UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAPHAEL SANT'ANNA GOMES

GENUVEM

Busca de sequências genéticas em plataformas de Big Data

RIO DE JANEIRO
2024

RAPHAEL SANT'ANNA GOMES

GENUVEM

Busca de sequências genéticas em plataformas de Big Data

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profa. Silvana Rossetto
Co-orientadora: Profa. Maria Luiza M. Campos

RIO DE JANEIRO
2024

CIP - Catalogação na Publicação

G633g

Gomes, Raphael Sant’Aanna
Genuvem: busca de sequências genéticas em plataformas de Big Data / Raphael Sant’Aanna Gomes. - Rio de Janeiro, 2024.
77 f.

Orientadora: Silvana Rossetto.
Coorientador: Maria Luiza Machado Campos.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. Alinhamento de sequências. 2. Bioinformática. 3. Blast. 4. Hadoop. 5. Spark. I. Rossetto, Silvana, orient. II. Campos, Maria Luiza Machado, coorient. III. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

RAPHAEL SANT'ANNA GOMES

GENUVEM
Busca de sequências genéticas em plataformas de Big Data

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 14 de Agosto de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 20/08/2024 20:02:19-0300
Verifique em https://validar.iti.gov.br

Silvana Rossetto, D.Sc. (UFRJ)

gov.br Documento assinado digitalmente
MARIA LUIZA MACHADO CAMPOS
Data: 20/08/2024 23:39:43-0300
Verifique em https://validar.iti.gov.br

Maria Luiza M. Campos, Ph.D. (UFRJ)

gov.br Documento assinado digitalmente
GISELI RABELLO LOPES
Data: 21/08/2024 06:53:55-0300
Verifique em https://validar.iti.gov.br

Giseli Rabello Lopes, D.Sc. (UFRJ)

gov.br Documento assinado digitalmente
VIVIAN DOS SANTOS SILVA
Data: 21/08/2024 10:45:49-0300
Verifique em https://validar.iti.gov.br

Vivian dos Santos Silva, D.Sc. (UFRJ)

gov.br Documento assinado digitalmente
RODRIGO JARDIM MONTEIRO DA FONSECA
Data: 21/08/2024 08:16:21-0300
Verifique em https://validar.iti.gov.br

Rodrigo Jardim, D.Sc. (IOC)

Em memória de minhas vovós, Eny e Neide.

AGRADECIMENTOS

Em primeiro lugar, agradeço aos meus pais, Sônia e Fernando, por tudo que representam e fizeram por mim e pela minha educação. À minha tia Fátima, que me educou e ensinou a caminhar. E à minha prima Mariana, por ser minha maior referência.

Às minhas orientadoras, Maria Luiza Campos e Silvana Rossetto, pela orientação, e pela compreensão e paciência durante o período que levei para concluir este trabalho. Estendo estes agradecimentos aos demais professores do Bacharelado em Ciência da Computação da Universidade Federal do Rio de Janeiro (UFRJ) e do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ), onde formei-me Técnico em Informática. A estas instituições, minha profunda gratidão por serem referência em educação pública, gratuita e de qualidade.

Ao professor Rodrigo Jardim, sempre solícito, por me apresentar o Genoogle, me receber em seu laboratório e apoiar todo o desenvolvimento deste projeto. Através da sua pessoa, agradeço à Fundação do Instituto Oswaldo Cruz (FIOCRUZ).

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e ao programa Ciência sem Fronteiras, agradeço pela bolsa de estudos que me permitiu estudar Genômica Computacional e Computação de Alto Desempenho em *Queen Mary, University of London* (QMUL). E agradeço à tia Deise *(in memoriam)*, cujo apoio foi fundamental para que este intercâmbio acontecesse.

Finalmente, agradeço aos meus amigos, em especial ao Xiao e Débora, pelos conselhos oferecidos ao longo do processo. E ao meu querido time da Tribo de Dados da Exame, agradeço pelos docinhos que me enviaram quando precisei me recolher para concluir esta monografia.

"Há grandeza nessa visão de que a vida, com suas diversas capacidades, teria sido insuflada pelo Criador em poucas formas, ou em uma só; e que, enquanto este planeta se mantém a girar conforme a imutável Lei da Gravidade, a partir de tão simples começo, infindáveis formas, das mais belas e maravilhosas, evoluíram e continuam a evoluir."

Charles Darwin

RESUMO

A busca de sequências em bases de dados permanece como uma das mais importantes tarefas em bioinformática, sendo o *Basic Local Alignment Search Tool* (BLAST) a principal ferramenta utilizada para inferência de homologia baseada em similaridade de sequências. Nas últimas décadas, o volume de dados publicados cresceu de tal forma que foi necessária a criação de estratégias de paralelização e distribuição das buscas. Uma das possibilidades é a distribuição da execução do BLAST em *clusters* de computadores com o apoio de plataformas de *Big Data*, como Apache Hadoop e Apache Spark. Nesta monografia, foi realizada uma revisão de trabalhos relacionados neste campo, e foi desenvolvida uma ferramenta para busca de sequências em nuvem baseada em Spark, o Genuvem. A validação foi conduzida na nuvem pública da *Google Cloud Platform* em diferentes tipos de *clusters* de computadores. Os resultados demonstraram que, para buscas suficientemente grandes, a solução escala com eficiência superior a 80%.

Palavras-chave: alinhamento de sequências; bioinformática; blast; hadoop; spark;

ABSTRACT

Sequence searching in databases remains one of the most important tasks in bioinformatics, with the Basic Local Alignment Search Tool (BLAST) being the primary tool used for homology inference based on sequence similarity. In recent decades, the volume of published data has grown to such an extent that it has been necessary to create parallelization and distribution strategies for searches. One possibility is to distribute the execution of BLAST in computer clusters with the support of Big Data platforms like Apache Hadoop and Apache Spark. In this monograph, a review of related works in this field was conducted, and a tool for cloud-based sequence searching, Genuvem, was developed. Validation was conducted on the Google Cloud Platform public cloud in different types of computer clusters. The results demonstrated that, for sufficiently large searches, the solution scales with an efficiency greater than 80%.

Keywords: bioinformatics; blast; cloud computing; hadoop; mapreduce; sequence alignment; spark;

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BLAST	*Basic Local Alignment Search Tool*
DAG	*Directed Acyclic Graph*
DNA	*Deoxyribonucleic Acid*
GCP	*Google Cloud Platform*
GCS	*Google Cloud Storage*
GFS	*Google File System*
HDFS	*Hadoop Distributed File System*
HSP	*High-scoring Segment Pair*
JAR	*Java Archive*
JVM	*Java Virtual Machine*
mRNA	RNA Mensageiro
NCBI	*National Center for Biotechnology Information*
NGS	*Next Generation Sequencing*
ORF	*Open Reading Frame*
PaaS	*Platform as a Service*
RAM	*Random-Access Memory*
RDD	*Resilient Distributed Dataset*
RNA	*Ribonucleic Acid*
WGS	*Whole Genome Shotgun*
URL	*Universal Resource Locator*
XML	*Extensible Markup Language*
YARN	*Yet Another Resource Negotiator*

LISTA DE SÍMBOLOS

b	bit
B	byte
bp	par de bases
GB	gigabyte
GHz	gigahertz
KB	kilobyte
MB	megabyte
MHz	megahertz
TB	terabyte

SUMÁRIO

1 INTRODUÇÃO

Em 1859, o naturalista britânico Charles Darwin publicou o livro "A Origem das Espécies", onde demonstrou que organismos se adaptam ao meio em que vivem de forma gradual, através de um mecanismo que chamou de Seleção Natural. O acúmulo de mudanças ao longo de gerações resultaria em novas espécies. A Teoria da Evolução de Darwin, contudo, não explicava como estas mudanças surgiam e eram passadas de um indivíduo para seus descendentes. Hoje, compreende-se que fatores genéticos e epigenéticos mediam esta herança. E que comportamentos e códigos sociais influenciam indivíduos, o meio ambiente e, consequentemente, a vida (JABLONKA; LAMB, 2005).

O estudo das sequências genéticas tem aplicações em áreas da saúde, biologia, e até na antropologia. O GenBank (SAYERS et al., 2023) é um das maiores bases de dados genéticos e seu tamanho vem dobrando a cada dois anos (NCBI, 2023a). A análise da similaridade pelo alinhamento de sequências ajuda a entender as relações entre indivíduos e espécies (CRISTIANINI; HAHN, 2007). O BLAST (ALTSCHUL et al., 1990) é um dos programas mais utilizados para busca de sequências em bases de dados, O custo computacional deste tipo de busca é elevado.

1.1 MOTIVAÇÃO

Desde o início dos anos 2000, as fabricantes de microprocessadores parecem ter atingido o teto no que diz respeito a aumento da frequência de *clock* de processadores. Fatores limitantes as fizeram priorizar a combinação de múltiplos núcleos em um processador em detrimento da velocidade nominal (MCCOOL; REINDERS; ROBISON, 2012). Enquanto isso, a quantidade de informação gerada pela humanidade cresce exponencialmente. O problema de processamento de dados em larga escala, chamado de *Big Data*, requer computação em *clusters*. Entretanto, a adaptação de algoritmos para execução paralela e distribuída não é trivial (CHAMBERS; ZAHARIA, 2018).

A popularização da computação em nuvem e das plataformas para processamento de dados em larga escala tornou mais simples a utilização de *clusters* para resolução de problemas de *Big Data* (WHITE, 2015). Na bioinformática, o Cloud BLAST (MATSUNAGA; TSUGAWA; FORTES, 2008) utilizou o Apache Hadoop para distribuir a execução de buscas de sequências com BLAST. O SparkBLAST (CASTRO, 2017) fez o mesmo utilizando Apache Spark. O Genoogle (ALBRECHT, 2009) implementou paralelismo e estruturas de dados eficientes para otimizar o algoritmo do BLAST, mas não suporta execução distribuída.

A motivação deste trabalho é estudar o Genoogle enquanto alternativa ao BLAST e avaliar oportunidades de otimização em sua arquitetura.

1.2 OBJETIVO GERAL

O principal objetivo deste trabalho é propor uma solução horizontamente escalável para buscas de sequências genéticas em bases de dados. O atingimento do resultado esperado requer o cumprimento dos seguintes objetivos específicos:

- Identificar oportunidades de otimização de algoritmos de buscas de sequência;
- Adaptar o algoritmo para execução em ambiente distribuído;
- Avaliar métricas de desempenho e de utilização de infraestrutura; e
- Comparar o desempenho computacional deste trabalho com trabalhos correlatos

1.3 METODOLOGIA

O presente estudo classifica-se como exploratório pois visa, a partir da revisão bibliográfica, identificar oportunidades de otimização. A revisão terá como ponto de partida o BLAST (ALTSCHUL et al., 1990), e o modelo de programação MapReduce (DEAN; GHEMAWAT, 2004).

A abordagem ao problema será quantitativa, uma vez que um dos objetivos específicos do projeto é validar a solução por meio de métricas de desempenho.

1.4 ESTRUTURA

Este trabalho está dividido em seis partes. No Capítulo 2 são revisados os conceitos da Biologia e da Ciência da Computação fundamentais para compreensão desta monografia. No Capítulo 3 são discutidos trabalhos relacionados que exploraram técnicas de computação paralela e distribuída para otimização de busca de sequências em bases de dados. O Capítulo 4 detalha a proposta e o desenvolvimento deste trabalho. No Capítulo 5, serão discutidos os resultados e limitações. Por fim, no Capítulo 6, são apresentadas as conclusões do trabalho, suas contribuições e sugestões de trabalhos futuros.

2 CONCEITOS BÁSICOS

O objetivo deste capítulo é apresentar a revisão de conceitos fundamentais para o entendimento deste trabalho. Primeiramente, é feita uma breve introdução à Bioinformática com foco em técnicas para alinhamento e buscas de sequências em bases de dados. Em seguida, são discutidas ferramentas de computação paralela e distribuída para processamento de grandes volumes de dados.

2.1 BIOINFORMÁTICA E GENÔMICA COMPUTACIONAL

O termo bioinformática foi definido por Hesper e Hogeweg (1970 apud HOGEWEG, 2011) como "o estudo de processos informáticos em sistemas biológicos". No entendimento dos autores, uma das propriedades que definem a vida é o processamento de informação em suas variadas formas. Citam como exemplos o acúmulo de informação durante a evolução, a transmissão de informação através do DNA e a interpretação desta informação em múltiplos níveis. Naquele momento, os principais tópicos em estudo estavam relacionados à Biologia Teórica, como a análise de padrões e modelagem dinâmica de processos biológicos.

Em Hogeweg (2011), o autor revisita o termo e afirma que, a partir do fim da década de 1980, este adquiriu um novo significado. Com o crescimento exponencial das bases de dados discutido adiante, bioinformática tornou-se sinônimo de genômica computacional. Isto é, a utilização e desenvolvimento de métodos computacionais para gestão e análise de dados de sequências genéticas, determinação de estruturas de proteínas e filogenia. Cristianini e Hahn (2007) concordam com esta visão, e afirmam que uma das principais formas de se analisar dados genéticos é através do alinhamento de sequências.

Nesta seção, serão apresentados os conceitos básicos da genética relacionados a alinhamentos de sequência. Em seguida, é feita a revisão dos algoritmos de alinhamentos e de ferramentas para busca de sequências em bases de dados.

2.1.1 Fundamentos da Genética

Todas as formas de vida conhecidas são constituídas por ácidos nucleicos e proteínas. O ácido desoxirribonucleico (DNA, do inglês *Deoxyribonucleic Acid*) é uma molécula que carrega as informações genéticas de um organismo (SELZER; MARHÖFER; KOCH, 2018). Consiste em duas fitas, de orientações opostas. As fitas são longas cadeias de nucleotídeos, moléculas menores caracterizadas por uma base nitrogenada: Adenina, Citosina, Guanina ou Timina. As bases são representadas por suas iniciais: A, C, G ou T. As fitas mantém-se ligadas quimicamente pelo pareamento entre as bases. A Adenina de uma fita liga-se com a Timina da outra, e Citosina liga-se com Guanina. Por exemplo, a sequên-

cia ATGCATGC é considerada complementar à sequência TACGTACG (CRISTIANINI; HAHN, 2007).

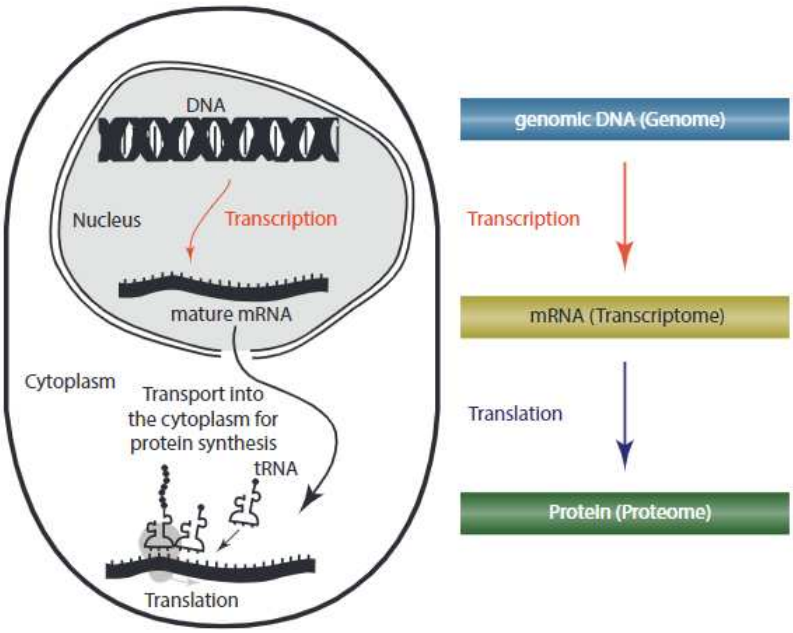
Quadro 1 – Os 20 aminoácidos e seus símbolos-padrão

Símbolo	**Aminoácido**	**Símbolo**	**Aminoácido**
A	Alanina	L	Leucina
R	Arginina	K	Lisina
N	Asparagina	M	Metionina
D	Ácido aspártico	F	Fenilalanina
C	Cisteína	P	Prolina
Q	Glutamina	S	Serina
E	Ácido glutâmico	T	Treonina
G	Glicina	W	Triptofano
H	Histidina	Y	Tirosina
I	Isoleucina	V	Valina

Fonte: Cristianini e Hahn (2007, p. 24)

O ácido ribonucleico (RNA, do inglês *Ribonucleic Acid*) participa da biosíntese de proteínas que controlam os processos celulares da vida. São formados por uma cadeia única de nucleotídeos. No RNA, a Timina é substituída por Uracila, representada pela letra U (SELZER; MARHÖFER; KOCH, 2018). As proteínas são moléculas que desempenham as mais variadas funções em um organismo. São formadas por cadeias de até 20 aminoácidos, listados no Quadro 1 (CRISTIANINI; HAHN, 2007).

Figura 1 – Dogma Central da Biologia Molecular



Fonte: Selzer, Marhöfer e Koch (2018, p. 6)

A relação entre ácidos nucleicos e proteínas é descrita pelo Dogma Central da Biologia Molecular, ilustrado na Figura 1. A informação genética é codificada pela cadeia de nucleotídeos do DNA. Para construir uma proteína, processos bioquímicos separam as fitas de DNA e sintetizam uma molécula de RNA mensageiro, complementar a uma das fitas. Em seguida, a sequência de nucleotídeos da molécula de RNA mensageiro é processada, três bases por vez. Esta tripla de bases, chamada de códon, codifica um aminoácido seguindo o código genético do Quadro 2. A tradução dos códons em aminoácidos termina quando uma das seguintes códons de parada for alcançado: TAA, TAG ou TCA. Após a tradução, a cadeia de aminoácidos passa por dobras que conferem uma estrutura tridimensional à proteína sintetizada (SELZER; MARHÖFER; KOCH, 2018).

Quadro 2 – O código genético padrão: códons e aminoácidos correspondentes

	A		**G**		**C**		**T**	
A	AAA	K	AGA	R	ACA	T	ATA	I
	AAG	K	AGG	R	ACG	T	ATG	M
	AAC	N	AGC	S	ACC	T	ATC	I
	AAT	N	AGT	S	ACT	T	ATT	I
G	GAA	E	GGA	G	GCA	A	GTA	V
	GAG	E	GGG	G	GCG	A	GTG	V
	GAC	D	GGC	G	GCC	A	GTC	V
	GAT	D	GGT	G	GCT	A	GTT	V
C	CAA	Q	CGA	R	CCA	P	CTA	L
	CAG	Q	CGG	R	CCG	P	CTG	L
	CAC	H	CGC	R	CCC	P	CTC	L
	CAT	H	CGT	R	CCT	P	CTT	L
T	TAA	*	TGA	*	TCA	S	TTA	L
	TAG	*	TGG	W	TCG	S	TTG	L
	TAC	Y	TGC	C	TCC	S	TTC	F
	TAT	Y	TGT	C	TCT	S	TTT	F

Fonte: Cristianini e Hahn (2007, p. 27)

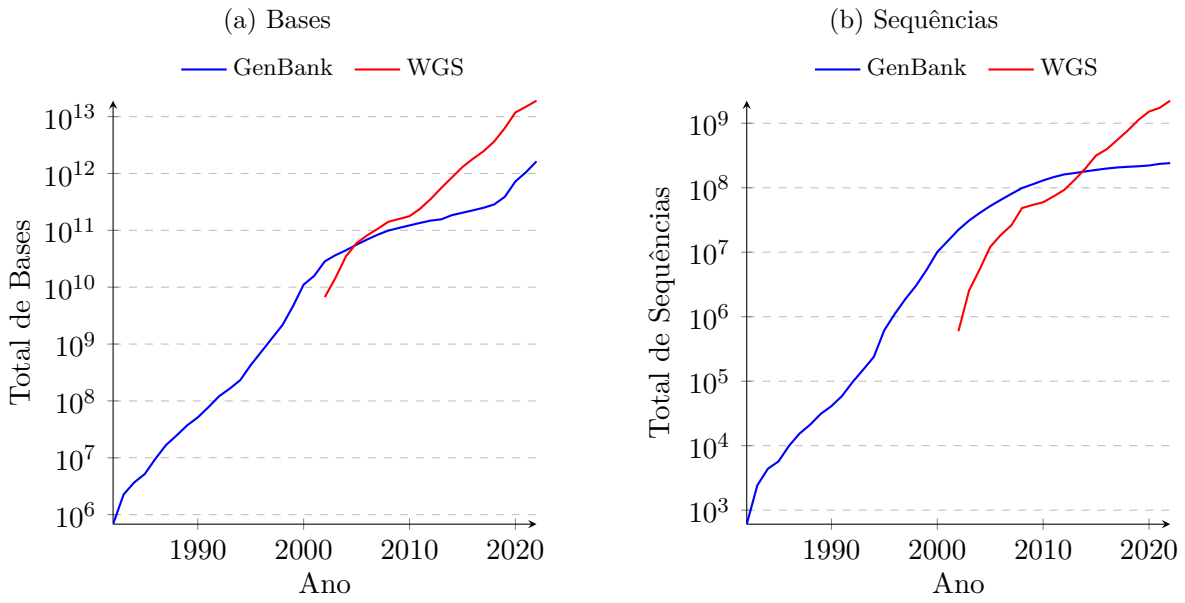
O genoma é a sequência completa do DNA de um organismo. As regiões de DNA que codificam uma proteína são chamadas de gene. Os genes são considerados unidades discretas de hereditariedade, pois transmitem as informações de um indivíduo para seus descendentes. O estudo das sequências genéticas e das funções dos genes permite entender como indivíduos e espécies se relacionam. A comparação entre duas sequências genéticas de indivíduos diferentes permite avaliar a similaridade entre elas. Sequências similares sugerem que os genes podem compartilhar um ancestral em comum, mas também podem ocorrer ao acaso. Genes que descendem de um ancestral comum e não são similares pelo simples acaso são chamados de homólogos (CRISTIANINI; HAHN, 2007). Aqueles que descendem de um ancestral comum, possuem a mesma função, mas pertencem a espécies diferentes são chamados de ortólogos. Parólogos são os genes homólogos que possuem

funções diferentes em uma mesma espécie (SELZER; MARHÖFER; KOCH, 2018).

2.1.2 Busca de Sequências

Uma das tarefas mais importantes na bioinformática é a busca de sequências em bases de dados, que consiste em identificar regiões de alta similaridade entre duas ou mais sequências distintas. Este procedimento viabiliza, por exemplo, a inferência de função de uma proteína ainda desconhecida a partir da comparação da sequência de aminoácidos desta com sequências de outras proteínas cuja função é conhecida (CRISTIANINI; HAHN, 2007).

