# Parallel and Distributed Computing

Project Assignment

## $k$-NEAREST NEIGHBORS:

.

## THE BALL ALGORITHM

**Version 1.2 (26/3/2021)**

2020/2021

2nd Semester

## Contents

## Revisions

| | |
|---|---|
| Version 1.0 (March 12, 2021) | Initial Version |
| Version 1.1 (March 12, 2021) | Root of the tree should have id 0 |
| Version 1.2 (March 26, 2021) | Remove limitation for range of tree ids |

## 1 Introduction

The purpose of this class project is to give students hands-on experience in parallel programming on both shared-memory and distributed-memory systems, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of the Ball Algorithm that is used in the process of finding the $k$-nearest neighbors (kNN) of a given point in an $n$ dimensional space.

The problem of the kNN of a point in a multidimensional space is a fundamental step in many supervised machine learning algorithms. For high-dimensional spaces, the Ball Tree is a data structure that can be used to help improve the efficiency of this search. The Ball Tree is a type of binary tree that can be built from the initial samples in $O(n \log n)$ time and that allows searching for the set of nearest $k$ samples of any point in the space in logarithmic time. Hence, for large datasets, once this tree has been computed, the search for the kNN of a given point becomes extremely efficient. This is specially significant because this search operation is typically carried out repeatedly for different points.

In this project we provide the search routine, you will be developing the routine that builds of the Ball Tree.

## 2 The Ball Tree Algorithm

The algorithm to build a Ball Tree recursively partitions the input sample points into two groups, each inside its hypersphere (a "ball"). The partitions are defined by dividing in half the direction of higher spread of points. The algorithm is the following:

1. get input sample points;
2. compute points $a$ and $b$, furthest apart in the current set;
3. perform the orthogonal projection of all points onto line $ab$;
4. compute the center, defined as the median point over all projections;
5. create two sets of points, $L$ and $R$, defined as the points whose projection lies to one side or the other of the center;
6. create a tree node storing:
   - the coordinates of the center;
   - the radius, defined as the maximum distance between the center and any point in the current set of points;
   - a link to two children, $L$ and $R$.
7. repeat from 2 for each partition $L$ and $R$, unless the partition has a single point.

This procedure creates a balanced binary tree. The leaves of the tree are the input sample points, hence for these nodes the center will have the coordinates of the point it corresponds to and the radius will be zero.

# 3 Implementation Details

## 3.1 Input Data

The input sample points will be randomly generated, please use the routine provided in Appendix A so that everybody uses the same starting values, allowing the results to be validated against a golden solution.

Your program should allow three command-line parameters:

1. number of dimensions;
2. number of samples;
3. seed for random generator.

all 32-bit integers.

## 3.2 Output Data

Your program should send to the standard output, `stdout`, the Ball Tree in the following text format: the first line will have two integers, number of dimensions and number of nodes in the tree, in this order; then there should one line per tree node with this information about the node, in this order:

```
node_id left_child_id right_child_id radius center_coordinates
```

The first three values are integers, the remaining are floating-point. Please use 6 decimal digits for these values. Naturally, the `center_coordinates` should have as many values as the number of dimensions.

The order of the lines is not relevant.

Please use 0 for the id of the root node and -1 for the children of the leaves, for the other nodes you are free to make your own id assignment, as long as it fits in a long int.

For example, considering the example given in Section 3.5, the program should output (node ids may differ, but should be consistent):

```
$ ballAlg 2 5 0
2 9
2 -1 -1 0.000000 2.777747 5.539700
3 -1 -1 0.000000 3.352228 7.682296
1 2 3 1.109138 3.064987 6.610998
5 -1 -1 0.000000 7.830992 7.984400
7 -1 -1 0.000000 8.401877 3.943829
8 -1 -1 0.000000 9.116474 1.975514
6 7 8 1.047009 8.759176 2.959671
4 5 6 4.070595 8.682547 4.003873
0 1 4 5.141101 5.463001 5.592565
```

The submitted programs should only print this tree information (and **nothing else!**) to the standard output, `stdout`. We can redirect the standard output to a file and run a query program (that we will be supplying) on the tree produced by your program. We will use this to validate your implementation, and it will only work if the format above is strictly observed.

The project **cannot be graded** unless you follow strictly these input and output rules!

