COL380: Assignment 0

Sachin 2019CS10722

January, 2022

1 Changes Made:

1.1 Makefile

1. Added flags -g and -pg (required for running of profiliers)
2. Compiled using -fopenmp
3. Added make test command to change variables using command line arguments (for testing)
4. Added make clean command

So run makefile with following optional arguments for testing:
make test r=< $a1$ > d=< $a2$ > n=< $a3$ > t=< $a4$ > x=< $a5$ >
where,
a1 = range file
a2 = data file
a3 = max data to read
a4 = number of threads
a5 = reps

by default, a1 = rfile, a2 = dfile, a3 = 1009072, t = 4, x = 3

1.2 classify.h

1. Returned *this in the operator+= function of Ranges class (as pointed out in piazza).
2. Added function **increaseVal**(unsigned int **id**, int **val**) in class Counter, that increase count of id by val.

1.3 classify.cpp

During parallel computation the given code is dividing between multiple threads in following way:-

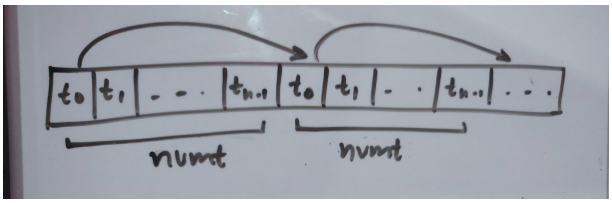

Figure 1: Original Code

I've changed it in following way:-

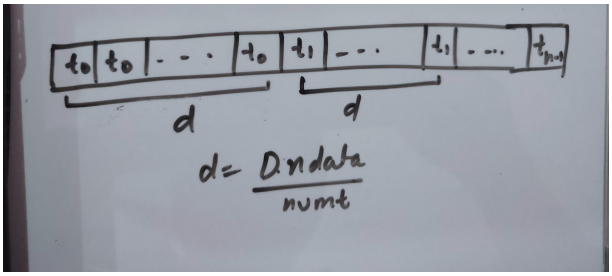

Figure 2: Modified Code

Moreover, for the part where data is sorted and stored in D2, given code does the following computation:-
$\{\forall r \in R \forall d \in D$: if(d.value = r) then add d in D2 with value r$\}$
I've changed that to following:-
$\{\forall d \in D$: add d in D2 with value d.value at its sorted position$\}$

Formula used to find sorted position of d:
$pos(i) = rangecount[r-1] + counts[r].get(t-1) + c[r].get(t)$
where,

t = thread number
$D_i = (k, r)$
$rangecount[r-1] = D_j : r_j < r$ (used also in original code)
$counts[r].get(t-1) = D_j : r_j = r$ but $t_j < t$

i.e. all data that lie in same range but are encountered by threads having id less than that of current (since I am giving continious set of data to threads)

