PDS: Homework Assignment 3

Andreas De Lille

Q1.1

The first error is visible in the console output and is about the threadID, which is passed to the thread by reference. the problem here is that all thread point to the same memory address as the iterator from the loop. If the iterator is incremented before a thread writes his id, then his ID will be wrong. The solution is simple, pass it by value.

Q1.2

The second error is more subtle, since the calculated pi value seems to be correct. However, pi is a shared value, the threads all write to this variable, therefore it should be protected by a mutex, if not then race condition occur. If two threads execute the pi+= element at the same time e.g. 2 thread do pi += 1 then the end result won't be the expected pi +2, but pi+1. Again the solution is simple, by using a mutex to protect the critical pi+= statement, race condition are removed.

As a sidenote, i would also like to point out that valgrind also mentioned cout, since it's essentially a single stream, shared between all the threads. Additionally, depending on the implementation of the randomgenerator, all 'random values' use the same seed. This means that the random numbers will be the same for each thread.

Q2.1 & Q2.2

# Threads	**Time (s) -O0**	**Speedup -O0**	**Time (s) -O3**	**Speedup -O3**
1	0.31s		0.13	
2	0.44	0.7045	0.06	2.1667
4	0.4275	0.7251	0.0325	4
8	0.21125	1.4675	0.02125	6.1176
16	0.228125	1.3589	0.019375	6.7097

Q2.3

When the -O3 parameter is used (all optimizations on), there is a near perfect speedup for 2 and 4 threads (the 2.1667 speedup is due to rounding errors). Using 8 threads still gives some speedup, using even more threads results in almost no additional speedup. This is due to the thread overhead; the usage of more threads causes more overhead which, in turn, results in lower speedups.

The difference between O0 and O3 is, obviously the little speedup in O0. The reason for this is false sharing; False sharing occurs when multiple threads (on multiple cores that each have their own cache) write to the same cache line. The MESI protocol will then mark the whole line as modified, thus the other cores have to refetch this cache line, even though the used variable (or in this case, position in the array) is not shared. To solve this the compiler has to align the array so that there is no overlap. Another solution is to create a local variable in the thread-scope, this will be allocated as close to the core as possible, which means that each variable will be in its own cache.