

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Execução da Aplicação Paralela Tolerante a Falhas FT-CUDAlign Integrada ao Escalonador SLURM

Gustavo Santana Lima

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientadora
Prof.a Dr.a Alba Cristina Magalhaes Alves de Melo

Brasília
2025



Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Execução da Aplicação Paralela Tolerante a Falhas FT-CUDAlign Integrada ao Escalonador SLURM

Gustavo Santana Lima

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof.a Dr.a Alba Cristina Magalhaes Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Eduardo Adilo Pelinson Alchieri
CIC/UnB

Prof. Dr. Guilherme Novaes Ramos
Coordenador do Curso de Engenharia da Computação

Brasília, 09 de Dezembro de 2025

Dedicatória

Dedico este trabalho, com muito carinho, aos meus pais, especialmente ao meu pai, Marcelo de Mesquita Lima. Você foi, e continua sendo, uma grande inspiração para mim. Desde cedo, me ensinou a importância da ética e do que é certo, sendo fundamental na formação do meu caráter. Além disso, sempre me incentivou a buscar o conhecimento e a me dedicar aos estudos. Se hoje cheguei até aqui, grande parte disso se deve ao seu apoio e seus ensinamentos. Sou imensamente grato por tudo.

Agradecimentos

Gostaria de começar os agradecimentos citando a minha família. Agradeço imensamente aos meus pais, que sempre acreditaram em mim e me proporcionaram a oportunidade de vir para Brasília, permitindo que eu trilhasse esse caminho e conquistasse todos os passos que se seguiram. Sem o apoio e dedicação de vocês, nada disso seria possível.

Quero também agradecer às minhas irmãs, que sempre estiveram ao meu lado, torcendo por mim e sendo minhas maiores apoiadoras. O apoio de vocês foi essencial e fez toda a diferença ao longo desses anos.

A minha namorada, Mariana, merece um agradecimento especial. Ela esteve ao meu lado, me incentivando, e motivando para que eu me dedicasse ao meu melhor trabalho. Sua presença e apoio foram fundamentais.

Por fim, gostaria de expressar minha sincera gratidão à Profa. Dra. Alba Cristina, que me orientou com excelência e contribuiu de maneira decisiva para o desenvolvimento deste trabalho. Seu conhecimento, paciência e dedicação foram indispensáveis para que eu chegasse a esse resultado.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho apresenta o projeto, implementação e avaliação da integração entre o escalonador SLURM e o FT-CUDAlign, uma ferramenta de alinhamento exato de sequências biológicas tolerante a falhas. O alto custo computacional dos algoritmos de alinhamento por programação dinâmica, frequentemente aplicados em sequências com milhões de nucleotídeos ou aminoácidos, demandam o uso de GPUs para redução do tempo de execução. A ferramenta FT-CUDAlign utiliza diversas GPUs para a comparação de sequências, característica que a torna candidata para execução em ambientes de supercomputação. No entanto, como ela não estava, até o momento, integrada ao escalonador de aplicações SLURM, não podia ser usada em tais ambientes.

Para possibilitar a execução automatizada do FT-CUDAlign em clusters e supercomputadores, foi desenvolvido um projeto, que culminou em um script SLURM dividido em quatro etapas principais: (A) Detecção dinâmica dos recursos disponíveis nos nós do cluster, incluindo IPs e GPUs; (B) Geração automática do arquivo de configuração necessário ao módulo controlador do FT-CUDAlign; (C) Ativação e monitoramento dos balanceadores de carga associados a cada GPU; (D) Inicialização e supervisão do módulo de controle do FT-CUDAlign, com identificação de possíveis falhas. A integração permitiu também que o FT-CUDAlign seja executado em ambientes heterogêneos, mantendo suas tratativas internas de tolerância a falhas e ampliando sua portabilidade para diferentes infraestruturas de alto desempenho.

Os experimentos realizados mostraram que o SLURM foi capaz de executar o FT-CUDAlign em diferentes cenários, identificar falhas de GPUs e fornecer informações úteis ao processo, enquanto o FT-CUDAlign permaneceu responsável pela recuperação a partir de checkpoints. Foram analisados ainda os *overheads* associados às execuções com e sem falhas, mostrando que o custo adicional cresce conforme a troca de contexto aumenta e o tempo de paralelismo diminui, especialmente em sequências menores.

Palavras-chave: SLURM, FT-CUDAlign, MASA-CUDAlign, Comparação de Sequências Biológicas

Abstract

This work presents the project, implementation and analysis of integration between the SLURM scheduler and FT-CUDAlign, a fault-tolerant tool for exact biological sequence alignment. The high computational cost of dynamic programming–based alignment algorithms, which are often applied to sequences containing millions of nucleotides or aminoacids, demanding the use of GPUs to reduce execution time. The FT-CUDAlign framework uses GPUs for sequence comparisons, a characteristic that makes it a suitable candidate for supercomputer environments. However, until now, it has not been integrated into the SLURM scheduler, preventing it from being used in these environments.

To automate the execution of FT-CUDAlign on clusters and supercomputers, a SLURM script was developed and divided into four main steps: (A) Dynamic detection of the resources available in the cluster nodes, including IPs and GPUs; (B) Automatic generation of the configuration file required by the FT-CUDAlign controller module; (C) Activation and monitoring of the balancers associated with each GPU; (D) Initialization and supervision of the FT-CUDAlign's controller, with identification of potential failures. This integration enables FT-CUDAlign to run in heterogeneous environments while preserving its internal fault-tolerance mechanisms and improving its portability across various high-performance infrastructures.

Experimental results presented that SLURM successfully executed FT-CUDAlign across different scenarios, identified GPU failures, and provided useful diagnostic information, while FT-CUDAlign remained responsible for recovery through checkpoints. Additionally, overhead analyses for executions with and without failures showed that overhead increases as context-switching costs grow and the period of full parallelism decreases, particularly for smaller sequences.

Keywords: SLURM, FT-CUDAlign, MASA-CUDAlign, Biological Sequence Comparisons

Sumário

1 Introdução **1**
1.1 Objetivos . . . 2
1.2 Estrutura do Trabalho . . . 2

2 Comparação de Sequências Biológicas **3**
2.1 Conceitos Básicos . . . 3
2.2 Tipos de Algoritmos . . . 4
2.3 Algoritmos Exatos para Comparação de Sequências . . . 5
2.4 Algoritmo de Needleman-Wunsh (NW) . . . 5
2.5 Algoritmo de Smith-Waterman (SW) . . . 7
2.6 Algoritmo de Gotoh . . . 8
2.7 Algoritmo de Hirschberg . . . 9
2.8 Algoritmo de Myers-Miller(MM) . . . 10

3 A ferramenta MASA-CUDAlign **12**
3.1 MASA-CUDAlign em uma GPU . . . 12
3.1.1 CUDAlign 1.0 . . . 12
3.1.2 CUDAlign 2.0 . . . 15
3.1.3 CUDAlign 2.1 . . . 19
3.2 CUDAlign com múltiplas GPUs . . . 20
3.2.1 CUDAlign 3.0 . . . 20
3.2.2 CUDAlign 4.0 . . . 21
3.2.3 MASA-CUDAlign-MultiBP . . . 21
3.3 FT-CUDAlign . . . 23
3.3.1 Checkpoint . . . 23
3.3.2 Arquitetura do FT-CUDAlign . . . 24
3.3.3 Detecção de erros . . . 24
3.3.4 Protocolos de fim da iteração . . . 25

4 O Gerenciador de Execuções de Aplicações SLURM 26
4.1 Visão Geral . . . 26
4.2 Arquitetura do SLURM . . . 27
4.3 Segurança . . . 29
4.4 Políticas de Escalonamento . . . 30
4.4.1 Backfill . . . 30
4.4.2 Prioridade do Job . . . 31
4.4.3 Preempção . . . 31
4.5 Execução . . . 32
4.5.1 Filtros e Configurações de Execução . . . 32

5 Projeto da Integração entre o FT-CUDAlign e o SLURM 34
5.1 Visão Geral . . . 34
5.2 Configuração dos Serviços e Execução dos Módulos . . . 36
5.2.1 Execução MUNGE . . . 36
5.2.2 SLURM . . . 36
5.2.3 Configuração e Execução do NFS . . . 38
5.2.4 Execução do CUDAlign e do FT-CUDAlign . . . 38
5.3 Execução do FT-CUDAlign no SLURM . . . 39
5.4 Execuções sem Falhas e com Falhas . . . 42

6 Resultados Experimentais 44
6.1 Sequências Selecionadas . . . 44
6.2 Ambiente de Testes . . . 45
6.3 Script de Execução do FT-CUDAlign . . . 45
6.4 Experimentos . . . 46
6.4.1 Cenários sem Falha . . . 46
6.4.2 Cenários com Falha . . . 47

7 Conclusão 50
7.1 Trabalhos Futuros . . . 50

Referências 52

Apêndice 53

A Primeira página do artigo Apresentado no WPOS 2025 54

B Corpo do Script Utilizado 56

Lista de Figuras

2.1 Comparação global entre S_0 e S_1, com $escore = -2$ 4
2.2 Comparação local entre S_0 e S_1, com $escore = +3$ 4
2.3 Matriz de Needleman-Wunsh 6
2.4 Alinhamento de Needleman-Wunsh 6
2.5 Matriz de Smith-Waterman 7
2.6 Alinhamento local ótimo para o algoritmo SW 8
2.7 Matrizes do algoritmo de Gotoh: F, H e E 9
2.8 Alinhamento local ótimo para o algoritmo Gotoh 9
2.9 Representação do ponto médio ótimo(*crosspoint*) na matriz de similaridade Myers-Miller [1] 11

3.1 Divisão da matriz em blocos. Em azul está a diagonal externa D_3. 13
3.2 Bloco Retangular representando as diagonais e o paralelismo interno. . . . 13
3.3 Bloco não retangular, células com colunas negativas se referem a células delegadas de blocos já processados. 14
3.4 Delegação de células entre blocos[2] 15
3.5 Estágio 1 do CUDAlign 2.0 [2] 15
3.6 Estágio 2 do CUDAlign 2.0 [2] 16
3.7 Estágio 3 do CUDAlign 2.0 [2] 17
3.8 Estágio 4, redução das partições a um tamanho máximo.[2] 18
3.9 Estágio 5, obtenção do alinhamento ótimo.[2] 18
3.10 Limite de células adjacentes para matriz com *gap* de valor fixo 19
3.11 Cálculo se a célula é *prunable* ou não 19
3.12 Alinhamento local ótimo da comparação entre cromossomo 21 do ser humano em relação ao cromossomo 21 do chimpanzé, a área vermelha apresenta alto escore, a área cinza representa baixo escore e a área preto representa áreas com blocos que não foram processados (*pruned*) [3] 20
3.13 Arquitetura do MASA-CUDAlign MultiBP [4] 22
3.14 Passo a passo do Controller ao identificar falha [5]. 24

4.1 Componentes do SLURM [6] 27
4.2 Arquitetura do SLURM [7] 28
4.3 Representação de *jobs*, nós e partições [6] 28
4.4 Exemplo simples de script com configurações e pré-requisitos a ser executado pelo SLURM 33

5.1 Representação mínima das configurações das máquinas utilizadas. 35
5.2 Configurações e conteúdo do slurm.conf do projeto 37
5.3 Definição dos diretórios utilizados e parâmetros do arquivo de configuração FT-CUDAlign 40
5.4 Função responsável pelo reconhecimento dos IPs, GPUs e montar o *Array* de portas para conexão entre *sockets*. 40
5.5 Escrita do arquivo de configuração do FT-CUDAlign (Parâmetro "$1" recebe a pasta contendo o par de sequências a ser comparado). 41
5.6 *Loop* de execução dos *balancers*, com identificação do retorno para reconhecimento de falhas. 41
5.7 Execução do script pós definições. 42
5.8 Projeto do Script de Execução 42

6.1 Arquivo de configuração do FT-CUDAlign gerado durante a execução em duas GPUs e um *breakpoint*. 46
6.2 Arquivo de configuração do FT-CUDAlign gerado durante a execução em três GPUs e dois *breakpoints*. 46
6.3 Tempos obtidos a partir de seis execuções: três utilizando um *breakpoint* e três utilizando dois *breakpoints*, para duas GPUs e sequências de 5M 6.1. . 47
6.4 Tempos obtidos a partir de seis execuções: três utilizando um *breakpoint* e três utilizando dois *breakpoints*, para duas GPUs e sequências de 7M 6.1. . 48
6.5 Tempos obtidos a partir de seis execuções: três utilizando um *breakpoint* e três utilizando dois *breakpoints*, para duas GPUs e sequências de 10M. . . . 48
6.6 Tempos obtidos através da execução com e sem falha utilizando três GPUs. 49

Lista de Tabelas

6.1 Especificações das sequências comparadas. 45
6.2 Especificações das GPUs do LAICO. 45
6.3 *Overheads* obtidos em cenário sem falha. 49
6.4 *Overheads* obtidos em cenário com falha. 49

Capítulo 1

Introdução

A Bioinformática é uma área da biologia que utiliza técnicas computacionais para processar, analisar e interpretar dados biológicos [8]. Entre suas aplicações mais relevantes está a comparação de sequências biológicas, um procedimento fundamental para a biologia molecular e evolutiva, que tem como objetivo identificar semelhanças entre diferentes genes, proteınas, diferentes genomas e até entre diferentes organismos. Diversos organismos possuem milhares e até milhões de nucleotıdeos e, dessa forma, o processo de comparação pode ser extremamente lento, exigindo soluções de alto desempenho.

Existem dois tipos de algoritmos para a comparação de sequências: os exatos, que garantem sempre o melhor resultado porém possuem uma complexidade quadrática de tempo; e os heurísticos, que embora não garantam o melhor resultado, possuem um tempo de execução menor. O foco deste trabalho de graduação é com base em algoritmos exatos, que utilizam programação dinâmica.

Diversas ferramentas foram desenvolvidas para acelerar a comparação de sequências biológicas por meio de paralelizações baseadas na utilização de *multicores*, FPGAs (*Field Programmable Gate Array*) e principalmente de GPUs (*Graphics Processing Units*) como é o caso das ferramentas MASA-CUDAlign [4] e o FT-CUDAlign [5].

A ferramenta MASA-CUDAlign [9] [4] utiliza o do algoritmo de Gotoh [10] eimplementa a comparação de sequências biológicas em *cluster* de múltiplas GPUs com técnicas de *Block Pruning* para redução de blocos calculados, acelerando o processo.

O FT-CUDAlign [5] é uma versão do MASA-CUDAlign que introduz mecanismos de tolerância a falhas com técnicas preventivas e técnicas reativas. A ferramenta é capaz de identificar falhas durante a execução e permite o retorno das execuções a partir de *checkpoints* salvos periodicamente, sem a necessidade de reiniciar o processo ou de intervenção manual, desse modo, caso uma ou mais GPUs apresentem falhas do tipo *fail-stop*, o FT-CUDAlign consiga se readaptar e continuar sua execução com a quantidade de GPUs ainda disponıveis.

A execução em grandes *clusters* e supercomputadores que permitem a execução de diversas aplicações simultâneamente, necessita de um escalonador de aplicações para controle das partições com controle de recursos e filas de execução. O *Simple Linux Utility for Resources Management* (SLURM) é amplamente utilizado como o escalonador principal em diversos *clusters*, inclusive em diversos supercomputadores, como o El Capitan nos Estados Unidos, o Marenostrum 5 na Espanha e o Santos Dumont no Brasil [11]. É um software código aberto, tolerante a falhas e um escalonador escalável tanto para *clusters* homogêneos quanto heterogêneos. Além disso, sua arquitetura modular permite a instalação de *plugins*, possibilitando a extensão de funcionalidades para aprimorar tarefas como autenticação, controle de permissões nas filas, análise e monitoramento de *jobs*, entre outras.

1.1 Objetivos

O FT-CUDAlign não pôde ser ser executado via SLURM e isso não permite sua utilização em clusters de alto desempenho ou em supercomputadores. Para eliminar essa restrição, o objetivo primário do presente trabalho de graduação é projetar, implementar e avaliar a integração do FT-CUDAlign integrado ao gerenciador de recursos SLURM, de forma a permitir a execução do FT-CUDAlign em *clusters* de máquinas Linux com GPU de forma completa, incluindo sua tolerância a falhas.

O trabalho visa também mostrar a viabilidade do uso do SLURM como plataforma de orquestração para a utilização das GPUs do FT-CUDAlign, garantindo maior robustez e flexibilidade em comparação de sequências biológicas de grande porte.

1.2 Estrutura do Trabalho

O trabalho está organizado de modo que o Capítulo 2 apresenta o tópico de Comparação de Sequências Biológicas e explica algoritmos utilizados para realizar a comparação. O Capítulo 3 apresenta uma visão geral da ferramenta MASA-CUDAlign e de sua atualização FT-CUDAlign. O Cápitulo 4 apresenta o gerenciador de aplicações SLURM. O Capítulo 5, por sua vez, é o responsável por apresentar o projeto de integração entre o FT-CUDAlign e o SLURM, que é o foco desse TCC. O Capítulo 6 discute os resultados experimentais obtidos a fim de analisar se foram satisfatórios, enquanto o Capítulo 7 conclui o trabalho de graduação e propõe trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

A comparação de sequências biológicas é uma das principais operações na biologia molecular e evolutiva. Seu principal objetivo é identificar semelhanças e diferenças entre genes, proteínas ou genomas de diferentes organismos [12]. Essas comparações permitem compreender relações evolutivas, revelando, por exemplo, quão próximas ou distantes duas espécies estão em relação a um ancestral comum, de modo a possibilitar a identificação de genes conservados ao longo da evolução, bem como variações genéticas que explicam características específicas de cada espécie.

