(pixel, primitive);
    rasterize(bestPrimitive);
```

Does this new implementation require any synchronization? Please refer to whether memory operations are independent/shared among threads when justifying your answer.

No synchronization required. Reads and writes are not shared among threads.

4.3) Work balance (4 points)

Compare the original implementation to the modified implementation in 4.2. For each implementation, is there any workload imbalance amongst the threads in the same block? What about threads in between blocks? You may assume the call to rasterize takes the same amount of time for each primitive.

First kernel had work imbalance between threads (some call rasterize more) and between blocks (some have an overall greater number of primitives in bounds). Second kernel had no work imbalance whatsoever (only one call to rasterize per thread).

For the below questions, assume you are running your revised kernel on a 6 core GPU with 64 warps and 32 threads each. The device has 64kB of available shared memory total per device.

4.4) Block size (3 points)

You try running your revised kernel with a thread block size of (64,64,1), but you receive a CUDA error. You run it again on a block size of (32,32,1) and your program works fine. Why might that be the case?

Warps x threads needs to be greater than the total number of threads per block.

4.5) Shared memory (3 points)

Running your revised kernel requires 16kB of shared memory per block with a block size of (32,32,1). Do you expect cores to have full occupancy (percentage of cores in use)? Why or why not?

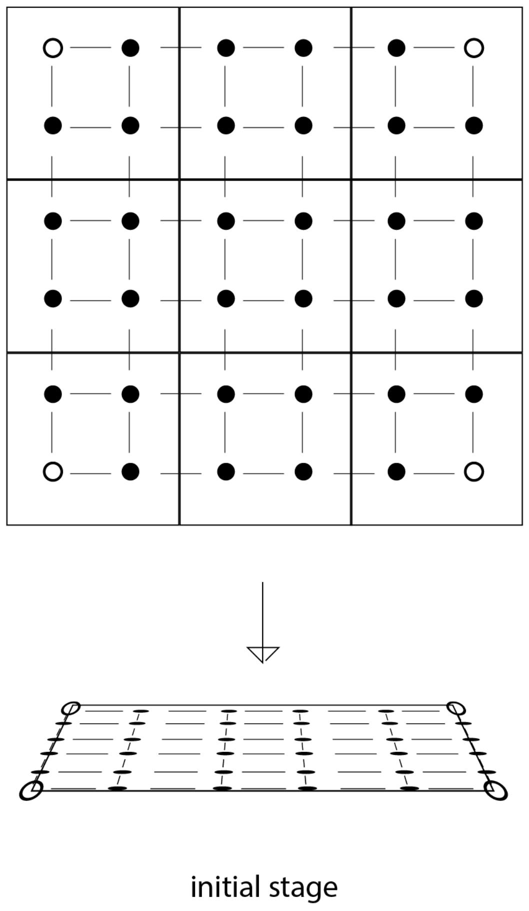
No, you are bounded by the total amount of shared memory.

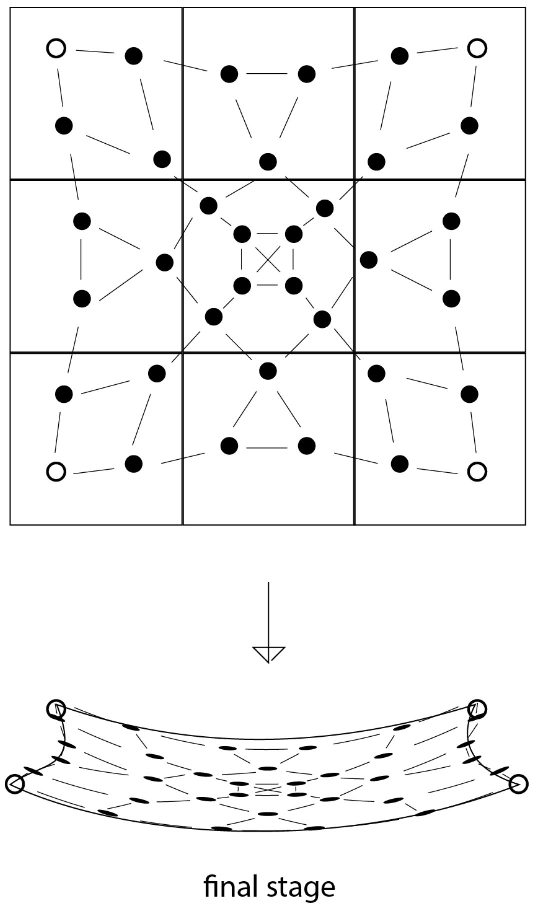
Problem 5: Workload-driven Performance

12 points. Mass-spring systems are large-scale particle-based environments used to produce fabric and clothing simulations. Each particle (denoted as a black dot) is a point mass with its own weight and is connected to other masses via springs (denoted as thin black lines). At each iteration, particles exchange forces (recall Newton's third law) while also feeling a net gravitational force downwards.

