Rank of performance
->Automatic reduction (0.005279 sec)
->Manual reduction b (0.006802 sec)
->Manual reduction a (0.007846 sec)
->Serial method (0.012798 sec)
->With atomic (0.033914 sec)
->With critical (0.041153 sec)

Questions
1. Why are the fastest method(s) fast?
2. Why are the slowest method(s) slow?
3. The \Manual Reduction A" and \Manual Reduction B" methods above appear very similar. If you
notice a performance discrepancy between the two, why?
4. The \Atomics" and \Critical" methods may appear similar in that they are both instructing OpenMP
to protect access to the num hits variable. If you notice a performance discrepancy between the two,
why?

The slowest methods are parallelization with atomic and critical statements inside the parallel for. The conclusion is that synchronizing the threads so that only one of them can access the line of code that is protected is very expensive, especially in a parallel for. Threads have to wait in each iteration of the for, making the process very slow. Parallelization with atomic and critical have similar speed, but atomic is faster. This is consistent with theory - we know atomic is very similar to critical but more efficient. In some architectures, it uses a dedicated assembly instruction for the operation.

Next comes the serial method, which help us notice that parallelization is not good at any cost. This is just an alert that, if not using properly and efficiently the synchronization methods, the serial method can sometimes be more efficient than the parallelization.

Following in time speed comes the manual reduction methods. This are quite efficient mainly because they parallelize the for loop but do not have the constraint of limiting the access of a line of code to one thread at a time in the parallel for loop, as the atomic and critical methods did. Now, we can see "manual reduction b" is quicker than "manual reduction a". This can be explained because, in the case of b method, the addition of the local num_hits (num_hits += num_hits_local) is done for every thread as soon as they finish their calculation of local_num_hits (and as soon as they can enter the atomic section), regardless if other threads have not finished their own calculation of local_num_hits. This is the main difference with "manual reduction a", which needs all threads to finish before it starts adding up the local num_hits to the global one.

Automatic reduction is clearly the fastest method. It is the native implementation in OpenMP for the same functionality as "manual reduction b" method, creating local num_hits and adding them up in the end to a global num_hits. Due to the native implementation, its more efficient than the manual method.

Discussion

Describe briefly how you would choose to efficiently implement the Monte Carlo dartboard Pi method if using pthreads instead of OpenMP. Also discuss how you would control access to the num hits state variable via pthreads so as to maximize efficiency.

If using pthreads, we should use pthread_mutex_t in order to control access to operations that have to be done one thread at a time. The mutex shall be locked before adding to the global_sum, and unlocked after the summation is done.

To maximize efficiency, we should create local variables of num_hits for each threat, and then add them to the global num_hit with the mutex, as we did in manual reduction b. This way, the computation of local num_hits can run in parallel without the restriction of the mutex.