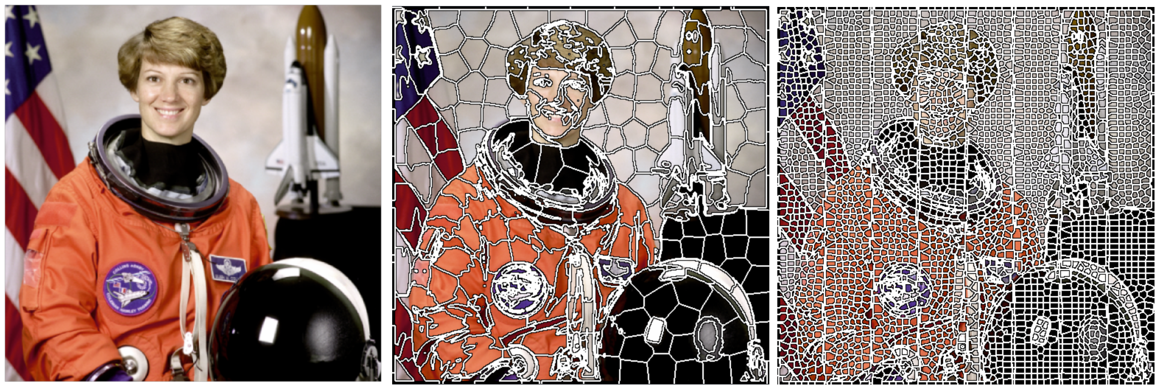
CS 7080 High Performance Computing
Homework 2
Multi-process parallel programming with shared memory

Haoxiang Zhang
PowPrint: hzny2

Due: September 30, 2021

1 Homework Description

This homework has two parts: basic code part and parallel part. For the basic code, we need to implement image segmentation based on Superpixels and clustering algorithm with given SLIC Superpixel Algorithm code. In the Fig.1, it shows how the SLIC Superpixels alogrithm works. Once the code could run successfully, we will make the code parallelized with forks, semaphores, shared memory, locks, and mutexes.



(a) Input image (b) Output image - 256 superpixels (c) Output image - 4096 superpixels

Figure 1: SLIC Superpixels Algorithm Example

2 Homework Details

For homework 2, first part, using SLIC's GetLABXYSeeds(), PerformSuperpixel(), and DrawContoursAround-Segments() functions to implement image segmentation and draw the boundaries. In Fig.1b and Fig.1c, it is easily to see the how good results that the superpixel algorithm did.

In this homework, I used semaphores, shared memory and fork to implement the parallel part. Shared memory is a block of RAM which all of the process can access it. In this assignment, all of the kseeds, distances, and labels will be stored in the shared memories. Semaphores are the tools to manage the processes. Semaphores has a value to tell the processes if they can start to run the code or not. If the value is great than zero, the process can run the code. If the value is zero, which means there is no available to run the code, the process has to wait until other processes post the value. And the fork, this is the function to replicate process which is exactly same as the parent process.

In the Algorithm 1, it shows the details about the whole program. In the PerformSuperpixel() function, the code about calculate distances is using multi-nested-loop, which cost a lot extra time. Therefore I decided to parallelize this part with child processes. However, when finish the distance calculation part, it should goes to recalculate centroid part, then go to next iteration. It is not good to fork processes every iteration. Child processes are doing the same thing, so I forked the processes before iterations. Then, in each iteration, child processes do the distance calculation and parent process do the centroid re-calculation.

Every time, parent process has to be run after all of the child processes finish their job. Thus, I created

Algorithm 1 SLIC SuperPixels with Multiprocess Parallelization

```
Input
    $F$      the input file path
    $P$      the integer level of parallelism
    $K$      the integer count of target superpixels
    $O$      the output filename
Output
    $img$    saved output image file
procedure HOMEWORK2($F, P, K, O$)
    read image
    convert RGB to Lab format
    getLABXYSeeds

    setup and initialize Semaphore
    define semaphore operations
    setup and initialize shared memory

    for all $p = 0 \rightarrow P$ do
        use fork() to replicate process
    end for
    for all $iteration$ do
        if Child process then
            wait() child semaphore, value - 1      ▷ Each child will run one time before parent process
            calculate distances
            post() parent semaphore, value + 1     ▷ Increase parent semaphore value until to P
        else if Parent process then
            wait() parent semaphore, value - P     ▷ Wait until all of the child process finish
            recalculate the centroid
            post() child semaphore, value + P      ▷ Release all the child semaphore value
        end if
    end for
    kill child processes using exit()

    DrawContoursAroundSegments()                   ▷ Draw the contour boundary
    save the image
end procedure
```

two semaphores. One is controlling when the child processes can run, the other semaphore is to control the parent process. Initilize child semaphore value as P, and parent semaphore value as 0. For the semaphore operations, I setup child wait and post operations, and setup parent wait and post operations.

