COL380: Assignment 0

Harshit Mawandia

January 15, 2023

1 Introduction

For this assignment we have used `perf`, which is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple commandline interface. Perf is based on the perf events interface exported by recent versions of the Linux kernel.

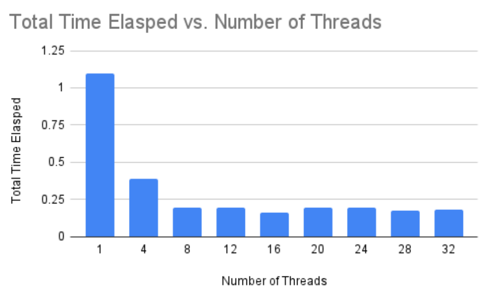
2 Setting up and Running Perf

We run different perf commands to analyse the given code and find the hotspots to optimise them.

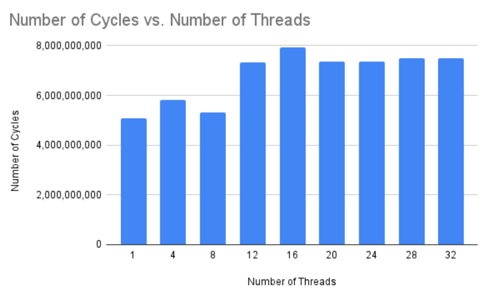
2.1 Perf stat

The perf stat command instruments and summarizes key CPU counters (PMCs). We vary the number of threads to find the plots of the time elapsed and the number of cycles.
For this section we have kept the **number of repetitions to 1**.

We plot the graph of Time elapsed vs Number of threads and Number of Cycles and Number of Threads.



(a) Plot of Total time elapsed vs Number of Threads



(b) Plot of Number of Cycles vs Number of Threads

We can see that initially the time elapsed decreases by the same factor as which the number of threads increases, while the number of cycles remain almost constant. After 12 threads, we see that the number of cycles increases and stabilises while the time take remains almost constant. The reason for this is that initially threads divide the cycles amongst themselves and reduces the time taken, but after a while concurrency cannot be achieved with too many threads so we see that time taken does not improve any more as the resources like CPU and memory are limited, but the number of cycles increases due to extra work for using the threads and synchronising them.

2.2 Perf Record

Now we use 4 threads and 10 repetitions for solving the subsequent problems.

1. We use `perf record -o perf_1.data` to generate perf_1.data submitted along with this report.
2. We use `perf report -i perf_1.data` with annotate option to inspect the perf_1.data file
3. The assembly instruction that takes the most time is
   `jg 33d3 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x93>`
   which uses **37.7%** of the time.
4. This code corresponds to the instruction `return(lo <= val && val <= hi);`
5. We add the `-g` flag to `CFLAGS=-std=c++11 -O2 -fopenmp -g` to view source code with assembly instructions.

3 Hotspot Analysis

1. We use `perf record -o perf_2_1.data` to generate perf_2_1.data submitted with this report.
2. We can see that as we reported in Section 2.2, the code
   `jg 33d3 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x93>`
   takes the most time

```
       │ bool within(int val) const { // Return if val is within this range
       │ return(lo <= val && val <= hi);
  0.30 │   cmp     0x4(%r11,%rax,8),%edx
 37.35 │ → jg      33d3 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x93>
  0.40 │   shl     $0x6,%rax
  0.07 │   add     %rbp,%rax
       │ _Z8classifyR4DataRK6Rangesj._omp_fn.0():
  0.06 │   mov     %r13d,0x4(%r12)
       │ // and store the interval id in value. D is changed.
       │ counts[v].increase(tid); // Found one key in interval v
```

Command that takes the most time

3. The problem with the code snippet that makes it the top hotspot is that it has the most number of cycles, i.e., it runs for the most number of times.
4. The code can be optimised to improve performance of this hotspot by improving the algorithm efficiency thereby reducing the number of cycles for which this code runs. If $|R|$ is the number of ranges and $|D|$ is the number of data entries, then this part of code currently has $|R| \cdot |D|$ number of iterations. This can be reduced to $|D|$ or $|D| \cdot log(|R|)$ number of iterations by using Hashmaps to store the ranges or using binary search to find the range for a particular data value.
5. We use `perf record` with `-e` or event flags to gain the output file. The command we use is
   `perf record -o perf_2_2.data -e branches,branch-misses,cache-misses,cpu-cycles,`
   `cycles,page-faults make run`

4 Memory Profiling

In this section we try to analyse the instances of false sharing and try to optimise them in our code by analysing the cache misses in the code and which part has the most instances of it.

