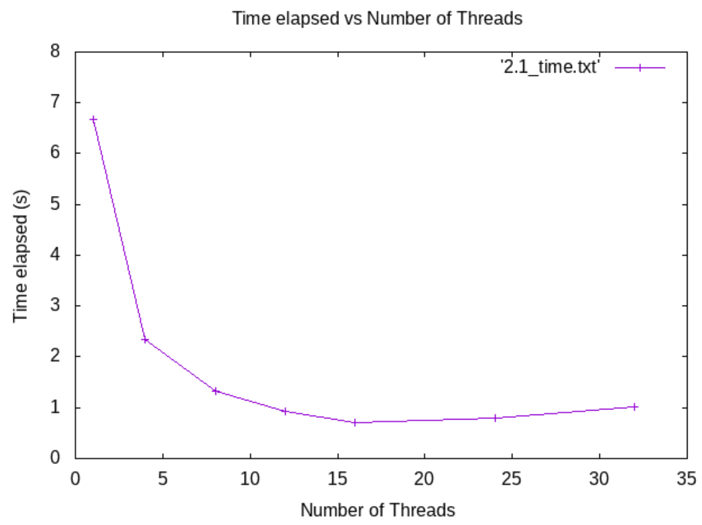
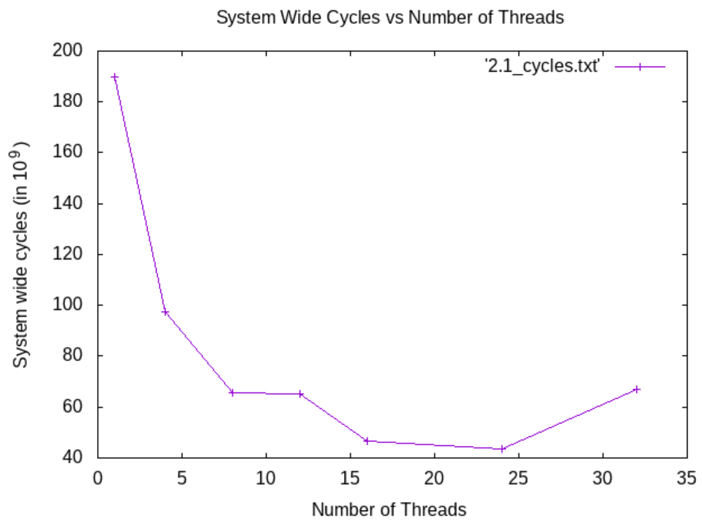
1 Perf Stat

Note: All readings here are done for 6 runs of the code, i. e. the default number of runs.





