CIS*3090 – Assignment 2, Part 1

High Level Pseudocode

```
Image *images;
Image *aldos;

int imageCount;
int imagesReadIn;

int workers;


char *aldodir, *imagedir;
ReadAldosIn(aldodir, &parallaldos, &parallaldoCount);

forall(threadID in(0..workers){
      localAldos = localize(aldos[])
      localImagesReadIn = localize(imageReadIn);
      localImageCount = localize(imageCount);
      localImages = localize(images[]);
      imagesProcessed = 0

      if(threadID == 0){
            ReadImagesIn(imagedir, &localImages, &localImageCount, &
localImagesReadIn);
      }
      else{
            wait for first image to be read in
            while imagesProcessed < localImageCount
                  wait for next image to be ready
                  localize the image we are on

                  for each aldo
                        figure out how many 'pixel chunks' we can divide
                         -the image into for this aldo / image
                        for each threadID^th 'pixel chunk'
                              check if the 'pixel chunk' contains the aldo
                                    if the 'pixel chunk' contains the aldo
                                          exclusive {printf("found one")}
                  imagesProcessed++
            }
      }
      barrier;
}
```

Pixel Chunk

A pixel chunk is a square sub-image, where the length of each side is the same length as the longest side of the current parallaldo. This is to make dealing with work distribution and aldo rotation easy. For example if the parallaldo was size 3x2, then the 'pixel chunks' for that parallaldo / image combination would be 3x3. There is an obvious case that would break with this setup. An example of this setup breaking is if both the parallaldo and the image are 3x2, as it would try to search a 3x2 image, in a 'single' iteration of 3x3.

Report

The main reason I organized my solution the way I did was for simplicity. As far as I can tell this is a fairly straight forward way to parallelize the find parallaldo problem. Based on my first assignment, I clearly have no idea how to write a fast program, because of this I choose the simplicity route first. Choosing the simple route first should allow me to have a simple base to start and restart from, as I use bench marking to figure out how to write faster programs.

Despite considering this solution fairly simple, I still hope it runs fairly fast. The code is setup to have a thread dedicated to reading in the files. The forall (0..workers) instead of (0..workers-1) is what spawns the extra thread. I'm hoping by having a process dedicated to reading in from a disk and the workers having more work to do, that the disk bottle neck will keep pace better with the other processes working with faster memory. The reason the thread responsible for reading in the files is in the 'if(threadID == 0)' of the 'forall' statement is because I had no idea how to show using Peril-L, that one thread is doing task A while concurrently every other thread is doing task B.

Once all the threads are created, they must wait until thread-0 reads in the first image. Once the first image is read in, the worker threads go to work. Workers keep track of how many images they've processed and compare it to how many images have been read in, and also how many images there are total. This allows the workers to know when there is work to do, and get to work as soon as there is work to do. If the workers largely outpace the reader thread, I'm not sure I'll have many ideas to try to make my program go faster.

The data flow is setup so that only the thread responsible for reading in the file can write to any global variable. This allows worker threads to read concurrently, and since the threads don't need to write back to where they read from, this should be the fastest possible setup. The only synchronization that is needed for any worker thread is localizing the data they use, and making printing to standard out exclusive, to ensure program output does not get garbled.

Global variables are localized by the workers as they are used. I don't really think I need the final barrier, it's mainly there because I feel like I didn't use enough of the Peril-L notation, but it may make sense if my program continued on after. I am heavily hoping I am not missing that could cause thrashing because of false sharing. Since only one thread writes data to global variables, and only writes to those variables once, thrashing shouldn't be possible.