$c[r].get(t) = D_j : r_j = r$ and $t_j = t$ but $j < i$

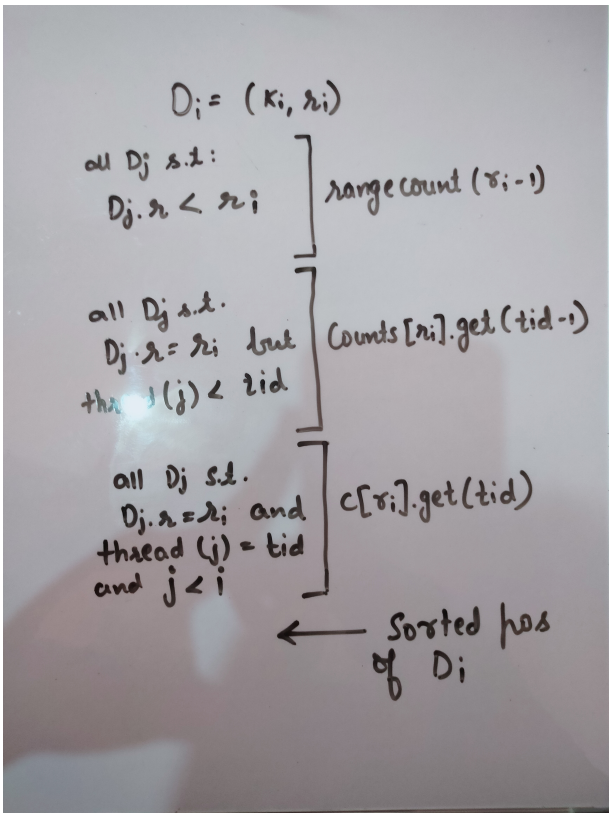

Figure 3: Formula Used

NOTE: Why I made these changes is explained in the follwoing sections where analysis of different profilers is mentioned.

2 perf

It is a detailed profiler that provides a lot of detailed data about the code, but I used the cache data of perf to optimize my code.

2.1 Original Code:

1. **perf stat -d -d -d make test**
   Using this I observed that L1-d cache misses are around 6% (irrepective of number of threads). But it was not clear where these misses are actually occuring, so to find that I used perf record command that generated perf.data file.

```
➜  A0 git:(master) ✗ sudo perf stat -d -d -d  make test type=0
./classify rfile dfile 1009072 4 3 0
405.461 ms
410.245 ms
422.693 ms
3 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 405.461 ms, Average was 412.8 ms

 Performance counter stats for 'make test type=0':

          5,037.00 msec task-clock                #    3.832 CPUs utilized
               121      context-switches          #    0.024 K/sec
                 4      cpu-migrations            #    0.001 K/sec
             9,380      page-faults               #    0.002 M/sec
    19,315,796,872      cycles                    #    3.835 GHz                      (30.96%)
    33,393,010,052      instructions              #    1.73  insn per cycle           (38.60%)
    11,874,274,197      branches                  # 2357.412 M/sec                    (38.48%)
         4,453,050      branch-misses             #    0.04% of all branches          (38.30%)
     6,233,301,874      L1-dcache-loads           # 1237.504 M/sec                    (38.30%)
       378,255,062      L1-dcache-load-misses     #    6.07% of all L1-dcache accesses  (38.32%)
         8,983,837      LLC-loads                 #    1.784 M/sec                    (30.72%)
           183,185      LLC-load-misses           #    2.04% of all LL-cache accesses  (31.68%)
   <not supported>      L1-icache-loads
         1,108,910      L1-icache-load-misses                                         (31.58%)
     6,471,342,958      dTLB-loads                # 1284.762 M/sec                    (31.40%)
         5,775,472      dTLB-load-misses          #    0.09% of all dTLB cache accesses  (31.30%)
            21,096      iTLB-loads                #    0.004 M/sec                    (31.10%)
            55,210      iTLB-load-misses          #  261.71% of all iTLB cache accesses  (30.99%)
   <not supported>      L1-dcache-prefetches
   <not supported>      L1-dcache-prefetch-misses

       1.314309004 seconds time elapsed

       5.022209000 seconds user
       0.015975000 seconds sys
```

Figure 4: Original Code: perf stat

2. **perf report**
   Following is the report:-

```
Samples: 9K of event 'cache-misses', Event count (approx.): 18685940
Overhead  Command   Shared Object        Symbol
  87.46%  classify  classify             [.] classifyOld
   2.66%  classify  classify             [.] classifyOld
   1.65%  classify  [kernel.kallsyms]    [k] 0xffffffff90605907
   0.93%  classify  classify             [.] repeatrun
   0.90%  classify  libstdc++.so.6.0.28  [.] std::num_get<char, std::
   0.88%  classify  [kernel.kallsyms]    [k] 0xffffffff902ff7ca
   0.67%  classify  [kernel.kallsyms]    [k] 0xffffffff90605f5e
   0.43%  make      [kernel.kallsyms]    [k] 0xffffffff90332a5b
```

Figure 5: Original Code: perf report

This shows that there are around 18685940 cache-miss events noted and 87% out them are from single part of code. Let's see what part of code that is.

