Parallel Spherematch

Ehsan Kourkchi
University of Hawai'i at Mānoa, Honolulu, HI, U.S.A.
ehsan20@hawaii.edu

***Abstract*—k-d trees are useful data structures to store multidimensional points in such a way to preserve the relative position of the data points. Using this structure, we find all possible pairs of data points, distributed on a sphere, that are within a particular angular separation. We use the *Astrometry.net* spherematch package which is originally developed in C with a Python interface for astronomers. We introduce new functions in the level of Python-C extension of the package that enables us to use the power of OpenMP. We find that however adding more threads increases the efficiency of spherematch, the total run-time is dominated by other operations that need to be done in Python. Therefore, we implement the other intensive operations in C where we can also use OpenMP. This together with breaking the entire job into several smaller sub-tasks help of to reduce the run-time of individual sub-tasks, and therefore the total queue waiting time, that can dominate the total run-time.**

I. Introduction

The analysis of two-point correlation functions of the spatial distribution of astronomical objects is a very well established way to understand how the objects might be grouped together. This provides a tool to study the scale of clustering. Data points are the coordinates of the observed objects. Although objects are distributed in 3-dimensional space, it is easier put a constraint on the line-of-sight dimension to form a 2-dimensional distribution of the objects on the surface of a sphere. Then positions can be expressed in spherical coordinates for which we can use all available tools of spherical geometry/mathematics.

In this study, the ultimate scientific goal is to understand how Quasars (quasi-stellar radio sources) are clustered in the early Universe. Quasars are extremely luminous star-like objects that produce strong emissions through the accretion of materials into a central black-hole of a galaxy with active nuclei. Quasars are mainly present in the early stages of the Universe where galaxy formation is in its very early stages. Since Quasars are gravitationally hosted by large massive dark matter haloes, studying their spatial distribution enables us to understand how the dark matter is structured.

Given the position of a set of Quasars, the following estimator can describe the degree of clustering in different angular scales

$$\xi(\theta) = \frac{\langle DD(\theta)\rangle - 2\langle DR(\theta)\rangle + \langle RR(\theta)\rangle}{\langle RR(\theta)\rangle}, \quad (1)$$

where $DD(\theta)$ is the number of data-data pairs with angular separation in the range of θ and θ+$\Delta\theta$ and $DR(\theta)$ is the number of data-random pairs also separated by θ [1]. The random catalog is a synthetic catalog which has the exact sky coverage and characteristics of the data catalog, but randomly distributed. The random catalog is constructed to be larger than the data catalog to reduce Poisson noise due to the pair counts that include random points. The idea of the above estimator is to measure how much the distribution of the observed Quasars is deviated from the random distribution as an indicator of the clustering strength.

In our study, we have 91,716 data points, for which we have 652,012 random points. The exhaustive way to estimate $\xi(\theta)$, is to calculate the angular distance of all pair of points. This can take a long time, since for 600,000 data points we need to calculate $\sim 10^{11}$ angular separations. Assuming that each calculation of angular separation takes more than 1 μs, we need at least 50 hours to go through all the data points. Therefor, re-estimation of $\xi(\theta)$ for different sets of data with different random catalogs seems to be impossible. In fact, the complexity of the exhaustive method is $O(n^2)$. On the other hand, we only need to calculate $\xi(\theta)$ for $\theta < 75^o$, which means that there is no need to know the angular separation of all pairs, while only a small fraction of the pairs have $\theta < 75^o$. This suggests that there might be some heuristic algorithms to avoid unnecessary calculations.

k-d trees, first introduced by Bentley (1975) [2], are multidimensional binary tree data structures to store k-dimensional data. In this structure, neighbor points are stored next to each other to form a hierarchical structure with low cost operations. A balanced k-d tree can be constructed in $O(knlogn)$ time [3]. The complexity of insertion and deletion is $O(logn)$, and search can be done in $O(nlogn)$ [2]. Because of its better performance, almost all algorithms, that try to match two sets of points, use k-d trees. Recently, buffer k-d trees have been introduced to provide a tool to process many nearest neighbor searches on many-core