Figura 2 – Estatísticas do GenBank e WGS de 1982 a 2022



Fonte: Adaptado de NCBI (2023a)

O GenBank (SAYERS et al., 2023), por exemplo, é uma base de dados pública, mantida pelo *National Center for Biotechnology Information* (NCBI), que reúne sequências de nucleotídeos e respectivas anotações. Cada entrada no GenBank inclui uma descrição da sequência, nome científico e taxonomia do organismo, além de referências bibliográficas e áreas significativas do ponto de vista biológico, como regiões codificantes para determinadas proteínas. A Figura 2 mostra que o volume de dados disponibilizados no GenBank cresceu significativamente desde 1982 e, mais recentemente, vem dobrando de tamanho a cada dois anos. A figura também mostra a evolução do *Whole Genome Shotgun* (WGS), uma das divisões do GenBank onde publicam-se sequências em construção, com ou sem anotações, atualizadas conforme o sequenciamento em questão progride.

2.1.2.1 Formato FASTA

O GenBank utiliza o formato FASTA para distribuir sequências de nucleotídeos e aminoácidos. Criado como padrão de entrada para o programa de alinhamento de sequências homônimo (PEARSON, 1990), tornou-se amplamente aceito por outras ferramentas até os dias atuais. A Figura 3 mostra um trecho de arquivo FASTA da sequência de DNA de um exemplar do vírus SARS-CoV-2 obtido do NCBI Virus[1] (HATCHER et al., 2017).

Figura 3 – Exemplo de sequência de DNA complementar do vírus SARS-CoV-2 em formato FASTA

```
>ON642517.1 |Severe acute respiratory syndrome coronaviru...
AGATCTGTTCTCTAAACGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAG
TGCACTCACGCAGTATAATTAATAACTAATTACTGTCGTTGACAGGACACGAGTAACTCG
TCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTGTTGCAGCCGATCATCAGCACATCT
AGGTTTTGTCCGGGTGTGACCGAAAGGTAAGATGGAGAGCCTTGTCCCTGGTTTCAACGA
GAAAACACACGTCCAACTCAGTTTGCCTGTTTTACAGGTTCGCGACGTGCTCGTACGTGG
CTTTGGAGACTCCGTGGAGGAGGTCTTATCAGAGGCACGTCAACATCTTAAAGATGGCAC
TTGTGGCTTAGTAGAAGTTGAAAAAGGCGTTTTGCCTCAACTTGAACAGCCCTATGTGTT
CATCAAACGTTCGGATGCTCGAACTGCACCTCATGGTCATGTTATGGTTGAGCTGGT...
```

Fonte: GenBank (2022)

O formato FASTA consiste em arquivo de texto plano que pode conter uma ou mais sequências genéticas codificadas conforme os alfabetos de nucleotídeos e aminoácidos mostrados na subseção 2.1.1. Uma sequência é identificada por uma linha de texto iniciada pelo caracter `>`. O restante da linha contém a descrição da sequência, que pode incluir campos como seu código de identificação, informações sobre a espécie, grupo de pesquisa que realizou o sequenciamento, dentre outros. As linhas seguintes contêm os caracteres que compõem a sequência em si. A sequência termina quando uma nova linha iniciada por `>` é encontrada, ou quando o fim do arquivo é alcançado.

2.1.2.2 Métodos de Alinhamento

O alinhamento de sequências pode ser entendido como um alinhamento entre cadeias de caracteres que representam uma sequência genética. Orengo, Jones e Thornton (2003) afirmam que, para compreender este processo, é importante considerar possíveis mutações durante a evolução de um organismo, que inclui a substituição, exclusão ou inclusão de nucleotídeos. A qualidade de um alinhamento de sequências pode ser melhorada ao permitir a inclusão de espaços nas sequências de entrada. A Figura 4 mostra um exemplo de alinhamento entre as sequências TGCTACGA e TCTAGA.

É possível observar que duas sequências de entrada podem ser alinhadas de formas diferentes, dependendo dos espaços inseridos no processo de alinhamento. Portanto, é

[1] NCBI Virus – https://www.ncbi.nlm.nih.gov/labs/virus/vssi/

Figura 4 – Exemplo de alinhamento não ótimo entre duas sequências de nucleotídeos

T	G	C	T	A	C	G	A
|		|	|	|			|
T	-	C	T	A	G	-	A

necessário haver um critério de pontuação para encontrar o alinhamento ótimo, ou seja, aquele com maior pontuação dentre todos os alinhamentos possíveis. Um critério simples, por exemplo, consiste em somar 1 ponto para cada posição equivalente entre duas sequências. Considerando este critério, o alinhamento da Figura 4 soma 5 pontos. Nota-se, contudo, que é possível melhorar esta pontuação alinhando-se as guaninas mais à direita em ambas as sequências. A Figura 5 mostra o alinhamento ótimo entre as sequências TGCTACGA e TCTAGA, com 5 pontos.

Figura 5 – Alinhamento ótimo entre duas sequências de nucleotídeos

T	G	C	T	A	C	G	A
|		|	|	|		|	|
T	-	C	T	A	-	G	A

Encontrar o alinhamento ótimo dentre todos os possíveis, entretanto, não é uma tarefa trivial. Needleman e Wunsch (1970) propuseram um algoritmo utilizando técnicas de programação dinâmica para encontrar o alinhamento ótimo entre duas sequências de ponta a ponta. O algoritmo utiliza uma matriz que é preenchida conforme uma função de pontuação e penaliza a inclusão de espaços nas sequências de entrada. Terminados os cálculos, o algoritmo constrói o alinhamento percorrendo a matriz partindo do canto inferior direito até o canto superior esquerdo. Assim, o algoritmo de Needleman-Wunsch calcula o alinhamento global ótimo entre duas sequências.

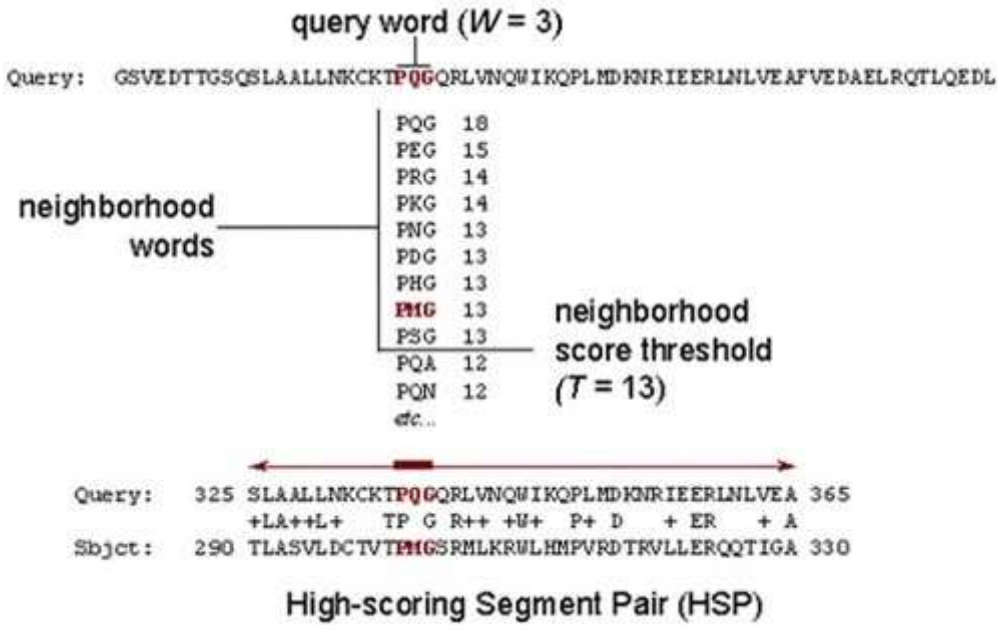
Smith, Waterman e Fitch (1981), por outro lado, propuseram um modelo diferente onde espaços para maximizar o alinhamento não são penalizados. Além disso, o caminho percorrido para construir o alinhamento parte da célula da matriz que contiver a pontuação mais alta e termina na primeira célula de valor 0. Com essa premissa, o algoritmo de Smith-Waterman identifica regiões de alta similaridade entre duas sequências, realizando portanto um alinhamento local ótimo.

A complexidade algorítmica de tempo e espaço de ambos os métodos é da ordem de $O(mn)$, onde m e n são os comprimentos das sequências de entrada. Portanto, o custo computacional em termos de tempo de execução e de utilização de memória RAM torna-os inviáveis para alinhamentos em larga escala.

2.1.2.3 BLAST

O BLAST (*Basic Local Alignment Search Tool*) é um pacote de ferramentas otimizadas para encontrar alinhamentos locais entre uma ou mais sequências de consulta contra sequências alvo armazenadas em bases de dados (ALTSCHUL et al., 1990).

Figura 6 – O algoritmo de busca do BLAST



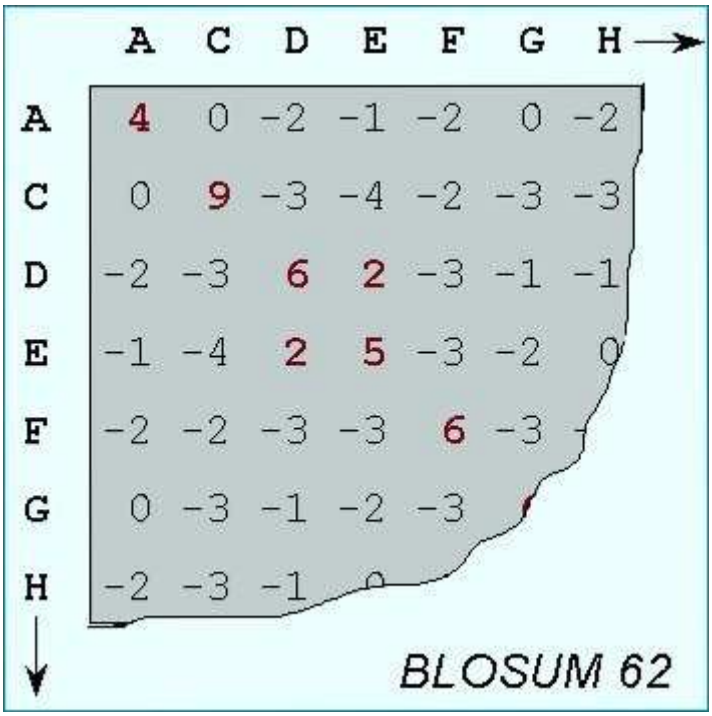
Fonte: Fassler e Cooper (2011)

O algoritmo consiste em dividir a sequência de entrada em palavras de comprimento fixo W. Cada palavra é utilizada como semente para buscar subsequências correspondentes na base de dados, composto por palavras de tamanho idêntico derivadas das sequências alvo. A Figura 6 mostra o alinhamento entre o fragmento PQG da consulta e o segmento correspondente PMG de um alvo da base de dados.

A pontuação S - do inglês, *score* - dos alinhamentos iniciais é calculada como a soma das pontuações de substituição e de inclusão de espaços. As substituições podem ser pontuadas de acordo com matrizes de substituição, como por exemplo a BLOSUM (Figura 7) ou PAM, ou utilizando penalidades simples (FASSLER; COOPER, 2011). Para a fase seguinte, são selecionados alinhamentos cuja pontuação S é igual ou superior a um limite T definido pelo usuário.

Os alinhamentos são estendidos pelo algoritmo em ambas as direções até que o fim das sequências seja alcançado, ou até que a pontuação atinja um valor inferior ao limite definido. Desta forma, o BLAST encontra alinhamentos entre pares de segmentos cuja pontuação é localmente máxima com respeito à busca realizada. Isto é, a pontuação não aumenta se encurtá-lo ou estendê-lo. Estes pares de segmento de alta pontuação são denotados pela sigla HSP (do inglês, *High-scoring Segment Pair*).

Figura 7 – Fragmento da matriz de pontuação BLOSUM 62



Fonte: Fassler e Cooper (2011)

A pontuação, entretanto, não é suficiente para avaliar a significância biológica de um alinhamento. Por exemplo, sequências de DNA de baixa complexidade, com repetições de bases, podem impactar negativamente os resultados encontrados. Os autores desenvolveram um conjunto de estatísticas para avaliar a significância de um alinhamento, conhecidas como estatísticas de Karlin-Altschul[2]. O *e-value* é a quantidade esperada de alinhamentos com pontuação igual ou superior a S que podem ocorrer ao acaso na base de dados em questão. Quanto mais próximo de zero, mais significativo é o alinhamento. Isto é, maiores são as chances do algoritmo ter identificado sequências homólogas de fato.

O BLAST é mais eficiente do que os métodos baseados em programação dinâmica. Sua implementação inicial, contudo, era sequencial e precisava escalar para processar o volume cada vez maior de dados genômicos. No Capítulo 3 serão apresentados trabalhos que otimizaram o BLAST aplicando técnicas de computação paralela e distribuída.

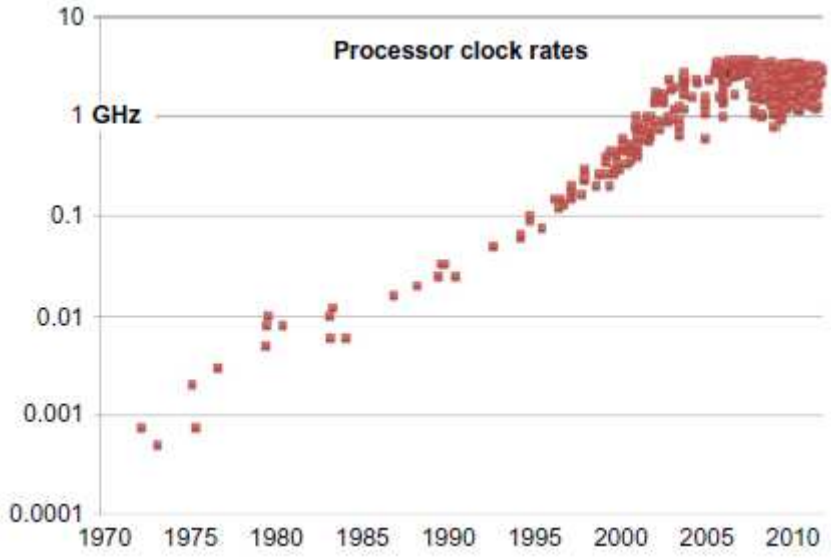
2.2 COMPUTAÇÃO PARALELA E DISTRIBUÍDA

Em 1965, George Moore, co-fundador da Intel, identificou que o número de transistores integrados em chips de silício dobravam aproximadamente a cada dois anos, uma observação que ficou conhecida como a Lei de Moore. O crescimento do número de transistores, e também os avanços na tecnologia destes componentes, viabilizaram o desenvolvimento

[2] As estatísticas são complexas, e o detalhamento da equação de Karlin-Altschul para o cálculo de *e-value* não é relevante para este trabalho.

de arquiteturas complexas. A velocidade dos microprocessadores (Figura 8) cresceu de 1 MHz em 1973 para 1 GHz em 2003, alcançou 3 GHz em 2005 e pouco cresceu desde então. Os principais fatores limitantes são o aumento no consumo de energia, dissipação de calor, e a latência e largura de banda dos chips de memória RAM. O desempenho de programas sequenciais tende a ser beneficiado por *clocks* maiores, pois este indica a quantidade de instruções que o processador pode executar por segundo. Isto ocorre sem qualquer necessidade de alteração de código-fonte (MCCOOL; REINDERS; ROBISON, 2012).

Figura 8 – Tendência de crescimento da taxa de *clock* dos microprocessadores a partir de 1970



Fonte: McCool, Reinders e Robison (2012, p. 9)

A partir de 2005, passou-se a priorizar a inclusão de mais núcleos de CPU para expandir a capacidade dos processadores. Para obter ganho de desempenho, divide-se um programa em múltiplas linhas de execução – ou *threads* –, executadas em paralelo por cada núcleo conforme agendamento definido pelo sistema operacional. Chambers e Zaharia (2018) acrescentam que, a partir do mesmo ano, o custo de armazenamento de dados seguiu tendência contrária à dos *clocks*. O preço de armazenamento de 1 TB de dados caiu pela metade a cada 14 meses, e permitindo que organizações coletassem, armazenassem e processassem grandes quantidades de informação.

Inicialmente, Cox e Ellsworth (1997) chamaram de *Big Data* a classe de problemas que envolvia o processamento de dados que não cabiam na memória RAM ou nos dispositivos de armazenamento de uma máquina. Ao longo dos anos, o termo adquiriu novo significado. McAfee e Brynjolfsson (2012) afirmam que a diferença entre o processamento de dados convencional e *Big Data* não está apenas no volume de informação, mas também na variedade de fontes e formatos, e a velocidade na qual os dados são gerados. Entende-se

então que esta classe de problemas requer computação de alto desempenho, em paralelo, pois uma máquina não é suficiente para armazenar ou processar todo o conjunto de dados. Por isso, utilizam-se *clusters* de computadores para a distribuição de tarefas entre máquinas – ou nós – que comunicam-se via rede e compartilham dados por meio de sistemas de arquivos distribuídos. Há décadas, são utilizadas grades (*grids*) para computação de alto desempenho. Grades são *clusters* de computadores que acessam um sistema de arquivos compartilhados hospedado em uma rede de área de armazenamento (SAN, do inglês *Storage Area Network*). Este modelo é adequado para tarefas cujo processamento faz uso intenso da CPU. Para algoritmos de *Big Data* que acessam volumes de dados massivos, o tráfego ne rede torna-se um gargalo, pois há risco de nós ficarem ociosos enquanto aguardam o término de uma transferência (WHITE, 2015).

A partir da década de 2000, popularizou-se o modelo de serviço de computação em nuvem. Em uma nuvem pública, o usuário contrata infraestrutura, plataforma e softwares como serviços sob demanda, e paga apenas pelos recursos utilizados durante o tempo contratado. Os recursos são gerenciados por um provedor, que garantem o nível de serviço conforme acordado em contrato. Os principais fornecedores de serviços de nuvem pública são a *Amazon Web Services* (AWS), *Google Cloud Platform* (GCP) e *Microsoft Azure*. Também é possível que uma organização mantenha sua própria infraestrutura de computação em nuvem. No modelo de nuvem privada, a organização provendencia um catálogo de serviços análogo aos de uma nuvem pública a seus usuários internos. Cabe à organização a manutenção da infraestrutura necessária para operar o serviço, a implantacão de mecanismos de segurança e de governança para gestão dos recursos computacionais.

2.2.1 Controle de Execução

Cormen et al. (2009) definem que um algoritmo paralelo é determinístico quando, para uma mesma entrada, o resultado é sempre o mesmo, independendente do modo que as tarefas são agendadas ou distribuídas. Quando há dependência entre tarefas, é necessário implementar mecanismos de sincronização, do contrário o programa comporta-se de forma não determinística.

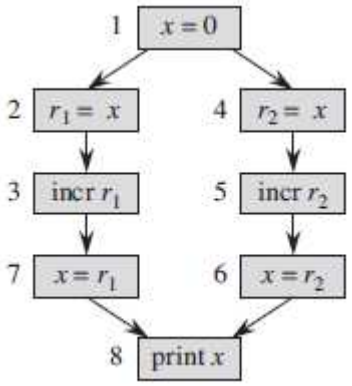
O comportamento não-determinístico advém de condições de corrida, que ocorrem quando múltiplas instruções acessam um mesmo endereço de memória, e pelo menos uma delas é uma operação de escrita. O Código 2.1 descreve um algoritmo simples para incrementar o valor de uma variável duas vezes, em paralelo e sem sincronização. O comando `parallel for` inicializa duas *threads* que executam a operação `x = x + 1`. Esta operação é executada em três instruções pela CPU: primeiro guarda-se o valor de x em um registrador r, incrementa-se o valor no registrador, e escreve-se o resultado de volta em x.

A Figura 9 mostra um exemplo de fluxo de execução do Código 2.1 onde a ordem de agendamento das instruções resultou em erro. O programa inicia com $x = 0$. Nos *steps* 2

Código 2.1 – Algoritmo sujeito a condição de corrida, adaptado de Cormen et al. (2009)

```
1 x = 0
2 parallel for i = 1 to 2
3     x = x + 1
4 print x
```

Figura 9 – Fluxo de execução do Código 2.1, onde observa-se condição de corrida entre duas *threads*



step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

Fonte: Cormen et al. (2009, p. 789)

e 3, a *thread* à esquerda guarda o valor de x no registrador r_1 e incrementa o valor em r_1. Contudo, antes que esta completasse a operação de incremento, a *thread* à direita guarda o valor de x, ainda 0, em r_2, incrementa r_2 e escreve o resultado em x, que assume o valor 1. No *step* 7, a primeira *thread* conclui sua execução e escreve o resultado armazenado em r_1, igual a 1, em x. Finalmente, o programa imprime no *step* 8 o valor 1, quando o esperado seria 2! Este erro pode ser corrigido se o acesso à variável x for mutuamente exclusivo. A exclusão mútua é um mecanismo de sincronização onde uma *thread* bloqueia o acesso de outras *threads* a uma região de memória compartilhada enquanto uma operação estiver sendo realizada. O Código 2.2 mostra o algoritmo corrigido. A *thread* que invocar o comando `lock` obtém a trava se estiver disponível, e então prossegue com a execução do programa adentrando a chamada região crítica – neste caso, a linha 4 do código. Como consequência desta implementação, o comando `parallel for` degenera para uma execução sequencial e, portanto, a paralelização não oferece vantagem neste caso. Inclusive, a sobrecarga causada pela aquisição e liberação das travas pode prejudicar o desempenho do algoritmo, que tende a desempenhar melhor em sua forma puramente sequencial.

O exemplo anterior mostra que mesmo um programa simples pode exigir adaptações para executar corretamente em paralelo. A exclusão mútua, quando mal implementada, pode levar uma *thread* a aguardar indefinidamente pelo acesso à região crítica, resultando em um *deadlock*. Plataformas de concorrência como o Cilk e OpenMP providenciam uma camada de software para coordenação, agendamento e gerenciamento que abstraem

Código 2.2 – Algoritmo com sincronização por exclusão mútua, adaptado de Cormen et al. (2009)

```
1 x = 0
2 parallel for i = 1 to 2
3     lock x
4     x = x + 1
5     unlock x
6 print x
```

parte da complexidade de criação e sincronização de *threads* (CORMEN et al., 2009). Problemas análogos podem acontecer em algoritmos distribuídos, que requerem mecanismos de gestão de recursos de *cluster*, coordenação de tarefas, controle de transações executadas pelo sistema e tolerância a falhas. Neste caso, o usuário precisa implementar estes mecanismos utilizando interfaces para programação de aplicações (API, do inglês *Application Programming Interface*) como o *Message Passing Interface* (MPI). O MPI possibilita o controle do fluxo de execução por troca de mensagens via rede, mas requer que o programador atente-se explicitamente aos requisitos mencionados (WHITE, 2015).

