UNIVERSIDADE FEDERAL DO PARANÁ
SETOR DE CIÊNCIAS EXATAS
DEPARTAMENTO DE INFORMÁTICA
INFORMÁTICA BIOMÉDICA

Cláudio Torres Júnior

Desafios na Correção de Sequências Genômicas

Curitiba
2022

Cláudio Torres Júnior

Desafios na Correção de Sequências Genômicas

Trabalho de Conclusão de Curso apresentado na Disciplina Trabalho de Graduação do Curso de Informática Biomédica da Universidade Federal do Paraná, como exigência parcial para obtenção do grau de Bacharel em Informática Biomédica.

Orientador: André Ricardo Abed Grégio

Co-Orientador: Antônio Bernardo de Carvalho (UFRJ)

Curitiba
2022

Agradecimentos

Inclua seus agradecimentos aqui. Há vários exemplos na internet.

"Democracia é oportunizar a todos o mesmo ponto de partida.
Quanto ao ponto de chegada, depende de cada um".
(Fernando Sabino)

Resumo

Com o avanço das tecnologias de sequenciamento do genoma, o tamanho dos *reads* (sequência de bases nitrogenadas inferidas pelos sequenciadores) gerados foram ficando maiores, como os resultantes dos equipamentos de 3ª geração, favorecendo estudos de sequência de novo e a análise estrutural entre genomas. Em contraponto, os *reads* gerados por essas tecnologias possuem uma alta taxa de erro, podendo chegar a 25% no *PacBio* e 35% no *Nanopore*. Nesse sentido, métodos de correção de *reads* foram desenvolvidos para contornar esse problema, com um custo mais acessível. Com isso não é necessário a utilização de sequenciadores de gerações passadas e nem de outras técnicas que já estão disponíveis em versões mais novas do *PacBio* por exemplo, que acabam encarecendo o estudo. Neste trabalho, faz-se um estudo empírico dos desafios (em relação ao consumo de processamento e memória) da correção de sequência genômica dependentes de linguagens de alto nível. Para tanto, apresenta-se inicialmente os conceitos básicos por trás do funcionamento da correção genômica e implementa-se, em Python, um corretor originalmente escrito na linguagem Awk, de forma a se testar o impacto de criação de estruturas de dados compartilhadas e do uso de *multiprocessing/multithreading*. Por fim, foram realizados testes para mostrar os pontos fortes e fracos de se utilizar uma linguagem de alto nível para o problema de correção de *reads*, bem como foram discutidas outras abordagens que podem ser utilizadas no futuro para sanar as dificuldades encontradas no projeto atual.

Palavras-chave: sequenciamento de genoma, reads, correção de reads.

Sumário

1 INTRODUÇÃO 1
1.0.1 Objetivo e contribuição 2
1.0.2 Organização 3

2 REVISÃO DE LITERATURA 4

3 MATERIAL E MÉTODOS 6
3.1 Material 6
3.1.1 Conjunto de Dados 6
3.1.2 Recursos Computacionais 6
3.2 Métodos 6

4 RESULTADOS E DISCUSSÃO 11
4.0.1 Ideia Geral da Implementação 11
4.0.2 Testes Comparativos 11
4.0.3 Desafios 15

5 CONCLUSÃO 19

REFERÊNCIAS 20

1 Introdução

O sequenciamento de genomas é uma tarefa de grande relevância em várias áreas, podendo ser aplicado na análise de microbiomas, identificação de microorganismos e diagnóstico e tratamento de doenças. Como exemplo disso, temos a sua utilização durante a pandemia da Covid-19, em que diversos cientistas do mundo todo juntaram esforços para descobrirem a estrutura do vírus responsável (LABORATORY; DIAGNOSIS, 2021). Mesmo após o sequenciamento completo do Coronavírus no início, os esforços não diminuíram para que suas variantes pudessem ser caracterizadas e estudadas. A partir do sequenciamento, pesquisadores puderam descobrir a qual família viral ele pertencia, além de terem identificado o seu comportamento desde a entrada no organismo até a chegada às células. Com isso, medicamentos e vacinas foram desenvolvidas, propiciando a criação e aplicação de políticas públicas.

O sequenciamento genômico vem evoluindo ao longo dos anos, incorrendo em maior desempenho (via paralelismo) e barateamento de custos das leituras. Isso ajuda a avançar a área e propiciar capacidade de análise por parte de outros pesquisadores, que podem realizar suas atividades em países/ambientes com pouco financiamento para testes ou compra de maquinário adequado. No início, o processo de sequenciamento era realizado através do método de Sanger, mas as limitações inerentes a ele e o desafio de sequenciar o restante do genoma humano fomentaram o desenvolvimento de tecnologias de próxima geração. Além de mais rápidas e baratas, o sequenciamento resultante dessas novas tecnologias permitiu a obtenção de sequências maiores (PARK; KIM, 2016).

Atualmente, estamos presenciando os passos iniciais da quarta geração de métodos para o sequenciamento genômico (LIN; HUI; MAO, 2021), mas isso não significa que os da geração passada estão defasados. Se a taxa de erro precisa ser baixa e o comprimento da sequência gerada precisar estar em valores pequenos (aproximadamente entre 250 e 1000 pares de bases), métodos como o de *Sanger* (1ª geração) ou *Illumina* (2ª geração) podem ser escolhidos (caso o foco seja minimizar ao máximo a taxa de erro, o segundo deve ser escolhido, o que acarreta também em uma diminuição da sequência gerada). Métodos com essas características costumam ser mais caros, pois, dentre outros motivos, precisam de sequências pequenas (caso o estudo necessite sequenciar moléculas grandes, essas sequências precisam ser subdivididas em pedaços menores) que serão clonadas centenas de vezes para que o processo consiga, ao final, sobrepor os resultados de cada clone (os pertencentes a mesma sequência) e ter como resultado uma fita consenso. Por esse motivo, o uso de métodos de 1^a ou 2ª geração acabam sendo inviáveis dadas as necessidades de sequenciamento de certos estudos.

O contorno deste problema inclui a aplicação dos métodos da terceira geração, os quais foram desenvolvidos para o estudo de moléculas maiores e até genomas completos,

como o *Single-Molecule real-time (SMRT) sequencing* da *Pacific Biosciences of California, Inc.* (*PacBio*. Nome também utilizado para se referir ao método) e o *Oxford Nanopore*. Levando em consideração o *SMRT sequencing*, esse método utiliza-se de longas cadeias de base para serem lidas (*long reads*), o que é de grande importância para a montagem e estudos de sequências de novo e a análise estrutural entre genomas (GOODWIN et al., 2015). Além disso, adota um método em tempo real para o sequenciamento, o que promove uma leitura mais rápida do que os métodos utilizados nos sequenciamentos de segunda geração (SGS) (RHOADS; AU, 2015), além de não necessitar que a sequência seja clonada diversas vezes. Daí o nome *Single-Molecule real-time*.