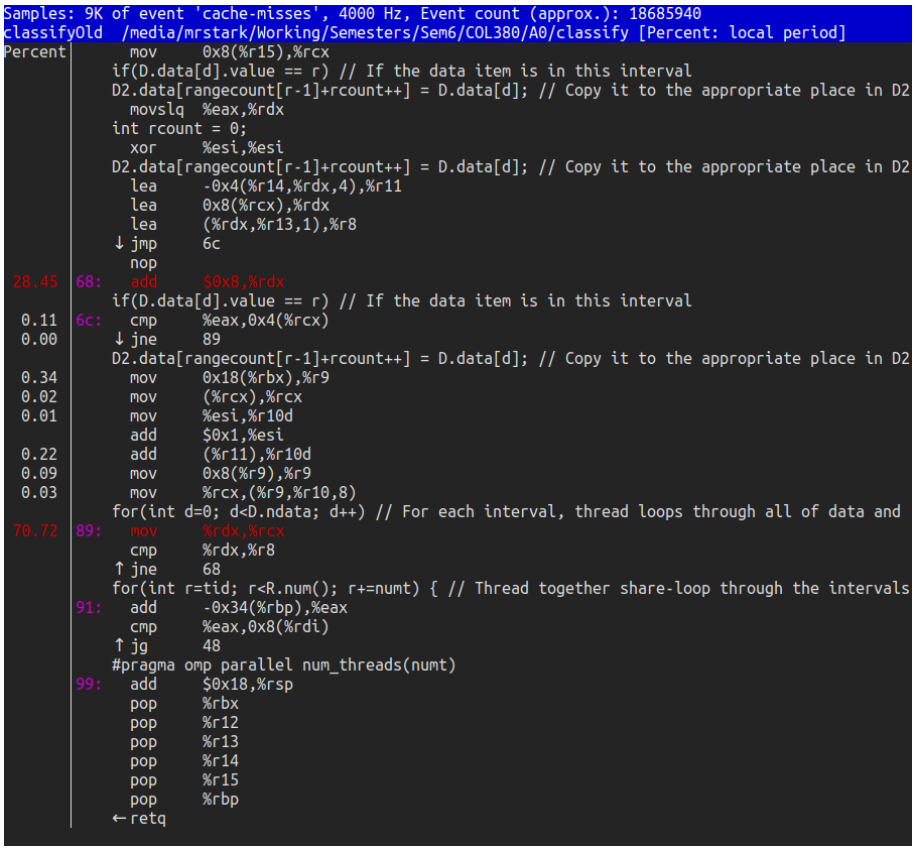

Figure 6: Original Code: cache analysis

Clearly, most of the cache misses are from the last part of classify function when we sort the data and store it in D2. More specifically the memory access of D.data[d].value and memory write of D2. This is because of false sharing between threads.

This motivated me to perform operation for a given thread such that if a cache line is cached in L1 of that thread then it should do future operations on the memory addresses on that cache line.

Hence I modified the code the way described above. But to do that I had to use the formula stated above, that requires some way to get number of data lie in same range as D_i but are before D_i in data list. Hence modified initial parallel computation (of assigning ranges to data) in such a way that counts[r].get(t) gives that quantity.

