COL380 Parallel and Distributed Programming
Assignment 0

Chinmay Mittal (2020CS10336)

January 16, 2023

2) Setting up and Running Perf

1) *Perf Stat*

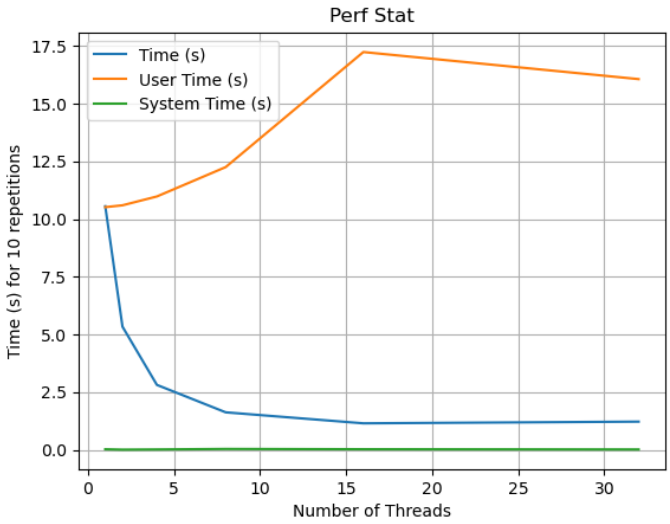

Figure 1: Perf Stat and Time statistics

Time: As we increase the number of threads, the time taken (real wall clock time) decreases in general, the minimum being at 16 threads, there is a slight increase in time for 32 threads. This might be because the CSS cluster has 16 CPUs (seen from *lscpu* command). Having more than 16 threads might not lead to any benefits.
The general trend is explained by the fact that we are dividing the computations across multiple cores each processing simultaneously decreasing the overall time. But as we increase the threads beyond the number of cores, the overhead from creating and managing threads over-weights the benefit of parallelization (limited by number of chores).
CPU cycles: The general trend is that the number of cycles increases as we increase the threads. Work distributed between different processors will be counted multiple times hence the affect of parallelization will not be seen in the number of cycles as compared to the time taken. When we create more threads, our code also has to spend CPU time to manage these threads i.e. things like creating these threads, scheduling them destroying them etc. This counts to the increase in CPU cycles with increase in number of threads.

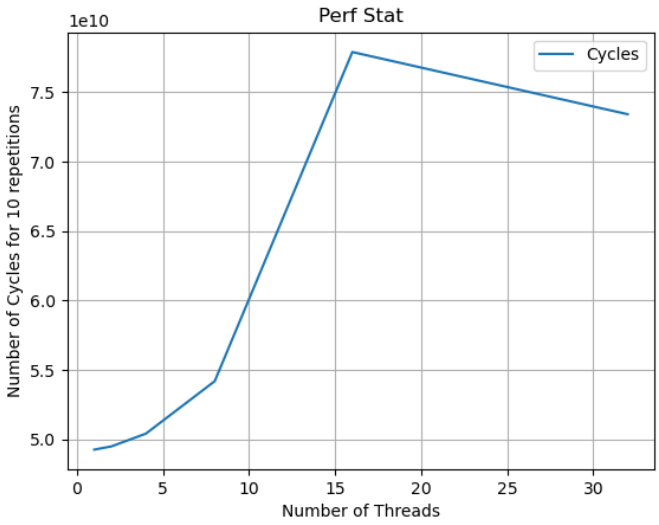

Figure 2: Perf Stat and Cycle statistics

2) *Perf Record*

Doing *perf record* and *perf report* on the base source code gives the folloing two hotspots in the code

```
Samples: 45K of event 'cycles', Event count (approx.): 52632171704
Overhead  Command   Shared Object            Symbol
  72.60%  classify  classify                 [.] classify
  21.66%  classify  classify                 [.] classify
```

Figure 3: Two code hotspots by perf record

These two correspond to two different parts of the *classify* function. These are the code blocks executed by the threads.
The first hotspot is for the following block:

```
#pragma omp parallel num_threads(numt)
{
    int tid = omp_get_thread_num(); // I am thread number tid
    for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
        int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
                                  // and store the interval id in value. D is changed.
        counts[v].increase(tid); // Found one key in interval v
    }
}
```

