Christos Axelos, AEM 1814, 2nd Assignment in High Performance computing 2017/18

OBSERVATIONS

- We add the **-qopenmp** flag to include the openMP libraries

- We run each time our program using the command: **./seq_main -i Image_data/texture17695.bin -o -b -n 10000 ,** so the number of clusters is 10000.

- Flag **-O0** gave very bad performance, so I used only the **-fast** flag

SYSTEM SPECS

- All the measurements were done using the below characteristics:
a) OS: Ubuntu LTS 16.04
b) Core Edition: 4.4.0-97-generic
c) CPU: Intel Core i5
d) NumOfProcessors: 4
e) Compiler: icc
f) Compiler’s edition: 18.0.0

RESULTS

- We begin with our **initial** code, trying to find the loops that increase the execution time. The bellow code is the initial code that we want to parallelize. It is from file **seq_kmeans.c**

Function: **seq_kmeans()**

```
do {
    delta = 0.0;
    for (i=0; i<numObjs; i++) {
        /* find the array index of nestest cluster center */
        index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                     clusters);

        /* if membership changes, increase delta by 1 */
        if (membership[i] != index) delta += 1.0;

        /* assign the membership to object i */
        membership[i] = index;

        /* update new cluster center : sum of objects located within */
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            newClusters[index][j] += objects[i][j];
    }

    /* average the sum and replace old cluster center with newClusters */
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            if (newClusterSize[i] > 0)
                clusters[i][j] = newClusters[i][j] / newClusterSize[i];
            newClusters[i][j] = 0.0;   /* set back to 0 */
        }
        newClusterSize[i] = 0;   /* set back to 0 */
    }

    delta /= numObjs;
} while (delta > threshold && loop++ < 500);
```

Function **find_nearest_cluster()**

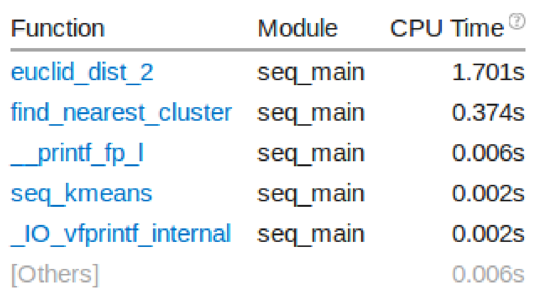
```
    /* find the cluster id that has min distance to object */
    index    = 0;
    min_dist = euclid_dist_2(numCoords, object, clusters[0]);

    for (i=1; i<numClusters; i++) {
        dist = euclid_dist_2(numCoords, object, clusters[i]);
        /* no need square root */
        if (dist < min_dist) { /* find the min and its array index */
            min_dist = dist;
            index    = i;
        }
    }
    return(index);
```

Function **euclid_dist_2()**

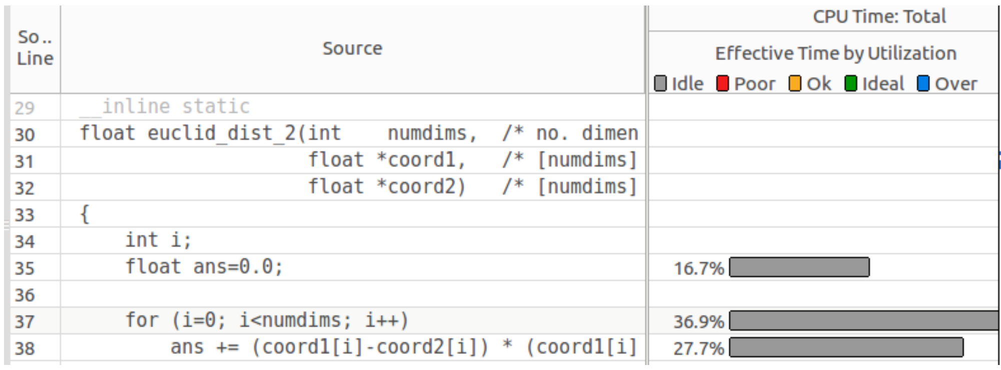
```
float euclid_dist_2(int    numdims,  /* no. dimensions */
                    float *coord1,   /* [numdims] */
                    float *coord2)   /* [numdims] */
{
    int i;
    float ans=0.0;

    for (i=0; i<numdims; i++)
        ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);

    return(ans);
```

- After running the sequential code, the timing that we get at **Venus** is 7.288691 seconds. Let’s see where is the biggest latency, using the **vtune amplifier**

- In the left image, we see the functions that cause the biggest latency. At first, we have to focus at the function **euclid_dist_2**



