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

Comparação Paralela de Sequências Biológicas em Múltiplas GPUs com Mecanismo de Tolerância a Falhas

Filipe Maia Soares

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Alba C. M. A. de Melo

Brasília
2025

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

Comparação Paralela de Sequências Biológicas em Múltiplas GPUs com Mecanismo de Tolerância a Falhas

Filipe Maia Soares

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Alba C. M. A. de Melo (Orientador)
CIC/UnB

Prof. Philippe O. A. Navaux
UFRGS

Prof. Aletéia P. F. de Araújo
UnB

Prof. Rodrigo B. de Almeida
Coordenador do Programa de Pós-graduação em Informática

Brasília, 13 de Março de 2025

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

Agradecimentos

O mestrado é um trabalho que envolve a participação direta ou indireta de várias pessoas. Gostaria de deixar registrado meu agradecimento a todos que de alguma forma contribuíram para a realização deste projeto.

O primeiro agradecimento não poderia deixar de ser à professora Dra. Alba Melo pela orientação, ensinamentos e parceria contínua. Sua atenção, paciência e dedicação são capazes de extrair o maior potencial dos alunos e produzir o incentivo necessário para vencer os desafios. Muito obrigado pela oportunidade de interagir e colaborar com a equipe de professores do Barcelona Supercomputing Center, aos quais também sou muito grato pela colaboração e auxílio no desenvolvimento deste trabalho. Agradeço também ao projeto CNPq/AWS 421828/2022-6, que possibilitou acesso, configuração e uso do AWS ParallelCluster.

Também agradeço ao Dr. Marco Figueirêdo, que foi extremamente solícito ao sanar dúvidas e a me ensinar mais sobre o MASA-CUDAlign. Ao Dr. Walisson Sousa, agradeço pela nossa amizade, conversas e por todo o auxílio dado desde o desenvolvimento do meu TCC até hoje.

Aos meus pais e irmão sou muito grato por me acompanharem nesta jornada. O suporte de vocês me permitiu passar por cima dos momentos difíceis e foi fundamental para conseguir concluir este trabalho. Minha namorada Rafaela foi uma parceira e tanto, me dando todo o apoio, trazendo incentivo e reflexões. Você é parte desta conquista.

Agradeço à minha amiga Mariana pelos conselhos e por me encorajar nos momentos de insegurança. Aos meus amigos Gabriel e Emilia, muito obrigado por me convencerem a vencer os desafios da vida.

A todos os meus amigos, obrigado por estarem presentes em minha vida. A convivência com vocês traz tranquilidade em meio às angústias. Este trabalho não seria possível sem o apoio de todos vocês.

Resumo

A comparação de sequências biológicas é uma das principais operações na área de Bioinformática. Os algoritmos exatos utilizados para compará-las aliado ao tamanho das sequências pode acarretar em horas ou até mesmo dias de execução. Neste contexto, é fundamental que tais aplicações estejam amparadas com mecanismos de tolerância a falhas, uma vez que a ocorrência de uma falha pode demandar a reexecução completa da aplicação, gerando prejuízos em termos de tempo e de custo. O *MASA-CUDAlign* é uma ferramenta de comparação de sequências paralela e exata, que implementa tolerância a falhas em apenas uma GPU. O objetivo da presente Dissertação de Mestrado consiste em propor e avaliar um mecanismo de tolerância a falhas para o *MASA-CUDAlign*, utilizando múltiplas GPUs, o chamado *FT-CUDAlign* (*Fault Tolerant CUDAlign*). O *FT-CUDAlign* é um mecanismo leve de tolerância a falhas, pois utiliza uma funcionalidade já existente na ferramenta *MASA-CUDAlign*, que é o armazenamento de algumas colunas em disco. Tais colunas são definidas no *FT-CUDAlign* como *checkpoints* síncronos. Além disso, foi proposto um protocolo de detecção e recuperação de falhas, usando estes *checkpoints*. A implementação do *FT-CUDAlign* foi feita em C e C++ e utiliza *sockets* para estabelecer a comunicação entre os processos. Os experimentos foram realizados em dois ambientes paralelos contendo de três a oito máquinas, com uma GPU em cada. Os resultados mostraram que o *overhead* adicionado pelo *FT-CUDAlign*, em execuções sem falhas, permaneceu abaixo de 11% para os experimentos realizados na nuvem da *AWS* e abaixo de 1% para os mesmos experimentos realizados no LAICO. Este *overhead* decresce à medida que se aumenta o tamanho das sequências. Os resultados também mostraram que o *FT-CUDAlign* se mostrou eficaz ao tratar cenários de falhas simples e múltiplas, sendo capaz de concluir a computação utilizando apenas as máquinas remanescentes.

