Bioinformatics, 2020, pp. 1–7

doi: DOI HERE
Advance Access Publication Date: Day Month Year
Paper

PAPER

Synchronized iteration for genomic data processing

Stefano Perna [1,*], Pietro Pinoli [2], Val Tannen [3], Stefano Ceri [2], Limsoon Wong [1,*]

[1]School of Computing, National University of Singapore, Singapore, 117417, Singapore
[2]DEIB, Politecnico di Milano, Milano, 20133, Italy
[3]Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA
[*]To whom correspondence should be addressed.
FOR PUBLISHER ONLY Received on Date Month Year; revised on Date Month Year; accepted on Date Month Year

Abstract

Motivation: Processing of large data files is unavoidable in genomic pipelines. Many tools that do this are either stand-alone languages or command-line tools. There is an impedance mismatch when these tools are used with a host programming language to support more complex analysis.
Results: A novel concept, Synchrony iterator, is introduced. It allows efficient algorithms underlying such tools to be easily expressed. As a demonstration, the powerful GenoMetric Query Language (GMQL) is emulated using Synchrony iterators in Scala/Python, and the resulting equivalents of these queries are very efficient. Notably, a user can freely intermix GMQL-like queries with other features of Scala/Python, thereby overcoming the impedance mismatch problem.
Availability: Implementations of Synchrony iterator and Synchrony GMQL, in both Scala and Python 3, are available at `https://www.comp.nus.edu.sg/~wongls/projects/synchrony`.
Contact: dcsstef@nus.edu.sg and wongls@comp.nus.edu.sg

Key words: Synchronized iteration, Synchrony iterator, GMQL, Genomic data processing.

Introduction

Some tools have emerged that exploit the intrinsic structures in genomic datasets to achieve efficiency in processing these large datasets. E.g., the GenoMetric Query Language (GMQL) [5] is a declarative language for processing large quantities of genomic regions, and BEDOPS [6] is a popular command-line toolkit for extracting and manipulating genomic regions.

A challenge, from the perspective of a programmer implementing these tools, is the impedance mismatch between the programming language used for the implementation and the kind of algorithms needed in these bioinformatics tools. Modern programming languages provide good support which makes programs manipulating large collections more readable. Yet it is still difficult to implement efficient algorithms (e.g. intersect collections of genomic regions under non-trivial matching conditions [9]) in an easy-to-understand way.

There are two further challenges, this time from a user-programmer's perspective. One is that, as these tools provide a rich set of operators on genomic data, the user-programmer has to familiarize himself with their thick manuals to use them effectively, even if he is a competent programmer. The other is that an impedance mismatch [2] often arises when using these stand-alone tools in conjunction with a host programming language, to perform more complex analysis on genomic data.

This paper addresses these three challenges. Firstly, we describe *Synchrony iterator*, a novel class of iterators enabling efficient sychronized iterations over multiple collections to be expressed in simple-to-understand comprehension syntax [1]. Synchrony iterator is efficient and can succinctly implement those efficient algorithms alluded to in the first challenge above.

Secondly, this implies that a user who has mastered Synchrony iterator, which is a single programming construct, as opposed to a thick manual of many genomic operators, is already able to implement typical genomic queries in a simple and efficient way. So, it also addresses the second challenge.

Thirdly, we use Synchrony iterator to emulate GMQL. This emulation results in a Scala/Python library, *Synchrony GMQL*. This library naturally embeds GMQL into Scala/Python. One can thus freely intermix GMQL-like operators into Scala and Python programs, addressing the third challenge. Remarkably, Synchrony GMQL is implemented in about 4,000/1,500 lines of Scala/Python codes, as counted by cloc. Despite the

simplicity of its implementation, Synchrony GMQL outperforms a local installation of GMQL [5]. This is a testimonial to Synchrony iterator as an elegant idea for expressing efficient synchronized iterations on multiple collections in a succinct and easy-to-understand way.

Methods

Basics of Synchrony iterator

We describe Synchrony iterator using our Python implementation, as Python is likely more familiar to the reader. For convenience, we refer to a Synchrony iterator as "eiterator", as it is an *enhanced* iterator. An eiterator ys = `EIterator`(Y, cl) can be constructed from any iterable object Y (e.g. a list); where cl is an optional argument which is a function for releasing resources held for Y when the eiterator is no longer needed. Although the eiterator ys can be used as an iterator of the elements of Y in the usual manner, it is endowed with several additional methods, including `syncedWith` which characterizes its conceptual novelty. The pertinent method `syncedWith` is described below, along with some supporting methods.

- ys.`syncedWith`(x, bf, cs) returns a list vs equivalent to `list(filter(`cs`, takewhile(lambda` $y, x : bf(y, x)$ `or` $cs(y, x)$, ys`))` and also updates ys to an eiterator equivalent to `dropwhile(lambda` $y, x : bf(y, x)$ `or` $cs(y, x)$`).prepend(`vs`)`.
- ys.`prepend`(vs) updates ys by prepending elements in the list vs to the front of ys.
- ys.`peekahead`(n) returns [ ] if there are fewer than n elements left in ys; otherwise, it returns $[y]$ where y is the nth element in ys, without removing y or any other elements from ys.
- ys.`close`() releases resources held for ys. It is automatically invoked when accessing the last element in ys, but it can be invoked earlier to terminate the iteration on ys midway.

