Assignment 0 - COL380

Aayush Goyal

October 2021

Contents

1 Introduction

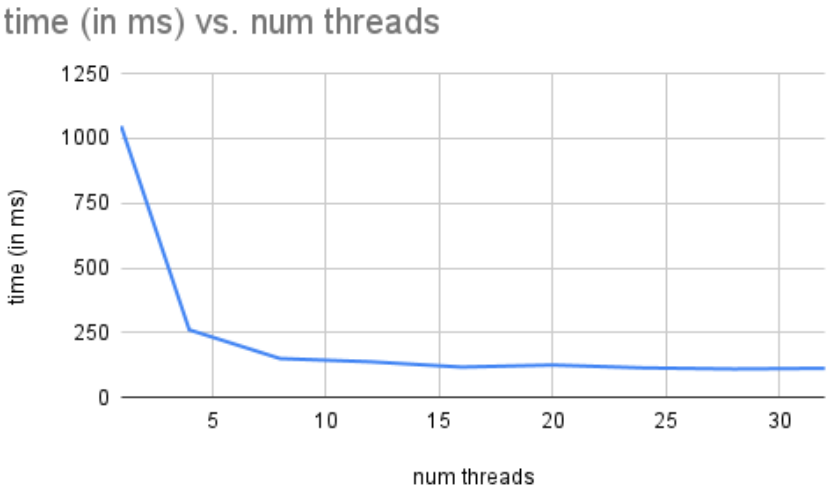
In this assignment we will use perf tool to analyze the performance of a program. We will also use the perf tool to analyze the performance of a program after doing different optimizations like removing false sharing, removing memory leaks, etc. We will compare the performance of the program before and after the optimizations based on different metrics like cache misses, branch misses, etc.

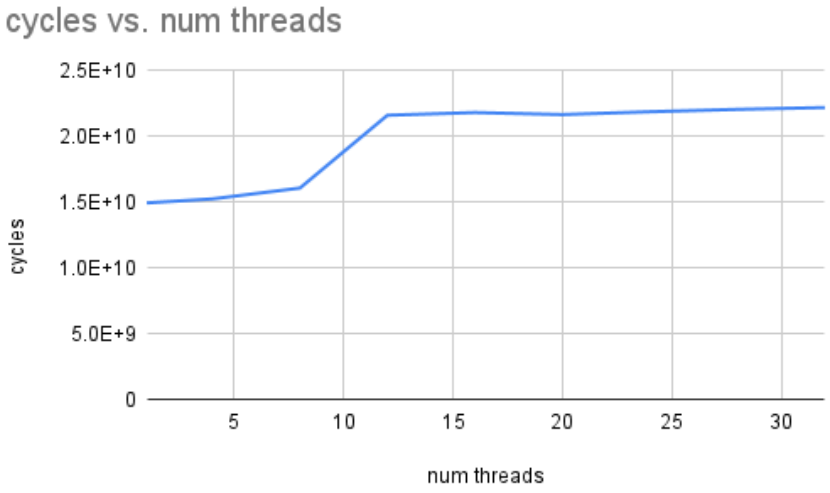
2 Setting up and Running Perf

I used css1 to check the runtime of the code. For the report files I have used perf on my local machine since it was not working on the css machine. Same issue was faced by a lot of students.

2.1 Perf stat

I have varied the number of threads as 1, 4, 8, . . . , 32 keeping the number of iterations fixed as 3. There is a significant decrease till we use 16 threads, because this is the maximum number of threads we can run in parallel on the css machine. After this point the run-time does not decrease significantly because they are not actually running on different threads. They are just running like concurrent processes after that and this doesn't reduce the run-time significantly. However using more and more threads adds the overhead of thread managaement and context switching. Hence we can observe their is an increase in the number of cycles required to run the program. So it is best to use not more than 16 threads.