Palavras-chave: Tolerância a falhas, Comparação de sequências biológicas, Checkpointing, GPU, MASA-CUDAlign

Abstract

Biological sequence comparison is one of the main operations in the field of Bioinformatics. The exact algorithms used for this comparison, combined with the size of the sequences, can result in hours or even days of execution. In this context, it is essential that such applications are equipped with fault tolerance mechanisms, as the occurrence of a failure may require a complete re-execution of the application, resulting in time and cost losses. *MASA-CUDAlign* is a parallel and exact sequence comparison tool that implements fault tolerance in a single GPU. The objective of this Master's Dissertation is to propose and evaluate a fault tolerance mechanism for *MASA-CUDAlign* using multiple GPUs, referred to as *FT-CUDAlign* (Fault Tolerant CUDAlign). *FT-CUDAlign* is a lightweight fault tolerance mechanism, as it leverages an existing feature of *MASA-CUDAlign*, which is the storage of certain columns on disk. These columns are defined in *FT-CUDAlign* as synchronous checkpoints. Additionally, a fault detection and recovery protocol was proposed, using these checkpoints. The implementation of *FT-CUDAlign* was carried out in C and C++ and uses sockets to establish communication between processes. The experiments were conducted in two parallel environments, consisting of three to eight machines, each equipped with a GPU. The results showed that the overhead added by *FT-CUDAlign*, in failure-free executions, remained below 11% for experiments conducted in the *AWS* cloud and below 1% for the same experiments performed in LAICO. This overhead decreases as the sequence size increases. The results also demonstrated that *FT-CUDAlign* was effective in handling both single and multiple failure scenarios, successfully completing computations using only the remaining machines.

Keywords: Fault tolerance, Biological sequence comparison, Checkpointing, GPU, MASA-CUDAlign

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ABFT Algortihm-Based Fault Tolerance.

AMT Assynchronous Many-Tasks.

API Application Programming Interface.

AWS Amazon Web Service.

BP Block Pruning.

C/R Checkpoint/Restart.

CUDA Compute Unified Device Architecture.

DMTCP Distributed MultiThreaded Checkpointing.

DNA Deoxyribonucleic Acid.

FPGA Field Programmable Gate Array.

FT-CUDAlign Fault-Tolerant CUDAlign.

GPU Graphics Processing Unit.

HPC High Performance Computing.

IoT Internet of Things.

IST Incremental Speculative Traceback.

LAICO LAboratório de sistemas Integrados e COncorrentes.

MASA Multi-platform Architecture for Sequence Aligners.

MBP Million Base Pairs.

MPI Message Passing Interface.

MTBF Mean Time Between Failures.

MTTF Mean Time To Failure.

MTTR Mean Time To Repair.

NCBI National Center for Biotechnology Information.

RAxML Randomized Axelerated Maximum Likelihood.

RNA Ribonucleic Acid.

SDC Silent Data Corruption.

SDK Software Development Kit.

SEDAR Soft Errors Detection and Automatic Recovery.

SSH Secure Shell Protocol.

ULFM User-Level Fault Mitigation.

VM Virtual Machine.

Capítulo 1

Introdução

A Bioinformática é uma área de pesquisa que utiliza de técnicas computacionais para auxiliar no estudo de dados biológicos [7]. Uma de suas principais atuações reside na comparação par a par de sequências biológicas, na qual selecionam-se duas sequências de Deoxyribonucleic Acid (DNA), Ribonucleic Acid (RNA) ou proteínas de interesse e obtém-se, como resultado da comparação, o escore e o respectivo alinhamento. Estes resultados são relevantes, pois permitem avaliar a similaridade e inferir propriedades acerca das sequências [8]. Existem dois tipos de algoritmos utilizados para a comparação de sequências: os exatos, que sempre produzem o melhor resultado, porém que possuem complexidade quadrática de tempo; e os heurísticos, que não garantem o melhor resultado, mas possuem, em média, tempo de computação bem menor [8]. O foco desta Dissertação de mestrado reside nos algoritmos exatos, que utilizam programação dinâmica e possuem complexidade quadrática de tempo de execução.

As sequências biológicas de DNA podem ser muito extensas, na ordem de milhões de pares de base (*Million Base Pairs (MBP)*) e, por esta razão, podem demorar horas ou até mesmo dias para serem comparadas. Com o intuito de reduzir o tempo de execução, têm-se optado por paralelizar a aplicação, subdividindo uma tarefa grande em várias partes menores, que são distribuídas entre unidades de processamento, como é o caso dos: *multicores*, Field Programmable Gate Array (FPGA) e Graphics Processing Unit (GPU) [9]. Dentre essas opções, as GPUs têm se destacado devido à sua facilidade de programação, popularização e alto desempenho.

Como os algoritmos exatos de comparação de sequências ainda podem demorar dias para serem executados, é fundamental estar preparado para o pior dos cenários: a ocorrência da falha. Isto porque a falha pode comprometer o andamento da aplicação, exigindo que sua execução seja reiniciada por completo e assim trazendo prejuízo nos quesitos temporal, financeiro e de recursos computacionais dedicados à execução da aplicação. Apesar de importante, a maioria das ferramentas da literatura ainda não implementa mecanis-

mos de tolerância a falhas, como é o caso da ferramenta de Ino e coautores [10] e do *CUDASW++* [11]. A ferramenta *MASA-CUDAlign* [4] [5] [12] (Multi-platform Architecture for Sequence Aligners (MASA), Compute Unified Device Architecture (CUDA)), por sua vez, possui um mecanismo de tolerância a falhas, o qual se restringe à utilização de apenas uma GPU.

Neste contexto, o objetivo desta dissertação consiste em propor, implementar e avaliar um mecanismo de tolerância a falhas para a ferramenta *MASA-CUDAlign*, utilizando múltiplas GPUs. O objetivo principal foi segmentado nos seguintes objetivos secundários: projetar o mecanismo de detecção de falhas; buscar a reconexão entre os módulos; detectar o último *checkpoint* válido; e reiniciar a computação a partir dele.