In order to provide more realistic results, our mass-spring system aims to remesh itself, meaning that it removes masses/springs from low-curvature areas and introduces masses/springs into higher curvature regions after every few iterations. Below is an example of our initial uniform mass-spring system, where the white dots on the edges represent the anchor points that will not move during the simulation. On the left, we show the remeshed results after several iterations, where more masses have accumulated near the center.

We've partitioned the computation into nine, equal-size squares (as drawn in the figure) and assign all dots within each square to one core. You may assume each mass takes 1 unit of local computation and each edge requires 1 unit of communication. A detailed diagram is shown below.





5.1) Workload balance (3 points)

What do you predict about the variance in local computation per processor, communication between processors, and arithmetic intensity in the final stage compared to the initial stage (does it increase or decrease)? Why is this?

Variance increases in all three. Cores do not have an equal number of nodes/outgoing edges (can count to prove it).

5.2) Total work and communication (3 points)

How does the total computation of the entire scene change when moving from the initial stage to the final stage? What about the total communication between processors? Why is that?

Computation stays the same. Communication decreases because of a tradeoff between computation and communication (more work done by one core, less to communicate overall).

5.3) Performance (3 points)

Based on the answers to the previous question, how does the execution performance of the final stage change compared to the initial stage? Why?

Worse execution performance due to workload imbalance/bottlenecks.

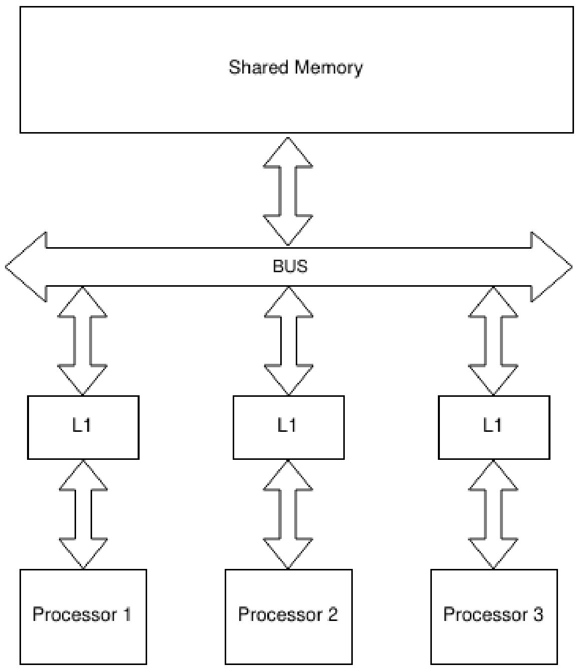
5.4) Improving work assignment (3 points)

Using a simple partitioning scheme, how would you rebalance the local computation per processor in the final stage? Using the taxonomy described in lecture, what type of partitioning scheme is this?

You can move around the outer nodes in the center cell to rebalance the work so that each cell again has 4 nodes of work. You could also use a quadtree and subdivide the center node. Both are semi-static partitioning schemes.