Every time, before run child process, each child process has to wait if there is available. If so, child semaphore value minus 1 and run the job. When they finish the job, post 1 value to parent semaphore value. Once P child processes are finish job, parent semaphore value has increased to P. Then, parent process will start its job, because when parent process is waiting, it needs P semaphore value to run the job. Before parent process done the work, the value of child semaphore still is zero. Once parent process finish the work and post P values to child semaphore value, child processes will start their next work again. All of the child processes will be killed after they finished all of the iterations, like Fig.2 shows.

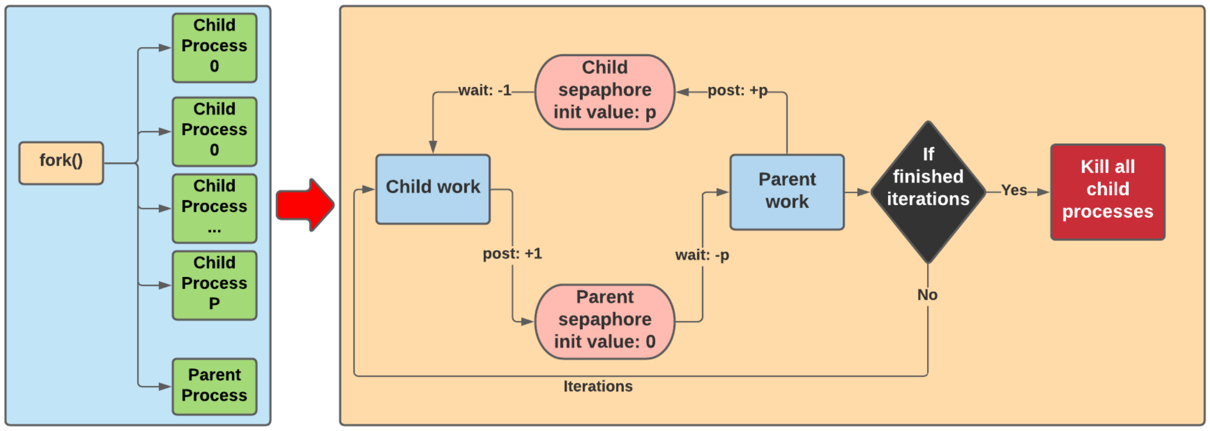


Figure 2: Parallelization Pipeline - Replicate processes before iterations, then terminate processes after iterations (Fork & Join Model)

3 Problems & Solutions

I added the wait(NULL) after killed all of child processes. This cause the problem that the waitpid() complains the error that **"Error waiting for child process"**. This is because, after killed child processes, calling wait() cannot detect any child process. All of the child process are gone. Therefore, just make sure there is no wait() function after all child processes killed.

For each iteration, the distance array values should be reset each iteration. I added the reset code at the beginning. This causes the problem that every time child processes in a new iteration, the distance array in shared memory will be reset. The distances values should be reset only once every iteration. Therefore the solution is that move the reset code to the end of the parent work and before parent process does post action.