O mecanismo proposto, chamado Fault-Tolerant CUDAlign (FT-CUDAlign) é a principal contribuição desta dissertação. É um mecanismo leve *application-specific* de *checkpoint/recovery* que utiliza colunas salvas pelo *MASA-CUDAlign* como *checkpoints*, sem introduzir *overhead* para tanto. Além de utilizar as colunas salvas como *checkpoints*, o FT-CUDAlign implementa um protocolo de detecção de falhas aproveitando-se da troca de mensagens entre os nodos durante a execução do *MASA-CUDAlign*. Ao ser detectada a falha, o FT-CUDAlign executa um protocolo de recuperação, re-executando a aplicação a partir do *checkpoint* mais recente. O FT-CUDAlign foi implementado em C e C++, utilizando *sockets* para estabelecer comunicação entre os processos. Os resultados experimentais na nuvem *AWS*, usando a ferramenta *ParallelCluster* [13], mostram que o *overhead* percentual das execuções sem falha paremenceu abaixo de 11%, no pior dos casos, e decresce de acordo com o aumento do tamanho das sequências. Outra conclusão observada é que o momento da falha impacta significativamente no *overhead*, pois quanto mais próximo do final da iteração ocorrer a falha, maior será o *overhead* de reexecução. Por fim, o *FT-CUDAlign* se mostrou eficaz ao tratar cenários de falhas simples e múltiplas, sendo capaz de finalizar a computação com os nodos sobreviventes. Evidentemente que, quanto mais falhas, maior é o *overhead* induzido na aplicação.

O Capítulo 2 introduz o tópico comparação de sequências biológicas e explica os algoritmos utilizados para compará-las. O Capítulo 3 aborda as principais técnicas de tolerância a falhas, em especial o *checkpointing* que é a técnica empregada neste trabalho. O Capítulo 4 detalha o funcionamento da ferramenta de Comparação de Sequências Biológicas *MASA-CUDAlign*, algoritmo no qual este trabalho baseia-se. O Capítulo 5, por sua vez, comenta sobre os trabalhos desenvolvidos na área de tolerância a falhas e apresenta uma tabela que compara as soluções adotadas neles, em relação ao trabalho desenvolvido nesta pesquisa. No Capítulo 6 é explicado o projeto da versão tolerante a falhas multi GPU do *MASA-CUDAlign*, o *FT-CUDAlign*. O Capítulo 7 detalha os experimentos, os ambientes de execução e avalia os resultados em termos de *overhead* e de efeito da falha

sob diferentes circunstâncias. Por fim, o Capítulo 8 conclui a dissertação, relembrando o que foi feito, avaliando os resultados e trazendo propostas de trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

A comparação de sequências biológicas é uma das áreas de estudo da Bioinformática, ramo da Biologia que utiliza de técnicas computacionais para auxiliar no estudo de dados biológicos [7]. Nesta operação, são selecionadas duas sequências de DNA, RNA ou proteínas, que serão comparadas, produzindo como resultado o escore e o alinhamento entre elas. Na natureza novas sequências biológicas são provenientes de modificações em sequências já existentes, seja por meio de inserções, deleções ou substituições de nucleotídeos (pares de bases) ou aminoácidos e, portanto, a comparação de sequências permite a inferência de propriedades de uma sequência conhecida na sequência de estudo [8]. Nesta proposta, serão tratadas sequências de DNA.

2.1 Escore e Alinhamento

A obtenção de um alinhamento se dá listando as sequências e pareando suas bases por colunas. Há três tipos de alinhamento: global, local e semi-global. No alinhamento global, as sequências são comparadas em sua totalidade e, portanto, nenhuma base é descartada. Já o alinhamento local tem o intuito de evidenciar a região mais similar entre as duas sequências e, por isso, ele delimita apenas um fragmento das sequências. Por fim, o alinhamento semi-global envolve as sequências quase que em sua totalidade, podendo excluir apenas os prefixos ou sufixos de cada, isto é, porção de bases que inicia ou que encerra a sequência. Os alinhamentos global, local e semi-global estão ilustrados, respectivamente, nas Figuras 2.1, 2.2 e 2.3, com pontuações $+1$ (*match*), -1 (*mismatch*) e -2 (*gap*).

O escore de duas sequências é obtido analisando as pontuações da comparação de seus elementos. Por exemplo, se as bases forem idênticas, ocorre um *match* e atribui-se uma

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

Figura 2.1: Alinhamento Global.

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

Figura 2.2: Alinhamento Local.

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

Figura 2.3: Alinhamento Semi-Global.

pontuação positiva a esta comparação. Por sua vez, se os elementos forem diferentes tem-se um *mismatch* e atribui-se uma pontuação negativa. Por fim, pode-se inserir um *gap* em uma das sequências. Neste caso, não realiza-se a comparação entre os elementos, porém ainda assim atribui-se uma pontuação negativa, que depende do modelo de penalização adotado: *linear* ou *affine*. No modelo *linear*, todos os *gaps* possuem a mesma pontuação, independentemente de sua disposição. Já no modelo *affine*, se um *gap* for inserido consecutivamente a outro, este é chamado de *gap* de extensão ($gap_{extension}$) e pontuará de uma forma. Caso contrário, ele é chamado de *gap* de abertura (gap_{open}), pois dá início a uma sequência de *gaps* e pontua de maneira diferente. Em seguida, soma-se as pontuações de cada comparação e obtém-se um valor de escore. Os escores das Figuras 2.1, 2.2 e 2.3 foram, respectivamente: -6, $+4$ e $+2$.

