IGD: high-performance search for large-scale genomic interval datasets

Jianglin Feng[1,✉] and Nathan C. Sheffield[1-4,✉]

[1]Center for Public Health Genomics, University of Virginia
[2]Department of Public Health Sciences, University of Virginia
[3]Department of Biomedical Engineering, University of Virginia
[4]Department of Biochemistry and Molecular Genetics, University of Virginia
✉ Correspondence: jf.xtable@gmail.com
✉ Correspondence: nsheffield@virginia.edu

Summary: Databases of large-scale genome projects now contain thousands of genomic interval datasets. These data are a critical resource for understanding the function of DNA. However, our ability to examine and integrate interval data of this scale is limited. Here, we introduce the integrated genome database (IGD), a method and tool for searching genome interval datasets more than three orders of magnitude faster than existing approaches, while using only one hundredth of the memory. IGD uses a novel linear binning method that allows us to scale analysis to billions of genomic regions. **Availability**: `https://github.com/databio/IGD`

INTRODUCTION

Genome and epigenome data are frequently represented as *genomic intervals*, defined by a chromosome plus *start* and *end* coordinates. Because this data type is so common, interval-based comparisons are fundamental to genomic analysis[1–4]. The most basic operation on interval data is computing the intersections of two interval sets. Several approaches have been developed to do this efficiently, such as the the nested containment list (NCList)[5], the interval tree[6], the R-tree[7], and the Augmented Interval List (AIList)[8], and many tools exploiting these data structures exist[1,9–12].

A more advanced task with intervals is *interval database search*, which requires comparing a query interval set against a database of multiple interval sets. This task extends the file-to-file comparison to a file-to-database comparison. To do this, the file-to-file approaches can be applied in a loop, but this does not take advantage of integrating across interval sets in the database, which can be done by creating a combined index that spans individual interval sets. This is the approach taken by GIGGLE, which is much faster for searching a large database than a naive file-to-file loop[3]. Here, we present a new approach called the *integrated genome database (IGD)*, which uses a novel indexing method called *linear binning*. We show that the IGD approach achieves substantially better performance than all existing approaches.

METHODS

Linear binning. An interval set has two different orders: the order of start values and the order of end values. To effectively process interval data, multi-level binning methods are usually used[2,7,10], where intervals are stored in bins of different sizes that are large enough to contain the intervals. In contrast, the linear binning approach used by IGD is a single-level binning (Fig. 1a). The whole genome is divided into equal-sized bins and intervals are put into bins they intersect. The intervals are not required to fit into the bins; instead, intervals that cross bin boundaries are stored in multiple bins. This duplication introduces two challenges: increase in size of the database, and a possibility of multiple counting. The number of duplicates can be adjusted by the selecting bin size, which therefore adjusts the amount of extra data required for linear binning. In our tests, the database size increases by only 5% to 20% for a bin size of 16,384 (2^{14}), which is much more efficient than multi-level binning or tree-based methods (see results in the next section). The multiple counting challenge, which occurs only when both the query and the data interval cross the same bin boundary, can be neatly solved by setting a rule: if the query crosses the left boundary of the bin, then any region in the bin that also crosses the left boundary will be skipped. In figure 1a, query q_2 intersects both interval $2a$ (which belongs to bin i) and interval $2b$ (bin $i+1$); since q_2 does not cross the left boundary of bin i, region $2a$ will count; but q_2 crosses the left boundary of bin $i+1$, so the intersection with region $2b$ will not count.

With the challenges of multiple counting and database size neatly solved, linear binning makes interval data behave similar to a dataset with a single order. More importantly, since a bin contains all regions that intersect the bin, the processing of intervals in each bin is independent. In contrast, the nodes of the tree-based or multi-level approaches are not independent and queries typically require traversing nodes or bins of different levels.

IGD data structure and query. IGD stores the bin data linearly and maintains a header that contains the location of the bins and the number of IGD data elements in each bin. Each IGD data element contains the index of the original dataset, the original genomic region ($start$ and end coordinates), and a value (signal level). For compatibility to other existing search tools, IGD also supports a data element type without the signal value. Since the region data is indexed linearly, a query to the database simply determines the index of the bin that the query region intersects and then loads only that bin into memory and searches within the bin. If a query crosses any bin boundaries, IGD loads and searches those bins one by one. Compared to file-to-file approaches, IGD does not need to load all the files and does not need to search all regions in every file, making IGD much more efficient (Fig. 1b).

Regions in a bin are presorted by the region $start$, so the search inside a bin can be done by first using a binary search to find the right-most interval (intervals with $start$ larger than or equal to the query end will not intersect the query) then scanning to the left (smaller $start$). This is very efficient because the bin size, *e.g.*, 16,384 bp, is small. Alternatively, the general search algorithm of AIList[8] may be used, but it requires preprocessing to add an extra