2.2 Modified Code:

1. **perf stat -d -d -d make test**
   Clearly, L1-d cache misses have gone down to 0.09%, that is a reasonably good number. In return runtime of code also decreased.

```
→  A0 git:(master) ✗ sudo perf stat -d -d -d  make test type=1
./classify rfile dfile 1009072 4 3 1
266.546 ms
278.11 ms
272.335 ms
3 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 266.546 ms, Average was 272.33 ms

 Performance counter stats for 'make test type=1':

          2,158.49 msec task-clock                #    2.380 CPUs utilized
               145      context-switches          #    0.067 K/sec
                 3      cpu-migrations            #    0.001 K/sec
             9,589      page-faults               #    0.004 M/sec
     8,559,084,790      cycles                    #    3.965 GHz                      (31.33%)
    15,743,839,250      instructions              #    1.84  insn per cycle           (38.92%)
     6,263,581,889      branches                  # 2901.840 M/sec                    (38.54%)
         3,287,377      branch-misses             #    0.05% of all branches          (38.21%)
     3,448,196,929      L1-dcache-loads           # 1597.507 M/sec                    (38.02%)
         3,113,004      L1-dcache-load-misses     #    0.09% of all L1-dcache accesses  (38.71%)
           233,622      LLC-loads                 #    0.108 M/sec                    (30.55%)
           119,274      LLC-load-misses           #   51.05% of all LL-cache accesses  (31.22%)
   <not supported>      L1-icache-loads
           727,085      L1-icache-load-misses                                         (31.26%)
     3,246,948,547      dTLB-loads                # 1504.271 M/sec                    (31.64%)
            27,507      dTLB-load-misses          #    0.00% of all dTLB cache accesses  (32.42%)
            12,719      iTLB-loads                #    0.006 M/sec                    (32.06%)
            17,078      iTLB-load-misses          #  134.27% of all iTLB cache accesses  (31.86%)
   <not supported>      L1-dcache-prefetches
   <not supported>      L1-dcache-prefetch-misses

       0.907034125 seconds time elapsed

       2.148087000 seconds user
       0.012055000 seconds sys
```

Figure 7: Modified Code: perf stat

2. **perf report**

```
Samples: 2K of event 'cache-misses', Event count (approx.): 3550238
Overhead  Command   Shared Object        Symbol
  25.64%  classify  classify             [.] classify
  16.56%  classify  classify             [.] classify
  15.98%  classify  classify             [.] repeatrun
   7.08%  classify  [kernel.kallsyms]    [k] 0xffffffff90605907
   4.65%  classify  libstdc++.so.6.0.28  [.] std::num_get<char, std::
   2.91%  make      ld-2.31.so           [.] _dl_sort_maps
   2.79%  classify  [kernel.kallsyms]    [k] 0xffffffff90605f5e
   1.79%  classify  [kernel.kallsyms]    [k] 0xffffffff90284cd2
   1.69%  classify  [kernel.kallsyms]    [k] 0xffffffff902ff6b6
```

Figure 8: Modified Code: perf report

This shows that there are around 3550238 (<< 18685940 that we got in original code) cache-miss events noted and that too evenly distributed.

Let's see the part of code that gave most cache misses in original code.

