A&A 672, A83 (2023)
https://doi.org/10.1051/0004-6361/202346015
© The Authors 2023

Astronomy & Astrophysics

Fast correlation function calculator

A high-performance pair-counting toolkit*

Cheng Zhao (赵成)

Institute of Physics, Laboratory of Astrophysics, École Polytechnique Fédérale de Lausanne (EPFL), Observatoire de Sauverny, 1290 Versoix, Switzerland
e-mail: cheng.zhao@epfl.ch

Received 29 January 2023 / Accepted 24 February 2023

ABSTRACT

Context. A novel high-performance exact pair-counting toolkit called fast correlation function calculator (FCFC) is presented.

Aims. With the rapid growth of modern cosmological datasets, the evaluation of correlation functions with observational and simulation catalogues has become a challenge. High-efficiency pair-counting codes are thus in great demand.

Methods. We introduce different data structures and algorithms that can be used for pair-counting problems, and perform comprehensive benchmarks to identify the most efficient algorithms for real-world cosmological applications. We then describe the three levels of parallelisms used by FCFC, SIMD, OpenMP, and MPI, and run extensive tests to investigate the scalabilities. Finally, we compare the efficiency of FCFC with alternative pair-counting codes.

Results. The data structures and histogram update algorithms implemented in FCFC are shown to outperform alternative methods. FCFC does not benefit greatly from SIMD because the bottleneck of our histogram update algorithm is mainly cache latency. Nevertheless, the efficiency of FCFC scales well with the numbers of OpenMP threads and MPI processes, even though speedups may be degraded with over a few thousand threads in total. FCFC is found to be faster than most (if not all) other public pair-counting codes for modern cosmological pair-counting applications.

Key words. methods: data analysis – methods: numerical – techniques: miscellaneous – large-scale structure of Universe

1. Introduction

Correlation functions are a useful statistical tool in cosmology that characterise the excess probability of finding tracers with given separations compared to a random distribution. Thus, they are a measure of the clustering pattern of a tracer distribution, which can then be used to infer the statistical quantities of the underlying density field. In the current standard cosmological paradigm, the distribution of matter results from tiny fluctuations in the primordial Universe, which evolve following gravitational instability and cosmic expansion. For this reason, correlation functions are crucial for our understanding of inflation and cosmic structure formation models (e.g. Bernardeau et al. 2002). The pair correlation function, also known as the radial distribution function, which is essentially the isotropic two-point correlation function (2PCF), is also a fundamental quantity in statistical mechanics, where it links microscopic details to macroscopic properties (Chandler 1987).

The measurement of 2PCFs of galaxies and quasars has been a key goal of massive spectroscopic surveys, such as the Baryon Oscillation Spectroscopic Survey (BOSS; Dawson et al. 2013), the Extended Baryon Oscillation Spectroscopic Survey (eBOSS; Dawson et al. 2016), and the ongoing Dark Energy Spectroscopic Instrument (DESI; DESI Collaboration 2016). With a data catalogue and the corresponding random sample, the 2PCF is generally measured using the Landy–Szalay (LS) estimator (Landy & Szalay 1993),

$$\xi = (\mathrm{DD} - 2\mathrm{DR} + \mathrm{RR})/\mathrm{RR}, \tag{1}$$

where DD, DR, and RR denote data–data, data–random, and random–random pair counts, respectively. Observational and simulated galaxy samples today usually consist of several million or more than several million galaxies. Robust clustering measurements further require random samples with typically ten times the objects. As a result, the brute-force pair-counting approach that evaluates N^2 pair separations, where N is the number of data points, is impractical. The calculation of 2PCFs from pair counts has become a practical challenge for modern cosmological analysis, not to for mention higher-order statistics like three-point correlation functions.

The evaluation of correlation functions is effectively a range-searching problem, which reports objects within a query range. Range searching is a fundamental topic in computational geometry. A variety of data structures and algorithms is available to solve range-searching problems with different objects and query ranges (see e.g. de Berg et al. 2008). In the context of cosmology, efficient correlation function calculators have also been studied extensively in the literature, from the pioneering work by Moore et al. (2001) to the recent remarkable development of Sinha & Garrison (2020). Meanwhile, significant efforts have been made to make full use of high-performance computing (HPC) resources, such as a large number of multi-core CPUs and GPUs (e.g. Dolence & Brunner 2008; Alonso 2012; Chhugani et al. 2012; Ponce et al. 2012). Different approximate methods

* The fast correlation function calculator is publicly available at https://github.com/cheng-zhao/FCFC

Open Access article, published by EDP Sciences, under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.
This article is published in open access under the Subscribe to Open model. Subscribe to A&A to support open access publication.

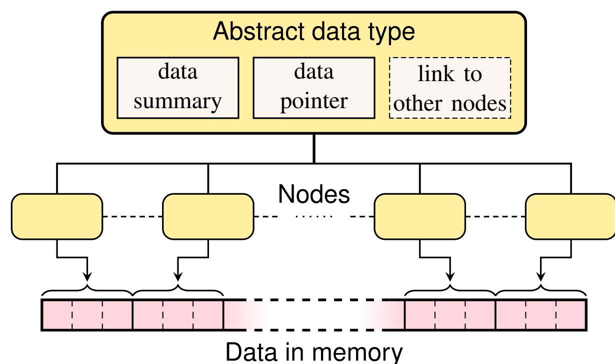

Fig. 1. Architecture of the data structures implemented in this work.

are explored widely as well (e.g. Zhang & Pen 2005; Slepian & Eisenstein 2015; Philcox et al. 2022).

Even though very many pair-counting codes are publicly available (e.g. Alonso 2012; Jarvis 2015; Rohin 2018; Donoso 2019; Sinha & Garrison 2020, and references therein), we introduce the fast correlation function calculator[1] (`FCFC`), a novel high-efficiency, scalable, portable, flexible, and user-friendly toolkit for exact pair counting. We focus on `FCFC` version 1.0.1 in this article, which supports 2PCFs for 3D data, with various commonly used binning schemes. It may be the fastest publicly available 2PCF calculator for modern cosmological datasets so far.

This paper is organised as follows. We begin with a comparison of different data structures for pair-counting problems in Sect. 2. Then, we introduce the pair-counting and histogram update algorithms used by `FCFC` in Sect. 3. In Sect. 4 we describe the performance of `FCFC` with different levels of parallelisms. A direct comparison between `FCFC` and `Corrfunc`, another efficient cosmological pair-counting code, is presented in Sect. 5. Finally, we conclude in Sect. 6.

2. Data structures

Data structures are a technique for organising and storing an input dataset in memory that allows efficient data access. Typically, it is not only unnecessary, but also inefficient to process the data all at once in a program. A well-designed data structure may prevent the retrieval of irrelevant data during data queries as much as possible; this is known as data pruning. Thus, data structures are usually crucial for efficient algorithms. To this end, several types of data structures for pair-counting applications have been proposed in the literature, including regular grids (Alonso 2012; Sinha & Garrison 2020), linked list (Donoso 2019), the k-d tree (Moore et al. 2001), and ball tree (Rohin 2018).

In general, a data structure sorts and partitions the dataset, and stores the data segments on different nodes, either by copying the data directly or by saving only the addresses in memory. Each node is typically defined as an abstract data type, which contains summaries of the associate data, but this is sometimes done implicitly, to quickly judge whether the data should be retrieved. Connections between different nodes may also be built to optimise the node traversal process. This architecture is illustrated in Fig. 1. Because the datasets for cosmological applications are large, we save only data pointers on the nodes, to make the latter more compact and reduce the chance of cache misses during node traversal. The raw data, which are accessed less often, are stored separately and continuously in the memory. In particular, during the construction of the data structures, the data are sorted in such a way that the points belonging to adjacent nodes are aligned continuously.

For pair-counting applications, it is crucial to be able to compute the separation ranges between nodes efficiently to omit nodes that are too far away or too close to each other, without visiting individual data points. For this purpose, we describe a few data structures in this section, including regular grids, the k-d tree, and a new variant of the ball tree, and compare their performance in terms of pair counting. Throughout this section, the costs of structure constructions are not counted for our benchmarks, as they generally take <1 % of the time used by the pair-counting processes. Moreover, throughout this section, we do not consider multiple histogram bins. This is because with the same histogram update algorithm, almost identical costs are added to the pair-counting process with different data structures, which makes the comparisons of the data-pruning efficiency less effective. The computational costs are all measured with a single Haswell CPU core (see Appendix A).

2.1. Regular grids

A simple way to partition a dataset is to divide the domain into regular axis-aligned grids, with unique identifiers for spatial indexing. Normally, the positions and extents of the grid cells can be expressed by polynomials of the identifiers, or indices. Therefore, distance ranges between different grid cells can be inferred from the differences of cell indices, which can be computed prior to the cell traversal process. This makes regular grids a potentially very efficient data structure for pair counting.

With the architecture shown in Fig. 1, only three passes through the dataset are required to construct regular grids for a catalogue: (1) find the minimum axis-aligned bounding box (AABB) of the catalogue to define grids (2) count the number of data points in each grid cell (3) group the data points based on the associate cell indices. Therefore, the construction of regular grids can be very efficient, with a time complexity of $O(N)$, where N denotes the total number of data points. In contrast, the storage consumed by regular grids is very sensitive to the number of grid cells and scales as $O(\prod_i N_{g,i})$, where $N_{g,i}$ indicates the number of cells along the ith dimension.

The efficiency of data pruning for regular grids largely depends on the cell sizes as well. An example is shown in Fig. 2, where the data partitions with two different cell sizes are illustrated, even though the choices of the cell sizes are likely unwise for real-world applications. Given the same reference point and maximum distance for an isotropic range search, the numbers of visited cells and data points are both significantly different when the number of cells per box side is varied. Here, data points belonging to different grid cells are arranged in column-major order, and gaps between adjacent memory visits are observed. Sorting the cells using a space-filling curve, such as the Hilbert curve, may improve the memory locality and reduce the chance of cache misses (see e.g. Springel 2005, for an application). Nevertheless, the improvement is expected to be marginal, as the memory jumps can never be entirely eliminated, and it is more difficult but possible to pre-compute the map from indices of grid cells to the distance ranges between cells.

The algorithm for pair counting with regular grids is as simple as traversing all grid cells that contain data points and successively visting cells that are separated within the distance range of interest, given the pre-computed offsets of indices. This

[1] https://github.com/cheng-zhao/FCFC

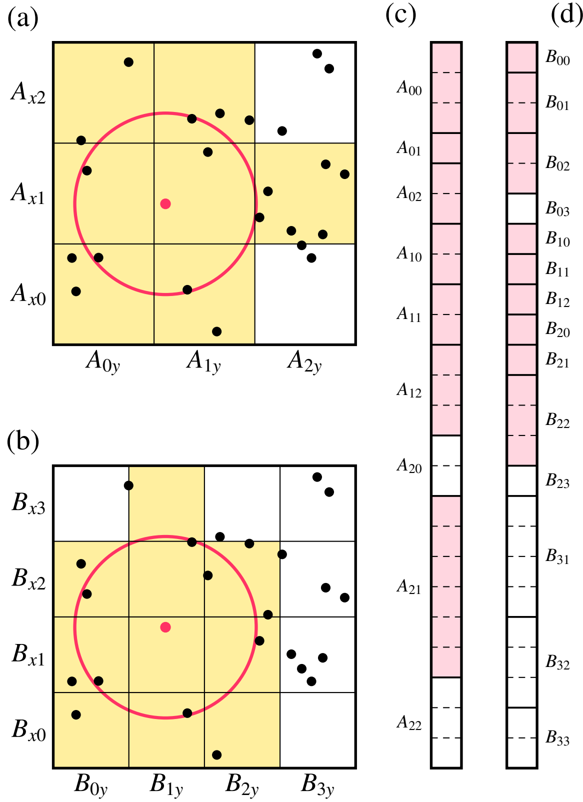

Fig. 2. Illustration of an isotropic range search using regular grids with different cell sizes. The points in panels a and b indicate a randomly generated dataset in 2D, with the current reference point marked in red. Yellow areas denote cells that are visited, given the query range indicated by red circles. Panels c and d show the arrangements of data points in memory for the column-major grid configurations in panels a and b, respectively. Pink regions indicate data points that are visited during the range search process. The cell sizes are chosen for illustration purposes and are not meant to be optimal.

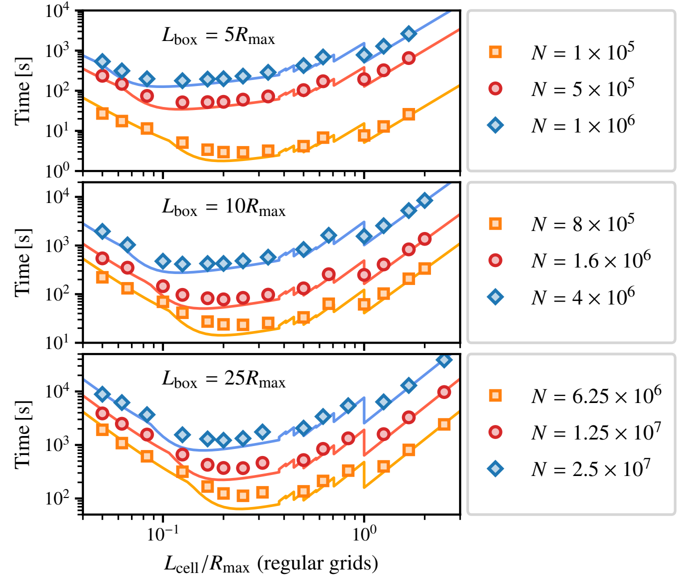

Fig. 3. Execution time of the grid-based pair-counting routine with different cell sizes and query ranges for periodic uniform random samples with different cubic box sizes and numbers of points. The solid lines show the best-fitting theoretical results detailed in Appendix C.

procedure is detailed in Appendix B. Consequently, the complexity of the algorithm depends not only on the number of grid cells that intersect with the query range, but also on the average number of data points in each cell. Apparently, when the side lengths of regular grid cells are increased, the number of cells to be visited is reduced, but there may be more unnecessary distance evaluations for pairs, as illustrated by Fig. 2. Thus, the choice of cell sizes is crucial for the efficiency of a grid-based pair-counting algorithm (see also Sinha & Garrison 2020, for relevant discussions).

We then performed a series of benchmarks with the pair-counting routine based on regular grids, which reports the number of pairs with separations smaller than R_{max}, and omits histogram bins of distances to separate the impacts of the data structure and histogram update algorithm (see Sect. 3.2 for details). For simplicity, the pair-counting procedure is based on cubic grid cells with a side length of L_{cell}, and run on N uniformly distributed random points in a periodic cubic volume with the box size of L_{box}. The execution time of the pair-counting processes with different settings are shown in Fig. 3, together with the theoretical model detailed in Appendix C. L_{box} and L_{cell} are expressed as factors of R_{max}, as the benchmark results are irrelevant to the units of lengths. The results confirm the sensitivity of computational costs to cell sizes. For the configurations we explored, the optimal L_{cell} is typically 0.1 to 0.5 times R_{max}.

2.2. k-d tree

A k-d tree (Bentley 1975) is a binary space-partition data structure that is commonly used for range-searching and nearest-neighbour algorithms. It partitions the k-dimensional space recursively with axis-aligned planes. Depending on the choice of the splitting planes, there are several variants of the k-tree structure. In this work, we chose the ‘optimised’ k-d tree introduced by Friedman et al. (1977), for which the space-partition planes are perpendicular to the dimension with the largest data variance, and split the dataset into two parts at the median point. Therefore, this variant always produces a balanced tree structure and is particularly useful for observational catalogues with an arbitrary survey geometry.