Conclui-se então que a implementação de algoritmos paralelos e distribuídos não é trivial. Porém, plataformas de *Big Data*, como o Apache Hadoop, permitem que programadores sem conhecimento técnico avançado criem algoritmos distribuídos e escaláveis.

2.2.2 Apache Hadoop

Na década de 2000, o Google publicou dois importantes trabalhos na área de computação distribuída. No artigo de Ghemawat, Gobioff e Leung (2003), foi descrito o *Google File System* (GFS), o sistema de arquivos do Google. O sistema foi desenvolvido internamente pela companhia para armazenar centenas de terabytes distribuídos entre milhares de máquinas. Estas máquinas eram computadores convencionais, mais baratos, com discos rígidos suscetíveis a falhas. Um dos diferenciais do GFS foi assumir que o *hardware*, com certeza, iria falhar. Para garantir a resiliência do sistema, os arquivos são armazenados em blocos, e cada bloco replicado automaticamente entre os nós do *cluster*. Desta forma, caso um nó falhasse, cópias das partições perdidas estariam disponíveis em outras máquinas. A divisão em blocos permite que um conjunto de dados qualquer seja acessado em paralelo pelos diferentes nós que contenham suas partições.

Em Dean e Ghemawat (2004), foi apresentado o MapReduce, um modelo de programação que oferece uma API para processamento de dados em larga escala. O projeto se inspirou no paradigma de linguagens de programação funcionais como o LISP, emprestando o nome das rotinas *map* e *reduce*. O usuário deve implementar apenas estas duas funções para processar dados armazenados no GFS. A plataforma se encarrega de distribuir a execução das tarefas, monitorá-las e resubmetê-las em caso de falha em um

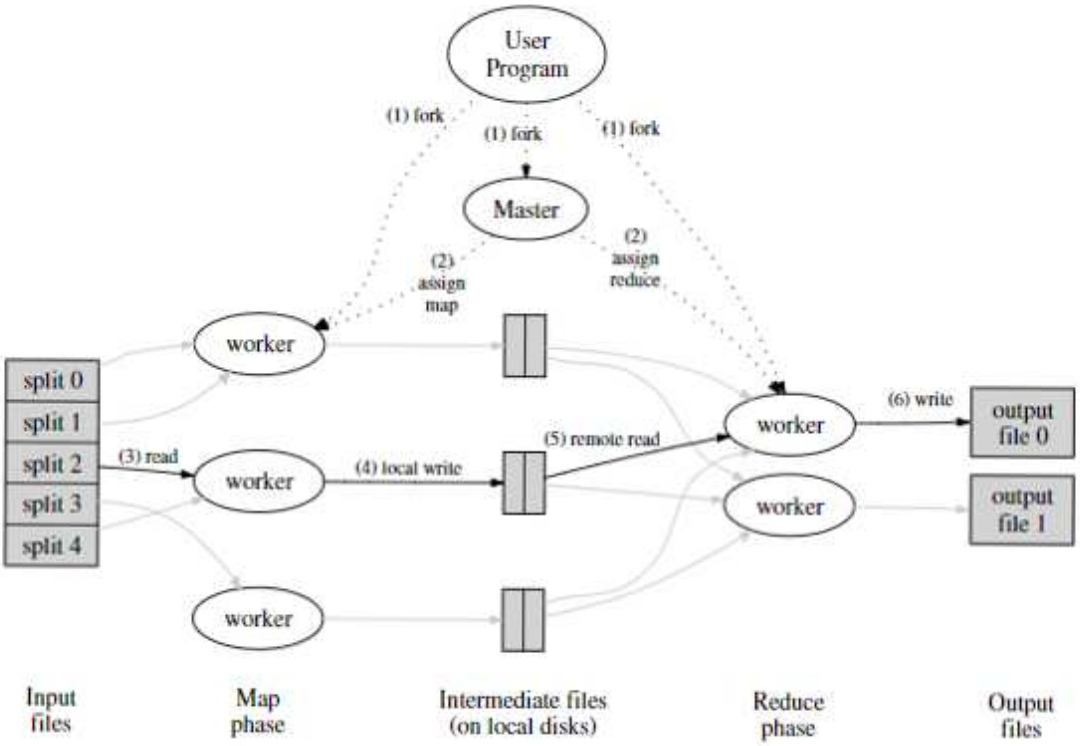
dos nós.

GFS e MapReduce, entretanto, eram *softwares* proprietários e já não são mais utilizados internamente pelo Google. Mas, conforme afirmaram Dean e Ghemawat (2004), a interface do MapReduce pode ser implementada de várias maneiras, conforme a necessidade e as característics do ambiente de execução. Em 2006, foi lançado o Apache Hadoop, implementação em código aberto do modelo MapReduce. O Hadoop é composto por três principais componentes: o MapReduce; o *Hadoop Distributed File System*, sistema de arquivos distribuído baseado no GFS; e o *Yet Another Resource Negotiatior* (YARN), o negociador de recursos encarregado de gerenciar as aplicações MapReduce (WHITE, 2015).

2.2.2.1 MapReduce

MapReduce é um modelo de programação inerentemente paralelo onde processamento é dividido em duas fases: *map* e *reduce*. O Hadoop disponibiliza uma API em Java onde o programador pode criar um *job* implementando as classes *Mapper* e *Reducer* (WHITE, 2015). A Figura 10 mostra a visão geral da execução de um programa MapReduce. Nela, vê-se que o nó *Master* distribui as tarefas para os *workers* correspondentes à fase em que atuam. Este processo é transparente para o usuário.

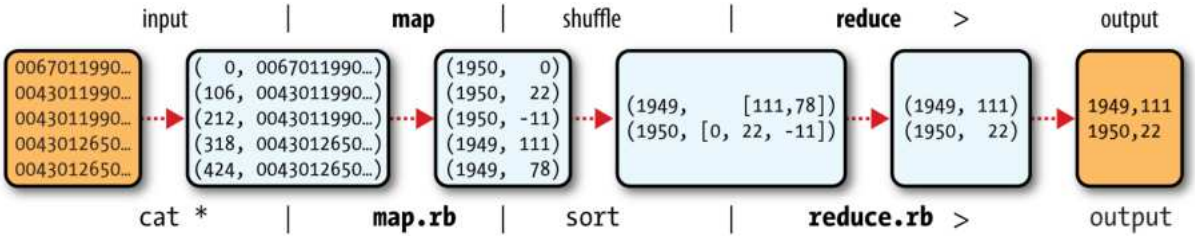
Figura 10 – Visão geral da execução de uma aplicação MapReduce



Fonte: Dean e Ghemawat (2004)

Durante o *map*, o *Mapper* processa um conjunto de pares chave-valor derivados de uma ou mais partições de entrada. A saída é um conjunto de pares chave-valor intermediários escritos no disco local. O motor de execução agrupa automaticamente os pares intermediários que possuam a mesma chave. Esta fase intermediária é chamada de *shuffle*. Na fase *reduce*, o *Reducer* recebe uma chave e o conjunto de valores agrupados correspondente àquela chave. Geralmente, uma operação de *reduce* executa agregações sobre o conjunto de valores recebidos, e produz apenas um ou nenhum valor de saída. Os resultados são escritos no sistema de arquivos ao fim do processo (DEAN; GHEMAWAT, 2004).

Figura 11 – Fluxo de dados em um programa MapReduce



Fonte: White (2015, p. 24)

A Figura 11 ilustra o fluxo de dados em uma aplicação MapReduce. Neste exemplo, a entrada foi um conjunto de arquivos de texto com séries históricas de dados meteorológicos. A plataforma leu estes arquivos e gerou pares de chave-valor para o *Mapper*, onde a chave representa o *offset* da linha no arquivo, e o valor é o conteúdo desta linha. Para cada entrada, o *map* desprezou a chave e gerou um par de chave-valor intermediário, respectivamente o ano correspondente à data da linha processada, e a temperatura observada. Finalmente, os *Reducers* receberam, para cada ano, uma lista de temperaturas observadas. Na saída, foram escritas tuplas com o ano, e a temperatura máxima observada naquele ano.

É interessante observar que, no exemplo de White (2015), as funções *map* e *reduce* não foram implementadas com a API em Java. Foram utilizados scripts em Ruby – `map.rb` e `reduce.rb` – para o processamento. Isto é possível graças ao *Hadoop Streaming*, um utilitário que permite a execução de *jobs* MapReduce utilizando programas externos. Para utilizar o *Streaming*, o programa deve receber dados pela entrada-padrão `stdin` e escrever o resultado na saída-padrão `stdout`. Esta funcionalidade confere flexibilidade ao MapReduce, pois possibilita diversificar o processamento de dados utilizando ferramentas que, de outro modo, não poderiam integrar-se à API em Java.

2.2.2.2 Hadoop Distributed File System

O HDFS, conforme mencionado anteriormente, é baseado no GFS. O sistema de arquivos é organizado em blocos, que são a menor unidade de dados que pode ser lida ou

escrita pelos clientes. Por padrão, um bloco HDFS possui 128 MB. Arquivos maiores que o tamanho-padrão são particionados em blocos menores e cada partição é armazenada como uma unidade independente (WHITE, 2015).

Como consequência da divisão em blocos, o HDFS pode armazenar arquivos maiores que a capacidade individual de um disco rígido no *cluster*. A gestão de recursos de armazenamento é simplificada pelo tamanho fixo dos blocos, é fácil para o HDFS calcular quanto blocos podem ser armazenados em um nó. E, conforme mencionado na subseção 2.2.2, a replicação de blocos aumenta a resiliência do sistema caso um nó seja perdido.

Os nós que armazenam blocos de dados são chamados de *Datanodes*. Aquele que mantém a lista de arquivos, metadados e árvore de diretórios do HDFS é o *Namenode*. Ao ler um arquivo, o cliente solicita ao *Namenode* a localização dos blocos, e em seguida solicita os dados aos respectivos *Datanodes*. Para escrever, o cliente requisita a criação de um novo arquivo e, se autorizado, escreve os dados em uma fila. O *Namenode*, por sua vez, determina os *Datanodes* que guardarão os blocos provenientes do cliente observando a capacidade de armazenamento disponível nos nós e o fator de replicação mínimo (WHITE, 2015).

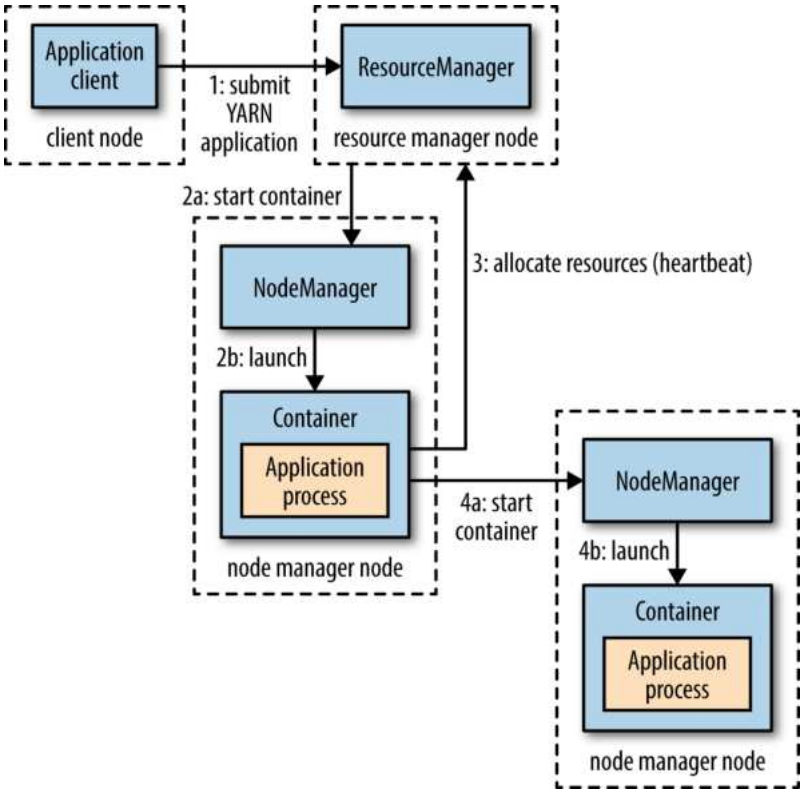
Como o *Namenode* guarda a localização de todos os blocos no *cluster* em memória, qualquer falha neste agente pode inutilizar o HDFS. Para mitigar riscos, o *Namenode* efetua cópias de segurança periódicas do estado do sistema de arquivos em um sistema de arquivos externo. Outra forma envolve adicionar um *Namenode* secundário, que assume o lugar do nó primário em caso de falhas. O desempenho do HDFS pode ser impactado se arquivos minúsculos forem armazenados em excesso, pois uma lista de arquivos e diretórios muito grande pode resultar em falha por falta de memória no *Namenode* (WHITE, 2015).

2.2.2.3 Yet Another Resource Negotiator

O YARN é um negociador de recursos introduzido a partir da segunda versão do Hadoop para otimizar a execução do MapReduce. A ferramenta provê APIs para requisitar e operar recursos do *cluster*, o que permitiu que outras aplicações além do MapReduce fossem adaptadas para utilizá-lo. A coordenação e alocação de recursos computacionais é realizada pelo nó que executa o *ResourceManager*, um dos serviços do YARN. O *NodeManager* é outro serviço, executado em cada nó, e responsável por inicializar e monitorar a execução de contêineres. E os contêineres executam os processos da aplicação conforme os recursos alocados pelo *ResourceManager* (WHITE, 2015).

A Figura 12 mostra o fluxo de execução de uma aplicação no YARN. A execução começa quando o cliente submete uma aplicação, que assume o estado pendente. Primeiramente, o *ResourceManager* avalia se os recursos solicitados pela aplicação são alocáveis. Se positivo, a aplicação é aceita. O *ResourceManager* determina a inicialização do *ApplicationMaster* em um dos nós com recursos disponíveis. No nó, o *NodeManager* inicializa o contêiner. O *ApplicationMaster*, então, passa a gerenciar a execução da aplicação e

Figura 12 – Negociação de recursos por uma aplicação YARN



Fonte: White (2015, p. 80)

mantém seu estado atualizado junto ao *ResourceManager*. O *ApplicationMaster* pode inicializar ou destruir contêineres conforme a necessidade da aplicação e a disponibilidade de recursos no *cluster*. Ao fim da execução, os recursos são desalocados e liberados.

As aplicações podem solicitar ao YARN a inicialização de contêineres em nós específicos. Esta funcionalidade permite, por exemplo, que um *Mapper* seja inicializado no nó que armazena o bloco do HDFS a ser processado por ele. Ao levar a tarefa computacional para onde o dado está localizado, e não o contrário, minimiza-se o volume de dados trafegados pela rede. Considerar a localidade dos dados, portanto, beneficia algoritmos de *Big Data* (WHITE, 2015).

Em caso de falha por algum erro na aplicação, cabe ao *ApplicationMaster* reagendar a tarefa perdida. Se o próprio *ApplicationMaster* falhar, o YARN tentará executar a aplicação novamente até um limite definido em suas configurações. Neste caso, cabe à aplicação implementar mecanismos para evitar retrabalho, ou seja, reexecutar tarefas que finalizaram com sucesso na tentativa anterior. Caso algum *NodeManager* deixe de enviar o status de execução para o *ResourceManager*, pode ser excluído do *cluster* e quaisquer tarefas em execução serão perdidas. E, novamente, cabe ao *ApplicationMaster* resubmetê-las em outro nó (WHITE, 2015).

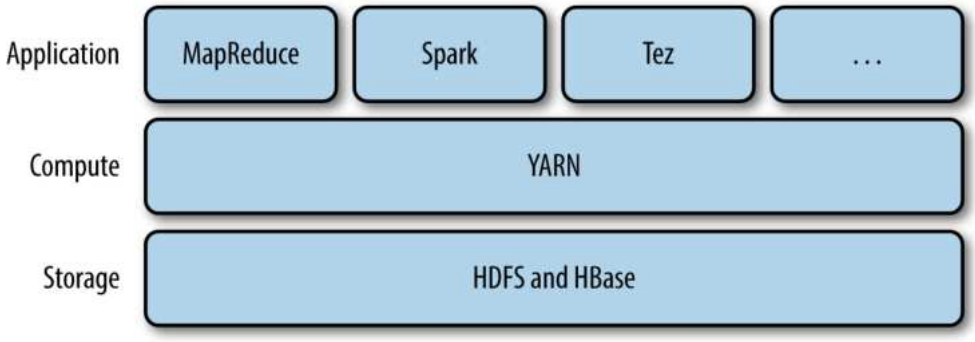
2.2.2.4 Limitações e Alternativas

Os subsistemas do Hadoop discutidos nesta seção abstraem parte significativa da complexidade que envolve computação paralela e distribuída. A API do MapReduce, assim como o *Streaming*, facilitam a implementação de aplicações distribuídas para problemas cujo fluxo de dados siga o padrão *map-shuffle-reduce*. Na prática, porém, é comum que o processamento de dados envolva sequências de transformações, agregações e junções com outros conjuntos (WHITE, 2015).

O MapReduce admite apenas um *Mapper* e um *Reducer* por *job*. Algoritmos que aplicam sequências de operações sobre um conjunto de dados, em MapReduce, são implementadas encadeando *jobs*, onde a saída de um *job* é utilizada como entrada do *job* seguinte. Esta é uma limitação da API do MapReduce, que não permite expressar de forma concisa séries de transformações sobre um conjunto de dados (WHITE, 2015). Também deve-se considerar que o tempo para agendamento de cada *job* e as sucessivas leituras e escritas em disco impactam negativamente o tempo de execução destes algoritmos em MapReduce (CHAMBERS; ZAHARIA, 2018).

De todo modo, o Hadoop deu origem a um ecossistema de ferramentas que facilitam este processo. O Apache Pig, por exemplo, fornece uma linguagem de *script* – *Pig Latin* – para programação de *pipelines* de transformação de dados. O Hive, criado pelo Facebook e também mantido pela Fundação Apache, é um sistema de *Data Warehousing* distribuído que permite a consulta de dados em larga escala utilizando a linguagem HiveQL, cuja sintaxe é bastante similar ao SQL adotado pelos SGBDs tradicionais. Internamente, ambas as ferramentas utilizam MapReduce para execução das tarefas (WHITE, 2015).

Figura 13 – Aplicações em *cluster* gerenciado pelo YARN



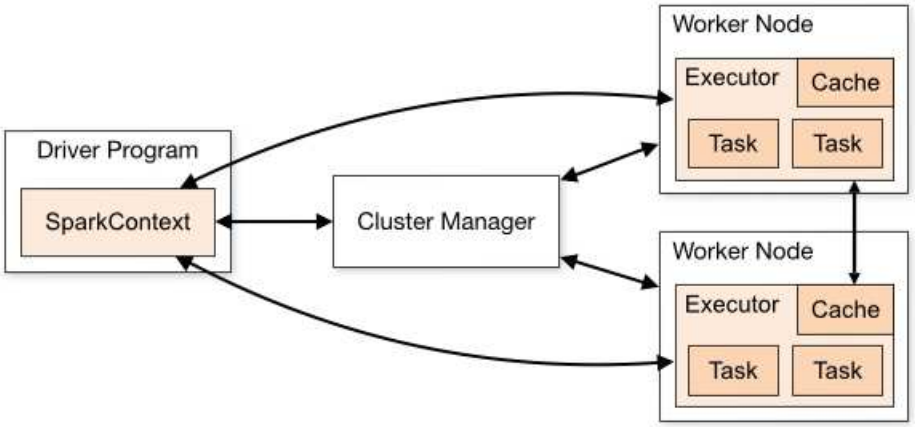
Fonte: White (2015, p. 79)

Há ainda ferramentas de *Big Data* associadas ao Hadoop, mas que não necessariamente compõem este ecossistema. A Figura 13 ilustra como os componentes de um cluster Hadoop se conectam. Na camada de aplicação, observam-se ferramentas que integram com YARN e HDFS, mas não dependem do MapReduce como motor de execução.

2.2.3 Apache Spark

O crescimento do volume de dados aumentou a demanda por processamento em larga escala. O MapReduce viabilizou a utilização de *clusters* de computadores convencionais para processamento massivo em lote garantindo, de forma transparente, a gestão eficiente de recursos computacionais e a tolerância a falhas. Mas há aplicações que dependem de operações em série sobre um conjunto de dado que, portanto, não encaixam no padrão *map-shuffle-reduce*. Por exemplo, algoritmos de aprendizado de máquina que aplicam uma função sucessivamente para otimizar parâmetros, ou análises interativas que envolvem diferentes consultas sobre um mesmo conjunto de dados. O Apache Spark[3], uma plataforma unificada para análise de dados em larga escala, foi criado como uma alternativa ao MapReduce. Seus principais diferenciais são o compartilhamento de dados em memória, e as APIs que permitem escrever concisamente operações e consultas complexas (ZAHARIA et al., 2010).

Figura 14 – Componentes de um *cluster* Spark



Fonte: The Apache Software Foundation (2023a)

Uma aplicação Spark consiste em um processo driver e um conjunto de processos executores. O *driver* é responsável pela distribuição de tarefas entre os executores, por monitorar o estado da aplicação e responder às interações do usuário. Os executores computam as tarefas designadas pelo *driver* e reportam o estado de execução. O *driver* pode incluir e remover executores conforme a carga de trabalho em andamento. A Figura 14 mostra estes atores e suas relações com os demais componentes do cluster. O Spark possui o próprio *Cluster Manager* (gerenciador de *cluster*), e oferece suporte ao Mesos, YARN e Kubernetes (The Apache Software Foundation, 2023b). Há ainda um modo local, indicado apenas para testes e pequenos experimentos, onde *driver* e executores são *threads* de uma mesma máquina (CHAMBERS; ZAHARIA, 2018).