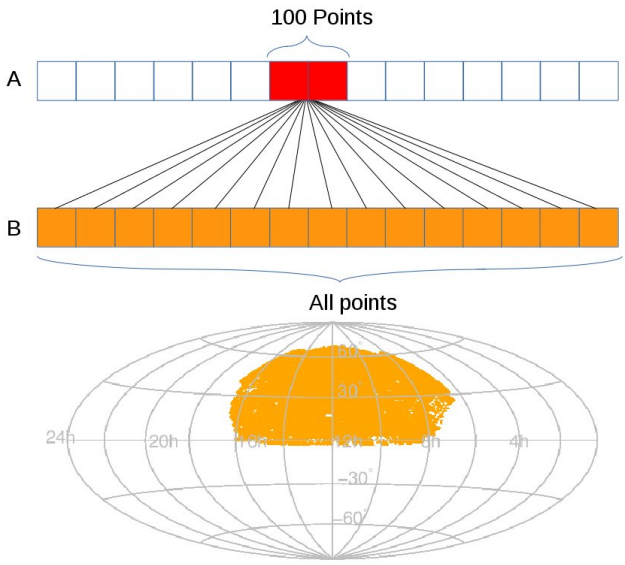

Figure 1: Top: The cross correlation of a fraction of data (A) with entire data-set (B). In this study, only 100 point is used to construct k-d tree at each step and then the angular separation of those pairs with $\theta < 75^o$ is calculated and returned. Bottom: The aitoff projection of the distribution of the random points (652,012 points).

devices like GPUs [4], [5]. For our purpose, the idea is to construct a k-d tree using all the points. Then for any given point, performing a query finds a set of nearest points in the tree within a specific angular separation (say $\theta < 75^o$). This way, k-d tree enables us to better avoid those points that have larger angular separations. In another word, since all the points are stored based on their spatial positions, i.e. neighbor points occupy the same region in the tree, it is easier to traverse the tree and avoid those far away points that are not of our interest.

II. Resources

A. Data

We have 91,716 observed Quasars, all located by Sloan Digital Sky Survey telescope[1]. For the purpose of estimating the two-point correlation function, a catalog of 652,012 random data point is constructed. All points in this synthetic random catalog are randomly distributed, however its other characteristics statistically mimic the observed data. Since the the number of points in the random catalog is much larger and the calculation time is dominated by the calculation of $\langle RR(\theta)\rangle$ in the Equation 1, we only focus on the random catalog in this report. Also we use data points and random points interchangeably. The spatial distribution of the interested random point is displayed in the bottom panel of the Figure 1. The line-of sight distance of all random points are almost the same, therefore we can assume that all point are distributed on the surface of a sphere as seen in this figure.

B. Computational Resources

We use University of Wyoming Mount Moran cluster[2] which consists of 284 nodes. Each node has two 8-core (16 cores in total) Intel E5-2670 (Sandy Bridge) 2.6 GHz processors, each with 20 MB cache. We use OpenMP multi-thread programs in this project and therefore we only use one node but different number of cores. We refer to this cluster as *Mt. Moran* throughout this report. This cluster uses *Slurm* for job-scheduling.

To test the codes and their performance, we sue a desktop PC with a quad-core 3.4 GHz CPU (Intel i7-3770). Each core can run two threads separately. Therefore, maximum 8-threads can be used when using OpenMP. We refer to this machine as *Desktop PC* hereafter.

III. Method

We use the *Astrometry.net-0.64* spherematch package[3] which has been developed in C with a Python friendly interface for the use of astronomers. This package provides all the necessary functions to match two sets of points distributed on the surface of a sphere and finds all the desired pairs. The goal is to manipulate this package to add the capability of OpenMP use. Originally, one can use "spherematch" in Python by calling the following function of the package

```
spherematch.match(xyz_small, xyz, r)
```

where $\texttt{xyz}_{small}$ is the coordinates of the first set of points based on which a k-d tree is constructed. `xyz` is the set of points that would be cross correlated with $\texttt{xyz}_{small}$, and `r` is the maximum angular separation of the desire pairs (for out purpose $r = 75^o$). the output of the functions is the indices of the data_{small}-data pairs. Since we are dealing with large number of points (∼600,000 points), constructing a k-d tree, that is traversed recursively, needs a large memory and it is practically impossible when we use all the points at once. Thus, the strategy is to breaking the data points into smaller sets (i.e. xyz_{small}) each of which has 100 points, and then cross correlated it with the entire data-set. We are interested to find the number of pairs for different intervals of angular separations. Therefore, for each 100 sub-set we calculate the angular histogram of pairs. Then we only save the histograms. This way

[1] http://www.sdss.org/

[2] https://arcc.uwyo.edu/

[3] http://astrometry.net/downloads/

we avoid storing information about the pairs, when we only store their statistics. At the end after covering the entire data-set, we add all the generated histograms to obtain the final result. The top panel in Figure 1 schematically shows how this process is done. The top array (A), shows how we span the data-set by dividing it into smaller pieces, each with 100 points. 100 is the number we found empirically after testing our codes on Mt. Moran cluster and on Desktop PC. Larger numbers causes extreme resource usage that leads the OS to terminate the program right away[4]. Array (B) represents the entire data-set against which 100 points get cross correlated.

