CS 610 Semester 2024–2025-I
Assignment 2

Divyansh
(210355)

6th September 2024

1 Problem 1

1.1 Performance Bug identification

We start finding performance bug in the `problem.cpp` code by using `perf c2c` hardware counter at a sampling rate of `75000` Hz. Here is the report generated by `perf`:

```
Total Load Opeations        :  6803
Load Local HITM             :  766
Total Shared Cache Lines    :  3
```

Shared Cache Line Distribution Pareto

Cacheline : 0x61734492b380

Local HITM	Data Address/Offset	Source:Line	NodeCpu List
1.82%	0x8	thread_runner(void*)+817	0{15}
8.07%	0x10	thread_runner(void*)+817	0{0}
20.05%	0x18	thread_runner(void*)+817	0{9}
19.01%	0x20	thread_runner(void*)+817	0{2, 10}
3.91%	0x28	thread_runner(void*)+394	0{0, 2, 9-10, 15}
13.80%	0x30	thread_runner(void*)+830	0{0, 2, 9-10, 12, 15}
15.89%	0x38	pthread_mutex_unlock.c:43	0{0, 2, 9-10, 12, 15}
13.02%	0x38	pthread_mutex_lock.c:48	0{0, 2, 9-10, 12, 15}
2.34%	0x38	lowlevellock.c:45	0{0, 2, 9-10, 12, 15}
2.08%	0x38	lowlevellock.c:42	0{0, 2, 9-10, 12}

Cacheline : 0x61734492b3c0

Local HITM	Data Address/Offset	Source:Line	NodeCpu List
14.99%	0x0	pthread_mutex_lock.c:94	0{0, 2, 9-10, 12, 15}
17.98%	0x4	pthread_mutex_lock.c:172	0{0, 2, 9-10, 15}
64.03%	0x8	pthread_mutex_lock.c:80	0{0, 2, 9-10, 15}
1.63%	0x8	pthread_mutex_lock.c:44	0{0, 2, 9-10, 12, 15}
1.36%	0x8	pthread_mutex_unlock.c:51	0{0, 2, 9-10, 12, 15}

Table 1: Cache Contention w/o any optimization

1.2 Identifying True Sharing

`True Sharing` can be seen very evidently in both of the hottest cachelines here. In the first cacheline (`0x61734492b380`) we can see that for the same cache offsets `0x28, 0x30, 0x38` there are multiple threads contending indicating hits in modified caches. Very similar pattern can observed in the second cacheline(`0x61734492b3c0`) at offsets of `0x0, 0x4, 0x8`.

CODE One easy observation here which will help in our further analysis that, generally `true sharing` is occuring at the source line of `pthread_mutex_lock` or `unlock` which also makes sense since multiple threads will be contending for the same lock. In code we can find that here:

```
    while (getline(input_file, line)) {

        pthread_mutex_lock(&line_count_mutex);
        tracker.total_lines_processed++;
        pthread_mutex_unlock(&line_count_mutex);

        /* .. some code .. */

            // true sharing: on lock and total word variable
            pthread_mutex_lock(&tracker.word_count_mutex);
            tracker.total_words_processed++;
            pthread_mutex_unlock(&tracker.word_count_mutex);

        }
    }
```

1.3 Solving True Sharing

This contentiona can be removed easily. Since we are acquiring lock on every iteration of loop, what we can do instead is that, maintain a local counter for each thread and once the thread is finished reading it will the update the counter. Here is how it impacts the performance:

```
Total Load Opeations       :  850
Load Local HITM            :  33
Total Shared Cache Lines   :  2

---------------------------------------------------------------------------------
                    Shared Cache Line Distribution Pareto
---------------------------------------------------------------------------------
                         Cacheline :  0x565585af8380

Local HITM  Data Address/Offset          Source:Line          NodeCpu List
  68.75%           0x10          thread_runner(void*) + 799   06, 10
  31.25%           0x20          thread_runner(void*) + 799   09
```

Table 2: Cache Contention w/o `true sharing`

Analysis We can see that number of modified cache hits has reduced as compared to the unoptimized case. Also, from our observation we can see that the contention for `mutex_lock` didn't come in the radar of `perf` indicating that its number has become too few.

1.4 Identifying False Sharing

In the first cacheline (`0x61734492b380`) we can that the core `0{0}` writes at the offset of `0x10`, the core `0{9}` writes at the offset of `0x18` of the same cachline and the cores `0{2,10}` writes at the offset of `0x20` of that exact same cacheline resulting in `false sharing` of this cache line.

CODE Similar to the case of `true sharing` we can observe that `false sharing` mostly occurs at the source line of `thread runner(void*)+817` where the threads are updating their individual word count in a shared array of `tracker.word_count` which results in contiguous memory of these counters. In code here we can see:

```
        while ((pos = line.find(delimiter)) != std::string::npos) {
            token = line.substr(0, pos);

            // false sharing: on word count
            tracker.word_count[thread_id]++;



            line.erase(0, pos + delimiter.length());
        }
```

Consider the `64 bytes` cacheline as an array of `16` blocks each of `4 bytes`, then we locate these counters as in the image below which resultings in conviction of this cacheline from private cache of all `5` threads.