Algorithm 1 shows the procedure we followed to construct the k-d tree for pair-counting purposes. The root node of the tree is associated with all the data points. For each non-leaf node, the two subsets of data after space partition are assigned to their two children, respectively. In addition, we stored the minimum AABB of points on each node for efficient data pruning because it is easy to evaluate the minimum and maximum distances between AABBs, which can be good estimates of the separation ranges of points on different nodes. Finally, the tree construction process was terminated when all leaf nodes contained at most n_{leaf} points.

Since the k-d tree is always balanced, there are in total $O(N)$ tree nodes for a fixed n_{leaf}. The storage cost of the tree is then $O(N)$. Computations of the minimum AABB and coordinate variances require only two passes through the dataset. Additionally, we split the data for the child nodes using the linear-time adaptive QUICKSELECT algorithm (`MedianOfNinthers`; Alexandrescu 2017). Therefore, the k-d tree construction can be accomplished in $O(N \log N)$ time, given the tree depth of $O(\log N)$.

Figure 4 shows the k-d tree constructed with $n_{leaf} = 3$ for the same sample points as in Fig. 2. The space partition is adaptive, and in this particular example, the tree is complete, with the same number of points on all leaf nodes. In addition, the

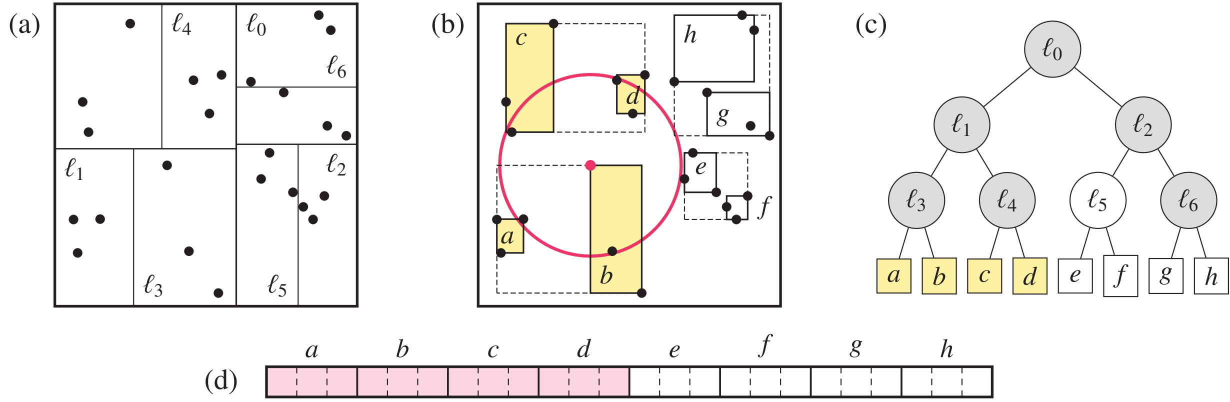

Fig. 4. Illustration of an isotropic range search using the k-d tree. Panels a, b, and c show the partitions of the data points (black dots) during the k-d tree construction, the resulting minimum axis-aligned bounding box of leaf nodes and their parents, and the diagram of the tree structure, respectively. In particular, non-leaf nodes in panel (c) are indicated by the corresponding dividing lines in panel a. The red point and circle in panel b indicate the reference point and radius for a range search. The grey regions in panel c and the yellow areas in panels b and c highlight the visited non-leaf and leaf nodes during this query. The retrieved data points are shown in pink in panel d.

Algorithm 1 KdTree_Build ($\mathcal{P}$, $n_{\rm leaf}$)

Input: a point set $\mathcal{P}$ and the capacity of leaf nodes.
Output: the root of a k-d tree for $\mathcal{P}$.
1: Create a new node v, with v.data ← $\mathcal{P}$.
2: v.bound ← MinimumAABB ($\mathcal{P}$) ▹ bounding volume of v
3: **if** cardinality($\mathcal{P}$) $\leq n_{\rm leaf}$ **then**
4: **return** v as a leaf node
5: **else**
6: Find the axis direction with the largest coordinate variance for all points in $\mathcal{P}$, and divide $\mathcal{P}$ into $\mathcal{P}_1$ and $\mathcal{P}_2$ with a splitting plane perpendicular to this direction, such that $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$, $\mathcal{P}_1 \cap \mathcal{P}_2 = \varnothing$, and cardinality($\mathcal{P}_1$) = $\lfloor$cardinality($\mathcal{P}$)/2$\rfloor$.
7: v.left ← KdTree_Build ($\mathcal{P}_1$, $n_{\rm leaf}$) ▹ left child of v
8: v.right ← KdTree_Build ($\mathcal{P}_2$, $n_{\rm leaf}$) ▹ right child of v
9: **return** v
10: **end if**

total AABB volume of nodes with the same depth can be significantly smaller than the volume of the full dataset, especially for leaf nodes, due to the gaps between the bounding boxes of different nodes. This implies a relatively high data-pruning efficiency, as it is easier to detect data groups that are too far away or too close to each other than in the grid-based method. For the example shown in Fig. 4, only four leaf nodes were visited after checking the distances between AABBs. Furthermore, since the leaf nodes visited are in the same branch of the tree, the associated data points are continuously aligned in memory, which indicates a high memory-access efficiency.

We used the dual-tree algorithm (see Sect. 3.1) to count pairs with the k-d tree, which traverses the tree nodes in a top-down manner. In brief, we skipped all the descendants of two nodes when the separation range between the minimum AABBs of these nodes was entirely inside or outside the query range for pair counting. In other words, a leaf node was only visited when the corresponding AABB intersected with the query range boundary of its counterpart during the tree traversal process, which usually is a leaf node as well. Consequently, the sizes of nodes from which the data points are retrieved are adaptive, and the number of visited nodes is greatly reduced compared to the grid-based approach, especially when the query range is large.

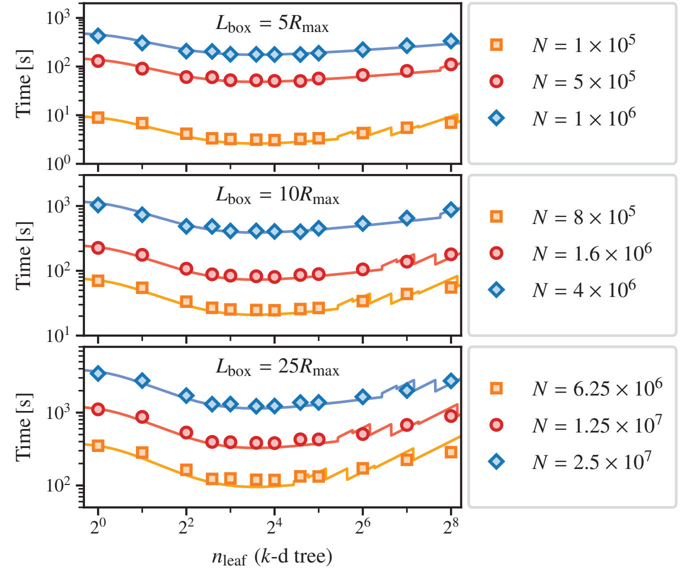

Fig. 5. Execution time of the pair-counting routine based on the k-d tree with different capacities of leaf nodes for periodic uniform random samples with different cubic box sizes and numbers of points. The solid lines show the best-fitting theoretical results detailed in Appendix C.

We then ran the pair-counting routine based on the k-d tree upon the same catalogues as were used for benchmarks of the grid-based method. Again, we considered a single histogram bin for separations smaller than $R_{\rm max}$. The results with different choices of leaf node capacity are shown in Fig. 5. The execution time of the pair-counting algorithm based on the k-d tree does not vary significantly as $n_{\rm leaf}$, especially when $4 \leq n_{\rm leaf} \leq 64$, compared to the strong cell-size dependence of the grid-based method (see Fig. 3; we provide more detail in Appendix C). Moreover, the optimal $n_{\rm leaf}$ is found to be 8 for almost all configurations presented in Fig. 5. This makes the k-d tree structure particularly useful in practice, as it is not necessary to explore different choices of $n_{\rm leaf}$ to maximise the pair-counting efficiency for different input samples[2].

[2] The optimal $n_{\rm leaf}$ may change if the costs of distance evaluations and node visits vary disproportionally; see Sect. 4.1.

2.3. Ball tree

Similar to the k-d tree, the ball tree (Omohundro 1989) is also a binary space partition tree that is useful for range queries, especially for high dimensions. In general, every node of a ball tree defines a hypersphere that contains all the points on the node. This makes it slightly easier to compute the minimum and maximum distances between two nodes than in the case of axis-aligned boxes for the k-d tree. However, for traditional ball-tree implementations (e.g. Moore 2000), the tree is not necessarily balanced, and the hyperspheres, or balls, can be significantly larger than the minimum bounding spheres of the points. As a result, both the dual-tree algorithm (see Sect. 3.1) and the data-pruning process are sub-optimal for pair counting (see the application in Rohin 2018, however). We then introduce a new variant of the ball–tree structure to circumvent these problems.

To construct a balanced ball tree, one way is to use the space partition scheme of the k-d tree. In this case, all subsets of data points are bounded by axis-aligned boxes, and data pruning with minimum bounding spheres is supposed to be less efficient than with the k-d tree because their volumes are generally larger than the corresponding minimum AABBs. Moreover, axis-aligned partition schemes may be sub-optimal for observational data with complicated shapes. To circumvent these problems, we followed the space partition approach introduced by Dolatshah et al. (2015), which defines the splitting plane based on a principal component analysis (PCA). In particular, the plane was chosen to be perpendicular to the most significant principal component of the data distribution, which is the direction with the largest variance of the data points. Thus, the resulting data subsets are statistically the least extended. In this way, the minimum bounding spheres of the ball-tree nodes are generally small enough in practice for efficient data pruning.

The next step was to compute the minimum bounding spheres of the subdivided datasets. In principle, the exact solution can be obtained in linear time using a randomised algorithm (Welzl 1991; Gärtner 1999). However, it is relatively slow for a large dataset. We then focused on the approximate algorithm introduced by Ritter (1990), which ensures that all the input data points are enclosed by the reported sphere, but it typically overestimates the radius by $\lesssim$20 % (e.g. Larsson 2008). This algorithm sets up an initial sphere with three points that are far away from each other and then goes through the rest of the data points. Whenever a point is found outside the sphere, a new sphere that encloses both the point and the previous sphere is constructed. Following Larsson (2008), we improved this algorithm by constructing a better initial sphere, which is defined by the extreme points along the directions of the first two principal components. In practice, the minimum bounding sphere of the four extreme points was computed exactly, and this sphere was updated the same way as in Ritter (1990).

The full procedure for the construction of our ball-tree variant is shown in Algorithm 2, which is very similar to that of the k-d tree (see Algorithm 1), and consumes $O(N)$ space as well since the tree is balanced. In practice, we relied on the symmetric QR algorithm (e.g. Golub & Van Loan 2013) for the 3D PCA. When the number of data points is large, the computing time for PCA is dominated by the covariance matrix evaluation, which requires two passes through the dataset. The update of the minimum bounding sphere needs another pass. Again, we used the adaptive QuickSelect algorithm for the data partition, but with a comparison rule that involves the first principal component of the dataset. Therefore, the time complexity for constructing a single ball-tree node is $O(N)$, and is $O(N \log N)$ for the whole tree. In practice, the ball-tree construction process is typically only marginally slower than that of the k-d tree.

Algorithm 2 BallTree_Build ($\mathcal{P}$, $n_{\rm leaf}$)

Input: a point set $\mathcal{P}$ and the capacity of leaf nodes.
Output: the root of a ball tree for $\mathcal{P}$.
1: Create a new node v, with v.data $\leftarrow \mathcal{P}$.
2: Compute $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, the first two principal components of $\mathcal{P}$.
3: $\mathcal{E} \leftarrow$ FindExtremePoints ($\mathcal{P}$, $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$)
4: $B \leftarrow$ MinimumBoundingSphere ($\mathcal{E}$)
5: **for all** $\boldsymbol{p} \in \mathcal{P} \setminus \mathcal{E}$ **do**
6: **if** $\boldsymbol{p}$ outside B **then** $B \leftarrow$ GrowSphere (B, $\boldsymbol{p}$) **end if**
7: **end for**
8: v.bound $\leftarrow B$ ▷ bounding volume of v
9: **if** cardinality($\mathcal{P}$) $\leq n_{\rm leaf}$ **then**
10: **return** v as a leaf node
11: **else**
12: Divide $\mathcal{P}$ into subsets $\mathcal{P}_1$ and $\mathcal{P}_2$, such that $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$, $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, $\max_{(\boldsymbol{p}_1 \in \mathcal{P}_1)}(\boldsymbol{p}_1 \cdot \boldsymbol{u}_1) \leq \min_{(\boldsymbol{p}_2 \in \mathcal{P}_2)}(\boldsymbol{p}_2 \cdot \boldsymbol{u}_1)$, and cardinality($\mathcal{P}_1$) = $\lfloor$cardinality($\mathcal{P}$)/2$\rfloor$.
13: v.left $\leftarrow$ BallTree_Build ($\mathcal{P}_1$, $n_{\rm leaf}$) ▷ left child of v
14: v.right $\leftarrow$ BallTree_Build ($\mathcal{P}_2$, $n_{\rm leaf}$) ▷ right child of v
15: **return** v
16: **end if**

An example of the ball tree constructed with the points as in Fig. 2 is shown in Fig. 6. The space partition lines are not axis-aligned in general, resulting in different data point groups than those of the axis-aligned partition scheme (see Fig. 4). Different nodes with the same depth do not share data points, but their bounding spheres may overlap. The bounding sphere of a node may not be fully inside that of its parent. This does not necessarily mean that the data-pruning efficiency is low, as the distances between different nodes are always examined in the top-down order. For the example in Fig. 6, the range-searching process involves one more leaf node than that of the k-d tree, but the visited data points are still stored continuously, indicating a good memory locality.

Similar to the case of the k-d tree, the tree-independent dual-tree algorithm (see Sect. 3.1) was used to count pairs with the ball tree. The benchmark results with the ball tree are shown in Fig. 7, with a single histogram bin for separations below $R_{\rm max}$. The dependences of the execution time measurements on $n_{\rm leaf}$ are similar to those of the k-d tree. The theoretical model was derived for the k-d tree (see Appendix C), but it also works well for the ball tree. This can be explained by the fact that the spatial partition schemes are similar for these two data structures for a periodic box. Again, the results are not sensitive to the choice of $n_{\rm leaf}$, and a leaf node capacity of 8 is found to be optimal for almost all cases.