There are three different levels in this package. All fundamental functions and applications are in C and we leave them intact. We are just interested in codes that are in `astrometry.net-0.64/libkd` directory of the package, that will hold all the manipulated codes and scripts. The very last level is in C and provides a platform for the Python C-extension (`pyspherematch.c`). This file contains a set of interface functions that talk to lower level C-functions. When the entire C-package is compiled, a shared library is produced that is accessible by a higher level python code (i.e. `spherematch_c.so`). The next higher level code is in Python and provides all the interface functions in python (i.e. `spherematch.py`) and it imports the shared library `spherematch_c`. This provides a platform for all other user-implemented codes in Python, normally developed by astronomers. We introduce our new functions with capability of using OpenMP in `pyspherematch.c` and then recompile the entire package to produce the updated library `spherematch_c.so`. In addition, we need to add more functions in `spherematch.py` to provide an interface for users to call our newly implemented functions in Python. It should be noted that `setup.py` needs to be updated to include OpenMP flags when compiling `pyspherematch.c`.

After breaking the codes, we move several operations from `spherematch.py` to `pyspherematch.c`. This way we have a better control over the pointers and the array of k-d trees that are created for multi-process purpose. At all levels of development, we constantly check the new codes against the original version to make sure that they produce the desired results. As seen in Figure 1, we can divide the array (B) into several pieces and run the query in parallel, while all processes use the same k-d tree that contains 100 points of the array (A). After all, we introduce the following function that uses OpenMP,

[4]Building k-d trees might not be the cause of termination. One needs to dig in more details to see how the tree is queried

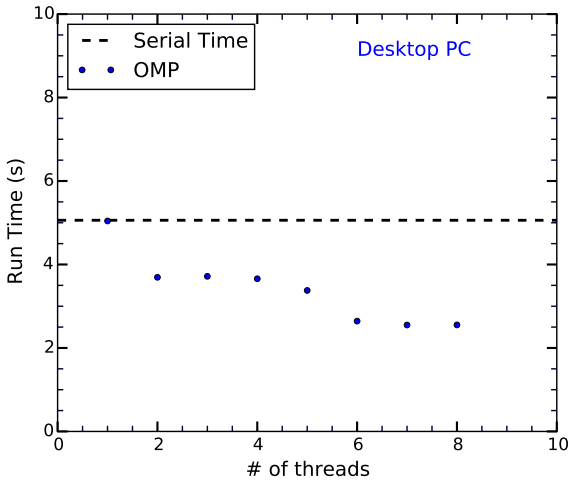

Figure 2: Run-time of each sub-task in terms of the number of threads for Desktop PC. Black dashed line represents the run-time of the original function, i.e. `spherematch.match`.

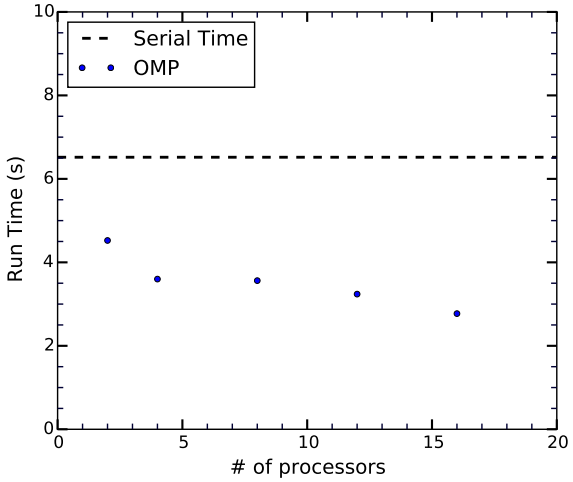

Figure 3: Same as Figure 2 for Mt. Moran cluster, using different number of processors, 2, 4, 8, 12, 16.