1. I used the command `perf mem record -o perf_3.data` to generate the required report.
2. The total number of mem loads are 8000+. The top 2 hotspots for most cache misses are:

```
       │  for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
       │    add      %ebx,%ecx
       │  _ZN7Counter8increaseEj():
 93.12 │    mov      (%rax),%edx
```

Hotspot-1 with 45% of total cache misses

```
       │  if(D.data[d].value == r) // If the data item is in this interval
       │    cmp      %eax,0x4(%rcx)
 95.08 │  → 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.
```

Hotspot-2 with 26.4% of total cache misses

3. The main reasons why the given hotspots are cache unfriendly are:
   - There are instances of false sharing, the data being accessed are not thread private.
   - The threads try to access data that are not aligned, so it results in higher cache misses as cache lines are unaligned
   - Alligned allocation
4. We try to optimise the code in three steps.
   - After 1st improvement we see that the 1st hotspot has vanished but the 2nd hotspot doesn't change. The total number of mem loads are still 8000+ as there is a new hotspot created.

```
       │  for(int i=start; i<end && i<total; i++) { // Threads together share-loop through all of Data
       │    cmp      %r10,%rdx
       │  → je       3400 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xc0>
```

Hotspot-1 vanishes but new Hotspot arises

```
bjdump:│  if(D.data[d].value == r) // If the data item is in this interval
       │    cmp      %eax,0x4(%rcx)
  95.08│  → jne      3321 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x81>
```

Hotspot-2 remains the same

```
Samples: 8K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 219152
Overhead  Command   Shared Object        Symbol
  45.63%  classify  classify             [.] classify
  30.71%  classify  classify             [.] classify
   6.07%  classify  libstdc++.so.6.0.28  [.] std::num_get<char, std::istreambu
```

Total 8000+ mem-loads, Number of hotspots = 3

- After 2nd improvement we see that the new hotspot has vanished but the 2nd hotspot doesn't change, but the cache-misses now reduce from **8000+ to 3000+**. The 2nd hotspot now becomes the 1st Hotspot.

```
       if(D.data[d].value == r) // If the data item is in this interval
0.03     cmp     %eax,0x4(%rcx)
98.75  → jne     3321 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x81>
```

Hotspot-2 remains same

```
Samples: 3K of event 'cpu/mem-loads,ldlat=30/P', Event count
Overhead  Command   Shared Object          Symbol
  74.10%  classify  classify               [.] classify
   7.36%  classify  libstdc++.so.6.0.28    [.] std::num_get<ch
```

Total mem-loads reduce to 3000, Number of hotspots = 2

- After 3rd improvement we see that the number of mem-loads reduces from **3000 to 1000**. The 2nd Hotspot also improves by a large margin and goes from 2200 mem-loads to around 500 mem-loads

```
Samples: 1K of event 'cpu/mem-loads,ldlat=30/P', Event count (appro
Overhead  Command   Shared Object          Symbol
  54.27%  classify  classify               [.] classify
  14.63%  classify  libstdc++.so.6.0.28    [.] std::num_get<char, std
```

Total mem-loads reduce to 1000

Now explaining what I have done in these three improvements

- Firstly, we try to access data in sequential order in the first for loop in classify.cpp

```
#pragma omp parallel num_threads(numt)
{
   int tid = omp_get_thread_num(); // I am thread number
   for(int i=tid; i<D.ndata; i+=numt) { // Threads toget
      int v = D.data[i].value = R.range(D.data[i].key);/
                         // and store the interval id in valu
      counts[v].increase(tid); // Found one key in inter
      // std::cout << "Thread " << tid << " found " << D
   }
}
```

(a) Initial Code

```
#pragma omp parallel num_threads(numt)
{
   int tid = omp_get_thread_num(); // I am thread numbe
   int total = D.ndata;
   int chunk = total/numt;
   int start = tid*chunk;
   int end = (tid+1)*chunk;
   for(int i=start; i<end && i<total; i++) { // Threads
      int v = D.data[i].value = R.range(D.data[i].key);
                         // and store the interval id in val
      counts[v].increase(tid); // Found one key in inte
      // std::cout << "Thread " << tid << " found " << 
   }
}
```

(b) 1st improvement

- In the second improvement, I have changed the counts[] array from `int[R.num()][tnum]` to `int[tnum][R.num()]` so that when we use `counts[tid].increase(v);` in a thread, we access the same horizontal array for each thread.

- In the third and last improvement, we again try to access data sequentially in the last for loop in the file classify.cpp. We make the following changes.

```
#pragma omp parallel num_threads(numt)
{
    int tid = omp_get_thread_num();
    for(int r=tid; r<R.num(); r+=numt) { // Thread together sh
        int rcount = 0;
        for(int d=0; d<D.ndata; d++) // For each interval, thre
            if(D.data[d].value == r) // If the data item is in
                D2.data[rangecount[r-1]+rcount++] = D.data[d];
    }
}
```

(a) Initial Code

```
#pragma omp parallel num_threads(numt)
{
    int tid = omp_get_thread_num();
    int total = R.num();
    int chunk = total/numt;
    int start = tid*chunk;
    int end = (tid+1)*chunk;
    for(int r=start; r<end && r<R.num(); r++) { // Thread tog
        int rcount = 0;
        for(int d=0; d<D.ndata; d++) // For each interval, thr
            if(D.data[d].value == r) // If the data item is in
                D2.data[rangecount[r-1]+rcount++] = D.data[d];
    }
}
```

(b) Final Code

5. We use the command `perf record -e cache-misses make run` to obtain the required perf data files.

5 Submission

The submission directory is as follows:

```
2020CS10348/
2020CS10348/A0
2020CS10348/A0/Makefile
2020CS10348/A0/classify.cpp
2020CS10348/A0/classify.h
2020CS10348/A0/perf_1.data
2020CS10348/A0/perf_2_1.data
2020CS10348/A0/perf_2_2.data
2020CS10348/A0/perf_3.data
2020CS10348/A0/perf_4.data
2020CS10348/A0/perf_5_1.data
2020CS10348/A0/perf_5_2.data
2020CS10348/report.pdf
```

The name of the zip folder is `2020CS10348.zip`