`word_counters of all 5 threads`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 1: false sharing of this cacheline because of contiguous memory of counters

1.5 Solving False Sharing

We can remove `False Sharing` by padding memory space in between the counters the threads. Idea is separate the counter memory to different cache lines so that they do not contend for the same cacheline. This can be visualized as in the image below similar to above image. This time the `word_counter` for each thread has been spread across multiple cache lines.

	counter of thread 1							
cacheline 1:	1	3	5	7	9	11	13	15
	counter of thread 2							
cacheline 2:	1	3	5	7	9	11	13	15
	counter of thread 3							
cacheline 3:	1	3	5	7	9	11	13	15
	counter of thread 4							
cacheline 4:	1	3	5	7	9	11	13	15
	counter of thread 5							
cacheline 5:	1	3	5	7	9	11	13	15

Figure 2: padding to have separate cacheline for each thread counter memory

Employing the above solution, the results from `perf` turns out like following:

```
Total Load Opeations        :  7435
Load Local HITM             :  71
Total Shared Cache Lines    :  2

---------------------------------------------------------------------------------
                     Shared Cache Line Distribution Pareto
---------------------------------------------------------------------------------
                          Cacheline :  0x5ebc5583e4c0

Local HITM  Data Address/Offset          Source:Line          NodeCpu List
   9.09%            0x0          thread_runner(void*) + 405   0{2, 10, 12, 14}
  10.61%            0x10         pthread_mutex_unlock.c:43    0{2, 10, 12, 14}
   6.06%            0x10         pthread_mutex_lock.c:48      0{2, 10, 12, 14}
   1.52%            0x10         lowlevellock.c:42            0{2, 10, 12, 14}
  66.67%            0x20         pthread_mutex_lock.c:80      0{2, 10, 12, 14}
   3.03%            0x20         pthread_mutex_unlock.c:51    0{2, 10, 12, 14}
   1.52%            0x20         pthread_mutex_lock.c:44      0{2, 10, 12, 14}
   1.52%            0x20         pthread_mutex_unlock.c:39    0{2, 10, 12, 14}

                          Cacheline :  0x5ebc5583e280

Local HITM  Data Address/Offset          Source:Line          NodeCpu List
    100%            0x0          pthread_mutex_unlock.c:80    0{2, 10, 14}
```

Table 3: Cache Contention w/o `false sharing`

Analysis We can see that there aren't many different offsets in the cacheline which are bein contended by one or two cores. Also, from our obervation we can follow that mostly contention are of `mutex_locks` which is `true sharing` issue.

1.6 Combined perfomance improvement

If remove both `false sharing` and `true sharing` from the source code. Then `perf` report gives empty `Pareto` table.

```
Total Load Opeations       :  1039
Load Local HITM            :  0
Total Shared Cache Lines   :  0

=================================================================================
                      Shared Cache Line Distribution Pareto
=================================================================================
```

Table 4: Cache Contention w/o `false sharing` & `true sharing`

The performance improvement can also be seen the speed ups in completing the task

Case	Time (in ms)	Local HITM
w/o any optimization	399	766
w/o false sharing	326	71
w/o true sharing	181	33
w/o false sharing & true sharing	116	0

Table 5: Speed-up performance

2 Problem 2

Details of this problem is entirely in the source file `problem2.cpp` in the assignment zip folder. Few details of the implementation:

1. The memory buffer is implemented using `queue` data structure with the size limited to `MAX_BUFFER_SIZE`.
2. `pthread_mutex_locks` & `condvars`: One lock on reading the input file to isolate reading. Another lock on writing content to memory buffer. Further, there is a lock and two condvars between the consumer and producer threads to maintain correctness.

3 Problem 3

The code we need to analyze is as follows:

```
for i = 1, N-2
    for j = i+1, N
        A(i, j-i) = A(i, j-i-1) - A(i+1, j-i) + A(i-1, i+j-1)
```

3.1 Dependency Analysis

For all pairs of Array `A` `read-write` access, we will assume that the write occured on $(I_o, J_o)^{th}$ iteration and the read happened on $(I_o + \Delta I, J_o + \Delta J)^{th}$ iteration.

1. Dependency in `A(i, j-i)` & `A(i, j-i-1)`:

$$I_o = I_o + \Delta I \qquad \& \qquad J_o - I_o = (J_o + \Delta J) - (I_o + \Delta I) - 1$$

   This results in $(\Delta I = 0, \Delta J = 1)$ hence the dependency vector is `(0, 1)`. We have the leftmost non-zero value as positive hence it is a **flow dependency**.

2. Dependency in `A(i, j-i)` & `A(i+1, j-i)`:

$$I_o = I_o + \Delta I + 1 \qquad \& \qquad J_o - I_o = (J_o + \Delta J) - (I_o + \Delta I)$$

   This results in $(\Delta I = -1, \Delta J = -1)$ with distance vector `(-1, -1)`. This is a **anti dependency** both the distances negative.

3. Dependency in `A(i, j-i)` & `A(i-1, i+j-1)`:

$$I_o = I_o + \Delta I - 1 \qquad \& \qquad J_o - I_o = (J_o + \Delta J) + (I_o + \Delta I) - 1$$

   This results in $(\Delta I = 1, \Delta J = -2I_o)$ hence the dependency vector is `(1, -/=)` which means ΔJ can be anything non-positive. Since, the left-most non-zero distance is positive, this is **flow dependency**.