CS426 - Parallel Computing
Project 03 - Report
Aldo Tali
ID: 21500097
Date: 01/05/2019

Part 1) Detailed description of gprof's profiling outputs

The Gprof command will profile the run time of the program based on the function calls that are made within it. Below I have attached the two profiling outputs for the sequential and the parallel parts of the programs. As seen in the outputs a normal gprof gives accumulative time for the running time of each function routine in sorted order. In this way it is easy to identify the bottlenecks of the program and which points need to be tackled for parallelization. In the second picture the report depicts how gprof also shows the information of each call in a graph manner, that is the subroutines of each function show the amount of time needed to execute each routine in the function. All this graph view is represented in terms of children of each function.
As seen from my gprof files, the most time consuming business in the sequentials part is the reading files, creating histograms, allocating matrices and computing distances. Since the memory read operations are I/O bound and they are dominating the run time and the gprof data will only show relative information, it is not possible to see the time improvements for such type of operations. In order to check the overall execution I used the build in **time ./executable** functionality in ubuntu so as to give comparable results and to build the visual graphs.

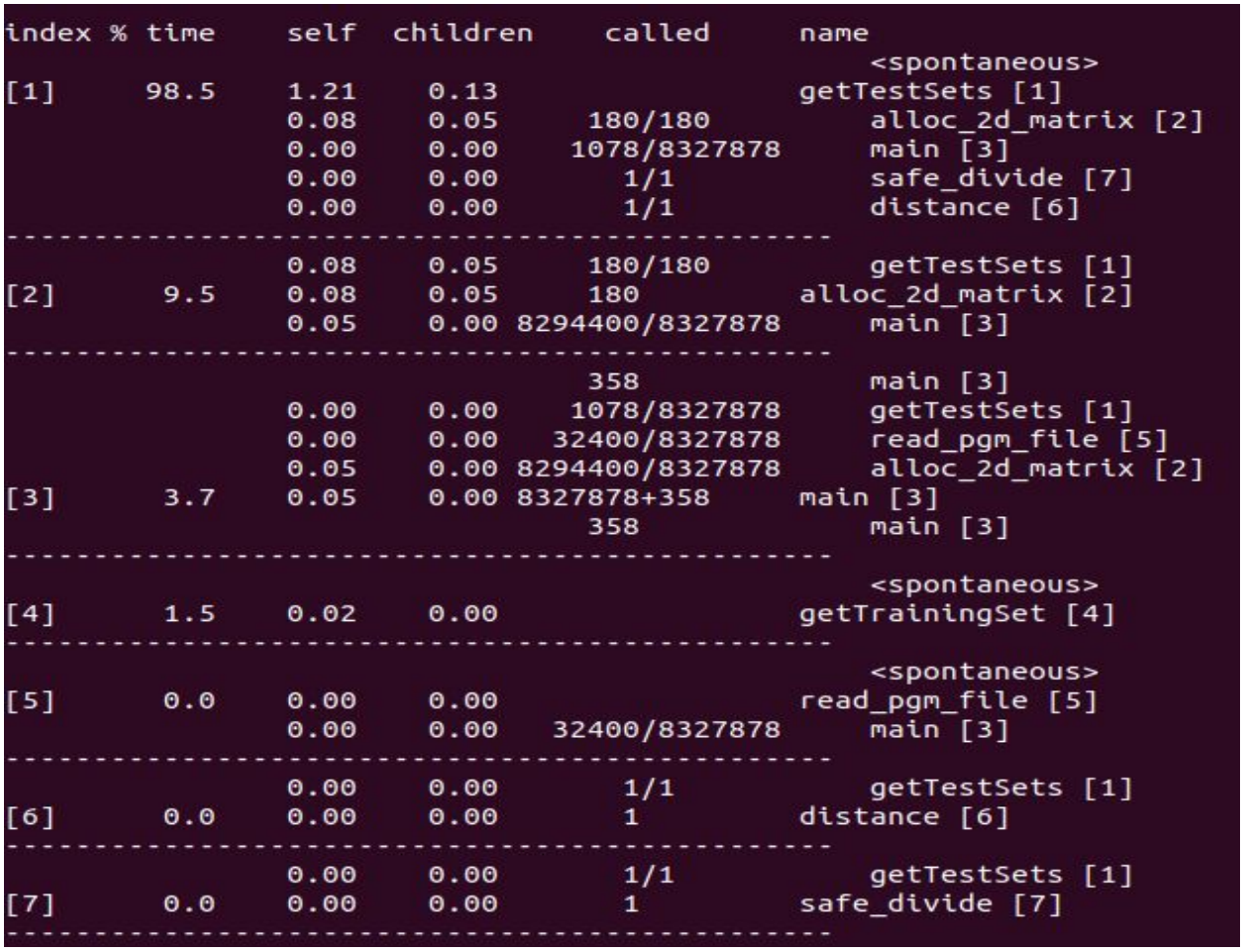
Figure 1.0 Sequential

Flat profile:

Each sample counts as 0.01 seconds.

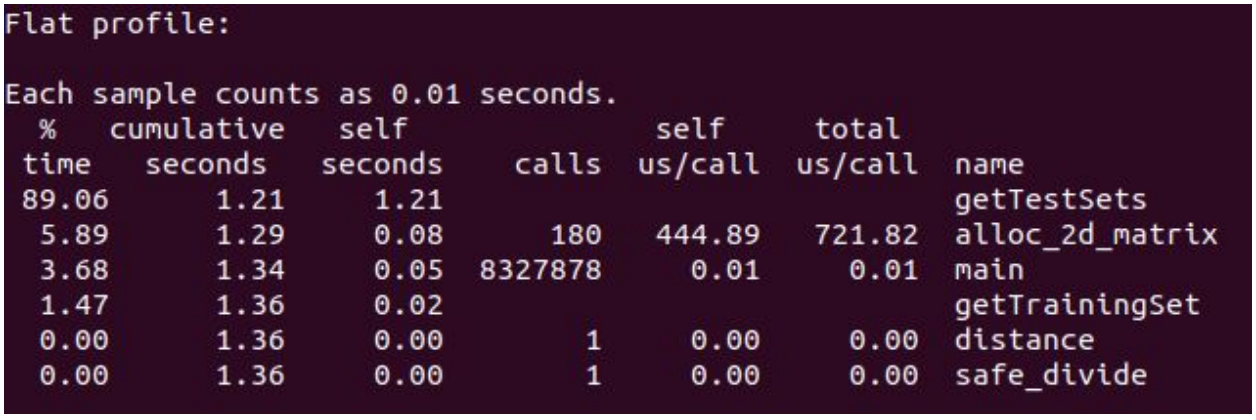
% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
21.82	0.22	0.22				read_pgm_file
18.84	0.41	0.19				create_histogram
17.35	0.59	0.18				alloc_2d_matrix
15.87	0.75	0.16				distance
10.91	0.86	0.11	1080	0.10	0.10	main
9.42	0.95	0.10				dealloc_2d_matrix
2.98	0.98	0.03	32400	0.00	0.00	find_closest
1.98	1.00	0.02				safe_divide
0.99	1.01	0.01	2	5.01	60.10	getTestSets
