CS426: Project #3 Report

Doruk Çakmakçı
21502293

Bilkent University — May 1, 2019

1 Description of Profiling Outputs Using gprof

For this project, code profiling is done in order to find what constitutes most to the running time of both the sequential and the parallel implementations. The gprof tool is used for profiling both implementations. The gprof tool is used as the following: First, the source code is compiled using -pg flag which produces gmon.out file which is used later on. Then, the executable produced from the compilation process is run using the necessary inputs to produce information that will be profiled. Finally using the gprof command, the produced executable and gmon.out file, the code profile is dumped to a text file(For this project, the resulting files are prof_sequential.txt and prof_omp.txt). In the profiling outputs, verbose information are omitted using -b flag to produce a nice output.

The output of gprof tool consists of the following components:

- **Flat Profile** consists of the information about how the total running time of the program is distributed to specific functions(e.g. distribution of timing to execution of functions) without considering the callee functions inside a function. Flat Profile has the following fields:
  - **%(in time units)** gives the percentage in terms of the total running time of the program spent in the function.
  - **cumulative(in milliseconds)** is the running sum of the number of seconds accounted seconds for by this function and those listed above it.
  - **self(in milliseconds)** is the number of seconds accounted for by this seconds function alone.
  - **calls** is the number of times this function is called.
  - **self(in milliseconds per call)** is the average number of milliseconds spent per call of the function
  - **total(in milliseconds per call)** is the average number of milliseconds spent per call of the function and its descendants per call
  - **name** is the name of the function.
- **Call Graph** describes the call tree of the program, and it is sorted with respect to the total amount of time spent in each function and its children. Call Graph has the following fields:
  - **index** is a unique identifier of the functions that combine to generate the total time. The lowest index is given to the most time consuming(Aggregate time that is generated by the function and the callee functions) function and the highest index is given to the minimum time consuming function(and called functions).
  - **%time** gives how much the function under consideration and the called functions inside this function contributes to the total timing in terms of percentage.
  - **self** shows how much time the function spends without taking the time of the callee functions.
  - **children** field lists the total amount of time propagated into this function by its children.
  - **called** field gives the number of times the function was called and lastly, name field gives the name of the function.

2 Description of Implementation Details

First, the following pragmas are used during implementation:

- **#pragma omp parallel for** is used to make a insert a parallel region around the for loop that is paralelized. Also provides an implicit barrier for thread syncronization at the end of the loop. To disable the implicit barrier, a nowait clause should be used within for loop. The loop index variable is implicitly made private to the thread. Also, clasuses such as single, atomic, critical, if, schedule, num_threads and much more clauses can be used in the scope of this pragma. Also, private, shared, default, firstprivate and lastprivate clauses can be used for declaring variable scopes.
- **#pragma omp atomic** can only be used for statements that are wanted to be atomically executed by the threads. In other words, it provides a critical region around the statement for which it is used.
- **#pragma omp critical** is used where only one thread must be executing a code segment at one time. When used, in the critical region, always 1 thread will be executing concurrently. This directive can be used with named critical region or unnamed critical region. Also, if option can be used to

In the following discussion, the parallelized segments are identified by the job they do and the line numbers in lbp_omp.c file. The sequential implementation is parallelized using OpenMP. Accoridng to profiling results of the sequential implementation, create_histogram and distance function should be parallelized. But further parallelism is also done to the implementation. First in create_histogram function, the following parallelisms are done:

