20goto10

CSC-410

Matrix Multiplication

# Ease of writing

In this group project, we decided to start from a similar known good algorithm for our solution, as this problem turned out to be extremely similar to the last problem. This means we had an easier time than usual implementing the sequential version. We already had an abstraction written which break up a two-dimensional array into a set of dynamically assignable units of work. This meant we just had to design a function which handled taking a coordinate pair and calculates the dot product of that index in the new solution. The only other problem we had to deal with was that our original solution was recursive, which caused problems with large input sets when the stack ended up popping. It wasn’t difficult to unroll the recursion, but it was one of the difficulties of the implementation path we took. Either way though, even had we just written a more direct for loop solution, the fact is the logic of this implementation is the easiest to understand what is happening. It took longer to write than the eventual threaded versions, but it is also doing far more work in making sure we had a solid algorithm. There were many different solution possibilities, and it would have been really hard to make a wrong move and break the algorithm making this easily the easiest part of the solution to write.

On the other hand, for the threaded portions we had an extremely easy time with the implementation due to the fact we were working from a known good design strategy. The original solution was design specifically to break the problem into a set of finite problems which are solvable on their own with no race conditions. This means we didn’t really have to modify the algorithm at all, we simply needed to implement the threading loop and handle the critical section. The only problem which showed up for us is attempting to return values outside of a critical segment to allow for unlocking. It appears different versions of GCC handle returning function pointers and local variables differently, which means we had to attempt this section of the code a handful of times before we found one that compiled on all of the test systems.

Even so the only real difference between PThreads and Open OP is how many lines of code you have to write. PThreads require you to do the setup of deciding how many thread, statically or with an algorithm, setting up a loop and a barrier to start them, clean them up, and have the main function wait for all of them. On the other hand, you need to add 2 lines, and 2 closures in Open MP to solve the same problem. All in all, for our algorithm, we were already setup for Open MP to be used with no struggle, which meant it was by far the easiest solution. Perhaps if we had chosen a different design approach it would have been more difficult to use Open MP, as it is less flexible, but since we were already following best practices, by breaking our problem apart before attempting to parallelize it, Open MP implemented cleanly with minimal effort.

In summary, the sequential version of the program was fairly straight forward and easy to reason about, while the PThread and Open MP versions were easy to implement but required different amount of boilerplate coding to get working. Open MP required much less code, but was also less flexible on implementations details. However, due to our design we needed no flexibility so Open MP was the cleanest implementation.

# Performance

We tested our three different solutions by varying the size of the square input matrices, along with varying the number of threads used in the pthread solution. To maintain a hands-off, high level approach with the OpenMP solution, we did not change the default thread count.

For very small array sizes, the sequential solution had by far and away the fastest runtime. Once we got up to the size 16 test case, the pthread solution began to overtake the sequential solution in terms of speed, though the 24 thread case didn’t run faster until the next step up in size (32). The OpenMP solution didn’t overtake the sequential solution until we got up to a matrix size of 128, and from that point on had a similar runtime to the pthread solutions. In some cases (sizes 1024 and 2048) it even gave the fastest results. The highest threadcount pthread solution seems to fare the worst, interestingly. At a sufficiently high matrix size, it was consistently outperformed by even the OpenMP solution.

These results suggest that for this problem, very small sizes are best solved sequentially. OpenMP only had a beneficial effect for large sample sizes, but when the overhead became a negligible part of the overall processing time, it performed very well overall. In most cases, the effect of varying the threadcount in the pthread solution was questionable. With many more iterations and test cases it might be possible to identify some trend, but here the differences between the pthread solutions seem to be driven as much by noise and pseudorandom variations as it is by the variation in threadcounts.

# Timing and Speed Tests

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Sequential** | |  | **pthread (6 threads)** | |  | **pthread (12 threads)** | |  | **pthread (24 threads)** | |  | **OMP** | |  |
|  | Size | Time (sec) |  | Size | Time (sec) |  | Size | Time (sec) |  | Size | Time (sec) |  | Size | Time (sec) |  |
|  | 2 | 0.000001 |  | 2 | 0.000104 |  | 2 | 0.000229 |  | 2 | 0.000207 |  | 2 | 0.002627 |  |
|  | 4 | 0.00001 |  | 4 | 0.000171 |  | 4 | 0.000149 |  | 4 | 0.000312 |  | 4 | 0.004988 |  |
|  | 8 | 0.00006 |  | 8 | 0.000148 |  | 8 | 0.000146 |  | 8 | 0.000322 |  | 8 | 0.002591 |  |
|  | 16 | 0.000194 |  | 16 | 0.000173 |  | 16 | 0.000175 |  | 16 | 0.000272 |  | 16 | 0.002585 |  |
|  | 32 | 0.000982 |  | 32 | 0.000421 |  | 32 | 0.000361 |  | 32 | 0.000399 |  | 32 | 0.002984 |  |
|  | 64 | 0.004358 |  | 64 | 0.003566 |  | 64 | 0.003409 |  | 64 | 0.001852 |  | 64 | 0.005779 |  |
|  | 128 | 0.021872 |  | 128 | 0.011035 |  | 128 | 0.015179 |  | 128 | 0.013684 |  | 128 | 0.013675 |  |
|  | 256 | 0.093523 |  | 256 | 0.040141 |  | 256 | 0.029731 |  | 256 | 0.04743 |  | 256 | 0.036287 |  |
|  | 512 | 0.581991 |  | 512 | 0.154523 |  | 512 | 0.144753 |  | 512 | 0.13952 |  | 512 | 0.153836 |  |
|  | 1024 | 5.400176 |  | 1024 | 2.178068 |  | 1024 | 2.092963 |  | 1024 | 2.24522 |  | 1024 | 1.998293 |  |
|  | 2048 | 106.6818 |  | 2048 | 33.376015 |  | 2048 | 32.353804 |  | 2048 | 32.929047 |  | 2048 | 31.98147 |  |
|  | 4096 | 918.78636 |  | 4096 | 298.750814 |  | 4096 | 314.233064 |  | 4096 | 319.352711 |  | 4096 | 312.575827 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |