INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 0

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL380 | Prof. Subodh Kumar

Due January 16, 2022

Contents

Chapter 1

Analysis of Original Code

I used three profiling/performance measurement tools, i.e., **gprof**, **valgrind** (tool = `cachegrind`) and **perf** (`stat` and `c2c`), to analyse the efficiency of the original code. I present my findings and intended optimisations below:

1.1 Profiler: gprof

The code was compiled with `-pg` compiler flags and then run using the following command to generate `gmon.out` file:

`./classify rfile dfile 1009072 4 3`

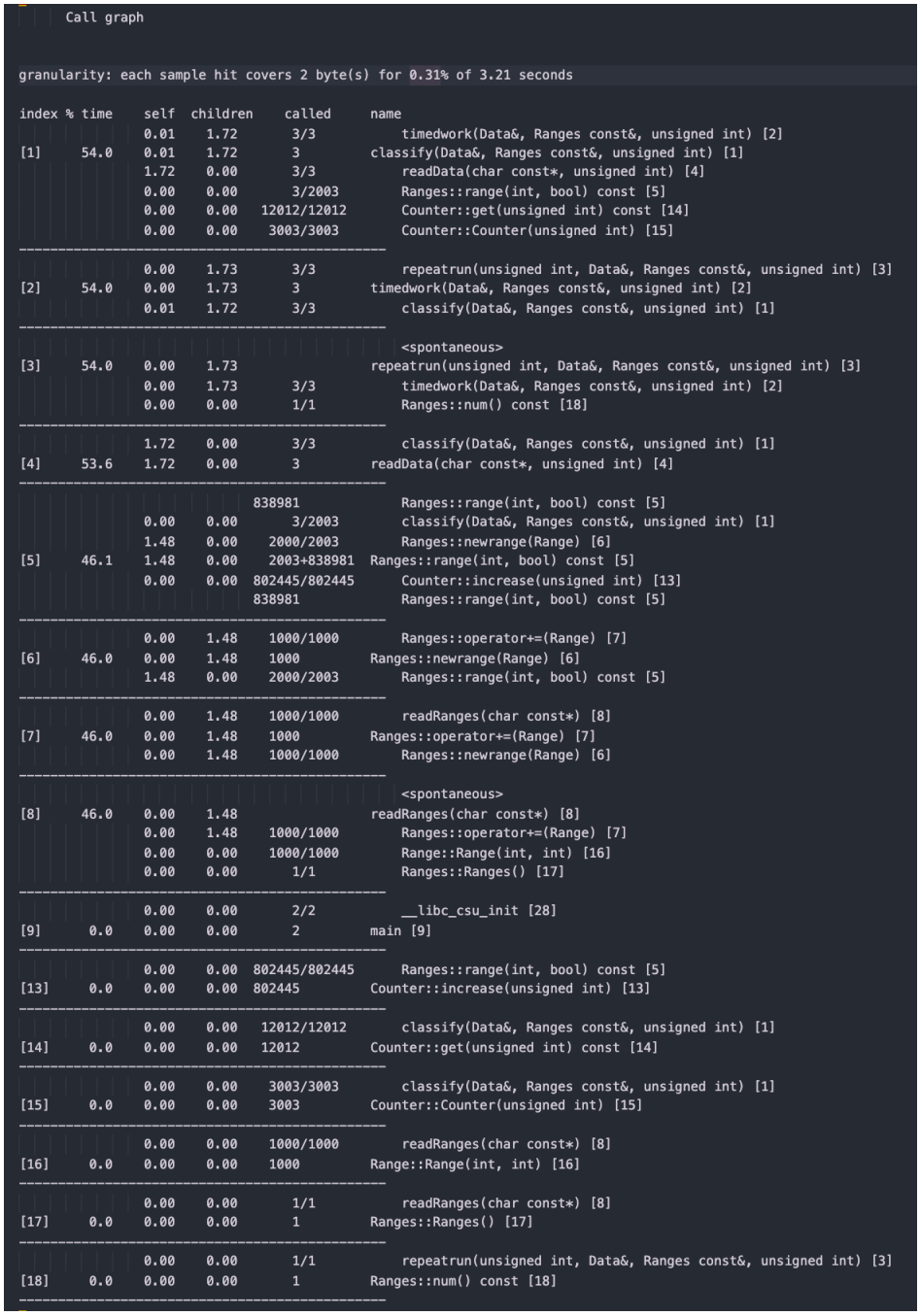
The report was generated by running `gprof` command on the output file (`gmon.out`):

