**COL380 - Assignment 1 - Profiling**

**Done by: Mohit Sharma (2019CS10372)**

**1) Analysis of gprof of original code:**

The gprof report is generated by compiling all the files with “-pg” flag. And then when we execute the code a file name gmon.out is generated.

To generate the report following command is used:

**gprof classify gmon.out**

For the original code the gprof report can be seen in orgGprof.txt file.

The file shows that the time taken by the code is mainly spend in classify function.

**2) Analysis of valgrind of original code:**

The valgrind report can be generated by compiling the files with “-g” flag.

Then following command is used for execution:

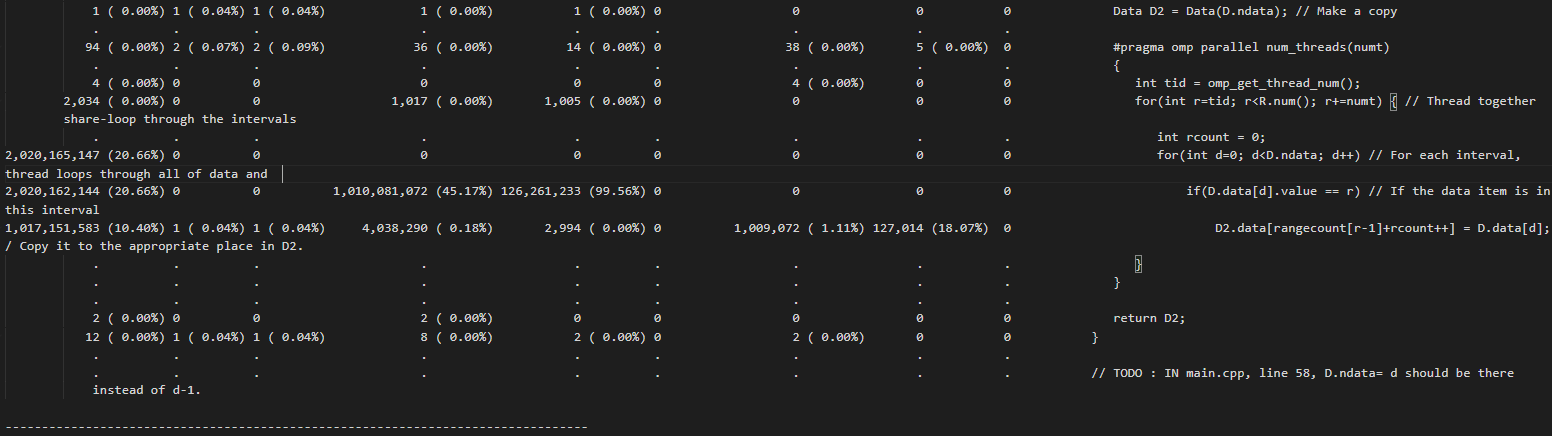
**valgrind --tool=cachegrind --log-file="cachegrind\_res.txt" --cachegrind-out-file="cachegrind\_res.out" ./classify rfile dfile 1009072 4 1**

Now to generate the annotated report use following command:

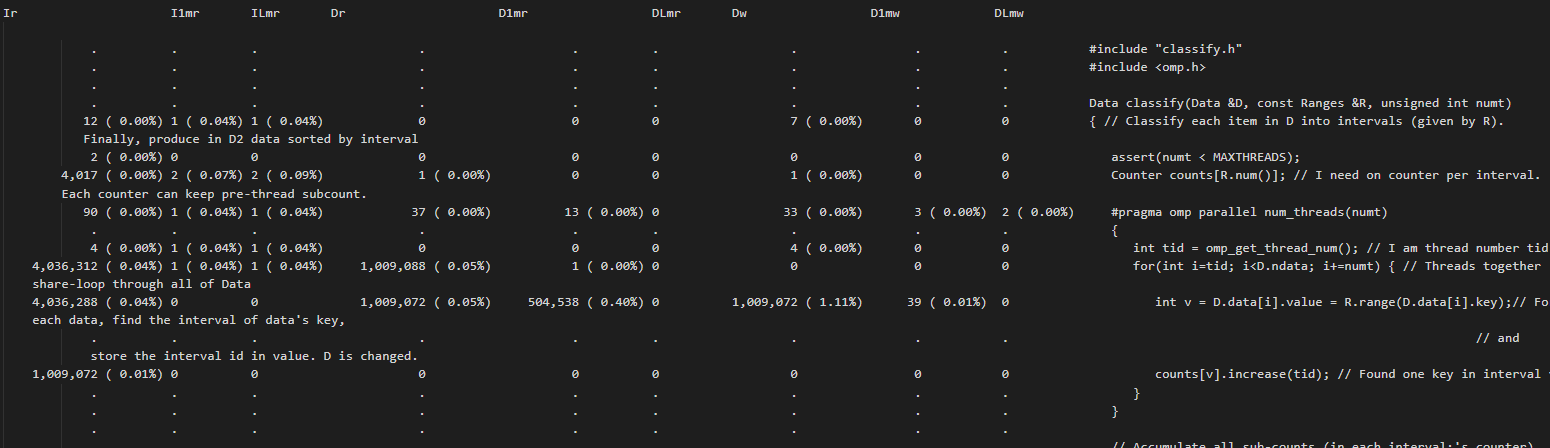
**cg\_annotate cachegrind\_res.out**

The annotated report of original code can be seen in orgCache.txt file.