A realização do processo de sequenciamento ocorre da seguinte forma: Primeiro, a dupla fita de DNA a ser sequenciada é isolada e adaptadores são ligados em suas pontas, formando uma sequência circular, chamada de *SMRT bell*. Após isso, um primer e a enzima polimerase são adicionados à essa sequência e colocados em células do equipamento, chamados *SMRT cell*, que possuem milhares de "aberturas" chamadas *Zero-Mode Waveguides* (ZMW) onde os *SMRT bell* se encaixam e começa a incorporação de novas bases à sequência a partir da replicação pela polimerase presente. A cada base setada, uma luz é emitida (de acordo com a base) e essa incorporação de nucleotídeos é capturada em tempo real. Nesse momento podem aparecer alguns erros, devido à uma leitura incorreta dos flashes de luzes captados pelo sensor. Cada *SMRT cell* possui aproximadamente 150k ZMW, sendo que em média somente 35k-70k produzem leituras de sucesso (RHOADS; AU, 2015). Por se tratar de uma leitura única e dada a ausência de clones para o cálculo de um consenso, a taxa de erro se eleva para aproximadamente 11% (podendo chegar a 20% no *PacBio* e até 35% no *Oxford Nanopore* (CHOUDHURY; CHAKRABARTY; EMRICH, 2018)), que são erros de deleção e/ou inserção na sua maioria (KORLACH, 2013).

Para amenizar a limitação das leituras com alta taxa de erro, alguns métodos são utilizados visando corrigir as sequências lidas de modo a torná-las mais próximas da sequência que deveria ter sido lida de início. Versões mais atuais do *PacBio* possuem uma continuação da etapa final do sequenciamento que utiliza o tempo de vida máximo da polimerase presente nos ZMW, fazendo com que a cadeia continue a ser replicada e ao final, um consenso é calculado a partir de todas as cadeias que foram geradas. Mas isso acaba se tornando caro, podendo inviabilizar todo o estudo sendo feito. Por esse motivo, métodos externos para correção de sequências foram desenvolvidos para minimizar esse erro mantendo um custo viável.

1.0.1 Objetivo e contribuição

Com o objetivo de investigar os desafios da correção de sequenciamento genômico e auxiliar na informatização adequada deste processo em um laboratório de genética de outra Universidade Federal, neste trabalho teve-se por foco o desenvolvimento de nova versão de corretor de sequências baseado em linguagem Awk, a fim de sanar suas limitações. Assim,

a contribuição principal deste trabalho é a implementação de um corretor em linguagem Python 3, fazendo uso de técnicas de paralelismo e compartilhamento em memória para dar vazão à correção de sequências. Além disso, apresenta-se testes de comparação entre os corretores de *reads* nas duas versões, sendo Awk a linguagem da ferramenta original e Python a versão atualizada e estruturada. Mostra-se que as escolhas feitas para a versão em Python permitiram que fosse possível corrigir vários *reads* em paralelo utilizando um genoma de referência compartilhado entre os processos, o que era impossível na primeira versão, que necessitava uma cópia do genoma de referência em cada processo. Por fim, mostra-se que, mesmo utilizando uma linguagem interpretada que é considerada lenta, conseguiu-se um resultado considerável, mas devido à algumas limitações, Python acaba sendo uma má escolha para genomas maiores. Para este desafio, levanta-se trabalhos futuros voltados para o desenvolvimento de uma terceira versão, em C++, que permitiria a correção do sequenciamento do genoma humano. Nesse sentido, mostra-se partes da implementação preliminar dessa terceira versão e discute-se brevemente o caminho a ser seguido.

1.0.2 Organização

O presente trabalho está organizado da seguinte forma: no Capítulo 2, apresentamos um *background* de alguns trabalhos relacionados à correção de reads; no Capítulo 3 falamos sobre o material utilizado durante os testes, como os dados processados e os de referência durante a correção e os recursos computacionais utilizados no desenvolvimento do projeto. Além disso descrevemos os métodos utilizados durante todo o processo e damos uma introdução em como o dicionário de *k-mers* e os *k-mers* em si são utilizados na correção; no Capítulo 4 apresentamos alguns resultados referentes à comparação entre as duas principais linguagens utilizadas no projeto, que foi Awk e Python. Em seguida, discutimos sobre os problemas encontrados e as limitações do Python para resolução do problema em questão e apresentamos possíveis melhorias utilizando C++ e o início do desenvolvimento da nova versão, mais especificamente a construção do dicionário de *k-mers*, utilizando essa linguagem; por fim, no Capítulo 5 apresentamos o que foi possível concluir com o trabalho em questão;

2 Revisão de Literatura

(MORISSE et al., 2020) propuseram uma ferramenta que permite a correção de *reads* grandes a partir de informações que estão na própria sequência, utilizando-se do alinhamento entre as sequências estudadas. Uma sobreposição entre os *reads* é calculada e separada em regiões menores para realizar o alinhamento. A metodologia utilizada se parece com uma das etapas utilizadas no corretor explicado neste trabalho, que é alinhar as sequências de *reads*. Enquanto (MORISSE et al., 2020) alinham os *reads* entre eles mesmos, nosso corretor realiza essa etapa entre os *reads* e uma sequência de referência (*k-mers* dessa sequência). Por não utilizar um genoma de referência carregado em memória, levando em consideração o mesmo organismo utilizado por nós nos testes, o consumo chegou a ser menos da metade do que alcançamos na etapa de criação do dicionário de *k-mers* e teve um tempo de execução 34% menor.

Apesar de considerado o estado da arte, o método referenciado anteriormente chega a ser muito caro computacionalmente, como visto em (CARVALHO; DUPIM; GOLDSTEIN, 2016). Nessa pesquisa foi possível mostrar que a utilização de *k-mers* de genomas sequenciados por técnicas com baixa taxa de erro (como o *Illumina*) possuem grande similaridade com o genoma real do organismo sequenciado, permitindo que fosse utilizado como referência para a correção de erros em *reads* resultantes das tecnologias da 3ª geração, como do *PacBio* por exemplo. Além disso, foi demonstrado que a performance da correção se dará a partir da relação de *k-mers* corretos pelo total de *k-mers*. Esse fato foi utilizado no nosso corretor, que utiliza os *k-mers* do genoma de referência, assim como suas frequências, para encontrar erros nos *reads* sendo processados.

Já (MITCHELL et al., 2020), fizeram uma comparação entre vários corretores, mostrando que os diversos parâmetros existentes nos algoritmos podem influenciar bastante no resultado da correção. Além disso, concluiu-se também que não houve um corretor perfeito para todos os datasets utilizados, indicando que cada ferramenta foi gerada no intuito de processar reads de certo conjunto de genomas, focando em domínios biológicos diferentes. Com isso podemos inferir que as heurísticas utilizadas pelo nosso corretor podem ter sido desenvolvidas com "vícios" que acabam ocasionando uma correção diferente da esperada como verdadeira. Vimos também que, a partir dos corretores apresentados, pode-se verificar que todos utilizam a linguagem C ou C++. Um dos fatores para a escolha foi discutido no Capítulo 4.

(KALLENBORN; HILDEBRANDT; SCHMIDT, 2020) mostra que essa estratégia utilizando frequência dos *k-mers* é importante pois *k-mers* que possuem erros não aparecem frequentemente e por isso é uma alternativa comum entre os corretores. Mas nem sempre é possível utilizar essa frequênci ideia. Caso o número de *reads* que cobrem uma região seja pequeno, haverá uma baixa cobertura dessa região, fazendo com que *k-mers* sem erros

possuam uma baixa frequência, atrapalhando a identidade do método. Outro fator que pode acarretar esse problema, é o fato de que as ferramentas de correção (assim como a estudada neste trabalho) podem utilizar diferentes valores de k, o que influencia na frequência dos novos *k-mers* calculados. (KALLENBORN; HILDEBRANDT; SCHMIDT, 2020) desenvolveu um método baseado em alinhamentos múltiplos dos *reads* de uma mesma região, realizando a correção na sequência consenso criada. Como resultado, conseguiram reduzir os falsos positivos que a abordagem dos *k-mers* dava em certos casos, mas acabou sendo mais caro computacionalmente, devido ter que realizar diversos alinhamentos entre todos os *reads* de uma dada região.