- **Line 13-16:** The elements of the hist parameter should be initialized to 0. Although this parallelism is not necessary, the for loop that initializes elements of hist array is parallelized using "#pragma omp parallel for" directive. With this directive any option available for "parallel" directive can be used. To be exact, the following options could be used: shared, private, default, firstprivate, lastprivate. Since this is a parallelism that is not essential to solve for the bottleneck, only the shared option is used on hist variable since all threads do mutually exclusive job on setting the elements of hist array(there is no race condition) and we need the updated elements of hist array.
- **Local Binary Pattern Analysis** As defined in the project documentation, local binary pattern analysis should be done on the pixels of the image. This analysis is divided to interior pixels, border pixels and edge pixels. These lines do the interior pixel analysis. The local binary pattern analysis is one of the main bottlenecks of the sequential implementation so these line segments are parallelized. Further discussion of these line segments are below:
  - **Line 22-48(Interior Pixels):** In order to parallelize this segment "#pragma omp parallel for" directive is used. The options available for this pragma are default, share, private, firstprivate and lastprivate. There also may be more advanced options for this pragma but the listed ones are enough for many applications. For this segment, I used private, shared and default options. Also, since the value of hist[index] must be updated in a mutually exclusive manner to eliminate race conditions between threads working in this parallel region, "#pragma omp atomic" directive is used. Instead of atomic, a critical region may be used to ensure atomic increment operation by using "#pragma omp critical" directive.
  - **Line 53-75, 77-99, 101-123, 125-147(Border Pixels):** In like manner with the analysis for Interior pixels, "#pragma omp parallel for" and "#pragma omp atomic" directives were used for the same use cases.
  - **Line 152-236(Edge Pixels):** The same operation is done for the 4 edge pixels, the pixels are processed in a sequential manner, similar to an unrolled loop.

Then, distance function(Line 239-259) is parallelized. There is a for loop for all elements of the input arrays, the loop is parallelized using "#pragma omp parallel for" directive and with private, shared and default clauses. In this for loop, distance variable must be updated mutually exclusively by threads so the updates to distance variable is done in a critical region. Critical region is declared unnamed. Also, instead of critical region, a reduction on the running variable may also be implemented as an alternative to the critical region.

Then, find_closest function is parallelized. The for loop for processing training set images is processed

since it is a bottleneck to the sequential implementation. The for loop is parallelized using "pragma omp parallel for" directive with private, shared and default clauses. Also, since minimum_distance variable must be updated mutually exclusively to eliminate race conditions, the updates to min_dist variable are surrounded with an unnamed critical region without any clauses.

Finally, some parts of the main program is parallelized. The allocation of the 2D matrices to hold histograms is done in a for loop. This fpor loop is parallelized using "pragma omp parallel for" directive with shared, private and default clauses since the operation done in the loop is mamory allocation. Then, the creation of the histograms for training images and test images is critical for decreasing the program running time. The for loops for generating the histograms are parallelized using "#pragma omp parallel for" directives with private, shared and default clauses.