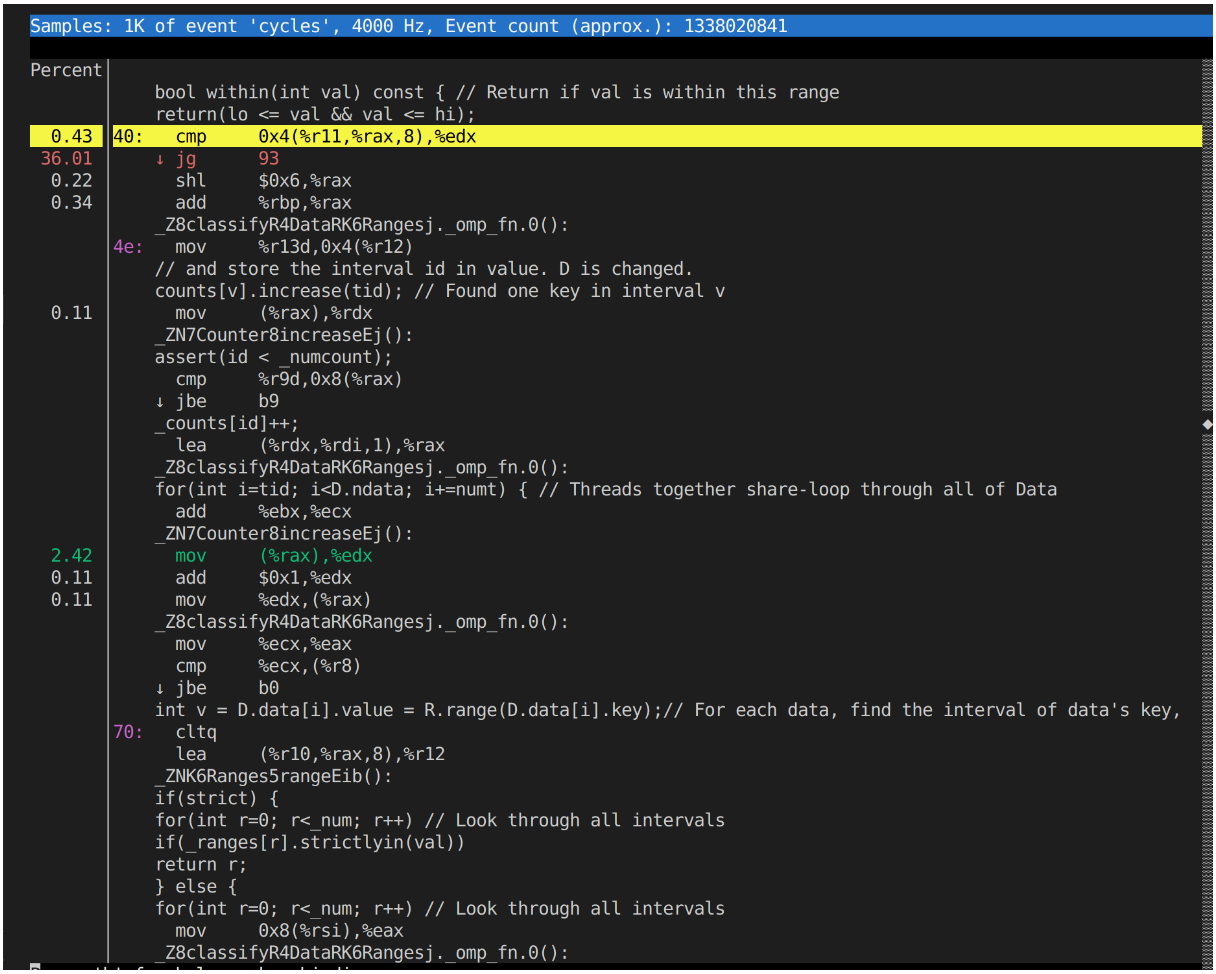
The pattern observed is that both system-wide cycles and time elapsed decrease when the number of threads increase from 1 but start increasing when the number crosses a certain limit (Which appears to be 16 threads here).
The reason for this is that initially, the threads are able to effectively use the parallelism available on the cluster machine by assigning each thread to a different core/CPU which reduces the overall runtime and hence the number of cycles. However, when the number of threads becomes too high, each thread cannot be assigned to a different core/CPU so the whole task cannot be done parallelly.

2 Perf Record

The assembly instruction which takes the most time (37.95%) is **jg 93**.
Using the **-g** compile flag in the **CFLAGS** variable in the makefile, we can see the source code along with the assembly which allows us to determine the source code portion this instruction maps to.
The part of the source code which maps to this instruction is:

```
if(_ranges[r].within(val))
    return r;
bool within(int val) const { // Return if val is within this range
   return(lo <= val && val <= hi);
}
```

