Parallel Computer Architecture and Programming
Written Assignment 3

50 points total. Due Monday, July 17 at the start of class.

Problem 1: Message Passing (6 pts)

A. (3 pts) You and your friend liked the lecture on message passing and decide to make your own message passing API for C++. You have already implemented and tested the SEND and RECEIVE functions and they work well. Now, your friend suggests that you also add support for LOCKS in your library. Do you agree with this suggestion or do you argue that it is unnecessary to do so in the message passing programming model? Why?

B. (3 pts) Your friend suspects that his program is slow because of high communication overhead (waiting for message sends and receives to complete). To make it faster, he attempts to overlap the sending of multiple messages by rewriting his code to use **asynchronous, non-blocking sends** instead of synchronous blocking sends. The result is the code shown below. (Assume the code below is run by thread 1 in a two thread program, where thread 1 sends data to thread 2.)

```
float mydata[ARRAY_SIZE];
int dst_thread = 2;

update_data_1(mydata); // update contents of mydata here

// send contents of mydata in a message to dst_thread
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);

update_data_2(mydata); // update contents of mydata here

// send contents of mydata in a message to dst_thread
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);
```

After running his code, your friend runs to you to say "my program no longer gives the correct results." Explain what is the bug?

Problem 2: Memory Consistency (8 pts)

A. (3 pts) Consider the following code executed by three threads on a **cache-coherent, relaxed consistency** memory system. Specifically, the system allows reordering of writes (W->W reordering) and in these cases makes no guarantees about when notification of writes is delivered to other processors. **You should assume that all variables are initialized to 0 prior to the code you see below.**

```
P1:                 P2:                 P3:
==================  ==================  ==================
x = 10;             while (!flag);      while (!flag);
flag = true;        print x;            print x;
print x;
```

You run the code and P2 prints "10". List what values might be printed by P1 and P3. Also explain why your answer shows that the system **does not** provide sequentially consistent execution.

B. (2 pts) Imagine you are given a memory write fence instruction (`wfence`), which ensures that all writes prior to the fence are visible to all processors when the fence operation completes. Please add the **minimal number of write fences** to the code in Part A to ensure execution consistent with that of a sequentially consistent.

C. (3 pts) Consider the following program which has four threads of execution. In the figure below, the assignment to `x` and `y` should be considered stores to those memory addresses. Assignment to `r0` and `r1` are loads from memory into local processor registers. (The print statement does not involve a memory operation.)

Processor 0	**Processor 1**	**Processor 2**	**Processor 3**

`x = 1`	`y = 1`	`r0 = y`
		`r1 = x`
		`print (r0 & ~r1)`
			`r0 = x`
			`r1 = y`
			`print (r0 & ~r1)`

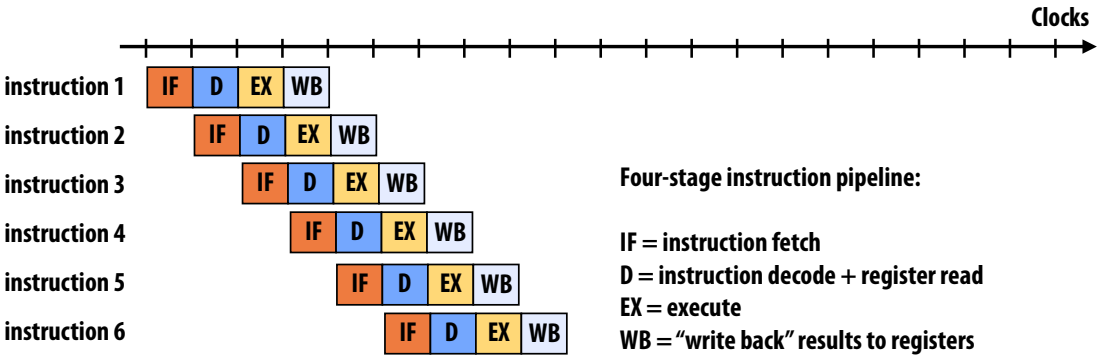
- Assume the contents of addresses `x` and `y` start out as 0.
- Hint: the expression `a & ~b` has the value 1 only when `a` is 1 and `b` is 0.

