TRABALHO DE GRADUAÇÃO

Cloudificação de Aplicações Multi-GPU de Bioinformática no *AWS ParallelCluster* da nuvem *Amazon*

Filipe Maia Soares

Brasília, 4 de Novembro de 2021

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

Cloudificação de Aplicações Multi-GPU de Bioinformática no *AWS ParallelCluster* da nuvem *Amazon*

Filipe Maia Soares

Relatório submetido como requisito parcial de obtenção de grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Dra. Alba Cristina Magalhães Alves de Melo, CIC/UnB
Orientador

Prof. Dr. Bruno Luiggi Macchiavello Espinoza, CIC/UnB
Examinador interno

Prof. Dra. Carla Maria Chagas e Cavalcante Koike, CIC/UnB
Examinador interno

Brasília, 4 de Novembro de 2021

FICHA CATALOGRÁFICA

MAIA SOARES, FILIPE
Cloudificação de Aplicações Multi-GPU de Bioinformática no AWS ParallelCluster da nuvem Amazon

[Distrito Federal] 2021.

vii, 70p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2021). Trabalho de Graduação – Universidade de Brasília.Faculdade de Tecnologia.

1. Computação em Nuvem 2.Comparação de Sequências
3. GPU

I. Mecatrônica/FT/UnB II. Controle e Automação

REFERÊNCIA BIBLIOGRÁFICA

MAIA S., FILIPE, (2021). Cloudificação de Aplicações Multi-GPU de Bioinformática no AWS ParallelCluster da nuvem Amazon. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-n°05, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 81p.

CESSÃO DE DIREITOS

AUTOR: Filipe Maia Soares

TÍTULO DO TRABALHO DE GRADUAÇÃO: Cloudificação de Aplicações Multi-GPU de Bioinformática no AWS ParallelCluster da nuvem Amazon.

GRAU: Engenheiro ANO: 2021

Universidade de Brasília

Campus Universitário Darcy Ribeiro - Asa Norte

70910-900 Brasília – DF – Brasil.

Agradecimentos

Gostaria de agradecer a professora Dra. Alba Melo pela orientação, paciência e por todos os ensinamentos. Sem a sua ajuda nada disso seria possível. Agradeço também ao Dr. Marco Figueirêdo por estar sempre disposto a ajudar e a compartilhar conhecimento. Ao doutorando Walisson Sousa, obrigado pela amizade, auxílio e apoio dado ao longo dessa jornada. Gratidão ao projeto CNPq/AWS $440014/2020 - 4$*, que possibilitou o acesso, configuração e uso do AWS ParallelCluster.*

A meus pais e meu irmão, muito obrigado por me darem todo o suporte de que precisei durante o curso e, especialmente nesse último ano, em que realizei o TCC. Aos meus amigos, obrigado pelas risadas, brincadeiras e por sempre estarem presentes nos momentos difíceis.

Filipe Maia Soares

RESUMO

A comparação de sequências biológicas é uma das principais áreas de estudo no campo da bioinformática, pois auxilia no entendimento das funcionalidades e constituição dos organismos. Os algoritmos exatos utilizados para realizar tais comparações apresentam complexidade quadrática de tempo, de maneira que a comparação de longas sequências exige a utilização de ambientes de alto desempenho, preferencialmente os que paralelizam a aplicação, com o intuito de reduzir ainda mais o tempo de execução. O *MASA-CUDAlign* é um exemplo de ferramenta que atende a esses requisitos e é o escopo de estudo deste trabalho. Em especial, a versão *Static-MultiBP* atingiu um impressionante desempenho de $82,82$ *TCUPS*. Além disso, a computação em nuvem vem ganhando muita visibilidade e utilização nos últimos anos devido à série de vantagens que proporciona ao usuário. Nesse contexto, o objetivo desta monografia consiste em realizar a "cloudificação" do *MASA-CUDAlign*, ou seja, prover uma solução para executar esta ferramenta no cluster virtual da nuvem da *Amazon*, o *AWS ParallelCluster*.

Palavras Chave: computação em nuvem, comparação de sequências biológicas, *MASA-CUDAlign*, *AWS ParallelCluster*

ABSTRACT

Biological sequence alignment is one of the main areas of study in the field of bioinformatics, as it helps to understand the functionalities and constitution of the organisms. The exact methods used to perform such comparisons have quadratic time complexity, so that the comparison of long sequences requires high-performance environments, specially those that parallelize the application, in order to further reduce execution time. *MASA-CUDAlign* is a tool that meets these requirements and is the scope of study of this work. In particular, the *Static-MultiBP* version achieved an impressive performance of 82.82 *TCUPS*. In addition, cloud computing has gained a lot of visibility and use in recent years due to the series of advantages it provides to the user. The objective of this monograph is to cloudify the *MASA-CUDAlign* tool, i.e., to provide a solution to run this tool in the *AWS ParallelCluster*, which is the virtual cluster of the *Amazon* cloud.

Keywords: *cloud computing*, *biological sequence alignment*, *MASA-CUDAlign*, *AWS ParallelCluster*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

A Bioinformática é uma área que utiliza técnicas computacionais para auxiliar no estudo e na análise de dados biológicos, podendo ser desdobrada em três frentes: armazenamento e consulta de dados; desenvolvimento de ferramentas para o tratamento de dados; e utilização de ferramentas para analisar e interpretar os dados [6]. Uma das principais aplicações da área consiste na comparação de sequências biológicas como, por exemplo, de proteínas, DNA ou RNA, possibilitando-se a geração do escore (pontuação) e do alinhamento, posteriormente utilizados para analisar a similaridade entre as sequências. As ferramentas de comparação podem utilizar dois tipos de algoritmos: os exatos, que sempre obtêm o escore ótimo e um ou mais alinhamentos que o produzem; e os heurísticos, que obtêm escore e alinhamentos relevantes, mas não necessariamente ótimos [7]. O foco desta monografia é nos algoritmos exatos, que utilizam programação dinâmica e possuem complexidade quadrática de tempo.

As sequências biológicas podem ser extremamente longas, da ordem de milhões caracteres, ou de pares de base. Isto, aliado ao custo quadrático de tempo de um algoritmo exato, pode tornar a sua comparação um grande desafio, fazendo assim necessário o uso de ambientes de alto desempenho. Para tentar reduzir o tempo de execução busca-se paralelizar a aplicação, subdividindo a execução em múltiplas unidades de processamento. As Unidades de Processamento Gráfico (*Graphics Processing Unit* – GPUs), por exemplo, vêm sendo amplamente utilizadas neste tipo de aplicação, uma vez que são capazes de atingir altas taxas de processamento, além de serem muito mais simples de se programar, em comparação com outras soluções que utilizam *hardware*, como é o caso dos Arranjos de Portas Programáveis em Campo (*Field Programmable Gate Array - FPGAS*) [8]. Ferramentas de comparação de sequências biológicas, como o *SW#* [9], o *CUDASW+* [10] e o *MASA-CUDAlign* [4] constituem algoritmos exatos, que utilizam GPUs em seu processamento.

A ferramenta *MASA-CUDAlign* é o escopo de estudo desta monografia. A primeira versão, *MASA-CUDAlign 1.0*, utiliza apenas uma GPU em sua execução e tem como objetivo obter o escore ótimo e a respectiva posição na matriz de similaridade. Para tanto, a principal premissa da ferramenta é paralelizar a aplicação por meio da técnica do *wavefront* (Seção 4.1). A versão seguinte, *MASA-CUDAlign 2.0* (Seção 4.2), além de calcular o escore ótimo e sua posição na

matriz, obtém um dos alinhamentos ótimos. Sua execução é divida em 6 estágios: o primeiro realiza os mesmos procedimentos adotados no *MASA-CUDAlign 1.0*; os estágios de 2 a 5 são responsáveis por obter o alinhamento e o último permite a visualização gráfica e textual do alinhamento.

O *MASA-CUDAlign 2.1* (Seção 4.3) traz a ideia do *Block Pruning*, que consiste em podar a matriz de similaridade, eliminando regiões que são incapazes de gerar um alinhamento ótimo. Já na versão 3.0 (Seção 4.4), a ferramenta traz a possibilidade de se utilizar múltiplas GPUs para executar o estágio 1, que é o mais demorado. A versão *MASA-CUDAlign 4.0* (Seção 4.5), por sua vez, inclui a utilização das múltiplas GPUs nos estágios de 2 a 4, que são responsáveis por realizar o *traceback* (construção do alinhamento).

A versão mais atual da ferramenta é o *Static-MultiBP* (Seção 4.6), que busca acelerar a execução combinando a técnica de *Block Pruning*, do *MASA-CUDAlign 2.1*, com a utilização de múltiplas GPUs, do *MASA-CUDAlign 3.0*. Esta versão foi escolhida para ser executada em um *cluster* de GPUs NVIDIA Volta ($512V100$) e obteve-se um desempenho expressivo de $82,82$ *TCUPS*, sendo esse: "*...o melhor desempenho obtido na literatura de ferramentas de comparação de sequências em GPU que usam o algoritmo Smith-Waterman e suas variantes, e o melhor desempenho entre ferramentas que comparam sequências longas (em qualquer dispositivo).*"[5]. Esse desempenho excepcional foi obtido em um *cluster* dedicado da empresa *NVidia*, com custo estimado em milhões de dólares e com equipe de manutenção trabalhando 24 horas. Como a maioria dos usuários não têm acesso a tal plataforma, para esses casos é necessário explorar outras alternativas, como a computação em nuvem. A computação em nuvem vem ganhando muita visibilidade e utilização nos últimos anos, trazendo uma série de vantagens, como: oferecer serviços sob demanda; elasticidade rápida, que garante ao usuário a capacidade de alocar e desalocar recursos rapidamente, de acordo com a necessidade; e economia com a infraestrutura, uma vez que os recursos são todos virtuais [11]. Atualmente, várias empresas oferecem este serviço, como a *Amazon Web Service (AWS)*, a *Google Cloud Platform* e a *Microsoft Azure*, sendo a *AWS* uma das que possui mais variedade de recursos a um custo razoável.

Nesse contexto, a ideia de executar a ferramenta *MASA-CUDAlign* na nuvem se torna, à princípio, muito interessante. O problema dessa abordagem é que um *cluster* possui ambiente com poder de computação e banda de rede dedicados, diferentemente do que ocorre em nuvem. Além disso, como o *MASA-CUDAlign* é uma uma aplicação na qual os nós se comunicam com frequência, seu desempenho seria possivelmente muito prejudicado no ambiente de nuvem, que utiliza da virtualização e de rede compartilhada. Dessa maneira, para simular um *cluster* na nuvem necessita-se de uma ferramenta que proporcione tais garantias. O *AWS ParallelCluster* (Seção 2.4.2) é uma ferramenta da *AWS* que permite a execução de aplicações de alto desempenho por meio da criação de *clusters* virtuais, a partir dos recursos fornecidos pela empresa. Esta ferramenta atende aos requisitos propostos na medida que proporciona a utilização dedicada dos nós, assim como uma rede privada em nuvem, chamada de *VPC* (Seção 2.4.4).

O objetivo do presente trabalho de graduação consiste em realizar a "cloudificação" do *MASA-CUDAlign*, ou seja, prover uma solução para executar esta ferramenta no *AWS ParallelCluster*. Para tanto, foi necessário subdividir este objetivo principal em objetivos menores. O primeiro

deles consistiu em criar um *cluster* na *AWS* utilizando a ferramenta *AWS ParallelCluster*. O objetivo seguinte buscou integrar e executar a ferramenta *MASA-CUDAlign* no *ParallelCluster*. Por fim, objetivou-se coletar e analisar os resultados de desempenho e custo no *ParallelCluster*.

O presente trabalho está organizado da seguinte maneira: O Capítulo 2 se inicia com um breve histórico da computação em nuvem, descrevendo suas principais influências, características e algumas das tecnologias que proporcionaram seu desenvolvimento e se encerra detalhando a nuvem da *AWS*. Já o Capítulo 3 aborda a comparação de sequências biológicas, bem como os principais algoritmos utilizados para compará-las. O Capítulo 4 tem seu foco na ferramenta *MASA-CUDAlign*, explicando o seu funcionamento em cada uma de suas versões. No Capítulo 5, de Projeto, é descrita a solução proposta nesta monografia para configurar e executar a ferramenta *MASA-CUDAlign* no *ParallelCluster*. No Capítulo 6 são analisados os resultados obtidos, com base no tempo e no custo, para as execuções realizadas. Por fim, o Capítulo 7 conclui a monografia, avaliando o que foi feito, comentando os resultados obtidos e trazendo propostas de trabalhos futuros.

Capítulo 2

Computação em Nuvem

O presente capítulo discorre sobre a origem da computação em nuvem, algumas das tecnologias que possibilitaram a sua criação e por fim descreve resumidamente suas principais características. Na Seção 2.1 é abordado um breve histórico da computação em nuvem, bem como as tecnologias de computação em *cluster* e em grade. A Seção 2.2 se refere à técnica de virtualização, fundamental para o desenvolvimento da computação em nuvem. Em seguida, a Seção 2.3 explica resumidamente o conceito de computação em nuvem, bem como suas principais características. Por fim, a Seção 2.4 aborda especificamente a nuvem da *AWS* e algumas de suas ferramentas e serviços.

2.1 Histórico da Computação em Nuvem

A computação em nuvem é um paradigma recente no âmbito da tecnologia, que consiste na provisão de recursos e aplicações computacionais através da Internet. Apesar de sua grande popularização ter se iniciado há apenas alguns anos, a ideia em si, bem como as tecnologias nela não são tão recentes. As primeiras noções do que conhecemos hoje como computação em nuvem surgiram na década de 1960, com John McCarthy. Ele acreditava que o avanço tecnológico causaria um grande barateamento da computação até que esta se tornaria um serviço público, assim como as linhas telefônicas da época, criando assim um novo ramo no âmbito industrial [12]. No entanto, a infraestrutura de hardware e software da época ainda não era capaz de implementar uma ideia como esta. Apenas anos à frente surgiriam as premissas, tecnologias e protocolos que influenciaram e proporcionaram o surgimento da computação em nuvem na forma atual. Os principais são: computação em grade e *cluster*; Internet, mais especificamente o protocolo World Wide Web (www); e a técnica de virtualização. A computação em *cluster* serviu, entre outros fatores, como uma forma de se obter maior poder de processamento utilizando vários computadores de menor desempenho ao invés de um único supercomputador tradicional. Isto permite a execução de processos mais complexos por um custo menor [13].

Nesta técnica, os gabinetes das máquinas são interconectados por uma rede local, rápida e de alto desempenho, que proporciona a execução paralela de aplicações. Dessa maneira, o processamento das tarefas se torna muito mais eficiente, uma vez que este sistema é capaz de

balancear e distribuí-las entre os dispositivos, que agora as executam simultaneamente.

Apesar de haver várias máquinas físicas, esse sistema é visto pelo usuário como uma única máquina. Outro ponto positivo é a tolerância à falhas, uma vez que o não funcionamento de um dos nós pode ser compensado com a redistribuição de processos entre os restantes.

De maneira análoga, a computação em grade também consiste na conexão de vários computadores, porém eles podem possuir arquiteturas distintas, pertencer a diversos domínios administrativos e estar espalhados em vários locais. Nesse contexto, apesar de utilizar os recursos, o usuário não necessariamente sabe onde eles residem fisicamente.

Foi a partir dessa perspectiva que surgiu o nome computação em grade, do inglês *grid computing*, que remete às *power grids*, que são as redes de energia elétrica. Assim como os recursos computacionais da computação em grade, a energia é provida sob demanda e, apesar de parecer ser proveniente de uma única fonte, ela costuma ser formada por várias outras que se uniram.

Uma das definições que enfatiza esse aspecto foi criada por Ian Foster e Carl Kesselman, na década de 1990. Ela diz que: *"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities"* [14]. Nota-se que a ideia de provisionamento de recursos em muito se assemelha ao conceito de computação em nuvem e por isso muitos autores consideram que ela, de fato, descende da computação em grade.

Alguns anos depois, devido ao desenvolvimento da computação em grade e de suas aplicações, sua definição foi sendo alterada e, nos anos 2000 Foster e Kesselman propuseram uma nova, englobando três pontos-chave. São eles [14]:

- Coordenar recursos que não estão submetidos a um controle centralizado
- Utilizar protocolos e interfaces padrão, abertos e de uso genérico
- Entregar serviços de qualidade não trivial

A coordenação de recursos descentralizados sugere que a computação em grade deve envolver diversas organizações. Nesse contexto, para que uma organização faça parte da grade e possa utilizar seu poder computacional, ela necessita acordar a contribuição que fará em termos de recursos e, com base nisso, o quanto ela vai poder usufruir.

O uso de interfaces padrão afirma que, para que a computação em grade possua uma interface transparente, apesar de ter recursos heterogêneos ela necessita de protocolos padrão, *middlewares*, *toolkits* e outras tecnologias que porporcionem tais aspectos. Essas tecnologias também são fundamentais para garantir segurança e estabelecer diferenças entre a utilização dos recursos no âmbito global (da grade) e local (da instituição que provém o recurso) [15].

Por fim, a computação em grade deve ser capaz de fornecer serviços que possibilitem a execução das aplicações considerando os requisitos de desempenho impostos pelo usuário.

Vale ressaltar que, como a computação em grade deve fornecer recursos para vários usuários da

rede, é necessário estabelecer um agendamento da ordem na qual os *jobs*, de cada um serão executados. Quanto maiores forem os requisitos, maior será a espera pela execução da aplicação, uma vez que ela só será realizada quando os recursos necessários estiverem disponíveis pela quantidade de tempo especificada para alocação. Após essa execução, os recursos podem ser simplesmente realocados para outras atividades [15].

2.2 Virtualização

2.2.1 Desenvolvimento da Virtualização

Virtualização significa abstrair recursos computacionais, como redes, CPU, memória etc, criando uma versão virtual destes [16]. Essa técnica facilita a gerência e pode melhorar a performance na utilização de recursos. Uma de suas principais aplicações consiste na criação de máquinas virtuais (MVs), instâncias de uma única máquina física, que são encapsuladas e dão ao usuário a impressão de ter acesso direto à ela.

A primeira utilização de máquinas virtuais data da década de 1960 e foi desenvolvida pela IBM em um de seus computadores *mainframes* para proporcionar a utilização de vários sistemas operacionais de maneira concorrente [17]. Na época, o hardware era extremamente caro e, por isso, o desenvolvimento dessa tecnologia foi muito importante, pois permitia que múltiplos usuários executassem suas aplicações em diversos sistemas operacionais, na mesma máquina física, de maneira concorrente. Isso possibilitou que os grandes *data centers* condensassem a quantidade de servidores físicos aumentando a de servidores virtuais.

As vantagens disso se refletem no menor gasto energético e de manutenção de hardware. Além disso, como as máquinas virtuais são independentes, não há risco de que as ações de um programador interfiram, de alguma maneira, na máquina virtual do outro.

O avanço tecnológico nas décadas de 1970 e 1980 levaram a um grande barateamento do hardware, o que causou uma drástica redução na utilização da virtualização. Entretanto, a partir da década de 1990 foram propostas diversas aplicações que utilizavam esta tecnologia, revivendo a sua utilização e, mais recentemente, ela se torna essencial no contexto de computação em nuvem.

Nesse contexto, a virtualização de recursos, como o *storage*, é fundamental para a computação em nuvem, uma vez que ela possibilita reunir a capacidade de armazenamento de vários dispositivos, apresentando-os como se fossem apenas uma unidade. Analogamente, é possível virtualizar um dispositivo de amplo armazenamento em subdivisões menores de memória e apresentar cada um deles como uma instância. Esta característica facilita a administração e compartilhamento de recursos que devem ser dinamicamente providos aos usuários [18].

Outro atrativo dessa tecnologia consiste na possibilidade de executar diferentes sistemas operacionais(SOs) em uma única máquina física, um SO em cada MV, dando assim mais possibilidades de utilização aos usuários. Entretanto, é necessário prover um ambiente apropriado para que as máquinas virtuais possam ser instanciadas e evitar que ambos os sistemas operacionais tentem obter o controle do hardware simultaneamente. Por esse e outros motivos foi criado o monitor de

máquina virtual (MMV), ou *hypervisor*.

2.2.2 Hypervisor

O monitor de máquina virtual, também chamado de *hypervisor* é uma camada de software que fica entre o hardware e as máquinas virtuais. A Figura 2.1 ilustra um modelo genérico de *hypervisor*.

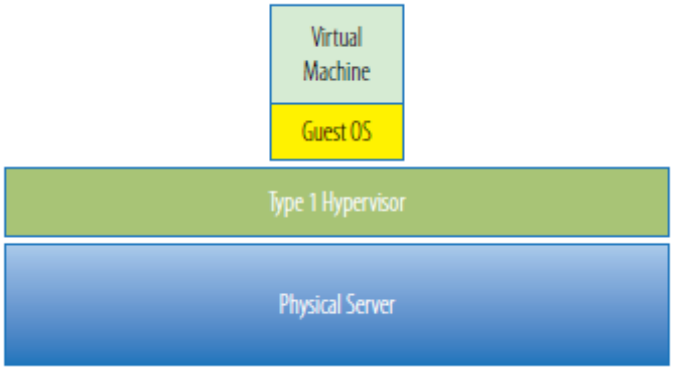


Figura 2.1: Exemplo de um monitor de máquina virtual genérico. [1]

Segundo Popek e Goldberg, o *hypervisor* possui três principais funções [1]:

- Prover um ambiente idêntico ao do hardware. Isto significa que os recursos disponíveis à maquina virtual devem ser apresentados e providos à ela de maneira adequada.
- Prover um ambiente com uma redução mínima no desempenho. Ou seja, o tempo de resposta deve ser pequeno e sem utilizar mais recursos do que o esperado para a respectiva tarefa.
- Ter controle do sistema de recursos. Ou seja, evitar que a MV utilize mais recursos do que foi reservado para ela e impedir o acesso entre máquinas virtuais.

De maneira simplificada, o *hypervisor* é encarregado de administrar os recursos para as máquinas virtuais. Para garantir isso, é necessário todas as instruções em modo protegido, enviadas pelas máquinas virtuais sejam interceptadas e tratadas por ele, que as executa e retorna o resultado da requisição. Além disso, o hypervisor necessita escalonar adequadamente as tarefas das máquinas virtuais, de maneira a prevenir que as requisições de uma delas sejam negligenciadas por muito tempo.

2.2.2.1 Tipos de Hypervisor

Há duas classes principais de hypervisor: tipo 1 e tipo 2. No tipo 1, o hypervisor é executado diretamente acima da camada de hardware, como mostra a Figura 2.2, e por isso esse modelo

também é chamado de *bare-metal*. Como o hypervisor possui acesso direto ao hardware, há pouco *overhead*, tornando esta implementação mais rápida e, dessa maneira permitindo o acesso de mais máquinas vituais nesse servidor. Além disso, esta implementação é mais segura, uma vez que as máquinas virtuais não possuem um sistema operacional tradicional que as une, tornando-as totalmente independentes.