0.00	1.01	0.00	180	0.00	0.00	prettyPrintArray

Figure 1.1 Sequential



```
index % time    self  children    called     name
                                               <spontaneous>
[1]     98.5    1.21    0.13                   getTestSets [1]
                0.08    0.05     180/180           alloc_2d_matrix [2]
                0.00    0.00    1078/8327878       main [3]
                0.00    0.00       1/1             safe_divide [7]
                0.00    0.00       1/1             distance [6]
-----------------------------------------------
                0.08    0.05     180/180           getTestSets [1]
[2]      9.5    0.08    0.05     180             alloc_2d_matrix [2]
                0.05    0.00 8294400/8327878       main [3]
-----------------------------------------------
                                 358               main [3]
                0.00    0.00    1078/8327878       getTestSets [1]
                0.00    0.00   32400/8327878       read_pgm_file [5]
                0.05    0.00 8294400/8327878       alloc_2d_matrix [2]
[3]      3.7    0.05    0.00 8327878+358        main [3]
                                 358               main [3]
-----------------------------------------------
                                                   <spontaneous>
[4]      1.5    0.02    0.00                   getTrainingSet [4]
-----------------------------------------------
                                                   <spontaneous>
[5]      0.0    0.00    0.00                   read_pgm_file [5]
                0.00    0.00   32400/8327878       main [3]
-----------------------------------------------
                0.00    0.00       1/1             getTestSets [1]
[6]      0.0    0.00    0.00       1           distance [6]
-----------------------------------------------
                0.00    0.00       1/1             getTestSets [1]
[7]      0.0    0.00    0.00       1           safe_divide [7]
-----------------------------------------------
```