You run the program on a four-core system and observe that both processor 2 and processor 3 print the value 1. Is the system sequentially consistent? Explain why or why not?

Problem 3: Avoiding Stalls in a Processor Pipeline (10 pts)

Consider a single core, single threaded processor that executes instructions using a simple four-stage pipeline. As shown in the figure below, each unit performs its work for an instruction **in one clock**. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with six **independent instructions** on this processor. *However, if instruction B depends on the results of instruction A, instruction B will not begin the IF phase of execution until the clock after WB completes for A.*



A. (1 pt) Assuming all instructions in a program are **independent** (yes, this a bit unrealistic) what is the instruction throughput of the processor?

B. (1 pt) Assuming all instructions in a program are **dependent** on the previous instruction, what is the instruction throughput of the processor?

C. (1 pt) What is the latency of completing an instruction?

D. (1 pt) Imagine the EX stage is modified to improve its throughput to two instructions per clock. What is the new overall maximum instruction throughput of the processor?

E. (3 pts) Consider the following C program:

```
float A[100000];
float B[100000];
// assume A is initialized here

for (int i=0; i<100000; i++) {
    float x1 = A[i];
    float x2 = 2*x1;
    float x3 = 3 + x2;
    B[i] = x3;
}
```

Assuming that we consider **only the four instructions in the loop body in this problem** (for simplicity, ignore the instructions for managing the loop predicate), what is the average instruction throughput of this program? (Hint: You should probably consider more than one iteration of the loop to get the correct answer).

F. (3 pts) Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-pseudocode. (not in words)

Problem 4: Particle Simulation (10 pts)

Consider the following code that uses a simple $O(N^2)$ algorithm to compute forces due to gravitational interactions between all N particles in a particle simulation. One important detail of this algorithm is that force computation is symmetric (`gravity(i,j) = gravity(j,i)`). Therefore, iteration `i` only needs to compute interactions with particles with index `j`, where `i<j`. As a result, the work done by the algorithm is $N^2/2$ rather than N^2.

In this problem, **assume the code is run on a dual-core processor, with infinite memory bandwidth.**

```
struct Particle {
   float force;  // for simplicity, assume force is represented as a single float
};

Particle particles[N];

void compute_forces(int threadId) {

  // thread 0 takes first half, thread 1 takes second half
  int start = threadId * N/2;
  int end = start + N/2;

  for (int i=start; i<end; i++) {

    // only compute forces for each pair (i,j) once, then accumulate force
    // into *both* particle i and j

    for (int j=i+1; j<N; j++) {
       float force = gravity(i, j);
       particles[i] += force;
       particles[j] += force;
    }
  }
}
```

The question is on the next page.

A. (4 pts) The function `compute_forces` above is run by two threads on a dual-core processor. There is a correctness problem with the code. Using only the synchronization primitive:

```
atomicAdd(float* addr, float val)
```

Fix the correctness bug in the code. **However, to get full credit your solution should be efficient—it should do better than making N^2 calls to `atomic_add`** (at least by an integer constant factor). Solutions that incur significant storage overhead or increase the amount of work done by the algorithm are not allowed.

B. (2 pts) There is also a significant **performance problem** in the implementation that results in a speedup that is significantly lower than $2\times$ on the two-core processor. What is the problem?

C. (4 pts) Give an implementation of `compute_forces` that (1) achieves good workload balance between the two threads (2) does not significantly increase the amount of work performed (work should be no more than $N^2/2 + O(N)$) and (3) does not use fine-grained `atomicAdd` synchronization. However, you are allowed to allocate O(N) storage and use a barrier. Pseudocode is fine.