`spherematch.match_esn_omp(xyz_small, xyz, r, n)`.

The arguments of this function are the same as the original function. It takes an extra parameter, n, which indicates the number of elements of the array (B) for running in multi-thread mode. For instance, setting $n = 100$ means that each thread would take 100 points while all threads loop through all divisions to cover the entire data-set, i.e. array (B).

A. Results

The strategy is to examine how much speed-up we can get by using OpenMP. Our random catalog includes 652,012 data points that defines 6,521 sub-tasks. We number all of these tasks based on their positions on the array. For example the red sub-array (A), in Figure 1, defines a task with IDs ranging from 0 to 6,521. To test the performance, and in order to sample the entire data-set at different regions on the sky, we use 7 sub-tasks with IDs of 0, 1000, 2000, 3000, 4000, 5000, 6000. We use 10 iterations for each sub-task and we take the median run-time value. We observe very small deviations in run-time of different sub-tasks, therefore we report the median value of the run-time for all 7 sub-tasks.

First we use the original form of the function, `spherematch.match`, which produces the black dashed line in Figure 2. We use the OpenMP version of this code, `spherematch.match_esn_omp`, to use multiple threads/cores and compare the results.

As seen tin Figure 2, the best achieved speed-up is $\sim$2 when the number of threads is larger than 5. For 2 threads, the speed-up is $\sim$1.4 which is less than two, meaning that we need to spend some overhead time to produce and reduce threads. Increasing the number of threads does not change the run-time, implying that the run-time is dominated by the overhead time. In addition, the measured small values for I/O time (i.e. at most 0.01 seconds) suggests that even running the original serial code independently on multiple cores might give us a better performance, than using OpenMP. Running the same experiment on Mt. Moran cluster leaves us with the same conclusion. Figure 3 shows that two processors run the code $\sim$1.4 times faster, and even with 16 cores, the speed-up is slightly better than 1.5.

We change the order of data points in order to see how it affects the performance. The idea is that successive query of the neighbor data points results in less cache miss, when we need to less frequently load the different parts of the k-d tree. It should be noted that our random catalog is originally ordered in such a way that the neighbor points are almost stored next to each other. To do that, we use HEALPix[5] nested indices with the resolution of $N_{side} = 256$. All points which are inside the same nested pixel would have the same nested index. Sorting the data points in terms of their nested indices results in an array whose points are physically close to each other on the sphere. In order to change this nice ordering fashion, we randomly shuffle the data array and explore its effect on the performance. Figure 4 shows the results which is the same as the results presented in Figure 3. Although we expect to see a worse performance when we shuffle the data points, still we can not see its effect, which is likely due to the small size of the k-d tree that entirely fits in the cache.

[5]http://healpix.jpl.nasa.gov/

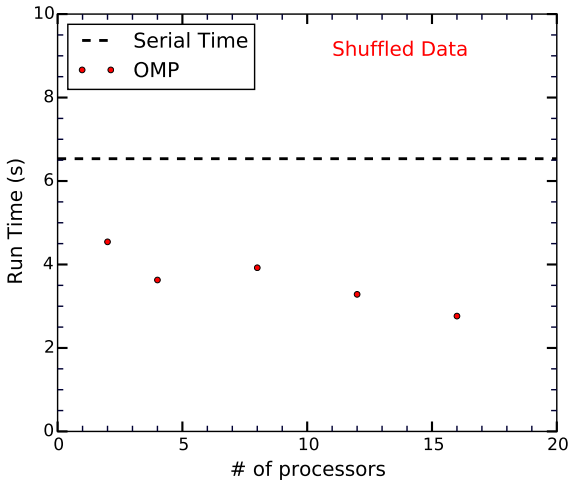

Figure 4: Same as Figure 3, but for shuffled data points.

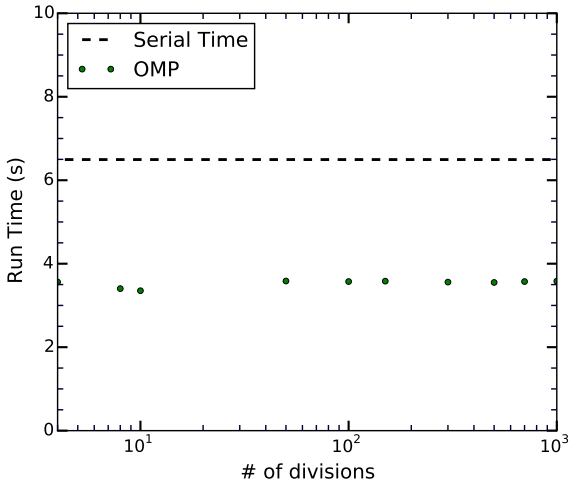

Figure 5: Run-time versus the size of the divided data (horizontal axis), using 4 processors on Mt. Moran cluster. *Note: The label of the x-axis in this plot needs to be replaced by "sub-array size".*