`gprof -a -b ./classify > gprof_original.txt`

This report can be found in the `gprof/original/` directory present in the submission. I have also attached a screenshot of the same in this document below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 53.66      1.72     1.72        3   574.15   574.15  readData(char const*, unsigned int)
 46.17      3.20     1.48     2003     0.74     0.74  Ranges::range(int, bool) const
  0.31      3.21     0.01        3     3.34   578.22  classify(Data&, Ranges const&, unsigned int)
  0.00      3.21     0.00   802445     0.00     0.00  Counter::increase(unsigned int)
  0.00      3.21     0.00    12012     0.00     0.00  Counter::get(unsigned int) const
  0.00      3.21     0.00     3003     0.00     0.00  Counter::Counter(unsigned int)
  0.00      3.21     0.00     1000     0.00     0.00  Range::Range(int, int)
  0.00      3.21     0.00     1000     0.00     1.48  Ranges::newrange(Range)
  0.00      3.21     0.00     1000     0.00     1.48  Ranges::operator+=(Range)
  0.00      3.21     0.00        3     0.00   578.22  timedwork(Data&, Ranges const&, unsigned int)
  0.00      3.21     0.00        2     0.00     0.00  main
  0.00      3.21     0.00        1     0.00     0.00  Ranges::Ranges()
  0.00      3.21     0.00        1     0.00     0.00  Ranges::num() const
```

Figure 1.1: Flat Profile



```
Call graph


granularity: each sample hit covers 2 byte(s) for 0.31% of 3.21 seconds

index % time    self  children    called     name
                0.01    1.72       3/3           timedwork(Data&, Ranges const&, unsigned int) [2]
[1]     54.0    0.01    1.72       3         classify(Data&, Ranges const&, unsigned int) [1]
                1.72    0.00       3/3           readData(char const*, unsigned int) [4]
                0.00    0.00       3/2003        Ranges::range(int, bool) const [5]
                0.00    0.00   12012/12012       Counter::get(unsigned int) const [14]
                0.00    0.00    3003/3003        Counter::Counter(unsigned int) [15]
-----------------------------------------------
                0.00    1.73       3/3           repeatrun(unsigned int, Data&, Ranges const&, unsigned int) [3]
[2]     54.0    0.00    1.73       3         timedwork(Data&, Ranges const&, unsigned int) [2]
                0.01    1.72       3/3           classify(Data&, Ranges const&, unsigned int) [1]
-----------------------------------------------
                                                  <spontaneous>
[3]     54.0    0.00    1.73                 repeatrun(unsigned int, Data&, Ranges const&, unsigned int) [3]
                0.00    1.73       3/3           timedwork(Data&, Ranges const&, unsigned int) [2]
                0.00    0.00       1/1           Ranges::num() const [18]
-----------------------------------------------
                1.72    0.00       3/3           classify(Data&, Ranges const&, unsigned int) [1]
[4]     53.6    1.72    0.00       3         readData(char const*, unsigned int) [4]
-----------------------------------------------
                              838981             Ranges::range(int, bool) const [5]
                0.00    0.00       3/2003        classify(Data&, Ranges const&, unsigned int) [1]
                1.48    0.00    2000/2003        Ranges::newrange(Range) [6]
[5]     46.1    1.48    0.00    2003+838981  Ranges::range(int, bool) const [5]
                0.00    0.00  802445/802445      Counter::increase(unsigned int) [13]
                              838981             Ranges::range(int, bool) const [5]
-----------------------------------------------
                0.00    1.48    1000/1000        Ranges::operator+=(Range) [7]
[6]     46.0    0.00    1.48    1000         Ranges::newrange(Range) [6]
                1.48    0.00    2000/2003        Ranges::range(int, bool) const [5]
-----------------------------------------------
                0.00    1.48    1000/1000        readRanges(char const*) [8]
[7]     46.0    0.00    1.48    1000         Ranges::operator+=(Range) [7]
                0.00    1.48    1000/1000        Ranges::newrange(Range) [6]
-----------------------------------------------
                                                  <spontaneous>
[8]     46.0    0.00    1.48                 readRanges(char const*) [8]
                0.00    1.48    1000/1000        Ranges::operator+=(Range) [7]
                0.00    0.00    1000/1000        Range::Range(int, int) [16]
                0.00    0.00       1/1           Ranges::Ranges() [17]
-----------------------------------------------
                0.00    0.00       2/2           __libc_csu_init [28]
[9]      0.0    0.00    0.00       2         main [9]
-----------------------------------------------
                0.00    0.00  802445/802445      Ranges::range(int, bool) const [5]
[13]     0.0    0.00    0.00  802445         Counter::increase(unsigned int) [13]
-----------------------------------------------
                0.00    0.00   12012/12012       classify(Data&, Ranges const&, unsigned int) [1]
[14]     0.0    0.00    0.00   12012         Counter::get(unsigned int) const [14]
-----------------------------------------------
                0.00    0.00    3003/3003        classify(Data&, Ranges const&, unsigned int) [1]
[15]     0.0    0.00    0.00    3003         Counter::Counter(unsigned int) [15]
-----------------------------------------------
                0.00    0.00    1000/1000        readRanges(char const*) [8]
[16]     0.0    0.00    0.00    1000         Range::Range(int, int) [16]
-----------------------------------------------
                0.00    0.00       1/1           readRanges(char const*) [8]
[17]     0.0    0.00    0.00       1         Ranges::Ranges() [17]
-----------------------------------------------
                0.00    0.00       1/1           repeatrun(unsigned int, Data&, Ranges const&, unsigned int) [3]
[18]     0.0    0.00    0.00       1         Ranges::num() const [18]
-----------------------------------------------
```

Figure 1.2: Call Graph

The *Flat Profile* gives information about different functions (only those that were profiled), such as %-time spent in the function, time spent only in that function (*self seconds*), cumulative sum of *self seconds*, number of times a function was invoked, time spent only in that function per call and time spent in that function and its descendants per call.
On the other hand, the *Call Graph* lays out the same information in a different format. This graph also gives information about which function called which (parent-child relationship).

Observations: **gprof** does not give any information regarding the cache-based performance of the code. Hence, I was not able to pin-point any memory-based bottleneck. However, it does indicate which parts of the code are slower than the others. Using the report generated, I observed that the function **Ranges::range(int, bool) const** was taking up a significant portion of time (46.17 %). This function is used to determine the range of a given data item.
Apart from this, I was not able to observe any other function-call based bottleneck in the code.

Optimisations: A very good optimisation to reduce the time spent in the *range*-function, is to use **binary search** instead of linear search (used in original code). However, *binary search* requires that the ranges are present in an increasing order. This condition is not guaranteed, as the input can be in any order and the code will accept it as long as the ranges are non-overlapping.

Conclusions: I implemented *binary search* and used it in my modified code (function name: `range_binary`). The resulting code was running **60 - 90 ms** faster for small number of threads (e.g. 4, 8 etc.) and at least **15 ms** faster for large number of threads (e.g. 48, 63 etc.). However, as the input specifications do not guarantee the correctness of binary search, I have not used it in the final submission. The implementation for the same can be found in the *classify.cpp* file.

1.2 Profiler: valgrind

I used the *cachegrind* tool of valgrind to analyse the cache-based performance of the original code, and use that information to improve the memory accesses and reduce cache-misses (data cache-misses).
The code was compiled with `-g` compiler flag and then run using the following command to generate `cachegrind.out.23443` file:
`valgrind --tool=cachegrind ./classify rfile dfile 1009072 4 3`
The final report was generated by running `cg_annotate` command on the `.out.` file:
`cg_annotate --auto=yes cachegrind.out.23443 > valgrind_original.txt`
This report can be found in the `valgrind/original/` directory present in the submission. Since, the report is too big to fit in this document, I have only attached screenshots of the summary and those parts of code that had a significant number of cache-misses (data).

