Assignment 0

Shreyansh Singh

January 16, 2023

2.1 Perf Stat

In the total time elapsed vs number of threads graph, the total time decreases as the number of threads increases. This is because as the number of threads increases, the time required decreases due to parallel processing. Different threads do different tasks concurrently instead of doing every task sequentially. But when the number of threads is greater than the number of CPUs (there are 16 CPUs), then the time taken slightly increases because performance may suffer as a result of the CPU switching between running various threads when there are more threads than available cores. This is due to the fact that the CPU must save the current thread's state, load the state of the subsequent thread, and then resume execution when switching between threads. This procedure, known as a "context switching," adds overhead and may reduce the program's overall speed.

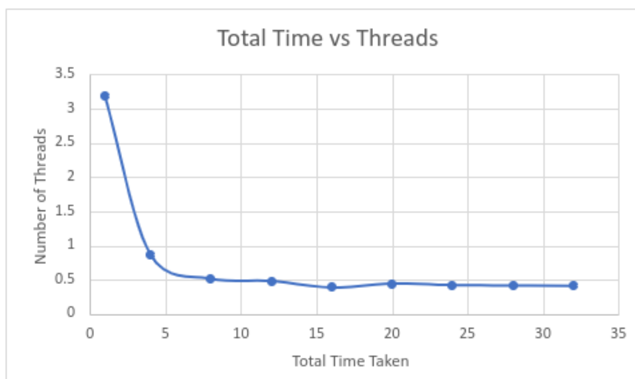

Figure 1: Total Time vs Threads

In general, as the number of threads increases, the number of cycles also increases, as employing more threads may result in a task requiring more cycles to finish. As number of threads increases, the task is broken into multiple sub-tasks, which decreases the time required, but managing threads requires extra work, which increases the number of cycles required to do the job and there is a decrease in the number of cycles when the number of threads is greater than the number of CPUs.

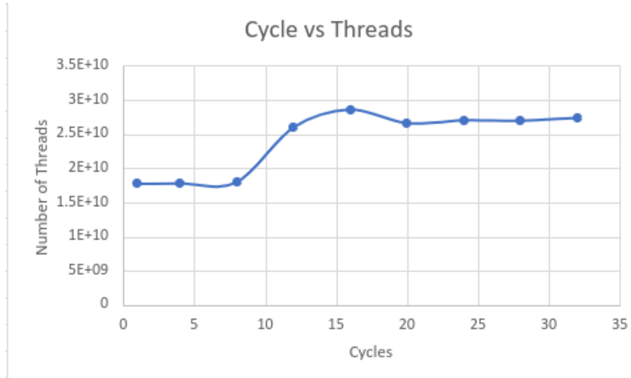

Figure 2: Cycles vs Threads

2.2 Perf Record

3. Assembly instruction, which takes the most time, is `jg 93`
4. This instruction corresponds to "return(lo <= val && val <= hi);" in within function in the classify.h file.
5. To show the source code along with the assembly instructions, we have to add the -g flag to CFLAGS in Makefile.

3 Hotspot Analysis

2. Hotspot is:

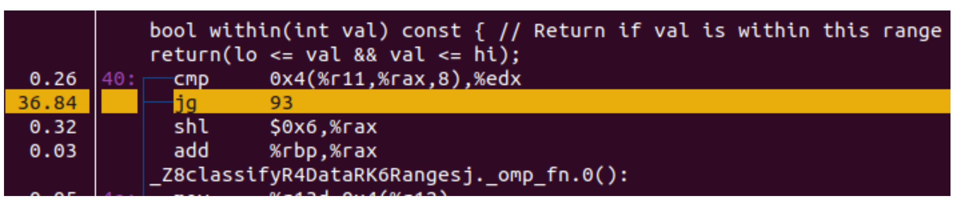

Figure 3: Hotspot

3. There are 2 comparisons and a logical AND operation that takes a lot of time to execute, and this function is called a number of times, which makes this instruction the hotspot.
4. Yes, the within function can be optimized. Optimised code:

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

In this code, we multiply the difference between val and lo with the difference between val and hi and if this value is non-positive, then val is in the range. This code snippet avoids the use of the logical AND operator and comparisons.
The original code executed in 267.551 milliseconds, whereas this update takes 72.33 milliseconds, and within function is no longer the top hotspot.

