

# Assignment 0 - COL380

Aayush Goyal

October 2021

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Setting up and Running Perf</b>	<b>1</b>
2.1	Perf stat . . . . .	2
2.2	Perf Record . . . . .	2
2.2.1	. . . . .	2
2.2.2	. . . . .	3
2.2.3	. . . . .	3
2.2.4	. . . . .	4
<b>3</b>	<b>Hotspot Analysis</b>	<b>5</b>
3.1	. . . . .	5
3.2	. . . . .	5
3.3	. . . . .	5
3.4	. . . . .	5
3.5	. . . . .	5
<b>4</b>	<b>Memory Profiling</b>	<b>5</b>
4.1	. . . . .	5
4.2	. . . . .	6
4.3	. . . . .	6
4.4	. . . . .	6
4.5	. . . . .	7
4.5.1	Runtime . . . . .	7
4.5.2	Cache Misses . . . . .	8

## 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.

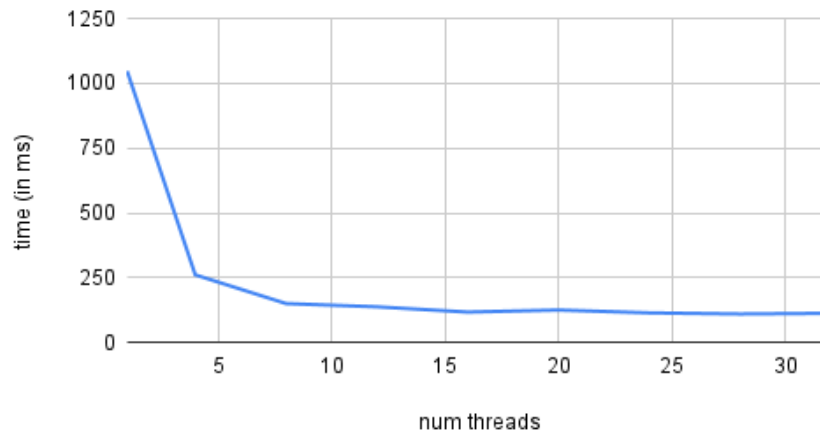
## 2 Setting up and Running Perf

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.

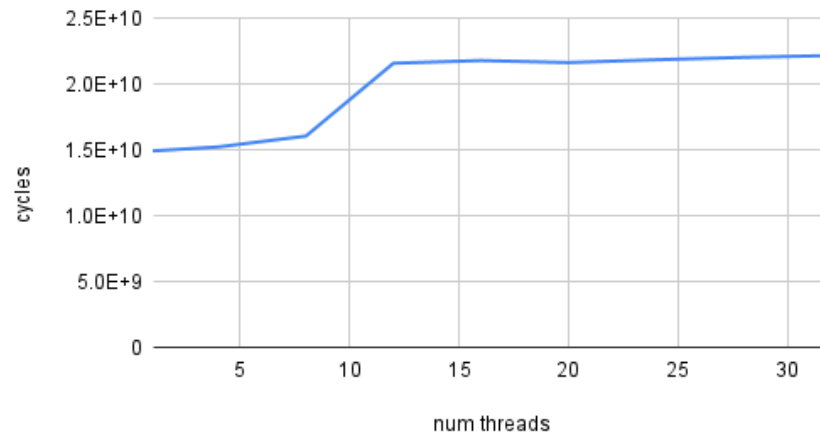
## 2.1 Perf stat

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 management and context switching. Hence we can observe an increase in the number of cycles required to run the program. So it is best to use not more than 16 threads.

