CIS*3090 A2- Part 2

Impressions of Parallel Studio and Pthreads

My impressions working with Parallel Studio is positive. I feel at home in Visual Studio, since C# is my favourite language, and Visual Basic.net was the first language I studied in depth. I also find I enjoy programming in C much more with an IDE. I found programming with C in an IDE much more enjoyable because of the immediate feedback. I felt much less hesitant to write and change code while using the IDE. Outside of the immediate syntax highlighting, I found being given a list of struct attributes to be very nice as well. Being given a list of struct attributes confirmed that I was accessing the struct correctly, and made me feel much more comfortable using structs, especially since I have a history of trying to avoid structs when I can do without them.

I didn't really use the analysis tools with Parallel Studio to drive my program development. I didn't use Parallel Studio to drive my program development because by the time I was ready to try to polish my programs performance, the Lockhart was already being heavily used.

Which leads to something I disliked about this assignment: the shared computing environment. Since other users were also trying to use 100% of Lockharts processing capability, I had to watch the system usage and try to time my program runs in between other student's runs. Worse than the students periodically using over 90% of the processor for a short time, was when multiple students would run their serial version at the same time. Multiple students running their serial versions at the same time made system usage sit noticeably above 0% for seemingly endless amounts of time.

Both the serial programs seemingly always running, and students periodically using as much processing power as they can made using the Parallel Studio tuning hard. It was hard to tell what changes to my program affected execution time by what amount, when my programs runtime fluctuated by over 50% between identical runs. I suppose this is the price I pay for leaving performance polishing so late.

My impression of Pthreads is also positive. Mainly because one of my goals was to use as little of the library as possible. Meaning I didn't want mutexs, and I didn't want barriers, I just wanted threads that ran unhindered by each other, this is an embarrassingly parallel problem after all. The setup, creation and joining of the Pthreads threads was really straight forward. Once the thread flow was in place, it was easy to move my sequential code onto the threads. Coupled with the straight forward cyclic work distribution I used, Pthreads were a breeze.

Changes made to pseudocode architecture during implementation

One of my goals when I wrote the pseudo code for part 1 was to aim for simplicity. Upon starting to implement the pseudo code however, I realized there were two areas I could simplify my pseudocode design further.

The first area I simplified my design was to read in all the images first in parallel. Originally I had one thread reading in images as the other threads did work. I switched to having all threads read all the files before doing anything else for two reasons. Reason one: I noticed a significant speed up when file

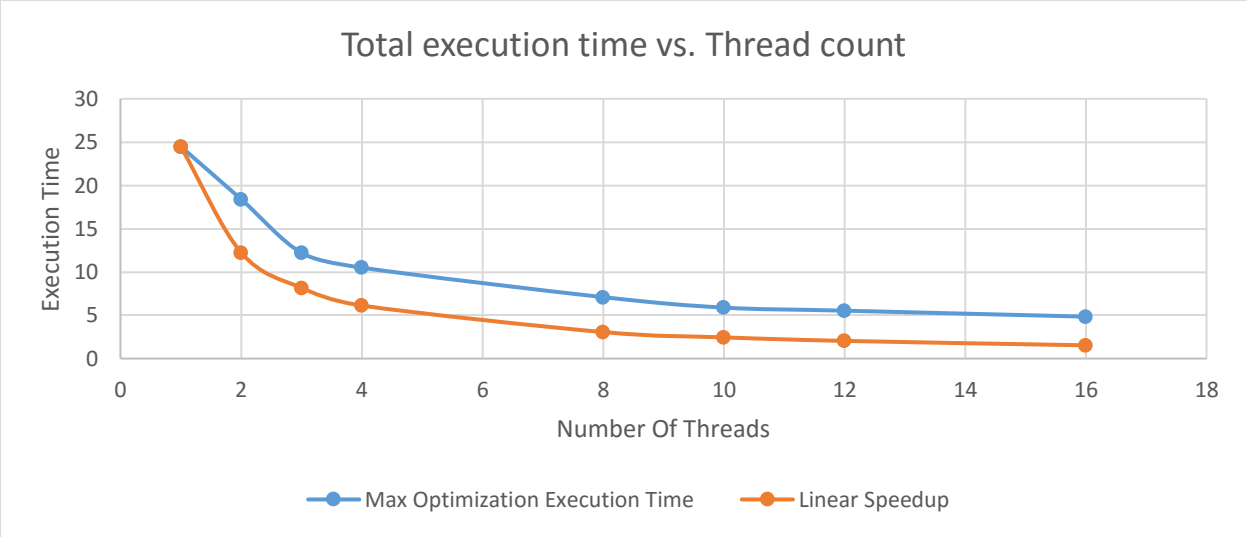
reading in parallel versus serial file reading (nearly 50% 1 thread vs. 16 threads). Two file reading in parallel took ~0.3s, while the program took ~4 seconds with 16 threads to complete. This made the complication of adding mutexs to allow for reading while working not seem worth it. Program time already was much larger than file reading time, and I was going to suffer the time to read in the initial required files to start in either case. Switching to having all file reading done in a stage prior to image processing, also eliminated any chance of false sharing during the image processing stage, since the images can't change once read in and thus cannot be false shared.

