CS 426 Parallel Computing

Project 3 Report

Alp Ege Baştürk

21501267

01.05.2019

Implementation Details: Sequential

- **Main**

The main program starts by allocating a matrix for histograms, then continues with 2 nested for loops. They iterate from 0 to sample_count_per_person, which is 20, for each person, making 360 iterations in total. These loops iterate over the matrix, allocate row, then call read_pgm_file(…) and create_histogram(…). This is the main functionality of the program.

After this loop finishes execution, main runs a similar loop for tests. However inner loop starts from k instead of 0 to only consider test results. find_closest(...) function is called people count times test image count in these nested loops.

- **Create Histogram**

This function has 2 for loops nested to iterate the image to create an histogram. Both loop start from 1 instead of 0 to zero pad the edges, and end one less than the height or width. apply_filter_on_pixel() function is called for each pixel.

- **Apply Filter On Pixel**

This is a simple function which compares center with neighbors and returns the decimal. It was written using bitwise operators on an u_int8_t decimal to make it optimal as possible.

- **Distance**

Returns distance of two vectors as described.

- **Find Closest**

Uses two for loops nested in order to iterate over saved histogram matrices' training part. Calls distance function on current matrix location and the test histogram and checks if it is the closest, updates if so.

Description of Gprof's Profiling Outputs

% time: the percentage of the total running time of the program used by this function.

Cumulative seconds: a running sum of the number of seconds accounted for by this function and those listed above it.

Self seconds: the number of seconds accounted for by this function alone. This is the major sort for this listing

calls: the number of times this function was invoked, if this function is profiled, else blank.

Self ms/call: the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call: the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

Name: the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

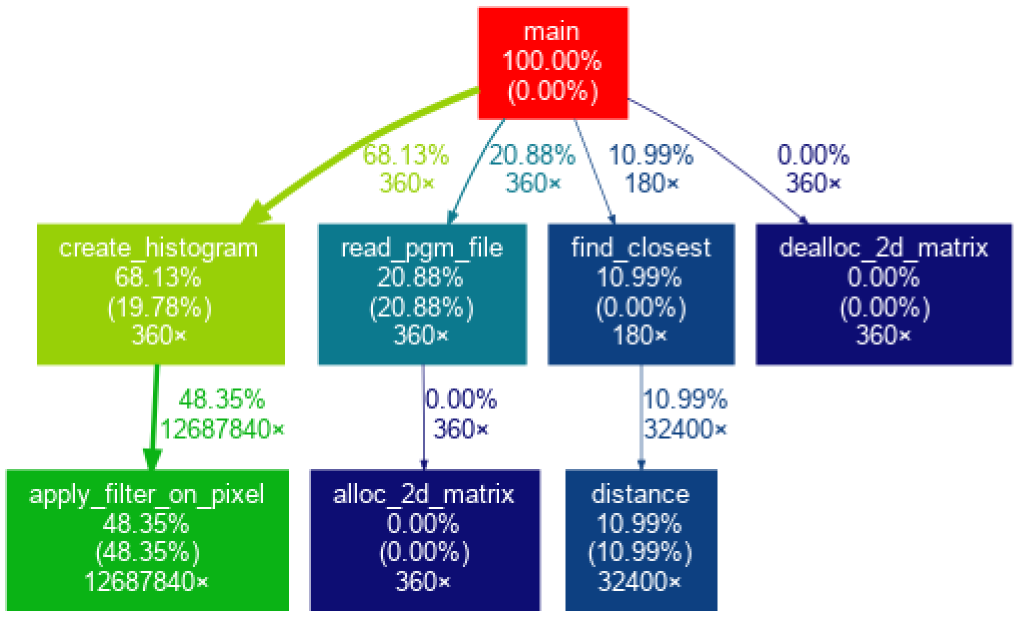
Gprof Analysis: Sequential

Following results are generated using k = 10 from the sequential implementation. Graph was generated using python code with GPL 3 license which converts various profiler outputs to visuals by jrfonseca [1].

