Assignment 0, COL380

Sarthak Singla, 2019CS10397

Analysis of Original code

Runtime

On a 4 core system, the given datafile dfile and range file rfile take on average 360ms with 4 threads.

Valgrind

Valgrind tool cachegrind was used for profiling cache behaviour.
cg_annotate was used to get a line by line cache behaviour**. The output is stored in file cg_val_old.**

```
==59047==
==59047== I   refs:      34,195,240,917
==59047== I1  misses:             3,563
==59047== LLi misses:             3,423
==59047== I1  miss rate:           0.00%
==59047== LLi miss rate:           0.00%
==59047==
==59047== D   refs:       6,444,388,996  (6,344,460,904 rd   + 99,928,092 wr)
==59047== D1  misses:       381,831,749  (  380,359,515 rd   +  1,472,234 wr)
==59047== LLd misses:       380,960,797  (  379,489,648 rd   +  1,471,149 wr)
==59047== D1  miss rate:            5.9% (          6.0%     +        1.5%  )
==59047== LLd miss rate:            5.9% (          6.0%     +        1.5%  )
==59047==
==59047== LL refs:          381,835,312  (  380,363,078 rd   +  1,472,234 wr)
==59047== LL misses:        380,964,220  (  379,493,071 rd   +  1,471,149 wr)
==59047== LL miss rate:             0.9% (          0.9%     +        1.5%  )
levio_sa@ZenBook:~/IITD Folder/Sem6/COL380/Submission/Original$
```

Terminologies:
I- Instruction
D- Data
LL(i/d)- Last level (instruction/data)

Observations:
D1 cache miss rate is 5.9% overall, 6% for read and 1.5% for write
LLd cache miss rate is 5.9% overall, 6% for read and 1.5% for write

- We can observe that the number of read requests is very high (6B) compared to write requests (1B).
- It can be observed that the cache miss rate is high, especially due to the high read miss rate. This points to possible false sharing.

Analysis of cg_val_old:

We are primarily interested in the last 6 columns which stand for:

Dr: D1 cache accesses(read)
D1mr: D1 cache misses(read)
DLmr: LL cache misses(read)
Dw: D1 cache accesses(write)
D1mw: D1 cache misses(write)
DLmw: LL cache misses((write)

We look for high cache miss places in the code. It is seen that the two parallel regions contribute to the maximum number of cache misses.

Gprof

We used gprof to profile the running times of the various components of code. Optimisations had to be turned off while compiling to achieve this. **The output has been stored in file gprof_old.**

The argument reps was set to 100 to ensure that the time spent in the actual work doesn't get hidden by the time in reading data and ranges. However, even with these precautions, time taken by some portions still remained insignificant.

From first table, we can see significant time spent in Ranges::range(), Ranges::within(), Counter::increase() and Data::Item::Item(), which are relevant to our current code.

The second table shows time spent in functions and their children. It shows Ranges::range(), Range::within(), Counter::increase() and Data::Data() to consume most time in our task.

These functions are there mostly because of being called multiple times.

Code Modifications

```
#pragma omp parallel num_threads(numt)
  {
     int tid = omp_get_thread_num(); // I am thread number tid
     for(int i=tid; i<D.ndata; i+=numt) { // Threads together
share-loop through all of Data
        int v = D.data[i].value = R.range(D.data[i].key);// For each
data, find the interval of data's key,
                     // and store the interval id in value. D is
changed.
        counts[v].increase(tid); // Found one key in interval v
     }
  }
```

This was the first parallel region of the original code. Here we have many read misses in the first assignment inside for loop, which we can see in cg_val_old.
We see that consecutive threads access consecutive portions in the array D.data. Therefore, here we may have an instance of false sharing. Multiple cores might want to write to the same cache line. For this each other cache line has to get invalidated. We modify the code as below:

```
int sblock = 16;
  int bblock = 16*numt;
  #pragma omp parallel num_threads(numt)
  {
     int tid = omp_get_thread_num();
     for(int x=0; x<D.ndata; x+=bblock){
        int y = ((x+bblock)<D.ndata)?(x+bblock):(D.ndata);
        int tmp = x+tid*sblock;
        for(int i=tmp; i<std::min(tmp+sblock,y);++i){
           int v = D.data[i].value = R.range(D.data[i].key);// For
each data, find the interval of data's key,
                      // and store the interval id in value. D is
changed.
           counts[v].increase(tid); // Found one key in interval v
        }
     }
  }
```