time (in ms) vs. num threads



cycles vs. num 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

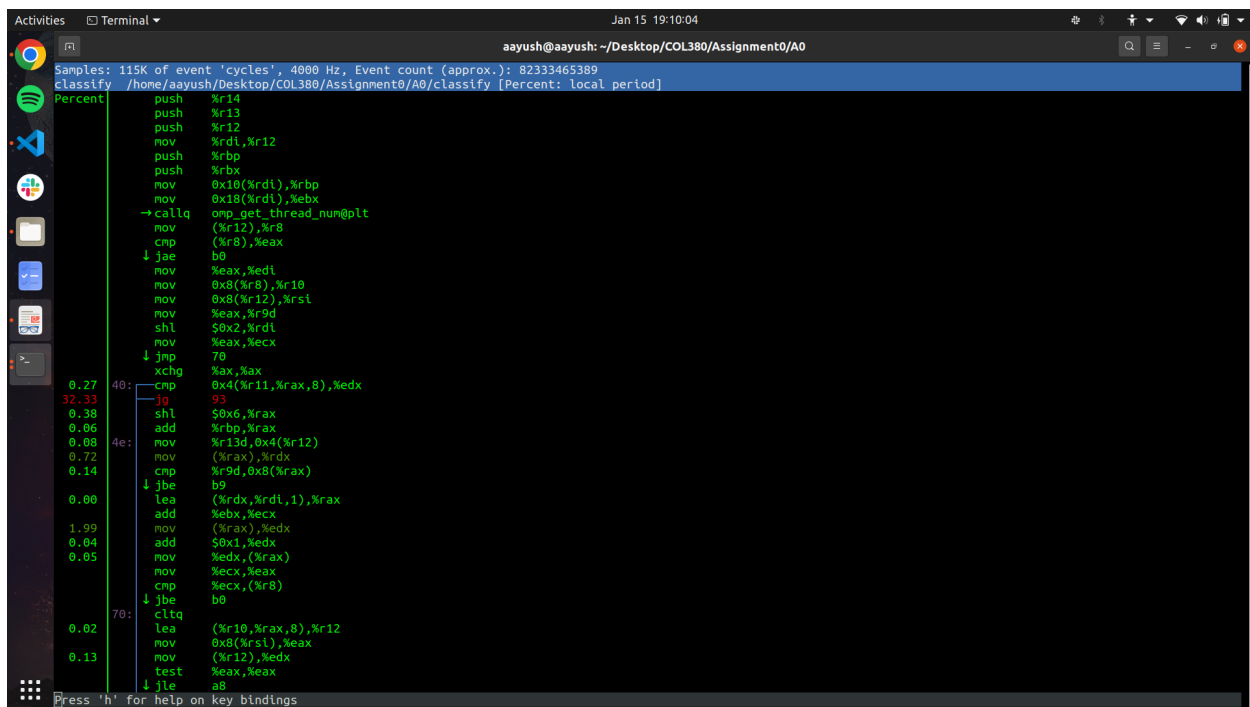
We obtained the perf data after running the previous command. This is not readable by humans and hence we need to use perf to read it. It is done using the following command `perf report -i <filename>`

### 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). `jpg` 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.



```

Samples: 115K of event 'cycles', 4000 Hz, Event count (approx.): 82333465389
classify /home/aayush/Desktop/COL380/Assignment0/A0/classify [Percent: local period]
Percent
push    %r14
push    %r13
push    %r12
mov     %rdi,%r12
push    %rbp
push    %rbx
mov     0x10(%rdi),%rbp
mov     0x18(%rdi),%ebx
→ callq  cmp_get_thread_num@plt
mov     (%r12),%r8
cmp     (%r8),%eax
↓ jae    b0
mov     %eax,%edi
mov     0x8(%r8),%r10
mov     0x8(%r12),%r10
mov     %eax,%rdi
shl     $0x2,%rdi
mov     %eax,%ecx
↓ jmp    70
xchg    %ax,%ax
40: cmp     0x4(%r11,%rax,8),%edx
32.33 ↓ jpg    93
0.38 shl     $0x6,%rax
0.06 add     %rbp,%rax
0.08 4e: mov     %r13d,0x4(%r12)
0.72 mov     (%rax),%rdx
0.14 cmp     %r9d,0x8(%rax)
↓ jbe    b9
0.00 lea     (%rdx,%rdi,1),%rax
add     %ebx,%ecx
1.99 mov     (%rax),%edx
0.04 add     $0x1,%edx
0.05 mov     %edx,(%rax)
mov     %ecx,%eax
cmp     %ecx,(%r8)
↓ jbe    b0
70: cltq
0.02 lea     (%r10,%rax,8),%r12
mov     0x8(%r12),%eax
0.13 mov     (%r12),%edx
test    %eax,%eax
↓ jle    a8
Press 'h' for help on key bindings
```

```

Samples: 115K of event 'cycles', 4000 Hz, Event count (approx.): 82333465389
classify /home/aayush/Desktop/COL380/Assignment0/A0/classify [Percent: local period]
Percent
mov 0x8(%rbx),%rdi
cmp 0x8(%rdi),%eax
jge 91
mov (%rbx),%r15
mov (%r15),%ebp
lea -0x1(%rbp),%r12d
shl $0x3,%r12
nop
40: test %ebp,%ebp
jje 89
mov 0x8(%r15),%rcx
movslq %eax,%rdx
xor %esi,%esi
lea -0x4(%r14,%rdx,4),%r11
lea 0x8(%rcx),%rdx
lea (%rdx,%r12,1),%r8
jno 64
nop
60: add $0x8,%rdx
64: cmp %eax,0x4(%rcx)
jne 81
mov 0x18(%rbx),%r9
mov (%rcx),%rcx
mov %esi,%r10d
add $0x1,%esi
mov (%r11),%r10d
mov 0x8(%r9),%r9
mov %rcx,(%r9,%r10,8)
81: mov %rdx,%rcx
cmp %rdx,%r8
jne 60
89: add %r13d,%eax
cmp %eax,0x8(%rdi)
jg 40
91: add $0x8,%rsp
pop %rbx
pop %rbp
pop %r12
pop %r13
pop %r14
pop %r15
retq
Press 'h' for help on key bindings