Continuando sobre as comparações entre os corretores, (DOHM et al., 2020) foi um pouco além e explicou que os tipos de erros existentes entre as tecnologias de sequenciamento são diferentes. Enquanto *PacPio* possui mais erros relacionados a substituição de transição (A é trocada por outra base diferente de seu complemento, como G por exemplo) e o *Nanopore* tem como maioria erros de substituições transversais (A é trocado pela base complementar, por exemplo). Além disso, realizaram experimentos entre os *k-mers* do genoma de referência e dos *reads* que seriam corrigidos, mostrando que havia uma relação entre as frequências de *k-mers* do genoma de referência e os *reads* a serem corrigidos, fato que é a base do nosso corretor, que utiliza um dicionário de frequências de *k-mers*.

3 Material e Métodos

3.1 Material

Neste capítulo, serão apresentados os conjuntos de dados e equipamento utilizados nos testes de correção de sequência, bem como os pacotes/módulos/bibliotecas e suas respectivas versões utilizados no desenvolvimento do corretor em Python 3.

3.1.1 Conjunto de Dados

Como referência de genoma correto (sem erros), utilizamos o sequenciamento do verme *Caenorhabditis elegans (resumidamente C. elegans)*. Esse organismo foi escolhido pois, além de ser um genoma pequeno (aproximadamente 100Mb), foi o primeiro organismo multicelular a ter o seu genoma completamente sequenciado (LISTED, 1998) e por isso podemos considerá-lo como um sequenciamento sem erros, o que facilita durante os testes e verificação da correção.

Para correção, utilizamos *reads* resultantes do sequenciamento do *C. elegans* feito pelo *PacBio*. Os *reads* estão divididos em 20 arquivos diferentes no fomato SAM (*Sequence Alignment Map*).

3.1.2 Recursos Computacionais

Foram utilizadas as linguagens Awk e Python para o processamento dos *reads* e a realização da correção. Especificamente, o processamento dos *reads* atualizado usou Python 3.9 e os seguintes pacotes:

- pbcore v1.2.1 para leitura de arquivos no formato *FASTA* e *FASTAQ*
- pysam v0.19.0 para leitura de arquivos no formato *SAM* e *BAM*
- edlib v1.3.9 para realizar o alinhamento entre sequências utilizando distâncias entre as edições

Os testes foram realizados em um servidor com Processador *Intel(R) Xeon(R) Platinum 8164 CPU @ 2.00GHz*, 208 *cores* (104 físicos) e 2Tb de RAM.

3.2 Métodos

Inicialmente, o código da correção estava em Awk. Os *reads* que seriam corrigidos estavam separados em 20 arquivos diferentes no formato SAM e eram usados como

input separadamente, já que não era possível paralelizar com o awk. Cada arquivo SAM possui uma certa quantidade de *reads*, que eram processados e corrigidos. Os problemas encontrados nesse método foi o tempo de execução e a memória utilizada quando era realizado a correção de *reads* de arquivos SAM diferentes.

Para contornar esse problema, o *stakeholder* (Prof. Dr. Bernardo Carvalho, Geneticista e *Principal Investigator* do projeto na UFRJ) iniciou a nova versão, dado que foi responsável pela confecção do corretor em Awk, mas demandou ajuda para implementação usando a linguagem Python, para que ele pudesse também participar da conversão e depois dar manutenção no código produzido. Toda a estrutura foi pensada de modo que cada arquivo SAM pudesse ser processado de forma separada, mantendo suas informações e variáveis encapsuladas por uma classe.

Quando era necessário rodar o código em awk em mais de um arquivo SAM (no caso seriam vários processos diferentes simulando um processamento paralelo), o arquivo com os *k-mers* utilizados como referência na correção precisava ser carregado em cada processo separadamente, o que acarretava em uma grande utilização de memória desnecessária, já que todos os arquivos SAM utilizariam sempre o mesmo arquivo de *k-mer* carregado para comparação.

Os *reads* que queremos corrigir, são os gerados pelos sequenciamentos da nova geração (3ª), que utilizam de métodos mais baratos do que as primeiras versões, como o *Illumina*. Um dos fatores, é que os *reads* são bem maiores (chegando a 100 mil pares de bases). Mas devido a isso, podem acontecer bem mais erros do que os métodos de sequenciamento das gerações passadas, devido a problemas na hora da leitura pelo sensor dos equipamentos. A maior parte dos erros do *PacBio*, por exemplo, são *indels* (inserção ou deleção de bases). Existe um outro modelo utilizado pelo *PacBio* que utiliza um método durante o sequenciamento que aproveita a vida máxima da polimerase para que sejam gerados *subreads* que são sobrepostos para gerar uma sequência consenso. O lado ruim dessa etapa extra, é que acaba deixando o processo bem mais caro. Por esse motivo, existem algumas técnicas de correção dos *reads* gerados pela versão "mais barata".

Para realizar a correção, utiliza-se *k-mers* de genomas de referência para a validação dos *reads*. Por possuir uma taxa baixa de erro (0.1%), sequenciamentos realizados pelo *Illumina* acaba sendo utilizado como referência para a correção. A partir dos *k-mers* considerados válidos (os do *Illumina* no caso), é gerado um dicionário desses e suas frequências.

k-mers são praticamente substrings de tamanho k em uma string, sendo a string uma sequência de nucleotídeos. Exemplo de um 16-mer na figura 1.

Disso temos que qualquer sequência de tamanho L terá L - K + 1 *k-mers*. Portanto, decompor uma sequência em *k-mers* permite que a análise seja feita a partir dessas subsequências menores, o que permite uma melhor correspondência. Por exemplo, se queremos saber a qual organismo uma sequência faz parte, basta decompô-la em *k-mers*

Figura 1 – Exemplo de *16-mer* em uma sequência

e comparar a frequência desses em relação aos organismos sendo estudados. Quanto maior a frequência, mais próximos são esses organismos.

O algoritmo que foi portado para Python realiza o processamento de cada *read* do sequenciamento sendo estudado *k-mer* por *k-mer*. Antes de processar um *read* com possível erro, realizamos a leitura de um arquivo que possua todos os *k-mers* do genoma de referência. Temos como exemplo o arquivo utilizado nos testes, em que o genoma de referência foi dividido em *48-mers*. Dois dicionários possuem como chave as primeiras 24 bases (formando um *24-mer*) e as bases restantes formam o próximo *24-mer*. Como valor, um dicionário possui a frequência de ocorrência da chave e o outro possui uma lista com cada *24-mer* complementar a chave. A Listagem 3.1 ilustra o conteúdo de um arquivo de k-mer (com k=8), enquanto que a Listagem 3.2 representa um trecho de um dicionário de frequência de k-mers (com k=4 e duas entradas).