## 3.3 Measuring Execution Time

We don't want to take into account the output of the tree in the execution time, therefore we need to have a way of measuring time inside the program. We will use the routine `omp_get_wtime()` from OpenMP, which measures real time (also known as "wallclock time"). This same routine should be used by all three versions of your project.

Hence, your programs should have a structure similar to this:

```
#include <omp.h>
<...>

int main(int argc, char *argv[])
{
    double exec_time;

    exec_time = -omp_get_wtime();
    pts = get_points(argc, argv, &n_dims, &n_points);

    root = build_tree();

    exec_time += omp_get_wtime();
    fprintf(stderr, "%.1lf\n", exec_time);

    dump_tree(root);     // to the stdout!
 }
```

In this way the execution time will be sent to the standard error, `stderr`. This separation allows, for instance, sending the standard output to `/dev/null`, which is specially useful for very large instances, eg:

```
$ ballAlg 50 1000000 0 > /dev/null
8.1
```

Because this time routine is part of OpenMP, you need the include `omp.h` and compile your program with the flag `-fopenmp`.

## 3.4 Implementation Notes

Please consider the following set of notes:

- To minimize numerical errors due to rounding, use the type `double` for the floating-point numbers;
- Use the routine in Appendix A for the initialization of the input sample points;
- To create the two partitions, use the first coordinate of the projection of a point and place in partition $L$ the points that have this coordinate smaller than that of the center, otherwise place the point in partition $R$ (note that in this way points that coincide with the center will be assigned to $R$ – you may assume that there is at most one point in this situation);

- To determine the points $a$ and $b$ furthest apart in the current set use the following approximate approach: first compute $a$ as the most distant point from the first point in the set, i.e., lower index relative to the initial set; then, compute $b$ as the most distant point from $a$. Keep this rule in mind through the recursion.

You should adhere to these rules to better guarantee the validation of your results!
Some helpful hints:

- The orthogonal projection $p_o$ of a point $p$ onto line $ab$ can be computed using:

$$p_o = \frac{(p-a)\cdot(b-a)}{(b-a)\cdot(b-a)}(b-a)+a$$

  where $\cdot$ stands for the inner product;

- The point that defines the radius is not necessarily $a$ or $b$;

- Although there are more efficient approaches, feel free to use a sorting algorithm (eg, `qsort`) to find the median. If the number of points is odd, the point in the middle of the ordering will be the center. If it is even, you need to compute the intermediate point between the two points in the middle (average of each coordinate). However, later on, for parallelization purposes, if you find it useful you can invest in your own implementation.

## 3.5 Sample Problem

We illustrate the workings of this algorithm through a simple example. Consider that we run our program with the following parameters:

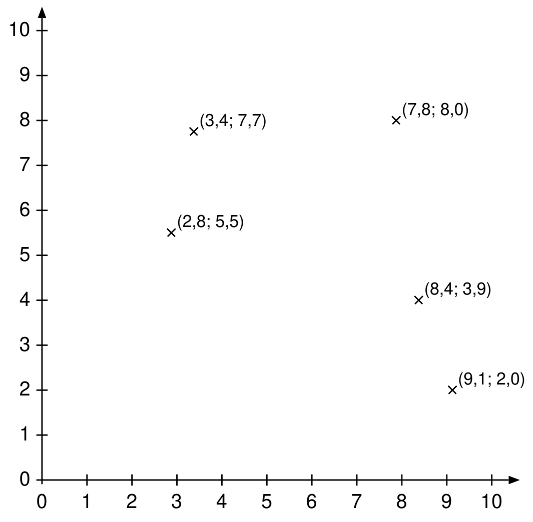
```
$ ballAlg 2 5 0
```

This will generate 5 points in a 2-dimensional space, using 0 as the seed for the pseudo-random number generator. The points that are generated are represented in Figure 1a.