Figure 4: Hotspot Code Block 1

The second hotspot is for the following block:

```
#pragma omp parallel num_threads(numt)
{
   int tid = omp_get_thread_num();
   for(int r=tid; r<R.num(); r+=numt) { // Thread together share-loop through the intervals
      int rcount = 0;
      for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
         if(D.data[d].value == r) // If the data item is in this interval
            D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
   }
}
```

Figure 5: Hotspot Code Block 2

The first code block in turn calls several other functions, cycle count breakdown for which is given as follows (obtained from *–call-graph* flag to *perf record*):

```
# Children      Self  Command   Shared Object       Symbol
# ........  ........  ........  ..................  ......................
#
  100.00%     0.00%  classify  libc-2.31.so        [.] __GI___clone (inlined)
          |
          ---__GI___clone (inlined)
             |
              --100.00%--start_thread
                        |
                        |--98.67%--0x7f81a6fa878d
                        |          |
                        |          |--76.17%--classify
                        |          |          |
                        |          |          |--73.78%--Ranges::range (inlined)
                        |          |          |          |
                        |          |          |           --48.43%--Range::within (inlined)
                        |          |          |
                        |          |           --2.08%--Counter::increase (inlined)
                        |          |
                        |           --22.50%--classify
                        |
                         --1.08%--0x7f81a6fa8767
                                   |
                                    --1.02%--0x7f81a6fab130
```

Figure 6: Breakdown of Code Block 1

The corresponding hottest assembly instruction for each of the code blocks is as follows:

```
 0.38 | 40:  cmp    0x4(%r11,%rax,8),%edx
36.99 |      jg     93
 0.40 |      shl    $0x6,%rax
 0.06 |      add    %rbp,%rax
```

Figure 7: Hottest Instruction for Code block 1

The percentage taken by this instruction is 72.60*36.99 = 26.8%

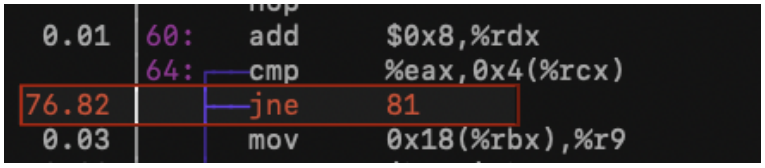

Figure 8: Hottest Instruction for Code block 2

The percentage taken by this instruction is 21.66*76.82 = 16.63%

Hence the assembly instruction which takes the most assembly time is the jg 93 jump instruction corresponding to the first code block

To find the corresponding source code instruction we need to add the *-g* flag in the *Makefile*

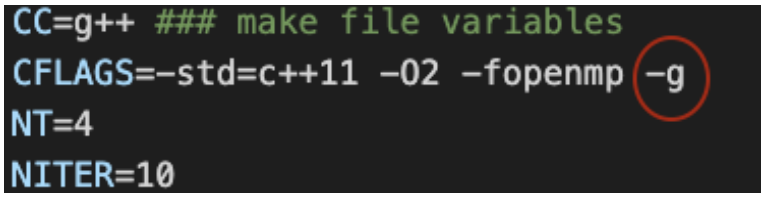

Figure 9: Compile time flag to see source code along with assembly

The *jg 93* hottest assembly instruction maps to the *return* statement in the first code block. This is in the *Range::within* function called by the *Ranges::range* function as seen from the source code along with *perf report*:

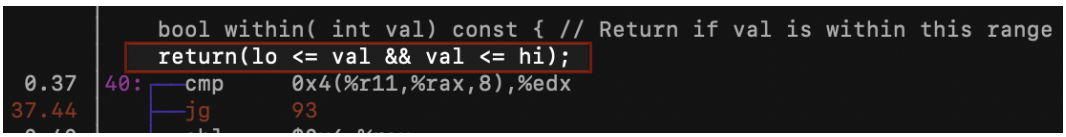

Figure 10: Source code for hottest assembly instruction

3) Hot Spot Analysis

As seen above in Figure 3 the bottleneck code is the first code block Figure 4. This code block takes 72.60 % time in total. The bottleneck instruction in this code is the branch instruction corresponding to the return statement Figure 10. The assembly code for this bottleneck return statement can be simplified as follows:

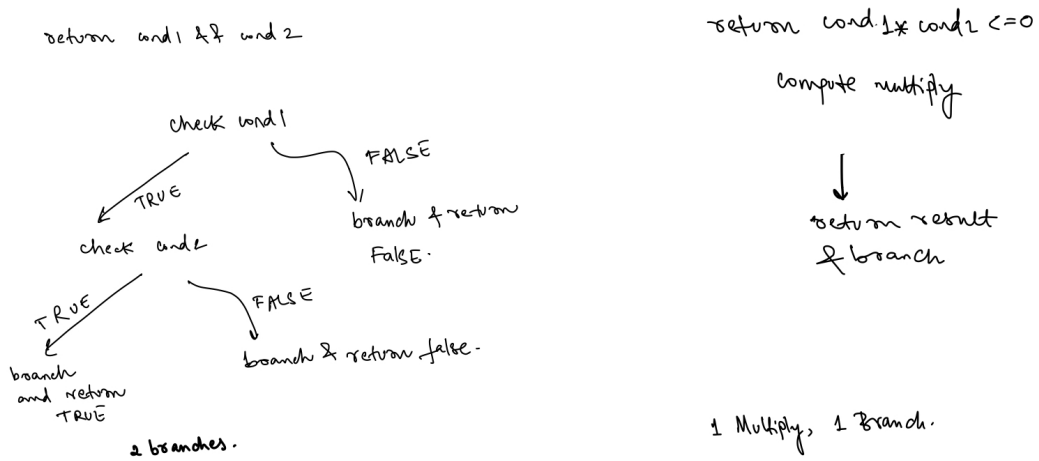

Figure 11: Assembly Code simplified for short circuited return and return with multiply

As we can see this return statement is implemented using two branch statements (due to short-circuiting of *&&*) we can improve this by implementing using an equivalent statement having only one branch instruction and a multiply instruction. This will be much more effective if the hardware-level branch prediction is ineffective.

```
bool within( long long int val) const { // Return if val is within this range
   //  return(lo <= val && val <= hi); // is equivalent to
   return (lo-val)*(hi-val) <= 0 ;
}
```

Figure 12: Optimizing Branch Mis-Prediction due to short-circuiting

The results of these optimizations are as follows:

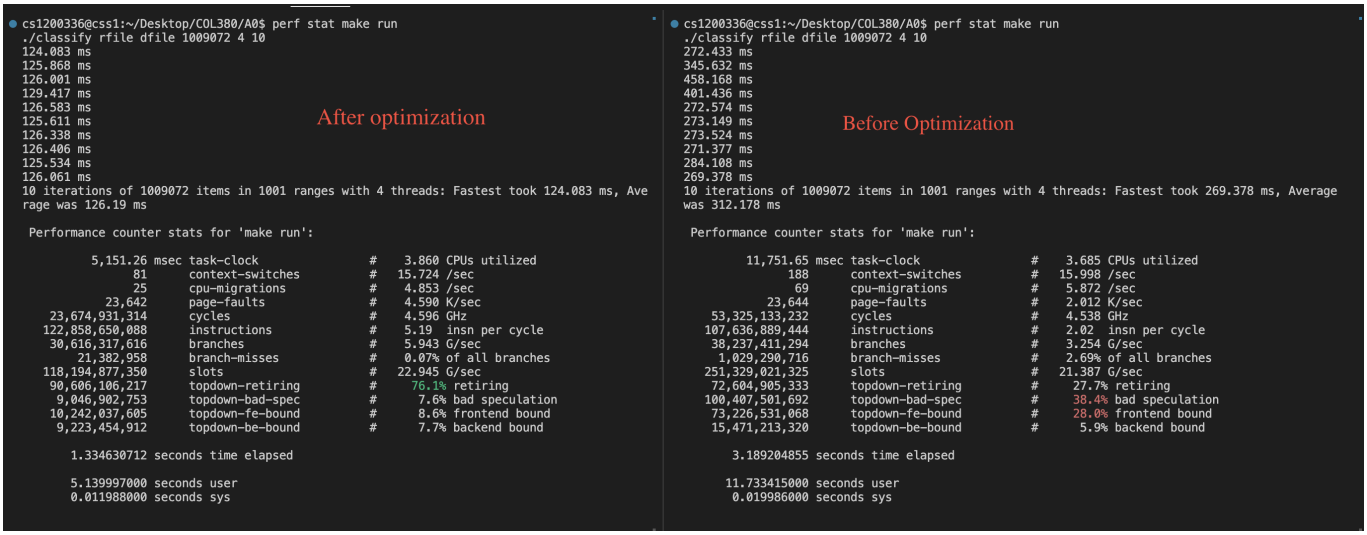

Figure 13: Optimizing Branch Mis-Prediction due to short-circuiting

The second bottleneck is algorithmic-ally inefficient and can thus be optimized, the base code being $O(|D||R|)$ in time complexity is redundant because for each range we are iterating over all possible elements in the data array. Since we already know the ranges for each element, we can instead iterate on the data itself and write to correct location without the need to iterate on all ranges making the time complexity $O(|D|)$.
The code for the following is as follows:

```
for( int d = 0 ; d< D.ndata ; d++){
    int r = D.data[d].value ;
    if(r%numt == tid){
        D2.data[rangecount[r-1]++] = D.data[d] ;
    }
}
```

Figure 14: Algorithmic Improvements

The results for the optimization are as follows:

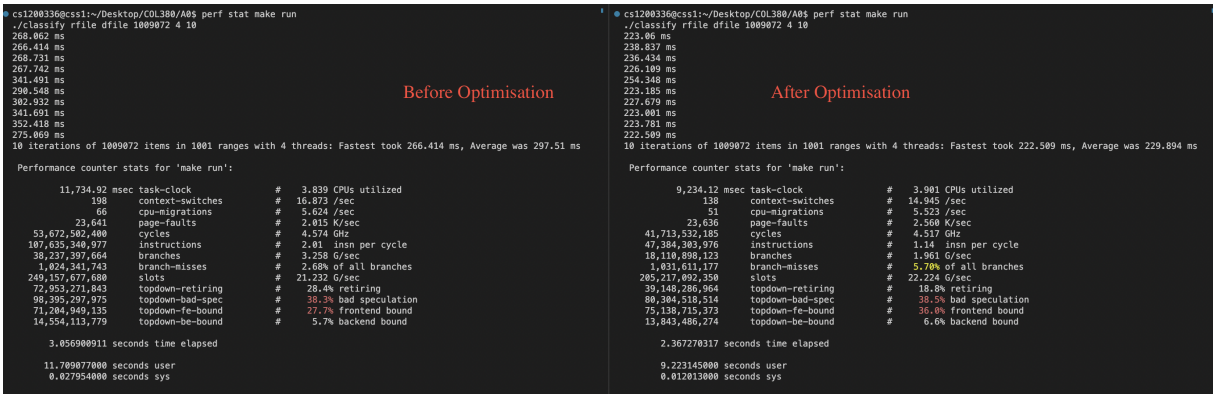

Figure 15: Results of Algorithmic Improvements

4) Memory Profiling

I ran command *perf mem record make run* and *perf report*.
Hotspots for CPU Memory Loads:

```
Samples: 235  of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 1834
Overhead  Command   Shared Object     Symbol
  72.62%  classify  classify          [.] classify
  12.99%  classify  classify          [.] classify
  12.34%  classify  [unknown]         [k] 0xffffffff9971263b
   1.19%  classify  classify          [.] classify
   0.20%  classify  [unknown]         [k] 0xffffffff996c63f2
   0.18%  classify  libc-2.31.so      [.] malloc
   0.16%  classify  [unknown]         [k] 0xffffffff9a01f45c
   0.15%  classify  [unknown]         [k] 0xffffffff99548c23
   0.09%  classify  [unknown]         [k] 0xffffffff9955b9da
   0.09%  classify  [unknown]         [k] 0xffffffff999f7c2c
```