The sequential execution took 2675.822 ms in total, and paralel execution took 1293.670544 on 4 threads. All results were obtained by 2.6 GHz 4 Core Intel CPU Machine. But the Turbo Boost was disabled and CPU frequency was set to 1.8 GHz in order to see more meaningful results since it was executing too fast with high frequencies.

Optimizations were disabled at the compile time to get meaningful results with the gprof.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
48.4	0.44	0.44	12687840	0	0	apply_filter_on_pixel
20.9	0.63	0.19	360	0.53	0.53	read_pgm_file
19.8	0.81	0.18	360	0.5	1.72	create_histogram
11	0.91	0.1	32400	0	0	distance
0	0.91	0	360	0	0	alloc_2d_matrix
0	0.91	0	360	0	0	dealloc_2d_matrix
0	0.91	0	180	0	0.56	find_closest



In my implementation, create histogram is called from main to create histograms for all samples, which makes 18 x 20 = 360 calls. This function iterates on the original image and calls apply_filter_on_pixel() function to make calculations with the neighbor cells. It can be seen that most of the time was spent in creating the histogram, more specifically in the apply_filter_on_pixel() function. This is the main computation function of this program.

In addition to the main computation, finding closest to a test image and file read took the remaining time spent. As it can be seen from the table, apply_filter_on_pixel took 0.44 self seconds and distance took 0.19 seconds which makes these, the top two bottlenecks.

Most of the time was spent in histogram creation and closest computation, Therefore initial optimizations were focused on these two points.

Implementation Details: Changes in Parallel

There are 4 debug flags which enable parallelization on 4 different parts of the program. This helps with analyzing which parts helped the performance. They are defined in the util.h as

```
#define DEBUG_OPT_HIST 1
#define DEBUG_OPT_MAIN 1
#define DEBUG_OPT_DIST 0
#define DEBUG_OPT_TEST 1
```

- **Main**

As explained before, create histogram and testing takes most of the time in the sequential implementation. Also time spent for reading pgm files is close to finding closest. Thefore, the first nested loop in the main function around line 173 is a good choice for parallelization. This both parallelizes create histogram and read_pgm. Parallelization was chosen only on the outer loop because dividing it in rows was enough to use all available threads, but also read files and relevant create histogram calls were assigned to same threads.

Separating matrix allocation, file read, and create histogram calls to 3 loops where 2 for loops are nested, was implemented for experimentation. In this case, #pragma omp parallel for collapse(2) was used but it did perform worse, which might be due to overhead, therefore single nested implementation was chosen.

In addition, create histogram has a parallelized for loop, which might increase scalability with more processors since it can divide work more for each create histogram call. This was not the case for the test machine with 4 cores, 8 threads with Hyper Threading.

Also the nested for loop which calls find closest are also parallelized with `#pragma omp parallel for collapse(2)`. This divides matrix iteration to threads. Collapse is a clause which makes OpenMP library convert perfectly nested loops to single loops and share work accordingly. This was used as an experiment here. Version without collapse(2) have given the similar results. Collapse version was chosen for diversity since both were similar.

```
    #if DEBUG_OPT_TEST
#pragma omp atomic
#endif
correct_count++;
    #if DEBUG_OPT_TEST
#pragma omp atomic
#endif
incorrect_count++;
```

atomic calls were necessary in order to prevent race conditions on shared variables.

- **Create Histogram**

The nested loop in create histogram, around line 49, was chosen to be parallelized. This is the loop where apply_filter_on_pixel() is called which takes most of the program time. Only the outer loop was parallelized, and collapse(2) was commented out because it wasn't providing any improvement since parallelization at the main function was already dividing work enough to fill all available threads. Collapse approach might be chosen if there are many threads available to divide work even more.

```
    #pragma omp atomic
((int*)hist)[tmp]++;
```

This atomic call was used in order to prevent wrong calculations on the histogram.