2.1 Conceitos Básicos

A comparação de sequências biológicas recebe como entrada duas sequências biológicas, produzindo um escore, que é uma métricade similaridade entre as sequências, e um alinhamento, onde as semelhanças e diferenças entre as sequências são ressaltadas. No caso de comparação de DNA há 4 possibilidades: Adenina (A), Citosina (C), Guanina (G), Timina (T) e caso o nucletídeo ainda não tenha sido identificado, é usado (N). Já para o RNA a Timina é substituída pela Uracila(U). A comparação das sequências se baseia em um sistema de pontuação, de modo que a maior pontuação significa maior semalhança entre as sequências de DNA/RNA, sendo usado um valor positivo para acertos (*matchs*) e valores negativos para erros (*mismatchs*) e espaços (*gaps*). A comparação também produz o alinhamento, onde cada nucleotídeo é posicionado de modo a produzir a melhor pontuação, considerando acertos, erros e espaços.

Existem dois tipos básicos de comparação: global e local. A comparação global consiste em alinhar as duas sequências completas de maneira que todos os caracteres das duas sequências façam parte do alinhamento. Já a comparação local consiste em alinhar trechos

das sequências, com o intuito de maximar a pontuação dos trechos e de encontrar genes semelhantes em espécies muito distintas, procurando genes evolutivamente dominantes [13].

Considere as sequências S_0 e S_1: $S_0 =$ `TATCCGTCC` e $S_1 =$ `AGCCGATG` das comparações global e local são apresentadas nas Figuras 2.1 e 2.2, respectivamente.

S_0:	T	A	T	C	C	G	T	C	C	
		|		|	|	|				
S_1:	–	A	G	C	C	G	A	T	G	
G:	-2	+1	-1	+1	+1	+1	-1	-1	-1	= -2

Figura 2.1: Comparação global entre S_0 e S_1, com $escore = -2$

S_0:	C	C	G		
	|	|	|		
S_1:	C	C	G		
L:	+1	+1	+1	=	+3

Figura 2.2: Comparação local entre S_0 e S_1, com $escore = +3$

2.2 Tipos de Algoritmos

Os algoritmos aplicados em problemas computacionais podem ser classificados em duas categorias: algoritmos exatos e algoritmos heurísticos [14]. Os algoritmos exatos são projetados para garantir a obtenção da solução ótima, ou seja, aquela que maximiza (ou minimiza) a função objetivo, independentemente da complexidade da entrada. No entanto, esses algoritmos frequentemente implicam em altos custos computacionais, tanto em termos de tempo de execução quanto de consumo de memória, o que pode inviabilizar sua aplicação em instâncias de grande porte.

Já algoritmos heurísticos usam abordagens heurísticas que não garantem que o resultado ótimo será produzido, porém são executados em menor tempo e geralmente consomem menos recursos. São mais eficientes e menos custosos do que algoritmos exatos, apresentando desempenho satisfatório na prática, especialmente quando aplicados a problemas de alta complexidade ou quando há restrição de recursos computacionais.

Tanto algoritmos exatos quanto heurísticos podem se beneficiar de técnicas de programação dinâmica, especialmente em contextos onde subproblemas se repetem e podem ser

reutilizados para compor soluções mais complexas. A programação dinâmica contribui significativamente para a eficiência desses métodos, otimizando o uso de tempo e memória.

No presente trabalho de graduação, o foco é em algoritmos exatos.

2.3 Algoritmos Exatos para Comparação de Sequências

Os algoritmos exatos para comparação de sequências recebem duas sequências S_0 e S_1 e como saída retornam o melhor alinhamento e a pontuação (escore de similaridade). Serão apresentadas inicialmente algoritmos que possuem *gaps* com valor constante, mas que podem ser extendidos para teorias mais complexas com *gaps* de valor variado.

Os algoritmos exatos levam em consideração:

1. As entradas S_0 e S_1 que são as sequências biológicas,
2. A comparação $s(x_i, y_i)$ entre os termos alinhados, sendo *match* ou *mismatch*,
3. O valor presente na célula (i, j) da matriz de programação dinâmica ($H_{(i,j)}$),
4. O valor "G" para *gap*.

Os algoritmos de programação dinâmica de Needleman-Wunsh, Smith-Waterman, Gotoh, Hirschberg e Myers-Miller serão apresentados nas seções 2.4, 2.5, 2.6, 2.7 e 2.8, respectivamente.

2.4 Algoritmo de Needleman-Wunsh (NW)

O primeiro algoritmo que será explicado é o algoritmo de Needleman-Wunsh[15]. Sendo ele um alinhamento global para comparação de Sequências Biológicas, foi um dos primeiros algoritmos utilizando programação dinâmica a resolver o problema da comparação de sequências, permitindo a utilização de gaps. O algoritmo se executa em 2 passos: o primeiro passo é o cálculo de uma matriz de programação dinâmica H, de tamanho $(m+1) \times (n+1)$ onde m é o tamanho do sequência S_0 e n é o tamanho da sequência S_1. Para cada índice é escrita a melhor pontuação possível para aquela célula, podendo ser calculada recursivamente. Para cada cálculo de $H_{(i,j)}$ há 3 possibilidades: utilizar o valor da comparação anterior e comparar os valores atuais $H_{(i-1,j-1)} + s_{(x_i,y_i)}$, a possibilidade de um gap na sequência S_1, $H_{(i.j-1)} + G$ ou um gap proveniente da sequência S_0 $H_{(i,j-1)} + d$. O valor $H_{(i,j)}$ é o máximo encontrado entre as 3 possibilidades.

A equação 2.1 ilustra a equação de recorrência do algoritmo NW.

$$H(i,j) = \max \begin{cases} H(i-1, j-1) + s(x_i, y_j) \\ H(i-1, j) + G \\ H(i, j-1) + G \end{cases} \tag{2.1}$$

A matriz de similaridade H tem seu primeiro termo $H_{0,0}$ preenchido com zero e segue até que todos os termos estejam preenchidos, de modo que a última célula $H_{(m+1,n+1)}$ é a célula que possui a melhor pontuação global para o sistema, ou seja, é a célula que contem o escore máximo do alinhamento global.

Já o segundo passo consiste em percorrer a matriz H de trás para a frente, de maneira a recuperar o alinhamento a partir da última célula $H_{(m,n)}$. O caminho ótimo é obtido por *backtracking* a partir dessa célula, seguindo o caminho que levou à pontuação máxima. Com o fim do passo a passo a complexidade resultante é de $O(mn)$ em tempo e espaço.

A figura 2.3 apresenta a matriz H do algoritmo NW para as entradas $S_0 = ACGTAGCT$ e $S_1 = AGTAGTAT$ com valores de *match*, *mismatch* e *gap* como $+1, -1$ e -2. Já a figura 2.4 apresenta o alinhamento produzido pelo algoritmo.

	-	A	C	G	T	A	G	C	T
-	0	-2	-4	-6	-8	-10	-12	-14	-16
A	-2	1	-1	-3	-5	-7	-9	-11	-13
G	-4	-1	0	0	-2	-4	-6	-8	-10
T	-6	-3	-2	-1	1	-1	-3	-5	-7
A	-8	-5	-4	-3	-1	2	0	-2	-4
G	-10	-7	-6	-3	-3	0	3	1	-1
T	-12	-9	-8	-5	-2	-2	1	2	2
A	-14	-11	-10	-7	-4	-1	-1	0	1
T	-16	-13	-12	-9	-6	-3	-2	-2	1

Figura 2.3: Matriz de Needleman-Wunsh

S_0:	A	C	G	T	A	–	G	C	T		
	|		|	|	|		|	|			
S_1:	A	–	G	T	A	C	G	C	A		
G:	+1	-2	+1	+1	+1	-2	+1	+1	-1	=	+1

Figura 2.4: Alinhamento de Needleman-Wunsh

2.5 Algoritmo de Smith-Waterman (SW)

O segundo algoritmo a ser estudado é o de Smith-Waterman [16], sendo ele uma modificação do algoritmo de Needleman-Wunsh, com a mesma complexidade de tempo e memória ($O(mn)$).

O algoritmo SW busca o melhor alinhamento local entre as sequências e, portanto, descarta alinhamentos com escores negativos. Para tanto, foram feitas três modificações do algoritmo NW. Primeiramente, a primeira linha e a primeira coluna da matriz H foram preenchidas com zeros. A segunda mudança é a utilização de zeros na equação de recorrência, pois não sao admitidos valores menores que zero. A equação 2.2 é a equação de recorrência do algoritmo SW.

$$H(i,j) = \max \begin{cases} 0 \\ H(i-1, j-1) + s(x_i, y_j) \\ H(i-1, j) + G \\ H(i, j-1) + G \end{cases} \tag{2.2}$$

	-	A	C	G	T	A	G	C	T
-	0	0	0	0	0	0	0	0	0
A	0	1	0	0	0	1	0	0	0
G	0	0	0	1	0	0	2	0	0
T	0	0	0	0	2	0	0	1	1
A	0	1	0	0	0	3	1	0	0
G	0	0	0	1	0	1	4	2	0
T	0	0	0	0	2	0	2	3	3
A	0	1	0	0	0	3	1	1	2
T	0	0	0	0	1	1	2	0	2

Figura 2.5: Matriz de Smith-Waterman

Nesse algoritmo, o alinhamento pode terminar em qualquer célula da matriz H e para a recuperação do alinhamento ótimo é utilizado o *backtracking* começando na célula de maior valor e terminando ao encontrar o primeiro zero. Essa é a terceira diferença em relação ao algoritmo de NW.

A figura 2.5 apresenta a matriz H calculada com o algoritmo SW para as entradas $S_0 = ACGTAGCT$ e $S_1 = AGTAGTAT$ com valores de *match*, *mismatch* e *gap* como $+1, -1$ e -2 e a Figura 2.6 apresenta o alinhamento produzido.

S_0:	G	T	A	G		
	|	|	|	|		
S_1:	G	T	A	G		
L:	+1	+1	+1	+1	=	+4

Figura 2.6: Alinhamento local ótimo para o algoritmo SW

2.6 Algoritmo de Gotoh

Os algoritmos de NW e SW possuem valor fixo para *gaps*. O grande diferencial do algoritmo de Gotoh[10] é o uso de uma função para a pontuação de *gaps*, o modelo *affine gap*. Para isso são necessários dois valores para pontuar uma sequência de *gaps*: a pontuação do primeiro *gap* (*Gfirst*) e a pontuação dos *gaps* subsequentes (*Gextend*) [12].

Uma vez que os valores atribuídos aos *gaps* possuem valor variável, a equação necessita passar por algumas alterações a fim de aceitar tais diferenças. Portanto, teremos gap_{first} e gap_{ext} como valores *gap*. Para o cálculo, são necessários duas matrizes de programação dinâmica adicionais (E e F). As equações 2.3, 2.4 e 2.5 são as equações de recorrência de Gotoh, para o caso da comparação local.

Assim, como o algoritmo de Smith-Waterman, a primeira linha e coluna da matriz H são inicializadas com zero. As matrizes E e F são utilizadas para calcular um conjunto de *gaps* nas sequências S_0 e S_1, respectivamente.

$$H(i,j) = \max \begin{cases} 0 \\ H(i-1,j-1) + s(x_i, y_j) \\ E(i,j) \\ F(i,j) \end{cases} \tag{2.3}$$

$$F(i,j) = \max \begin{cases} H(i-1,j) + G_{first} \\ F(i-1,j) + G_{ext} \end{cases} \tag{2.4}$$

$$E(i,j) = \max \begin{cases} E(i,j-1) + G_{ext} \\ H(i,j-1) + G_{first} \end{cases} \tag{2.5}$$

H	-	A	C	G	C	G	T	A	G
-	–	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
A	–	-4	-4	-4	-4	-4	-4	-4	-4
C	–	-2	-4	-4	-4	-4	-4	-2	-4
G	–	-3	0	-4	-2	-4	-4	-3	-4
A	–	-4	-1	2	-2	0	-4	-4	-2
A	–	-2	-2	1	0	-1	-2	-2	-3
C	–	-2	-3	0	-1	-2	-3	0	-4
G	–	-3	0	-1	-1	-3	-4	-1	-2
T	–	-4	-1	2	-2	1	-3	-2	-2
A	–	-4	-2	1	0	0	3	1	2

(a) Matriz F

F	-	A	C	G	C	G	T	A	G
-	0	0	0	0	0	0	0	0	0
A	0	2	0	0	0	0	0	2	0
C	0	0	4	0	2	0	0	0	0
G	0	0	0	6	2	4	0	0	2
A	0	2	0	2	4	0	2	2	0
A	0	2	0	1	0	2	0	4	0
C	0	0	4	0	3	0	0	0	2
G	0	0	0	6	2	5	1	0	2
T	0	0	0	2	4	1	7	3	2
A	0	2	0	1	0	2	3	9	5

(b) Matriz H

J	-	A	C	G	C	G	T	A	G
-	–	–	–	–	–	–	–	–	–
A	$-\infty$	-4	-2	-3	-4	-4	-4	-4	-2
C	$-\infty$	-4	-4	0	-1	-2	-3	-4	-4
G	$-\infty$	-4	-4	-4	2	1	0	-1	-2
A	$-\infty$	-4	-2	-3	-2	0	-1	-2	-2
A	$-\infty$	-4	-2	-3	-3	-4	-2	-3	0
C	$-\infty$	-4	-4	0	-1	-1	-2	-3	-4
G	$-\infty$	-4	-4	-4	2	1	1	0	-1
T	$-\infty$	-4	-4	-4	-2	0	-1	3	2
A	$-\infty$	-4	-2	-3	-3	-4	-2	-1	5

(c) Matriz E

Figura 2.7: Matrizes do algoritmo de Gotoh: F, H e E

O algoritmo de Gotoh é eficiente para encontrar regiões localmente semelhantes em sequências que possuem trechos geneticamente distintos, pois penaliza mais a abertura de gaps do que sua extensão. Isso torna o alinhamento mais favorável à identificação de blocos conservados entre regiões divergentes, permitindo detectar conjuntos de subsequências semelhantes mesmo em meio a grandes diferenças [10].

A complexidade de tempo do algoritmo é $(O(mn))$, semelhante ao encontrado nos algoritmos de Needleman-Wunsh e Smith-Waterman. Assim como sua complexidade de espaço, que embora ocupe $3(mn)$ de espaço, ainda possui complexidade $(O(mn))$, semelhante aos algoritmos de Smith-Waterman e Needleman-Wunsh. A Figura 2.7 apresenta as matrizes de Gotoh para as sequências $S_0 = ACGCGTAG$ e $S_1 = ACGAACGTA$. Para *match, mismatch,* gap_{first} *(primeiro gap) e* gap_{ext} *(demais gaps)* foram usados, respectivamente $2, -3, -4$ e -1.

A figura 2.8 apresenta o alinhamento produzido pelo algoritmo de Gotoh.

S_0:	A	C	G	–	–	C	G	T	A		
	|	|	|			|	|	|	|		
S_1:	A	C	G	A	A	C	G	T	A		
L:	+2	+2	+2	-4	-1	+2	+2	+2	+2	=	+9

Figura 2.8: Alinhamento local ótimo para o algoritmo Gotoh

2.7 Algoritmo de Hirschberg

O algoritmo de Hirschberg[17], diferente dos demais vistos até o momento, permite obter o alinhamento global ótimo entre duas sequências com uma complexidade de espaço em $(O(n))$, mantendo a complexidade de tempo em $(O(mn))$. O algoritmo proposto se baseia na simplificação do problema, particionando as sequências de modo que a concate-

nação do resultado ótimo da comparação entre as subsequências, formem um alinhamento ótimo.

Para isso é necessário dividir a sequência S_0 ao meio e realizar o alinhamento global ótimo considerando toda a sequência S_1, de modo a obter o vetor C para a primeira metade de S_0 e o vetor C' para a segunda metade do vetor S_0. Após esse passo, deve encontrar a coluna que maximiza a soma $C + C'$ e ao descobrir essa coluna j, encontra-se um ponto que pertence ao alinhamento ótimo(*crosspoint*), de modo que esse é o fim da primeira iteração.

O procedimento é aplicado recursivamente às subsequências. A proxima iteração ocorrerá utilizando as duas partições do vetor S_1, sendo a primeira partição a primeira metade de S_0 que será redividida em duas partes com a subsequência de S_1 até a coluna j encontrada previamente. Na segunda partição, o processo se repete, utilizando a segunda metade de S_0 com a subsequência de S_1 que vai da coluna $j + 1$ até o fim da sequência. A recursão prossegue até que as subsequências reduzam-se a um ou nenhum elemento.

Como a matriz completa não é mantida na memória durante toda a execução do algoritmo e não há utilização de *backtracking* tradicional, o alinhamento é reconstruído por meio da concatenação dos resultados obtidos nas chamadas recursivas, a partir dos casos base.

