Assignment 2
Implementation Plan
TMA881 – High Performance Computing

October 7, 2019

1 Summary of concepts

1.1 Threads

From the lectures, we know that the use of POSIX threads is a good way to achieve the parallelization needed to speed up our computations.Threads share access to global variables that are declared in the program header. They also share files and heap memory. We will use this in our code by letting our computation threads read and update gobal matrices for initial values, attractors and convergences. Threads do not share thread local variables created in the stack memory. Threads can therefore create local variables with the same name.

`pthread_create` creates a new thread and makes it executable. As arguments with this command, we also tell the thread which function to execute. This function must take in a `void*` argument and return a `void*`. The actual type of the argument sent into the function needs to be recovered inside the function, thus the programmer need to make sure that the correct datatype is sent in.

When a thread has been created, it is up to the operation system to schedule the execution of that thread. As programmers, we don't affect the order in which our threads are executed. Access to variables and files that are shared between threads therefore need to synchronized. We will be using mutexes for synchronization. A mutex, which stands for *mutually exclusive*, is a global variable and thus shared between all threads. If several threads can access and modify the value of a global variable, this must not happen at the same time. We will obtain the mutex lock by calling `mutex_lock` for a specific mutex before accessing the global variable. When we are done modifying it, we unlock the mutex with `mutex_unlock` and the value can be accessed by other threads.

1.2 Memory Access

From Assignment 1 we have learned that the way we access memory will highly affect affect the performance of our program. Indirect addressing and locality of reference are both connected to how we access memory and will affect the runtime of our program.

Indirect addressing is when memory is accessed by using more than one step. For example using a pointer that points to the address of another pointer which in its turn points to the address of the intended memory location is a typical way of using indirect addressing. This is something we want to avoid to ensure good performance for our program.

When optimizing a program it is important to utilize the locality of reference to avoid unnecessary loss of performance due to excessive memory search (Computer Organization | Locality and Cache friendly code, geeksforgeeks). Locality is commonly divided into two parts, temporal and spatial locality. With temporal

locality we assume that the accessed memory is likely to be used again and it is therefore stored in the cache. If we would then try to access the same memory again it will be easier to use since the cache memory is faster than the main memory. The spatial locality refers to that we are likely to try to access the nearby data of our currently used memory location. We could see this in Assignment 1 when we tried to calculate the row and column sums of a matrix. The row sums were a lot faster to calculate due to the fact that we used one pointer to a memory location for each individual row summation. As for the column sums we accessed only one element per pointer and therefore could not utilize the spatial locality.

When building our program we will have to be aware of how we access data objects and how we continue to use them in order to assure that we utilize locality. We will be able to use this when searching for convergence in our program by accessing each element of our output picture in a way that minimizes the amounts of times we have to refer to the main memory.

1.3 Writing to PPM files

The program will write the information about which root that is the attractor and how many iterations each point needed for convergence to a PPM file, which can then be interpreted as a picture. We are going to use a P3 PPM file, which means that it uses RGB-colouring, and is written using ASCII-symbols. A PPM file consists of a header which specifies what type it is (P3 in our case), how many rows and column it is (where one element will correspond to one pixel in the picture) and a maximum value for each colour, which corresponds to the intensity of the colour in the picture. More concretely, the start of our textfile will look lite this:

P3
L L
M

Where L will be the amount of lines, as specified when calling the program. We will decide later what M will be, and it will probably be different for the attractorfile and the convergencefile. For the attractorfile, we will need at most 9 different colors, one for each root, so we can't choose M=1 since then we only have $2^3 = 8$ different colors, one for each combination of RGB, but it should work good with M=2. For the convergencefile any big number should do good, maybe 1000. After the header, the colorvalues should come. Each pixel will be represented by three number, each separated by a space, each pixel should also be separated by a space, and each line by a newline-symbol. To get a gray color, you give each color the same value. The writing could, and will probably be done using `fwrite`.

1.4 Parsing command line arguments

To parse the arguments giving the number of lines and columns for the output picture, the degree of the polynomial and the number of computing threads our plan is to simply look at the arguments passed to our main function and inspect whether the first argument is the rows or the number of threads, and then initzialize global variables corresponding to the number of rows/columns, threads and degree. This will be done in the same way as in assignment 0. We can assume that our program will be run by the command `./newton -tT -lL d` or `./newton -lL -tT d`, where `T` will be the number of computation threads and `L` will be size of our our matrices. Since the `l` and `t` argument can be passed in an arbitrary order, our main function first compares the letters in the first two arguments and then initializes `T` and `L` to the values provided by the user. The last argument will be the degree of our polynomial.

In case the user provides a wrong number of arguments, inserts a degree above 9 or a number of lines equal to or greater than 100000, our program will send back an error message with instructions on how to call the functions, and then exit.

2 Intended program layout

The program can be, quite naturally, divided into two subtasks. The computation of the Newton iteration for each of the points, and the writing of the attractors and number of iterations for each point two the output PPM files.

For the computational part, the program will be split up into a few functions. One, called `check`, which will take a complex double as an input, and check if it has met any of the stopping criteria, i.e if the real or imaginary part is bigger than 10^{10}, for which it will return 0, if it is less than 10^{-3}, for which it will also return 0, and if it has converged to a root, for which it will return a natural number corresponding to that root. For this we will also need to calculate the actual roots, which we will do, and save as a global variable. We will create another function called `newton_iteration` which will take a complex double as input and update the complex number using the newton method, and give it as output. We will, as suggested in the assignment description, hardcode this iteration step for each different degree of the polynomial, for faster computations. Furthermore we will have a function called `compute_rows` which will serve as the mainfunction for the computational threads. which iterates through one row, of the matrix, and generates one vector of computed attractors and one with the corresponding iterations, which it finds using `check` and `newton_iteration`. These are then stored into global matrises which will be the output when done.

The writing to PPM files will be happening in a function called `write_to_file`. This function will create the appropriate filenames and headers for our two output files. Inside this function, we will also prepare a string from each row in our result matrices. Each of these strings will correspond to one row in the output PPM file, and contain the RGB codes corresponding to the roots and the number of iterations needed to converge (or diverge). The strings will be prepared dynamically while the other threads are running. We will keep track of which rows are computed by an array `row_done`, initially filled with zeros. When a thread is done computing one row it changes the corresponding element in `row_done` to 1. In order to synchronize the access to `row_done`, we will use a mutex `row_done_mutex`. Our plan is that `write_to_file` also will write the prepared strings to the output files. After all strings are prepared, we will execute `fwrite` to write them in the right order to the file.

3 References

Computer Organization | Locality and Cache friendly code. In geeksforgeeks. Gathered 2019-10-04 from https://www.geeksforgeeks.org/computer-organization-locality-and-cache-friendly-code/