[3] Apache Spark – http://spark.apache.org/

2.2.3.1 Interfaces para Programação de Aplicações

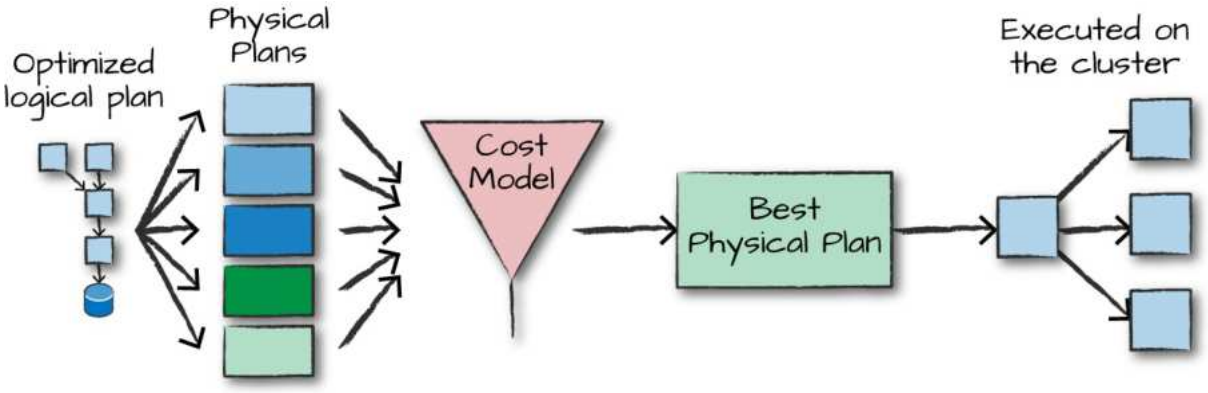
O Spark possui APIs de baixo e alto nível para manipulação de dados. As APIs de alto nível, que possuem maior nível de abstração, dividem-se em três: *Dataset*, *DataFrame* e Spark SQL (CHAMBERS; ZAHARIA, 2018). Estas APIs possuem métodos análogos às operações em tabelas de bancos de dados relacionais, como *select*, *where*, *join*, *groupBy*, dentre outras (The Apache Software Foundation, 2023b).

O *Dataset* consiste em uma coleção de objetos fortemente tipados, e por isso só está disponível para Java e Scala. A principal vantagem em sua utilização é a verificação, em tempo de compilação, de erros de sintaxe e de compatibilidade de tipos de dados (CHAMBERS; ZAHARIA, 2018).

O *DataFrame* abstrai o conceito de tabelas relacionais. Esta estrutura possui colunas nomeadas e tipos definidos por um esquema. O esquema da tabela pode ser declarado pelo usuário, ou inferido em tempo de execução baseado em uma amostra dos dados. Internamente, o *DataFrame* é equivalente a um *Dataset* de objetos do tipo *Row*: objeto genérico que representa uma coleção de campos genéricos (The Apache Software Foundation, 2023b). A sintaxe das operações em *DataFrame* são verificadas em tempo de compilação, mas a conformidade das linhas com o esquema do *DataFrame* ocorre em tempo de execução (CHAMBERS; ZAHARIA, 2018).

O Spark SQL possibilita a realização de consultas utilizando SQL, como um banco de dados relacional. Esta API facilita a migração de usuários que já conhecem SQL, mas não tem experiência com programação. As operações em Spark SQL e *DataFrame* são equivalentes, sem diferença de desempenho. A sintaxe da consulta, porém, é analisada apenas em tempo de execução (CHAMBERS; ZAHARIA, 2018).

Figura 15 – Geração de plano físico a partir das APIs estruturadas do Spark



Fonte: Chambers e Zaharia (2018, p. 63)

As manipulações de dados efetuadas com APIs estruturadas definem um plano lógico de execução. Antes de executá-lo, o Spark invoca o otimizador Catalyst (Figura 15), que aplica um conjunto de regras para gerar o plano lógico otimizado. O plano físico

escolhido corresponde a uma série de operações de baixo nível que é executada no *cluster*. Em síntese, o Spark compila código escrito com APIs de alto nível para RDD, uma API de baixo nível (CHAMBERS; ZAHARIA, 2018).

O *Resilient Distributed Dataset* (RDD) é a API mais antiga e de menor nível de abstração do Spark, mas que provê maior controle sobre o fluxo de dados da aplicação. Formalmente, o RDD é um conjunto de dados particionado e imutável, que só pode ser criado a partir de dados armazenados em sistemas de arquivos estáveis, ou a partir de outros RDDs. Cada RDD guarda a sequência lógica de conjunto de dados e operações utilizados para computá-lo, ou seja, preservam a linhagem de dados. A linhagem forma um grafo acíclico direcionado (DAG, do inglês *Directed Acyclic Graph*) indicando as dependências do RDD e suas partições. Em caso de falha, o Spark consegue computar novamente apenas as partições perdidas, sem reiniciar o processo inteiro (ZAHARIA et al., 2012). As operações em RDD possuem sintaxe similar às linguagens de programação funcionais. Por exemplo, *map*, *reduce*, *filter* (The Apache Software Foundation, 2023b).

2.2.3.2 Ações e Transformações

As operações que derivam RDDs a partir de outro são chamadas de transformações, e são classificadas em estreitas ou largas. As estreitas são aquelas onde cada partição de entrada contribui para apenas uma partição de saída. Por exemplo *map* e *filter*. Em transformações largas, uma partição de entrada contribui para mais de uma partição de saída. Este efeito, chamado de *shuffle* (ou embaralhamento), é o reparticionamento físico dos dados e pode ser custoso quando o volume de dados movido entre nós for significativo. São exemplos de transformações largas *sort*, *groupByKey* e *join* (CHAMBERS; ZAHARIA, 2018).

Também é possível transformar um RDD utilizando programas externos. A transformação *pipe*, similar ao *Hadoop Streaming*, executa um programa qualquer definido pelo usuário. O processo recebe os dados do RDD pela entrada-padrão `stdin`. A saída é escrita em `stdout`, onde cada linha é interpretada pelo Spark como um registro do RDD resultante (The Apache Software Foundation, 2023b).

As transformações são consideradas operações "preguiçosas" pois não são executadas de imediato. O RDD é efetivamente computado quando uma ação é invocada e dispara a execução de um *job*. Este tipo de operação retorna um valor para o *driver*, ou escreve o conteúdo do RDD em um sistema de arquivos. São exemplos de ações *collect*, *count* e *save* (ZAHARIA et al., 2012).

Um *job* em Spark é essencialmente a execução de uma ação, e é dividido em estágios. Cada estágio (*stage*) corresponde a um grupo de tarefas que podem ser executadas simultaneamente em múltiplas máquinas. Uma tarefa (*task*) contém um conjunto de transformações aplicadas a uma partição de dados em um único executor. O Spark tenta executar o máximo de tarefas em um mesmo estágio. Quando há transformações estrei-

tas planejadas em sequência, a execução ocorre em *pipeline*: o fluxo de dados de uma transformação para outra acontece em memória, sem escritas intermediárias em disco. As transformações largas, devido ao *shuffle*, resultam em escrita em disco e implicam na adição de um novo estágio ao *job* (CHAMBERS; ZAHARIA, 2018).

2.2.4 Apache Zeppelin

O Apache Zeppelin[4] é uma ferramenta para criação e edição de *notebooks* baseado na *web* com foco em análise de dados interativas e colaborativas. O Zeppelin é compatível com linguagens de programação tradicionalmente utilizadas por analistas de dados, como Python e R, e possui integração nativa com Spark. É possível criar e executar *jobs* Spark pelo *notebook* utilizando as APIs em Scala e Python, ou executar consultas estruturadas diretamente em Spark SQL.

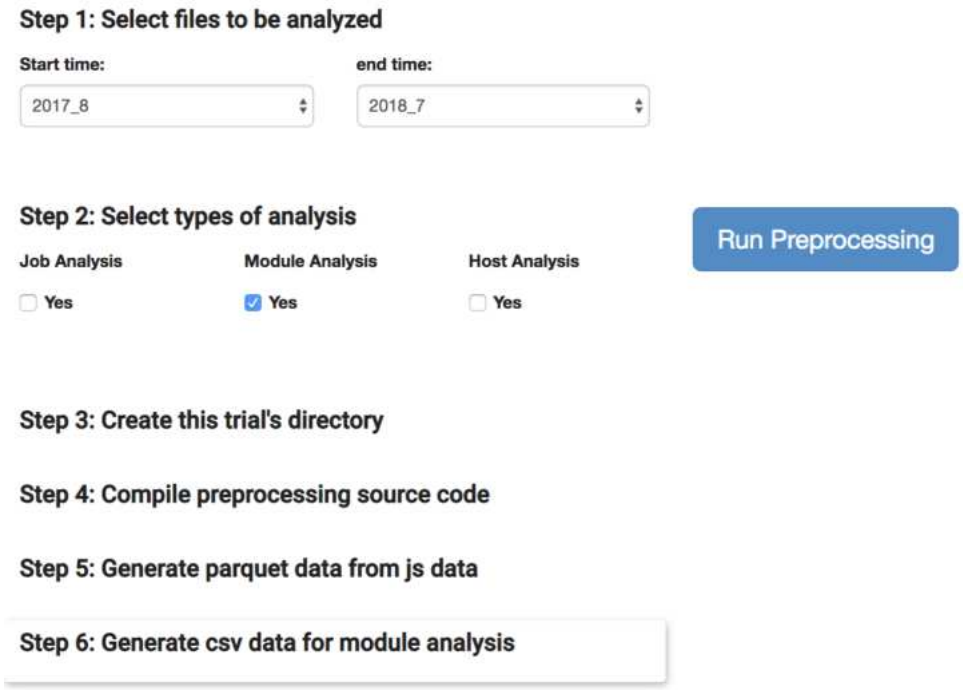
A interface de um *notebook* é composta por células que executam códigos na linguagem de preferência do usuário. As células podem ser executadas interativamente e o resultado visualizado imediatamente. O contexto de execução do Zeppelin, acessível pela variável `z`, possui utilitários que estendem a funcionalidade do *notebook*. Por exemplo, o comando `z.show` exibe o resultado de uma célula em tabelas e gráficos gerados nativamente. Isto é particularmente útil para criação de paineis de análise de dados ou relatórios (CHENG et al., 2018).

As células podem ainda conter *Dynamic Forms* – formulários dinâmicos –, que são objetos interativos em HTML para entrada de parâmetros. O comando `z.textbox`, por exemplo, cria um campo de texto. Esta funcionalidade permite a construção de interfaces amigáveis, onde o usuário final pode personalizar a execução de uma célula sem alterar o código-fonte. Também é possível agendar a execução de *notebooks*. Desta forma, a aplicação executa as células do *notebook* no horário agendado, e atualiza dados e visualizações automaticamente. A Figura 16 mostra uma das interfaces criadas por Liu et al. (2018).

Os trabalhos estudados nesta seção demonstraram que o Zeppelin é uma ferramenta poderosa para análise de dados, mas também para o desenvolvimento de *jobs* em Spark. Na seção 4.4 serão mostradas imagens do ambiente interativo criado para apoiar o desenvolvimento deste trabalho.

[4] Apache Zeppelin – http://zeppelin.apache.org/

Figura 16 – Exemplo de formulários dinâmicos em um *notebook* do Zeppelin



Fonte: Liu et al. (2018)

3 TRABALHOS RELACIONADOS

Programas como o BLAST utilizam heurísticas para otimizar a busca de sequências em bases de dados, sem garantir resultados ótimos. Devido ao aumento expressivo da quantidade de genomas sequenciados nos últimos anos, tornou-se necessário otimizar os algoritmos de busca para processar o crescente volume de informação. Neste capítulo, serão apresentados trabalhos que recorreram a técnicas de computação paralela e distribuída para otimização de buscas de sequências com BLAST.

3.1 MPIBLAST

A utilização de bases de dados genéticas grandes tem como consequência uma alta taxa de operações de leitura e escrita em disco durante a execução do BLAST. Darling, Carey e Feng (2003) criaram o mpiBLAST, que distribui a execução do algoritmo em *clusters* de computadores divindo a base de dados entre os nós em tamanhos aproximadamente iguais. O *cluster* deve ser previamente configurado com uma instalação do BLAST e deve-se mapear um local de rede para compartilhamento de arquivos.

O algoritmo foi implementado utilizando MPI para troca de mensagens. A execução é coordenada por um nó mestre que lê um arquivo de consulta do disco e a distribui integralmente para todos os nós trabalhadores do *cluster*. Cabe ao mestre controlar qual partição da base de dados será copiada para um trabalhador, e cada nó trabalhador é responsável por reportar o estado de execução local ao mestre. Uma vez designada a partição da base, o trabalhador deve copiá-la da rede para o seu dispositivo de armazenamento. A busca é realizada executando o BLAST instalado no nó contra a partição local da base. Os resultados de cada trabalhador são enviados de volta ao nó mestre, que os consolida e ordena por *score*. O mpiBLAST suporta todos os tipos de saída do BLAST, incluindo XML, HTML e texto delimitado.

Os testes foram realizados em um *cluster* com 240 nós equipados com processadores de 667 MHz e 640 MB de memória RAM, onde cada um recebeu um fragmento da base de dados com tamanho total de 5,1 GB. Foi feita uma consulta inicial, cujo tempo de execução não foi mensurado, para que os trabalhadores executassem o BLAST e carregassem sua respectiva partição da base de dados em memória. Os dados da base ficam no *cache* do sistema operacional, portanto não são recarregados do disco em execuções posteriores. Em seguida, uma consulta com sequências de genes de uma bactéria foi executada. O tempo de execução em um único trabalhador foi de aproximadamente 1.344 minutos, ou 22,4 horas. A execução levou 8 minutos com 128 trabalhadores, representando um *speedup* da ordem de 168.

O *speedup* superlinear, isto é, de valor superior à quantidade de nós adicionados é

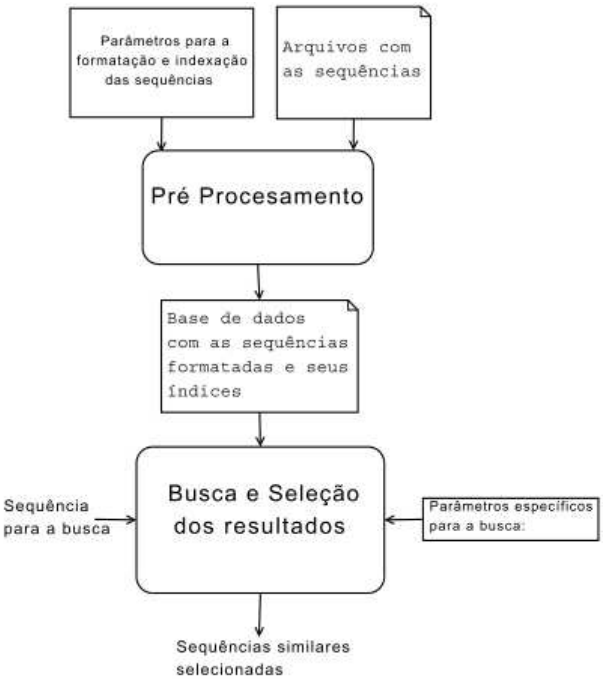
explicado pelo tamanho da base. Quando o BLAST executa uma busca contra uma base cujo tamanho é superior à memória, caso da execução em um único nó, não é possível guardar os dados em *cache*. Consequentemente, operações de leitura do disco rígido são realizadas conforme as sequências da base são alinhadas à consulta, impactando severamente o tempo de execução. Ao se particionar a base entre os nós, os fragmentos locais menores que a memória são carregados inteiramente e mantidos em *cache*.

Como limitações, Darling, Carey e Feng (2003) afirmam que o mpiBLAST não possui mecanismos de tolerância a falha. Ou seja, em caso de falha de processamento ou perda de algum nó, o sistema não é capaz de recuperar a execução. Outra limitação apontada é a ausência de paralelização da consulta e, por isso, a entrada deve ser enviada a todos os nós trabalhadores.

3.2 GENOOGLE

O trabalho de Albrecht (2009) consistiu em uma avaliação abrangente das estruturas de dados e técnicas de otimização empregadas nas ferramentas de busca de sequências disponíveis à época. A pesquisa deu origem ao Genoogle[1] uma ferramenta que utiliza técnicas de indexação e programação paralela com o objetivo de acelerar as buscas de sequências por similaridade. A Figura 17 mostra a visão geral do algoritmo.

Figura 17 – Visão geral do Genoogle

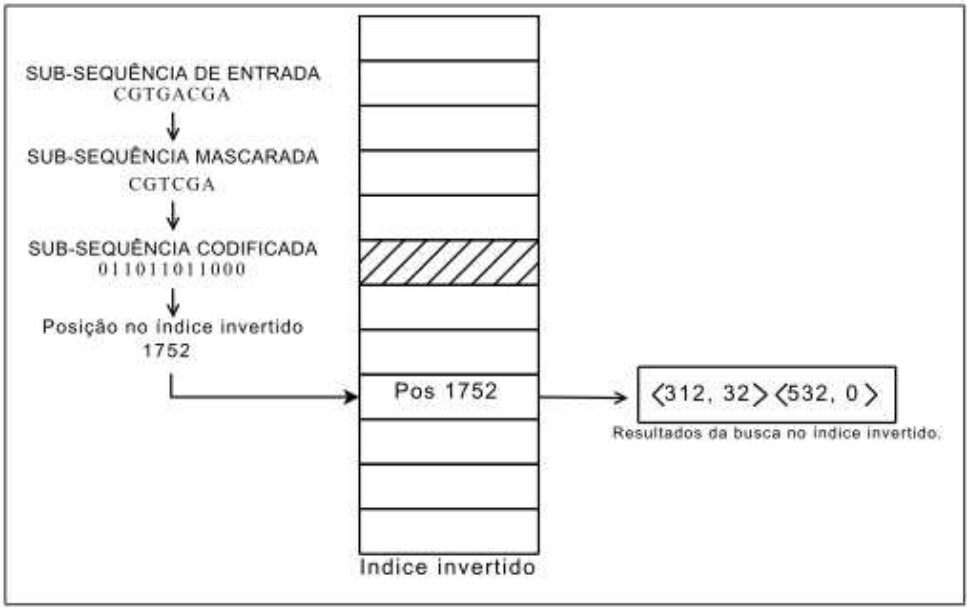


Fonte: Albrecht (2009)

[1] felipealbrecht/Genoogle – https://github.com/felipealbrecht/Genoogle

Como no BLAST, as sequências que compõe a base de dados precisam ser formatadas previamente. As sequências são quebradas em palavras não sobrepostas de tamanho k definido pelo usuário. Por exemplo, para a sequência ATCGTCGA e k igual a 4, obtém-se as subsequências ATCG e TCGA. Em seguida, é construído o índice invertido utilizado durante as buscas. Esta estrutura permite obter a localização de uma subsequência contida nas sequências da base de dados em tempo constante. Em contrapartida, um índice com volume considerável de entradas pode não caber na memória disponível. Para melhorar a sensibilidade das buscas e reduzir o total de entradas no índice, o algoritmo utiliza máscaras de *bits* de tamanho k, também definidas pelo usuário, que indicam quais posições das subsequências devem ser ignoradas pelo codificador. Por exemplo, ao aplicar-se a máscara 1101 na subsequência ATCG, o resultado é ATG. Ou seja, o valor 0 indica que a base correspondente àquela posição deve ser desprezada, enquanto 1 denota sua manutenção. O pré-processamento das sequências encontra-se exemplificado na Figura 18.

Figura 18 – Codificação e indexação das sequências de entrada



Fonte: Albrecht (2009)

O processo de busca tem como entrada um arquivo FASTA contendo uma ou mais sequências de consulta. Estas sequências passam pelo processo de codificação descrito anteriormente, com a diferença que as palavras agora são sobrepostas para melhorar a sensibilidade da busca. Isto é, considerando o exemplo anterior ATCGTCGA, obtém-se as subsequências ATCG, TCGT, CGTC, GTCG e TCGA, nas quais também é aplicada a máscara e a codificação para um valor numérico. O algoritmo utiliza estes fragmentos para consultar o índice invertido a fim de identificar regiões de alta similaridade, ou sementes.

A Figura 19 mostra o processo de semeadura e extensão do Genoogle, onde HSPs são estendidos em ambas as direções observando-se o *e-value*. Caso haja HSPs sobrepostos,

Figura 19 – Busca e extensão de HSPs

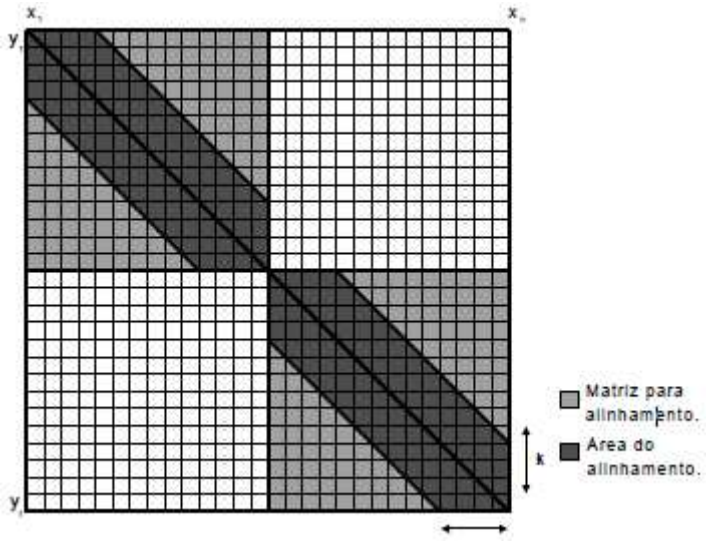
Fonte: Albrecht (2009)

ambos são agrupados em um único HSP. Após a extensão, os HSPs mais longos são selecionados e alinhados por uma versão otimizada do algoritmo de Smith-Waterman (SMITH; WATERMAN; FITCH, 1981). As sequências são divididas em trechos alinhados separadamente. Esta otimização, segundo Albrecht (2009), reduz o tamanho da matriz de alinhamento e utiliza 75% menos memória RAM. Como o HSP garante alta similaridade, limita-se a quantidade de espaços inseridos nas subsequências alinhadas. Com isso, apenas as células mais próximas à diagonal da matriz são preenchidas, reduzindo também o tempo de alinhamento. A Figura 20 mostra o alinhamento de um HSP onde as sequências foram divididas em dois trechos, alinhados por matrizes menores.

O Genoogle foi avaliado em uma máquina com dois processadores *Xeon*, cada um com 4 núcleos, e 16 GB de memória RAM disponíveis. Foi utilizado uma base de dados com tamanho total igual a 4,25 GB construído a partir do projeto genoma humano e de sequências de referência de outras espécies. Também foram criados 11 conjuntos de sequências de entrada, cada um contendo 11 sequências de tamanho variável entre 80 e 1.000.000 de pares de base. O objetivo dos experimentos foi mensurar o ganho de desempenho do Genoogle com relação ao BLAST com e sem paralelismo. Os resultados mostraram que o speedup do Genoogle sem paralelismo variou de 16 a 42,61 para entradas com 1.000.000 bp e 500 bp respectivamente. O ganho deu-se, principalmente, pela utilização de índices invertidos no processo de busca. Com paralelismo, os ganhos obtidos foram menores, mas ainda significativos. O *speedup* variou de 7 vezes com entradas de 80 bp, a 29,50 com 100.000 bp.