Here, we provide contiguous blocks of 16 to each thread to process. This prevents frequent cache line invalidation as happened earlier. Also, we have kept this number small, so that when the last level cache reads a block from memory, different portions can be assigned to different threads without requiring another fetch. An analysis of cg_val_new shows that the read misses do reduce.

```
#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.
     }
  }
```

Here we have large number of read misses in the if statement. In the subsequent instruction, we have large number of write misses. But we also have large number of LL write misses, which means this is due to the accesses inside D2.data array being random and not local. Here also, false sharing is a possibility in accesses to rangecount array by threads. We modify the code as follows:

```
unsigned int *rct = new unsigned int[R.num()];
  #pragma omp parallel num_threads(numt)
  {
     int tid = omp_get_thread_num();
     int a = ((tid)*R.num())/numt;
     int b = ((tid+1)*R.num())/numt;
     for(int d=0; d<D.ndata; ++d){
        int r = D.data[d].value;
        if(a<=r && r<b){
           D2.data[((r==0)?0:rangecount[r-1])+rct[r]++] = D.data[d];
        }
     }
  }
```

Here we assign contiguous values of r to the various threads instead of separated ones, to reduce false sharing. A comparison of cg_val_old and ccg_val_new shows we reduce many cache misses.

We do not modify anything in the functions pointed out by gprof, as we did not find scope there. From an analysis of valgrind, we could see that not many cache misses happened there.

We also parallelised the calculation of rangecount array, but as we didn't have to do algorithmic improvements, we have left that out.

We also experimented with padding arrays, so that each element resides in a separate cache line. However, this did not yield much difference in runtime or cache misses. This may be due to the extra overhead in creating a larger data structure, and also the overhead of refetching cache line for neighboring elements may not be suitable for the current purpose.

Analysis of Modified Code

Runtime

```
levio_sa@ZenBook:~/IITD Folder/Sem6/COL380/Submission/New$ make run
./classify rfile dfile 1009072 4 3
169.913 ms
169.018 ms
171.003 ms
3 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 169.01
8 ms, Average was 169.978 ms
```

On a 4 core system, the given datafile dfile and range file rfile take on average 170ms with 4 threads.

This is a significant improvement over the previous 360ms and can be attributed to the improved memory performance and better distribution of workflow among threads.

Valgrind

Valgrind tool cachegrind was used to for profiling cache behaviour. cg_annotate was used to get a line by line cache behaviour**. The output is stored in file cg_val_new.**

```
==6073==
==6073== I   refs:      16,107,858,112
==6073== I1  misses:             3,582
==6073== LLi misses:             3,443
==6073== I1  miss rate:           0.00%
==6073== LLi miss rate:           0.00%
==6073==
==6073== D   refs:       3,440,645,979  (3,337,690,518 rd   + 102,955,461 wr)
==6073== D1  misses:         3,601,350  (    2,131,740 rd   +   1,469,610 wr)
==6073== LLd misses:         3,511,266  (    2,042,747 rd   +   1,468,519 wr)
==6073== D1  miss rate:             0.1% (          0.1%     +         1.4%  )
==6073== LLd miss rate:             0.1% (          0.1%     +         1.4%  )
==6073==
==6073== LL refs:            3,604,932  (    2,135,322 rd   +   1,469,610 wr)
==6073== LL misses:          3,514,709  (    2,046,190 rd   +   1,468,519 wr)
==6073== LL miss rate:              0.0% (          0.0%     +         1.4%  )
```

Observations:
D1 cache miss rate is 0.1% overall, 0.1% for read and 1.4% for write
LLd cache miss rate is 0.1% overall, 0.1% for read and 1.4% for write

- We managed to reduce the D1 and LLd cache miss rates from 5.9% to 0.1% which is a highly significant improvement.
- We can see a huge improvement in read misses. Some of it is due to successfully removing some false sharing.
- Some improvement in write misses is also there.

Analysis of cg_val_new:
We can see significant improvement in cache performance if we compare cg_val_old and cg_val_new side by side.

In the first parallel region, the number of read misses gets reduced by 3 times approximately.

In the second parallel region, the number of read misses gets reduced by approximately 300 times, and the number of write misses also gets reduced, though by a smaller amount.

Gprof

The output has been stored in file gprof_new.

As we did not modify the functions per se, gprof comparison of the old and new files is not very well suited. Still, we can observe a significant shift in time taken by function classify, in the modified code.

Even here, for many functions,gprof just shows 0 time, as several functions take a lot more time.

NOTE-ALL RESULTS ARE WITH DEFAULT COMMAND LINE ARGUMENTS