```

## 2.2.5

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

```

Samples: 107K of event 'cycles', 4000 Hz, Event count (approx.): 77123181053
classify /home/aayush/Desktop/COL380/Assignment0/A0/classify [Percent: local period]
Percent
assert(id < _numcount);
_counts[id]++;
mov %eax,%edi
_Z8classifyR4DataRK6Rangesj_omp_fn.0():
int v = D.data[i].value = R.Range(D.data[i].key); // For each data, find the interval of data's key,
mov 0x8(%r8),%r10
mov 0x8(%r12),%r1d
mov %eax,%r9d
_ZN7Counter8IncreaseEj():
shl $0x2,%rdi
mov %eax,%ecx
jmp 70
xchg %ax,%ax
_ZNKSRange6withinEi():
lo = a;
hi = b;
}

bool within(int val) const { // Return if val is within this range
return(lo <= val && val <= hi);
40: cmp 0x4(%r11,%rax,8),%edx
jne 93
shl $0x6,%rax
add %rbp,%rax
_Z8classifyR4DataRK6Rangesj_omp_fn.0():
mov %r13d,0x4(%r12)
// and store the interval id in value. D is changed.
counts[v].increase(tid); // Found one key in interval v
0.77: mov (%rax),%rdx
_ZN7Counter8IncreaseEj():
assert(id < _numcount);
cmp %r9d,0x8(%rax)
jbe b9
_counts[id]++;
lea (%rdx,%rdi,1),%rax
_Z8classifyR4DataRK6Rangesj_omp_fn.0():
for(int l=tid; l<D.ndata; l+=numt) { // Threads together share-loop through all of Data
add %ebx,%ecx
_ZN7Counter8IncreaseEj():
2.06: mov (%rax),%edx
0.03: add $0x1,%edx
0.03: mov %edx,%rax
Press 'h' for help on key bindings

```

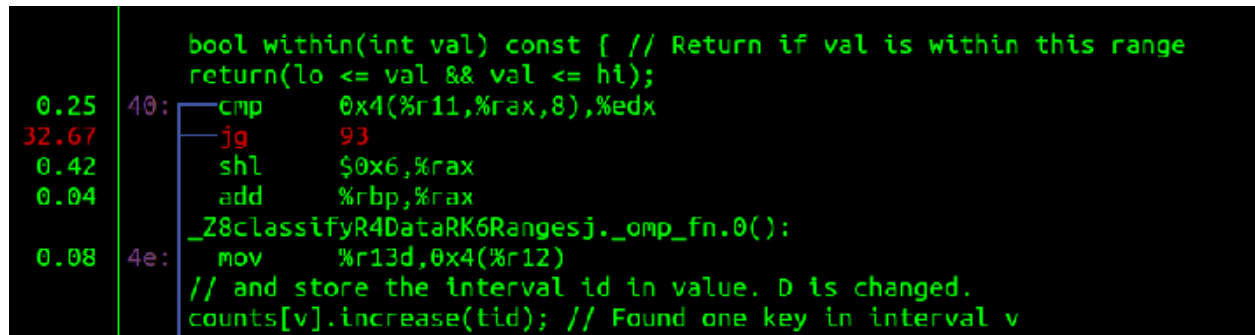
## 3 Hotspot Analysis

### 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.



```
bool within(int val) const { // Return if val is within this range
return(lo <= val && val <= hi);
0.25 40: cmp     0x4(%r11,%rax,8),%edx
32.67 41: jg      93
0.42 42: shl     $0x6,%rax
0.04 43: add     %rbp,%rax
0.08 4e: mov     %r13d,0x4(%r12)
// and store the interval id in value. D is changed.
counts[v].increase(tid); // Found one key in interval v
```

### 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. Moreover 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

1. The first hotspot is in the first for loop of classify function.

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

2. Second is in the last for loop of the classify function.

```
0.05      add     $0x0,%rcx
          if(D.data[d].value == r) // If the data item is in this interval
          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)
```

## 4.3

After reading the perf report I have analyzed different issues in the code. Their is false sharing happening and also instances of cache misses which can be avoided. The issues are:

1. In the first for loop of the classify function and the last for loop of it, data is being accessed in a cyclic fashion in the threads. All the threads are trying to update the values in the same cache line. So every time, the cache line has to be updated in all of them. This is an instance of false sharing. We are not saving any time since we are updating the same cache line in all the threads. This can be resolved by dividing the data into contiguous chunks instead of cyclic chunks.
2. Second issue is when we are updating the `count[v].increase(tid)`. Since the size of `counts[v]` array is small, the entire array gets stored in the cache. But when we are updating the `count[v].increase(tid)` in one thread, it needs to be updated in the other ones as well. Hence another instance of false sharing. This can be resolved using padding.