2.8 Algoritmo de Myers-Miller(MM)

O algoritmo de Myers-Miller[18] se baseia na união dos conceitos dos algoritmos de Hirschberg e Gotoh, ou seja, será calculado *affine gap* em espaço linear. O alinhamento é reconstruído a partir dos resultados das chamadas recursivas, iniciando nos casos base.

Diferente de Hirschberg, o algoritmo de Myers-Miller utiliza o modelo de *affine gaps* implementado por Gotoh, para determinação da pontuação e do alinhamento global, utilizados na comparação entre as sequências dadas na entrada. Porém se baseia no mesmo conceito do *crosspoint* de Hirschberg para definição do alinhamento ótimo em tempo linear.

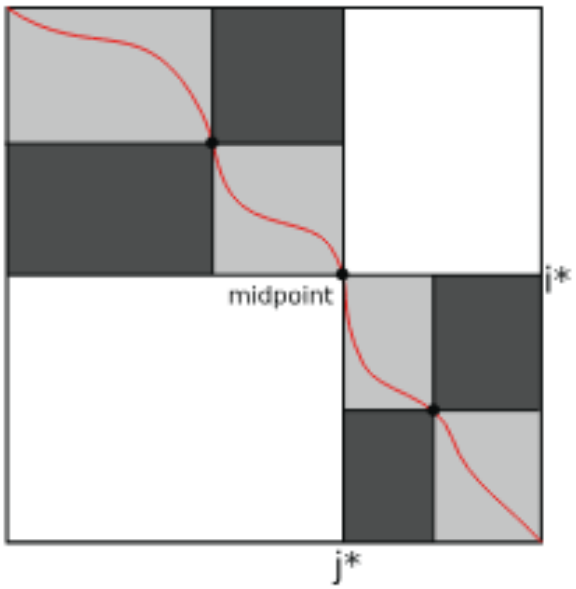


Figura 2.9: Representação do ponto médio ótimo(*crosspoint*) na matriz de similaridade Myers-Miller [1]

O algoritmo de Myers-Miller tornou-se útil em contextos de comparação de genomas ou proteínas entre espécies evolutivamente distantes, onde as sequências podem ter tamanhos muito grandes e diferenças significativas. Embora o modelo de Gotoh seja utilizado para tal, sua demanda por espaço de memória torna-se um problema em cenários de larga escala, enquanto o algoritmo de Myers-Miller necessita de menor espaco.

Capítulo 3

A ferramenta MASA-CUDAlign

A ferramenta CUDAlign vem sendo desenvolvida desde 2010 e passou por diversas versões. Em suas primeiras versões, executava-se somente em GPUs NVIDIA. Em 2016, foi proposta a arquitetura MASA, que dividiu o código em duas partes: dependente de plataforma e independente de plataforma. A partir daí, todas as versões de CUDAlign são chamadas MASA-CUDAlign.

O MASA (*Multi-Platform Architecture for Sequence Aligners*) é um *framework* de infraestrutura flexível e customizável, que permite ao usuário três variantes de alinhamento (local, semi-global e global), em múltiplas instalações de software e hardware [9], propondo formas genéricas de *block pruning* e múltiplas formas de se implementar o paralelismo, de modo a ser aplicável em três diferentes plataformas de hardware: CPU, GPU e Intel Xeon Phi, e três diferentes plataformas de software: OmpSs [19], OpenMP e CUDA.

Na seção 3.1 explicamos como o MASA-CUDAlign é executado em uma GPU e na seção 3.2 discorremos sobre a execução do MASA-CUDAlign em diversas GPUs.

3.1 MASA-CUDAlign em uma GPU

3.1.1 CUDAlign 1.0

No CUDAlign 1.0 [2], a equação de recorrência de Gotoh (Equação 2.6) é calculada em paralelo por diversas *threads* que se executam em uma única GPU com arquitetura CUDA da NVIDIA. Seu resultado retorna a melhor pontuação(*score*) da comparação de duas sequências S_0 e S_1.

A divisão de trabalho entre as *threads* usa o conceito de *wavefront* em blocos com dois tipos de paralelismo: o primeiro é o paralelismo externo e o segundo é paralelismo interno nos blocos.

O paralelismo externo é definido como o agrupamento das células para formar os blocos de tamanho $(R \times C)$, em que os valores de R e C são definidos conforme o número de blocos concorrentes e o número de *threads* por bloco, sendo o número de linhas $R = \alpha T$ e o número de Colunas $C = \frac{n}{B}$. Além disso, tem-se α, que é uma constante inteira que representa o número de linhas que cada *thread* processa, T é o número de *threads*, n o tamanho da sequência S_1 e B é o número de blocos por linha.

$G_{0,0}$	$G_{0,1}$	$G_{0,2}$
$G_{1,0}$	$G_{1,1}$	$G_{1,2}$
$G_{2,0}$	$G_{2,1}$	$G_{2,2}$
$G_{3,0}$	$G_{3,1}$	$G_{3,2}$

Figura 3.1: Divisão da matriz em blocos. Em azul está a diagonal externa D_3.

Feito isso, a técnica de *wavefront* é aplicada nas diagonais externas, chamando a execução paralela de cada diagonal externa usando o algoritmo de Gotoh, após o término do cálculo da diagonal anterior. Essas diagonais variam de tamanho 1 a B, conforme ilustrado na Figura 3.1.

O paralelismo interno ocorre dentro dos blocos, de maneira semelhante ao paralelismo externo. As células são aplicadas em diagonais e cada *thread* é responsável por calcular as linhas αk a $\alpha(k+1)-1$ do bloco, na direção da esquerda para a direita. Seguindo essa lógica, todas as *threads* devem calcular a mesma diagonal, ao mesmo tempo, evitando assim que falte algum dado para alguma das *threads* processar. Ou seja, as *threads* devem ser sincronizadas ao final do cálculo de cada diagonal. O processamento das diagonais internas está ilustrado na Figura 3.2.

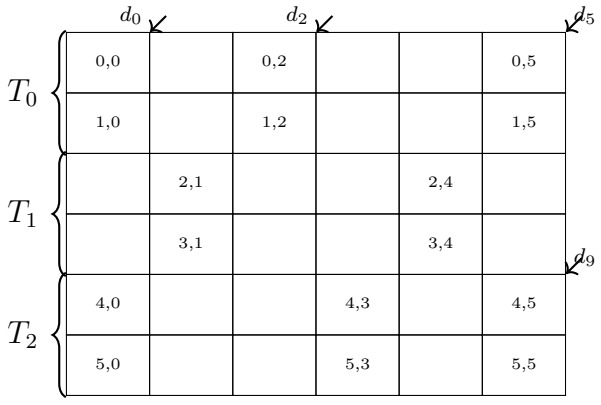


Figura 3.2: Bloco Retangular representando as diagonais e o paralelismo interno.

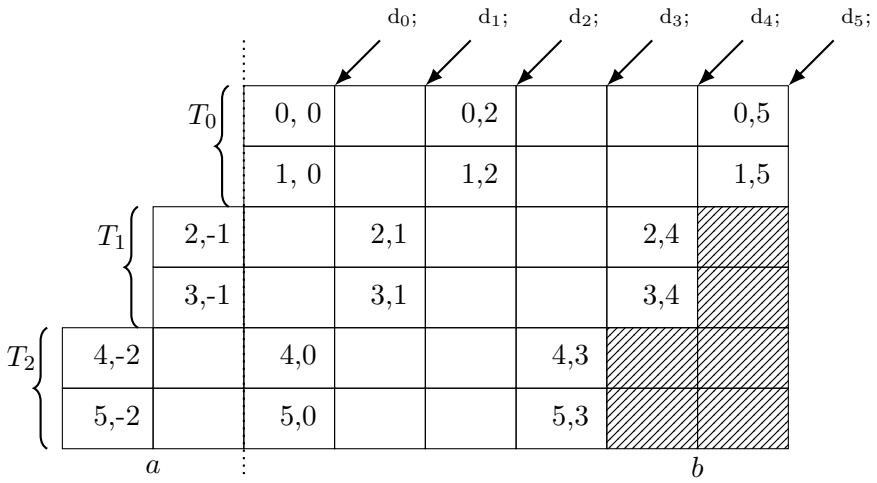


Figura 3.3: Bloco não retangular, células com colunas negativas se referem a células delegadas de blocos já processados.

Seguindo uma análise do bloco na Figura 3.2, apenas as diagonais d_4, d_5 e d_6, possuem máximo paralelismo, de modo que em outras diagonais, como d_0, d_1, d_9 e d_{10}, apenas uma *thread* está sendo utilizada, reduzindo, assim, o nível de paralelismo do bloco e perdendo eficiência. De maneira a explorar melhor o paralelismo, foi proposta a *delegação de células* [9], onde o bloco é processado até a última célula de paralelismo máximo. As células pendentes devem ser processadas pelo próximo bloco, a fim de aumentar o tempo de paralelismo máximo (Figura 3.3).

No entanto para que isso seja possível, a parte delegada do bloco mais a direita de cada diagonal externa deve ser delegada para que o bloco mais a esquerda da próxima diagonal a processe. O mesmo é válido para o final da matriz, de um jeito que novos blocos devem ser criados a fim processar as delegações deixada, aumentando o número de diagonais externas necessárias para processar tais blocos (Figura 3.4).

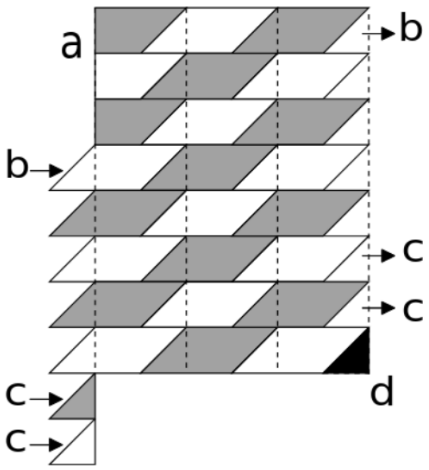


Figura 3.4: Delegação de células entre blocos[2]

3.1.2 CUDAlign 2.0

O CUDAlign 2.0 [3] obtém o alinhamento ótimo resultante da comparação de duas sequências biologicas com uma quantidade restrita de memória, utilizando estratégias de Myers-Miller(Seção 2.8) de forma a obter o alinhamento local ótimo em espaço linear. Foram propostas as otimizações *matching* baseado em objetivo, execução ortogonal e a divisão balanceada, que trazem maior eficiência e reduzem a quantidade de reprocessamento da matriz. O CUDAlign 2.0 se executa em cinco estágios.

O primeiro estágio se baseia na obtenção da pontuação máxima e a coordenada dessa célula, baseando-se no algoritmo apresentado CUDAlign 1.0 (Seção 3.1.1), com a modificação de que algumas linhas da matriz, chamadas linhas especiais, são salvas em disco para serem usadas no estágio 2. As linhas especiais são obtidas do barramento horizontal em intervalos fixos de x blocos. Desse modo, apenas linhas múltiplas de R podem ser linhas especiais.

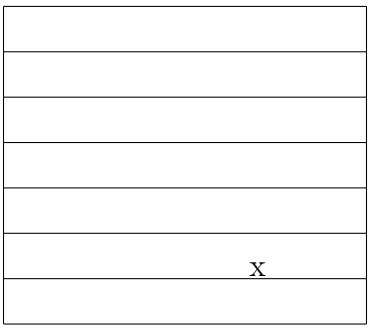


Figura 3.5: Estágio 1 do CUDAlign 2.0 [2]

O segundo estágio consiste no *backtracing* parcial de modo a encontrar em qual ponto o alinhamento ótimo cruza as linhas especiais, de modo a encontrar também o ponto de início do alinhamento ótimo. Para isso o procedimento executado é no formato de alinhamento semiglobal e o *matching* baseado em objetivo, que consiste em, a partir da célula de melhor pontuação, encontrar o escore-alvo para cada linha especial. Quando encontra, essa execução é paralisada e começa a próxima a partir desse valor alvo, para o valor alvo da próxima linha especial, evitando computação desnecessária e melhorando a otimazação de tempo do algoritmo.

O paralelismo do estágio 2 não pode ser feito de forma de blocos horizontais, iguais ao estágio 1, para isso, ele deve ser feito de forma vertical e para isso foi proposta a segunda otimização do algoritmo, a execução ortogonal.

Essa execução consiste na leitura de blocos, no formato ortogonal, verticalmente, fazendo com que a partir do escore já processado, encontre o escore-alvo na linha especial, ou encontre o valor de início do alinhamento.

Assim como o estágio 1, o estágio 2 salva algumas colunas em disco para que possam ser usadas no estágio 3 a fim de descobrir mais pontos do alinhamento.

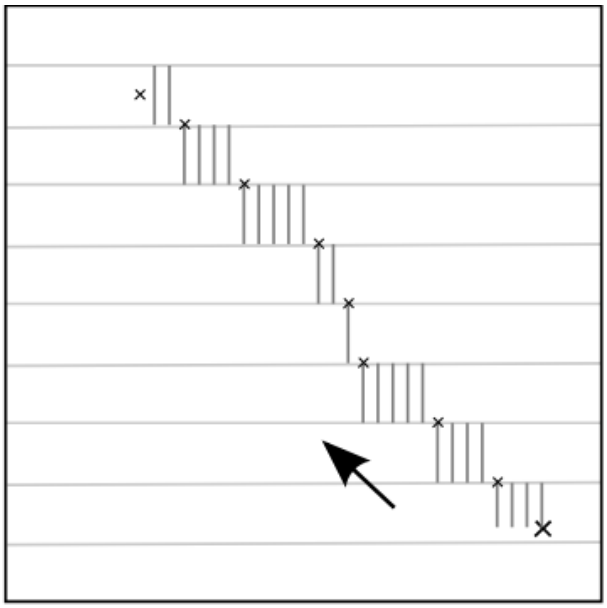


Figura 3.6: Estágio 2 do CUDAlign 2.0 [2]

O estágio 3 tem o objetivo de aumentar a quantidade de pontos que cruzam o alinhamento ótimo. Diferente do estágio 2, esse estágio já possui as coordenadas de início e fim, sendo utilizado para determinar pontos do alinhamento ótimo nas colunas salvas no passo anterior.

Uma vez que não há intersecção e nem dependência entre as partições, é possível que esse passo possa ser realizado de forma paralela sem sincronia ou até mesmo por GPUs distintas.

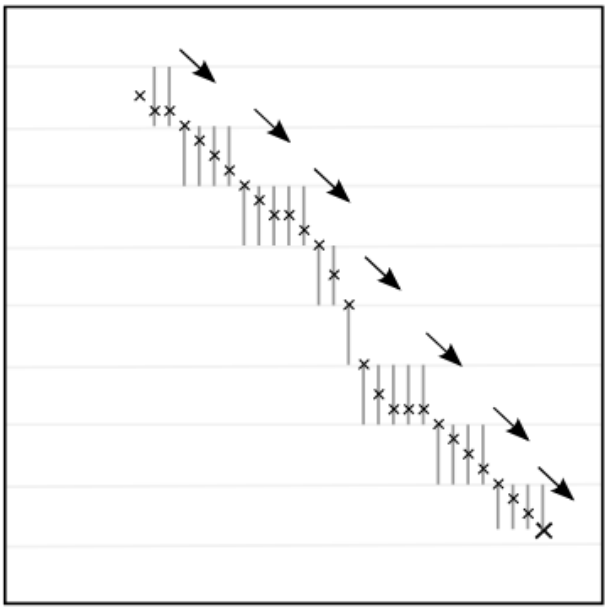


Figura 3.7: Estágio 3 do CUDAlign 2.0 [2]

O estágio 4, assim como o passo anterior, tem o intuito de de reduzir o tamanho de cada partição entre coordenadas sucessivas. Utiliza-se o algoritmo de Myers-Miller (Seção 2.5), para que de forma iterativa, reduza o tamanho das partições até um tamanho máximo definido de forma empírica. Da mesma forma que o estágio 3, não há intersecção ou dependência entre as partições, podendo ser feita com várias *threads* de forma paralela e sem sincronia.

No estágio 4 foi implementada a terceira otimização do CUDAlign 2.0, a divisão balanceada, que consiste que a matriz processada pelo algoritmo Myers-Miller possa ser dividida tanto em linha quanto em coluna pela metade, pegando o ponto central, de modo a ser definido a depender de qual coordenada é maior, evitando assim que partições muito desproporcionais se mantenham por muitas iterações. De forma concorrente a essa otimização, ocorre também a execução ortogonal dessas partições, sendo valor alvo, a média dos valores nas coordenadas de início e fim da partição, reduzindo o processamento necessário para a definição das próximas iterações.

