Report on LAB5

ZHANG QIANHAO / 1004654377

CHEN JINGFENG / 1000411262

Parallelization strategy: barrier parallelization

- We choose barrier parallelization to avoid the release and join of worker threads at every iteration.
  With a barrier the threads will wait for each other until all of them finished this iteration. This is necessary because each thread will rely on the correctness of its neighbor of boundary column, which is written by its neighbor worker.
- We implemented the barrier in barrier.c and barrier.h, with pthread mutex and pthread cond.

Optimization:

- By running gprof, we found 'BOARD', 'mod' and 'alivep' are the bottleneck for sequential implementation. In the optimization of sequential part, we focused on how to minimize the calling to these functions.
  Our first intuition comes from lab2, where the image is sparse. We decided to implement 2 hashtables, 1 storing only those alive pixels from previous iteration and 1 storing only those who may be alive in this iteration.
  The hash is based on the memory address of the pixel byte. Each item has a special data 'alive_neighbor_count' which is incremented each time an alive neighbor is read.
  This way our iteration only need to be on those may-be-alive pixels, which is at most current alive pixels plus their neighbors.
  Each time we finish a iteration of alive pixels, we can directly know the aliveness of this pixel of the next round by looking at its 'alive_neighbor_count'. This could greatly reduce the number of loop size when the image is sparse like lab2.
  By running gprof, we found the bottleneck is the insertion of pixel into hashtable, which we have tried to optimize. The outcome is that hashtable implementation can only beat the original sequential implementation when the image is super-sparse (< 1%). But this is not the case for our test case (~4%). The test case is dense at start, and is with about 4% density after 5000 iterations.
  This is probably because that the sequential implementation already has the best cache performance, and the hash lookup mostly causes a cache miss. Although the sparsity can save us 99% of the pixel lookup, the operations and cache performance in a cache lookup then makes each lookup 100x times more expensive.
- Given that, we then focused on how to optimize the original sequential code about how to without changing the loop structure, but meanwhile reduce the funciton call. Specifically, 'alivep' still has to be called upon each pixel because of the loop structure, we focused on how to minimize 'mod' and BOARD.

1. Avoid unnecessary read.
   This is based on a observation on the sequential implementation: each pixel in the image

is read for 9 times.
Below is part of the image:

nw	**n**	**ne**
w	c	e
sw	s	se
a	b	c

where c is our center of the lookup window, our next read will be on a, b, c. So the aliveness of c can be determined by itself given its neighbors. When reading a, b, and c there is no need to re-read north and center columns, just slide them: w = sw, nw = w, etc. Then update sw, s and se.

In this case, each pixel, instead of being read for 9 times (neighbors and itself), is read only for 3 time (itself and 2 cross-column neighbors).

To even reduce this number further, We tried to use a window of size 3-by-4:

jnw	**jw**	**jjw**	**jjne**
w	jc	jjc	jje
sw	js	jjs	jjse

So that we read 2 cross-column neighbors each time, and ideally the read for each pixel will be reduced to 2, but it turns out to be not beneficial. This is probably because we need to compute the aliveness of jc and jjc by invoking alivep which will use all the values above, which brings too much register pressure. (Unlike 3-by-3 window, where westside and eastside in the same column can be merged into one variable since they are always needed together. This saves register usage.) We finally use a lookup window of 3-by-3.

2. loop peeling;
   This is divided into 2 levels:
   At column level, we choose to split the thread worker into 3 types.
   if the worker deals with top part of the image, its window's starting column needs to be wrapped around ncols. This is hand-coded.
   if the worker deals with middle part of the image, no wrap-around is needed.
   if the worker deals with the bottom part of the image, its window's ending column needs to be wrapped around ncols. This is hand-coded.
   By peeling this loop, we avoided any invoking of mod at column level.
   At row level, we choose to split the iteration. Our first lookup window will be centered at the bottom of the row, so that the only pixel that needs to be wrapped around will be treated outside the loop. So in the loop of rows, no wrap-around is needed.
   Combined, we get rid of the function usage of 'mod'.
3. loop unrolling;
   Firstly we re-ordered the iteration to be column-major, because we found the image in memory is column-major (by looking at BOARD macro function)
   At row level, we found that the sliding of the lookup window can be unrolled.

After some testing we found the unrolling size 16 gives the best performance.
By parallelization we figure the speedup to be ~8, and by reducing reading the speedup of reading is ~3, by loop unrolling and loop peeling there should be some constant speedup for no 'mod'. We reckon the combined speedup to be around 30.