Jacob Nehama
Oct 16 2017
Parallel Programming CP341
Professor Dan Ellsworth

Block Project Report

Abstract: One paragraph capturing why the program is interesting and your results

By applying a stencil to every pixel in the image we can modify its appearance via a 3x3 kernel filter. When this stencil is applied through openmp's parallel implementations we can get increased execution time. In fact, this program takes full advantage of openmp, achieving 4x speedup and 100% efficiency on the 4 available cores of our cluster server mcscn.

Introduction: Description of the problem and why it is interesting

The main problem associated with this program is its lack of optimized performance. When a developer applies a stencil to an image in order to modify its appearance, it needs to be applied to every last pixel within the image if we really want to get the full effect of the utilized filter. In the case of my program, this was the main issue that needed to be solved to reach an optimal performance level on the 4 available cores of mcscn. This was the main problem that was holding back the programs performance but along the way I ran into many more, mostly related to improper scheduling and synchronization of openmp parallelization optimizations. It was interesting to see/figure out why some of these openmp optimizations were successful and why some weren't while attempting to use the multiple tools that are offered to the developer by intel with the sole purpose of achieving maximum performance.

Design: Short description of the program design. This section should call out where/how parallelism was used and any challenges in adding that parallelism.

This program reads in a jpg image as its argv[1] argument, processes its coordinates and then puts it into a 1 dimensional output array. A stencil making a call to/using a 3x3 kernel is then applied to every pixel within the image to modify the image in order to utilize the filter. The one dimensional array is then re-copied with its new filter attributes and sent to output where it can be viewed to see its new appearance, while of course keeping track of the execution time via a "stopwatch." In attempt to achieve maximum performance I had many trials and tribulations with openmp, some optimizations worked, others destroyed the performance, and then there were also some that changed nothing. One of the simpler tactics I initially used when increasing performance was the principle of fusing for loops within an openmp parallel loop.
Ex:

```
#pragma omp parallel for
```

```
for (int n = 0; n < rows*cols; ++n){
  int i = n/rows;
  int j = n%rows;

  //for(int i = 0; i < rows; ++i) {
  //for(int j = 0; j < cols; ++j) {
```

These modified, fused for loops using the power of openmp parallelization give us a very slight speedup because the compiler has less instructions to read through while the program is running on multiple parallel threads. After successfully recording this speedup, I attempted to test out what I had just proven by modifying all of the for loops within my program that were applicable to use for loop fusing. It turns out that the gcc compiler already attempts to auto vectorize for loops that are not used in openmp for loop declarations. Any attempt to fuse for loops outside of openmp for loop declarations are a waste of time and have already been taken care of by some of the compilers black magic. Another challenge I experienced while attempting to implement parallel programming within my program was noticed when execution time significantly decreased when applying openmp for loops to regular for loops that only iterated through a small number of items (ie the creation of the kernels). Since these regular for loops were accessing such small amounts of memory/using only a few instructions, the overhead of the openmp for loop caused the execution time to increase. The openmp optimization that resulted in the greatest speedup was used when applying the stencil to the image. Every pixel from each row and column of the image needed to be iterated through in specific order to properly apply the stencil to the entirety of the image. The openmp for loop, due to a lack of proper synchronization, caused the image to be displayed incorrectly once the filter was applied. I looked deeper into openmps scheduling and found out that the runtimes have a default schedule of schedule(static), meaning that openmp statically assigns loop iterations to different threads. Sometimes different threads take different amounts of time within the compiler thus leading us to our problem in this case. The collapse clause (#pragma omp parallel for collapse(2)) turned out to be just what I needed. The iterations of the associated loops to which the clause applies are collapsed into one larger iteration space that is then divided according to the schedule clause. The sequential execution of the iterations in the associated loops determines the order of the iterations in the collapsed iteration space, resulting in an increased speedup and nearly 100% efficiency when applying our stencil.

Performance Analysis: Describe how you measure and compare the performance of your application and include discussion of your results.

Similar to the assignments we did in class, my program measures and compares performance by recording the difference in execution time between the serial and openmp utilizing parallel versions. My program achieves a fantastic speedup and has optimal efficiency in its parallel versions. The serial version, executes in approximately 1600ms(+-20ms) while the parallel version executes consistently at 400ms(+-20ms). Considering that we are running the program on the mcscn server with 4 cores, we have achieved what is the approximate maximum optimized speedup for this version.

(Speedup = t1/tp = 1600ms/400ms = 4x speedup through mcscns 4 cores + the proper workload balance). This also means that we are at approximately 100% efficiency, distributing the workload of the program in the best possible way throughout the cpus (efficiency = t1/P(tp) = 1600/4(400) = 1).

Conclusion: What are your conclusions regarding the performance of your solution and what follow-on work might you want to do with your codebase

After completing both the serial and parallel versions of my block project I can say that I am quite happy with its overall end performance. With the utilization of openmp's tools and a few code syntax modifications, I achieved the optimal efficiency and speedup in my program. If one were to become familiar enough with the internals of the compiler and the high level scheduling tools of openmp, they could achieve close to maximum efficiency on any program they are writing, regardless of code complexity. If I could go back and change anything about my program, it would be to implement a more complex serial version. I tried at first to write something that was too difficult and it backfired on me resulting in a mad start-from-scratch scramble to get something serial running. This in turn, left me with a program that is not as interesting as some of my classmates (although it does get great speedup/efficiency). With that being said, this codebase is going to be pretty dead to me in the means of follow on work, even though I did learn a lot about achieving speedup/max efficiency!