Parallel and Distributed Computing
Spring Semester - 2020/2021

k-Nearest Neighbors: The Ball Algorithm

Group 15 - Diogo Moura (86976), Francisco Romão (90074), João Figueiredo (90108)

24th April 2021

I. Serial Implementation

A. Building the binary tree

Starting from a large group of points, the two most distant points, a and b, are computed. It is used a recursive greedy algorithm, making the algorithm more efficient, but the solution isn't guaranteed to be the two most distant points, but an approximation. The worst-case scenario can be inefficient - $O(n\_points * n\_points * n\_dims)$. Despite this, empirically, in average, the recursion is done only for k points - $O(k * n\_points * n\_dims)$ - with k much smaller than n_points.

From points a and b we can calculate the center of the line that connects the two points and the projection of every other point on that line. The center is used to separate the points' projections on the left and on the right of the center. These two new groups of points (not the projections of the points but the points themselves) will form the children of the root node. To continue the algorithm, it is applied the same procedure, for each child, but now using the group of points associated to them. On each node, it is needed to save the coordinates of the center and a value for the radius. The radius is calculated starting from the center to the most distant point (not his projection on the ab line). In the end, there will be no more points to separate and the bottom tree layer child won't have anymore children. It's the end of the algorithm.

B. Specification

1) build_tree(): The build_tree function is a recursive function. The parent node calls it for both left and right children, spawning the tree nodes from root to leafs.

This function receives a set of points, the tree vector and additional, not so relevant, arguments.

If the set has only two points, the construction of the ball is hard coded for better performance. Otherwise, the order of functions/procedures called is the following:

- $recursive\_furthest\_apart()$: Finds the two furthest points in a set. Makes use of a more CPU-intensive function, $furthest\_point\_from\_coords$, which goes through the entire set and it is explained in depth in section I-B.2.
- $project\_pts2line()$: Computes a value for each point correlated with its position on a line. Allows points projections to be sorted within a line - without spending the computation power associated with the projection. Detailed in II-B.2.
- $getKsmallest()$: Finds the K smallest value in a set, sparing the computation power required to sort all the values and take the one that separates the higher half from the lower half **[1]**. Detailed in I-B.4.
- $find\_idx\_from\_value()$: Since the $getKsmallest()$ returns a value, this function iterates the set to return its memory position.
- $orthogonal\_projection()$: Performs the orthogonal projection of point in a line defined by two points. Useful to compute the center of the ball.
- $compare\_with\_median()$: Through the $build_tree$ recursion each child node should handle the higher or lower half. This rearranges the set only ensuring that each point is in its respective half.
- radius computation: $furthest\_point\_from\_coords$ is called with the ball center as an argument. The $recursive\_furthest\_apart$ output cannot be employed since it might not reach an absolute solution. The distance between the point found and the center is the ball radius.

2) $recursive\_furthest\_apart$: The function $recursive\_furthest\_apart$ implements a recursion. Initially a point is chosen. At each recursion level,

the furthest point from the current one is discovered, until there is a point that is further apart from the last chosen.

The function $furthest\_point\_from\_coords$ implements a cycle that looks for the furthest point between all the points and its used once at each recursion level. The initial point chosen influences the performance of the algorithm. In attempt to address this problem, the furthest apart points from the parent node were used to start the recursion in the children nodes. For a one million point problem, this reduced the number of calls by a tenth. However the operations required to implement this prevented any run time improvement and it was therefore discarded.

3) $project_pts2line$*:* This function receives the points and the starting and ending point of the line. The main objective is to compute a reduced version of the orthogonal projection. This version is stripped of the computations that are common to all points, or constants. Then, comparing the projections is still possible.

4) $getKsmallest()$*:* Sorting a set of values to compute the median is not very efficient. $getKsmallest$ is an algorithm, presented in **[1]**, that finds the k^{th} smallest element of a set in linear time. Since the median of the set with n elements is the $(n-1)/2$ element for an odd n, or the average between $n/2$ and $(n-1)/2$ for an even n, this algorithm renders itself as very useful. For odd n it gets the median immediately and for even n, aided by a function that finds the lowest and nearest neighbor of a value in a array, this method saves valuable time.