```
==23443== I   refs:      34,207,795,679
==23443== I1  misses:             3,569
==23443== LLi misses:             3,410
==23443== I1  miss rate:           0.00%
==23443== LLi miss rate:           0.00%
==23443==
==23443== D   refs:       6,448,230,466  (6,345,256,694 rd   + 102,973,772 wr)
==23443== D1  misses:       381,828,821  (  380,356,508 rd   +   1,472,313 wr)
==23443== LLd misses:         3,971,689  (    2,992,791 rd   +     978,898 wr)
==23443== D1  miss rate:            5.9% (          6.0%     +         1.4%  )
==23443== LLd miss rate:            0.1% (          0.0%     +         1.0%  )
==23443==
==23443== LL refs:          381,832,390  (  380,360,077 rd   +   1,472,313 wr)
==23443== LL misses:          3,975,099  (    2,996,201 rd   +     978,898 wr)
==23443== LL miss rate:             0.0% (          0.0%     +         1.0%  )
```

Figure 1.3: Valgrind Summary

Observations: *Cachegrind* gives information about number of instruction/data references and respective number of *cache-misses*. In case of data, it also gives information about *write* misses. The misses are specified for **L1** and **LL** caches (LL = L3 on my machine). For clear reasons, LL cache-misses are more expensive than L1 cache-misses. However, a large number of L1 cache-misses can also slow down the program tremendously. Hence, I modified the code so as to reduce total *cache-misses* in general.
Figure 1.3 is self-explanatory and simply gives a summary of total number of references and misses, and the miss-rate. Figure 1.4 and 1.5 annotate the source-code to give the exact location in code where majority of the misses are occuring. The **six** columns on the left represent *Dr, D1mr, DLmr, Dw, D1mw, DLmw* respectively (from left to right). Here, r stands for read, w for write, 1 for L1 and L for LL cache. These two figures indicate code-locations where a significant number of cache-misses are occurring, thereby, opening door for optimisations.

