Comparison of Partial Sum Aggregation Techniques

Philip Nelson

17th September 2021

Motivation

I was curious to discover the cost of a barrier statement and compare mutex guarded aggregation to single threaded vector aggregation. The results were partially expected and surprising. I expected the tree style aggregation to perform worse on increasing number of threads due to the increased cost of synchronization and single threaded vector aggregation to perform the best because I thought it would have the least amount of synchronization and overhead.

Methods

I implemented three methods for aggregating the partial sums of the PI approximation:

- A single thread adding up all partial sums from a vector
- Threads working together to add partial sums in a tree style aggregation
- Threads adding partial sums to a global sum guarded my a mutex

To synchronize the threads, I was going to use `std::barrier` but it turns out it is a c++20 feature that was not available on my compiler, so instead of upgrading `gcc` I implemented a simple, reusable barrier with `std::mutex` and `std::condition_variable`. The tests were run on a 6 core / 12 thread Intel Xeon CPU. The approximation was run with $2.15e9$ iterations on varying number of threads. Each test was only run once so the data is likely noisier than it could have been.

Results

As I expected, the tree like aggregation performed worse with larger numbers of threads, see figure 2. This is likely due to extra cost incurred by synchronization at every step of the aggregation algorithm. I would be interested in running this test on a processor with many times more cores and I think it could perform much better. It is interesting to note that while the tree style aggregation might perform better in a message passing context because the cost of passing a message is so expensive, that does not hold for a multi thread context.

I was surprised to find that the mutex guarded global sum method performed the best.

The other interesting feature of note is that all the methods performed very similarly up to the hardware limit, 12 threads. Since I was timing the execution of the whole approximation function, the portion of time devoted to aggregation is likely small in comparison. It would be interesting to time just the aggregation step to see a different view of the performance.

Figure 1 also includes the single threaded program for reference.

Table 1 shows the speedup attained at 12 threads.

Table 1: Speedup for 12 threads

	12 threads (ms)	Speedup
one sum	395.74	5.49
tree sum	387.99	5.60
mutex sum	379.76	5.72

Future Work

It would be interesting to time just the aggregation step

It would be interesting to run on a CPU with a larger hardware limit, ie an AMD Ryzen Threadripper.