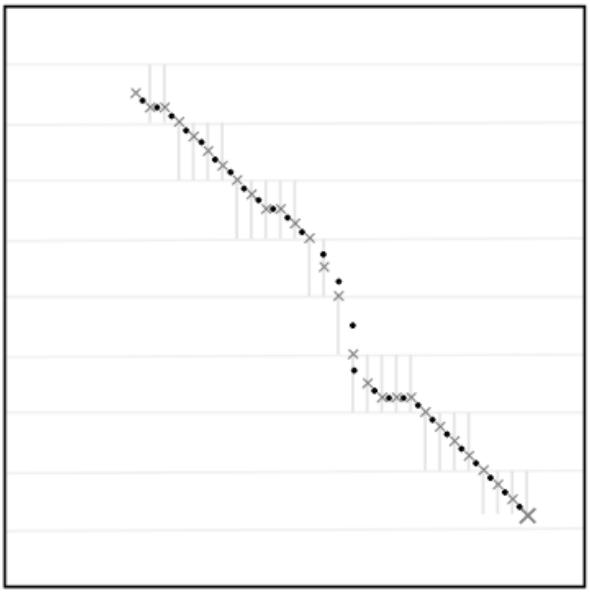


Figura 3.8: Estágio 4, redução das partições a um tamanho máximo.[2]

No estágio 5 é obtido o alinhamento ótimo total, executando o algoritmo de alinhamento global de Needleman-Wunsch (Seção 2.4), onde os alinhamentos são concatenados e a união desses alinhamentos resulta no alinhamento ótimo completo. Após a redução das partições no estágio 4 e a necessidade de um tamanho máximo constante, o estágio 5 ganha bastante eficiência e o alinhamento ótimo pode ser obtido em memória linear.



Figura 3.9: Estágio 5, obtenção do alinhamento ótimo.[2]

3.1.3 CUDAlign 2.1

O CUDAlign 2.1 [3] propôs a estratégia de *Block Pruning(BP)* que permite que blocos da matriz não sejam calculados (descartados) se com certeza não levarem ao alinhamento ótimo. O descarte ocorre caso a maior pontuação possível para aquele bloco, seja menor que a melhor pontuação já cálculada em blocos anteriores, de modo a ser impossível que o alinhamento ótimo passe por aquele bloco.

A matriz de programação dinâmica possui uma dependência dos valores das células adjacentes, possuindo um valor máximo e um valor minímo. Por meio dessa propriedade, é possível obter os intervalos que cada célula pode ou não obter, e caso possua o caminho de *traceback* que conecta duas células, há uma possibilidade ainda menor de valores possíveis. Assim é possível identificar, e ignorar células da matriz, antes mesmo de serem calculadas (Figura 3.10)

$H_{0,j-1} = (1-j)G$	$H_{0,j} \geq -jG$
$H_{1,j-1} \geq (2-j)G - mi$	$H_{1,j} \leq (1-j)G + ma$

Figura 3.10: Limite de células adjacentes para matriz com *gap* de valor fixo

Desse modo é possível calcular o maior valor que aquela célula pode gerar, de modo que considerando a célula $H_{i,j} = 0$, o maior valor possível será se tudo para frente for *match*, portanto o valor máximo desse alinhamento será $min(\Delta i, \Delta j) * ma$, sendo ma o valor de *match*. Caso esse valor seja menor que o limite inferior(*bound*), definido empiricamente ou através da definição de algum método que sobreescreva o *bound* durante a execução, utilizando a projeção mínima da célula de escore máximo já encontrado, essa célula e todo o seu alinhamento será considerado *prunable* e poderá ser desconsiderada. [3]

$$H_{i,j}^{max} \leq bound$$

Figura 3.11: Cálculo se a célula é *prunable* ou não

No entanto, uma vez que o cálculo a cada célula gera um grande *overhead*, a fórmula foi adaptada para o uso em blocos, como proposto pelo CUDAlign 2.1.

No formato proposto, para sequências semelhantes, o *Block Pruning* acelera o processamento e evita que diversas áreas que não contribuem para solução do alinhamento ótimo da comparação sejam processadas. A Figura 3.12 representa um dos casos que o *Block Pruning* se mostrou eficaz.

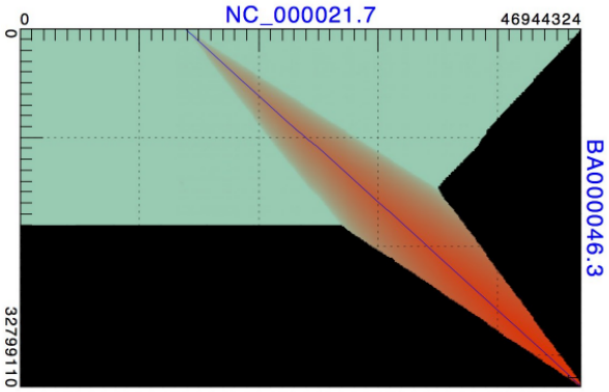


Figura 3.12: Alinhamento local ótimo da comparação entre cromossomo 21 do ser humano em relação ao cromossomo 21 do chimpanzé, a área vermelha apresenta alto escore, a área cinza representa baixo escore e a área preto representa áreas com blocos que não foram processados (*pruned*) [3]

3.2 CUDAlign com múltiplas GPUs

3.2.1 CUDAlign 3.0

O CUDAlign 3.0 [3], assim como a primeira versão, produz apenas a pontuação ótima e não o alinhamento, no entanto, possui em sua arquitetura, algo não visto nas implementações já comentadas, que é a implementação Multi-GPU, em que mais de uma GPU será usada para o cálculo de escore máximo nas comparações de sequências biológicas. O uso de diversas GPUs consiste em separar intervalos de colunas para cada GPU, de modo que em casos de GPUs homogêneas, cada intervalo terá igual tamanho e em casos de GPUs heterogêneas, o intervalo pode variar a depender da proporção de sua capacidade de processamento.

As GPUs necessitam de células calculadas por GPUs vizinhas e constantemente utilizam *buffers* para a troca de dados. Cada GPU é dividida em três *threads*, a *thread* gerente e duas de comunicação, sendo a primeira responsável pelo gerenciamento da execução GPU e as *threads* de comunicação são responsáveis pela comunicação entre as GPUs.

A comunicação é baseada em *buffers* circulares associados às *threads* de comunicação e podem ser usados com *buffer* de entrada ou saída de dados. Esses *buffers* são responsáveis por reduzir a latência e evitar que as GPUs necessitem esperar que a outra receba o dado para poderem continuar sua execução, aumentando o paralelismo associado. Casos os buffers de entrada ou de saída estejam cheios ou vazios, gera atraso na comparação e significa que o *wavefront* deve ser balanceado para que nenhuma GPU fique bloqueada esperando dados e perdendo eficiência.

3.2.2 CUDAlign 4.0

O grande diferencial do CUDAlign 4.0 [3] foi a implementação da obtenção do alinhamento ótimo com múltiplas GPUs. Para que a recuperação pudesse ser feita de maneira eficiente foi proposta a estratégia de *Incremental Speculative Traceback(IST)* que permite que GPUs ociosas possam especular valores de *crosspoints* e, caso acertem, recuperar parte do alinhamento com base nesses valores.

Usando como base o passo a passo proposto no CUDAlign 2.0 (Seção 3.1.2), foram feitas modificações para que fosse possível a execução em múltiplas GPUs. No primeiro passo, a modificação principal foram as linhas especiais serem salvas em diretórios exclusivos para cada GPU e uma implementação que permite que salvar as linhas em memória RAM para reduzir a quantidade de operações de entrada/saída do disco.

Para o segundo passo, foi proposto o *Pipelined Traceback(PT)*, permitindo a criação de um pipeline que realiza os passos 2 a 4 nas diferentes partições de forma paralela e a cada *crosspoint*, a GPU responsável repassa as coordenadas à GPU a sua esquerda para que essa, possa calcular o próximo crosspoint, gerando dependências. Agregado a isso, após a resolução das dependências, os passos 2, 3 e 4 podem ser calculados de forma independente em pipeline.

Já o estágio 5, por ser rápido e exigir pouca necessidade computacional, é realizado apenas pela primeira GPU recuperando os *crosspoints* já calculados pelas demais GPUs. No entanto, para evitar ociosidade das GPUs, o *Incremental Speculative Traceback* foi uma técnica usada no estágio 2 que consiste em cada GPU calcular *crosspoints* locais definidos na sua partição, enquanto as últimas GPUs terminam os cálculos do estágio 1.

Desse modo, caso os *crosspoints* coincidam com o alinhamento definido pelo *crosspoint* encontrado pela última GPU, não haverá a necessidade de reprocessamento e os *crosspoint* serão imediatamente repassados para a GPU a sua esquerda. Caso não coincidam, a GPU irá calcular seus *crosspoints* a partir do *crosspoint* correto, enviado pela GPU à sua direita.

Os *crosspoints* especulativos podem ser passados para as GPUs a esquerda, aumentando ainda mais, em caso de acerto, o grau de especulação e reduzindo ainda mais o tempo de processamento dos dados no estágio 2 e agilizando o *traceback*.

3.2.3 MASA-CUDAlign-MultiBP

O MASA-CUDAlign-MultiBP é a última versão do MASA-CUDAlign e tem o intuito de utilizar o *Block Pruning* desenvolvido no CUDAlign 2.1 (Seção 3.1.3) com múltiplas GPUs, tendo como base o CUDAlign 4.0 (Seção 3.2.2).

Para fazer o *pruning* com várias GPUs, a estratégia adotada foi o envio periódico do melhor escore obtido em cada GPU para a GPU diretamente a sua direita, de modo

circular, de modo que a primeira GPU receba de volta o valor da última GPU. O formato de conexão é feito através de buffers de comunicação, semelhantes aos apresentados no CUDAlign 3.0.

De forma concorrente, o MASA-CUDAlign MultiBP[4], foi projetado para trabalhar tanto em ambientes homogêneos como ambientes heterogêneos, ajustando as partições de colunas para cada GPU de forma a maximizar o processamento.

Para esse algoritmo foram propostas dois formatos para o *Block Pruning*, o *Static-MultiBP* e o *Dynamic-MultiBP* tendo como diferença principal a atualização da carga de trabalho por cada GPU de forma dinâmica na implementação do *Dynamic-MultiBP*. Para tanto, foram definidos ciclos com *breakpoint* e ao fim de cada execução, a proporção de colunas de cada GPU é recalculada, criando um balanceamento de carga.

Assim, se tornou necessário a criação de um módulo *Decision Maker* para definir em qual formato o algoritmo deve rodar, podendo ser definido pelo usuário ou uma análise interna de similaridade das matrizes, e através de um limiar pré-definido. Se a similaridade for menor que o limiar, o método escolhido é o *Static-MultiBP* e, caso seja maior ou igual ao limiar, o método passa a ser o *Dynamic-MultiBP*.

A Figura 3.13 ilustra o MASA-CUDAlign MultiBP.

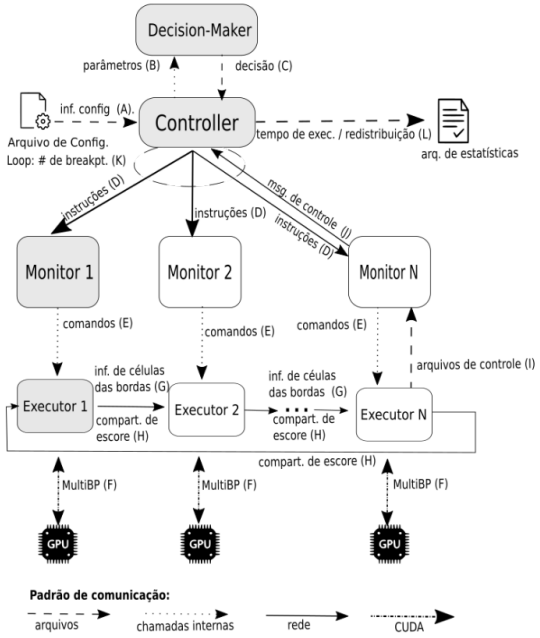


Figura 3.13: Arquitetura do MASA-CUDAlign MultiBP [4]

3.3 FT-CUDAlign

O FT-CUDAlign (*Fault Tolerance CUDAlign*) [5] é uma proposta de atualização do MASA-CUDAlign Dynamic-MultiBP, que implementa um mecanismo de tolerância a falhas, salvando certas colunas em disco e permitindo que essas colunas sejam consideradas como checkpoints para retorno em casos de falha.

O tratamento de falhas pode ser feito com duas categorias distintas, as técnicas proativas e as técnicas reativas. A primeira consiste em detectar e prevenir erros, antes que eles possam ocorrer, enquanto a segunda categoria consiste em tratar erros e lidar com eles após a falha. O FT-CUDAlign utiliza a técnica reativa.

3.3.1 Checkpoint

Os *checkpoints* ou pontos de salvamento consistem no salvamento do contexto de um processo em disco e, no caso de falha, a execução do processo é retomada a partir do contexto salvo. O FT-CUDAlign usou a técnica específica de aplicação, onde não se salva o contexto completo mas somente as informações necessárias para a retomada da execução, reduzindo o tamanho do checkpoint. Sendo assim, no FT-CUDAlign é salva somente a coluna que corresponde ao *breakpoint*.

Em casos de erro, não é necessário reprocessar toda a matriz, pois o reprocessamento é feito a partir dessa última coluna salva. Para determinar se o *checkpoint* é válido, deve-se analisar o estado da execução e se foi validado, sendo considerado um estado consistente e livre de erros.

Há diversos tipos de *checkpoints*, dentre eles o não coordenado, que permite que vários *checkpoints* ocorram de maneira independente para diferentes processos, o que reduz a necessidade de sincronização e evita *overhead*. No entanto esse método pode entrar em efeito cascata e com diversos erros, pode retornar a execução para o ponto de início do código, além da criação de *checkpoints* muito próximos, o que reduziria o paralelismo e, uma vez que escritas em disco são instruções demoradas, geraria atraso na execução e desperdício de memória.

Existem também, os *checkpoints* coordenados, que são sincronizados de modo a impedir o efeito cascata, permitindo um uso menor e mais otimizado de memória, mas tendo uma latência maior devido à sincronização. O FT-CUDAlign usa *checkpoints* coordenados, específicos de aplicação.

3.3.2 Arquitetura do FT-CUDAlign

Baseado no MASA-CUDAlign-MultiBP, a arquitetura do FT-CUDAlign se utiliza dos módulos de Controle e Executores, com o intuito de detectar falhas. No entanto, uma vez que apenas o Controlador pode lidar e tratar com as falhas, foi assumido que esse módulo não apresenta falhas.

No FT-CUDAlign, os *checkpoints* ocorrem após cada *breakpoint*, salvando a última coluna e em caso de falhas, a execução recomeça no último *breakpoint*. Uma primeira rotina de validação das GPUs é feita na fase de pré-execução e caso detecte falha, aquela GPU especifica é retirada da lista de válidas, conforme a Figura 3.14.

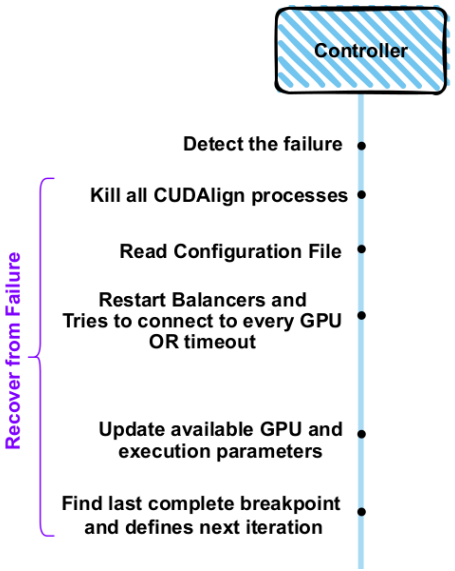


Figura 3.14: Passo a passo do Controller ao identificar falha [5].

O segundo passo de detecção de erros ocorre na fase de execução, e em casos de erro, a execução retorna para o último checkpoint válido. Há uma nova análise para detectar possíveis GPUs inativas e recalcular a proporção de colunas de cada GPU (*splits*) e, somente após isso, a iteração é reinicializada. Assim que a iteração é concluída, mais duas rotinas de validações são chamadas, sendo elas sincronizadas, a primeira consiste em detectar erros entre o fim do processamento da matriz e o fim da iteração e a segunda garante o estado da iteração para que a próxima possa ser inicializada.

3.3.3 Detecção de erros

Inicialmente, é feita a detecção de erros em pré-execução, que consiste em análise das chamadas de *socket* entre o controlador e os monitores de cada GPU, de modo a utilizar *ACKs* para chamadas de sincronização e conexão com cada GPU antes da submissão de

tarefas. Caso detecte falha, seja por *socket* inválido ou inoperante, deve haver rebalanceamento da divisão de dados (*splits*), das partições de trabalho (*parts*) e das colunas de carga (*load-columns*) entre as demais GPUs consideradas ativas, de modo a garantir a continuidade da execução.

A técnica reativa ocorre durante a execução e consiste na análise das chamadas de conexão entre os monitores de GPUs e seus adjacentes, uma vez que esses, são os únicos capazes de identificar erro em tempo de execução, seja por conta do não recebimento das células processadas pela GPU adjacente, seja pela não conexão com a GPU ao fim da execução, detectando assim falha no monitor e fazendo com que o *controller* inicie o tratamento de falhas. Uma nova análise dos monitores com GPUs válidas deve ocorrer e o rebalanceamento de carga ocorre, para o reinício da execução a partir do último checkpoint válido existente.

3.3.4 Protocolos de fim da iteração

Para o FT-CUDAlign, diferente do MASA-CUDAlign-MultiBP, foi implementada uma detecção de falhas antes das operações de terminação que consistem em finalizar a escrita e limpar as estruturas de dados. Desse modo, as estruturas daquele monitor só são apagadas em caso de sucesso da GPU seguinte ou no fim da execução da última GPU para aquela iteração. Com isso, caso uma GPU falhe durante a execução, os dados da GPU anterior podem serem utilizados para reprocessamento, evitando que o código entre em *deadlock*, tendo que voltar para o último checkpoint.