2.4. Comparison of the data structures

In order to identify the optimal data structure among those discussed so far for real-world pair-counting problems, we performed two additional sets of benchmarks with both periodic and non-periodic datasets. For tests with periodic boundary conditions, which is the case for cosmological simulations, we generated uniformly distributed random points in a cubic volume with a box size of $L_{\rm box}$. Then, to mimic the geometry of the observational data in redshift bins, we cut the cubic catalogues at $R_{\rm out} = L_{\rm box}$ and $R_{\rm in} = L_{\rm box}/2$ with respect to a corner of

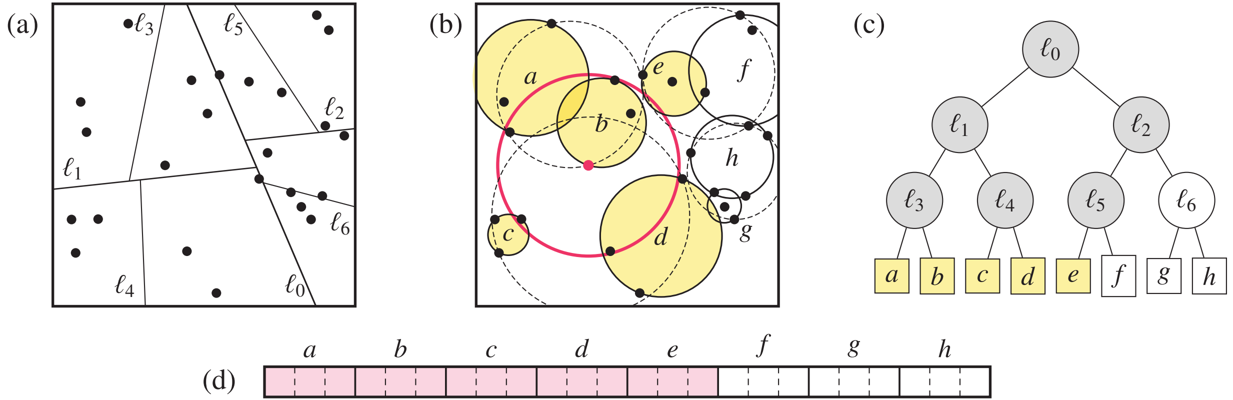

Fig. 6. Same as Fig. 4, but for a variant of the ball tree. Panel a shows the partition lines based on the PCA, and panel b shows the resulting minimum bounding spheres of leaf nodes and their parents.

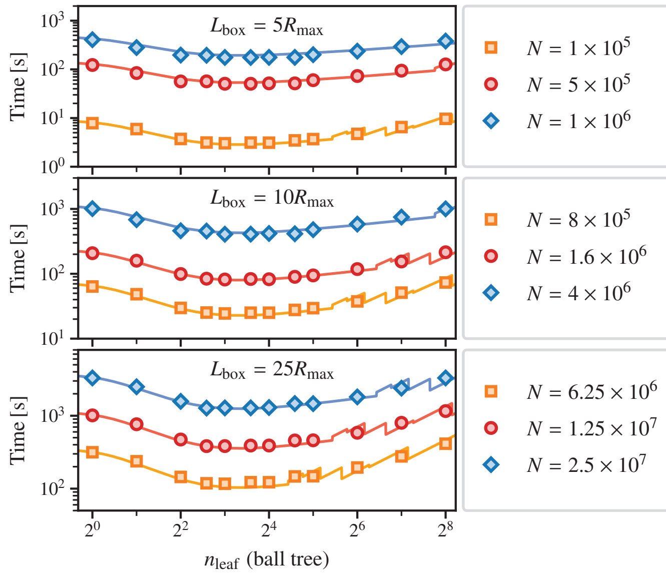

Fig. 7. Execution time of the pair-counting routine based on the ball tree with different capacities of leaf nodes for periodic uniform random samples with different cubic box sizes and numbers of points. The solid lines show the best-fitting theoretical results detailed in Appendix C.

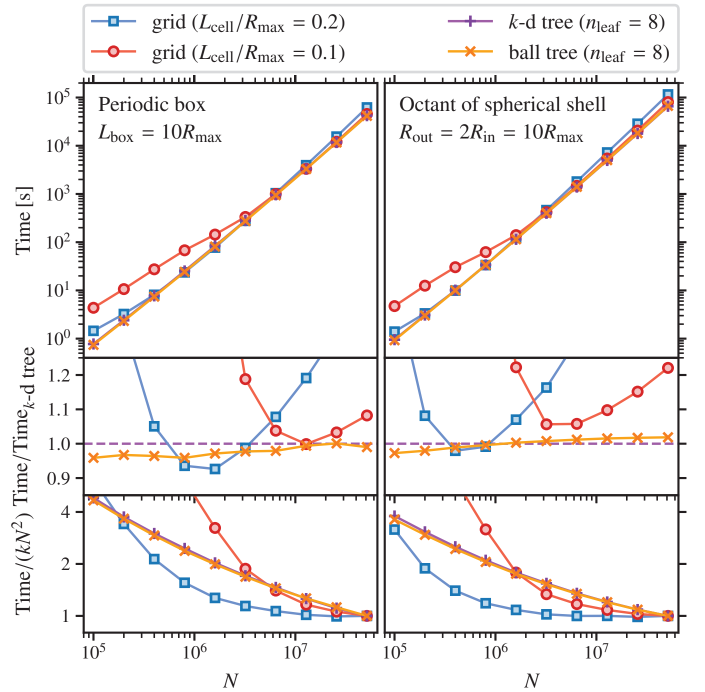

Fig. 8. Comparisons of the computational costs of pair-counting routines based on different data structures for periodic uniform random samples with different numbers of points in a cubic volume (left) and for sections between R_{in} and R_{out} of the same catalogues with respect to a corner of the boxes, where R_{out} is equal to the box size (right). The middle and bottom panels show results normalised by the costs with the k-d tree and kN^2, respectively. Here, the value of k is chosen such that the normalised costs at $N = 5 \times 10^7$ are unity.

the boxes, and took the sections in between as our non-periodic samples, which are essentially octants of spherical shells. The reason for using random samples is that the most computationally challenging pair-counting tasks in practice are usually with random catalogues because high-density randoms are required for clustering measurements.

We counted pairs in $[0, R_{max})$ as a whole to exclude costs from the histogram update process, as the goal was only to examine the data-pruning efficiencies of different data structures. For all tests, we set $L_{box} = 10R_{max}$, which is typical for modern cosmological applications, for instance, pair-counting with separations up to 200 h^{-1} Mpc, for simulations with a side length of 2 h^{-1} Gpc. We considered only cubic grid cells for regular grids, but with two choices of cell sizes, $0.1R_{max}$ and $0.2R_{max}$, which are near optimum for most cases shown in Fig. 3. We set $n_{leaf} = 8$ for the k-d and ball trees because it is the most favourable for almost all cases in Figs. 5 and 7.

For the benchmarks, we used the pair-counting algorithm described in Appendix B for regular grids and the dual-tree algorithm presented in Sect. 3.1 for the trees. Both algorithms were highly optimised to permit a reasonable comparison of the data-pruning efficiency of the data structures. We also note that identical pair counts are produced with all the data structures used. Our benchmark results with different input sample sizes are shown in Fig. 8. The performance of the two tree structures is very similar, with differences <5% for all our tests. Again, we find that the efficiency of the grid-based method is sensitive to the choice of cell size. In particular, the optimal cell size decreases as the sample size increases. This is because too many data points per cell may result in a large number of unnecessary pair-separation evaluations, while the cost of traversing cells can

be significant when many cells are almost empty. In contrast, the sizes of tree nodes are adaptive.

In terms of the execution time, when the number of data points is $\lesssim 10^6$, regular grids with the optimal cell size can be slightly better than the trees, but the improvement is only $\lesssim 5\%$ compared to the ball tree. However, when a sub-optimal cell size is used, the computing time with regular grids can be as long as twice that of the trees. When the number of data points is $\gtrsim 10^7$, the tree structures are always more efficient regardless of the choice of cell size for regular grids, especially for the non-periodic and non-cubic catalogues, but the speedups may be marginal in practice if accounting for the cost of the histogram update process. Meanwhile, for regular grids, the scaling of the computational cost with respect to the sample size reaches N^2 earlier than those of the tree structures. The scaling of the tree structures is better than N^2 even at $N = 5 \times 10^7$.

To conclude, both the k-d tree and the ball tree perform better than regular grids for modern and next-generation cosmological pair-counting problems with $\gtrsim 10^7$ objects in the data or random catalogues because the computational costs is lower in general, and because fine-tuning parameters are absent that depend on the input samples and strongly affect the performance. In particular, given the second reason, we did not test the grid-based method hereafter. There is no essential difference in the efficiencies of the two tree structures, and therefore, we implemented both the k-d tree and ball tree in the `FCFC` toolkit.

2.5. Discussion of the data structures for pair counting

A variety of data structures are available for different range-searching problems in the field of computational geometry (e.g. de Berg et al. 2008). The basic idea of time-efficient data structures is to allow for the report of groups of points directly without visiting them individually. Therefore, the complexity of a pair-counting algorithm can in principle be better than $O(N^2)$, which is the complexity of a brute-force approach that examines all data pairs. However, for cosmological applications, it is generally necessary to count pairs in thin separation bins. The 2PCFs are typically measured in (s, μ) or (σ, π) bins for anisotropic information, which are given by

$$s = |\boldsymbol{s}| = |\boldsymbol{s}_2 - \boldsymbol{s}_1|, \tag{2}$$

$$\pi = \frac{|\boldsymbol{s} \cdot \boldsymbol{l}|}{|\boldsymbol{l}|}, \tag{3}$$

$$\sigma = \sqrt{s^2 - \pi^2}, \tag{4}$$

$$\mu = \pi / s, \tag{5}$$

where $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ denote the coordinates of two points that form a pair, and $\boldsymbol{l}$ is the line-of-sight vector. For observational data, $\boldsymbol{l}$ is typically defined as

$$\boldsymbol{l}_{\rm obs} = \boldsymbol{s}_2 + \boldsymbol{s}_1, \tag{6}$$

while for simulations, the plane-parallel line of sight is usually assumed, for example,

$$\boldsymbol{l}_{\rm sim} = \hat{\boldsymbol{e}}_z = (0, 0, 1). \tag{7}$$

The complicated binning schemes make it difficult to find pairs of large data groups with separations all in the same bin. For instance, the typical number density of modern galaxy samples is $\rho \sim 10^{-3}\,h^3\,{\rm Mpc}^{-3}$. Then there is only one point in a cubic volume with a box size of $10\,h^{-1}\,{\rm Mpc}$ on average, which is already larger than the commonly used separation bin width of $5\,h^{-1}\,{\rm Mpc}$ for 2PCFs. This problem may be less severe for galaxy catalogues with strong clustering patterns. The most challenging tasks normally are pair counting with random samples for the normalisation of 2PCFs, however. In most cases, individual pairs have therefore to be visited to update the pair-counting histograms.

For a 3D periodic box, when $\rho R_{\rm max}^3 \gg 1$, the total number of pairs with separations in $[0, R_{\rm max})$ can be estimated by

$$\hat{N}_{\rm pair} = N \cdot \rho \frac{4\pi R_{\rm max}^3}{3} = N^2 \cdot \frac{4\pi}{3} \left(\frac{R_{\rm max}}{L_{\rm box}} \right)^3. \tag{8}$$

Since $\hat{N}_{\rm pair} \propto N^2$, the complexity of a real-world pair-counting algorithm is generally ineluctably $O(N^2)$. For this reason, the aim of the data structures described in this work is to reduce the constant factor hidden in the complexity by efficient data pruning. After all, $\hat{N}_{\rm pair}/N^2$ can be as small as $\sim 10^{-3}$ when $L_{\rm box} = 10 R_{\rm max}$. Hence, the pair-counting algorithm with a well-designed data structure can still be faster than the brute-force approach by a few orders of magnitude.

It is possible to reduce the complexity of the pair-counting process for certain cosmological problems in principle, however. As an example, because σ and π are independent with the plane-parallel line of sight, the evaluation of (σ, π) pair counts for periodic simulations can benefit from developments of orthogonal range queries (de Berg et al. 2008). For instance, following the spirit of the range tree (e.g. Lueker 1978), a binary tree with the z coordinates can be constructed, and at each node, there can be an associate k-d tree or range tree for the x and y coordinates. Then, groups of pairs in π bins can be reported in logarithmic time, and individual pair visits are only required for the associated 2D subtrees. This improves the overall pair-counting complexity with additional storage space. We leave detailed studies of this case to a future work.

For future samples with unprecedented number densities, isotropic pair counting with s bins alone can potentially be improved as well. For a given reference point (x_0, y_0, z_0), the pair-counting process is equivalent to a spherical range search, that is, to finding all points (x, y, z) within a certain radius R,

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 < R^2. \tag{9}$$

When we define $w \equiv x^2 + y^2 + z^2$, the condition can be rewritten as

$$2x_0 x + 2y_0 y + 2z_0 z - w - w_0 + R^2 > 0. \tag{10}$$

Therefore, the 3D spherical range-searching problem is converted into a 4D half-space range search, that is, finding all the points (x, y, z, w) above a given hyperplane. This is a well-known problem in computational geometry, and data structures exist that are able to accomplish the query in logarithmic time. Tradeoffs between the query time and storage costs are also possible (see de Berg et al. 2008; Agarwal 2017, for reviews). However, these data structures and algorithms are generally very difficult to implement in practice. We leave them for future developments.

It is possible to further boost the efficiency of pair counting hugely by allowing inexact solutions. For instance, there are data structures for approximate range queries with controlled errors that can be adjusted to vary the query time and storage costs (e.g. da Fonseca & Mount 2010). There are also 2PCF estimators that pixelate the volume (Alonso 2012), neglect the extent of tree

Algorithm 3 PAIRCOUNT_DUALTREE ($\mathcal{N}, \mathcal{S}, \mathcal{H}$)

Input: a stack $\mathcal{N}$ for pairs of tree nodes, the separation range $\mathcal{S}$ of interest, and the histogram $\mathcal{H}$ for storing pair counts.

1: Pop a pair of tree nodes $\{v_1, v_2\}$ from $\mathcal{N}$.
2: **if** DISTANCERANGE (v_1.bound , v_2.bound) $\cap \mathcal{S} = \varnothing$ **then**
3:     **return** ▹ descendants of both nodes are pruned
4: **else if** DISTANCERANGE (v_1.bound , v_2.bound) $\subseteq \mathcal{S}$ **or** v_1 and v_2 are both leaves **then**
5:     **for all** $\boldsymbol{p}_1 \in v_1$.data, $\boldsymbol{p}_2 \in v_2$.data **do**
6:         $d \leftarrow$ DISTANCE ($\boldsymbol{p}_1, \boldsymbol{p}_2$)
7:         **if** $d \in \mathcal{S}$ **then** update histogram $\mathcal{H}$ with d **end if**
8:     **end for**
9: **else if** neither of v_1 and v_2 is a leaf node **then**
10:     Push $\{v_1$.right, v_2.right$\}$ and $\{v_1$.right, v_2.left$\}$ onto $\mathcal{N}$.
11:     Push $\{v_1$.left, v_2.right$\}$ and $\{v_1$.left, v_2.left$\}$ onto $\mathcal{N}$.
12: **else if** v_1 is a leaf **then**
13:     Push $\{v_1, v_2$.right$\}$ and $\{v_1, v_2$.left$\}$ onto $\mathcal{N}$.
14: **else** ▹ v_2 is a leaf, but v_1 is not
15:     Push $\{v_1$.right, $v_2\}$ and $\{v_1$.left, $v_2\}$ onto $\mathcal{N}$.
16: **end if**

nodes that are far away from each other (Zhang & Pen 2005), or make use of fast Fourier transforms (e.g. Pen et al. 2003). It is important to validate these approximate methods in terms of the accuracies on different scales with modern cosmological data. We will perform relevant tests and combine the exact and inexact methods in `FCFC` to achieve a higher efficiency with tuneable precision in a follow-up paper.