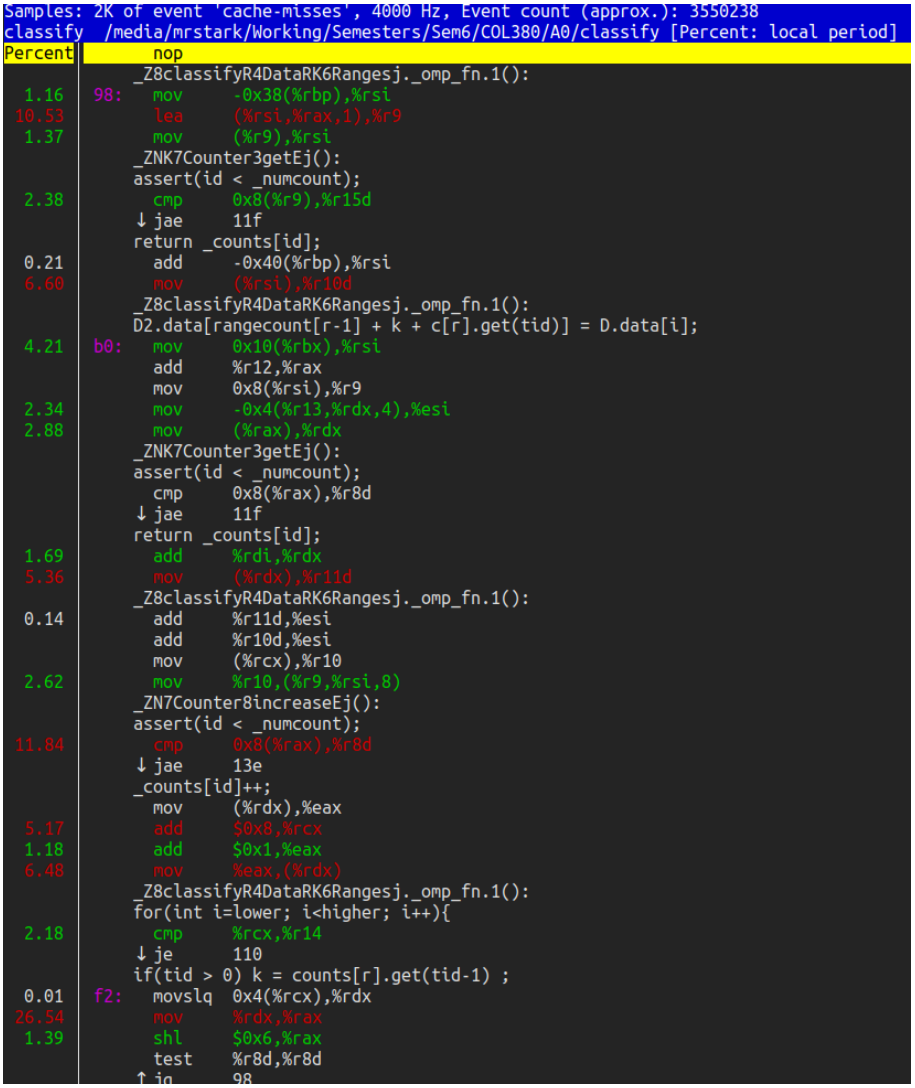

Figure 9: Modified Code: cache analysis

Clearly, cache misses are not concentrated on few instructions but are evenly distributed.

2.3 Comparision:

I wrote a python script to run original and modified code on varying number of threads and plotted the graph of cache misses. Here are the results for L1-d cache misses noted from perf:-

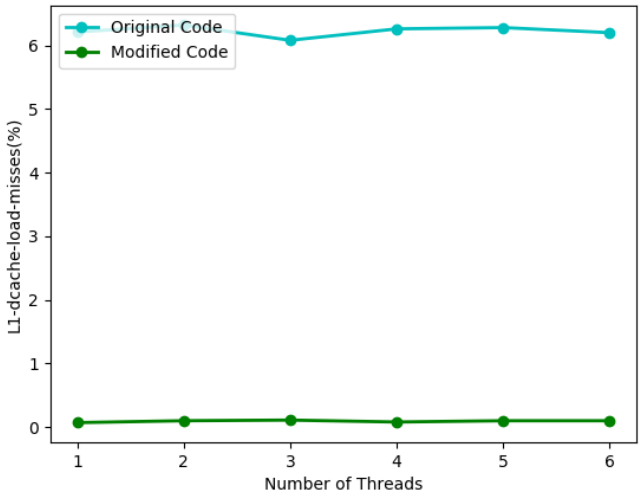

Figure 10: Cache Comparision: perf

NOTE: Number of reps is kept 10 for all the cache miss percent computation.

It is evident from graph that Modified code has better cache behaviour than original code (in fact it has negligible cache misses).

3 valgrind

Valgrind is also a detailed profiler that provides not only cache analysing tools but also tools to analyse the runtime of different parts of code and compare them. It also provides features using which we can check memory leaks inside code.
I cheked for any memory leak, there wasn't any. Thereafter I used cachegrind tool of valgrind to study the cache behaviour of modified code that I got through optimisations done using perf.

I used the command:

'valgrind –tool=cachegrind ./classify rfile dfile 1009072 4 3'

NOTE: One thing that I noticed while using valgrind is that, out of all 3 profilers this took the highest time to run the code. Moreover other 2 did not had any resonable overhead on time but valgrind took a lot more time than simply running the same code (without any profiler).