Já no fim da iteração, para a criação do *checkpoint*, uma última validação foi criada, permitindo uma análise dos arquivos criados e se houve erro na última GPU. No FT-CUDAlign, um arquivo deve ser criado antes da deleção das estruturas de dados para aquela GPU e após a deleção, mais um arquivo deve ser criado a fim de validar o checkpoint e este possa ser considerado em estado livre de erros. Caso o segundo arquivo não seja criado em um tempo pré-determinado, é entendido que houve erro na execução dessa GPU e deve ser chamada a rotina de recuperação e reanálise das GPUs.

Capítulo 4

O Gerenciador de Execuções de Aplicações SLURM

4.1 Visão Geral

O *SLURM* (*Simple Unix Utility for Resource Management*) é um escalonador de tarefas e gerenciador de recursos para computadores organizados como cluster de alto desempenho [7]. O cluster gerenciado pelo SLURM está ilustrado na Figura 4.1. Existem quatro tipos de nós, cada um com papel específico no gerenciamento. Os nós de submissão são os nós aos quais os usuários se conectam para submeter *jobs* (tarefas) à fila de execução e acompanhar sua execução. Os nós de *head* são os responsáveis por gerenciar as filas de execução, decidir em que nós trabalhadores, os jobs vão executar, os nós tralbahadores, responsáveis por executar o *job* que lhe for enviado e finalmente, os nós de banco de dados, que possuem dados permanentes dos usuários. O SLURM possui três funções principais [6].

1. Alocar nós de computação para que os usúarios possuam acesso à recursos do cluster, sendo exclusivos ou não-exclusivos e possam executar as tarefas submetidas (*jobs*), controlando a quantidade de recursos e período de tempo necessário.

2. Oferecer um ambiente para execução e monitoramento das tarefas submetidas aos nós alocados, promovendo uma análise em tempo real.

3. Controlar a fila de submissão, arbitrando conflitos e controlando recursos sobre as submissões.

O SLURM [7] foi desenvolvido com o objetivo de ser uma ferramenta de uso genérico, com foco na simplicidade, permitindo que usuários finais compreendam seu funcionamento e possam estender suas funcionalidades conforme necessário. Por ser um sistema de

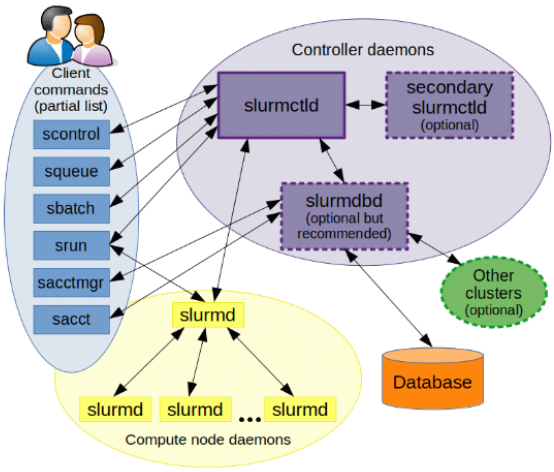


Figura 4.1: Componentes do SLURM [6]

código aberto, sob licença GNU GPL, o SLURM permite modificações e a distribuição de versões customizadas, além de oferecer suporte à adição de funcionalidades por meio de um mecanismo modular baseado em *plugins*.

Projetado para ser portável e escalável, o SLURM é implementado em linguagem C e utiliza o sistema de configuração GNU Autoconf, o que facilita sua adaptação para diferentes sistemas operacionais baseados em UNIX. Sua arquitetura permite escalabilidade eficiente em *clusters* compostos por milhares de nós. Os usuários podem especificar os requisitos de recursos computacionais e intervalos de execução para seus *jobs*, o que possibilita uma inicialização mais rápida e otimizada, em comparação a abordagens tradicionais de gerenciamento.

No que diz respeito à robustez, o SLURM é capaz de lidar com falhas em um ou mais nós de execução sem comprometer os *jobs* em execução [6]. Quando ocorrem falhas, os nós afetados são isolados, permitindo que os demais continuem sendo utilizados normalmente. Dessa forma, o impacto é minimizado e a continuidade da execução das tarefas é preservada.

O sistema também oferece recursos de segurança, empregando tecnologias de criptografia para autenticação mútua entre usuários e serviços. Além disso, o SLURM pode ser configurado com diferentes plugins de segurança, de forma a atender a requisitos específicos de ambientes computacionais mais restritos.

4.2 Arquitetura do SLURM

O SLURM tem uma arquitetura baseada em nós, que possui um ou mais *daemons* para cada nó, chamados de *slurmd* e um daemon central chamado de *slurmctld* responsável pelo

monitoramento dos nós e controle de comandos (Figura 4.2). As entidades gerenciadas pelos *daemons* do SLURM incluem os nós, que representam os recursos computacionais disponíveis, e as partições, que agrupam os nós em conjuntos lógicos conforme a Figura 4.3.

Os *jobs* também são controlados pelos *daemons*, onde cada *job* representa a alocação de recursos para cada usuário por um tempo determinado, e os *job* steps são os conjuntos de tarefas executadas dentro do *job*. Os *jobs* são alocados em conjuntos de nós em uma partição, podendo ser executado em execução direta ou separado em *job* steps e executados paralelamente ou de forma independente utilizando os nós daquela partição.

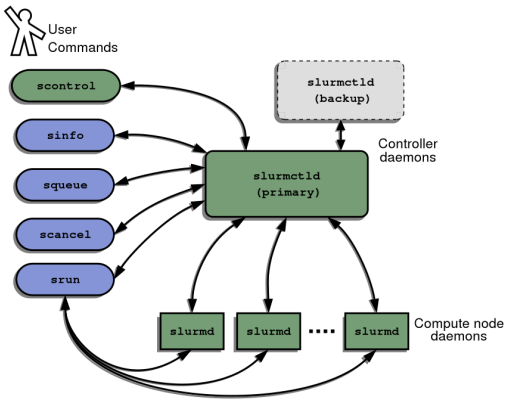


Figura 4.2: Arquitetura do SLURM [7]

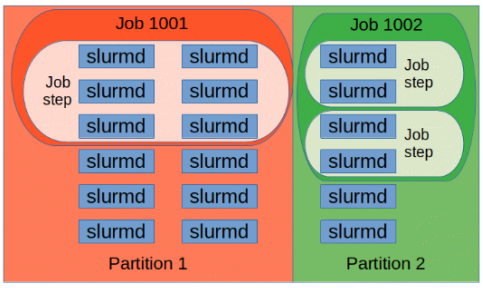


Figura 4.3: Representação de *jobs*, nós e partições [6]

Os *daemons slurmd* são *multithreaded* e se executam em um nó do *cluster*. Ao iniciar, o arquivo de configuração é lido e qualquer dado ou informação de estado prévio é salvo e recuperado, notificando o controlador (*slurmctld*) de que está ativo e pode receber tarefas. Ao receber, pode executá-las e devolver o status de execução, ficando livre para novas atividades ou, em caso de falha, ficar inativo.

Os *daemons* são tradicionalmente empregados para executar e monitorar o estado das tarefas submetidas. Entretanto, sua função não se restringe a esse papel. Esses processos também assumem a responsabilidade pelo gerenciamento dos fluxos de entrada

e saída padrão (*stdin*, *stdout* e *stderr*). Essa função é essencial para prevenir bloqueios de execução, que podem ocorrer quando os *buffers* de saída ficam cheios,o que acarretaria na paralização do *job*. Ao interceptar e redirecionar esses fluxos, os *daemons* asseguram a continuidade da comunicação entre os nós de execução e o usuário, além de viabilizar a execução remota. Dessa forma, constituem elementos centrais para a coordenação e o suporte à execução de *jobs* paralelos em ambientes de computação de alto desempenho.

O *daemon slurmctld*, ou controlador, é responsável pelo controle da execução dos *jobs*. Ao ser iniciado, também lê o arquivo de configuração do SLURM e carrega informações de estado definidas previamente. Sendo tolerante a falhas, o estado do controlador é salvo periódicamente em disco, além de salvamento instatâneo para alterações incrementais.

É composto por três componentes principais[7]:

1. O Gerenciador de Nós, responsável por analisar o estado de cada nó, utilizando um *daemon slurmd* para receber esse estado, conferindo se o estado é valido e pode ser considerado para receber um trabalho.

2. O Gerenciador de Partições, responsável por alocar nós aos *jobs*, se baseando na resposta do gerenciador de nós para detectar nós validos e na resposta do Gerenciador de *jobs* para saber qual *job* deve ser inicializado e sua necessidade por recursos.

3. O Gerenciador de *jobs* é responsável por aceitar ou recusar as solicitações de *jobs* dos usuários e organizá-los em uma fila de prioridades, alocando o *job* de maior prioridade às partições conforme disponibilidade, e passando o estado necessário para cada nó a fim que o *job* comece sua execução.

4.3 Segurança

As informações de configuração e estado são públicas, assim como a permissão dos usuários do *cluster* enviarem seus *jobs* e terem permissão de cancelar seus próprios *jobs*. Dois formatos de autenticação via plugin foram apresentados: AuthD e munge.

O Authd [20], é um plug-in de software que fornece e verifica credenciais de usuário que contenham assinatura criptográfica baseada na criptografia de chave pública RSA. Nesse modelo, um par de chaves RSA é instalado em todos os nós do *cluster*. O *daemon* authd em cada nó assina credenciais com a chave privada, e os serviços do *cluster* validam a autenticidade do usuário utilizando a chave pública compartilhada.

O Munged faz a requisição da função ***getpeername()*** gerando uma credencial contendo a identificação única do usuário, a identificação do grupo, o tempo de vida e a origem da conexão, usando a credencial assinada com *Message Authentication Code(MAC)*. Após isso a credencial é criptografada com uma chave secreta compartilhada entre todos os

nós. Desse modo, quando o *slurmd* executa o *job* submetido, a credencial é validada pelo *munged* e permitindo a execução do *job* naquele nó, impedindo que a credencial seja usada novamente de forma inapropriada.

Além disso o *slurmctld* guarda uma credencial do *job* step para cada alocação. Essa credencial contém dados como ID do usuário, *job* ID, step ID, lista de nós alocados e tempo de validade. Cada *slurmd* valida a credencial para garantir que o *job* pode de fato ser executado naquele nó, e utiliza essa credencial também para autenticar os fluxos de entrada, saída e erro padrão do *job*.

4.4 Políticas de Escalonamento

O SLURM foi desenvolvido com foco em simplicidade e desempenho para execução de *jobs*. O escalonador padrão denominado *built-in*, avalia os *jobs* em ordem estrita de prioridade e é acionado apenas em eventos específicos, como submissão ou finalização de *jobs*, ou modificações no arquivo de configuração. Alternativamente, é possível configurar a execução periódica do escalonador por meio do parâmetro *sched_interval*.

4.4.1 Backfill

No escalonador padrão, caso algum dos *jobs* de alta prioridade esteja pendente, nenhum outro *job* na partição será executado, a fim de evitar que a competição de recursos com o *job* prioritário pendente. Porém o SLURM oferece um plug-in de escalonamento *backfill[6]* que considera o tempo de execução e a quantidade de recursos necessários para determinar se e quais *jobs* de prioridade mais baixa podem ser executados sem que recursos sejam retirados de *jobs* de prioridade mais alta, permitindo que que *jobs* se iniciem na partição e aumente a eficiência da utilização de recursos.

O *backfill* realiza reservas por nós completos, mesmo quando o *job* não exige todos os recursos do nó, com o intuito de melhorar a performance e evitar problemas em partições de nós. Uma lista ordenada de job-partição é criada para determinar a ordem de execução e possibilidade de executar um *job* de prioridade mais baixo, durante a execução de outros *jobs* na partição. Em casos de *jobs* submetidos em múltiplas partições, o escalonador pode avaliar as opções, mas inicia o *job* na partição com menor tempo de espera estimado.

Contudo, um *job* com múltiplas reservas pode bloquear outros *jobs* ao reservar recursos em partições com maior prioridade, impedindo sua própria execução antecipada em partições de menor prioridade que compartilhem os mesmos nós.

O escalonador é custoso e gera interrupções no sistema para ser executado, de modo que as interrupções do sistema para escalonamento são liberadas a cada dois segundos permitindo outros processamentos. Mesmo após essa liberação o *backfill* pode continuar

sua execução evitando os *jobs* recém-submetidos, melhorando a eficiência do escalonador, mas atrasando a análise e na execução de *jobs* recém-submetidos.

4.4.2 Prioridade do Job

O SLURM utiliza uma política multifatorial de prioridade chamada de *FairShare*[21], baseada em fatores como *Fair-Share Factor*, que consiste em analisar o histórico de recursos usados por cada usuário e quem utilizou menos recurso deve ter maior prioridade, *Age Factor*, permitindo que *jobs* mais antigos subam sua prioridade para terem uma chance maior de serem executados, *Job Size Factor*, permitindo que *jobs* que utilizem menos recursos possam ter maior prioridade e *Partition Factor*, que prioriza *jobs* com base em partições ou filas aos quais foram submetidos.

4.4.3 Preempção

Há a possibilidade de que o SLURM utilize preempção para *jobs* [6], de modo a pausar os menos prioritários e transferir os recursos para um job de maior prioridade. Baseado na configuração, *jobs* que podem ser preemptados, podem ser cancelados, podem ser readicionados à fila, ou suspensos e executados após o término do *job* que causou a preempção ou mesmo terem recursos compartilhados alocados em conjunto com o *job* que gerou a preempção.

A preempção pode ser configurada considerando o *PriorityTier* da partição ou através da sua *Quality of Service(QoS)*. Assim, são definidos quais *jobs* podem interromper a execução de outros e quais podem ser interrompidos, além de permitir diferentes mecanismos de preempção para diferentes partições ou diferentes *QoS*.

Quando um *job* é submetido, o SLURM deve deve tentar executá-lo sem preempção e caso não seja possível iniciar o *job* pendente sem preempção, o *plugin* simula a preempção dos *jobs* em uma lista ordenada por prioridade para verificar após quais preempções o *job* pendente pode iniciar. A lista de *jobs* que podem ser interrompidos é reorganizada para colocar primeiro o *job* que deve ser interrompido por último, e os outros são ordenados de acordo com a quantidade de recurso que eles utilizam e o *job* pendente precisa. Depois, o sistema refaz a simulação da preempção com essa nova ordem. Isso pode fazer com que alguns sejam preemptados fora da ordem de prioridade, mas diminui o total de preempções necessárias.

Ao se utilizar a preempção em união com o *backfill*, caso um *job* seja preemptado para que outro possa ser executado, todo o nó será alocado para o novo *job*, evitando assim a fragmentação do nó, conflitos de recursos e reduzindo a complexidade da análise de recursos, deixando o escalonador mais simples e eficiente.

4.5 Execução

A execução de *jobs* no SLURM é realizada através de dois comandos principais, *sbatch* e *srun*, cada um com comportamentos diferentes entre si e funcionalidades diferentes.

O comando *sbatch* é utilizado para submeter scripts de execução à fila de tarefas SLURM. Uma vez submetido o *job* é enfileirado em uma fila de prioridades, e deve aguardar até que tenha disponibilidade de recursos, sendo executado a depender das políticas de prioridade, preempção e partição.

Por outro lado o comando *srun* deve ser utilizado para execuções interativas e imeditadas, pois possui um comportamento de prioridade máxima, travando os recursos solicitados até que todos estejam disponíveis e a partir do momento que todos os recursos solicitados estiverem disponíveis, o *job* é executado, sem necessidade de enfileiramento.

4.5.1 Filtros e Configurações de Execução

Os comandos *sbatch* e *srun* permitem a definição de filtros de recursos e parâmetros de execução para controlar o ambiente em que o *job* será executado. Esses filtros especificam quantidade de nós, quantidade de tarefas (tasks), número de GPUs e, opcionalmente, nós específicos destinados à execução.

```
#Filtro associado a quantidade de nós direcionados ao job (3 Nós)
sbatch -N3 script.sh

#Filtro em relação a quantidade de tasks direcionadas ao job (3 tasks)
sbatch -n3 script.sh

#Filtro em relação a quantidade de GPUs direcionada ao job (2 GPUs)
#sbatch --gres=gpu:2 script.sh

#Especifica o nó que irá executar o job (Nó "cuda")
sbatch -w cuda script.sh
```

Esses filtros podem ser combinados livremente conforme a necessidade do experimento, permitindo uma alocação precisa dos recursos computacionais. Além disso, essas configurações podem ser definidas diretamente dentro do script *bash* a ser executado, utilizando diretivas iniciadas por "#SBATCH".

A figura 4.4 apresenta um exemplo de script bash configurado para submissão ao SLURM, com parâmetros mínimos de execução.

