Final Project OpenMP 1 - KMEANS

Student: Tiago de Souza Oliveira

Toolkit:

- icc -g -pg -qopenmp -o kmeans_omp kmeans_omp.c
- gprof -l kmeans_omp > kmeans_omp_gprof.out
- more kmeans_omp_gprof.out
- For sbatch:

openmp > $ job_kmeans_omp.sh

```
#SBATCH -o %x-%J.out
#SBATCH -e %x-%J.error

#SBATCH -J omp_job_kmeans       # Job name
#SBATCH -o omp_job_kmeans.o%j   # Name of stdout output file(%j expands to jobId)
#SBATCH -e omp_job_kmeans.o%j   # Name of stderr output file(%j expands to jobId)

#SBATCH --time=0-00:05:00 #requested time to run the job
#SBATCH -c 32 #(64 cores per job)
#SBATCH -t 00:10:00 #(10 min of execution time)
#SBATCH --mem=16GB #(4GB of memory)
#SBATCH --exclusive

export OMP_NUM_THREADS=8
time ./kmeans_omp
```

By sticking to the Incremental Parallelization strategy, I tried to focus on the most expensive parts of the application. From that perspective, the gprof report gives some hints as it shows below in terms of routine and subroutine time consumption, computational cost, call frequency and so on.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ps/call  ps/call  name
 92.27    415.97   415.97                             distEucl (kmeans_omp.c:355 @ 401f8c)
  5.06    438.79    22.81                             distEucl (kmeans_omp.c:354 @ 401fee)
  1.28    444.54     5.76                             recalculateCenters (kmeans_omp.c:322 @ 401d8c)
  0.72    447.81     3.27                             recalculateCenters (kmeans_omp.c:321 @ 401d14)
  0.18    448.60     0.79                             __libm_sqrt_ex
  0.13    449.18     0.58                             findClosestCenters (kmeans_omp.c:292 @ 401b88)
  0.12    449.73     0.55                             createRandomVectors (kmeans_omp.c:151 @ 40143b)
  0.08    450.09     0.36                             recalculateCenters (kmeans_omp.c:320 @ 401dee)
```

```
granularity: each sample hit covers 2 byte(s) for 0.00% of 451.11 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[5]      0.2    0.79    0.00                 __libm_sqrt_ex [5]
-----------------------------------------------
                0.05    0.00 150000000/150000000     findClosestCenters (kmeans_omp.c:292 @ 401b88) [6]
[14]     0.0    0.05    0.00 150000000         distEucl (kmeans_omp.c:349 @ 401f53) [14]
-----------------------------------------------
                0.00    0.00 1500000/1500000     findClosestCenters (kmeans_omp.c:293 @ 401bf0) [21]
[25]     0.0    0.00    0.00 1500000         argMin (kmeans_omp.c:368 @ 402019) [25]
-----------------------------------------------
                0.00    0.00       1/15          kMeans (kmeans_omp.c:188 @ 401669) [69]
                0.00    0.00      14/15          kMeans (kmeans_omp.c:188 @ 4016a4) [71]
[26]     0.0    0.00    0.00      15         findClosestCenters (kmeans_omp.c:283 @ 401b2e) [26]
```

The gprof report tells essentially that one routine alone consumes more than 95% of the time of processing. The second one relies at a far lower level.

- distEuclid
- and recalculateCenters

Although I have tried to also improve parallelism for other functions, for most of them it results that the fork-join or thread-startup side effect cost was higher than the speeding up parallelism benefit as the more threads I added to the tests.

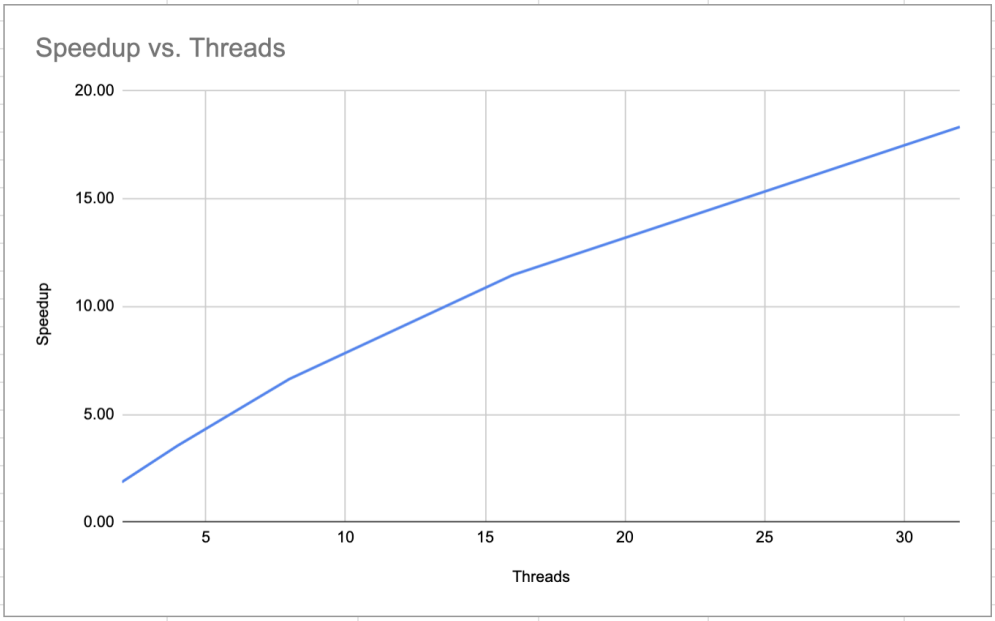
When focusing on parallelizing loops inside distEuclid function, I noticed that the more threads I added to the application the lower it becomes. So I realized that the root cause for that could be external loops or upper level loops creating such unnecessary fork-join overhead.

Therefore, by placing the OMP parallelism at the function level before calling distEuclid (findClosestCenters) results in a better balance for speedup and consistency in the application.

Speedup for running in a node with 64 cores per job configuration

	Threads OMP (findClosestCenters)				
Threads	2	4	8	16	32
Speedup	1.84	3.53	6.61	11.44	18.30
	299	156	83	48	30
	300	156	83	48	30
parallel (s)	298	155	83	48	30
Average (s)	299	155.67	83	48	30

	549
	549
sequence (s)	549
Average (s)	549



OMP in findClosestCenters function

