CS194-15 Fall 2016 Midterm 1

Name:	
Student Id:	
Person to your left:	
Person to your right:	

Q1:	(46 points)
Q2:	(25 points)
Q3:	(28 points)

Read all of the following information before starting the exam:

- Show all work, clearly and in order, if you want to get full credit. I reserve the right to take off points if I cannot see how you arrived at your answer (even if your final answer is correct).
- Circle or otherwise indicate your final answers.
- Please keep your written answers brief; be clear and to the point. I will take points off for rambling and for incorrect or irrelevant statements.
- This test has 3 problems. Check to make sure that your test booklet has all pages!
- Good luck!

1. (*46 points*) **Short Answer (13 questions)**

a. (*3 pts*) What is load imbalance? Give an example.

Load Imbalance is what occurs when 1 thread does a lot more “work” than another thread and subsequently takes a lot longer to finish. For example in the Mandelbrot set, some of the darker regions take a lot longer to calculate than the lighter regions. So if one thread has to do the dark regions and another the light, the dark thread would take much longer.

b. (*2 pts*) What is the difference between concurrency and parallelism?

Concurrency is when multiple tasks are logically active at the same time. Parallelism is when they are actually active at the same time.

c. (*4 pts*) What are TLBs and how do they impact performance?

A translation lookaside buffer is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval. When an address is searched in the TLB and not found, we must go to the physical memory, which takes a lot longer.

d. (*2 pts*) In Homework 1, why were profiled times incorrect if we did not print-out the final index after pointer chasing?

Because the compiler would optimize the program and not store the result when it sees that we never use it. Thus it saves many clock cycles and we get a faster profiled time.

e. (*4 pts*) What is Moore's Law, specifically what does it say about the growth in the number of transistors?

Moore's law says the number of transistors per square inch on an integrated circuits will double every 18 months

f. (*4 pts*) Given more transistors through Moore's Law, why use them to create caches? What are the advantages and disadvantages of making caches larger? Why create a hierarchy of L1 and L2 caches?

Given the processor-memory gap, in that memory is improving at a much slower rate than processors, we used a tiered approach to caches to hide this gap. Creating a larger cache means we have slower lookup times and thus a lower bandwidth. Having a smaller cache is the opposite. Thus to take advantage of a high bandwidth and fast lookup times, with a larger amount of memory. We use a cache hierarchy.

g. (*5 pts*) Why do we use blocked version of matrix multiply routines? What is a blocked for loop? Explain why this will have significant speed increases?

Blocking a for loop means that we only computer on a block that is cache aligned. (the width is the size of one line of the cache) this takes advantage of the spatial locality). Cache blocking attempts to maximize the ratio of cache hits to cache misses.

h. (*8 pts*) Name and briefly explain 4 (we covered 5 in lecture) micro-architectural improvements that have led to significant processor speed increases.

Superscalar, pipelining, out of order execution, SIMD, hardware assisted multithreading.

i. (*4 pts*) A non-pipelined basic processor has a latency of 64 ns for each instruction. If we split the processor into 16 perfectly pipelined stages, what is the new latency per instruction? What was the throughput before and after the pipelining? Make sure to use the correct units. Also please ignore any additional latency induced by the introduction of pipeline stages.

The latency stays 64 ns. Thoughput before: 1 inst / 64 ns. Throughput after: 1 inst / 4 ns. Throughput is increased by 16.

j. (*2 pts*) How does pipe and filter pattern help to enforce modularity?

Each filter in pipe and filter is its own “module” with a defined input and output that can be treated a black-box to other programmers.

k. (*2 pts*) How does modularity help the chief architect of a software development team?

It helps the architect plan out workflow, and high level decisions on priorities, and to be able to convey the architecture efficitively to the team.

l. (*2 pts*) Which of the following is true? (circle 1 or 2 answers)

[X] 1. When you use gcc to compile OpenMP directives, the resulting assembly code looks a lot like pthreads.