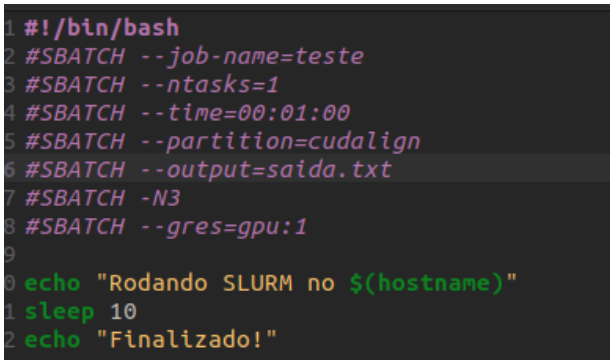


Figura 4.4: Exemplo simples de script com configurações e pré-requisitos a ser executado pelo SLURM

Nesse exemplo, o script contém diretivas "#SBATCH"que especificam:

- -job-name: O nome do *job*, exibido na fila de execução;
- -ntasks: O número de tarefas a serem utilizadas;
- -time: O tempo máximo de execução permitido antes do encerramento automático;
- -partition: O nome da partição onde o *job* deve ser executado (definido no arquivo slurm.conf);
- -output: O arquivo de saída onde o log será armazenado;
- -N3: O número mínimo de nós necessários para iniciar a execução, no exemplo foram 3;
- -gres=gpu:1: A quantidade de GPUs que cada nó deve possuir para execução do *job*.

Essas diretivas são interpretadas pelo SLURM no momento da submissão, permitindo que o agendador gerencie os recursos de forma eficiente e automatizada. No exemplo da Figura 4.4 o *job* teste é executado com uma tarefa simples para entendimento dos parâmetros e necessidades e configuração dos parâmetros que o SLURM é capaz de interpretar e permitir interações para sua execução.

Capítulo 5

Projeto da Integração entre o FT-CUDAlign e o SLURM

Esse capítulo descreve a configuração experimental utilizada para a implementação e execução dos algoritmos CUDAlign e do FT-CUDAlign, ambos baseados em GPUs com arquitetura CUDA. O ambiente foi implantado em um cluster de máquinas Linux, configurado com o gerenciador de recursos SLURM, o serviço de autenticação MUNGE, o sistema de arquivos distribuído e as aplicações MASA-CUDAlign e FT-CUDAlign.

A configuração foi planejada para permitir a execução paralela dos experimentos, garantindo a escalabilidade, reprodutibilidade e o uso eficiente dos recursos computacionais disponíveis.

5.1 Visão Geral

Na nossa solução, consideramos um cluster de máquinas em um conjunto de máquinas com GPU que se comunicam através de uma rede local e utilizam o escalonador SLURM. O projeto foi feito de forma a manter uma configuração homogênea entre as máquinas do cluster, diferenciando-se apenas a máquina responsável pelos serviços de controle do SLURM. Para isso foi definida uma máquina de controle, encarregada de possuir o *daemon* controlador do SLURM (*slurmctld*) e de atuar como servidor NFS, exportando o diretório a ser compartilhado e utilizado em execuções do FT-CUDAlign. Além das funções de controle, essa máquina também possui um nó de execução permitindo-a ser usada como máquina de trabalho. Além disso, para este projeto, adotou-se o pressuposto de que a máquina de controle não apresentará falhas durante a execução.

Cada uma das máquinas de trabalho, possui uma configuração idêntica, contendo:

- MUNGE, utilizado para autenticação entre os nós do cluster e essencial para o funcionamento do SLURM;

- SLURM com um *daemon* de trabalho ativo (slurmd), responsável pelo escalonamento, gerenciamento e execução de *jobs*;
- NFS, utilizando um *NFS Client* para permissão de acesso a um diretório comum necessário para troca de informações na execução do FT-CUDAlign;
- FT-CUDAlign, garantindo que todos os nós possuam a aplicação alvo do TCC. Necessário também o MASA-CUDAlign como pré-requisito para execução do FT-CUDAlign.

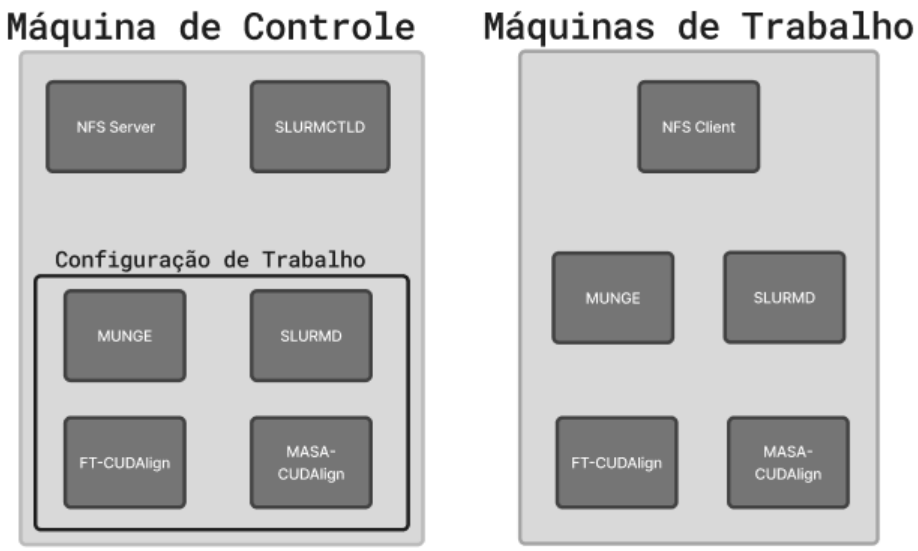


Figura 5.1: Representação mínima das configurações das máquinas utilizadas.

A Figura 6.1 apresenta os módulos que se executam na máquina de controle e nas máquinas de trabalho. A máquina de controle é um servidor NFS e possui os arquivos que serão acessados de maneira remota pelas máquinas de trabalho (clientes NFS). Os arquivos em questão são os dependentes para execução contínua do FT-CUDAlign, ficando dentro da pasta "share" no diretório compartilhado, esses arquivos são os responsáveis pela sincronização e conexão entre as máquinas e GPUs, onde cada máquina é responsável pelo registro do *socket* ou arquivo ao qual seus dados são direcionados, um arquivo com o sinal de término de sua execução para que a GPU anterior leia a consiga interpretar os resultados obtidos no arquivo binário com os resultados obtidos pela GPU em questão.

5.2 Configuração dos Serviços e Execução dos Módulos

5.2.1 Execução MUNGE

O MUNGE [22] possui a função de autenticação e autorização segura entre os nós do cluster, o que permite ao SLURM validar a identidade dos processos que submetem os *jobs*.

A configuração inicia com a geração de uma chave secreta no nó controlador através do comando `"mungekey"` em um terminal com as bibliotecas do MUNGE carregadas. Essa chave é armazenada em `"/etc/munge/munge.key"` ou no caminho relativo passado no momento da instalação e deve ser compartilhada entre todas as máquinas com as devidas permissões, para que o MUNGE seja capaz de fazer a conexão entre as máquinas. Após o compartilhamento, o serviço é iniciado com o comando a seguir, seguindo ajuste nos caminhos dos parâmetros:

```
munged --verbose   --key-file=.../etc/munge/munge.key   \
--socket=.../var/run/munge.socket.2   \
--pid-file=.../var/run/munged.pid
```

Para a primeira execução do comando, os arquivos de *socket* e *pid* serão criados no momento da inicialização do MUNGE, não sendo necessário mais comandos de inicialização. Depois da execução do comando, o *daemon* do MUNGE entra em execução.

Para sua execução, o MUNGE depende de dois pacotes fundamentais. O *libgpg-error* para tratar falhas de autenticação e a validação da criptografia e o *libgcrypt*, que permite a criptografia e descriptografia da chave privada.

5.2.2 SLURM

Para o funcionamento adequado, o Slurm necessita da instalação prévia do MUNGE e do compartilhamento das chaves de autenticação entre os nós. Além disso, necessita de um arquivo de configuração único (*slurm.conf*) que deve estar presente em todas as máquinas, caso contrário, os nós de trabalho se tornam incapazes de se conectar ao nó de controle. O conteúdo do arquivo *slurm.conf* está ilustrado na Figura 5.2.

Para esse projeto também se torna essencial que o arquivo de configuração de recursos (*gres.conf*), onde devem ter um arquivo configurado por nó de execução e contendo o nome do nó, a quantidade de GPUs utilizadas para o nó, o modelo de cada um e onde está hospedada, seguindo o padrão:

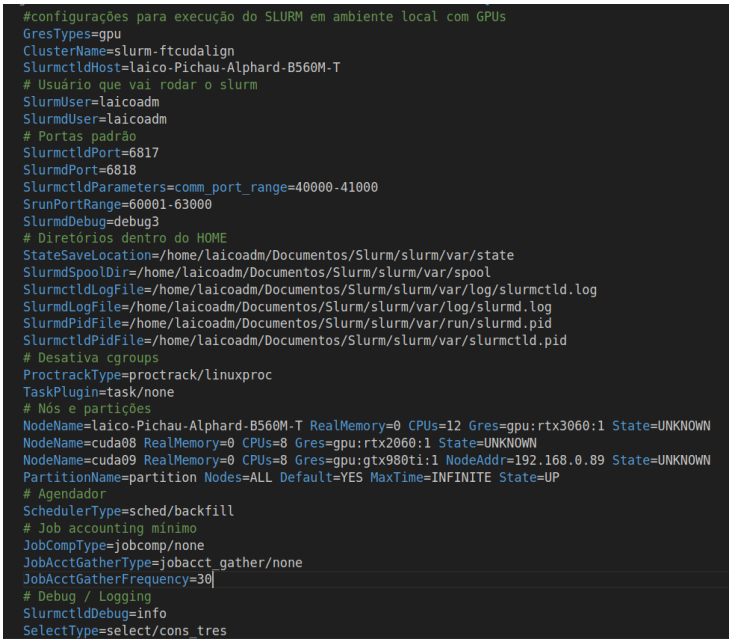


Figura 5.2: Configurações e conteúdo do slurm.conf do projeto

```
NodeName=cuda09 Name=gpu Type=rtx3060 File=/dev/nvidia0
NodeName=cuda09 Name=gpu Type=rtx2080 File=/dev/nvidia1
```

Os nós podem ser ativados de duas formas, a depender da funcionalidade responsável para cada nó. Nos dois casos é passado como parâmetro o caminho até o arquivo de configuração.

```
#Nó de Trabalho
slurmd .../etc/slurm.conf

#Nó de Controle
slurmctld  .../etc/slurm.conf
```

Além disso, a execução de *jobs* também depende do método de submissão:

- Jobs executados com o comando `srun script.sh`, devem possuir o arquivo "script.sh", todos os nós responsáveis pela execução.
- Jobs executados através do comando `sbatch script.sh`, devem estar contidos na máquina referente ao nó de trabalho com menor ID a ser alocada para o SBATCH.

O SLURM, por padrão, utiliza o identificador numérico dos nós como critério primário para decidir em qual nó um *job* será executado, desde que os requisitos mínimos solicitados sejam atendidos. Além disso, os nós são organizados em ordem alfabética crescente com

base no parâmetro *NodeName*. Dessa forma, nós cujo nome aparece mais próximo do início do alfabeto recebem identificadores menores e, consequentemente, tendem a ser selecionados com maior frequência durante as execuções.

5.2.3 Configuração e Execução do NFS

Tanto o MASA-CUDAlign quanto o FT-CUDAlign dependem de um diretório compartilhado entre todos os nós participantes da execução distribuída. Esse diretório é utilizado para troca de dados, checkpoints, arquivos intermediários e sincronização dos resultados parciais entre GPUs.

Para essa necessidade, foi configurado um servidor NFS (*Network File System*), onde o nó controlador também é responsável pelo serviço de dados. Para isso, o nó controlador deve ser capaz de exportar a pasta compartilhada e permitir acesso às outras máquinas, via adição da estrutura `"/diretorio_compartilhado IPs_permitidos (permissões)"` ao arquivo "/etc/exports", além da liberação de permissões via *chmod* e *chown* para que os *NFS Clients* tenham o acesso devido ao diretório.

Já para as máquinas de serviço, detentoras do *client*, a montagem do NFS se baseia no comando `"mount IP_do_servidor:diretorio_compartilhado diretório_do_client"`. Para esse projeto é essencial que todos os nós montem o diretório no mesmo caminho, garantindo que os scripts e executáveis referenciem corretamente os arquivos de entrada e saída.

5.2.4 Execução do CUDAlign e do FT-CUDAlign

Os frameworks CUDAlign 4.0 (Seção 3.2.2) e FT-CUDAlign (Seção 3.3) constituem o ponto chave da execução experimental desse projeto. Ambos foram executados no ambiente configurado com suporte a CUDA e integrados ao SLURM para submissão e balanceamento de tarefas entre GPUs.

A execução do MASA-CUDAlign necessita de todas as máquinas em execução do início ao final, enquanto o FT-CUDAlign possibilita o tratamento de falhas e continuação mesmo após erros. O comando de execução "`./balancer N°GPU diretorio_compartilhado`" é responsável pela inicialização do *balancer* e deve ser inicializado para cada GPU a ser utilizada na comparação e o comando "`./controller /caminho_conf_file /diretorio_compartilhado`" inicia a execução do *controller* usando o arquivo de configuração /caminho_conf_file.

5.3 Execução do FT-CUDAlign no SLURM

A execução do FT-CUDAlign no SLURM ocorre através da submissão de um *job* ao SLURM e depende de alguns parâmetros a serem passados, tanto internos ao *job*, quanto externos (na hora de submissão). O primeiro parâmetro interno a ser mapeado, é a necessidade da alocação dos recursos de GPU do nó e para isso deve ser passado como parâmetro SLURM, `"#SBATCH -gres=gpu:all"` de modo a permitir e alocar todas as GPUs presentes nos nós alocados.

Um parâmetro externo deve ser associado para execução do *job* sendo esse na chamada de submissão de `"sbatch -N`x `arquivo_de_submissao"`, onde x determina quantos nós devem ser alocados, de modo a permitir uma maior dinâmicidade ao *job* e permitir que seja escalável para executar tanto com quantidades baixas de nós e GPUs quanto com quantidades maiores.

Após a alocação dos nós e GPUs, os parâmetros a serem mapeados serão os necessários para execução do FT-CUDAlign em seu arquivo de configuração, sendo:

- A quantidade de GPUs totais a serem utilizadas na execução, nesse caso definido pela soma total das GPUs presentes em cada nó;
- A quantidade de *breakpoints*, nesse caso fixado em 2;
- Pares IP/Porta mapeados para conexão em cada GPU;
- Quantidade de GPUs por máquina que participam da execução, nesse caso são todas as GPUs dos nós;
- Se há ou não *Block Pruning*;
- A proporção de colunas que cada GPU irá executar (*split*), nesse caso deixamos fixo em uma quantidade igual para cada GPU;
- As sequências a serem usadas na aplicação;
- Parâmetro WAIT com o tempo de espera para recebimento/envio do melhor escore local;
- O modo de execução, para o projeto será sempre *dynamic* para execução dinâmica do MASA-CUDAlign;
- Parâmetro PROG para utilização do *decision maker*, capaz de avaliar e prever qual modelo seria o mais eficiente (*Static* ou *Dynamic*);
- Parâmetro GFLOPS com GFLOPS de cada GPU a fim de avaliar a distribuição de linhas processadas por GPU e recalcular o *split* usado.

No script bash definido para execução SLURM a porta padrão usada será 5234 e caso o nó possua mais de uma GPU, a lógica das portas segue uma ordem crescente para execução. As definições dos parâmetros e diretórios utilizados para confecção do script estão apresentadas na Figura 5.3. A quantidade de GPUs, IPs e o *Array* contendo as portas de conexão entre os *sockets* foi mapeada usando instruções Linux e comandos *srun* para terem acesso aos dados de cada máquina, conforme Figura 5.4.

```
#define diretórios usados
ftDIR=/home/laicoadm/Documentos/Filipe/FT-dynBP
sharedDIR=/home/laicoadm/Documentos/Gustavo-Slurm/CUDA/SharedDir
folders=(2-5M 3-7M 4-10M) #Pastas com as sequências utilizadas por execucao

#Define parametros de rede e GPU
total_gpus=0 #Pegar o total de gpus para o arquivo de configuracao
nodes_gpus=() #pegar a quantidade de gpus de cada maquina para o balancer
nodes_ips=()
IPs=""
GPUs=""
BasePort=5234
PortArray=""
SEQ=""
SPLIT=""
GFLOPS=""
```

Figura 5.3: Definição dos diretórios utilizados e parâmetros do arquivo de configuração FT-CUDAlign

```
#getData:Pega a quantidade de gpus, os IPs e faz o PortArray para arquivo FT-CUDAlign
getData() {

for n in $(scontrol show hostnames $SLURM_NODELIST); do
    ip=$(srun -N1 -w $n hostname -I | awk '{print $1}')
    nodes_ips+=($ip)
    gpus=$(srun -N1 -w $n nvidia-smi -L 2>/dev/null | wc -l)
    nodes_gpus+=($gpus)
    total_gpus=$((total_gpus+gpus))
    for ((j=0; j<gpus; j++)); do
        if [[ -z "$IPs" ]]; then
            IPs="$ip"
            GPUs="$j"
            PortArray="$BasePort"
        else
            IPs="$IPs,$ip"
            GPUs="$GPUs,$j"
            PortArray="$PortArray,$((BasePort+j))"
        fi
    done
done

}
```

Figura 5.4: Função responsável pelo reconhecimento dos IPs, GPUs e montar o *Array* de portas para conexão entre *sockets*.