3.1 Original Code:

Following are the results:-

```
➜  A0 git:(master) ✗ valgrind --tool=cachegrind ./classify rfile dfile 1009072 4 3 0
==58448== Cachegrind, a cache and branch-prediction profiler
==58448== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==58448== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==58448== Command: ./classify rfile dfile 1009072 4 3 0
==58448==
--58448-- warning: L3 cache found, using its data for the LL simulation.
--58448-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--58448-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
27876 ms
27983.4 ms
28034 ms
3 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 27876 ms, Average was 27964.4 ms
==58448==
==58448== Process terminating with default action of signal 27 (SIGPROF)
==58448==    at 0x4BB865A: __open_nocancel (open64_nocancel.c:45)
==58448==    by 0x4BC636F: write_gmon (gmon.c:370)
==58448==    by 0x4BC6BCE: _mcleanup (gmon.c:444)
==58448==    by 0x4AEC15D: __cxa_finalize (cxa_finalize.c:83)
==58448==    by 0x10A8D6: ??? (in /media/mrstark/Working/Semesters/Sem6/COL380/A0/classify)
==58448==    by 0x4011F5A: _dl_fini (dl-fini.c:138)
==58448==    by 0x4AEBA26: __run_exit_handlers (exit.c:108)
==58448==    by 0x4AEBBDF: exit (exit.c:139)
==58448==    by 0x4AC90B9: (below main) (libc-start.c:342)
==58448==
==58448== I   refs:      34,157,935,239
==58448== I1  misses:             3,526
==58448== LLi misses:             3,260
==58448== I1  miss rate:           0.00%
==58448== LLi miss rate:           0.00%
==58448==
==58448== D   refs:       6,439,112,481  (6,339,172,478 rd   + 99,940,003 wr)
==58448== D1  misses:       381,858,706  (  380,386,273 rd   +  1,472,433 wr)
==58448== LLd misses:         1,305,474  (      388,978 rd   +    916,496 wr)
==58448== D1  miss rate:            5.9% (          6.0%     +        1.5%  )
==58448== LLd miss rate:            0.0% (          0.0%     +        0.9%  )
==58448==
==58448== LL refs:          381,862,232  (  380,389,799 rd   +  1,472,433 wr)
==58448== LL misses:          1,308,734  (      392,238 rd   +    916,496 wr)
==58448== LL miss rate:             0.0% (          0.0%     +        0.9%  )
[1]    58448 profile signal  valgrind --tool=cachegrind ./classify rfile dfile 1009072 4 3 0
```

Figure 11: Original Code: valgrind

Instructions cache had no misses and data cache had around 6% misses (Confirming the results got from perf).

LL cache is the L3 cache for my system, that too had ngligible misses. So we only need to optimise the cache behaviour of L1 data cache (that I already did using perf results).