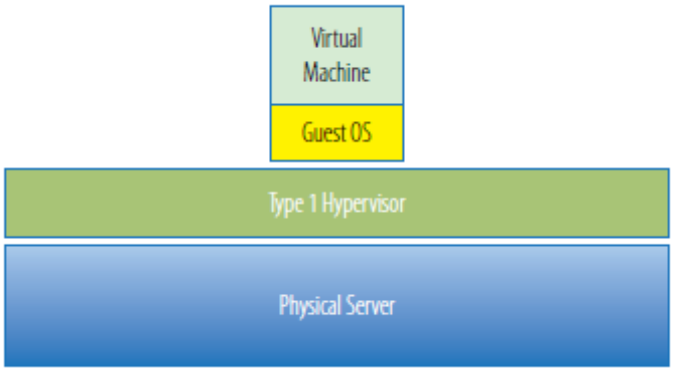


Figura 2.2: Exemplo de um monitor de máquina virtual do tipo 1. [1]

Já no tipo 2, representado na Figura 2.3, o hypervisor é executado abaixo das máquinas virtuais e acima do sistema operacional tradicional. Nessa estrutura, a instalação e utilização do hypervisor são mais simples, uma vez que o sistema operacional tradicional já administra todos os recursos disponíveis. A principal desvantagem desse modelo reside no fato de que as instruções protegidas devem ser interceptadas, tanto pelo hypervisor quanto pelo sistema operacional antes de seguir para o hardware. Da mesma maneira, o retorno da operação, segue o mesmo passo-a-passo, porém no sentido inverso. Esse conjunto de interrupções causa grande overhead, tornando essa abordagem mais lenta que a primeira. Além disso, esta estrutura também utiliza mais hardware, uma vez que inclui tanto as ações do hypervisor, quanto as do sistema operacional tradicional.

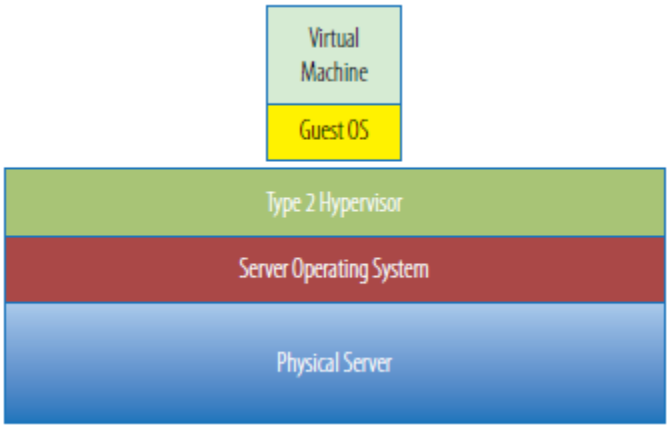


Figura 2.3: Exemplo de um monitor de máquina virtual do tipo 2. [1]

2.2.3 Vantagens da Virtualizaçao

Analisando os pontos principais abordados nas seções 2.2.1 e 2.2.2, é possível resumir as principais vantagens da virtualização em alguns tópicos [19]:

- Agregar e Compartilhar Recursos: como dito anteriormente, é possível reunir recursos de vários dispositivos e apresentá-los como um só, assim como dividir recursos de um dispositivo e apresentá-lo como várias unidades menores. Além disso a quantidade de recursos da máquina virtual é facilmente reconfigurável.
- Realocação Eficiente: a virtualização de recursos permite grande flexibilidade em (des)alocá-los de acordo com a necessidade.
- Consolidação de Servidores Físicos: permite uma utilização mais eficiente do hardware disponível, que é compartilhado com vários usuários. Isso permite redução de gastos com energia e manutenção.
- Isolamento entre MVs: proporciona segurança e independência entre usuários.
- Encapsulamento de MVs: pelo fato de máquinas virtuais serem constituídas por um conjunto de arquivos reconfiguráveis, elas podem ser rapidamente transferidas entre servidores como se fossem arquivos comuns.

2.3 Visão Geral de Computação em Nuvem

A ideia de computação em nuvem, como dito na Seção 2.1, já tinha sido concebida desde a década de 1960, porém ainda não havia recebido este nome. Foi apenas em uma conferência no ano de 2006 que o CEO da *Google*, Eric Schmidt utilizou e posteriormente popularizou o termo computação em nuvem, "cloud computing". Em sua fala ele se referia a importância deste novo paradigma para a computação [20]: *"I don't think people have really understood how big this opportunity really is. It starts with the premise that the data services and architecture should be on servers. We call it cloud computing as they should be in a "cloud"somewhere".*

Já no final de 2006, grandes empresas, como a *Google* e a *Amazon* começaram a prover tais serviços de computação via internet, utilizando o termo de Schmidt, e descreveram o uso da computação em nuvem da seguinte maneira: *"It is the movement of application services onto the Internet and the increased use of the Internet..."* [20].

Entretanto, esta definição desagradou vários especialistas da área, que alegavam ser muito genérica e pouco descritiva. Isso motivou alguns autores, como Foster, Buya, Vaquero dentre outros, a criar suas próprias definições. Como cada um descreve a computação em nuvem colocando sua perspectiva acerca do tema, que é relativamente novo, não há uma definição padronizada do que realmente é computação em nuvem, nesse contexto, uma das mais utilizadas e aceitas atualmente é a da NIST, que diz o seguinte [11]: *"Cloud computing is a model for enabling ubiquitious, convenient, on-demand network access to a shared pool of configurable computing resources (e.g.,*

networks,servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.".

Esta definição trata de alguns pontos-chave, que descrevem a computação em nuvem como conhecemos hoje. São eles:

- Serviço sob Demanda: o usuário pode requisitar mais recursos de acordo com sua necessidade e de maneira automática, via internet.
- Amplo Acesso à Rede: o usuário pode acessar os recursos computacionais em qualquer local que tenha acesso à rede e sob diferentes plataformas, como desktops, notebooks, celulares, tablets etc.
- Fonte de recursos: Os recursos físicos e virtuais providos ao usuário são dinamicamente alocados e desalocados por servidores de várias regiões, de acordo com sua demanda.
- Elasticidade Rápida: Os recursos são rapidamente liberados ao usuário e podem ser alocados e removidos com facilidade.
- Medição de Serviço: os servidores têm controle da "quantidade" de recursos utilizados pelo usuário e tornam essas informações transparentes.

A computação em nuvem vêm ganhando mais visibilidade e utilização tanto devido a esses fatores, quanto pela questão econômica. Muitas empresas optam pela computação em nuvem para evitar altos investimentos em infraestrutura. Além disso, o modelo é capaz de proporcionar ao usuário amplo acesso a recursos, que podem ser pagos de acordo com sua utilização e apenas no final do período contratado.

2.3.1 Modelos de Serviço

Há três modelos de serviço básicos fornecidos pela computação em nuvem. Cada um deles proporciona recursos de acordo como nível de abstração desejado pelo usuário [21].

- Software as a Service (SaaS): dá ao usuário acesso às aplicações do provedor, que rodam na nuvem. Dessa maneira, não é ncessário se preocupar com a manutenção do sistema. As aplicações são acessadas via web, o que facilita seu acesso.
- Platform as a Service (PaaS): neste modelo, o servidor provê uma plataforma em nuvem, que proporciona um ambiente para o usuário desenvolver suas aplicações, sem necessariamente saber o quanto de recursos está utilizando.
- Infrastructure as a Service (IaaS): modelo de menor nível de abstração. O servidor provém recursos virtualizados ao usuário, como armazenamento, capacidade de processamento, acesso à rede etc. O usuário, por sua vez, é quem administra tais recursos.

A escolha de qual serviço será contratado depende da aplicação desejada e, consequentemente, da necessidade do usuário.

2.3.2 Modelos de Implantação

Há quatro modelos de implantação de nuvem, cada um deles especifica a acessibilidade do serviço.

- Nuvem Pública: neste modelo, todas as aplicações e serviços são abertos e facilmente acessíveis ao público.
- Nuvem Privada: todo o acesso à infraestrutura da nuvem é restrito a uma empresa. A administração da estrutura pode ser realizada pela própria empresa ou por outras.
- Nuvem Híbrida: consiste em uma mistura dos modelos público e privado. Dessa maneira, para aplicações críticas utiliza-se a nuvem privada, enquanto que para as não críticas utiliza-se a pública.
- Nuvem Comunitária: o acesso à sua infraestrutura é restrito a um conjunto de empresas. Assim como no modelo público, a administração pode ser feita por qualquer uma das empresas da nuvem comunitária ou por uma terceira.

2.4 A Nuvem AWS

A *Amazon* foi criada em 1994 e iniciou sua atuação com a venda de livros. Ao longo dos anos ela foi se especializando no ramo de tecnologia e atualmente é uma das principais referências no campo de computação em nuvem, ao lado de outras grandes empresas como a *Azure*, da *Microsoft*, e a *Google Cloud Platform*. Foi em meados dos anos 2000 que a empresa iniciou a instalação de sua infraestrutura ao longo de vários locais do mundo. Além disso, para facilitar a administração de seus recursos computacionais, a *Amazon* investiu consistentemente em soluções automatizadas, para solucionar os problemas mais comuns de infraestrutura [22].

Posteriormente, a *Amazon* percebeu que sua infraestrutura bem organizada poderia ser rentável por meio do provisionamento de recursos como um serviço e assim, em 2006, foi criada a *Amazon Web Service (AWS)*. Uma de suas premissas, que a destacou de outras empresas competidoras, foi o seu modelo de custeio, no qual o usuário pode utilizar o quanto quiser dos recursos computacionais e apenas paga de acordo com a sua utilização.

A *Amazon Web Service* (*AWS*) é uma plataforma *web* de computação em nuvem, criada pela *Amazon*, que oferece os três principais modelos de serviço: *IaaS*, *PaaS* e *SaaS*. Suas ferramentas e soluções são amplamente utilizadas por diversas organizações e instituições, e seus *datacenters* estão espalhados por vários países. Algumas das principais utilidades fornecidas por ela são poder computacional, armazenamento de dados e redes de fornecimento, entrega e distribuição de conteúdo.

2.4.1 Elastic Compute Cloud (EC2)

O *Elastic Compute Cloud* (*EC2*) é um serviço *web*, criado em 2006 pela nuvem da *Amazon* (*AWS*), que proporciona ao usuário poder computacional redimensionável [23]. Dessa maneira, o usuário pode escolher quais máquinas (instâncias) deseja utilizar, assim como algumas de suas configurações de ambiente, tais como espaço de armazenamento, tipo de rede, sistema operacional, dentre outras.

O *EC2* disponibiliza uma ampla gama de instâncias, cada uma com suas respectivas especificações, como, tipo e quantidade de GPUs e VCPUs, largura da banda de rede, tipo de armazenamento, dentre outras. Além disso, tais instâncias são agrupadas em famílias de acordo com sua especialidade, por exemplo, aquelas com maior capacidade de processamento gráfico pertencem à família g, enquanto as que são otimizadas para operações em memória pertencem à família m e assim sucessivamente. Essa classificação auxilia a escolha de instâncias que se mais adequam ao projeto e que podem ser facilmente alocadas ou desalocadas, bem como os recursos utilizados por elas.

Ainda no *EC2* existe um serviço chamado de *Elastic Block Store* (*EBS*), que consiste em um tipo de armazentamento, projetado pela *Amazon*, para trabalhar em conjunto com as instâncias do *EC2*. Dessa maneira, ao reservar instâncias é possível alocar vários volumes de *EBS* de tamanho desejado e mantê-los ativos, ou não, após o encerramento da instância. A vantagem de manter esses volumes ativos consiste em não descartar os arquivos armazenados nele durante a execução da instância. Porém, uma solução alternativa e mais interessante consiste em transferir os arquivos do *EBS* para o *S3* (explicado na Seção 2.4.3), uma vez que o *S3* utiliza apenas o espaço necessário para armazenar seus arquivos, além de ser mais barato.

2.4.2 Parallel Cluster

O *Parallel Cluster* é uma ferramenta, que possibilita a computação de aplicações de alto desempenho por meio da utilização de *clusters* (Seção 2.1), formados a partir de instâncias da *AWS* [24]. A Figura 2.4 apresenta, de maneira simplificada, a sua arquitetura.

O funcionamento da ferramenta ocorre da seguinte maneira. Primeiramente, o administrador preenche, em um arquivo de configurações, as especificações do *cluster* e solicita, ao servidor da *AWS*, a sua criação. Algumas dessas configurações são: credenciais do usuário, escalonador de tarefas, sistema operacional, quantidade de instâncias mínima e maxima, e tipo de instância utilizada para o mestre e para os escravos, também chamados de nós computacionais.

Em seguida, a ferramenta denominada *Cloud Formation* irá iniciar a criação do *cluster*, segundo as definições dadas pelo usuário. O *cluster* sempre deve possuir pelo menos um mestre e nenhum ou mais nós. O mestre é responsável por atribuir tarefas aos nós e estes, por sua vez, tem a função de executar tais tarefas simultaneamente, proporcionando o paralelismo da aplicação.

Após a inicialização do *cluster*, o usuário pode conectar diretamente seu terminal ao do mestre do *cluster* por meio do protocolo *SSH*. Uma vez conectado ao mestre, o usuário tem acesso aos

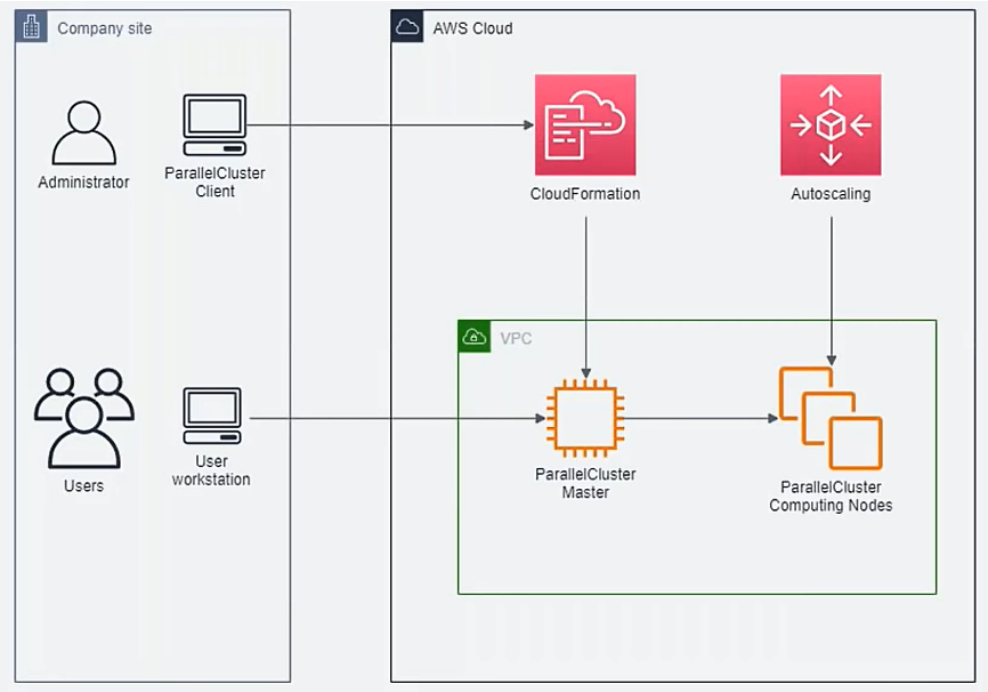


Figura 2.4: Arquitetura simplificada do *Parallel Cluster*. Retirada de [2].

demais nós do *cluster*, também utilizando o protocolo *SSH*.

Uma vez conectado ao *cluster*, o usuário pode submeter vários processos ao escalonador, que os organiza em uma tabela de processos e gerencia suas execuções. Por fim, de acordo com a capacidade de processamento demandada, a ferramenta de *Autoscaling* entra em ação, adicionando ou retirando nós computacionais, de acordo com a necessidade da aplicação em execução, mas sempre respeitando os limites definidos no arquivo de configuração do usuário.

2.4.3 Simple Storage Service (S3)

O *Simple Storage Service* (*S3*) é um serviço, criado pela *AWS* em 2006, que permite armazenar e recuperar dados, via *web* [25]. Os dados armazenados no *S3* são vistos como objetos, que necessitam ser armazenados em contêineres, chamados de *buckets*. Um usuário pode ter vários *buckets*, em *datacenters* de diversas regiões do mundo, porém cada um deve possuir um nome globalmente exclusivo, ou seja, diferente de todos os outros *buckets* existentes na *Amazon*. Uma vez armazenados, os dados podem ser (re)organizados da maneira que o usuário preferir.

2.4.4 Virtual Private Cloud (VPC)

Uma *Virtual Private Cloud (VPC)* é uma rede privada criada na nuvem. Apesar de estar na nuvem, a VPC é isolada, ou seja, restrita à conta do usuário e portanto não pode ser visualizada nem conectada por outros usuários. À cada VPC é associado um IP elástico, que consiste em um endereço IPV4, que a identifica. É possível inicializar diversos serviços da AWS na VPC,

como instâncias, volumes, buckets, dentre outros. Neste sentido, as VPCs são fundamentais para o cluster, uma vez que possibilitam a comunicação entre as instâncias e a criação de um sistema de arquivos (volume EBS) único e compartilhado.

Regiões são áreas geográficas separadas, que provêm os serviços da AWS, de maneira independente. Ao criar uma VPC é necessário especificar uma região na qual ela irá residir. Dentro de cada região existem várias zonas de disponibilidades, também independentes entre si e que agrupam os recursos da região. Dessa maneira, a falha de uma zona de disponibilidade não traz consequências para a outra e o mesmo ocorre no caso das regiões. Neste contexto, utilizar recursos em diferentes zonas pode ser interessante, uma vez que diminui a chance de falhas, porém agrupar recursos em uma mesma zona pode ser mais interessante, uma vez que a proximidade entre eles pode gerar um melhor desempenho da aplicação.

Uma VPC comporta sub-redes, que podem ser públicas ou privadas. As públicas são assim chamadas pois têm acesso público à *internet* e portanto podem tanto vir dados da *internet* em direção à *VPC*, quanto o contrário. Já as sub-redes privadas podem ter acesso à *internet*, porém este fluxo é restrito, de maneira que somente é possível enviar dados da sub-rede para a *internet*, garantindo assim segurança e privacidade dos recursos contidos nela.

A Figura 2.5 ilustra um exemplo no qual a conta A cria toda a infraestrutura da VPC, incluindo sub-redes, tabela de rotas e gateways. As contas B, C e D pertencem à mesma organização de A e por isso podem se conectar e utilizar a sua VPC e seus recursos, porém não podem realizar modificações na estrutura. Neste exemplo foram criadas duas sub-redes: um pública, para a conta D e uma privada para as contas A e C. Neste caso as contas são separadas em diferentes sub-redes, pois a aplicação da conta D exige interação com o público, enquanto as aplicações das contas B e C não podem ser acessadas por outros usuários. Em cada um dos casos são utilizados diferentes serviços da *AWS*. Ambas as sub-redes se conectam ao mesmo roteador, que gerencia a tabela de rotas, proporcionando diversas conexões, entre elas a conexão com a internet [3].

Durante a criação de um cluster utilizando a ferramenta ParallelCluster, definem-se algumas informações básicas sobre a VPC, como a região de alocação e a quantidade de sub-redes a serem criadas: uma pública para o mestre e outra privada aos nós; ou uma única rede pública para todas as instâncias. Esta escolha depende da aplicação do usuário. Em seguida, o ParallelCluster direciona todos os recursos utilizados por ele para a mesma zona de disponibilidade e a VPC, por sua vez, realiza todas as configurações necessárias para proporcionar uma comunicação entre as instâncias e os outros recursos, poupando muito trabalho do usuário.

2.5 Comentários Finais

Apesar de ser um paradigma recente, as primeiras noções de computação em nuvem são datadas da década de 60, entretanto a tecnologia da época não era capaz de implementar a computação em nuvem como conhecemos. Atualmente há várias empresas que oferecem tal serviço, sendo a *AWS* uma das principais, pois possui uma ampla variedade de serviços a um custo razoável. O *ParallelCluster* é uma ferramenta da *AWS* que permite a execução de aplicações de alto desempe-

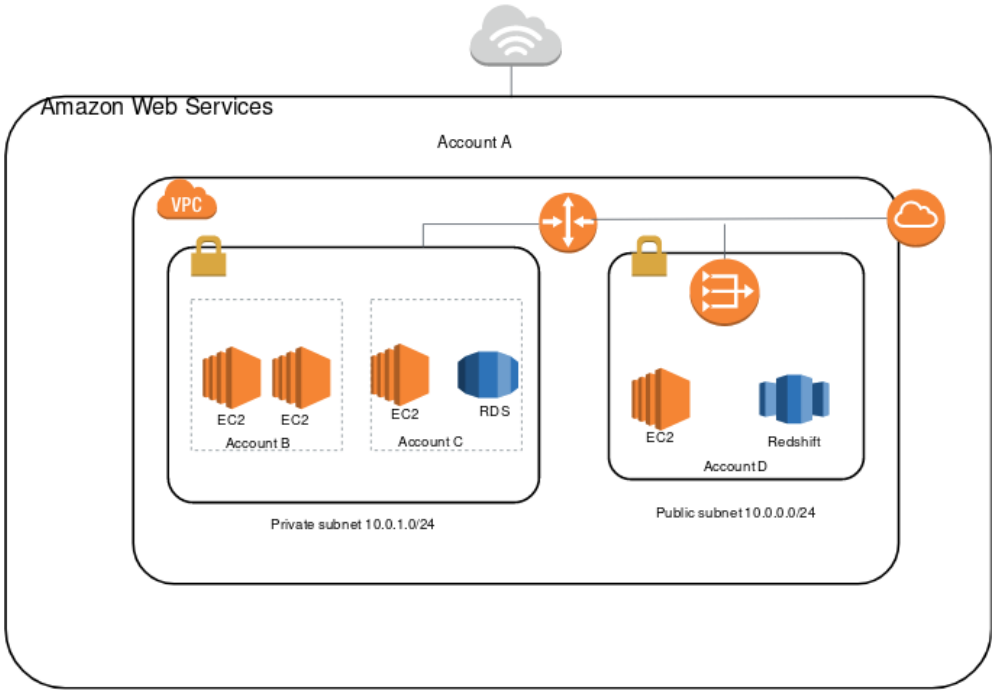


Figura 2.5: Exemplo de projeto com sub-redes pública e privada. Retirada de [3]

nho por meio da criação de *clusters* em nuvem. Dessa maneira, o *ParallelCluster* combina as três tecnologias abordadas no capítulo: computação em *cluster*, uma vez que tanto os nós computacionais, quanto a rede *VPC* são dedicados ao usuário; computação em *grade*, pois para criar o *cluster* é necessário reservar as instâncias da *AWS*, que nem sempre estão disponíveis; e virtualização, pois todas as instância que compõe o *cluster* são máquinas virtuais. Nesse contexto, a possibilidade de implementar um *cluster* em nuvem, que garanta poder de computação e banda de rede é o que possibilita a execução de aplicações de alto desempenho, como as ferramentas de comparação de sequências biológicas.

Capítulo 3

Comparação de Sequências Biológicas

3.1 Conceitos Básicos

A comparação de sequências biológicas é um dos tópicos mais estudados no ramo da Biologia Molecular. A sua importância reside no fato de que novas sequências tendem a surgir a partir de modificações em outras já existentes. As regiões de semelhança entre as sequências costumam gerar as mesmas propriedades das antigas nas novas, podendo ser impactantes na constituição da estrutura molecular ou em suas funcionalidades. Dessa maneira, a análise das características de uma determinada sequência, seja ela de DNA, RNA ou de uma proteína, se torna muito mais simples a partir do momento que já se conhece uma outra, similar à que está sendo alvo do estudo [7].