Para as sequências utilizadas, foram passados os caminhos do diretório para cada comparação, tendo sido separado uma pasta para cada par de sequências. Após as definições e o mapeamento dos pârametros dinâmicos (IP, GPUs e Portas), é executada a função responsável pela criação do arquivo de configuração do FT-CUDAlign na máquina que

contém o nó de menor Id associado, responsável pela execução do módulo *controller* do FT-CUDAlign (Figura 5.5).

```
#writeFile:Escreve o arquivo de configuracao com base no folder selecionado
writeFile(){
#$1 -> folder da sequencia

path=$sharedDIR/sequences/$1
configFile="$ftDIR/Configs/config$1"
files=("$path"/*.fasta)
str1="${files[0]}"
str2="${files[1]}"
SEQ="$str1,$str2"
#Cria o arquivo config referente a sequencia do folder
cat <<EOF > $configFile
GPUS $total_gpus
BREAK 1
IP $IPs
PORT $PortArray
GPU# $GPUs
BP 0
SPLIT $SPLIT
SEQ $SEQ
WAIT 0
MODEL dynamic
PROG ./dynamic
GFLOPS $GFLOPS
EOF

}
```

Figura 5.5: Escrita do arquivo de configuração do FT-CUDAlign (Parâmetro "$1" recebe a pasta contendo o par de sequências a ser comparado).

O próximo passo consiste em um *loop* que deve executar em cada GPU e ativar o módulo *balancer* a fim de iniciar a execução e a função responsável está representado na Figura 5.6. O script fica responsável por analisar o retorno de cada *balancer* para identificar possíveis falhas. Por último o módulo de controle é ativado e a partir desse momento a execução do FT-CUDAlign realmente se inicia, script final presente na Figura 5.7.

```
#runBalancer: Executa os Balancer, forçando a execucao em uma gpu e espera final da execucao
runBalancers(){
for n in $(scontrol show hostnames $SLURM_NODELIST); do
        gpus=$(srun -N1 -w $n nvidia-smi -L 2>/dev/null | wc -l)
        for ((count=0;count<gpus;count++)); do
        (
                srun -N1 -w $n --gres=gpu:1 ./balancer $count $path
                ret=$?
                echo "Retorno foi $ret"
        ) &
        done
done
}
```

Figura 5.6: *Loop* de execução dos *balancers*, com identificação do retorno para reconhecimento de falhas.

```
#Inicio da execucao do script
#Executa o getData
getData

#necessita do getData para saber o total de gpus participantes execucao
SPLIT=$(echo $(yes 30 | head -n $total_gpus) | tr ' ' ',')
GFLOPS=$(echo $(yes 14000 | head -n $total_gpus) | tr ' ' ',')


mkdir -p "$ftDIR/Configs"
for folder in "${folders[@]}"; do
        writeFile $folder
        cd $ftDIR
        runBalancers
        ./controller $configFile $path
        ret=$?
        echo "Retorno do Controller foi $ret"
done
```

Figura 5.7: Execução do script pós definições.

5.4 Execuções sem Falhas e com Falhas

Para esse projeto, dois formatos devem ser testados, sendo eles a execução do projeto sem falha e a execução com falha do FT-CUDAlign. Nas execuções sem falhas, vamos verificar se o FT-CUDAlign integrado ao SLURM funciona como anteriormente, ou seja, quando a sua execução era iniciada sem o SLURM.

Ou seja, na execução sem falhas o objetivo é apenas a obtencão do resultado final correto em tempo hábil. Para o caso com falhas, algumas funcionalidades devem ser testadas e os testes devem conter a falha do módulo *balancer*, onde o SLURM deve ser capaz de identificar o erro para retorno ao usuário. A falha no módulo *balancer* vai ser criada através da chamada de sistema `"pkill balancer"` e deve ocorrer entre o primeiro e o segundo *breakpoints*.

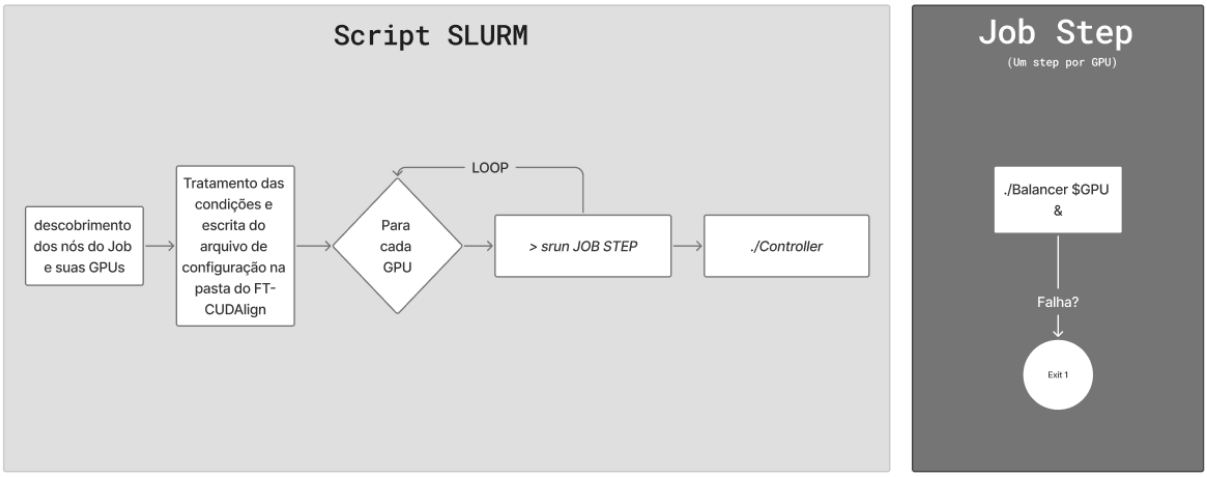


Figura 5.8: Projeto do Script de Execução

A Figura 5.8 apresenta a arquitetura geral do projeto. No script SLURM, o primeiro passo consiste em identificar os nós alocados para o *job* e suas respectivas GPUs, permitindo determinar quais endereços IP e quais GPUs farão parte da execução distribuída. Em seguida, é gerado automaticamente o arquivo de configuração do FT-CUDAlign, responsável por informar ao módulo *controller* todos os IPs, GPUs e demais parâmetros necessários, conforme descrito na Seção 5.3.

O terceiro passo envolve um *loop* que, para cada GPU mapeada, executa um comando *srun* direcionado ao nó correspondente, iniciando um subterminal responsável por ativar o *balancer* associado e monitorar seu retorno. Após a ativação de todos os *balancers*, o script inicializa o módulo *controller*, utilizando como entrada o arquivo de configuração criado no segundo passo.

O script SLURM completo confeccionado no projeto está em apêndice.

Capítulo 6

Resultados Experimentais

Os objetivos desse capítulo são verificar se o FT-CUDAlign executa via SLURM e verificar se o script SLURM é capaz de interpretar falhas das GPUs e/ou falhas do módulo controlador do FT-CUDAlign.

A solução proposta no Capítulo 5 (Projeto da Integração entre o FT-CUDAlign e o SLURM) possibilitou a execução paralela e distribuída dos experimentos de alinhamento em um ambiente controlado, com autenticação segura via MUNGE, gerenciamento eficiente de recursos através do SLURM e compartilhamento de dados pelo NFS.

O ambiente criado foi desenvolvido para simular o comportamento esperado em ambientes de supercomputadores, permitindo escalabilidade e melhor consumo dos recursos.

6.1 Sequências Selecionadas

As sequências reais de DNA selecionadas para realização dos experimentos estão apresentadas na Tabela 6.1, onde temos o tamanho, nome do organismo e id das sequências foram fornecidas, conforme registros do website *National Center for Biotechnology Information* [23].

Foram selecionadas as sequências de tamanhos 5MBP, 7MBP e 10MBP para a realização das execuções. Essas sequências foram testadas pela viabilidade de execução delas no ambiente de teste, uma vez que a complexidade de tempo é da ordem de ($O(n^2)$) e sequências maiores exigiriam uma quantidade de tempo muito longo de execução. Além disso, cabe notar que as máquinas do laboratório (LAICO) são utilizadas por diversos alunoes que desenvolvem seus projetos, sendo um ambiente compartilhado. Com isso, execuções muito longas teriam impacto em outros grupos de pesquisa.

Seq.	Id	Organismos	Tamanho
5MBP	AE016879.1	Bacillus anthracis str. Ames	5.227.293
	AE017225.1	Bacillus anthracis str. Sterne	5.228.663
7MBP	NC_003997.3	Bacillus anthracis str. Ames	5.227.293
	NC_005027.1	Rhodopirellula baltica SH	7.145.576
10MBP	NC_014318.1	Amycolatopsis mediterranei U32	10.236.715
	NC_017186.1	Amycolatopsis mediterranei S699	10.236.779

Tabela 6.1: Especificações das sequências comparadas.

6.2 Ambiente de Testes

O ambiente utilizado para a coleta e análise de resultados foi o Laboratório de Sistemas Integrados e Concorrentes da Universidade de Brasília (LAICO), utilizando até três nós, onde cada nó possui uma GPU para execução dos experimentos. A descrição das características das máquinas com GPUs está na Tabela 6.2.

GPU	Modelo	Memória	Núcleos	Clock	Arquitetura
1	RTX 2060	6GB	1920	1680 MHz	Turing
2	GTX 980ti	6GB	2816	1076 MHz	Maxwell
3	RTX 3060	12GB	3584	1777 MHz	Ampere

Tabela 6.2: Especificações das GPUs do LAICO.

6.3 Script de Execução do FT-CUDAlign

O script desenvolvido na Seção 5 foi executado no laboratório LAICO com duas e três máquinas. Ele foi capaz, no primeiro passo, de identificar os IPs dos nós assim como sua quantidade de GPUs a partir de chamadas padrões Linux agregadas ao uso do ***srun*** do SLURM que permite a execução de instruções sem a necessidade da utilização recorrente de chamadas "ssh".

O próximo passo consiste na criação ou alteração do arquivo de configuração para as sequências a serem processadas. No script foi criado um laço para que pudessem ser processadas várias sequências em ordem para ganhar tempo e evitar submissões múltiplas de *jobs*, retirando a necessidade de múltiplos tempos de espera para execução dos processos.

O terceiro passo consiste na criação de mais um laço de repetição para que a cada GPU mapeada nos passos anteriores, *job steps* contendo a ativação e monitoração dos *balancers* criados para execução e ao fim desse laço, quanto todos os *balancers* disponíveis

```
GPUS 2
BREAK 1
IP 192.168.0.158,192.168.0.111
PORT 5234,5234
GPU# 0, 0
BP 0
SPLIT 30,30
SEQ  /home/laicoadm/Documentos/Gustavo-Slurm/CUDA/SharedDir/sequences/2-5M/AE016879.1.fasta,/home/laicoadm/Documentos/Gustavo-Slurm/CUDA/SharedDir/sequences/2-5M/AE017225.1.fasta
WAIT 0
MODEL dynamic
PROG ./dynamic
GFLOPS 14000,14000
```

Figura 6.1: Arquivo de configuração do FT-CUDAlign gerado durante a execução em duas GPUs e um *breakpoint*.

```
GPUS 3
BREAK 2
IP 192.168.0.158,192.168.0.111,192.168.0.89
PORT 5234,5234,5234
GPU# 0,0,0
BP 0
SPLIT 30,30,30
SEQ  /home/laicoadm/Documentos/Gustavo-Slurm/CUDA/SharedDir/sequences/2-5M/AE016879.1.fasta,/home/laicoadm/Documentos/Gustavo-Slurm/CUDA/SharedDir/sequences/2-5M/AE017225.1.fasta
WAIT 0
MODEL dynamic
PROG ./dynamic
GFLOPS 14000,14000,14000
```

Figura 6.2: Arquivo de configuração do FT-CUDAlign gerado durante a execução em três GPUs e dois *breakpoints*.

estiverem ativos, o módulo *controller* deve ser iniciado e a partir desse ponto a execução e tratativa de falhas do FT-CUDAlign está em andamento.

6.4 Experimentos

Nos scripts desenvolvidos o SLURM se mostrou capaz de executar o FT-CUDAlign em um ambiente de cluster de até três máquinas Linux. Além disso, o SLURM também foi capaz de identificar falhas dos *balancers* de execução do FT-CUDAlign assim como identificar falhas do *controller*.

No entanto, o próprio FT-CUDAlign já possui mecanismos de tolerância a falha dos *balancers*, assim como cálculo dos novos *splits* e reinicio após alguma falha. Dessa forma, o script SLURM atuou mais como um gerenciador de recursos e monitor de execução, enquanto FT-CUDAlign manteve controle sobre as tratativas de falha.

6.4.1 Cenários sem Falha

Para as tratativas apresentadas aqui, seis testes foram feitos para cada par das sequência selecionadas, divididos em duas categorias: com a utilização de um *Breakpoint* (1BP) e com a utilização de dois *Breakpoints* (2BP). A Figura 6.3 apresenta os tempo obtidos em execuções sem falha e a diferença entre os *overheads* obtidos. Para esses testes foram usadas duas GPUs do ambiente, sendo elas a GPU(1) e a GPU(3) da Tabela 6.2 e consistiram principalmente na execução do script apresentado, com três execuções cada.

Como pode ser visto na Figura 6.3, o aumento do número de *breakpoints* aumenta o tempo de execução da aplicação. No caso da Comparação de $5M \times 5M$, o aumento foi de 22%.

A Figura 6.4 apresenta os tempos da comparação $7M \times 5M$. Da mesma forma que o caso anterior, o tempo de execução aumenta quando o número de *breakpoints* aumenta. Nesse caso, o aumento foi de 15%.

Finalmente, a Figura 6.5 apresenta os resultados para a comparação de $10M \times 10M$, com o mesmo comportamento das comparações anteriores. Para essa comparação, o aumento do tempo de execução foi de 6%. Com isso, nota-se que o aumento do tempo de execução devido ao aumento de *breakpoints* é inversamente proporcional ao tamanho das sequências comparadas. Ou seja, quanto maiores as sequências, menor o *overhead* acrescentado. Isso ocorre porque o paralelismo escala progressivamente até atingir o paralelismo máximo, porém, em sequências menores, quanto maior o tempo gasto com trocas de contexto, menor é o intervalo efetivo em que ocorre o paralelismo máximo, ampliando assim o custo do *overhead*. Os *overheads* mencionados nessa Seção estão ilustrados na Tabela 6.3.

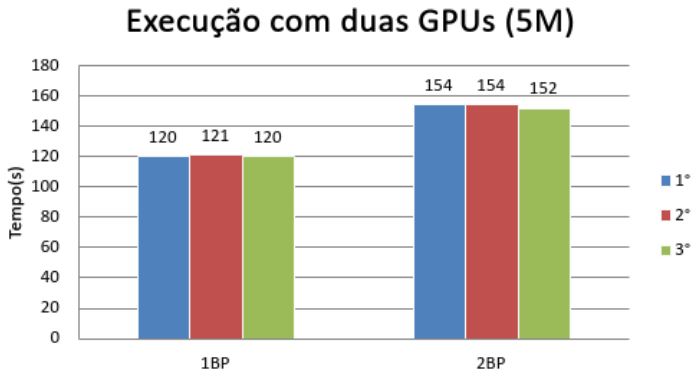


Figura 6.3: Tempos obtidos a partir de seis execuções: três utilizando um *breakpoint* e três utilizando dois *breakpoints*, para duas GPUs e sequências de 5M 6.1.

6.4.2 Cenários com Falha

Foram testados casos de falha para as mesmas comparações de sequências apresentadas na Seção 6.4.1 e presentes na Tabela 6.1. Para esses casos de teste, foi usada uma

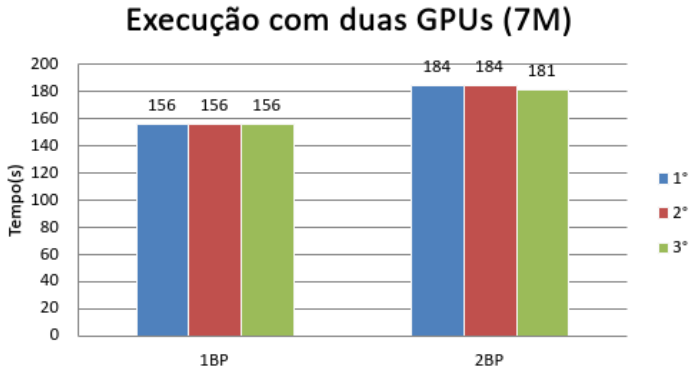


Figura 6.4: Tempos obtidos a partir de seis execuções: três utilizando um *breakpoint* e três utilizando dois *breakpoints*, para duas GPUs e sequências de 7M 6.1.

