Assignment 2: Threads

Relevant concepts

Cache locality, memory fragmentation

The CPU has different memory entities: registers, cache (L1, L2, ...), (main) memory (RAM) and disk; of these, registers are the fastest, then cache, then memory and finally disk. The CPU registers are so small that they only hold data for a few clock cycles while the CPU does calculations on them. Thus, the CPU constantly swaps data with its different memory entities. However disk is multiple times slower than memory, which in turn is multiple times slower than cache. Thus, it would be a catastrophy if the program has to swap in data from disk, and a smaller catastrophy if it has to swap in data from memory.

When data is swapped into cache from memory (and to memory from disk) it is done in lines, i.e. the principle of spatial locality is assumed; if one pice of data is needed by a program, adjacent data is probably also needed. The CPU also assumes the principle of temporal locality; if on piece of data is used once it will probably be used again. However, thinking about temporal locality will probably not be that relavant when implementing this program.

When the CPU swaps in the requested data, some fixed size of adjecent data is also included. This makes it possible to avoid much data swapping with memory by programming in a way that makes sure to allocate data in a adjacent manner and to do calculations on all adjacent data that the memory swapped in before doing another swap or moving on.

This will be considered and utilized in the program when allocating arrays and when fine-tuning how many calculations each loop and thread will perform before moving on.

Inlining

As we learned in assignment 1, using inlining can make function calls faster for small functions that are called a large amount of times. Compilers do this automatically unless it is disabled, and as a programmer you can give the compiler hints for which functions to inline by using the `static inline` keywords in C.

When implementing the program, we will focus on decoupling the code and breaking out functionality into small function according to the single responsibility principle. Since the compiler automatically uses inlining to optimize the code, we will probably not have to think so much about which functions to inline. However, if we find a small function that is called a large amount of times, we will experiment wih using the `static inline` keywords. One such function could be the calculation of the next x-value in each iteration of Newton's method.

Thread communication (mutexes etc.)

As we learned in the lecture on parallel computation, the runtimes of parallel programs are heavily affected by communication between threads. When using shared memory, mutexes need to be used to coordinate memory access in order to make sure that concurrent reads and writes don't corrupt the data. Since a mutex can only be held by a single thread at any time, acquiring a mutex can cause all other threads to be locked. This means that unnecessary usage of mutexes by a single thread can cause an entire program to slow down, which is why great care needs to be had when working with mutexes.

In C, mutexes are used in the following way:

```
pthread_mutex_tsome_mutex some_mutex;
void some_function() {
  pthread_mutex_lock(&some_mutex);
  // Access shared variable.
  pthread_mutex_unlock(&some_mutex);
}
```

In the program that we're writing for this assignment, we will need to use a matrix stored in shared memory to communicate results between the worker threads and the write thread. While the program can be written so that all worker threads write to disjunct sets of indices, the write thread needs to read from all indices, and since reading and writing the same index concurrently is undefined behaviour, we will need to use a mutex to coordinate these accesses.

There are a lot of things to think about when designing a program that uses mutexes. We need to make sure that the program doesn't end up in a state of deadlock, where no thread is able to continue with its execution, or that a single thread starves other threads by rapidly unlocking and re-locking a mutex. In general, we also want to minimize the amount of time where the mutexes are locked, which can be achieved by for example copying the result of the shared variable to a local variable, and then doing computations using the local variable instead of the shared one.

We will use worker threads for performing calculations and a write thread for writing the results to the PPM files. This requires communication between the threads over which results are ready to be written. A *ready* array will be implemented in which the worker thread set a value to 1 if the same index in a `results` array is ready to be written. The writer thread then fetches the result from the `results` array and writes it to file.

Using a *ready* matrix will allow for the worker to keep working with the `results` array without having to worry about undefined behavior, as the writer will only read from the indices that are marked in the *ready* array as ready. This will allow for less mutex usage. In our program, it might make more sense to make each *ready* index represent more than one value on the `results` array. This will reduce the amount of locking needed in the worker threads, but it will delay the communication of results to the writer thread. We will test a couple of different approaches here to find out what works best for our program.

Static vs dynamic load balancing

In the lecture on parallel computation, we also learned about static and dynamic load balancing, which are two different ways of balancing work between threads. Say that we have r worker threads. In our case, static load balancing would mean to split the range of x-values that we want to compute Newton's method for into r disjunct, equally sized intervals, and then let each thread compute the results for all points in one of the intervals. This works fine if the workload for each interval is roughly the same, as then all threads should theoretically finish their computations at around the same time. However, if this is not the case, some threads will likely finish their computations earlier than others, leading to idle threads and suboptimal CPU utilization.