- **Apply Filter On Pixel**

This is a simple function with optimized bitwise operations, therefore it wasn't parallelized.

- **Distance**

Calculations inside this function are not parallelized because of the reasons given for create histogram. Main function already parallelizes the nested loop for testing, which also uses collapse(2). This is enough to fill all available threads. Also find closest has a similar optimization, which was disabled with `DEBUG_OPT_DIST 0` since it wasn't providing any improvement because of the lack of hardware. Thus parallelizing distance wouldn't help since the work was divided enough up to that point.

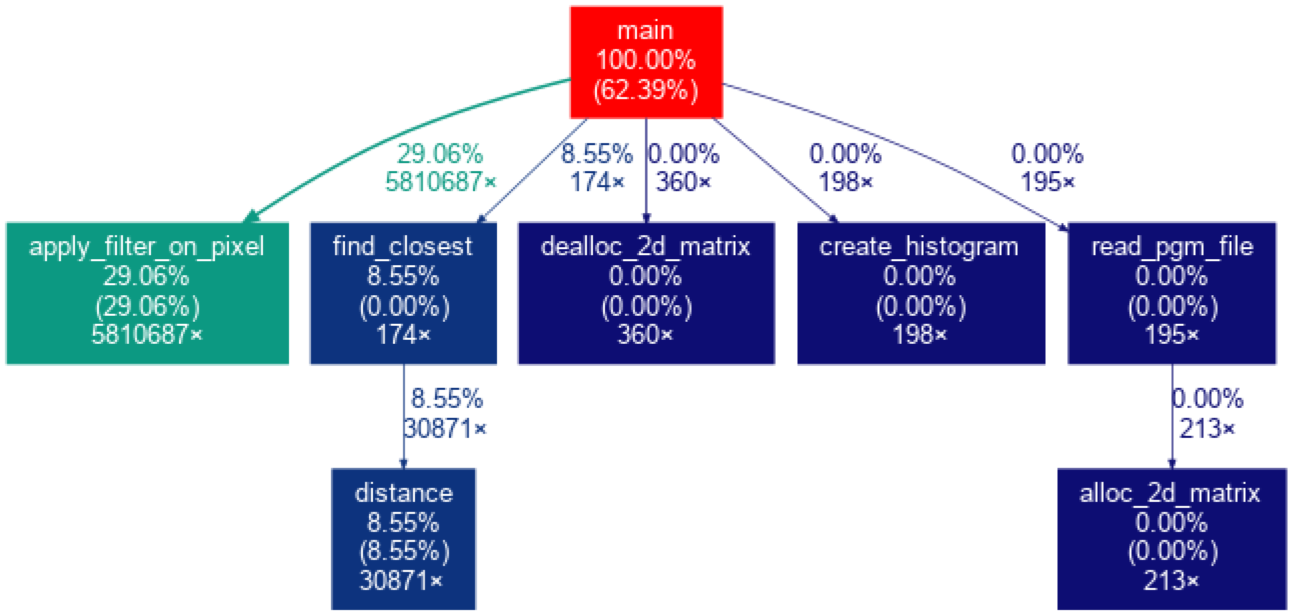
- **Find Closest**

Nested for loops around line 121 are parallelized with `#pragma omp parallel for collapse(2)` but this was disabled during testing since it didn't provide any benefit due to lack of hardware since caller main already paralellizes the nested loop.