As pontuações de *match*, *mismatch* e *gap* estabelecidas no parágrafo anterior foram arbitrárias, escolhidas apenas com o intuito de exemplificar o procedimento de cálculo do escore. A obtenção de valores mais adequados para cada um desses cenários pode ser obtida por meio de estudos estatísticos que buscam obter alinhamentos mais semelhantes àqueles observados na natureza [8].

O objetivo dos *gaps* é levar em consideração as inserções e deleções de bases nas sequências, que ocorrem na natureza. Entretanto, a adição de *gaps* provoca defasagens

entre as bases das sequências que dificultam, consideravelmente, o procedimento de comparação, tornando-o inviável de ser feito manualmente em sequências mais extensas. Por esta razão, foram implementados os algoritmos de comparação de sequências exatos, que serão explicados a seguir.

2.2 Algoritmos de Comparação de Sequências

2.2.1 Needleman-Wunsch (NW)

O algoritmo de *Needleman-Wunsch* realiza uma comparação global entre as sequências e obtém o escore ótimo, bem como o respectivo alinhamento que resulta nele. O modelo de penalidade adotado é do tipo *linear* e, portanto, todos os *gaps* pontuam o mesmo tanto. A sua execução é dividida em duas fases: na primeira calcula-se a matriz de programação dinâmica, e na segunda recupera-se o alinhamento [14].

Cálculo da Matriz

Considere duas sequências S_0 e S_1 de tamanhos m e n, que formam uma matriz A, de coordenadas (i, j), sendo i o índice da linha e j o da coluna. Assim, as linhas da matriz representarão os elementos da sequência S_0, enquanto as colunas representarão S_1. A primeira linha e coluna, entretanto, são constituídas por valores de inicialização: $A_{0,j} = -G \cdot j$ e $A_{i,0} = -G \cdot i$, dando à matriz as dimensões $(m+1)$x$(n+1)$. O elemento $A_{0,0}$, por sua vez, recebe o valor nulo.

A matriz de similaridade é então calculada segundo a Equação de Recorrência 2.1, que seleciona o valor máximo entre a comparação dos elementos $S_0[i]$ e $S_1[j]$, a qual pode resultar em um *match* ou *mismatch*, ou a inserção de *gap* em uma das sequências.

$$A_{i,j} = max \begin{cases} A_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ A_{i,j-1} + G \\ A_{i-1,j} + G \end{cases} \tag{2.1}$$

Note que cada elemento (i, j) depende apenas das coordenadas imediatamente anteriores a ele: $(i-1, j)$, $(i, j-1)$ e $(i-1, j-1)$, conforme mostra a Figura 2.4.

Recuperação do Alinhamento (*Traceback*)

O objetivo desta fase é recuperar o alinhamento ótimo (*traceback*). Durante a fase de cálculo da matriz associa-se, a cada célula, um ponteiro responsável por indicar a qual dos três predecessores o elemento atual é proveniente. O alinhamento é então recuperado seguindo os ponteiros, a partir do último elemento da matriz, $H_{m,n}$, em sentido inverso

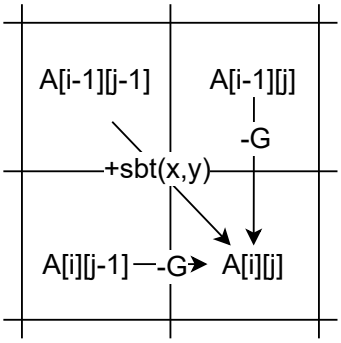


Figura 2.4: Cálculo do escore de um elemento.

ao de processamento, até atingir o início da matriz, $H_{0,0}$. Vale ressaltar que um elemento pode ser simultaneamente proveniente de mais de um predecessor, propiciando que sejam obtidos múltiplos alinhamentos ótimos. As Figuras 2.5 e 2.6 apresentam, respectivamente, a matriz e o alinhamento ótimo, obtidos da utilização do algoritmo de *Needleman-Wunsch*, em um cenário hipotético.

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 2.5: Algoritmo de *Needleman-Wunsch*.

S_0 :	A	A	T	G	G	C	G	T	-	G	C	T	A	C
S_1 :	A	A	-	A	T	T	G	T	A	G	C	G	-	A

Figura 2.6: Alinhamento (*Needleman-Wunsch*).

Análise de Complexidade

Se por um lado o algoritmo de *Needleman-Wunsch* é vantajoso devido à sua simplicidade, por outro lado exige o processamento completo da matriz de similaridade, que possui dimensões $(m+1)x(n+1)$. Se as sequências tiverem tamanhos próximos, é possível dizer que a complexidade, tanto espacial, quanto temporal do algoritmo é quadrática, ou seja, da ordem $O(n^2)$ [14].