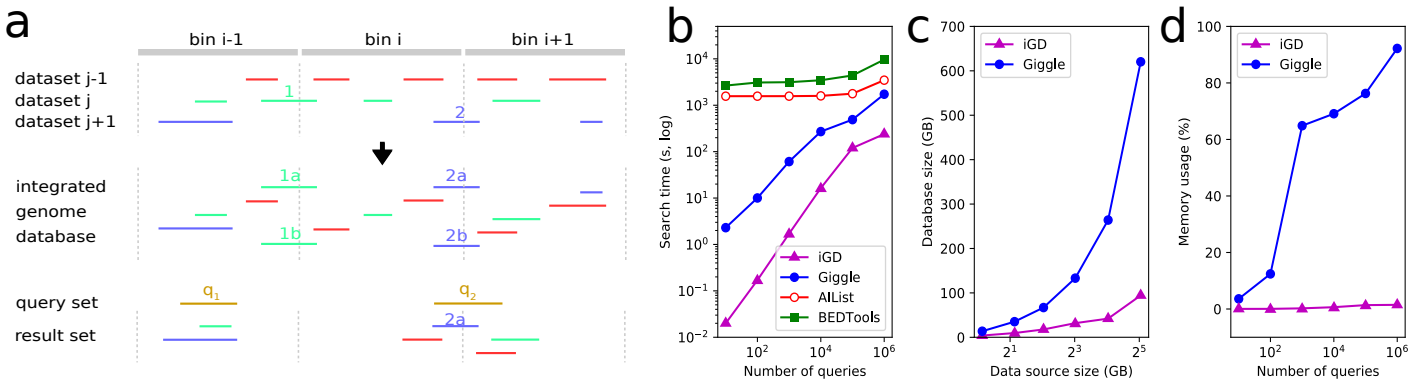

Fig. 1: a) Linear-binning indexing. The genome is divided into equal-sized bins and regions are put into any bins they intersect. Regions that cross boundaries are stored in multiple bins. For example, region 1 is placed in both bin $i-1$ and bin i. Regions in each bin are then sorted by their starts. A search query interval $q2$ crosses the boundary of bin i and bin $i+1$, so region 2 is double counted. IGD solves this by counting only $2a$ (see text); b) Performance comparisons with existing methods on 5660 interval datasets. Search times are shown for IGD, GIGGLE, AIList and BEDTools on 6 query datasets of 10^1 to 10^6 regions; c) Sizes of IGD and GIGGLE databases created from 6 different subsets of a UCSC interval set collection with input size ranging from 2^0 to 2^5 GB of compressed data; d) Peak memory usage for IGD and GIGGLE for the 6 queries. GIGGLE did not complete the query of 10^6 regions.

element (the augmenting value) for each region, which increases the storage by one third and also affects the data loading speed.

RESULTS

To benchmark the speed and memory use of the IGD approach, we assembled a test set of human query and database interval sets. All tests in this work were performed using a single core on a 2.8 GHz personal computer with 16 GB memory and an external 1TB SSD disc. We created the IGD databases with a bin size of 8,192 (2^{13}), corresponding to about 376,000 bins for human genome.

We first compared IGD to existing approaches for the database search task. For query sets with 100 or fewer intervals, IGD is four to five orders of magnitudes faster than file-to-file approaches AIList and BEDTools (Fig 1b). This is expected, as these approaches load all data from 5660 files while IGD only loads data from very few bins. GIGGLE uses a database approach based on B+ tree indexing scheme, which is faster than the file-by-file methods, but IGD is two orders of magnitude faster still. For larger query intervals, IGD is also several orders of magnitude faster than any existing approach.

In terms of disk use, the IGD index increases linearly with the size of input data source, while GIGGLE increases exponentially (Fig 1c). For the full set UCSC data source, GIGGLE uses 640GB disc space while IGD only needs 98GB. IGD is also faster in database creation: the complete database takes IGD about 2 hours to create, while GIGGLE takes more than 4 hours.

Furthermore, IGD uses nearly negligible computer memory while GIGGLE needs substantial memory even for small query sets (Fig. 1d). For example, with 1,000 query regions, IGD requires only 38 MB of memory, while GIGGLE uses 10,570 MB memory, more than 270 times as much. These results suggest that linear-binning indexing is superior to B+tree indexing in terms of database creation time, search time, and search memory use.

We have shown how IGD presents a fast alternative for interval search operations. Furthermore, the independence of IGD bins and the small memory use make IGD amenable to parallel processing, which can further increase the searching speed. In additional to calculating number of overlaps, IGD can also calculate the seqpare metric, which provides a novel way to assess similarity between region sets[13]. As the size of genomic interval data increases, we believe the speed and memory advances achieved by IGD will make it a valuable tool for fast analysis in the future.

FUNDING

This work was supported by the University of Virginia School of Medicine and the University of Virginia 4-VA program.

REFERENCES

1. Sheffield NC & Bock C. *Bioinformatics* **32,** 587–589 (2016).

2. Li H. *Bioinformatics* **27,** 718–719 (2011).

3. Layer RM *et al. Nature Methods* **15,** 123–126 (2018).

4. Jalili V *et al. IEEE Transactions on Knowledge and Data Engineering* **31,** 2008–2021 (2019).

5. Alekseyenko AV & Lee CJ. *Bioinformatics* **23,** 1386–1393 (2007).

6. Cormen TH *et al.* (2001).

7. Kent WJ *et al. Genome Res.* **12,** 996–1006 (2002).

8. Feng J *et al. Bioinfomatics* **btz407,** (2019).

9. Richardson JE. *Journal of Computational Biology* **13,** 1457–1464 (2006).

10. Quinlan AR & Hall IM. *Bioinformatics* **26,** 841–842 (2010).

11. Neph S *et al. Bioinfomatics* **28,** 1919–1920 (2012).

12. Dozmorov MG. *Bioinformatics* **33,** 3323–3330 (2017).

13. Feng SC *et al. bioRxiv* (2020). doi:10.1101/2020.04.05.026732