Gprof Analysis: Parallel

Following results are generated using k = 10 with 4 threads from the parallel implementation.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
62.44	0.73	0.73				main
29.08	1.07	0.34	5810687	0.06	0.06	apply_filter_on_pixel
8.55	1.17	0.1	30871	3.24	3.24	distance
0	1.17	0	360	0	0	dealloc_2d_matrix
0	1.17	0	213	0	0	alloc_2d_matrix
0	1.17	0	198	0	0	create_histogram
0	1.17	0	195	0	0	read_pgm_file
0	1.17	0	174	0	575.18	find_closest



The results are obtained as explained for the sequential analysis. As it can be seen from the call stack graph, OpenMP optimized some function calls. For example apply_filter_on_pixel() looks it was embedded in to the main function instead of being called from create histogram. This may be due to OpenMP detecting parallelization on the caller, main function's nested for loop, and parallelization at the callee, create_histogram's nested for loop, and regenerate all of these in the main call.

In the OpenMP case, call stack looks distorted, which may be due to effects of the OpenMP library optimizations. It can be seen that apply_filter_on_pixel was reduced from 48.35% to 29.06%. This is probably the most useful and meaningful result obtained from the gprof for this implementation. Also time for find closest was reduced by a small amount.

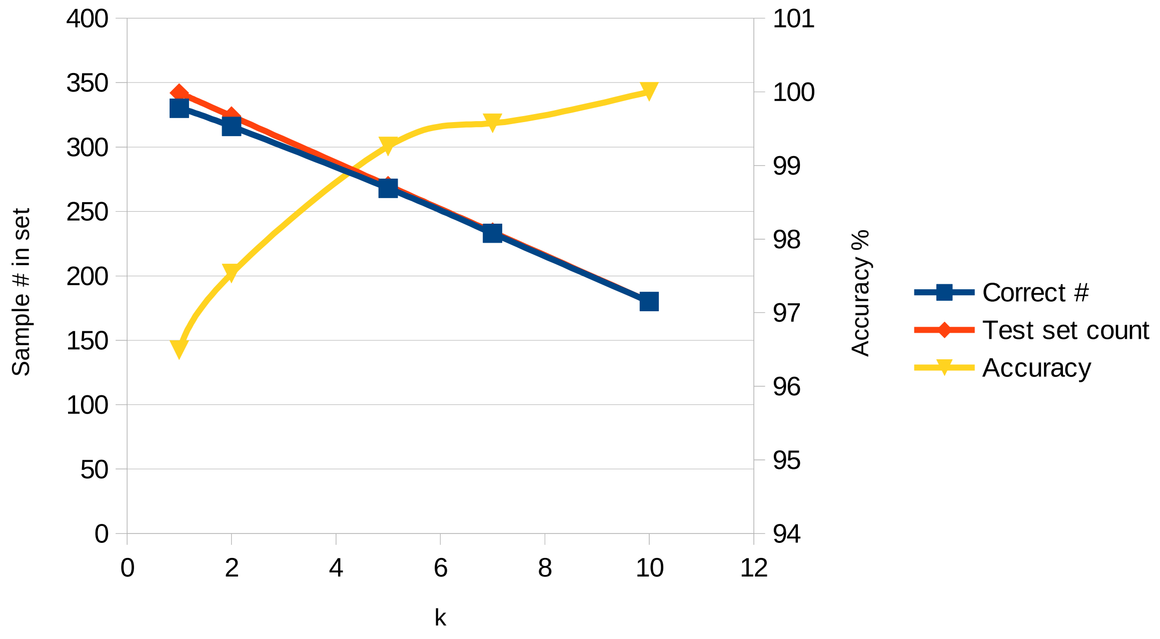
In this gprof result, read_pgm_file takes no time, which is wrong. This might be due to read_pgm_file being inside the parallelized for loop, and calls are optimized so gprof failed to profile it. Gprof profiles code compiled with -pg option so if external library calls are not compiled accordingly, then the profiler might give such meaningless results.

Results & Plots

Tests machine had Intel i7-4720HQ CPU with 4 cores at 2.6 GHz. However Turbo Boost was disabled and CPU frequency was set to 1.8 GHz using cpu_freq to obtain meaningful results. Sequential time measurements for the parallel implementation are ignored in the discussions since they were only ~20 ms and it was mostly the deallocation and cleanup part.

