High Performance Computing
Individual assignment

Desislava STOYANOVA

31st May 2016

1 Comparisons

1. Create an array of randomly created stars

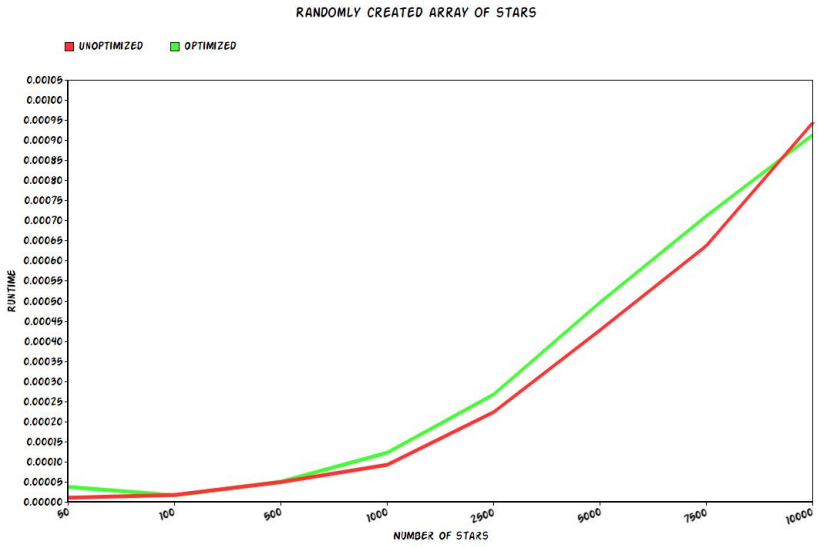

Figure 1: Create an array of randomly generated stars

2. Sort the array in terms of the distance between each point and the origin

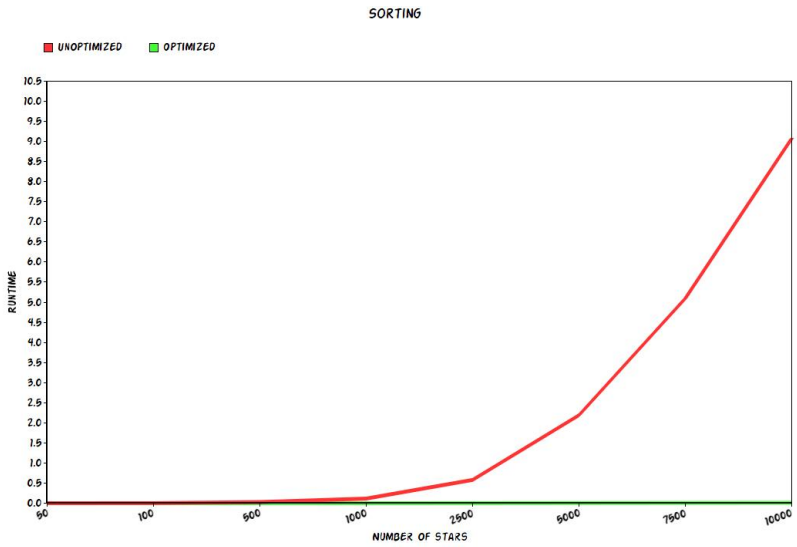

Figure 2: Sort the array of stars

In the initial version of the project, we have been using a simple **Insertion sort** in order to sort the array of stars. On one hand, it was a suitable option for smaller arrays since

the complexity in the best possible case is equal to $O(n)$. However, for larger arrays, it does not perform that well and having in mind that its upper bound is equal to $O(n^2)$ in the worst possible case, we decided to improve the sorting by using **Merge sort**.

Moreover, the **sort** function has been updated in a way that it invokes Insertion sort for arrays of size lower than 43 elements and Merge sort for larger arrays since Insertion sort performs better for smaller input sets.

Apparently, that gave the result that we were expecting. For input set of stars with size lower than 1000 elements, both algorithms perform almost equally well. In contrast, the runtime of Insertion sort increases dramatically for arrays that consist of more elements, while the Merge sort performs considerably better.

3. Allocate the matrix

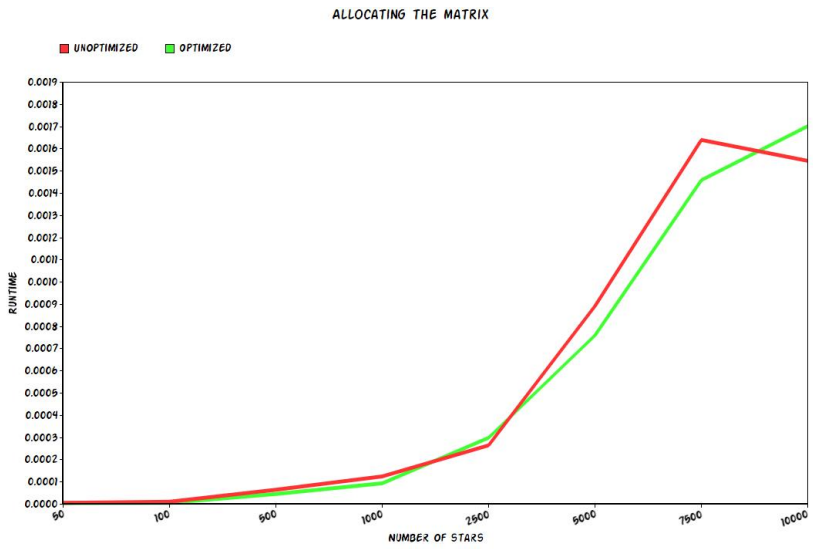

Figure 3: Allocate the matrix

We allocate the matrix using the function **malloc**. Initially, a pointer of type **(float_t**)** is created that points to the heap memory. After that, using a **for** loop, we allocate all of the subarrays that identify the matrix. This way of allocating the memory has been used in both implementations and it is not surprising that no significant improvement is visible on the graph.

