hictk: blazing fast toolkit to work with .hic and .cool files

Roberto Rossini[1] and Jonas Paulsen[1,2*]

1. Department of Biosciences, University of Oslo, 0316 Oslo, Norway; 2. Centre for Bioinformatics, Department of Informatics, University of Oslo, 0316 Oslo, Norway
*Correspondence: jonas.paulsen@ibv.uio.no

Abstract

Motivation

Hi-C is gaining prominence as a method for mapping genome organization. With declining sequencing costs and a growing demand for higher-resolution data, this trend is expected to persist. Therefore, efficient tools for processing Hi-C datasets at different resolutions are crucial. Over the past decade, the .hic and Cooler file formats have become the de-facto standard to store interaction matrices produced by Hi-C experiments in binary format. Yet, interoperability between the two formats is lacking, making it unnecessarily difficult to convert between the two formats, or writing applications that can natively process both formats. Therefore, there is a pressing need for high-performance tools that can efficiently handle both file formats.

Results

We developed hictk, a toolkit that can transparently operate on .hic and .cool files with excellent performance. The toolkit is written in C++ and consists of a C++ library with Python bindings as well as CLI tools to perform common operations directly from the shell, including converting between .hic and .mcool formats. We benchmark the performance of hictk and compare it with other popular tools and libraries. We conclude that hictk significantly outperforms existing tools while providing the flexibility of natively working with both file formats without code duplication.

Availability and Implementation

The hictk library, Python bindings and CLI tools are released under the MIT license as a multi-platform application available at github.com/paulsengroup/hictk. Pre-built binaries for Linux and macOS are available on bioconda. Python bindings for hictk are available on GitHub at github.com/paulsengroup/hictkpy.

Introduction

Chromosome Conformation Capture-based methods (3C) are invaluable tools for studying genome- and chromatin organization. High-throughput 3C-techniques such as Hi-C involve probing and quantifying pairwise proximities throughout entire genomes' three-dimensional (3D) structure. Recently, there has been a remarkable increase in the production of Hi-C data both for the purpose of genome assembly (Burton *et al.*, 2013; Kaplan and Dekker, 2013; Selvaraj *et al.*, 2013) and for high-resolution mapping of genome 3D organization (Lieberman-Aiden *et al.*, 2009). With the steady decrease of sequencing costs, this trend is unlikely to stop, putting to the test the performance of available computational tools to process and analyze Hi-C data.

Sequencing data produced by Hi-C experiments are usually processed with specialized pipelines from ENCODE (github.com/ENCODE-DCC/hic-pipeline), 4DNucleome (github.com/4dn-dcic/docker-4dn-hic), nf-core/hic (doi.org/10.5281/zenodo.7994878) and Distiller (doi.org/10.5281/zenodo.7309110). The main file output by these pipelines stores a matrix-representation of all pairwise HiC-interactions within and between the reference chromosomes.

These interaction matrices are typically very large (9.5 trillion values at 1 kbp resolution in the human genome) and thus cannot be represented as dense matrices in plain-text format.

Two popular file formats are currently used to efficiently store this kind of data: Cooler (Abdennur and Mirny, 2019) and .hic (Durand, Robinson, *et al.*, 2016). Both formats represent interaction matrices as sparse, compressed and chunked matrices. However, they do so in substantially different ways, requiring specialized libraries for reading or writing each format. As a consequence, application developers need to write two versions of the same code to read data from files: one for .hic files and another for Cooler files. Yet, responsibility of format conversion is usually placed on the users of the tools. Notably, converting between the two formats is not trivial or computationally efficient, as the only reliable way is to first convert input files to an intermediate text representation, and then create the output files from this intermediate representation. This process is inefficient and error prone, as users are tasked with verifying the accuracy of the text format and ensuring proper sorting.

Files stored in the .hic format undergo processing through several tools: JuiceboxGUI (Durand, Robinson, *et al.*, 2016) serves as a visualization tool for .hic files, offering an interactive interface where users can visualize .hic matrices at different resolutions, with various normalizations and annotation tracks. JuicerTools and HiCTools (Durand, Shamim, *et al.*, 2016) facilitate the generation of .hic files, and straw (Durand, Robinson, *et al.*, 2016) functions as a versatile .hic file reader. Notably, straw is accessible as a library for multiple programming languages such as C++, Matlab, R, and Python. In contrast, JuiceboxGUI and HiCTools are Java-based applications designed to efficiently handle .hic files.

Cooler files on the other hand, are processed with the homonymous suite of tools and can be visualized using HiGlass (Abdennur and Mirny, 2019; Kerpedjiev *et al.*, 2018). The Cooler file format is based on HDF5 (www.hdfgroup.org/HDF5), while the CLI and library are written in Python. The Cooler file format specification defines three formats: single-resolution (.cool), multi-resolution (.mcool) and single-cell (.scool) Cooler files. Single-resolution Cooler files store interactions using a compressed sparse row (CSR) scheme; multi-resolution Coolers are collections of .cool files at different resolutions; single-cell Cooler files are meant to store single-cell Hi-C data and consist of collections of .cool files: one file for each cell. HiGlass is a client-server application written in JavaScript and Python. Contrary to files in Cooler format, files in .hic format can be visualized using the UCSC genome browser.

Finally, a tool named hic2cool is available to easily convert from .hic to .mcool. hic2cool is written in Python and borrows code from straw and cooler. However, hic2cool is relatively slow (see benchmarks) and is not capable of reading the latest version of .hic files (github.com/aidenlab/hic-format).

We have developed the hictk toolkit with the aim of streamlining reading, writing and converting .hic and Cooler files within the same library, and doing so with excellent performance. hictk consists of a library and CLI components. The library allows reading from both .hic and Cooler files, as well as writing to Cooler files. The CLI makes it easy to perform common operations directly from the shell, including converting files between Cooler and .hic formats.

Materials and methods

Overview (library)

hictk is implemented in C++ and was designed with computational- and memory efficiency and composability in mind. To achieve this, hictk heavily relies on iterators to lazily traverse collections of pixels. From the user perspective, there is no difference between reading interactions from a .hic, .cool or .mcool file. To read interactions for a region of interest, users first create a file object by providing the path to a file and the resolution in bp to be used. The file object implements a fetch method that takes as input several optional parameters, including a query range (e.g. chr1:0-10,000,000). The fetch method returns a PixelSelector object, providing begin() and end() methods allowing pixel traversal for the queried range (see Table 1). PixelSelector objects also support returning interaction frequencies directly as sparse or dense matrices. Iterators are implemented to strike a good balance between memory usage and performance, requiring few KBs of memory when fetching small regions on coarse matrices up to hundreds of MBs when fetching large queries from high resolution matrices while traversing millions of pixels per second (see benchmarks for more details).
The library supports reading files with integer and floating-point interaction frequencies, with and without matrix balancing using algorithms such as iterative correction and eigenvector decomposition (ICE) or Knight-Ruiz (KR) (Imakaev *et al.*, 2012; Knight and Ruiz, 2012). For .hic files, expected and observed/expected interaction frequencies are also supported. This level of flexibility is achieved through runtime polymorphism enabled by the visitor pattern (Palsberg and Jay, 1998). Beside reading interactions, the library provides a set of operations that can be used as building blocks for more complex applications, such as functions to create .cool, .mcool and .scool files, merge multiple .cool files, coarsening interactions from .hic and .cool files and more.
Unrecoverable errors are propagated through exceptions, letting the caller decide whether an application using hictk should abort.
Beside performance, one of the advantages of writing hictk in C++ is the ability to write bindings for popular scripting languages such as Python and R through e.g. pybind11 github.com/pybind/pybind11 and Rcpp (Eddelbuettel, 2013). We currently provide bindings for the Python language through the hictkpy package. In the result section we show how to update a Python application to use hictkpy instead of cooler to read contact matrices, making the application capable of reading .hic files with few changes.

Table 1: Code snippets showing how to use hictk and hictkpy to read a subset of interactions from .hic and .mcool files.

```
#include <algorithm>
#include <cstdint>

#include <fmt/format.h>
#include <hictk/file.hpp>


hictk::File file{"interactions.hic", 10'000};
const auto selector = file.fetch("chr1:10,000,000-20,000,000");

// Write pixels to stdout
std::for_each(selector.template begin<std::int32_t>(),
              selector.template end<std::int32_t>(),
              [](const auto &pixel) {
                  fmt::print("{}\n", pixel);
              });
```

```
import hictkpy

import sys


file = hictkpy.File("interactions.mcool", 10_000)
selector = file.fetch("chr1:10,000,000-20,000,000")
df = selector.to_df()

# Write pixels to stdout
df.to_csv(sys.stdout, sep="\t")
```

Overview (CLI)

The CLI interface of hictk is built around facilities provided by the library component.
The interface consists of a single executable, named hictk with several subcommands (see Table 2).

Table 2: list and brief description of subcommands available from hictk command-line interface.

Subcommand	Description
convert	Convert matrices between .hic and Cooler formats.
dump	Write interactions from .hic or Cooler files to the terminal.
load	Generate a Cooler file from pixels or pairs of interactions in text format.
merge	Merge multiple Cooler files using the same reference assembly.
validate	Validate Cooler and .hic files.
zoomify	Convert single-resolution cooler files to multi-resolution cooler files by coarsening.

hictk convert, dump and validate accept both .cool and .hic files as input, while the rest of the subcommands only support .cool, .mcool and .scool files. Conversion from .mcool to .hic is performed by wrapping JuicerTools and HiCTools.

Testing

hictk is thoroughly tested with a continuous integration pipeline (CI) which executes hictk unit tests (80%+ coverage) on Linux, macOS and Windows using different versions of GCC, Clang, Apple-Clang and MSVC.

CLI tools are further tested with 17 integration tests, covering the most common use-cases of each subcommand.

Finally, fetch functionality is automatically tested every time a pull request is merged, or weekly, with thousands of random queries to ensure that hictk and cooler return identical results for the same query.

Benchmark system and data

Benchmarks were run on a system with the following specifications:

- AMD EPYC 7742 (2x 64 cores)
- 2048 GB (32x 64 GB, RDIMM DDR4 2933 MT/s eight-channel)
- RHEL 8.8 (Linux v4.18.0-425)

Input and output files were stored on a ramdisk to avoid IO bottlenecks.

The benchmark suite is written using Nextflow DSL 2 and was executed using Nextflow v23.04.1 (Di Tommaso *et al.*, 2017) limiting parallelism to at most 32 CPU cores to ensure results are not biased by high CPU load.

The following files were downloaded from ENCODE (ENCODE Project Consortium, 2012; Luo *et al.*, 2020; Hitz *et al.*, 2023) and UCSC FTP server and were used across benchmarks:

- ENCFF447ERX: pairs of interactions for GM12878 in text format (4DN-DCIC pairs) (identifiers.org/encode:ENCFF447ERX)
- ENCFF301CUL: list of Topologically Associating Domains (TADs) for GM12878 (identifiers.org/encode:ENCFF301CUL)
- hg38.chrom.sizes: list of chromosomes for hg38 (Lander *et al.*, 2001)

Interactions from ENCFF447ERX were used to generate interaction matrices in .hic v8, .hic v9 and .mcool formats using JuicerTools v1.22.01 (Durand, Shamim, *et al.*, 2016), HiCTools v3.30.00 (Durand, Shamim, *et al.*, 2016) and cooler v0.9.2 (Abdennur and Mirny, 2019) respectively. This dataset was selected as it is one of the largest dataset publicly available, with close to 7 billion interactions (see Table 3). Each file contains 13 resolutions ranging from 10 bp to 2.5 Mbp. We decided to run our benchmarks with resolutions up to 10 bp, as this is the highest resolution available in .hic files released as part of ENCODE4 (www.encodeproject.org/pipelines/ENCPL839OAB) (ENCODE Project Consortium, 2012; Luo *et al.*, 2020; Hitz *et al.*, 2023). Unless specified otherwise, each benchmark was repeated 10 times and median values and standard error were used for plotting.

Table 3: File size, number of interactions and resolutions of .hic and .mcool files used for benchmarking.

	hic v8	hic v9	Multi-resolution cooler
File size (GBs)	176	77	53
# interactions	6,974,747,832		
# of non-zero entries (10 bp resolution)	5,873,705,034		5,873,722,963
Resolutions	10, 100, 500, 1 kbp, 5 kbp, 10 kbp, 25 kbp, 50 kbp, 100 kbp, 250 kbp, 500 kbp, 1 Mbp, 2.5 Mbp		

The code used to generate the results presented in the next section is hosted on GitHub at github.com/paulsengroup/2023-hictk-paper and is archived on Zenodo at doi.org/10.5281/zenodo.8331042. Code used to generate matrices in .hic v8, .hic v9 and .mcool format can be found under preprocessing/ while code used for benchmarking is available inside the benchmarks/ folder. The repository README contains instructions on how to run the preprocessing and benchmark steps. Files used as input for benchmarks are automatically downloaded or generated by the preprocessing workflows and are also deposited at the Gene Expression Omnibus under identifier GSE242815 (identifiers.org/geo:GSE242815). Benchmark results and matrices used for visualization purposes throughout the paper are deposited on Zenodo at doi.org/10.5281/zenodo.8331156.

Results

hictk is a header-only C++17 library accompanied by command-line tools to transparently operate on .hic and .cool files. hictk source code is available on GitHub at github.com/paulsengroup/hictk under the MIT license. Pre-built binaries for hictk can be obtained from bioconda (Linux and macOS). Docker images are available on GHCR.io and DockerHub. hictk can be built from source on Linux, macOS and Windows using a compiler toolchain supporting C++17, CMake v3.25+ and optionally Conan v2+. The C++ API can be used by other projects by including hictk source code using CMake or by installing hictk with Conan. hictkpy can be obtained through PyPI and bioconda or installed from source.

We characterize performance and memory requirements of hictk by comparison with cooler (Abdennur and Mirny, 2019), straw (Durand, Robinson, *et al.*, 2016) and hic2cool (github.com/4dn-dcic/hic2cool).

Converting file formats

Converting .hic to .cool or .mcool files is achieved through the hictk convert subcommand with

```
hictk convert input.hic output.mcool
```

The above command will create a .mcool by copying all resolution and balancing vectors available in the .hic file given as input. Fig. 1. shows the result of converting .hic files to .mcool. Interaction frequencies are fully preserved by the conversion process. Correctness of the conversion is tested automatically by the hictk CI pipeline every time a pull request or commit is submitted to hictk's repository on GitHub.

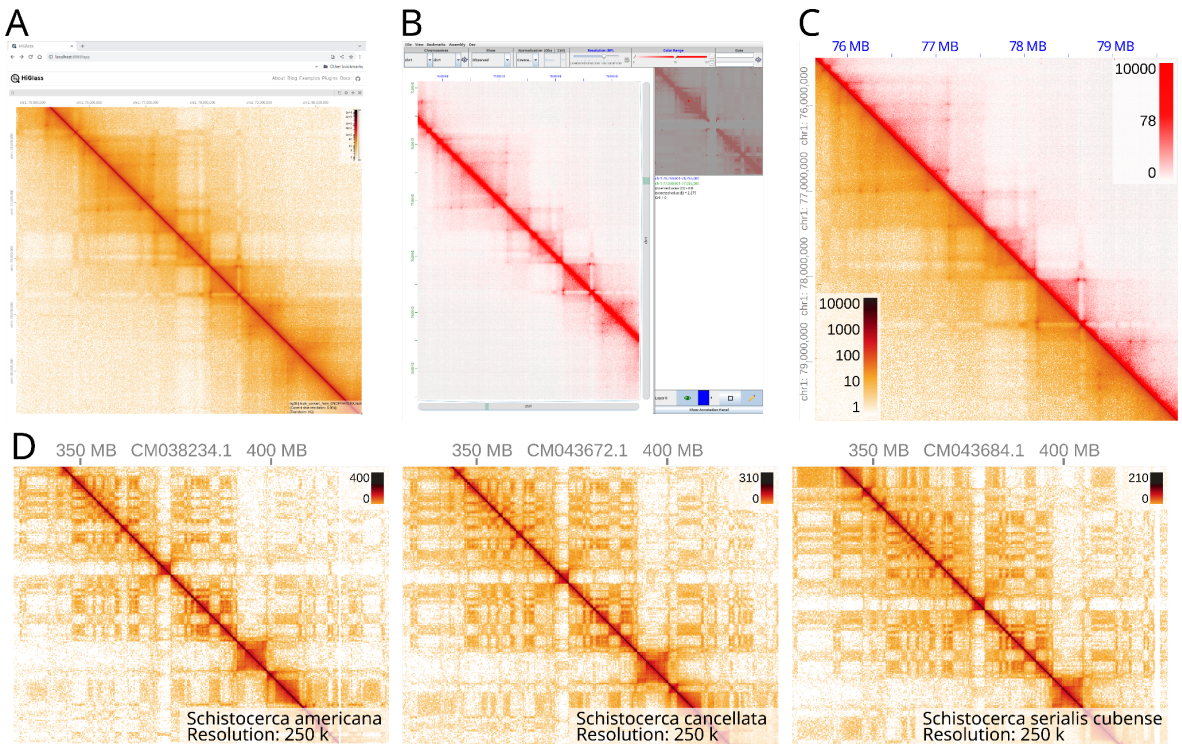

Fig. 1: VC-normalized interaction matrices for hg38 chromosome 1 in .mcool and .hic visualized using HiGlass and JuiceboxGUI, respectively. **A**: Interactions from a .mcool file visualized using HiGlass. **B**: Interactions from a .hic file visualized using JuiceboxGUI. **C**: Merged view of interactions visualized with HiGlass and JuiceboxGUI. **D**: Hi-C Interactions for three species of grasshopper from the DNA Zoo (Dudchenko *et al.*, 2017, 2018) visualized using HiGlass after converting matrices from .hic to .mcool using hictk convert.

We assess the performance of hictk convert by comparing the time required for converting files from .hic to .mcool format using hic2cool and hictk convert. For hictk we show the performance when converting from .hic v8 and v9 files. For hic2cool we only show performance when converting .hic v8 files, as this is the most recent standard supported by hic2cool.

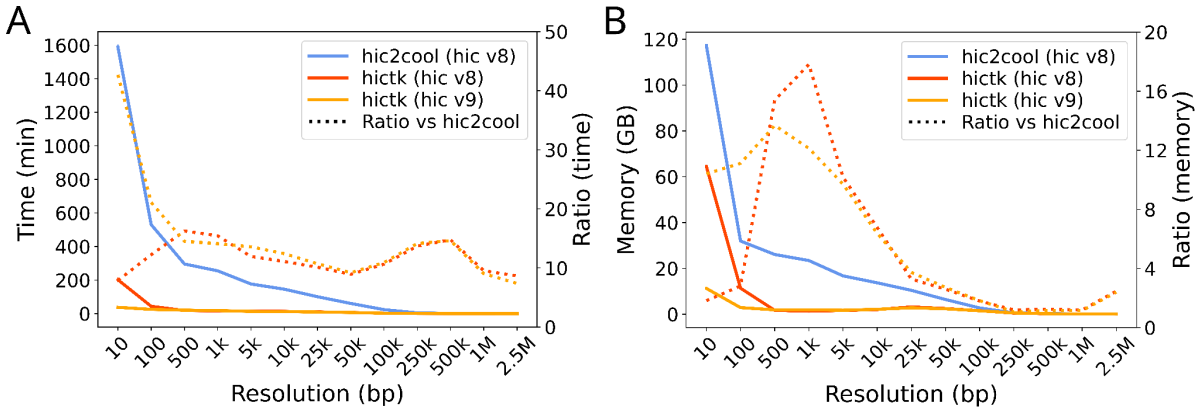

Fig. 2: Benchmark results comparing the performance of hictk convert and hic2cool when converting .hic files to .cool format for different resolutions of the Hi-C dataset. Each combination of file format and tool is colored differently. Dotted lines show the relative improvement in performance and memory usage of hictk compared to hic2cool. **A**: Time required to convert a .hic file to .cool format. The Y axis on the right shows the performance improvement achieved when running hictk compared to hic2cool. **B**: Peak memory usage during conversion from .hic to .cool format. The right Y axis shows reduction of memory requirements when using hictk compared to hic2cool.

As seen in Fig. 2A, when converting .hic v8 files, hictk is 7-16 times faster than hic2cool across all tested resolutions. It also maintains a comparable memory footprint at coarse resolution and notably consumes significantly less memory at finer resolutions (only 1/14 at 500 bp resolution) (Fig. 2B). When handling .hic v9 files, the disparity in performance and memory utilization becomes even more pronounced at higher resolutions. hictk outperforms hic2cool by a factor of 42 at a resolution of 10 bp, while consuming only one-tenth of the memory. At resolutions above 500 bp, hictk converts .hic v8 and v9 with a similar performance. At finer resolutions, however, performance is generally better for .hic v9 files. This performance difference is due to the new indexing strategy used in .hic v9 files, which leads to more compact indexes and interaction blocks that are less sparse and can be decompressed more efficiently.

hictk convert can also convert matrices from .mcool to .hic v8 and v9.

```
hictk convert input.mcool output.hic --juicer-tools-jar juicer_tools.jar
```

This will generate a .hic file containing all resolutions found in the .mcool file. If balancing weights are available in the .mcool, those will be copied in the output .hic.

Conversion is achieved by wrapping JuicerTools or HiCTools and writing interactions to a temporary file while ensuring they are properly sorted and in a format understood by JuicerTools and HiCTools. Conversion takes 31 h 49 min and uses up to 164 GBs of RAM (median values over 3 runs). To the best of our knowledge, hictk is the first tool to enable easy conversion from .mcool to .hic format.
We conclude that hictk convert is a drop-in replacement for hic2cool that is much more efficient at converting .hic files to .mcool format. Furthermore, hictk convert can be used to greatly simplify converting .mcool files to .hic format.

Reading interactions

Reading interactions from .hic and .cool files is done with the hictk dump subcommand:

```
hictk dump input.hic --resolution 1000
hictk dump input.cool
hictk dump input.mcool::/resolutions/1000
```

This will write to stdout genome-wide interactions sorted by genomic coordinates in COO format (row, column, interaction count). Option --join can be used to output interactions in bedgraph2 format (chrom1, start1, end1, chrom2, start2, end2, interaction count). When available, balancing weights can be applied to interactions by passing the weight name to option --balance.

To fetch interactions overlapping a region of interest options --range and --range2 can be used:

```
hictk dump input.cool --range chr1
hictk dump input.cool --range chr1:0-10,000,000 --range2 chr2:0-5,000,000
```

Next, we compare the performance of reading interactions from .hic8, .hic9 and .mcool files using cooler, straw and hictk. We measure the performance of dumping genome-wide as well as single chromosome interactions. Measurements for dumping genome-wide interactions using straw are missing as this operation is not supported by straw. Contrary to hictk and cooler, straw returns interactions sorted by block ID. Interactions are thus not guaranteed to be sorted by genomic coordinates. For this reason we also show the performance of a version of straw modified by us to sort interactions by genomic coordinates before returning them to the user (see Supplementary Text 1).

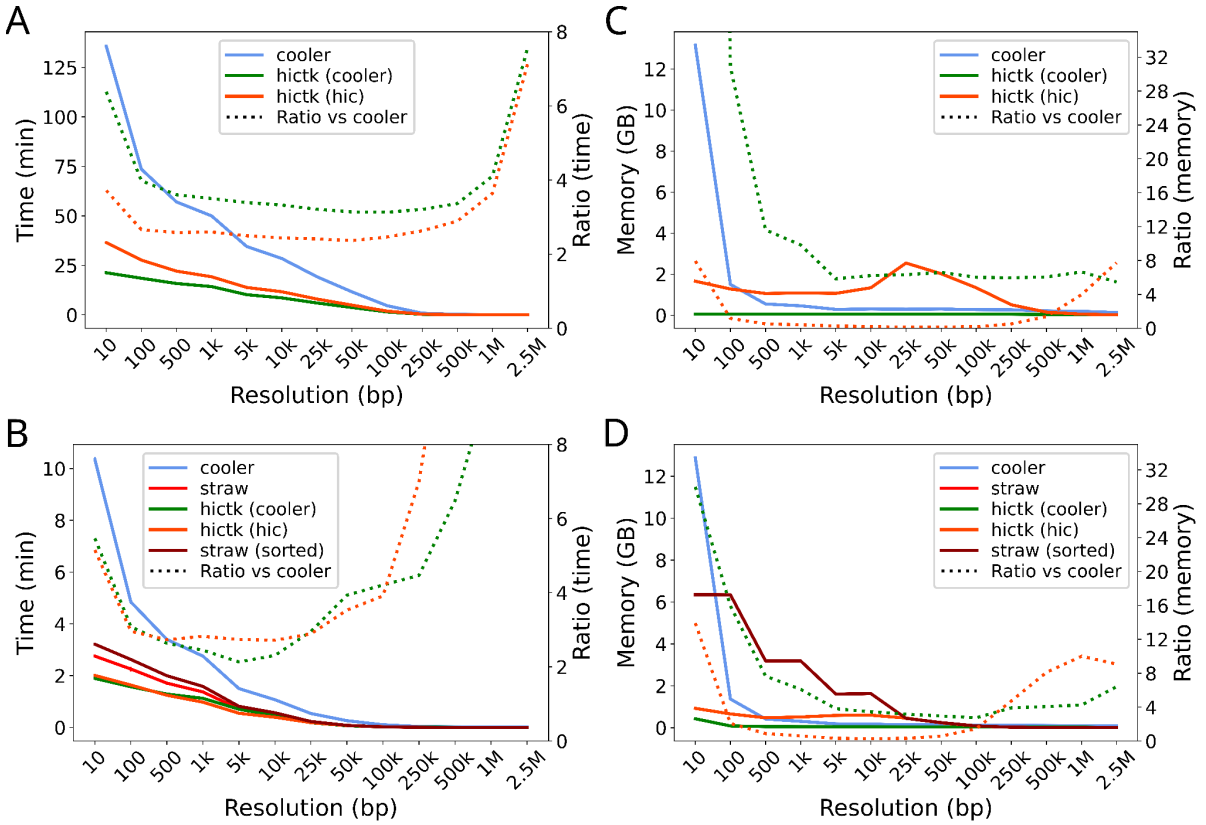

Fig. 3: Benchmark results comparing the performance of hictk dump, cooler dump and straw for different resolutions of the Hi-C dataset. Dotted lines show the relative improvement in performance and memory usage of hictk compared to cooler. **A**: Time required to dump genome-wide interactions. The second vertical axis (right) shows the relative performance improvement achieved when running hictk compared to cooler. **B**: Time required to dump interactions for chromosome 1. The second vertical axis (right) shows the performance improvement achieved when running hictk compared to cooler. **C**: Peak memory usage while dumping genome-wide interactions. The second vertical axis (right) shows reduction of memory requirements when using hictk compared to cooler. **D**: Peak memory usage while dumping interactions for chromosome 1. The second vertical axis (right) shows reduction of memory requirements when using hictk compared to cooler.

Our benchmark shows that hictk is 3 to 6 times faster than cooler and 1.5 times faster than straw for dumping genome wide (Fig. 3A) and chromosome 1 (Fig. 3B) interactions. Compared to straw, hictk uses significantly less memory across all tested resolutions regardless of file format (Fig. 3C and D). This is because hictk streams interactions using iterators, while straw buffers all interactions in memory before returning them to the user. The comparison with cooler memory requirements is more nuanced: when dumping interactions from .hic files, hictk uses more memory than cooler for resolutions greater than 1 kbp and approximately the same or less for higher resolutions; when dumping interactions from .cool files, hictk always requires less memory than cooler (Fig. 3C and D).

Dumping interactions from .hic files with hictk is slightly slower than performing the same operation on .cool files (Fig. 3A and 3B), as in this case interactions are stored separately for each pair of chromosomes, and hictk has to stitch together interaction for pairs of chromosomes in the right order before printing them to stdout. This is also the reason why hictk requires more memory to dump genome-wide interactions from .hic files than from .cool files (Fig. 3C and D). Dumping genome-wide interactions from .cool files is trivial, as interactions are already stored in a single vector sorted by genomic coordinates: this allows

us to dump interactions without having to load the file index, which is why hictk uses very little memory to perform this operation (less than 15 MBs at 10 bp resolution) (Fig. 3C).
In conclusion, hictk dump is significantly faster than available tools when dumping interactions for the entire genome or a single chromosome. Furthermore, hictk dump is the only tool to require less than 4 GBs of memory across all resolutions.

Creating .cool files from interactions in text format
Interactions in text format can be loaded into a .cool file using hictk load. Supported formats are 4DN-DCIC pairs, validPairs, bedgraph2 and COO.

```
hictk load -f 4dn hg38.chrom.sizes 1000 output.cool < interactions.txt
```

Interactions do not necessarily need to be sorted. However, if they are, the --assume-sorted flag can be used to efficiently load interactions in a single pass.

Next we benchmark the performance of creating .cool files from interactions in text format using cooler cload and hictk load. We do not compare hictk with juicer_tools because the hictk library itself is not capable of writing .hic files. The benchmark consists of ingesting interactions in 4DN-DCIC pairs format into a .cool file. Both tools were run assuming interactions are in an unspecified order using a chunk size of 50,000,000.

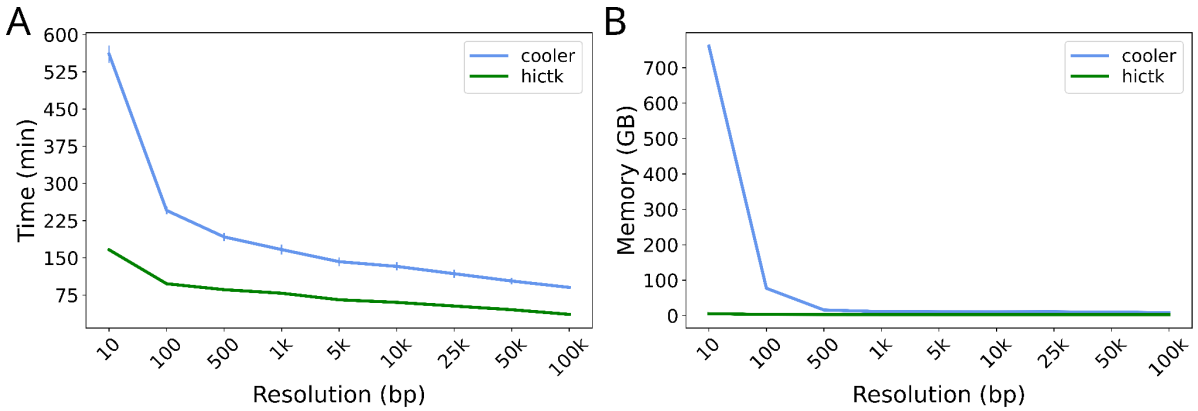

Fig. 4: Benchmark result comparing the performance of hictk load and cooler cload when creating .cool files at different resolutions. Error bars show the standard error of time measurements taken across multiple runs of the same benchmarks. **A**: Time required to ingest interactions at different resolutions. **B**: Peak memory usage while ingesting interactions at different resolutions.

To generate a .cool file at 10 bp resolution, hictk outperforms cooler by a factor of 2-2.5 (Fig. 4A). Memory requirements of hictk are also modest compared to those of cooler, necessitating 9 and 16 GBs of memory at 1 kbp and 10 bp, respectively, while cooler requires 11 and 760 GBs to process the same resolutions (Fig. 4B).
Both hictk and cooler ingest interactions in two stages. In the first stage a fixed number of interactions are buffered and sorted in memory. Sorted interactions are then written to temporary files. Once all interactions have been processed, intermediate files are merged into the output cooler. This second stage is the most memory intensive in both applications. hictk uses a k-way merge algorithm using pairs of iterators and a priority queue which can merge k-files buffering exactly k-pixels in memory. In practice, for performance reasons, hictk

buffers more than one pixel per file, but the number is low enough to require in the order of 1-100 MBs of memory per intermediate file.
We conclude that hictk load is a valid alternative to cooler load across a wide range of resolutions, both in terms of processing speed and memory requirements.

Creating multi-resolution coolers
Single-resolution cooler files can be converted to multi-resolution cooler files using hictk zoomify.

```
hictk input.cool output.mcool --resolutions 1000 2000 5000 10000 …
```

The conversion is achieved by iterative coarsening, where a low resolution matrix is generated from a high resolution one.

We compare the performance of cooler coarsen and hictk zoomify when generating low resolution .cool files from higher resolution files by coarsening. cooler coarsen supports processing files using multiple CPU cores, and was tested using 1, 4 and 8 cpu cores. hictk zoomify does not support multi-threading and so numbers were obtained using a single CPU core.