2.2 Perf Record

2.2.1

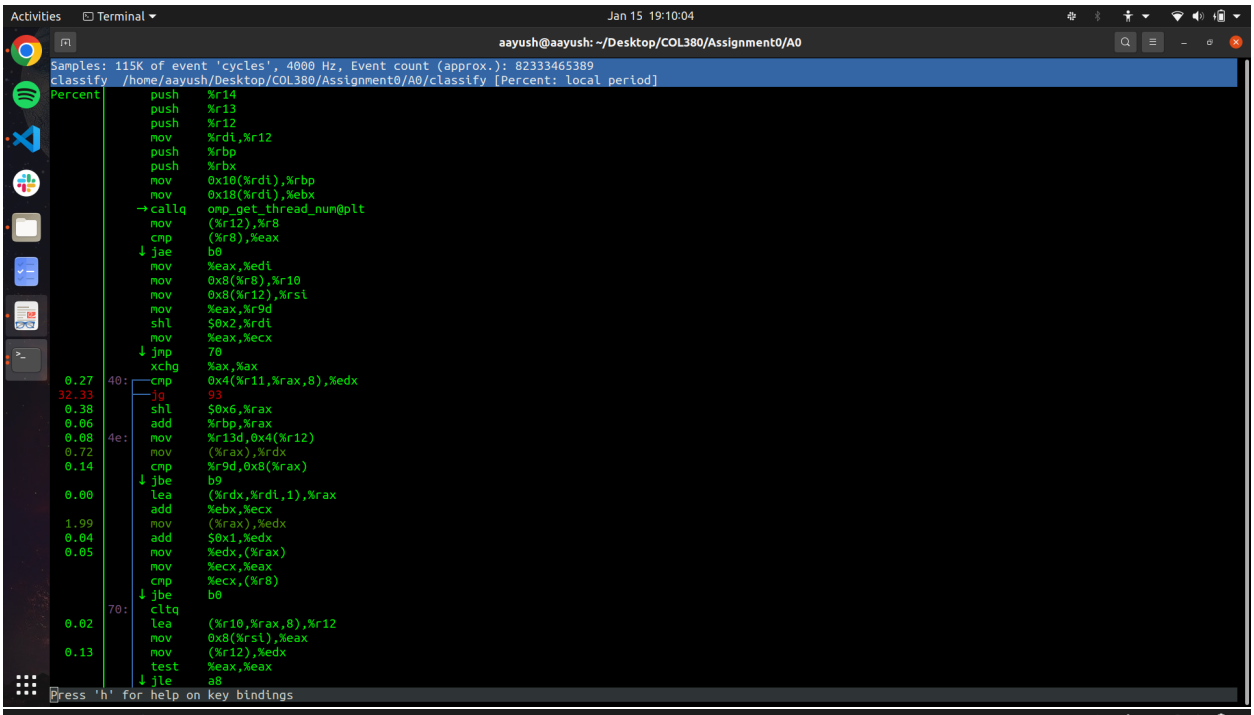
After running the `perf record make run` command, we get the `perf.data` file. I have saved it with separate name as mentioned in the assignment.

