Assignment 6: testing different loops

11/11/21

Michael Skarlatov

# Experimental setup

For the experiment I had to create a matrix multiplication algorithm in C++ and evaluate the performance of this algorithm with different loop access orders. To do this I used Visual Studio and created a new C++ project and made the algorithm similar to how it was presented in the assignment. I then recreated this algorithm switching the loop order to create the different variations of the algorithm using all possible loop orders. To test these algorithms I used the clock()feature that was given in the assignment hint to count how many clicks it took to perform the multiplication and present the number of seconds the algorithm ran. All of this was run a my PC which has the following specs.   
A picture containing text

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

These specs were found by either looking in my computers settings or checking the processes tab of the task manager window for my computer’s CPU.

# Experimental results

The results of running my applications can be seen in the following console output.

Text

Description automatically generated

The output of C[2] is supposed to represent the result of the matrix multiplication to show that the multiplication is correct but since this test is mostly to figure out the speed of the multiplication I do not bother with resetting the output matrix C after each multiplication. Instead, I just add the result of the multiplication to the previous result. Since I am initiating two matrices called A and B to be a size of 1000\*1000 filled with 1s the resulting multiplication should be 1000. This is why if we look at the output for C in each iteration of the algorithm it increases by 1000. From the results it can be seen that the loop order of jki was the fastest while the loop order ikj was the slowest. Because of this I used the loop order jki in another algorithm in an attempt to speed up computation. This algorithm used parallel programing and multiple cores from my CPU to perform the multiplication. The output for this can be seen below.

Text

Description automatically generated

This parallel programing was done using opm which I had to enable in visual studio for it to be used so it may not work on a different environment with out changes to the environment running the source code. It can be seen though that by running the multiplication in parallel there was a speedup of almost 2.3/0.3 = 7.6 times. This is pretty significant speedup to the multiplication process and is talked about more the explanation of the results.

# The result explanation

The results showed that there were essentially three groups of differing time ranges. There was the group that was both close to being the fastest which was jki and kji, the group that was closest to being the slowest which was ikj and kij, and finally the group that was somewhere in between which was everything else. For the group that was the fastest jki and kji they were many times faster then the slowest loop orders. This is because of the data locality of the values of i. i is a value that easy to keep in cache memory and is able to be accessed by using small jumps that don’t require us to go all the way to main memory. This is known to be the case because by looking at the main line of code that performs the multiplication C[i + j \* n] += A[i + k \* n] \* B[k + j \* n]; i is not used in the multiplication of n and is there for easier get from cache. Because of this when it is the innermost loop, we are performing more small jumps then we are performing large jumps for each loop iteration. In the same way this is also the reason for why the slowest loop orders are ikj and kij. j in this case is a value that is not easy to store in cache and is one that requires more jumps to main memory to access. This can be seen because it has to be multiplied by n twice in the actual matrix multiplication line. Because of this we end up performing more long jumps to memory when it is places within other loops. The rest of the loop orders jik and ijk both have k as the value that it is retrieving for the multiplication and in this case, it is both easy to retrieve and difficult because of how it used in the line for matrix multiplication. There for it is making a short easy jump and another long jump to main memory when ever accessing k. This slows it down enough that it is not the fastest order but still not the slowest. However, to speed up this process even more I can just use more of my CPU to perform the computation. This is done through parallel threads that can be used to perform the multiplication simultaneously. I can do this because my CPU has multiple cores and so by assigning multiple cores to perform different threads for the multiplication, I can substantially increase the multiplication time.

# Conclusion

In conclusion it’s important to take a look a data locality when creating instructions that will be performed many times. Access times for information that will be reused a lot will affect the performance of an application and the more the information is used the more the affect on performance will be felt. This example is a great example of something commonly used, matrix multiplication, in large scale problems with possibly hundreds of thousands more points of information. I also find it important to look at how to appropriately use as much of your machines resources as well since the way I was able to speed up the multiplication even further was by using more of my CPUs resources that it still had available. Knowing what extra cores or other resources your computer has for processing is important and utilizing them can significantly speed up the performance of applications that are used frequently. In this case by using both data locality and parallelism the time taken using matrix multiplication to say train a predictive model could be reduced from hours to possibly minutes.