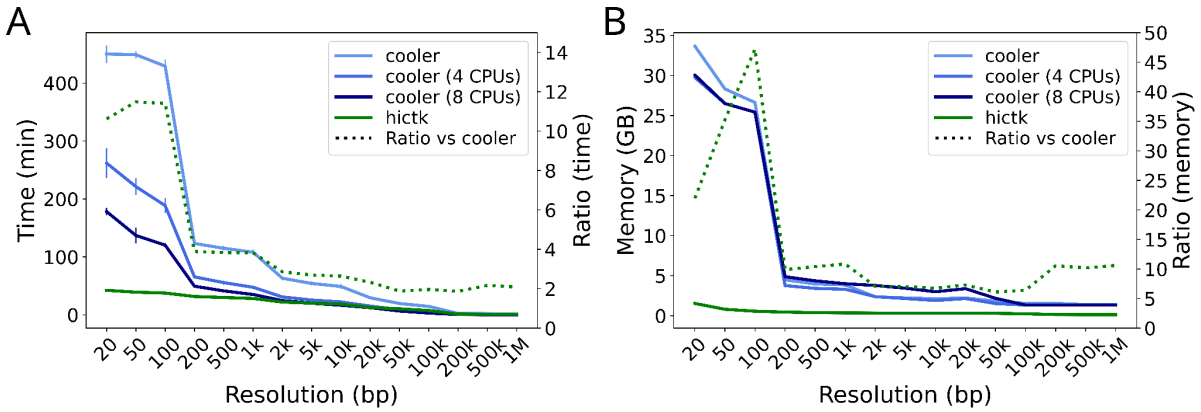

Fig. 5: Benchmark results comparing the performance of hictk zoomify and cooler coarsen for different resolutions of the Hi-C dataset. The horizontal axis shows the resolution after coarsening. The greatest common divisor between the target resolution and resolutions listed in Table 1 was used as base resolution. Error bars show the standard error of time measurements taken across multiple runs of the same benchmarks. Dotted lines show the relative improvement in performance and memory usage of hictk compared to cooler using 1 CPU core. **A**: Time required to coarsen .cool files at different resolutions. The second vertical axis (right) shows the performance improvement achieved when running hictk compared to cooler using 1 CPU core. **B**: Peak memory usage while coarsening .cool files. The second vertical axis (right) shows reduction of memory requirements when using hictk compared to cooler using 1 CPU core.

When using a single CPU core, hictk is faster than cooler across all tested resolutions. Yet, when cooler is allowed to use 4 or 8 CPU cores, hictk is relatively slower at low resolutions (greater than 20kbp). However, hictk is still 5 times faster when processing high resolutions (Fig. 5A).
Generating a 20 bp .cool file from a 10 bp file takes less than 45 minutes with hictk and approximately 3.5 hours with cooler, even when utilizing 8 CPU cores (Fig. 5A). Furthermore, hictk requires significantly less memory than cooler across all resolutions,

using between 130 MBs and 1.5 GBs of RAM, while cooler requires at least 1.3 GBs and up to 35 GBs (Fig. 5B).
hictk takes advantage of a memory-efficient data structure to represent bin tables, which only requires storing chromosome names, lengths and the prefix sum of chromosome lengths, making memory requirements independent of resolution. This data structure allows for efficient mapping between bin identifiers (i.e. matrix rows and columns) and genomic coordinates. Given the above, coarsening in hictk amounts to the following operations: map interactions from the old coordinate system to the new one, then aggregate interactions mapping to the same pair of bins. Both operations are performed using iterators, buffering at most pixels overlapping one row at the lowest resolution. Interactions are aggregated using a hashmap and a priority queue. An implementation using binary-search trees was tested but found on average to be slower.

In conclusion, hictk zoomify can be used as a more efficient alternative to cooler coarsen and cooler zoomify to generate multi-resolution Cooler files by coarsening.

Fetching interactions overlapping pairs of TADs
Next, we sought to assess the performance of the library behind hictk by measuring the time required to fetch and sum interactions overlapping a pair of TADs. The benchmark consists of two parts, one testing the performance within chromosomes (cis queries) and the other between chromosomes (trans queries). Each part runs 5000 queries generated by random sampling with replacement on TADs from ENCFF301CUL (identifiers.org/encode:ENCFF301CUL). Cis queries are generated by using the same coordinates for both dimensions, while trans queries are generated by using coordinates from TADs mapping to different chromosomes. For each query we measure the time and memory required to fetch and sum interactions returned by the query.
We compare the performance of hictk C++ API with cooler and hic-straw Python API using benchmark scripts fetch_and_sum_cooler.py, fetch_and_sum_straw.py and executable fetch_and_sum_hictk hosted on GitHub at github.com/paulsengroup/2023-hictk-paper