Synchrony iterator is an efficient mechanism for "synchronizing" iterations on two or more collections. To appreciate this aspect, let X and Y be two lists. Let $(x << x' \mid X)$ mean x appears before x' in X, and $(y << y' \mid Y)$ mean y appears before y' in Y. A predicate $bf(y, x)$ is said to be "monotonic" wrt (X, Y) if (i) for each y in Y, $bf(y, x)$ implies $bf(y, x')$ whenever $(x << x' \mid X)$; and (ii) for each x in X, $bf(y, x)$ implies $bf(y', x)$ whenever $(y' << y \mid Y)$. A predicate $cs(y, x)$ is said to be "antimonotonic" wrt a monotonic predicate $bf(y, x)$ if (i) for each y in Y, $bf(y, x)$ and $\neg cs(y, x)$ implies $\neg cs(y, x')$ whenever $(x << x' \mid X)$; and (ii) for each x in X, $\neg bf(y, x)$ and $\neg cs(y, x)$ implies $\neg(y', x)$ whenever $(y << y' \mid Y)$.

Theorem 1 (cf. [9]) *Let* `syncmap` *be the program below.*

```
def syncmap(X,Y,bf,cs,f):
  ys = EIterator(Y)
  return [f(x,ys.syncedWith(x,bf,cs)) for x in X]
```

Let X and Y be two lists, $bf(y, x)$ be monotonic wrt (X, Y), $cs(y, x)$ be antimonotonic wrt $bf(y, x)$, and $f(x, vs)$ has time complexity $O(|vs|)$. Then,

1. `syncmap`(X, Y, bf, cs, f) *produces the same list as* $[f(x,$ `[`y `for` y `in` Y `if` $cs(y, x)$`])` `for` x `in` $X]$.
2. `syncmap`(X, Y, bf, cs, f) *has time complexity* $O(|X|+k|Y|)$, *provided there is some k such that for each y in Y, there are at most k elements x in X satisfying $cs(y, x)$.*

From part 1 of the theorem, `syncmap`(X, Y, bf, cs, f) produces a list same as that produced by $[f(x,$ `[`y `for` y `in` Y `if` $cs(y, x)$`])` `for` x `in` $X]$. Suppose for each y in Y, there are at most k elements x in X satisfying $cs(y, x)$. Then $[f(x,$ `[`y `for` y `in` Y `if` $cs(y, x)$`])` `for` x `in` $X]$ has quadratic complexity, $O(|X| \cdot k|Y|)$. In contrast, by part 2 of the theorem, `syncmap`(X, Y, bf, cs, f) has linear complexity, $O(|X| + k|Y|)$. So, the Synchrony iterator-based program is far more efficient.

This tremendous gain in efficiency illustrates the powerful synchronized iteration mechanism effected by the seemless interaction between Synchrony iterator and comprehension syntax. For each x in X, vs = ys.`syncedWith`(x, bf, cs) is computed. By definition of `syncedWith`, vs is same as `list(filter(`cs, `takewhile(lambda` $y, x : bf(y, x)$ `or` $cs(y, x)$, ys`))`. The function `takewhile` stops the iteration on ys as soon as it encounters an element y such that both $bf(y, x)$ and $cs(y, x)$ are false. This is due to part (ii) of the antimonotonicity of cs, $\neg bf(y, x)$ and $\neg cs(y, x)$ implies $\neg cs(y', x)$ for all subsequent elements y' in Y. So, the iteration on Y for this x is safely stopped early, avoiding a full iteration on Y. Recall also from the definition of `syncedWith` that ys is updated to an eiterator equivalent to `dropwhile(lambda` $y, x : bf(y, x)$ `or` $cs(y, x)$`).prepend(`vs`)`. This means on the next x, the iteration on Y does not start from the beginning of Y. Rather it starts from the elements of vs prepended previously, and continues on to where the iteration for the previous x stops. This is due to part(i) of the antimonotonicity of cs, $bf(y, x)$ and $\neg cs(y, x)$ implies $\neg cs(y, x')$ for all x' coming after x. So, provided bf is monotonic wrt (X, Y), the antimonotonicity of cs implies that each x in X is "seen" by a different segment of Y, with overlaps $|vs| \leq k$ on average, resulting in $O(|X| + k|Y|)$ complexity.

Synchrony iterators on large files

Our Synchrony iterator implementation provides the class `EFile` for representing large data files. Let f be a file containing a list of entries. Let df be an incremental deserializer function for reading entries in f; i.e. df sequentially fetches a few entries at a time, on demand. Let sf be a serializer function for writing entries in f. Then `OnDiskEFile`(f, sf, df) constructs an `EFile` representing the file f, while `TransientEFile`(vs, sf, df) constructs an `EFile` representing an iterable or eiterator vs that can be serialized to and deserialized from disk using sf and df. An `EFile` vf provides a number of methods, including:

- vf.`eiterator`(), which produces a new Synchrony iterator on vf. If vf represents a file on disk, this eiterator uses the incremental deserializer associated with vf to read a few sequential entries in vf at a time as needed. So, a Synchrony iterator keeps only a small segment of a large file in memory.
- vf.`serialized`(), which serializes vf to disk.
- vf.`sorted`(), which sorts vf on disk using the canonical ordering defined on vf. The sorting is done by chopping vf into m chunks, sorting each chunk in memory and writing to a temporary file, then merging the m files. The in-memory sorting has linearithmic complexity, while the final merging has linear complexity. However, due to disk access, the former is dominated by the latter; thus, near-linear performance is observed when vf has many items.
- vf.`mergedWith`$(f_1, ..., f_n)$, which merges vf, f_1, ..., f_n, assuming all these `EFile` objects are already pre-sorted using their respective canonical ordering.

BED files and sample files

We use Synchrony iterator on genomic data files, in particular, BED files compatible with the popular BEDOPS [6] toolkit. A BED file is an on-disk list of BED entries. A BED entry is a region on a chromosome and has some annotations associated with it. It is represented in our implementation as an object in the class `Bed`. A region on chromosome ch, starting at position st, ending at position en, on strand sn, that has annotations {`name` = nm, `score` = sc, ...} is represented as a `Bed` object x = `Bed`({`chrom` : ch, `chromStart` : st, `chromEnd` : en, `strand` : sn, `name` : nm, `score` : sc, ...}). Any annotation ℓ on the BED entry represented by a `Bed` object x can be accessed as $x.\ell$, e.g. x.`chrom` is the chromosome of the BED entry.

We often want to compare BED entries based on their chromosomal positions or loci. So, a predicate object p = `BedPred`(y, x) provides predicates on the loci of `Bed` objects y and x, including the followings:

- p.`isBefore`(), which is `True` iff the locus of y is before the locus of x in the "canonical ordering" of loci. Loci are canonically ordered by (`chrom`, `chromStart`, `chromEnd`, `strand`).
- p.`overlap`(n), which is `True` iff the locus of y overlaps the locus of x by at least n bases.
- p.`dl`(n), which is `True` iff the locus of y does not overlap that of x and the distance between them is less than n bases.

It is a simple exercise to prove the proposition below. Thus these predicates (and others which we have omitted here) can be used in conjunction with Synchrony iterator.

Proposition 2 *Let X and Y be two BED files sorted based on the canonical ordering on loci; and p =* `BedPred`(y, x) *is a predicate object on y and x, which are respectively* `Bed` *objects representing some BED entries in Y and X. Then p.*`isBefore`() *is monotonic wrt (X, Y), and p.*`overlap`(n) *and p.*`dl`(n) *are both antimonotonic wrt p.*`isBefore`().

BED files are represented by the class `BedEFile`, a subclass of `EFile`. This class has two main constructors: `OnDiskBedEFile`(f) for constructing a `BedEFile` object representing the BED file f, and `TransientBedEFile`(vs) for constructing a `BedEFile` object representing an iterable or eiterator vs of `Bed` objects.

Some metadata can also be associated with an entire BED file. A BED file and its associated metadata is called a "sample." Our Synchrony iterator implementation provides a class `Sample` for representing samples. Its constructor `Sample`(vf, d) constructs a `Sample` object representing the `BedEFile` object vf and its metadata d (which is a dictionary.) Let s be a `Sample` object. Then s.`bedFile` returns its `BedEFile` object; $s.\ell$ returns the value of its metadata labelled as ℓ in the dictionary; s.`bedFileUpdated`(vf) constructs a new `Sample` object with `BedEFile` object vf and the same metadata as s; and s.`metaUpdated`(d) constructs a new `Sample` object with metadata d and the same `BedEFile` object as s.

A sample file is an on-disk list of samples. Sample files are represented by the class `SampleEFile`, a subclass of `EFile`. This class also has two main constructors: `OnDiskSampleEFile`(f) for constructing a `SampleEFile` object representing the sample file f, and `TransientSampleEFile`(vs) for constructing a `SampleEFile` object representing an iterable or eiterator vs of `Sample` objects.

Overview of GMQL

GMQL [5] features a list of operators to create, store, and process datasets defined in common genomic data formats. Some GMQL operators mirror relational algebra operators, e.g. SELECT, PROJECT and GROUP. Some GMQL operators are genomic-specific, e.g. MAP, JOIN, and COVER.

GMQL is optimized for sample files containing many samples, with each sample having a large BED file. A GMQL query is decomposed into metadata and region operations. Metadata operations are evaluated before region operations. Usually, the effect of a metadata operation is to filter or remove samples from the input. Thus, only samples contributing to the result are passed to data loaders. For region operations, GMQL achieves high performance by binning the genome into chunks and comparing different bins concurrently [4].

For benchmarking, we deploy GMQL on a local installation of Apache Spark, which simulates a small cluster on a single multicore machine. We refer to this as the GMQL *command-line interface*, or CLI. The machine is a laptop with 2.6 GHz 6-Core i7, 16 GB 2667 MHz DDR4, 500 GB SSD.

Synchrony emulation of GMQL

Many queries have common idiomatic structures, which can be abstracted as re-usable query operators. GMQL operators are examples of these. We use Synchrony iterator to implement a Scala/Python library, Synchrony GMQL, that replicates all GMQL operators; i.e. Synchrony iterator is used instead of GMQL's binning strategy. Two examples (MAP and COVER) are provided below, to illustrate the ease and succinctness of the emulation. Full details of the emulation of all GMQL operators are at `www.comp.nus.edu.sg/~wongls/projects/synchrony`.

Emulation of MAP

Consider the GMQL MAP query, MAP() $\mathcal{U}$ $\mathcal{V}$, where $\mathcal{U}$ and $\mathcal{V}$ are SampleEFile objects. For each `Sample` s in $\mathcal{U}$, each `Sample` t in $\mathcal{V}$, and each `Bed` entry x in s.`bedFile`, this query counts the number of `Bed` entry y in t.`bedFile` whose locus overlaps with the locus of x. If naively executed, the time complexity of this query is $\Omega(|\mathcal{U}| \cdot |\mathcal{V}| \cdot m^2)$, assuming each BED file has m entries.

It has a very succinct and far more efficient implementation below as `maps`$(\mathcal{U}, \mathcal{V})$, which forms every pair of samples and processes the BED files of the pair by `mapr`(X, Y) to count overlaps using Synchrony iterator; see the four lines of codes delineated as the function `synchro`(xs, ys). By Theorem 1, despite its simplicity, this implementation has complexity $O(|\mathcal{U}| \cdot |\mathcal{V}| \cdot (k+1)m)$, when no region overlaps more than $k << m$ other regions. Notably, the linear part $(k+1)m$ is achieved without using any specialized interval indices (e.g. [3].)

```
def bf(y,x): return BedPred(y,x).isBefore()
def cs(y,x): return BedPred(y,x).overlap(1)

def mapr(X,Y):
  def synchro(xs,ys):
    for x in xs:
      ss = lambda y: BedPred(y,x).sameStrand()
      vs = filter(ss, ys.syncedWith(x,bf,cs))
      yield Bed({**x.dict(),'count': len(vs)})
  xs,ys = (X.eiterator(),Y.eiterator())
  tr = synchro(xs,ys)
  cl = lambda : (xs.close(),ys.close()))
  return TransientBedEFile(EIterator(tr,cl))
```

```
def maps(U, V):
  def aux(S,T):
    for s in S:
      for t in T:
        b = mapr(s.bedFile,t.bedFile)
        yield s.bedFileUpdated(b)
  S,T = (U.serialized(),V.serialized())
  it = aux(S,T)
  cl = lambda : it.close()
  return TransientSampleEFile(EIterator(it,cl))
```

Emulation of COVER

Consider the GMQL query COVER(n, m) $\mathcal{U}$, where $n, m > 0$ and $\mathcal{U}$ is a `SampleEFile`. Conceptually, it produces "maximal" regions in the underlying genome that each overlaps n to m number of regions in the BED files of the samples in $\mathcal{U}$.

It is implemented below as `covers`(`minmax`,$\mathcal{U}$), where `covers`(`betw`(n, m),$\mathcal{U}$) realizes the query COVER(n, m) $\mathcal{U}$, while `covers`(`atleast`(n),$\mathcal{U}$) realizes COVER(n, ANY) $\mathcal{U}$.

```
def atleast(n): return lambda i: n <= i
def betw(n,m): return lambda i: n <= i <= m

def coverr(minmax,XS):
  # mm comprises all regions in XS.
  first,rest = (XS[0],XS[1:])
  mm = first.mergedWith(*rest).serialized()

  # rf comprises start and end positions in mm
  def mp(mm,f):
    it = mm.eiterator().map(f)
    tf = TransientBedFile(it)
    return tf.serialized()
  ss = mp(mm,lambda r: r.start())
  ee = mp(mm,lambda r: r.end())
  rf = ss.mergedWith(ee.sorted)
         .distinct().serialized()
  ss.destruct(); ee.destruct()

  # make histogram of regions, histo. each bar
  #   in histo is a segment, with a count on
  #   no. of regions in mm overlapping it.
  # omit segments whose count fails minmax.
  def synchro(minmax,rf,mm):
    for x in rf:
      vs = mm.syncedWith(x,bf,cs)
      ht = len(vs)
      if ht == 0 or not(minmax(ht)): continue
      en = min([v.chromEnd for v in vs])
      yield Bed({'chrom': x.chrom,
                 'chromStart': x.chromStart,
                 'chromEnd': en,
                 'name': "", 'score': 0,
                 'strand': ".", 'count': ht})
  hh = synchro(minmax,rf,mm)
  ch = lambda : (rf.destruct(),mm.destruct())
  histo = EIterator(hh,ch)

  # concatenate adjoining segments in histo.
  def concat(histo):
    aj = lambda y,x: BedPred(y,x).overlap(0)
    while histo.hasNext():
      acc = histo.next()
      while (histo.hasNext() and
             aj(acc,histo.peekahead(1)[0])):
        nxt = histo.next()
        acc.chromEnd = nxt.chromEnd
      yield acc
  eit = concat(histo)
  cl = lambda : histo.close()
  return TransientBedEFile(EIterator(eit,cl))

def covers(minmax,U):
  b = coverr(minmax,[s.bedFile for s in U])
  return TransientSampleEFile([Sample(b,{})])
```

Here, `covers` first extracts `XS` = [s.`bedFile for` s `in` $\mathcal{U}$], which are the BED files associated with the samples in $\mathcal{U}$. Then all the regions in these BED files are extracted in a new BED file `mm` and all the start and end positions of these regions are extracted into another new BED file `rf`. A Synchrony iterator is then used in the function `synchro`(`minmax`,`rf`,`mm`) to generate for each x in `rf`—where x is overlapped by i number of regions in `mm`, and i satisfies the constraint `minmax`—a new region with the same start position as x but with its end position set to the nearest end position among the regions in `mm` that overlap x. This list of new regions is denoted as `histo` in the program. Finally, the function `concat`(`histo`) merges regions in `histo` that overlap each other, to produce the maximal regions desired.

All the steps above are linear in the total number of regions in all the samples, except for a substep needed in producing `rf`: The substep is the sorting of end positions (`ee`), which has the usual linearithmic time complexity of sorting. However, this linearithmic component is masked by disk access when processing large BED files. I.e., the overall time performance observed in practice is very close to linear. Again, this is achieved without using any specialized genomic interval indices (e.g. [3].)

Performance comparisons

To show that Synchrony GMQL, which is based on Synchrony iterator, has similar or better performance than GMQL CLI, we have chosen four representative operators: MAP, COVER, JOIN, and SELECT. The first three showcase the power of Synchrony iterator when processing large genomic data files, while the last shows that the emulation is also efficient for GMQL operators which do not need Synchrony iterator.

For MAP, the GMQL query chosen is MAP() $\mathcal{U}$ $\mathcal{V}$, which we saw earlier. The equivalent Synchrony GMQL query is `mapS`(`mapR`())($\mathcal{U}, \mathcal{V}$). Note that, in contrast to the `maps` and `mapr` functions shown earlier, `mapS` and `mapR` are Synchrony GMQL functions that fully emulate MAP in all its parameters.

For COVER, the chosen query is COVER(1, ANY) $\mathcal{U}$. Its Synchrony GMQL equivalent is `coverS`(`coverR`(`atleast`(1)))($\mathcal{U}$). Different from the `covers` and `coverr` functions shown earlier, `coverS` and `coverR` are Synchrony GMQL functions that fully emulate COVER in all its parameters.

For JOIN, the GMQL query chosen is JOIN(DL(0); output: int) $\mathcal{U}$ $\mathcal{V}$. For each `Sample` s in $\mathcal{U}$, each `Sample` t in $\mathcal{V}$, this query searches for all regions in t.`bedFile` which have distance less than 0 to any region in s.`bedFile`; i.e. it looks for regions in `Sample` t's BED file that overlap with regions in `Sample` s's BED file. This query outputs the intersection (viz. overlapping area) of each pair of matching regions. The equivalent in Synchrony GMQL is `joinS`(`joinR`(`Overlap`(1), `output` = `intersect`))($\mathcal{U}, \mathcal{V}$). Here, `joinS`(f)($\mathcal{U}, \mathcal{V}$) forms all possible pairs of `Sample` objects in ($\mathcal{U}, \mathcal{V}$), and applies f to each of these

pairs; `joinR(Overlap(1), output = intersect)` is the Synchrony iterator-based function for joining two BED files, where the `Overlap(1)` predicate makes the semantic intention more clear.

For SELECT, the GMQL query chosen is SELECT(region: (chr==chr1 OR chr==chr2)) $\mathcal{U}$. For every `Sample` s in $\mathcal{U}$, this query selects all regions found on the first or second chromosome of s.`bedFile`. The equivalent in Synchrony GMQL is `onRegion(selectR(lambda` r : r.`chrom == "chr1" or` r.`chrom == "chr2"))`$(\mathcal{U})$. Here, $onRegion(f)(\mathcal{U})$ applies f to s.`bedFile` for each s in $\mathcal{U}$, and `selectR(lambda` r : r.`chrom == "chr1" or` r.`chrom == "chr2"`) selects regions on chr1 or chr2.

Performance is compared based on the criteria below.

1. *Execution time.* The queries are run on 9 datasets, cf. the Datasets section, spanning the two dimensions of number of samples and number of lines (i.e. regions) per sample. Execution time includes writing query results to disk.
2. *Time complexity.* Each query is also run on a second group of datasets. These alternative test cases are meant to show how the time complexity (viz., execution time) increases with respect to either the number of samples, or the number of regions per sample, when the other measure is fixed.
3. *Memory usage.* Scala executes on top of the Java Virtual Machine (JVM). To verify that Synchrony GMQL maintains its performance even when a low amount of memory is allocated to the JVM, the queries are run two times; once by allocating 2GB of memory to the JVM, and another time by allocating 128MB of memory.

As GMQL CLI is based on Scala, we compare our Synchrony GMQL implementation in Scala with it. Our Scala implementation can be run in strictly sequential mode and also in sample-parallel mode (In this mode, each BED file is processed sequentially, but the BED files of different samples are processed in parallel on all 6 cores of the test machine.) We also show the performance of our Synchrony GMQL implementation in Python for information purpose; it is expected to be an order or two of magnitude slower as it is interpreter-based.

Datasets

Table 1. describes all datasets used in the above comparisons. There are 9 reference datasets, named SS through BB, that provide examples of different type of input data, viz., either with large amount of samples, many regions per samples, or a combination of both. All regions come from transcription factor ChIP-Seq experiments (i.e., regions are TF binding sites.) Data was extracted from two public repositories, ENCODE [8] and Cistrome [10]. Synthetic metadata is created to match each sample. The name labeling provides a cue of the contents. The first letter (S, M, or B) refers to the number of samples in the dataset (1 sample, 10 samples, or 100 samples). The second letter refers to the number of regions in each sample (1,000 regions, 10,000 regions, or 100,000 regions). E.g., MS is a dataset with 10 samples, each containing 1,000 regions. 13 additional datasets (s1l1000 through s1l1000000) are used for two different scaling studies wrt the number of samples (at fixed number of regions), and wrt the number of regions (at fixed number of samples). For the use-case described in the Discussion section, we use a dataset of 10 ChIP-seq experiments from cell line MCF-7, and a set of gene regions from RefSeq. We synthetically increased the number of genes by replicating the regions 10 times, to better simulate larger scenarios.

Table 1. Datasets used in Synchrony GMQL performance testing.

Name	Sample	Regions	Size [MB]
SS	1	1000	0.08
SM	1	10000	0.86
SB	1	100000	9.58
MS	10	1000	0.95
MM	10	10000	8.74
MB	10	100000	96.00
BS	100	1000	14.42
BM	100	10000	86.33
BB	100	100000	696.35
s1l1000	1	1000	0.06
s5l1000	5	1000	0.34
s10l1000	10	1000	0.67
s20l1000	20	1000	2.13
s50l1000	50	1000	5.15
s75l1000	75	1000	7.52
s100l1000	100	1000	10.20
s1l1	1	1	0.00
s1l10	1	10	0.00
s1l100	1	100	0.01
s1l10000	1	10000	0.60
s1l100000	1	100000	7.09
s1l1000000	1	1000000	71.25
MCF-7	10	424096	24.90
ncbiRefSeqGenes10X	1	314364	49.70

Discussion

Use case: TFBS found in promoters

Consider this fairly complex query: *Given a set of transcription factor binding sites (TFBS) from a single transcription factor (TF), and a set of genes of interest, all of which are located on the positive strand of human DNA, identify all TFBS that are found in the promoters of said genes.* For simplicity, assume that both the TFBS and the genes sample files, `TL` and `GL`, are already sorted according to chromosome, start, end, and strand. Assume also that for each gene, the promoter is located between 2,000 bp upstream and 1,000 bp downstream of its transcription start site (TSS), and the TSS is located at the very first base of the gene.

In GMQL, this query is written below using a combination of PROJECT, MAP, and SELECT: PROJECT takes care of resizing each gene to locate the promoter area; MAP counts how many promoters intersect each TFBS; and SELECT filters out all TFBS that do not match any promoters.

```
PLS = SELECT(region: strand == +) GL;
PRM = PROJECT(region_update:
          start as start - 2000,
          stop as start + 1000) PLS;
MAT = MAP() TL PRM;
RES = SELECT(region: count >0) MAT;
```

The same query is expressed below in Python via Synchrony GMQL functions mirroring the GMQL operators.

```
pls = selectR(lambda r: r.strand == "+")
prm = projectR({
  'chromStart': lambda r: r.chromStart - 2000,
  'chromEnd': lambda r: r.chromStart + 1000})
```

```
PRM = onRegion(lambda r: prm(pls(r)))(GL)
MAT = mapS(mapR())(TL, PRM)
RES = onRegion(selectR(lambda r: r.count>0))(MAT)
```

We test the two queries using MCF-7 as the TFBS list `TL` and ncbiRefSeqGenes10X as the gene list `GL`; cf. Datasets section. Sequential Synchrony GMQL produces its results in 2.75 seconds (average of 10 runs, stdev = 0.09), while parallel Synchrony GMQL does this in 1.94 seconds (average of 10 runs, stdev = 0.07.) GMQL CLI finishes in 56.14 seconds (average of 10 runs, stdev = 2.26.) Python Synchrony GMQL completes in 35.44 seconds (average of 10 runs, stdev = 0.88.) So, Scala Synchrony GMQL is 20 to 30 times faster than GMQL CLI on this example, while the Python version is 1.58 times faster.

Naturally embedded genomic queries

So far, our queries all have `SampleEFile` as input and output. This is because we are emulating GMQL: It is a standalone query language which can only take `SampleEFile` as input and produce `SampleEFile` as output. For more complex processing, the input/output is often needed to be something else. So, one has to switch between GMQL and a more general host programming language, and suffers the inconvenience of impedance mismatch. Since Synchrony GMQL is a natural embedding of GMQL-like features into Scala/Python, this impedance mismatch disappears.

The short Python program below illustrates this.

```
[ (p, tf, s)
    for p in PRM
    for tf in TL
    for b in [mapr(tf.bedFile, p.bedFile)]
    for f in [selectR(lambda r: r.count > 0)(b)]
    for s in [f.serialized()] ]
```

In this program, which also implements the "TFBS found in promoters" query, `PRM` and `TL` are as defined earlier and evaluate to `SampleEFile`; and `mapr` is as defined earlier (in the section on emulating MAP) and evaluates to `BedEFile`. The output is a list of triples $(\texttt{p}, \texttt{tf}, \texttt{s})$, where `p` is a `Sample` from `PRM`, `tf` is a `Sample` from `TL`, and `s` is a `BedEFile` containing those regions in `tf.bedFile` that overlap some regions in `p.bedFile`. The twist here is that the output is no longer a `SampleEFile`, but is the triple $(\texttt{p}, \texttt{tf}, \texttt{s})$. In a situation where `GL` (and thus `PRM`) contains multiple samples (each being a separate list of gene promoters), outputting such a triple is natural for keeping track of the provenance of `s` (i.e. it is derived from which `p` and `tf`.) This illustrates the free mixing of Synchrony GMQL operations and results with any other features in the host programming language. This natural embedding of efficient genomic querying capability into a host programming language brings great convenience in more complex data processing and analysis pipelines.

Execution time comparisons

Figure 1 shows the results of running the MAP, JOIN, COVER, and SELECT on datasets SS through BB. For all four queries, Synchrony GMQL outperforms GMQL CLI by large margins on nearly all datasets. The exceptions are MAP and JOIN on BB, where GMQL CLI outperforms sequential Synchrony GMQL. On BB, 10000 pairs of samples are compared. GMQL CLI is able to do these in parallel on all 6 cores of the test machine, while sequential Synchrony GMQL is strictly sequential. Parallel Synchrony GMQL processes samples in parallel

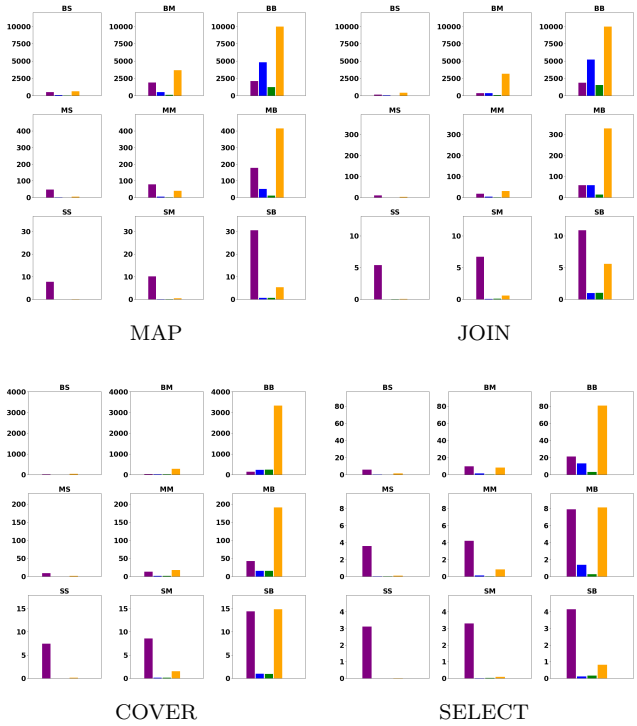


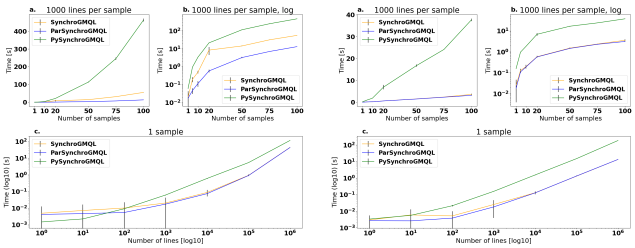
Fig. 1. Timing of GMQL CLI and Synchrony GMQL for MAP, JOIN, COVER, and SELECT queries on 9 reference datasets SS, ..., BB. Time in seconds. Each average is done over 30 runs, except for BM and BB, which are done up to 5 runs due to time constraints. *Purple:* GMQL CLI. *Blue:* Sequential Synchrony GMQL. *Green:* Parallel Synchrony GMQL. *Yellow:* Python Synchrony GMQL.

but BED files sequentially. Thus, parallel Synchrony GMQL is faster than sequential Synchrony GMQL on all queries on multiple samples, but has similar efficiency on the COVER query as COVER first merges the BED files in the samples into a single BED file. GMQL CLI is parallel at both sample and BED file levels, via binning [4]. GMQL CLI's poorer performance suggests that the brute-force parallelism provided by a SPARK implementation of quadratic algorithms on MAP and JOIN cannot compete with Synchrony GMQL's Synchrony iterator, until the number of samples and/or regions is sufficiently high to keep all 6 cores on the test machine fully busy.

Time complexity as a function of sample and region counts

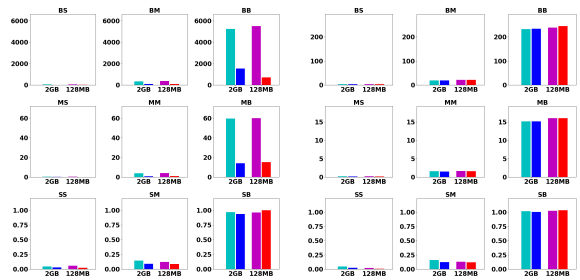
The query $\texttt{joinS}(\texttt{joinR}(\texttt{Overlap}(1), \texttt{output} = \texttt{intersect}))(\mathcal{U}, \mathcal{V})$ compares each U in $\mathcal{U}$ to each V in $\mathcal{V}$, and computes the intersection of each region in U to each region in V. Assuming each region in V overlaps at most k regions in U, and each BED file has at most m regions, the intersections of regions are computed by a Synchrony iterator in $O((k+1)m)$ time complexity; cf. Theorem 1. So the overall time complexity is $O(|\mathcal{U}| \cdot |\mathcal{V}|(k+1)m)$. This theoretical upperbound is consistent with the charts for JOIN in Figure 2, when one ignores the subseconds part of the figure caused by fluctuations due to systems start-up; the quadratic component $|\mathcal{U}| \cdot |\mathcal{V}|$ in the upper charts, the linear component $(k+1)m$ in the lower chart.