A operação de comparação se inicia definindo as duas sequências de entrada. Em seguida, é necessário realizar o seu alinhamento, que consiste em listar as sequências em duas linhas e parear seus elementos, por colunas. O alinhamento pode ser realizado de três maneiras principais: global, local ou semiglobal. O alinhamento global analisa ambas as sequências por inteiro e assim é utilizado quando as sequências possuem tamanhos semelhantes. Já o alinhamento local busca o fragmento das sequências mais parecido. Por fim, no alinhamento semiglobal, não se analisa as sequências por inteiro, porém só é possível remover os préfixos ou sufixos de cada, ou seja, uma cadeia de elementos que iniciam ou que encerram a sequência. Tanto o alinhamento global quanto o semi-global buscam maximizar a quantidade de caracteres similares, enquanto o alinhamento local, além disso, tenta favorecer subsequências longas. As Figuras 3.1, 3.2, 3.3 mostram, respectivamente um exemplo de alinhamento global, local e semiglobal. Os valores utilizados para comparação foram $match = +1$, $mismatch = -1$ e $gap = -2$.

$$
\begin{array}{lcccccccccccc}
S_0: & G & T & C & C & G & A & C & A & T & C & A & G \\
S_1: & - & - & - & - & G & A & C & A & A & T & A & C \\
\hline
E: & -2 & -2 & -2 & -2 & +1 & +1 & +1 & +1 & -1 & -1 & +1 & -1 \quad = -6
\end{array}
$$

Figura 3.1: Exemplo de comparação global

$$
\begin{array}{lcccccccccccccc}
S_0: & * & * & * & * & G & A & C & A & * & * & * & * & & \\
S_1: & & & & & G & A & C & A & * & * & * & * & & \\
- & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
E: & & & & & +1 & +1 & +1 & +1 & & & & & & = +4
\end{array}
$$

Figura 3.2: Exemplo de comparação local

$$
\begin{array}{lcccccccccccccc}
S_0: & * & * & * & * & G & A & C & A & T & C & A & G & & \\
S_1: & & & & & G & A & C & A & A & T & A & C & & \\
- & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
E: & & & & & +1 & +1 & +1 & +1 & -1 & -1 & +1 & -1 & & = +2
\end{array}
$$

Figura 3.3: Exemplo de comparação semiglobal

Uma vez obtido o alinhamento, realiza-se uma comparação elemento a elemento. Caso eles sejam iguais, ocorre um *match* e a comparação desses elementos recebe uma pontuação de valor positivo. Se os elementos pareados forem diferentes, diz-se que ocorreu um *mismatch* e a pontuação atribuída à comparação possui valor negativo. Por fim, pode-se inserir um *gap* em uma das sequências. Neste caso, não realiza-se o alinhamento entre esses elementos, mas é atribuída uma pontuação negativa devido à inserção do *gap*, de acordo com o modelo de penalização adotado. Caso o modelo seja simples (chamado de moldeo linear), a penalização é constante, porém se for o modelo *affine-gap*, há uma pontuação para abertura do *gap*, chamada de *gap-open*, e outra caso haja a sua extensão, chamada de *gap-extension*. A penalização do *gap-extension* costuma ser menor, em módulo, do que a do *gap-open*, tornando a extensão menos prejudicial para o escore do alinhamento do que a sua abertura [7]. Ao final das comparações, realiza-se uma somatória de todos os pontos contabilizados e obtém-se o escore do alinhamento. No caso das Figuras 3.1, 3.2 e 3.3, os escores obtidos para os alinhamentos global, local e semi global foram, respectivamente, -6, $+4$ e $+2$.

Vale ressaltar que as respectivas pontuações são elaboradas com base em estudos estatísticos. Neles são levados em consideração o impacto que os *matches*, *mismatches* e *gaps* contribuem para a similaridade entre as sequências [7].

A inserção de *gaps* nas sequências provoca defasagens na comparação dos elementos, proporcionando assim, diversas possibilidades de alinhamentos. Este efeito se destaca muito na prática, pois sequências de DNA, RNA ou proteínas costumam ser muito extensas, na ordem de milhares ou milhões de bases. Dessa maneira, essa ampla gama de possibilidades dificulta uma dedução simples dos efeitos que a inserção de um determinado *gap* pode provocar no escore ótimo. Portanto, necessita-se de algoritmos exatos, que sejam capazes de identificar o escore ótimo e seus respectivos alinhamentos.

3.2 Algoritmos de Comparação de Sequências Biológicas

3.2.1 Simbologia

Esta Seção apresenta alguns algoritmos de comparação de sequências biológicas. Para explicá-los foram utilizados um conjunto de símbolos padronizados, que estão listados na Tabela 3.1, assim como suas respectivas definições.

Símbolos	Descrição
S_0 e S_1	Vetores contendo as bases do par de sequências selecionadas para a comparação
m e n	Tamanho, respectivamente, das sequências S_0 e S_1
i e j	Índices, respectivamente, dos vetores S_0 e S_1
$A_{i,j}$	Elemento da matriz de similaridade na posição (i,j)
G	Penalidade de inserção de um *gap*
sbt($S_0[i]$, $S_1[j]$)	Função que atribui pontuação referente à comparação das sequências nas posições $S_0[i]$ e $S_1[j]$
λ	Função que atribui penalidade ao modelo do affine gap
G_{open}	Penalidade do *gap* que inicia a sequência de *gaps*
G_{ext}	Penalidade dos *gaps* que dão continuidade à sequência de *gaps*
k	Tamanho da sequência de *gaps*
H	Matriz (Gotoh) que armazena o escore ótimo para cada par de índices da comparação
E	Matriz (Gotoh) que armazena escores referentes à abertura e extensão de *gaps* na sequência S_0
F	Matriz (Gotoh) que armazena escores referentes à abertura e extensão de *gaps* na sequência S_1
C	Vetores coluna utilizados em MM para obter os *crosspoints*
$i*$	Linha da coluna média, por onde passa o *crosspoint*

Tabela 3.1: Tabela de símbolos.

3.2.2 Needleman-Wunsch (NW)

O algoritmo de *Needleman-Wunsch* realiza uma comparação global entre duas sequências e objetiva obter seu escore ótimo, bem como os alinhamentos que resultam nele. Neste caso, a inserção do *gap* segue o modelo linear, no qual não se faz distinção se ele é de abertura ou de extensão e portanto, possui penalidade constante. O algoritmo se dá em duas etapas. Na primeira realiza-se o cálculo da matriz de similaridade e na segunda recupera-se o alinhamento ótimo [26].

Etapa 1 - Calculo da matriz de similaridade:

Suponha duas sequências S_0 e S_1, de tamanhos m e n. A matriz de similaridade, A, terá dimensão $(m+1)$x$(n+1)$, sendo cada elemento dado por $A_{i,j}$ onde i e j são, respectivamente, os índices de linha e coluna da matriz. Primeiramente é necessário inicializar a primeira linha da

matriz de similaridade com valores negativos, segundo a fórmula $A_{0,j} = -j \cdot G$, onde G é o valor da penalidade linear de *gap*. A mesma coisa ocorre para a primeira coluna, $A_{i,0} = -i \cdot G$ e o elemento inicial recebe valor nulo, $A_{0,0} = 0$.

A partir de então pode-se iniciar o cálculo da matriz. À cada um de seus elementos é atribuído um valor, que representa o escore ótimo do alinhamento gerado, partindo dos índices iniciais e encerrando naqueles do elemento analisado, ou seja, $S_0[1..i]$ e $S_1[1...j]$. Nesse contexto, há 3 possibilidades para a obtenção desse escore e a que tiver o maior valor será a escolhida.

A primeira possibilidade ocorre a partir da comparação entre os caracteres $S_0[i-1]$ e $S_1[j-1]$, na qual analisa-se o resultado, ou resíduo $(sbt(a,b))$, da comparação, ou seja, se ocorreu um *match* ou *mismatch*. Já a segunda possibilidade, consiste na situação de inserção de um *gap* na sequência S_0, a partir do alinhamento $S_0[1...i]$ e $S_1[1...j-1]$. Por fim, na terceira situação, analisa-se a inserção do *gap* na sequência S_1 a partir do alinhamento $S_0[1...i-1]$ e $S_1[1...j]$. A equação que descreve a escolha de possibilidades está representada em 3.1.

$$A_{i,j} = max \begin{cases} A_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ A_{i,j-1} + G \\ A_{i-1,j} + G \end{cases} \tag{3.1}$$

Nota-se da equação 3.1, que ela simplesmente analisa os elementos da linha, coluna ou linha e coluna anteriores ao que está sendo calculado e verifica qual valor é maior: se é a penalidade do *gap* adicionada ao valor do elemento da linha ou coluna anteriores; ou se é o valor do resíduo somado ao valor do elemento da linha e coluna anterior. A Figura 3.4 exemplifica esta situação.

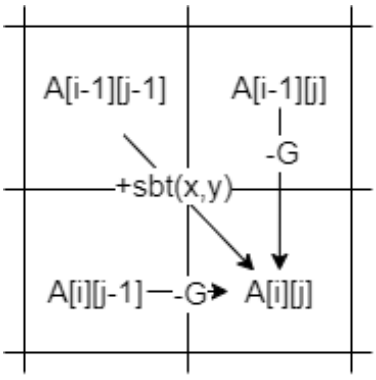


Figura 3.4: Determinação do escore de um elemento genérico

Etapa 2 - Recuperação do Alinhamento Ótimo (Traceback)

O objetivo desta etapa é reconstituir o alinhamento global ótimo. Na etapa anterior, além de se calcular o escore atribuiu-se um ponteiro, à cada elemento, que indica qual dos elementos anteriores, $A[i-1][j]$, $A[i][j-1]$ ou $A[i-1][j-1]$, o originou. Sendo assim, a recuperação do alinhamento ótimo se dá percorrendo todos os ponteiros, do final da matriz, no canto inferior direito, para seu início, no canto superior esquerdo. Vale ressaltar que o escore de cada elemento pode ser proveniente de um à três dos seus predecessores, possibilitando que haja múltiplos alinhamentos

ótimos. A Figura 3.5 representa a matriz de similaridade e um de seus alinhamentos ótimos, com escore $E = -3$. A pontuação atribuída foi: *match* $= +1$, *mismatch* $= -1$ e *gap* $= -2$.

D		A_1	A_2	T_3	G_4	G_5	C_6	G_7	T_8	G_9	C_{10}	T_{11}	A_{12}	C_{13}
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26
A_1	-2	1	-1	-3	-5	-7	-9	-11	-13	-15	-17	-19	-21	-23
A_2	-4	-1	2	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20
A_3	-6	-3	0	1	-1	-3	-5	-7	-9	-11	-13	-15	-15	-17
T_4	-8	-5	-2	1	0	-2	-4	-6	-6	-8	-10	-12	-14	-16
T_5	-10	-7	-4	-1	0	-1	-3	-5	-5	-7	-9	-9	-11	-13
G_6	-12	-9	-6	-3	0	1	-1	-2	-4	-4	-6	-8	-10	-12
T_7	-14	-11	-8	-5	-2	-1	0	-2	-1	-3	-5	-5	-7	-9
A_8	-16	-13	-10	-7	-4	-3	-2	-1	-3	-2	-4	-6	-4	-6
G_9	-18	-15	-12	-9	-6	-3	-4	-1	-2	-2	-3	-5	-6	-5
C_{10}	-20	-17	-14	-11	-8	-5	-2	-3	-2	-3	-1	-3	-5	-5
G_{11}	-22	-19	-16	-13	-10	-7	-4	-1	-3	-1	-3	-2	-4	-6
A_{12}	-24	-21	-18	-15	-12	-9	-6	-3	-2	-3	-2	-4	-1	-3
Score: -3														

Figura 3.5: Matriz de similaridade obtida segundo o algoritmo de *Needleman-Wunsch*. Imagem construída utilizando a plataforma Freiburg:Teaching - Needleman-Wunsch

Análise de Complexidade

Uma das vantagens desse algoritmo consiste na simplicidade do método e de seus cálculos, entretanto sua complexidade em termos de tempo de processamento e uso de memória deixam a desejar. Isto porque para obter o alinhamento ótimo é necessário calcular toda a matriz de similaridade, que possui $(m+1)\cdot(n+1)$ posições. Dessa maneira, se considerarmos sequências de mesmo tamanho, a complexidade deste algoritmo se torna quadrática, $O(n^2)$, o que dificulta ou até impossibilita a comparação de sequências muito grandes, dependendo do hardware disponível.

Caso apenas o escore ótimo seja necessário, a complexidade do algoritmo pode ser reduzida em termos de memória. Isto se deve pelo fato de que é possível calcular uma linha de escores partindo apenas da anterior e o mesmo ocorre para as colunas. Assim, para a obtenção do escore ótimo, o armazenamento de duas linhas ou de duas colunas seria suficiente, diminuindo a complexidade para $O(m)$. A desvantagem dessa abordagem é o fato de que o escore por si só não traz muitas informações biológicas, enquanto o alinhamento, por sua vez, constitui uma importante ferramenta para se avaliar o resultado da comparação e realizar as devidas conclusões [4].

3.2.3 Smith-Waterman (SW)

O algoritmo de *NW* obtém o escore e alinhamento ótimos, a partir de uma comparação global, o que proporciona a avaliação das sequências como um todo. Entretanto, a análise de sub-sequências também é extremamente relevante para a área de biologia molecular, especialmente quando deseja-se descobrir se elas compartilham de uma mesma origem evolutiva. Isto porque duas sequências com descendência comum podem divergir em grande parte de sua cadeia, tornando o escore global baixo, mas ainda apresentar regiões similares, que comprovam sua origem comum. Neste contexto, algoritmos de alinhamento local são capazes de identificar tais regiões de semelhança e assim obter uma análise mais direcionada [7].

O algoritmo de *Smith-Waterman* identifica o alinhamento local ótimo [27] e é muito semelhante ao de *NW* (Seção 3.2.2), porém possui três aspectos principais que o diferenciam. O primeiro aspecto está na Equação 3.1, que agora deve incluir o termo 0 na operação de máximo, como mostra a Equação 3.2. Esta modificação garante a ausência de escores negativos na matriz e, consequentemente, ao invés de dar continuidade a um alinhamento com valores negativos, a alteração proporciona o recomeço de novos alinhamentos a partir da reiniciação com o escore 0. A segunda modificação consiste em inicializar a primeira linha e coluna da matriz de similaridade com zeros, uma vez que ela não deve conter valores negativos. A terceira diferença ocorre no alinhamento, pois ao invés de sempre utilizar a última célula da matriz como origem do alinhamento, em *SW* o alinhamento se inicia no elemento de maior escore da matriz, percorrendo-a no sentido inverso de sua construção e se encerra ao atingir uma célula com escore nulo.

$$A_{i,j} = max \begin{cases} A_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ A_{i,j-1} + G \\ A_{i-1,j} + G \\ 0 \end{cases} \tag{3.2}$$

A Figura 3.6 representa o alinhamento gerado pelo algoritmo de *SW*, utilizando a mesma sequência e parâmetros da Seção 3.2.2 anterior. Vale ressaltar que a complexidade deste algoritmo, tanto em relação ao tempo de processamento, quanto à utilização de memória permaneceu a mesma: $O(mn)$.

S		A_1	A_2	T_3	G_4	G_5	C_6	G_7	T_8	G_9	C_{10}	T_{11}	A_{12}	C_{13}
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A_1	0	1	1	0	0	0	0	0	0	0	0	0	1	0
A_2	0	1	2	0	0	0	0	0	0	0	0	0	1	0
A_3	0	1	2	1	0	0	0	0	0	0	0	0	1	0
T_4	0	0	0	3	1	0	0	0	1	0	0	1	0	0
T_5	0	0	0	1	2	0	0	0	1	0	0	1	0	0
G_6	0	0	0	0	2	3	1	1	0	2	0	0	0	0
T_7	0	0	0	1	0	1	2	0	2	0	1	1	0	0
A_8	0	1	1	0	0	0	0	1	0	1	0	0	2	0
G_9	0	0	0	0	1	1	0	1	0	1	0	0	0	1
C_{10}	0	0	0	0	0	0	2	0	0	0	2	0	0	1
G_{11}	0	0	0	0	1	1	0	3	1	1	0	1	0	0
A_{12}	0	1	1	0	0	0	0	1	2	0	0	0	2	0
Score: 3														

Figura 3.6: Matriz de similaridade obtida segundo o algoritmo de *Smith-Waterman*. Imagem construída utilizando a plataforma Freiburg:Teaching - Smith-Waterman

3.2.4 Gotoh

O algoritmo de *Gotoh* [28] busca substituir o modelo do *linear gap*, utilizado nos algoritmos de *Needleman-Wunsch* e *Smith-Waterman*, pelo *affine gap*. Nesse caso, a penalidade dos *gaps* deixa de ser constante: os de abertura terão uma penalização maior e os de extensão, menor. Dessa maneira, a pontuação de gap é dada por:

$$\lambda(k) = G_{open} + (k-1) \cdot G_{ext} \tag{3.3}$$

, sendo k a quantidade de *gaps* consecutivos, G_{open} a penalização por abertura de *gap* e G_{ext} a penalização por sua extensão.

Para carregar a informação da consecutividade de *gaps*, o algoritmo de *Gotoh* propõe a utilização de três matrizes: H, E e F. A matriz H armazena os escores ótimos para cada par de índices da comparação. Já a matriz E carrega os escores referentes à abertura e extensão de *gaps* na sequência S_0, enquanto a matriz F faz o mesmo para a sequência S_1 [29].

A primeira etapa deste algoritmo consiste em inicializar as matrizes. Assim como na seção 3.2.2, o elemento inicial é zerado, $H_{0,0} = 0$, enquanto a primeira linha e coluna de H são inicializadas de acordo com a penalização dinâmica de *gaps*, conforme a Equação 3.3 e assim gerando as seguintes fórmulas: $H_{i,0} = \lambda(i)$ e $H_{0,j} = \lambda(j)$. No caso da matriz E, seu primeiro elemento

também é zerado $E_{0,0} = 0$, sua primeira coluna é inicializada com $E_{i,0} = -\infty$ e a primeira linha é ignorada, $E_{0,j} = X$. A inicialização da matriz F é feita de maneira análoga, porém a operação realizada na primeira linha de E passa a ser feita na coluna de F e vice-versa, ou seja, $F_{0,0} = 0$, $F_{0,j} = -\infty$ e $F_{i,0} = X$.

Após a inicialização das matrizes, pode-se iniciar o seu preenchimento. Dessa maneira, as equações de recorrência são adaptadas, de maneira a introduzir a dinâmica de *affine gap* ao algoritmo, como mostram as Equações 3.4, 3.5 e 3.6

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ E_{i,j} \\ F_{i,j} \end{cases} \tag{3.4}$$

$$E_{i,j} = max \begin{cases} H_{i,j-1} + G_{open} \\ E_{i,j-1} + G_{ext} \end{cases} \tag{3.5}$$

$$F_{i,j} = max \begin{cases} H_{i-1,j} + G_{open} \\ F_{i-1,j} + G_{ext} \end{cases} \tag{3.6}$$

Por fim, realiza-se o *traceback*, recuperação do alinhamento ótimo, partindo do último elemento da matriz, $H_{m,n}$ até seu primeiro elemento, $H_{0,0}$. Durante esse retorno verifica-se a matriz e o elemento que originaram o atual. Cada referência à um elemento da matriz E representa a introdução de um *gap* na sequência S_0 e avanço na S_1, enquanto que as referências à matriz F representam a introdução de *gap* em S_1 e avanço em S_0. As referências à matriz H representam o avanço de ambas as sequências.

Vale ressaltar que, tanto em termos de tempo de processamento, quanto de espaço de armazenamento a ordem de complexidade do algoritmo permaneceu em $O(mn)$, embora o triplo de espaço e computação sejam necessários para realizá-lo.

3.2.5 Myers & Miller

A ampla utilização de armazenamento verificada nos algoritmos das Seções 3.2.2, 3.2.3 e 3.2.4, dificulta ou até impossibilita a obtenção do alinhamento ótimo para sequências muito longas. Nesse contexto, *Hirschberg* [30] elaborou um algoritmo que soluciona o problema da *LCS (Longest Common Subsequence)* utilizando complexidade de espaço linear $O(m+n)$, ao invés da quadrática, $O(mn)$, adotada pelos outros algoritmos. Posteriormente, *Myers e Miller* [31] se baseiam nesta ideia de *Hirschberg*, adaptando o algoritmo de *Gotoh* para operar em espaço linear [4].

O algoritmo de *MM* busca, primeiramente obter o *crosspoint*, ou seja, elemento da matriz, localizado na coordenada média das colunas, pelo qual passa o alinhamento ótimo. Para obter tal elemento, é necessário inicialmente calcular toda a coluna central, $n/2$. Entretanto, ao contrário dos algoritmos anteriores, em *MM* este processo não é realizado por meio do armazenamento completo da matriz de similaridade, mas de apenas quatro colunas. Isto porque, o cálculo do

escore de qualquer elemento da matriz pode ser obtido utilizando-se apenas a coluna atual e a anterior, ao elemento em questão. Dessa maneira, *MM* utiliza quatro vetores coluna para calcular, iterativamente, os escores das colunas mais externas da matriz, até a central. Dois desses vetores são utilizados para obter o valor da coluna no sentido normal da matriz, isto é, $C_{i,0} \rightarrow C_{i,\frac{n}{2}}$ e os outros dois no sentido reverso, $C_{i,n} \rightarrow C_{i,\frac{n}{2}}$. Após a obtenção das colunas centrais, realiza-se o cálculo da coluna central resultante, que é dada por meio da adição, elemento a elemento, das colunas centrais no sentido direto e reverso. Em seguida, verifica-se a linha da coluna central resultante que especifica o elemento de maior escore. Esse é o *crosspoint*.

A linha, $i*$, e coluna, $n/2$, do *crosspoint* são então utilizadas para dividir a matriz em quatro, como mostra a Figura 3.7. Sabe-se que o alinhamento segue do canto superior esquerdo da matriz, para o canto inferior direito e, consequentemente, ele não pode passar por um ponto que envolva incrementar a coordenada de uma linha e decrementar a da coluna e vice-versa, pois o alinhamento passaria a seguir no sentido oposto. Dessa maneira, as duas regiões em cinza, não podem fazer parte do alinhamento e, portanto, elas são descartadas, deixando apenas as regiões em verde e assim, reduzindo a área de processamento para metade da inicial. Este algoritmo será executado recursivamente, para cada uma das seções em verde, encontrando seus *crosspoints*, dividindo suas seções em quatro e descartando metade de cada área, até que se obtenha todos os pontos do alinhamento ótimo.

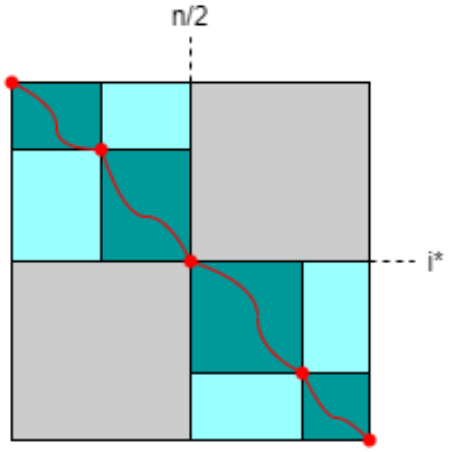


Figura 3.7: Representação da divisão da matriz realizada no algoritmo de Myers & Miller.

