Research in Astronomy and Astrophysics, 23:105011 (11pp), 2023 October https://doi.org/10.1088/1674-4527/acee50

3DT-CM: A Low-complexity Cross-matching Algorithm for Large Astronomical Catalogues Using 3d-tree Approach

Yifei Mu[1,2], Ce Yu[1,2], Chao Sun[1,2], Kun Li[1,2], Yajie Zhang[1,2], Jizeng Wei[1,2], Jian Xiao[1,2], and Jie Wang[3]
[1] College of Intelligence and Computing, Tianjin University, Tianjin 300350, China; weijizeng@tju.edu.cn
[2] Technical R&D Innovation Center, National Astronomical Data Center, Tianjin 300350, China
[3] Xinjiang Astronomical Observatory, Chinese Academy of Sciences, Urumqi 830011, China
Received 2023 May 15; revised 2023 July 10; accepted 2023 July 31; published 2023 September 20

Abstract

Location-based cross-matching is a preprocessing step in astronomy that aims to identify records belonging to the same celestial body based on the angular distance formula. The traditional approach involves comparing each record in one catalog with every record in the other catalog, resulting in a one-to-one comparison with high computational complexity. To reduce the computational time, index partitioning methods are used to divide the sky into regions and perform local cross-matching. In addition, cross-matching algorithms have been adopted on high-performance architectures to improve their efficiency. But the index partitioning methods and computation architectures only increase the degree of parallelism, and cannot decrease the complexity of pairwise-based cross-matching algorithm itself. A better algorithm is needed to further improve the performance of cross-matching algorithm. In this paper, we propose a 3d-tree-based cross-matching algorithm that converts the angular distance formula into an equivalent 3d Euclidean distance and uses 3d-tree method to reduce the overall computational complexity and to avoid boundary issues. Furthermore, we demonstrate the superiority of the 3d-tree approach over the 2d-tree method and implement it using a multi-threading technique during both the construction and querying phases. We have experimentally evaluated the proposed 3d-tree cross-matching algorithm using publicly available catalog data. The results show that our algorithm applied on two 32-core CPUs achieves equivalent performance than previous experiments conducted on a six-node CPU-GPU cluster.

Key words: methods: data analysis – catalogs – techniques: miscellaneous

1. Introduction

Cross-matching plays a vital role in astronomical data fusion, enabling the correlation of records from different catalogs. In recent years, with the development of multi-messenger astrophysics, more sky survey projects were implemented, which provided multi-band astronomical catalog data. Additionally, advancements in observational precision and duration have resulted in massive catalog data sets. For example, Gaia DR2 (GAIA 2018) contains over 160 million sources, 2MASS (2MASS 2006) holds about 470 million sources, and each data release from version 12–17 of SDSS (SDSS 2022) has collected more than 1.2 billion records of catalog objects. Thus, efficient cross-matching algorithms are critical for astronomical scientific computing using massive catalogs.

The differences in observation instruments and calibration methods cause acceptable slight disturbance, and making it untrivial to correlate the same celestial object in two different catalogs. Cross-matching is the method used to correlate two records within slightly different coordinates. The classic cross-matching formula between two points is an angular distance computation conducted from the Equatorial cosine theorem or the haversine formula. When the angular distance is less than the threshold determined by hyper-parameters, cross-matching is considered successful. Cross-matching each record in catalog A with each record in catalog B is the most apparent method for cross-matching two catalogs, but it is time-consuming as astronomical catalog size increases (Szalay et al. 2004).

To accelerate the cross-matching process in astronomical data fusion, various methods have been employed to reduce computation time, including data division (Gray et al. 2007) (Gorski et al. 2005) (Szalay et al. 2007) and parallel computing (Zečević et al. 2019) (Li et al. 2019) (Zhao et al. 2009). Index partitioning methods have been particularly useful, but they suffer from source leaking at the border of each area. To address this issue, mixed indexing (Yu et al. 2020) and border data redundancy methods (Jia & Luo 2016) (Zhang et al. 2023) have been introduced.

Hardware structure has also been utilized to speed up the cross-matching process, with various parallel computing methods being employed (Zečević et al. 2019) (Zhang et al. 2023). However, few of these methods have focused on improving the cross-matching algorithm itself. In this paper, we propose a novel approach using kd-tree, a data structure that efficiently handles dimensional spatial information. By

Table 1
Notations

Notation	Description
(α, δ)	(Right ascension, decl.) in Equatorial Coordinate System
(x, y, z)	Corresponding position of (α, δ) in Euclidean Coordinate System
$d_A(O_1, O_2)$	Angular distance of objects O_1 and O_2 in Equatorial Coordinate System
$d_E(O_1, O_2)$	Euclidean distance of objects O_1 and O_2 in Euclidean Coordinate System
θ_A	$\theta_A = 3\sqrt{r_1^2 + r_2^2}$, to represent the threshold of error radius, where r_1, r_2 are the error radius of catalog 1 and catalog 2
θ_E	To represent the threshold of error radius in Euclidean Coordinate
N, M	Record number of catalog 1 and catalog 2
N_{area}	The sky area number after using index partitioning method

converting the Equatorial distance computation into an equivalent Euclidean distance computation, we demonstrate that kd-tree can be used to accelerate the query process. We show that performing the coordinate conversion leads to a significant improvement in performance since kd-tree pruning in Equatorial coordinate systems has high computational complexity.

Our contributions can be summarized as follows. First, we propose a novel 3d-tree based cross-matching algorithm and parallelize it in a big-memory computer, achieving performance comparable to that of index partitioning methods. Second, we efficiently utilize the resources of our environment, both in terms of memory and CPU, to achieve the best performance within just 3 minutes, even with billions of records. Lastly, we provide insights into why the performance of our 3d-tree based algorithm is superior to that of the traditional 2d-tree, as discussed in Section 3.2.

The rest of this paper is organized as follows: Section 2 provides background information of cross-matching and kd-tree. Section 3 introduces our efficient 3d-tree method. Section 4 presents the evaluation of our work, and Section 5 concludes this paper.

2. Background and Related works

2.1. Cross-matching Function

Position-based cross-matching is a method to calculate distance of two celestial bodies and determine if it is less than the error radius. In Equatorial Coordinate System, each record's position is represented in (α, δ) with ranges in [0, 360) and [−90, 90]. Then using Equatorial cosine theorem (1) or haversine formula (2), the angular distance between two records is calculated. Table 1 summarizes the major notations used in this paper.

