

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.

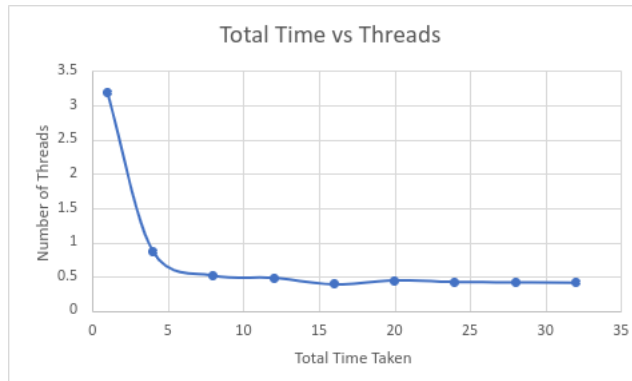


Figure 1: Total Time vs Threads

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.

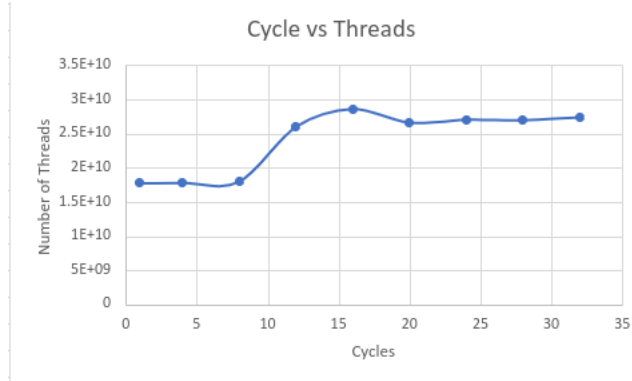


Figure 2: Cycles vs Threads

2.2 Perf Record

3. Assembly instruction, which takes the most time, is `jq 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:

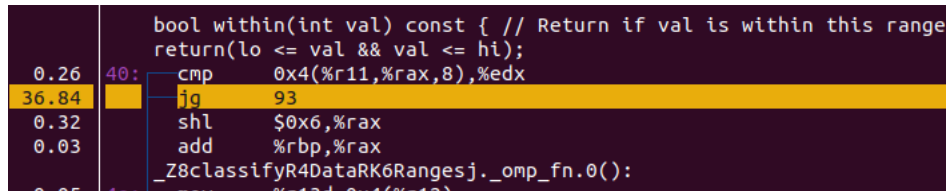


Figure 3: Hotspot

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:

```

// If the data item is in this interval
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.
movslq %eax,%rdx
int rcount = 0;
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 64
nop
0.02 60: add $0x8,%rdx
if(D.data[d].value == r) // If the data item is in this interval
0.12 64: cmp %eax,0x4(%rcx)
97.42 jne 81
D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
0.29 mov 0x18(%rbx),%r9
mov (%rcx),%rcx
mov %esi,%r10d
add $0x1,%esi
0.57 add (%r11),%r10d
0.40 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
0.97 81: mov %rdx,%rcx
cmp %rdx,%r8
↑ jne 60
for(int r=tid; r<R.num(); r+=numt) { // Thread together share-loop through the intervals
89: add %r13d,%eax
cmp %eax,0x8(%rdi)
0.12 ↑ jg 40
#pragma omp parallel num_threads(numt)

```

Figure 4: Hotspot 1

```

// The total number of keys in the interval is changed
counts[v].increase(tid); // Found one key in interval v
0.97 mov (%rax),%rdx
_ZN7Counter8increaseEj():
assert(id < _numcount);
0.99 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():
90.54 mov (%rax),%edx
add $0x1,%edx
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
70: cltq
lea (%r10,%rax,8),%r12
_ZNK6Ranges5rangeEib():
return r;
} else {
for(int r=0; r<_num; r++) // Look through all intervals 2
if(_ranges[r].within(val))
return r;
}

```

Figure 5: Hotspot 2

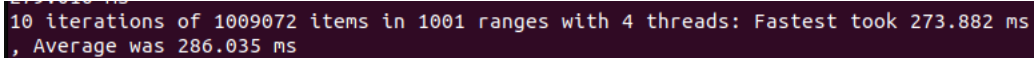
3. Cache unfriendly code is :

```
#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:



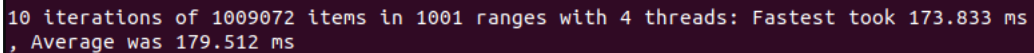
```
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 273.882 ms
, Average was 286.035 ms
```

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 :



```
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 173.833 ms
, Average was 179.512 ms
```

Figure 7: Time

Now the hotspot is:

	<pre> 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]; movslq (%rax),%rdx 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; } </pre>
--	---

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