Figure 16: Memory Load Hotspots

Hotspots for CPU Store Loads:

```
Samples: 32  of event 'cpu/mem-stores/P', Event count (approx.): 5494896
Overhead  Command   Shared Object     Symbol
  54.47%  classify  classify          [.] repeatrun
  23.11%  classify  classify          [.] classify
   5.25%  classify  [unknown]         [k] 0xffffffff997143c5
   5.06%  classify  [unknown]         [k] 0xffffffff996c4282
   4.75%  classify  [unknown]         [k] 0xffffffff99684750
   3.60%  classify  classify          [.] classify
   2.95%  classify  classify          [.] classify
   0.29%  classify  [unknown]         [k] 0xffffffff996e2cdf
   0.28%  classify  [unknown]         [k] 0xffffffff9a201017
   0.23%  classify  [unknown]         [k] 0xffffffff9a00e5c2
   0.01%  classify  [unknown]         [k] 0xffffffff9943b825
   0.00%  classify  [unknown]         [k] 0xffffffff99406140
   0.00%  classify  [unknown]         [k] 0xffffffff994763c5
   0.00%  classify  [unknown]         [k] 0xffffffff99647120
```

Figure 17: Memory Store Hotspots

Further analysis of Load Hotspots:

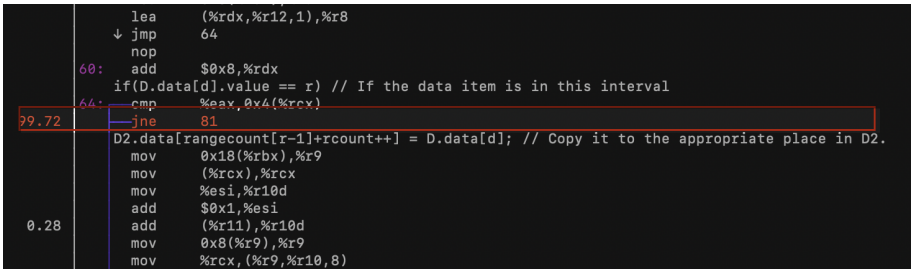

Figure 18: First Memory Load Hotspot

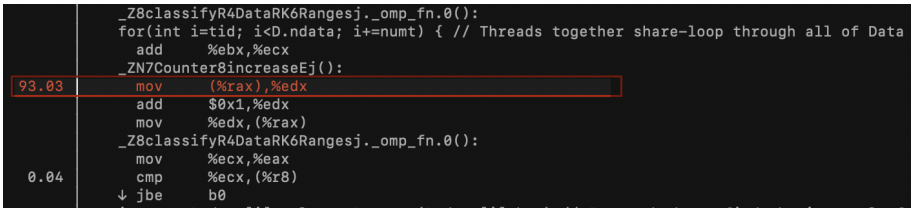

Figure 19: Second Memory Load Hotspot

False Sharing

There are several instances of false sharing in the *classify* function, arising from the fact that all threads are writing to locations which are consecutive and hence part of the same cache line thus requiring reloads from memory.

Eg. 1:

```
int tid = omp_get_thread_num(); // I am thread number tid
for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
   int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
```

Figure 20: False Sharing in Data array

Here threads are processing the data items in an interleaved manner, the first element being processed by the first thread the second element by the second thread and so on. When data is being written back (assignment to *D.data[i].value*) threads thus write to consecutive memory locations which require cache reloads from main memory.

Eg. 2:

```
counts[v].increase(tid); // Found one key in interval v
```

Figure 21: False Sharing in Counter

Another instance of false sharing in the code is the counter increment in the same loop as Figure 20. Here counter can be thought of as a 2D array, with rows as ranges and columns as the thread numbers. Thus when two different threads find elements in the same range they increment values in memory locations that are close to each other in the address space (hence part of same cache line) leading to false sharing.

Another point to note is that in all these cases threads are not exploiting the spatial locality of the cache. Each thread is writing to memory locations in the cache which are not consecutive in the address space. Eg. 3:

```
int tid = omp_get_thread_num();
for(int r=tid; r<R.num(); r+=numt) { // Thread together share-loop through the intervals
   int rcount = 0;
   for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
      if(D.data[d].value == r) // If the data item is in this interval
         D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
}
```

Figure 22: False Sharing in Processing Ranges

Here again, the ranges are being processed in an interleaved manner by all the threads, the first range being processed by the first thread, the second range being processed by the second thread and so on. Thus when the threads write to *D2.data*, they write to locations that are consecutive in the address space and hence part of the same cache line leading to false sharing.

