COE 4DS4 – Lab #7 Report

Group Number: 33
Moshiur Howlader / 001316948 / howlam@mcmaster.ca
Ryan Ganeshan / 001322407 / ganesr3@mcmaster.ca
March 14th, 2018

Exercise 1:
We have added new mailboxes for bubblesort start, bubble sort done, generate array start, and mergeSortDone. Everything but the last mailbox mentioned was used to arbitrate the 4 sort tasks. gen_array_i() task were used to generate our arrays. bubble_sort_i() were used to sort the arrays individually. merge_sort() was used to combine the individually sorted arrays into a bigger array and sort via bubble sort. The main start and end of the main Total Program Counter was implemented in the sorting(). For our test, Total Program Counter (PC) was 6,488,972,937, while the individual bubble_sort_i() PC were 1,597,159,733, and the sort function was 4,847,106,113. The numbers make sense since the bubble sort function is ran concurrently, and the final sort function plus the sort function runtime adds up to roughly the total run time of program approximately.

Exercise 2:
We are implementing this code as positive height BMP. The major change in the code is the implementation of the R, G, and B WriteQueue in the pipe flow. From the source code, instead of computing and passing Y value through the Y mailbox, we have passed the RGB values through the RGB mailbox in the in the BGR order. Majority of the changes made were in compute_Y_task(), where we implemented the 4 filter equation in the order appropriate for the positive height (quadrant 3 filter is implemented in quadrant 1, and quadrant 1 filter is implemented in quadrant 3). This flip is done since in positive height, the pixels are read bottom up. Also the reading done at the beginning is done in the 2,3,0,1 order since we are reading positive height (bottom up). Finally, in Process_Y, checking for whether the Queue became full after posting the fresh new RGB values, we would either increment the index for that queue or empty it, if it became full to arbitrate the flow of the queue data to SD_Write.
Exercise 3:
Our scheme for storing the sparse matrix was as follows. We allocated memory to hold up to 70,000 random values from -1000 to 1000, as well as an array to hold the column locations that those values. We also had a 1000 index array, where each index corresponded to how many non-zero values were generated per row (so we can traverse our mentioned value and column arrays). These arrays were dynamically allocated, and then freed at the end of the while loop, which can account for multiple runs. We obeyed the memory constraints for all parts of this exercise. We modified the struct within the shared memory to house all the information needed to utilize CPU 2 to help with the matrix multiplication. We passed 5 rows' information at a time by allocating 5x70 memory locations of appropriate size to accommodate different data types (short int, int, alt_32). To perform the calculation, we set up the struct to contain all necessary information for CPU2 to fo 5 Y[i] calculations. At this time, we set a flag that tells CPU 2 to perform the calculation. then we perform 5 Y[i] calculations on CPU1 while waiting for CPU 2 to modify that same flag, notifying CPU1 that it has finished its calculations. We iterate through the 1000 necessary Y values in this fashion. Overall, we got our exercise to a "mostly" working state. The CPU 2 calculations did not return correctly. There was probably a arithmetic algorithm errror, or data type error. We were unable to solve this after several hours. Also, it is necessary to toggle sw 17 in between runs of the exercise. The PB flag system allows for multiple runs.