Analisando a questão da complexidade de tempo de processamento deste algoritmo, nota-se que, além de se calcular a matriz completa na primeira passada (recursão), em *MM* reduz-se a área de processamento pela metade a cada iteração. Assim, para descobrir a quantidade total de área processada ao final do procedimento, considere que a área processada a cada iteração é um elemento de uma progressão geométrica, de razão $q = \frac{1}{2}$. Além disso, a equação da soma de infinitos termos de uma progressão geométrica decrescente, ou seja, de razão $0 < q < 1$, é dada por $S = \frac{a_1}{1-q}$, sendo a_1 o primeiro termo da progressão, que nesse caso representa a área total da matriz, A_M. Aplicando essa equação, deduz-se que a quantidade total de área de processamento, A_T, deve ser menor que este somatório, uma vez que, na prática, suas divisões não serão realmente

infinitas:

$$\begin{aligned} A_T &< S \\ A_T &< \frac{a_1}{1-q} \\ A_T &< \frac{A_M}{1-\frac{1}{2}} \\ A_T &< 2 \cdot A_M \end{aligned} \tag{3.7}$$

Dessa maneira, como o máximo de área calculada é menor que duas vezes a área total da matriz, a ordem de complexidade do tempo de processamento continua $O(mn)$, porém como são armazenadas apenas algumas colunas da matriz de similaridade, a complexidade em termos de armazenamento é linear, $O(m+n)$.

3.3 Comentários Finais

A comparação de sequências biológicas é uma das principais aplicações no ramo da biologia molecular, pois facilita o estudo das características dos organismos. Para realizá-la foram implementados diversos algoritmos exatos, como o de *Needleman-Wunsch (NW)*, *Smith-Waterman (SW)*, *Gotoh* e *Myers & Miller (MM)*, cada um com suas respectivas métricas e complexidades em termos espacial e temporal. Tais algoritmos são amplamente utilizados por ferramentas de comparação de sequências biológicas. O *MASA-CUDAlign*, por exemplo, utiliza *Smith-Waterman*, *Gotoh* e *Myers & Miller* como base para obter a matriz de programação dinâmica e, consequentemente, seu escore ótimo e o alinhamento que o gera.

Capítulo 4

Ferramenta MASA

4.1 CUDAlign 1.0

O *CUDAlign 1.0* é uma ferramenta de comparação de sequências biológicas, que busca obter o escore ótimo e sua respectiva posição na matriz de similaridade, M, usando GPUs (placas gráficas). Para tanto, utiliza-se o algoritmo de Gotoh, seção 3.2.4, que adota o modelo de penalização *affine gap*, porém com a particularidade de que, as três matrizes H, E e F, são agrupadas em uma única matriz M. Uma das principais características do *CUDAlign 1.0* consiste na paralelização da aplicação, utilizando a técnica do *wavefront*, tanto dentro dos blocos, quanto entre eles. O *CUDAlign* 1.0 e as suas versões posteriores, que serão explicadas nas seguintes seções, foram desenvolvidas por Sandes [4].

4.1.1 Paralelismo Externo

O paralelismo que ocorre entre os blocos é chamado de externo [4]. Cada bloco é um conjunto de elementos da matriz M, agrupados a cada R linhas e C colunas. O primeiro passo do paralelismo externo consiste em redimensionar a matriz de similaridade, para que fique em termos de blocos, ao invés de elementos. Analisando uma comparação envolvendo duas sequências de tamanho m e n, a matriz de similaridade terá dimensão $m \times n$ e, após a divisão em blocos, sua nova dimensão será dada por $\frac{m}{R} \times \frac{n}{C}$. Essa divisão está representada na Figura 4.1, na qual a dimensão da matriz é alterada de 8×8 para 4×4.

As dimensões R e C, que definem um bloco, não são arbitrárias e portanto para calculá-las é necessário definir previamente os valores de três parâmetros: B, T e α, que dependem dos parâmetros da GPU utilizada. Os dois primeiros representam, respectivamente, a quantidade de blocos e de threads que serão executados em uma chamada de kernel, enquanto que o terceiro consiste na quantidade de linhas que cada *thread* ficará responsável por processar. Uma vez obtidos os três parâmetros, define-se que os B blocos ocupam todas as colunas da matriz M e que cada uma das T *threads* ficará responsável por excutar α linhas do bloco, de maneira que as dimensões dos blocos são dadas por $R = \alpha \cdot T$ e $C = \frac{n}{B}$. No exemplo da Figura 4.1, os parâmetros

adotados foram $\alpha = 1$, $T = 2$ e $B = 4$.

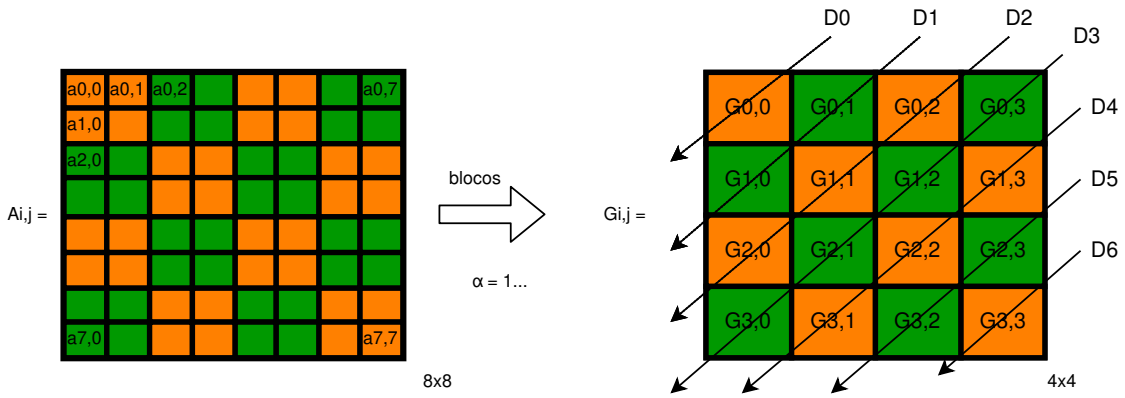


Figura 4.1: Matriz de similaridade transformada de elementos para blocos.

Após obter o valor desses parâmetros é possível reorganizar a matriz em blocos ao invés de elementos, como mostra a Figura 4.1 e, em seguida, agrupar os blocos em diagonais para que seja possível utilizar a técnica *wavefront*. Cada diagonal externa é definida por $D_k = \{G_{ij} | i+j = k\}$, ou seja, os elementos são agrupados pela soma de suas coordenadas na matriz. A consequência dessa equação é que a quantidade de elementos, em cada diagonal, varia de 1 à B, sendo a quantidade máxima de diagonais em uma matriz dada por $|D| = B + \frac{m\alpha}{T} - 1$.

A última etapa do paralelismo externo consiste em aplicar a técnica do *wavefront*. Esta técnica realiza uma chamada ao kernel para cada diagonal externa, que processa todos os blocos da respectiva diagonal. Finalizado o processamento desses blocos, realiza-se uma sincronização, e a CPU reinvoca o kernel para iniciar o processamento de uma nova diagonal externa. Esse procedimento continua sendo realizado até que se processe a matriz por completo.

4.1.2 Paralelismo Interno

O paralelismo interno ocorre dentro do bloco e entre seus elementos. De maneira similar ao que foi feito no paralelismo externo, no interno é necessário agrupar os elementos em diagonais internas, mas segundo a expressão $d_k = \{(i,j) | \lfloor \frac{i}{\alpha} \rfloor + j = k\}$ [4], conforme é ilustrado na Figura 4.2, na qual diferentes diagonais estão pintadas de verde ou laranja. No exemplo dessa imagem, foram especificadas as coordenadas dos elementos das diagonais d_0, d_1 e d_3 de forma a exemplificar a separação das diagonais. Vale ressaltar a notação de *floor* utilizada na expressão anterior, que implica que todos os números decimais serão arredondados para baixo.

Para garantir o paralelismo interno utiliza-se T *threads*, sendo que cada *thread* T_k executa α linhas, ou seja, da linha αk até a linha $\alpha k + \alpha - 1$. No exemplo da Figura 4.2, $\alpha = 2$ e $T = 4$. O sentido de computação é da esquerda para a direita e sempre executa-se o máximo de *threads* possível, em paralelo. Além disso, para utilizar a técnica do *wavefront*, a *thread* T_k deve estar sempre adiantada em relação à *thread* T_{k+1} e assim, enquanto T_k executa a coluna j, T_{k+1} executa a coluna $j - 1$. Essa regra garante que um elemento não seja calculado previamente a algum

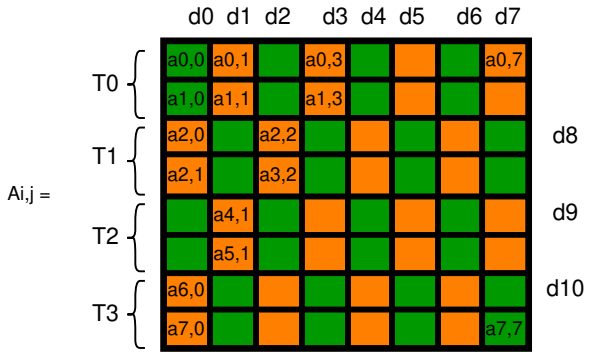


Figura 4.2: Representação do paralelismo interno.

de seus três predecessores. Para garantir o cumprimento dessa regra, realiza-se, ao final de cada diagonal, uma sincronização, impedindo que uma *thread* inicie a execução da diagonal seguinte antes que as outras terminem a atual.

4.1.3 Delegação de Células

Assim como mencionado nas seções 4.1.1 e 4.1.2 anteriores, o paralelismo do *CUDAlign 1.0* ocorre tanto externamente quanto internamente aos blocos, porém o grau de paralelismo varia de maneira diretamente proporcional à quantidade de blocos (diagonal externa) ou elementos (diagonal interna) processados ao mesmo tempo [4]. Analisando o caso da Figura 4.2 percebe-se que as suas três primeiras diagonais, de d_0 à d_2, processam menos de 8 células e utilizam menos *threads* do que foi reservado para este bloco e portanto possuem menor nível de paralelismo. Já as diagonais d_3 à d_7 utilizam todos os recursos providos ao bloco, indicando que há maior grau de paralelismo nessa região.

Assim, para tentar obter maior aproveitamento do paralelismo, o algoritmo *CUDAlign* utiliza uma proposta denominada *delegação de células*. Nessa técnica apenas as C primeiras diagonais internas serão processadas, pois é nelas que o paralelismo é considerado máximo, deixando o restante pendente para a execução do próximo bloco. As diagonais pendentes, por sua vez, são calculadas juntamente com as primeiras diagonais do bloco seguinte, de maneira que o paralelismo continua máximo. A Figura 4.3 ilustra que as diagonais d_8 à d_{10}, pendentes do bloco i, são continuadas e executadas, respectivamente nas diagonais d_0 à d_2 do bloco $i+1$. Dessa maneira, o bloco $i+1$ executa 8 diagonais, de d_0 à d_7, com paralelismo máximo e deixa o restante pendente para o próximo bloco.

Vale ressaltar que a técnica da delegação de células posterga a execução das diagonais finais de cada bloco, de maneira que, em teoria, a forma do bloco continua retangular, porém na prática ele se comporta na forma de um paralelogramo. A Figura 4.3 ilustra esse efeito por meio da linha de contorno azul. Além disso, a Figura 4.4 representa os vários blocos da matriz, em formato de paralelogramo, bem como a delegação células, tanto para blocos na mesma linha, quanto para blocos de outras linhas, como é o caso de b_1 e b_2. Já o conjunto de células delegadas, c_1 e c_2,

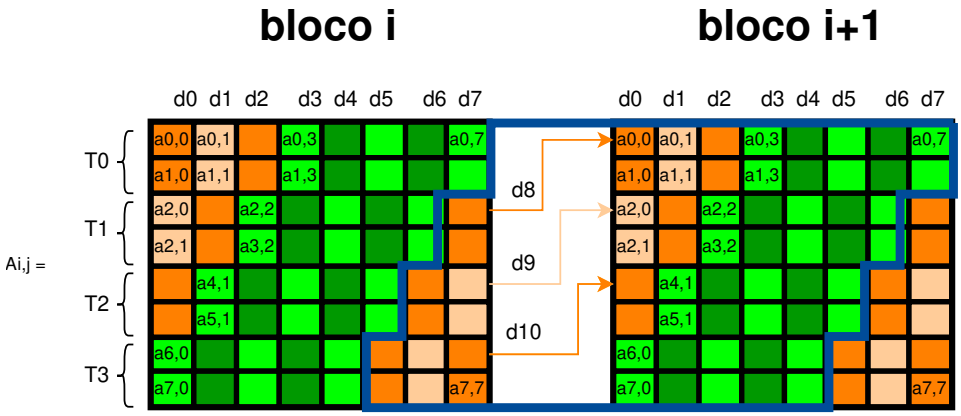


Figura 4.3: Representação da delegação de células.

são executadas juntamente a outros blocos, porém elas não dão continuidade a eles. Por fim, é necessário criar um bloco extra apenas para executar as células em d, uma vez que não há um bloco seguinte para delegar o seu processamento.

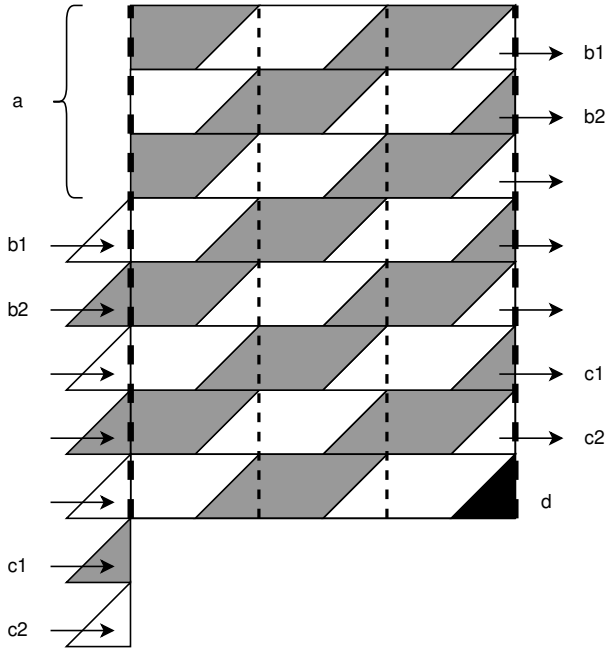


Figura 4.4: Representação externa da delegação de células. Adaptado de [4].

4.1.4 Divisão de Fases

A técnica da delegação de células, explicada na seção 4.1.3 anterior, melhora o grau de paralelismo, mas traz consigo um problema: as diagonais pendentes de um bloco possuem dependências para um bloco da diagonal seguinte. Além disso, o *CUDA*, plataforma utilizada para executar o *CUDAlign 1.0*, pode executar os blocos de uma diagonal em qualquer ordem e, tanto em série,

quanto em paralelo. Isto é problemático, pois dependendo da ordem de execução dos blocos, as dependências geradas na fase de delegação de células podem resultar em erros no cálculo da matriz de similaridade [4].

Analisando-se a Figura 4.5 percebe-se que há três blocos, sendo o de número 1 pertencente a uma diagonal externa e os blocos 2 e 3 pertencentes à diagonal posterior. Dessa maneira, o bloco 1 será executado primeiro e suas diagonais internas finais, indicadas pelo triângulo de cor amarela, ficarão pendentes para o bloco 3. Suponha que no início da execução da diagonal seguinte o *CUDA* decida executar primeiramente o bloco 2. Nesse caso haverá um problema, pois o cálculo das últimas diagonais internas do bloco 2, representadas pelo triângulo vermelho, dependem das células contidas no triângulo amarelo, cujo processamento ficou pendente para o bloco 3. Isto fará com que o bloco 2 leia e utilize valores indefinidos no cálculo dos elementos restantes, de maneira a preencher a matriz com valores incorretos.

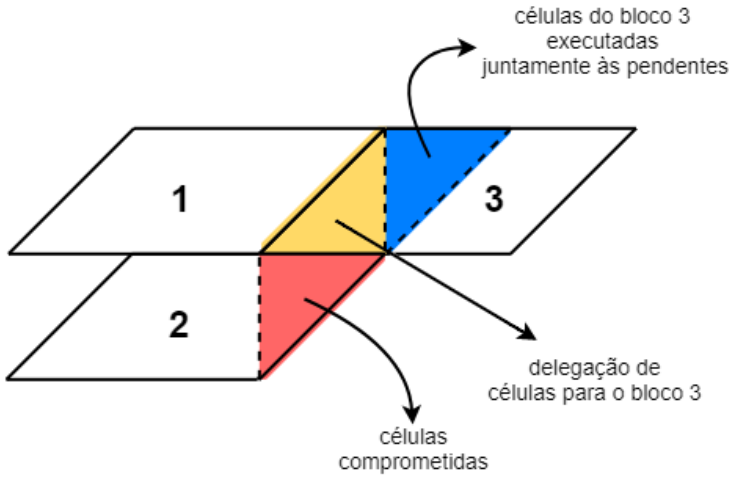


Figura 4.5: Exemplo do problema da dependência de dados. Adaptado de [4]

Para resolver tal problema é necessário primeiramente processar todas as diagonais pendentes e então realizar a sincronização entre blocos, de maneira a atualizar os valores calculados, para só então calcular o restante do bloco. Nesse contexto, a maneira que o *CUDAlign 1.0* resolve isso é por meio da divisão da execução em duas fases: a fase longa e a fase curta.

- **Fase Curta**: Nesta fase realiza-se o cálculo das $T - 1$ diagonais internas pendentes da diagonal externa anterior. Esta fase é chamada de fase curta, pois o número de colunas de um bloco costuma ser muito maior que o seu número de *threads*. Assim, esta fase processa muito menos diagonais internas do que a anterior.
- **Fase Longa**: Ocorre após a fase longa. Esta fase é responsável pela execução das $C-(T-1)$ diagonais internas de cada bloco da diagonal externa atual.

A Figura 4.6 representa tal separação de fases, na qual todos os números .1 representam a fase curta, enquanto todos os .2 representam a fase longa. Note que o processamento do conjunto de

células 2.1, na fase curta, garante a independência entre os blocos 2 e 3, na fase longa.

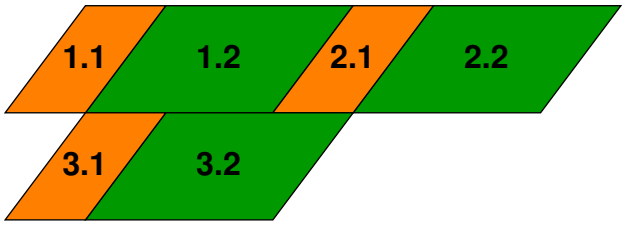


Figura 4.6: Ilustração da solução da dependência de dados, utilizando a divisão de fases. Adaptado de [4]

4.2 CUDAlign 2.0

O *CUDAlign 2.0* é uma versão aprimorada do *CUDAlign 1.0* e tem por objetivo obter, além do escore ótimo e as coordenadas desse ponto, um dos alinhamentos locais ótimos que resultam nele, utilizando memória linear e tempo reduzido [4]. Para alcançar tal feito, o *CUDAlign 2.0* baseia-se em *Myers & Miller* (Seção 3.2.5) e *FastLSA*. O algoritmo é separado em seis estágios, que serão detalhados a seguir. A Figura 4.7 ilustra e explica brevemente o procedimento realizado em cada estágio.

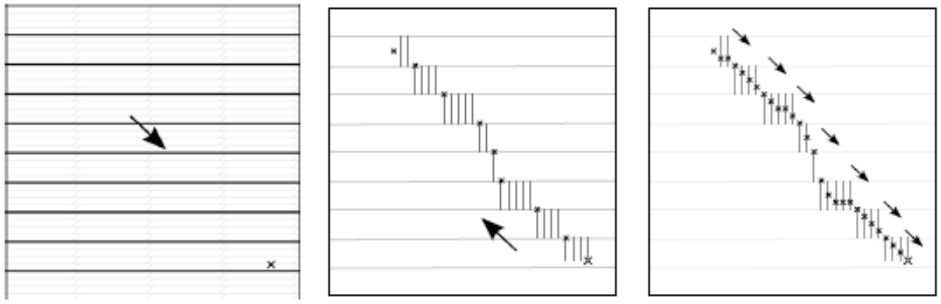
4.2.1 Estágio 1

Este estágio é muito similar ao que ocorre no *CUDAlign 1.0*, pois sua função é obter o elemento de escore ótimo da matriz. A diferença entre eles é que o estágio 1, além de obter tal elemento, também armazena, em disco, algumas linhas da matriz, chamadas de linhas especiais, que serão utilizadas posteriormente, no estágio 2.

4.2.2 Estágio 2

O objetivo desse estágio consiste em encontrar, em cada uma das linhas especiais, armazenadas no estágio anterior, um elemento pelo qual passa o alinhamento ótimo. O *CUDAlign 2.0* utiliza de duas técnicas para realizar esse processo: o procedimento de *matching* baseado em objetivo; e a execução ortogonal.

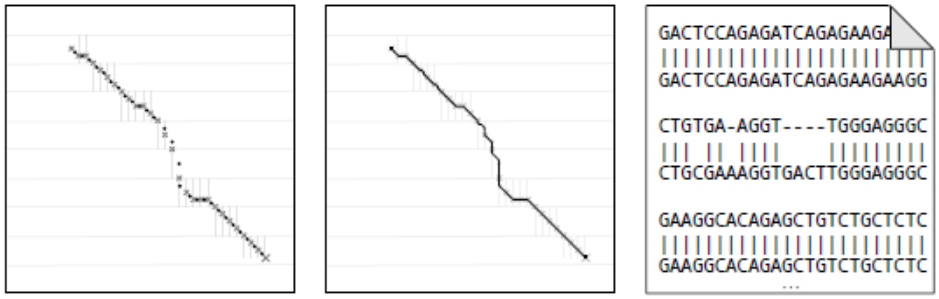
Matching baseado em objetivo: no algoritmo de *Myers & Miller*, todas as linhas de uma determinada região são processadas, em sentido inverso, até obter-se a linha central da região em questão. Em seguida, compara-se a soma de cada elemento das linhas centrais obtidas nos sentidos direto e inverso, de maneira a encontrar o ponto de valor máximo, sendo ele pertence ao alinhamento. O estágio 2 do *MASA-CUDAlign* utiliza uma ideia similar a esta, mas com algumas modificações. Diferentemente de *Myers & Miller*, o ponto de início do *traceback* é o de escore local ótimo e, à cada novo ponto calculado, pertencente ao alinhamento, obtém-se o escore do elemento



(a) O Estágio 1 encontra o escore ótimo e sua posição. Linhas especiais são salvas em disco.

(b) O Estágio 2 encontra as coordenadas nas quais o alinhamento ótimo intercepta as linhas especiais. Colunas especiais são salvas em disco.

(c) O Estágio 3 encontra as coordenadas que interceptam o alinhamento ótimo pelas colunas especiais salvas no Estágio 2.



(d) O Estágio 4 executa o algoritmo de Myers e Miller em cada partição formada por coordenadas sucessivas.

(e) O Estágio 5 obtém o alinhamento completo concatenando o alinhamento de cada partição.

(f) O Estágio 6 permite a visualização textual e gráfica do alinhamento ótimo obtido.

Figura 4.7: Estágios do *CUDAlign* 2.0. Retirado de [4].

da próxima linha especial, chamado de escore-alvo. Dessa maneira, não é necessário somar todos os elementos das linhas, pois basta interromper o processo ao achar as coordenadas da célula, cujo escore for igual ao escore-alvo.

Execução Ortogonal: Outra vantagem de já se saber o escore-alvo é que, ao invés de se calcular todas as linhas, por completo, até obter a linha central, o estágio 2 calcula individualmente as colunas, compreendidas entre as linhas especiais, até que se obtenha o elemento cujo escore das posições das duas linhas somadas seja igual ao escore-alvo. Este procedimento é chamado de execução ortogonal, pois o sentido de execução muda de horizontal (linhas) para vertical (colunas). A principal vantagem deste método consiste na redução da quantidade de processamento necessária para se obter os pontos do alinhamento.

4.2.3 Estágio 3

O estágio 3 é muito similar ao estágio 2. Seu objetivo é obter mais pontos do alinhamento e, para isso, utiliza o mesmo procedimento adotado previamente. A diferença reside no fato de que os pontos calculados no estágio anterior, agora delimitam regiões, com pontos de início e fim bem definidos, pelas quais o alinhamento ótimo deve passar. O conhecimento dessas regiões permite

a criação de partições. Cada partição fica responsável por executar uma região e, pelo fato de as partições serem independentes, pode-se processá-las em qualquer ordem, possibilitando o seu paralelismo e tornando o processo muito mais rápido do que no estágio anterior.

4.2.4 Estágio 4

O objetivo do estágio 4 é calcular iterativamente mais pontos do alinhamento, até que o tamanho da partição seja menor do que um determinado valor máximo. Cada iteração obtém aproximadamente o dobro de pontos conhecidos, de forma que são necessárias algumas iterações até que a partição atinja o tamanho máximo desejado. Assim como no estágio anterior, as partições são independentes e portanto, podem ser executadas em paralelo.

O algoritmo utilizado no estágio 4 para obter os pontos do alinhamento consiste numa otimização de *Myers & Miller* (Seção 3.2.5). Isto porque, ao invés de sempre realizar a divisão das partições pela linha central, no estágio 4 as divisões podem ser realizadas tanto na linha quanto na coluna central. A divisão é realizada no meio da maior das coordenadas. A vantagem dessa otimização é proporcionar partições com dimensões mais próximas, evitando que fiquem muito largas ou compridas, e assim reduzir a quantidade de divisões necessárias para se alcançar o tamanho desejado.

4.2.5 Estágio 5

O estágio 5 termina de identificar os últimos pontos do alinhamento, completando-o através da utilização do algoritmo de *Needleman-Wunsch* (Seção 3.2.2) nas regiões restantes da partição. O tamanho máximo das partições, definida no estágio anterior, possibilita que o estágio 5 ocorra mais rapidamente. Além disso, como o tamanho das partições é constante, o uso de memória também é, proporcionando a obtenção do alinhamento ótimo com o uso de memória linear.

4.2.6 Estágio 6

Este é o último estágio do *CUDAlign 2.0* e é optativo. Sua função é apenas de representar o alinhamento obtido, seja por meio de texto ou graficamente. Este estágio é útil quando é necessária uma análise mais detalhada do alinhamento obtido e, para auxiliar nesse processo, foi desenvolvida uma interface gráfica que facilita a visualização e interação com o alinhamento.

4.3 CUDAlign 2.1

O *CUDAlign 2.1* traz uma otimização ao algoritmo, chamada de *Block Prunning*. Sua função é acelerar o processamento do estágio 1, que calcula a matriz de programação dinâmica, quando se tem por objetivo obter apenas um alinhamento local ótimo. Para tanto, essa técnica realiza o descarte de alguns blocos, reduzindo assim a área total de processamento. Vale ressaltar que nem

todos os blocos são descartáveis, apenas aqueles cujo escore é tão baixo, que é matematicamente impossível fazerem parte do alinhamento ótimo.

4.3.1 Definições

Suponha duas sequências, S_0 e S_1, de tamanho m e n, bem como uma função $p(i, j)$, que identifica a pontuação atribuída ao resultado da comparação, isto é, $p(i,j) = ma$, em caso de *match* ou $p(i,j) = mi$, em caso de *mismatch*.

Todas as células da matriz possuem uma distância, em número de células, de sua coluna atual, até a coluna final, Δ_j, bem como de sua linha atual à final, Δ_i. Nesse contexto, as células cujas coordenadas apresentam menor distância, em linhas do que em colunas ($\Delta_i < \Delta_j$), são chamadas de $\Delta_i - cells$ e as que apresentam menor distância em colunas ($\Delta_j < \Delta_i$), são chamadas de $\Delta_j - cells$, conforme está representado na Figura 4.8

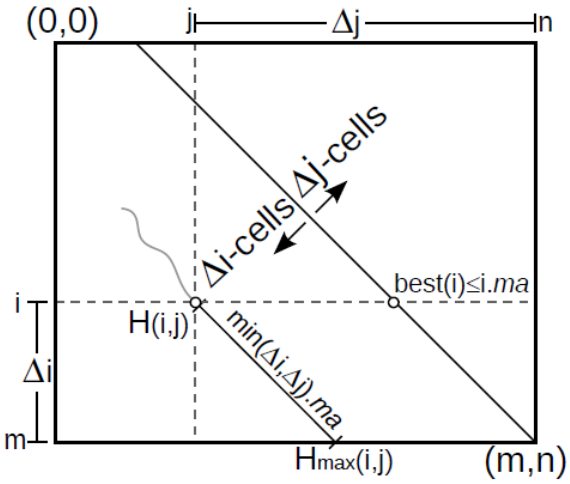


Figura 4.8: Esquemático de *Prunning*. Retirado de [4].

Sendo $H(i, j)$ o escore de uma célula de coordenadas (i, j), o escore máximo possível, de um alinhamento gerado por ela, é dado por $H_{max}(i, j) = ma \cdot min(\Delta_i, \Delta_j)$, ou seja, esse escore máximo ocorre quando a comparação de todos os elementos, das sequências $S_0[i]$ e $S_1[j]$, seguintes à essa célula, resultarem em *match* [4]. Note que, quando um *match* ocorre, tanto a coordenada i, quanto a coordenada j, das sequências $S_0[i]$ e $S_1[j]$, são incrementadas em 1 célula. Assim, a quantidade de *matches* de um alinhamento é limitada à menor distância, da célula em questão, ao final da matriz.

4.3.2 Procedimento de Pruning

O valor do maior escore obtido até o momento é denominado $best(i)$, que será sempre igual ou menor ao escore máximo: $best(i) \leq i \cdot ma$, Figura 4.8. Nesse contexto, uma célula é considerada *prunable*, e portanto pode ser descartada, se o escore, ao final de seu alinhamento, não é capaz

de superar o maior escore obtido até as células de sua linha, ou seja, $H_{max}(i,j) < best(i)$. Dessa maneira, é possível evitar o cálculos de diversas regiões da matriz, o que acelera bastante o algoritmo. Além disso, o procedimento de *pruning* pode ser generalizado para descartar blocos, ao invés de apenas células, e por isso o nome *Block Pruning*. O descarte de blocos é vantajoso, pois descarta maiores regiões, mais rapidamente.

4.4 CUDAlign 3.0

O *CUDAlign* 2.1 busca reduzir o tempo de execução do estágio 1 por meio do descarte de blocos e, consequentemente, da área total de processamento. Já o *CUDAlign* 3.0 utiliza uma abordagem diferente, por meio da utilização de várias GPUs [4].

4.4.1 Arquitetura Multi-GPU

A primeira medida dessa abordagem é atribuir, para cada GPU, um intervalo de colunas que ela deve processar. O tamanho do intervalo atribuído à cada GPU varia de acordo com seu poder de processamento, isto é, GPUs com menor capacidade de processamento recebem intervalos menores, enquanto as com maior poder de processamento, recebem intervalos menores. Se todas as GPUs tiverem a mesma capacidade, os intervalos possuirão o mesmo tamanho. Dessa maneira, o tempo de processamento de cada GPU se torna mais próximo, equilibrando a carga de trabalho.

À cada GPU atribui-se um processo composto por três *threads*, sendo duas de comunicação e uma de gerenciamento. A *thread* gerente tem como função administrar a execução da GPU à qual está vinculada, bem como de transferir as células calculadas entre a GPU e as *threads* de comunicação. As *threads* de comunicação, por sua vez, tem como objetivo transferir as células entre processos, de maneira assícrona, por meio de *sockets*. A Figura 4.9 ilustra as *threads* de gerenciamento e de comunicação, bem como suas interações inter e intra GPU. Note que tanto a última quanto a primeira GPU possuem apenas uma *thread* de comunicação, visto que elas não precisam, respectivamente, receber e enviar células para outras GPUs [4].

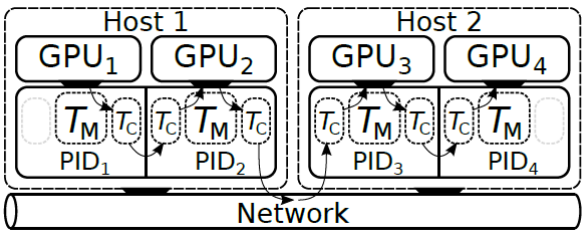


Figura 4.9: Ilustração das *threads* de comunicação e gerenciamento de cada GPU . Retirado de [4].

4.4.2 Buffers de Comunicação

Cada *thread* de comunicação está associada a um *buffer* circular, que serve como armazenamento para o envio e retirada de dados. A utilização desses *buffers* permite que a transferência de dados entre os processos ocorra de maneira assíncrona e, portanto, sem a necessidade de bloquear os processos das GPUs. Isso, por sua vez, é vantajoso pois ajuda a reduzir o efeito da latência referente à comunicação entre os processos, bem como o *overhead* e variações na qualidade de serviço da rede.

A utilização dos *buffers* pode trazer vários benefícios ao processamento multi-GPU, entretanto, para que a sua utilização tenha um bom desempenho, é fundamental que haja um bom balanceamento de processamento entre as GPUs. Isto porque, se uma GPU escreve muito mais rápido no *buffer* do que a outra consegue ler, eventualmente o *buffer* ficará sem espaço para armazenar novos dados. Analogamente, se a GPU de escrita não processar tão rapidamente quanto a de leitura, será necessário bloquear o processo da GPU seguinte para que os dados necessários sejam preenchidos. Dessa maneira, é necessário estabelecer um equilíbrio entre as GPUs, de maneira que a velocidade de escrita de uma e leitura da outra seja próxima [4].

4.5 CUDAlign 4.0

Diferentemente das contribuições das versões anteriores, o *CUDAlign* 4.0 busca otimizar o processo de obtenção do *traceback* com várias GPUs, por meio de modificações nos estágios de 2 à 4. Para tanto, utiliza-se de duas estratégias: o *Pipelined Traceback (PT)* e o *Incremental Speculative Traceback (IST)* [4].

4.5.1 Pipelined Traceback (PT)

O estágio 2 do *CUDAlign* busca obter as coordenadas de alguns *crosspoints*, de maneira a delimitar regiões pelas quais o alinhamento ótimo pode passar. Entretanto, o fato de não se conhecer as coordenadas dos crosspoints, e portanto as respectivas partições do alinhamento, no início do estágio 2, gera uma cadeia de dependência entre as partições e uma consequente serialização da computação, dificultando a utilização de múltiplas GPUs de maneira eficiente.

A técnica de PT tenta resolver este problema. Para tanto, considere a GPU p como a última GPU executada no estágio 1. Ela própria iniciará a execução do estágio 2, buscando obter, a partir do ponto de escore ótimo, as coordenadas do próximo *crosspoint*. Após a obtenção do *crosspoint*, a GPU p passa então essas coordenadas para a GPU anterior, $p-1$, que por sua vez calculará o próximo *crosspoint* e assim sucessivamente, até o final do estágio 2. Além disso, como o próprio nome já diz, o *PT* realiza o *traceback* segundo o modelo *pipeline* e assim, após enviar o *crosspoint* para a GPU anterior, a própria GPU p já inicia o estágio 3 e, em seguida, o estágio 4. Este procedimento é adotado por todas as GPUs subsequentes. Já os estágios 5 e 6, por exigirem menor tempo computacional, são executados posteriormente utilizando apenas uma GPU [4].

A Figura 4.10 ilustra a técnica do *Pipelined Traceback*, com o eixo x representando a GPU que está sendo executada e o eixo y representando o tempo transcorrido. Note que o tempo t_a corresponde ao tempo que determinada GPU i demora para iniciar sua execução, visto que no *wavefront* não há como todas as GPUs serem iniciadas simultaneamente. Ainda no estágio 1, t_1 representa o tempo de computação da GPU i e t_b o tempo na qual ela permanece ociosa. Por fim, note que t_2, t_3 e t_4 são, respectivamente, os tempos da execução parcial de cada estágio e que eles são processados em *pipeline*, conforme explicado previamente.

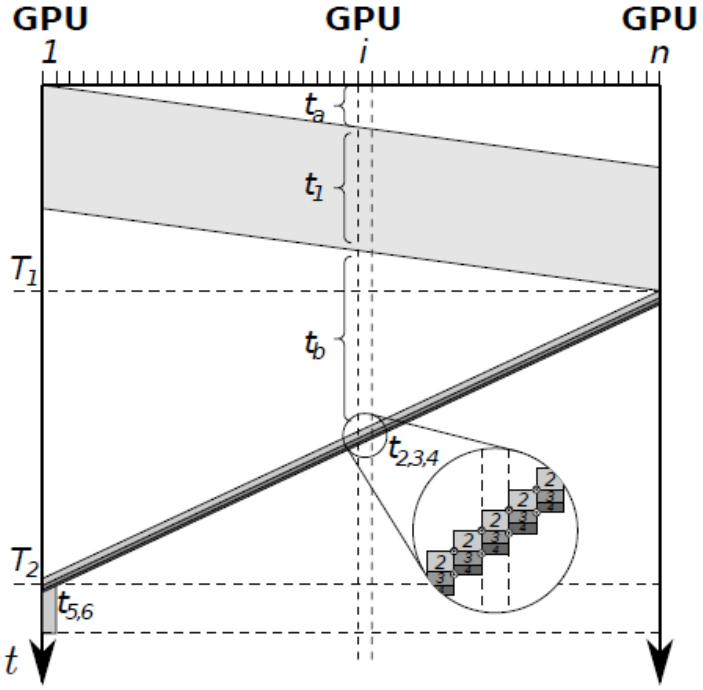


Figura 4.10: Técnica do *Pipelined Traceback*. Retirado de [4].

4.5.2 Incremental Speculative Traceback (IST)

Analisando-se várias execuções do *Pipelined Traceback*, notou-se que os *crosspoints* gerados normalmente coincidiam com pontos de escore máximo das colunas intermediárias. Nesse contexto, o *IST* realiza uma série de especulações de possíveis candidatos à *crosspoints* antes mesmo de eles serem calculados, justamente para tentar otimizar o uso das múltiplas GPUs, bem como para reduzir o tempo total de execução.

As especulações são iniciadas ainda no estágio 1, logo após cada GPU encerrar a sua execução e são finalizadas até que se encerre o processamento da especulação ou até que o processamento do estágio 2 retorne à GPU em questão. Então a GPU que estiver executando o estágio 2 recebe o ponto especulado e verifica se este possui o escore-alvo. Em caso positivo, esta GPU simplesmente passa adiante as coordenadas do *crosspoint* e continua seu *pipeline*. Caso contrário, a GPU terá que recalcular o ponto de *crosspoint* e, consequentemente, os limites da partição. A Figura 4.11 ilustra esse procedimento.

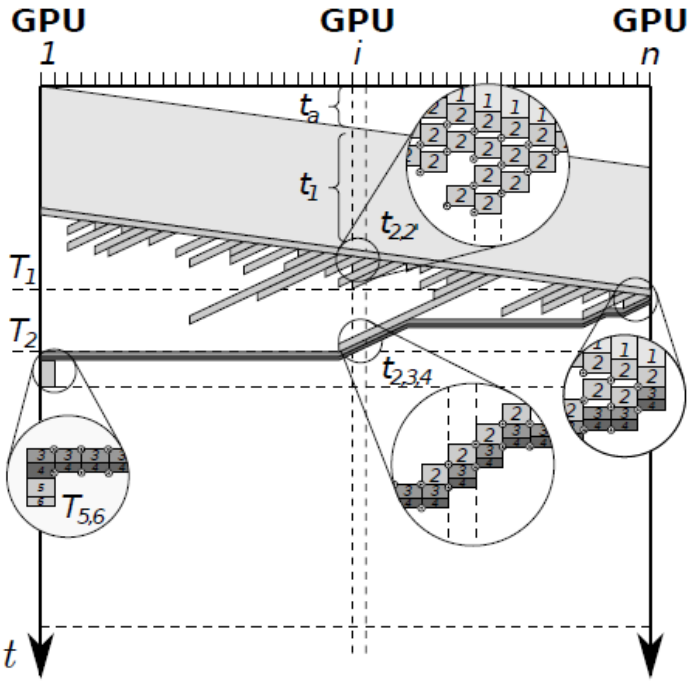


Figura 4.11: Técnica do *Incremental Speculative Traceback*. Retirado de [4].

4.6 Static-MultiBP

Como visto anteriormente, o *CUDAlign* 2.1 beneficia o algoritmo de comparação de sequências biológicas por meio da técnica de *Block Pruning*. Tanto o *CUDAlign* 3.0 quanto o 4.0 executam o algoritmo em múltiplas GPUs, porém não incorporam a técnica de *Block Pruning*, da versão 2.1. O *Static-MultiBP* foi então desenvolvido por Figueirêdo Júnior [5] e implementa algumas modificações no algoritmo do *CUDAlign*, desenvolvido por Sandes [4], de maneira a tornar possível a utilização da técnica de *Block Prunning* em múltiplas GPUs e, consequentemente, acelerar a execução do estágio 1. Vale ressaltar que, além do *Static-MultiBP*, também foi implementado o *Dynamic-MultiBP*, porém como não foi utilizado neste trabalho, ele não será explorado nas próximas seções.

4.6.1 Contextualização do Problema

Dois fatores constituíram obstáculos para a implementação do *Static Multi-BP*. O primeiro está relacionado à forma como as coordenadas das células são atribuídas, em múltiplas GPUs, enquanto o segundo se relaciona ao desconhecimento do processamento de dados entre diferentes GPUs.

Assim como explicado previamente, uma das primeiras ações do *CUDAlign* 3.0 consiste em dividir o total de colunas em partições. Cada GPU receberá uma partição, de tamanho proporcional ao seu poder de processamento. Além disso, os índices de cada partição são relativos, o que significa que sempre se iniciam em (1,1) e encerram no tamanho da partição, (x,y). Entretanto,

a atribuição relativa de índices é problemática ao procedimento de *pruning*, uma vez que este utiliza das coordenadas globais para verificar a descartabilidade de uma célula, ou bloco. Este foi o primeiro obstáculo.

Para que uma célula, ou bloco, seja descartado é necessário comparar o melhor escore que determinada célula poderia gerar ao final de seu alinhamento, com o melhor escore obtido, até então, na matriz. Porém, como cada GPU tem acesso apenas às colunas calculadas por ela mesma, frequentemente pode-se desconhecer escores maiores obtidos por outras GPUS, reduzindo a quantidade de blocos selecionados para descarte. Este foi o segundo obstáculo. Na Seção 4.6.2 será mostrado como o *Static Multi-BP* tratou esses problemas.

4.6.2 Projeto do Static Multi-BP

Para solucionar o primeiro problema, o *Static Multi-BP* ajusta o índice da primeira coluna da partição de acordo com sua posição na matriz completa e, a partir disso, todos os índices seguintes são recalculados tendo este como base. Além disso, no *CUDAlign* 2.1, como só se tinha uma GPU, considerava-se a distância ao final da matriz como a coordenada final da partição da respectiva GPU. Já no *Static Multi-BP*, considera-se esta distância como a última coluna da última GPU e a última linha da GPU que está realizando o cálculo.

O segundo problema se concentra no fato de cada GPU apenas conhecer o melhor escore local corrente (*CLBS - Current Local Best Score*), ou seja, o escore de sua partição. A solução proposta pelo *Static Multi-BP* consiste em cada GPU enviar periodicamente seu *CLBS* às outras. Para que o envio desses dados prejudique o mínimo possível o processamento da matriz, foram implementadas as *threads assíncronas*, T_S, que se comunicam de maneira assíncrona e independente em relação às outras *threads* do *CUDAlign*. A interação dessas *threads* está representada na Figura 4.12. Cada GPU possui uma *thread* assíncrona, que recebe a numeração dentro do intervalo de 1 à P, sendo P a quantidade total de GPUS. Além disso, cada *thread* assíncrona, i se comunica com a seguinte, $((i+1) mod P)$, em uma topologia em anel, de forma que a última se comunica com a primeira. As *threads* se comunicam periodicamente informando seu CLBS para a seguinte. Ao receber o CLBS da *thread* anterior compara-se se o escore recebido é maior que o CLBS e, se for, o valor do CLBS é atualizado com o valor rebido, que passará a ser utilizado nas operações de *pruning* [5].

4.7 Comentários Finais

O objetivo deste capítulo foi explicar as diferentes versões do *MASA-CUDAlign*, ferramenta de comparação de sequências biológicas escolhida para ser executada no *AWS ParallelCluster*. As duas primeiras versões da ferramenta definem o funcionamento geral do algoritmo, enquanto as seguintes implementam otimizações e melhorias em seus estágios. Conhecer o funcionamento desta ferramenta foi fundamental para entender os requisitos necessários para executá-la no *ParallelCluster*, que serão abordados no próximo capítulo, assim como a metodologia adotada nesta monografia.

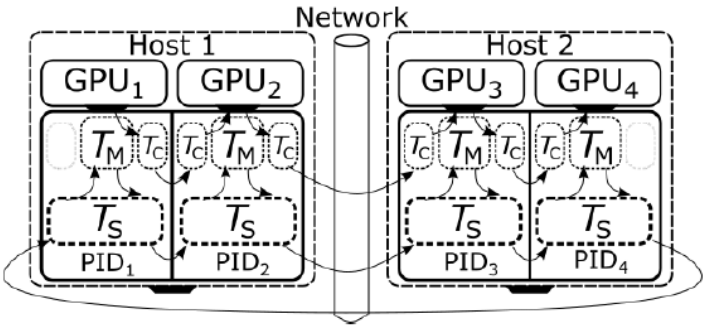


Figura 4.12: Inclusão das *threads assíncronas* no modelo de comunicação do *CUDAlign*. Retirado de [5].

Capítulo 5

Projeto de Integração do MASA-CUDAlign ao AWS ParallelCluster

O projeto desenvolvido nesta monografia tem por objetivo utilizar a ferramenta *AWS ParallelCluster* para configurar e criar um *cluster*, na nuvem da *Amazon*, e, a partir disso, integrar este ambiente à ferramenta de comparação de sequências biológicas, *MASA-CUDAlign*, na versão *Static-MultiBP*. O *cluster* é formado por uma instância mestre, que é responsável por designar tarefas às demais instâncias, chamadas de nós computacionais, que de fato as executam.

5.1 Requisitos

O projeto implementado nesta monografia é de caráter exploratório, ou seja, analisa algumas possibilidades de configurações de *cluster*, de maneira que outros usuários possam posteriormente utilizá-lo para identificar os modelos que melhor se adequem a seus requisitos, em termos de custo e tempo de execução. As restrições adotadas neste trabalho foram: utilizar instâncias (máquinas virtuais), que custassem menos de 1$ por hora; e selecionar um tipo de instância contendo poucas *VCPUs*, de maneira a respeitar o limite estabelecido pela *AWS* e possibilitando a utilização de mais nós computacionais.

5.2 Instalação e Configuração do ParallelCluster

A primeira etapa deste projeto consistiu em instalar e configurar a ferramenta *AWS ParallelCluster*. Esta é responsável por automatizar a criação de todos os recursos do *cluster*, como as instâncias, o tipo e quantidade de armazenamento alocado, a rede *VPC (Virtual Private Cloud)* que conecta os *hosts*, dentre outros. Antes porém de criar um *cluster*, o usuário deve preencher um arquivo de configurações, que registra as especificações desejadas.

5.2.1 Seleção de Instâncias do *Parallel Cluster*

Um dos principais parâmetros definidos no arquivo de configuração é o tipo de instância que comporá os *hosts* do *cluster*, ou seja, suas unidades de processamento. A *AWS* possui uma ampla variedade de instâncias, cada uma com diferentes especificações em termos de GPU, memória, preço, dentre outras. Neste contexo, surge o questionamento de por qual delas optar.

O artigo [32] seleciona um conjunto de instâncias, da *AWS*, para executar a ferramenta *MASA-CUDAlign*, em uma GPU, e em seguida as compara, de maneira a verificar qual delas proporciona o melhor custo-benefício. Usamos as instâncias consideradas no artigo para a nossa decisão, que baseou-se nos quesitos custo, processamento gráfico e tempo de execução. Primeiramente utilizou-se o custo para selecionar apenas as instâncias mais baratas (abaixo de 1$) e que tivessem uma boa capacidade de processamento gráfico (foco na família g), uma vez que o *MASA-CUDAlign* requer boas GPUs para o seu processamento. Já o tempo de execução relatado em [32] foi utilizado para avaliar a relação custo-benefício de cada instância. Ao final dos testes, a instância que apresentou os melhores resultados foi a *g4dn.xlarge*. Baseando-se neste resultado, este trabalho também optou por utilizar a instância *g4dn.xlarge* para a implementação do *ParallelCluster*.

Todas as instâncias da *AWS* possuem uma quantidade de *Virtual Central Processament Unit (VCPUs)*, ou seja, CPUs virtuais. A *AWS* impõe um limite, da quantidade máxima de *VCPUs*, que determinado usuário pode utilizar, restringindo assim suas opções de execução. No caso do *cluster*, esta medida impacta diretamente na quantidade de instâncias que este poderá usufruir.

A *g4dn.xlarge*, por exemplo, possui 4 *VCPUs*, de maneira que, para criar um *cluster* com um mestre e quatro escravos, ou seja, cinco instâncias, seria necessário um limite de 20 *VCPUs*, porém o limite inicial disponibilizado era de 0 *VCPUs*, para a família *g*. Para resolver este problema, foi necessário solicitar, à *AWS*, o aumento dos limites impostos. Isto foi feito diversas vezes, pois a quantidade de *VCPUs* requisitada nunca era plenamente atendida, mas apenas incrementada aos poucos. Ao final desse processo, o limite alcançado foi de 36 *VCPUs*, o que possibilitou a utilização de até nove instâncias do tipo *g4dn.xlarge*, um mestre e oito escravos.

5.2.2 Arquivo de Configuração

Assim como dito no início da Seção 5.2, antes de criar o *cluster* é necessário preencher um arquivo de configurações, contendo todas as especificações desejadas pelo usuário. As etapas desse procedimento, que serão descritas a seguir, foram baseadas no *AWS ParallelCluster User Guide*, que é um manual confeccionado pela *AWS* para ensinar o usuário a utilizar o *ParallelCluster* [33].

Primeiramente é necessário instalar, na máquina do usuário, duas dependências: *python* 3 e o *pip*. Estas são necessárias para possibilitar o *download* do *AWS CLI*, ferramenta utilizada para gerenciar serviços da *AWS*. Em seguida é necessário executar esta ferramenta e configurá-la com as credenciais do usuário e região de associação dos serviços da *AWS*.

Em seguida é necessário instalar alguma ferramenta capaz de criar ambientes virtuais isolados, de maneira que qualquer configuração indesejada ou prejudicial ao usuário fique restrita ao

ambiente virtual criado pela ferramenta. Após ativado o ambiente virtual, instala-se a ferramenta *AWS ParallelCluster* e então pode-se iniciar o preenchimento do arquivo de configuração. O preenchimento das especificações será realizado via terminal e, uma vez concluído, tais informações serão armazenadas no arquivo de configuração, que poderá ser novamente utilizado para criar um *cluster* com as mesmas configurações. Vale ressaltar que, além das especificações apresentadas no terminal, também é possível adicionar diversas outras, que estão listadas no *website* da *AWS* [34].

A primeira informação a ser definida é a região na qual o *cluster* irá residir, Figura 5.1. Nos testes realizados, a região *us-east-1*, que corresponde ao Norte da Virgínia, foi a que apresentou os menores preços e por isso foi escolhida. Em seguida, deve-se escolher o escalonador de tarefas, Figura 5.2. Escolheu-se o *slurm* pois este é muito utilizado por ambientes de alto desempenho que executam ferramentas de comparação de sequências biológicas. Em relação ao sistema operacional, optou-se pelo *ubuntu 1804*, Figura 5.3.

```
Allowed values for AWS Region ID:
1. ap-northeast-1
2. ap-northeast-2
3. ap-south-1
4. ap-southeast-1
5. ap-southeast-2
6. ca-central-1
7. eu-central-1
8. eu-north-1
9. eu-west-1
10. eu-west-2
11. eu-west-3
12. sa-east-1
13. us-east-1
14. us-east-2
15. us-west-1
16. us-west-2
AWS Region ID [us-east-1]:
```

Figura 5.1: Seleção de região.

```
Allowed values for Scheduler:
1. sge
2. torque
3. slurm
4. awsbatch
Scheduler [slurm]:
```

Figura 5.2: Escolha do escalonador.

```
Allowed values for Operating System:
1. alinux2
2. centos7
3. centos8
4. ubuntu1804
5. ubuntu2004
Operating System [alinux2]: 4
```

Figura 5.3: Seleção do sistema operacional.

A próxima configuração compreende a quantidade máxima e mínima de nós computacionais (Seção 2.4.2). O modelo de execução do *MASA-CUDAlign* requer um número fixo de instâncias. Para manter essa quantidade constante é preciso selecionar a mesma quantidade de instâncias em ambos os campos, Figura 5.4. Ainda em relação às instâncias, escolheu-se o tipo *g4dn.xlarge*, conforme explicado na Seção 5.2.1.

```
Minimum cluster size (instances) [0]: 2
Maximum cluster size (instances) [10]: 2
Head node instance type [t2.micro]: g4dn.xlarge
Compute instance type [t2.micro]: g4dn.xlarge
```

Figura 5.4: Quantidade e tipo de instância.

Por fim, deve-se criar uma *VPC (Virtual Private Cloud)* (Seção 2.4.4). Nos testes realizados, observou-se a necessidade de selecionar a opção *Automate VPC creation* a cada criação de um novo *cluster*, pois caso contrário o IP elástico não será automaticamente desassociado da respectiva *VPC*, de maneira que essa desassociação deverá ser feita manualmente para só então ser possível deletar a *VPC* e interromper a sua cobrança, Figura 5.8. A outra configuração da *VPC* diz respeito à configuração da rede, isto é, se o mestre deve ficar em uma sub-rede privada e os nós numa pública ou se todos devem compartilhar uma mesma sub-rede pública. Nos testes realizados verificou-se não haver diferença entre as opções, no que diz respeito ao funcionamento do *Static-MultiBP* no *cluster*.

```
Automate VPC creation? (y/n) [n]: y
Allowed values for Network Configuration:
1. Head node in a public subnet and compute fleet in a private subnet
2. Head node and compute fleet in the same public subnet
Network Configuration [Head node in a public subnet and compute fleet in a private subnet]: 1
```

Figura 5.5: Criação da *VPC*.

Vale ressaltar que a criação de uma *VPC* só é permitida aos usuários que possuírem permissões mais restritas na *AWS*. Aos demais usuários, a criação da *VPC* é interrompida por uma mensagem de erro. A conta utilizada, neste projeto, até então não possuía as permissões necessárias para criação da *VPC* e esta lista das permissões, além de extensa, não foi bem definida pela *AWS*, de maneira que não se sabia quais delas eram necessárias. Devido a esses fatores, optou-se por utilizar uma conta de administrador para dar continuidade ao projeto.

5.2.3 Criação e Funcionalidades do *Cluster*

Realizado o preenchimento de todas as informações, o arquivo de configuração está completo e então é possível iniciar um novo *cluster* por meio do comando:

```
pcluster create nome_do_cluster
```

Após cerca de 10 minutos o cluster será criado. Para se conectar ao mestre é necessário utilizar um par de chaves, criado pelo usuário, no seguinte comando. A Figura 5.6 apresenta a tela do terminal conectado ao *cluster*.

```
pcluster ssh nome_do_cluster -i nome_da_chave.pem
```

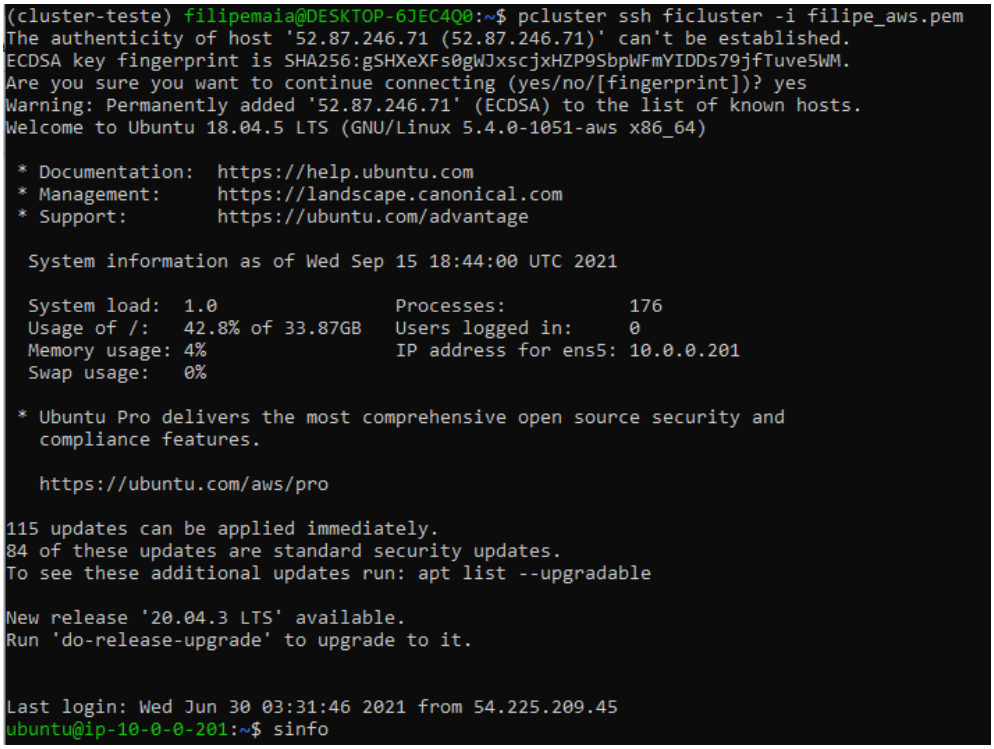


Figura 5.6: Terminal do usuário conectado ao mestre.

Uma vez criado o *cluster*, é possível acessar o site da *AWS* e visualizar os diversos serviços utilizados pelo *ParallelCluster*, como: instâncias, *VPC* e volumes, respectivamente nas Figuras 5.7, 5.8 e 5.9.

É importante salientar que o *cluster* já é entregue ao usuário com alguns recursos básicos instalados e configurados, como: o escalonador de tarefas, *CUDADriver* e sistema operacional. Além disso, o sistema de arquivos é único e compartilhado entre todas as instâncias, sejam elas mestre ou nós computacionais.

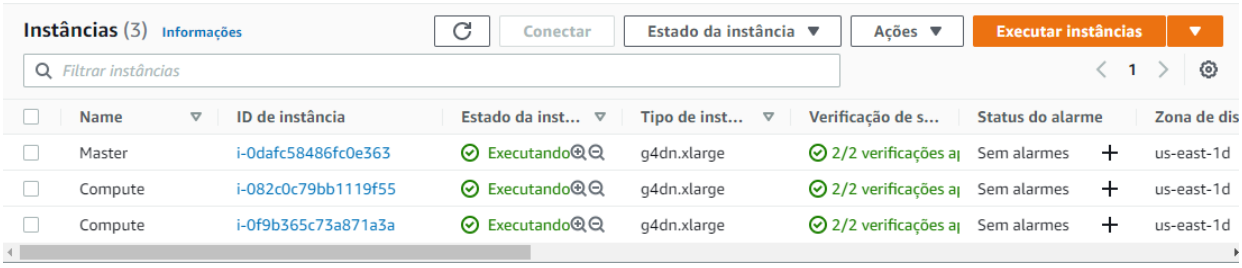


Figura 5.7: Lista de instâncias criada pelo *cluster* (1 mestre e 2 nós).

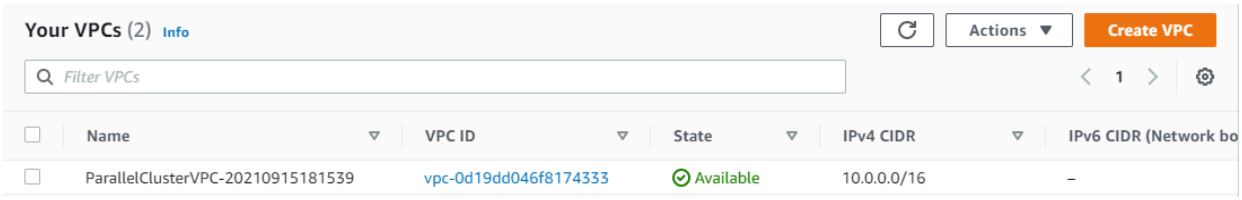


Figura 5.8: *VPC* criada pelo *cluster*.

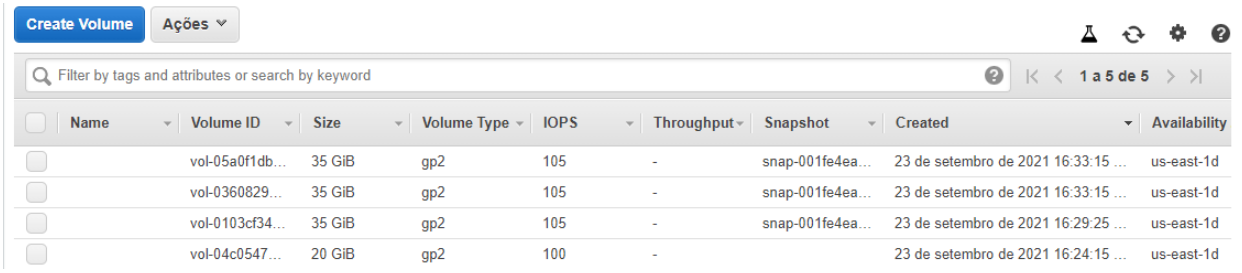


Figura 5.9: Volumes alocados pelo *cluster*.

Por fim, foi feita uma lista com alguns comandos utilizados neste trabalho para interagir com o *cluster*.

- SCP: Utilizado para transferir arquivos da máquina do usuário para o *cluster* -> scp -i nome_da_chave.pem nome_do_arquivo DNS_IPV4_público
- SINFO: Visualiza as instâncias que compõem o *cluster*, seus nomes e estados (ativo ou inativo) -> sinfo
- SQUEUE: Imprime na tela a tabela de processos do escalonador -> squeue
- SBATCH: Envia um programa à fila de processos do escalonador -> sbatch nome_do_programa

5.3 Integração do MASA-CUDAlign ao ParallelCluster

5.3.1 Instalação e Configuração do MASA-CUDAlign

O primeiro requisito para proporcionar a integração do *MASA-CUDAlign* ao *ParallelCluster* consiste em realizar a instalação desta ferramenta. A versão do *MASA-CUDAlign* escolhida para este projeto foi a *Static-MultiBP* (Seção 4.6) uma vez que esta combina a estratégia de múltiplas GPUs, em paralelo, com a técnica de *Block Pruning*.

Dessa maneira, utilizou-se o comando *scp* (Seção 5.2.3) para transferir o arquivo do *Static-MultiBP* e as sequências a serem comparadas, da máquina do usuário, para o *cluster*. Em seguida realizou-se a instalação da ferramenta e suas variáveis de ambiente foram configuradas.

5.3.2 Visão Geral

A Figura 5.10 apresenta, simplificadamente, as etapas necessárias para executar o *Static-MultiBP* no *ParallelCluster*. Uma vez que o usuário tenha se conectado ao *cluster* e configurado a ferramenta *MASA-CUDAlign*, é necessário adotar um escalonador de tarefas para gerenciar as cargas de trabalho. O escalonador utilizado neste projeto foi o *SLURM (Simple Linux Utility for Resource Management)*. A escolha deste foi realizada no preenchimento do arquivo de configurações (Seção 5.2.2) e, assim como explicado na Seção 5.2.3, este já vem instalado e configurado no ambiente do *ParallelCluster*.

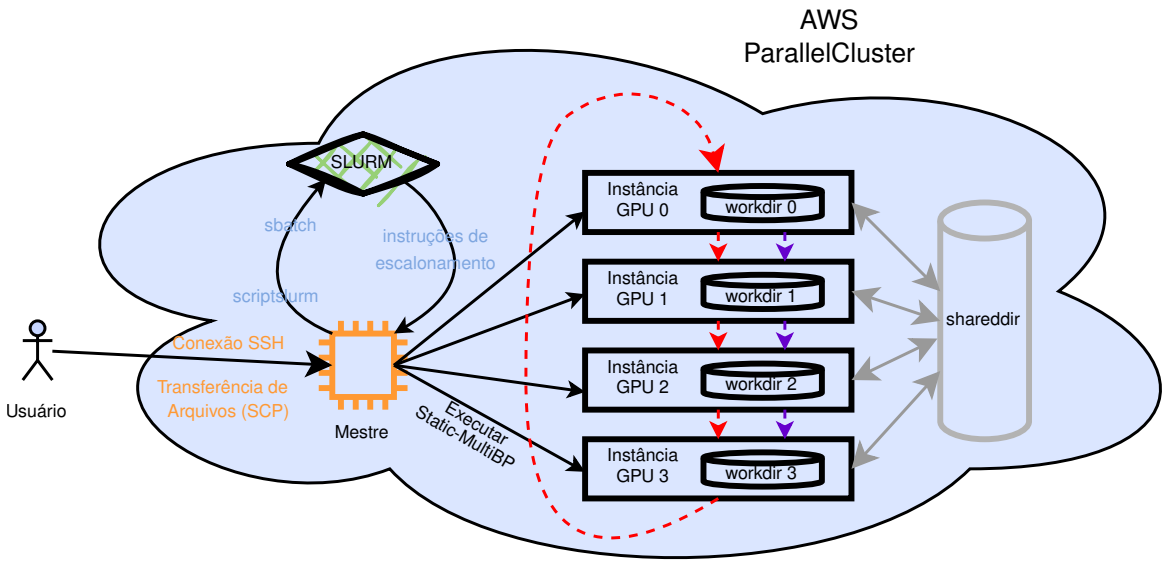


Figura 5.10: Execução do *Static-MultiBP* no *ParallelCluster*.

Ainda analisando a Figura 5.10, são utilizados dois *scripts* para a submissão das execuções, o *sbatch-nvidia-x* e o *scriptslurm* (Seções 5.3.3.1 e 5.3.3.2). Ambos foram desenvolvidos por Figueirêdo Júnior [35] e posteriormente modificados para se adequarem ao ambiente de nuvem. No

primeiro são estabelecidas as configurações do *job*, enquanto no segundo são estabelecidos os comandos que propiciam a comunicação entre as GPUs. Ambos os *scripts* são enviados do mestre do *cluster* ao escalonador, que os processa e retorna as instruções de escalonamento. O mestre utiliza essas instruções para designar as tarefas de cada nó computacional. Cada nó calcula a respectiva parte da matriz e compartilha os dados calculados com o nó seguinte (seta em roxo, Figura 5.10). Na estratégia de *Block Pruning* a área de poda é proporcional ao valor do maior escore obtido até o momento. No *Static-MultiBP*, cada *host* envia periodicamente o valor do seu maior escore obtido ao vizinho da direita, utilizando uma topologia em anel (Figura 4.12), que é representada por setas vermelhas na Figura 5.10.

Cada nó possui um diretório *workdir* utilizado para armazenar dados da própria execução. Este diretório é privado e não deve ser acessado por outros, de maneira evitar inteferência de dados entre as GPUs. Além disso, também é criado um único diretório *shareddir*, que serve para propiciar o compartilhamento de dados entre as GPUs. Este diretório deve estar em um sistema de arquivos compartilhado, de maneira todas as GPUs possam acessá-lo.

5.3.3 Scripts para Automatizar a Execução do Static-MultiBP

5.3.3.1 Funcionamento do Script Sbatch-NVIDIA-x

O *script sbatch-nvidia-x* define as características de execução. A Figura 5.11 apresenta as configurações utilizadas.

```
1  #!/bin/bash
2  #
3  #SBATCH --job-name=cudalign2
4  #SBATCH --ntasks=2
5  #SBATCH --nodes=2
6  #SBATCH --time=09:59:00
7  #
8
9  gpus=1
10
11 srun scriptslurm.sh $1 $gpus
12
```

Figura 5.11

Os comandos apresentados na Figura 5.11 são explicados na lista abaixo:

- **#SBATCH –job-name=cudalign2:** Atribui um nome ao processo;
- **#SBATCH –ntasks=2:** Define a quantidade de tarefas que serão designadas para executar o job referente ao script submetido;
- **#SBATCH –nodes=2:** Define a quantidade de nós computacionais que serão utilizados na execução;

- **#SBATCH –time=09:59:00:** Tempo máximo para finalização da execução;
- **#gpus=1:** Define a quantidade de GPUs por instância;
- **srun scriptslurm.sh $1 $gpus:** Submete o script "scriptslurm" para execução. Também é necessário informar dois parâmetros, que neste caso são: nome que designa as sequências a serem comparadas e quantidade de GPUs por instância.

5.3.3.2 Funcionamento do Scriptslurm Original

O *scriptslurm* tem a função de iniciar a execução do *Static-MultiBP* nas GPUs e estabelecer a comunicação entre elas. Primeiramente será detalhado o funcionamento do *script* original, desenvolvido em [35]. O fragmento abaixo representa um pseudocódigo simplificado do funcionamento do *script*. O programa foi implementado em *Shell Script* e recebe dois parâmetros de entrada, do terminal: $1, *string* que identifica as sequências a serem comparadas; $2, quantidade de GPUs por instância.

```
sequencias = parametro_terminal1
qtd_gpus = parametro_terminal2

cria lista com enderecos IPs
loop n vezes
    ip_atual = lista.IP(n)
    ip_anterior = lista.IP(n-1)
    ip_posterior = lista.IP(n+1)

switch case
    1-3M
    2-5M
    .
    .
    chrY
case
cria 1 shareddir
cria 1 workdir por GPU

socket GPU(n-1) -> GPU (n)
socket GPU(n) -> GPU (n+1)
```

A etapa inicial do *script* consiste em identificar as instâncias disponibilizadas para o *cluster* e criar uma lista contendo os seus IPs. Para tanto, o *script* utiliza a função *getips*, que obtém uma lista de nomes padronizados dos *hosts* disponíveis e os converte em endereços IP. Em seguida, utiliza-se o ID do processo para percorrer esta lista e obter o endereço IP do *host* corrente. A partir disso também são obtidos os endereços IP dos *hosts* anterior e posterior. O conhecimento desses IPs permitirá a comunicação, via *socket* entre as GPUS.

A seguir é feito um *switch-case*, utilizando o segundo parâmetro passado pelo terminal, para

identificar qual comparação será executada. Nessa etapa também são criados os diretórios *shared-dir* e *workdir*, que proporcionam, respectivamente o armazenamento de arquivos compartilhados e privados ao processo.

