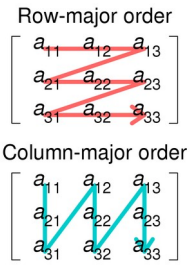
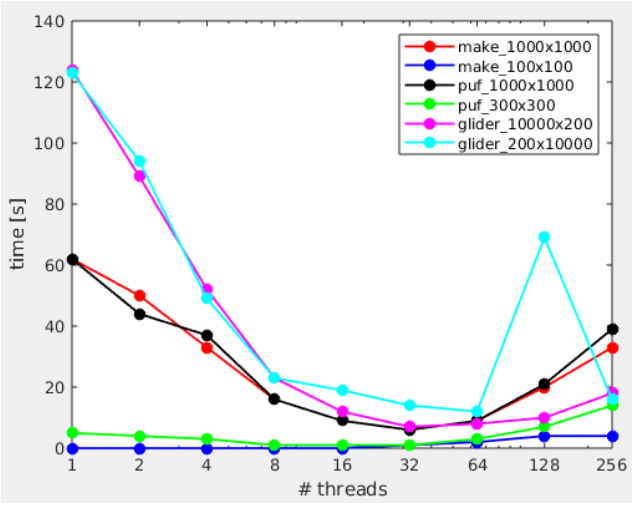
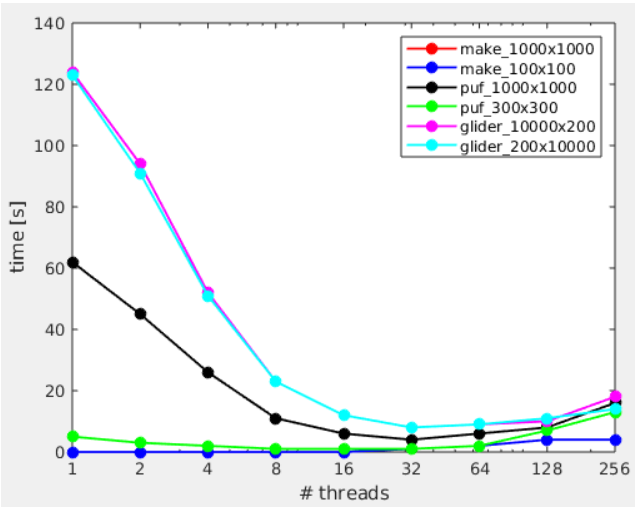
The general idea of the parallelization is to separate the columns of the grid in chunks and process each chunk with one dedicated thread. The reason why I divide the grid into columns and not into rows is that the memory layout of the *Game Of Life* grid is in column-major order.
The cells of one column are continous in memory while the cells of one row have an offset of the number of rows in memory. With this approach spatial locality is exploited better which results in an improved performance.



My first approach for the chunk separation was simple. Each thread processes an equal number of columns, while the last created thread additionally calculates the remaining columns, if the number of columns is not exactly dividable by the number of threads.
While it makes sense at the first sight, it turned out that it has a load balance issue since the last thread might have to process up to *"nprocs – 1"* more columns than the other threads, which results in poor performance. The other threads have to wait for this single thread to finish at the barrier for synchronization. This happens each iteration and results in a lot of unused processing time.

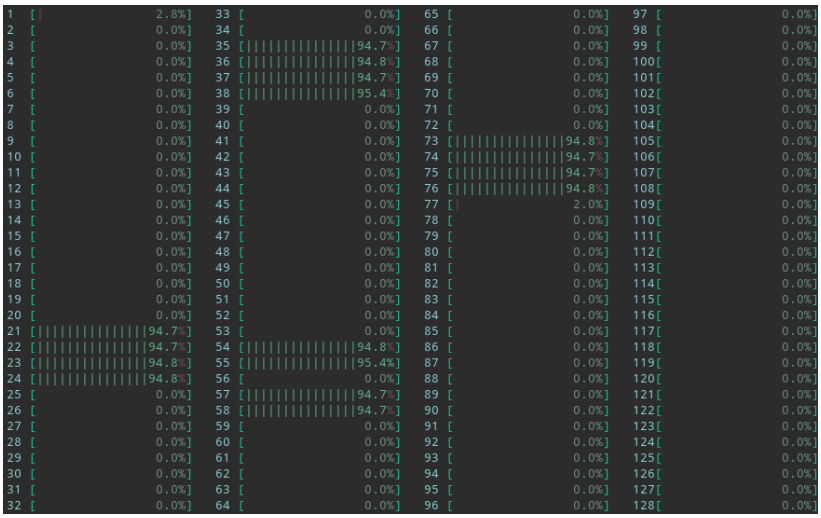
My second approach solves this issue. Instead of giving the remaining columns to the last thread, they are evenly distributed over all threads. Since the number of remaining columns is always less than the number of threads, all threads process a maximum of one column more than the other threads.