3. Algorithms

Algorithms are another fundamental building block of a program in addition to data structures. A good algorithm may accomplish computational tasks efficiently by taking advantage of the layout of input datasets in memory given the data structure, or making use of memorisation to avoid redundant computations. The most important algorithms used by `FCFC` are those for identifying pairs within desired separation ranges, and updating histogram bins given a large number of (multi-dimensional) pair separations, which are usually the most time-consuming tasks for a correlation function calculator.

3.1. Tree-independent dual-tree algorithm

With the tree structures described in the previous section, a considerable fraction of unnecessary distance evaluations can be avoided provided an algorithm that detects node pairs that are not in the separation range of interest as early as possible. To this end, it is preferred to traverse trees in a top-down manner because when the separation range between a pair of parent nodes is entirely outside or inside the query range, all their descendant nodes can be omitted. In particular, for the latter case, we visited the data associated with the parent nodes directly to avoid unnecessary tree node visits. We then end up with Algorithm 3, which is an improved version of the dual-tree algorithm introduced by Moore et al. (2001). Our dual-tree algorithm is tree independent (see also Curtin et al. 2013b). Therefore, it is applicable to all binary space-partition tree structures in principle. The complexity of the dual-tree algorithm should depend on the tree structure, but in practice, the pair-counting efficiencies with the k-d and ball trees are quite similar (see Fig. 8).

Our algorithm traverses the tree in the so-called depth-first order, as it uses less memory than the breadth-first order for a balanced tree. This is because at a given level, the depth of the balanced binary tree is generally smaller than the width. We then maintain a stack for pairs of tree nodes to avoid recursive function calls in typical depth-first dual-tree algorithms (Moore et al. 2001; March et al. 2012). This increases the scalability of the algorithm with parallelisation, as different threads are able to work independently given their private stacks for dual nodes (see Sect. 4.2). The overhead due to the stack memory cost for recursive function calls is also mitigated. We do not directly report the total number of pairs from two nodes, as is done in Moore et al. (2001), because the examination of individual pairs is usually necessary for histogram updates with separation bins (see Sect. 2.5 for details).

Although not shown explicitly, the implementation of Algorithm 3 in `FCFC` is further optimised for some specific but common cases. For auto pair counts, we discard node pairs $\{v_2, v_1\}$ when $\{v_1, v_2\}$ is (going to be) visited to avoid duplicated pair examinations, as v_1 and v_2 belong to the same tree. In addition, following Sinha & Garrison (2020), we do not inspect individual pairs of data points for wrapping large separations when periodic boundary conditions are enabled. Instead, we compute the offsets of coordinates for the periodic wrapping of node pairs given their bounding volumes, and apply the offsets directly to all the associate data points. In this way, a large number of periodic boundary detections are avoided.

Moreover, the dual-tree algorithm can be directly applied to angular pair counts and can easily be extended for higher-order statistics, such as three- or four-point correlation functions. We leave relevant developments to future work.

3.2. Update of pair-counting histograms

The cost of histogram updates in Algorithm 3 can be considerable, as there are usually numerous pairs within the query range, which scales with $O(N^2)$ for most cases (see Eq. (8)). Therefore, the complexity of the histogram update process is usually $O(N^2)$, which is independent of data structures and algorithms. Nevertheless, it is possible to reduce the hidden constant factor with a smart algorithm. In general, this factor relies on the number of bins and the distribution of separations, which are then crucial for comparing the performance of different histogram update algorithms. In practice, pair separations are usually computed from the squared distances. We therefore sampled squared distances randomly following their expected distributions with a periodic box (see Appendix D for details) for the histogram update algorithm benchmarks. We assumed monotonically increasing histogram bins for the tests. In reality, this can be fulfilled by pre-sorting the bins. We also required the bins to be continuous, which is a common scenario in practice. Furthermore, we used zero-based bin indices throughout this work.

3.2.1. Comparison-based methods

A direct way of locating the histogram bins of given separation values is to compare them with the bin edges. In this case, the squared distances can be compared against pre-computed squared bin edges, without evaluating square roots for the actual separations. This improves the efficiency and the numerical stability of the algorithms. A commonly used method for this purpose is the binary search algorithm. The average complexity of this algorithm is $O(\log N_{\rm bin})$, where $N_{\rm bin}$ denotes the number

of histogram bins. This complexity is optimal for comparison-based methods when the separations are distributed uniformly across the bins and come in random order.

However, in reality, there are usually more pairs with larger separations (see Appendix D). Thus, it is worthwhile to consider a simple algorithm that continuously traverses the histogram bins in reverse order, that is, starting from the bin for the largest separations (see Sinha & Garrison 2020). The worst-case complexity of this algorithm is $O(N_{\rm bin})$. However, the average computational cost can be lower than that of the binary search algorithm, especially when the distribution of separations across the bins is highly asymmetric.

In principle, comparison-based methods can be further improved by taking advantage of the locality of separation values during the pair-counting process. This is particularly true for the tree structures discussed in Sect. 2, which groups nearby data points together. In this case, the splay tree is a potentially useful data structure for histogram updates, with which frequently accessed bins can be visited more quickly (Sleator & Tarjan 1985). Nevertheless, the performance of comparison-based methods is limited by the $N_{\rm bin}$ dependences and unavoidable conditional branches, which are harmful to the performance of instruction-level parallelism with modern pipelined processors. Given also the high efficiency of alternative algorithms introduced later, we did not implement a splay tree in this work.

3.2.2. Index-mapping functions

The pair separation histogram can be updated in constant time and without branches when it is possible to map squared distances directly onto the indices of the corresponding histogram bins. For evenly spaced bins on both linear and logarithmic scales, which are the most common configurations in practice, the index-mapping forms are simple. Thus, it is of practical interest to examine index-mapping algorithms for these specific cases.

For uniform linear separation bins in the range of $[s_{\rm min}, s_{\rm max})$, the index of the bin for a given squared distance s^2 is

$$\mathrm{i}_{\rm lin}(s^2) = \left\lfloor \frac{\sqrt{s^2} - s_{\rm min}}{\Delta_{\rm lin} s} \right\rfloor, \quad s^2_{\rm min} \leq s^2 < s^2_{\rm max}, \tag{11}$$

where $\Delta_{\rm lin} s$ indicates the width of the bins. Because the pair-counting process is independent of coordinate units, we can rescale data-point coordinates and histogram bins by $(1/\Delta_{\rm lin} s)$ in advance to eliminate the division in Eq. (11), thus improving the overall efficiency of the pair-counting algorithm. Eventually, we need only three operations for the evaluation of bin index for each valid pair, which are square root, subtraction, and floor.

Similarly, the index of squared distance s^2 for logarithmic bins in the range of $[s_{\rm min}, s_{\rm max})$ can be obtained by

$$\mathrm{i}_{\rm log}(s^2) = \left\lfloor \frac{\frac{1}{2}\log s^2 - \log s_{\rm min}}{\Delta_{\rm log} s} \right\rfloor, \quad s^2_{\rm min} \leq s^2 < s^2_{\rm max}. \tag{12}$$

Here, $\Delta_{\rm log} s$ is the width of the bins on logarithmic scale. Again, we can rescale all coordinates and histogram bins to further improve the efficiency. For instance, with a rescaling factor of $(1/s_{\rm min})$, the $(\log s_{\rm min})$ term in Eq. (12) can be omitted. Then, if pre-computing the factor $(2\Delta_{\rm log} s)^{-1}$, we end up with one logarithm, one multiplication, and one floor for the index mapping.

Although the complexity of index-mapping algorithms is only $O(1)$, which outperforms those of comparison-based methods, the actual computing time largely depends on the efficiency of the index calculations. A considerable number of comparisons can be accomplished during the evaluation of the logarithm in Eq. (12). Therefore, histogram update algorithms based on index-mapping functions are not necessarily faster than the methods described in Sect. 3.2.1, especially when $N_{\rm bin}$ is small. To make the constant-time complexity effective, we need more efficient index-mapping methods than the direct function evaluations, not to mention the limited numerical precision of these functions.

3.2.3. Index lookup tables

A common way of accelerating the evaluation of a numerical function is to look up pre-computed values from a table. This technique can be very efficient if the domain of the function is discrete and reasonably small. In general, index-mapping functions for histogram updates do not fulfil this condition, as the squared distances can be of any value inside $[s^2_{\rm min}, s^2_{\rm max})$. Nevertheless, when the edges of histogram bins are integers, only the integer part of a squared distance determines the index of the histogram bin. In this case, we can create an index lookup table, the keys to which are the integer parts of all possible squared distance values. The index-mapping process can then be completed by truncating squared distances and looking up indices in the table. Moreover, the efficiency of this method can benefit from the data locality with the tree structures discussed previously, which reduces the cache-miss rate of table lookup.

This method is also applicable when all the histogram bin edges can be converted into integers by a common rescaling factor, as it is permissible to rescale the histogram bins together with the coordinates of data points. This is a common scenario in practice. For instance, given equally spaced separation bins with $s_{\rm min} = 0$, the rescaling factor that converts all bin edges into integers is simply the inverse of the bin width. However, because the length of the lookup table is

$$N_{\rm table} = \lfloor s^2_{\rm max} \rfloor - \lfloor s^2_{\rm min} \rfloor, \tag{13}$$

when the (rescaled) distance range is wide, the table may be too large to fit in the CPU caches. As a result, the lookup efficiency can be significantly downgraded due to the expensive memory accesses. One solution to this problem is to rescale the histogram bins by a factor that is smaller than 1. The bin edges are the not guaranteed to be integers, however. When we also consider cases in which the bin edges cannot all be converted into machine-representable integers, an index lookup algorithm that does not rely on integer bin edges is necessary.

For non-integer bin edges, we have to take care of non-injective lookup table entries. This is because squared distances belonging to different separation bins may share the same integer part. In this case, we can record the index ranges for non-injective entries and use a comparison-based method to further identify the exact index for a given squared distance. In practice, we used the reverse traversal algorithm (see Sect. 3.2.1) because it is simple. It is worth noting that this hybrid index-lookup method is able to deal with separation bins with arbitrary bin edges and widths as long as the bins are continuous.

The efficiency of this method depends on the rescaling factor of the histogram bins. When the factor is small, there is a higher chance of encountering non-injective table entries, which requires further comparisons that are relatively slow. In contrast, large rescaling factors yield large tables that may increase

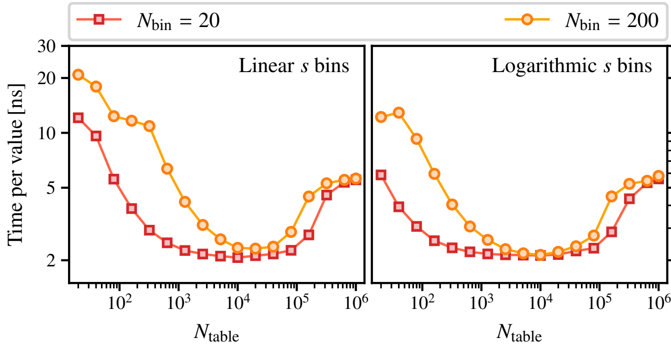

Fig. 9. Performance of the separation histogram update routine based on the hybrid index-lookup algorithm with different lookup table sizes. The execution time is measured with 6.4×10^9 randomly sampled squared distances in the range of $[0, 200^2)\,h^{-2}\,\text{Mpc}^2$. Both linear and logarithmic separation bins are tested, with s in ranges of $[0, 200)$ and $[0.1, 200)\,h^{-1}\,\text{Mpc}$, respectively. There are also two different numbers of bins, 20 and 200, for the two binning schemes.

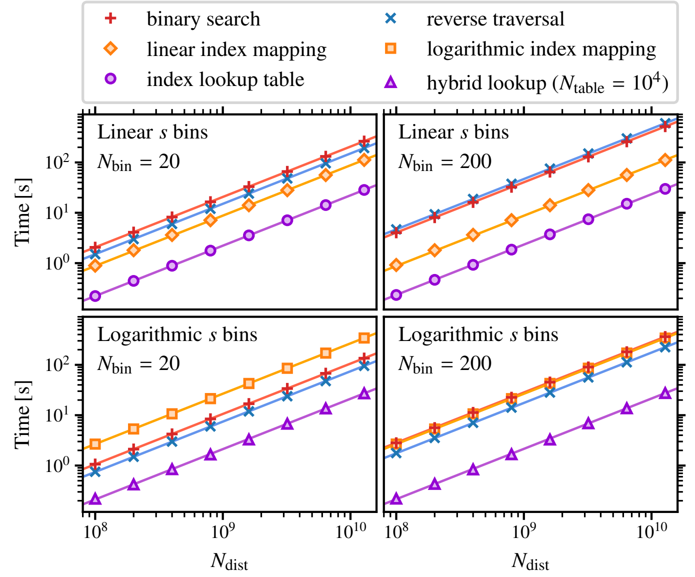

Fig. 10. Execution times of various histogram update algorithms for different histogram bins and numbers of squared distances sampled in the range of $[0, 200^2)\,h^{-2}\,\text{Mpc}^2$. The separation ranges of the linear and logarithmic bins are $[0, 200)$ and $[0.1, 200)\,h^{-1}\,\text{Mpc}$, respectively. Two different numbers of bins are also tested. The solid lines show the best-fitting straight lines with a constant execution time per squared distance.

the cache-miss rate. In principle, the optimal rescaling factor depends on the CPU cache sizes and should be estimated through benchmarks.

3.2.4. Comparison of the histogram update algorithms

In order to compare the performance of the different histogram update algorithms discussed so far, and to choose the optimal table size for the hybrid index-lookup method, we performed a series of benchmark tests on Haswell CPUs with squared distance values sampled randomly following Appendix D. In particular, the squared distances were sampled in the range of $[0, 200^2)\,h^{-2}\,\text{Mpc}^2$. We examined both linear and logarithmic separation bins, which are the most commonly used binning schemes in practice, with s in ranges of $[0, 200)$ and $[0.1, 200)\,h^{-1}\,\text{Mpc}$, respectively. To inspect the $N_{\rm bin}$ dependences of the algorithms, we further tested two different numbers of histogram bins, 20 and 200, for both binning schemes. Some of the algorithms required rescaling of the squared distances, which can be achieved by pre-processing the coordinates of all data points in reality. The computational cost of this pre-processing step is $\mathcal{O}(N)$, which is generally much smaller than that of the histogram update process with $\mathcal{O}(N^2)$ pairs. Thus, the costs of histogram update routines we report do not include those for rescaling separations.

Given 6.4×10^9 random squared distances, the execution times of the hybrid index-lookup algorithm with different lookup table sizes and histogram bins are shown in Fig. 9. A table with $\sim 10^4$ entries is always almost optimal, regardless of the separation bin configurations. The indices of histogram bins can be represent by 8- or 16-bit integers in most cases, and therefore, the memory cost of a table with $\sim 10^4$ entries is about 10 to 20 KB, which fits in the level-1 (L1) cache of most modern CPUs for supercomputers. This explains the optimality of the table size. The optimal histogram update cost per squared distance value is about 2 ns for all the separation bin configurations in Fig. 9, which correspond to barely ~ 5 Haswell CPU cycles, that is, slightly larger than the four-cycle latency of L1 cache accesses (Fog 2022). This means that we achieve almost the maximum theoretical efficiency for histogram updates. Thus, we always chose separation rescaling factors that yielded $N_{\rm table} \sim 10^4$ for the hybrid index-lookup algorithm.