2.2.2 Smith Waterman (SW)

O algoritmo de *Smith Waterman* objetiva a obtenção do alinhamento local ótimo [15]. A sua metodologia é muito similar ao algoritmo de *Needleman-Wunsch*, porém traz três diferenças. A primeira consiste na adição do valor 0 à equação de recorrência, resultando na Equação 2.2, que garante a inexistência de valores negativos na matriz.

$$A_{i,j} = max \begin{cases} A_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ A_{i,j-1} + G \\ A_{i-1,j} + G \\ 0 \end{cases} \tag{2.2}$$

A segunda modificação consiste em alterar os valores de inicialização da primeira linha e coluna da matriz para 0. A última modificação está na forma de recuperação do alinhamento, que inicia-se do ponto de escore ótimo, em sentido inverso ao processamento da matriz, até atingir um ponto de valor nulo. As Figuras 2.7 e 2.8 apresentam, respectivamente, a matriz de programação dinâmica e o alinhamento local ótimo obtido utilizando o algoritmo de *Smith Waterman* nas mesmas sequências da Seção 2.2.1. Vale ressaltar que a complexidade deste algoritmo também é de ordem $O(n^2)$.

2.2.3 Gotoh

Diferentemente das soluções mencionadas até então, que operam segundo o modelo de penalização *linear gap*, o algoritmo de *Gotoh* adota o modelo *affine gap*, na qual os *gaps* pontuam diferentemente de acordo com o seu tipo: abertura ou extensão [16]. A Equação 2.3 calcula a pontuação dos *gaps* de acordo com seu tipo:

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 2.7: Algoritmo de *Smith Waterman*.

S_0 :	*	*	*	G	G	C	G	*	*	*	*	*	*
S_1 :	*	*	*	A	G	C	G	*	*	*	*	*	*

Figura 2.8: Alinhamento (*Smith Waterman*).

$$\lambda(k) = G_{open} + (k-1) \cdot G_{ext} \tag{2.3}$$

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

O cálculo da matriz é realizado por meio de três matrizes: H, E e F. A matriz E representa a inserção de *gaps* na sequência S_0 e armazena o escore que cada elemento da matriz teria caso fosse adicionado um *gap* naquela posição. A matriz F opera de maneira análoga, porém representa a inserção de *gaps* na sequência S_1. Por fim, a matriz H contém o valor máximo entre a comparação dos elementos de S_0 e S_1 e a inserção de *gaps* em uma das sequências.

A matriz H inicializa sua primeira linha e coluna segundo o modelo de penalização de *gaps* da Equação 2.3, de maneira que $H_{i,0} = \lambda(i)$ e $H_{0,j} = \lambda(j)$. A matriz E, por sua vez, inicializa sua primeira linha com $E_{0,i} = -\infty$, e a primeira coluna não é levada em consideração e, portanto, $E_{j,0} = X$. Por fim, a matriz F inicializa a primeira coluna

com $F_{j,0} = -\infty$ e a primeira linha é descartada $F_{i,0} = X$. Em todas as matrizes o primeiro elemento recebe o valor nulo. Após a inicialização é realizado o processamento das matrizes H, E e F, segundo as Equações de recorrência 2.4, 2.5 e 2.6.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ E_{i,j} \\ F_{i,j} \end{cases} \tag{2.4}$$

$$E_{i,j} = max \begin{cases} H_{i,j-1} + G_{open} \\ E_{i,j-1} + G_{ext} \end{cases} \tag{2.5}$$

$$F_{i,j} = max \begin{cases} H_{i-1,j} + G_{open} \\ F_{i-1,j} + G_{ext} \end{cases} \tag{2.6}$$

A recuperação do alinhamento no algoritmo de *Gotoh* segue as mesmas regras de *Needleman-Wunsch*, diferenciando-se apenas pelo fato de ser necessário identificar tanto as coordenadas do elemento anterior, quanto a sua matriz de origem.

A complexidade temporal e espacial de *Gotoh* continua tendo a mesma ordem de grandeza $O(mn)$ que os algoritmos de *Needleman-Wunsch* e *Smith Waterman*, porém como são processadas 3 matrizes, demora o triplo do tempo e consome o triplo do espaço.

2.2.4 Myers-Miller

A complexidade quadrática de armazenamento, utilizada nos algoritmos descritos nas Seções 2.2.1, 2.2.2 e 2.2.3, dificulta consideravelmente a comparação de sequências mais extensas. Neste contexto, o algoritmo de *Myers-Miller* [17] surge com o intuito de reduzir este custo computacional. A inspiração para este algoritmo é proveniente dos estudos de *Hischberg*, que elaborou um algoritmo para solucionar o problema da *Longest Common Subsequence* [18]. Tal ideia foi adaptada ao algoritmo de *Gotoh*, reduzindo sua complexidade espacial para a ordem linear $O(m + n)$.

Diferentemente das abordagens anteriores, o algoritmo de *Myers-Miller* não guarda a matriz como um todo. Ao invés disso, o cálculo da matriz ocorre através de apenas duas colunas que a processam em sentido direto, até atingir a coluna central, $\frac{m}{2}$. Simultaneamente, outras duas colunas processam a matriz, em sentido inverso, até atingir a coluna central. Em seguida, somam-se os elementos de cada uma dessas colunas. As coordenadas do elemento de maior escore desta soma identificam o *crosspoint*, ou seja, o elemento da coluna que cruza o alinhamento ótimo.

A matriz é então dividida na coluna central e na linha do *crosspoint*, resultando em quatro partes, conforme ilustra a Figura 2.9. Como o alinhamento sempre se inicia no

ponto $(0, 0)$ e segue em direção ao final da matriz, não é possível que ele retorne para cima ou para a esquerda. Esta condição garante que as regiões em cinza não farão parte do alinhamento ótimo e que possam ser descartadas, eliminando metade da área de processamento, que agora se concentra apenas nas regiões em verde. Este mesmo procedimento de divisão e exclusão é realizado iterativamente até que o alinhamento seja obtido por completo.

