Colin Mitchell

u0535755

CS6013

Analysis – C++ and OpenMP Parallel Reductions

For the following timings, all timings were obtained by filling an array with each element equal to 1, and then averaging the time it too to add the elements of the array together, using 3 different functions (using plain C++, a custom OpenMP version, and the built-in OpenMP reduction). I repeated each function 10,000 times and took the average to get the results shown.

The following chart shows when I tested my C++ function with different array sizes. I had started with arrays of 1000 and 2000 elements, but I saw no time improvement when splitting the work among threads, in fact It took longer to add the elements of the array. I increased the array size to see if that made a difference, and did so using powers of 10 until I ultimately reached an array size of 10 million. It was with the array size of 1 million that a significant time reduction was observed with multiple threads, and while the same held true for the array with 10 million elements, the inclusion of those times made the graph hard to read, so I made two charts, one with timings for array sizes 1 thousand, 2 thousand, 10 thousand, 100 thousand, and 1 million, with another graph showing array sizes 1 million and 10 million.

As can be seen, for every array of size less than 1 million, it took more time with multiple threads compared to a single thread. I would argue that this happened because there is a cost to calling multiple threads and then joining them again, so the array needs to be of a significant size to show any benefit to using multiple threads. The array of size 1 million showed improvement with each additional thread, until it reached 4 threads, and the time started to increase with each thread, ultimately getting slower with each additional thread, with using 11-16 threads actually being slower than using a single thread.

With an array of 10 million, there was a similar trend, with each additional thread making the function run faster (more or less) until 5 threads. Adding more threads than that were still better than a single thread with an array of that size (it never became slower than with a single thread), but slower than 5 threads.

The computer I was running these tests on has 4 physical cores, so it actually makes sense that with the array sizes where I was able see an improvement with more threads, it continued to improve until I was adding more threads than there were cores.

Here are the graphs for the custom OpenMP version of the function. I ran the same experiments as above.

One difference with this function was that we actually are able to observe an improvement in timings with a smaller array size, starting with 100,000. While slight, you can see that timings improve with each thread until 6 threads are added, and then 9 threads the time has become longer than with a single thread. Significantly, with an array of size 1 million, the timings never got worse than with a single thread, like it did with the plain C++ version.

With an array of 10 million, the timings improve until 4 threads, like before, but instead of getting worse, they plateaued and stay about the same as I added more threads.

Below are the graphs with the built-in OpenMp reduction.

Here we see similar results to the custom OpenMP version. There is an improvement in the array of size 100,000 as we add threads, again plateauing at 4 threads. However, with this one, we see that the timings don’t really get worse again until we get to 9 threads. This also makes sense, as while my computer has 4 physical cores, there are 8 virtual cores.

One surprising thing is that the graph for the array size 1 million looks almost exactly the same as the one for the custom OMP function, including a slightly longer time for 5 threads, then settling back down until 8 threads. Again, with the larger array size the timings never get slower than with a single thread as we add more and more threads.

Here, the interesting thing is how there is an obvious increase in time as we go from 4 to 5 threads, and again from 8 to 9. I would guess that this was because there are 4 physical cores and 8 virtual cores on my computer so maybe adding one more than each of those shows a performance hit.

Next I have a single chart showing what happens when I increase the array size and threads simultaneously. When doing this, I started with an array of size 100,000, which was were I first started to see some type of improvement with the last experiment. Also, I wanted to make sure that as I increases size linearly along with the threads that I would hit the 1 million array size that worked so well before. I basically increased the size by 100,000 with each thread that I added, so I started with one thread and a size of 100,000, and ended with 16 threads and a size of 1.6 million. The idea is that as more threads are added, each thread is doing the same amount of work, so increasing linearly should show us something.

All three functions increased in time with each increase of threads and array size, but clearly the custom OpenMP version performed the best. It is interesting to note that the C++ version is basically linear in the amount of time taken, both OMP functions seem to hold ground until that fifth thread is added, and then again until the 9th thread is added. See the charts in detail below:

This illustrates that the OMP versions do quite well at not increasing time as the size increased, as long as the physical and then virtual cores were available.

In conclusion, my takeaway from this is that multithreading in this case is only worth it with huge amounts of data. However, performing operations that aren’t just adding the elements of an array together may show more improvement sooner. The OpenMP versions did well, although the custom one ended up performing better than the built-in one. Also, it doesn’t make sense to split into more threads than your machine has processing cores, whether physical or virtual.