Problem 6: Cache Coherence

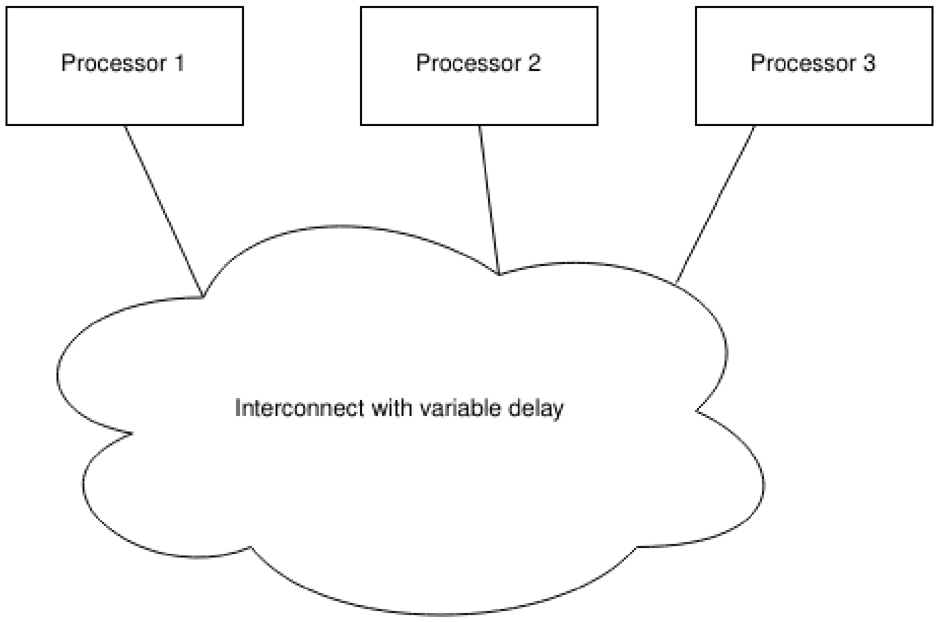
15 points. Recall the implementation of an MSI, snooping-based cache coherence protocol we have discussed in class. Assume an atomic bus, with access guarded by an arbiter. A high level diagram of this is shown here:



As we can see, every processor is connected to each other through a single bus. This is nice for when a processor must broadcast a message out, as this operation happens atomically, and all processors respond accordingly. For instance, if Processor 1 sends out a BusRdX, all other processors would respectively invalidate their instances of the memory location at the same time. The arbiter acts as a synchronization point, ensuring all processors observe the request at the same time.

6.1) MSI without a bus (5 points)

For this problem, we are going to instead connect processors to each other over a point-to-point network that does not guarantee that messages are seen by all processors at the same time.. The shared memory is abstracted away in order to focus on the communication between processors themselves. (Continued on next page.)



Now assume we use an unchanged MSI protocol over this point-to-point network. Consider the following sequence of events, all operating on some shared address X. *Fill in the coherence state (M, S, or I) for each processor after each event:*

Execution order	P1's state	P2's state	P3's state
	I	I	S
P1 sends a BusRdX			
P2 receives BusRdX from P1			
P2 sends BusRd			
P1 receives Ack from P2			
P3 receives BusRd from P2			
P2 receives Ack from P3			
P3 receives BusRdX from P1			
P1 receives Ack from P3			

(Continued on next page)

The above state transitions are not a cache coherent execution. Using the definition of coherence from lecture, explain why not. Which property of coherence is violated?

Execution order	P1's state	P2's state	P3's state
	I	I	S
P1 sends a BusRdX	I ->M	I	S
P2 receives BusRdX from P1	I ->M	I	S
P2 sends BusRd	I ->M	I ->S	S
P1 receives Ack from P2	I ->M	I ->S	S
P3 receives BusRd from P2	I ->M	I ->S	S
P2 receives Ack from P3	I ->M	S	S
P3 receives BusRdX from P1	I ->M	S	I
P1 receives Ack from P3	M	S	I

Execution order	P1's state	P2's state	P3's state
	I	I	S
P1 sends a BusRdX	I	I	S
P2 receives BusRdX from P1	I	I	S
P2 sends BusRd	I	I	S
P1 receives Ack from P2	I	I	S
P3 receives BusRd from P2	I	I	S
P2 receives Ack from P3	I	S	S
P3 receives BusRdX from P1	I	S	I
P1 receives Ack from P3	M	S	I

\

6.2) Update vs invalidate (5 points)

```
int *p1, *p2;
void Thread1() {
      int x = *p1;
      for (int i = 0; i < 100000; i++) {
            *(p2) += x;
      }
}
void Thread2() {
      int y = *p2;
      for (int i = 0; i < 100000; i++) {
            *(p1) += y;
      }
}
```

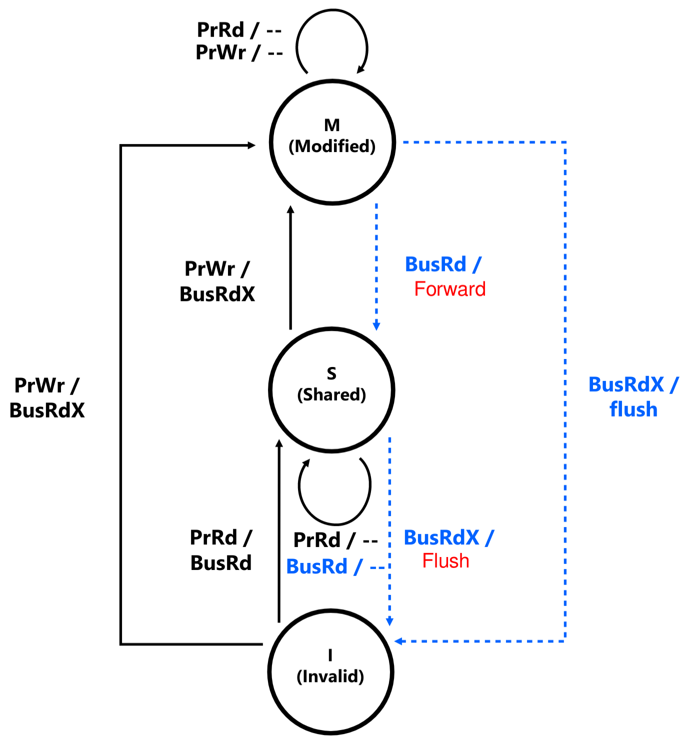
Alice and Bob ran the above program with two threads. Alice's machine uses an invalidation-based coherence protocol; whereas Bob's uses an update-based coherence protocol. Which do you expect to perform better? What is the bus traffic going to look like for each machine on the above program? Explain your answer.