Além do desempenho, foi avaliada a qualidade dos resultados das buscas efetuadas pelo Genoogle. Para isto, verificou-se a capacidade do algoritmo de identificar os mesmos

Figura 20 – Matriz de alinhamento do algoritmo Smith-Waterman otimizado



Fonte: Albrecht (2009)

alinhamentos encontrados pelo BLAST. 95% dos alinhamentos com *e-value* menor ou igual a 10^{-35} também foram encontrados pelo Genoogle. Para *e-value* de até 10^{-15}, 70% dos alinhamentos foram encontrados. Foi demonstrado que a porcentagem varia de acordo com os parâmetros de indexação e busca utilizados.

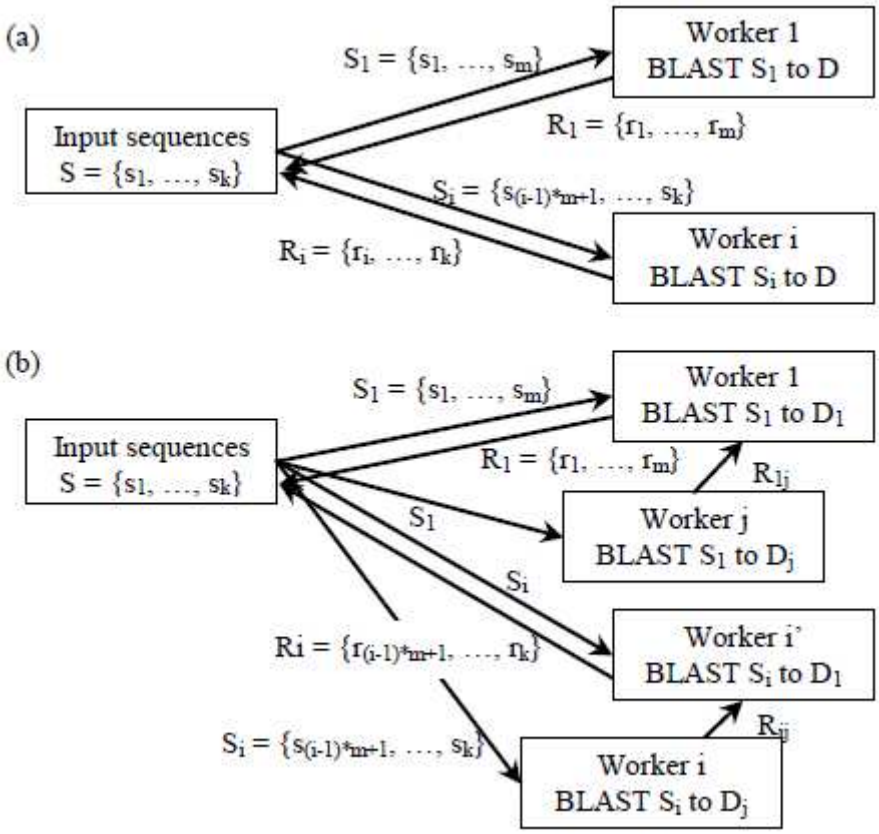
A pesquisa concluiu que a utilização de índice invertido associado às técnicas de paralelização empregadas melhora o desempenho das buscas consideravelmente, sem prejuízo à significância dos alinhamentos encontrados. Até o momento, nenhum outro trabalho deu continuidade ao Genoogle.

3.3 CLOUDBLAST

O CloudBLAST de Matsunaga, Tsugawa e Fortes (2008) utilizou MapReduce para paralelizar a execução do BLAST sem modificá-lo. O trabalho sugeriu duas abordagens de paralelização ilustradas na Figura 21. A primeira consiste na segmentação da entrada S e a execução de múltiplas instâncias do BLAST para cada segmento s_k contra uma cópia local da base de dados D em cada nó. Na segunda, divide-se S e também D entre os nós do *cluster*. Desta forma, a base de dados pode crescer, e cada fragmento D_k caber no disco e na memória local sob o custo dos nós se comunicarem entre si para consolidar os resultados.

A paralelização do BLAST foi realizada com a extensão Hadoop *Streaming*, onde cada *Mapper* executou a ferramenta tendo como entrada a partição das sequências correspondente. Não foi implementado um *Reducer* para este caso. Os autores afirmam que, apesar de terem executado a solução apenas com o BLAST, pode-se utilizá-la com outras ferramentas, como o HMMER, MegaBLAST e outras variantes do BLAST.

Figura 21 – Visão geral do CloudBLAST



Fonte: Matsunaga, Tsugawa e Fortes (2008)

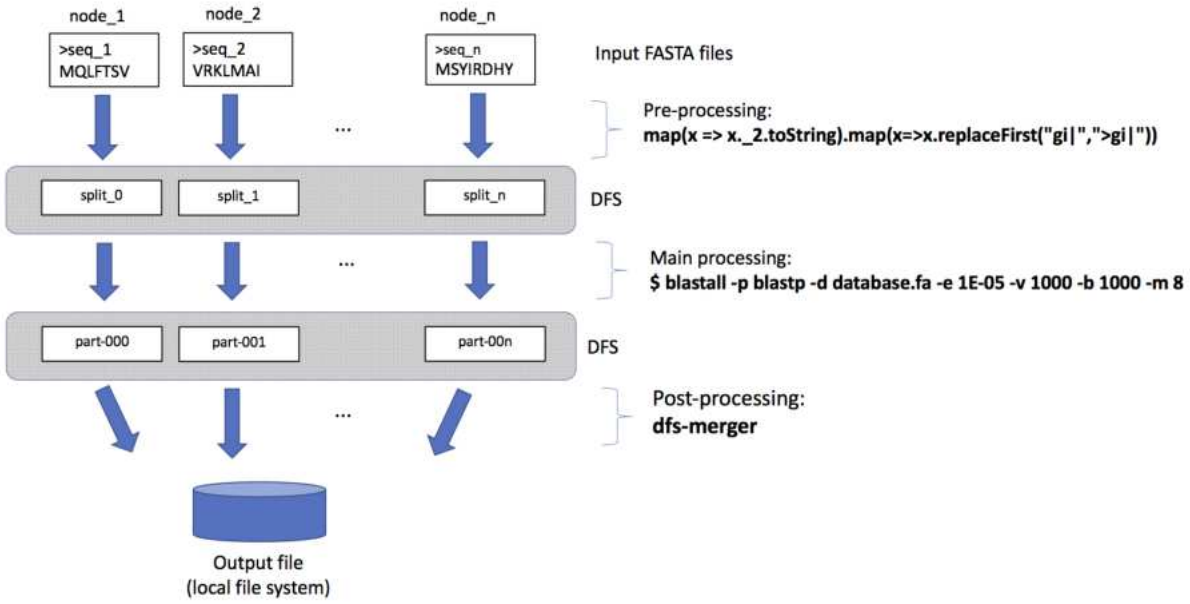
Matsunaga, Tsugawa e Fortes (2008) avaliaram a escalabilidade do CloudBLAST em *clusters* de computadores instalados na Universidade de Chicago e na Universidade da Flórida, conectados via internet. Os autores executaram o mpiBLAST sob as mesmas condições. O ganho de desempenho do CloudBLAST em relação ao BLAST foi melhor que o apresentado pelo mpiBLAST em todos os cenários de teste. O BLAST levou 43.76 horas para executar consultas longas, com 960 sequências com comprimento médio de 1116 bp. Neste cenário, utilizando 64 processadores de máquinas localizadas nas duas universidades, o CloudBLAST alcançou *speedup* da ordem de 57 vezes e o mpiBLAST entregou 52. Nas consultas curtas, de comprimento médio de 443 bp, a execução sequencial do BLAST durou 17,1 horas. Este tempo foi reduzido para 19,5 minutos com o CloudBLAST, e 25 minutos com o mpiBLAST.

Além do desempenho, o CoudBLAST apresentou como vantagem o desenvolvimento mais simples, pois o MapReduce abstrai toda a parte de coordenação e comunicação entre nós, enquanto no mpiBLAST esta lógica precisa ser implementada. Outra vantagem observada é que o CloudBLAST pode executar novas versões do BLAST assim que lançadas, de maneira transparente.

3.4 SPARKBLAST

O SparkBLAST[2] (CASTRO, 2017) foi desenvolvido com o objetivo de oferecer uma ferramenta escalável e de fácil operação para paralelizar a execução do BLAST. Na pesquisa, identificou-se a oportunidade de utilizar a transformação *pipe* do Spark para distribuir a entrada e paralelizar as consultas em um esquema similar ao CloudBLAST. Ao contrário deste, o SparkBLAST não explorou o paralelismo através da divisão da base de dados.

Figura 22 – Visão geral do SparkBLAST



Fonte: Castro (2017)

O algoritmo é dividido em três etapas, conforme mostra a Figura 22. Primeiramente, é realizado o pré-processamento da entrada onde o arquivo FASTA da consulta é lido do HDFS, tratado e dividido entre os nós. Na etapa principal de processamento, o BLAST é invocado pelo *pipe* e as buscas executadas em paralelo contra as cópias locais da base de dados. Os resultados são armazenados no HDFS, um arquivo por tarefa. No pós-processamento, as partições dos resultados são consolidadas em um arquivo único no caminho de saída especificado pelo usuário.

A solução foi validada por meio de dois experimentos. Cada teste foi executado seis vezes por configuração, e a média dos tempos de execução foi tomada para calcular o *speedup* e eficiência tanto do CloudBLAST quanto do SparkBLAST em relação às respectivas execuções em um único nó. O primeiro foi executado na *Google Cloud Platform* (GCP) em *clusters* de 1 a 64 nós e 2 núcleos por nó. Foi utilizada uma base de dados de 36,7 MB com 91.108 sequências obtidas do proteoma de 11 bactérias. A consulta utilizou estas mesmas sequências de modo a encontrar os melhores *hits* recíprocos para inferência

[2] sparkblastproject/v2 – https://github.com/sparkblastproject/v2

de genes ortólogos. O CloudBLAST apresentou *speedup* igual 1,61 e eficiência de 80% na execução em 2 nós, enquanto o SparkBLAST obteve *speedup* de 1,99 e eficiência próxima a 100%. Com 64 nós, o CloudBLAST executou 37 vezes mais rápido com eficiência de 58%. Neste cenário, o SparkBLAST obteve speedup igual a 41,81 e eficiência de 65%.

O segundo experimento foi realizado na nuvem da Azure em *clusters* de 1 a 63 nós, com 4 núcleos por nó. Foram utilizadas duas bases de dados com 805 MB e 11 GB respectivamente. A consulta continha 86.968 sequências e tamanho igual a 35 MB. O *speedup* do SparkBLAST foi superior ao do CloudBLAST em todos os cenários de teste com a primeira base de dados. Com o CloudBLAST, não foi possível executar consultas na base de dados de 11 GB por falta de memória. O *speedup* obtido com o SparkBLAST, por outro lado, variou de 3,73 com 1 nó a 177,64 no *cluster* de 63 nós[3]. O autor atribui este resultado ao uso de memória mais eficiente por parte do Spark.

Os resultados obtidos pelo SparkBLAST demonstraram a escalabilidade da solução. Os ganhos de desempenho superiores ao CloudBLAST e a possibilidade de utilizar bases de dados maiores do que a memória disponível mostraram ainda que o Spark é uma solução mais interessante que o MapReduce para a paralelização do BLAST.

3.5 TRABALHOS RECENTES

Outros trabalhos exploraram técnicas diferentes para distribuir a execução do BLAST. O SparkyBlast (CORES; GUIRADO; LERIDA, 2021) reimplementou o algoritmo de busca e alinhamento utilizando Pyspark. Para otimizar o acesso aos dados, o programa divide as sequências da base de dados e índices invertidos em blocos e os armazena no Cassandra[4], um sistema gerenciador de bancos de dados distribuído. Um aspecto interessante desta solução é que o armazenamento em blocos, em essência, permite que os nós do *cluster* armazenem fragmentos da base de dados, ao invés de uma réplica completa. Consequentemente, permite a construção de grandes bases que não caberiam na memória de uma única máquina. Por outro lado, embora o *speedup* em relação ao BLAST tenha validado a solução, o ganho obtido não mostrou-se proporcional à quantidade de CPUs disponíveis por experimento. Os autores entendem que a etapa de extensão teve o desempenho prejudicado pois o SparkyBlast foi implementado em Python, cujo desempenho é inferior à linguagem C utilizada pelo BLAST.

Mais recentemente, Camacho et al. (2023) propuseram o ElasticBLAST[5] com o propósito de reduzir a complexidade para implantar e gerenciar a infraestrutura em nuvem utilizada para paralelizar o BLAST. No entanto, o ElasticBLAST não utiliza Hadoop ou Spark para a distribuição de consultas. A solução define um fluxo de trabalho baseado

[3] Nos experimentos da Azure, o autor calculou o *speedup* com base no número total de processadores do *cluster*, e não o de nós.

[4] Apache Cassandra – https://cassandra.apache.org/

[5] ElasticBLAST – https://blast.ncbi.nlm.nih.gov/doc/elastic-blast/

no modelo Plataforma como Serviço (PaaS, do inglês *Platform as a Service*), que utiliza serviços gerenciados em nuvens públicas para facilitar a gestão de recursos. São suportados os provedores de serviço AWS e GCP. A base de dados é copiada em cada nó, portanto o processo está sujeito às mesmas limitações de memória e armazenamento do SparkBLAST. Como diferencial, o ElasticBLAST seleciona as máquinas e recursos mais apropriados para as buscas considerando o tamanho da base e das consultas. Inclusive, pode optar por não distribuir a execução caso a base de dados utilizada não seja suficientemente grande. Com isso, a solução é capaz de otimizar não apenas o tempo de consulta, mas também os custos de infraestrutura.

4 GENUVEM

O BLAST (ALTSCHUL et al., 1990) consolidou-se como a principal ferramenta de busca de sequências em bases de dados genômicos. Com o crescimento acelerado de bases como o GenBank (NCBI, 2023a), fez-se necessária a paralelização e distribuição do algoritmo. O mpiBLAST (DARLING; CAREY; FENG, 2003) implementou um mecanismo para execução do BLAST em *clusters* de computadores com MPI. Os projetos CloudBLAST (MATSUNAGA; TSUGAWA; FORTES, 2008) e SparkBLAST (CASTRO, 2017) demonstraram a viabilidade da utilização de MapReduce (DEAN; GHEMAWAT, 2004) e Spark (ZAHARIA et al., 2010) para distribuir a execução do BLAST sem alterações em seu código-fonte.

O Genoogle (ALBRECHT, 2009) foi bem sucedido em utilizar técnicas de programação paralela e estruturas de dados otimizadas para obter alinhamentos de significância comparáveis ao BLAST em tempo significativamente menor. O algoritmo, porém, não suporta execução em *clusters* e não houve continuidade em seu desenvolvimento após a publicação do trabalho. A partir da pesquisa realizada, identificou-se a oportunidade de otimizar a execução do Genoogle utilizando plataformas de *Big Data*. Nas seções a seguir, será apresentada a ferramenta criada para executar o Genoogle em *clusters* de computadores, o Genuvem.

4.1 HISTÓRICO E ESCOPO DE TRABALHO

O escopo inicial consistiu em reimplementar o Genoogle utilizando a API MapReduce do Hadoop. Já no início da pesquisa, verificou-se que as etapas do algoritmo precisariam ser implementadas como uma sequência de *jobs* MapReduce encadeados. Também, a etapa de extensão e junção de HSPs deve ser repetida enquanto houver HSPs sobrepostos. Portanto, para cada iteração, seria necessário adicionar um novo *job* à sequência de execução. Considerando as limitações elencadas na subseção 2.2.2.4, entendeu-se que o agendamento de *jobs* e as sucessivas operações de leitura e escrita no HDFS impactariam significativamente o tempo de execução. Decidiu-se, então, que este modelo não seria adequado para o problema.

Na segunda tentativa, cogitou-se reimplementar o Genoogle utilizando a API de RDD em Scala do Spark por sua capacidade de processamento em memória, mais adequada para algoritmos iterativos. Foi desenvolvido um protótipo para a codificação e indexação de sequências reutilizando as técnicas de otimização do Genoogle, como máscaras e filtros de sequências de baixa complexidade. O fluxo implementado naquele momento seguiu o modelo da Figura 18 da seção 3.2. Porém, a complexidade da reimplementação da busca, extensão e alinhamento de HSPs mostrou-se alta para o propósito desta monografia e

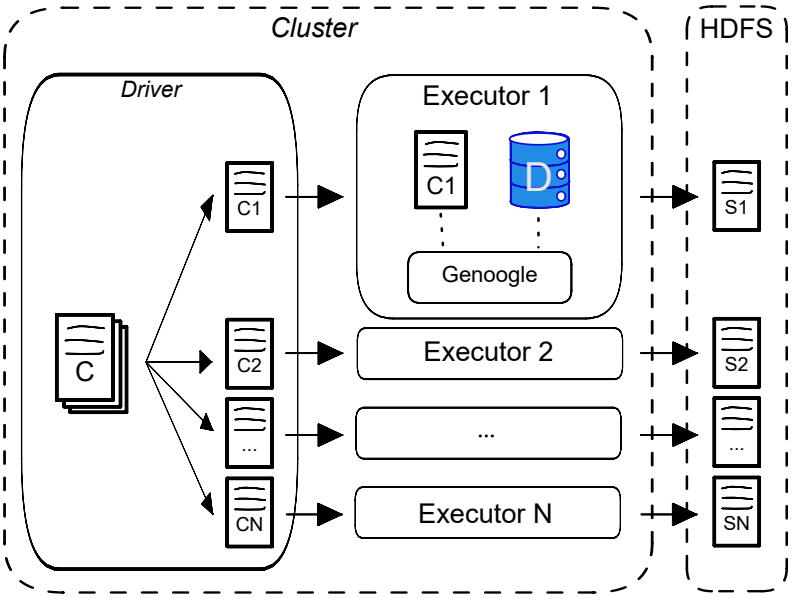
decidiu-se por não reimplementar estas etapas.

Finalmente, notou-se que o uso da transformação *pipe* do Spark, conforme utilizada pelo SparkBLAST, viabilizaria a execução distribuída do Genoogle sem a necessidade de reimplementá-lo, simplificando o desenvolvimento. Assim, definiu-se como escopo deste trabalho a adaptação do modelo estudado para distribuir a execução do modo de busca em lote do Genoogle.

4.2 MODELO CONCEITUAL

Para a distribuição do Genoogle foi idealizado um modelo de execução similar ao SparkBLAST, baseado na paralelização da consulta. O processo ilustrado na Figura 23 inicia com a divisão das sequências de entrada. Seja C o conjunto de um ou mais arquivos em formato FASTA. O *driver* deverá particioná-lo em lotes de consulta c_k tal que $1 \leq k \leq N$, onde N é o total de executores disponíveis. Caberá ao *driver* distribuir as tarefas aos executores.

Figura 23 – Visão geral do Genuvem



O k-ésimo executor receberá um único lote de consultas c_k e, em seguida, executará o Genoogle contra a cópia local da base de dados D. Ao fim da busca, o arquivo de saída s_k gerado pela ferramenta será copiado para o HDFS e o processo é concluído.

O produto deste fluxo é um diretório no HDFS com os resultados em XML gerados pelo Genoogle sobre cada partição da consulta. A divisão da base de dados entre os nós não foi considerada parte do escopo deste trabalho, e será sugerida como trabalho futuro adiante.

4.3 IMPLEMENTAÇÃO

O Genuvem foi implementado como um conjunto de *scripts* Spark que transformam os arquivos FASTA de entrada em um RDD de sequências e distribuem, com o apoio de um *shell script* auxiliar, a execução de buscas com Genoogle. Como pré-requisito, os nós do *cluster* devem possuir uma instalação do Genoogle, uma cópia integral da base de dados e uma cópia do *shell script* auxiliar `run_genoogle.sh`.

Os *scripts* detalhados nas seções a seguir foram desenvolvidos em Scala 2.12 e APIs compatíveis com Spark 3.3.1 e Hadoop 3.3.4. Não foi realizada qualquer alteração no código-fonte do Genoogle para este projeto. Detalhes adicionais sobre configuração e implantação do Genuvem estão disponíveis no Apêndice A.

4.3.1 Paralelização das sequências de entrada

Arquivos FASTA são arquivos de texto plano. A API do Spark fornece o comando `textFile` que abstrai a leitura de um ou mais arquivos em simultâneo. Por padrão, cada linha dos arquivos de texto de entrada são considerados registros independentes. Porém, o formato FASTA define que o caractere `>` marca o cabeçalho de uma sequência, e as linhas seguintes correspondem à sequência em si. A sequência termina quando um novo marcador é introduzido, ou o arquivo termina. Portanto, para que o Spark faça a leitura correta, é necessário alterar o delimitador de registros para o caractere `>`. Desta forma, cada sequência contida em um ou mais arquivos FASTA é lida em sua completude, e o número de partições do RDD resultante corresponde à quantidade total de sequências lidas. O Código 4.1 mostra como é feita a leitura e tratamento dos arquivos de entrada.

Código 4.1 – Leitura e tratamento dos arquivos de consulta

```
1 val nodeCount = sc.getConf.getInt("spark.executor.instances", 1)
2 val conf = sc.hadoopConfiguration
3 conf.set("textinputformat.record.delimiter", ">")
4
5 val rdd = sc.textFile(queryPath)
6   .map(x => x.trim())
7   .filter(x => x.nonEmpty)
8   .map(x => '>' + x)
9   .repartition(nodeCount)
```

Após a leitura, é feito o tratamento das cadeias de caracteres que representam os cabeçalhos e respectivas sequências. Primeiro, são removidos espaços em branco no início ou no final das cadeias de caracteres processadas. Partições vazias, isto é, aquelas cujo comprimento da cadeia de caracteres seja 0, são removidas. Em seguida, o caractere `>`, que não é considerado parte do registro pelo Spark, é reintroduzido no início de cada partição. Desta forma, cada partição contém um cabeçalho FASTA e sua sequência correspondente.

Logo, o número de partições resultante da leitura e tratamento dos arquivos de entrada é igual ao número de sequências contidas nestes arquivos.

