COL380 Assignment 0

Sayam Sethi

January 2022

Contents

1 Code Analysis and Plan of Action

The following observations were made after looking at the original code and analysing using various tools:

1. Both threading loops had issues of cache misses. This was because of the loop having the property of skip-by-numt instead of a linear loop. This leads to lesser spacial locality and more cache misses as a result.

2. `counts[v].increase(tid)` was leading to false sharing. This was due to the increase of adjacent array elements in different threads.

3. The second threading loop was not dividing the work equally among all the threads. This was becuase each bin won't have an equal number of elements, therefore, the work is not fairly divided across all threads.

4. Additionally, in the second loop, for each range, the code was looping over the entire D array, this lead to excessive read and write misses, apart from the wasted time complexity.

`perf` was initially used to compare results with different modifications, however this was discarded in exchange of `cachegrid` tool of `valgrind` since `perf` had very high fluctuations across multiple runs of the same code. This led to inconsistent deductions and difficulty in removing noise from the results of the tool.

Running `gprof` on the original code signified that `readRanges` took the most amount of time. This analysis wasn't helpful. Additionally, running `gprof` on the optimised code gave some absurd results that could not be inferred at all. Therefore, the output of `gprof` tool was discarded. However, the functioning of the tool has been well understood.

The improvements were made sequentially, i.e., performance was compared with the previous optimisation to find out if the optimisation helped in improving cache performance.

2 Reducing Cache Misses

The first improvement that was made was to reduce cache misses in the iterations of the loops. The first loop was changed to having single increments instead of an increment by `numt`. When this was run without any tool, the time taken to execute almost doubled.

On further analysis, the reason turned out to be the non-randomness of the given data. This was resolved by `random_shuffle` on the given data in the `readData` function.

After this, the actual runtime didn't improve, however, the last level cache misses reduced for both read and write. Therefore this optimisation was successful in improving cache performace.

Next, the second loop threading was inspected. The change here was implemented in a different way as compared to the previous loop since direct replication of the modification led to worse results in terms of running time instead.

It was observed that the work is not being divided equally across all the threads since all bins did not have the same size. Therefore, the indices were divided such that the number of writes in each thread would approximately be the same. This was done by creating a new array `indices`.

On running `cachegrind` on this output, it was observed that the last level cache misses were reduced in the case of writes. There was no improvement in the reads.

3 Reducing False Sharing

As mentioned in the first section, the cause of false sharing had been narrowed down. While experimenting with it, removing `alignas(32)` for the `Counter` class had led to great improvements in the cache misses in the last layer when reading and writing.

The reason for this was narrowed down to the first threading loop where the `counts` array was being repeatedly fetched and the `_counts` array was being modified. The additional padding provided by `alignas` was leading to less number of elements of the array being present in the cache which led to more cache misses.

After this, the data in the `_counts` array was padded such that each index mapped to $index \times 32 = index << 5$. This led to slight improvements in last level cache read and write misses. However, there was more than 50% improvement in `l2_rqsts.all_rfo` and slight improvement in `l1d.replacement` on the `perf` tool.

4 Analysis of the Bottleneck and its Improvement

After the above improvements, it was observed that the most number of misses were happening in the second loop where the data had to be written to `D2` and be read from `D`. The reason for this was the non-sequential nature of the data reads and updates. Entire `D.data` was being scanned multiple times in each thread. Additionally the writes were in such a fashion that encouraged false sharing.

The above shortcomings were modified by changing the code to the following:

```
50 #pragma omp parallel num_threads(numt)
51   {
```

```
    int tid = omp_get_thread_num();
    int len = indices[tid + 1] - indices[tid], offset = indices[tid];
    int temp_counts[len];
    memcpy(temp_counts, rangecount + offset, len * sizeof(int));
    for (int i = 0; i < D.ndata; ++i) {
      int r = D.data[i].value;
      if (r >= offset && r < offset + len) {
        D2.data[temp_counts[r - offset]++] = D.data[i];
      }
    }
  }
```

The reason for the change was to perform all operations in a single pass of `D` and update adjacent values in `D2` within the same thread. This led to about 43% decrease in last level cache write misses and 99.6% decrease in L1 cache read misses. However, there was a 7% increase in the last level cache read misses and 87.4% increase in the L1 cache write misses. The increase was because of copying the data from `rangecount` array to the local array.

A previous iteration of the code involved updating `rangecount` directly instead of copying it to `temp_counts` and then updating it. This also had similar improvements in L1 cache read misses, but very little improvement in last level cache write misses and similar amounts of decrease in improvement in the other two.

This is why a local array was used to reduce the misses. The decreased improvement in the two parameters is outweighed by the drastic improvement in the other two parameters. Additionally, the running time of the program is almost halved in comparison to the previous version.

5 Miscellaneous Changes

- The `makefile` was modified to include the `fopenmp` flag even when compiling.
- There was an out of bounds access in `classify.cpp` which involved accessing `rangecount[-1]` in the second loop. This issue has been fixed in the final version.
- An `issorted` method has been defined in the `Data` class to ensure that the output is sorted. `assert(D2.issorted())` has been called after the timing in `timedwork` function.