Due to the differences of observational method, equipment and condition, there are slightly disturbance in position of the same object in different records. Thus, a distance threshold based on the calibration errors and other considerations, also called a search radius, is used to determine if two positions correspond to the same celestial object.

$$d_A = \sqrt{(\alpha_1 - \alpha_2)^2\left(\frac{\delta_1 + \delta_2}{2}\right)^2 + (\delta_1 - \delta_2)^2} \leqslant \theta \quad (1)$$

$$d_A = 2\arcsin\left(\sqrt{\sin^2\left(\frac{\delta_1 - \delta_2}{2}\right) + \cos(\delta_1)\cos(\delta_2)\sin^2\left(\frac{\alpha_1 - \alpha_2}{2}\right)}\right) \leqslant \theta \quad (2)$$

2.2. Kd-tree

In computer science, a kd-tree (short for *k*-dimensional tree) is a space-partitioning data structure used to organize points in a *k*-dimensional space. These trees are useful for several applications, such as multidimensional search operations (Wan et al. 2019) involving a search key (e.g., range search shown in Figure 1) and creating point cloud regions (Zhou et al. 2008) (as shown in Figure 2). kd-trees are a special case of binary space partitioning trees, and each node contains attributes listed in Table 2 (Moore 1991).

The "dom-elt" field in a node of a binary tree represents the domain-vector of the exemplar, while the "range-elt" field represents the range-vector. The "dom-elt" component is an index for the node, which divides the space into two subspaces based on the splitting hyperplane of the node. All the points in the "left" subspace are represented by the left subtree, while the points in the "right" subspace are represented by the right subtree. The splitting hyperplane is a plane that passes through the "dom-elt" and is perpendicular to the direction specified by the "split" field. Let "*i*" be the value of the "split" field. A point is considered to be on the left of "dom-elt" only if its "ith" component is less than the "*i*th" component of "dom-elt". The complimentary definition holds for the "right" field. If a node has no children, the splitting hyperplane is not required.

2.3. Cross-matching Related Works

The original cross-matching method cost little time when astronomical data are under small volume. With data volume grows, large-scale cross-matching that involve all or a large fraction of the sky cannot be performed on demand.

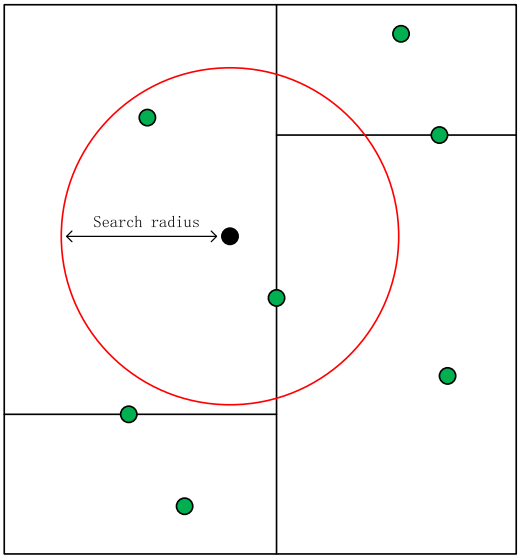

Figure 1. Range search in 2d-tree.

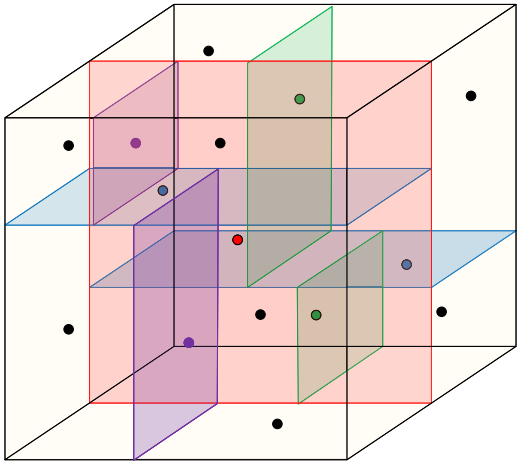

Figure 2. Diagram of 3d-tree, where each plane separates its space into two subspaces and the point cloud is divided into regions.

Table 2
Kd-tree Attributes

Field Name	Field Type	Description
Dom-elt	Domain-vector	A point from k dimension space
Range-elt	Range-vector	A point from 2^*k dimension space
Split	Integer	The splitting dimension
Left	Kd-tree	A kd-tree representing those points to the left of the splitting plane
Right	Kd-tree	A kd-tree representing those points to the right of the splitting plane

Contemporary cross-matching method facilitates index partitioning method to decrease calculation complexity. Zone (Gray et al. 2007) and the Hierarchical Equal Area isoLatitude Pixelisation (Healpix) (Gorski et al. 2005) are mostly used methods. The common idea of them is that, the whole sky is partitioned into a fixed number of regions and each celestial object in the same region are assigned to the same index or indices. Therefore, only adjacent regions will be used to perform cross-matching, which will reduce the computational complexity.

Zones method is proposed facing demand of SQL. Its a way of bucketing two-dimensional spaces (or 2+D spaces) to give dynamically computed bounding boxes for queries. The basic idea is to map the celestial sphere into zones, each zone is a decl. stripe of the sphere with some fixed height as shown in Figure 3 (left). Healpix is a hierarchical sky partitioning developed at NASA to facilitate fast and accurate statistical and astrophysical analysis of massive full-sky data sets. At level 0, the sky is divided into 12 pixels. Then, at each successive level, the pixels are divided into four new pixels like Figure 3 (right).

In terms of Zone method, Szalay et al. (2004) proposed and implemented the zone-based cross-matching with some SQL extensions in a single Microsoft SQL server. Nieto-Santisteban et al. (2006, 2007) implemented a parallel zones algorithm on multiple server and cross match of two million-record catalogs was done under 20 minutes with eight servers. Kumar et al. (2009) extended the parallel zone-based cross-matching to a hybrid MySQL cluster and the optimized algorithm was able to cross-match two catalogs with size of 3 million objects and 30551 objects respectively in seven seconds. Wang et al. (2013) and Budavari & Lee (2013); Lee & Budavári (2013) parallelized the zones algorithm on a single GPU and a single node with multiple GPUs, respectively. With the aid of GPUs, these methods completed cross-matching of million-scale catalogs in a few seconds. For cross-matching partially overlapping catalogs, Fan et al. (2013) optimized the original zone algorithm by first filtering out irrelevant objects with sky coverage information. AXS (Zečević et al. 2019) extended Zones algorithm to adapt it for a distributed, shared-nothing architecture. It is applied in cluster with 28 executors and using spark to achieve less-than minute performance in billion degree data sets.

