Brian Rodriguez

Quiz 4

1. The original double-checking locking pattern initializes and writes to the Helper object without a lock. This means that the initialization and writes can be done out of order to the compilers dismay. The author goes on about different types of implementations of the double-checking locking pattern, all of which did not work. He makes the suggestion to just use a static singleton pattern for all threads as opposed to creating a new Helper Object for each Foo object. This guarantees that any thread with access to the object will be able to see writes resulting from initializing the field. The role of java volatiles allows the developer to guarantee that the compiler cannot reorder with respect to any previous read or write. Meaning that any read or write to the java cannot be reordered to the compilers dismay.

2. Synchronizes-with implies happens-before because it uses similar semantics. By this I mean that is uses acquire-release and write-release semantics. This ensures that in order to make a change to the Object being worked on that their must be a consumption of an atomic variable first, then once the work has been done the thread doing the work must release the atomic variable for other threads to do work.

3. The double-check locking pattern was fixed in C++11, when C++ got a new memory model and an addition of atomic variables which enables a wide variety of double-checking locking patterns. The DCLP used in the diagram depicts the seamless transition of a acquire-release of a C++ atomic variable. The diagram depicts the guarantee that all the writes that happen in the first thread will be seen by the second thread. Without the atomic fence, then there is no guarantee that the writes will be seen by other threads. This also works because of the singleton pattern used as the instance variable.

4. The acquire-release constructs that are available in C++ are contained within the atomic variables themselves. By default, the atomic variables are sequentially consistent and allow the programmer to maintain specific ordering of the program to a certain degree. The atomic variable itself has store and load constructs which enable to the developer to see if a thread can acquire or release the variable being consumed.

5. Atomics are implemented in OpenMP using the CAS operation. The CAS operation essentially swaps two registers if the first register is equal to the first argument. If the operation fails and the swap cannot happen then the thread was not able to access that variable and must try again. This ensures that the work of the thread will be done but the thread has to wait to try again. The speaker begins to talk about how data transfer when you start to scale up your machine is the actual overhead of multithreading and not the CAS instruction. He says that the more you scale programs and machines the cost of loading more and more data will start to dominate the operation of CAS. The large server is doing a comparison between static and dynamic scheduling. Static

scheduling is out racing dynamic since static scheduling does not need to reassign a problem size to each thread being ran. Tim Mattson mentions that atomics deal with operations on the hardware level when the instructions are available, alluding to the fact that CAS is used as a hardware specific operation to be done when using atomics.

6. The speaker stressed on the fact that when threads share the same space in main memory, the system has to do more overhead work to make sure that a thread is not working on memory that is invalid. He suggests to use private data and reading is fine. The memory hierarchy diagram depicts a cc-NUMA architecture with a couple of memory caches and shared caches, so that the threads can use the same shared memory spaces. The speaker cannot stress enough about data placement. It seems as if data placement alone can be the center of many parallel algorithms that have been created. Initializing and reading from an array reads in a chunk of the array allowing for better cache coherence. The role of profiling is so that the developer can understand where idle time and compute time each thread is undergoing. In the example that he showed, the graphs clearly showed that the master threads were undergoing a lot of barrier initialization. This is a depiction of false sharing in the program that he created. His mentioning about cache blocking is a bit hidden but when he changes his code to use the thread's id to split the work up he eliminates blocking completely. He uses the thread's id to access different parts of the array being worked on.

7. This example was a bit tough to understand at first. I believe that he meant to say not "index c" but construct c or maybe it was "index I" instead of index c. It would make more sense. In the example, index j is being used and so is index i. They are being used to calculate the struct c, since they are being used to compute c, it would make more sense that c is a private variable to eliminate the race conditions when computing c.

8. Tasks are a convenient way for an OpenMP program to handle parts of the program in a paralleled fashion. Tasks allow for threads to do some work, then when the task is finished, the threads are all waited on so that the program can be executed again. Depending on the program this is not always the case unless other wised specified by a #pragma omp taskawait clause.

9. OpenMP's implementation is very clever but requires of a lot of under the hood work for it to be implemented with the original source code it is given. There are many stages of compiling the code it is given, the three stages are the frontend, middle, and backend stage. Both serve their own purpose in transforming the original code it is given. The stages optimize aspects in the code as well as make optimization for the later stages simpler. The outlining OpenMP does is very clever in the fact that it will optimize the code that is using the directives into it's own function and split the work up by having threads being created outside of the functions. When the reduction directive is present, the compilation uses clever tricks like creating a local variable so that there are no race conditions and uses critical directives to signal that adding to the sum is a critical section.

10. OpenMP and TBB are very powerful libraries, both serve their own purpose as a parallel library. TBB is more of a templated library meaning that you are allowed more degrees of freedom while maintaining the originality of the developer's code. OpenMP on the other-side is more of a cookie-cutter library. It uses directives to modify the existing code so that it can be compiled in a very specific way. OpenMP to me is more of a very fast prototyping library and may not scale as well as TBB. TBB on the other hand takes some time to build code but the way the code is build serves its purposed to be used in production. If I were doing my final project in C or C++ then I would use TBB. Although a couple of weeks ago, I enjoyed OpenMP more but now that I have spent time with both libraries I have more of a passion towards TBB and will take the extra work it takes to build a scalable application, rather than a fast prototype.