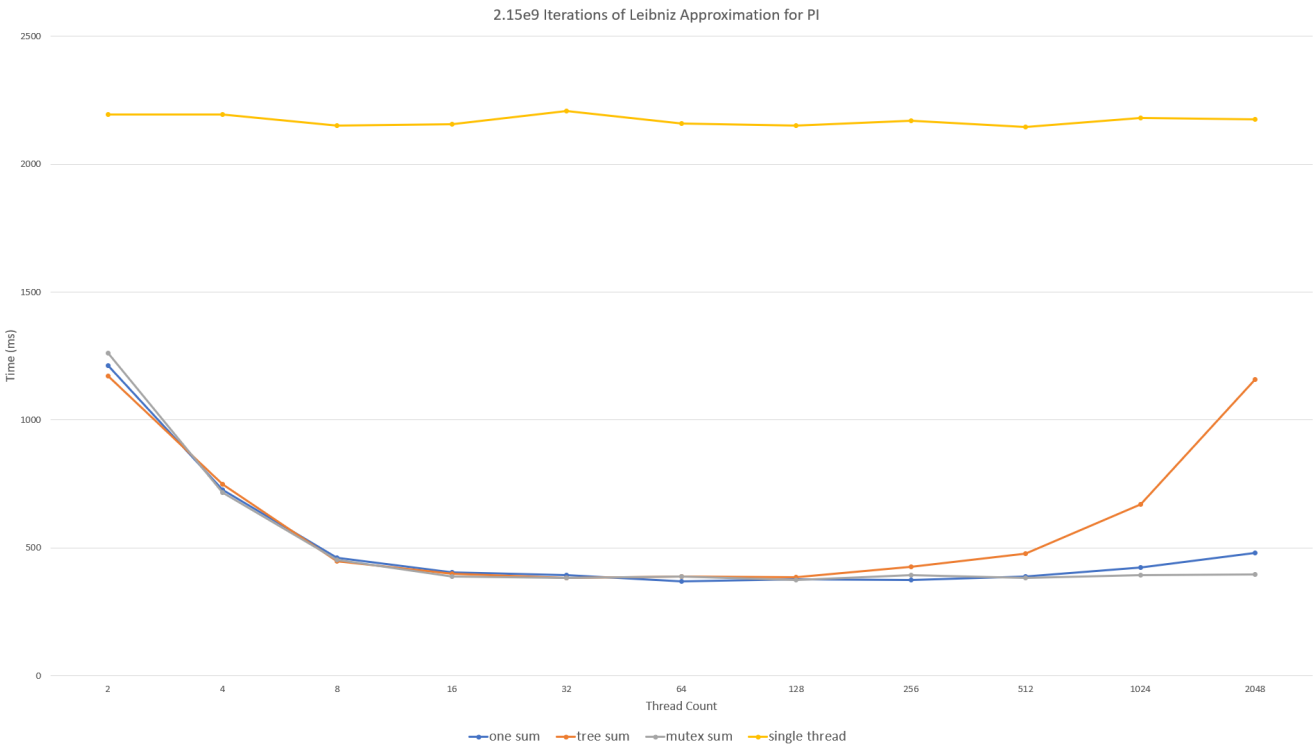

Figure 1: Thread Count increasing as powers of 2

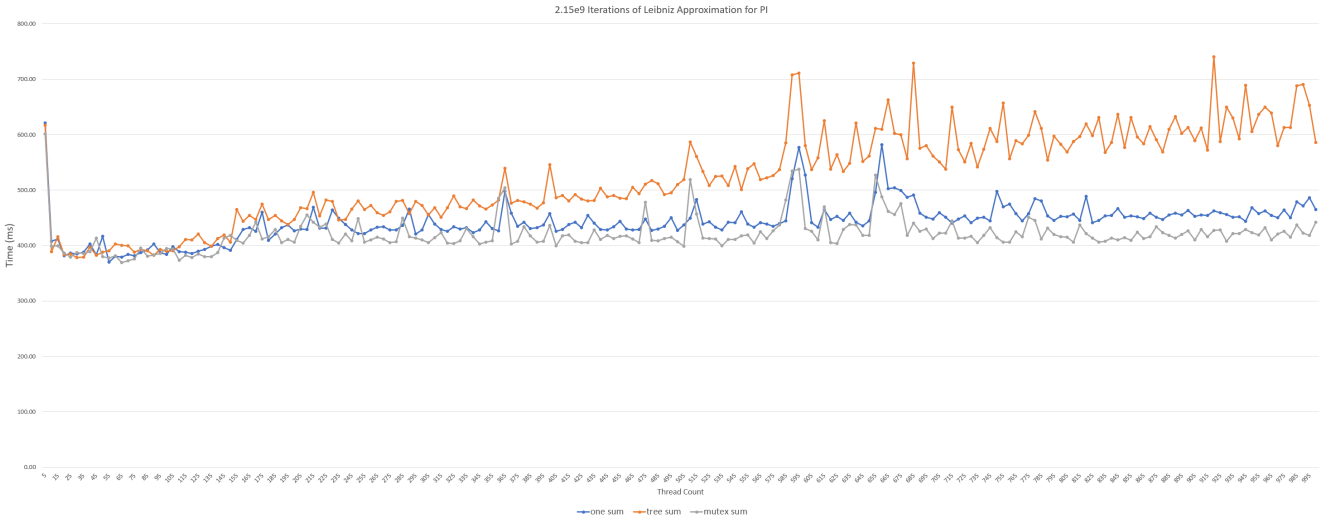

Figure 2: Thread Count increasing from 2 - 1000 by 5

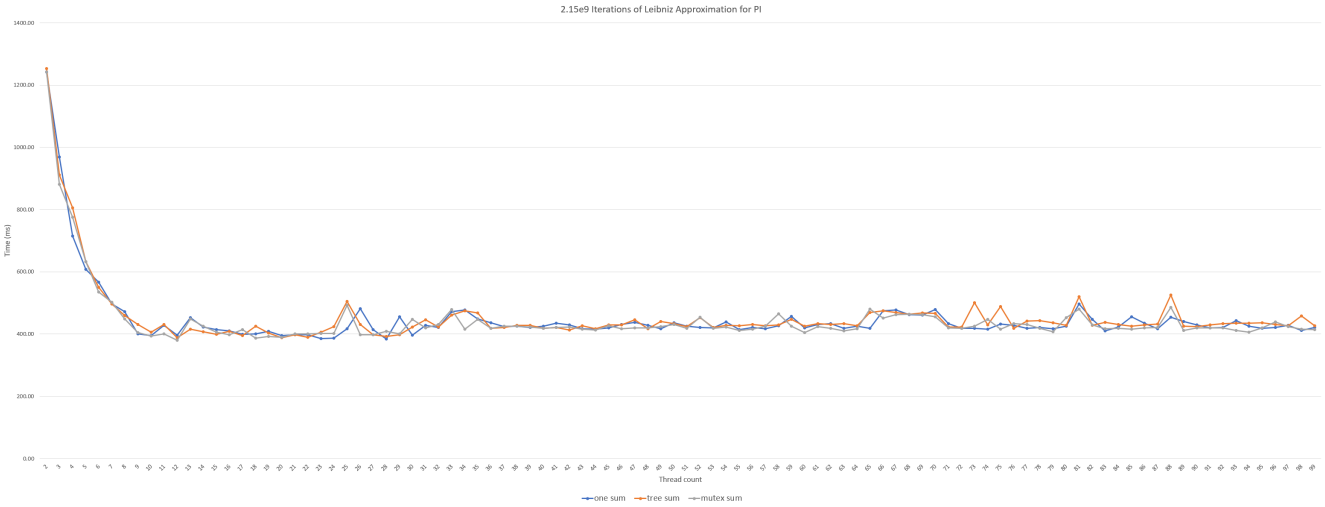

Figure 3: Thread Count increasing from 2 - 100 by 1