## 4.4

1. As discussed in the above section about the instances of false sharing, I have tried to remove them by dividing the data into contiguous chunks instead of dividing the work in a cyclic manner. By allotting a big chunk to the thread we can avoid false sharing. All the adjacent array values belong to the same thread only and hence need not be updated in the other threads. This results in a significant speedup. I have implemented this change in both the first loop of classify function and the last loop of it. The time taken is 185.252ms which is a significant improvement over the initial time taken of 310.047ms.

2. I have also removed the separate function call to `within()` and made it inline. This also reduced the run time to some extent since function calls are heavy, they require some significant amount of push and pop operations.

After doing these optimizations, I ran the `perf mem record make run` command and saved the report with appropriate name as mentioned. The hotspots earlier are not taking as much time as they were taking earlier. This is evident from the following screenshot.

1. The first hotspot is in the first for loop of classify function. This is not taking any significant percent of time now.

```

cmovb %edi,%esi
for(int i=start; i<end; ++i) { // Threads together share-loop through all of Data
  cmp %ecx,%esi
  → jle 33c8 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xb8>
  movslq %ecx,%r11
  not %ecx
  int v = D.data[i].value = R.range(D.data[i].key); // For each data, find the interval of data's key,
  mov 0x8(%r9),%r10
  mov 0x8(%rbp),%rdi
  lea (%rcx,%rsi,1),%eax
  _ZN7Counter8increaseEj():
  _counts[i] = 0;
}

```

2. Second is in the last for loop of the classify function. This is also not taking any significant percent of time now as compared to it was taking before.

```

xor %edx,%edx
int r = D.data[i].value;
11.73 movslq 0x4(%rsi,%rdx,8),%rcx
if ( start<=r && r< end ) D2.data[rangecount[r]++] = D.data[i];
  cmp %ecx,%eax
  → jg 32fd <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x5d>
  cmp %ecx,%r10d
  → jle 32fd <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x5d>
  lea 0x0(%rbp,%rcx,4),%r8
0.37 mov 0x10(%rbx),%rdi

```

Thus the optimizations have been successful in reducing the time taken by the hotspot of the code. The mem accesses have also decreased. All this has collectively resulted in decreased time taken by the code.

## 4.5

### 4.5.1 Runtime

Initially the time taken was 310.047ms

```

./classify rfile dfile 1009072 4 10
319.743 ms
318.498 ms
313.901 ms
339.938 ms
310.047 ms
312.495 ms
311.574 ms
314.727 ms
312.773 ms
310.254 ms
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 310.047 ms, Average was 316.395 ms

```

After optimization time is 185.252ms

```

./classify rfile dfile 1009072 4 10
186.129 ms
185.941 ms
185.252 ms
185.523 ms
187.021 ms
187.108 ms
186.603 ms
186.791 ms
187.515 ms
186.709 ms
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 185.252 ms, Average was 186.459 ms
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.216 MB perf.data (5600 samples) ]
cs1190452@css3:~/COL380/Assignment0/A0$ perf report
cs1190452@css3:~/COL380/Assignment0/A0$

```

Hence we can see after doing the optimizations there is a significant decrease in the time taken.

#### 4.5.2 Cache Misses

Initially there were 8k events of cache misses

Samples: 8K of event 'cache-misses', Event count (approx.): 4446008			
Overhead	Command	Shared Object	Symbol
31.12%	classify	classify	[.] 0x00000000000003321
14.43%	classify	classify	[.] 0x00000000000003300
5.85%	classify	[unknown]	[k] 0xfffffffff8ddf90c0
4.18%	classify	classify	[.] 0x000000000000033d3
2.11%	classify	[unknown]	[k] 0xfffffffff8d2c6c18

Now there are only 5k events of cache misses after optimization

Samples: 5K of event 'cache-misses', Event count (approx.): 4559236			
Overhead	Command	Shared Object	Symbol
13.43%	classify	classify	[.] 0x000000000000032fd
11.56%	classify	classify	[.] 0x000000000000032ea
6.11%	classify	classify	[.] 0x000000000000032d0
4.74%	classify	[unknown]	[k] 0xfffffffff8ddf90c0
4.36%	classify	classify	[.] 0x000000000000032e3
3.71%	classify	classify	[.] 0x000000000000032d9
3.33%	classify	classify	[.] 0x00000000000003399
2.27%	classify	classify	[.] 0x00000000000003301

Hence there have been a significant decrease in the number of cache misses, which has resulted in a significant decrease in the time taken.