Listing 3.1 – Arquivo com quatro exemplos de k-mer, k=8.

```
ACTGTTTT
ACTGTCGT
ACCCTTCG
ACTGGGGG
```

Listing 3.2 – Dicionário de frequência de k-mers, k=4.

```
ACTG: 3
ACCC: 1
```

A Listagem 3.3 ilustra um dicionário de correspondências no qual a chave representa um 4-mer e o valor, um dicionário com os potenciais 4-mer subsequentes esperados.

Listing 3.3 – Dicionário de correspondências de k-mers e subsequências possíveis esperadas.

```
ACTG: \{1: TTTT, 2: TCGT, 3: GGGG\}
ACCC: \{1: TTCG\}
```

A seguir, descreve-se o conjunto de passos que corresponde ao processo de verificação de corretude de uma dada sequência. Supondo uma *read* completa dada como abaixo, temos em verde o que seria o 24-mer:

AAAAACCGAAAAAAGTGTGGACTT CCCGCGTGAAAAC

A partir dele, desliza-se por uma janela de quatro bases (um parâmetro do algoritmo também) e verifica se o novo 24-mer resulta em um *k-mer* válido:

AAAA ACCGAAAAAAGTGTGGACTT CCCG CGTGAAAAC

Se sim, salva-se a sequência como uma sequência *kvcorr* e retorna-se ao início.

Se não, os 4 pares de bases que foram adicionados ao final (já que houve o deslizamento de 4 bases no início, as bases em azul formam o final desse novo *24-mer*) devem conter algum erro.

Como saber qual destas bases é a errada? A Figura 2 mostra uma tabela que foi criada a partir de informações contidas no arquivo SAM referente ao *read* sendo processado e cada valor indica a probabilidade daquele erro ter acontecido. Quanto menor o valor, maior essa probabilidade. Do exemplo, temos que provavelmente a base **C** era na realidade a base **G** e o erro ocorreu devido a uma substituição de bases.

error data	C	C	C	G
insertion qv	16	20	15	17
deletion qv	14	27	22	18
deletion base	A	N	T	A
substit. qv	19	21	6	27
substit. base	A	T	G	T

Figura 2 – Tabela de probabilidade de erros em leituras (inserção, remoção e substituição)

Se a partir da tabela de erros e da base possivelmente errada o novo *k-mer* encontrado for válido, salvamos a nova sequência **CCGG** como um *kvcorr*. Se não, indica que não foi possível, a partir da tabela, determinar qual base está errada.

Se não tivermos sucesso na busca anterior, vamos para o método **Blind**. Retornamos ao primeiro passo, antes do slide das 4 bases e fazemos o slide de base em base agora.

AAAAACCGAAAAAAGTGTGGACTT CCCGCGTGAAAAC

A AAAACCGAAAAAAGTGTGGACTTC CCGCGTGAAAAC

AA AAACCGAAAAAAGTGTGGACTTCC CGCGTGAAAAC

AAA AACCGAAAAAAGTGTGGACTTCCC GCGTGAAAAC

AAAA ACCGAAAAAAGTGTGGACTTCCCG CGTGAAAAC

Pelo exemplo, temos que o último *slide* fez com que o *k-mer* em questão não fosse válido. Tentamos retornar ao estado original dessa sequência antes de acontecer o erro, testando todos os possíveis problemas que podem acontecer:

- **Inserção**: Remover o G
- **Deleção**: Substituir G por AG, CG, GG, TG
- **Incompatibilidade**: Substituir G por A, C T

Se após essa verificação encontrarmos somente um *k-mer* válido, salvamos a nova sequência (por exemplo **CCCT**) como um *kvcorr*. Se mais de uma edição for um *k-mer* válido, realizamos um outro processamento. Se não obtivermos um *k-mer* válido, iniciamos outro procedimento para verificar os erros no *read*.

Os métodos anteriores podem não ser o suficiente para encontrar o erro do *read* e por esse motivo realizamos um outro procedimento chamado **Bimer**. A partir dele é possível alinhar a sequência sendo analisada com a sequência de referência. Antes do alinhamento, devemos pegar o último *k-mer* válido (chamaremos de chave) e pegar as bases associadas (chamaremos de valor) à ele no dicionário compartilhado de *k-mer* (no nosso caso, os *k-mers* de referência são as chaves e as bases associadas a eles serão os valores. Caso existam vários valores associados ao mesmo *k-mer*, esses serão salvos em uma lista, para serem verificados de modo mais fácil). Após esse passo, basta realizar o alinhamento do valor com a sequência sendo analisada utilizando o *edlib*.

4 Resultados e Discussão

Dado o exposto nos capítulos anteriores, foi implementada uma versão em Python a partir do corretor em Awk, visando testar sua eficiência e potencial de ser utilizado para correção de sequenciamento de genomas maiores, como o humano.

Neste capítulo, em linhas gerais, serão discutidos detalhes da implementação feita em Python, bem como serão mostrados testes preliminares e os resultados obtidos da comparação do programa original com o novo programa. O foco dado foi ao gerenciamento da memória, uma vez que os genomas utilizados nos testes são relativamente pequenos ($\approx$100 milhões de pares de bases), enquanto que o genoma humano necessita de um espaço bem maior para carregamento na memória e correção (3 bilhões de pares de bases).

4.0.1 Ideia Geral da Implementação

Pensou-se em criar uma classe geral em Python para carregar o dicionário de *k-mer*, que chamaremos de classe global. Tendo como exemplo o genoma do *C. elegans*, ao carregar o arquivo de *k-mers* em um dicionário, aproximadamente 40Gb de RAM eram utilizados no processo, mostrando que seria inviável carregar várias vezes esse arquivo em memória para genomas maiores. Para processar os arquivos SAM, uma classe chamada *EachProcess*, com os métodos para essa etapa foi desenvolvida, que utiliza a classe global para ter acesso ao dicionário de *k-mer*.

Com isso, utilizando o módulo *multiprocessing* do Python, conseguimos fazer com que cada arquivo SAM fosse aberto simultaneamente, sendo cada um uma instância diferente da classe EachProcess. As variáveis de cada classe seriam globais para cada arquivo SAM e as variáveis da classe global seria global entre todos os arquivos. Desse modo, conseguimos paralelizar a correção e manter um dicionário de *k-mer* compartilhado entre todos os processos, diminuindo em X vezes (X sendo o número de arquivos SAM abertos) a quantidade de memória utilizada na versão em Awk.

Ao ser aberta, cada sequência de *read* presente no arquivo SAM em questão, é processada separadamente. As variáveis locais (globais para cada processo) são inicializadas com os parâmetros importantes de cada read (método prepare_data_array). Dependendo da tecnologia utilizada para a geração dos *reads*, os parâmetros podem ser diferentes.

4.0.2 Testes Comparativos

A seguir, temos a comparação ao carregar os *k-mers* de referência na memória entre a versão em Awk e Python. Ao todo, foram carregados 201.416.236 24-mers, sendo 96.5% únicos (194.394.842).

Na Figura 3, mostra-se teste feito com somente um arquivo de *reads*. Percebe-se que a estrutura utilizada para salvar os *k-mers* em Python é mais otimizada do que em Awk, consumindo menos memória para carregar todos os *k-mers*.