Function	Module	CPU Time
euclid_dist_2	seq_main	1.701s
find_nearest_cluster	seq_main	0.374s
__printf_fp_l	seq_main	0.006s
seq_kmeans	seq_main	0.002s
_IO_vfprintf_internal	seq_main	0.002s
[Others]		0.006s



So.. Line	Source	CPU Time: Total Effective Time by Utilization ▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over
29	`__inline static`	
30	`float euclid_dist_2(int    numdims,  /* no. dimen`	
31	`                    float *coord1,   /* [numdims]`	
32	`                    float *coord2)   /* [numdims]`	
33	`{`	
34	`    int i;`	
35	`    float ans=0.0;`	16.7%
36		
37	`    for (i=0; i<numdims; i++)`	36.9%
38	`        ans += (coord1[i]-coord2[i]) * (coord1[i]`	27.7%

- The 36.9% of total time is wasted in **line 37**, so we try here our first optimizations.

- I tried to optimizise, applying paralelism to the for-loop with 2 different ways

code (a)

```
float ans=0.0;
float ans_i;
#pragma omp parallel private (ans_i)
{
  ans_i = 0;
  #pragma omp for
  for (i=0; i<numdims; i++)  {
    ans_i += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
  }
  #pragma omp critical
  {
    ans += ans_i;
  }
}
```

code (b)

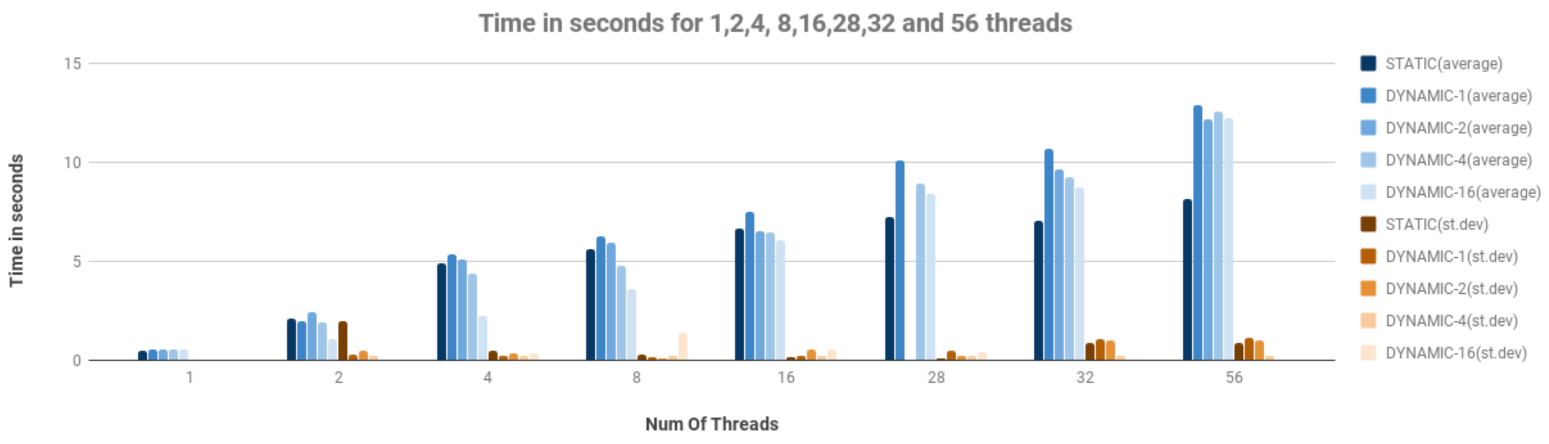
```
int i;
float ans=0.0;

#pragma omp parallel for reduction (+:ans)
for (i=0; i<numdims; i++)  {
  ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
}
```

- However the performance is getting a lot worser. The reason why the above optimizations increase so much the latency is the extremely big **overhead** that we add by creating and destroying threads each time we call function **euclid_dist_2**. And this function is called 884750000 times! Furthermore we can’t use the **nowait** keyword, because we must wait all threads to finish and then write the results

- So we must find a way to parallelize that function without destroying our threads each time we return from that function. We will not create our threads inside **euclid_dist_2**, but inside the function **find_nearest_cluster()**

- Lets try now parallelizing the code in **find_nearest_cluster()**

- After running the code for different number of threads and using different type of scheduling, we improved the performance , but still is **worser** a lot more than the sequential


Time in seconds for 1,2,4, 8,16,28,32 and 56 threads
Time in seconds
Num Of Threads
STATIC(average)
DYNAMIC-1(average)
DYNAMIC-2(average)
DYNAMIC-4(average)
DYNAMIC-16(average)
STATIC(st.dev)
DYNAMIC-1(st.dev)
DYNAMIC-2(st.dev)
DYNAMIC-4(st.dev)
DYNAMIC-16(st.dev)