CIS3090 – A3 Documentation

# Impressions of Parallel Studio and OpenMP

I can’t really say my experience with Parallel Studio changed much between the two assignments. The highlighting and auto-completion continued to be very helpful. Outside of the highlighting and auto-completion, I did use the Parallel Studio tools to look at my programs performance a little deeper. Interestingly the non-optimized version has a CPI rate of ~0.9, but the optimized version has a CPI of 1.4.

My experience with OpenMP was pleasant. To convert my program from a Pthreads program to an OpenMP program was simple. All I had to do in this conversion was wrap my A2 parallel implementation with a number of OMP for loops. The part I was stuck on the longest had to do with how every thread needed to look at every image. But only certain threads should read certain files. Once I realized my mistake was making my threads cyclically distribute the images when they should all touch every image using OpenMP was a piece of cake.

Even though using OpenMP was a piece of cake, I would much rather use Pthreads, for a number of reasons. I find the lack of ability to put barriers in OpenMP for loops very strange. Because I couldn’t a barrier in an OpenMP for loop, I had to create four OpenMP for loops, where one OpenMP for loop with barriers would have done the job in a much more human readable way. I would also rather use Pthreads, because I’m under the impression that Pthreads are much more related to other methods of creating multithreaded programs. That is to say I’m assuming that Pthreads are more related to Java and .Net threads than OpenMP is to Java and .Net threads. I’m also under the impression that if you’ve written a multithreaded cyclic work distributed Pthreads program (where the work is distributed based on the threads ID), it is easy to convert to an OpenMP program.

# Use of Parallel Inspector and Amplifier

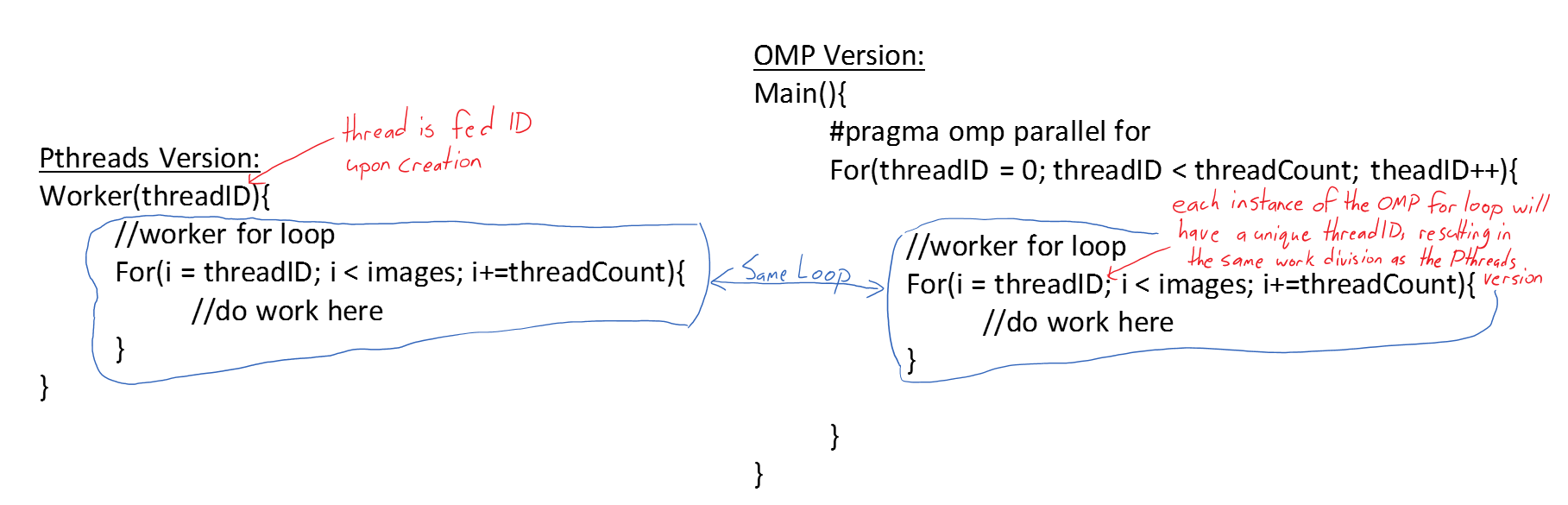
Using Amplifier, I was able to discover my program had a CPI rate of 1.4 using the advanced hotspots analysis. Apparently a CPI of 1 is OK, and a CPI of 0.25 is the best possible for most modern processors. So this this is clearly an area that can lead to significant performance improvements. After running the advanced hotspots analysis a few times, I was able to find that the high CPI rate was being caused by the most executed ifs statement in my program. I suspect that this if statement has such a high CPI because of the fact that the memory these if statements are comparing are entirely pointer based, but I’m not sure. Before the rematch is possible for this assignment, I intend to see if locally caching the entire row the thread is working on in an image has any effect on the CPI rate. I also intend to look into more about CPI in general in order try and improve my programs speed.

Other than pointing out where I could potentially improve my program performance, neither Parallel Inspector nor Amplifier really helped me on this assignment. This assignment was a matter of moving one of my functions into pieces, and wrapping each piece in an OMP for loop. No tools were needed for this assignment, so the tools did not help me in this instance, but nor did they hurt me. These tools will almost certainly be useful when I try to squeeze the last drop of untapped performance out of my program for the rematch, if I make it into the top three.

# Changes made to my programs Architecture and Data Structure

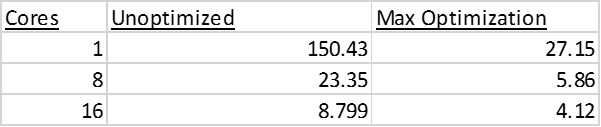
No changes were made to my programs data structure, all images/aldos were stored in their entirety in a 2D struct array, same as before. No attempts to speed up my OMP version were made, my OMP version performs similarly, but defiantly slower than my Pthreads version. I tried to change the programs structure as little as possible, however changes to program structure were unavoidable because of the fact that OMP does not allow barriers inside OMP for loops.

To begin, I entirely deleted the code that managed my serial implementation of the program. Once the serial code was gone, I got rid of the function that created my Pthreads. I then moved the code from the function that called all the worker functions in loops, into my main, and surrounded each of these “worker-loops” with OMP for loops. The OMP for loops were setup to provide the worker loops/worker functions with the ID of the OMP thread. Since this is all that was required with the Pthreads version, I now had a working OMP program. None of functions that did actual work were changed, only the code that managed other code had to be changed.



# The Effect of Optimization

Optimization had a very significant effect on program execution time. The most dramatic run being between the optimized and optimized serial program run, where the optimized run took 150 seconds, and the optimized version took 27.15 seconds. The differences between the other runs is much less extreme due to the fact that the optimized version very quickly begins running into the parallel program scaling wall, and the non-optimized version gets the chance to catch up.



# Program Performance

My OpenMP program was about as scalable as my Pthread program. However my OpenMP is indisputably slower than my Pthreads version. While the OpenMP version does gain impressive speed gains from more cores, the amount of speedup gained from each core very noticeably decreases. Like my Pthread program, my OpenMP programs execution time resembles the linear speedup time, but with an offset of around 2 seconds. So it would appear this 2 seconds of overhead has nothing to do with OpenMP or Pthreads. This overhead must then be related to either the reading in of the files, or the actual image searching algorithm itself, both of which probably play a large role in it, both of which I will look at reducing to improve my programs speed.

# Mean Data



# Graphs