```
double findClosestCenters( double patterns[][Nv], double centers[][Nv], int classes[], double ***distances ) {

    double error = 0.0 ;
    size_t i, j ;

    //placing the parallel for here in order to reduce overhead of creating/destroying threads per (i * j) in the nested loop inside functions
    #pragma omp parallel for private(j) reduction(+:error) schedule(static)
    for ( i = 0; i < N; i++ ) {
            for ( j = 0; j < Nc; j++ )
                (* distances)[i][j] = distEucl( patterns[i], centers[j] ) ;
            classes[i] = argMin( (* distances)[i], Nc ) ;
            error += (* distances)[i][classes[i]] ;
        }


    return error;
}
```

Adding OMP also in recalculateCenters function

After adding parallelism to the second most time expensive function(recaculateCenters), the following statistics could be observed. In summary, the speedup could get better as expected with the caveat of correctly placing the barrier to keep the consistency as before for the sequential implementation.

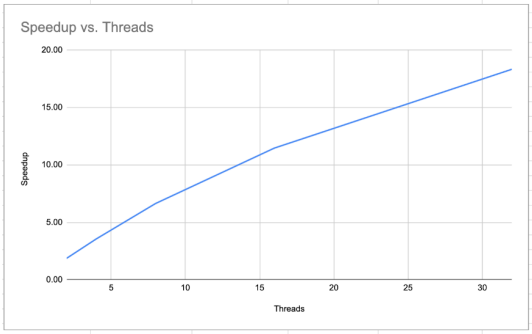
The chart tells us that the application has such linear performance behavior with respect to threads added to speedup.

Speedup for running in a node with 64 cores per job configuration (findClosestCenters, recalculateCenters)

Threads OMP (findClosestCenters)

Threads	2	4	8	16	32
Speedup	1.84	3.53	6.61	11.44	18.30
	299	156	83	48	30
	300	156	83	48	30
parallel (s)	298	155	83	48	30
Average (s)	299	155.67	83	48	30

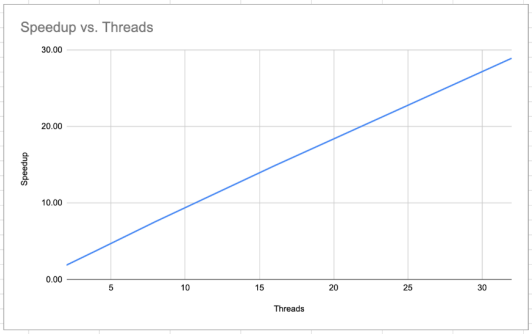
	549
	549
sequence (s)	549
Average (s)	549



Threads OMP (findClosestCenters, recalculateCenters)

Threads	2	4	8	16	32
Speedup	1.87	3.75	7.56	14.84	28.89
	293	147	73	37	19
	294	146	73	37	19
parallel (s)	293	146	72	37	19
Average (s)	293.33	146.33	72.67	37	19.00

	549
	549
sequence (s)	549
Average (s)	549



OMP in recalculateCenters function

```
309  void recalculateCenters( double patterns[][Nv], double centers[][Nv], int classes[], double ***y, double ***z ) {
310
311      double error = 0.0 ;
312
313      size_t i, j;
314
315      double t1, t2;
316      t1=omp_get_wtime();
317
318      #pragma omp parallel for private(j) reduction(+:error)
319      for ( i = 0; i < N; i++ ) {
320          for ( j = 0; j < Nv; j++ ) {
321              (* y)[classes[i]][j] += patterns[i][j] ;
322              (* z)[classes[i]][j] ++ ;
323          }
324      }
325
326      //barrier to make sure that all threads have finished the calculation of tmp arrays
327      #pragma omp barrier
328
329      // update step of centers
330      for ( i = 0; i < Nc; i++ ) {
331          for ( j = 0; j < Nv; j++ ) {
332              centers[i][j] = (* y)[i][j]/(* z)[i][j] ;
333              (* y)[i][j] = 0.0 ;
334              (* z)[i][j] = 0.0 ;
335          }
336      }
337
338      t2=omp_get_wtime();
339      printf("\nTime taken for recalculateCenters is %g seconds\n\n", t2-t1);
340
341      return ;
342  }
```