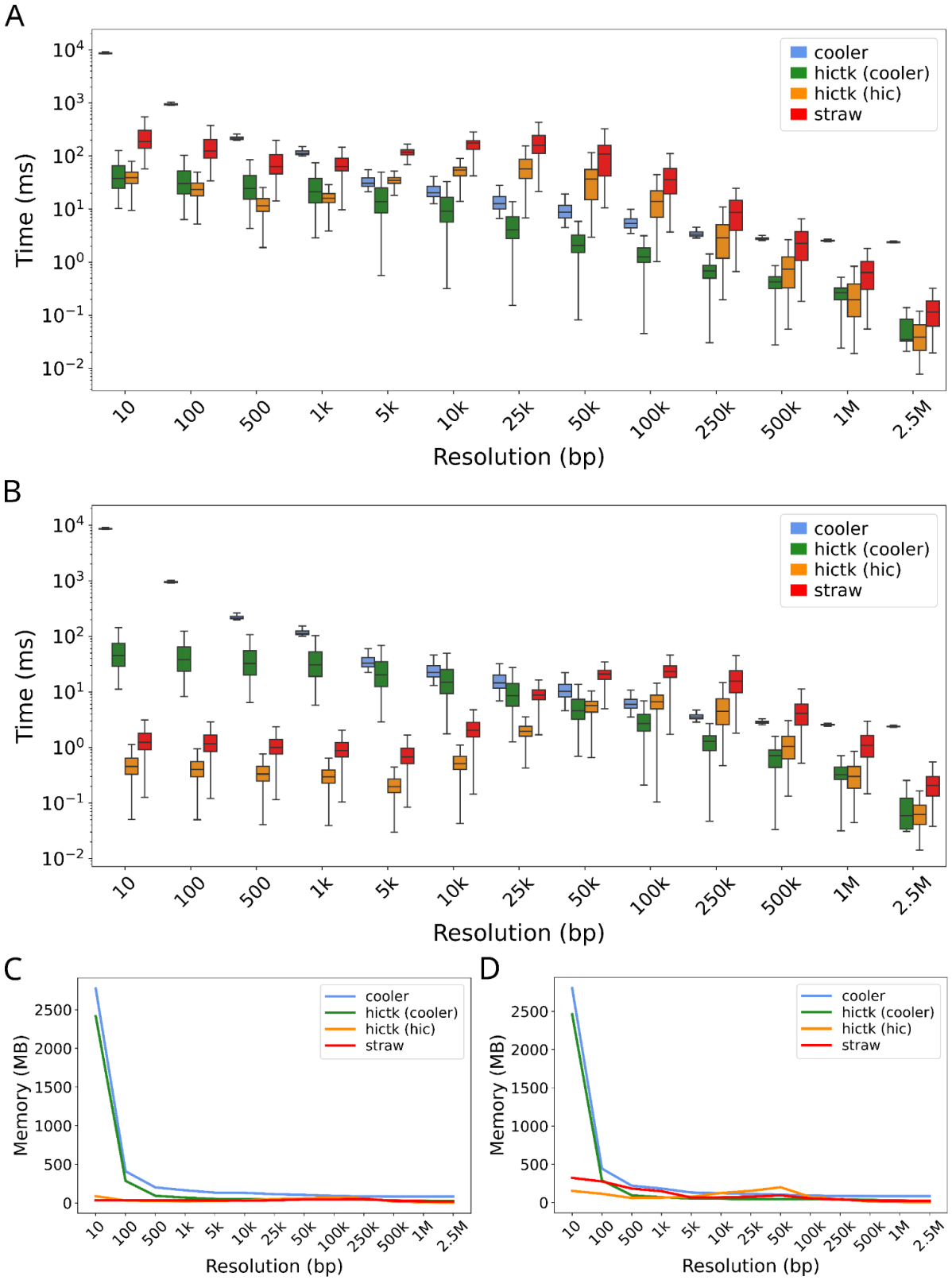

Fig. 6: Benchmark results comparing performance when fetching interactions overlapping pairs of TADs. Queries were run for different resolutions of the Hi-C dataset. **A**: Time required to process queries for pairs of TADs located on the same chromosome (cis queries). **B**: Time required to process queries for pairs of TADs located on different chromosomes (trans queries). **C**: Peak memory usage when processing cis queries. **D**: Peak memory usage when processing trans queries.

For cis queries, the performance of hictk in reading .cool files stands out as the fastest among the tested tools (Fig. 6A). It is faster for resolutions ranging from 5 kbp to 2.5 Mbp (with the exception of 1.0 Mbp). Notably, when working with resolutions of 1 kbp or higher, hictk's efficiency in reading .hic files becomes even more pronounced. In contrast, cooler lags behind as the slowest performer in both high and low-resolution scenarios, while straw is the slowest tool at intermediate resolutions. Overall, hictk is 3-6 times faster than straw at reading .hic files and 2-220 times faster than cooler at reading .cool files.
When processing trans queries, hictk is once again the fastest among the tested tools (Fig. 6B). In general, hictk reading .cool file is faster when processing 50 kbp to 2.5 Mbp resolutions, while hictk reading .hic files becomes the fastest tool at resolutions higher than 50 kbp. Finally, at high resolutions, straw is faster than cooler, while at low resolution cooler is faster than straw. Overall, hictk is 2-6 times faster than straw at reading .hic files and 1.5-185 times faster than cooler at reading .cool files.
The memory requirements of all tested libraries when processing cis queries are similar for medium and low resolution, with hictk and straw consistently using slightly less memory than cooler (Fig. 6C). At resolutions higher than 500 bp the memory required to process .cool files grows significantly for both cooler and hictk, as both libraries have to load a relatively large file index into memory in order to process queries. When processing .hic files, both straw and hictk require approximately the same amount of memory across all tested resolutions (Fig. 6C). When processing trans queries, all libraries require approximately the same amount of memory to serve queries at 500 bp resolutions and lower (Fig. 6D). At higher resolutions, the memory requirements for processing .cool files are once again significantly greater due to reading the file index into memory. However, hictk consistently requires slightly less memory than cooler to process .cool files (Fig. 6D). The memory requirements of straw and hictk are similar for resolutions of 5 kbp and lower, while at higher resolutions the memory requirements of hictk are slightly lower (Fig. 6D).

In conclusion, hictk is consistently faster than cooler and straw when reading .cool and .hic files respectively. In general, hictk achieves peak performance when reading from .hic and .cool files at high and low resolutions respectively. Finally, cooler performance is degraded at very high resolutions, with it being slower than other tools by up to 4 orders or magnitude, suggesting that under these circumstances the benchmark is testing code that has not been optimized for very high resolutions.

Updating existing applications to use hictk

One of the features of hictk is the ability to transparently operate on .hic and Cooler files, simplifying writing applications reading files in these formats. To showcase this capability we modified dejunlin/hicrep (Lin *et al.*, 2021), a Python implementation of the HiCRep algorithm used to calculate the stratum-adjusted correlation coefficient (SCC) (Yang *et al.*, 2017).
The original version of dejunlin/hicrep uses the cooler library to read files in .cool and .mcool format. We forked dejunlin/hicrep and updated the application to use hictkpy instead of cooler, enabling hicrep to read files in .hic, .cool and .mcool formats. The update involved adding 4 new lines of code, removing 59 and modifying 48 (see Supplementary Text 2).

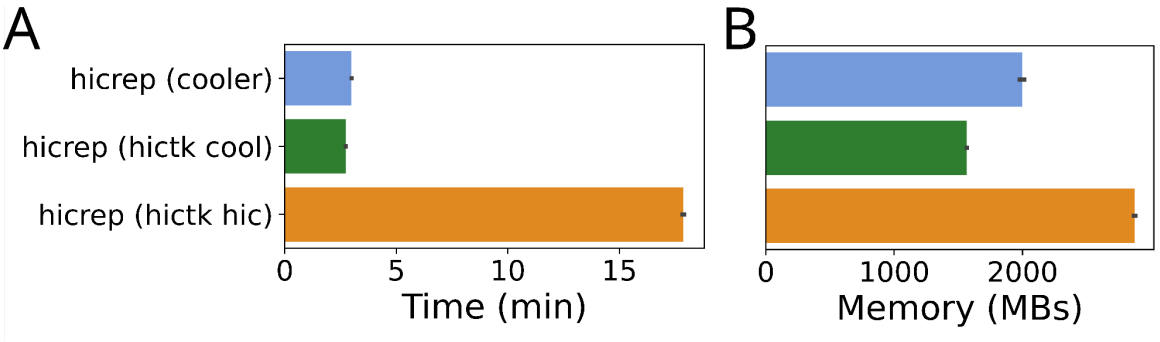

Fig. 7: Benchmark results comparing the performance of hicrep using hictk and cooler as libraries to fetch interactions at 25 kbp resolution. Error bars show the standard error of time and memory measurements taken across multiple runs of the same benchmarks. **A**: Time required to compute the SCC for all human chromosomes. **B**: Peak memory usage while computing the SCC for all human chromosomes.

For reading .cool files, performance is similar using cooler or hictk (Fig. 7A). Note, however, that computing the SCC on .hic files takes much longer than on .cool files due to a normalization step carried out by hicrep, which requires traversing .hic files in their entirety. This is not required when processing .cool files because the normalization factor is pre-computed and stored in the file. Memory requirements are slightly reduced by using hictk instead of cooler to process .cool files (Fig. 7B). However, this is not the case when processing .hic files, where hictk requires more memory than cooler or hictk reading .cool files.