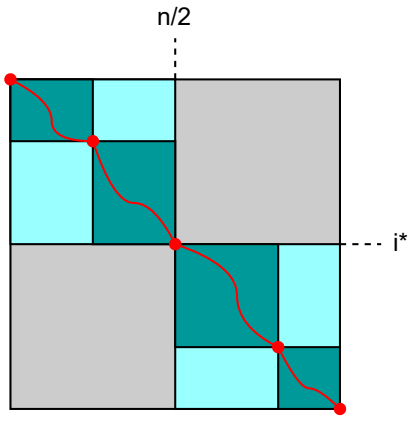


Figura 2.9: Algoritmo de *Myers e Miller*.

Assuma que a área de processamento a cada iteração do algoritmo segue uma progressão geométrica de razão $q = \frac{1}{2}$. Para verificar a área total de processamento, após todas as iterações, é necessário calcular a soma de infinitos termos de uma progressão geométrica, que é dada por: $S = \frac{a_1}{1-q}$, sendo a_1 a área processada na primeira iteração que, neste caso, é a área da matriz A_M. Como o número de iterações não é infinito, tem-se que a área total processada deve ser menor do que a soma dos inifitos termos da progressão, resultando na Equação 2.7.

$$\begin{aligned} A_T &< S \\ A_T &< \frac{a_1}{1-q} \\ A_T &< \frac{A_M}{1-\frac{1}{2}} \\ A_T &< 2 \cdot A_M \end{aligned} \tag{2.7}$$

Como a área total é menor do que duas vezes a área da matriz, tem-se que o tempo de processamento continua sendo da ordem $O(mn)$. Entretanto, como o cálculo da matriz utiliza apenas quatro colunas, a complexidade espacial passa a ser linear, isto é, $O(m+n)$.

Capítulo 3

Tolerância a Falhas

A Computação de Alto Desempenho (*High Performance Computing (HPC)*) é uma área da computação que permite a resolução de problemas complexos que exigem alto poder de processamento e, muito frequentemente, dias de computação. Para que a resolução desses problemas seja realizável, em tempo hábil, os sistemas de HPC se utilizam de vários elementos de computação para obter a paralelização da aplicação, possibilitando atingir altas taxas de processamento, que atualmente podem chegar à escala de um quatrilhão de operações de ponto flutuante por segundo (10^{15} operações), chamada peta-escala, e mais recentemente atingiram a exa-escala (1 quintilhão de operações de ponto flutuante por segundo - 10^{18} operações) [19].

Ainda que alguns sistemas de HPC estejam atingindo tais taxas de processamento, vários dos problemas a serem processados ainda podem demorar horas, dias, ou até mesmo semanas para finalizarem sua execução. Além disso, a utilização de mais elementos de computação nesses sistemas também traz um ponto negativo, que é o aumento da probabilidade de falhas. Nesse contexto, para garantir a finalização da execução, com sucesso, deseja-se que o sistema seja tolerante a falhas.

3.1 Definições

A falha, a falta e o erro são anomalias de software que, apesar de serem utilizados coloquialmente com o mesmo significado, possuem diferentes definições. O **erro** se refere a alguma ação que produz um resultado incorreto. A **falta**, por sua vez, é a manifestação do erro e, portanto, ainda que o erro exista, é possível que a falta nunca ocorra. Por fim, a **falha** é um evento na qual o sistema ou um de seus componentes não se comporta de acordo com os limites estabelecidos [20]. As falhas podem ser classificadas em três tipos [21]:

- **Transitórias:** duram certo período de tempo e não são recorrentes;

- **Intermitentes:** periodicamente surgem e desaparecem;
- **Permanentes:** após surgirem, continuam ativas até que sejam corrigidas.

Assim sendo, tolerância a falhas refere-se à capacidade de um sistema de continuar operante, mesmo diante de uma ou mais falhas [19]. A implementação de técnicas de tolerância a falhas traz inevitavelmente *overhead* à aplicação e, portanto, a escolha apropriada da técnica é fundamental para o projeto, tanto para que a solução empregada seja eficaz, quanto para maximizar o custo-benefício da técnica.

3.2 Métricas

A disponibilidade avalia a probabilidade de o sistema estar operante em um determinado instante de tempo. Obter alta disponibilidade é algo que se almeja em todos os sistemas suscetíveis à falha e, para tanto, é necessário implementar eficientes técnicas de tolerância a falhas. Mais especificamente, para que um sistema seja altamente disponível, deseja-se que o tempo de recuperação seja o mais breve possível e o intervalo entre as falhas o maior possível, pois ainda que elas ocorram em grande quantidade, o sistema ainda será capaz de rapidamente retornar à atividade. Existem diversas métricas para avaliação da disponibilidade, as quais são apresentadas na Figura 3.1, e descritas a seguir [21]:

- ***Mean Time To Failure (MTTF)***: tempo médio que o sistema, estando ativo, demora para falhar.
- ***Mean Time To Repair (MTTR)***: tempo médio para que ocorra a reparação do sistema, pós falha.
- ***Mean Time Between Failures (MTBF)***: tempo médio entre faltas. O intervalo se inicia com a ocorrência de uma falta e conta o período de recuperação, passando pela atividade do sistema, até a ocorrência da falha seguinte.