4 Memory Profiling

2. The two hotspots are:

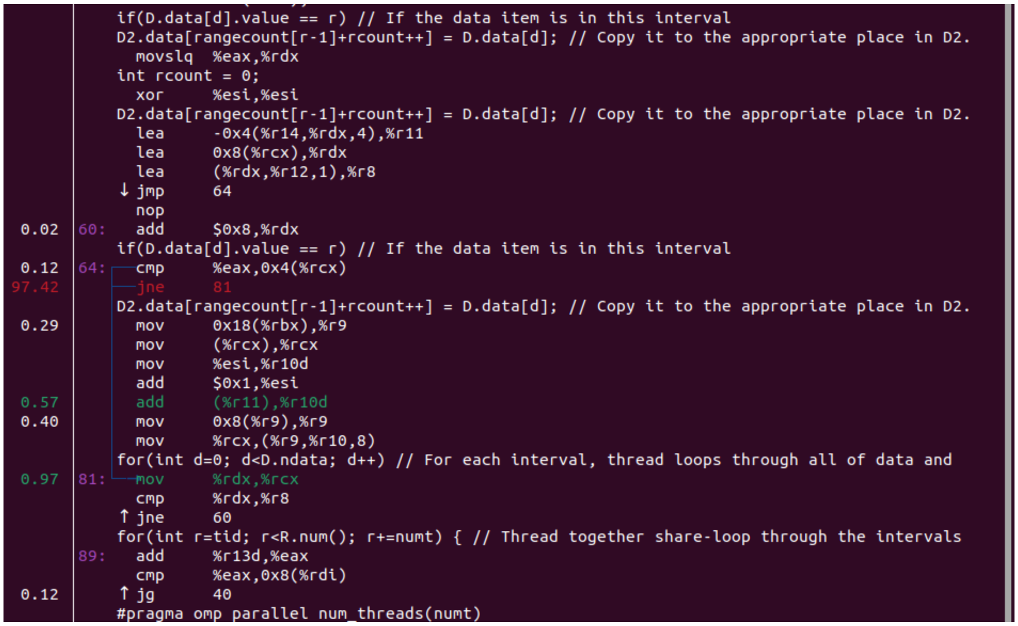

Figure 4: Hotspot 1

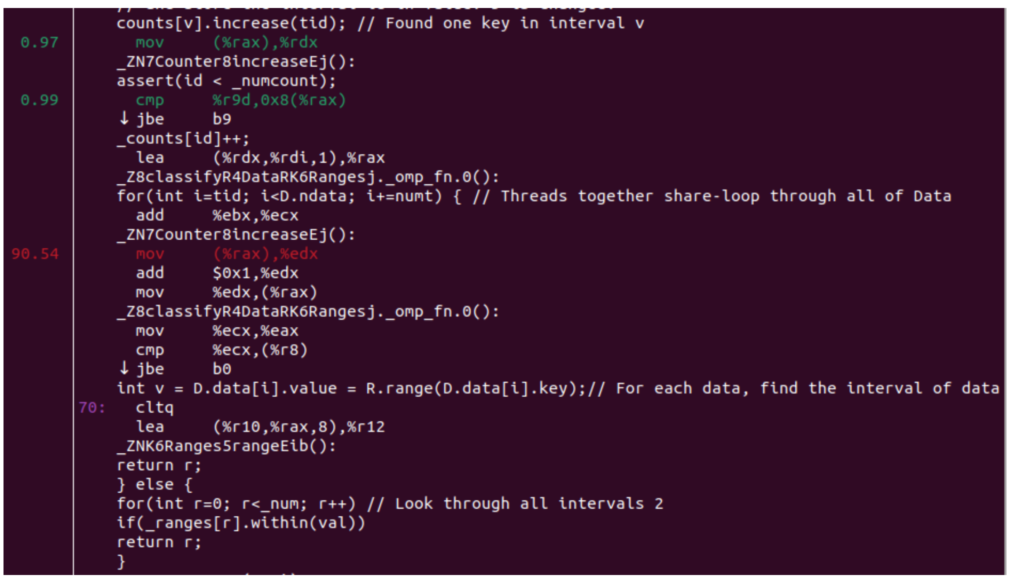

Figure 5: Hotspot 2

3. Cache unfriendly code is :

```
#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.
  }
}
```

Due to its non-locality memory access, this code is cache-unfriendly. The inner loop iterates through all of the data items while the outside loop iterates in a non-sequential fashion through the intervals. As a result, the cache is not being used effectively, which significantly reduce performance.

4. The time taken by original code is:

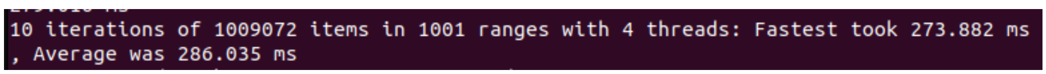

Figure 6: Original Time

To handle the false sharing in the code snippet mentioned in part 3, we can replace it with following code:

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

The time taken after this improvement is :

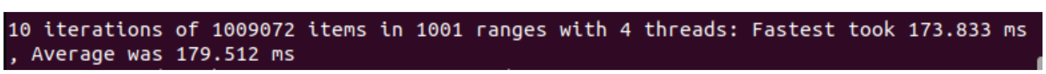

Figure 7: Time

Now the hotspot is:

```
       │    for(int d=0; d<D.ndata; d++){
       │      mov      (%rbx),%edx
       │      test     %edx,%edx
       │    → je       35f3 <classify(Data&, Ranges const&, unsigned int)+0x273>
       │      mov      0x8(%rbx),%rcx
       │      sub      $0x1,%edx
       │      lea      0x4(%rcx),%rax
       │      lea      0xc(%rcx,%rdx,8),%rsi
       │      xchg     %ax,%ax
       │    D2.data[rangecount[D.data[d].value-1]] = D.data[d];
 42.45 │      movslq   (%rax),%rdx
 34.60 │      mov      -0x4(%rax),%rcx
       │      add      $0x8,%rax
  1.50 │      mov      -0x4(%r13,%rdx,4),%edx
       │      mov      %rcx,(%r8,%rdx,8)
       │    rangecount[D.data[d].value-1]++;
 14.24 │      movslq   -0x8(%rax),%rdx
  2.68 │      addl     $0x1,-0x4(%r13,%rdx,4)
       │    for(int d=0; d<D.ndata; d++){
       │      cmp      %rax,%rsi
       │    → jne      35d0 <classify(Data&, Ranges const&, unsigned int)+0x250>
       │    }
       │
       │    return D2;
       │    }
```

Figure 8: Hotspot

After optimizing the within function as done in Hotspot Analysis, the final time taken is:

```
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 10.9478 ms
, Average was 12.3879 ms
```

Figure 9: Time