```
          .          .          .          .          .          .  {
          0          0          0         12          8          8      int tid = omp_get_thread_num(); // I am thread number tid
  3,027,240          3          0          0          0          0      for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop throug
  3,027,240  1,513,614  172,203 6,054,432          3          0        int v = D.data[i].value = R.range(D.data[i].key);// For each data, find
          .          .          .          .          .          .                                 // and store the interval id in value. D is changed.
          .          .          .          .          .          .        counts[v].increase(tid); // Found one key in interval v
          .          .          .          .          .          .      }
```

Figure 1.4: First omp parallel section

Problem: A total of 1,513,614 data read misses are occurring in L1 cache while executing the instruction shown in Figure 1.4. This is a huge overhead. While experimenting, I found that traversing the whole D.data array from D.data[0] to D.data[D.ndata - 1] leads to around 380,000 L1 cache-misses on average. Hence, having 1,513,614 misses, which is around **4** times the ideal case, is definitely a bottleneck in the given code. This increased misses, are possibly due to the fact that different threads are accessing alternating data items in memory. Therefore, a thread is not being able to use the spatial locality of items efficiently. The cache-line size on my machine is 64 bytes and hence, can only fit a maximum of 8 D.data items (8 bytes each). If threads are alternating among items, a single thread can hit at most twice in the same cache-line before experiencing a miss (4 threads were used for this analysis). Also, since the threads are possibly executing on different cores (8-core processor), each thread will have a different L1-cache, and will thus, end up having almost the same number of misses, while traversing the D.data array, i.e. 380,000. Multiplying this number by 4 (number of threads), gives 1,520,000 which is a good estimate of the actual number of cache-misses.

Optimisation: I optimised the code by splitting up the D.data array into chunks of 12 consecutive items and scheduling their processing on different threads. The *consecutive-ness* allows for a greater degree of spatial locality to be used effectively by each thread (upto 8-items in the same cache-line). The reason why I chose 12 consecutive items, even though a maximum of 8 items can be present in a single cache-line is given below:

- The array is allocated dynamically, and there is no guarantee that the starting element will be located at a 64-byte boundary. Hence, 8-consecutive elements need not lie in the same cache-line.

- Dividing array into chunks helps in reducing the number of L1 cache-misses (spatial locality), however, if the chunks are disjoint, this can increase the number of LL cache-misses, which is shared across threads. Hence, accessing items which are present in some other cache-line helps in reducing the number of LL cache-misses for some other thread, as that cache-line is already possibly present in LL (shared cache).
- Given the above reasoning, 12 seems a very natural number $= 8 + \frac{8}{2}$, i.e., incorporating best of both worlds (L1 and LL).
- Last but not the least, I did extensive testing with different values of chunk size, ranging from 4 to 16, and found that 12 was giving the best performance as far as program runtime is considered. I also used $\frac{D.ndata}{numt}$, but this was leading to increase in runtime (possibly due to large number of LL cache-misses).

```
          0          0         0         0       0       0          int rcount = 0;
      3,003      2,991        46         0       0       0          for(int d=0; d<D.ndata; d++) // For each interval, thread loops th
3,030,243,216 378,783,407 2,802,992         0       0       0              if(D.data[d].value == r) // If the data item is in this interv
  12,108,864      5,991     1,569 3,027,216 381,042 380,604                  D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it
          .          .         .         .       .       .      }
          .          .         .         .       .       .  }
```

Figure 1.5: Second omp parallel section

Problem: A total of 378,783,407 data read misses are occurring in L1 cache while executing the instruction shown in Figure 1.5. This is a huge overhead. As mentioned before, traversing the whole D.data array from D.data[0] to D.data[D.ndata - 1] leads to around 380,000 L1 cache-misses on average. Hence, having 378,783,407 misses, which is around **1000** times the ideal case, is definitely a bottleneck in the given code. This increased number of misses can be easily justified. Notice that for each range number, a thread is traversing the whole data array. In other words, the D.data array is being traversed from start to end a total of 1001 times (including the empty range). Since, a single traversal results in around 380,000 L1 cache-misses, it is not surprising that 1001 such traversals are leading to around 380,000,000 L1 cache-misses. An important point here is that the size of L1 cache (data) is only 32768 bytes (= 4096 data items). Hence, L1 cache cannot simultaneously store all 1009072 data items. Therefore, repeated traversals (in same order), will incur almost the same number of cache-misses.

Optimisation: Main idea behind reducing the cache misses is to prevent a thread from traversing the D.data array multiple times, as that is redundant and unnecessary. Hence, in order to optimise the code, I ensure that each thread only traverses the D.data array exactly once. This strategy is equivalent to inter-changing the order of the two *for*-loops. Since, each thread is traversing the data array exactly once, there will be roughly $numt \times 380,000$ cache-misses, which is still 250 times less than that in original code (in case $numt = 4$). The implementation of this optimisation can be found in *classify.cpp* file.

Conclusion: By doing the above two optimisations, I was able to reduce a large number of cache-misses and improve my program's efficiency. The analysis reports of the modified code can be found in **Chapter 2** for comparison.

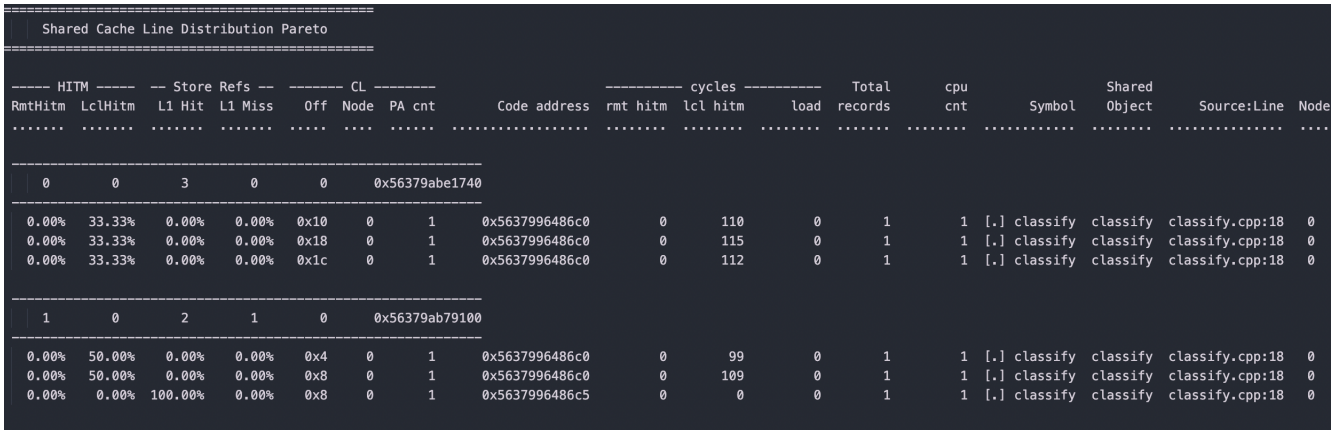
1.3 Profiler: perf

I used the *stat* and *c2c* option of perf to analyse performance of the original code (general and cache-based). Using report generated by *c2c* tool, I was able to identify and remove instances of **false-sharing** from the code.
The code was compiled with `-g` flag and then run using the following two commands:
`perf stat -d ./classify rfile dfile 1009072 4 3`
`perf c2c record ./classify rfile dfile 1009072 4 3`
The final report was generated by using the `report` command of perf:
`perf c2c report > perf_original.txt`
This report can be found in the `perf/original/` directory present in the submission. Since, the report is too big to fit in this document, I have only attached screenshots of the performance summary (stat) and a few (out of 262) cache-lines which were being shared *falsely* among the threads.