3 Hotspot Analysis

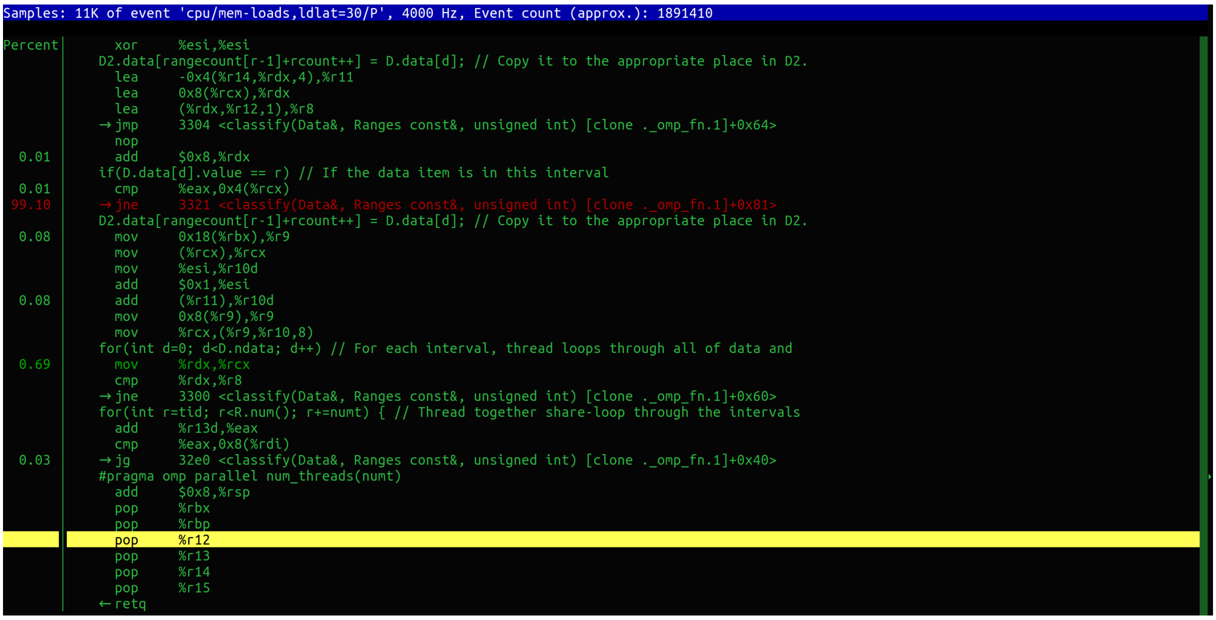


```
Samples: 1K of event 'cycles', 4000 Hz, Event count (approx.): 1338020841

Percent│
       │     bool within(int val) const { // Return if val is within this range
       │     return(lo <= val && val <= hi);
  0.43 │40:   cmp      0x4(%r11,%rax,8),%edx
 36.01 │    ↓ jg       93
  0.22 │      shl      $0x6,%rax
  0.34 │      add      %rbp,%rax
       │     _Z8classifyR4DataRK6Rangesj._omp_fn.0():
       │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
  0.11 │      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 i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
       │      add      %ebx,%ecx
       │     _ZN7Counter8increaseEj():
  2.42 │      mov      (%rax),%edx
  0.11 │      add      $0x1,%edx
  0.11 │      mov      %edx,(%rax)
       │     _Z8classifyR4DataRK6Rangesj._omp_fn.0():
       │      mov      %ecx,%eax
       │      cmp      %ecx,(%r8)
       │    ↓ jbe      b0
       │     int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
       │70:   cltq
       │      lea      (%r10,%rax,8),%r12
       │     _ZNK6Ranges5rangeEib():
       │     if(strict) {
       │     for(int r=0; r<_num; r++) // Look through all intervals
       │     if(_ranges[r].strictlyin(val))
       │     return r;
       │     } else {
       │     for(int r=0; r<_num; r++) // Look through all intervals
       │      mov      0x8(%rsi),%eax
       │     _Z8classifyR4DataRK6Rangesj._omp_fn.0():
```

The problem which makes this portion of code the bottleneck lies in the classification algorithm. The classification algorithm uses linear search to determine the range a value lies in which causes this function to be called much more than it needs to.
The code can be optimized by changing the linear search to a binary search.