3 Results

3.1 Accuracy vs K

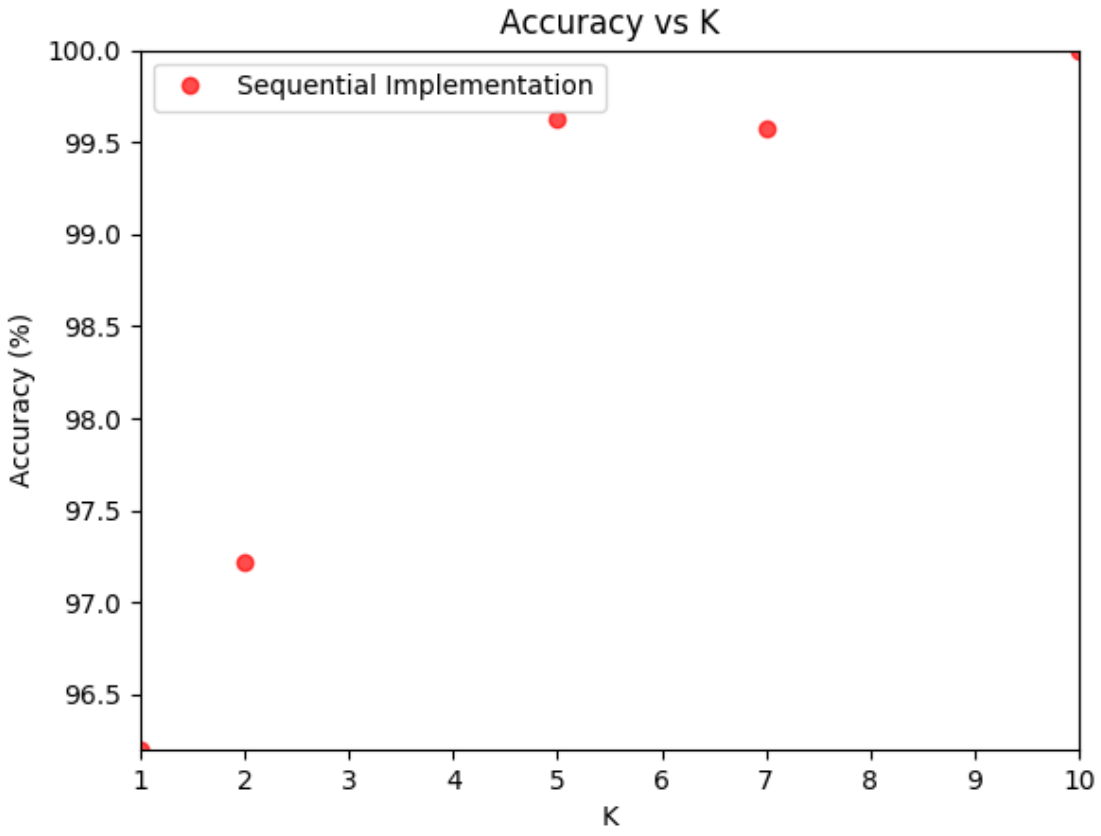

Figure 1: Accuracy vs. K value for sequential implementation

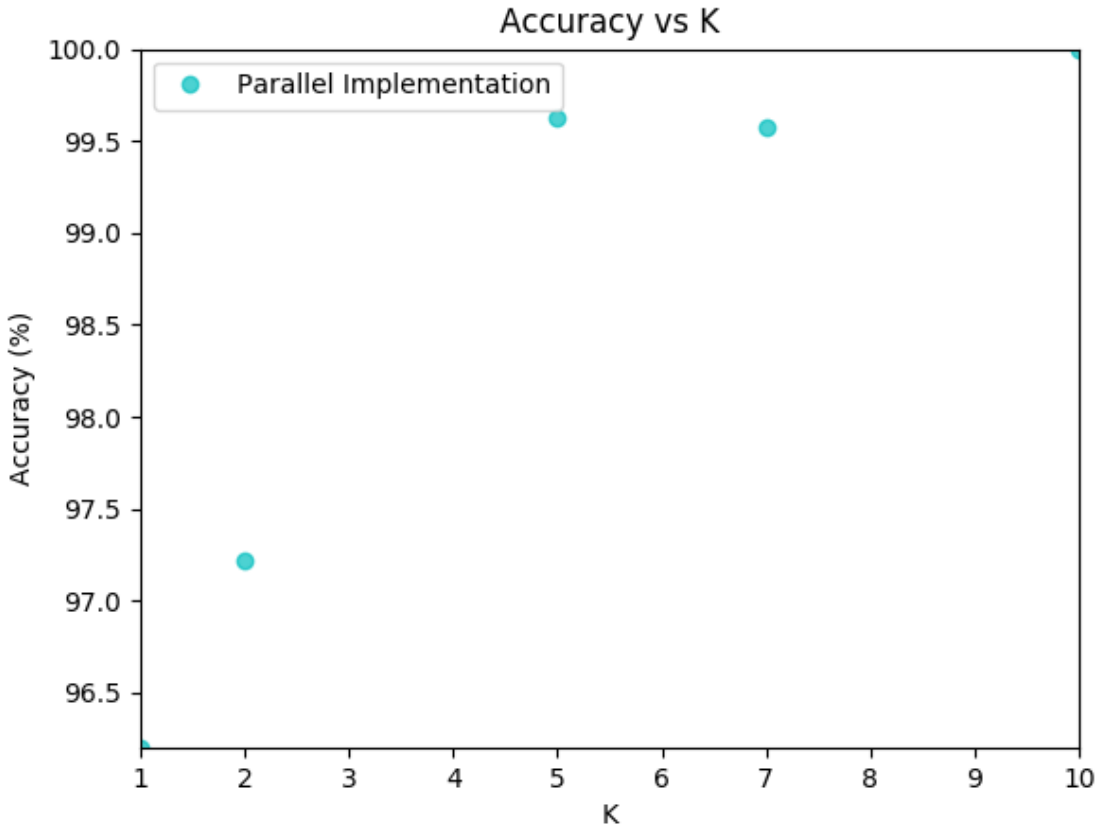

Figure 2: Accuracy vs. K value for parallel implementation

3.1.1 Discussion

The parallel and sequential implementation must not differ in terms of accuracy for same K value since this project is based on a static problem and the results are not time varying. We see that the results of parallel and sequential execution are same. Naturally, we see the accuracy is increasing with K value. This is due to increased number of training examples to base the prediction of test examples. However, the accuracy of K = 7 is less than K = 5. This is due to presence of outliers for each person's images. Mainly, the 20 images belonging to a person must have similar characteristics from a perspective but it seems that there exists an outlier in the images of a person.