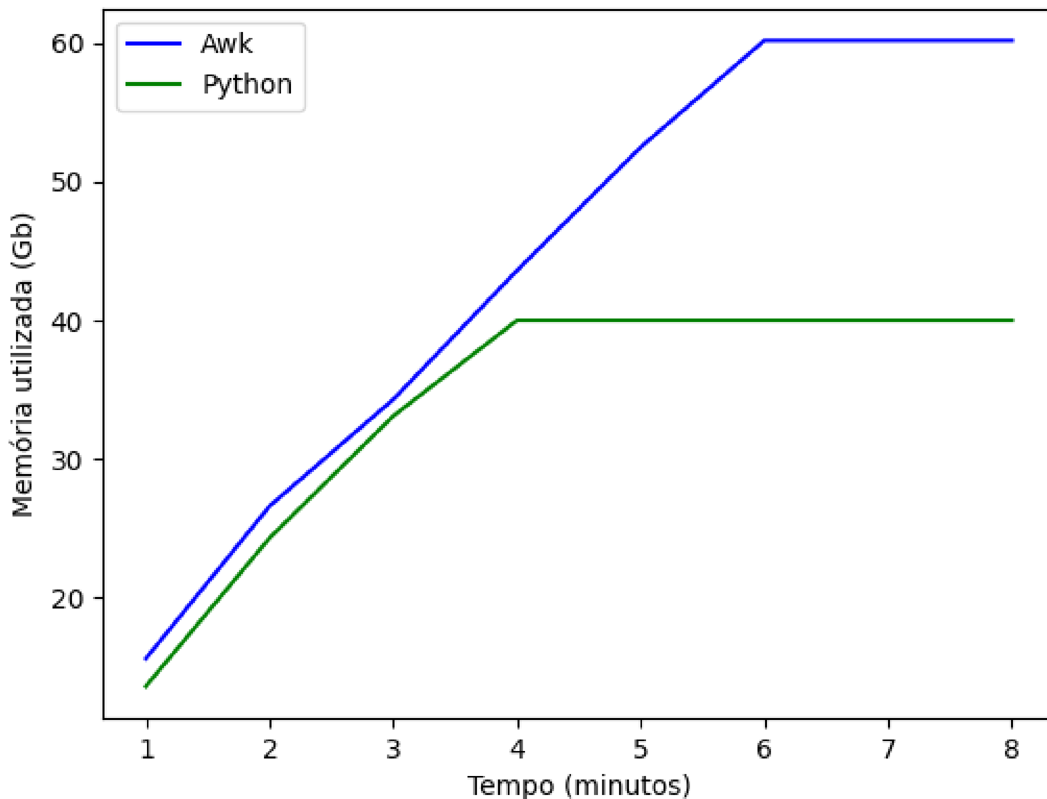

Figura 3 – Relação entre tempo e memória para carregar o arquivo de *k-mers* entre a versão em Awk e Python - sequencial

Quando passamos para o processamento de dois arquivos de *reads* simultaneamente, percebemos a grande diferença de utilizar um dicionário de *k-mers* compartilhado. Como nao é possível realizar tal procedimento em Awk, o único modo de realizar o processamento de dois arquivos ao mesmo tempo é realizando duas execuções da ferramenta, que intuitivamente já sabemos o resultado: irá duplicar a memória utilizada. Na Figura 4 vemos isso acontecendo, enquanto o consumo de memória do Python permanece idêntico ao da Figura 3.

Na Figura 5, mostra-se um gráfico com a relação de tempo gasto para correção das *reads* de um único arquivo SAM. Enquanto o Awk está processando um arquivo por vez, a versão em Python já está processando todos os arquivos SAM paralelamente devido à aplicação de *multiprocessing*. Percebe-se que devido a ter-se ganho bastante desempenho em relação à versão do corretor em Awk, é possível realizar a correção de vários arquivos sequencialmente com Python e mesmo assim ainda obter resultados mais rápidos.

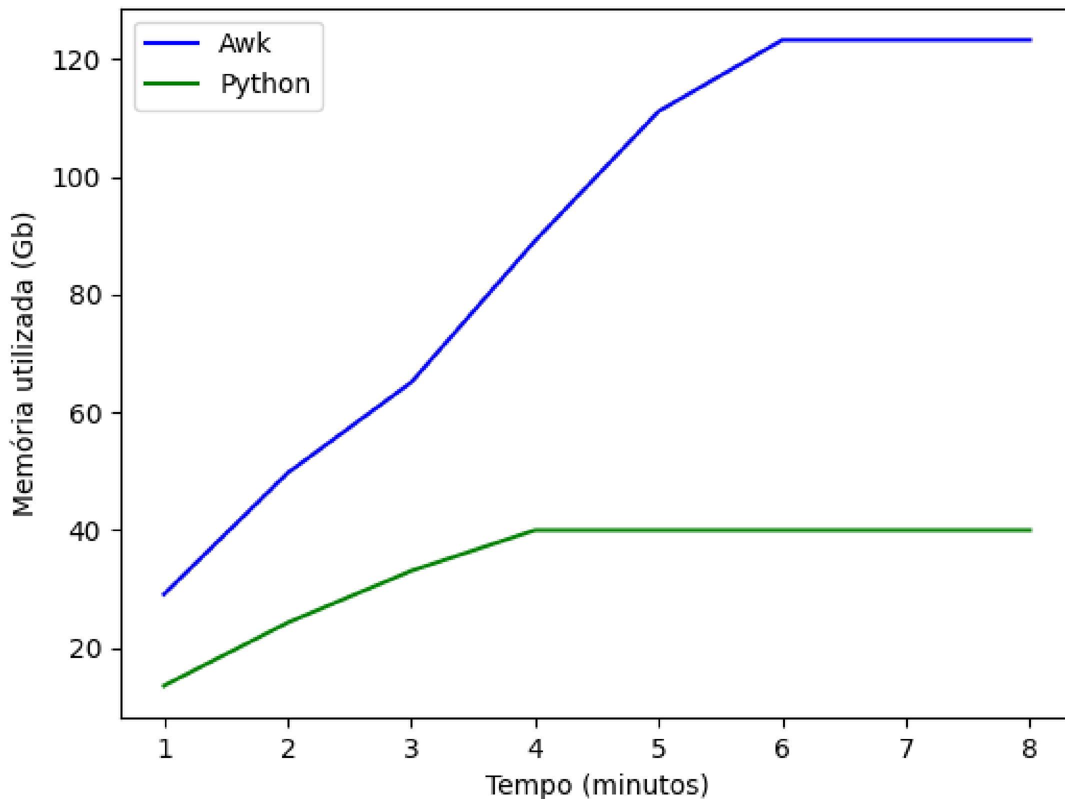

Figura 4 – Relação entre tempo e memória para carregar o arquivo de *k-mers* entre a versão em Awk e Python - paralelo (somente Python está em paralelo. Awk possui duas execuções simultâneas)

Nas Figuras 3 e 4, quando a criação do dicionário é finalizada, o consumo de memória permanece constante. Esse seria o modelo ideal para as duas versões, mas isso não acontece na versão em Python. Devido à limitações da linguagem, constatamos que em um determinado momento, haverá vários *k-mers* utilizados para comparação. Essas strings acabam se acumulando, não dando tempo para o *garbage collector* (gc) limpar a memória (leva um tempo até o gc identificar qual memória liberar), resultando no gráfico apresentado na Figura 6.

É possível observar na Figura 6 que em toda a execução do Awk, a memória permanece constante após carregarmos o arquivo com os *k-mers*. Já na versão em Python, temos um crescimento durante toda a execução do teste, chegando a ultrapassar a memória utilizada pelo Awk. O aumento e queda repentina de memória próxima aos 5 minutos é devido uma cópia realizada do dicionário de *k-mers* e a deleção da classe global.