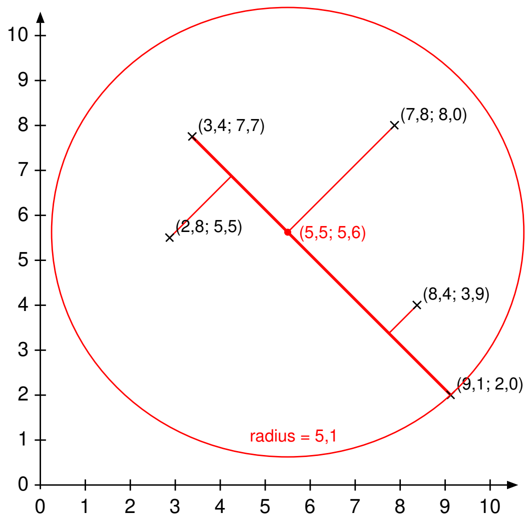
The two points that are identified has being furthest apart are $a = (3{,}4; 7{,}7)$ and $b = (9{,}1; 2{,}0)$, defining the direction of largest spread as the wider line segment in red in Figure 1b. All points are then orthogonally projected onto this line and the center for the root node is determined as $(5{,}5; 5{,}6)$, the intermediate projection onto line $ab$. The furthest point from the center is point $b$, which defines the radius of this node as $5{,}1$. We now observe that there are two points to the "left" of the center and three points to the "right" (remember the center goes here), defining the points in partitions $L$ and $R$, respectively.

The process above is repeated separately for the set of points in $L$ and $R$. Figure 1c shows the balls for each of these two partitions. The center for the left node of the root will be $(3{,}1; 6{,}6)$, with radius $1{,}1$, and for the right node the center is $(8{,}7; 4{,}0)$ with radius $4{,}1$. Each partition in the left node will have a single point, hence recursion stops there. On the right node, partition $R$ still has two points, thus the process needs to be repeated there. This defines a new ball at level 3 with center $(8{,}8; 3{,}0)$ and radius $1{,}0$, as depicted in Figure 1d.

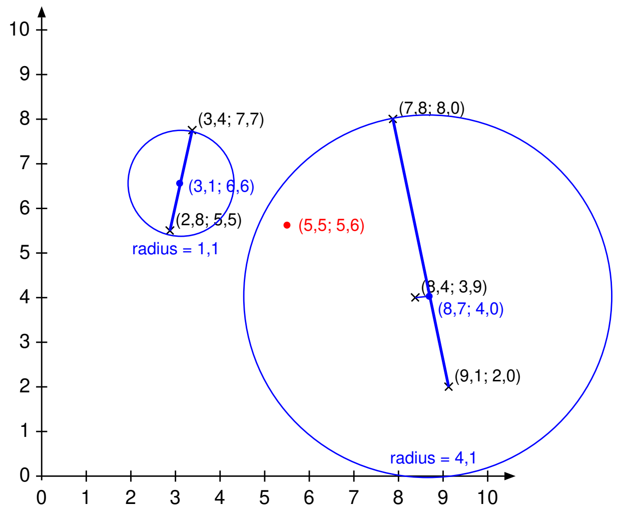
The resulting ball tree for the illustrative example is represented in Figure 2. In blue are the indexes of the tree nodes so you can compare with the output file in Section 3.2.



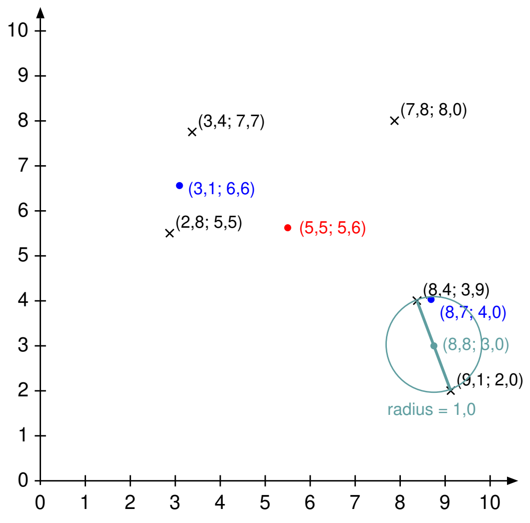
(a) Initial set of points.



(b) First ball.



(c) Second level balls.



(d) Third level balls.

**Figure 1: Balls defined at each level of the tree.**

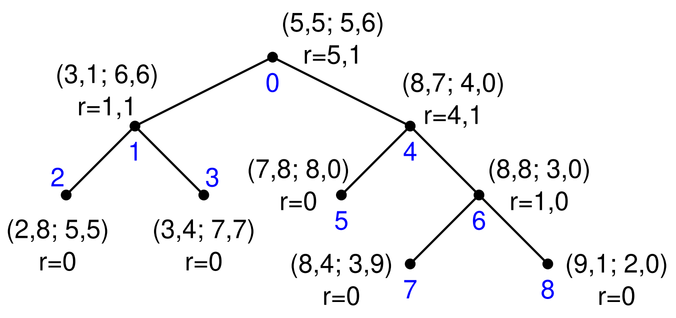


**Figure 2: Final ball tree for the illustrative example.**

## 4 Part 1 - Serial implementation

Write a serial implementation of the algorithm in C (or C++). Name the source file of this implementation `ballAlg.c`. As stated, your program should expect three input parameters.