We furthermore compared the performance of different histogram update algorithms with the same input squared separation sequences and histogram bins. The results are presented in Fig. 10. We did not use the hybrid method for linear separations bins because the bin edges are integers and lookup tables are directly applicable. In all cases, the execution time scales linearly with $N_{\rm dist}$, the number of squared distances sampled. It is not surprising that the comparison-based methods, binary search and reverse traversal, are sensitive to both the binning scheme and number of separation bins. The performance of index-mapping algorithms also largely depends on the binning schemes, but not on $N_{\rm bin}$. This can be explained by the different costs of the index-mapping functions. The linear index-mapping method expressed by Eq. (11) is faster than the comparison-based methods for the examined $N_{\rm bin}$ values; and the logarithmic mapping shown in Eq. (12) is generally less efficient, especially when compared to the reverse traversal algorithm. In contrast, the index-lookup algorithms are insensitive to the configurations of separation bins, and they outperform all the other methods in all the cases presented here. The lookup cost for each squared distance value is always ~ 2 ns on average.

We then implemented the index-lookup methods in FCFC for pair counting because they are highly efficient and can deal with arbitrary separation bins. In particular, for linear separation bins, we computed the smallest positive factor that converts both edges of the first bin into integers. Given the separation ranges rescaled by this factor, if the $N_{\rm table}$ expressed by Eq. (13) is $\lesssim 3 \times 10^4$, we used the index-lookup table for integer bin edges directly. For all the other cases, either the $N_{\rm table}$ computed in this way is too large or the separation bins are not evenly spaced, we relied on the hybrid index-lookup method with $N_{\rm table} \sim 10^4$. In order to eliminate potential numerical errors due to rescaling, we always chose a rescaling factor that was a power of the radix used by floating point representations and that yielded a lookup table size that was closest to 10^4. In this case, the rescaling only changes the exponent of almost all floating-point numbers, so the mantissas

were untouched and no additional numerical errors were introduced. When the factor was chosen, we rescaled all histogram bins and the coordinates of the data points accordingly.

4. Parallelisation

Modern multi-core vector processors are able to run multiple independent instructions simultaneously on different pieces of data. HPC clusters are usually equipped with hundreds or thousands of such CPUs. To make full use of the computing facilities, the computational task needs to be broken down into similar sub-tasks, and different levels of parallelisms need to be used.

4.1. SIMD

Single instruction, multiple data (SIMD) refers to a type of data-level parallelism that permits operations of multiple data (i.e. a vector) with a single instruction[3]. For instance, most of the modern x86 CPUs support advanced vector extensions (AVX), which provides 256-bit registers for eight single-precision or four double-precision floating-point numbers to be processed simultaneously. A number of CPUs also support advanced vector extensions 2 (AVX2), which is an extension of AVX with the same register width, but more instructions, or even AVX-512, which permits 512-bit SIMD operations. AVX-512 consists of multiple extension sets. We focus on AVX-512 foundation (AVX-512F) in this work because it is available for all AVX-512 implementations and is sufficient for our application.

SIMD is potentially able to boost the performance of a pair-counting code because distances between different pairs of data points can be evaluated at once, which has to be processed for each individual pair with the conventional sequential (also known as scalar) approach. Thus, the traversal of points on pairs of tree nodes can be highly accelerated. In contrast, SIMD does not help the data-pruning process much because the maintenance of the dual-node stack (see Sect. 3.1) cannot be parallelised with vector operations. In this case, larger tree nodes and fewer node comparisons are preferred for a better overall pair-counting efficiency, so that the optimal leaf node capacity may change with different register widths.

To explore the optimal $n_{\rm leaf}$ for the k-d tree and ball tree with AVX and AVX-512, we performed a new set of benchmarks with both the scalar and vectorised dual-tree algorithms on the Haswell and Knights Landing CPUs (see Appendix A). Because only distance evaluations were vectorised here, which involves barely elementary arithmetic, the speedups are expected to be very similar for newer CPUs. In particular, we checked both single- and double-precision arithmetics by using the `float` and `double` data types in the C progamming language, as illustrated in Fig. 11. Similar to the tests in Sect. 2, we measured the execution time of the pair-counting algorithm, which reports the number of pairs with separations smaller than $R_{\rm max}$ for 4×10^6 uniformly distributed random points in a cubic box with a side length of $L_{\rm box} = 10R_{\rm max}$. It has been shown previously that the optimal $n_{\rm leaf}$ is not sensitive to the specifications of the input samples. Therefore, we did not vary the box size or the number of data points here. Figure 11 shows that with SIMD, $n_{\rm leaf} = 32$ is near optimal for almost all cases. Moreover, when $n_{\rm leaf} \gtrsim 32$, theoretical maximum speedups with SIMD are generally achieved. For instance, AVX is able to process four double-precision numbers at once, and the actual speedups of the AVX-vectorised algorithms are indeed ∼4 with respect to the scalar counterparts. It implies that the efficiency of the dual-tree algorithm is not likely to be surpassed by the SIMD-vectorised pair-counting algorithm based on regular grids. The speedup can be larger than the number of floating-point numbers processed simultaneously. This might be due to additional efficiency boosts with the fused multiply-add (FMA) instructions that are available with most modern SIMD implementations.

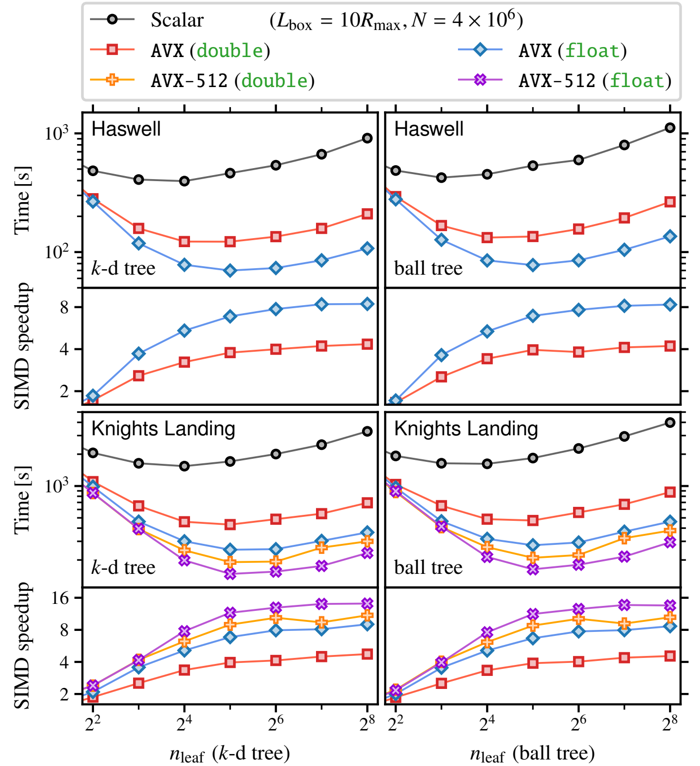

Fig. 11. Execution time of the scalar, AVX-vectorised, and AVX-512-vectorised pair-counting routines based on the k-d and ball trees, with different capacities of leaf nodes, for a periodic uniform random sample. Results for both Haswell and Knights Landing CPUs are shown. Speedup is measured as the ratio of the computing time of the scalar code to that of the vectorised counterpart.

Our histogram update process, however, may benefit from SIMD. On one hand, the index-lookup methods are sufficiently fast for the access of CPU caches to have become the bottleneck (see the discussions in Sect. 3.2.4). On the other hand, AVX does not provide instructions for reading lookup tables and maintaining histograms. In this case, only the floor operation can be vectorised, while the rest of the histogram update process has to be implemented in scalar. The more recent AVX2 instruction provides the `gather` operation, which loads multiple elements from non-contiguous memory locations, and might useful for loading lookup tables and histogram counts. However, there is still no instruction for the update of histogram with AVX2. Only with AVX-512 are both `gather` and `scatter` operations available, where `scatter` stores multiple data at different memory locations at once. Therefore, AVX-512 permits a full vectorisation of our histogram update algorithm.

We then vectorised the histogram update process with different SIMD instruction sets and performed benchmarks on a number of different CPUs using 10^{10} randomly generated squared separation sequences with the same binning schemes as in Sect. 3.2.4. For AVX-512, we maintained private histograms

[3] We implement SIMD with the intrinsics available in the C programming language throughout this work.

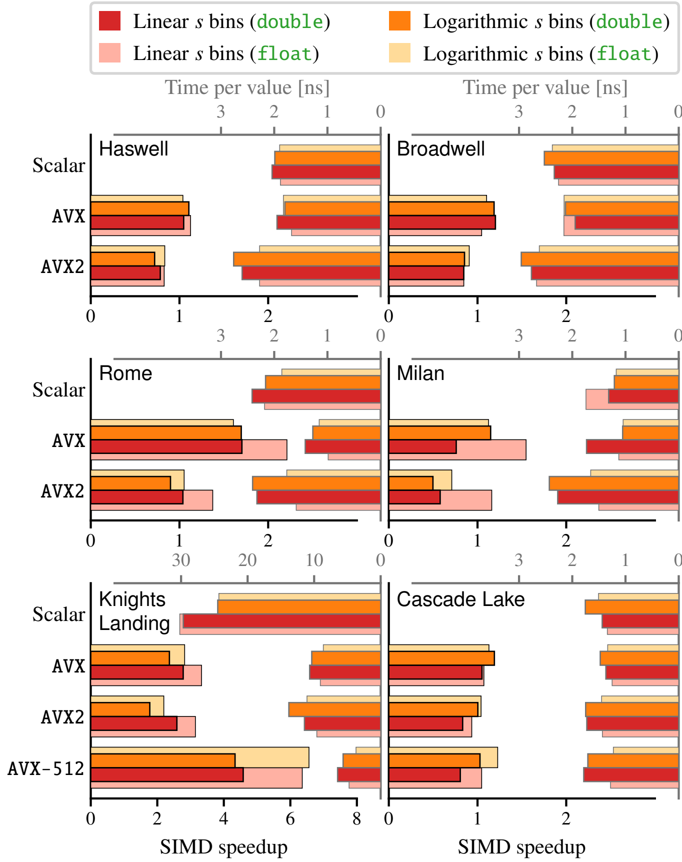

Fig. 12. Execution time of the histogram update algorithms measured upon 10^{10} random squared distances, and the speedups of the vectorised algorithms with respect to the scalar counterparts on different CPUs with different histogram bin settings and precisions of floating-point numbers.

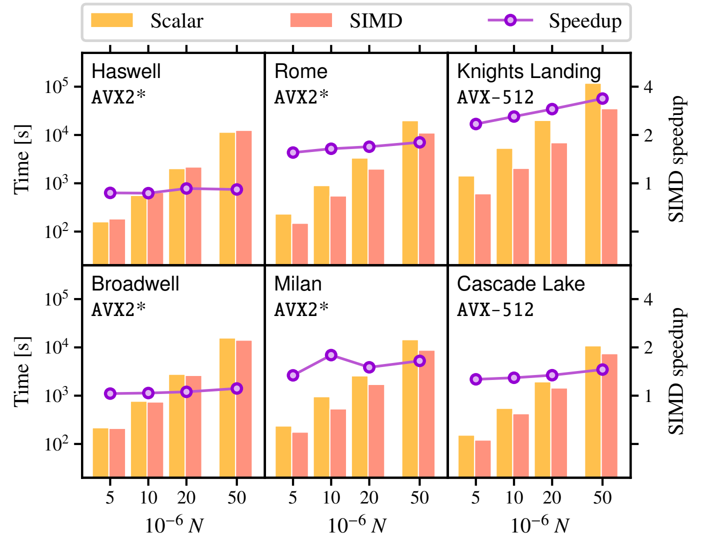

Fig. 13. Execution time of the scalar and vectorised versions of `FCFC` on different CPUs (bars) for the full pair-counting procedure with 200 linear s bins in $[0, 200)\,h^{-1}$ Mpc and 120 μ bins in $[0, 1)$, run upon periodic random samples in a cubic box with a side length of $3\,h^{-1}$ Mpc. The purple lines indicate the speedups of the SIMD-parallelised versions with respect to the scalar counterparts. `AVX2*` indicates AVX2, but without the `gather` instructions.

for individual vector elements to avoid conflicts, rather than relying on the conflict detection instructions (AVX-512CD). In this way, we eliminated costs due to conflict detection and branching by trading off memory usage. The averaged processing time of each squared separation value, as well as the speedups of the vectorised versions with respect to the scalar counterparts, are illustrated in Fig. 12. Improvements with SIMD are almost always marginal, except for the Knights Landing CPU. This can be explained by the limits of cache throughputs. After all, for most of the CPUs tested, the cost of processing one square distance value is barely a few nanoseconds with the scalar code. Moreover, with AVX, the main components of the histogram update algorithm are not vectorised. The inclusion of the `gather` instruction alone with AVX2 is harmful to the efficiency of index lookups, possibly because the algorithm is not fully vectorised, and there are additional micro-operations than memory loads (see Chapter 15, Intel Corporation 2022). The index-lookup algorithm can be significantly accelerated by AVX-512 on the Knights Landing CPU, while for Cascade Lake, the performance of the vectorised and scalar codes is very similar. This shows that AVX-512 is only useful when the histogram update procedure is significantly slower than the latency of cache access. Given these benchmark results, `FCFC` makes use of `gather` only when `scatter`, or AVX-512, is available. This does not mean that AVX2 is useless because we benefit from the versatile vectorised integer arithmetics introduced by AVX2.

To further examine whether or to which degree SIMD is beneficial to the full pair-counting procedure, including both distance evaluations and histogram update, we compared the entire runtime of the scalar and vectorised `FCFC` on different CPUs for auto pair counts upon periodic cubic random catalogues with a box size of $3\,h^{-1}$ Gpc, with 200 linear s bins in $[0, 200)\,h^{-1}$ Mpc and 120 μ bins in $[0, 1)$, which is a common setting in practice. The results are presented in Fig. 13. We conclude that SIMD is generally useful, although the overall improvement can be marginal on certain CPUs. Thus, we always enabled SIMD parallelisation throughout this work.

4.2. OpenMP

Open multi-processing[4] (OpenMP) is a high-level application programming interface (API) that provides a set of compiler directives, library routines, and environment variables for multi-thread parallelisms with shared memory. It is usually possible to parallelise a program with high scalability using OpenMP, with little modification of the serial code. Therefore, multi-threading with OpenMP is generally easy to implement to take advantage of multi-core processors. It is thus used extensively in cosmological applications, including pair-counting programs (e.g. Alonso 2012; Donoso 2019; Sinha & Garrison 2020).

However, it is not trivial to parallelise our dual-tree algorithm (see Algorithm 3) with high scalability. The update of the dual-node stack has to be executed by one thread at a time to prevent race conditions. This may result in additional overhead. It is possible to reform the algorithm as a recursive function, but then there additional costs accrue due to recurrent function calls and creations of threads for subtasks. One way to eliminate these expenses is maintaining a private stack on each thread. To this end, the dual-node stack has to be initialised with multiple elements that can be assigned to different threads and run independently. In this way, the initialisation and allocation of node pairs are crucial for the load balancing of the parallelised dual-tree algorithm.

