P3: Threaded Programming

Sergio López and Xavier Marquès

February 24, 2021

3 Threaded Mandelbroat

In order to make a threaded version of the mandelbroat, first we need to create an struct to pass all arguments we need: 2 integers for the pixels and a pointer. Next step is to create the thread function where we pass the struct as a void pointer and then we recast it back and store the arguments into new variables to execute the function generate_mandelbrot_region. Now we can proceed in the main by creating an array of structs for the arguments and a 2D array for the id's of the thread. Then we make a double loop, one iteration per region, and in each one we fill the variables of the corresponding struct and create a thread. Finally we must wait all the threads to finish.

The difference between the threaded version and the single version is appreciable. In average, the threaded version it costs around 100ms but in contrast the single version costs around 350ms generating the image, more than three times than the threaded version. So we have noticed that the threaded version uses better the resources of a computer. Due to one of the students who have tested this programs has a Virtual Machine with a linux distribution installed, we have seen that in a virtual machine there's no difference in the time execution. We suppose that it is because a virtual machine has less resources and limited by the computer host.

4 Scrambled Mandelbroat

1. We proceed similarly to the previous part but in this case we're creating N threads and we're passing more arguments, so we need to create another struct with more values. Since we've created N threads and each of them uses 2 regions of the images to perform the interchange, the threads are accessing the same regions at the same time corrupting the performance of the program. And, indeed, we have a synchronization problem.

2. One solution is to use one single lock. This solves the problem of synchronization but gives another problem. When a region is locked by a thread,

then other threads that want to access the same region must loop until the condition of free becomes true and this cost resources and time.

Using a single lock solution, the program costs approximately 48ms to scramble the image.

3. To implement one lock per region, we have to take care about the order in which we lock, other way, the program reaches into a deadlock state. To avoid this, we need to impose the program to perform the lock always in the same order, in our case, we imposed to lock first the lower region, that is, the one with lower indices. So before doing the interchange we check on the indices, if the lower region is the first we execute as always, if the lower region is the second one, we swap the values.

   Using a one lock per region solution, the program costs 24ms to scramble the image. Half times than using a single lock solution.

4. To implement this monitor solution, we proceed the same way as the multi lock, but instead we lock using the functions provided in the utils.c
   By using monitors, now we have this condition variable that allows the threads to release the lock and enter in a sleep mode until it has access to the shared data. Thanks to that, the thread don't need to loop causing busy waiting. Using the monitor solution, the program costs more or less the same than the lock per region solution, in concrete, 26ms, but also is much better than using only one lock.