CS61C Fall 2018 GS Worksheet 2: RISC-V

Question 1: "Hello, I'm Johnny Cache"

Suppose we have an 8-bit system with a 32 B cache with a 2 word block size. Assume we're trying to access the following addresses in order:

0xAC, 0x18, 0xF6, 0x44, 0x9A, 0x3A, 0xE4, 0x00

a) Assuming a direct-mapped cache, calculate how many bits would go to the tag, index, and offset.

b) Assuming a direct-mapped cache, fill in the table below with what the cache will look like after all the memory accesses above have executed. The first has been filled in for you.

				0xAC			

c) Bonus Question: Why do we use the upper bits for the tag? Why not the index instead? (i.e. why do we use T:I:O instead of I:T:O)

Question 2 (SP15 Guerilla Section): Cache-22

LRU: Least Recently Used replacement policy. This policy is used in caches where data from memory has multiple locations within the cache it can potentially go (this is not possible on a direct-mapped cache). The LRU policy is used when there is new data to be loaded into the cache and every block it can be loaded into is full. Whichever of the blocks that can be replaced was least recently used is the one that gets replaced. Any data that block contained is written back to a higher level of cache.

Compare the performance of three cache designs for a byte-addressed memory system:

- **Cache 1**: A direct-mapped cache with four blocks, each block holding one word.
- **Cache 2:** A 16B 2-way set associative cache with 4B blocks and LRU replacement policy.
- **Cache 3:** A 16B fully associative cache with 4B blocks and LRU replacement policy.

For the following sequences of memory accesses starting from a cold cache, calculate the **miss rate** of each cache if the accesses are repeated for an arbitrarily large number of times. All addresses are given in decimal (not hexadecimal).

a) Memory Accesses: 0, 4, 0, 4, (repeats)

Cache 1: __________ Cache 2: __________ Cache 3: __________

b) Memory Accesses: 0, 16, 32, 0, 16, 32, (repeats)

Cache 1: __________ Cache 2: __________ Cache 3: __________

c) Memory Accesses: 0, 4, 8, 12, 16, 0, 4, 8, 12, 16, (repeats)

Cache 1: __________ Cache 2: __________ Cache 3: __________

d) Memory Accesses: 0, 4, 8, 12, 16, 12, 8, 4, 0, 4, 8, 12, 16, 12, 8, 4, (repeats)

Cache 1: __________ Cache 2: __________ Cache 3: __________

Question 3 (SU16 FINAL): Pragmatic Parallel Programming Practice

a) Circle ONE option below that best describes the result of running the following parallel code.

```
// given 512-element integer array A, count the occurrences of 61
int count = 0;
#pragma omp parallel for
{
      for (int i = 0; i < 512; i++) {
            if (A[i] == 61) count++;
      }
}
```

(1) Always correct

(2) Incorrect because of false sharing

(3) Incorrect because of data race

(4) Incorrect because of data dependency

b) To tackle data dependency, we can enforce the correct order of execution to ensure data dependency is always met. Consider the following situation, we have three functions:

```
void harvey();           void patterson();           void garcia();
```

These functions fill the three arrays `H[]`, `P[]`, and `G[]` sequentially. For example, if you call `harvey()` twice, then `H[0]` and `H[1]` will be filled. `harvey()` and `patterson()` work independently, but `garcia()` depends on the data in `H[]` and `P[]`. Specifically, to compute `G[i]`, `garcia()` requires `H[i]` and `P[i]` to be already computed. Fill in the blanks below to ensure the program works correctly.

```
omp_set_num_threads(3);                      // allocate three threads
int h_idx, p_idx, g_idx = 0, 0, 0;           // track indices
#pragma omp parallel
{
      int tid = omp_get_thread_num();
      while (g_idx < 1000) {                 // stop when garcia() is done
            if (tid == 0) {
                  harvey();
                  h_idx++;
            } else if (tid == 1) {
```

```
                        patterson();
                        p_idx++;
                } else {
                        while(_______________________________________);
                        ______________;
                        ______________;
                }
        }
}
```

Question 4: Spark (SP16 Final)

Given a set of documents, complete the functions so the code will do the following sequentially:

1. Count the total number of times each word shows up in a document
2. Finds the document that the words show up in the most times. If there is a tie between two documents, choose the document with the lower document ID.
3. Finally returns a set of key value pairs where the key is the document ID and the value is a list of all the words that showed up the most frequently in that given document.

```
def find_all_words(document):
      # Returns a list of all the words in a document.
      # Words are converted to lower-case, represented as strings.
      # Assume this is implemented.

def flatmap_func(pair):
      document_id = pair[0]    # This is an integer
      document = pair[1]
      ________________________________
      ________________________________

def count(v1,v2):
      ________________________________

def map_func(pair):
      ________________________________
      ________________________________
      ________________________________

def compare(v1,v2):
      ________________________________
      ________________________________
      ________________________________
      ________________________________
      ________________________________
      ________________________________
      ________________________________

def transform_func(pair):
      ________________________________
      ________________________________
      ________________________________

if __name__ == "main":
      rdd = sc.parallelize(documents).flatMap(flatmap_func) \
               .reduceByKey(count).map(map_func).reduceByKey(compare) \
               .map(transform_func).groupByKey().collect()
```

Question 5: Sparky, the Robot Dog

Sparky, the robot dog, is trying to learn how to play the highly competitive game of tug-of-war. He knows that he can use a two-step minimax to decide when to pull harder, and when to conserve his energy. However, at the day of the competition, he realizes that his method is too slow to make decisions in real time!

Given Sparky's original Python function below (he says it's okay if you do not understand why it works), can you use MapReduce to help him be ready to play at the 61C post-final party? Heuristic is some arbitrary function, and is already implemented as efficiently as possible.

```
def run( energy, oppEnergy, distanceToCenter ):
      choices = []
      for en in range(0, energy):
            states = []
            for op in range(0, oppEnergy):
                  delta = (en - op) / 2
                  states += [heuristic(distanceToCenter + delta, \
                                                energy - en, oppEnergy - op)]

            oppChoice = min(states)
            choices += [oppChoice]

      return choices.index(max(choices))
```

Question 6: Why Can't You Use Parallelism at a Gas Station? It Might Cause a Spark (SP15 Final)

a) Optimize `factorial()` using SIMD intrinsic(AVX).

```
    double factorial(int k) {
          int i;
          double f = 1.0;
          for (i = 1; i <= k; i++) {
                f *= (double) i;
          }
          return f;
    }
```

You might find the following intrinsics useful:

Function Header	Action
`__m256d _mm256_loadu_pd(double *s)`	**returns** vector(s[0],s[1],s[2],s[3])
`void _mm256_store_pd(double *s, __m256d v)`	**stores** p[i] = v_i **where** i = 0,1,2,3
`__m256d _mm256_mul_pd(__m256d a, __m256 b)`	**returns** vector(a_0b_0,a_1b_1,a_2b_2,a_3b_3)

```
double factorial(int k) {
      int i, j;
      double f_init[] = {1.0, 1.0, 1.0, 1.0};
      double f_res[4];
      double f = 1.0;

      // initialize f_vec
      __m256d f_vec = ________________________;

      // vectorize factorial
      for (i = 1; i <= _________; _________) {
            double l[] = {
                  (double) _________, (double) _________,
                  (double) _________, (double) _________};
            __m256d data = ________________________;
            _________ = ________________________;
      }

      // reduce vector
      __________________________________;
      for (j = 0; j < 4; j++) {
            ________________________;
      }
      // handle tails
      for ( ; i <= k; i++) {
            ________________________;
      }

      return f;
}
```

b) Cache Coherence: We are given the task of counting the number of even and odd numbers in an array, A, which only holds integers greater than 0. Using a single thread is too slow, so we have decided to parallelize it with the following code:

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
        int result[2] = {0, 0};
        int i,j;
        omp_set_num_threads(threads);

        #pragma omp parallel for {
            for (j=0; j < size; j++) {
                result[ (A[j] % 2 == 0) ? 0 : 1 ] += 1;
            }
        }

        printf("Even: %d\n", result[0]);
        printf("Odd: %d\n", result[1]);
}
```

As we increase the number of threads running this code:

i) Will it print the correct values for Even and Odd? If not, explain the error.

ii) Can there be false sharing if the cache block size is 8 bytes?

iii) What about 4 bytes?