The core loop of this algorithm aims to find a prediction of the k^{th} smallest number in a very short time. After that, the predicted k is compared with the whole group and checked if there are $k-1$ elements lower than the predicted k. If not, this same algorithm is evoked again, but this time with the group of points in which the real k element is contained i.e supposing k is equal to four, and there are 5 elements lower than the predicted k, in this case the algorithm would be called again on the lower set of elements.

The prediction of the k smallest element is done with the following tasks: first the cluster of elements grouped by sets of five elements and a set with the remainder. Each set is sorted and returned the k^{th} smallest value, independently. At this point the number of elements on the cluster was divided by five, resulting in the k^{th} smallest element from each set. If the number of ks are less or equal to five the method goes to computing the k^{th} smallest of the ks, on the other hand, if there are more than 5 ks this algorithm is called recursively until it is left with no more than 5 ks, when the final predicted k^{th} smallest is returned.

The choice of 5 elements was done based on multiple tests with different values in which five elements came out to be the hot-spot of the least time to compute the median.

II. Parallel Implementation

A. Approach used

The approach used for parallelization is to parallelize the inner functions of the algorithm of the ball algorithm on the first node.

Then, parallelize the tree construction, at two levels (outtermost and inner levels), dividing the threads in half to each of the nodes (build_tree parallelization) and also parallelizing the inner functions, until there is one node for each thread.

At this point, the parallelization stops occurring at the lower inner level functions of the ballAlg algorithm and occurs only at the node construction level (outtermost level).

In summary, there is a *cutoff* for the parallelization of the algorithms, after which the parallelization is done only in the tree node construction. This *cutoff* is defined as the execution point where each thread can be assigned a node without being stalled ($threads\_available <= 1$).

B. Decomposition and Synchronization

The number of points/vector is split statically and sieved for each child thread in the parallelization of the algorithms.

1) $furthest\_point\_from\_coords$*:* The function $furthest\_point\_from\_coords$ implements a cycle that looks for the furthest point between all the points and its used once at each recursion level.

The iterations from the function cycle are divided evenly between tasks. Each task is composed of a certain number of iterations which are assigned statically.

Since there is dependence between each recursion of the function $recursive\_furthest\_apart$ it can't be parallelized. However, the function $furthest\_point\_from\_coords$ can be parallelized since it's called once on each recursion level and there is no dependence between points.

It is necessary the use of synchronization, since a private structure is used for each child thread to save the maximum and respective index in the array, but

a reduction is performed at the end. This reduction requires synchronization.

```
typedef struct max_n_idx{
    double maximum;
    long int index;
}max_n_idx;

max_n_idx max_n_idx_max(max_n_idx a, max_n_idx b) {
    return a.maximum > b.maximum ? a : b;
}

#pragma omp declare reduction(max_n_idx_max_: struct max_n_idx: omp_out=max_n_idx_max(omp_out, omp_in))

if (threads_available > 1)
{
    #pragma omp parallel for reduction(max_n_idx_max_:max_point)
    for (long i = 0; i < n_points; i++)
    {
        if ((curr_dist = squared_distance(n_dims, base_coords, pts[i])) > max_point.maximum)
        {
            max_point.maximum = curr_dist;
            max_point.index = i;
        }
    }
}
```

Fig. 1. Parallel recursive furthest Apart

2) project_pts2line: This function implements a cycle that iterates over all the points, calculating the projection for each point.

The iterations from the function cycle are divided evenly between tasks . Each task is composed of a certain number of iterations which are assigned statically.

It isn't required any type of synchronization since privatization is made. Each thread operates separately on its own auxiliary data structures.

The privatization implementation is comprised of an array with elements equal to the number of threads available of type $double*$. Then, a for loop to *alloc* each $double*$ pointer to a space in memory to be used by the thread. This way we may avoid false sharing, given that reserving the whole continuous space and then addressing each memory block by a referencing array could lead to a cache line from one thread be filled with space reserved for another thread, which is avoided if the positions in memory are not contiguous.

```
if(threads_available > 1)
{
    double **p_minus_a = (double**) malloc(sizeof(double*) * omp_get_max_threads());

    for(int i = 0; i < omp_get_max_threads(); i++)
    {
        p_minus_a[i] = (double *)malloc(n_dims * sizeof(double));
    }

    #pragma omp parallel for
    for (int i = 0; i < n_points; i++)
    {
        projections[i] = orthogonal_projection_reduced(n_dims, pts[i], a,
            p_minus_a[omp_get_thread_num()%omp_get_max_threads()], b_minus_a);
    }

    for(int i = 0; i < omp_get_max_threads(); i++)
        free(p_minus_a[i]);

    free(p_minus_a);
}
```

Fig. 2. Parallel version of project_pts2line

3) Median: This function implements a cycle that iterates over a set of vectors (the $getKSmallest$ function original vector is divided into groups of 5 points that constitute this set of vectors), calculating the median for each vector.

