COL380 Assignment0
Satyam Kumar Modi, 2019CS50448

Analysis using **gprof** tool

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.27      2.86     2.86                             readRanges(char const*)
  0.00      2.86     0.00        3     0.00     0.00  classify(Data&, Ranges const&, unsigned int)
  0.00      2.86     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z8classifyR4DataRK6Rangesj
```

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.35% of 2.86 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    2.86    0.00                 readRanges(char const*) [1]
-----------------------------------------------
                0.00    0.00       3/3           timedwork(Data&, Ranges const&, unsigned int) [14]
[9]      0.0    0.00    0.00       3         classify(Data&, Ranges const&, unsigned int) [9]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [18]
[10]     0.0    0.00    0.00       1         _GLOBAL__sub_I__Z8classifyR4DataRK6Rangesj [10]
-----------------------------------------------
```

Analysis of the original code

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.22      1.31     1.31                             readData(char const*, unsigned int)
  0.00      1.31     0.00        3     0.00     0.00  classify(Data&, Ranges const&, unsigned int)
  0.00      1.31     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z8classifyR4DataRK6Rangesj
```

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.76% of 1.31 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    1.31    0.00                 readData(char const*, unsigned int) [1]
-----------------------------------------------
                0.00    0.00       3/3           repeatrun(unsigned int, Data&, Ranges const&, unsigned int) [11]
[7]      0.0    0.00    0.00       3         classify(Data&, Ranges const&, unsigned int) [7]
-----------------------------------------------
                0.00    0.00       1/1           _fini [16]
[8]      0.0    0.00    0.00       1         _GLOBAL__sub_I__Z8classifyR4DataRK6Rangesj [8]
-----------------------------------------------
```

Analysis of the modified code

Gprof is a profiling tool which could provide info only about the runtime of different sections of the code and the number of times a function was called while executing a program. We notice that since we don't have a return type of **repeatRun** function, we are not getting any data for **repeatRun** and **classify** function. Thus, this data is not very useful for optimisation purposes.

Analysis using **Valgrind**

```
==89172== I   refs:      34,131,610,901
==89172== I1  misses:             3,646
==89172== LLi misses:             3,500
==89172== I1  miss rate:           0.00%
==89172== LLi miss rate:           0.00%
==89172==
==89172== D   refs:       6,435,354,278  (6,335,417,676 rd   + 99,936,602 wr)
==89172== D1  misses:       381,861,951  (  380,389,407 rd   +  1,472,544 wr)
==89172== LLd misses:       375,952,290  (  374,480,979 rd   +  1,471,311 wr)
==89172== D1  miss rate:            5.9% (          6.0%     +        1.5%  )
==89172== LLd miss rate:            5.8% (          5.9%     +        1.5%  )
==89172==
==89172== LL refs:          381,865,597  (  380,393,053 rd   +  1,472,544 wr)
==89172== LL misses:        375,955,790  (  374,484,479 rd   +  1,471,311 wr)
==89172== LL miss rate:             0.9% (          0.9%     +        1.5%  )
```

Cache data obtained from original code

```
==90468== I   refs:      16,240,240,924
==90468== I1  misses:             3,838
==90468== LLi misses:             3,692
==90468== I1  miss rate:           0.00%
==90468== LLi miss rate:           0.00%
==90468==
==90468== D   refs:       3,585,037,429  (3,435,682,504 rd   + 149,354,925 wr)
==90468== D1  misses:         2,690,998  (    1,016,401 rd   +   1,674,597 wr)
==90468== LLd misses:         2,585,136  (      948,461 rd   +   1,636,675 wr)
==90468== D1  miss rate:            0.1% (          0.0%     +         1.1%  )
==90468== LLd miss rate:            0.1% (          0.0%     +         1.1%  )
==90468==
==90468== LL refs:            2,694,836  (    1,020,239 rd   +   1,674,597 wr)
==90468== LL misses:          2,588,828  (      952,153 rd   +   1,636,675 wr)
==90468== LL miss rate:             0.0% (          0.0%     +         1.1%  )
```

Cache data obtained from modified code

Valgrind is a profiling tool that profiled a program based on memory leakages, cache management, etc. Here, we focused mainly on cache based data for our profiling. We observed that the first for loop can be more optimized if a thread can access contiguous data. Thus, in the modified code, I have alloted contiguous part of loops of length **num_threads** to each thread. Apart from that, I have statically scheduled loop with a gap of 50. This helped to reduce the amount of data cache miss in the first for loop by 74%. Further, we can notice that the D1 miss rate has also come down from 5.9% to 0.1%.

Algorithmic optimisations

Next, we have added some algorithmic optimisation in the last for loop. We noticed in the original code that the last for loop ran **(R.num()*D.ndata)** times. We tried to reduce this to only **D.ndata** times. For this, we have used an **index** array in the first for loop which records the interval to which an integer belongs. And using this, we only had to loop **D.ndata** times in the final loop. This significantly reduced the runtime of the code from an average of **390ms** to **130ms.**

Ir	D1mr	Source
		--- From '/home/satyam/Documents/sem2/COL380/A1/classify.cpp' ---
		Counter counts[R.num()]; // I need on counter per interval. Each counter can keep pre-thread subco…
252	37	#pragma omp parallel num_threads(numt)
		{
12		int tid = omp_get_thread_num(); // I am thread number tid
12 108 912	3	for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
12 108 888	1 513 057	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.
3 027 216	12 126	counts[v].increase(tid); // Found one key in interval v
		}
		}

Cache data for Loop 1

Ir	D1mr	Source
		--- From '/home/satyam/Documents/sem2/COL380/A2/classify.cpp' ---
		Counter counts[R.num()];
		int index[D.ndata];
76 062	41	#pragma omp parallel num_threads(numt)
		{
24		int tid = omp_get_thread_num(); // I am thread number tid
36		#pragma omp for schedule(static, 50)
168	1	for(int i=0; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
8 461 554	75	int temp[numt];
17 406 492		for (int j = 0; j < numt && i+j < D.ndata; j++)
		{
14 379 276	393 523	int v = D.data[i+j].value = R.range(D.data[i+j].key);// For each data, find the interval of data's key,
		// and store the interval id in value. D is changed.
3 027 216	12 010	counts[v].increase(tid); // Found one key in interval v
6 054 432		temp[j] = v;
		}
756 804		memcpy(index + i, temp, numt * sizeof(int));
		}
		}

Cache data for Loop 2