Many works also employed HEALPix as the partitioning scheme and performed cross-matching of two catalogs in partitioned regions. Zhao et al. (2009) performed cross-matching of SDSS DR6 (100 million objects) and 2MASS (470 million objects) in 32 minutes on a single SQL server with MPI. Pineau et al. (2011) used two hyper-threaded quad-core CPUs to finish cross-matching of 2MASS (470 million objects) with USNOB1 (1 billion objects) in 30 minutes. Jia et al. (2015) took an indexed-loop join approach utilizing the HEALPix index and cross-matched billion-record catalogs on a seven node CPU-GPU cluster under 10 minutes. MASJ (Jia & Luo 2016) proposed a GPU based algorithm to fully use the performance of GPU in dividing healpix area and parallel

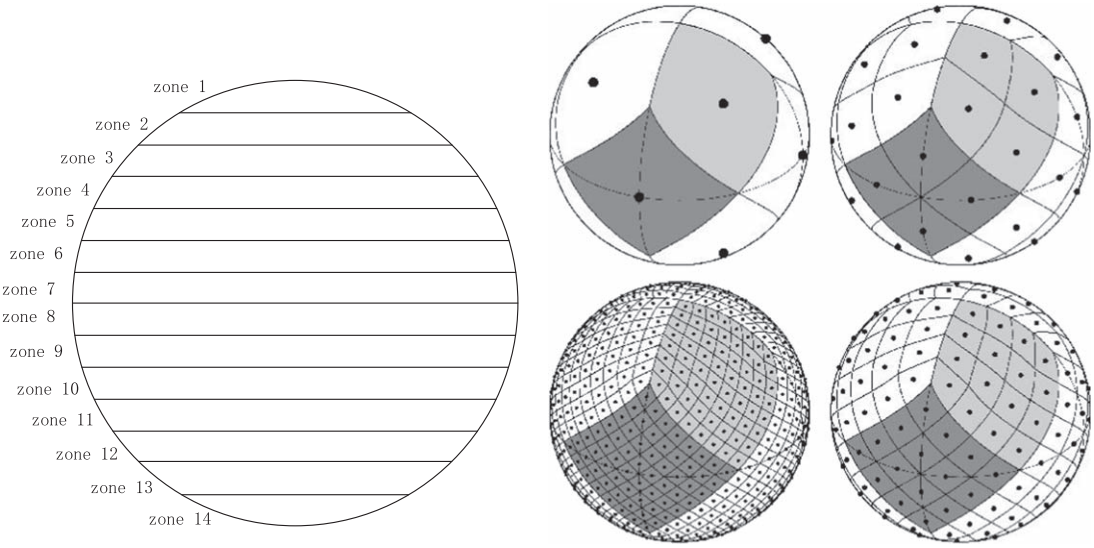

Figure 3. Zone(left) (Gray et al. 2007) and Healpix(right) (Gorski et al. 2005) index partitioning schematics.

cross-matching. It uses 12 GPU to accelerate cross-matching process and achieve minutes degree speed-up under billions of records. HLC2 (Zhang et al. 2023) also presented an efficient cross-matching framework under CPU-GPU architecture. When applying cross-matching algorithm in database system, Han et al. (2016) used Q3C index method in PostgreSQL database and provided four different cross match functions. Boehme et al. (2023) made use of probabilistic cross-identification method (Budavári & Szalay 2008) to cross-match LoLSS with other surveys at higher frequencies.

While most methods above using index partitioning and highly parallel architecture did not change the cross-matching algorithm itself, where cross-matching is considered as one by one comparison calculation. In this aspect, Bai et al. (2018) used machine learning method in cross-matching and achieved 91.9% accuracy. Shi et al. (2019) facilitated Hungarian algorithm to cross-match in crowded regions of the sky. Pineau et al. (2011) proposed 2d-tree based cross-matching method and achieve about 30 minutes in cross-matching 2MASS and USNOB1 under 24 threads. In this paper, we propose a superior approach using a 3d-tree under multi-threading, achieving comparable performance to multi-GPU methods, and show the advantages of kd-tree based cross-matching in algorithm.

2.4. Spherical Distance Calculation

The minimum distance from a query point to a circle (latitude or longitude circle) is used to prune in 2d-tree. Its easy for the latitude circle, which is the latitude difference, but more complex for the longitude circle. There are two methods to compute it.

As shown in Figure 4, to calculate the minimum distance from point A to ⊙NOB, we can first use spherical cosine theorem. Assuming point B is the intersection where $\widehat{AB}$ is the minimal angular distance from A to ⊙NOB. The ∠ANB is the longitude difference of A and B while $\angle B$ is a right angle. By calculating a system of equations as shown in Equation (3), we can obtain the angular distance of $\widehat{AB}$ as Equation (4).

$$\begin{cases} \cos\widehat{AB} = \cos\widehat{AN}\cdot\cos\widehat{BN} + \sin\widehat{AN}\cdot\sin\widehat{BN}\cdot\cos\ \angle\ \mathrm{ANB} \\ \cos\widehat{AN} = \cos\widehat{AB}\cos\widehat{BN} \end{cases} \tag{3}$$

$$\widehat{AB} = \arccos\sqrt{\cos^2\widehat{AN} + \sin^2\widehat{AN}\cos^2\ \angle\ \mathrm{ANB}} \tag{4}$$

Second, we can use the analytic geometry method. Point A is considered as a vector $\overrightarrow{OA}$ and the normal vector of ⊙NOB is calculated. The complementary angle of the angle between these two vectors is the angular distance of $\widehat{AB}$. In this case, we should transform the (α, δ) into (x, y, z) and use Equation (5).

$$\begin{aligned} x_1 &= \cos(\alpha)\cdot\sin(\delta),\ y_1 = \sin(\alpha)\cdot\sin(\delta) \\ x_2 &= \cos(\alpha_{\mathrm{border}}),\ y_2 = \sin(\alpha_{\mathrm{border}}) \\ \widehat{AB} &= \frac{\pi}{2} - \arccos(x_1\cdot x_2 + y_1\cdot y_2) \end{aligned} \tag{5}$$

3. Efficient 3d-tree Design for Large Catalogue

3.1. 3d-tree Cross-matching Algorithm

In this paper, we consider cross-matching as a process where every record in catalog B makes a query in the whole catalog A to find several spatially nearest points and build a kd-tree on catalog A to to accelerate this type of query. This way, the

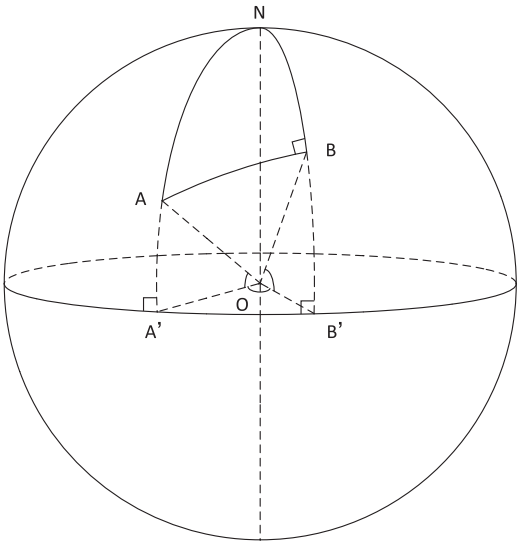

Figure 4. $\overset{\frown}{AB}$ is the minimal angular distance from query point A to a border $\odot NOB$

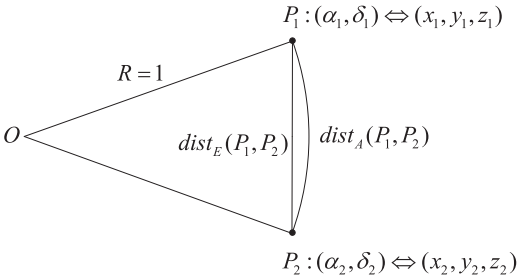

Figure 5. One angular distance in standard ball corresponded with a certain chord length.

complexity of the entire algorithm turns to building and querying the kd-tree, which will be further discussed in Section 3.2.