Accuracy Results Using Sequential Implementation

k	Correct #	Test set count	Accuracy
1	330	342	96.49122807
2	316	324	97.530864198
5	268	270	99.259259259
7	233	234	99.572649573
10	180	180	100

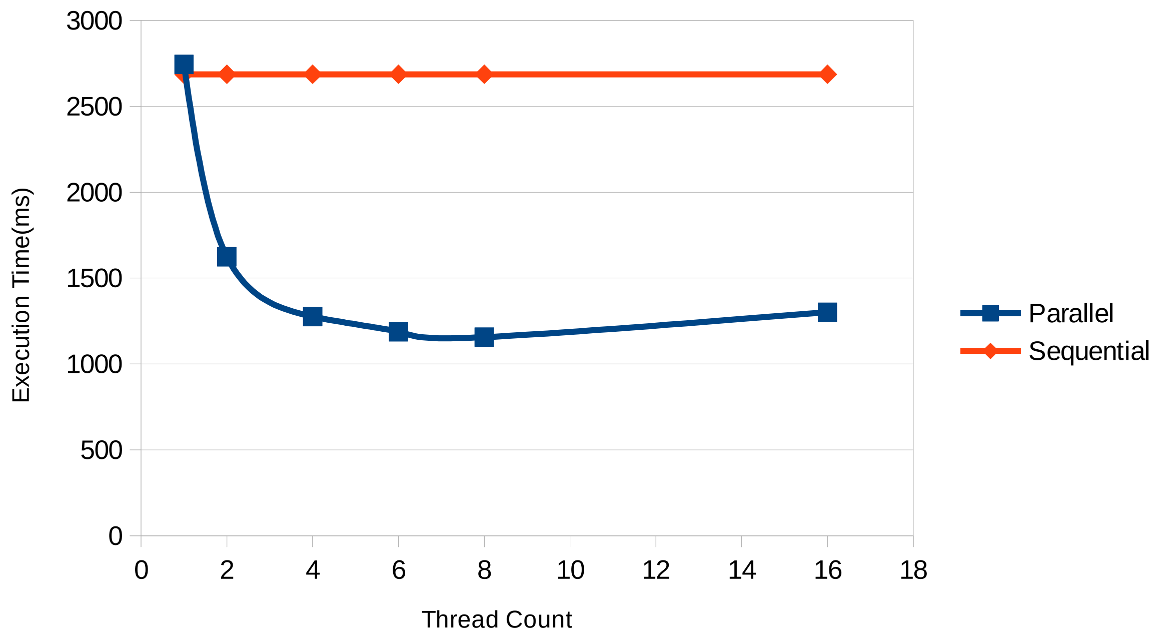


As it can be seen, accuracy increases while the training set increases. Accuracy was high for k=1 and it went up to 100% for k=10.

Timing Results Using Parallel Implementation

Following results are obtained for k = 10 with fixed k, variable thread count.

Thread #	Parallel	Sequential
1	2744.459934	2686.814
2	1624.543135	2686.814
4	1276.372035	2686.814
6	1187.538449	2686.814
8	1156.071107	2686.814
16	1301.279995	2686.814



As it can be seen, parallel execution improves the results greatly, especially reduced CPU frequency shows this difference better. Parallel implementation with single thread was worse than the sequential implementation, this might be due to additional overhead of the libraries. The results improved most for the 2 threads, but returns were diminishing with the increasing thread counts. Test machine had 4 cores, and 8 threads with HT, therefore improvement was expected up to 4 threads. It is also consistent with the results of previous MPI projects. The curve becomes flat around 8 threads which is different from the previous MPI results. It seems Hyper Threading helped for this implementation. At 16 threads performance is degraded, this might be due to additional overhead for threads, which can’t get any improvement from parallelization since no physical resources are left.

References

[1] jrfonseca, “gprof2dot”. *GitHub*. Available: https://github.com/jrfonseca/gprof2dot