```
Performance counter stats for './classify rfile dfile 1009072 4 3':

       3982.948183      task-clock:u (msec)       #    3.621 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
              6451      page-faults:u             #    0.002 M/sec
        9817846975      cycles:u                  #    2.465 GHz
       29942869748      instructions:u            #    3.05  insn per cycle
       10874296603      branches:u                # 2730.213 M/sec
          3938437      branch-misses:u           #    0.04% of all branches
        6446286153      L1-dcache-loads:u         # 1618.471 M/sec
         391974074      L1-dcache-load-misses:u   #    6.08% of all L1-dcache hits
          33282400      LLC-loads:u               #    8.356 M/sec
           2058309      LLC-load-misses:u         #    6.18% of all LL-cache hits

       1.100027249 seconds time elapsed
```

Figure 1.6: Performance Statistics

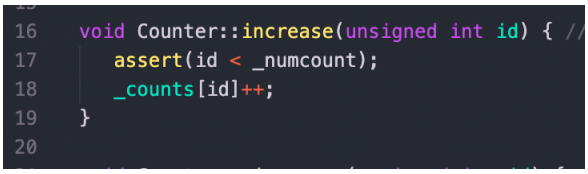
Figure 1.6 shows the output of `stat` command. It shows general performance-based statistics of the program along with cache loads and misses. Extensive analysis regarding cache-misses has already been done in **Section 1.2**, and hence, will not be repeated in this section. However, one quantity of interest here is the **IPC** count which is 3.05 for the original code. In following chapter, it will be shown that removing false sharing from the code actually leads to an increase in IPC and therefore, increase in parallelism.



```
=================================================
    Shared Cache Line Distribution Pareto
=================================================

----- HITM -----  -- Store Refs --  ------- CL --------                    ---------- cycles ----------    Total       cpu                 Shared
RmtHitm  LclHitm   L1 Hit  L1 Miss    Off  Node  PA cnt       Code address  rmt hitm  lcl hitm      load  records       cnt        Symbol    Object      Source:Line  Node
.......  .......  .......  .......  .....  ....  ......  ..................  ........  ........  ........  .......  ........  ............  ........  ................  ....

------------------------------------------------------------
  0        0        3        0        0      0x56379abe1740
------------------------------------------------------------
  0.00%   33.33%    0.00%    0.00%   0x10     0       1      0x5637996486c0         0       110         0        1         1  [.] classify  classify  classify.cpp:18   0
  0.00%   33.33%    0.00%    0.00%   0x18     0       1      0x5637996486c0         0       115         0        1         1  [.] classify  classify  classify.cpp:18   0
  0.00%   33.33%    0.00%    0.00%   0x1c     0       1      0x5637996486c0         0       112         0        1         1  [.] classify  classify  classify.cpp:18   0

------------------------------------------------------------
  1        0        2        1        0      0x56379ab79100
------------------------------------------------------------
  0.00%   50.00%    0.00%    0.00%   0x4      0       1      0x5637996486c0         0        99         0        1         1  [.] classify  classify  classify.cpp:18   0
  0.00%   50.00%    0.00%    0.00%   0x8      0       1      0x5637996486c0         0       109         0        1         1  [.] classify  classify  classify.cpp:18   0
  0.00%    0.00%  100.00%    0.00%   0x8      0       1      0x5637996486c5         0         0         0        1         1  [.] classify  classify  classify.cpp:18   0
```

Figure 1.7: Shared Cache Statistics

Figure 1.7 shows two cache-lines, i.e. 0x56379abe1740 and 0x56379ab79100, and code blocks that are writing to it simultaneously. This therefore, represents an instance of **false-sharing**. In the report it can be seen that there are 262 such cache lines. An important point to uncover here is that 251 of 262 cache-lines are being shared *falsely* over a single block of code, i.e. classify.cpp:18.