To use kd-tree, we should first determine k. The eigen dimensions of the sphere is two, which means we can use $k=2$ to build and query the 2d-tree. For example, we can use Equatorial coordinate, which is widely used in astronomy and previous works in cross-matching, to compute angular distance (Equations (1), (2)). Compared with 2d-tree on the plane, we demonstrated that the 3d-tree is better concerning about the feature of tree pruning and computer calculation, which is discussed in Section 3.3.

Traditional cross-matching is based on Equatorial coordinate system. To use 3d-tree in cross-matching algorithm, we should represent records in 3d Euclidean space and create an equivalent 3d cross-matching formulate. First, every point in Equatorial coordinate can be easily transformed into corresponding Euclidean coordinate using Equatorial coordinate transformation.

Second, the 3d-Euclidean distance and haversine distance are one-to-one correspondence. In geometry, the 3d-Euclidean distance between two points represents a chord length of the sphere, whose corresponding arc length is the angular distance (sky sphere is a standard ball), as shown in Figure 5. So, we can use Equation (6) to get an equivalent distance formula under 3d Euclidean coordinate.

The same transformation can be done to the threshold θ_A, with domain area of $d_A/2$ in $[0,\pi]$, where the sine function is monotonically increasing, and by using the inequality theorem. Then, we can get θ_E and the cross-matching function in 3d Euclidean coordinate like Equation (8).

$$\begin{cases} x = r\cdot\sin\left(\frac{\pi}{2}-\delta\right)\cdot\cos\alpha \\ y = r\cdot\sin\left(\frac{\pi}{2}-\delta\right)\cdot\sin\alpha \\ z = r\cdot\cos\left(\frac{\pi}{2}-\delta\right) \end{cases} \tag{6}$$

$$d_E = 2 * \sin(d_A/2) \tag{7}$$

$$d_E = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2} \leqslant \theta_E$$
$$= 2\cdot\sin\left(\frac{3\sqrt{r_1^2+r_2^2}}{2}\right) \tag{8}$$

Using the above formula, we can construct 3d-tree on catalog A and perform cross-matching on 3d-tree. To construct a 3d-tree from data in catalog A, we use the Algorithm 1. The pivot-choosing procedure we manage involves rotation options of each $k=$ three-dimension, which is suitable for the spherical points. Then we perform range-searches in the 3d-tree as described in Algorithm 2 for each record in catalog B. The results of range-searches are the cross-matching results of catalog A and B.

Algorithm 1. Building a 3d-tree for catalog A

Input: data set, for all records in catalog A, of type exemplar-set
Output: 3dt, of type 3d-tree
1 **if** data set is empty **then**
2 return the empty 3dtree
3 **end**
4 Call pivot-choosing procedure which returns two values:
5 ***ex*** ← the medium member of *data set*
6 ***split*** ← the splitting dimension
7 ***d***← domain vector of *ex*
8 ***data set'*** ← *data set* with *ex* removed
9 ***r***← range vector of *data set*
10 ***datasetLeft*** ← $\{d' \in dataset' \mid d'_{split} \leqslant d_{split}\}$
11 ***datasetRight*** ← $\{d' \in dataset' \mid d'_{split} > d_{split}\}$
12 3***dtLeft***← recursively build 3d-tree for *data setLeft*
13 3***dtRight***← recursively build 3d-tree for *data setRight*
14 3***dt*** ← <*d*, *r*, *split*, 3*dtLeft*, 3*dtRight*>

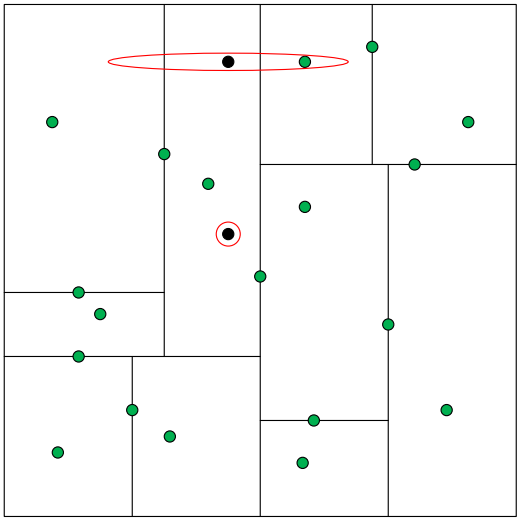

Figure 6. An example of pruning in 2d-tree cross-matching algorithm. Green points are in 2d-tree and black points are query point. Near the north pole, the query circle (red oval) span across two boarder of 2d-tree which cause inefficient prune.

Algorithm 2. Range searching in a 3d-tree for a record in catalog B

Input: 3dt, of type 3d-tree
Input: target, for a record in catalog B, of type domain vector
Input: hr, of type hyperrectangle in 3d-tree
Input: dist, for the error radius, of type double precision floating point
Output: result, for the cross-matching results, of type domain vector array
1 **if** 3dt is empty **then**
2 exit
3 **end**
4 ***s*** ← *split* field of 3*dt*
5 ***pivot*** ← *dom* − *elt* field of 3*dt*
6 **if** *distance*(***target***, ***pivot***) ⩽ ***dist*** **then**
7 ***result***.*push_back*(***pivot***)
8 **end**
9 cut ***hr*** into two sub-hyperrectangles ***left-hr*** and ***right-hr*** through ***pivot*** and perpendicular to dimensions
10 **if** *distance* (***target***,***left-hr***) ⩽ ***dist*** **then**
11 recursively call range searching with parameters:
13 (3 ***dtLeft, target, left-hr, dist***)
13 **end**
14 **if** *distance* (***target***,***right-hr***) ⩽ ***dist*** **then**
15 recursively call range searching with parameters:
16 (3 ***dtRight, target, right-hr, dist***)
17 **end**

3.2. The Benefit of Kd-tree Method than Traditional Method

Using an index partitioning based cross-matching algorithm, the best complexity is $o(N_{\text{area}})$ and the worst complexity is $O(M*N)$. For example, when two catalogs are from observations pointed to totally different sky area, the $T(n) = (N_{\text{area}}) * \text{UnitTime}$. However, when two catalogs records belong to one sky area, $T(n) = (M*N) * \text{UnitTime}$. N_{area} is a hyperparameter that needs to be determined by the user and has an uncertainty.

The average time complexity is around $\Theta(M*N/N_{\text{area}})$, but it cannot be guaranteed due to the differences in data distribution and index partitioning method. If the index partitioning method cant effectively reduce regional records, the complexity will not decrease either. As a result, the calculating time fluctuations can be high and depend on index partitioning method and the chosen N_{area} hyperparameter.

