BioHaskell in Reality

A Final Project Project Report
Presented to Prof. Thomas Austin

Department of Computer Science
San Jose State University

By
AArohi Chopra (aarohi.chopra@sjsu.edu)
Linh Le (linh.le01@sjsu.edu)
May 15, 2023

Table of Contents

I. Project Objective

Bioinformatics is an exponentially growing field driven with ever increasing needs in the fields of genomics and sequence alignment. Dealing with problems involving biological data can be extremely tricky since single genomes can include up to 3 billion base pairs(for humans). This calls for creation of pipelines with massive compute and run as efficiently as possible. It also needs to ensure that no artifact is being introduced in the results because single mutations can lead to misdiagnosis impacting patient outcomes. To minimize the introduction of artifacts researchers use various quality control and DNA analysis techniques. These can be extremely time consuming and computationally challenging due to the huge amounts of data involved.

The field of bioinformatics can significantly benefit from utilizing functional programming to solve the above mentioned issues. The project tries to perform DNA analysis and compare 3 different languages to assess if a functional approach shows a significant edge over the others. Python is the most popular language used for any bioinformatics tools, implementing the analysis steps in Haskell, Python and Java can help evaluate and compare the respective performance characteristics. Haskell also has its own library to perform biological analysis.

II. History and Background

BioHaskell is a domain-specific language (DSL) designed for computational biology and bioinformatics. It combines the power and expressiveness of Haskell, a functional programming language, with specific features and libraries tailored to address the unique challenges of biological data analysis and modeling. The development of BioHaskell can be traced back to the early 2000s when the field of bioinformatics was experiencing rapid growth, with an increasing need for efficient and reliable tools for data analysis

and manipulation. Haskell, known for its strong static typing, lazy evaluation, and expressive type system, was seen as a suitable language for addressing the complex and dynamic nature of biological data.
Strong type system enables strong typing and also prevents common programming errors. Since data integrity is crucial for biological data it prevents introduction of artifacts providing more accurate results. This is also maintained because of immutable data structures implemented by haskell. The immutability also allows referential transparency, which simplifies reasoning about the behavior of functions and facilitates parallel and concurrent programming. Lazy evaluation is another feature extremely useful for genomic data which benefits from demand driven execution and also makes efficient memory usage. Haskell also provides powerful higher-order functions, type classes, and monads, enabling the creation of elegant and reusable abstractions. These abstractions can simplify complex bioinformatics algorithms, improve code modularity, and facilitate code reuse. Haskell's type system also supports strong abstraction mechanisms, allowing the definition of precise and generic types that capture the semantics of bioinformatics concepts.
Over the years, the BioHaskell community has grown, contributing to the development of libraries, tools, and resources specifically tailored for computational biology and bioinformatics. These include modules for sequence analysis, genomics, protein structure prediction, phylogenetics, and many other areas of biological research. Its functional programming paradigm and rich ecosystem make it a valuable resource for researchers and developers in the field, offering a powerful and flexible platform for analyzing and modeling biological data. Even though the project tried to initially use Biohaskell to program the Haskell files it posed significant challenges due to installation issues.
Haskell's focus on type safety, immutability, purity, expressiveness, and efficient concurrency make it a powerful language for developing reliable and performant bioinformatics applications. Its functional programming paradigm and strong ecosystem of libraries make it well-suited for tackling the complex computational challenges in bioinformatics research and analysis.

III. Technical Approach/Methodology

We present four applications:

- **Kmer count**: it is a fundamental technique in Bioinformatics that involves counting the number of possible subsequences of specified length k in raw reads. It is used in genome assembly, error detection and correction and even in taxonomic classification.
- **AT and CG contents and percentages**: used to calculate the GC-content, which is the ratio of guanine (G) and cytosine (C) nucleotides to the total number of nucleotides in a DNA sequence. GC-content is an important factor in various biological processes, including DNA stability, gene regulation, and evolutionary studies.
- **T->U transcription:** in DNA, where thymine (T) is replaced by uracil (U) during RNA synthesis. The T->U transcription in DNA plays a critical role in accurately transferring genetic information, enabling diverse RNA functions, distinguishing RNA from DNA, supporting RNA modifications, and providing regulatory control over gene expression.
- **Fasta and Fastq Readers**: which are file formats used in bioinformatics for storing nucleotide or protein sequences and their associated quality scores. FASTA and FASTQ readers provide efficient data organization, enable sequence retrieval, ensure compatibility with bioinformatics tools, facilitate quality assessment, streamline analysis workflows, and support data sharing.

For the first 3 applications, we use three different languages including Haskell, Python, and Java to measure the time performance of each function.

The code for first three application are provided below:

Code snippet for K-Mer counting

Haskell

```haskell
countKmers :: Int -> String -> [(String, Int)]
countKmers k str = toList $ fromListWith (+) [(take k $ drop i str, 1) | i <- [0..length str - k]]
```

Java

```java
public static List<Map.Entry<String, Integer>> countKmers(int k, String str) {
    Map<String, Integer> counts = new HashMap<>();
    for (int i = 0; i <= str.length() - k; i++) {
        String kmer = str.substring(i, i + k);
        counts.merge(kmer, 1, Integer::sum);
    }
    List<Map.Entry<String, Integer>> tuples = new ArrayList<>(counts.entrySet());
    Collections.sort(tuples, Map.Entry.comparingByValue(Comparator.reverseOrder()));
    return tuples;
}
```

Python

```python
def count_kmers(k, seq):
    counts = defaultdict(int)
    for i in range(len(seq) - k + 1):
        kmer = seq[i:i+k]
        counts[kmer] += 1
    tuples = sorted(counts.items(), key=lambda x: x[1], reverse=True)
    return tuples
```

Code snippet for AT-CG Counting

Haskell

```haskell
countDinucleotides :: String -> (Int, Int)
countDinucleotides dna = go dna (0,0)
    where go [] counts = counts
          go [_] counts = counts
          go (x:y:xs) (atCount,cgCount)
            | [x,y] == "AT" = go (y:xs) (atCount+1,cgCount)
            | [x,y] == "CG" = go (y:xs) (atCount,cgCount+1)
            | otherwise = go (y:xs) (atCount,cgCount)
```

Java

```java
for (int i = 0; i < dna.length() - 1; i++) {
    String dinucleotide = dna.substring(i, i + 2);
    if (dinucleotide.equals("AT")) {
        atCount++;
    } else if (dinucleotide.equals("CG")) {
        cgCount++;
    }
}
```

Python

```python
for i in range(len(dna)-1):
    dinucleotide = dna[i:i+2]
    if dinucleotide == "AT":
        at_count += 1
    elif dinucleotide == "CG":
        cg_count += 1
```

Code snippet for T -> U Transcription

Haskell

```haskell
transcribe :: Char -> Char
transcribe 'T' = 'U'
transcribe nucleotide = nucleotide
```

Java

```java
public String getRNASequence() {
    // Replace all occurrences of T with U
    return sequence.replaceAll("T", "U");
}
```

Python

```python
def get_rna_sequence(self):
    # Replace all occurrences of T with U
    return self.sequence.replace("T", "U")
```

IV. Requirements

These requirements are available to SJSU students at no cost.

For Fasta and Fastq Readers in Python, we are required to download the bioPython library which is available for free online or can be downloaded via Pip command.

V. Results

Below are the result measures in units of time (milliseconds) for three applications to compare how fast each language executes a specific function.

We take DNA sequences of length 1,000,000 and apply it to all 3 functions.

K-Mer Counting

Milliseconds	Haskell	Java	Python
Average	<1	165.6	218.8

AT-CG Counting

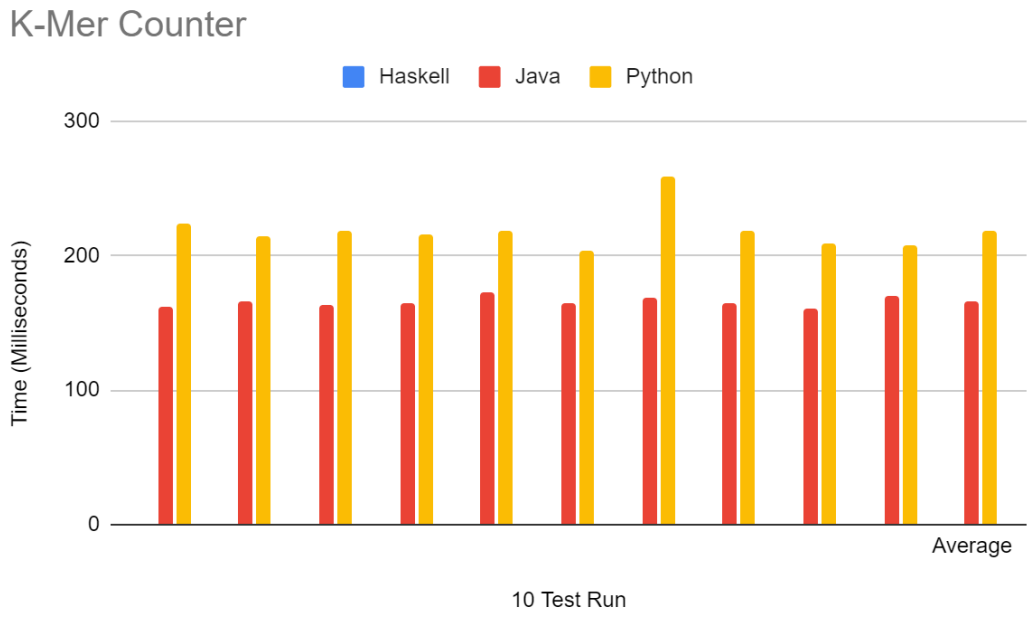
Milliseconds	Haskell	Java	Python
Average	<1	30.7	180.1

T -> U Transcription

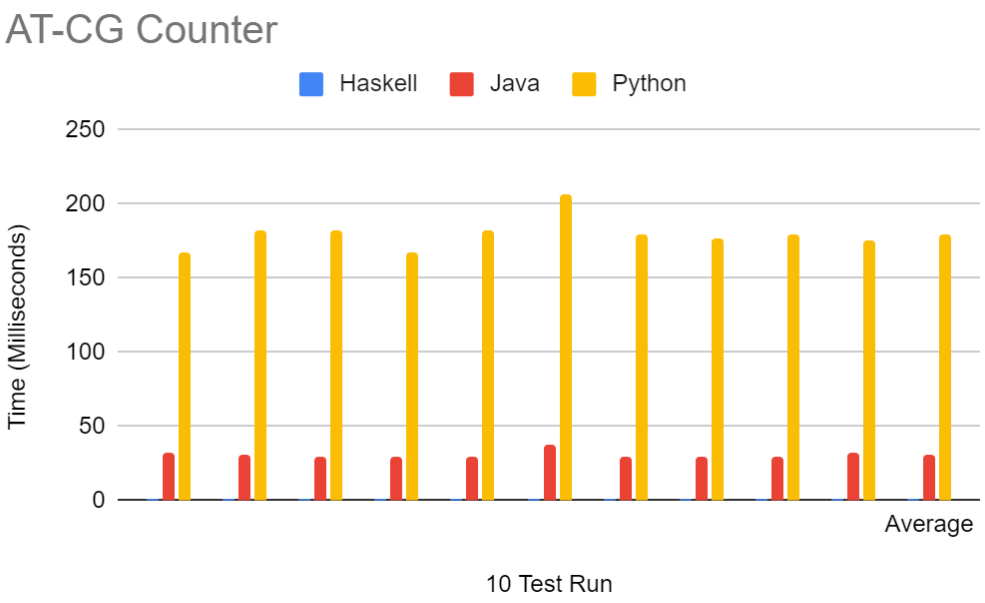
Milliseconds	Haskell	Java	Python
Average	282.1	59	56.7

The results show that Haskell is relatively fast when it comes to mathematical computation like K-Mer Counting and AT-CG Counting. These applications usually involve computing a large amount of numbers and since Haskell is a functional language, it is the best language out of all 3 for this job.

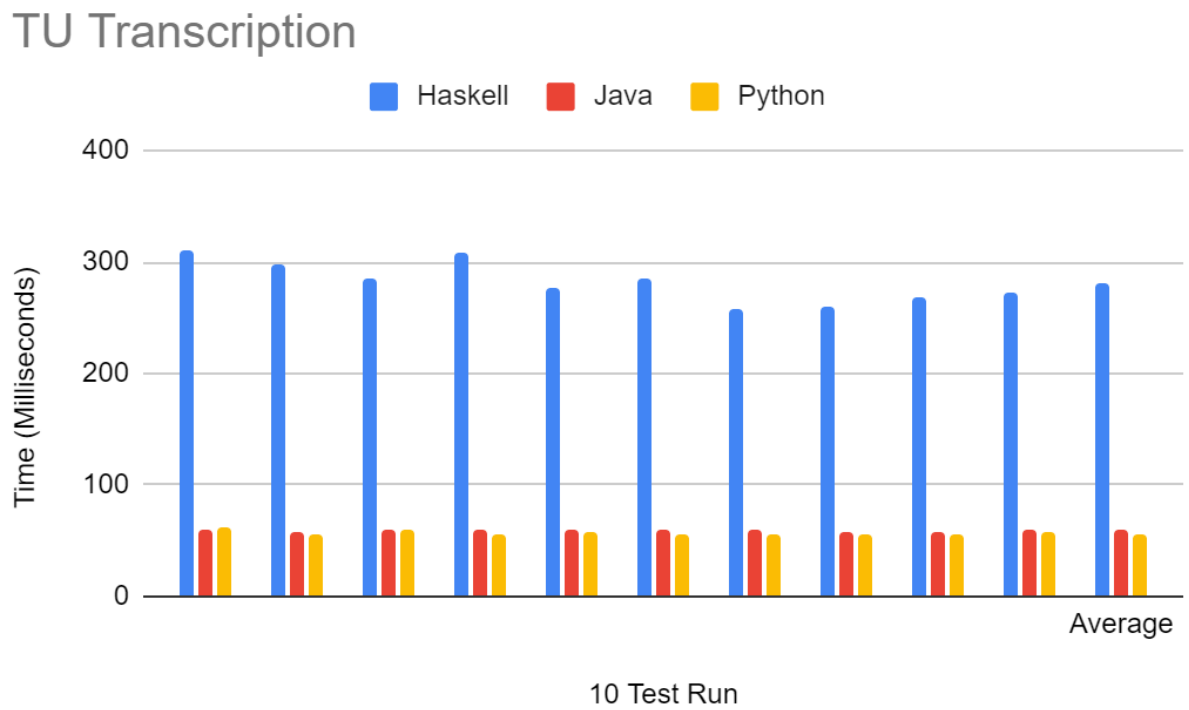
For T -> U Transcription, Haskell isn't very fast because this function requires string manipulation and Haskell isn't very strong when it comes to this area.



This is the chart to show the average time after 10 runs. As the result points out, Haskell beat the other 2 languages by a long shot.



Same result for AT-CG Counter. Haskell is super fast when it come to counting.



For TU Transcription, Haskell can be relatively slower when it comes to string manipulation compared to other programming languages due to its immutable nature and the way string are represented. Lazy evaluation introduces overhead when working with strings, as evaluating and manipulating individual characters or substrings require traversing the entire string, leading to potential space and time inefficiencies.

Fasta and Fastq Reader:

Language	Fasta File	Fastq File
Python	********** ENTRY NUMBER: 11 SEQUENCE ID: seq10 SEQUENCE: FDSWDEFVSKSVELFRNHPDTTRYVVKYRHCEGKLVLK Elapsed time: 0.02208423614501953 seconds (base) Aarohis-MacBook-Pro:python aarohichopra$ p	********** ENTRY NUMBER: 4567 SEQUENCE ID: e0292b9f-6d60-459b-b510- SEQUENCE: GGTGTGCTTCGTTCGATTTTCGAAGTA ATGCGGGTTCAAAATCCTCTATCTGAAAGATGGGCAAT SEQUENCE QUALITY: [4, 5, 8, 3, 3, 3, 9, 5, 5, 7, 6, 3, 5, 3, 12, 12, 13, 12 6, 27, 27, 25, 22, 15, 15, 20, 14, 6, 13, 14, 19, 19, 24, 25, 24, 10, 10, 6 8, 7, 4, 5, 5, 3, 3, 8, 12, 8, 5, 3, Elapsed time: 0.620847225189209 second
Java	********** ENTRY NUMBER: 11 SEQUENCE ID: seq10 SEQUENCE: FDSWDEFVSKSVELFRNHPDTTRYVVKYRHCEGKLVL Execution time: 34 milliseconds	********** ENTRY NUMBER: 4567 SEQUENCE ID: e0292b9f-6d60-459b-b510-3b552347d56f ru SEQUENCE: GGTGTGCTTCGTTCGATTTTCGAAGTAGGTTGATTTTATGGA TGCGGGTTCAAAATCCTCTATCTGAAAGATGGGCAATGAGCAGGAGTCACAA SEQUENCE QUALITY: +%&)$$$###'+/,&&#))*))%%&%$#&%$'&% 9++'(',.4-+++0<=)0)*(&$$%'+$%/:.03.*&03%()(%&&$$)-)8 Execution time: 397 milliseconds
Haskell	("seq10","FDSWDEFVSKSVELFRNHPDTTRYVVKYRH Total time: 0.002289s	FastqFile {seqID = "@161c457e-1c5e-4a09-9653 quence = "CATGCTACGTTCTGCTATCGTTTGATGTTTACGC AATATCAGCACCAACAGAAACACAAGACACCGACAACTTTCTTA .$)#%''$4$,233&///6.3)796$%'&$,%'%*,'')#\"$# FastqFile {seqID = "@e0292b9f-6d60-459b-b510 quence = "GGTGTGCTTCGTTCGATTTTCGAAGTAGGTTGAT TGCGGGTTCAAAATCCTCTATCTGAAAGATGGGCAATGAGCAGG -:;74)(/10253$$'<<:7005/'.*'+0'',(&1$(%'*%'1 Total time0.612952s

Surprisingly, Python, despite being the most popular language and having specialized libraries for bioinformatics, performed relatively worse in terms of performance. On the other hand, Haskell, despite

its rudimentary and unoptimized implementation, exhibited better performance. Java, with the assistance of specialized libraries that provide abstraction and parallelization, appeared to perform the best. It is worth noting that the basic Haskell version might not have achieved the same level of performance due to the absence of similar specialized libraries.

References

[1] Vissidarte, "Where and how can Haskell be used in the Biopharma Industry? what are the software/app dev-related pain points? what are holes that Haskell can fill in MATLAB, R and other popular languages in this space? how strong is Biohaskell, and how can it be improved?," *Reddit*, 13-Nov-2013. [Online]. Available: https://www.reddit.com/r/haskell/comments/1qki1w/where_and_how_can_haskell_be_used_in_the/. [Accessed: 10-Apr-2023].

[2] M. J. Gajda, "Applications," GitHub, 28-Oct-2020. [Online]. Available: https://github.com/BioHaskell/biohaskell.github.io/wiki/Applications. [Accessed: 10-Apr-2023].

[3] G. R, "A systematic approach to dynamic programming in bioinformatics," *Bioinformatics (Oxford, England)*, 16-Aug-2000. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/11099253/. [Accessed: 10-Apr-2023].

[4] K. Malde, E. Coward, and I. Jonassen, "Fast sequence clustering using a suffix array algorithm," *Bioinformatics (Oxford, England)*, 01-Jul-2003. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/12835265/. [Accessed: 10-Apr-2023].

[5] U. Köhler, "BioHaskell: Read Fasta File," *Stack Overflow*, 14-Feb-2014. [Online]. Available: https://stackoverflow.com/questions/21802526/biohaskell-read-fasta-file. [Accessed: 10-Apr-2023].

[6] Free Software Foundation, Inc, *Biocore*, 19-Sep-2011. [Online]. Available: https://malde.org/~ketil/biohaskell/biocore/. [Accessed: 10-Apr-2023].

[7] J. Hughes, Why Functional Programming Matters, The Computer Journal, Volume 32, Issue 2, 1989, Pages 98–107, https://doi.org/10.1093/comjnl/32.2.98