CSM 61C Spring 2020

Parallelism

Exam Question Compilation Solutions

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

- Fall 2018 Midterm Q5 A,B,E
- Fall 2017 Final Q2
- Fall 2017 Final Q10
- Fall 2015 Final F2
- Fall 2015 Final F3
- Spring 2015 Final F2
- Fall 2019 Final Q10a, Q10b, Q10c
- Spring 2018 Final Q10
- Spring 2018 Final Q11
- Summer 2018 Final Q9
- Summer 2018 Final Q10
- Summer 2018 Final Q13

Q5) *What's that smell? Oh, it's potpourri!* (18 points)

a) What's the ideal speedup of a program that's 90% parallel run on an ∞-core machine? ____10____

1/s = 1/(1/10) = **10x**

b) How many times faster is the machine in (a) than a 9-core machine in the ideal case? ____2____

1/[s + (1-s)/9] = 1/[1/10 + (9/10)/9] = 1/[1/10 + 1/10] = 1/[2/10] = 10/2 = 5 (so **2x** faster)

e) The value of memory pointed to by x1 is 10. Two cores run the following code concurrently:

`lw x2,0(x1)`
`addi x2,x2,1`
`sw x2,0(x1)`
	`lw x3,0(x1)`
	`add x3,x3,x3`
	`sw x3,0(x1)`

...what are possible values of the memory afterward? (select ALL that apply)

☐10 ■11 ☐12 ☐13 ☐14 ☐15 ☐16 ☐17 ☐18 ☐19 ■20 ■21 ■22

Q2: MapReduce

We want to provide more insight into the Yelp dataset used in Project 4. **Fill in the blanks for the Python code in the answer sheet** to return a review with the highest word count per star label using Spark (Multiple reviews could have the same word count. If so, the code should return just one of them). A sample of input and output is given:

Sample Input (in yelp_data.txt): (review_id, num_stars, review_text)

```
7xGHiLP1vAaGmX6srC_XXw 5 Great place
KpRwKYyQ93ypyDSdA7IXfw 3 Okay place
ZWlXWc9LHPLiOksrp-enyw 1 Bad place fine service
```

Sample Output: (num_stars, (review_id, num_words))

```
(5, (7xGHiLP1vAaGmX6srC_XXw, 2))
(3, (KpRwKYyQ93ypyDSdA7IXfw, 2))
(1, (ZWlXWc9LHPLiOksrp-enyw, 4))
```

```
def parseLine(line):
    tokens = line.split(" ")
    review_text = tokens[2:]
    return (tokens[1], (tokens[0], len(review_text)))

def findMax(value1, value2):
    if value1[1] > value2[1]:
        return value1
    else:
        return value2

if __name__ == '__main__':
    # Assume you have a Spark Context set up
    sc = SparkContext()
    raw_reviews = sc.textFile("yelp_data.txt")
    results = raw_reviews\
                .map(parseLine).reduceByKey(findMax)
```

Q10: If only you could parallelize taking this test…

The following code snippets count the number of nonzero elements in the array.

```
int omp1(double arr[], int size) {
        int nonzero = 0;
        #pragma omp parallel for reduction(+:nonzero)
        for (int i = 0; i < size; i++) {
            if (arr[i] != 0)
                nonzero++;
        }
        return nonzero;
}
```

```
int omp2(double arr[], int size) {
        int nonzero = 0;
        #pragma omp parallel for private(nonzero)
        for (int i = 0; i < size; i++) {
            if (arr[i] != 0)
                nonzero++;
        }
        return nonzero;
}
```

```
int omp3(double arr[], int size) {
        int nonzero = 0;
        #pragma omp parallel for private(nonzero)
        for (int i = 0; i < size; i++) {
            if (arr[i] != 0)
                #pragma omp critical
                nonzero++;
        }
        return nonzero;
}
```

```
int omp4(double arr[], int size) {
        int nonzero = 0;
        #pragma omp parallel for
        for (int i = 0; i < size; i++) {
            if (arr[i] != 0)
                #pragma omp critical
                nonzero++;
        }
        return nonzero;
}
```

```
double omp5(double arr[], int size) {
        int n = 4;
        int res[4] = {0,0,0,0};
        // num_threads(4) runs the code block with 4
           threads.
        #pragma omp parallel num_threads(4) {
            int thread_id = omp_get_thread_num();
            for (int i = thread_id; i < size; i+=n) {
                if (arr[i] != 0)
                     res[thread_id]++;
            }
        }
        return res[0] + res[1] + res[2] + res[3];
}
```

```
double omp6(double arr[], int size) {
        int n = 4;
        int res[4] = {0,0,0,0};

        #pragma omp parallel num_threads(4) {
            int thread_id = omp_get_thread_num();
            #pragma omp for
            for (int i = thread_id; i < size; i+=n) {
                if (arr[i] != 0)
                    res[thread_id]++;
            }
        }
        return res[0] + res[1] + res[2] + res[3];
}
```