3.2 Execution Time vs Number of Threads

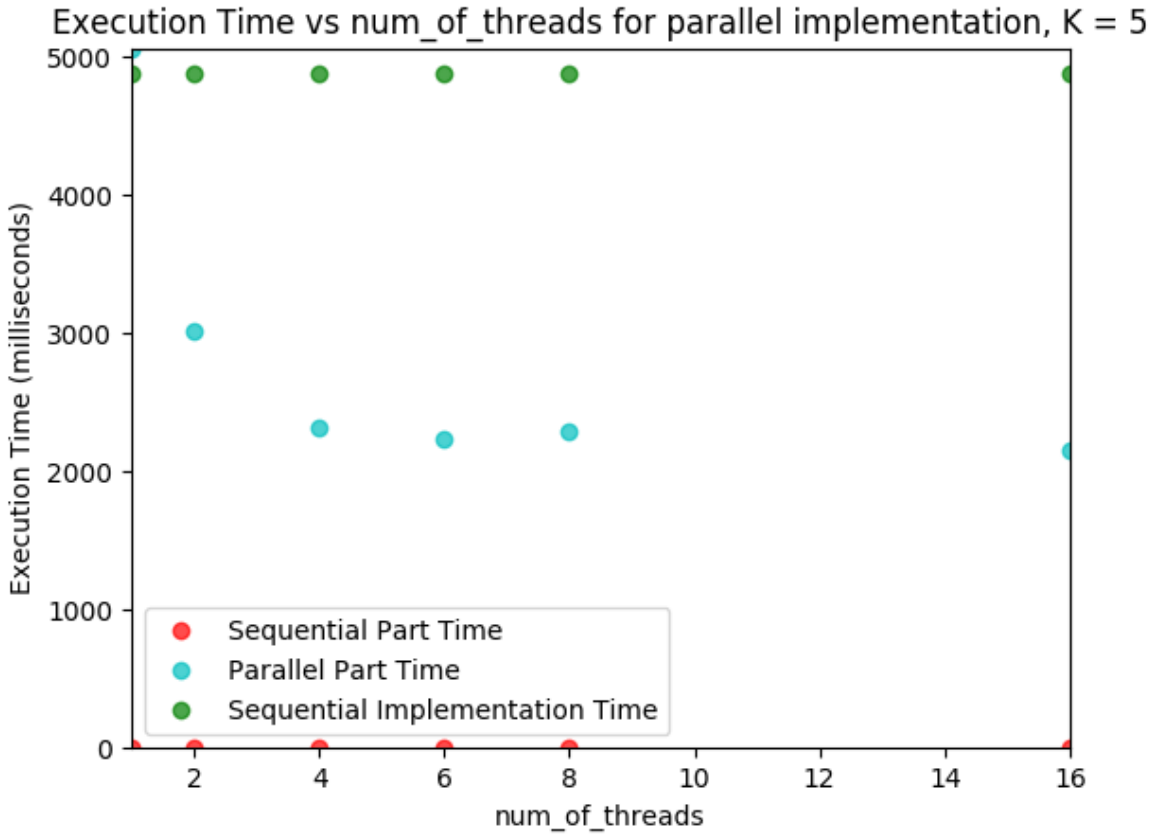

Figure 3: Execution time vs num_of_threads for parallel implementation. K is set to 5.

3.2.1 Discussion

As seen from the plot, as the number of threads increase linearly, parallel running times decrease in exponential manner with diminishing returns. After num_of_threads is increased beyond 4, the threads can not be utilized as much. The utilization of threads is dependent to the value of computation per thread. We can infer that adding more threads not always decrease runtime of a program. The optimal number of threads is dependent to the amount of paralelism and the nature of the problem.

We see that sequential part time is always 0. This is because printing is done in the scope of parallel time computation even though printing does not running time drastically. The sequential part only consists of minor memory allocations in the scale of $O(1)$. As a design decision, I put the for loop for prediction and result printing to parallel timing region. This is because find_closest function has a parallel implementation. I also tried adding the mentioned for loop to the sequential timing region. Then, the sequential time was non 0 and increasing with the num_of_threads. I tried to put printing in the sequential part but sequential part still remained 0. Refer to the lbp_omp.c for parallel timing regions and sequential timing regions.

This plot also contains the results of the sequential implementation. The running time of the sequential implementation does not change with respect to the num_of_thread. The difference between green dots and blue dots for specific K values gives the speedup in terms of milliseconds. We see that speedup achieves its maximum value(2721ms) in num_of_threads = 16 case.