A última etapa do processo consiste em, de fato, executar o *Static-MultiBP*. Para tanto, o mestre deve enviar um comando, à cada GPU, ordenando a execução da ferramenta, de acordo com os parâmetros definidos pelo escalonador e estabelecendo a comunicação entre as GPUs. O comando enviado para cada GPU possui a mesma estrutura, mas parâmetros com valores diferentes. O fragmento a seguir apresenta a estrutura deste comando.

```
cudalign --stage-x --no-flush --blocks=y --shareddir=./nome_shareddir --workdir
    =./nome_workdir --gpu=z --split-proporcao --part=n --load-column=socket://IP_
    GPU_anterior:porta_anterior --flush-column=socket://IP_GPU_atual(127.0.0.1):
    porta_posterior sequencia1 sequencia2
```

Note que é preciso especificar os seguintes quesitos:

- *stage*: qual(is) estado(s) do *Static-MultiBP* deve(m) ser executado(s).
- *no-flush*: não salvar *Special Rows*.
- *blocks*: refere-se ao parâmetro B (Seção 4.1.1), que define a quantidade de blocos para as quais serão divididas as colunas da matriz.
- *shareddir*: diretório para compartilhamento de dados entre os processos. Esta pasta deve estar em um sistema de arquivos compartilhado por todas as GPUs. Seu nome deve ser o mesmo no comando enviado a todas as instâncias.
- *workdir*: diretório único para cada GPU. Serve para que cada GPU armazene os dados de sua execução e não deve ser compartilhado, para evitar que os dados de uma interfiram nas demais.
- *gpu*: índice que define qual das GPUs da instância está sendo executada.
- *split*: índice que define a proporção de colunas da matriz que cada GPU ficará responsável por processar.
- *part*: indica qual parte da execução esta *GPU* ficará responsável por executar.
- *flush-column*: à este parâmetro devem ser informados o endereço IP da própria GPU, bem como a porta destinada à comunicação entre esta e a próxima. Ao utilizar este comando, a GPU em questão fica esperando uma comunicação, via socket, com a GPU para a qual ela enviará os dados processados.
- *load-column*: este parâmetro indica o outro lado da comunicação, via *socket*. Deve-se informar o endereço IP da GPU anterior, de onde serão carregados os dados, assim como a porta, aberta pela GPU anterior, por onde os dados serão transferidos.

- *sequencia1* e *sequencia2*: caminho para as sequências que serão comparadas.

A comunicação é realizada entre GPUs, que podem ser de um mesmo *host* ou de *hosts* diferentes. Dessa maneira, o *scriptslurm* necessita identificar à qual GPU, de qual *host* da lista será enviado o comando. Esta identificação é fundamental para definir os parâmetros de cada comando.

Por exemplo, a primeira GPU do primeiro *host* não recebe dados de nenhuma outra, uma vez que ela mesma inicia o processamento da matriz. Dessa maneira, o comando enviado a esta GPU não deve incluir o parâmetro *load-column*, que é responsável por carregar dados de outra GPU. Entretanto, esta GPU ainda deve enviar os dados processados para a próxima GPU do mesmo *host* e portanto deve incluir o parâmetro *flush-column*.

Além de incluir ou não determinados parâmetros, identificar a GPU do *host* à qual está sendo construído o comando também é importante para definir os valores das variáveis. Por exemplo, os parâmetros *load-column* e *flush-column* necessitam definir corretamente os valores das variáveis porta e endereço IP, da GPU de destino da comunicação.

5.3.4 Execução Manual

Os procedimentos descritos na Seção 5.3.3.2 explicam o funcionamento do *scriptslurm* implementado por Figueirêdo Júnior, que não pôde ser diretamente aplicado ao *ParallelCluster*. Até então todos os parâmetros e variáveis eram definidos manualmente. Dessa maneira, sempre que se desejava alterar a quantidade de GPUs no *cluster* era necessário redefinir todos os parâmetros da Seção 5.3.3.2. Por este processo ser extremamente trabalhoso, decidiu-se modificar o *scriptslurm*, de maneira a se adequar ao ambiente do *AWS ParallelCluster*.

5.3.4.1 Script Modificado

O objetivo desta seção consiste em apresentar e explicar as modificações realizadas no *scriptslurm*, para que este se tornasse compatível com as características do *ParallelCluster*. A primeira das alterações ocorreu na função *getips*, que converte os nomes das instâncias em IPs e os organiza em uma lista. No *ParallelCluster* as instâncias também possuem nomes padronizados, que são dados da seguinte maneira.

```
compute-st-tipo_da_instancia-[1-n]
```

O tipo de instância utilizado em todos os testes foi o *g4dn.xlarge*. O parâmetro n indica a quantidade de nós computacionais adotados no *cluster*. Dessa maneira, os nomes compartilham da mesma estrutura, variando-se apenas a numeração ao final do nome, como é mostrado no fragmento abaixo, para um caso de 4 nós computacionais.

```
Exemplos de nomes:
```

```
compute-st-g4dn.xlarge-1
compute-st-g4dn.xlarge-2
compute-st-g4dn.xlarge-3
compute-st-g4dn.xlarge-4
```

Entretanto, diferentemente do *cluster* utilizado no contexto do *script* original, o *AWS ParallelCluster* consegue utilizar os nomes das máquinas no lugar de seus IPs e vice-versa, evitando a necessidade de realizar a tradução dos nomes em IPs. Além disso, como neste caso *cluster* é totalmente dedicado ao usuário, utilizou-se o comando *scontrol*, do *SLURM*, para retornar uma lista com os nomes de todas as instâncias ativas, Figura 5.12.

```
22 # Função que recupera os nomes dos hosts
23 function getips ()
24 {
25     nodelist=`scontrol show hostnames compute-st-$instancia-[1-$nos]`
26     nodeips=$nodelist
27     echo $nodeips
28 }
```

Figura 5.12: Listagem dos nomes das instâncias.

Note, da Figura 5.12, que a função *scontrol* necessita especificar o nome das instâncias utilizadas no *cluster*, bem como a quantidade de nós computacionais utilizados. Para tanto foram criadas as variáveis *instancia* e *nos*, que podem ser alteradas de acordo com as especificações do *cluster*.

A outra modificação realizada se deu na etapa de comunicação entre as GPUs. O *script* original foi projetado para trabalhar sempre com múltiplas GPUs por instância, o que não é o caso da instância *g4dn.xlarge*, escolhida para formar o *cluster*, que possui apenas uma. Dessa maneira, no script modificado, não é necessário verificar a posição da GPU para a qual será enviado o comando, apenas a posição do host.

5.4 Equação de Previsão de Tempo

Em ambiente com recursos limitados é fundamental ter alguma maneira de estimar o tempo de execução da comparação de duas sequências. Em particular, ao utilizar o *SLURM*, deve ser especificado o tempo máximo de execução (Figura 5.11). Dessa maneira, em [5] foi desenvolvida a Equação 5.1, que realiza esta previsão de tempo utilizando uma instância e a ferramenta *StaticMultiBP*, com *Block Pruning* ativo.

$$\log_{10}(t) = -3,036 + 0,979 \cdot \log_{10}(m \cdot n) - 3,044 \cdot \log_{10}(CP) - 1,001 \cdot \log_{10}(BW) + 0,777 \cdot \log_{10}(1 - BP) \tag{5.1}$$

Os valores de m e n se referem aos tamanhos das sequências, em quantidade de bases. Já CP se refere ao poder computacional da GPU, que é dada pelo produto entre a quantidade de núcleos

e a sua frequência de operação, em MHz. O parâmetro BW consiste na largura de banda da memória, dada em Gb/s. Finalmente, BP é o valor absoluto da taxa de pruning.

O valor da taxa de *pruning (BP)* é teórico e depende apenas das sequências utilizadas na comparação, porém só pode ser obtido após a realização da comparação. Dessa maneira, não é possível utilizar a Equação 5.1 para obter uma estimativa de tempo antes de se realizar a comparação de sequências cuja taxa de *pruning* é desconhecida. Apesar disso, caso a taxa de *pruning* e as informações da GPU, listadas no parágrafo anterior, sejam conhecidas, é possível realizar a estimativa de tempo para outros ambientes, como foi o caso da previsão realizada no *ParallelCluster*, cujos resultados serão descritos na Seção 6.4.

5.5 Comentários Finais

O projeto desenvolvido nesta monografia consistiu em realizar a "cloudificação" do *MASA-CUDAlign*. Esta não foi uma tarefa simples e, por isso, foi subdividida em objetivos menores. Inicialmente realizou-se a instalação e configuração do *AWS ParallelCluster*, por meio do preenchimento do arquivo de configurações, com as informações do usuário. Em seguida, realizou-se a integração do *MASA-CUDAlign* ao *ParallelCluster*, por meio da modificação de dois *scripts*, que realizam a submissão das execuções. Por fim, foi realizada uma previsão do tempo de execução das sequências, para as comparações que utilizam apenas uma GPU. No próximo capítulo serão discutidos os testes realizados neste projeto, bem como os resultados obtidos.

Capítulo 6

Resultados Experimentais

O presente capítulo possui como objetivo descrever os experimentos realizados no projeto, ou seja, avaliar os parâmetros e escolhas adotados para a construção e execução do *ParallelCluster*. Além disso, também foi feita uma análise dos resultados obtidos com enfoque na comparação destes utilizando uma, duas, quatro ou oito GPUs. Por fim realizou-se uma previsão do tempo de execução do algoritmo *Static-MultiBP* utilizando *Block Pruning* em apenas uma GPU, com base na equação desenvolvida em [5].

6.1 Sequências Utilizadas

Foram selecionadas dez sequências biológicas para comparação, no *ParallelCluster*, com tamanhos variando de $3MBP$ até $83MBP$(*Million Base Pairs*). A escolha destas foi baseada no estudo realizado por [5], que já havia utilizado grande parte delas para obter seus resultados. A Tabela 6.1 apresenta as sequências escolhidas, em ordem crescente de tamanho, e algumas de suas especificações, tais como: identificação, nome e tamanho, além da taxa de *Block Pruning* da comparação. Note que as últimas cinco sequências comparam cromossomos do ser humano com os do chimpanzé. As sequências foram obtidas do *website* do *National Center for Biotechnology Information (NCBI)*, que disponibiliza uma base de dados com diversas informações genômicas e biomédicas [36].

6.2 Ambiente de Execução

O ambiente utilizado para a execução do algoritmo *Static-MultiBP* (Seção 4.6) consiste em um *AWS ParallelCluster*, constituído por um mestre e $1, 2, 4$ ou 8 nós computacionais do tipo *g4dn.xlarge*, cada um possuindo uma única GPU *NVIDIA T4*, de arquitetura *Turing*. As especificações desta GPU encontram-se na Tabela 6.2. A região escolhida para a criação do *cluster* foi o Norte da Virgínia, no leste do Estados Unidos (*us-east-1*), por apresentar, em média, os menores preços. O sistema operacional utilizado foi o *ubuntu* na distribuição 1804 e o escalonador de tarefas selecionado foi o *slurm*. Note, da Tabela 6.2, que a performance da rede depende do tipo

Comp.	Seq.	Id	Nome	Tamanho	BP
3M	Seq1	BA000035.2	*Corynebacterium efficiens YS-314*	3147090	0
	Seq2	BX927147.1	*Corynebacterium glutamicum ATCC 13032*	3282708	
5M	Seq1	AE016879.1	*Bacillus anthracis str. Ames*	5227293	0,53
	Seq2	AE017225.1	*Bacillus anthracis str. Sterne*	5228663	
7M	Seq1	NC_003997.3	*Bacillus anthracis str. Ames*	5227293	0
	Seq2	NC_005027.1	*Rhodopirellula baltica SH 1 chromosome*	7145576	
10M	Seq1	NC_014318.1	*Amycolatopsis mediterranei U32 chromosome*	10236715	0,53
	Seq2	NC_017186.1	*Amycolatopsis mediterranei S699 chromosome*	10236779	
23M	Seq1	NT_033779.4	*Drosophila melanogaster chromosome 2L*	23011544	0
	Seq2	NT_037436.3	*Drosophila melanogaster chromosome 3L*	24543557	
47M	Seq1	BA000046.3	*Pan troglodytes DNA, chromosome 22*	32799110	0,48
	Seq2	NC_000021.7	*Homo sapiens chromosome 21*	46944323	
chr21	Seq1	NC_000021.9	*Homo sapiens chromosome 21*	46709983	0,34
	Seq2	NC_006488.4	*Pan troglodytes chromosome 21*	33445071	
chrY	Seq1	NC_000024.10	*Homo sapiens chromosome Y*	57227415	0,07
	Seq2	NC_006492.4	*Pan troglodytes chromosome Y*	26350515	
chr22	Seq1	NC_000022.11	*Homo sapiens chromosome 22*	50818468	0,29
	Seq2	NC_006489.4	*Pan troglodytes chromosome 22*	37823149	
chr17	Seq1	NC_000017.11	*Homo sapiens chromosome 17*	83257441	0,22
	Seq2	NC_006484.4	*Pan troglodytes isolate Yerkes chimp pedigree*	83181570	

Tabela 6.1: Parâmetros das sequências selecionadas.

de instância escolhida, de maneira que instâncias mais robustas e caras possuem melhor serviço de rede. Também é importante ressaltar que a frequência de *clock* da *g4dn.xlarge* se inicia com um valor mais baixo($585MHz$), que pode ser aumentada até o valor máximo de $1590MHz$, à depender da carga de trabalho.

Instância	Última atualização de preço verificada em 03/10/2021			
g4dn.xlarge	GPUs	Memória (GB)	Performance da Rede (GB/s)	Preço ($/h)
	1	16	Até 25	0,526
	Arquitetura		Clock (MHz) / BW (GB/s)	Núcleos
	Turing		585 ->1590 / 320	2560

Tabela 6.2: Especificações da instância *g4dn.xlarge*.

6.3 Resultados Obtidos

Ao todo foram realizadas duzentas execuções, pois foram feitas dez comparações de sequências, cada uma executada cinco vezes para cada quantidade de GPUs (1, 2, 4 e 8). O objetivo desses testes foi avaliar a quantidade GPUs que proporciona o melhor custo-benefício em termos de tempo e de custo. A partir das cinco repetições de cada comparação foram calculadas as médias de tempo e de desvio padrão, de maneira a proporcionar a análise de confiabilidade dos dados obtidos nos testes. A Tabela 6.3 apresenta os tempos médios de execução e outros resultados, como taxa de *GCUPS* (Seção 6.3.2), *Speedup*, desvio padrão e o custo médio referente à cada comparação.

6.3.1 Tempo de Execução

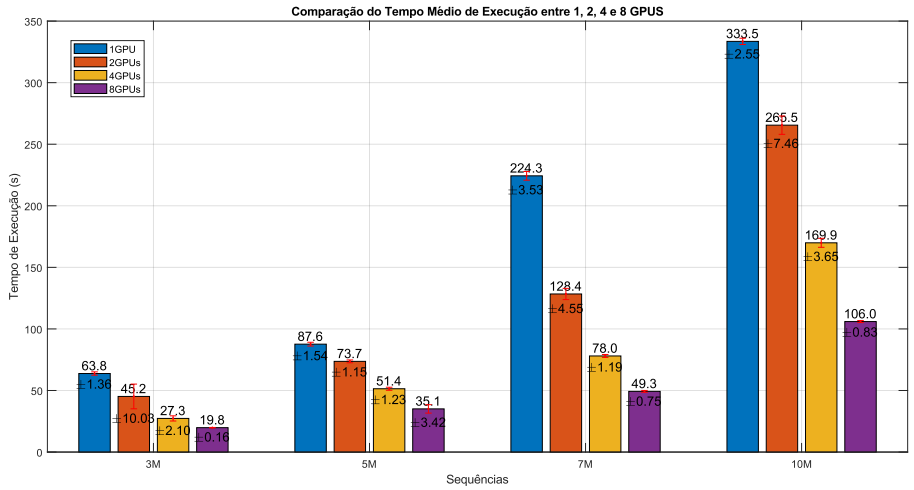
As sequências da Tabela 6.3 foram organizadas em ordem crescente de tamanho. Este parâmetro é um dos que mais influencia no tempo de execução e é possível verificar este efeito analisando a tabela, uma vez que, de maneira geral, à cada comparação que se passa, o tempo de execução tende a aumentar.

Outro ponto importante de se observar das tabelas é o fato de que, à exceção de três casos, o desvio padrão permaneceu abaixo de 5%, indicando que houve pouca variação entre os resultados e portanto certa repetibilidade entre os testes, no ambiente de nuvem.

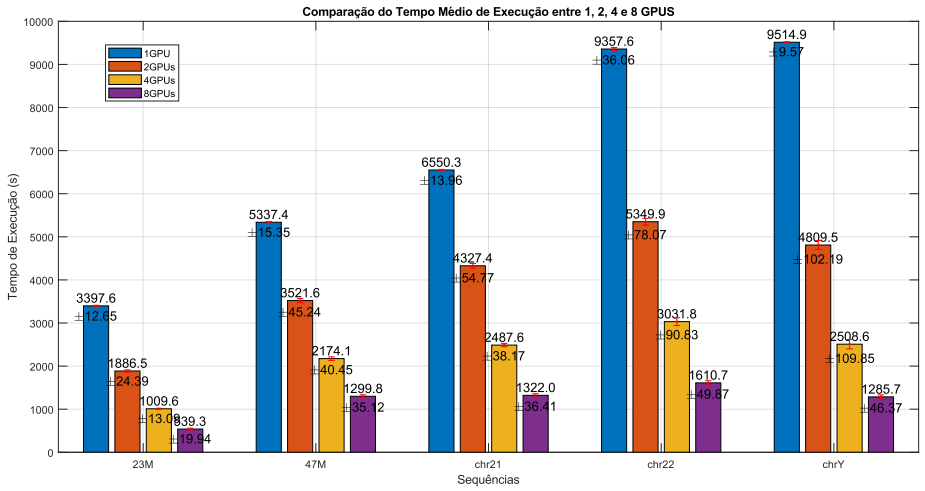
As Figuras 6.1a, 6.1b e 6.1c apresentam, de maneira mais ilustrativa, o tempo de execução e o desvio padrão das comparações das sequências menores, maiores e do cromossomo 17, para a quantidade de GPUs correspondente. Assim como explicado anteriormente, é possível visualizar que o desvio padrão, em quase todos os casos foi extremamente baixo. Além disso, verificou-se que o tempo de execução não reduziu na mesma proporção de aumento da quantidade de GPUs, como seria esperado em um caso ideal. As comparações que mais se aproximaram desse resultado, foram os cromossomos Y e 17, pois possuem maior grau de paralelismo.

Comparação	Nós	Tempo de Execução Médio (s)	GCUPS	Speedup	Desvio Padrão	Custo Estimado
3M	1	63,785275	161,96	1,00	2,14%	0,009
	2	45,16916718	228,72	1,41	22,21%	0,020
	4	27,32081994	378,14	2,33	7,70%	0,020
	8	19,806811	521,59	3,22	0,83%	0,026
5M	1	87,63424844	311,88	1,00	1,75%	0,013
	2	73,66043282	371,05	1,19	1,55%	0,032
	4	51,42015154	531,54	1,70	2,39%	0,038
	8	35,0673094	779,41	2,50	9,75%	0,046
7M	1	224,3298312	166,50	1,00	1,57%	0,033
	2	128,3898031	290,93	1,75	3,54%	0,056
	4	78,05077032	478,56	2,87	1,52%	0,057
	8	49,2910977	757,78	4,55	1,53%	0,065
10M	1	333,5279625	314,19	1,00	0,77%	0,049
	2	265,4982813	394,70	1,26	2,81%	0,116
	4	169,928125	616,68	1,96	2,15%	0,124
	8	105,9811250	988,77	3,15	0,78%	0,139
23M	1	3397,59515	166,23	1,00	0,37%	0,496
	2	1886,544075	299,38	1,80	1,29%	0,827
	4	1009,559225	559,44	3,37	1,30%	0,738
	8	539,251475	1.047,35	6,30	3,70%	0,709
47M	1	5337,3618	288,48	1,00	0,29%	0,780
	2	3521,59945	437,23	1,52	1,28%	1,544
	4	2174,1439	708,20	2,45	1,86%	1,588
	8	1299,7808	1.184,61	4,11	2,70%	1,709
chr21	1	6550,3091	238,50	1,00	0,21%	0,957
	2	4327,3748	361,01	1,51	1,27%	1,897
	4	2487,57895	628,01	2,63	1,53%	1,817
	8	1322,0065	1.181,70	4,95	2,75%	1,738
chrY	1	9514,878	158,49	1,00	0,10%	1,390
	2	4809,4608	313,54	1,98	2,12%	2,108
	4	2508,5925	601,12	3,79	4,38%	1,833
	8	1285,650725	1.172,92	7,40	3,61%	1,691
chr22	1	9357,6414	205,41	1,00	0,39%	1,367
	2	5349,9165	359,28	1,75	1,46%	2,345
	4	3031,75745	633,99	3,09	3,00%	2,215
	8	1610,7068	1.193,34	5,81	3,10%	2,118
chr17	1	37845,5512	182,99	1,00	0,36%	5,530
	2	18971,244	365,05	1,99	5,16%	8,316
	4	9565,5702	724,00	3,96	0,41%	6,988
	8	5703,7852	1.214,19	6,64	4,94%	7,500

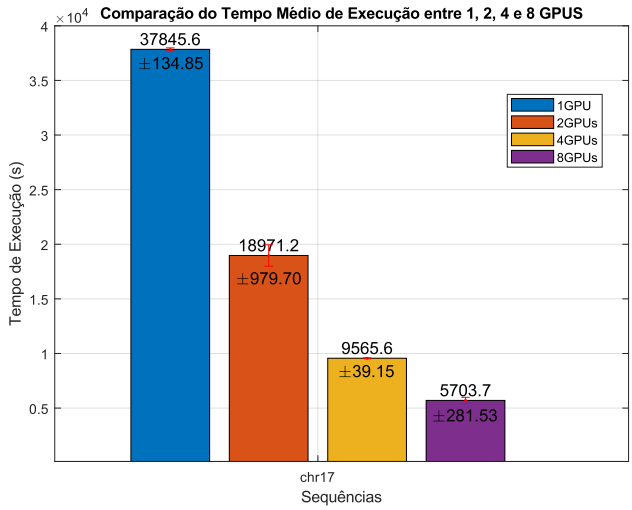
Tabela 6.3: Valores médios das execuções realizadas para 1, 2, 4 e 8 GPUs.



(a) Tempo médio de execução e desvio padrão das sequências menores (3M a 10M).



(b) Tempo médio de execução e desvio padrão das sequências maiores (23M a 57M).



(c) Tempo médio de execução e desvio padrão para o cromossomo 17 (83M).

6.3.2 Desempenho

O parâmetro *GCUPS* diz respeito à quantidade, em bilhões, de células da matriz processadas por segundo e é utilizado como indicador de desempenho. Dessa maneira, ao analisar uma mesma comparação, quanto mais GPUs forem utilizadas, maior será o paralelismo e, consequentemente, maior a taxa de *GCUPS*. Este efeito pode ser observado em todas as sequências da Tabela 6.3. Nos testes realizados, o melhor desempenho obtido foi de $1.214,19$ *GCUPS* ou $1,21419$ *TCUPS*, no cromossomo 17, utilizando 8 GPUs.