```
16  void Counter::increase(unsigned int id) { //
17      assert(id < _numcount);
18      _counts[id]++;
19  }
20
```

Figure 1.8: classify.cpp:18

As Figure 1.8 shows, this code is nothing but the increments that are done in the counter by different threads. One important point here is that different threads have different sub-counters, so the problem is not about accessing the same location simultaneously. However, a single cache-line is 64 bytes on my machine, and hence, in case of four threads, *four* such sub-counters are going to be present in the same cache-line (Counter is 32-byte aligned). Hence, there exists **false-sharing** between threads, even when they are writing to different memory locations.

Optimisations: One straightforward way to reduce this contention among threads, will be to pad the integer sub-counters, so that no two of them lie in the same cache line. However, this is extremely inefficient as integer is just 4 bytes as compared to 64 bytes (size of cache-line). Another way to reduce false sharing is to have different counters for different threads altogether. This can be achieved using **std::vector<std::vector<int>>**, i.e. a 2-D dynamic array of size $numt \times R.num()$, where each individual vector corresponds to counter for a different thread.

Conclusion: By doing vector-based optimisation, even if two threads need to update counter for a particular range, they will be updating memory locations which are in different cache-lines with high probability. Hence, false sharing is removed. Refer **Chapter 2** for results.

Chapter 2

Analysis of Modified Code

I used three profiling/performance measurement tools, i.e., **gprof**, **valgrind** (tool = `cachegrind`) and **perf** (`stat` and `c2c`), to analyse the efficiency of the modified code (optimised). I present my observations and results below:

2.1 Profiler: gprof

The code was compiled with `-pg` compiler flags and then run using the following command to generate `gmon.out` file:

```
./classify rfile dfile 1009072 4 3
```

The report was generated by running `gprof` command on the output file (`gmon.out`):

```
gprof -a -b ./classify > gprof_modified.txt
```

This report can be found in the `gprof/modified/` directory present in the submission. I have also attached a screenshot of the same in this document below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 91.10      1.21     1.21     2003     0.60     0.60  Ranges::range(int, bool) const
  6.02      1.29     0.08        3    26.70    28.37  readData(char const*, unsigned int)
  1.51      1.31     0.02        3     6.68    35.65  classify(Data&, Ranges const&, unsigned int)
  0.75      1.32     0.01                             repeatrun(unsigned int, Data&, Ranges const&, unsigned int)
  0.38      1.33     0.01  1009072     0.00     0.00  Item::Item()
  0.38      1.33     0.01     1000     0.01     1.21  Ranges::operator+=(Range)
  0.00      1.33     0.00     1000     0.00     0.00  Range::Range(int, int)
  0.00      1.33     0.00     1000     0.00     1.21  Ranges::newrange(Range)
  0.00      1.33     0.00        3     0.00    35.65  timedwork(Data&, Ranges const&, unsigned int)
  0.00      1.33     0.00        2     0.00     0.00  main
  0.00      1.33     0.00        1     0.00     0.00  Ranges::Ranges()
  0.00      1.33     0.00        1     0.00     0.00  Ranges::num() const
```

Figure 2.1: Flat Profile

Observations: It can be seen from Figure 2.1, that the function **Ranges::range(int, bool) const** still runs for majority of time (91.10 %). This is expected, as I did not make any change to this function. On the other hand, since, I have optimised *classify()* function, the total time spent in this function (per call) has dropped from 578.22 ms (Figure 1.1) to 35.65 ms (Figure 2.1). The *Call Graph* for the modified code conveys similar information and can be found in the `gprof_modified.txt` file (submitted).

To understand the implications of optimisation suggested in **Chapter 1**, I also implemented *binary search* in place of linear search. The *Flat Profile* obtained in that case is displayed in Figure 2.2.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 42.16      0.08     0.08        3    26.70    30.04  readData(char const*, unsigned int)
 42.16      0.16     0.08        3    26.70    26.70  Ranges::range_binary(int, bool) const
 10.54      0.18     0.02        3     6.68    63.42  classify(Data&, Ranges const&, unsigned int)
  5.27      0.19     0.01  1009072     0.00     0.00  Item::Item()
  0.00      0.19     0.00     2000     0.00     0.00  Ranges::range(int, bool) const
  0.00      0.19     0.00     1000     0.00     0.00  Range::Range(int, int)
  0.00      0.19     0.00     1000     0.00     0.00  Ranges::newrange(Range)
  0.00      0.19     0.00     1000     0.00     0.00  Ranges::operator+=(Range)
  0.00      0.19     0.00        3     0.00    63.42  timedwork(Data&, Ranges const&, unsigned int)
  0.00      0.19     0.00        2     0.00     0.00  main
  0.00      0.19     0.00        1     0.00     0.00  Ranges::Ranges()
  0.00      0.19     0.00        1     0.00     0.00  Ranges::num() const
```

Figure 2.2: Flat Profile (using binary search)

Observations: The use of *binary search* in place of linear search reduced the execution time by **60-90 ms** when using small number of threads (like 4, 8 etc.) and by at least **15 ms** when using large number of threads (like 48, 63 etc.).

Conclusion: I implemented *binary search* and used it in my modified code (function name: `range_binary`). The resulting code experienced a significant reduction in execution time. However, as the input specifications do not guarantee the correctness of binary search, I have not used it in the final submission. The implementation for the same, however, can be found in *classify.cpp*.

2.2 Profiler: valgrind

I used the *cachegrind* tool of valgrind to analyse the cache-based performance of the modified code, and compare it with that of original code
The code was compiled with `-g` compiler flag and then run using the following command to generate `cachegrind.out.4265` file:
`valgrind --tool=cachegrind ./classify rfile dfile 1009072 4 3`
The final report was generated by running `cg_annotate` command on the `.out.` file:
`cg_annotate --auto=yes cachegrind.out.4265 > valgrind_modified.txt`
This report can be found in the `valgrind/modified/` directory present in the submission. Since, the report is too big to fit in this document, I have only attached screenshots of the summary and those parts of code that had a significant number of cache-misses in original code (Figure 1.4-1.5).