Figure 2.0 Parallel



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 89.06      1.21     1.21                             getTestSets
  5.89      1.29     0.08      180   444.89   721.82  alloc_2d_matrix
  3.68      1.34     0.05  8327878     0.01     0.01  main
  1.47      1.36     0.02                             getTrainingSet
  0.00      1.36     0.00        1     0.00     0.00  distance
  0.00      1.36     0.00        1     0.00     0.00  safe_divide
```

Part 2) Detailed description of your implementation details

In this implementation there are three main places where pragma parallelization were used. The first two consist of the reading of the training and the real sets of images of each person. The last one is the create_histogram function. In the process of implementing pragmas using a reduction pragma on the distance function was also considered however, since the level of multithreading overpasses the hardware availability of my device the improvement is not eminent. There are four main pragmas that the parallel implementation uses in this submission. These pragmas are `#pragma omp parallel, #pragma omp parallel for , #pragma omp atomic` and `#pragma omp barrier`. Of course for the measuring of the parallel time the omp_get_wtime(); function was used.Below, the report discusses each use case of the pragmas used as part of the implementation.

Part 2.1 Parallelizing Training Set Read

Code Piece

```
#pragma omp parallel private (i,j,tid,p)
{
     double start,end;
     …. Extra code here …
      #pragma omp parallel for
     for (i = tid; i < noPeople; i+= nthreads) {
      }
}
```

In the above representation a parallel for pragma was used to allocate the training sets in the code. The function takes i, j, p as private loop variables so as not to influence of each thread. In addition each thread id the tid is also declared as private. In order to ensure that each thread accesses only the files that pertain to the thread, tid is used as loop variable increment. In here the pragma for loop for each thread is running only in multiples of tid locations. There are not too many alternatives for this kind on operation, however we could have specified a scheduling system such as a dynamic scheduling. Nonetheless considering that the files are exactly the same size and the run times are almost the same, then it was obvious to use the defaults. Another option to use here is the no wait sine the operations are not interdependent by some other thread business.

Part 2.2 Parallelizing the Test cases

Code Piece

```
#pragma omp parallel private (i,j,tid,p)
{
     double start,end;
     …. Extra code here …
      #pragma omp parallel for

      for (i = tid; i < noPeople; i+= nthreads) {
            testHistograms[i] = (int **)malloc(sizeof(int*) * 20-k);

      }
}
```

In the above representation a parallel for pragma was used to allocate the training sets in the code. The function takes i, j, p as private loop variables so as not to influence of each thread. In addition each thread id the tid is also declared as private. In order to ensure that each thread accesses only the files that pertain to the thread, tid is used as loop variable increment. In here the pragma for loop for each thread is running only in multiples of tid locations. There are not too many alternatives for this kind on operation, however we could have specified a scheduling system such as a dynamic scheduling. Nonetheless considering that the files are exactly the same size and the run times are almost the same, then it was obvious to use the defaults. Another option to use here is the no wait sine the operations are not interdependent by some other thread business.

Part 2.3 Parallelizing the Create Histogram

Code Piece

```
#pragma omp parallel private (number,power,tid,j)
 {
       nthreads = omp_get_num_threads();
       tid = omp_get_thread_num();
       j = 0; number = 0; power = 7;