[ ] 2. When you use gcc to compile pthreads code, the resulting assembly code looks a lot like OpenMP directives.

m. (*4 pts*) Consider the following code. Under what conditions do the two addition instructions take 1 cycle and what micro architectural improvements allow this?

```
int a, b, c, d;
    …
    …
    …
b = a + b;
c = c + d;
```

If all the values are loaded into the registers already, these two addition instructions could take 1 cycle, because of Superscalar the processor has multiple addition logical units.

2. (*25 points*) **Roofline Model**

```
//A = a vector that we'll compute the histogram of. Assume that each value A[i] is in the range [0, 1].
//N = length of A
//H = histogram output
//B = number of bins in histogram H. (initialized to zeros)
void histogram(float* A, int N, float* H, int B)
{
  #pragma omp parallel for //for part (d)
  #pragma omp parallel { //for part (e)
  #pragma omp single { //for part (e)
   for(int i = 0; i < N; i++) //iterate over input

      for(int j = 0; j < B; j++) //iterate over histogram bins
        #pragma omp task { //for part(e)
       float bin_begin = float(j)/B; //which histogram bin are we looking at?

        if (A[i] <= bin_begin)
          #pragma omp critical {//for part (d)
          #pragma omp critical {//for part(e)
          H[j] = H[j] + A[i]
          }
          }
          break; //move onto next index i
        }
  }
  }
}
```

Listing 1: Serial Naive Histogram Computation

These are examples of floating-point operations: $- \; + \; \times \div > < \leq \geq$

a. (*5 pts*) The number of floating-point operations in histogram( ) depends on the input data. Given fixed values of N and B, what is the **minimum** number of FLOPS that need to be computed? If N=100, give an example of a vector A that would hit this minimum number of FLOPS.

3N Flops (one FLOP to compute bin_begin; one FLOP to compare A[i] to bin_begin, and one FLOP to update H[j]).

A = [0, 0, 0, 0, 0, … , 0]

b. (*5 pts*) Given N and B, what is the **maximum** number of FLOPS that could need to be computed by histogram( )? Give an example vector A that would hit this maximum number of FLOPS.

2BN FLOPS (we will also accept 2BN + N FLOPS)

A = [1,1,1,1, … ,1] (we will also accept A = [(B-1)/B, (B-1)/B,…(B-1)/B].

c. (*3 pts*) If the program takes T seconds, what is the peak GFLOP/S this program would achieve?

NB *10e-9/ T

d. (*6 pts*) Using OpenMP, **parallelize** histogram( ) using the **omp for**. You can rewrite the code below, or you can write your changes directly on Listing 1. Be careful of race conditions!

SEE CODE

e. (*6 pts*) Using OpenMP, **parallelize** histogram( ) using the **omp task**. You can rewrite the code below, or you can write your changes directly on Listing 1. Be careful of race conditions!

SEE CODE

For reference, here are some OpenMP compiler directives:

```
    #pragma omp parallel
    #pragma omp parallel for
    #pragma omp single
    #pragma omp critical
    #pragma omp task
    #pragma omp taskwait
```

as well as some OpenMP runtime library functions:

```
    int omp_get_num_threads(void);
    int omp_get_thread_num(void);
```