The iterations from the function cycle are divided evenly between tasks. Each task is composed of a certain number of iterations which are assigned statically.

Each iteration operates on different elements of the median and point vectors, so it isn't required any synchronization or privatization.

```
if (threads_available > 1)
{
    #pragma omp parallel for if(n_items>1000)
    for(int i=0; i<full_splits; i++)
    {
        medians[i] = sorted_median(vector + 5*i, 5);
    }

    if(semi_splits != 0)
    {
        medians[full_splits] = sorted_median(vector + 5*full_splits, semi_splits);
    }

    n_medians = full_splits + (semi_splits!=0 ? 1 : 0);
}
```

Fig. 3. Parallel version of median

C. Load Balancing

The load balancing is performed by defining the nodes' computations (of the tree construction) as tasks. Since all the nodes of the same level require almost the same workload, this is a decomposition that achieves load balancing by itself. Each processing unit is assigned a node task (from the tree construction) by getting them from the task pool, without being stalled.

Note that the parallelization on the lower level functions is performed with a condition ($threads\_available > 1$) that allows the parallelization to be executed in the inner level functions when the tree width isn't enough for the parallelization to be executed only at the outermost level (node tasks from the tree construction), without the processors being stalled.

This assured that the processing units are making inner computations instead of waiting stalled for new, still unexisting, node tasks (from the tree construction).

The inner ball algorithms' functions such as $furthest\_point\_from\_coords$, $project\_pts2line$ and $median$ achieve parallelization by dividing the iterations of the cycles performed in them. Since each iteration takes exactly the same level of computation, a good load balancing is achieved by dividing evenly and statically the iterations among the tasks. Each task is

then composed of a certain number of iterations of the cycle. Each processing unit is assigned to one of this tasks and each processing unit task completion time will be fairly the same. Also a low level granularity isn't required due to the same fact that all the iterations take almost exactly the same level of computation.

D. Performance Results

The results of the execution can be seen on the table I.

The program scales well with the number of threads. The speedup isn't linear, but scales well.

The speedup suffers particularly for high dimensional problems when the number of threads isn't equal to the number of CPU cores, such as when there are 8 threads for only 4 CPU cores for points with 50 dimensions.

The speedup didn't scale for 8 threads, since there were only 4 processing units for the CPU used. This means that the speedup relative to the number of processing units reduced relatively to 4, 2 or 1 threads. This was expected since there are overheads associated with the task creation and parallelization, namely thread and context creation and allocation and synchronization. Also, there are portions of the algorithm that are executed in serial, which constrain the speedup, as explained by the Ahmdal's Law.

The speedup scales almost linearly with the number of threads, provided there is the same number of CPU cores to execute them. Despite this, when the number of threads increases, the speedup relative to the number of processing units is reduced due to overheads and serialization of portions of the code.

TABLE I

RUNTIMES - LAB13, COMPUTER 4

Arguments	T_{serial}	T_{P1thd}	S_{P1thd}	T_{P2thd}	S_{P2thd}	T_{P4thd}	S_{P4thd}	T_{P8thd}	S_{P8thd}
20 1000000 0	5.3	5.3	1.00	3	1.77	1.9	2.79	2.6	2.04
50 1000000 0	9.8	10	0.98	5.7	1.72	3.6	2.72	5.1	1.92
3 5000000 0	13	12.8	1.02	6.9	1.88	4.2	3.10	4.2	3.10
4 10000000 0	31.2	30.6	1.02	16.8	1.86	9.9	3.15	10.1	3.09
3 20000000 0	63.2	61.6	1.03	33.2	1.90	19.9	3.18	19.9	3.18
4 20000000 0	69	67.1	1.03	36.6	1.89	22	3.14	22.5	3.07

REFERENCES

1 http://www.cs.cornell.edu/courses/cs2110/2009su/Lectures/examples/MedianFinding.pdf