We also play with the size of divisions, n, when using `spherematch.match_esn_omp`. Figure 5 shows the run-time using 4 processors on Mt. Moran cluster for variety of n parameters. As seen, the efficiency does not depend on the size of sub-arrays (because the sizes are such small that doesn't matter to see the effect of cache miss) and therefore we use $n = 100$ for all of our other calculations.

Table I: The number of pairs with $\theta < 75^o$ for different sub-tasks.

sub-task ID	No. of pairs
0	57,062,498
1000	57,881,907
2000	55,562,024
3000	55,856,268
4000	56,762,397
5000	56,946,478
6000	56,518,878

IV. More Analysis

For our real analysis and to calculate $\xi(\theta)$ in Equation 1, we need to count the number of pairs for each angular separation interval, θ and $\theta+\Delta\theta$. Therefore, for each sub-task we need to find the histogram of angular separation of desired pairs with $\theta < 75^o$.

Normally this process is done in Python, however considering the huge number of pairs given each sub-set (i.e. 60 million pairs), it is in-efficient to calculate the histogram in Python. Table I shows the number of the matched points for 7 chosen sub-tasks.

Check out Table I.

In order to improve the efficiency, we implement the histogram calculation and all other mathematics in C by adding more specialized function in `pyspherematch.c`, which also uses OpenMP for both 'spherematch' and 'histogram calculations'. The new function is as follows

```
spherematch.match_sss_omp(xyz_small,
xyz, r, s, low, powmin, powmax,
n_bins, 100),
```

where `xyz`$_s$`mall`, `xyz` and `r` are as before. `'s'` is an array that contains a single value for each point and has the same size as the array `'xyz'`. For each pair, a combinations of the associated `'s'` values is used to calculate the `'s'` value of the pair. At the end, the histogram of `'s'` value of the pairs is the desired parameter . `'powmin'`, `'powmax'` are the values to determine the minimum and maximum angular limit of the histogram, and `'n_bins'` is the number of bins.

We compare the performance of the above C-implemented function with that of `spherematch.match_esn_omp` when it is used together with built-in mathematical functions in Python. Figure 6 shows how the performance is improved when we implement everything in C. Black dashed line shows the run-time of the Python code for each sub-task and blue points represent the run-time of the new function, `spherematch.match_sss_omp`.

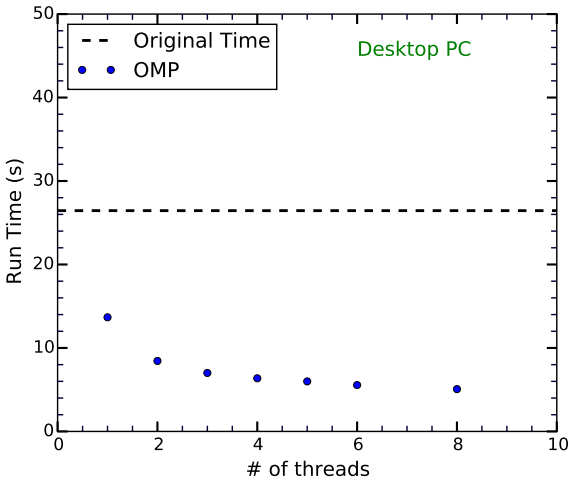

Figure 6:
Run-time of `spherematch.match_sss_omp` for different number of threads on Desktop PC. Black dashed line represents the run-time of the Python code when it only calculates the histograms of the resulting pair produced by `spherematch.match_esn_omp`.

With only one processor, this function runs ∼1.8 times faster and its speed-up reaches to ∼5 when using 8 threads. Figure 7 shows the same results on Mt. Moran cluster when different number of cores are in use. In this case, comparing with Python performance (dashed line), `spherematch.match_esn_omp` results in a speed-up of ∼1.6 while the best performance is achieved using 4, 5 or 6 core where speed-up is ∼5.3. Relative to the single core run, using 4 cores results in the speed-up of ∼2.5. As previously mentioned, this suggests that the performance of single process run might be better if multiple cores run the code separately and produce their own histograms. This way, we can achieve the nominal speed-up of p, when using p processors, since starting up each sub-task and I/O does not take so much time.