3.2 Modified Code:

Following are the results:-

```
→  A0 git:(master) ✗ valgrind --tool=cachegrind ./classify rfile dfile 1009072 4 3 1
==58656== Cachegrind, a cache and branch-prediction profiler
==58656== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==58656== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==58656== Command: ./classify rfile dfile 1009072 4 3 1
==58656==
--58656-- warning: L3 cache found, using its data for the LL simulation.
--58656-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--58656-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
11044.5 ms
10696.2 ms
10648 ms
3 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 10648 ms, Average was 10796.2 ms
==58656==
==58656== Process terminating with default action of signal 27 (SIGPROF)
==58656==    at 0x4BB865A: __open_nocancel (open64_nocancel.c:45)
==58656==    by 0x4BC636F: write_gmon (gmon.c:370)
==58656==    by 0x4BC6BCE: _mcleanup (gmon.c:444)
==58656==    by 0x4AEC15D: __cxa_finalize (cxa_finalize.c:83)
==58656==    by 0x10A8D6: ??? (in /media/mrstark/Working/Semesters/Sem6/COL380/A0/classify)
==58656==    by 0x4011F5A: _dl_fini (dl-fini.c:138)
==58656==    by 0x4AEBA26: __run_exit_handlers (exit.c:108)
==58656==    by 0x4AEBBDF: exit (exit.c:139)
==58656==    by 0x4AC90B9: (below main) (libc-start.c:342)
==58656==
==58656== I   refs:      16,037,563,768
==58656== I1  misses:             3,532
==58656== LLi misses:             3,270
==58656== I1  miss rate:           0.00%
==58656== LLi miss rate:           0.00%
==58656==
==58656== D   refs:       3,437,933,225  (3,334,656,550 rd   + 103,276,675 wr)
==58656== D1  misses:         2,286,507  (      800,934 rd   +   1,485,573 wr)
==58656== LLd misses:         1,143,242  (      297,292 rd   +     845,950 wr)
==58656== D1  miss rate:            0.1% (          0.0%     +         1.4%  )
==58656== LLd miss rate:            0.0% (          0.0%     +         0.8%  )
==58656==
==58656== LL refs:            2,290,039  (      804,466 rd   +   1,485,573 wr)
==58656== LL misses:          1,146,512  (      300,562 rd   +     845,950 wr)
==58656== LL miss rate:             0.0% (          0.0%     +         0.8%  )
[1]    58656 profile_signal  valgrind --tool=cachegrind ./classify rfile dfile 1009072 4 3 1
```

Figure 12: Modified Code: valgrind

Clearly cache misses of L1 cache are reduced and also matched with the results got using perf.

3.3 Comparision:

I wrote a python script to run original and modified code on varying number of threads and plotted the graph of cache misses data got through valgrind. Result is shown on next page.

The graph is more or less same as that we got from that of perf.

NOTE: Number of reps is kept 3 for all the cache miss percent computation.

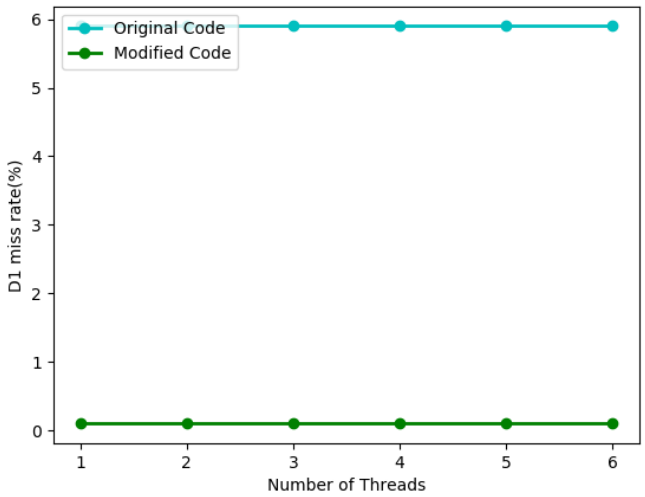

Figure 13: Cache Comparision: valgrind

4 gprof

This is a very old profiler (in use since 80's) and provides many featues to analyse the code on the basis of make parameters, like function taking most amount of time, function call graph etc. But one drawback of this profiler is that it only analyse the main thread of process. Hence I was not able to do any optimisation using the results of this profiler.

4.1 Original Code:

gprof –flat-profile classify

```
➜  A0 git:(master) ✗ gprof --flat-profile classify
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.20      2.96     2.96                             readRanges(char const*)
  0.00      2.96     0.00        3     0.00     0.00  classifyOld(Data&, Ranges const&, unsigned int)
  0.00      2.96     0.00        3     0.00     0.00  timedwork(Data&, Ranges const&, unsigned int)
  0.00      2.96     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11classifyOldR4DataRK6Rangesj
  0.00      2.96     0.00        1     0.00     0.00  _GLOBAL__sub_I_type

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

Figure 14: Original Code: gprof –flat-profile

4.2 Modified Code:

gprof –flat-profile classify

