CS 61C Summer 2016
Guerrilla Section 6: Parallelism

a. Version 1:

```
result[0] = 0;
#pragma omp parallel
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
    int sum = 0;
    for (int j = 0; j < ARR_SIZE; j++) {
        sum += arr[j] + i;
    }
    result[i] = sum + result[i - 1];
}
```

Correctness:
A. Always Correct
B. Sometimes Correct
C. Always Incorrect

Speed:
A. Faster
B. Same
C. Slower

Correctness: A
Speed: **C**

b. Version 2:

```
result[0] = 0;
#pragma omp parallel for
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
    int sum = 0;
    for (int j = 0; j < ARR_SIZE; j++) {

    }
    #pragma omp critical
    result[i] = sum + result[i - 1];
}
```

Correctness: **B**
Speed: **A**

c. Version 3:

```
result[0] = 0;
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for (int j = 0; j < ARR_SIZE; j++) {
        sum += arr[j] + i;
    }
    result[i] = sum + result[i - 1];
}
```

Correctness: **A**
Speed: **A**

d. Consider the correctly parallelized version of the serial code above.
   i. Could it ever achieve perfect speedup? **No**
   ii. What law provides the answer to this question? **Amdahl's Law**

Problem 2 (adapted from Sp15 Final):

i. 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:

`__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, __m256d 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 = __mm256_loadu_pd(f_init);
   // vectorize factorial
   for (i = 1 ; i <= k / 4 * 4; i += 4) {
      double l[] = {
         (double) i, (double) i + 1,
         (double) i + 2, (double) i + 3 };
      __m256d data = _mm256_loadu_pd(l);
      f_vec = _mm256_mul_pd(f_vec, data);
   }
   // reduce vector
   _mm256_store_pd(f_res, f_vec);
   for (j = 0 ; j < 4 ; j++) {
      f = f * f_res[j];
   }
   // handle tails
  for ( ; i <= k ; i++) {
       f *= (double) i;
   }
     return f;
}
```

Problem 3: Cache Coherence (adapted from Sp15 final):

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 parallellize 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.

No, there may be a data race.

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

Yes

iii. What about 4 bytes?

No

Problem 4 (taken from Fa14 Final):

Imagine we're looking at Facebook's friendship graph, which we model as having a vertex for each user, and an undirected edge between friends. Facebook stores this graph as an adjacency list, with each vertex associated with the list of its neighbors, who are its friends. This representation can be viewed as a list of degree 1 friendships, since each user is associated with their direct friends. We're interested in finding the list of degree 2 friendships, that is, an association between each user and the friends of their direct friends.

You are given a list of associations of the form `(user_id, list(friend_id))`, where the `user_id` is 1st degree friends with all the users in the list.

Your output should be another list of associations of the same form, where the first item of the pair is a `user_id`, and the second item is a list of that user's 2nd degree friends. **Note:** a user is not their own 2nd degree friend, so the list of second degree friends must not include the user themselves.

Write pseudocode for the mapper and reducer to get the desired output from the input. Assume you have a set data structure, with `add(value)` and `remove(value)` methods, where `value` can be an item or a list of items. You can iterate through a list with the `for item in items` construct. You may not need all the lines provided.

```
map(user_id, friend_ids):
    for friend in friend_ids:
        emit(friend, friend_ids)

reduce(key, values):
    second_degree_friends = set()
    for value in values:
        second_degree_friends.add(value)
    second_degree_friends.remove(key)
    emit(key, second_degree_friends)
```