as well as some OpenMP scheduling:

```
    schedule(static);
    schedule(dynami
void worker(S *struct_ptr) {

        int action = rand() % 100;

        if ( (action == struct_ptr->action)  {
                pthread_mutex_lock(struct_ptr->not_likely_lock);
                struct_ptr->not_likely_counter++;

                struct_ptr->not_likely_array[rand() % 100]++;
                pthread_mutex_unlock(struct_ptr->not_likely_lock);

        } else {
                pthread_mutex_lock(struct_ptr->likely_lock);
                struct_ptr->likely_counter++;

                struct_ptr->likely_array[action]++;
                pthread_mutex_lock(struct_ptr->likely_lock);
        }
}


void foo(int N) {
        pthread_t tid[N];
        S *predefined_struct;

        //initialize and load in random values to predefined_struct
                ...

        pthread_mutex_init(predefined_struct->likely_lock, NULL);
        pthread_mutex_init(predefined_struct->not_likely_lock, NULL);
        for ( int  i  =  0 ;   i  <  N;   i ++)   {

                pthread_create(&tid[i], NULL, &worker, predefined_struct);

        }

        for (int i = 0; i < N; i++){
                pthread_join(&tid[i]);
        }
        pthread_mutex_destroy(predefined_struct->likely_lock);
        pthread_mutex_destroy(predefined_struct->not_likely_lock);

 }
```

3. (*28 points*) **Pthreads, etc…**

a. (*14 pts*) Parallelize the code above using PThreads (reference functions below), be careful of data races and concurrency issues since we are passing by reference the same data structure to each thread (we don't want to accidentally overwrite data or seg fault). You can pseudocode for up to 10 points. You need to create N threads (`pthread_t tid [N];`)

b. (*4 pts*) Write a memory efficient data structure for `struct S` based on the code above.

```
struct S {
   int action;
   pthread_mutex_t *likely_lock;
   int likely_counter;
   int[100] likely_array;
   pthread_mutex_t *not_likely_lock
   int not_likely_counter;
   int[100] not_likely_array;
};
```

c. (*4 pts*) Write a memory inefficient data structure `struct S` based on the code above and justify its inefficiency using the memory hierarchy.

```
struct S {
   int action;
   int[100] not_likely_array;
   int[100] likely_array;
   int not_likely_counter;
   pthread_mutex_t *likely_lock;
   int likely_counter;
   pthread_mutex_t *not_likely_lock;
};
```

d. (*6 pts*) Now use hold-and-cold splitting in order to write a memory efficient data structure for `struct S`. Once again please justify your answer. You can assume that the code for handling your data structure does not need to be exactly the same as the original code in the question; in other words assume that there exists similar but not identical code that performs the same task in spirit. (Hint: You will need 2 data structures)

```
struct S {
 int action;
 pthread_mutex_t *likely_lock;
 int likely_counter;
 int[100] likely_array;
 struct S2* not_likely_struct


 };
```

```
struct S2 {
 pthread_mutex_t *not_likely_lock;
 int not_likely_counter;
 int[100] not_likely array;



};
```

Pthread reference:

pthread_create( )

```
int pthread_create (
pthread_t *thread,
const pthread_attr_t *attr,
void *(*start_routine)(void *),
void *arg);
```

pthread_join( )

```
int pthread_join (
pthread_t thread,
void **value_ptr);
```

pthread_mutex_lock( )

```
int pthread_mutex_lock (
pthread_mutex_t *mutex);
```

Locks an unlocked mutex. If the mutex is already locked, the calling thread blocks until the thread that currently holds the mutex releases it.

pthread_mutex_unlock( )

```
int pthread_mutex_unlock (
pthread_mutex_t *mutex);
```

Unlocks a mutex. The scheduling priority determines which blocked thread is resumed. The resumed thread may or may not succeed in its next attempt to lock the mutex, depending upon whether another thread has locked the mutex in the interval between the thread's being resumed and its issuing the *pthread_mutex_lock* call.

pthread_mutex_init( )

```
int pthread_mutex_init (
pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr);
```

Initializes a mutex with the attributes specified in the specified mutex attribute object. If*attr* is NULL, the default attributes are used.

pthread_mutex_destroy( )

```
int pthread_mutex_destroy (
pthread_mutex_t *mutex);
```

Destroys a mutex.

Locking syntax:

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
      …
pthread_mutex_lock(&lock);
      … //critical section
pthread_mutex_unlock(&unlock);
      …
pthread_mutex_destroy(&lock);
```

Page left blank intentionally