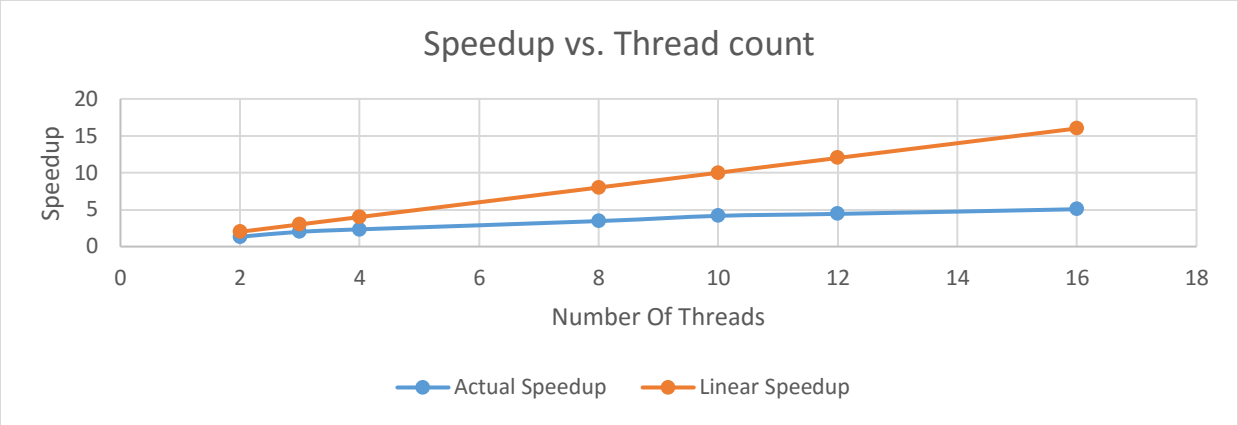
The second area I simplified my pseudo code design, was my work distribution. My original plan of splitting the image into square chunks and allocating the chunks out threads seems to be both heavy weight and offer no advantage over my new work allocation strategy. My new strategy was to split up the images, where each core got every coresth row in each image to process in a cyclic fashion. I thought this to be a good idea since it would result in very balanced work allocation.