No gráfico da Figura 7, tem-se uma relação entre a memória privada e compartilhada da versão Python. Até aproximadamente 4 minutos, há a criação do dicionário compartilhado de *k-mers*. No momento não existe, entretanto, memória compartilhada, pois os processos-filhos ainda não foram criados. Próximo aos 5 minutos temos o fato

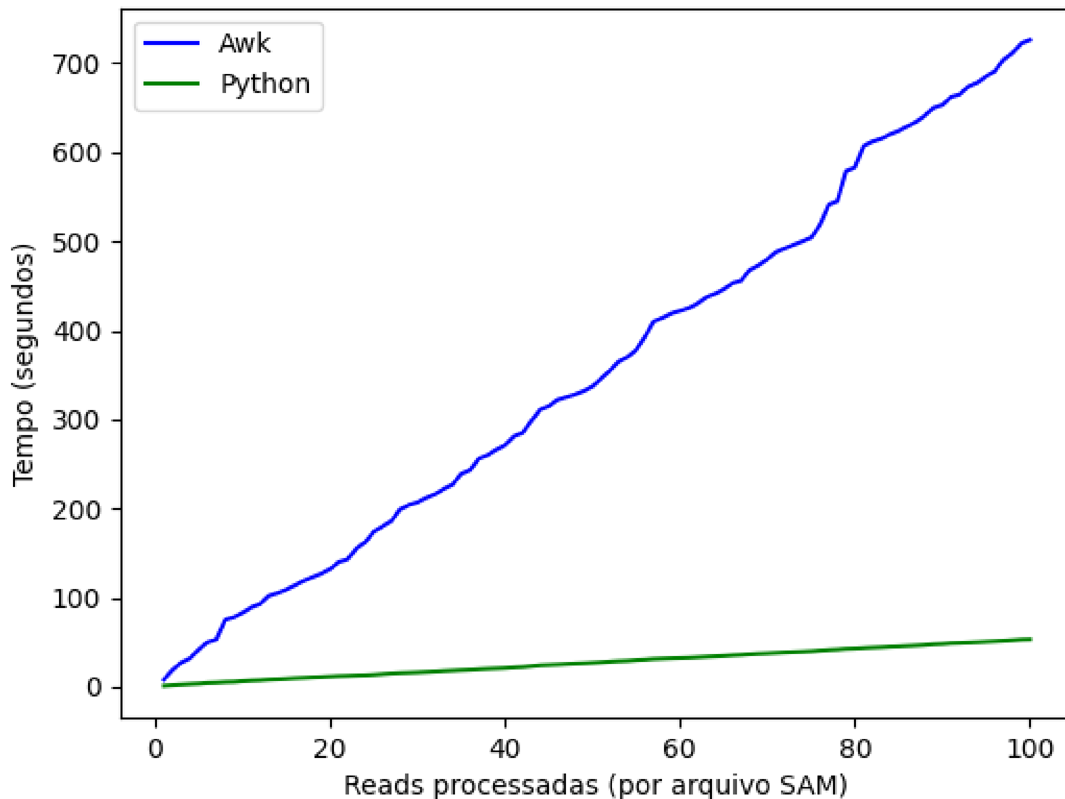

Figura 5 – Relação entre tempo de execução entre Awk e Python para a correção de um arquivo de reads

explicado na Figura 6 e a partir desse momento, já com os novos processos criados para cada arquivo de *reads*, a memória compartilhada permanece constante durante toda a execução enquanto a memória privada (nesse caso é a soma da memória privada de todos os processos) começa a crescer.

Uma alternativa para o problema foi mudar a abordagem: em vez de utilizar multiprocessos, passou para *multithreading*. Fazendo essa modificação, não mais se observou o vazamento de memória. Porém, o programa ficou extremamente lento em comparação à abordagem multiprocessos, conforme pode-se observar na Figura 8. Cada bloco representa o mesmo arquivo SAM (o primeiro relativo à versão com multithreading e o segundo utilizando multiprocessos) e a última coluna é o tempo de processamento do *read* correspondente. A diferença de tempo é extremamente grande, fato que pode ser explicado a partir do *Global Interpreter Lock* (GIL), que funciona como um *mutex* para o gerenciamento da memória. Essa etapa não é *thread-safe* e o GIL consegue garantir que um programa que utilize várias *threads* não tenha a memória corrompida quando essas tentam alterar a memória ao mesmo tempo (HATTEM, 2016). Com isso, temos que a utilização de várias *threads* é prejudicada pelo GIL, pois somente uma *thread* por vez pode acessá-lo.

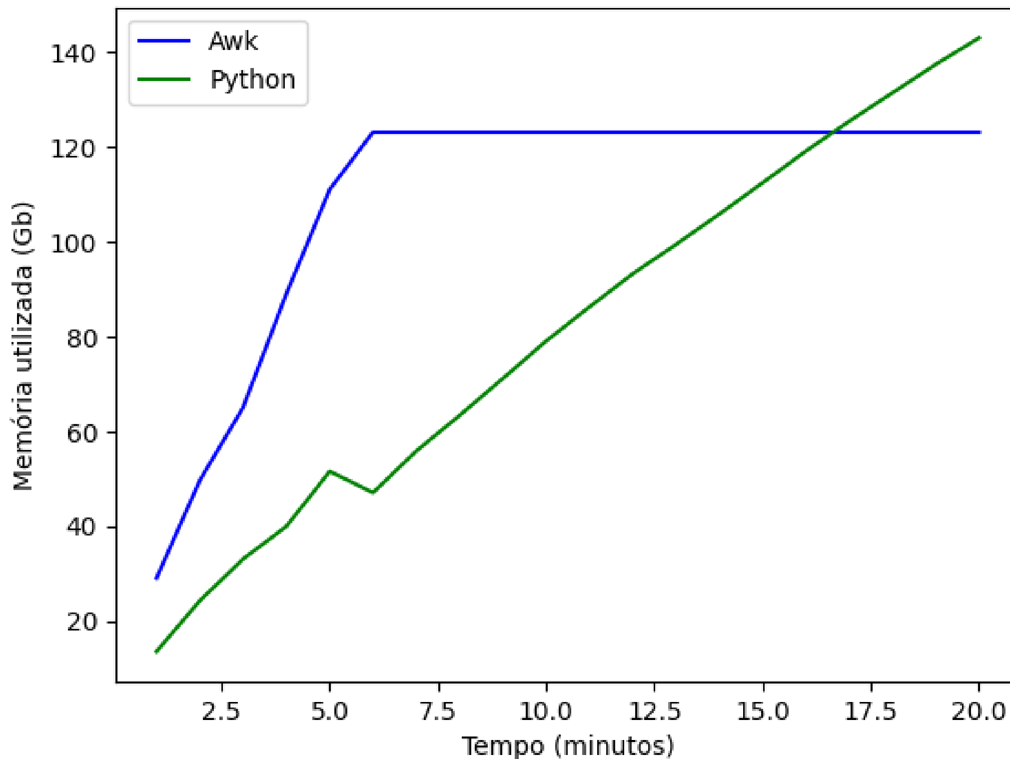

Figura 6 – Relação entre tempo e memória para correção de *reads* de dois arquivos distintos entre a versão em Awk e Python - paralelo (somente Python está em paralelo. Awk possui duas execuções simultâneas)

4.0.3 Desafios

Para resolver a situação que se apresentou, seria mais interessante utilizar uma linguagem que permite ao programador ter um controle maior da memória que será liberada. Desse modo, visando possibilitar o tratamento de genomas maiores sem ocasionar sobrecarga de memória, concluiu-se que a melhor abordagem seria reimplementar o código feito em Python, desta vez utilizando C++. Por ser uma linguagem compilada, C++ conseguir transferir o código-fonte diretamente em linguagem de máquina, diferente de Python que primeiramente passa por um interpretador que irá traduzir o código em *byte code* e durante a execução transfere o código para a linguagem de máquina.