Como o Genoogle carrega os índices em memória no início de cada execução, não é desejável que múltiplos processos executem em um nó. Para mitigar este problema, a transformação `repartition` define o total de partições igual ao número de executores disponíveis. A análise que levou a esta decisão será detalhada na seção 5.3.

4.3.2 Execução do Genoogle em ambiente distribuído

O Código 4.2 mostra a execução da transformação *pipe* sobre o RDD de consultas de entrada. São passados como argumento para o *pipe* o caminho local do *script* auxiliar `run_genoogle.sh`, o nome da base de dados de referência e o diretório do HDFS onde serão armazenados os resultados. A ação `collect` inicia, então, a execução do processo. Neste ponto, o *driver* distribui as tarefas para os executores e aguarda até que todos completem com sucesso. Para simplificar o entendimento, as definições de variáveis de ambiente e outras configurações foram omitidas nos *scripts* desta seção.

Código 4.2 – Execução distribuída do Genoogle

```
1  val runArgs = Seq("/app/genoogle/run_genoogle.sh", databank, hdfsPath)
2
3  val outputFiles = rdd
4      .pipe(runArgs)
5      .collect
```

Código 4.3 – Trecho principal do *shell script* auxiliar `run_genoogle.sh`

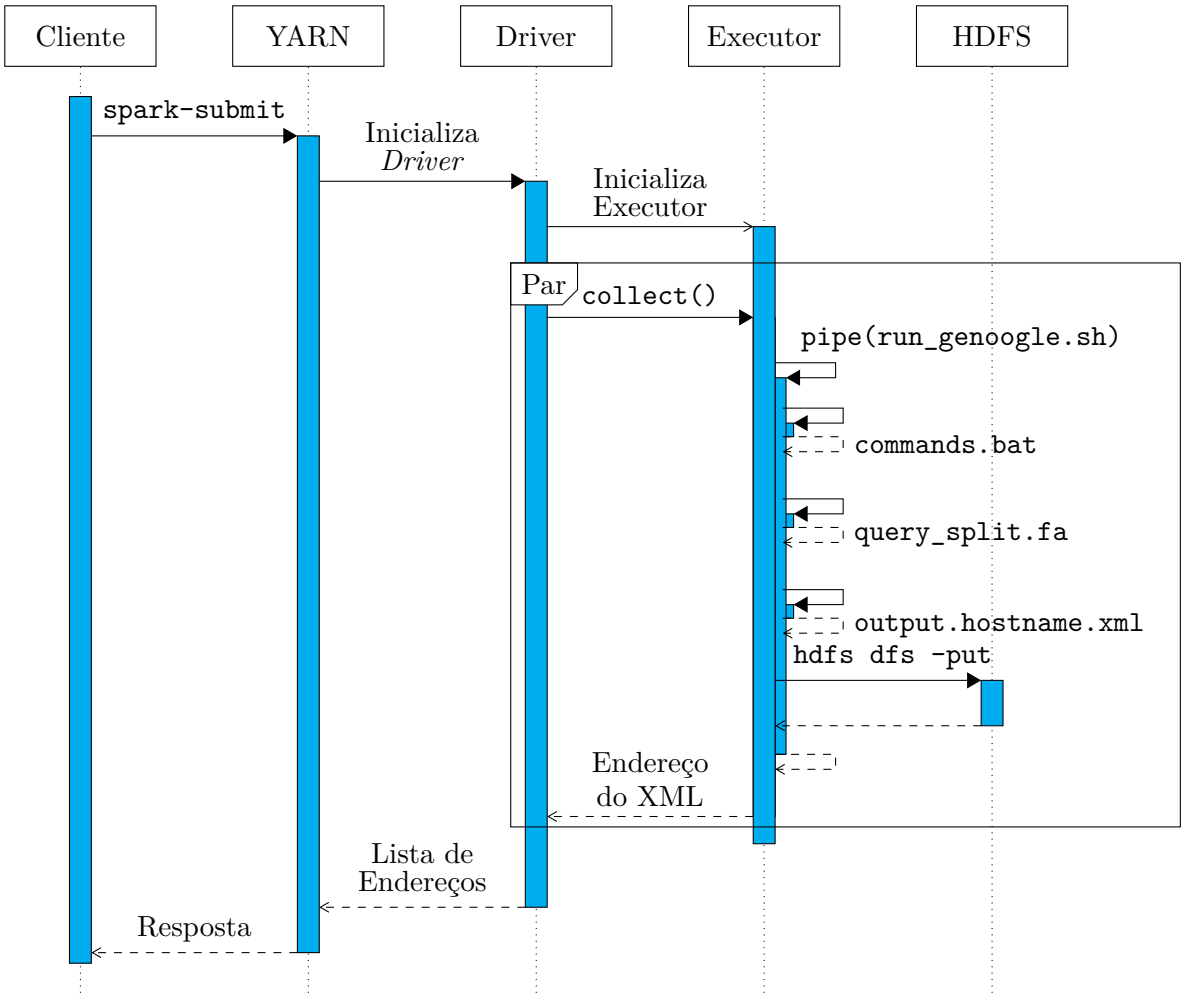
```
1  CMD="search $DATABANK ./query_split.fasta ./output.$HOSTNAME\nexit"
2  echo -e "$CMD" >commands.bat
3
4  true >query_split.fasta
5  while read -r LINE; do
6    echo "${LINE}" >>query_split.fasta
7  done
8
9  java -server -Xms4086m -Xmx12288m \
10   -classpath "${GENOOGLE_HOME}/genoogle.jar:${GENOOGLE_HOME}/lib/*" \
11   bio.pih.genoogle.Genoogle -b commands.bat &>>genooogle.log
12
13 hdfs dfs -put ./output.$HOSTNAME.xml "$TASK_OUTPUT"
14 echo "$TASK_OUTPUT"
```

Devido às limitações da interface do Genoogle, o Genuvem não o executa diretamente. O BLAST recebe os parâmetros de consulta do SparkBLAST pelo *pipe* e escreve os *hits* na saída padrão em formato de tabela, com um resultado por linha. Então, o Spark

consegue tratar e consolidar estes resultados em um RDD. Com o Genoogle, o comando `search` obriga que a consulta seja lida de um arquivo FASTA armazenado em disco. Já a saída, disponível apenas em XML, não é compatível com o formato esperado de um resultado por linha do *pipe*. Considerando o objetivo de distribuir o sistema sem alterar o seu código fonte, decidiu-se pela criação do *script* auxiliar listado no Código 4.3.

O *script* `run_genoogle.sh` é responsável por preparar os arquivos locais necessários para a execução do Genoogle em cada executor. Primeiro é criado o arquivo de comandos em lote `commands.bat`. O comando `search` recebe três parâmetros obrigatórios: o identificador da base de dados, o caminho do arquivo local de consulta e o caminho do arquivo de saída. O comando `exit` encerra o Genoogle após a conclusão da busca.

Figura 24 – Sequência de interações entre componentes do *cluster* Genuvem



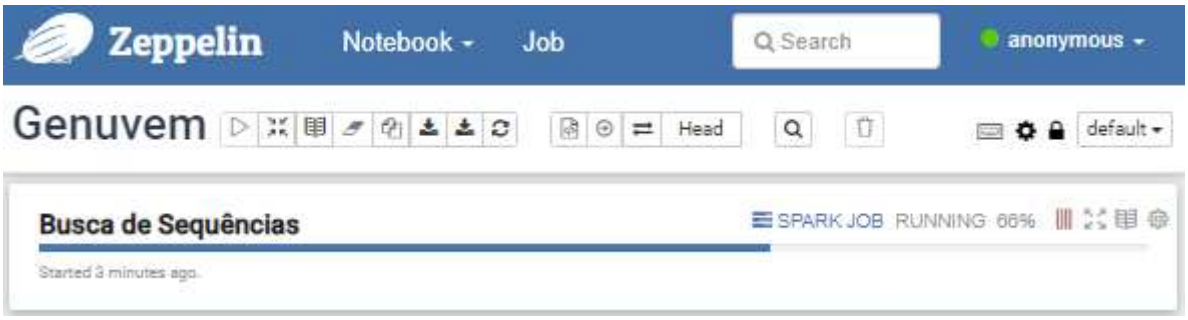
O arquivo de consulta local é construído dinamicamente a partir do fragmento da consulta processada. O trecho entre as linhas 4 e 7 do Código 4.3 lê da entrada-padrão o fragmento de consulta e escreve seu conteúdo, linha a linha, neste arquivo. Então, o Genoogle executa os comandos em lote. A saída de texto é redirecionada para o arquivo `genoogle.log` para que os executores não transmitam cada linha de volta para o *driver* como registros do RDD resultante. Ao término da execução, o arquivo XML local com

o resultado da busca é copiado para o diretório HDFS especificado, o endereço final no HDFS é escrito na saída-padrão, o *driver* coleta a lista de arquivos XML gerados e encerra a execução. O diagrama de sequência da Figura 24 ilustra o processo fim-a-fim, com foco nas principais etapas e artefatos descritos nesta seção.

4.4 AMBIENTE INTERATIVO

Os códigos apresentados neste capítulo foram desenvolvidos em um *notebook* Zeppelin. Os recursos de interação e visualização de dados foram úteis para construir uma interface de usuário para o Genuvem. A integração com o Spark permite o acompanhamento em tempo real da execução de um *job*, como mostra a Figura 25.

Figura 25 – Zeppelin: barra de progresso indicando o andamento de uma consulta



A Figura 26 mostra a interface após a conclusão. Os campos Banco de Dados e Consulta permitem que o usuário execute nova busca facilmente. A tabela de resultados, construída pelo Código 4.4, traz informações sobre as sequências alinhadas e métricas de interesse, como pontuação e *e-value*.

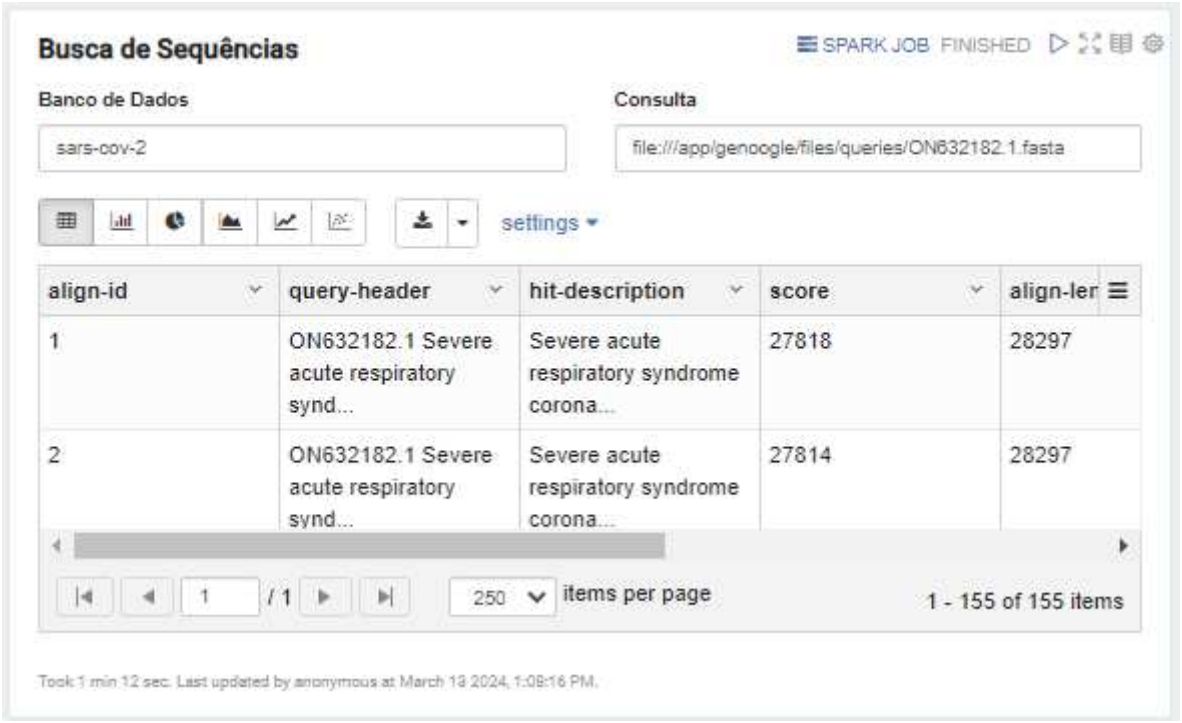
Código 4.4 – Leitura e consolidação dos resultados de busca

```
val xmls = spark.read
    .option("rootTag", "Genoogle")
    .option("rowTag", "iteration")
    .xml(s"/results/*.xml")

val results = xmls
    .withColumn("hit", explode($"results.hits.hit"))
    .withColumn("hsp", explode($"hit.hsps.hsp"))
    .select("*", "hit.*", "hsp.*")
    .drop("results", "hit", "hsps", "hsp", "_number")
    .withColumn("align-id", row_number().over(
        Window.orderBy(asc("_query"), desc("_normalized-score"))
    ))

z.show(results)
```

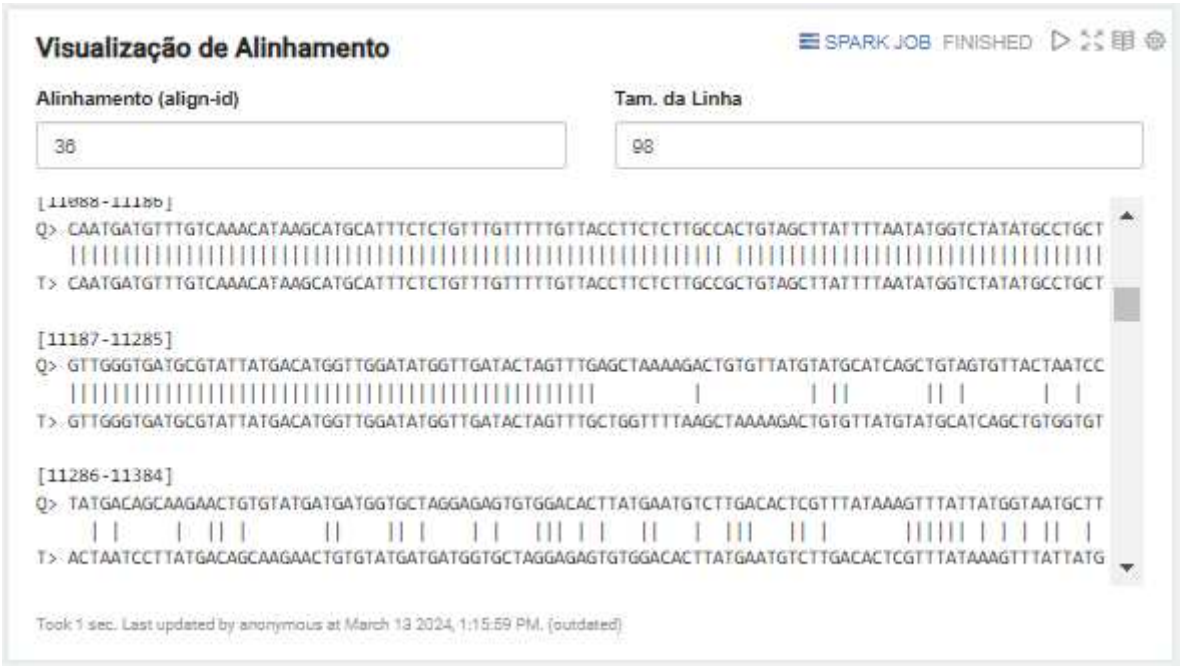
Figura 26 – Zeppelin: formulários dinâmicos e tabela interativa de resultados



Os resultados são obtidos dos XMLs de saída do Genoogle armazenados no HDFS e consolidados em um *DataFrame*. Na sequência, estruturas aninhadas do XML são transformadas em novas colunas pela transformação `explode` e é adicionado o identificador `align-id` para cada alinhamento. A tabela, desenhada em tela pelo comando `z.show`, oferece recursos de paginação, ordenação, filtros, e exportação de dados.

A Figura 27 mostra a interface de visualização de alinhamentos. O Código 4.5 filtra a tabela de resultados conforme o `align-id` definido pelo usuário. Os campos `query` e `targt` contém as subsequências de consulta e o alvo correspondente. Ambas podem conter o caractere `-`, que indica um espaço introduzido. `align` é o alinhamento em si, composto por caracteres comuns às duas subsequências nas respectivas posições. Para simplificar a visualização, as posições alinhadas são indicadas com um traço vertical.

Figura 27 – Visualização de alinhamento entre consulta (Q) e alvo (T)



Código 4.5 – *Script* para visualização de alinhamentos

```
1  val row = results
2      .filter($"align-id" === alignId)
3      .select("query", "align", "targt")
4      .first
5
6  val query  = row.getAs[String]("query")
7  val align  = row.getAs[String]("align").replaceAll("[ATCG]", "|")
8  val target = row.getAs[String]("targt")
9
10 var start = 0
11 while (start <= query.length){
12     val end = Math.min(query.length(), start + limit)
13
14     println(s"[$start-$end]")
15     println("Q> " + query.substring(start, end))
16     println("   " + align.substring(start, end))
17     println("T> " + target.substring(start, end))
18     println
19
20     start = end + 1
21 }
```

5 RESULTADOS

Neste capítulo, são discutidos os resultados das medições do tempo de processamento de consultas executadas no Genuvem em *clusters* de diferentes tamanhos, comparando-as à execução do Genoogle em um nó isolado. O objetivo dos experimentos realizados foi avaliar o ganho de desempenho com a utilização de *clusters* e demonstrar a escalabilidade da solução.

5.1 MÉTRICAS DE AVALIAÇÃO DE DESEMPENHO

O *speedup* relativo (Equação 5.1) compara o tempo T_1, necessário para resolver um problema com uma unidade de processamento, e o tempo T_P, decorrido ao paralelizar o trabalho entre P unidades (MCCOOL; REINDERS; ROBISON, 2012).

$$\textit{speedup} = S_P = \frac{T_1}{T_P} \tag{5.1}$$

A eficiência computacional (Equação 5.2) é a razão entre o *speedup* e o número de unidades de processamento. Esta medida, geralmente expressa em porcentagem, quantifica o retorno de investimento em unidades de processamento.

$$\text{eficiência} = \frac{S_P}{P} = \frac{T_1}{PT_P} \tag{5.2}$$

Uma tarefa perfeitamente paralelizável resultaria em *speedup* linear onde $S_P = P$. Ou seja, o ganho de desempenho seria igual à quantidade de unidades de processamento utilizada e, consequentemente, eficiência igual a 100%. A paralelização de algoritmos implica em carga adicional de trabalho ou de trocas de contexto que podem impactar o tempo de execução. Por este motivo, não é comum alcançar *speedup* linear.

5.2 CONFIGURAÇÃO DE AMBIENTE

Os experimentos foram realizados na nuvem da *Google Cloud Platform* (GCP). Para o Genoogle foi utilizada uma máquina do tipo *e2-standard-2*, equipada com processador *Intel(R) Xeon(R) CPU* de 2.20GHz com 2 núcleos, 16 GB de memória RAM e 100 GB de espaço em disco. O Genuvem foi executado em *clusters* Hadoop com um nó mestre, 2 a 32 nós trabalhadores e o mesmo tipo de máquina. Optou-se por esta configuração para viabilizar consultas de longa duração com o menor custo possível.

Não houve motivação biológica para este estudo. A criação dos conjuntos de dados relatada nesta seção buscou a obtenção de alinhamentos longos, com duração significativa

para a análise de desempenho. Buscou-se também minimizar o requisito de armazenamento ou memória RAM para que fosse compatível com as máquinas de baixo custo utilizadas.

A base de dados foi construída a partir de genomas completos de SARS-CoV-2[1], sequenciados entre janeiro e dezembro de 2021 e disponibilizados no GenBank (NCBI, 2023b) em formato FASTA. Foram obtidas 252.430 sequências de referência com comprimento médio de 29.796 bp, e total de 7.521.461.371 bp. Os arquivos FASTA obtidos ocuparam um espaço total de 7,15 GB.

Quadro 3 – Configurações de codificação e indexação de bases de dados do Genoogle

Parâmetro	**Valor**
sub-sequence-length	11
mask	111010010100110111
number-of-sub-databanks	1
low-complexity-filter	5

O comando `run_console.sh -g`, do próprio Genoogle, foi utilizado para formatar a base de dados e construir o índice. Os parâmetros listados no Quadro 3 seguem o modelo de configuração que acompanha o Genoogle e não foram alterados para este estudo. As sequências formatadas e seus metadados resultaram em 2,58 GB, aproximadamente 36% do tamanho original. A base gerou 417.858.965 subsequências candidatas a indexação. O filtro de baixa complexidade reduziu o total de entradas no índice para 416.408.598 subsequências, com tamanho final igual a 3,28 GB. Os arquivos gerados totalizaram 5,87 GB, e o tempo decorrido foi de 42 minutos. Todas as máquinas receberam cópias destes arquivos e do arquivo XML de configuração da ferramenta.

Quadro 4 – Parâmetros de busca do Genoogle

Parâmetro	**Valor**
max-sub-sequence-distance	30
min-hsp-length	11
extend-dropoff	5
max-hits-results	50
max-threads-index-search	2
max-threads-extend-align	2
min-query-slice-length	1000
query-split-quantity	1

Para as buscas, foram gerados arquivos de consulta no formato Multi-FASTA, contendo de 4 a 1024 genomas completos de SARS-CoV-2 sequenciados a partir de 2022. Dos parâmetros de busca listados no Quadro 4, foram alterados apenas `max-hits-results`, `max-threads-index-search` e `max-threads-extend-align`. *max-hits-results* define o total

[1] *SARS-CoV-2 Resources* - NCBI – https://www.ncbi.nlm.nih.gov/sars-cov-2/

de alinhamentos retornados por sequência de entrada. Optou-se por reduzi-lo para 50 para diminuir o uso de memória RAM em consultas maiores. Os dois últimos definem a quantidade de duas *threads* para a busca nos índices, e para a etapa de extensão e alinhamento de HSPs. Decidiu-se por este valor em função da quantidade de CPUs disponíveis em cada nó.

5.3 ANÁLISE DE CARREGAMENTO E USO DE MEMÓRIA

O estudo dos registros de execução do Genoogle foi realizado para avaliar o tempo de carregamento da aplicação. O objetivo desta análise foi compreender o impacto do tamanho da base de dados no tempo de busca e na utilização de memória RAM.