To avoid or decrease the worst situation, index partitioning based cross-matching method try to use as large an N_{area} argument as possible. However this brings up another question, boundary leakage. The larger the N_{area} argument, the higher the leakage percentage grows. To address this problem, an additional method is needed to cancel the boundary leakage problem, which is an external cost that is not cheap. When a record is confirmed to be in the boundary, it is copied into the neighboring region, and this increases cross-matching times in this area and management overhead, making it difficult to analyze and unstable. As a result, the N_{area} parameter cannot grow to big. In practice, the N_{area} parameter is chosen by experience or experimentation.

The kd-tree algorithm is not affected by the distribution of data or index partitioning method. It automatically adapts the input data distribution and divides it according to the median number of certain dimensions. The complexity of it is determined and stable. To build a kd-tree, the complexity is $\Theta(N * \log N)$, and making a range search has $O(k * N^{1-\frac{1}{k}})$ complexity. As a result, cross-matching using kd-tree method has a complexity of $O(M * k * N^{1-\frac{1}{k}})$, which is algorithmically superior than the traditional method. The average complexity of kd-tree cross-matching is related to the result of cross-matching process. If the average cross-matching number is Q, then the average complexity is $\Theta(M * Q * k * \log N)$ which is independent of index partitioning method or hyperparameters.

Table 3
Calculation Time of Different Operation Type among Three Methods

Operation Type	2d-tree		3d-tree
	Method 1	Method 2	
Trigonometric	4	4	0
Exponential	1	0	0
Multiplication	4	4	3

3.3. The Benefit of 3d-tree than 2d-tree

To perform kd-tree in the cross-matching algorithm, the instinct option is to use Equatorial coordinate with R.A. and

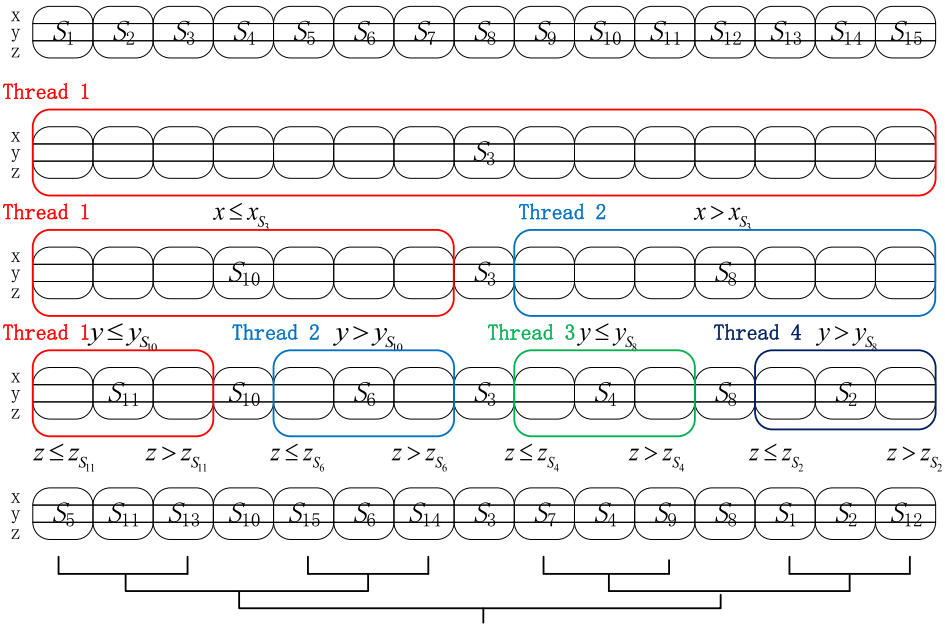

Figure 7. Multi-thread build process in 3d-tree, where each node fold two threads to find medium number of a certain dimension in sub-trees.

decl. and build a 2d-tree for search. However, it is not the best choice. There are two main reasons why we perform a conversion from the traditional cross-matching formula in Equatorial Coordinate System into a new formula under Euclidean Coordinate space, and to convert the 2d-tree into 3d-tree.

3.3.1. Pruning Efficiency

Pruning is crucial in kd-tree and determines the performance of the searching process. When using Equatorial coordinate, the cross-matching process is equivalent to a cone search on a sphere. To perform a cone search on a sphere, one can make a disk with radius r around a point P(α, δ) and thus make an Equatorial circle.

When using 2d-tree on a sphere, the sphere is stretched into a rectangle with R.A. in [0, 2π) and decl. in [$-\pi$, π]. On the Equator, the shape of the search cone is exactly a circle, but near the poles, the cone is similarly stretched into an oval like Figure 6. In this case, pruning near the poles can cause a big problem because they cannot be pruned due to the span of the oval. This significantly decreases the efficiency of pruning in kd-tree.

On the contrary, using a 3d-tree is a good alternative to avoid this issue. In a 3d-tree, the search is described as searching a ball from *P(x, y, z)* with radius *r*. This ball will not be stretch, and the pruning efficiency will remain stable everywhere, which is the guarantee of the low complexity of 3d-tree cross-matching algorithm.

3.3.2. Calculation in One Prune

Under kd-tree, every time it decides to prune into a sub-kd-tree, a target-node computation is needed to determine whether the search circle (or ball) overlaps with the sub-kd-tree range. So if the UnitTime of this target-node computation is reduced, the whole efficiency of kd-tree based cross-matching algorithm is improved. Considering the trigonometric and exponential operations are much more expensive than multiplication and addition operations in computer, the UnitTime in 3d-tree should be much less than that of 2d-tree.

The comparison of different operation time among these three methods is shown in Table 3. In 2d-tree, the distance from the target to the sub-kd-tree border is geometrically a big circle through the poles or a small circle parallel to the Equator. To achieve it, the minimum distance from the point to the circle as described in Section 2.4 is calculated. In contrast, using Euclidean method, we can simply use Equation (9) without doing trigonometric operation. The last experimental result in 4.2 is consistent with the theoretical analysis, where 3d-tree shows 41.0% speed up in one prune.

$$d = (x_1 - x_{\text{border}})^2 + (y_1 - y_{\text{border}})^2 + (z_1 - z_{\text{border}})^2 \quad (9)$$

3.4. Accelerate in Multi-thread

The construction of 3d-tree is straightforward and the key operation is to choose the median number as dividing point. For every non-leaf node, it records six boundaries of the region to

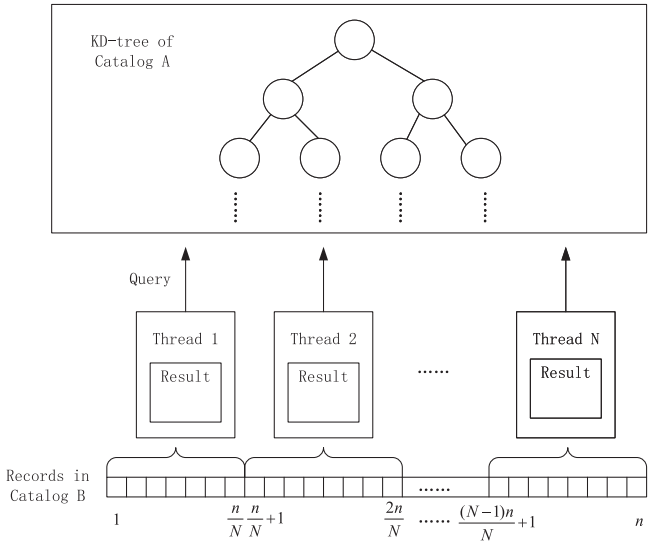

Figure 8. Multi-thread query process in 3d-tree, where each thread handles $\frac{n}{N}$ records and keeps its own results.

enable rapid pruning during searches. To parallelize it with multiple threads, we folk two threads in the upper layers of tree to handle its left-sub-tree and right-sub-tree, as shown in Figure 7.