1. Which function(s) produce(s) the correct answer? 1, 4, 5

2. Assume cache loading/reloading is slower than thread switching. Which is the slowest correct function if:
   a. the size of the cache block is equal to `4 * sizeof(int)`? 5
   b. the size of the cache block is equal to `sizeof(int)`? 4

3. Which is the fastest correct function? 1

4. Please select the **correct and best** option to fill in the blanks to complete the SIMD implementation of counting the number of nonzero elements in a vector.

```
int sse(double arr[], int size) {
  int num_doubles = ______(a)______;

  int nonzero = 0;
  double *ans = calloc(num_doubles,
                       sizeof(double));

  //Assume sizeof(double) = 8
  __m128d ans_vec = _mm_setzero_pd();
  __m128d zeros = _mm_setzero_pd();
  __m128d ones = _mm_set1_pd(1.0);
  __m128d mask, data;
  int inc = _____________(b)_____________;

  int cutoff = _____________(c)_____________;

  for (int i = 0; i < cutoff; i += inc) {
    data = _mm_loadu_pd ((__m128d *) (arr+i));
    mask = _____________(d.i)_____________;
    mask = _____________(d.ii)_____________;
    ans_vec = _mm_add_pd(ans_vec, mask);
    _____________(e.i)_____________

  }
  _____________(e.ii)_____________

  for (int i = 0; i < num_doubles; i++)
    nonzero += ans[i];

  for (int i = cutoff; i < size; i++) {
   if (arr[i] != 0)
     nonzero++;
  }
  return nonzero;
}
```

a)
- A. 2
- B. 4
- C. 8
- D. 128

b)
- A. num_doubles;
- B. 1
- C. 4
- D. sizeof(__m128d)
- E. sizeof(double)

c)
- A. size/inc*inc
- B. size
- C. inc
- D. size*inc/inc

d.i)
- A. _mm_cmpeq_pd(zeros, data);
- B. _mm_cmpneq_pd(zeros, data);

d.ii)
- C. _mm_and_pd(mask, ones);
- D. _mm_and_pd(mask, zeros);

e.i)
- A. _mm_storeu_pd(ans,ans_vec);
- B. _mm_loadu_pd(ans, ans_vec);
- C. Leave blank/do not add any code

e.ii)
- D. _mm_storeu_pd(ans,ans_vec);
- E. _mm_loadu_pd(ans, ans_vec);
- F. Leave blank/do not add any code

F-2: Not all *optimizations* are created equal (8 points)

For this question, you will be looking at several different versions of the same code that has been, or at least has tried to be, optimized. For each of the versions, indicate the correctness and speed with the appropriate letter:

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

Speed:
A. Faster
B. Same
C. Slower

For reference, here is the serial version of the code:

```
#DEFINE RESULT_ARR_SIZE 8
#DEFINE ARR_SIZE 65536

result[0] = 0;
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];
}
```

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: The answer is actually A, always correct, contrary to the published solutions. This is simply due to the fact that because this code block is wrapped in #pragma omp parallel, every thread will simply be overwriting the result array with the same numbers in each index. This can be proved formally by induction, but the simple explanation is that since result[0] is always 0 for all threads, and every thread will calculate the same sum in the inner for loop, then result[1] must be the same for all threads, even if they overwrite one another. If result[1] must be the same for all threads, and they all calculate the same sum in the inner for loop, then result[2] must be

correct for all threads. This same argument can be made for the rest of the result array. result[0] is the key element that makes this code always correct.