3.3 Técnicas de Tolerância a Falhas

As técnicas de tolerância a falhas são divididas em dois tipos: as proativas, que buscam prever a ocorrência da falha e assim prevenir a sua ocorrência; e as reativas, que objetivam mitigar ao máximo os danos causados pela falha. A Figura 3.2 apresenta algumas das principais técnicas de tolerância a falhas. O foco desta dissertação é nas técnicas reativas, mais especificamente, o *checkpointing* (destacado em vermelho).

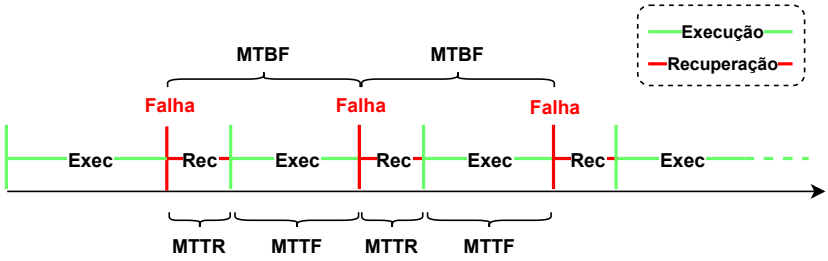


Figura 3.1: Métricas de avaliação da disponibilidade de um sistema, adaptado de [1].

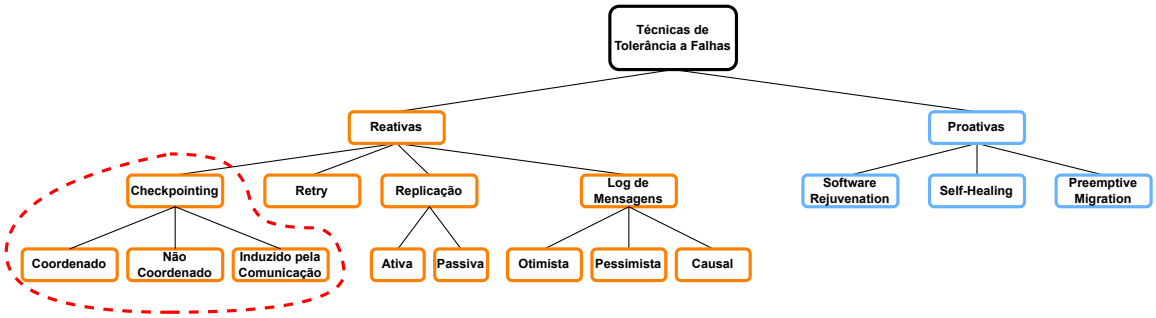


Figura 3.2: Técnicas de tolerância a falhas agrupadas pelas abordagens proativa e reativa.

3.3.1 Proativas

As abordagens proativas têm por objetivo monitorar a execução da aplicação para tentar prever cenários de falha e evitar que ocorram. Para tanto, durante a fase de monitoramento são executados algoritmos que constantemente calculam a probabilidade de ocorrência da falha. As principais técnicas proativas são: *Software Rejuvenation*, *Self-Healing* e Migração Preemptiva [22].

Software Rejuvenation

O *Software Rejuvenation* é uma técnica criada para operar em aplicações que demoram longos períodos de tempo (dias ou semanas) para serem executadas e que, por consequência, sofrem com o envelhecimento de seus processos. O vazamento de memória, o não *release* de *locks*, o vazamento de descritores e o corrompimento de dados são algumas das causas do envelhecimento de processos, que, ao longo do tempo, vão degrandado a perfomance da aplicação, podendo levá-la à falha [23].

Para tentar evitar essa situação, a técnica de *Software Rejuvenation* periodicamente e preventivamente reinicia a execução da aplicação, enfileirando as mensagens de entrada, descarregando estruturas de dados da memória, recriando processos a partir de um estado salvo via *checkpoint*, dentre outras ações [23]. Durante este período, a aplicação fica

indisponível, gerando atrasos na sua execução. É importante ressaltar que, diferentemente de um cenário de falha, a renovação de processos é uma ação planejada e portanto o sistema pode se preparar para a sua ocorrência, de maneira a não desencadear erros na aplicação, bem como menores atrasos na execução.

Self-Healing

Self-Healing é uma técnica proposta para que o sistema automaticamente detecte a falha, realize um diagnóstico do que está causando a falha e, então, atue adequadamente para reparar o problema e recuperar a execução da aplicação [24].

Migração Preemptiva

O método da migração tem por objetivo prever a ocorrência de uma falha e então, preventivamente, transferir parte da aplicação, do nó defeituoso para outro nó seguro, que dará continuidade à execução. Para que esta técnica funcione adequadamente, é necessário prever, com precisão, o local, o momento e o tipo de falha que ocorrerá. A transferência da execução pode ser realizada em nível de processo ou de máquina virtual. Além disso, costuma-se também transferir arquivos *logs*, pois eles permitem reduzir o tempo de inatividade, bem como recuperar mais rapidamente o sistema da falta [19].

Vale ressaltar que nem sempre as técnicas proativas realizarão previsões corretas e, portanto, é possível tanto prever erroneamente a ocorrência de uma falta, quanto não prever a falta e ela ainda assim ocorrer. Este problema é agravado em grandes sistemas *HPC*, como aqueles de peta ou exa escala, uma vez que nestes é necessário realizar uma grande quantidade de previsões. Por este motivo, as abordagens proativas são frequentemente utilizadas em conjunto com técnicas reativas, como as de recuperação, pois ainda que a técnica proativa falhe, ainda haverá como recuperar o sistema.