The search operation involves searching from the root node and comparing the distance from target point to its child tree region. The distance is 0 when the point is in this region and the minimal Euclidean distance from the point to the cuboid boundary. If the distance is larger than the threshold, this child tree is cut, and the leaf node it reaches is the cross-matching success point. Because each query point has no relation with others, we can simply divide them into subsets, and each thread handle one subset, as shown in Figure 8.

3.5. Comparison with Cone Search

A cone search is used to find all records that lie within a certain radius of a given longitude/latitude. It can be considered a form of cross-matching, where catalog A serves as the reference catalog, catalog B contains only one search record, and the error radius serves as the given radius. The traditional pairwise-based cross-matching algorithm is not used in cone search process because, first its optimization methods aim to provide parallelism in cross-matching each region. It does not consider the granularity and maintenance the cross-matching results of each record. Second, when using static index partitioning methods, cross-matching can have boundary problem and boundary data redundancy method (Jia & Luo 2016) is used to avoid it. The redundancy method use the error radius as redundancy radius, and if the given radius is large, the unnecessary data redundancy in catalog A (records unrelated with the search point) will cause the redundancy procedure to be less efficient than cone search in RMDB.

In the 3d-tree cross-matching algorithm, catalog A constructs a 3d-tree data structure and each record in catalog B makes a cone search in 3d-tree with the error radius. Therefore it is actually a set of cone searches in catalog B. In this aspect, the 3d-tree cross-matching algorithm can be used as a cone search algorithm without an extra cost. The difference is that, cross-matching algorithm only maintain location and number information, while cone search requires more attribute columns to perform data analysis. Cone search mostly depends on RMDB (e.g., Koposov & Bartunov 2006) with using spatial search tree. 3d-tree is also a good choice to replace it.

Table 4
Datasets

Dataset	Description (degree)	File Size	Number of Objects
2MASS	$\alpha \in [0°, 360°]$ $\delta \in [-89.°9928, 89.°9901)$	7.02 GB	470,992,970
WISE	$\alpha \in [0°, 360°]$ $\delta \in [-89.°9946, 89.°9983)$	11.14 GB	747,634,026
SDSS	$\alpha \in [0°, 360°]$ $\delta \in [-17.°7573, 84.°9799)$	18.34 GB	1,231,051,050

4. Experiment

In this section, we first describe the experimental setup and then present our experimental results.

4.1. Experiment Setup

We conducted our experiments on a Huawei TaiShan 200 Server, which is a multi-processor and big-memory computer. It features a Kunpeng 9205220 processor with dual processors of 32-cores and 2.6 GHz processor specifications. Each core integrates 64 KB L1 ICache, 64 KB L1 DCache and 512 KB L2 Cache. L3 Cache capacity is 24MB 64 MB (1 MB/Core). Meanwhile, we equipped the server with 192 GB memory

Table 5
All Test Cases and Corresponding Computational Time are Presented (Time in seconds)

Test Case	Catalogue A	Catalogue B	3d-tree			2d-tree		
			Building Time	Cross-matching Time	Total Time	Building Time	Cross-matching Time	Total Time
T1	2MASS	2MASS	35	16	61	28	89	125
T2	WISE	WISE	107	38	169	85	315	417
T3	SDSS	SDSS	150	118	305	128	1003	1161
T4	2MASS	WISE	29	219	270	30	232	283
T5	WISE	2MASS	107	17	152	84	129	236
T6	2MASS	SDSS	31	25	88	28	107	159
T7	SDSS	2MASS	156	5	202	131	54	221
T8	WISE	SDSS	103	56	203	83	307	427
T9	SDSS	WISE	146	191	385	145	405	589

Note. Building time represents the time cost of building 3d-tree on catalog A, cross-matching time represents the time cost of range searching for all records in catalog B and total time includes all time cost of the program.

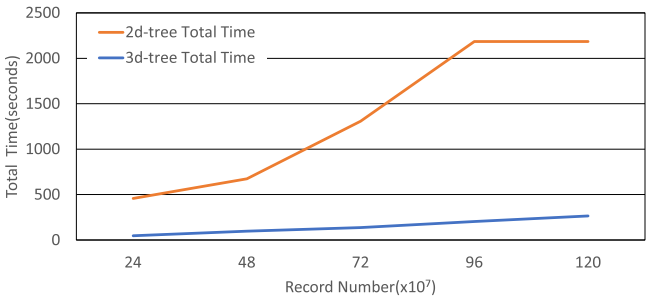

Figure 9. Total execution time increase linearly with data volume growth under 3d-tree method while the slope of 2d-tree is larger. The execution time of 2d-tree between 9.6×10^8 and 1.2×10^9 are similar because the cross-matching results here are few.

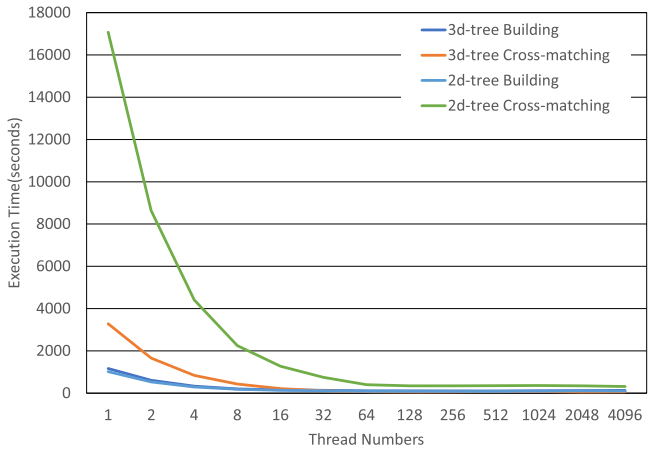

Figure 10. Execution Time of different procedures under multi-thread. Best thread number is 128 in our environment.

capability to meet the demand of 3d-tree. The operation system used was Ubuntu 22.04.1 LTS arm64 and the gcc version was 11.3.0. The -O3 optimization option was enabled in compilation.