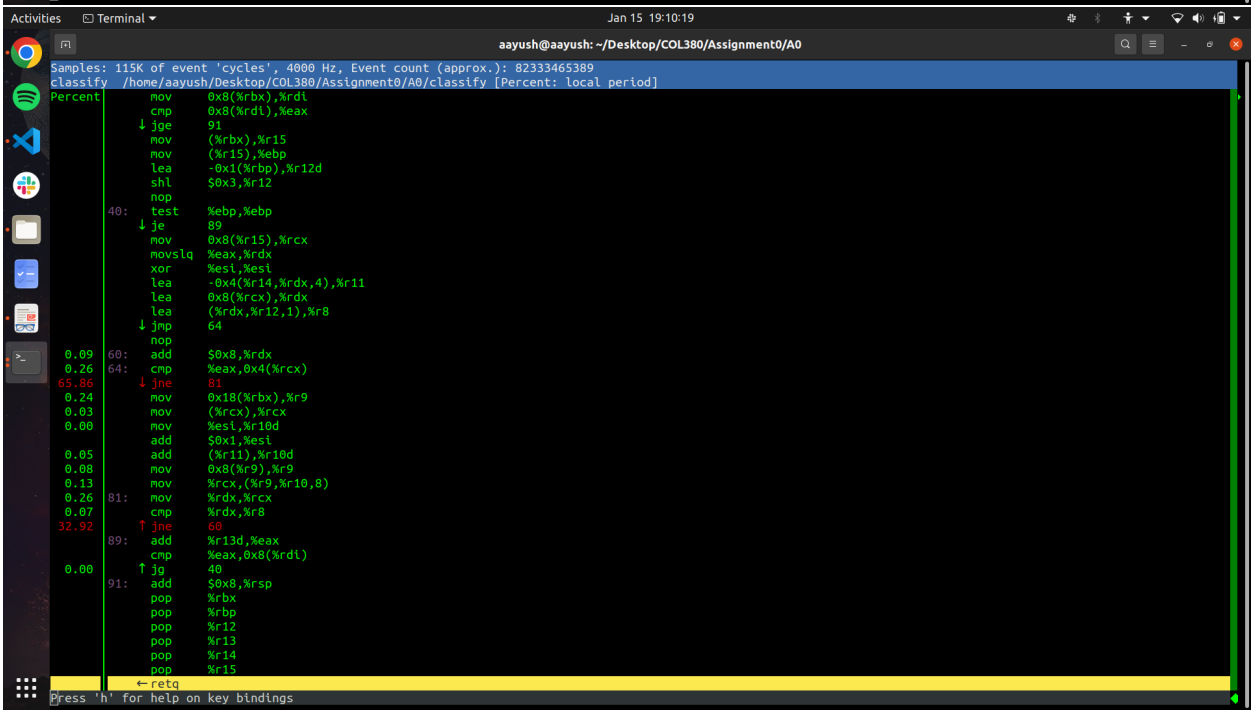
2.2.2

We obtained the perf data after running the previous command. This is not readable by humans and hence we need to use perf to read it. It is done using the following command `perf report -i <filename>`

2.2.3

The instruction marked in red takes the most amount of time. perf marks the instructions which take the most amount of time (or whatever metric we are using). `jg 93` instructions takes the most time.



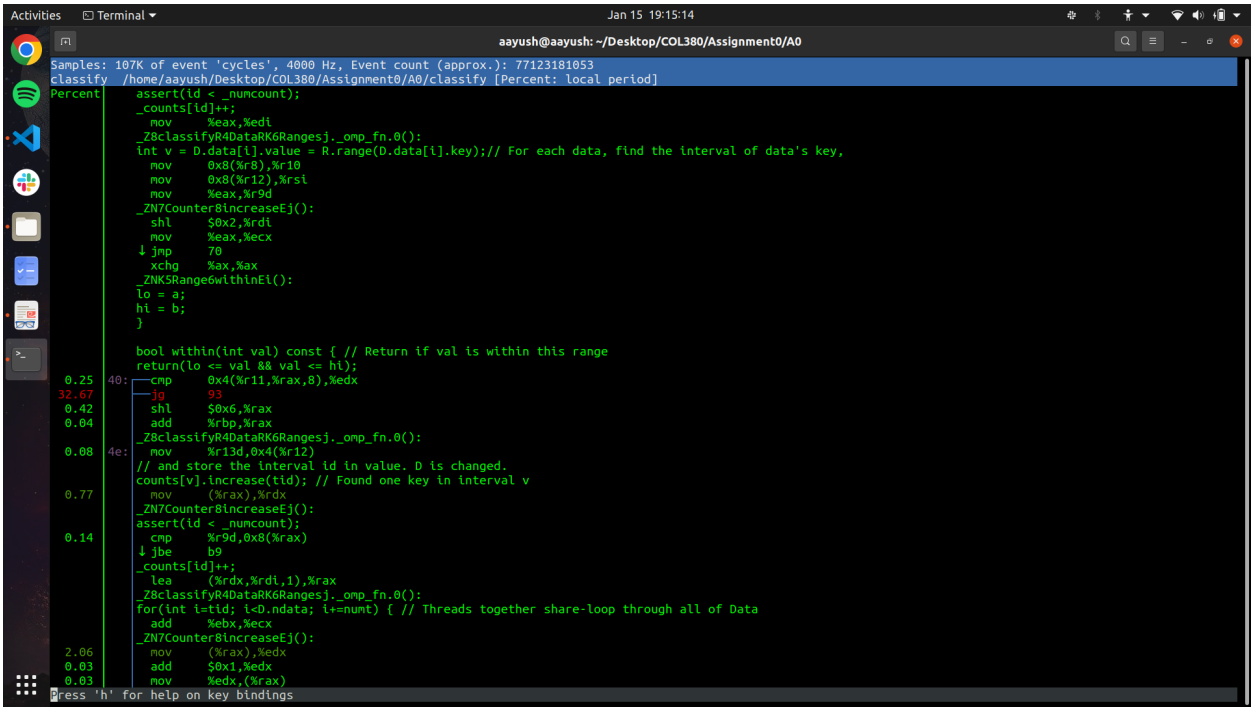


2.2.4

Yes this can be easily done by adding `-g` flag in the CFLAGS during the compilation in the Makefile. Now after this we can see the annotated assembly code along with which source code it points to.

