Sanjana Kambalapally 999147873
MD Rayun Mehrab 999036582

Among the three loops given, we identified that the board had a dependence between the loop iterating over generation. Hence, the i and j loops were to be parallelized. The following is a list of optimizations our team implemented for HW 5.

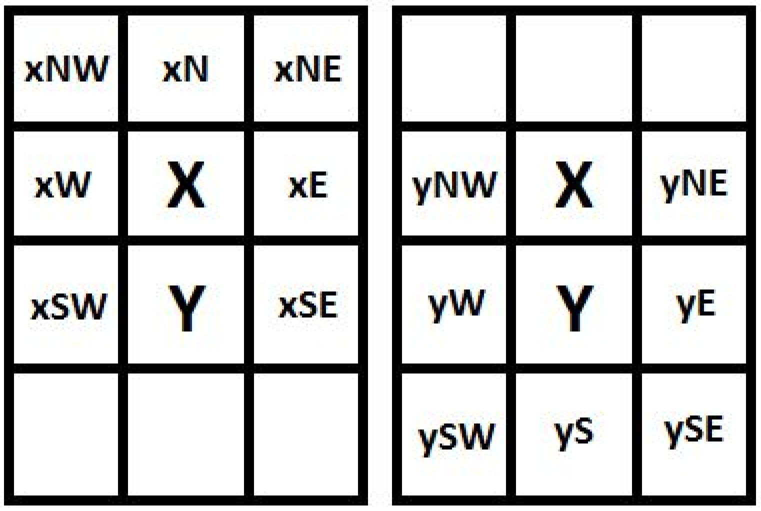
Parallelization using pthreads: 8 pthreads are created for every generation such that each thread can run on an individual CPU. Once all the threads complete execution for a generation, the boards are swapped before continuing to the next generation.

Loops switching: The data in the input board is stored in column order and not row order. Switching the i and j loops reduced the number of cache misses.

Code Motion: Ignoring tiling which is explained later, moving the *Integer* calculations for *jwest* and *jeast* outside the i for-loop allowed for them to not be recalculated for every new i as they were loop invariant.

Formula: The mod formula that is used to calculate the *inorth*, *isouth*, *jwest* and *jeast* coordinates have been optimized using the ternary operators and inlined to improve performance.

Barriers: Since each generation's evaluation depends on the previous generation's values, the threads cannot get ahead and move to the next gen before the other threads finish evaluating the current one. This requires creating and joining the threads for every generation separately. For large number of iterations, this creates a performance issue. Pthread barriers can used in such case so that the threads aren't created and joined for every generation. The barriers ensure that each thread waits at the end of its evaluation for the current gen. Once all the threads finish executing the code for its current gen and reach the barrier wait point, they all resume and move on to the next. This improved the performance by 2s.



Loop iteration memory sharing and Tiling: In order to decrease the cache miss rate, loop tiling was implemented for both i and j loops. Since there are 8 threads, the i-size for the tile block cannot be larger than nrows/NUM_THREADS, so it was set as such. The j-size for the tile block was set at nrows/2. In order to do tiling, the inner j loop had to be moved inside the outer i loop and thus, the code motion optimization done above to *jwest* and *jeast* came undone for each iteration of outer i. However, this code was still invariant to the inner i loop (i2) so it was still kept out of it. Tiling, however, didn't provide as much of a performance boost as did memory sharing. As can be seen in the example picture above, for each inner loop i iteration (i2), the previous iteration (X) 's neighbour life values xW, X, xE, xSW, xS and xSE are the same as the next iteration (Y) 's neighbour life values yNW, yN, yNE, yW, Y, yE. As such, memory sharing has been implemented so that for each iteration of the inner loop i (i2), the previous iteration values are taken instead of doing redundant calculations. Taking it one step further, for each inner j loop iteration's first pixel (when i2 = i), 6/9 neighbour values can be taken from the previous iteration's first pixel as well, similar to i. So when the memory sharing is implemented, it ensures that the BOARD macro doesn't have to be called each time redundantly, which accesses the inboard array every time. There are still some redundancies that can be removed, e.g the j loop iterations can share memory for each value of i. But since that requires using an array anyway, which will not change things.