```
==4265== I   refs:      16,168,243,596
==4265== I1  misses:             3,730
==4265== LLi misses:             3,570
==4265== I1  miss rate:           0.00%
==4265== LLi miss rate:           0.00%
==4265==
==4265== D   refs:       3,456,299,301  (3,349,586,943 rd   + 106,712,358 wr)
==4265== D1  misses:         3,625,881  (    2,168,338 rd   +   1,457,543 wr)
==4265== LLd misses:         2,639,044  (    1,469,949 rd   +   1,169,095 wr)
==4265== D1  miss rate:             0.1% (          0.1%     +         1.4%  )
==4265== LLd miss rate:             0.1% (          0.0%     +         1.1%  )
==4265==
==4265== LL refs:            3,629,611  (    2,172,068 rd   +   1,457,543 wr)
==4265== LL misses:          2,642,614  (    1,473,519 rd   +   1,169,095 wr)
==4265== LL miss rate:              0.0% (          0.0%     +         1.1%  )
```

Figure 2.3: Valgrind Summary

From Figure 2.3, it can be seen that the number of cache-misses and miss-rate, both have reduced (compare with Figure 1.3). This shows that I was able to successfully improve cache-based performance of the code. Below, I break up this analysis into two parts and show the results of the optimisations proposed in **Chapter 1**.

```
        24         0         0         0       0       0        while(index < D.ndata) {
 3,279,486         0         0         0       0       0           for(int j = index; j < (index + k) && j < D.ndata; j ++) {
 6,054,444   630,672    11,517 6,054,432       0       0              int v = D.data[j].value = R.range(D.data[j].key);
 6,054,444       755       141         0       0       0              counts[tid][v] ++;
         .         .         .         .       .       .           }
   252,270         0         0        12       0       0           index += k * numt;
         .         .         .         .       .       .        }
```

Figure 2.4: First omp parallel section

Optimisation: As specified in **Chapter 1**, I first optimised the code by splitting up the D.data array into chunks of 12 consecutive items ($k = 12$) and scheduled their processing on different threads. The *consecutive-ness* allows for a greater degree of spatial locality to be used effectively by each thread (upto 8-items in the same cache-line), thereby, reducing the total number of cache-misses. This can also be seen from Figure 2.4. Earlier, there were a total of 1,513,614 cache-misses in L1, now, there are only 630,672 such cache-misses. Hence, the optimisation has helped in reducing the number of cache-misses in first parallel section.