[4] `https://www.openmp.org`

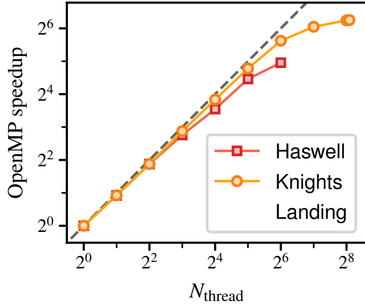

Fig. 14. Speedups of the OpenMP-parallelised FCFC with respect to the serial version on Haswell and Knights Landing CPUs with different numbers of OpenMP threads, measured using a periodic cubic random catalogue with $N = 5\times10^7$, $L_{\rm box} = 3\,h^{-1}$ Gpc, and with 200 linear s bins in $[0, 200)\,h^{-1}$ Mpc and 120 μ bins in $[0, 1)$. The dashed line denotes the theoretical maximum speedup. SIMD is enabled in all cases.

In principle, Algorithm 3 can be run with a single thread until the dual-node stack is sufficiently large, and then the node pairs can be distributed to different threads. However, with the depth-first tree traversal order, node pairs on the stack differ significantly in size as the number of points on each node depends mainly on the level (or depth) of the node. In this case, the word loads of different threads are normally highly unbalanced, which is harmful to the efficiency of the parallelised program. To circumvent this problem, we relied on the breadth-first tree traversal order for the initialisation of node pairs, which were then stored in a queue rather than a stack. Thus, after each iteration, the node pairs in the queue were all at the same level and consisted of a similar number of data points. When the queue was large enough, we distributed the node pairs to different OpenMP threads. The work loads were still not perfectly balanced in general, however, because the numbers of pairs within the query range can vary among different node pairs. It should be possible to further increase the performance of the parallelised dual-tree algorithm by using better scheduling strategies, such as the work-stealing technique (e.g. Blumofe & Leiserson 1999). For instance, the scaling efficiency of the 2PCF algorithm developed by Chhugani et al. (2012) is remarkable even with over 25 000 threads[5]. We leave relevant investigations to a future work.

The performance of the OpenMP-parallelised FCFC on different CPUs is shown in Fig. 14. The benchmarks were performed with a periodic cubic random sample with 5×10^7 points and a box size of $3\,h^{-1}$ Gpc. Similar to the case in Sect. 4.1, we measured auto pair counts with 200 linear s bins in $[0, 200)\,h^{-1}$ Mpc and 120 μ bins in $[0, 1)$. The speedup scales quite well with the number of threads when there are $\lesssim$32 OpenMP threads. With more threads, the speedups significantly deviate from the theoretical maximum values on both CPUs, possibly because of the non-negligible overheads of maintaining a large number of threads, as well as the imperfect work balancing. The scalability of the OpenMP-parallelised FCFC is reasonably good. Therefore, it is always recommended to enable OpenMP for pair-counting tasks with FCFC.

4.3. MPI

The message-passing interface (MPI) is a standard that defines a communication protocol for high-performance parallel computing on distributed memory systems. It permits multi-process

[5] However, the algorithm of Chhugani et al. (2012) is mainly useful for isotropic 2PCFs with a small number of separation bins, and it is therefore not general enough for actual cosmological applications.

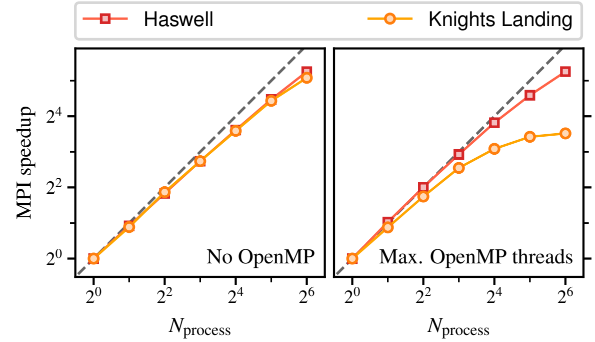

Fig. 15. Speedups of the MPI-parallelised FCFC with respect to the version without MPI on Haswell and Knights Landing CPUs with different numbers of MPI processes, measured using periodic cubic random catalogues with $N = 5 \times 10^7$ (left) and 5×10^8 (right), $L_{\rm box} = 3\,h^{-1}$ Gpc, and with 200 linear s bins in $[0, 200)\,h^{-1}$ Mpc and 120 μ bins in $[0, 1)$. The dashed lines indicate the theoretical maximum speedup. The left panel shows results without OpenMP, and the right panel presents results with the maximum available numbers of OpenMP threads. SIMD is enabled in all cases.

programs that are able to make use of almost all computing resources of a cluster in principle. In practice, MPI is commonly used along with OpenMP. In this hybrid paradigm, MPI is typically used across the computing nodes or sockets of a cluster, while OpenMP is used within nodes or sockets to reduce communication overhead and memory usage. Thus, a better scalability may be achieved than in pure MPI or OpenMP manners.

Our parallelised dual-tree algorithm discussed in Sect. 4.2 is applicable to the MPI parallelism. After creating the queue with node pairs at the same tree level, bulks of tasks can be assigned to different MPI processes, and then the breadth-first tree traversal procedure is repeated on each process to generate subtasks for threads if OpenMP is enabled in the meantime. In this way, the pair-counting routine is executed independently by different processes, and no communication is needed. In practice, the trees and dual-node queues are constructed on a single process and are broadcasted so that all processes maintain a local copy. The node pairs in the queue are then dynamically assigned to processes using the remote accessible memory (RMA) routines to achieve better load balancing. Thus, the only additional steps for MPI compared to the OpenMP-parallelised version are the synchronisations of trees and lookup tables among all processes, as well as the gathering of pair-counting results at the end.

The performance of the MPI-parallelised FCFC with and without OpenMP is presented in Fig. 15 for auto pair counts upon periodic cubic random samples with 5×10^8 and 5×10^7 points, respectively, and the same box size and binning scheme as in Sect. 4.2. For FCFC with MPI, but without OpenMP, the speedups scale well with the number of MPI processes on the Haswell and Knights Landing nodes. The trends are similar to those shown in Fig. 14, for which only OpenMP was enabled. This is expected as we distributed the work loads in the same way. When enabling OpenMP along with MPI and running FCFC with the maximum available number of OpenMP threads (64 on Haswell and 272 on Knights Landing), the speedups are basically unchanged on Haswell, but there is a significant degradation in the efficiency when the number of processes is $\gtrsim$8. This may be due to the fact that small imbalances of work loads become critical with thousands of independent threads running simultaneously. As discussed in Sect. 4.2, we leave the

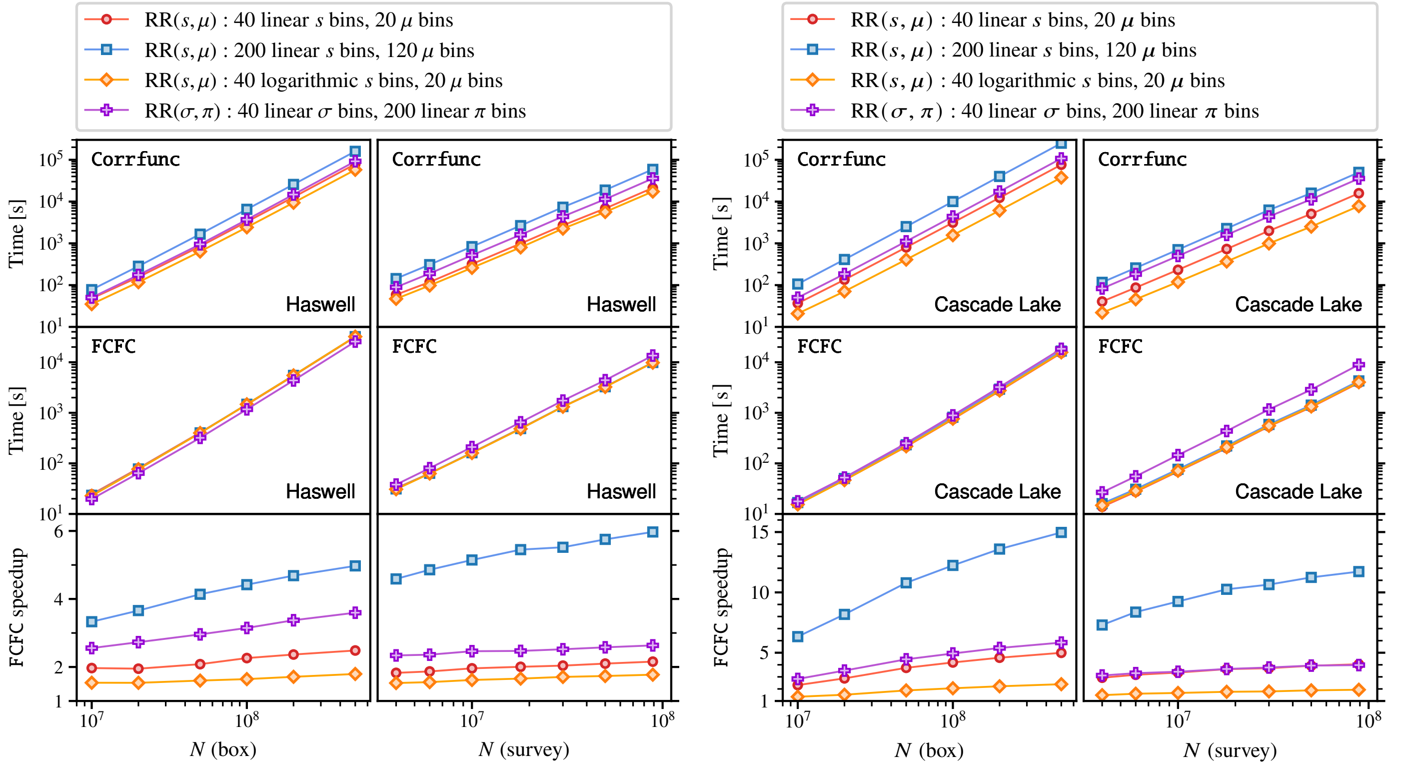

Fig. 16. Performance of FCFC and Corrfunc for auto pair counts with periodic and survey-like random samples with different numbers of points. Both OpenMP and SIMD parallelisms are enabled. The codes are run on entire nodes with all cores of Haswell and Cascade Lake CPUs.

exploration of a better work-load scheduler to a forthcoming paper.

5. Comparison with related work

To determine whether FCFC is useful in practice, it is important to run it with real-world applications and to compare the efficiency against related pair-counting tools. Sinha & Garrison (2020) have performed extensive benchmarks with a number of different pair-counting codes, and they concluded that Corrfunc outperforms all the other publicly available tools they tested, including SciPy cKDTree[6] (Virtanen et al. 2020), Scikit-learn KDTree[7] (Pedregosa et al. 2011), kdcount[8], Halotools (Hearin et al. 2017), TreeCorr (Jarvis 2015), CUTE (Alonso 2012), MLPACK RangeSearch (Curtin et al. 2013a), and SWOT[9], for auto pair counts with logarithmic bins upon simulation catalogues with $\gtrsim 10^5$ objects in a cubic volume of $1100^3\,h^{-3}\,\mathrm{Mpc}^3$. For simplicity, we therefore compare FCFC (version 1.0.1[10]) only with Corrfunc (version 2.4.0[11]) in this work.

The most expensive tasks in reality are usually random–random pair counts. We therefore focused only on auto pair counts with random catalogues. Due to the differences in boundary periodicity and line of sight for pair counting with simulation and observational data (see Sect. 2.5), we examined two sets of randoms: (1) 5×10^8 uniformly distributed random points in a cubic box with a side length of $3\,h^{-1}$ Gpc to mimic the random catalogue for a periodic simulation (1) the actual random samples for the BOSS DR12 data[12], with weights enabled for pair counting. These two random catalogues were further randomly down-sampled for benchmarks with smaller datasets. For all catalogues, we performed pair counts with the following binning schemes:

(1) 40 linear s bins in $[0, 200)\,h^{-1}$ Mpc and 20 linear μ bins in $[0, 1)$;
(2) 200 linear s bins in $[0, 200)\,h^{-1}$ Mpc and 120 linear μ bins in $[0, 1)$;
(3) 40 logarithmic s bins in $[0.1, 200)\,h^{-1}$ Mpc and 20 linear μ bins in $[0, 1)$;
(4) 40 linear σ bins in $[0, 200)\,h^{-1}$ Mpc and 200 linear π bins[13] in $[0, 200)\,h^{-1}$ Mpc.

The performance of FCFC and Corrfunc on the Haswell and Cascade Lake[14] nodes with double-precision arithmetics is shown in Fig. 16. Here, we enabled OpenMP and SIMD for both codes[15] and ran them on an entire computing node with all available resources, that is, 64 threads with AVX2 on Haswell, and

[6] https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html
[7] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html
[8] https://doi.org/10.5281/zenodo.1051242
[9] https://github.com/jcoupon/swot
[10] https://github.com/cheng-zhao/FCFC/releases/tag/v1.0.1
[11] https://github.com/manodeep/Corrfunc/releases/tag/2.4.0

[12] We merge 'random0_DR12v5_CMASSLOWZTOT_North.fits.gz' and 'random1_DR12v5_CMASSLOWZTOT_North.fits.gz' in https://data.sdss.org/sas/dr12/boss/lss/, to form a random sample with $\sim 9 \times 10^7$ objects.
[13] The numbers of σ and π bins are different, as Corrfunc only allows linear π bins with a width of $1\,h^{-1}$ Mpc.
[14] We did not test with Knights Landing CPUs as Corrfunc requires more advanced AVX-512 instructions than those available on Knights Landing.
[15] Corrfunc is not MPI-parallelised.

36 threads with AVX-512 on Cascade Lake. The pair-counting results from the two codes are identical, and therefore, we only compare their efficiencies here. FCFC is faster than Corrfunc for all cases. For logarithmic bins, the speedups of FCFC are relatively small, while with linear bins, especially when the bin counts are large, the speedups can be prominent. For the binning scheme (2), which is commonly used in practice for the ease of rebinning with different bin widths, FCFC can be five and ten times faster than Corrfunc with $\gtrsim 10^8$ objects on Haswell and Cascade Lake CPUs, respectively. Speedups are generally consistent with those of the index-lookup algorithms for histogram update (see Sect. 3.2.4). Thus, we conclude that the high efficiency of FCFC is mainly due to the novel histogram update algorithm.

6. Conclusions

We have presented FCFC, a high-performance software package for exact pair counting. It is highly optimised for cosmological applications, but should be useful for calculations of all kinds of two-point correlation functions or radial distribution functions with 3D data. We mainly focused on the efficiency and scalability of the tool in this paper, but FCFC is also portable, flexible, user-friendly, and applicable to a number of different practical problems, such as the calculation of radial distribution functions in statistical mechanics. A brief guide to the toolkit can be found in Appendix E.

We have compared three different data structures for pair-counting applications, that is, regular grids, the k-d tree, and a novel variant of the ball tree. For the tree structures, we make use of an improved dual-tree algorithm for pair counting. We showed that the performance of regular grids is sensitive to the choice of grid size. With a sub-optimal grid size, the efficiency of the pair-counting procedure can be substantially degraded. In contrast, the tree-based methods are almost always optimal for a fixed capacity of leaf nodes, and thus there is no free parameter for the tree constructions. In particular, we set $n_{\rm leaf} = 8$ for the scalar version of FCFC, while $n_{\rm leaf} = 32$ was used for the SIMD-vectorised implementation. When the number of data point is sufficiently large, both trees outperform regular grids regardless of the grid size, but the improvements may be marginal for cosmological catalogues with 10^7–10^8 objects. The efficiencies of the two tree structures are similar, however. Therefore, we implemented both tree structures in FCFC.