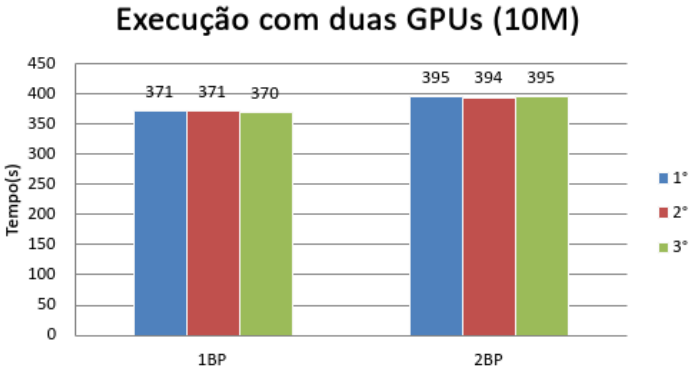


Figura 6.5: Tempos obtidos a partir de seis execuções: três utilizando um *breakpoint* e três utilizando dois *breakpoints*, para duas GPUs e sequências de 10M.

configuração com três GPUs e dois *breakpoints* (2BP). Para testar a tolerância a falhas, após o início do segundo *breakpoint*, foi criado um *fail-stop* na GPU(2), de modo a gerar um erro intermitente, permitindo que a GPU esteja apta a participar da próxima execução.

Os resultados obtidos são apresentados na Figura 6.6. Como pode ser visto, há um *overhead* considerável quando ocorrem falhas, mesmo em casos de falhas intermitentes. A ocorrência de uma falha exige tempo adicional para realinhamento do processamento, reorganização dos *splits* e verificação das GPUs que ainda possuem estado válido. Esse processo aumenta o tempo total de troca de contexto e, consequentemente, o *overhead*. Para a comparação $5M \times 5M$, o *overhead* em termos de tempo de execução ficou em 32%. Nas comparações $7M \times 5M$ e $10M \times 10M$ os *overheads* foram de 15% e 17%, o que indica *overheads* similares para essas duas comparações. A Tabela 6.4 apresenta esses *overheads*.

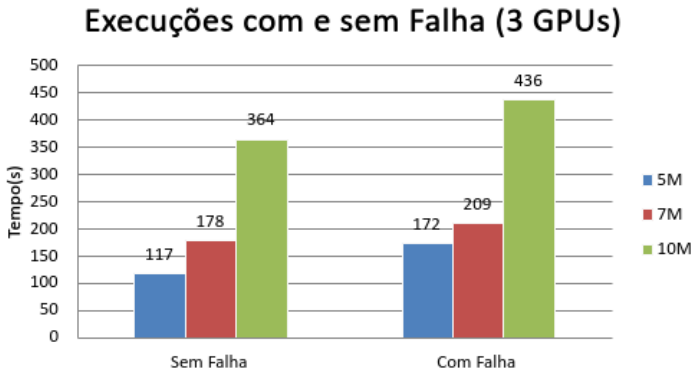


Figura 6.6: Tempos obtidos através da execução com e sem falha utilizando três GPUs.

Sequências	Overhead
5MBP	22%
7MBP	15%
10MBP	6%

Tabela 6.3: *Overheads* obtidos em cenário sem falha.

Sequências	Overhead
5MBP	32%
7MBP	15%
10MBP	17%

Tabela 6.4: *Overheads* obtidos em cenário com falha.

Capítulo 7

Conclusão

O presente trabalho apresentou a integração entre o escalonador SLURM e o FT-CUDAlign, propondo um modelo estruturado em quatro etapas principais:

- Detecção dinâmica dos recursos disponíveis nos nós do cluster, incluindo GPUs, endereços IP e mapeamento estrutural;
- Geração automática do arquivo de configuração necessário ao controlador do FT-CUDAlign;
- Ativação automática dos *balancers* já mapeados para o uso correto de cada GPU e monitoração dos *balancers* criados;
- Ativação e monitoração do módulo de *controle*, sendo capaz de identificar falhas desse módulo para possíveis melhorias futuras.

O FT-CUDAlign pôde ser executado de forma automática e adaptável a diferentes configurações de hardware, evidenciando a viabilidade da solução para ambientes heterogêneos. Além disso, verificou-se que o SLURM foi capaz de identificar falhas em GPUs durante a execução, fornecendo informações importantes para o controle do processo, ainda que as tratativas internas de recuperação continuem sendo responsabilidade do FT-CUDAlign. Esse comportamento confirma que a integração adiciona robustez sem interferir no mecanismo original de tolerância a falhas do algoritmo. Essa integração amplia o potencial do FT-CUDAlign a diversas infraestruturas de alto-desempenho, incluindo supercomputadores.

7.1 Trabalhos Futuros

Para trabalhos futuros, três caminhos foram identificados:

- A execução em *clusters* com múltiplas GPUs por nó e supercomputadores para análise de eficiência e escalabilidade da automação desenvolvida, onde podem explorar o balanceamento interno de cargas e comunicação entre GPUs;
- O SLURM nativamente já possui mecanismos simples para tolerância a falha e os scripts desenvolvidos foram pensados para tratar falhas no módulo de controle do FT-CUDAlign. Recomenda-se integrar essas funcionalidades às verificações internas do controlador, possibilitando a implementação de tolerância a falhas também no módulo de controle do FT-CUDAlign, uma limitação ainda não coberta pelo modelo atual;
- Sugerimos também a adoção da elasticidade dinâmica, combinando os mecanismos de tolerância a falhas do SLURM com sua capacidade de redimensionamento de *jobs* para permitir ao FT-CUDAlign expandir dinâmicamente a quantidade de *balancers* durante a execução, acelerando o processamento.

Referências

[1] O. Sandes, Edans Flavius de e Alba Cristina M.A. de Melo: *Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu.* IEEE Transactions on Parallel and Distributed Systems, 45:1009 – 1021, 2012. ix, 11

[2] O. Sandes, Alba C. M. A. de Melo Edans F. de: *Cudalign: using gpu to accelerate the comparison of megabase genomic sequences.* PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 10:137 – 146, 2010. ix, 12, 15, 16, 17, 18

[3] O. Sandes, Edans F. de: *Algoritmos paralelos exatos e otimizações para alinhamento de sequências biológicas longas em plataformas de alto desempenho.* 2015. ix, 15, 19, 20, 21

[4] Figueirêdo Júnior, Marco Antônio Caldas de: *Comparação paralela de sequências biológicas em múltiplas gpus com descarte de blocos e estratégias de distribuição de carga.* 2021. ix, 1, 22

[5] Filipe Maia Soares, Alba C. M. A. de Melo: *Comparação Paralela de Sequências Biológicas em Múltiplas GPUs com Mecanismo de Tolerância a Falhas.* Tese de Doutoramento, Universidade de Brasília, 2025. ix, 1, 23, 24

[6] SchedMD: *Slurm Workload Manager Documentation*, 2024. `https://slurm.schedmd.com`, Acessado em: 14 jul. 2025. x, 26, 27, 28, 30, 31

[7] Andy B. Yoo, Morris A. Jette e Mark Grondona: *Slurm: Simple linux utility for resource management.* 9th International Workshop, JSSPP 2003, páginas 44 – 60, 2003. x, 26, 28, 29

[8] Luscombe, Nicholas M, Dov Greenbaum e Mark Gerstein: *What is bioinformatics? a proposed definition and overview of the field.* Methods Inf Med ; 40(04): 346-358, 2001. 1

[9] O. Sandes, Guillermo Miranda, Xavier Martorell Eduard Ayguade George Teodoro Alba C. M. A. de Melo Edans F. de: *Masa: a multi-platform architecture for sequence aligners with block pruning.* ACM Transactions on Parallel Computing (TOPC), 2:1–31, 2016. 1, 12, 14

[10] Gotoh, Osamu: *An improved algorithm for matching biological sequences.* Journal of Molecular Biology, 162:705–708, 1982. 1, 8, 9

[11] Acessado em: 23 Nov. 2025. 2

[12] Mount, David: *Bioinformatics: Sequence and Genome Analysis*, volume 2° Edition. Cold Spring Harbor Laboratory Press, 2004. 3, 8

[13] Durbin, Richard, Sean R Eddy Anders Krogh e Graeme Mitchison: *Biological sequence analysis: probabilistic models of proteins and nucleic acids.* Cambridge university press, 1998. 4

[14] Tahami, H., & Fakhravar H.: *A literature review on combining heuristics and exact algorithms in combinatorial optimization.* European Journal of Information Technologies and Computer Science, 2(2):6 – 12, 2022. 4

[15] Saul B. Needleman, Christian D. Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology, 48:443–453, 1970. 5

[16] SMITH, T. F. e M. S. WATERMAN: *Identification of common molecular subsequence.* Journal of Molecular Biology, 147:195–197, 1981. 7

[17] Hisrchberg, D. S.: *A linear space algorithm for computing maximal common subsequences.* Communications of the ACM, 18:341–343, 1975. 9

[18] Eugene W. Myers, Webb Miller: *Optimal alignments in linear space.* Bioinformatics, 4:11–17, 1988. 10

[19] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia Jesús Labarta Luis Martinell Xavier Martorell e Judit Planas: *Ompss: A proposal for programming heterogeneous multi-core architectures.* Parallel Processing Letters, 21(02):173–193, 2016. 12

[20] Chun, Brent N.: *AuthD Home Page.* `http://www.theether.org/authd/`, Acessado em: 14 jul. 2025. 29

[21] Research Services - Boston College: *Job priority & fair share*, 2022. `https://rs.bc.edu/job-priority-fair-share/`, Acessado em: 14 jul. 2025. 31

[22] Dunlap, Chris: *Munge uid 'n' gid emporium*, 2022. `https://github.com/dun/munge/wiki`, Acessado em: 9 Set. 2025. 36

[23] Biotechnology Information, National Center for: *National library of medicine.* `https://www.ncbi.nlm.nih.gov/nuccore`, Acessado em: 17 out. 2025. 44

Apêndice A

Primeira página do artigo Apresentado no WPOS 2025

Execução Paralela Tolerante a Falhas da Aplicação FT-CUDAlign Integrado ao Escalonador SLURM

Gustavo Santana Lima
Departamento de Ciências de Computação
Universidade de Brasília , Brasil
guts.santana@gmail.com

Alba Cristina Magalhaes Alves de Melo
Departamento de Ciências de Computação
Universidade de Brasília , Brasil
alves@unb.br

***Resumo*—O objetivo deste trabalho é projetar, implementar e avaliar a integração da aplicação paralela FT-CUDAlign ao gerenciador de aplicações SLURM, de forma a permitir sua execução escalável e tolerante a falhas em ambientes de clusters e supercomputadores, de modo a demonstrar a viabilidade do uso do SLURM como plataforma de orquestração para o FT-CUDAlign, garantindo maior robustez e flexibilidade em comparações de sequências biológicas de grande porte.**

I. Introdução

A comparação de sequências biológicas é uma das ferramenta de suma importância na biologia molecular e evolutiva. Tendo como objetivo identificar semelhanças entre diferentes genes, proteínas, genomas e até entre diferentes organismos. No entanto, diversos genes possuem milhares e até milhões de nucleotídeos diferentes, de modo que, mesmo utilizando diversas GPUs, essa comparação é lenta e demora muito tempo para ser executada.

A ferramenta a ser integrada é o FT-CUDAlign [1], sendo esse, um comparador de sequências biológicas desenvolvido com o intuito de implementar tolerância a falha, ao software do MASA-CUDAlign. De modo que caso uma ou mais GPUs falhem, o software consiga se readaptar e continuar sua execução com a nova quantidade de GPUs disponíveis.

A ferramente MASA-CUDAlign [2] é a base do FT-CUDAlign. É um comparador de sequências biológicas baseado em cluster de múltiplas GPUs, que executa o algoritmo de Smith-Waterman [3], permitindo a utilização de técnicas de *Block Pruning* para redução de blocos calculados e conexão de dados entre as múltiplas GPUs.

O objetivo deste trabalho é projetar, implementar e avaliar a integração do FT-CUDAlign ao gerenciador de recursos SLURM [4], de forma a permitir sua execução escalável e tolerante a falhas em ambientes de clusters e supercomputadores. De modo, a demonstrar a viabilidade do uso do SLURM como plataforma de orquestração para o FT-CUDAlign, garantindo maior robustez, flexibilidade e desempenho em comparações de sequências biológicas de grande porte.

II. Comparação de Sequências Biológicas

A comparação de sequências biológicas é uma operação que recebe como entrada duas sequências biológicas, podendo essas serem de DNA, RNA, ou proteínas [6]. Baseia-se em alinhamento e escore [5], sendo realizada caractere a caractere e para cada posição a ser comparada, há três resultados possíveis: acerto (*match*), erro (*mismatch*) e salto (*gap*) e cada um com um valor específico. O acerto ocorre quando os caracteres das duas sequências são idênticos, o erro, quando os caracteres da comparação são distintos e o salto ocorre quando há o deslocamento em uma das sequências a fim de permitir um escore maior. O alinhamento por sua vez, é a representação gráfica correspondente a cada acerto, erro ou salto do escore obtido.

Existem dois tipos básicos de comparação: o global e o local. A comparação global consiste em alinhar as duas sequências completas, de maneira que todos os caracteres das duas sequências façam parte do alinhamento. Enquanto a comparação local consiste em alinhar trechos das sequências com o intuito de maximizar a pontuação de trechos e de encontrar genes semelhantes, de modo que apenas os caracteres que ativamente geram a pontuação máxima estejam presentes no alinhamento [7]. Considere as sequências S_0 = `TATCCGTCC` e S_1 = `AGCCGATG`. Um possível resultado da comparação local é apresentado na Figura 1.

S_0:	C	C	G		
	|	|	|		
S_1:	C	C	G		
L:	+1	+1	+1	=	+3

Figura 1. Comparação local entre S_0 e S_1, com $escore = +3$.

Nesse artigo será considerado o algoritmo de Smith-Waterman(SW) [3] para alinhamento local. O primeiro passo é o cálculo de uma matriz de programação dinâmica **H**, de tamanho $(M + 1) \times (N + 1)$, sendo M e N os tamanhos das sequências S_0 e S_1, onde a primeira linha e a primeira coluna devem ser referenciadas com 0 em todos os índices. Os valores das demais células são calculados com a Equação 1.

Apêndice B

Corpo do Script Utilizado

```
#!/bin/bash
#SBATCH -J cudalign_test
#SBATCH --gres=gpu:all
#SBATCH -p partition
#SBATCH -o "Refatorado.txt"
#SBATCH -t 00:30:00


#define diretrios usados
ftDIR=/home/laicoadm/Documentos/Filipe/FT-dynBP
sharedDIR=/home/laicoadm/Documentos/Gustavo-Slurm/CUDA/SharedDir
folders=(2-5M 3-7M 4-10M) #Pastas com as sequncias utilizadas por execucao

#Define parametros de rede e GPU
total_gpus=0 #Pegar o total de gpus para o arquivo de configuracao
nodes_gpus=() #pegar a quantidade de gpus de cada maquina para o balancer
nodes_ips=()
IPs=""
GPUs=""
BasePort=5234
PortArray=""
SEQ=""
SPLIT=""
GFLOPS=""


### DEF Functions that will be used
#getData:Pega a quantidade de gpus, os IPs e faz o PortArray para arquivo FT-CUDAlign
getData() {

for n in $(scontrol show hostnames $SLURM_NODELIST); do
    ip=$(srun -N1 -w $n hostname -I | awk '{print␣$1}')
    nodes_ips+=($ip)
    gpus=$(srun -N1 -w $n nvidia-smi -L 2>/dev/null | wc -l)
    nodes_gpus+=($gpus)
    total_gpus=$((total_gpus+gpus))
    for ((j=0; j<gpus; j++)); do
        if [[ -z "$IPs" ]]; then
            IPs="$ip"
            GPUs="$j"
            PortArray="$BasePort"
        else
            IPs="$IPs,$ip"
            GPUs="$GPUs,$j"
            PortArray="$PortArray,$((BasePort+j))"
        fi
    done
done

}

#writeFile:Escreve o arquivo de configuracao com base no folder selecionado
writeFile(){
```

```
#$1 -> folder da sequencia

path=$sharedDIR/sequences/$1
configFile="$ftDIR/Configs/config$1"
files=("$path"/*.fasta)
str1="${files[0]}"
str2="${files[1]}"
SEQ="$str1,$str2"
#Cria o arquivo config referente a sequencia do folder
cat <<EOF > $configFile
GPUS $total_gpus
BREAK 1
IP $IPs
PORT $PortArray
GPU# $GPUs
BP 0
SPLIT $SPLIT
SEQ $SEQ
WAIT 0
MODEL dynamic
PROG ./dynamic
GFLOPS $GFLOPS
EOF

}

#runBalancer: Executa os Balancer, forando a execucao em uma gpu e espera final da execucao
runBalancers(){
for n in $(scontrol show hostnames $SLURM_NODELIST); do
       gpus=$(srun -N1 -w $n nvidia-smi -L 2>/dev/null | wc -l)
       for ((count=0;count<gpus;count++)); do
       (
              srun -N1 -w $n --gres=gpu:1 ./balancer $count $path
              ret=$?
              echo "Retorno␣foi␣$ret"
       ) &
       done
done
}



#Inicio da execucao do script
#Executa o getData
getData

#necessita do getData para saber o total de gpus participantes execucao
SPLIT=$(echo $(yes 30 | head -n $total_gpus) | tr '␣' ',')
GFLOPS=$(echo $(yes 14000 | head -n $total_gpus) | tr '␣' ',')


mkdir -p "$ftDIR/Configs"
for folder in "${folders[@]}"; do
       writeFile $folder
       cd $ftDIR
       runBalancers
       ./controller $configFile $path
       ret=$?
       echo "Retorno␣do␣Controller␣foi␣$ret"
done
```

(Botão de Instalação)