O parâmetro *speedup*, mostrado na Tabela 6.3, é dado pela razão entre o tempo de comparação utilizando n GPUs e o tempo de comparação utilizando apenas uma GPU, Equação 6.1.

$$Speedup = \frac{tempo_{nGPUs}}{tempo_{1GPU}} \tag{6.1}$$

O*speedup* indica o quão mais rápido é executar uma mesma comparação, variando apenas a quantidade de GPUs. No caso do *MASA-CUDAlign*, o valor de *speedup* ideal deveria ser igual à quantidade de GPUs utilizada, uma vez que, se a capacidade de processamento aumenta na proporção x, o tempo de execução deve cair na proporção $1/x$. Neste caso, o tempo total de execução seria o mesmo, independente da quantidade de GPUs utilizada, tornando sempre mais vantajosa a utilização de mais GPUs, porém isto não é verificado na prática. Nos testes realizados, as comparações que mais se aproximaram deste resultado foram os cromossomos Y e 17, com *speedups* de $7,40$ e $6,64$, respectivamente. Cabe ressaltar que, para as comparações *3M*, *5M* e *10M*, os *speedups* ficam um pouco abaixo de $4x$ para 8 GPUs, o que indica que um *cluster* com menos GPUs seria mais adequado para essas comparações, dado os tamanhos das sequências. A Figura 6.2 apresenta os *speedups* obtidos para cada comparação.

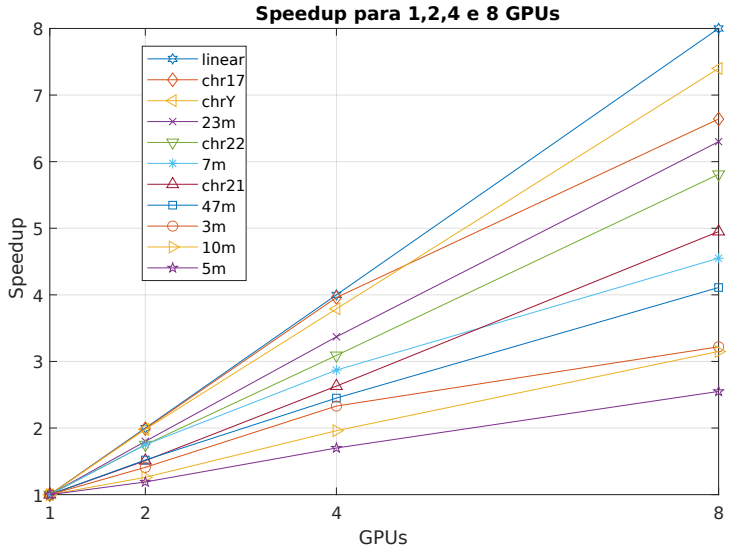


Figura 6.2: *Speedups* das sequências utilizando 1, 2, 4 e 8 GPUs.

6.3.3 Custo-Benefício do Conjunto de Comparações

A última coluna da Tabela 6.3 apresenta o Custo Estimado para cada comparação, que é calculado da seguinte maneira:

$$Custo_{estimado} = \frac{n_{hosts} \cdot t_{medio} \cdot Preco_{instancia}}{3600}$$

, sendo

- n_{hosts}: quantidade de instâncias, que compõem o *cluster*;
- t_{medio}: tempo médio de execução da respectiva comparação;
- Preço(/h): Preço da instância do tipo *g4dn.xlarge*, em $\frac{\$}{h}$

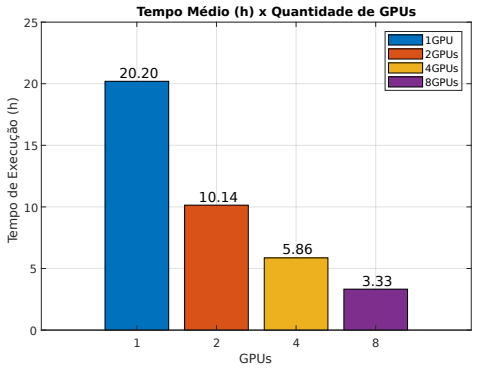
A variável quantidade de instâncias leva em consideração tanto o *host* mestre, que distribui tarefas, quanto os nós computacionais, que de fato, as executam. Dessa maneira, a quantidade total de instâncias, no caso em que se utilizam dois, quatro ou oito nós computacionais, é igual à quantidade de instâncias, acrescida em uma unidade. Entretanto, ao utilizar apenas um nó computacional, o *host* mestre não é alocado, uma vez que utiliza-se apenas uma GPU para executar todas as instruções e assim, a quantidade de instâncias é apenas igual à qunatidade de nós computacionais, no caso 1.

Para verificar a configuração que traz o melhor custo-benefício ao usuário na realização das dez comparações, agrupou-se os custos e tempos da Tabela 6.3, de acordo com a quantidade de GPUs utilizada. Este resultado encontra-se na Tabela 6.4.

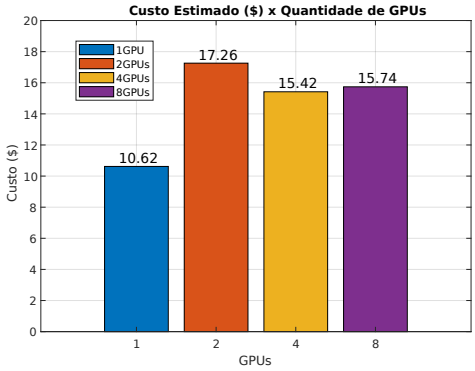
Nós	Tempo Total (h min s)	Custo Total Estimado ($)
1	20h 11min 52s	10,62
2	10h 56min 19s	17,26
4	5h 51min 44s	15,42
8	3h 19min 31s	15,74

Tabela 6.4: Relação tempo e custo para comparar todas as sequências em 1, 2, 4 e 8 GPUs.

Além dessa tabela, também foram implementados dois gráficos, Figuras 6.3a e 6.3b, para ilustrar tal comparação. Analisando-os em conjunto é possível verificar que, ao dobrar a quantidade de nós computacionais, o tempo de execução cai para próximo da metade, numa proporção um pouco abaixo de 50%.



(a) Tempo médio para comparação de todas as sequências.



(b) Custo estimado para a execuçãod e todas as sequências.

Dessa maneira, se o custo das execuções fosse proveniente exclusivamente dos nós computacionais, este iria aumentar à cada vez que se duplicasse a quantidade de GPUs, mas numa pequena proporção, Tabela 6.5. Entretanto, apesar de a instância mestre não contribuir para acelerar o processamento, seu custo ainda é contabilizado igual ao das outras. Além disso, à medida que se diminui a quantidade de GPUs, o tempo de execução aumenta e consequentemente o custo por instância também. Assim, o custo do mestre é muito representativo quando se utiliza menos instâncias e vai se diluindo à medida que são utilizadas mais instâncias. Por exemplo, em um caso com três instâncias o mestre é responsável por 33% do custo, enquanto que com cinco instâncias é 20% e com nove, 11%. Isto explica o motivo de as execuções utilizando dois nós serem as mais caras, pois apesar de se dobrar a quantidade de nós, o mesmo não ocorre para a quantidade total de instâncias, além de ser o caso em que o mestre possui o maior custo, verificar fragmento a seguir. Analisando as comparações utilizando 4 e 8 nós este efeito é amenizado pelo fato de o mestre ter uma proporção menor do total de instâncias.

Custo Total		Custo dos Nós		Custo do Mestre
Instâncias	Preço ($)	Nós	Preço ($)	Preço ($)
3	17,26	2	11,51	5,75
5	15,41	4	12,33	3,08
9	15,71	8	13,99	1,75

Tabela 6.5: Subdivisão dos custos.

Por fim, é importante destacar que, apesar de o custo das comparações aumentar à medida que se utiliza mais GPUs, este aumento é pouco significativo em comparação com o tempo de execução, que reduz significativamente. Entretanto, a escolha de qual configuração utilizar dependerá dos requisitos e dos recursos do usuário. Se o preço for o fator determinante para o usuário, a melhor opção seria utilizar apenas um nó. Já se o custo não for um fator limitante e o objetivo consistir em executar o mais rapidamente possível, 8 nós seria a opção mais adequada. Considerando a bateria de testes utilizada, a opção menos indicada é utilizar dois nós computacionais, uma vez que este caso demanda maior tempo e custo do que nos casos utilizando mais instâncias.

6.4 Previsão de Tempo

Assim como explicado na Seção 5.4, a presente monografia utilizou a equação 5.1, elaborada por [5], para verificar a previsão do tempo de execução do *Static-MultiBP* no ambiente do *ParallelCluster*. Infelizmente esta equação não foi projetada para um caso de múltiplas GPUs e portanto apenas foram comparados os tempos obtidos com os estimados, para o caso de apenas um nó computacional.

A instância *g4dn.xlarge*, escolhida para compor os *hosts* do *ParallelCluster*, possui como GPU uma *NVIDIA Tesla T4*. Os parâmetros desta GPU, necessários para o cálculo da Equação 5.1 foram retirados de [37] e estão listados na Tabela 6.2. Vale ressaltar que esta GPU possui um *clock* base de $585MHz$, para economizar energia e, de acordo com a necessidade, este pode aumentar para até $1590MHz$. Como a comparação de sequências requer muito processamento da GPU, o valor mais adequado e utilizado na equação foi o de maior frequência ($1590MHz$). A equação utilizada para a previsão, com os parâmetros da *g4dn.xlarge* é mostrada na Equação 6.2. A Tabela 6.6 mostra os tempos obtidos, estimados e o erro entre eles.

$$\begin{aligned} \log_{10} t &= \frac{-7,817 + 0,979 \log_{10}(m \cdot n) + 0,777 \log_{10}(1 - BP)}{1000} \\ t &= 10^{\frac{-7,817+0,979 \log_{10}(m \cdot n)+0,777 \log_{10}(1-BP)}{1000}} \end{aligned} \tag{6.2}$$

Comparação	Tempo Obtido (s)	Tempo Estimado (s)	Erro (%)
1 (3m)	63,79	78,32	22,79
2 (5m)	87,63	112,95	28,90
3 (7m)	224,33	276,06	23,06
4 (10m)	333,53	421,99	26,52
5 (23m)	3.397,6	3.931,77	15,72
6 (47m)	5.337,36	6.335,18	18,70
8 (chr21)	6.550,31	7.715,49	17,79
10 (chrY)	9.514,88	9.718,54	2,14
9 (chr22)	9.357,64	10.002,55	6,89
7 (chr17)	37.845,55	37.827,45	-0,05

Tabela 6.6: Comparativo entre os tempos obtidos e os estimados.

Analisando os dados da Tabela 6.6, nota-se que todos os valores de erro permaneceram abaixo de 30%, sendo o maior deles de $28,90\%$ e a média dos erros foi de $16,25\%$. Dessa maneira, os resultados obtidos, apesar de não estarem totalmente acurados, ainda são bons, pois são capazes de entregar uma razoável estimativa de tempo, ao usuário. Nota-se também que, para as comparações com as maiores sequências (chrY, chr22 e chr17) o erro é menor, enquanto o erro é maior para as comparações de menores sequências (3M, 5M, 7M e 10M).

6.5 Comentários Finais

Este capítulo apresentou as características do ambiente de execução e analisou os resultados obtidos. Ao todo foram realizadas duzentas execuções, pois foram feitas dez comparações de sequências, cada uma executada cinco vezes para cada quantidade de GPUs (1,2,4 e 8). A discussão de resultados focou nos quesitos custo, tempo e desempenho. Ao final do capítulo foi analisada a equação de previsão de tempo adotada no projeto.

Capítulo 7

Conclusão e Trabalhos Futuros

7.1 Conclusão

A presente monografia propôs, implementou e avaliou uma solução para integrar a ferramenta *MASA-CUDAlign* ao *AWS ParallelCluster*. O projeto se deu em três etapas: a primeira consistiu em criar o *cluster*, sendo necessário preencher um arquivo de configurações com as especificações desejadas, escolher as instâncias que comporiam o *cluster* e posteriormente conectar-se à ele. Na segunda instalou-se a ferramenta *Static-MultiBP* e foram configuradas as suas variáveis de ambiente. Por fim, automatizou-se as configurações das comparações, por meio da adaptação dos *scripts* utilizados (*sbatch* e *scriptslurm*), compatibilizando o seu funcionamento com o ambiente do *ParallelCluster*. Após a integração ao cluster, feita nas 3 etapas mencionadas, estimou-se o tempo necessário para execução do código, a partir de uma equação elaborada em [5] e dos parâmetros da GPU selecionada para compôr as instâncias do cluster.

Para realização dos testes foram selecionadas 10 sequências biológicas, cada uma executada 5 vezes para cada tamanho de *cluster*. O *AWS ParallelCluster* foi configurado com 1, 2, 4 e 8 instâncias de GPU e o foco de análise se deu nos quesitos tempo e custo.

Primeiramente foi possível observar que a dispersão entre os tempos de execução das comparações, medida pelos desvios padrão, permaneceu abaixo de 5%, indicando certa repetibilidade entre os testes. Também verificou-se que o tamanho das sequências tem grande influência sobre o tempo de execução, uma vez que, à medida que se comparava sequências maiores, o tempo de execução também aumentava significativamente. Isso era esperado, pois o algoritmo executado possui complexidade quadrática de tempo.

Em seguida, foi feita uma análise detalhada de desempenho. Nos testes realizados, o melhor desempenho foi obtido na comparação do cromossomo 17 ($83M$x$83M$), utilizando 8 GPUS, e foi de $1,21$ *TCUPS*, o que é um ótimo resultado. Já analisando-se o parâmetro *speedup*, percebeu-se que a redução do tempo, alterando a quantidade de nós, foi mais significativa nos cromossomos 17 e Y ($57M$x$26M$).

Também analisou-se o custo-benefício da quantidade de nós computacionais utilizadas no *clus-*

ter. A escolha de quantos nós utilizar depende da aplicação do usuário e da sua limitação de recursos. Por exemplo, se o fator limitante for o custo, vale mais à pena utilizar apenas 1 nó. Já se o tempo for um fator mais relevante, a utilização de 8 nós se torna mais adequada. O único caso que não vale à pena é utilizar 2 nós, pois, neste caso, apesar de o tempo reduzir consideravelmente em relação à utilização de apenas 1 nó, o custo do mestre é muito representativo, tornando as comparações demasiadamente caras. Vale ressaltar que, apesar de o custo aumentar à medida que se utiliza mais nós, esta diferença não é muito expressiva em relação ao tempo de execução, que reduz consideravelmente, numa proporção de quase 50%.

Por fim, a previsão do tempo de execução utilizando a equação 6.2 se mostrou boa na nuvem *AWS* com o *ParallelCluster*, principalmente na comparação de sequências maiores, nas quais o erro de previsão foi consideravelmente baixo. Nos demais casos, o tempo estimado consegue dar uma boa noção ao usuário, apesar de apresentar um erro maior.

O projeto de "cloudificação"do *MASA-CUDAlign* demorou meses para ser concretizado, uma vez que para alcançar tal objetivo foi necessário: testar diversas configurações de *cluster*; solicitar e esperar o aumento dos limites de *VCPUs* da *AWS*; tratar com problemas de restrição de acesso; configurar e modificar os *scripts* para submissão das execuções; dentre outros. Caso outro usuário da *AWS* desejasse executar o *MASA-CUDAlign* no *ParallelCluster*, com as mesmas especificações, este processo seria bem mais rápido, uma vez que os *scripts* já estão devidamente configurados e já foi definido um passo a passo para instalar e executar o *MASA-CUDAlign* no *ParallelCluster*. Nesse contexto, estima-se que seria necessário cerca de três horas para configuração do *cluster* e uma hora para iniciar a execução das sequências.

7.2 Trabalhos Futuros

Esta monografia utilizou a nuvem da *Amazon* para "cloudificar" a ferramenta *MASA-CUDAlign*. Um dos possíveis trabalhos futuros consiste em implementar o *cluster*, na nuvem de outras empresas, como a *Azure*, a *Google Cloud Platform* e a *IBM Cloud* e então analisar o comportamento do *MASA-CUDAlign* nesses ambientes. Além disso, as instâncias utilizadas no *cluster* deste trabalho possuíam apenas uma GPU e, por esse motivo, pretende-se futuramente testar a execução desta ferramenta em instâncias com múltiplas GPUs.

Nesta monografia utilizou-se a Equação 5.1 para prever o tempo de execução de cada comparação, em apenas uma GPU, porém, como o objetivo do trabalho foi construir um *cluster*, pretende-se futuramente expandir essa equação, de maneira que ela possa ser utilizada para múltiplas GPUs.

Outra proposta de trabalho futuro aborda a técnica de *Block Pruning*. A ideia consiste em comparar um par de sequências biológicas em um algoritmo heurístico, e armazenar o valor do escore calculado, que será sempre menor ou igual ao do escore ótimo. Em seguida, executar a mesma comparação, dessa vez na ferramenta *MASA-CUDAlign*, passando para ela o escore obtido pelo algoritmo heurístico, de maneira que este possa ser utilizado como referência para a poda de blocos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] PORTNOY, M. *Virtualization essentials*. [S.l.]: John Wiley & Sons, 2012.

[2] TALKS, A. O. T. *Your First HPC Cluster on AWS*. Disponível em: <https://www.youtube.com/watch?v=UnQM7cX2y0E&ab$_c$*hannel* = *AWSOnlineTechTalks*>.*Acessoem* : 26$_/$08$_/$2021.

[3] AMAZON WEB SERVICE. *Exemplo: compartilhamento de sub-redes públicas e privadas*. Disponível em: <https://docs.aws.amazon.com/pt$_b$*r*/*vpc*/*latest*/*userguide*/*example* − *vpc* − *share.html*>.*Acessoem* : 21$_/$09$_/$2021.

[4] SANDES, E. F. d. O. Algoritmos paralelos exatos e otimizações para alinhamento de sequências biológicas longas em plataformas de alto desempenho. 2015.

[5] JÚNIOR, M. A. C. d. F. Comparação paralela de sequências biológicas em múltiplas gpus com descarte de blocos e estratégias de distribuição de carga. 2021.

[6] LUSCOMBE, N. M.; GREENBAUM, D.; GERSTEIN, M. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, Schattauer GmbH, v. 40, n. 04, p. 346–358, 2001.

[7] DURBIN, R. et al. *Biological sequence analysis*. [S.l.]: Cambridge university press Cambridge, 1998.

[8] SANDES, E. F. D. O.; BOUKERCHE, A.; MELO, A. C. M. A. D. Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification. Association for Computing Machinery, New York, NY, USA, v. 48, p. 63:1–63:36, 2016.

[9] KORPAR, M.; ŠIKIĆ, M. Sw#–gpu-enabled exact alignments on genome scale. *Bioinformatics*, Oxford University Press, v. 29, n. 19, p. 2494–2495, 2013.

[10] LIU, Y.; WIRAWAN, A.; SCHMIDT, B. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC bioinformatics*, Springer, v. 14, n. 1, p. 1–10, 2013.

[11] BADGER, M. L. et al. *Cloud computing synopsis and recommendations*. [S.l.]: National Institute of Standards & Technology, 2012.

[12] YOUNGE, A. J. et al. Efficient resource management for cloud computing environments. In: IEEE. *International conference on green computing*. [S.l.], 2010. p. 357–364.

[13] SADASHIV, N.; KUMAR, S. D. Cluster, grid and cloud computing: A detailed comparison. In: IEEE. *2011 6th International Conference on Computer Science & Education (ICCSE)*. [S.l.], 2011. p. 477–482.

[14] FOSTER, I. What is the grid? a three point checklist. *GRID today*, v. 1, p. 32–36, 01 2002.

[15] FOSTER, I. et al. Cloud computing and grid computing 360-degree compared. In: IEEE. *2008 grid computing environments workshop*. [S.l.], 2008. p. 1–10.

[16] XING, Y.; ZHAN, Y. Virtualization and cloud computing. In: *Future Wireless Networks and Information Systems*. [S.l.]: Springer, 2012. p. 305–312.

[17] CHIUEH, S. N. T.-c.; BROOK, S. A survey on virtualization technologies. *Rpe Report*, v. 142, 2005.

[18] CAFARO, M.; ALOISIO, G. Grids, clouds, and virtualization. In: _____. [S.l.: s.n.], 2010. p. 1–21. ISBN 978-0-85729-048-9.

[19] JAIN, N.; CHOUDHARY, S. Overview of virtualization in cloud computing. In: IEEE. *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. [S.l.], 2016. p. 1–4.

[20] DAYLAMI, N. The origin and construct of cloud computing. *International Journal of the Academic Business World*, v. 9, n. 2, p. 39–45, 2015.

[21] BUYYA, R.; BROBERG, J.; GOSCINSKI, A. M. *Cloud computing: Principles and paradigms*. [S.l.]: John Wiley & Sons, 2010.

[22] MARQUES, R. *O que é Amazon AWS?* Disponível em: <https://www.homehost.com.br/blog/tutoriais/o-que-e-amazon-aws/>. Acesso em: 26/08/2021.

[23] AMAZON WEB SERVICE. *Perguntas frequentes sobre o Amazon EC2*. Disponível em: <https://aws.amazon.com/pt/ec2/faqs/>. Acesso em: 26/08/2021.

[24] AMAZON WEB SERVICE. *AWS ParallelCluster User Guide*. Disponível em: <https://docs.aws.amazon.com/parallelcluster/latest/ug/aws-parallelcluster-ug.pdf>. Acesso em: 26/08/2021.

[25] AMAZON WEB SERVICE. *O que é o Amazon S3?* Disponível em: <https://docs.aws.amazon.com/pt_br/AmazonS3/latest/userguide/Welcome.html>. Acesso em: 26/08/2021.

[26] NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, Elsevier, v. 48, n. 3, p. 443–453, 1970.

[27] SMITH, T. F.; WATERMAN, M. S. et al. Identification of common molecular subsequences. *Journal of molecular biology*, Elsevier Science, v. 147, n. 1, p. 195–197, 1981.

[28] GOTOH, O. An improved algorithm for matching biological sequences. *Journal of molecular biology*, Elsevier, v. 162, n. 3, p. 705–708, 1982.

[29] HASARD. *Bioinformatics: Gotoh Algorithm*. apr 2016. Disponível em: <http://anythingtutorials.blogspot.com/2016/04/bioinformatics-gotoh-algorithm.html>.

[30] HIRSCHBERG, D. S. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, ACM New York, NY, USA, v. 18, n. 6, p. 341–343, 1975.

[31] MYERS, E. W.; MILLER, W. Optimal alignments in linear space. *Bioinformatics*, Oxford University Press, v. 4, n. 1, p. 11–17, 1988.

[32] BRUM, R. C. et al. A fault tolerant and deadline constrained sequence alignment application on cloud-based spot gpu instances. In: SPRINGER. *European Conference on Parallel Processing*. [S.l.], 2021. p. 317–333.

[33] AMAZON WEB SERVICE. *AWS ParallelCluster User Guide*. Disponível em: <https://docs.aws.amazon.com/parallelcluster/latest/ug/aws-parallelcluster-ug.pdf>. Acesso em: 13/09/2021.

[34] AMAZON WEB SERVICE. *Configuration - AWS ParallelCluster User Guide*. Disponível em: <https://docs.aws.amazon.com/parallelcluster/latest/ug/configuration.html>. Acesso em: 22/09/2021.

[35] FIGUEIREDO, M. et al. Parallel fine-grained comparison of long dna sequences in homogeneous and heterogeneous gpu platforms with pruning. *IEEE Transactions on Parallel and Distributed Systems*, v. 32, n. 12, p. 3053–3065, 2021.

[36] NATIONAL CENTER FOR BIOTECHNOLOGY INFORMATION (NCBI). *NCBI Main Page*. Disponível em: <https://www.ncbi.nlm.nih.gov/>. Acesso em: 29/09/2021.

[37] TECH POWER UP. *NVIDIA Tesla T4*. Disponível em: <https://www.techpowerup.com/gpu-specs/tesla-t4.c3316>. Acesso em: 22/09/2021.

ANEXOS

I. SCRIPTS SBATCH E SCRIPTSLURM

Repositório do *github* contendo os *scripts sbatch* e *scriptslurm*: https://github.com/Filipemaia01/Scripts-de-Automatizacao-do-Static-MultiBP