Discussion

We present hictk, an efficient toolkit to easily and transparently operate on .cool and .hic files. The inexorable increase in number and size of Hi-C datasets makes having access to efficient and scalable tools a necessity. We show that hictk outperforms existing tools across a broad range of resolutions, often by orders of magnitude, while requiring significantly less memory when processing high resolution interaction matrices.

We designed the hictk library to abstract file format from applications, enabling developers to write applications supporting both .hic and .cool formats without code duplication. We also provide a tool to automate conversion between .hic and .mcool formats, allowing end-users to easily migrate between file formats. Taken together, these two features relieve users and developers from having to worry about which file format their Hi-C data is stored.

Nowadays, Hi-C datasets are routinely generated by individual research groups as well as large consortia such as 4DNucleome (Dekker *et al.*, 2017; Reiff *et al.*, 2022), ENCODE (ENCODE Project Consortium, 2012; Luo *et al.*, 2020), Vertebrate Genomes Project (VGP) (Rhie *et al.*, 2021) and the DNA Zoo (Dudchenko *et al.*, 2017). Data processing of Hi-C raw sequencing data consists of four main steps: quality control, mapping, identification of ligation junctions and finally interactions binning and counting. These steps are often automated through data processing pipelines. Developing pipelines by choosing tools that prioritize correctness and efficiency will result in reliable and high-performance pipelines, reducing the computational requirements for Hi-C data processing. hictk is continuously tested by automated tests, can bin and count interactions (hictk load and zoomify) and convert between formats (hictk convert) with excellent performance, making it a good candidate for inclusion in Hi-C data processing pipelines.

Finally, Hi-C data is mainly generated for two purposes: to aid scaffolding during the

assembly of genomes, and to study the 3D spatial organization of genomes. Datasets generated for assembly purposes are in theory suitable for further downstream analysis even though this is rarely done in practice for various reasons, one of the reasons being format incompatibilities. hictk convert is capable of converting .hic files v6-9 to .cool or .mcool and vice versa, facilitating the reuse of publicly available datasets without the need of remapping.
We conclude that hictk is a valid alternative to existing tools providing a level of flexibility and performance suitable to process current and next-generation Hi-C datasets.

Acknowledgements

We thank the ENCODE Consortium and the lab of Erez Aiden for contributing with ENCODE data used as part of this study.

Unpublished genome assemblies and sequencing data for Schistocerca americana, Schistocerca cancellata and Schistocerca serialis cubense are used with permission from the DNA Zoo Consortium (www.dnazoo.org).

We thank the DNA Zoo and the Behavioral Plasticity Research Institute (BPRI) for providing the genome assemblies for Schistocerca americana, Schistocerca cancellata and Schistocerca serialis cubense.

JP acknowledges funding from the Norwegian Research Council (project 324137).

References

Abdennur,N. and Mirny,L.A. (2019) Cooler: scalable storage for Hi-C data and other genomically labeled arrays. *Bioinformatics*, **36**, 311–316.
Burton,J.N. *et al.* (2013) Chromosome-scale scaffolding of de novo genome assemblies based on chromatin interactions. *Nat. Biotechnol.*, **31**, 1119–1125.
Dekker,J. *et al.* (2017) The 4D Nucleome Project. *Nature*, **549**, 219.
Di Tommaso,P. *et al.* (2017) Nextflow enables reproducible computational workflows. *Nat. Biotechnol.*, **35**, 316–319.
Dudchenko,O. *et al.* (2017) De novo assembly of the genome using Hi-C yields chromosome-length scaffolds. *Science*, **356**, 92–95.
Dudchenko,O. *et al.* (2018) The Juicebox Assembly Tools module facilitates de novo assembly of mammalian genomes with chromosome-length scaffolds for under $1000. *bioRxiv*, 254797.
Durand,N.C., Robinson,J.T., *et al.* (2016) Juicebox Provides a Visualization System for Hi-C Contact Maps with Unlimited Zoom. *Cell Systems*, **3**, 99–101.
Durand,N.C., Shamim,M.S., *et al.* (2016) Juicer Provides a One-Click System for Analyzing Loop-Resolution Hi-C Experiments. *cels*, **3**, 95–98.
Eddelbuettel,D. (2013) Seamless R and C++ Integration with Rcpp Springer New York.
ENCODE Project Consortium (2012) An integrated encyclopedia of DNA elements in the human genome. *Nature*, **489**, 57–74.
Hitz,B.C. *et al.* (2023) The ENCODE Uniform Analysis Pipelines. *bioRxiv*.
Imakaev,M. *et al.* (2012) Iterative correction of Hi-C data reveals hallmarks of chromosome organization. *Nat. Methods*, **9**, 999–1003.
Kaplan,N. and Dekker,J. (2013) High-throughput genome scaffolding from in vivo DNA interaction frequency. *Nat. Biotechnol.*, **31**, 1143–1147.

Kerpedjiev,P. *et al.* (2018) HiGlass: web-based visual exploration and analysis of genome interaction maps. *Genome Biol.*, **19**, 1–12.
Knight,P.A. and Ruiz,D. (2012) A fast algorithm for matrix balancing. *IMA J. Numer. Anal.*, **33**, 1029–1047.
Lander,E.S. *et al.* (2001) Initial sequencing and analysis of the human genome. *Nature*, **409**, 860–921.
Lieberman-Aiden,E. *et al.* (2009) Comprehensive mapping of long-range interactions reveals folding principles of the human genome. *Science*, **326**, 289–293.
Lin,D. *et al.* (2021) HiCRep.py: fast comparison of Hi-C contact matrices in Python. *Bioinformatics*, **37**, 2996–2997.
Luo,Y. *et al.* (2020) New developments on the Encyclopedia of DNA Elements (ENCODE) data portal. *Nucleic Acids Res.*, **48**, D882–D889.
Palsberg,J. and Jay,C.B. (1998) The essence of the Visitor pattern. In, *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*., pp. 9–15.
Reiff,S.B. *et al.* (2022) The 4D Nucleome Data Portal as a resource for searching and visualizing curated nucleomics data. *Nat. Commun.*, **13**, 1–11.
Rhie,A. *et al.* (2021) Towards complete and error-free genome assemblies of all vertebrate species. *Nature*, **592**, 737–746.
Selvaraj,S. *et al.* (2013) Whole-genome haplotype reconstruction using proximity-ligation and shotgun sequencing. *Nat. Biotechnol.*, **31**, 1111–1118.
Yang,T. *et al.* (2017) HiCRep: assessing the reproducibility of Hi-C data using a stratum-adjusted correlation coefficient. *Genome Res.*, **27**, 1939–1949.