3.3 Execution Time vs K

3.3.1 Parallel Implementation

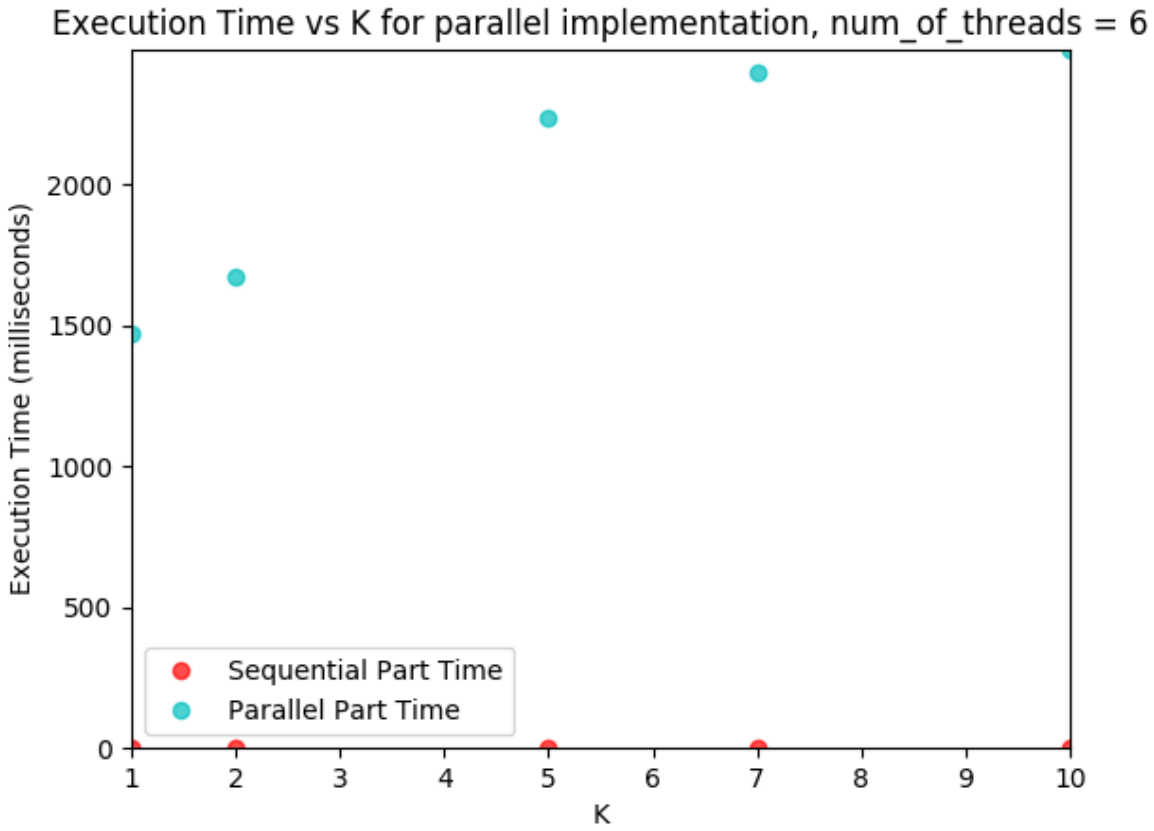

Figure 4: Execution time vs K for parallel implementation. num_of_threads is set to 6.

3.3.2 Discussion

From the plot, we can infer that the running time of the parallel implementation also depends on the value of k. As the value of k increases, parallel running time increases since computation per thread increases.