       #pragma omp parallel for
       for (i = tid; i < num_rows; i+=nthreads){
```

```
      …. Code extra goes here ….
      }
      #pragma omp atomic
           hist[number] = hist[number] + 1;
}
```

In the above representation a parallel for pragma was used to access the decimal number computation of the image pixels.In here unlike the previous statements an atomic statement is used in order to make the update on the respective entry without conflicts. The other possibilities to the atomic pragma would be the critical of the barrier pragmas. However in this case if we were to use any of those the concurrency level would be lowered since at each update in order to ensure consistency all other threads except for tid would have to wait for the barrier lock to be released.

Part 2.4 Parallelizing Closest Distance

Code Piece

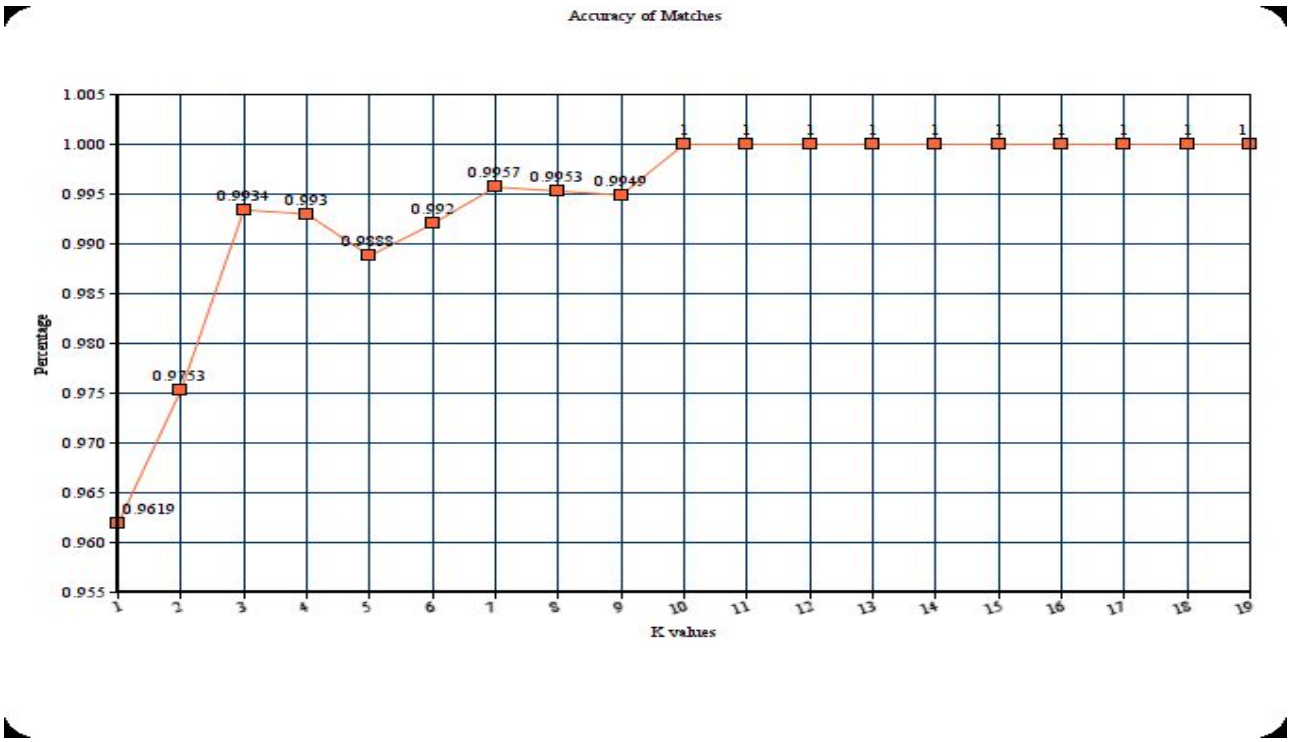
```
#pragma omp parallel private (i,tid,j)
 {
       nthreads = omp_get_num_threads();
       tid = omp_get_thread_num();


       #pragma omp parallel for reduction(min:min)
       for (i = tid; i < size; i+= nthreads){
            if (min < 0){
                min = (double)
distance(training_set[i][j],test_image,size);
                personId = i;
           } else {
                dist = (double)
distance(training_set[i][j],test_image,size);
```

```
                if (dist < min){
                    min = dist;
                    personId = i;
                }

