Assignment 2 - High Performance Computing

Relevant concepts

- **Stack allocation**. We have learned in Assignment 0 that there are two types of allocated memory: Stack and heap allocated memory. One acquires stack allocated memory by using array declarations inside of functions, as opposed to global declarations of arrays.

  Memory allocated on the stack is limited in size, but tends to be faster. Moreover, stack allocated memory is thread local and therefore provides an opportunity to untangle the mutual impact of parallel threads on one another. Consequentially, it is an important consideration to employ stack allocated memory in the innermost iteration steps, i.e., the Newton iteration for individual point on the complex plane.

  We plan to test this concept by comparing runtimes of variants of our program using stack and heap allocated memory in the innermost iteration steps.

- **Heap allocation**. The concept of heap allocation that was introduced in Assignment 0 should also be used because it is safer (less likely to run out of memory than stack) and better pratice to allocate large arrays in heap since stack has a much limited size than heap. However heap allocation tends to be slower because of the way it is more complicated than stack allocation. It is possible to make a heap allocation using the function malloc and calloc.

- **Reducing memory fragmentation**. Memory allocation of all matrices are done with non-fragmanted memory allocation. When memory is allocated in a contiguous way, it results in less cache misses and it becomes easier for the CPU to predict the next memory to fetch compared to the scenario where the memory is allocated in a fragmanted way. This is because different fragments of memory are located in random places of the RAM.

- **Writing to file**. The fact that the output of the program should be written to a .ppm file requires one to use the knowledge aquired in the Write to files task in Assignment 0. The learning outcome that the fprintf is slower than fwrite function is also beneficial. The write thread writes to file row by row where is each contains the attractors and convergences in RGB encoded manner.

- **Parsing command line arguments**. We have learned in Assignment 0 to parse the command line arguments that have a flag or not. Since it is the requirement of the assignment to receive number of threads, number of pixels in an edge of the output file and the exponent of the polynomial as input arguments, it is a must to use the concept that we have learned about parsing command line arguments. The function getopt is suitable to implement this concepts as

it is a very versatile way to parse command line arguments even if the order of the arguments is unpredictable.

- **Valgrind**. Valgrind is a useful tool to catch if there are any memory leaks in the program which helps one to trach down any misimplementation of memory management. Since this assignment requires heap allocation, valgrind is pretty useful to track down problems. One can call it using the command “valgrind ./newton -t3 -l1000 5”.

- **Locality**. Locality task in assignment 1 is also useful since this assignment requires one to iterate over large matrices as fast as possible. Caring about locality significantly reduces the caches misses which improves the performance. Iterations over initial conditions, attractors and convergences matrices are done in a way that the outer loop iterates over rows while the inner loop iterates over the columns which is the most efficient way to decrease the number of cache misses.

- **Inlining**. As it was also shown in assignment 1, defining small functions as inline increases the performance and readability. The inline functions are still implemented as regular functions but since they are inlined, the compiler copies the function body to where it is called so that the overhead of calling the function is avoided. This makes a significant performance improvement especially if there are frequently called small functions. To make a function inline, the keywords “static inline” are added before the return type of the function. It should be noted that static inlining may increase the compile time as it is an extra task for the compiler.

- **Avoiding dirty caches (Scott Meyer’s code::dive presentation)**. As Scott Meyer has pointed out, if a thread will be working on a global chunk of memory, it is important that the thread first take a local copy of the relevant chunk, manipulate it and finally update the global memory by copying the local chunk to where it is belong to in the global memory. Otherwise, when a thread manipulates even a single bit in a cache line, that whole cache line becomes entirely dirty and needs to be updated for all other threads. This brings additional load and avoiding it helps achieving a linear scaling between number of threads and performance speed-up.

Intended program layout

The program starts with initialization of global variables, parsing commandline inputs and precomputation of the roots of the proposed polynomial. For memory allocations, stack allocation is used for small-sized variables and heap allocation is used for large variables such as 2D matrices that contain initial conditions (IC), attractors and convergences of each pixel.

It should be noted that since an initial condition has both a real and an imaginary part, it is represent as a double complex value using the built-in complex header.

At this point, per instructions the program naturally splits into two subtasks: The computation of the Newton iteration and the writing of results to the two output files. Each of them will be implemented in a separate function, that are intended to be run as the main function of corresponding POSIX threads.

Subtask 1: Computation of Newton Iteration

Each thread owns a block of rows of the initial conditions matrix and processes them using the compute_main function which implements the logic behind computation of newton iteration.

The computation of the Newton iteration can be further split up into two subtasks. First one is to iterate over the rows that are owned by the thread and to calculate the index of the root that a particular IC converges for each IC inside a row. The second subtask is to make the locall calculated attractors and convergences global.

- ## Subtask 1.1: Finding Root

This logic is implemented with 2 nested loops where the outer loop iterates over the rows and the inner loop iterates over the pixels of those rows and calls the find_root function for each pixel.

- ## find_root function

The find_root function calls the newton_update_func to update a given IC value until the newly computed value’s squared norm becomes less than epsilon squared (too close to origin) or greater than 1.0e20 (too big) or the difference between itself and the value calculated in the previous iteration is lower than epsilon squared (converged). If the latter occurs, the squared norm between the newly calculated value and each of the precalculated roots is calculated one by one until one of the roots makes the squared norm less than epsilon squared.

- ## Subtask 1.2: Writing Locally Calculated Data To Global

The second subtask of the computation of the Newton iteration is to make the locally calculated root indices and convergences global so that the write thread can access them.

Subtask 2: Writing to File

As for the writing to file, we have identified 2 independent subtasks. First one is to check if the item (Row of attractors matrix) is ready to be written to the file. If not, the writing thread goes to a 20 milliseconds sleep. After waking up it checks if the item is ready again. If it is ready, the program continues with subtask 2.2 which is explained below in detail.

- ## Subtask 2.2: Writing RGB values to file

However, if the item is ready, the thread takes a local copy of the item_done array and iterates over the local copy. Along the iteration, for each item that is 1, the thread prepares two different char buffers to be written to the file. First buffer represents the corresponding RGB color of each pixel's root index while the second buffer represents the row of the convergence data encoded as RGB values. After filling the two buffers, they are written to their corresponding files. This is the part where the "writing to file" task from assignment 0 is implemented. Finally, write thread goes to checking if the next item is ready to be written to file.