a2/kmeans/omp_reduction_kmeans.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "kmeans.h"
/*
 * TODO: include openmp header file
 */
#include <omp.h>

// square of Euclid distance between two multi-dimensional points
inline static double euclid_dist_2(int numdims,    /* no. dimensions */
                                   double *coord1, /* [numdims] */
                                   double *coord2) /* [numdims] */
{
    int i;
    double ans = 0.0;

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

    return ans;
}

inline static int find_nearest_cluster(int numClusters,  /* no. clusters */
                                       int numCoords,    /* no. coordinates */
                                       double *object,   /* [numCoords] */
                                       double *clusters) /* [numClusters][numCoords] */
{
    int index, i;
    double dist, min_dist;

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

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

void kmeans(double *objects,     /* in: [numObjs][numCoords] */
            int numCoords,       /* no. coordinates */
            int numObjs,         /* no. objects */
            int numClusters,     /* no. clusters */
```

```
52              double threshold,    /* minimum fraction of objects that change membership */
53              long loop_threshold, /* maximum number of iterations */
54              int *membership,     /* out: [numObjs] */
55              double *clusters)    /* out: [numClusters][numCoords] */
56  {
57      int i, j, k;
58      int index, loop = 0;
59      double timing = 0;
60
61      double delta;        // fraction of objects whose clusters change in each loop
62      int *newClusterSize; // [numClusters]: no. objects assigned in each new cluster
63      double *newClusters; // [numClusters][numCoords]
64      int nthreads;        // no. threads
65
66      nthreads = omp_get_max_threads();
67      printf("OpenMP Kmeans - Reduction\t(number of threads: %d)\n", nthreads);
68
69      // initialize membership
70      for (i = 0; i < numObjs; i++)
71          membership[i] = -1;
72
73      // initialize newClusterSize and newClusters to all 0
74      newClusterSize = (typeof(newClusterSize))calloc(numClusters, sizeof(*newClusterSize));
75      newClusters = (typeof(newClusters))calloc(numClusters * numCoords,
    sizeof(*newClusters));
76
77      // Each thread calculates new centers using a private space. After that, thread 0 does
    an array reduction on them.
78      int *local_newClusterSize[nthreads]; // [nthreads][numClusters]
79      double *local_newClusters[nthreads]; // [nthreads][numClusters][numCoords]
80
81      /*
82       * Hint for false-sharing
83       * This is noticed when numCoords is low (and neighboring local_newClusters exist
    close to each other).
84       * Allocate local cluster data with a "first-touch" policy.
85       */
86      // Initialize local (per-thread) arrays (and later collect result on global arrays)
87      for (k = 0; k < nthreads; k++)
88      {
89          local_newClusterSize[k] = (typeof(*local_newClusterSize))calloc(numClusters,
    sizeof(**local_newClusterSize));
90          local_newClusters[k] = (typeof(*local_newClusters))calloc(numClusters * numCoords,
    sizeof(**local_newClusters));
91      }
92
93      timing = wtime();
94      do
95      {
96          // before each loop, set cluster data to 0
97          for (i = 0; i < numClusters; i++)
98          {
99              for (j = 0; j < numCoords; j++)
100                 newClusters[i * numCoords + j] = 0.0;
```

```
            newClusterSize[i] = 0;
        }

        // reset delta before each iteration; it will be updated via reduction in the parallel region
        delta = 0.0;

        /*
         * TODO: Initiliaze local cluster data to zero (separate for each thread)
         *
         * We now use an OpenMP parallel region where:
         *  - Each thread zeroes its own local_newClusterSize/local_newClusters.
         *  - The object loop is distributed with 'omp for' and 'reduction(+ : delta)'.
         *  - A single thread reduces the per-thread local arrays into the shared arrays.
         */
#pragma omp parallel private(i, j, k, index)
        {
            int tid = omp_get_thread_num();
            int T   = omp_get_num_threads();   // actual number of threads in this team

            /* per-thread zeroing (first-touch initialization of local cluster data) */
            for (i = 0; i < numClusters; i++)
                local_newClusterSize[tid][i] = 0;
            for (i = 0; i < numClusters * numCoords; i++)
                local_newClusters[tid][i] = 0.0;

            // Distribute objects across threads and compute per-thread contributions.
            // delta is accumulated using a reduction to avoid atomics on a shared variable.
#pragma omp for reduction(+ : delta)
            for (i = 0; i < numObjs; i++)
            {
                // find the array index of nearest cluster center
                index = find_nearest_cluster(numClusters, numCoords,
                                             &objects[i * numCoords], 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 centers : sum of all objects located within (average will be performed later)
                /*
                 * TODO: Collect cluster data in local arrays (local to each thread)
                 *       Replace global arrays with local per-thread
                 */
                local_newClusterSize[tid][index]++;
                for (j = 0; j < numCoords; j++)
                    local_newClusters[tid][index * numCoords + j] += objects[i * numCoords + j];
            }
```

```
151
152             /*
153              * TODO: Reduction of cluster data from local arrays to shared.
154              *       This operation will be performed by one thread
155              *
156              * Here we use 'omp single' so that exactly one thread accumulates
157              * all per-thread local arrays into the shared newClusterSize/newClusters.
158              */
159 #pragma omp single
160             {
161                 for (k = 0; k < T; k++)   // only sum over the threads actually in this
    team
162                 {
163                     int *srcS = local_newClusterSize[k];
164                     double *srcC = local_newClusters[k];
165                     if (!srcS || !srcC)
166                         continue;
167                     for (i = 0; i < numClusters; i++)
168                     {
169                         newClusterSize[i] += srcS[i];
170                         for (j = 0; j < numCoords; j++)
171                             newClusters[i * numCoords + j] += srcC[i * numCoords + j];
172                     }
173                 }
174             } /* implicit barrier after single */
175         }     /* end parallel region */
176
177         // average the sum and replace old cluster centers with newClusters
178         for (i = 0; i < numClusters; i++)
179         {
180             if (newClusterSize[i] > 0)
181             {
182                 for (j = 0; j < numCoords; j++)
183                 {
184                     clusters[i * numCoords + j] = newClusters[i * numCoords + j] /
    newClusterSize[i];
185                 }
186             }
187         }
188
189         // Get fraction of objects whose membership changed during this loop. This is used
    as a convergence criterion.
190         delta /= numObjs;
191
192         loop++;
193         printf("\r\tcompleted loop %d", loop);
194         fflush(stdout);
195     } while (delta > threshold && loop < loop_threshold);
196     timing = wtime() - timing;
197     printf("\n nloops = %3d (total = %7.4fs) (per loop = %7.4fs)\n", loop, timing, timing
    / loop);
198
199     for (k = 0; k < nthreads; k++)
200     {
```

```
        free(local_newClusterSize[k]);
        free(local_newClusters[k]);
    }
    free(newClusters);
    free(newClusterSize);
}


```