        }

}
```

In the above representation a reduction for pragma was used to compute the minimum distance. Each thread will compute its local minimum and at the end the global minimum will be stored in min. The other options would include to make min as a private variable and then include a critical section with an explicit barrier. This option was not preferred to to efficiency concerns.

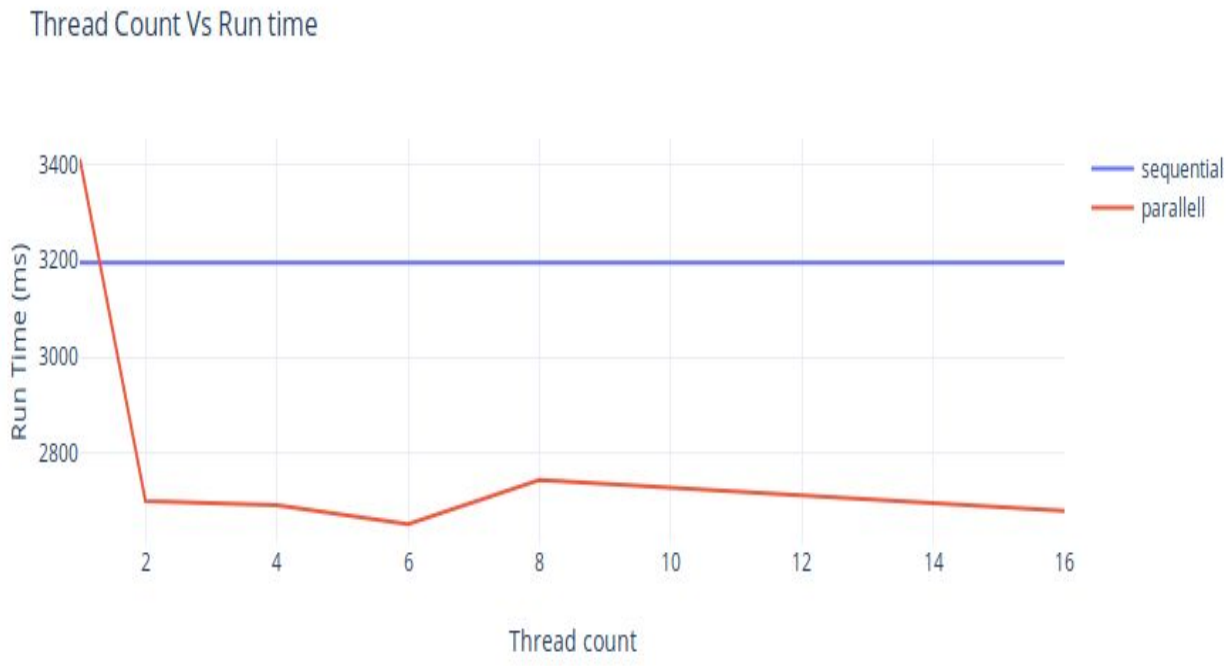
Part 3) Plot for accuracy results.

Below the report presents the graph for the data of different k values to the percentage of correct results. Further discussion follows on Part 5.



Part 4) Plot for execution times with different threads.

Below the report presents the graph for the data of different thread number values to the run time of results. Further discussion follows on Part 5.



Part 5) Discussion of your results.

As seen in the graph in part 4, the accuracy of training sets compared to the k values converges on k >= 10. For all these values there are 0 erroneous results. The lesser k values have at most 13 erroneous results. In the process although it may look counter intuitive that there are hills in the error numbers, however as the training sets increase the probability that the minimum distance is lower than the correct person's id also increases which explains why the hills are present in that graph. For the 5th part graph judging from the gprof results, after parallelizing the reads of the training sets and running the create histogram concurrently the time required to run the set is lowered by 0.67 seconds on average. However for numThreads =1 the actual running time is larger than the sequential part. This is normal since the overheads introduced by the parallelism structs in this case add up to the execution times of the overall program. However as seen in gprof the create_histogram time computation decreased which is reflected in the graph. However the time improvement is not large as such since most of the costly operations such as the file reads are memory I/O bound. As such even with the improvements in the lesser influential functions the bottleneck is the I/O operations and the parallelism benefits are not highly obvious.