3.3.3 Sequential Implementation

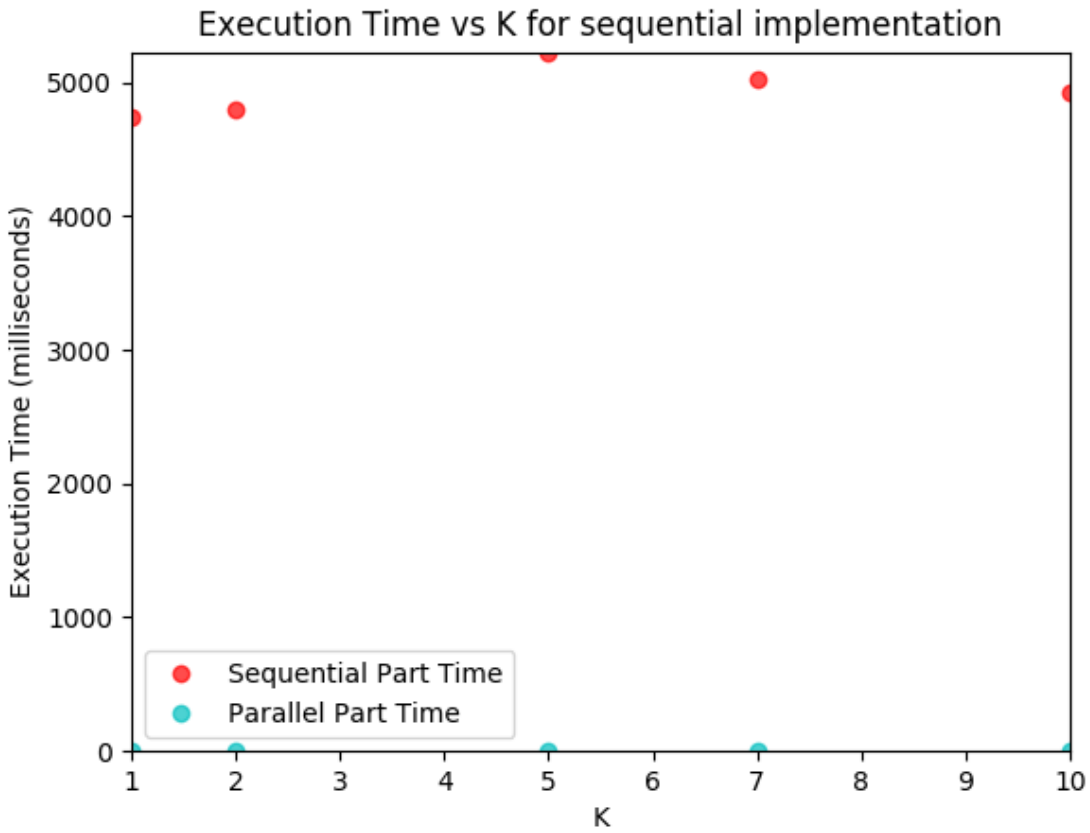

Figure 5: Execution time vs K for sequential implementation.

3.3.4 Discussion

From the plot we can infer that sequential time still increases with respect to the K value but the increase is not as drastic as parallel implementation. This is because as K increases training set size increases but test set size decreases. Hence the total number of images stay the same. Histograms of both training set and test set are computed. For lower K values, larger number of test set examples are compared(computed distance between a training set image and test set image) with smaller number of training set examples. For higher K values, smaller number of test set examples are compared with larger number of training set examples. Hence in both cases total computation is similar to each other. The changes in the results for different K values is due to the current state of the computer where the code is executed on. On the other hand parallel runtime is always 0 because sequential implementation does not have a parallel part.

3.4 Profiling Outputs Using gprof

The following analysis is based on the previously generated prof_sequential.txt and prof_omp.txt files.

3.4.1 Sequential Implementation

By investigating the flat profile for sequential program, we see that create_histogram function contributes 70% of the total runtime, distance function contributes 14.93% and read_pgm_file function also contributes 14.93% percent of the total running time. The other functions also contribute to the total runtime as well but they are not influential as the mentioned functions. This %time analysis shows that the main bottleneck of the sequential implementation are calls to create_histogram, distance and

read_pgm_file functions. Therefore, the flat profile shows that it would be meaningful to parallelize these portions of the implementation to make this code faster. Note that distance function generates its portion of running time through the calls from find_closest to it(this will be more apparent in the call graph analysis). Also by looking to the self field, we see that create_histogram takes 0.47 seconds(the most time consuming), read_pgm_file and distance functions take 0.10 seconds each. Other functions also consume time it appears that the most influential(bottleneck) functions are the mentioned. The calls field shows that distance function is called 32400 times, create_histogram and read_pgm_file functions are called 360 times each. In like manner, by looking at the calls and self field of the flat profile, we see that it meaningful to parallelize create_histogram and distance functions. Note that read_pgm_file function is given in the util file so it is not attempted to be parallelized.

By investigating Call Graph for the sequential program, we see that main and its descendants contribute 100% to the total running time. This is meaningful since the main function is run in C by default. create_histogram function and its descendants contribute 70% to the total running time. Since the percentage of contribution is same for create_histogram function in both call graph and flat profile, create histogram should not do a significant job by calling another declared function. This is indeed the case with the create_histogram function. A similar inference can be done for distance and read_pgm_file functions. On the other hand, 14.9% contribution from find_closest function and its distance is due to the distance function calls in find_closest function. In fact, the distance function is only called in find_closest function.

3.4.2 Parallel Implementation

By investigating the flat profile and call graph together, we see that compared to the sequential program, there is a different result in terms of time proportional division to the functions. According to the profiling outputs for different threads for K = 10, main function and read_pgm_file functions contribute most to the total time. But this should not be the case. After a research on using gprof with multithreaded programs, I found out that gprof only profiles the main thread but not the threads created by OpenMP. Therefore, we can not infer a correct conclusion from these results. For example, create_histogram function did the most job in sequential implementation however, it is listed as 0 in all of the parallel profiling results. The two explanations for this results is that either the sequential implementation is parallelized in a good proportion or the main thread only runs read_pgm_file function. In conclusion, gprof is not suitable for multi-threaded program profiling and a more sophisticated profiler should be used.