Tabela 1 – Tempo médio (em segundos) de carregamento da base de dados e índice

Execução	Base de Dados	Índice	Total
1	0,19	17,03	17,22
2	0,19	17,48	17,67
3	0,20	18,76	18,96
4	0,74	22,08	22,82
5	0,20	17,61	17,81
6	0,20	17,17	17,37
7	0,17	17,77	17,94
8	0,20	17,68	17,87
9	0,20	17,41	17,60
10	0,20	17,43	17,63
Média	0,25	18,04	18,29

O comando `echo exit | ./run_console.sh` foi executado dez vezes em sequência, e o Genoogle fechado logo após o carregamento. Durante a inicialização, apenas os metadados da base e das sequências são carregados. De acordo com os dados da Tabela 1 esta etapa levou, em média, menos de um segundo. Os dados das sequências são acessados no disco conforme os HSPs encontrados durante a fase de consulta e, por isso, não há impacto. Já o índice é mantido em memória durante todo o ciclo de vida da aplicação. Foram necessários em torno de 18 segundos para carregá-lo.

O `jmap` é um utilitário que acompanha a instalação do Java e foi utilizado para analisar o uso de memória após a inicialização. O comando `jmap -dump:live` dispara a execução do coletor de lixo da JVM e lista os objetos remanescentes no espaço de *heap*. A Figura 28 traz a saída do programa. Segundo mostra a linha 3, a coleção de metadados das sequências armazenadas – objetos da classe `StoredSequenceInfo` – ocupou aproximadamente 14 MB de memória. As duas primeiras linhas correspondem ao índice, objeto do tipo `long[][]` representado pela convenção de nomes de classe para tipos primitivos em Java[2]. No total,

[2] *Class - Java Platform SE 7* – https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#getName()

Figura 28 – Utilização de memória após inicialização do Genoogle

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:        178711     3334128288  [J
   2:             1       16777232  [[J
   3:        252431       14136136  bio.pih.genoogle.io.proto.Io$StoredSequ...
   4:           936        1491712  [Ljava.lang.Object;
   5:          3967         304488  [C
   6:           469         234688  [B
   7:          1295         147448  java.lang.Class
   8:          3920          94080  java.lang.String
   9:           809          25888  java.util.concurrent.ConcurrentHashMap$...
  10:           610          19520  java.util.HashMap$Node
  ...
Total        449168     3367590584
```

o índice e suas entradas ocuparam aproximadamente 3,1 GB, ou 20% da capacidade da máquina.

Embora pareça desprezível, o tempo de carregamento impactaria o desempenho caso fosse executada uma busca para cada sequência de entrada. Em um cenário com 256 sequências, a soma dos tempos de inicialização implicaria no equivalente a **76 minutos** de carregamento sem qualquer alinhamento computado. E no caso de execuções concomitantes, cada processo carregaria uma cópia do índice em memória. Como consequência, a fração disponível para buscas reduziria significativamente e levaria a falhas de execução por falta de recursos.

Código 5.1 – Comando para submeter o Genuvem em um *cluster* de 4 nós