2.2.5

Make the following change `CFLAGS=-std=c++11 -O2 -g` and we can see the souce code along with the assembly instructions.



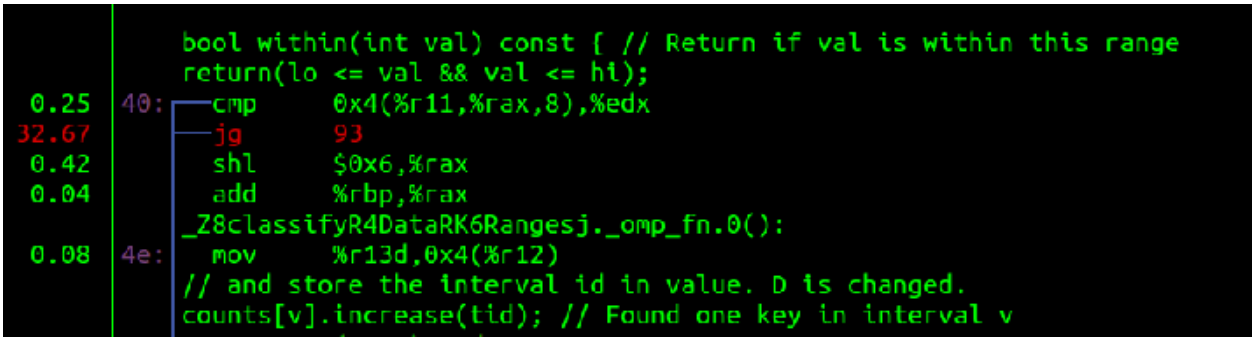
3 Hotspot Analysis

3.1

As shown in the screenshot above, the report is now showing source code along with assembly code. I have saved the perf data file with the appropriate name as mentioned in the assignment.

3.2

Most amount of time is taken by the `jg 93` instruction. This is corresponding to the `within()` function.



3.3

The above mentioned instruction is probably a hotspot because the function is getting called a lot of times. For every data item we are traversing through the entire list of ranges. Morever the excessive time being taken at this point is because of poor caache utilization and false sharing. The above need to be resolved for making it more efficient.

3.4

Yes, it is possible to optimize it further. One basic optimization possible in this case is that instead of using this as a function we can make it inline function. That is instead of doing another function call we can simply replace `within()` in the `Range::range()` function. This results in decent speedup because function calls are heavy. They require a lot of push and pop operations. Removing them will definitely make the code faster. The memory and cache related optimizations have been discussed in the further sections.

3.5

To get the list of events which perf can record, run the command `perf list`. I ran the following command: `perf record -e branch-instructions,branch-misses,cache-misses,page-faults,cpu-cycles make run` I have saved the perf.data file with the appropriate name as mentioned in the assignment.

4 Memory Profiling

4.1

I ran `perf mem record make run` and saved the report with appropriate name as mentioned.

4.2

Amongst the various hotspots, the following are the 2 major ones

```
                add     $0x8,%rdx
      if(D.data[d].value == r) // If the data item is in this interval
 0.05   cmp     %eax,0x4(%rcx)
98.49   → jne     3321 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x81>
      D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
 0.38   mov     0x18(%rbx),%r9
        mov     (%rcx),%rcx
        mov     %esi,%r10d
        add     $0x1,%esi
 0.12   add     (%r11),%r10d
 0.17   mov     0x8(%r9),%r9
        mov     %rcx,(%r9,%r10,8)
```

```
      for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
        add     %ebx,%ecx
      _ZN7Counter8increaseEj():
87.72   mov     (%rax),%edx
        add     $0x1,%edx
        mov     %edx,(%rax)
      _Z8classifyR4DataRK6Rangesj._omp_fn.0():
 0.02   mov     %ecx,%eax
 0.01   cmp     %ecx,(%r8)
        → jbe     33f0 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xb0>
```

4.3

4.4

4.5

Initially the time taken was 310.047ms Initially there were 8k events of cache misses
After optimization time is 185.252ms Now there are only 5k events of cache misses