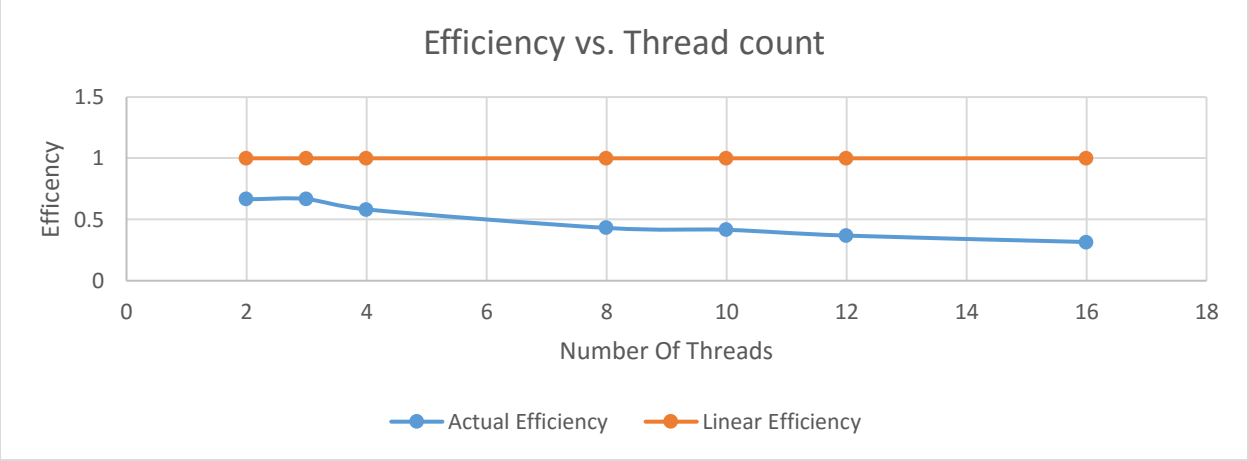
The first few cores would always be responsible for checking one more row than the last few cores (unless things divided perfectly). However early escaping if the aldo had been found in the image, made the first few cores having more rows irrelevant. Early escaping made the first cores having more rows to check irrelevant because there was a very small likely hood the aldo would be located in the last few rows. Both the chance of an aldo being in the last few rows, and the significance of having one extra row to check became much less for larger images, where load balancing mattered the most. With this allocation method, my program on average uses over 13 cores.

My program in the end was moderately scalable. I say my program is moderately scalable because it does clearly benefit significantly from having up to 16 cores to use. However it is clear that as more and more cores are added, the scaling will become worse and worse, due to the overhead mentioned several times above.

Graphs







Graph observations

Looking at the first graph, my programs execution time follows the curve of linear speedup. However my program appears to have a constant overhead amount that shrinks at a rate close to zero or doesn't shrink as the number of cores increase.

Looking at the second graph, my program clearly has speedup occurring as the number of cores increase. However the speedup is much slower compared to the linear speedup. This is probably caused by the same overhead visible in the first graph. The second graph appears to have a rate of speedup that slightly decreases the more cores that are added. This makes sense, as the overhead that doesn't shrink will account for a larger and larger percentage of the programs execution time as the rest of the sections execution time is shrunk. Eventually the speedup should level off, due to the non-shrinking overhead being the only meaningful execution time remaining.

Looking at the third graph, my programs efficiency drops the more cores are added. This is to be expected given the first two graphs. The non-shrinking overhead seems to execute in about the same amount of time regardless of the number of cores thrown at it. So for a portion of my program, more core are being thrown at it and the end result is the same. This is the source of the downward efficiency, the overhead also guarantees that my program will never reach an efficiency of 1.

Optimization and execution time

Optimization made a very significant difference for the serial execution. The optimized serial version executed in about 1/3 of the time of the non-optimized version. For the parallel execution however the gap closes. The optimized 16 thread version executed in about 5/6 of the time of non-optimized version.

Data

Cores	Max Optimization Execution Time
1	24.49
2	18.4086
3	12.2131
4	10.5016
8	7.0955
10	5.8872
12	5.5328
16	4.84

Cores	Unoptimized	Max Optimization
1	66.337991	24.032688
8	8.4575	7.0811
16	5.960486	4.84