Assignment 3: OpenMP

Relevant concepts

Loop collapsing

In the lecture on OpenMP we learned about how nested independent loops can be parallelized using the “collapse” keyword. This fuses the loops so that they are iterated on using a combined variable, which can improve load balancing as non-nested programs are easier to parallelize effectively.

In the cell_distances program, we will need to have nested for-loops for computing the distances between all the points, and here using loop collapsing might increase performance.

Synchronization

There are a few different ways of doing synchronization in OpenMP, such as using “critical”, “atomic”, and “reduction”. The “critical” keyword defines a critical section that only a single thread can execute at any given time, and the “atomic” keyword marks an instruction as atomic, which can remove concurrency-related issues. The “reduction” keyword on the other hand lets us define an operation that should be performed by all threads to combine each of their individual results. In the lecture we learned that we should use reduction whenever possible, as it is usually the fastest of the three methods.

In the cell_distances program, we will need a mechanism for gathering the counts of the different distances from all of the threads. This can be done by using array reduction with the plus operator, which was demonstrated in the lecture as well.

Tasks

As mentioned in the lecture, OpenMP tasks are code blocks that are processed independently of each other. They can be triggered by using the “task” keyword, and waited upon by using the “taskwait” keyword. This mechanism can be used to trigger a background task to be executed while some other computation is being run.

In the cell_distances program, since we won’t be able to fit all of the cells into memory simultaneously, we will need to read the input file in chunks. Here, it could be beneficial to trigger the reading of the next chunk in the file before doing the computations on the current chunk, as then both the reading and the computation could be done in parallel by different threads.

Schedulers

“Scheduling” refers to how work is split up among multiple threads. This is synonymous to load balancing as seen in the lecture on parallel computing. OpenMP provides a few different schedulers, such as static, dynamic, and guided, which each have their respective use cases. The static scheduler introduces the least amount of overhead, but it can lead to suboptimal CPU utilization due to its simplistic nature. Our plan is to start with the default scheduler in OpenMP, and then try different schedulers if the default one proves to be too slow for our needs.

Elementary functions

As we learned in the lecture on optimization, and got to experience first hand in assignment 2, elementary, and especially numerical functions can contribute a lot to the overall runtime of a computational-heavy program. One way in which this will affect the implementation of the cell_distances program is the rounding of distances to two decimal points. The obvious way of doing this would be to use the “round” function in the math library, but since round off errors are tolerated to a certain extent it might make more sense to simply cast the resulting double to an integer, thus always rounding the distance down to the nearest two decimal points.

Intended program layout

In order to keep the memory usage below the specified limit we can’t read all the points into memory at the same time, and since we need to compute the distances between all points in the entire file, it will be inevitable to read the same point multiple times. However, we can keep file reading to a minimum by reading the input file in chunks. The way we’ll do this is to have two nested loops. The outer loop will read a chunk from the file and compute the distances between all the points in that chunk. The inner loop will then go through the rest of the file in chunks, and compute the distances between all points in the outer chunk and the current inner chunk. A subtask of the program will thus be to read a single chunk from the file, starting at a certain offset.

Since the positions of all cells are specified using exactly 3 decimal points, the number of values needed to represent a position in one dimension is 20000 (−10.000 to +10.000). This means that we can encode all possible cell locations using three shorts instead of three floats or doubles, which saves us at least 12 bytes per cell. This will also be beneficial for numerical calculations, as computers work better with integers compared to floating numbers. Another subtask of the program will thus be to parse a single input line and return it as three shorts, and a subtask of this is to convert a string representing a single position to a short.

In order to parse a position as a short, we can make use of the fact that the numerical value of the character for a digit minus 48 is equal to the numerical value of the actual digit. The parsing can then be done by subtracting 48 from the numerical value of each digit and multiplying it with a power of ten that corresponds to its position in the number, and taking the sum of all resulting values.

From the way we read the input file, we can see that we need two subtasks for computing distances for chunks: one for computing the distance between all points in a single chunk and one for computing the distance from all points in one chunk to all points in another chunk. The execution of these functions will probably take up most of the execution time for the program, and will thus need to be parallelized. Since both functions involve two nested for loops, they can both be parallelized using “#pragma omp parallel for”. The two loops in the latter function can be collapsed using the “collapse” keyword, but this will not be possible for the two loops in the former function since they will not be independent, since we don’t want to compute the distance between two points twice.

The maximum distance between two points is sqrt(20^2*3) ≈ 34.64. Since the distances should be specified using two decimal points, there are only 3465 unique distances. The best way to save the counts of these is to create an array of 3465 counts, and to multiply the rounded distance by 100 to map it to an element in the array. When computing the distance, we can make all calculations using the shorts that we use to represent positions, and then convert the result back to a short. As mentioned in the section on elementary functions, the best way to do this conversion is probably to simply cast the resulting value.

Once all the distances have been computed and their counts have been stored, it will be trivial to loop through the array and print all the values.

If we use three shorts to represent a cell, each cell will use 6 bytes of memory. This means that we can fit approximately 180 million cells in memory without exceeding the 1 GiBi byte limit. If we set our chunk size to 100 000 (which is equal to the number of lines in the largest test file), we will use approximately 1.2 MB of memory for storing cells. The array used to store the distances will use 13 860 bytes (if each count is stored as a 32 bit long), which means that we are well below the limit.