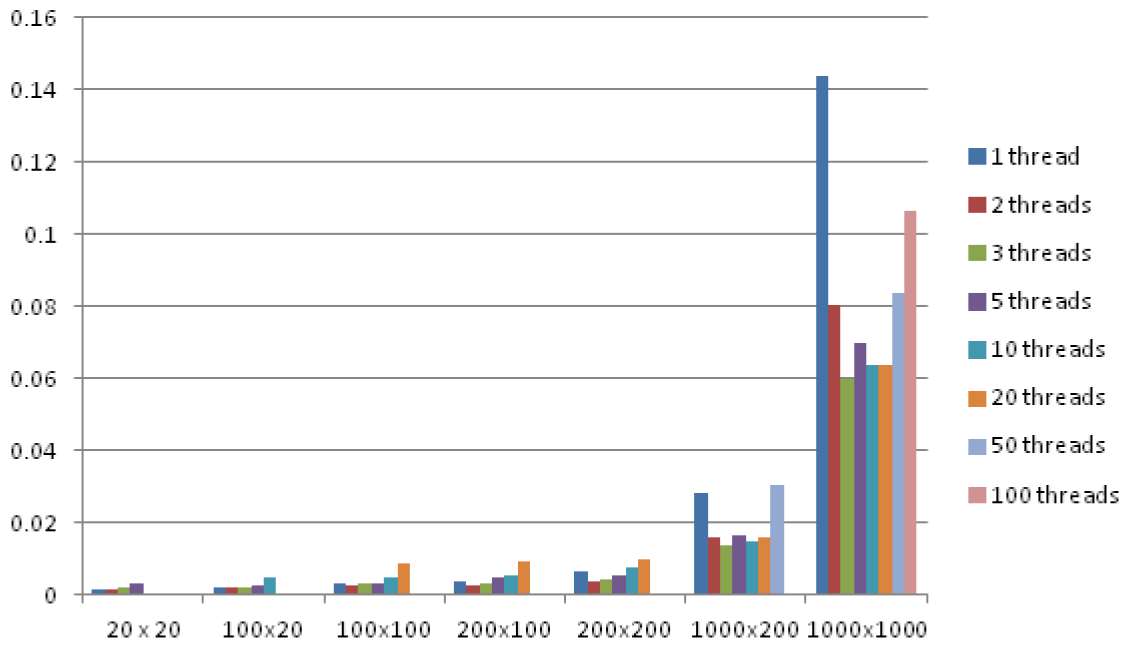
Computer Operating Environments

To obtain results about the performance of the application I made a few small changes to the code. The code that I timed is simply how long it takes for all of the threads calling the make_sphere function to finish. For this test there are no calls to Sleep() and the make_sphere threads are the only threads that were run. A median of the results found was taken and compiled in to the table below. The test was run on a laptop with an Intel Core i5 M 460 @ 2.53GHz CPU.

# of threads ARRAY SIZE	1 thread	2 threads	3 threads	5 threads	10 threads	20 threads	50 threads	100 threads
20x20	0.001296	0.001473	0.001907	0.002897				
100x20	0.001852	0.001952	0.002189	0.002372	0.004588			(all times in s)
100x100	0.002259	0.001967	0.002270	0.002672	0.004961	0.008535		
200x100	0.003518	0.002778	0.003148	0.004615	0.005360	0.009172		
200x200	0.006336	0.003905	0.004082	0.005139	0.007454	0.010053		
1000x200	0.028078	0.015720	0.013877	0.016217	0.014910	0.016083	0.030306	
1000x1000	0.143793	0.080289	0.059679	0.069689	0.063677	0.063644	0.083504	0.106462



This graph shows the median times the program took to calculate all vertices for spheres of varying sizes, with varying numbers of threads.

When collecting the results i noticed there were some cases of extreme outliers, sometimes the threads would complete their tasks very quickly and other times they would take significantly longer, as a result I decided to take the median of the results and just focus on average cases. However these outlying times are interesting to note, they may just be related to the CPU and fluctuations in background programs, but it's also possible that they were caused by other factors.

The Results I collected show that at very low numbers of calculations to be performed there is no benefit to introducing threading to a program, the overhead of starting threads outweighs any time saved in performing the calculations faster. As more calculations are required we can begin to see the benefit of introducing threads to our program, for the middle three sphere sizes two threads was the ideal number, and the difference between running one thread or three threads was pretty small. At even larger numbers of calculations it is much clearer the value in using threads, Running 3 threads for the last 2 sphere sizes more than halved the time taken for the vertex array to be filled. Though there is some fluctuation in the results (perhaps these might be even out with a larger sample size of results) the general picture is pretty clear that running between 2 and 5 threads can likely give you a performance boost when performing a large number of calculations.

From these results I would say that either two or three threads would be the best solution, depending on the intensity of processing required. Even at a relatively small number of calculations having threads did not significantly decrease time taken for the threads to execute. Even at much larger number of calculations running two threads is acceptable since it’s a significant improvement over a single thread. However if intensive processing is expected then I would recommend three threads, since if my results are to be trusted having three threads is a good amount and often gave best or second best performance.

The CPU these results were tested on is a dual core processor, since each core can run more than one thread, and there is also the main thread being executed, our results make sense. There is little point in creating additional threads if the CPU is already being fully used, but if it is not running an additional thread should speed up execution of the task. I theorise that if the test were run on a computer with more cores and greater processing power the results would be a little different, favouring greater numbers of thread. If the test were run on a CPU with just a single core or less processing power, the faster times would more likely favour only running one or two threads.