CS508 - Assignment I Report

Aaditya Arora B17071

Note:

1. Instruction of executing a program is written inside each program as comment.
2. File structure `P1/p1.c` `P2/p2.c` `P3/p3.c`

P1.

There are two different dependencies in loop -

1. Loop independent dependencies : dependence exists within an iteration. i.e., If the loop is removed, the dependence still exists.
2. Loop carried dependencies: dependence exists across iterations. i.e., If the loop is removed, the dependence no longer exists.

```
49  void selection_sort(int* arr, int start, int end) {
50    int min_index;
51    for(int i = start ; i < end ; i++) {
52      min_index = i;
53      for(int j = i+1 ; j < end ; j++) {
54        if(arr[j] < arr[min_index])
55          min_index = j;
56      }
57      _swap(&arr[i], &arr[min_index]);
58    }
59  }
```

In selection sort : from statements 52 - 57 there are loop independent dependencies.
For *loop i* (outer loop) If we draw an iteration-space traversal graph.
i = 0 : 0-1-2-3-end
i = 1 : 1-2-3-4-end
….
i = end : end
Above graph shows that for given `i` we are traversing the array from `i` to `end` and at given iteration `i` we are swapping a `i-1` indexed element with an element coming from from `i` to `end`.
So, the next iteration i depends on previous iteration i-1. So, there is a loop carried dependencies for outer loop i.

For parallelising - We can split the array into 4 parts and then sort them individually in each process using selection sort and then merging them in the end in the master process.

P2.

```
73  int main(int argc, char const *argv[]){
74    int width, height, channels;
75    unsigned char* img = NULL;
76    char* img_name = argc > 1 ? argv[1] : "test.jpg";
77    img = load_image(img_name, &width, &height, &channels);
78    if(img == NULL) {
79      printf("Error: Image does not exist :/ \n");
80      return 0;
81    }
82    unsigned char** quad = NULL;
83    int quad_size[4] = {0,0,0,0};
84    quad = split_image_to_quad(img, width, height, quad_size);
85    int* histo[4] = {NULL, NULL, NULL, NULL};
86    for(int i = 0 ; i < 4 ; i++) {
87      histo[i] = (int*)malloc(sizeof(int)*8);
88      get_histogram(quad[i], histo[i], quad_size[i]);
89    }
90    print_to_file("output.txt", histo);
91    return 0;
92  }
```

The tasks for creating a set of 8-bin histogram are:

1. Reading Image
2. Dividing the read image array into 4 quadrants.
3. Create a 8 dimensional histogram feature vector for each quadrant.
4. Print 4*8 matrix to file.

There are no loop-carried dependencies in any of the four tasks. We can easily parallelise task 3. As we can see in `for loop` from line number `86 - 89`. There are no loop carried dependencies. We can split the work of getting 8-bin histogram vectors into 4 different processors.

For parallelising - we will load the whole image onto master process and then splitting the image data for 4 quadrants in master process and then passing all the 4 data stream into different process for calculating 8-bin histogram and then sending back the 8 dim vector to master process and then printing 4*8 into a file in the master process.

P3.

There are two things different in P3 from P2.

1. We need to read two images.
2. Computing hellinger distance.

There are also no loop-carried dependencies in this problem.

The task of reading an image and representing it as four 8-d histogram vectors is dependent only on image. So, we can parallelise the operation of getting a histogram of two images parallelly. The next step is calculating the HD.

```
101    // hellinger distance
102    float hd = 0;
103    for(int i = 0 ; i < 4 ; i++) {
104      for(int j = 0 ; j < 8 ; j++) {
105        hd += sqrt(histo2[i][j]*histo1[i][j]);
106      }
107    }
108    printf("HD = %.3f\n", hd/4);
```

There are no loop-carried dependencies in calculating HD and it can be easily parallelised.

For Parallelising : We will load two images on two different processes and then compute the four 8-d histogram vectors and pass it to the master process then for computing HD we can have four different process computing for each `i` and then summing up out of each process in master process.

[REFERENCE]

[1] https://people.engr.ncsu.edu/efg/506/s10/www/lectures/notes/lec5.pdf