Speed: C, Slower. Due to the overhead of spawning multiple threads to complete the same iterations for all of the for loops.

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++) {
        sum += arr[j] + i;
```



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

Correctness: B, Sometimes correct. This is because of the data dependency on the previous element of result in the last line of code. If result[i-1] is not calculated, result[i] will be wrong. However, in theory, if all of the threads lined up and were scheduled such that result[1] was calculated, then result[2], then result[3], etc., then this code would be correct.

Speed: A, faster. Because we split up the outer for loop between threads, this outer for loop can be done concurrently, which would be faster than the naïve version. The critical section just applies to adding to the result array.

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, always correct. This is because the inner for loop is parallelized to many threads, and preserves the correctness of the sum variable with the reduction keyword. The result array is written to sequentially by in the outer for loop with no false sharing/race conditions.

Speed: A, faster. This is because we can assume that the parallel for in the inner for loop speeds up the execution of the loop enough to dwarf any overhead. Thus, this code would run faster than the naïve version.

d. Consider the correctly parallelized version of the serial code above.

i. Could it ever achieve perfect speedup? F. Amdahl's law says that speedup is limited by the non-parallelizable portion of any code, and that perfect speedup is not possible.

ii. What law provides the answer to this question? Amdahl's Law, per above

F-3: *Map* and *Reduce* are 2nd degree friends - when you also Combine (8 points)

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.

14/17

MapReduce

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

In our map phase, we want to somehow associate a user with their second degree friends. The key here is to recognize that every friend in a user_id's friend_ids list is a second degree friend to every other friend in that list (linked through the user_id). Thus, we want to emit the tuples (friend, friend_ids) to signify that friend is second degree friends with everyone else in a user_id's friends list.

```
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)
```

In reduce, we simply combine all of the second degree friends lists which correspond to a specific user, and then remove that user from the list of second degree friends.

SID: _____

F2: *Why can't you use parallelism at a gas station? It might cause a spark.* (10 points)

1. 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);   // initialize to all 1's (multiplicative identity)

  // vectorize factorial
  for (i = 1 ; i <= 4*(k/4) ; i+=4) {   // largest multiple of 4 ≤ k
    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);   // → put result into f_res array
  for (j = 0 ; j < 4 ; j++) {
    f = f * f_res[j];              // ↖ reduce into f
  }
  // handle tails
  for ( ; i <= k ; i++) {
    f += (double) i;               // ↖ manually handle tail case
  }
  return f;
}
```

SID: _____

F2: (continued)

2. 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]);

}
```

We can pick ANY ordering

Thread 0	Thread 1
lw (1)	lw (3)
add (2)	add (4)
sw (5)	sw (6)

These are the assembly ops req. for the increment

If we execute in the order shown above, then both threads load the old value of res[0] before incrementing instead of going sequentially

As we increase the number of threads running this code:

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

Not always. Consider the case where we have 2 threads, and each has an index that corresponds to an even #. See the explanation above.

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

Yes, the result array can be stored in a single block, so writing to any element will remove the block from all other caches.

c) What about 4 bytes?

No, false sharing is only applicable when the data being accessed by the threads are distinct. If the block is 4 bytes, then a single element of result is in a cache block, so we would invalidate blocks in other caches ONLY if they had the SAME element of the result array. Thus, false sharing is technically not possible.

Q10) Parallelism and Potpourri (30 pts = 6 + 4 + 3 + 3 + 3 + 3 + 4 + 4)

a) What are all the possible values of `x` and `y` after execution has completed if the code were run on two cores concurrently?

`x`: ☐ 0 ☐ 1 ■ 2 ■ 3 ☐ 4
☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9

`y`: ☐ 0 ☐ 1 ☐ 2 ■ 3 ■ 4
■ 5 ■ 6 ■ 7 ☐ 8 ☐ 9

