Assignment 2 report

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 an 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.

Dynamic memory and memory fragmentation Fragmentation can occur when we allocate dynamic memory in noncontiguous blocks. The memory “between” the blocks is unallocated and can be too small to be usable. These gaps summarize leaving substantial amounts of memory wasted. In addition to that the noncontiguous allocated memory can slow down memory access. When dynamically allocating memory it is imperative to free the memory, once the process is finished, in order to avoid wasting memory space (memory leakage).

Reading and writing files Files can be read with fread() and written with fwrite(). These functions are less complex than fscanf() or fprint() and therefore faster. fwrite() and fread() are using buffer to write or read. This results in faster computation.

Command line arguments It is possible to pass arguments to a program via command line. If intended, these additional inputs allow the user to change the program behaviour. POSIX library contains an easy way to implement the parsing of command line arguments.

Locality According to wikipedia locality is defined as:

> *the tendency of a processor to access the same set of memory locations repetitively over a short period of time* -[wikipedia: Locality of reference]

Exploiting this principle we can use this to access arrays in a faster way, increasing performance. Most arrays, or a big portion of them, can be put within one cache line. Caching reduces direct access to memory and keeps part of the data in one quickly accessible location. It is important to remember, that accessing data from different parts of the memory will always take more time than accessing data from cache.

In the assignments we looked at techniques like loop unrolling to optimize performance.

Alignment and Padding Different types of variables require different amounts of memory.

Variables should be declared in groups sorted by type. This way the compiler can allocate memory more efficient. This avoids waste of memory space.

HDD vs SSD While a single SSD is generally faster than an HDD gantenbein uses one SSD and a HDD-raid. As seen in the tests of Assignment 1, this raid increases performance of the HDDs leading to a similar performace as the SSD. Therefore the operation for writing files should not have a big overhead.

Threads and Mutexes

> *A thread of execution is the smallest sequence of programmed instructions in order to achieve the parallelism.* -[wikipedia: Thread (Computer science)]

In principle, the threads can be executed in parallel, by different processors. To avoid conflic on shared variables they have to be synchronized, a naive method is the use of mutexes.

Mutexes will allow global variables (used for data tranferring) to be accessed by one single thread at a time. In order to assure deterministic behaviour it is imperative that the variable is accessed by no more than one thread at a time.

Load Balancing Some problems can be split up into independent subproblems. These problems will be distributed among several threads, simplifying the work done by a single thread.

Possible Bottlenecks Even with fast processors the performance is bound by other factors, some of them are disk access, data trasfer done in memory, the cache misses, TLB misses, branch prediction misses. These bottlenecks have to be kept in mind while assesing to the benchmarks.

Intended program layout

Overall layout The main goal of the assignment is to compute newton iteration according to the problem’s defintion on the course website. The output file format is PPM, which will reppresent the root’s convergence of the function.

All the work is splitted up among different threads, in particular:

- Master thread: it is in charge of creating and managing the working threads and the necessary data structure.

- Working threads: they are in charge of resolving the given problem.

Division in subtasks * Thread management - create threads - give tasks to threads - wait for threads - prepare all the data structures for threads * Computation + calculate roots + split into subproblems (it is a possible improvement if the computation is too slow) + write to global variables (data transfer) + check conditions for convergence + compute the next value of x

- Writing
  - assign colours to roots
  - open files

- write files
- create result pictures

Resolution of each subtask.

Thread management The Master thread is in charge of the thread management, it will parse the argument given to the program from the command line and handle properly the behaviour of the program accordingly,i.e. creating the required number of computation threads, the degree of the function and the required rows and colums of the output picture. It will also wait the computation threads to finish and as last step, a wait for the writing thread which may be slower than others since it has to deal with writing operation on the HDD.

Computation Depending on the thread id each computation thread will be in charge to compute the roots' values of the function for a given row until some convergency conditions have been reached or number of iteration is reached, as last step it will handle the writing of the global variables used for data transfer, using a mutex for avoiding parallel writing that could lead to an undeterministic behaviour.

A clever implementation could be that each computation thread measures its performance. At a certain amount of iterations it would be considered as slow and is capable of creating (and waiting) sub-threads which could, in principle, speed up the overall computation.

Writing The writing thread will open the files according to their names then use a busy form of waiting for the data for writing them in the proper order. Regularly reading the global variables, the thread will wait for a row to be finished and then start converting it into a string and mapping each value to a corresponding colour. The amount of colors depends on the degree of the function + 2 extra cases (convergence to infinity and to zero).

Another implementation could be to create a writing thread for each file since the data of each is independent from the others.

There could be also the possibility to create a thread which is in charge of only mapping colour to a data. Moreover, the thread must set the global variable item_done properly, this variable is needed for the writing thread to correctly write on the files.