```
         0         0         0        12       0       0        int tid = omp_get_thread_num();
12,108,900         0         0         0       0       0        for(int d = 0; d < D.ndata; d ++) {
12,108,864 1,513,620 1,445,649         0       0       0           int r = D.data[d].value;
         0         0         0         0       0       0           if(tid == r % numt) {
15,136,080     2,293     1,355 6,054,432 381,042 354,842               D2.data[rangecount[r - 1] + rangeIndex[r] ++] = D.data[d];
         .         .         .         .       .       .           }
```

Figure 2.5: Second omp parallel sections

Optimisation: I optimised the code by ensuring that each thread traverses the D.data array exactly once, contrary to $\frac{R.num()}{numt}$ times in the original code. This way, there are roughly ($numt = 4$) $\times$ 380,000 cache-misses (1,513,620), which is 250 times less than that in original code. This improved the cache-based performance of the code by reducing both the number of memory accesses, as well as the number of cache-misses (both L1 and LL). Hence, the optimisation is working successfully.
Note: Number of write misses are still the same (roughly), which is expected as the D2.data is still being written into completely (as before).

Conclusion: By doing the above two optimisations, I was able to reduce a large number of cache-read-misses and improve my program's efficiency.

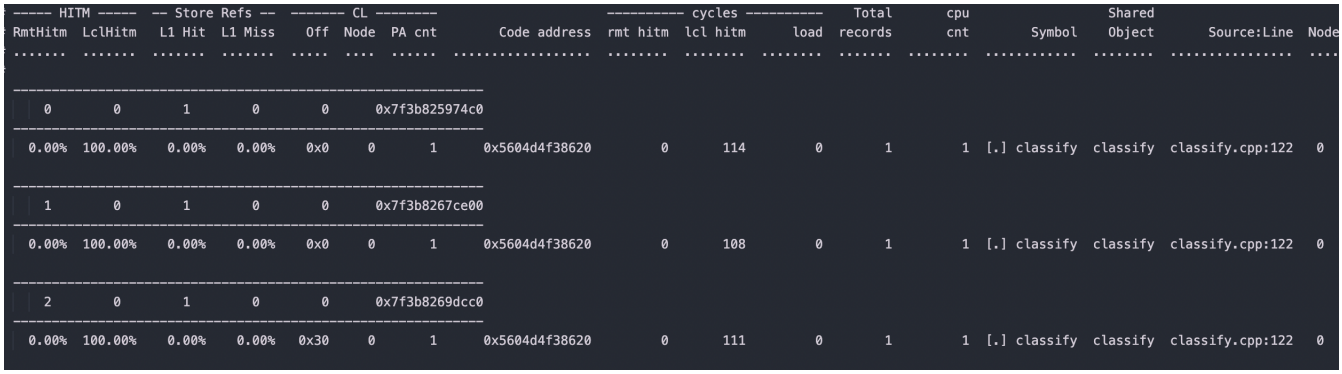
2.3 Profiler: perf

I used the *stat* and *c2c* option of perf to analyse performance of the modified code (general and cache-based). Using report generated by *c2c* tool, I was able to identify and remove instances of **false-sharing** from the original code.
The code was compiled with `-g` flag and then run using the following two commands:
`perf stat -d ./classify rfile dfile 1009072 4 3`
`perf c2c record ./classify rfile dfile 1009072 4 3`
The final report was generated by using the `report` command of perf:
`perf c2c report > perf_modified.txt`
This report can be found in the `perf/modified/` directory present in the submission. Since, the report is too big to fit in this document, I have only attached screenshots of the performance summary (stat) and cache-lines that are shared among threads.

```
Performance counter stats for './classify rfile dfile 1009072 4 3':

       1579.972389      task-clock:u (msec)       #    2.310 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
             9308      page-faults:u             #    0.006 M/sec
       3347853894      cycles:u                  #    2.119 GHz
      11605996070      instructions:u            #    3.47  insn per cycle
       4779828584      branches:u                # 3025.261 M/sec
          3227616      branch-misses:u           #    0.07% of all branches
       3332396636      L1-dcache-loads:u         # 2109.149 M/sec
          4190805      L1-dcache-load-misses:u   #    0.13% of all L1-dcache hits
           484744      LLC-loads:u               #    0.307 M/sec
           217781      LLC-load-misses:u         #   44.93% of all LL-cache hits

       0.683931409 seconds time elapsed
```

Figure 2.6: Performance Statistics

Figure 2.6 shows the output of `stat` command. It shows general performance-based statistics of the program along with cache loads and misses. As can be seen, the IPC of the program has increased to 3.47 from 3.05, which was expected as I have removed a large fraction of *false-sharing* from the code. Hence, the modified code is more effective in parallel computation as compared to the original code.



----- HITM -----		-- Store Refs --		------- CL -------				---------- cycles ----------			Total	cpu		Shared		
RmtHitm	LclHitm	L1 Hit	L1 Miss	Off	Node	PA cnt	Code address	rmt hitm	lcl hitm	load	records	cnt	Symbol	Object	Source:Line	Node
0	0	1	0	0		0x7f3b825974c0										
0.00%	100.00%	0.00%	0.00%	0x0	0	1	0x5604d4f38620	0	114	0	1	1	[.] classify	classify	classify.cpp:122	0
1	0	1	0	0		0x7f3b8267ce00										
0.00%	100.00%	0.00%	0.00%	0x0	0	1	0x5604d4f38620	0	108	0	1	1	[.] classify	classify	classify.cpp:122	0
2	0	1	0	0		0x7f3b8269dcc0										
0.00%	100.00%	0.00%	0.00%	0x30	0	1	0x5604d4f38620	0	111	0	1	1	[.] classify	classify	classify.cpp:122	0

Figure 2.7: Shared Cache Statistics

Figure 2.7 shows 3 (out of 12) cache-lines that are being shared among threads. Contrast this number with 262 as in the case of original code. This indicates that I was successful in removing *false-sharing* to a great extent in the program.

Conclusion: By doing vector-based optimisation (mentioned in **Chapter 1**), even if two threads need to update counter for a particular range, they will be updating memory locations which are in different cache-lines with high probability. Hence, false sharing is removed to a great extent.

Chapter 3

Timing Characteristics

This chapter involves some miscellaneous information regarding the original and modified code.

3.1 How to Run

- Run `make classify`
- Run `make run` or `./classify <rfile> <dfile> <nData> <numt> <reps>`

3.2 Direct Comparison

I ran both codes on HPC with different number of threads and noted their average execution time (over 10 runs). The results for two extremes are mentioned below:

- 4 threads:
  - original: **246.747 ms**
  - modified: **106.41 ms**
- 48 threads:
  - original: **59.1083 ms**
  - modified: **34.9011 ms**