Invalidation protocol (Alice) is better. These threads do not actually communicate once they enter the loop. E.g., in Thread1(), p1 is read once before the loop, but is not accessed thereafter, and similarly for Thread2 and p2. An invalidation-based protocol will give each thread exclusive access to the variable it is modifying in the loop and require no further bus traffic.

Note some answers stated that "invalidation protocols are better at writes". This is not true in general, and it is not a correct explanation. (Some answers actually said "update protocols are better at writes", saying that Bob would perform better -- this isn't true in general either.) The correct answer in this question involves the communication pattern between the threads.

6.3) Modifying MSI (5 points)

Your friend modifies the MSI protocol as shown below (all changes are in red):



Specifically, your friend wants to modify the transition from M → S to avoid writing data back to main memory. Instead, a processor in M will respond to a BusRd request by downgrading to S and forwarding the modified data to the requestor, but will *not* flush the data to memory.

Additionally, to preserve coherence, the protocol will now flush the data to memory when a BusRdX is received in S (the S → I transition). Why is this transition necessary to preserve coherence? (Give an example.)

Flushing on S → I is necessary for write propagation. Imagine P0 writes X, then P1 reads X. Both have X in S, and the value has not been propagated to memory yet. Now if P0 and P1 evict X, the update P0 made will be lost.

Note: Many answers said "P2 writes X" as the final step -- this is not necessarily a valid example, as P2 could get the value forwarded from one of the sharers on-chip. But I gave credit for it anyway.

Some answers also talked about write serialization, reasoning that values must appear at memory in the write order to be properly serialized. This is incorrect. So long as the *values seen by each processor* respect write serialization, there is no issue with write serialization. In a system that forwards the dirty value to P2, and never writes the value written by P0 back to memory, there is no coherence problem.

What are the performance implications of this modified MSI protocol? Do you expect it will perform better than the original MSI protocol from lecture? Explain your answer.

There are some exceptions, but as a general rule this version of MSI will perform *much* worse. The number of flushes to memory will grow with the number of sharers. Since reads are far more common than writes, that means the flush traffic to memory will increase enormously. (The protocol works exactly as described -- there is no separate dirty bit to distinguish “dirty sharers” from “clean sharers”, nor do we have any “owner bit” to guarantee that only one sharer writes back dirty data. This is not MOESI.)