The query $\texttt{coverS}(\texttt{coverR}(\texttt{atleast}(1)))(\mathcal{U})$ has theoretical time complexity which is linearithmic in the total number of regions in $\mathcal{U}$, and is likely linear in practice (when the linearithmic component is masked by disk access.) This is consistent



JOIN COVER

Fig. 2. Performance analysis of Sychrony GMQL for JOIN and COVER. **a.** Timing as a function of sample count; each sample has a fixed number of regions (1,000.) **b.** Same as a., but time in log-scale. **c.** Timing as a function of region count; each data point corresponds to one sample with a given number of regions. Average of 30 runs (error bars = stdev.)



JOIN COVER

Fig. 3. Timing of sequential and parallel Synchrony GMQL for JOIN nd COVER in two situations: 2GB allocated to the JVM (cyan and blue, resp. sequential and parallel), and 128MB allocated (margenta and red, resp.) All times (y-axis) in seconds.

with the observation in the charts for COVER in Figure 2 where, modulo some fluctuations at very low amounts of data and at systems start-up, linearity is observed.

The query `mapS(mapR())`$(\mathcal{U}, \mathcal{V})$ also has theoretical time complexity $O(|\mathcal{U}||\mathcal{V}|(k+1)m)$, with scaling behaviours similar to JOIN. The query `onRegion(selectR(lambda` r : r`.chrom == "chr1" or` r`.chrom == "chr2"))`$(\mathcal{U})$ has theoretical time complexity linear in the total number of regions in $\mathcal{U}$ as well. Both their scaling charts are omitted as these add no further insight.

Execution times in constrained memory situation

Figure 3 shows that the execution time of JOIN and COVER in Synchrony GMQL do not differ much when a lot (2GB) or little (128MB) memory is given to the JVM. Similar memory efficiency is observed for MAP, and SELECT in Synchrony GMQL; charts omitted. So, Synchrony iterator does not consume much memory even when there are large amounts of input data.

Conclusion

Synchrony iterator is a paradigm for expressing, in easy-to-understand comprehension syntax, efficient genomic data processing algorithms that require synchronized iterations on two or more streams of ordered genomic regions. We have demonstrated how Synchrony iterators can be used to emulate the powerful genomic query language GMQL in a succinct and efficient way. We have shown that the resulting emulation, Synchrony GMQL, is more efficient than a local installation of GMQL. Thus, Synchrony iterator is an elegant solution to impedance mismatch issues that often arise when designing and implementing genomic data processing pipelines.

Synchrony iterator is designed to keep the technicalities of synchronized iterations from the user, needing only simple definitions of the "is before" (denoted as `bf` in the text) and the "can see" (`cs`) predicates to function correctly and efficiently. While Synchrony iterator is not specifically designed for bioinformaticians, its use is natural in genomic dataset processing. This is because genomic regions are naturally ordered based on their loci, and because it is often the case that researchers are interested in questions of proximity between loci.

Funding

SP and LW were supported by National Research Foundation, Singapore, under its Synthetic Biology Research and Development Programme (Award No: SBP-P3); and by Ministry of Education, Singapore, Academic Research Fund Tier-1 (Award No: MOE T1 251RES1725). In addition, VT was supported in part by a Kwan Im Thong Hood Cho Temple Visiting Professorship, and LW was supported in part by a Kwan Im Thong Hood Cho Temple Professorship. SC and PP were supported by ERC AdG 693173 "Data-driven Genomic Computing (GeCo)." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors, and do not reflect the views of these grantors.

References

1. P. Buneman, L. Libkin, D. Suciu, et al. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
2. G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of ACM-SIGMOD 84*, pages 316–325, 1984.
3. J. Feng, A. Ratan, and N. C. Sheffield. Augmented interval list: A novel data structure for efficient genomic interval search. *Bioinformatics*, 35(23):4907–4911, 2019.
4. A. Gulino, A. Kaitoua, and S. Ceri. Optimal binning for genomics. *IEEE Transactions on Computers*, 68(1):125–138, 2018.
5. M. Masseroli, A. Canakoglu, P. Pinoli, et al. Processing of big heterogeneous genomic datasets for tertiary analysis of next generation sequencing data. *Bioinformatics*, 35(5):729–736, 2019.
6. S. Neph, M. S. Kuehn, A. P. Reynolds, et al. BEDOPS: High-performance genomic feature operations. *Bioinformatics*, 28(4):1919–1920, 2012.
7. K. D. Pruitt, T. Tatusova, G. R. Brown, and D. R. Maglott. NCBI Reference Sequences (RefSeq): Current status, new features, and genome annotation policy. *Nucleic Acids Research*, 40(D1):D130–D135, 2012.
8. C. A. Sloan, E. T. Chan, J. M. Davidson, et al. ENCODE data at the ENCODE portal. *Nucleic Acids Research*, 44(D1):D726–D732, 2016.
9. L. Wong. Addressing an intensional expressiveness gap of comprehension syntax, 2021. Manuscript available from `https://www.comp.nus.edu.sg/~wongls/projects/synchrony/v5-wls-natural2021.pdf`.
10. R. Zheng, C. Wan, S. Mei, et al. Cistrome data browser: Expanded datasets and new tools for gene regulatory analysis. *Nucleic Acids Research*, 47(D1):D729–D735, 2019.