Relevant concepts

False sharing: In the example of threads in the lectures notes, Raum uses a list “done” which is initialized to all zeros. Once an item is done, its corresponding element in the list “done” changes to 1. However, this results in false sharing, because different threads will write to adjacent elements. This means that if one thread updates an element in “done”, another thread that will update an adjacent element will have to “reload” its cache memory, which takes time. Instead we add a last column in the converge / attractor matrices that we change from say 0 to 1 when a row is done.

Parallelism: In order to speed up the program we use parallelism, which means that threads are created that perform different tasks. We used one thread that writes to the files and n threads that perform the calculations, where n is set by the user. Although parallelism can significantly speed up the program, it also introduces a set of new problems that have to be dealt with. For instance, when different threads use the same variables. To keep track of the current row in the compute threads, we used an offset as input when creating these threads. Another solution might be to have a global variable that is incremented. With this solution a mutex lock would have to be used when reading this variable so that another thread cannot increment it before it is used.

Explicitly inlining: Instead using a power-function when implementing the newton algorithm, it is better to write out all the multiplications, i.e. x^3 -> x*x*x.

Fwrite instead of fprintf (pre-format strings): There are two common functions for printing to a file: fwrite and fprintf. In order to increase the performance of a program, fwrite should be used (it is much faster than fprintf). When using fwrite, one has to pre-format the output that is passed to the function.

Synchronization between threads to avoid data accesses when accessing shared variables: One important thing when using threads is that each thread should work on its own items, i.e. no thread should work on the same item as another thread. The writer thread waits until the first item is done, independently of the other items. When that item is done, it writes that item and then writes the finished consecutive items until it reaches an unfinished item, say item 5. Now, it waits until item 5 is done. This process goes on until all items are done.

Data types (as small as possible): In order to fit as much data in the cache as possible we use small data types.

Using nano to avoid reduce busy wait: In the write thread, when waiting for an item to finish, nano can be used in order to reduce busy wait, i.e. one shouldn’t continuously check for finished items (polling). However, it should be noted that this is not an optional solution, but it reduced busy wait.

Intended program layout

Parsing command-line arguments: In this part we used out solution from assignment 0, meaning that we used strcpy to remove the “-” and then atoi to convert to integer.

Synchronization of compute and write threads: Every thread is given an offset. For example, the first thread (thread 0) starts with offset zero and calculates row 0, n_threads, 2*n_thread, …. The second thread (thread 1) starts with offset one and calculates row 1, n_threads+1, 2*n_threads+1, …. This implies that no threads will ever work on the same row.

Data transfer between compute and write thread: There are two main global variables in the program to hold the values to be written to file; attractors and convergences. These variables hold lists of strings and are initialized on the heap before starting any threads. The last value on each row is set to “N”. When the compute threads start they write to these variables and replace the last character of each row from “N” to “Y” when it is done. The write thread, on the other hand, reads the attractors and convergences variables. It knows that a row is finished when the last character is “Y”, in which case it writes the entire row to the respective file.

Checking the convergence and divergence conditions: Using the reverse triangle inequality, one sees that if the norm of the iterate is too large or too small it cannot be close to any of the roots, and one does not need to check it has reached any of the roots (which saves some compute time). However if the norm is around 1, we check if any of the distances between the iterate and the roots are less than 10^-3. But to avoid taking the square root we instead check if the squared distances are less than 10^-6.

Computation of x_n in the iteration step: We simplified the iteration step to z_{n+1} = 1/d + z_n(d-1) + (z*)^(d-1), where d is the degree and z* is the conjugate of z. Instead of using the pow function we hardcoded powers. So we wrote z*z*z*z instead of pow(z, d).

Writing to file: To increase speed, we use fwrite instead of fprintf. In order to get fwrite to work effectively we simply hard-code formatted color strings.