In dynamic load balancing, the work would instead be gradually handed out to each worker thread. The extreme case would be to consider the computation of Newton's method for a single x-value as a single work load. Each worker thread would then be assigned a single x-value to compute the result for, and then ask for more work when it is done computing the root for that x-value. The main thread will then give the worker thread a new x-value to work on, and this will continue until the roots for all x-values have been calculated. For uneven workloads, dynamic load balancing will result in better CPU utilization compared to static load balancing, but it also incurs additional communication costs and is more complicated to implement. The communication costs can be alleviated by splitting up the work into larger chunks. For example, we can consider the computation of Newton's method for an entire row in the picture as a single work load, instead of the computation for a a single x-value.

When implementing our program, we will start by implementing static load balancing as it is a lot easier to implement and will probably give us pretty good results. We will then run some benchmarks to see how the program utilizes the available cores, and if it turns out that the workload is uneven and that this is significantly affecting the performance of the program, we will look into implementing dynamic load balancing instead.

HDD vs SSD for writing files

The program will write its result to a PPM-file, which involves writing data to disk. On Gantenbien we have access to both SSD and HDD storage, which have different read and write speeds. In general, SSDs are faster than HDDs, but with the HDDs being set up in a RAID configuration they can become as fast, or even faster than the SSD.

In this program, writing to file will probably be a relatively small part of the execution time. However, since changing from SDD to HDD and vice versa is very simple (simply change the path of the file), writing to both SSD and HDD will be tested to see how much it affects the total execution time of the program.

For a real, industry scenario, it should be considered that even if the gain in total run time is small or even medium on SSD compared to HDD, it could not be worth using SSDs, as they are a lot more expensive than HDDs.

Intended program layout

The program can be naturally split into three subtasks, handled by three different threads: the main thread, the worker thread, and the writer thread. The task of the main thread is to set up the environment and start the other threads, the task of the worker threads is to calculate the root for a range of x-values, and the task of the writer thread is to write the results of the calculations done by the worker threads to file. Each task will be implemented in a separate function, which will be run as the main function of a corresponding POSIX thread. These main functions will also be split up into smaller functions for each identified subtask.

For the main function, the subtasks include parsing of command line arguments, allocating memory for shared variables, and starting the worker and write threads. As mentioned in the section discussing different load balancing methods, we will start by implementing static load balancing. The main thread will do this by dividing the work so that each worker thread is responsible for calculating the root for all x-values in a number of consecutive rows in the picture. All worker threads will be assigned the same number of rows, +/- 1 in case the number of rows can't be evenly divided across the threads. When starting the worker threads, the main function will pass the row to start on, the number of rows to calculate, and the size of the picture to the thread.

The worker thread function will use the information passed from the main function to calculate the actual x-values that it should use as starting points for its Newton's method calculations. In a sequential for-loop, it will calculate the first x-value, run Newton's method for this value, calculate the next x-value, and so on until it is done with all of the x-values in all of its assigned rows. The act of performing Newton's method for a single x-value will be broken out into a separate function, which in turn has its own subtasks. These tasks are to check if the current x-value is out of bounds or close enough to a root, and then calculate the next x-value.

Two separate functions will be used for calculating the function value and its derivative for given values of x and d. These functions will be called a large number of times, so it may be a good idea to mark them as `static inline` in order to give the compiler a hint that it could be benificial to inline them. As for checking if an x-value is close enough to a root, since it is assumed that the degree of the polynomial is less than 10, we can check the degree given to the program in the main function and populate a global array with the roots for that specific degree. When checking whether an x-value is close to a root, the worker threads will simply loop through the values in this global array and check if the x-value is close to any of the values.

When the computation for an x-value is done, the result will be stored as a struct in a matrix stored in shared memory. This struct will contain the root to which the computation converged, and the number of iterations it took. As discussed in the section on thread communication, a separate *ready* matrix will be used to communicate the availability of a result to the writer thread. This matrix will be protected by a mutex in order to protect against concurrent reads and writes.

The writer thread function will start by writing the headers of the two PPM files, and then start checking the *ready* matrix to see if a result is available. When it sees that a result is available, it will read it from the result matrix and convert it to the format needed for the two PPM files. It will then write the converted values to the actual files.

Benchmarking will be done outside the of the program using the `time` command in bash.