ECE459 Midterm W2013

Question 1

1. Not covered. In general, #pragma omp master is likely to be faster than #pragma omp single because it doesn't have an implicit barrier at the end of the block, while #pragma omp single does.
2. Since there are more threads than cores in the 9-thread version, there is added overhead from context-switching between the threads, since only 8 of them can run at a time. There is also potentially additional resource contention.
3. Not covered?
4. Not covered. The volatile qualifier ensures that a variable's value is read directly from memory each time it's accessed, preventing certain compiler optimizations. It does not guarantee atomicity or thread safety: multiple threads can still access the variable concurrently.
5. Not covered? r1, r2 = 0
6. Not covered.
7. Not covered.
8. Not covered.
9. Not covered.
10. If the iterations of the outermost loop have dependencies and the middle/inner loops do not; or if the outermost loop has less iterations than the number of cores but the middle/inner loops have more.

Question 2

a) Not covered. xor is a smaller instruction than mov. Also, processors optimize by recognizing the xor to self, making it a zero-cycle statement.

b) Not covered?

Variable	First line	Second line	Type of dependency
eax	1	2	RAW
eax	2	3	WAR
eax	3	4	RAW

Line 1 must execute before line 2. Line 2 must execute before line 3. Line 3 must execute before line 4.

c) We use a new register.
    a. add eax, 1
    b. mov ebx, eax
    c. xor edx, edx
    d. add edx, ecx

    Line 1 must execute before line 2. Line 3 must execute before line 4.

Question 3

Not covered.

Question 4

a) When two threads call isPrime with the same value of v concurrently, they might both determine that v is not prime and try to update pflag[v] simultaneously. We can use a mutex to eliminate the race condition, by initializing the mutex before calling is_prime, locking the mutex before updating pflag[v], and unlocking afterwards. The race condition is benign, as all threads set pflag[v] to the same value of 0, so the order of writes does not impact the output.

b) Not covered? The pInstance = new Singleton line involves allocating memory for the Singleton object, constructing it in the memory, and assigning the memory address to the variable pInstance. Due to instruction reordering, the memory address may be assigned before the object is constructed. Hence we might have an execution as follows.
    a. Thread 1 enters instance(), sees that pInstance = 0, acquires the lock, sees that pInstance = 0, and executes pInstance = new Singleton. It allocates memory and assigns the memory address to the variable pInstance.
    b. Thread 2 sees pInstance != 0 and returns the incomplete Singleton.