Make sure to include a Makefile so that the simple command

```
$ make
```

should generate an executable with the same name as the source file (minus the extension). Also, to be uniform across groups, in this makefile please use the `gcc` compiler with optimization flag `-O3`.

This version will serve as your base for comparisons and must be as efficient as possible.

## 5 Part 2 - OpenMP implementation

Write an OpenMP implementation of the algorithm, with the same rules and input/output descriptions. Name this source code `ballAlg-omp.c`. You can start by simply adding OpenMP directives, but you are free, and encouraged, to modify the code in order to make the parallelization more effective and more scalable. Be careful about synchronization and load balancing!

Important note: in order to test for scalability, we will run this program assigning different values to the shell variable `OMP_NUM_THREADS`. Please do not override this value in your program, otherwise we will not be able to properly evaluate the scalability of your program.

## 6 Part 3 - MPI implementation

Write an MPI implementation of the algorithm as for OpenMP, and address the same issues. Name this source code `ballAlg-mpi.c`.

For MPI, you will need to modify your code substantially. Besides synchronization and load balancing, you will need to create independent tasks, taking into account the minimization of the impact of communication costs. You are encouraged to explore different approaches for the problem decomposition.

Note that this distributed version should permit running larger instances, namely instances that do not fit in the memory of a single machine.

Extra credits will be given to groups that present a combined MPI+OpenMP implementation.

## 7 What to Turn in, and When

You must eventually submit the sequential and both parallel versions of your program (**please use the filenames indicated above**), and a table with the times to run the parallel versions on input data that will be made available (for 1, 2, 4 and 8 parallel tasks for both OpenMP and MPI, and additionally 16, 32 and 64 for MPI).

You must also submit a short report about the results (2-4 pages) that discusses:

- the approach used for parallelization
- what decomposition was used
- what were the synchronization concerns and why
- how was load balancing addressed
- what are the performance results, and are they what you expected

You will turn in the serial version and OpenMP parallel version at the first due date, with the short report, and then the serial version again (hopefully the same) and the MPI parallel version at the second due date, with a new report. The code, makefile and report will be uploaded to the Fenix system in a zip file. **Name these files** as `g<n>omp.zip` and `g<n>mpi.zip`, where `<n>` is your group number.

1st due date (serial + OMP): **April 24**, until 23:59.
  Note: your project will be tested in the lab class right after this due date.

2nd due data (serial + MPI): **May 22**, until 23:59.
  Note: your project will be tested in the lab class right after this due date.

## A Routine to Generate Initial Sample Points

```
#include <stdio.h>
#include <stdlib.h>

#define RANGE 10

double **create_array_pts(int n_dims, long np)
{
    double *_p_arr;
    double **p_arr;

    _p_arr = (double *) malloc(n_dims * np * sizeof(double));
    p_arr = (double **) malloc(np * sizeof(double *));
    if((_p_arr == NULL) || (p_arr == NULL)){
        printf("Error allocating array of points, exiting.\n");
        exit(4);
    }

    for(int i = 0; i < np; i++)
        p_arr[i] = &_p_arr[i * n_dims];

    return p_arr;
}


double **get_points(int argc, char *argv[], int *n_dims, long *np)
{
    double **pt_arr;
    unsigned seed;
    long i;
    int j;

    if(argc != 4){
        printf("Usage: %s <n_dims> <n_points> <seed>\n", argv[0]);
        exit(1);
    }
    *n_dims = atoi(argv[1]);
    if(*n_dims < 2){
        printf("Illegal number of dimensions (%d), must be above 1.\n", *n_dims);
        exit(2);
    }
    *np = atol(argv[2]);
    if(*np < 1){
        printf("Illegal number of points (%ld), must be above 0.\n", *np);
        exit(3);
    }

    seed = atoi(argv[3]);
    srandom(seed);

    pt_arr = create_array_pts(*n_dims, *np);
    for(i = 0; i < *np; i++)
        for(j = 0; j < *n_dims; j++)
            pt_arr[i][j] = RANGE * ((double) random()) / RAND_MAX;

    return pt_arr;
}
```