Assignment 3 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.

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 a reduced numbers of disk access and as consequence an increase into the performance.

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, in our case it will only define the maximum number of threads. 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.

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 and reading files should not have a big overhead.

OpenMP >OpenMP is an application programming interface (API) to support the shared-memory multiple-thread form of parallel application development. > *T.Sterling, M.Anderson, M. Brodowicz (2018) High Performance Computing, Modern Systems and Practices.

OpenMP is suitable for fine-grained parallelism. Compared to POSIX threads it provides the possibility of a much shorter and more decriptive code implementation.

Threads are the primary means for parallel computation. Each thread is a schedulable entity within which a sequence of instructions are executed, it has private variables and internal control at its disposal. OpenMP provides the flexibility to define the type of scheduling the programmer prefers. This is done in an efficient and concise code. Specified pragmas are used to define the characteristics of the parallelization tailored for a specific proram. Furthermore, offloading can be used as a simple way to exploit co-processors. On each device there could be a team of threads adjusted to execute a specific type of problem. The collection of the teams of threads is a league,one for each device. For instance, a for loop can be split over a number of threads, together constituting a team that is executing the whole loop in parallel.

Intended program layout

Overall layout The main goal of assignment 3 is to compute and count distances between points in 3-dimensional space, using OpenMP as instructed in the problem's implementation details on the course website. The output of the program will be printed on standard output, which will consist of two column, the first being the distance between the two 3-dimensional points and the second the number of times this distance was computed among all distances. Pratically speaking, it consists into a master thread which will take into account the reading from files, and the printing to standard output, and a team of threads which will compute the distance and update the relative index in the data structure

Input The points in the 3-dimensional space are given in a file. To read this file we have to analyze it with the help of fread(), fseek() and so on. The input coordinates are read as strings and saved into two buffers (window on the file),partially overlapped. Then the strings are parsed for computing the distances saving the points as floats, every window is parsed and combined with the other points from the other window. Exploiting the fact that the distances are symmetric, the number of distance calculation will be (n^2)/2 with n the number of points into the file.

Division in subtasks

- Read and parsing the file: We read from the file to a buffer of size 2048 bytes, so that we load efficient chunks of data into RAM. Loading the whole file into memory would not be feasible because even if it fits space for program code will be scarse, and especially frequently used commands will not have enough space available making the execution way slower.

- Computation of the distances and increment of the corresponding counts: Thanks to the fixed inputs, it can be computed the maximum distance that out program will calculate (346.40) from

the formula of the euclidian distance. This allows to create a vector of length 34640(all the possible distances with a difference of 1/100), implicitly sorted and for each and every entries the numbers of occurencies of the given index (distance*100) will be saved (reducing the critical section of our program in the parallel part)

- Output : For printing the results we are using fprintf() and post the result to the stdout. Printing the distance values and its relative frequency.

- Memory management : The assumption are that the fixed input points is respected, allowing also to simplification in terms of buffers used for parsing the file. The buffers,or windows, are built in such a way that they are reducing the number of total disk acess and in the meanwhile they are able to create all the possible combinations between points, thus calculation of distances. For this reason, the windows are capable of store at each time 1000 points (2400 bytes) as strings since it is known that the length of a row is 24 bytes.

- Parallelization: The parallelization will occure when the two windows have to be parsed, the outmost loop will be executed by just one single thread while the innermost loop will be the real parallelized part, which will also containt the critical section (the update of the frequency's vector). It is well known that in a cuncurrent enviroment the bottleneck is the critical section and the duration of its duration can seriosly create performance problem but in our case the update operation (the critical section) is just an increment of the by one of an entry of a vector, with a constant access time, which means a very short critical section that leads to an overall increase of the program's performances.