Porém, a mera tradução direta das funções não traria resultados muito vantajosos, pois algumas estruturas podem causar um *overhead* desnecessário. Em Python, O dicionário de frequências de *k-mers* foi feito com as chaves sendo strings e os valores podendo ser strings (caso a frequência seja única, o valor é o *k-mer* correspondente e não o número “1”) ou números inteiros.

Devido a esse cenário de *overhead* e outros detalhes da implementação em Python codificados anteriormente a este projeto e/ou legados da versão em Awk, deu-se início a fase de implementação em C++. O objetivo era o de corroborar a necessidade de

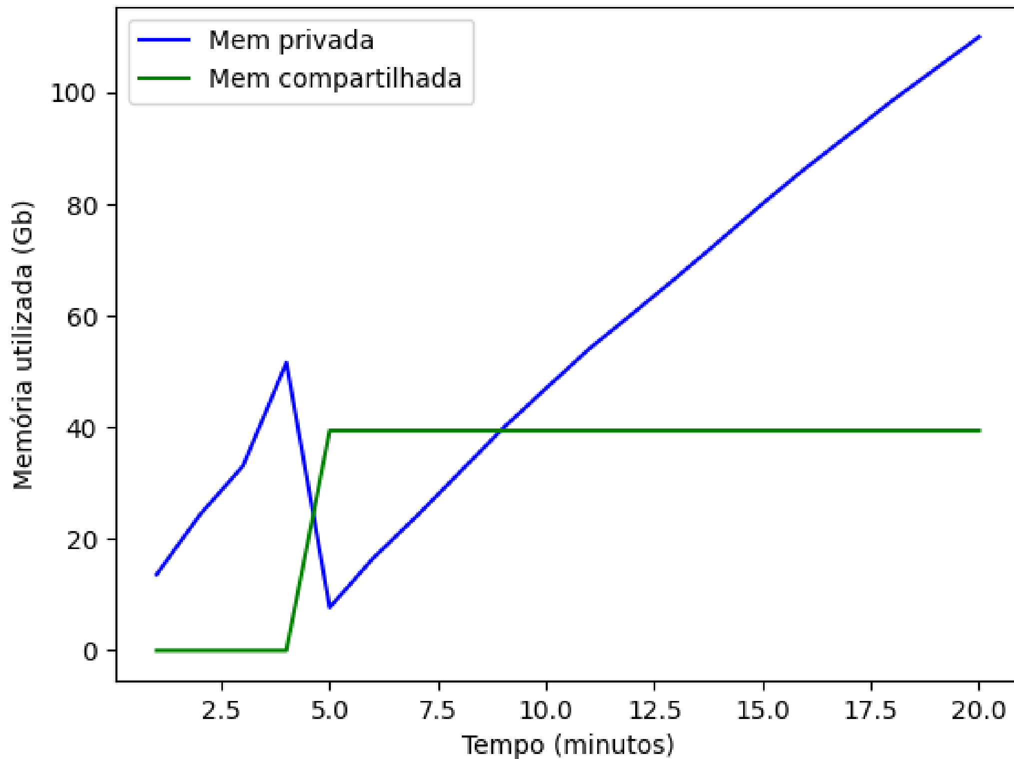

Figura 7 – Relação entre tempo e memória privada e compartilhada da versão Python

```
head claudio_vp27fev_temp2_qv4bpFR_PR0/output/vp27fev_temp2_qv4bpFR_PR0_x19.cel_sel1.dcs_summary_log
readID                                                                        def$ size_raw    time
     1     m140930_013011_sidney_c100699772550000001823139903261597_s1_p0/37632/0_20167    20167   636.1
     2     m140930_013011_sidney_c100699772550000001823139903261597_s1_p0/37644/0_18766    18766   536.0
     3  m140930_013011_sidney_c100699772550000001823139903261597_s1_p0/37652/2108_29560    27452   942.5
```

```
head vp27fev_temp1_qv4bpFR_PR0/output/vp27fev_temp1_qv4bpFR_PR0_x19.cel_sel1.dcs_summary_log
readID                                                                        def$ size_raw    time
     1     m140930_013011_sidney_c100699772550000001823139903261597_s1_p0/37632/0_20167    20167     7.8
     2     m140930_013011_sidney_c100699772550000001823139903261597_s1_p0/37644/0_18766    18766    11.3
     3  m140930_013011_sidney_c100699772550000001823139903261597_s1_p0/37652/2108_29560    27452    14.2
```

Figura 8 – Tempo (em segundos) de execução do multithreading vs multiprocessing para 3 reads

uma reestruturação completa, incluindo a tomada de novas decisões de projeto que contemplassem mudanças profundas e efetivas na forma de se representar os dados e realizar a entrada e saída internamente entre as funções.

Por exemplo, portou-se inicialmente dicionário de frequências de *k-mers* visando realização dos primeiros testes. Ao passá-lo para C++, isso levou a um acréscimo de tempo quando utilizamos um *map* que podia aceitar tanto *strings* quanto números inteiros como valor. Assim, imediatamente realizou-se uma mudança ao se deixar tudo como *string*, partindo então para a criação de uma função que verifica se a *string* é um número inteiro ou não. Essa verificação acabou deixando a execução um pouco mais rápida. Outro fator que pode ter influenciado na velocidade de execução dessa etapa é o fato de que as variáveis em C++ já possuem o seu tipo conhecido antes da execução, diferente de Python que precisa checar e realizar conversões do tipo da variável durante a execução sempre que a variável em questão for modificada.

A Figura 9 mostra que há uma redução de tempo de carregamento entre os dicionários em C++ e Python, indicando a diferença de declaração de variáveis entre as duas linguagens. Em Python tudo acaba sendo um objeto, o que acarreta em um *overhead* maior do que em C++.

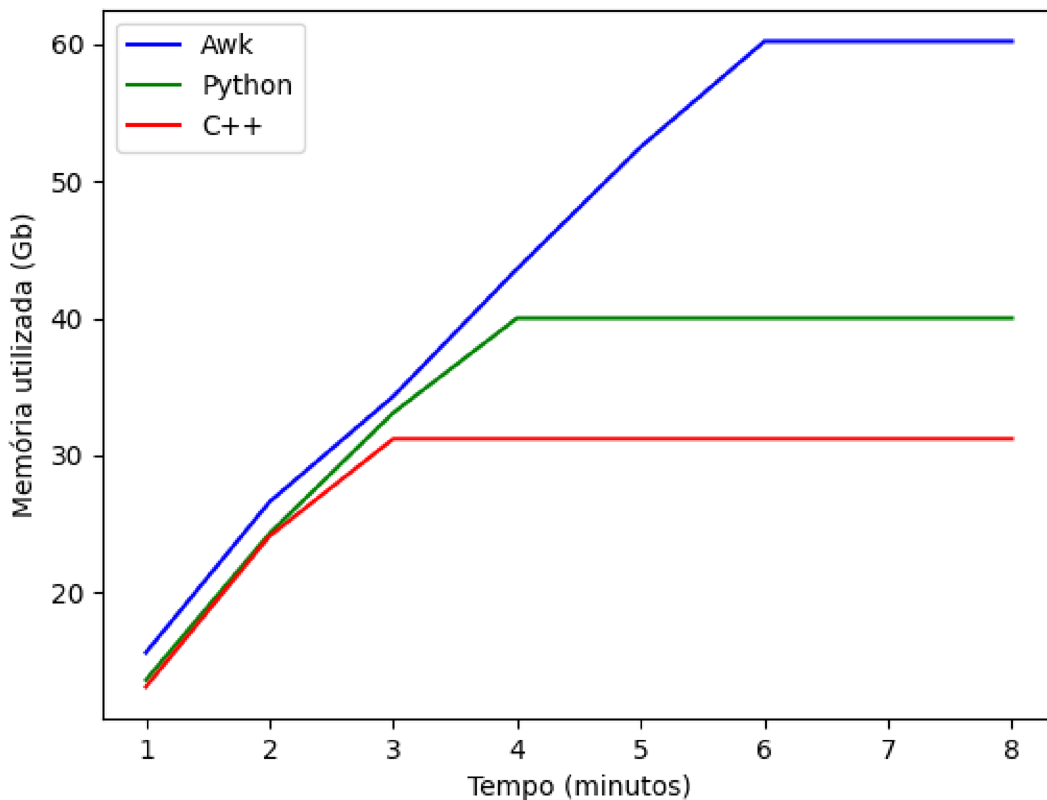

Figura 9 – Relação entre tempo e memória para carregar o arquivo de *k-mers* entre a versão em Awk, Python e C++ - sequencial

Nota-se que, embora o *stakeholder* tenha demandado uma linguagem específica (Python) em decorrência de sua necessidade de manutenção posterior independente, melhores resultados poderão ser obtidos simplesmente migrando-se o programa para C++. Para ser possível analisar competitivamente um genoma como o humano, a migração da linguagem por si só (isto é, sem alteração no fluxo das funções, tipos de dados e estruturas) pode não ser suficiente, deixando espaço para inúmeras otimizações.

5 Conclusão

A partir do trabalho realizado, pode-se perceber que a utilização de corretores, apesar de vantajosa, pode ter um limitador importante que é a memória disponível para carregar o genoma que será utilizado como referência. Utilizando um organismo com aproximadamente 100Mb tivemos um consumo em torno de 40Gb de memória para carregar o genoma de referência e construir o dicionário de *k-mers*, um valor muito inferior se tivessemos escolhido o genoma humano, que possui um tamanho de 3Gb.

Além disso, percebemos a diferença existente entre as linguagens de programação utilizadas na ferramenta de correção estudada. Apesar de termos conseguido uma melhora significativa de tempo de execução da correção dos *reads*, passamos a ter um novo problema relacionado a memória utilizada, o que acabou piorando o nosso limitador. Python possui suas vantagens e facilidade de implementação, mas para realizar tarefas com alto número de comparações e cópias entre strings, uma linguagem de mais baixo nível como C/C++ seria uma alternativa melhor pois, devido ao fato de os *k-mers* compartilhados não serem modificados, não seria necessário realizar a cópia dos *k-mers* e sim a criação de ponteiros no lugar.

Outro fato importante, é que para termos uma base sólida de referência para uma noção maior da correção (se está sendo realizada corretamente ou não) se dá a partir de sequenciamentos realizados por tecnologias de gerações anteriores que possuem uma maior acurácia (como o *Illumina*). Isso acaba sendo um limitador, pois a maior parte da correção se dará por heurísticas probabilísticas e isso pode nos dar informações erradas, o que foi demonstrado pelo número de etapas existentes para determinar se uma base está errada ou não.

Referências

CARVALHO, A. B.; DUPIM, E. G.; GOLDSTEIN, G. Improved assembly of noisy long reads by k-mer validation. *Genome Res*, v. 26, n. 12, p. 1710–1720, 12 2016.

CHOUDHURY, O.; CHAKRABARTY, A.; EMRICH, S. J. Hecil: A hybrid error correction algorithm for long reads with iterative learning. *Scientific Reports*, v. 8, n. 1, p. 9936, Jul 2018. ISSN 2045-2322. Disponível em: <https://doi.org/10.1038/s41598-018-28364-3>.

DOHM, J. C. et al. Benchmarking of long-read correction methods. *NAR Genomics and Bioinformatics*, v. 2, n. 2, 05 2020. ISSN 2631-9268. Lqaa037. Disponível em: <https://doi.org/10.1093/nargab/lqaa037>.

GOODWIN, S. et al. Oxford Nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome. *Genome Res*, v. 25, n. 11, p. 1750–1756, Nov 2015.

HATTEM, R. van. *Mastering Python*: Master the art of writing beautiful and powerful python by using all of the features that python 3.5 offers. [S.l.]: Packt Publishing Ltd, 2016.

KALLENBORN, F.; HILDEBRANDT, A.; SCHMIDT, B. CARE: context-aware sequencing read error correction. *Bioinformatics*, v. 37, n. 7, p. 889–895, 08 2020. Disponível em: <https://doi.org/10.1093/bioinformatics/btaa738>.

KORLACH, J. Understanding accuracy in smrt ® sequencing. In: . [S.l.: s.n.], 2013.

LABORATORY, C.-.; DIAGNOSIS. *Genomic sequencing of SARS-CoV-2: a guide to implementation for maximum impact on public health*. 2021. Disponível em: <https://www.who.int/publications/i/item/9789240018440>.

LIN, B.; HUI, J.; MAO, H. Nanopore technology and its applications in gene sequencing. *Biosensors*, v. 11, n. 7, 2021. ISSN 2079-6374. Disponível em: <https://www.mdpi.com/2079-6374/11/7/214>.

LISTED, N. authors. Genome sequence of the nematode C. elegans: a platform for investigating biology. *Science*, v. 282, n. 5396, p. 2012–2018, Dec 1998.

MITCHELL, K. et al. Benchmarking of computational error-correction methods for next-generation sequencing data. *Genome Biology*, v. 21, n. 1, p. 71, Mar 2020. ISSN 1474-760X. Disponível em: <https://doi.org/10.1186/s13059-020-01988-3>.

MORISSE, P. et al. Consent: Scalable long read self-correction and assembly polishing with multiple sequence alignment. *bioRxiv*, Cold Spring Harbor Laboratory, 2020. Disponível em: <https://www.biorxiv.org/content/early/2020/04/24/546630>.

PARK, S. T.; KIM, J. Trends in next-generation sequencing and a new era for whole genome sequencing. *Int Neurolol Journal*, Suppl. 2, n. 20, p. S76–83, 2016.

RHOADS, A.; AU, K. F. Pacbio sequencing and its applications. *Genomics, Proteomics & Bioinformatics*, v. 13, n. 5, p. 278–289, 2015. ISSN 1672-0229. SI: Metagenomics of Marine Environments. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1672022915001345>.