Problem 5: Angry Students (8 pts)

Your friend is developing a smartphone game that features many students chasing after professors for making long homeworks. Simulating students is expensive, so your friend decides to parallelize the computation using one thread to compute and update the student's positions, and another thread to simulate the student's angriness. The state of the game's N students is stored in the global array `students` in the code below).

```
struct Student {
   float position;   // assume 1D position for simplicity
   float angriness;
};

Student students[N];

////////////////////////////////

void update_positions() {
   for (int i=0; i<N; i++) {
      students[i].position = compute_new_position(i);
   }
}

void update_angriness() {
   for (int i=0; i<N; i++) {
      students[i].angriness = compute_new_angriness(i);
   }
}

////////////////////////////////

// ... initialize students here

std::thread t0(update_positions);
std::thread t1(update_angriness);
t0.join();
t1.join();
```

Questions are on the next page...

A. (3 pts) Since there is no synchronization between thread 0 and thread 1, your friends expect near a perfect 2× speedup when running on a **two-core processor that implements invalidation-based cache coherence**. They are shocked when they obtain a much lower speedup. Why is this the case? (For this problem assume that there is sufficient bandwidth to keep two cores busy – "the code is bandwidth bound" is not an answer we are looking for... but there is something else that is slowing the program down.)

B. (5 pts) Modify the program to correct the performance problem. You are allowed to modify the code and data structures as you wish, **but you are not allowed to change what computations are performed by each thread and your solution should not substantially increase the amount of memory used by the program.** You only need to describe your solution in pseudocode (compilable code is not required).

Problem 6: Parallel Histogram Generation (Yet Again) (8 pts)

Your friend implements the following parallel code for generating a histogram from the values in a large input array `input`. For each element of the input array, the code uses the function `bin_func` to compute a "bin" the element belongs to (`bin_func` always returns an integer between 0 and `NUM_BINS-1`), and increments a count of elements in that bin. Her port targets a small parallel machine with only two processors. *This machine features 64-byte cache lines and uses an invalidation-based cache coherence protocol.* Your friend's implementation is given below.

```
float input[N];                   // assume input is initialized and N is a very large
int   histogram_bins[NUM_BINS];   // output bins
int   partial_bins[2][NUM_BINS];  // assume bins are initialized to 0
                                  // assume partial_bins is 64-byte aligned (this means
                                  // that the address of partial_bins[0][0] modulo 64 == 0
                                  // (it is at the start of a cache line)

/////////////////////////// Code executed by thread 0 ////////////////////////////
for (int i=0; i<N/2; i++)
   partial_bins[0][bin_func(input[i])]++;

barrier();  // wait for both threads to reach this point

for (int i=0; i<NUM_BINS; i++)
    histogram_bins[i] = partial_bins[0][i] + partial_bins[1][i];

/////////////////////////// Code executed by thread 1 ////////////////////////////
for (int i=N/2; i<N; i++)
   partial_bins[1][bin_func(input[i])]++;

barrier();  // wait for both threads to reach this point
```

A. (3 pts) Your friend runs this code on an input of 1 million elements (`N`=1,000,000) to create a histogram with eight bins (`NUM_BINS`=8). She is shocked when the program obtains much less than a linear speedup. She sadly believes she needs to completely restructure the code to eliminate load imbalance. You take a look and recommend that she not do any coding at all, and just create a histogram with 16 bins instead. Who's approach will lead to better parallel performance? Why?

B. (3 pts) Inspired by his new-found great performance, your friend concludes that more bins is better. She tries to use the provided code from part A to compute a histogram of 10,000 elements with 2,000 bins. She is shocked when the speedup obtained by the code drops. Improve the existing code to scale near linearly with the larger number of bins. (Please provide pseudocode as part of your answer – it need not be compilable C code.)

C. (2 pts) Your friend changes `bin_func` to a function with *extremely high arithmetic intensity (high ratio of math instructions to memory access)*. (The new function requires 100000's of instructions to compute the output bin for each input element). If the **original histogram code provided in part A** is used with this new `bin_func` do you expect scaling to be better, worse, or the same as the scaling you observed using the old `bin_func` in part A? Why? (Please ignore any changes you made to the code in part B for this question.)