We also introduced a new histogram update algorithm based on index-lookup tables to speed up the increment of separation bins for correlation functions. For non-integer bin edges, the lookup table was used together with a comparison-based reverse traversal algorithm to locate histogram bins. Thus, our index lookup method is applicable to arbitrary binning schemes, including multi-dimensional bins for anisotropic measurements. According to the comprehensive benchmarks with different practical binning schemes, the index lookup method is shown to be considerably faster than the other commonly used histogram update algorithms for all cases.

Then, we parallelised FCFC with three levels of common parallelisms, that is, SIMD of vector processors, shared-memory OpenMP, and distributed memory MPI, with which it is possible to make full use of all computing resources of a cluster in principle. The key gredient of FCFC, that is, the index-lookup algorithm for separation bin updates, benefits only little from SIMD because the main bottleneck is likely to be the latency of CPU cache accesses. Nevertheless, the efficiency of FCFC scales well with the numbers of MPI processes and OpenMP threads, as long as the total number of threads does not exceed a few thousands. When the number of threads is too large, the performance boost due to parallelisation may be downgraded.

Finally, we compared OpenMP- and SIMD-parallelised FCFC and Corrfunc with the same number of computing resources, input catalogues, and binning schemes for pair counting. We find that FCFC is faster than Corrfunc for all the cases tested. The speedup is the most prominent with a large number of linear separation bins. FCFC can be more than ten times faster than Corrfunc on modern AVX-512 CPUs for catalogues containing $\sim 10^8$ objects and for pair counting with 200 linear s bins and 120 μ bins, which is a common setting for 2PCF calculations in practice. Thus, FCFC is a very promising tool for modern and future cosmological clustering measurements.

We will further extend our methods for more cosmological applications in the future, such as angular and high-order clustering statistics, including in particular three- and four-point correlation functions. Approximate methods will also be explored to further speed up the measurements with tolerable errors. Moreover, we will implement more advanced load-balancing schemes to further increase the scalability of FCFC, and hopefully make use of GPU acceleration.

Acknowledgements. I thank Charling Tao, Chia-Hsun Chuang, Daniel Eisenstein, and Lehman Garrison for useful discussions on pair-counting algorithms. This work is supported by the Swiss National Science Foundation (SNF) 'Cosmology with 3D Maps of the Universe' research grants 200020_175751 and 200020_207379. FCFC also benefits from a number of open-source projects, such as Fast Cubic Spline Interpolation (https://doi.org/10.5281/zenodo.3611922) (Hornbeck 2020), MedianOfNinthers (https://github.com/andralex/MedianOfNinthers) (Alexandrescu 2017), and sort (https://github.com/swenson/sort). The benchmarks in this work are run on the Baobab and Yggdrasil HPC clusters at Université de Genève (UNIGE), as well as the National Energy Research Scientific Computing Center (NERSC) (https://ror.org/05v3mvq14), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

Agarwal, P. K. 2017, in A Journey Through Discrete Mathematics: A Tribute to Jiří Matoušek, eds. M. Loebl, J. Nešetřil, & R. Thomas (Cham: Springer International Publishing), 1
Alexandrescu, A. 2017, in Leibniz International Proceedings in Informatics (LIPIcs), 75, 16th International Symposium on Experimental Algorithms (SEA 2017), 24, eds. C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, & R. Raman (Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik), 24:1
Alonso, D. 2012, ArXiv e-prints [arXiv:1210.1833]
Bentley, J. L. 1975, Commun. ACM, 18, 509
Bernardeau, F., Colombi, S., Gaztañaga, E., & Scoccimarro, R. 2002, Phys. Rep., 367, 1
Blumofe, R. D., & Leiserson, C. E. 1999, J. ACM, 46, 720
Chandler, D. 1987, Introduction to Modern Statistical Mechanics (Oxford University Press)
Chhugani, J., Kim, C., Shukla, H., et al. 2012, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12 (Washington, DC, USA: IEEE Computer Society Press)
Curtin, R. R., Cline, J. R., Slagle, N. P., et al. 2013a, J. Mach. Learn. Res., 14, 801
Curtin, R. R., March, W. B., Ram, P., et al. 2013b, ArXiv e-prints [arXiv:1304.4327]
da Fonseca, G. D., & Mount, D. M. 2010, Comput. Geometry, 43, 434
Dawson, K. S., Schlegel, D. J., Ahn, C. P., et al. 2013, AJ, 145, 10
Dawson, K. S., Kneib, J.-P., Percival, W. J., et al. 2016, AJ, 151, 44
de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. 2008, Computational Geometry: Algorithms and Applications, 3rd edn. (Berlin, Heidelberg: Springer), 386

DESI Collaboration (Aghamousa, A., et al.) 2016, ArXiv e-prints [arXiv:1611.00036]
Dolatshah, M., Hadian, A., & Minaei-Bidgoli, B. 2015, ArXiv e-prints [arXiv:1511.00628]
Dolence, J., & Brunner, R. J. 2008, in The 9th LCI International Conference on High-Performance Clustered Computing
Donoso, E. 2019, MNRAS, 487, 2824
Fog, A. 2022, The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers
Friedman, J. H., Bentley, J. L., & Finkel, R. A. 1977, ACM Trans. Math. Softw., 3, 209
Gärtner, B. 1999, in Algorithms – ESA' 99, ed. J. Nešetřil (Berlin, Heidelberg: Springer), 325
Golub, G., & Van Loan, C. 2013, Matrix Computations, Johns Hopkins Studies in the Mathematical Sciences (Johns Hopkins University Press)
Hearin, A. P., Campbell, D., Tollerud, E., et al. 2017, AJ, 154, 190
Hornbeck, H. 2020, ArXiv e-prints [arXiv:2001.09253]
Intel Corporation 2022, Intel 64 and IA-32 Architectures Optimization Reference Manual
Jarvis, M. 2015, Astrophysics Source Code Library [record ascl:1508.007]
Landy, S. D., & Szalay, A. S. 1993, ApJ, 412, 64
Larsson, T. 2008, in Linköping Electronic Conference Proceedings, 34, Proceedings of the Annual SIGRAD Conference, Stockholm, 27
Lueker, G. S. 1978, in 19th Annual Symposium on Foundations of Computer Science (sfcs 1978), 28
March, W. B., Connolly, A. J., & Gray, A. G. 2012, in Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12 (New York, NY, USA: Association for Computing Machinery), 1478
Moore, A. W. 2000, in Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence, UAI'00 (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.), 397
Moore, A. W., Connolly, A. J., Genovese, C., et al. 2001, in Mining the Sky, eds. A. J. Banday, S. Zaroubi, & M. Bartelmann, 71
Omohundro, S. M. 1989, Five Balltree Construction Algorithms, Tech. Rep. TR-89-063, International Computer Science Institute
Pedregosa, F., Varoquaux, G., Gramfort, A., et al. 2011, J. Mach. Learn. Res., 12, 2825
Peebles, P. J. E., & Hauser, M. G. 1974, ApJS, 28, 19
Pen, U.-L., Zhang, T., van Waerbeke, L., et al. 2003, ApJ, 592, 664
Philcox, O. H. E., Slepian, Z., Hou, J., et al. 2022, MNRAS, 509, 2457
Ponce, R., Cárdenas-Montes, M., Rodríguez-Vázquez, J. J., Sánchez, E., & Sevilla, I. 2012, in Astronomical Data Analysis Software and Systems XXI, eds. P. Ballester, D. Egret, & N. P. F. Lorente, ASP Conf. Ser., 461, 73
Reid, B., Ho, S., Padmanabhan, N., et al. 2016, MNRAS, 455, 1553
Ritter, J. 1990, in Graphics Gems, ed. A. S. Glassner (San Diego: Morgan Kaufmann), 301
Rohin, Y. 2018, Astron. Comput., 25, 149
Sinha, M., & Garrison, L. H. 2020, MNRAS, 491, 3022
Sleator, D. D., & Tarjan, R. E. 1985, J. ACM, 32, 652
Slepian, Z., & Eisenstein, D. J. 2015, MNRAS, 454, 4142
Springel, V. 2005, MNRAS, 364, 1105
Szapudi, I., & Szalay, A. S. 1997, ArXiv e-prints [arXiv:astro-ph/9704241]
Virtanen, P., Gommers, R., Oliphant, T. E., et al. 2020, Nat. Methods, 17, 261
Welzl, E. 1991, in New Results and New Trends in Computer Science, ed. H. Maurer (Berlin, Heidelberg: Springer Berlin Heidelberg), 359
Zhang, L. L., & Pen, U.-L. 2005, New A, 10, 569

Table A.1. Specifications of CPUs on the computing nodes used for the benchmarks.

CPU name	# of sockets	# of threads	AVX2	AVX-512
Haswell[16]	2	64	Yes	No
Broadwell[17]	2	20	Yes	No
Knights Landing[18]	1	272	Yes	Yes
Cascade Lake[19]	2	36	Yes	Yes
Rome[20]	2	128	Yes	No
Milan[21]	2	256	Yes	No

Appendix A: Benchmark specifications

We list the node specifications with different CPU architectures used for the benchmarks in this work in Table A.1. The CPU scaling governor are all set to 'performance', with the turbo boost enabled. We relied on the `gcc` compiler[22] for all our tests, with the compilation flags `-O3` and `-march=native` always enabled. For the Haswell and Knights Landing nodes, the compiler version was 7.5.0, while for all the other nodes, the version of `gcc` was 11.2.0. For tests with MPI, we made use of the Open MPI library[23] version 4.1.2, with processes bound to CPU cores.

For all benchmarks[24] in this work, each program was independently run 12 times. The execution time is then reported as the averaged cost of ten runs after excluding the longest and shortest cases. All programs were run with entire nodes to avoid resource contention from external jobs.

Appendix B: Pair-counting algorithm based on regular grids

Algorithm 4 presents the pair-counting routine used for the benchmarks in Sect. 2.1. We skipped empty cells without attempting to access the associate data points because the number of points belonging to each cell is stored as the 'data summary' of our architecture shown in Fig. 1. This reduces the chance of cache misses, especially for small cells. In addition, to speed up the evaluation of C', we pre-computed the indices of all cells within the query range with respect to a reference cell and saved the offsets of indices. Thus, C' can easily be obtained by adding offsets to the index of cell c.

[16] https://ark.intel.com/content/www/us/en/ark/products/81060/intel-xeon-processor-e52698-v3-40m-cache-2-30-ghz.html
[17] https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e52630-v4-25m-cache-2-20-ghz.html
[18] https://ark.intel.com/content/www/us/en/ark/products/94035/intel-xeon-phi-processor-7250-16gb-1-40-ghz-68-core.html
[19] https://ark.intel.com/content/www/us/en/ark/products/192443/intel-xeon-gold-6240-processor-24-75m-cache-2-60-ghz.html
[20] https://www.amd.com/en/product/8761
[21] https://www.amd.com/en/product/10906
[22] https://gcc.gnu.org/
[23] https://www.open-mpi.org/
[24] The benchmark codes for different data structures and histogram update algorithms are publicly available at https://github.com/cheng-zhao/FCFC/tree/main/benchmark.

Algorithm 4 PAIRCOUNT_GRID $(C, \mathcal{S}, \mathcal{H})$

Input: a list C with all grid cells, the separation range $\mathcal{S}$ of interest, and the histogram $\mathcal{H}$ for storing pair counts.
1: **for all** non-empty cell $c \in C$ **do**
2: $\quad C' \leftarrow \{c' \in C \mid \textsc{DistanceRange}(c, c') \subseteq \mathcal{S}\}$
3: $\quad$ **for all** non-empty cell $c' \in C'$ **do**
4: $\quad\quad$ **for all** points $\boldsymbol{p}_1$ in c, points $\boldsymbol{p}_2$ in c' **do**
5: $\quad\quad\quad d \leftarrow \textsc{Distance}(\boldsymbol{p}_1, \boldsymbol{p}_2)$
6: $\quad\quad\quad$ **if** $d \in \mathcal{S}$ **then** update histogram $\mathcal{H}$ with d **end if**
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

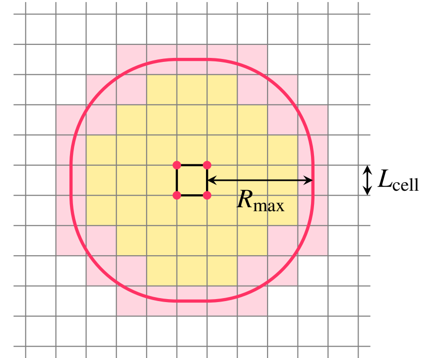

Fig. C.1. Grid cells to be visited (coloured areas) for a reference cell (black square) and an isotropic query range with a radius of $R_{\rm max}$. Yellow regions indicate cells that are entirely inside the query range, and pink zones denote cells that intersect the boundary of the query range, which is shown in red. The side length of every cell is denoted by $L_{\rm cell}$.

Appendix C: Complexities of pair-counting algorithms based on different data structures

We analysed the complexity of pair-counting processes based on different data structures in a simplified case, in which the data points were distributed uniformly in a 3D periodic cubic box with a side length of $L_{\rm box}$, and the distance range of interest was given by $[0, R_{\rm max})$, with $R_{\rm max} \ll L_{\rm box}$. This is a realistic and interesting scenario in practice because the most challenging datasets for pair counting are generally from large periodic simulations.

The complexity of a pair-counting algorithm consists of two parts: (1) $N_{\rm node}$, the number of tree nodes or grid cells that are visited (2) $N_{\rm pair}$, the number of pairs of data points that are examined. Apparently, in the small-node/cell limit, $N_{\rm node}$ dominates the complexity, while $N_{\rm pair}$ is more relevant for large nodes or cells. We then estimated both $N_{\rm node}$ and $N_{\rm pair}$ for different data structures.

C.1. Regular grids

For cubic datasets, it is obvious that the cells of regular grids are best cubes. In this case, the query range and grid cells to be visited for a single reference cell are illustrated in Fig. C.1. Given the edge length $L_{\rm cell}$ of all grid cells, the number of cells to be visited for any given reference cell, denoted by $\mathfrak{N}_{\rm tot}$, depends solely on $\hat{L}_{\rm cell} \equiv L_{\rm cell}/R_{\rm max}$, as it does not change when $L_{\rm cell}$ and $R_{\rm max}$ are simultaneously rescaled with the same factor. $\mathfrak{N}_{\rm tot}$ can be decomposed into two components:

$$\mathfrak{N}_{\rm tot} = \mathfrak{N}_{\rm inner} + \mathfrak{N}_{\rm edge}, \tag{C.1}$$

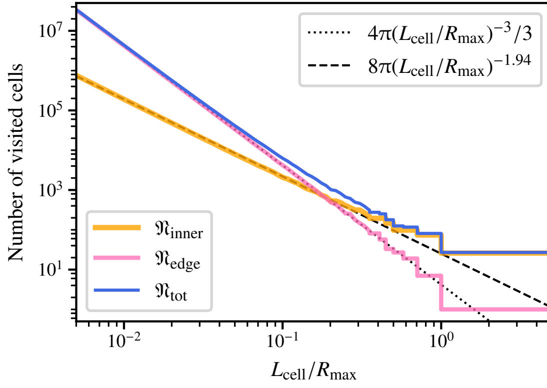

Fig. C.2. Number of regular grid cells to be visited for each reference cell with a side length L_{cell}, given a spherical range searching with the maximum distance of R_{max}, and a periodic box that is sufficiently large. Here, $\mathfrak{N}_{tot} = \mathfrak{N}_{inner} + \mathfrak{N}_{edge}$, where $\mathfrak{N}_{inner}$ and $\mathfrak{N}_{edge}$ indicate the number of cells that are fully and partially inside the query range, which correspond to the yellow and pink regions in Fig. C.1, respectively. The dashed and dotted black lines show analytical formulae that fit the numerical results in the small-cell limit well.

where $\mathfrak{N}_{inner}$ and $\mathfrak{N}_{edge}$ indicate the numbers of cells that are fully and partially inside the query range, as shown in yellow and pink in Fig. C.1, respectively. These numbers can be evaluated numerically, and the results are shown in Fig. C.2, together with two empirical analytical formulae that fit $\mathfrak{N}_{inner}$ and $\mathfrak{N}_{edge}$ well in the small-cell limit. In particular, when $\hat{L}_{cell} \lesssim 0.5$,

$$\mathfrak{N}_{inner}(\hat{L}_{cell}) \approx 4\pi\,\hat{L}_{cell}^{-3}/3, \tag{C.2}$$

$$\mathfrak{N}_{edge}(\hat{L}_{cell}) \approx 8\pi\,\hat{L}_{cell}^{-1.94}. \tag{C.3}$$

In contrast, $\mathfrak{N}_{inner}$ and $\mathfrak{N}_{edge}$ are constants of 1 and 26, respectively, when $\hat{L}_{cell} \geq 1$.

