Day 1

** Concepts relevant from Assignment 0 and 1 **

1. Assignment 0:
    1. Memory allocation (non cotinuous in our code), and correct freeing: In assigment 0 we learned the advantages and dissadvantages of each type of memory allocation: stack allocation is faster but one has to be certain that an array fits on the stack, and heap is used in the oposite case. Heap allocation can be structured to be “flexible” which is generally slower than allocating the whole array at the same time, but it allows to allocate and free memory in a more taylored way for our program. This flexible approach to allocate memory can allow us to “create” and free a specific row to write the resykts of the Newton method in a memory efficient way. We will test between continuos and non continuos memory allocation for the arrays that will store the root ids, and number of iterations, and decide which, if any, improve our benchmark.

    2. fopen, fwrite (faster than fprintf). In assignment 0 we learned the difference between writing characters, or writing bits directly to a file, and understood that for speed purposes the correct choice is to write bits directly using fwrite. This approach is going to be used from the begining since we are certain that fwrite command is much faster than fprinf and satisfies our needs.

    3. Parsing command line arguments In assignment 0 we learned how to write a program that accepts or needs arguments to function, we learned an understood the syntaxis to created an efficient way of parsing arguments in a way that it recognizes every pattern, i.e. -arg1, arg1… We have found that using the function opt simplifies this task.

    4. Make files Make files are an amazing tool to save time in the compilation proces of a “big” project, because it allows to perform changes and compile them avoiding small syntax mistakes, like forgeting to link a library, or use the correct optimization flag.

2. Assignment 1: 1.Correct benchmarking In assigment 1 we learned that the correct way to perform benchamarking was to run the specific task many times and then get the mean in order to have a realistic result not subject to small volatilities. In this assigment benchmarking is a must, and so the proccess has to be optimal. Among other things, we use the benchmarking techniques to compare our power function with cpow, in order to be sure that our implementation was faster, and should be used in our code.

    2.Inline code to make it faster, small functions. In assigment 1 we learned that inlining a function, specially if it is small, can improve the performance of a program, if the function its called iteratively many times. d.

    3.Use static inline to explicitly suggest the compiler to inline a function. We use static inline to “suggest” to the compiler to inline certain functions that we consider would be better placer in the main code at compilation. As an example we wrote our own power function, optimizing it to be faster than cpow and we use static inline for it, because it is called many times when solving

Newton´s method.

4.Locality, computing initial points moving throug the rows. We learned in assignment 1 that accessing memory its faster if we acces addresses that are close or adyacent, and not jump from one place to another in memory, in our program we implemented this concept by letting each thread compute the roots for a whole row before jumping to the next.

6.Use of valgrind and GDB to analize code and check for improvemen options Finally we learned in assigment 1 that this tools can be good allies in order to understand the performance of our code. with Valgrind we check for memory leaks or incorrect allocation and freeing procedures, and with GDB we can go to the origin or a bug, or a line where the code crashes or does something unexpected, and check what is happening in there, and the actual value of the variables at that point.

Day 2

** Program Layout & List of Subtasks **

1. Initialization Setup: Argument Parser, setup global variables
2. Implement synchronization and threading for : compute thread and write thread
3. compute_main Function: Check x_k for convergence
4. compute_next Function: Compute x_k+1
5. Setup color mapping and write to file.

Day 3-4

** Implementation & Revision of Subtasks **

1. Argument Parsing & Global variables
    1. Check if all needed values are available.
    2. Value of ‘l’ should not exceed 1000.
    3. Value of ‘d’ should not exceed 9 and must be greater than 0.
    4. Study “Assignment 0:Parsing command line arguments” for motivation.
    5. Study global variables and setup mutex.

2. compute_next function
    1. Objective: Find x_k+1 in the most effective way.
    2. Implementation 1: x_k+1 = x_k - (cpow(x_k, d)-1)/(d*cpow(x_k, d-1))
    3. Remark for Implementation 1: VERY SLOW
    4. Implementation 2:
        1. compute_next is implemented as inline function
        2. Compute theta = atan(imaginary(x_k), real(x_k))
        3. Compute r = real(x_k)*real(x_k) + imaginary(x_k)*imaginary(x_k)

4. Using De Moivre's theorem we get z^n = r^n * cis(n*thetha) Compute z = (pow(r,d) * (cos(d*thetha) + i*sin(d*thetha)) / (d* pow(r,d-1) * (cos((d-1)*thetha) + i*sin((d-1)*thetha))
5. x_k+1 = x_k - z

3. compute_main function
    1. Objective: Check if the given point (x_k) is converging to a root or no.
    2. Store roots for the given function with all possible values of d=1,2…9 in a 9x9 matrix.
    3. Norm of a complex number is defined as sqrt(R^2 + I^2) = epsilon, where R and I are real and imaginary coefficietns of the complex number.
    4. Instead of using csqrt() and cabs(), defined in complex.h we implement norm in a different way. This is done by squaring the equation in above given equation. Therefore, R^2 + I^2 = epsilon^2. Which helps us avoid time consuming functions like csqrt() and cabs().
    5. Setup initial matrix A having r rows and c columns where r = c.
    6. Run a Loop_1 from k = A[0][0]…A[r][c]
        1. Set a variable converged = False, to keep track of whether x_k has converged.
        2. If k = 1 => x_k = x_0
        3. Criteria 1: Check if Absolute value of real(x_k) and imaginary(x_k) both must be <= 10000000000.
        4. Criteria 2: Check if norm of x_k must be <= (10^-3)^2.
        5. Criteria 3: Run a Loop_2 from i = 1…d difference = x1 - root[d-1][i] Check if the norm of difference <= (10^-3)^2
        6. If any of the above criteria is true at any point we set converged = True => break Loop_2 iteration and pass the value to write thread; If it is False we compute x_k+1 = compute_next(x_k).

Day 5

** Functional Program **

1. We achieved a function program which prints a PPM file same as the sample newton file.
2. Currently we are working on methods to reduce the execution time of our program in half.