```
int x = 1;
int y = 1;
#pragma omp parallel
{
   x += 1;
   y = x + y;
}
```

SHOW YOUR WORK

x: Both cores could see x = 1 for the first line -> x will end up as 2; otherwise, one core will set x = 2 first, and the other core will set x = 3

y: The key point here is that the two cores are updating shared, global variables -- these updates can occur throughout execution! Let's call core i's execution of line j "ij" -> this results in 4 lines executed (11, 12, 21, 22). Execution order (core 1 and 2 could be flipped for equivalent results):
(11, 21 together), (12, 22 together): y -> 3
(11, 21 together), 12, 22: y -> 5
11, 21, (12, 22 together): y -> 4
11, 12, 21, 22: y -> 6
11, 21, 12, 22: y -> 7

Another way to look at this is to realize for each thread when it executes the line "y = x + y", x equals either 2 or 3. Since each thread adds either +2 or +3 to y, both threads together add +4, +5, or +6.

Additionally, we know that if both threads read y before writing back to it, only one thread will actually properly increment it, giving us +2 or +3. .

b) A Job involves four Tasks, and the % of time spent in each Task is shown in the table. If we buy accelerators that speed up f by 2x and k by 8x what's your *total speedup*?

Task	%
f	10% → 2x
g	4%
h	6%
k	80% → 8x

4x

SHOW YOUR WORK

Old: 10+4+6+80
New: 5+4+6+10
Old/New = 100/25 = 4x

c) Which of the following were discussed in the MapReduce lecture? (select all that apply)

☐ Workers specialize: A "map" worker that finishes their map task early are *only* given a new map task.
Map workers can be reassigned map or reduce tasks

■ The system automatically reassigns tasks if a worker "dies", providing automatic fault-tolerance.

☐ MapReduce was specifically designed for custom high-end machines and custom high-end networks.
Commodity hardware and networks

☐ Hadoop was better than Spark, since Hadoop offered better performance, lazy evaluation, and interactivity.
vice-versa

d) Virtual memory allows us to: (select all that apply)

■ Pretend that programs do not have to share the address space with other programs.

■ Have more stable and secure computer systems.

☐ Divide the entire address space into 4 sections specifically for static, code, heap, and stack.
Nope, that can happen with or without VM

☐ Provide the illusion that the computer has access to storage the size of DRAM but at the speed of disk.
vice-versa

Problem 10 ***[F-2] All Kinds of Parallelism*** **(18 points)**

SIMD within a Register (SWAR) in RISCV: You are planning to obfuscate some messages before they get released to the world. Instead of doing it properly (via encryption), you want a simpler implementation. Your first idea is to add 1 to each character, e.g. turning "aabb" into "bbcc"? If we apply this method to "a quick brown fox jumps over the lazy dog", it becomes "b!rvjdl!cspxo!gpy!kvnqt!pwfs!uif!mb{z!eph"! It looks promising. And the implementation is plain and simple (both in C and RISCV):

```
void obfuscate(char* d, size_t n) {
     for (int i = 0; i < n; i++) {
          d[i] += 1;
     }
}
```

```
obfuscate:
     beqz a1, END
     add t0, x0, x0
LOOP:
     lb t1, 0(a0)
     addi t1, t1, 1
     sb t1, 0(a0)
     addi t0, t0, 1
     addi a0, a0, 1
     blt t0, a1, LOOP
END:
     ret
```

Things look great so far. Then you realize that you learned all kinds of crazy techniques to speedup a function in CS61C and this looks very similar to the many SIMD examples you have seen. Although RISCV does have SIMD instructions via a vector extension set, we want to implement our own version of RISCV SIMD. Our idea is to pack multiple characters into a single 32-bit integer. In fact, we do not even need to load and pack the data: four characters have the same width of an integer. Assume `d` is word aligned and that all input characters in the message are less than 254. Also assume `n` is the number of characters in the message and that register `a0` holds the value of `d` and register `a1` holds the value of `n`.

(a) Complete the following implementation for a vectorized version of `obfuscate`:

Solution:

```
void obfuscate_vec(char* d, size_t n) {
     for (int i = 0; i < n / 4 * 4; i += 4) {
          *( (int*) (d + i) ) += INC;
     }
     /* handle tail cases */
     for (int i = n / 4 * 4; i < n; i++) {
          d[i] += 1;
     }
}
```

(b) Refer to the constant `INC` in the code above. What should the value of `INC` be such that `obfuscate_vec` works correctly? Write your answer in hexadecimal.

Solution: 0x01010101

(c) **Loop Unrolling:** You can optimize this procedure further! Loop unrolling is supposed to reduce the number of branch instructions. Complete the following:

Solution:
```
void obfuscate_vec_unroll(char*
d, size_t n) {
     for (int i = 0; i < n / 8 * 8; i += 8)
{
          *((int*) (d + i)) += INC;
          *((int*) (d + i + 4)) += INC;
     }

     /* handle tail cases */
     for (int i = n / 8 * 8; i < n; i++) {
          d[i] += 1;
     }
}
```

Solution:
```
obfuscate_vec_unrolled:
     beqz a1, END
     add t2, a1, x0
     srli t2, t2, 3
     slli t2, t2, 3
     beqz t2, TAIL
     add t0, x0, x0
     li t3, INC
LOOP_VEC:
     lw t1, 0(a0)
     add t1, t1, t3
     sw t1, 0(a0)
     addi a0, a0, 4
     lw t1, 0(a0)
     add t1, t1, t3
     sw t1, 0(a0)
     addi a0, a0, 4
     addi t0, t0, 8
     blt t0, t2, LOOP_VEC
TAIL:
     add t0, t2, x0
LOOP_TAIL:
     lbu t1, 0(a0)
     addi t1, t1, 1
     sb t1, 0(a0)
     addi t0, t0, 1
     addi a0, a0, 1
     blt t0, a1, LOOP_TAIL
END:
     ret
```

(d) Given a message of length `n` characters, how many instructions are needed after loop unrolling? Express your answer in terms of `n`, such as `3n + 4`. In addition, what is the speed up when `n` is approaching infinity in comparison to the **original non-optimized function** `obfuscate`? Count pseudo-instructions as 1 instruction. You do not need to simplify your expressions.

# of Instructions: $(7 + (n/8) * 10 + 1 + (n\%8) * 6 + 1)$ Speedup: 4.8X

(e) You decide to further improve the code with thread parallelism using 4 threads! Fill in proper OpenMP directive to the blank below:

Solution:

```
void obfuscate_vec_unroll(char* d, size_t n) {
    #pragma omp parallel for
    for (int i = 0; i < n/8*8; i+=8) {
        *((int*) (d + i)) += INC;
        *((int*) (d + i + 4)) += INC;
    }
    # handle tail cases
    for (int i = n/8*8; i < n; i++) {
        d[i] += 1;
    }
}
```

Someone tells you to add `#pragma omp parallel` at Location A in the code above. If you do this, which statement is true about the second `for` loop (the tail case)?

- ○ Always Incorrect
- ○ Always Correct, slower than serial
- ● Sometimes Correct
- ○ Always Correct, faster than serial

(f) Denote the speedup when `n` is approaching infinity of `obfuscate_vec_unroll` from part (d) as "S". Suppose the overhead of running OpenMP is negligible in comparison to the rest of the code, and we can run **four** threads, what is the maximum speed up compared to the **original non optimized function** `obfuscate`?

Solution: $4 * S$

(g) **WSC and Amdahl's Law**: The above program now runs in the cloud with many machines. `obfuscation_vec_unroll` is 90% of all execution (AFTER applying SWAR, unrolling, and OpenMP), and `obfuscation_vec_unroll` can be parallelized across machines.

(i) If we run `obfuscate_vec_unroll` on a cluster of 16 machines, what is the speedup? You may leave your answer as an expression.

Solution: $1/(0.1 + 0.9/16) = 6.4$

(ii) What is the maximum possible speedup we can achieve if we have an unlimited number of machines?

Solution: $1/0.1 = 10.0$

Problem 11 ***[F-3] Pikachu Learns Spark*** **(16 points)**

We are given the entire dataset of every Pokémon and we want to **find the mean of all Pokémon id numbers by type**. Some Pokémon have dual types so that Pokémon's id number will contribute to the average total of both types. For example, Kyurem is both a dragon and an ice type so his id number will contribute to both type's sum when considering the average. **Fill in the blanks for the Python code below**. Use the following Spark Python functions when necessary: `map, flatMap, reduce, reduceByKey`.

Sample input (pokemon_id, pokemon_name, pokemon_types):