Given N data points that are uniformly distributed, with a number density of $\rho = N/L_{box}^3$, the number of pair separations to be computed for the full dataset is a function of $\mathfrak{N}_{tot}(\hat{L}_{cell})$,

$$N_{pair}^{grid} = \rho\,\mathfrak{N}_{tot}(\hat{L}_{cell})L_{cell}^3 \cdot N = N^2(L_{cell}/L_{box})^3\mathfrak{N}_{tot}(\hat{L}_{cell}). \tag{C.4}$$

Ideally, the number of points in cells that are entirely inside the query range can be reported directly. This is impractical for real-world pair-counting problems with multiple separation bins, however. Therefore, we processed individual points of these cells in any case (see Sect. 2.5 for more discussions). The total number of cells that are visited can be estimated by

$$N_{node}^{grid} = \mathfrak{N}_{tot}(\hat{L}_{cell}) \cdot \min\{(L_{box}/L_{cell})^3, N\}, \tag{C.5}$$

where $(L_{box}/L_{cell})^3$ is the number of all grid cells, and the term $\min(L_{box}^3 L_{cell}^{-3}, N)$ indicates an approximation of the number of cells containing data, which reduces to N in the small-cell limit because most cells are empty in this case.

Because N_{pair}^{grid} and N_{node}^{grid} dominate the computational cost at the large- and small-cell ends, respectively, it is not difficult to find that the complexity of the grid-based pair-counting algorithm scales with $O(L_{cell}^3)$ when $L_{cell} \gtrsim R_{max}$, while it is $O(L_{cell}^{-3})$ if $L_{cell} \ll R_{max}$. These relations are consistent with the measurements shown in Fig. 3, where the best-fitting $(aN_{pair}^{grid} + bN_{node}^{grid})$ curves are also illustrated. Here, a and b are constants obtained from least-squares fits to the measurements with all different configurations. The agreement between the data and the model is good in general, especially for the large- and small-cell ends.

C.2. *k*-d tree

When constructing the k-d tree upon a periodic cubic box with uniform data distribution, the subdivided volumes after space partition are expected to be small cubes with a cell size of $(n_{leaf}/\rho)^{1/3}$. In this case, the number of k-d tree leaf nodes with the partitioned volumes intersecting the query boundaries is close to that of regular grids, which are shown as pink regions in Fig. C.1, but with some important differences. Firstly, the query range given a reference k-d tree node is slightly smaller than that of regular grids, as we measure distances between nodes using their minimum AABBs, which are generally smaller than the corresponding grid cells. Similarly, it is possible that the AABB of a node does not cross the query range boundary, even if the corresponding subdivided volume intersects with it. For instance, when there is only a single point on each leaf node, no leaves intersect with the boundary of the query range as the AABBs reduce to the points, which can only be inside or outside the range. For both reasons, the number of leaf nodes with their minimum AABBs intersecting the query boundary is smaller than the prediction of $\mathfrak{N}_{edge}(\hat{n}_{leaf}^{1/3})$, and may be modelled with an additional term,

$$\mathfrak{N}_{leaf} = \eta(n_{leaf})\,\mathfrak{N}_{edge}(\hat{n}_{leaf}^{1/3}), \tag{C.6}$$

where

$$\hat{n}_{leaf} \equiv n_{leaf}\,\rho^{-1}R_{max}^{-3}. \tag{C.7}$$

When R_{max} is large, the reduction of the query range is not significant. In this scenario, η is dominated by the fact that the AABBs of leaf nodes are less likely to intersect with the query range boundaries than regular grids. The lower limit of η is given by the ratio of the AABB volume to that of a grid cell, which is $[(n_{leaf} - 1)(n_{leaf} + 1)]^3$ for uniformly distributed points. For simplicity, we assume

$$\eta(n_{leaf}) \approx \left(\frac{n_{leaf} - 1}{n_{leaf}}\right)^3, \tag{C.8}$$

which fulfils the condition $\eta(1) = 0$, and approaches 1 when n_{leaf} is sufficiently large.

Since the tree structure is self-similar, the number of node separation evaluations, $N_{node}^{k\text{-d}}$, can be solved recursively. For instance, for a k-d tree that contains n_{leaf} data points per leaf node at most, with $n_{leaf} > 1$, further dividing the leaves into two parts is as if a new tree with a leaf capacity of $(n_{leaf}/2)$ were constructed. Moreover, if the separation range between two original leaf nodes intersects the boundary of the query range, the separations between their two children are checked for pair counting with the new tree. Consequently, we have

$$N_{node}^{k\text{-d}}(\frac{n_{leaf}}{2}) - N_{node}^{k\text{-d}}(n_{leaf}) = \frac{4N}{n_{leaf}} \cdot \mathfrak{N}_{leaf}(\hat{n}_{leaf}^{1/3}), \tag{C.9}$$

where (N/n_{leaf}) is an approximation of the total number of leaf nodes for the original k-d tree. Since the number of visited node is only significant when there are many nodes, in which case $\hat{n}_{leaf}$ is small, we consider here only the small-cell end of $\mathfrak{N}_{leaf}$. Given also Eqs. (C.3), (C.6), and (C.8), the right-hand side of this recursive equation is a Laurent polynomial of n_{leaf}, which yields the following analytical solution:

$$N_{node}^{k\text{-d}} \propto \left(15 - \frac{18}{n_{leaf}} + \frac{8.3}{n_{leaf}^2} - \frac{1.3}{n_{leaf}^3}\right) \cdot \frac{\pi N^{1.65} R_{max}^{1.94}}{n_{leaf}^{1.65} L_{box}^{1.94}}. \tag{C.10}$$

When considering the number of pair separations that are evaluated during the dual-tree pair-counting process, we can count the number of leaf nodes that are not entirely outside the query range, even though the algorithm may terminate without visiting all leaves. This is because for each node of the tree, the associated dataset is the union of those on all the corresponding descendant leaf nodes. Therefore, the total number of pair separations computed for the full dataset is

$$\begin{aligned} N_{\rm pair}^{k\text{-d}} &= n_{\rm leaf}\left(\mathfrak{N}_{\rm inner} + \mathfrak{N}_{\rm leaf}\right) \cdot N \\ &= n_{\rm leaf} N \left[\mathfrak{N}_{\rm inner}(\hat{n}_{\rm leaf}^{1/3}) + (n_{\rm leaf} - 1)^3 n_{\rm leaf}^{-3}\, \mathfrak{N}_{\rm edge}(\hat{n}_{\rm leaf}^{1/3})\right]. \end{aligned} \tag{C.11}$$

Therefore, in the large-node limit, the complexity of the pair-counting algorithm based on the k-d tree scales with $O(n_{\rm leaf}^{0.35})$, while it is a Laurent polynomial of $n_{\rm leaf}$ for small tree nodes (see Eq. (C.10)). The best-fitting $(aN_{\rm pair}^{k\text{-d}} + bN_{\rm node}^{k\text{-d}})$ curves are shown in Fig. 5, where the constants a and b are obtained by least-squares fits to all measurements. The theoretical complexity agrees remarkably well with the data for almost all cases.

Since the ball tree is a similar data structure as the k-d tree, especially for cubic periodic boxes, the derivations for the k-d tree should work for the ball tree as well, but the relation in Eq. (C.8) may be slightly different due to a different representation of the node bounding volume. We then fit the theoretical complexity from Eqs. (C.10) and (C.11) to the measurements shown in Fig. 7, and the agreement is excellent.

Appendix D: Random sampling of squared pair separations

For periodic boxes, Eq. (8) shows that the total number of pairs with separations below $R_{\rm max}$ scales with $R_{\rm max}^3$. In this case, the probability distribution function (PDF) of pair separations satisfies

$$P(s) \propto s^2. \tag{D.1}$$

The goal is to reproduce this distribution with uniform random sequences in the range $[0, 1)$, which are the direct outputs of most random number generation algorithms in practice. Denoting this random number as x, we have then $P(x) = 1$, and we need to find the relation $s(x)$, such that Eq. (D.1) holds.

When transforming a variable x to y, with $y(x)$ being monotonic, the PDFs of x and y satisfy

$$P_y(y) = P_x(x(y)) \left|\frac{{\rm d}x}{{\rm d}y}\right|. \tag{D.2}$$

Given this relation, we find

$$s(x) \propto x^{1/3}. \tag{D.3}$$

In other words, to sample randomly squared pair separations in the range $[0, 1)$, we barely need to compute $x^{2/3}$ for uniform random variables x generated in the same range. To extend the maximum separation to $R_{\rm max}$, the conversion is simply

$$s^2 = x^{2/3} \cdot R_{\rm max}^2. \tag{D.4}$$

Appendix E: Quick guide to `FCFC`

As of version 1.0.1, `FCFC` supports the following 2PCFs: $\xi(s)$, $\xi(s,\mu)$, $\xi(\sigma,\pi)$, $\xi_\ell(s)$, and $w_{\rm p}(\sigma)$, where

$$\xi_\ell(s) = (2\ell + 1) \int_0^1 \xi(s,\mu)\mathcal{L}_\ell(\mu)\,{\rm d}\mu, \tag{E.1}$$

$$w_{\rm p}(\sigma) \approx 2 \int_0^{\pi_{\rm max}} \xi(\sigma,\pi)\,{\rm d}\pi. \tag{E.2}$$

Here, $\mathcal{L}_\ell$ denotes the Legendre polynomial of order ℓ. The correlation function estimator is user-defined and can be arbitrary. It accepts both periodic and non-periodic input catalogues in ASCII text, FITS, and HDF5 formats. In particular, the supports of FITS and HDF5 formats require the `CFITSIO`[25] and `HDF5`[26] libraries. Except for the optional libraries for file formats, as well as the OpenMP and MPI libraries for the corresponding parallelisms, `FCFC` does not depend on any other external library. It is fully compliant with the ISO C99[27] and IEEE POSIX.1-2008[28] standards. Therefore, `FCFC` can be easily compiled with most modern C compilers and operating systems.

The specifications of a pair-counting task can be passed to `FCFC` via a configuration file or command-line options. We introduce here a few relevant settings for different practical scenarios. For instance, the 2PCF of a periodic simulation catalogue is generally measured using the Peebles–Hauser estimator (Peebles & Hauser 1974),

$$\xi = {\rm DD}/{\rm RR} - 1, \tag{E.3}$$

where RR can be computed analytically. In this case, the relevant configurations of `FCFC` can be

```
CATALOG       = sim_data.txt
CATALOG_LABEL = D
PAIR_COUNT    = DD
CF_ESTIMATOR  = DD / @@ - 1
```

Here, `CATALOG` denotes the filename of the input catalogue, and `CATALOG_LABEL` sets the label of this catalogue. `PAIR_COUNT` defines the sources of catalogues forming pairs, so '`DD`' indicates auto pair counts of the catalogue '`D`'. Finally, `CF_ESTIMATOR` sets the correlation function estimator, where '`@@`' denotes the analytical RR pair counts. Apparently, the estimator is basically set in the same form as Eq. (E.3).

Similarly, given observational luminous red galaxy (LRG) and emission line galaxy (ELG) samples with the filenames '`LRG_data.txt`' and '`ELG_data.txt`', together with the corresponding random catalogues '`LRG_rand.txt`' and '`ELG_rand.txt`', respectively, the auto 2PCFs of LRGs and ELGs as well as the cross 2PCFs between LRGs and ELGs can be computed at once with the following `FCFC` settings:

```
CATALOG       = [LRG_data.txt, LRG_rand.txt,
                 ELG_data.txt, ELG_rand.txt]
CATALOG_LABEL = [L, R, E, S]
PAIR_COUNT    = [LL, LR, RR, EE, ES, SS,
                 LE, LS, RE, RS]
CF_ESTIMATOR  = [(LL - 2 * LR + RR) / RR,
                 (EE - 2 * ES + SS) / SS,
                 (LE - LS - RE + RS) / RS]
```

[25] https://heasarc.gsfc.nasa.gov/fitsio/
[26] https://www.hdfgroup.org/solutions/hdf5/
[27] https://www.iso.org/standard/29237.html
[28] https://ieeexplore.ieee.org/document/4694976

The Szapudi–Szalay estimator (Szapudi & Szalay 1997) is used for the cross correlation here,

$$\xi^{\times} = (\mathrm{D_L D_E} - \mathrm{D_L R_E} - \mathrm{R_L D_E} + \mathrm{R_L R_E})/\mathrm{R_L R_E}, \tag{E.4}$$

where the subscripts 'L' and 'E' denote the catalogues for LRGs and ELGs, respectively.

Because the `libast` library[29] is embedded in `FCFC`, human-readable expressions can be used not only for the correlation function estimators, but also for numerical values read from the input catalogues. For example, to compute auto pair counts of the BOSS DR12 combined sample[30] in the redshift range $0.2 < z < 0.5$, we have to use weights to correct for systematics and reduce variance, with the total weight given by (Reid et al. 2016)

$$w_{\mathrm{tot}} = w_{\mathrm{FKP}}\, w_{\mathrm{sys}}\, (w_{\mathrm{cp}} + w_{\mathrm{noz}} - 1), \tag{E.5}$$

where w_{FKP}, w_{sys}, w_{cp}, and w_{noz} indicate the `WEIGHT_FKP`, `WEIGHT_SYSTOT`, `WEIGHT_CP`, and `WEIGHT_NOZ` columns of the data catalogue, respectively. In this case, `FCFC` can be configured with

```
POSITION  = [${RA}, ${DEC}, ${Z}]
SELECTION = ${Z} > 0.2 && ${Z} < 0.5
WEIGHT    = ${WEIGHT_FKP} * ${WEIGHT_SYSTOT} *
            (${WEIGHT_CP} + ${WEIGHT_NOZ} - 1)
```

Here, `${X}` indicates the column `X` of the input FITS catalogue.

For more details on the configurations of `FCFC`, we refer to the documentation of the toolkit[31].

[29] https://github.com/cheng-zhao/libast

[30] https://data.sdss.org/sas/dr12/boss/lss/galaxy_DR12v5_CMASSLOWZTOT_North.fits.gz

[31] https://github.com/cheng-zhao/FCFC/blob/main/README.md