Fixing False Sharing

False sharing for the loops can be fixed by making the threads process contiguous ranges of Data / Ranges. This prevents them from writing to addresses which are part of the same cache line.
Eg. Fixing 20

```
int tid = omp_get_thread_num(); // I am thread number tid
int size = D.ndata / numt ;
int modulo = D.ndata % numt ; // these many initial threads will process one element extra
int low_limit = tid*size + std::min(tid, modulo) ;
int high_limit = low_limit + size ;
if( tid < modulo ) high_limit ++ ;
if(tid == numt - 1 ){
   high_limit = D.ndata ;
}
for(int i = low_limit ; i < high_limit ; i ++ ){
   // this loops through all ranges to find which range contains the data item
   int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
                  // and store the interval id in value. D is changed.
   counts[tid].increase(v) ;
}
```

Figure 23: Fixing False sharing in Data array

False sharing in the counter can be fixed by rearranging the indices by making the first index the thread and the second index the range, this way all the range counts of a particular thread will be consecutive in memory.

Eg. Fixing 21

```
Counter counts[numt] ;
for( int i = 0  ; i < numt ; i ++ ){
   counts[i] = Counter(R.num()) ;
}
```

Figure 24: Fixing False sharing in the Counter

Why mitigating False Sharing is not that Effective

In experiments I did after mitigating false sharing, program run time did not decrease much. One reason is that false sharing is not the memory bottleneck of the program as can be seen from *perf mem record*. Also in all cases of false sharing, it occurs in an outer loop. For eg. in Figure 20 between memory writes to the data array (outer loop) there is also another for loop in the *range* function (inner loop). This decreases the influence of false sharing.

In 22, two threads process parts of memory which are already somewhat separated. They are separated by the number of elements in a range $\approx \frac{|D|}{|R|}$

Other Optimizations

As seen in Figure 18 the bottleneck in the memory loads occur at the branch statement (from if condition), this occurs because if the condition is true then data has to be fetched from memory to perform the subsequent operation. The condition in if is mostly false in our case, thus we will be fetching data many times when it is actually not required. A strategy to improve this is to ensure that condition is mostly true by negating the original condition and write the expensive memory fetches in the else statement. This is more effective for branch prediction leading to smaller number of memory loads.

```
for(int r = low_limit ; r < high_limit ; r ++ ){
   int rcount = 0;
   int store = rangecount[r-1] ;
   for(int d=0; d<D.ndata; d++){ // For each interval, thread loops through all of data and
      if( D.data[d].value != r) // If the data item is in this interval
         continue ;
      else
         D2.data[store+rcount++]= D.data[d]; // Copy it to the appropriate place in D2.4
   }
}
```

Figure 25: Optimizing If Else Memory Loads

Memory Optimization results

I have done the analysis only implementing the changes affecting cache utilization and not the changes mentioned before.

```
Samples: 1K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 94653
Overhead  Command   Shared Object        Symbol
  66.87%  classify  classify             [.] classify
   8.63%  classify  libstdc++.so.6.0.28  [.] std::num_get<char, std::istreambuf_i
   2.15%  classify  [unknown]            [k] 0xffffffffac6431d7
   1.33%  classify  [unknown]            [k] 0xffffffffac88fe11
   1.22%  classify  [unknown]            [k] 0xffffffffac6498c1
   1.07%  classify  [unknown]            [k] 0xffffffffac6069dd
   0.89%  classify  [unknown]            [k] 0xffffffffad361fd2
   0.87%  classify  [unknown]            [k] 0xffffffffac910f5a
   0.80%  classify  [unknown]            [k] 0xffffffffac6498e2
   0.74%  classify  [unknown]            [k] 0xffffffffac610aa1
   0.73%  classify  [unknown]            [k] 0xffffffffac610a7c
   0.70%  classify  [unknown]            [k] 0xffffffffad362085
   0.61%  classify  [unknown]            [k] 0xffffffffad4015bd
   0.57%  classify  [unknown]            [k] 0xffffffffacc8d33e
   0.47%  classify  [unknown]            [k] 0xffffffffad361fe5
   0.47%  classify  [unknown]            [k] 0xffffffffac9057fa
   0.42%  classify  [unknown]            [k] 0xffffffffac649916
   0.39%  classify  [unknown]            [k] 0xffffffffac6431fd
   0.39%  classify  classify             [.] classify
```

