Brian Grant

Results of Monte Carlo Method on integration.

I implemented a lock-less version of the multithreaded Monte Carlo integration, with the hope of cutting down on execution time by eliminating any waiting for the lock. I also kept variables local, and minimized my usage of dynamic memory to minimize execution time via memory accesses.

The speedup increases somewhat linearly as the number of threads increase. This is because initially each thread has its own dedicated core. There is a drop in speedup at around 8 threads. I have thought hard about this, and I can not come up with a reason as to why this might have happened. In previous executions this did not happen, so I am assuming that this was just some sort of anomaly which occurred in the hardware.

There is a spike in efficiency at 12 threads because this is when each core is utilized, and each core only has one thread, thus there is no hyperthreading to slow down execution. After 12 there is a drop in speedup. This is because the threads that are on a core by themselves finish first, and then have to wait for the other multithreaded cores to finish. This is a result of an implementation detail in my code.

I split the number of samples into roughly equal amounts and gave each thread the same workload. If I had instead given each thread a smaller workload to then replenish when it was finished with the given workload, the threads which shared the same core would have done less work, and thus each thread would have finished at roughly the same time.

To Implement this I would have had a locked global count of number of samples taken. Each thread would be given say, 100 samples to execute initially, and the counter would be incremented by 100. Then when it finished that 100 samples, it would obtain the lock, and gather another 100 samples, once again incrementing the counter by 100. This would continue until the last thread takes the last amount of samples which would be less than or equal to 100. The exit condition would be that this global counter equals the specified parameter for total samples taken. If I implemented this I would have had to use locks. Hopefully my lock-less implementation sped up the execution due to not having to wait for a lock, amortizing the slow down which resulted from this load imbalance.

There is a large drop in speedup around 24 threads. This is because each core allows two hyperthreads. After 24 threads the scheduler needs to be utilized to share the resource between each thread on the same core. This results in context switching, which takes time, decreasing speedup.

After the drop around 24 the speedup increases, but at a much slower rate than it did between 1 and 12 threads. I can speculate that speedup slightly increases after the large drop because we set cpu affinity. Even though there are context switches taking place after 24, the threads on a certain core remain on that core when they are descheduled, this decreases cache misses in the associated cache. If we did not set the affinity, the descheduled thread could be scheduled on a different core, resulting in a cold start penalty. This would reduce performance.

The final drop in my graph takes place at roughly 36 threads. This is because after 36 threads, a core now has to context switch between 2 threads, one for each hyper thread. There is a total of 4 threads on one core, while the other cores are contending with only three. The threads on the cores with less threads finish first and have to wait for the 4 threads on the single core to finish. This pattern continues and the load is imbalanced. But once again, as threads increase, speedup starts to slowly increase as

well. I speculate that if we were to continue adding threads, we would see another drop in speedup at around 48 threads for the same reason.

This was a very interesting assignment. If I were to do it again, I would have implemented the load balancing, smaller chunk implementation described above.