To evaluate our implementation, we used three real-world data sets, namely Sloan Digital Sky Survey (SDSS)(DR12), Two Micron All Sky Survey (2MASS), and Wide-field Infrared Survey Explorer (WISE). All of them are point source catalogs observed by optical telescopes. The detailed description of each data set is listed in Table 4. 2MASS and WISE (WISE 2013) are two million-record catalogs. SDSS is one of the largest and most detailed astronomical surveys, with the twelfth data release (SDSSDR12 2015) containing 1.2 billion records used in our evaluation.

Table 5 lists an overview of the nine test cases designed for our evaluation. Three of them (T1–T3) are self-matching cases on the same catalog, while the remaining cases (T4–T9) are cross-matching cases on two different catalogs. In our experiments, the error radius r_1 and r_2 were set as one arc second, which is commonly used in astronomical observations.

We conducted five experiments to demonstrate the performance of 3d-tree cross-matching algorithm. We ran each experiment ten times and reported the best run. The variation among all runs was less than 10%. The results of 3d-tree cross-matching algorithm have the same precision as those of the 2d-tree and traditional methods.

4.2. Results

First, to demonstrate the progressive performance of kd-tree based cross-matching algorithm and verify its complexity, we conducted experiments on self-cross-matching execution time using different volumes of data of SDSS, as shown in Figure 9. The results show the execution time of both 3d-tree and 2d-tree based cross-matching algorithms displays a linear increasing trend with the increase in data volume in both building and cross-matching procedures.

Second, to demonstrate the scalability and find the best performance under our hardware environment, we conducted an experiment using different thread number in building and cross-matching procedures. As shown in Figure 10, the execution time decreases quickly with the increase in thread numbers, and stabilizes stable near 64–128 threads where it

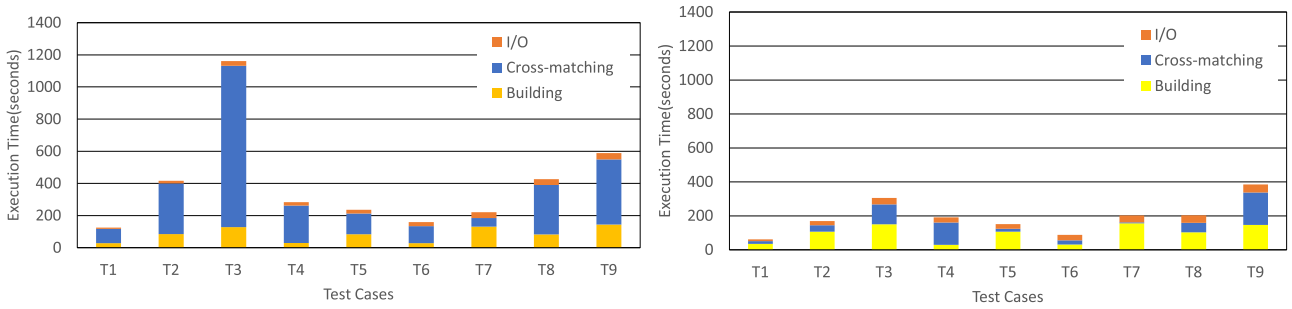

Figure 11. Time cost in three procedure of 2d-tree(left) and 3d-tree(right).

Table 6
Performance Comparison with other Methods

Cross-matcher	Record Number	Time Cost
Zhao et al. (2009)	470,992,970 × 100,106,811	32min
Du et al. (2014)	470,992,970 × 100,106,811	23min
Soumagnac & Ofek (2018)	470,992,608 × 563,908,224	53 min
Pineau et al. (2011)	470,992,970 × 1,042,618,261	30 min
Riccio et al. (2019)	1,000,000 × 10,000,000	800 s
Zhang et al. (2023)	467,555,800 × 102,890,000	260 s
Zečević et al. (2019)	1,692,919,135 × 747,634,026	226 s
Jia & Luo (2016)	747,634,026 × 1,231,051,050	229s
3d-tree cross-matching	747,634,026 × 1,231,051,050	203s

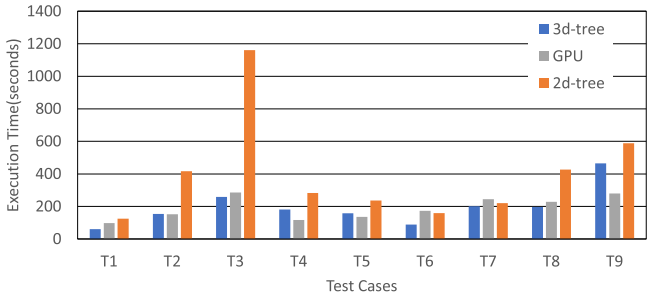

Figure 12. Multi-thread build process in 3d-tree, where each node fold two threads to find medium number of a certain dimension in sub-trees.

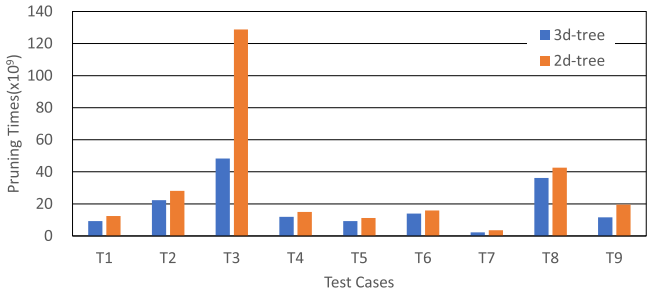

Figure 13. Multi-thread build process in 3d-tree, where each node fold two threads to find medium number of a certain dimension in sub-trees.

achieves the best scalability of our environment (64 cores in total). The building procedure achieves 12× speed-up and the cross-matching procedure achieves 60× speed up compared to that of single thread.

Third, Figure 11 shows the examined time cost in I/O, building and cross-matching procedures. The I/O time of both methods depends on data volume and number of cross-matching results, and takes 18.9% in 3d-tree on average. When using 2d-tree, the main expenses occur in cross-matching which takes 54.7%–86.4% of the total execution time (except for case 7 where catalog B is an order of magnitude smaller than catalog A). While in 3d-tree, it only takes 30.6% on average. In the building procedure, 2d-tree performs better than 3d-tree because of fewer dom-elt and border contents. Because of this feature, using smaller catalog in building and larger catalog in range searching is sufficient in 3d-tree. Notice that the cross-matching time of test cases involving WISE data set is longer than others, because the object density of WISE is larger than other data sets, which requires more time to prune in 3d-tree.