```
1  spark-submit \
2    --master yarn \
3    --num-executors 4 \
4    --executor-memory 12g \
5    --executor-cores 2 \
6    --conf spark.task.cpus=2 \
7    --class Genuvem \
8    genuvem.jar \
9    sars-cov-2-2021 \
10   sars-cov-2-2022-128.fasta
```

A conclusão desta análise levou à decisão de reparticionar a entrada (vide Código 4.2) e a restringir as instâncias do Genoogle a apenas uma por nó. Esta restrição é feita modificando os parâmetros de execução da aplicação Spark. No exemplo do Código 5.1, a consulta `sars-cov-2-2022-128.fasta` contra a base de dados `sars-cov-2-2021` é submetida em um cluster de 4 nós. O comando solicita 2 CPUs e 12 GB de memória RAM por

executor e define o mesmo número de CPUs por tarefa. Assim, o Spark inicializa apenas um executor por nó que, limitado pelo número de CPUs, processa uma tarefa por vez. Consequentemente, executa no máximo uma instância do Genoogle por nó.

5.4 ANÁLISE DE DESEMPENHO

Nesta seção são analisados os resultados obtidos pelo Genuvem em comparação ao Genoogle. O tempo de execução do comando `run_console.sh -b commands.bat` em uma máquina foi utilizado como parâmetro de comparação. Para o Genuvem, foi considerada a duração do comando `spark-submit` executado em *clusters* de 2, 4, 8, 16, e 32 nós. Cada busca foi executada seis vezes, a Tabela 2 mostra o tempo médio consolidado. Estes resultados foram utilizadas para calcular o *speedup* e a eficiência por tamanho de *cluster* com o objetivo de validar a escalabilidade horizontal da solução. Outros dados que serviram de base para esta análise encontram-se no Apêndice B.

Tabela 2 – Tempo médio (em segundos) de execução por quantidade de sequências de entrada

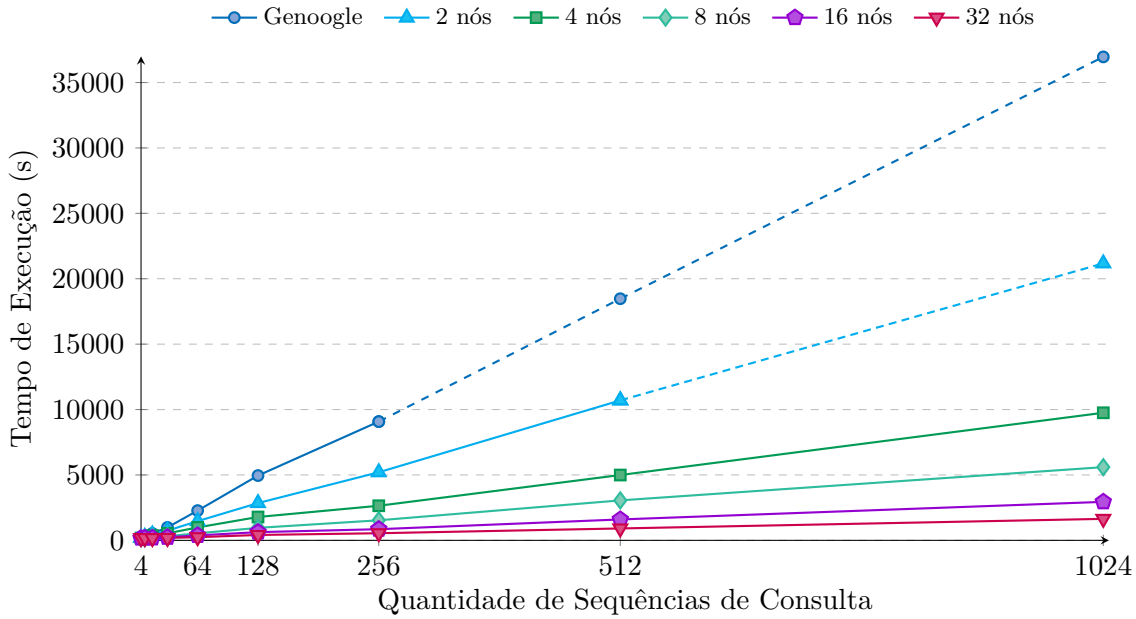
	Genoogle	**Genuvem**				
Nós	**1**	**2**	**4**	**8**	**16**	**32**
4 seqs.	140,00	192,00	180,67	161,83	167,67	172,00
8 seqs.	259,83	268,67	211,67	157,67	175,83	176,00
16 seqs.	531,50	452,67	368,50	240,17	184,17	171,67
32 seqs.	989,00	740,00	535,50	314,33	233,83	186,67
64 seqs.	2275,33	1467,83	988,67	536,17	377,17	242,67
128 seqs.	4953,17	2849,50	1785,83	954,50	627,50	411,83
256 seqs.	9079,33	5204,67	2644,50	1533,83	852,83	544,50
512 seqs.	18471,59[a]	10699,33	4993,83	3055,50	1598,17	910,00
1024 seqs.	36959,67[a]	21173,10[a]	9753,33	5596,60	2943,67	1643,33

Nota: [a] Os valores indicados na tabela foram estimados por regressão linear.

O Genoogle falhou ao processar consultas de 512 e 1024 sequências. O registro de execução armazenado nas máquinas apontou falha por falta de memória após, aproximadamente, 9 horas de execução. Este problema ocorre pois o Genoogle acumula os HSPs em memória até o fim dos alinhamentos, para então produzir o arquivo XML de saída. O Genuvem, no *cluster* de 2 nós, também falhou ao processar 1024 sequências. Neste caso, o algoritmo dividiu a consulta em dois lotes de 512 e distribuiu aos executores. Consequentemente, as instâncias locais não processaram este volume, falhando pelo mesmo motivo do Genoogle em execução isolada.

A Figura 29 mostra o tempo de execução por tamanho da consulta, onde cada série representa uma configuração de *cluster* utilizada. A busca de 4 sequências de entrada utilizando Genoogle em nó isolado levou 140 segundos. O Genuvem obteve desempenho pior em todos os *clusters*. Para consultas maiores, o gráfico mostra que o tempo de execução

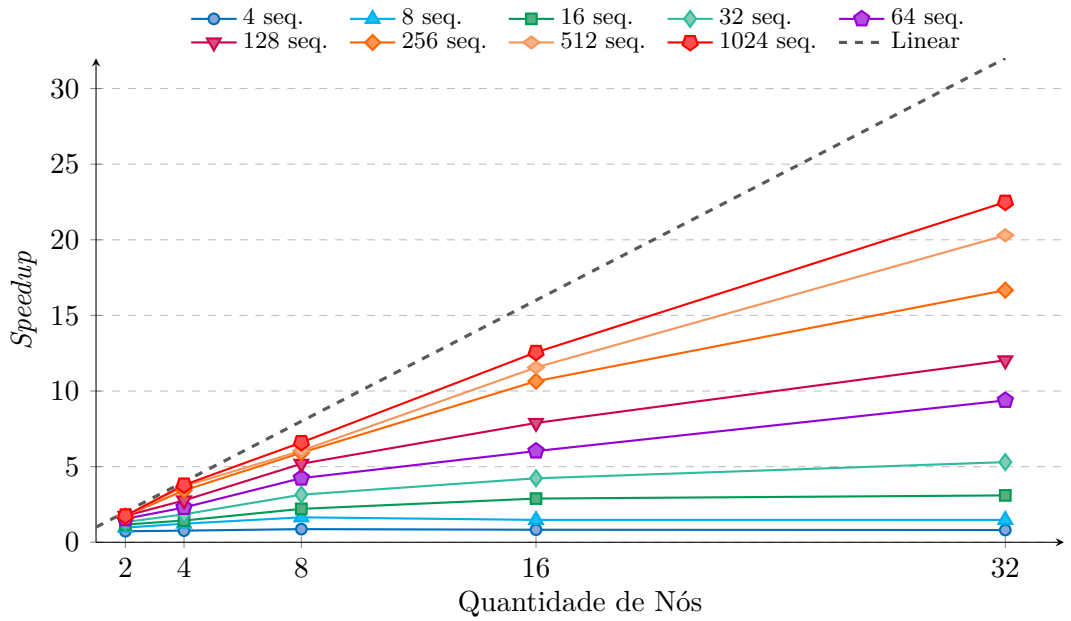
Figura 29 – Tempo de execução por tamanho da entrada para o Genoogle e Genuvem em diferentes *clusters*



cresceu conforme a quantidade de sequência de entrada, com uma taxa de crescimento inversamente proporcional à quantidade de nós. Em média, o Genoogle levou 151 minutos para buscar 256 sequências de entrada. Com o Genuvem em 32 nós, o mesmo processo levou apenas 9 minutos. A busca de 1024 sequências deu-se em 27 minutos no mesmo *cluster*, enquanto as estimativas mostram que no Genoogle demoraria mais de 10 horas.

A Figura 30 mostra a tendência de crescimento do *speedup* conforme aumenta-se o número de nós e a carga de trabalho. Foram observados *speedups* inferiores a 1 nos

Figura 30 – *Speedup* por nós de processamento para cada tamanho da entrada

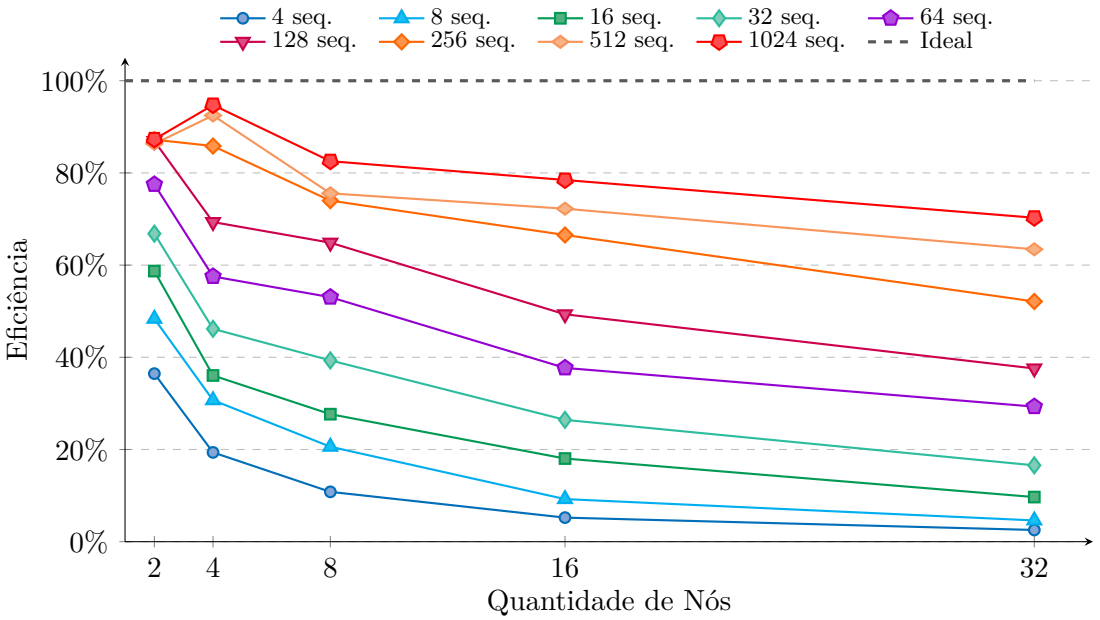


experimentos de 4 e 8 sequências, o que demonstra o impacto do Hadoop e Spark na latência de tarefas cuja carga de trabalho não é grande o suficiente para aproveitar os recursos disponíveis. Por outro lado, o Genuvem obteve *speedup* próximo do linear ao executar buscas acima de 128 sequências nos *clusters* de 2, 4 e 8 nós.

O gráfico da Figura 31 mostra que o ganho de desempenho com as consultas maiores resultou em eficiência superior a 80%. A maior eficiência observada em cenários sem estimativas foi 87,22%, referente à busca de 256 sequências em *cluster* de 2 nós. O Genuvem em 32 nós não apresentou eficiência superior a 80% em nenhum cenário. A carga de trabalho das consultas realizadas não foi suficiente para explorar todo o potencial do *hardware* disponível deste *cluster*.

Em síntese, os resultados obtidos demonstraram que a utilização do Genuvem é escalável, alcançando *speedup* próximo do ideal e eficiência superior a 80% em consultas suficientemente grandes. A distribuição da execução viabilizou o processamento de grandes consultas que, de outro modo, não seriam processados pelo Genoogle. Finalmente, os alinhamentos obtidos por ambas as ferramentas foram comparados. Verificou-se que, em todos os casos, os resultados retornados foram idênticos. Por este motivo, não foram discutidas a significância biológica e a sensibilidade das buscas realizadas.

Figura 31 – Eficiência por nós de processamento para cada tamanho da entrada



5.5 LIMITAÇÕES

Como observado, o Genoogle acumula os alinhamentos de todas as sequências de entrada em memória. Isto prejudicou a execução de consultas acima de 256 sequências e impactou a execução do Genuvem em um dos cenários de teste. Como consequência, o *speedup* das consultas com tamanho 512 e 1024 foram estimados por regressão linear.

Assim como o SparkBLAST, o Genuvem distribui a carga de trabalho paralelizando as sequências de consulta. Este método limita o tamanho da base de dados à quantidade de memória disponível por nó e impossibilita a comparação de genomas de organismos mais complexos, cujas sequências tem comprimento na ordem de bilhões de pares de base.

Já a alocação de 2 núcleos de CPU para cada tarefa Spark, faz com que cada executor possa realizar apenas uma busca de sequências por vez. No mundo real, é provável que sejam feitas mais de uma consulta ao mesmo tempo, o que pode acarretar em enfileiramento de tarefas e aumentar a latência do ponto de vista do usuário.

Outro fator limitante é a divisão igualitária da consulta por número de executores. Em caso de falha no processamento de uma sequência de entrada, todas as as demais daquela partição serão realizadas novamente, e impacta sua eficiência. Encontrar a quantidade ideal de sequências de entrada por tarefa é um exercício que será deixado como sugestão de trabalho futuro.

Por fim, neste trabalho não houve comparação com o CloudBLAST ou SparkBLAST pois queria-se demonstrar o ganho de desempenho do Genuvem em relação ao Genoogle. Todavia, seria interessante comparar o Genuvem e aquelas ferramentas a partir de um estudo de caso real, com fundamentação biológica, e avaliar não apenas o desempenho computacional mas também a significância dos alinhamentos obtidos. Esta análise, por sua complexidade e custo com provedores de nuvem, também será sugerida como trabalho futuro.

6 CONCLUSÃO

O BLAST consolidou-se como o uma das principais ferramentas para busca de sequências genômicas em bases de dados. Dele, derivaram projetos que buscaram otimizar a execução do algoritmo para comportar o volume crescente de sequências genômicas disponibilizadas ano a ano. O Genoogle obteve ganhos significativos de desempenho ao utilizar índices invertidos para acelerar o processo de semeadura de HSPs, a paralelização da fase de extensão, e o uso eficiente de memória com a otimização das estruturas de dados utilizadas para armazenamento da base de dados. Contudo, ambas as ferramentas não suportam execução em ambiente distribuído.

O CloudBLAST inovou ao utilizar a API Hadoop Streaming para distribuir a execução do BLAST sem alterar o seu código fonte. Baseado neste modelo, o SparkBLAST obteve ganhos de desempenho ainda maiores ao utilizar o operador *pipe* do Spark para a mesma finalidade. Ambos os projetos demonstraram que é possível escrever programas distribuídos para busca de sequências utilizando plataformas de *Big Data*, que abstraem a complexidade de gestão de recursos e coordenação de tarefas.

O principal objetivo deste trabalho foi propor uma solução horizontalmente escalável para buscas de sequências. A pesquisa realizada mostrou que nenhum outro projeto deu continuidade ao Genoogle até o momento. Identificou-se então a oportunidade de otimizar a ferramenta utilizando modelo similar ao do SparkBLAST. Foram desenvolvidos *scripts* para adaptar o Genoogle à interface do operador *pipe* do Spark. Um notebook do Apache Zeppelin foi utilizado como protótipo de visualização dos resultados das buscas.

A solução foi validada no ambiente de nuvem da *Google Cloud Platform*. Foram criados conjuntos de dados de genoma do vírus SARS-CoV-2 para a base de dados e para as consultas. As buscas foram paralelizadas distribuindo as consultas entre os nós dos *clusters*, e confrontaram genomas contra genomas. O desempenho das execuções do Genuvem foi avaliado em função do *speedup* em relação ao Genoogle e da eficiência alcançada. Os resultados demonstraram que a solução escala horizontalmente, isto é, o desempenho melhora conforme aumenta-se o número de nós. Em consultas suficientemente grandes, obteve-se eficiência acima de 80%. A distribuição de buscas por divisão da base de dados, assim como no SparkBLAST, não foi implementada. Esta limitação restringiu o tamanho da base à memória dos executores.

Concluiu-se, então, que este trabalho atingiu seus objetivos planejados, pois identificou no Genoogle uma oportunidade de otimização e demonstrou a escalabilidade da solução adaptada à execução sobre o Spark.

6.1 CONTRIBUIÇÕES

O presente trabalho resultou nestas contribuições:

- Adaptação do Genoogle para execução em ambiente distribuído e escalável utilizando Spark; e
- Desenvolvimento de ambiente interativo para execução de busca de sequências, e visualização de resultados e alinhamentos.

O código-fonte e a documentação dos *scripts* e demais artefatos foram disponibilizados no endereço https://github.com/rsantanna/genuvem. Também encontram-se neste repositório as instruções para configurações de ambiente de desenvolvimento local e em nuvem, visando facilitar a reprodução dos experimentos por terceiros.

Os arquivos FASTA utilizados para construção da base de dados e das consultas executadas foram disponibilizados na GCP no endereço https://console.cloud.google.com/storage/browser/genuvem-public.

6.2 TRABALHOS FUTUROS

Primeiramente, é possível realizar estudo das configurações de CPU e memória para executores e tarefas. Por exemplo, máquinas com mais recursos podem ter duas ou mais instâncias do Genoogle em execução, e os parâmeros de executores e tarefas podem ser variados para identificar a configuração ótima para uma dada base de dados.

Para mitigar o enfileiramento de tarefas e problemas de eficiência decorrentes de divisão estática da consulta, o Genoogle pode ser carregado como serviço durante a inicialização dos nós do cluster e manter os índices em memória. Ao invés de utilizar a interface de busca em lote, pode-se explorar a API do modo *WebService*.

A funcionalidade do Genoogle para fragmentação da base de dados não foi explorada neste trabalho. É possível adaptar os *scripts* do Genuvem para tal, e assim viabilizar consultas em bases de dados maiores que as capacidades de memória e armazenamento de uma única máquina.

A avaliação de desempenho do Genuvem em relação ao SparkBLAST e CloudBAST pode ser interessante para avaliar se os ganhos de desempenho do Genoogle em relação ao BLAST permanecem após anos de desenvolvimento interrompido.

E a exemplo do SparkyBlast, um possível e complexo trabalho é a reimplementação do Genoogle utilizando Spark e/ou outras ferramentas para processamento de dados em larga escala.

REFERÊNCIAS

ALBRECHT, F. F. **Algoritmo otimizado para comparação de sequências e busca em base de dados**. Dissertação (Mestrado em Sistemas e Computação) — Instituto Militar de Engenharia, Rio de Janeiro, 2009. Disponível em: http://www.comp.ime.eb.br/pos/?pc=p_detd&num=622.

ALTSCHUL, S. F. et al. Basic local alignment search tool. **Journal of Molecular Biology**, v. 215, n. 3, p. 403–410, 10 1990. ISSN 00222836. Disponível em: https://doi.org/10.1016/S0022-2836(05)80360-2.

CAMACHO, C. et al. ElasticBLAST: accelerating sequence search via cloud computing. **BMC Bioinformatics**, v. 24, p. 117, 3 2023. ISSN 1471-2105. Disponível em: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-023-05245-9.

CASTRO, M. R. d. **SparkBLAST : utilização da ferramenta Apache Spark para a execução do BLAST em ambiente distribuído e escalável**. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de São Carlos, São Carlos, 2017. Disponível em: https://repositorio.ufscar.br/handle/ufscar/9114.

CHAMBERS, B.; ZAHARIA, M. **Spark: The Definitive Guide Big Data Processing Made Simple**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2018. ISBN 1491912219.

CHENG, Y. et al. Building Big Data Processing and Visualization Pipeline through Apache Zeppelin. In: **Proceedings of the Practice and Experience on Advanced Research Computing**. New York, NY, USA: ACM, 2018. p. 1–7. ISBN 9781450364461. Disponível em: https://doi.org/10.1145/3219104.3229288.

CORES, F.; GUIRADO, F.; LERIDA, J. L. High throughput BLAST algorithm using spark and cassandra. **The Journal of Supercomputing**, Springer, v. 77, n. 2, p. 1879–1896, 2 2021. ISSN 0920-8542. Disponível em: https://doi.org/10.1007/s11227-020-03338-3.

CORMEN, T. H. et al. **Introduction to Algorithms**. [S.l.: s.n.], 2009. 1312 p. ISBN 978-0-262-03384-8.

COX, M.; ELLSWORTH, D. Application-controlled demand paging for out-of-core visualization. In: **Proceedings. Visualization '97 (Cat. No. 97CB36155)**. [S.l.: s.n.], 1997. p. 235–244.

CRISTIANINI, N.; HAHN, M. W. **Introduction to Computational Genomics: A Case Studies Approach**. USA: Cambridge University Press, 2007. ISBN 0521856035.

DARLING, A. E.; CAREY, L.; FENG, W. C. The design, implementation, and evaluation of mpiBLAST. 1 2003. Disponível em: https://www.osti.gov/biblio/976625.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: **Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6**. USA: USENIX Association, 2004. (OSDI'04), p. 10.

FASSLER, J.; COOPER, P. **BLAST Glossary**. National Center for Biotechnology Information, 2011. Disponível em: https://www.ncbi.nlm.nih.gov/books/NBK62051/.

GENBANK. **Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/USA/MT-MTPHL-3966153/2022, complete genome**. National Library of Medicine, 2022. Disponível em: https://www.ncbi.nlm.nih.gov/nuccore/ON642517.1/. Acesso em: 26 jan. 2022.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The Google file system. In: **Proceedings of the nineteenth ACM symposium on Operating systems principles**. New York, NY, USA: ACM, 2003. p. 29–43. ISBN 1581137575. Disponível em: https://doi.org/10.1145/945445.945450.

HATCHER, E. L. et al. Virus Variation Resource – improved response to emergent viral outbreaks. **Nucleic Acids Research**, Oxford University Press, v. 45, n. D1, p. D482–D490, 1 2017. ISSN 0305-1048. Disponível em: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkw1065.

HESPER, B.; HOGEWEG, P. Bioinformatica: een werkconcept. **Kameleon**, v. 1, n. 6, p. 28–29, 1970.

HOGEWEG, P. The Roots of Bioinformatics in Theoretical Biology. **PLoS Computational Biology**, v. 7, n. 3, p. e1002021, 3 2011. ISSN 1553-7358. Disponível em: https://dx.plos.org/10.1371/journal.pcbi.1002021.

JABLONKA, E.; LAMB, M. J. **Evolution in four dimensions: Genetic, epigenetic, behavioral, and symbolic variation in the history of life**. Cambridge, Massachusetts, USA: The MIT Press, 2005. 472 p. ISBN 0-262-10107-6.

LIU, F. et al. Integrated HPC Scheduler Data Processing Workflow using Apache Zeppelin. In: **2018 IEEE International Conference on Big Data (Big Data)**. IEEE, 2018. p. 4254–4260. ISBN 978-1-5386-5035-6. Disponível em: https://doi.org/10.1109/BigData.2018.8622085.

MATSUNAGA, A.; TSUGAWA, M.; FORTES, J. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. In: **2008 IEEE Fourth International Conference on eScience**. IEEE, 2008. p. 222–229. ISBN 978-1-4244-3380-3. Disponível em: http://doi.org/10.1109/eScience.2008.62.

MCAFEE, A.; BRYNJOLFSSON, E. Big data: the management revolution. **Harvard business review**, v. 90, p. 60–6, 68, 128, 10 2012. ISSN 0017-8012.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured Parallel Programming: Patterns for Efficient Computation**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123914439.

NCBI. **GenBank and WGS Statistics**. National Library of Medicine, 2023. Disponível em: https://www.ncbi.nlm.nih.gov/genbank/statistics/. Acesso em: 08 fev. 2023.

NCBI. **SARS-CoV-2 Resources**. National Library of Medicine, 2023. Disponível em: https://www.ncbi.nlm.nih.gov/sars-cov-2/. Acesso em: 08 fev. 2023.

NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. **Journal of Molecular Biology**, v. 48, n. 3, p. 443–453, 1970. ISSN 00222836. Disponível em: http://www.bbc.com/news/world-us-canada-38781302.

ORENGO, C.; JONES, D.; THORNTON, J. **Bioinformatics: genes, proteins, and computers**. New York: BIOS Scientific Publishers Limited, 2003. 320 p. ISBN 9781859960547.

PEARSON, W. R. Rapid and sensitive sequence comparison with FASTP and FASTA. **Methods in enzymology**, v. 183, n. 1988, p. 63–98, 1990. ISSN 0076-6879. Disponível em: http://doi.org/10.1016/0076-6879(90)83007-v.

SAYERS, E. W. et al. GenBank 2023 update. **Nucleic acids research**, NLM (Medline), v. 51, n. D1, p. D141–D144, 1 2023. ISSN 1362-4962. Disponível em: http://doi.org/10.1093/nar/gkac1012.

SELZER, P. M.; MARHÖFER, R. J.; KOCH, O. **Applied Bioinformatics**. Cham: Springer International Publishing, 2018. ISBN 978-3-319-68299-0.

SMITH, T. F.; WATERMAN, M. S.; FITCH, W. M. Comparative biosequence metrics. **Journal of Molecular Evolution**, v. 18, n. 1, p. 38–46, 1981. ISSN 00222844.

The Apache Software Foundation. **Cluster Mode Overview - Spark 3.2.1 Documentation**. 2023. Disponível em: https://spark.apache.org/docs/3.2.1/cluster-overview.html. Acesso em: 08 fev. 2023.

The Apache Software Foundation. **Spark 3.2.1 Documentation**. 2023. Disponível em: https://spark.apache.org/docs/3.2.1/. Acesso em: 08 fev. 2023.

WHITE, T. **Hadoop: The Definitive Guide**. 4th. ed. [S.l.]: O'Reilly Media, Inc., 2015. ISBN 1491901632.

ZAHARIA, M. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: **Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation**. USA: USENIX Association, 2012. (NSDI'12), p. 2.

ZAHARIA, M. et al. Spark: Cluster computing with working sets. In: **Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing**. USA: USENIX Association, 2010. (HotCloud'10), p. 10.

GLOSSÁRIO

alinhamento processo de alinhamento entre duas ou mais sequências para identificar o grau de similaridades entre as mesmas e é classificado de duas formas.
1) **global** alinhamento por todo o comprimento de duas sequências de entrada.
2) **local** alinhamento de regiões de alta similaridade.

BLAST *Basic Local Alignment Search Tool* é um algoritmo para comparação de sequências utilizado para busca de alinhamentos locais ótimos em uma base de dados.

e-value *expectation value*, ou valor esperado de alinhamentos com pontuação maior ou igual ao *score* do *hit* em questão que podem ser obtidos ao acaso. 1, *veja* hit

hit correspondência entre uma consulta e uma referência, cuja significância é determinada pelo *score* e *e-value*.
homologia similaridade atribuída a descendentes de um ancestral comum.
HSP *high-scoring segment pair*, ou par de segmentos de alta pontuação, é um alinhamento entre duas subsequências que indica uma região de alta similaridade.

score pontuação de um hit em função da quantidade de elementos alinhados e espaços introduzidos, em geral calculado com o auxílio de uma função ou matriz de pontuação. 1, *veja* hit

APÊNDICE A – DOCUMENTAÇÃO

O código-fonte do Genuvem foi disponibilizado no GitHub no seguinte repositório: https://github.com/rsantanna/genuvem. Nele, encontram-se o os *scripts* detalhados no Capítulo 4. O projeto possui a seguinte estrutura de pastas:

- **conf**: arquivos de configuração do Genoogle e do Genuvem
- **docker**: imagens do Docker
- **docs**: documentação detalhada do projeto
- **files**: base de dados e arquivos FASTA de consulta
- **notebook**: diretório de *notebooks* do Zeppelin
- **scripts**: diretório de *scripts* auxiliares para implantação e execução do Genuvem
- **src**: código-fonte do Genuvem

A documentação detalhada e atualizada encontra-se no repositório para consulta. Neste apêndice, será fornecida uma explicação em alto nível do processo de implantação da ferramenta e seus ambientes.

A.1 CONFIGURAÇÕES

Conforme explicado no Capítulo 4, o Genuvem orquestra a execução do Genoogle em cada nó. Na pasta de configurações, encontra-se o arquivo `genoogle.xml` onde o usuário define os parâmetros de codificação da base de dados e de execução das buscas. Os parâmetros mais relevantes para esta monografia serão detalhados adiante no Capítulo 5. A documentação completa da configuração do Genoogle encontra-se em seu repositório[1].

O Genuvem, enquanto aplicação, não possui arquivos de configuração de execução. Apenas de implantação do ambiente em nuvem. No arquivo `genuvem-env.sh`, o usuário define o nome dos *buckets* de armazenamento do *Google Cloud Storage* que serão utilizados para guardar os arquivos do Genuvem na GCP.

- **RESOURCES_BUCKET**: armazena a base de dados do Genoogle e as consultas
- **TEMP_BUCKET**: armazenamento de arquivos temporários do Hadoop e Spark
- **DATAPROC_BUCKET**: *bucket* de uso interno dos serviço de processamento de dados da GCP

[1] felipealbrecht/Genoogle – https://github.com/felipealbrecht/Genoogle

A.2 AMBIENTE DE DESENVOLVIMENTO

O projeto disponibiliza duas imagens Docker[2] para ambiente de desenvolvimento local. A imagem *hadoop-snc* implanta um contêiner para simular um *cluster* Hadoop de um único nó. Este contêiner viabiliza o desenvolvimento local da aplicação, incluindo testes em conjuntos de dados pequenos, sem incorrer no custo de um cluster real em nuvem. O ambiente é inicializado com o comando `docker compose up` e possui Genoogle, Spark e Zeppelin configurados.

A imagem *genoogle* configura um contêiner com apenas o Genoogle. Pode ser utilizada para executar a ferramenta sem a necessidade de compilar e construir o projeto localmente. O comando `docker compose run genoogle` executa o Genoogle em modo interativo. O usuário pode ainda passar os parâmetros `-g` para executar o codificador da base de dados e `-b <arquivo de comandos>` para executar comandos no modo de execução em lote.

A.3 CODIFICAÇÃO E INDEXAÇÃO

Este projeto contém uma base de dados configurada na seção `genoogle:databanks` que aponta para um arquivo FASTA de exemplo. Para codificar e indexar a base que acompanha o projeto, basta executar o comando `docker compose run genoogle -g`.

O codificador do Genoogle armazena os arquivos da base e do índice invertido no diretório de arquivos do projeto. Caso o usuário altere os parâmetros de codificação ou adicione novos arquivos FASTA, é necessário executá-lo novamente para formatar a base.

A.4 IMPLANTAÇÃO EM NUVEM

A pasta de *scripts* guarda as automações para implantação do Genuvem na *Google Cloud Platform*. Uma vez concluídas as configurações e codificação, o usuário deve executar o script `setup.sh` que criará automaticamente os *buckets* no GCS e copiará a base de dados, arquivos de consulta e demais arquivos de apoio.

O *script* `create_cluster.sh <nome> <tamanho>` cria um *cluster* Hadoop na GCP de nome e quantidade de nós definido pelo usuário. Durante a inicialização dos nós, o *script* `bootstrap.sh` será executado para instalação do Genoogle e cópia da base de dados do GCS para o armazenamento local.

[2] Docker – https://www.docker.com/

APÊNDICE B – DADOS ADICIONAIS

Este apêndice contém dados adicionais que fundamentam os resultados analisados no Capítulo 5, organizados por quantidade de sequências de entrada utilizadas em cada medição.

B.1 4 SEQUÊNCIAS – 119 KB, 120.716 NUCLEOTÍDEOS

Tabela 3 – Tempo de execução em segundos do experimento de 4 sequências

	Genoogle	**Genuvem**				
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	142,00	205,00	199,00	197,00	179,00	172,00
Execução 2	140,00	194,00	180,00	163,00	167,00	170,00
Execução 3	155,00	191,00	174,00	156,00	165,00	172,00
Execução 4	152,00	176,00	172,00	148,00	156,00	156,00
Execução 5	126,00	202,00	194,00	161,00	177,00	185,00
Execução 6	125,00	184,00	165,00	146,00	162,00	177,00
Média	140,00	192,00	180,67	161,83	167,67	172,00
Desvio Padrão	12,60	10,90	13,26	18,52	8,85	9,53
Coef. de Var.	9,00%	5,68%	7,34%	11,44%	5,28%	5,54%
Speedup	1,00	0,73	0,77	0,87	0,83	0,81
Eficiência	100,00%	36,46%	19,37%	10,81%	5,22%	2,54%

B.2 8 SEQUÊNCIAS – 238 KB, 241.432 NUCLEOTÍDEOS

Tabela 4 – Tempo de execução em segundos do experimento de 8 sequências

	Genoogle	**Genuvem**				
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	258,00	263,00	213,00	152,00	169,00	177,00
Execução 2	253,00	296,00	217,00	156,00	158,00	175,00
Execução 3	278,00	244,00	205,00	174,00	217,00	185,00
Execução 4	268,00	264,00	201,00	161,00	171,00	175,00
Execução 5	258,00	268,00	223,00	147,00	161,00	166,00
Execução 6	244,00	277,00	211,00	156,00	179,00	178,00
Média	259,83	268,67	211,67	157,67	175,83	176,00
Desvio Padrão	11,84	17,20	7,97	9,27	21,51	6,13
Coef. de Var.	4,56%	6,40%	3,76%	5,88%	12,23%	3,48%
Speedup	1,00	0,97	1,23	1,65	1,48	1,48
Eficiência	100,00%	48,36%	30,69%	20,60%	9,24%	4,61%

B.3 16 SEQUÊNCIAS – 475 KB, 483.258 NUCLEOTÍDEOS

Tabela 5 – Tempo de execução em segundos do experimento de 16 sequências

	Genoogle	**Genuvem**			
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	536,00	447,00	362,00	246,00	200,00	175,00
Execução 2	543,00	445,00	394,00	252,00	181,00	174,00
Execução 3	514,00	480,00	340,00	227,00	195,00	170,00
Execução 4	492,00	442,00	394,00	216,00	179,00	182,00
Execução 5	580,00	442,00	315,00	213,00	178,00	161,00
Execução 6	524,00	460,00	406,00	287,00	172,00	168,00
Média	531,50	452,67	368,50	240,17	184,17	171,67
Desvio Padrão	29,76	14,96	35,84	27,80	10,87	7,12
Coef. de Var.	5,60%	3,31%	9,73%	11,57%	5,90%	4,15%
Speedup	1,00	1,17	1,44	2,21	2,89	3,10
Eficiência	100,00%	58,71%	36,06%	27,66%	18,04%	9,68%

B.4 32 SEQUÊNCIAS – 950 KB, 966.917 NUCLEOTÍDEOS

Tabela 6 – Tempo de execução em segundos do experimento de 32 sequências

	Genoogle	**Genuvem**			
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	980,00	704,00	554,00	318,00	216,00	193,00
Execução 2	933,00	785,00	518,00	314,00	207,00	214,00
Execução 3	983,00	725,00	505,00	286,00	252,00	175,00
Execução 4	1011,00	753,00	555,00	337,00	227,00	181,00
Execução 5	1022,00	698,00	555,00	312,00	238,00	181,00
Execução 6	1005,00	775,00	526,00	319,00	263,00	176,00
Média	989,00	740,00	535,50	314,33	233,83	186,67
Desvio Padrão	31,88	36,62	22,04	16,48	21,37	14,84
Coef. de Var.	3,22%	4,95%	4,12%	5,24%	9,14%	7,95%
Speedup	1,00	1,34	1,85	3,15	4,23	5,30
Eficiência	100,00%	66,82%	46,17%	39,33%	26,43%	16,56%

B.5 64 SEQUÊNCIAS – 1,9 MB, 1.933.974 NUCLEOTÍDEOS

Tabela 7 – Tempo de execução em segundos do experimento de 64 sequências

	Genoogle	**Genuvem**				
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	2053,00	1480,00	969,00	552,00	378,00	238,00
Execução 2	2625,00	1437,00	911,00	520,00	377,00	234,00
Execução 3	2325,00	1547,00	930,00	511,00	343,00	233,00
Execução 4	2200,00	1418,00	1035,00	553,00	369,00	258,00
Execução 5	2341,00	1412,00	1046,00	553,00	391,00	252,00
Execução 6	2108,00	1513,00	1041,00	528,00	405,00	241,00
Média	2275,33	1467,83	988,67	536,17	377,17	242,67
Desvio Padrão	205,99	54,78	60,06	18,86	20,98	10,15
Coef. de Var.	9,05%	3,73%	6,07%	3,52%	5,56%	4,18%
Speedup	1,00	1,55	2,30	4,24	6,03	9,38
Eficiência	100,00%	77,51%	57,54%	53,05%	37,70%	29,30%

B.6 128 SEQUÊNCIAS – 3,8 MB, 3.867.900 NUCLEOTÍDEOS

Tabela 8 – Tempo de execução em segundos do experimento de 128 sequências

	Genoogle	**Genuvem**				
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	4879,00	3100,00	1843,00	937,00	697,00	440,00
Execução 2	5117,00	2875,00	1706,00	918,00	625,00	390,00
Execução 3	5015,00	2814,00	1682,00	969,00	623,00	404,00
Execução 4	4954,00	2900,00	1834,00	866,00	580,00	432,00
Execução 5	4910,00	2705,00	1830,00	1036,00	595,00	414,00
Execução 6	4844,00	2703,00	1820,00	1001,00	645,00	391,00
Média	4953,17	2849,50	1785,83	954,50	627,50	411,83
Desvio Padrão	99,95	148,02	71,92	60,80	41,17	20,87
Coef. de Var.	2,02%	5,19%	4,03%	6,37%	6,56%	5,07%
Speedup	1,00	1,74	2,77	5,19	7,89	12,03
Eficiência	100,00%	86,91%	69,34%	64,87%	49,33%	37,58%

B.7 256 SEQUÊNCIAS – 7,4 MB, 7.752.518 NUCLEOTÍDEOS

Tabela 9 – Tempo de execução em segundos do experimento de 256 sequências

	Genoogle	**Genuvem**			
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	9048,00	5147,00	2512,00	1414,00	830,00	565,00
Execução 2	10044,00	5362,00	2594,00	1554,00	894,00	562,00
Execução 3	8696,00	5380,00	2626,00	1509,00	835,00	530,00
Execução 4	8764,00	5270,00	2503,00	1610,00	845,00	521,00
Execução 5	8815,00	4953,00	2807,00	1602,00	897,00	531,00
Execução 6	9109,00	5116,00	2825,00	1514,00	816,00	558,00
Média	9079,33	5204,67	2644,50	1533,83	852,83	544,50
Desvio Padrão	499,74	163,88	141,04	72,42	34,36	19,25
Coef. de Var.	5,50%	3,15%	5,33%	4,72%	4,03%	3,54%
Speedup	1,00	1,74	3,43	5,92	10,65	16,67
Eficiência	100,00%	87,22%	85,83%	73,99%	66,54%	52,11%

B.8 512 SEQUÊNCIAS – 14,8 MB, 15.491.705 NUCLEOTÍDEOS

Tabela 10 – Tempo de execução em segundos do experimento de 512 sequências

	Genoogle	**Genuvem**			
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	...	9916,00	4929,00	2884,00	1562,00	898,00
Execução 2	...	11170,00	5119,00	2943,00	1606,00	941,00
Execução 3	...	10600,00	5143,00	3222,00	1484,00	874,00
Execução 4	...	9337,00	5038,00	3081,00	1562,00	944,00
Execução 5	...	11397,00	4687,00	3136,00	1678,00	909,00
Execução 6	...	11776,00	5047,00	3067,00	1697,00	894,00
Média	18471,59[a]	10699,33	4993,83	3055,50	1598,17	910,00
Desvio Padrão	...	932,41	167,96	124,14	79,82	27,62
Coef. de Var.	...	8,71%	3,36%	4,06%	4,99%	3,04%
Speedup	1,00	1,73	3,70	6,05	11,56	20,30
Eficiência	100,00%	86,32%	92,47%	75,57%	72,24%	63,43%

Notas: [a] Média estimada por regressão linear conforme a Tabela 5.4.
Sinal convencional utilizado:
... Dado numérico não disponível

B.9 1024 SEQUÊNCIAS – 29,7 MB, 30.997.232 NUCLEOTÍDEOS

Tabela 11 – Tempo de execução em segundos do experimento de 1024 sequências

	Genoogle	**Genuvem**				
Nós	**1**	**2**	**4**	**8**	**16**	**32**
Execução 1	...	...	9383,00	0,00	2837,00	1646,00
Execução 2	...	...	9521,00	5632,00	2792,00	1669,00
Execução 3	...	...	9505,00	5404,00	2997,00	1648,00
Execução 4	...	...	10774,00	5497,00	2984,00	1676,00
Execução 5	...	...	9335,00	5661,00	3003,00	1620,00
Execução 6	...	...	10002,00	5789,00	3049,00	1601,00
Média	36959,67[a]	21173,10[a]	9753,33	5596,60	2943,67	1643,33
Desvio Padrão	...	...	553,42	2288,71	103,41	28,61
Coef. de Var.	...	...	5,67%	40,89%	3,51%	1,74%
Speedup	1,00	1,75	3,79	6,60	12,56	22,49
Eficiência	100,00%	87,28%	94,74%	82,55%	78,47%	70,28%

Notas: [a] Média estimada por regressão linear conforme a Tabela 5.4.
Sinal convencional utilizado:
... Dado numérico não disponível