4 Memory Profiling



```
Samples: 11K of event 'cpu/mem-loads,ldlat=30/P', 4000 Hz, Event count (approx.): 1891410

Percent│       xor     %esi,%esi
       │     D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
       │       lea     -0x4(%r14,%rdx,4),%r11
       │       lea     0x8(%rcx),%rdx
       │       lea     (%rdx,%r12,1),%r8
       │     → jmp     3304 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x64>
       │       nop
  0.01 │       add     $0x8,%rdx
       │     if(D.data[d].value == r) // If the data item is in this interval
  0.01 │       cmp     %eax,0x4(%rcx)
 99.10 │     → 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.
  0.08 │       mov     0x18(%rbx),%r9
       │       mov     (%rcx),%rcx
       │       mov     %esi,%r10d
       │       add     $0x1,%esi
  0.08 │       add     (%r11),%r10d
       │       mov     0x8(%r9),%r9
       │       mov     %rcx,(%r9,%r10,8)
       │     for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
  0.69 │       mov     %rdx,%rcx
       │       cmp     %rdx,%r8
       │     → jne     3300 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x60>
       │     for(int r=tid; r<R.num(); r+=numt) { // Thread together share-loop through the intervals
       │       add     %r13d,%eax
       │       cmp     %eax,0x8(%rdi)
  0.03 │     → jg      32e0 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x40>
       │     #pragma omp parallel num_threads(numt)
       │       add     $0x8,%rsp
       │       pop     %rbx
       │       pop     %rbp
       │       pop     %r12
       │       pop     %r13
       │       pop     %r14
       │       pop     %r15
       │     ← retq
```

From the above image we can see that the top 2 hotspots in the code are **jne 3321** (99.10% of time) and **mov %rdx,%rcx** (0.69% of time).
The instruction **jne 3321** corresponds to:

```
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.
```

The instruction **mov %rdx,%rcx** corresponds to:

```
for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
```

In the portion of the code shown in the first hotspot, each thread works on the data points which have modulo of the thread ID. This is cache unfriendly because different threads are attempting to access contiguous portions of memory, i. e. Thread 0 accesses address 0, thread 1 accesses address 1, etc. which are likely on the same cache line which leads to the threads accessing the same cache line concurrently leading to false sharing.
This problem can be made cache friendly by having each thread itself do a contiguous portion of the work, i. e. for 100 samples and 4 threads, thread 0 does 0-24 and so on. This leads to minimization of concurrent same cache line accesses by different threads.
On observing the code, we see another case of false sharing when the interval of the data's key is found, having the same issue as the previous case and therefore the same solution.

On optimizing we get:

```
Percent│     nop
       │   int rcount = 0;
       │   for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
       │     mov     (%rbx),%rdx
       │     mov     (%rdx),%eax
       │     test    %eax,%eax
       │   → je      3343 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0xa3>
       │     mov     0x8(%rdx),%rdx
       │     lea     -0x1(%rax),%esi
       │     lea     0x8(%rdx),%rax
       │     lea     (%rax,%rsi,8),%r8
       │   int rcount = 0;
       │     xor     %esi,%esi
       │   → jmp     331c <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x7c>
       │     nop
       │     add     $0x8,%rax
       │   if(D.data[d].value == r) // If the data item is in this interval
  48.22│     cmp     0x4(%rdx),%ecx
       │   → jne     333b <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x9b>
       │   D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
   1.55│     mov     0x18(%rbx),%r10
  49.37│     mov     (%rdx),%rdx
       │     mov     %esi,%r11d
       │     add     $0x1,%esi
   0.46│     add     -0x4(%rbp,%r9,4),%r11d
   0.39│     mov     0x8(%r10),%r10
       │     mov     %rdx,(%r10,%r11,8)
       │   for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
       │     mov     %rax,%rdx
       │     cmp     %rax,%r8
       │   → jne     3318 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x78>
       │   for(int r=start; r<start+((tid==numt-1)?div+mod:div); r++) { // Thread together share-loop through th
       │     add     $0x1,%r9
       │     mov     %r9d,%ecx
       │     cmp     %r9d,%edi
       │   → jg      32f8 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x58>
       │   #pragma omp parallel num_threads(numt)
       │     pop     %rbx
       │     pop     %rbp
       │     pop     %r12
       │     pop     %r13
```

Initial cache misses: 4746131
Final cache misses: 4177119
We see an improvement in cache misses.