V. Conclusion

In order to calculate the level of clustering of a set of points distributed on a sphere, $\xi(\theta)$, we employ k-d trees to find the pairs whose angular separations is smaller than 75^o by using the C-Python package *Astrometry.net*. The time complexity of our calculations is dominated by autocorrelation analysis of a catalog of 652,012 random pints. In this package, `spherematch` is a function that finds all the desired pairs where we can modify its performance by changing its very last-level interface code, that is in C, that provides a platform for all other Python calls. We introduced new OpenMP friendly `spherematch` functions, that help

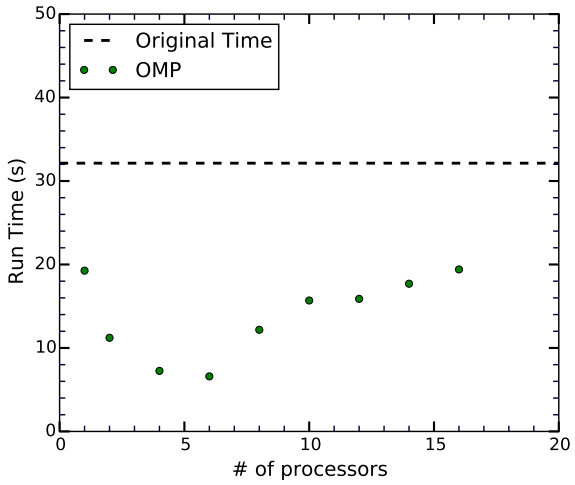

Figure 7: Same as Figure 6, for different number of processors on Mt. Moran cluster. The best performance is achieved with 5 processors where speed-up is $\sim$3.5 relative to the single processor run.

us to improve the performance of the calculations. More through analysis reveals that the run-time of these codes are dominated by the overhead time of thread creation and annihilation. On the other hand, considering the large number of pairs, and time expensive mathematical processes in Python, we implemented all other process in C, as well. This enabled us to achieve the speed-up of $\sim$5 when using 4 processors. We need to divide our big task into smaller sub-task with short run-times. Decreasing the run-time helps us to reduce the queue waiting time when submitting the array of sub-tasks. Finally, applying for 4 processors on Mt. Moran for each sub-task, we could calculate $\langle RR(\theta)\rangle$ in about 4 hours. The total needed sequential time for the entire task is 13.5 hours when everything is implemented in C. The run-time estimation of the original sequential Python code is $\sim$65 hours. The next challenge is to optimize this code to be run in less than an hour on the cluster. Since we need to estimate the uncertainty of $\xi(\theta)$ by removing different patch of the sky and repeating the entire calculation, reducing the run-time would be a great. One idea is to store the information of the pairs once they are discovered and then trying to use them over and over. However the pairs remains the same, storing and recovering their information needs a huge memory resource and I/O time, which is even more time consuming than the original calculations. Therefore, at this moment it sounds reasonable to store the information of pairs, by calculating their statistics on the fly.

VI. Acknowledgment

We acknowledge the use of Mount Moran cluster at the University of Wyoming under the HOD (Halo Occupation Distribution) project.

References

[1] S. Eftekharzadeh, A. D. Myers, M. White, D. H. Weinberg, D. P. Schneider, Y. Shen, A. Font-Ribera, N. P. Ross, I. Paris, and A. Streblyanska, "Clustering of intermediate redshift quasars using the final SDSS III-BOSS sample," *Monthly Notices Royal Astronomical Society*, vol. 453, pp. 2779–2798, Nov. 2015.

[2] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 18, pp. 509–517, 1975.

[3] R. A. Brown, "Building a balanced k-d tree in $o(kn \log n)$ time," *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 1, pp. 50–68, March 2015. [Online]. Available: http://jcgt.org/published/0004/01/03/

[4] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs," in *Proc. of The 31st International Conf. on Machine Learning*, 2014, pp. 172–180.

[5] F. Gieseke, C. Eugen Oancea, A. Mahabal, C. Igel, and T. Heskes, "Bigger Buffer k-d Trees on Multi-Many-Core Systems," *ArXiv e-prints*, Dec. 2015.