4 Conclusion

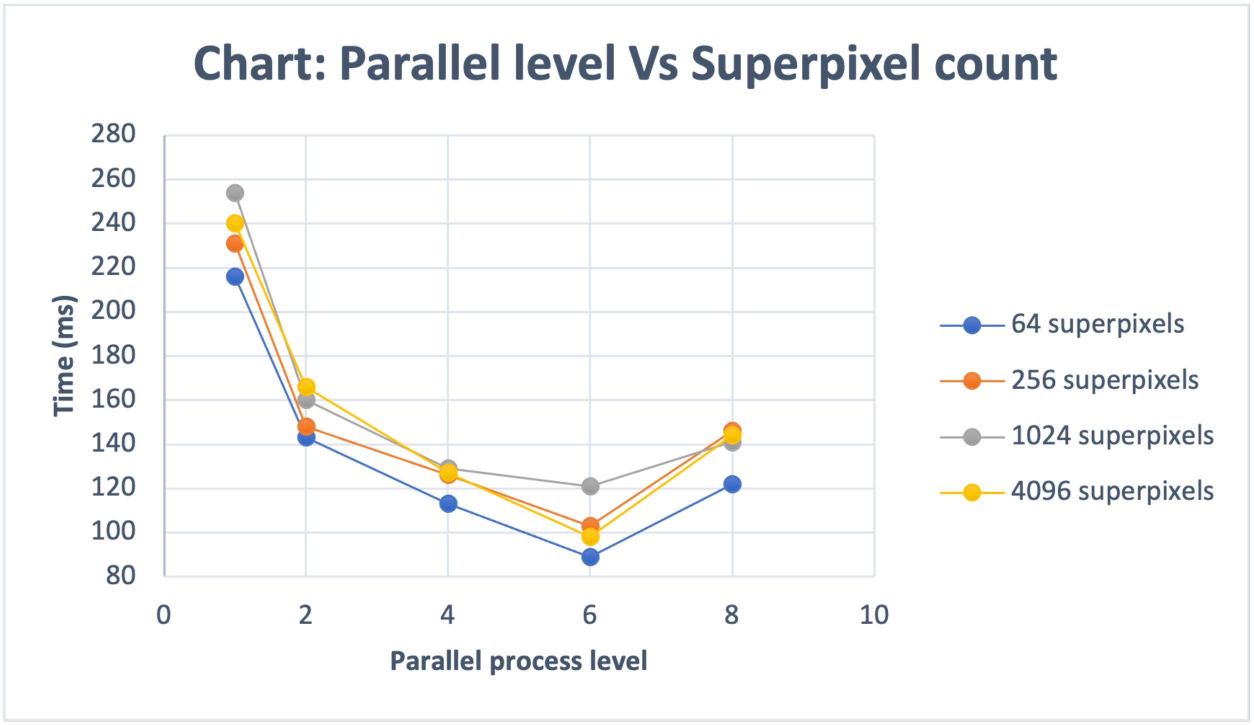


Figure 3: Caption

Fig.3 shows the comparison with different number of superpixel and different parallel level. We can easily find that with more paralleled processes the speed is faster, because the distance and labeling part is paralleled which saves a lot of time. The reason that the time cost more when have 8 processes to parallel, I think it may be maximum number of cores is 6. For the extra 2 processes, they may run the job separately. Besides, more superpixels causes more time to finish.

5 Lesson Learned

After this assignment, I am clear about how to setup semaphores and shared memories especially how to design a good way to control the processes with semaphore operations. Based on the results with different parallel level, it clearly shows the effect that speed up the processing time. Furthermore, I learned how to use **errno** to print out the detailed error. Errno could show the exactly what error is caused.

When do multi-processing, processes will not access to a variable. Therefore, we need to setup shared memories for each variables that processes need. Each shared memory need a pointer, a ID, a key, and a Flag. Using shmget() to register a space for the shared variable with given size which is same as the size of the variable type. Then, assign the space to the pointer. After that, each process only need the pointer then they can access the variables.

To manage each process, we can use semaphore to control the work flow processes. To setup semaphore, we also need Id, key, and flag. We also need to tell the semaphore how many work can pass to semget(). Every time a semaphore pass semget(), the semaphore count will minus one and the process need to wait if the semaphore count is zero until some other process release the semaphore. To control how many semaphores will post or wait, we can setup a **sumbuf** (semaphore operations). Then the semaphore knows how many semaphore need to release one a work is down or how many semaphore a work need to start.

For setting up the parallelization, I have learned how to set two different semaphores to manage two different work together. For example, I did parallel on distance computation and labeling, but re-calculate centroids part need to wait all the parallel part done. Therefore, I setup that parallel part need p semaphores if there is p processes parallelized. Each process (child process) will take one semaphore to start work. One they have done the work, release one semaphore to the one control re-calculate centroid part (parent process). When parent process wait until it has p semaphores, it will start work. Before parent process finish, child process semaphore still has zero until parent process is done and release p semaphores to let child processes do the work again.

I also learned how to end child process separately from parent process. When create child processes, I give them an identified ID. Once the parallel part is done, end the child processes with the ID.