```
646 Kyurem Dragon Ice
25 Pikachu Electric
257 Blaziken Fire Fighting
```

Sample output (Type, Number):

```
(Dragon, 587)
(Electric, 412)
```

Solution:

```
def parseLine(line):
    tokens = line.split(" ")
    types = tokens[2:]
    results = []
    for type in types:
        results.append((type, (tokens[0], 1))
    return results


def reduceFunc(v1, v2):

    return (v1[0] + v2[0], v1[1] + v2[1])


def average(k, v):

    return (k, v[0] / v[1])


pokemonData = sc.parallelize(pokemon)

out = pokemonData.flatMap(parseLine)

                 .reduceByKey(reduceFunc)

                 .map(average)
```

SID: _____

Question 9: SIMD Once Told Me (10 pts)

In lecture we showed you how to use Intel Intrinsics to perform an efficient Matrix Multiplication. For this question you are going to perform a similar operation, an inner product between an **m x n** matrix **A** and an **n x 1** vector **b**. This formula can be expressed as:

$$\mathbf{C_i} = \sum_{j=1}^{n} A_{i,j}\, b_j$$

To make this question more interesting, rather than working with the 128 bit registers we showed in lecture, we will instead work with 256 bit registers. You can select from the following SIMD functions, each of which has a brief explanation.

`__m256d _mm256_setzero_pd ()`	Produce a `__m256d` with all 0s.
`__m256d _mm256_loadu_pd (double *)`	Loads the doubles at the ptr's address into a `__m256d`.
`__m256d _mm256_load_pd (double *)`	Loads the doubles at the ptr's address into a `__m256d`. This is faster, but the ptr passed in must always be divisible by 32 (or else a segfault will occur).
`void _mm256_storeu_pd (double *, __m256d)`	Stores the contents of the `__m256d` in the memory location pointed to.
`void _mm256_store_pd (double *, __m256d)`	Stores the contents of the `__m256d` in the memory location pointed to. This is faster, but the ptr passed in must always be divisible by 32 (or else a segfault will occur).
`__m256d _mm256_fmadd_pd (__m256d A, __m256d B, __m256d C)`	Returns `(A * B) + C` by performing operations on packed doubles.

You may assume that all pointers passed into the function and any stack variables allocated are at addresses divisible by 32. You may also assume **`sizeof (double) = 8`.** Fill in the function below. You must use the **fastest load and store** whenever possible.

/* Computes an inner product between **SM**, which is a **M x N** matrix, and **SV**, which is
 * is a **N x 1** vector. This result is stored in **D**, which is a **M x 1** vector. */

```
void inner_product_simd (double *d, double *sm, double *sv, unsigned m, unsigned n) {
      __m256d matrix_part;
      __m256d vector_part;
      __m256d total;
      double arr[4];
      for (int i = 0; i < m; i += 1) {
            total = _mm256_setzero_pd ();
            for (int j = 0; j < n / 4 * 4; j += 4) {
                  if ((i * n + j) % 4 == 0) {  // Check if the address is aligned.
                        matrix_part = _mm256_load_pd (sm + i * n + j);
                  } else {
                        matrix_part = _mm256_loadu_pd (sm + i * n + j);
                  }
                  vector_part =  _mm256_load_pd (sv + j);
                  total = _mm256_fmadd_pd (matrix_part, vector_part, total);
            }
           _mm256_store_pd (arr, total);
            d[i] = arr[0] + arr[1] + arr[2] + arr[3];
      }
      if (n / 4 * 4 != n) {  // Check if there is a tail case.
            for (int i = 0; i < m; i += 1) {
                  for (int j = n / 4 * 4; j < n; j += 1) {
                        d[i] += sm[i * n + j] * sv[j];
                   }
            }
      }
}
```

SID: _____

Question 10: Hashtag Pragma (10 pts)

In OMP we have critical sections which need to be resolved via atomic instructions in assembly. When trying to execute this OpenMP code in multiple threads:

```
#pragma omp critical
{
      x++
}
```

a student produces this assembly to implement it:

```
try:  lw t0 0(s0) #load in the value of x from the address
      addi t2 t0 1
      amoswap.w.aq t1, t2, (s0)
      bne t1 t2 try #check if the value of x was the one previously loaded
End:
```

1. This code DOES NOT ensure the critical section works properly. Select **all of the following** options that accurately describes the code the student produced.
   - Ⓐ The code increments x every time it attempts store. This problem can be solved by restoring x to the value of `t0` before trying again.
   - Ⓑ Amoswap successfully changes the value at the address of `s0` and returns the old value in memory in a single, uninterrupted operation.
   - Ⓒ If instead of a `lw`, `addi`, and `amoswap`, a single `amoadd` instruction was used to add 1 to x, then the code would work properly.
   - Ⓓ The code can still work properly in some cases.

When working with fully associative caches, you learned that a **cache will search each block in parallel**. Let's envision how this could be simulated in software using openmp. Imagine you have a cache struct that looks like this:

```
typedef struct cache {
      uint32_t *blocks;
      unsigned num_blocks;
} cache;
```

Additionally imagine a cache block stores both data and metadata in a **single 32 bit value** and does so **big endian**. It has the following details:

- Bit 31 (MSB): Valid Bit
- Bit 30: Dirty Bit
- Bits 29 - 16 (14 total bits): Tag
- Bits 15 - 0 (16 total bits): Data

2. Complete the function for handling the **parallel search of each cache block**. It takes in a cache struct, a pointer to data (where data should be stored), and a tag. It should update two shared variables; the first a boolean value indicating if the data was stored in the cache; the second a pointer whose contents should be updated with the data. For this portion **assume we will have as many hardware threads as we do cache blocks.**

```
bool search_cache (cache *cache, uint16_t *data_ptr, uint16_t tag) {
	bool found = false;
	unsigned num_blocks = cache->num_blocks;
	#pragma omp parallel for
	{
	for (int i = 0; i < num_blocks; i++) {
		uint32_t contents = cache->blocks[i];
		uint32_t valid = contents & (0x80000000);
		if (valid) {
			uint32_t block_tag = (contents >> 16) & (0x3FFF);
			if (block_tag == tag) {
				uint32_t data = contents & (0xFFFF);
				#pragma omp critical
				{
					found = true;
					*data_ptr = data;
				}
			}
		}
	}
	}
	return found;
}
```

3. Now let's **assume we may have fewer hardware threads than cache blocks**. Which of the follow statements about the code above represent ways the code can be optimized? Select **all** that are true:

Ⓐ When a thread finds that it contains the data, it can break out of the loop.

Ⓑ As we increase our number of threads we will see little to no speedup because a critical section is included.

Ⓒ If our cache works properly, no matter how many threads exist in hardware we can completely remove the critical pragma.

Ⓓ If the number of hardware threads is near the number of cache blocks, then we run the risk of having false sharing occur in our code.

Question 13: South Spark (6 pts)

In this question you will write Spark to find the mode of a list of values and how often it occurs. As a refresher the mode is the number that appears most often. If there is a tie you can select any of the options. **Fill in the blanks for the Python code below.** Use the following Spark Python functions when necessary: **map**, **flatmap**, **reduce**, **reduceByKey**. Here is a sample input and output:

```
#Input: [1, 2, 1, 2, 3, 4, 5, 6, 4, 2, 1, 3, 3, 1, 1, 2, 2, 1]
#Output: (1, 6)

def output_data (val):
        return  (val, 1)

def compute_count (a, b):
        return  a + b

def find_max_occurrence (a, b):
        if a[1] > b[1]:
                return a
        return b

#values = list (numbers)
modeData = sc.parallelize (values)

modeData.map (output_data)
            .reduceByKey (compute_count)
            .reduce (find_max_occurrence)
```