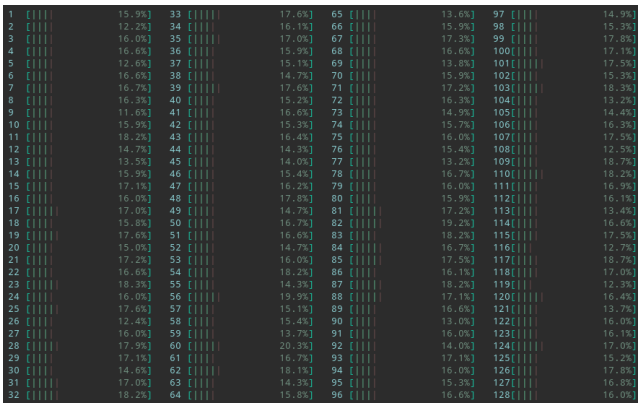


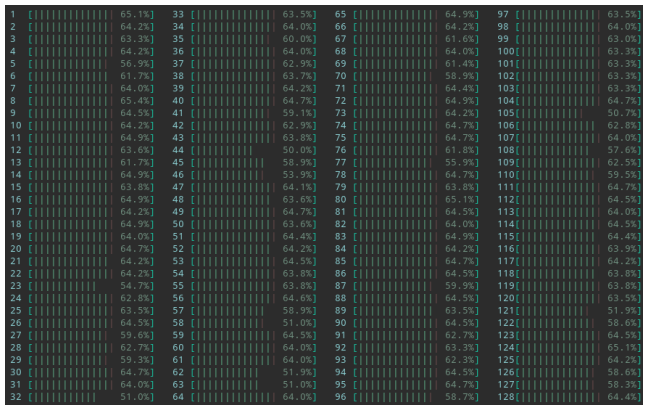
The left chart visualizes the first approach and the right one shows the second approach. Especially for "glider_200x10000" with 128 threads the approaches differ extremely. This makes sense since *"200 mod 128"* is 72. This means the last thread needs to calculate 72 more columns each iteration and the other threads have to wait for it. Each column contains 10.000 cells which are 720.000 more cells just for one thread. The alleged performance improvement for 256 threads is a consequence from the program limiting the number of threads to the number of columns and *"200 mod 200"* is 0.

The chart illustrates that the program is not perfectly scalable. For example using two threads instead of one does not reduce the required calculation time by half. To achieve this it usually requires four threads. This is explained by the fact that not the full program is parallelizable because the update of the grid has to be updated in a sequential manner (Amdahl's law).



This screenshot proves that the parallelization works as expected. The command issued was *./glife sample_inputs/glider 16 10000 10000 200*. As you can see 16 threads are almost completely utilised while all other ones are idling. Except for one (core 1) but this is caused by the *htop* process.





These pictures demonstrate the different CPU utilisation for different parameters. The left one is started with the grid size 500x1.000 while the right one has been given the size 500x10.000. As you can see the cores on the right one are way more utilised than on the left one. The reason is the management overhead of the left one is bigger because the barrier synchronization happens in shorter intervalls than on the right one.

In summary the application is very scalable but usually only for the half to full number of physical cores as the number of threads. Using the number of virtual cores actually decreases the performance. Using more threads than there are virtual cores decreases the performance even more. This is expected because the synchronization overhead grows if using more threads but they can not be processed in parallel because of the lack of CPUs. This behavior could be observed on the provided server (64 physical cores, 128 virtual cores) and also on a computer with an Intel i7-7500U (two physical cores, four virtual cores).