```
→  A0 git:(master) ✗ gprof --flat-profile classify
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 99.60      1.67     1.67                             readRanges(char const*)
  0.60      1.68     0.01                             readData(char const*, unsigned int)
  0.00      1.68     0.00        3     0.00     0.00  classify(Data&, Ranges const&, unsigned int)
  0.00      1.68     0.00        3     0.00     0.00  timedwork(Data&, Ranges const&, unsigned int)
  0.00      1.68     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11classifyOldR4DataRK6Rangesj
  0.00      1.68     0.00        1     0.00     0.00  _GLOBAL__sub_I_type

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

Figure 15: Modified Code: gprof –flat-profile

5 Time Comparision:

Finally when all analysis was done I plotted the number of threads vs runtime graph for original and modified code. Here are the results:-

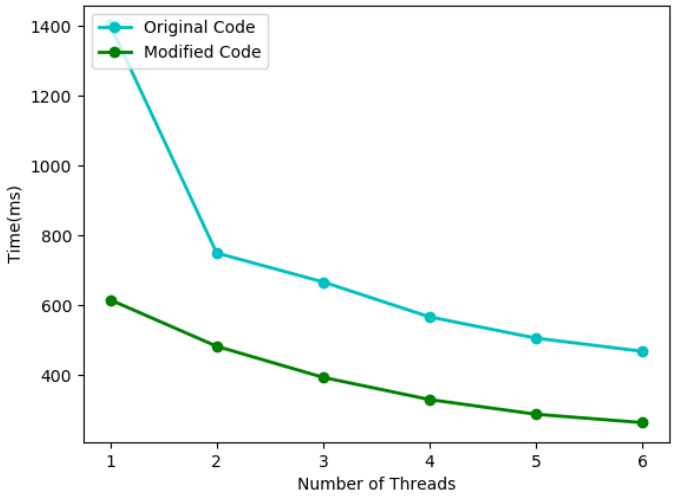

Figure 16: Time Comparision: local

I also plotted the same graph by running the plotting script on palasi VM to double check. Here are the results:-

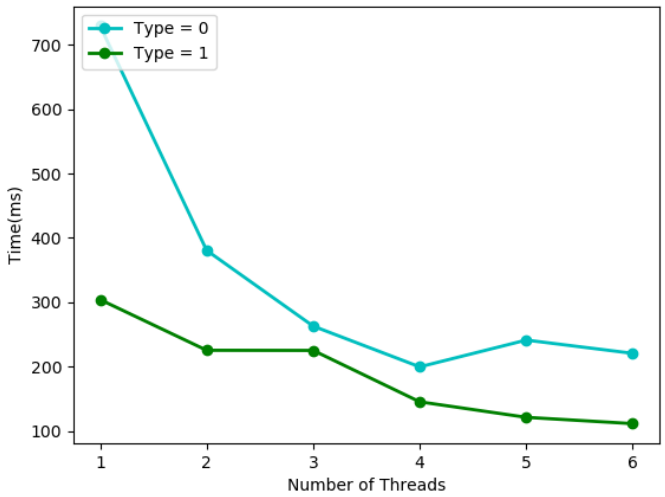

Figure 17: Time Comparision: VM

6 Folder Structure:

Following are the files in the folder:-

1. All files provided beforehand (classify.h, classify.cpp, Makefile modified).
2. **report.pdf:** Pdf containing analysis procedure and conclusions.
3. **test.py:** Python code I used for testing and plotting.
4. **perf-O.txt:** Output of perf on original code (with number of threads 1 to 6 and number of repetations = 10).
5. **perf-M.txt:** Output of perf on modified code (with number of threads 1 to 6 and number of repetations = 10).
6. **valgrind-O.txt:** Output of valgrind on original code (with number of threads 1 to 6 and number of repetations = 3).
7. **valgrind-M.txt:** Output of valgrind on modified code (with number of threads 1 to 6 and number of repetations = 3).
8. **gprof-O.txt:** Output of gprof on original code (with number of threads 1 to 6 and number of repetations = 10).
9. **gprof-M.txt:** Output of gprof on modified code (with number of threads 1 to 6 and number of repetations = 10).