Figure 26: CPU load hotspots after optimization

There have been improvements compared to Figure 16. Specific Assembly instruction hot spot, again there are some improvements compared to 18:

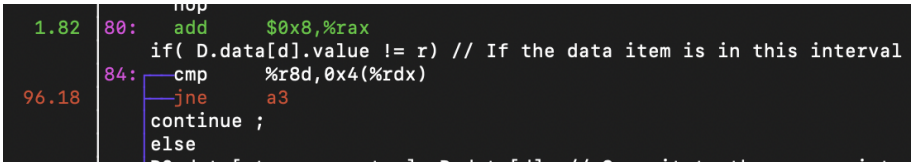

Figure 27: CPU load Assembly hotspots after optimization

Comparing cache misses:

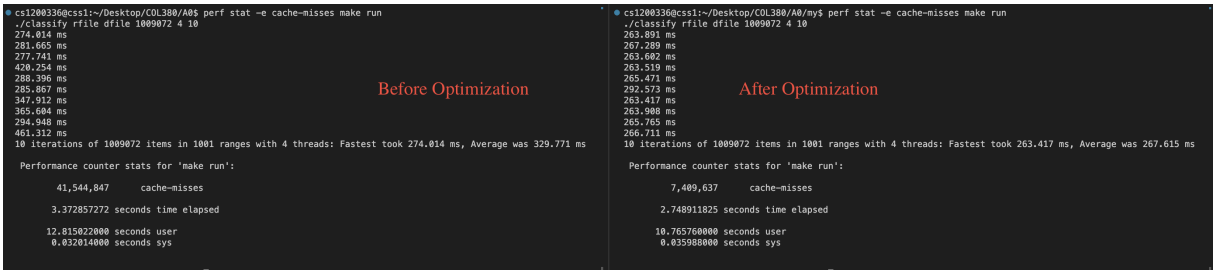

Figure 28: Results of Optimization on Cache Misses

All optimization results

Note: I have also submitted algorithmic optimizations in the final submission.

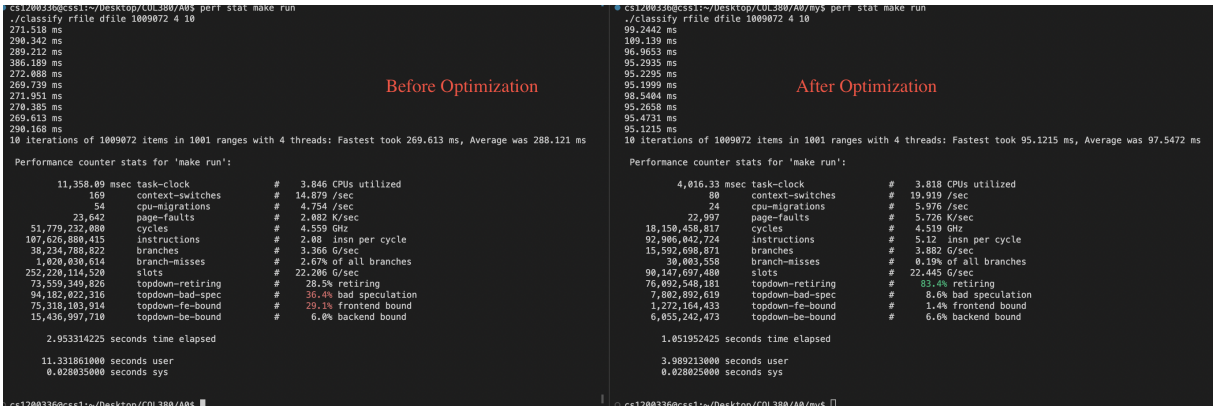

Figure 29: Results of All Optimizations Combined