Fourth, we compared the performance of 2d-tree and 3d-tree methods with other cross-matching methods as shown in Table 6. It shows great speed-ups compared to the former six cross-matches and has no boundary problem. The last two cross-matches perform on high-performance computing architecture. Zečević et al. implements cross-matching on CPU cluster of 28 nodes, and Jia & Luo implements it on Multi-GPU cluster with 12 GPUs used. The 3d-tree cross-matching algorithm has a comparable performance with the last two cross-matches using only multi-core CPU. Meanwhile, we compared 2d-tree and 3d-tree methods with the Multi-GPU method (Jia & Luo 2016) in Figure 12 and selected the same data sets and test cases. It shows that 3d-tree method obtain approximate performance than GPU method within lower equipment requirements and even better in five test cases, demonstrating the algorithmic superiority of 3d-tree. The kd-tree algorithm also can be used in GPU and distributed architecture to further increase the calculation efficiency.

Finally, we have identified the main reason of performance gap between 2d-tree and 3d-tree methods in Figure 13. The pruning time of 2d-tree is 48.0% more than that of 3d-tree and the computational time in one prune is more than 41.0% on average.

This finding confirms the analysis of the two weaknesses of the 2d-tree cross-matching algorithm discussed in Section 3.3.

5. Conclusions

In this paper, we presented a cross-matching algorithm that is based on 3d-tree, which we developed by converting the algorithm into a kd-tree problem. This approach achieved good performance in many data sets and is algorithmically superior to traditional cross-matching methods. Additionally, it demonstrates great scalability in a multi-thread environment and does not have a source leakage problem because we did not use any data division method. We also compared the performance of the 3d-tree approach with the 2d-tree method and measured the performance gap between them.

The experiments were conducted on a single computer with 64 cores. To further accelerate this method using better equipment, data division method can be adapt to achieve less building tree time and better concurrency. Additionally, many high-performance computing architectures (e.g., Garcia et al. 2008; Patwary et al. 2016) can support kd-tree well. However, one problem with the 3d-tree method is the huge memory cost during building. So finding a less memory-intensive algorithm could further improve the performance of cross-matching.

Acknowledgments

This work is supported by the National Key Research and Development Program of China (2022YFF0711502), the National Natural Science Foundation of China (NSFC) (12273025 and 12133010). Data resources are supported by China National Astronomical Data Center (NADC), CAS Astronomical Data Center and Chinese Virtual Observatory (China-VO).

ORCID iDs

Yifei Mu https://orcid.org/0009-0008-7599-566X

References

2MASS 2006, 2MASS Data Access, https://irsa.ipac.caltech.edu/Missions/2mass.html
Bai, Y., Liu, J.-F., & Wang, S. 2018, RAA, 18, 118
Boehme, L., Schwarz, D. J., de Gasperin, F., Roettgering, H. J. A., & Williams, W. L. 2023, A&A, 674, A189
Budavari, T., & Lee, M. A. 2013, Xmatch: GPU Enhanced Astronomic Catalog Cross-Matching, Astrophysics Source Code Library, ascl:1303.021
Budavári, T., & Szalay, A. S. 2008, ApJ, 679, 301
Du, P., Ren, J., Pan, J., & Luo, A. 2014, SCPMA, 57, 577
Fan, D., Budavári, T., Szalay, A. S., Cui, C., & Zhao, Y. 2013, PASP, 125, 218
GAIA 2018, GAIA Data Release 2, https://cosmos.esa.int/web/gaia/dr2
Garcia, V., Debreuve, E., & Barlaud, M. 2008, in 2008 IEEE Comp. Soc. Conf. Comput. Vis. Pattern Recognition Workshops (Piscataway, NJ: IEEE), 1
Gorski, K. M., Hivon, E., Banday, A. J., et al. 2005, ApJ, 622, 759
Gray, J., Nieto-Santisteban, M. A., & Szalay, A. S. 2007, CoRR, abs/cs/0701171
Han, B., Zhang, Y.-X., Zhong, S.-B., & Zhao, Y.-H. 2016, RAA, 16, 178
Jia, X., & Luo, Q. 2016, in Proc. 28th Int. Conf. Sci. and Statistical Database Management (New York: ACM), 1
Jia, X., Luo, Q., & Fan, D. 2015, in 2015 IEEE 21st Int. Conf. Parallel and Distributed Syst. (ICPADS) (Piscataway, NJ: IEEE), 617
Koposov, S., & Bartunov, O. 2006, adass, 351, 735
Kumar, V. S., Kurc, T., Saltz, J., et al. 2009, in 2009 IEEE Int. Symp. Parallel & Distributed Processing (Piscataway, NJ: IEEE), 1
Lee, M., & Budavári, T. 2013, adass XXII, 475, 235
Li, B., Yu, C., Li, C., et al. 2019, PASP, 131, 054501
Moore, A. 1991, An introductory tutorial on kd-treesTechnical Report No. 209, Computer Laboratory, University of Cambridge
Nieto-Santisteban, M., Thakar, A., Szalay, A., & Gray, J. 2006, adass, 351, 493
Nieto-Santisteban, M. A., Thakar, A. R., & Szalay, A. S. 2007, in The 2007 NASA Science Technology Conference (NSTC2007) (Baltimore, MA: Johns Hopkins University)
Patwary, M. M. A., Satish, N. R., Sundaram, N., et al. 2016, in 2016 IEEE Int. Parallel Distributed Proc. Symp. (IPDPS) (Piscataway, NJ: IEEE), 494
Pineau, F.-X., Boch, T., & Derriere, S. 2011, adass, 442, 85
Riccio, G., Brescia, M., Cavuoti, S., et al. 2016, in IAU Symp. 325, 12 (Cambridge: Cambridge Univ. Press), 327
SDSS 2022, SDSS Data Release, https://sdss4.org
SDSSDR12 2015, SDSS Data Release, Data Access for SDSS DR12 Overview, https://sdss4.org/dr12/data_access/
Shi, X., Budavari, T., & Basu, A. 2019, ApJ, 870, 51
Soumagnac, M. T., & Ofek, E. O. 2018, PASP, 130, 075002
Szalay, A. S., Fekete, G., O'Mullane, W., et al. 2004, arXiv:cs/0408031
Szalay, A. S., Gray, J., Fekete, G., et al. 2007, arXiv:cs/0701164
Wan, S., Zhao, Y., Wang, T., et al. 2019, Fut. Gen. Comput. Syst., 91, 382
Wang, S., Zhao, Y., Luo, Q., Wu, C., & Xv, Y. 2013, in IEEE 9th Int. Conf. e-Sci. (Piscataway, NJ: IEEE), 326
WISE 2013, WISE Data Access, https://irsa.ipac.caltech.edu/Missions/wise.html
Yu, C., Li, K., Tang, S., et al. 2020, MNRAS, 496, 629
Zečević, P., Slater, C. T., Jurić, M., et al. 2019, AJ, 158, 37
Zhang, Y., Yu, C., Sun, C., et al. 2023, MNRAS, 519, 6381
Zhao, Q., Sun, J., Yu, C., et al. 2009, in ICA3PP 2009 (Berlin: Springer), 604
Zhou, K., Hou, Q., Wang, R., & Guo, B. 2008, ACM Trans. Graph., 27, 126