ECE454 LAB 5 report

Xian Zhang 1000564766
Qili Lu 1000354537

Performance Summary:

8 cores machine:
Original version: 140 seconds
Optimized version: 2.05 seconds

Algorithm Level optimization:

"Premature optimization is the root of all evil" -- Donald Knuth

To maximize the performance of gol program, we first start to think about a more efficient algorithm before the actual system programming level optimization. We design a new simple algorithm to reduce the redundant calculation from original code. The key idea is to store the neighbour count along with its alive status of each cell into a buffer. The extra storage space is thus O(n). With the count stored, we do not need to recalculate the count every time every cell. Instead, we only update the count when a cell is detected to be updated to a new alive state at next generation. For example, if a cell is turn to alive from dead at next generation then it will add 1 to all of its 8 neighbour cells on their neighbour count. This way many redundant calculation to neighbour count is removed. At stable stage, there is only a few cells keep updating their state.

Another method we used is for alive() function which is used to check a cell's status for next generation. Instead checking if a cell is alive by comparing its neighbour count every time, we decide that we can use a mapping table to help so. Since the count is always less than 8 we use the previous 3 bits of a char type variable to represent its count and a fourth bit to represent whether the cell is alive or not. The totally 4 bits will then be used as an index into our mapping table of 32 entries to find out its next state. We decide that the mapping table is frequently accessed and will be stored into cache, and the access to cache is much more efficient than a comparison calculation from alive function.

To further reduce redundant work, we use 2 frame buffers for each thread. At stable stage, some part the game board might be free of updates, it is waste of computation to check each cells for updating. We setup a threshold, when the current generation is larger than the threshold we decide that it is in stable stage. Then we check if the current frame is seen before by using memcpy for this frame and our frame buffer. If the frame is seen before we simply copy the next generation frame from frame buffer to the board and update the border rows which might change during border update. If the frame is not seen, we copy the new frame into frame buffer and replace the oldest frame in frame buffer.

System level optimization:

Loop Unrolling:
In function parallel_game_of_life_new(), we used loop unrolling to speedup the update process. In the original code, when we trying to update the count for each cell, we update one cell every time. If we used loop unrolling, we will update two cell in each loop, by checking if the cell is updating, and update number by calling UPDATE macro defined in the code.

Loop Interchange:
In function parallel_game_of_life_new(), we used loop interchange to speedup by change the order of accessing the loop. We access the row first, and then access each column in a row-wise fashion. This increase the spatial locality and cache locality, thus utilize cache more efficiently.

Loop invariant code motion:
Loop invariant like inorth=i+1 and isouth=i-1 is unchanged every row so that we decide to move the expression to outer loop and save some efficiency.

Space-Time trade-off:
Boundary check from the original code is very time consuming, especially when it is called inside the inner loop of a heavy calculation function and it uses modulo. We remove the boundary check and the use of modulo operation by first calculate the boundary first before the looping of row/column starts. To do so, we have to use some redundant code in function like update_lower_hori_boundary_parallel(), update_row_parallel_upperself(). We improve the time efficiency with tradeoff of code size.

Function Inlining:
Both update_verti_boundary_i_j0() and update_verti_boundary_i_jl() have been set as inline function because in the main function, they have been called most frequently. Since the call to an inline function is replaced with the function itself, the overhead of building a parameter list and calling the function is eliminated in the calling function. Since there is no function call, the overhead associated with entering the function and returning to the caller is eliminated in the called function. Making both function inline greatly reduce the overhead, and speedup the program.

Multi-threading:
To further speed up the gol program we use 16 threads to parallelize the game of life main calculation part, considering a 8 cores 16 hyperthreads machine. For each thread we assign same number of rows as workload. The workload number is simply the number of total rows divided by the number of threads. Mutex lock is used to ensure synchronization among borther rows between two neighbour threads. pthread_barrier() is used to force synchronization between each generation for all threads.