From the file we can observe that a lot of D1 cache read misses takes place during the read of Data in the second part of the parallel threads usage in classify function (line 42 of classify.cpp). The instruction is used to access Items in array D.



Other major bottleneck in performance can be observed in the D1 cache write miss happening during writing of Items in the array D2(line 49 of classify.cpp). Also, in the first part of parallel thread execution (line 12 of classify.cpp), where the Items in array D are being read, a lot of D1 cache read misses can be observed.



**3) Analysis of gprof of modified code:**

The gprof report of modified code can be found in the modGprof.txt file.

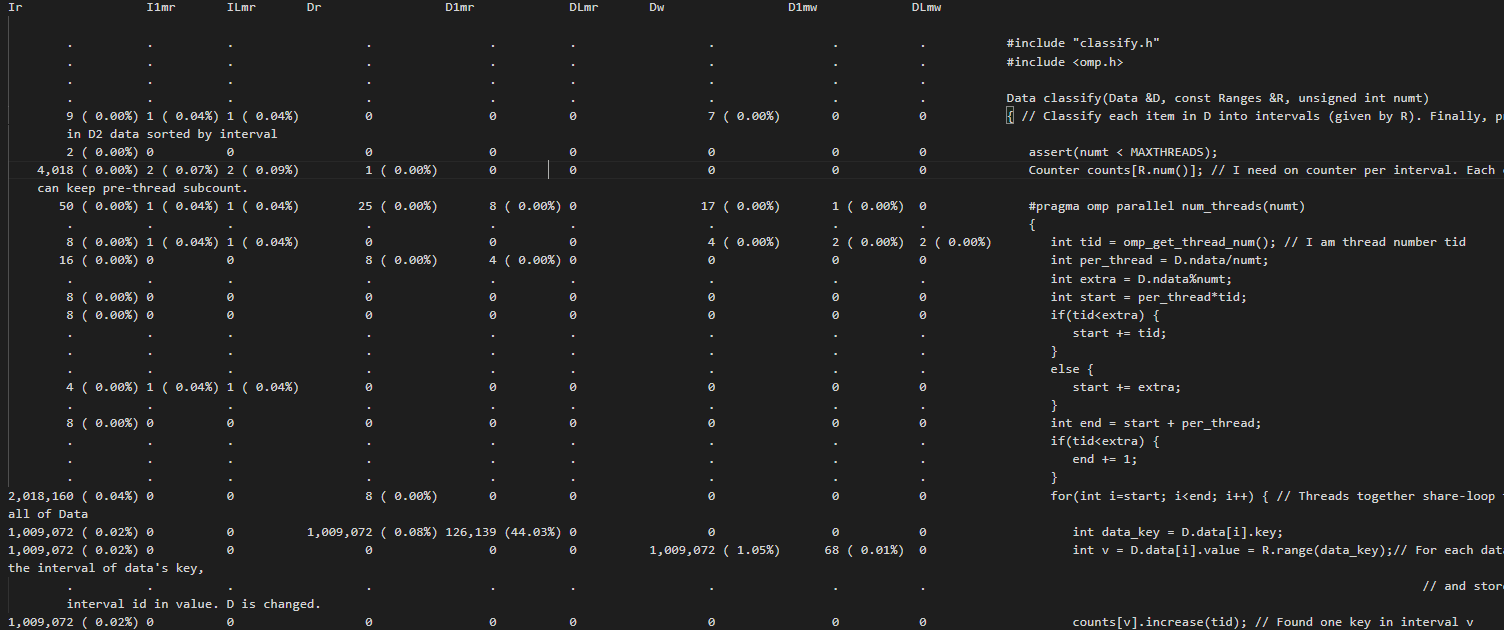
In the report we can see that the time taken by the classify function had reduced (earlier the cumulative time was 2.34 sec and now it was 1.34 sec).

**4) Analysis of valgrind of modified code:**

The annotated report for modified code can be found in the modCache.txt file.

In the report we can observe that the cache read misses had been reduced significantly.

For the first parallel part the cache read misses have been halved.



For the second parallel part, the cache read misses have almost reduced to 0.

Though the runtime reduced to large extent when only doing changes for the second parallel part, and then the changes in code for first parallel part increased the runtime. Even after the decrease of cache read misses in first parallel part the runtime increased because the threads have an unequal amount of work to be done. For each data we search the whole ranges until we find a range in which the data fits. Because of this the last thread have all data that belongs in the last few ranges and hence it traverses all the ranges for these data. This distribution of data is not supposed to happen always and a evenly spread data will instead decrease the runtime and hence I have included this optimization in the submission.

**5) Reasoning for changes:**

There were a lot of cache read misses in the old code reports while accessing the data D. This was happening because the work is divided in the threads in such a way that different threads accessed the same cache lines of the data D. As in first parallel part the data I divided in such a way that thread 0,1,2,3 will be first accessing the data indexed 0,1,2,3 in D and hence they all would be reading and writing in same cache lines and hence would cause the cache miss rates to be high.

So, I divided the data D to threads in blocks. So, like if there are 4 threads then first ¼ of data D will be given to thread 1, next ¼ to thread 2 and next ¼ to thread 3 and last ¼ to thread 4. This way each thread will work on different cache lines.

For second parallel part we have an algo which check for each range all the data items. I reduced the complexity to check only all the data and hence time decreased a lot. The first parallel part optimization is also done in this part.