4. Fill in the matrix with elements

It is noticeable that one of the functions that met a considerable improvement was the one related to filling the matrix with elements. In the initial version of the project, we used two nested **for** loops, and also calculated the value for each element of the matrix, which was happening in the nested loop.

In order to improve the performance, we decided to use a technique covered during the lab sessions which is called *loop unrolling*. To be more concrete, we increased the step of the nested **for** loop from **i++** to **i+=4** and in this way not only one, but four instructions could be done without interruptions for checking the loop condition.

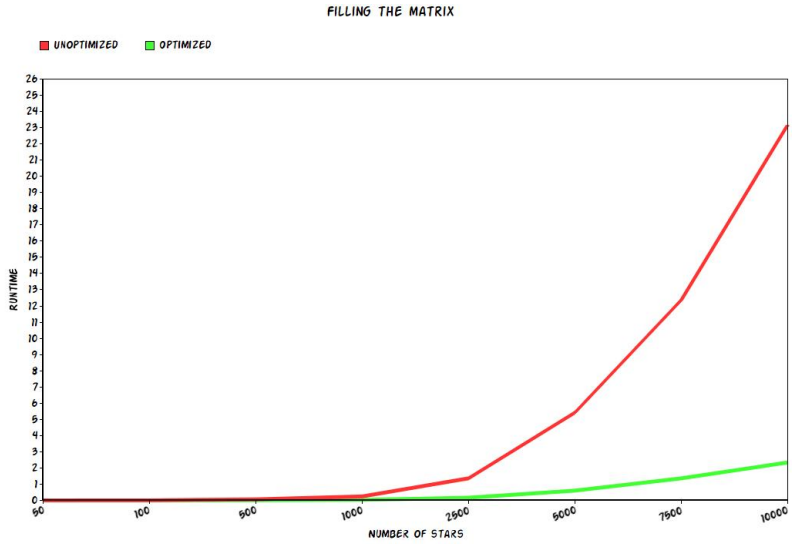

Figure 4: Fill in the matrix with the required values

As you can notice on the graph above, for arrays of size 1000 and less, both algorithms follow similar pattern and have a similar runtime. However, the larger the input is, the slower the unoptimized version appears to be. Moreover, while the unoptimized version of the algorithm goes up rapidly in terms of runtime performance with increasing the size of the input set, the runtime of the optimized version meets a steady growth and happens to have much more satisfying performance.

5. Generate the histogram

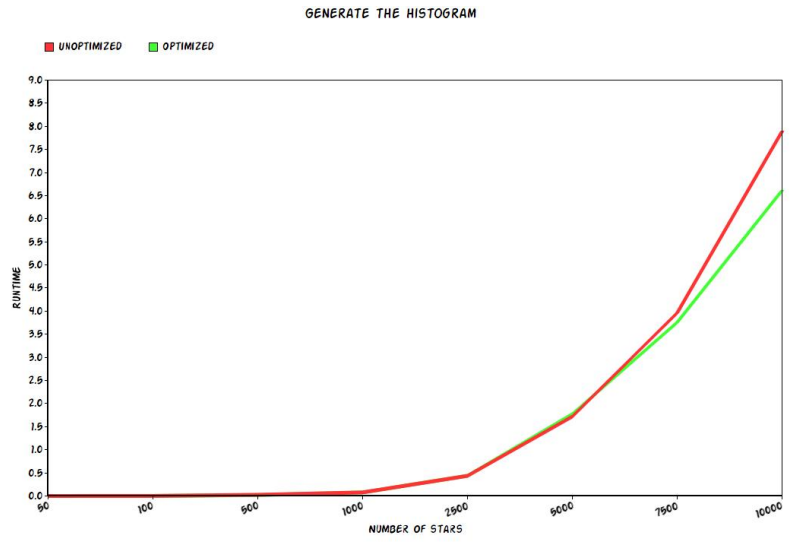

Figure 5: Generate the histogram

We have included some minor improvements in the function related to the histogram and how it is generated. For example, we have removed the arithmetic expressions that appear too often and we have stored them in variables. However, we have not met any significant improvement, but we suggested that if blocking was used (the technique presented during lab session 3), the performance would be much better.

2 How has the code been optimized?

In order to improve the overall performance of the program, we have used several techniques covered during the lab sessions. Some of them include:

1. Loop optimization
2. Faster boolean evaluation
3. Memory allocation/deallocation - the memory leaks form the initial version have been removed in the optimized version of the project. Now, the allocation and the deallocation respectively are properly used.
4. Avoiding too many small function calls - we rather store them in variables for future use.
5. The keyword **restrict** has been used whenever possible - in functions that take more than one pointers to the memory as arguments.
6. The keyword **const** has been used whenever possible - most frequently in functions' declarations where the arguments are used only to read the data, we want to ensure that the data is not going to be changed. Also, the array that we use for storing the spectral types is now defined as **const**.
7. Instruction-level parallelism (ILP) - loop unrolling.

3 Instructions

1. Open the terminal and navigate to the directory containing the unoptimized or optimized version of the project.
2. Run the command **make**.
3. Run the command **./main N** where **N** is the number of stars.