Patrick Alarcon

800792239

**Project Report – ITCS 3145**

At this point we have learned 3 different techniques for parallelization that we can apply for our course project: PThreads, MPI and OpenMP. Unfortunately, it seems that we aren’t able to learn CUDA in time in order to also use this technique for our project. So as a result, I will be talking about the advantages and disadvantages of the different techniques that we have used in the creation of this project.

**PThreads**

Pthread were the first parallelization technique that we learned this semester. Having had previous, albeit shallow, experience in using it from a previous class. In the scope of this program, I would say that this was the middle ground between the different techniques. The Pthread program runs fast and is the most flexible and controlling of all the techniques because you must program all aspects of the parallelization. It is, however, a shared memory program which means you must watch out for race conditions which may cause the code to not work correctly. In addition, taking the serialized code and converting into one appropriate for Pthreads may be time-consuming. In building this code, I realized that Pthreads

-Runs faster than MPI

-Provide more fine-grained control over the parallelization

-Is available to all POSIX compliant systems without the need to install outside packages

However

-Adaptation from serial code can become complex

-Shared memory context means you must be on the look out for race-conditions at any point that the threads may be accessing the same memory location.

-Usage of mutexes, barriers and semaphores may be expensive in overhead.

In terms of the program, using Pthread was the hardest in terms of adapting from a serial code due to potential race conditions, dividing up the array (if the array isn’t evenly divisible by the number of threads, that fact must be taken about into account when dividing up the array between threads as otherwise the last thread may access an unallocated memory location and create a seg fault). However, outside of that, the pthread program was fast running and again, allows for greater control on exactly how the code is parallelized.

**MPI**

MPI was the second parallelization technique we learned, and different from PThread and OpenMP in that MPI is not exclusively used in a shared memory context. This is because MPI can work as a method to do distributed memory programming. MPI can either be the hardest of the 3 to program, of the easiest depending on what you want to do. MPI, since each thread has its own memory, would be less expensive in terms of overhead than the other two. Each of the process also has its own variables so the risk of causing race conditions are considerably lower than with the other two techniques. However, MPI has to go through many changes when adapting from a serial code, they can be harder to debug (as I learned the hard way), and runs a bit slower because all the threads need to communicate between a network, and the placement of method calls can be the difference between a code running smoothly and the threads locking due to the thread not receiving a message. So to reiterate

-MPI can be harder or easier to code than Pthreads

-For example, if an array is easily divisible by the number of threads, then there is no problem using MPI Gather/Scatter etc. However if it is not, you have to do a separate code in order to calculate displacements and strides used for each thread in order to use Gatherv/Scatterv

-MPI is less expensive to use

-Each thread uses its own private variables so the risk of race conditions is greatly diminished

However

-Can be harder to debug

-Performance is slower than Pthread and OpenMP due to having to communicate between the threads

-The change from serial to parallel requires much more changes than with Pthread and especially OpenMP

In the scope of this program, the MPI program runs slower than both the PThread and OpenMP ones. I would imagine this is due to the fact that they have to communicate between the threads in the network. In terms of dividing up the arrays, this usually is easier to divide up since you know the array lengths. However, since this the two arrays containing the pixels use unsigned char arrays, C doesn’t have the capability to get the lengths. As a result, you have to know before hand about the array lengths in order to divide up the array and the broadcast it to all the threads in order to use it when doing threadwork. Speaking of which, the fact that the variables are private to each thread is great for preventing race conditions. However, this belies the fact that if one thread gets information from the user (which is usually the case with the main thread), it must broadcast that variable to all the other threads, otherwise the other threads will not have any data for their respective variable. This caused me a bit of trouble since I presumed that width and height were available to all the threads from the start. This wasn’t the case however and I had to broadcast them before I was able to get the program to work correctly. This also leads to the fact that MPI is harder to debug than the other two techniques, especially if you’re not using a static analysis tool. Overall, I’d have to say that MPI is not my most favorite technique to use.

**OpenMP**

OpenMP is the latest technique we have learned, and it is the easiest of the three to use. This is because OpenMP is higher level, there isn’t much need to change the program from its original serial code, the code is easier to understand and it is easier to debug compared to MPI while running just as fast as a Pthread program. The problem is that like most higher-level paradigms, it doesn’t allow for much fine-grained control. But honestly, if you want to parallelize the program without the need to have much control as to what exactly you have to parallelize, OpenMP is the best of the 3. The only change I needed to do to the original code was add a pragma for loop, a statement to get the thread count and change the method parameters in order to pass on the thread count. Outside of that, OpenMP did almost all the work for me while running as fast as Pthreads