3.3.2 Reativas

As técnicas reativas têm por objetivo reduzir os efeitos de uma falha após a sua ocorrência. As técnicas que serão abordadas nesta seção são: *retry*, replicação ativa e passiva, *checkpointing* e *log* de mensagens.

Retry

Esta é a técnica mais simples utilizada atualmente. Ela simplesmente reexecuta requisições que tenham falhado, utilizando o mesmo recurso computacional que anteriormente [22].

Replicação

A técnica da replicação consiste em criar várias cópias de cada tarefa e executá-las em diferentes nós computacionais, com o objetivo de evitar que a falha de um nó impeça a finalização da execução de alguma das tarefas [25]. A replicação pode ser feita de duas formas: ativa e passiva.

Na replicação passiva, o usuário apenas interage com uma das cópias, chamada de primária. A réplica primária então processa e responde a requisição do usuário. Além disso, ela também se comunica com as demais réplicas com o propósito de atualizar seus estados [26].

Já na replicação ativa, o usuário envia uma requisição a todas as réplicas, que vão processar e retornar a resposta da requisição ao usuário, que esperará apenas pela primeira resposta. Vale ressaltar que, para utilizar a replicação ativa, os servidores devem ser determinísticos [26].

Checkpointing

Os *checkpoints* são como imagens do estado de cada processo do sistema, em um determinado momento [27]. Após a ocorrência de uma falha, é possível utilizá-los para retornar a tal estado de execução, evitando assim a necessidade de reiniciar a execução da aplicação do início. A técnica que restaura a execução a partir de um *checkpoint* é chamada de recuperação por retorno.

Em um sistema paralelo, o *checkpoint* do sistema consiste no conjunto de *checkpoints* de todos os processos ativos. Há diversas maneiras de realizar o *checkpoint*, porém as três formas primárias são: o *checkpoint* não coordenado, o *checkpoint* coordenado e o *checkpoint* induzido pela comunicação [27]. A depender do tipo de *checkpoint* utilizado, a recuperação por retorno pode levar a aplicação a um estado consistente ou inconsistente em relação ao envio e recepção de mensagens.

Estado Consistente e Inconsistente

O estado inconsistente pode ocorrer após a recuperação por retorno, caso haja inconsistências em relação ao envio e recepção de uma mensagem. Por exemplo, no estado de um processo B consta que uma mensagem foi recebida pelo processo A. Porém, no estado de A não consta que a mensagem foi enviada. Este estado é dito inconsistente, pois não é possível uma mensagem ser recebida sem antes ter sido enviada.

Note que, no caso contrário, ou seja, se no estado do processo A consta o envio de uma mensagem com destino ao processo B, e no estado de B não consta a sua recepção, o estado é considerado consistente, pois a mensagem pode ainda estar trafegando na rede.

O conjunto mais recente e consistente de *checkpoints* é chamado de linha de recuperação [2]. A Figura 3.3 apresenta exemplos que ilustram os estados consistente e inconsistente, bem como a linha de recuperação.

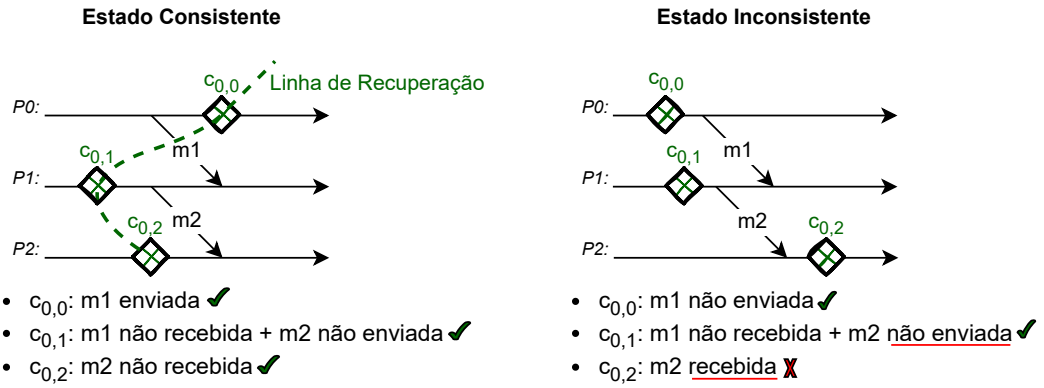


Figura 3.3: Exemplo de estado consistente e inconsistente, adaptado de [2].

É importante destacar que o estado inconsistente não pode existir no sistema e, portanto, ao atingir um estado inconsistente, a aplicação tentará recuperar a execução de um conjunto de *checkpoints* anterior.

Checkpoint Não Coordenado

No *checkpoint* não coordenado, cada processo realiza o respectivo *checkpoint* no momento que julgar mais oportuno. Esta abordagem é vantajosa, pois os processos possuem maior autonomia, uma vez que não necessitam interromper a execução de importantes regiões de código para realizar o *checkpoint*. Além disso, nesta abordagem não é necessária a sincronização dos processos, evitando *overheads* inerentes a esta ação. Em contrapartida, esta abordagem apresenta uma série de desvantagens, tais como: *a)* Possibilidade de ocorrência do efeito dominó, condição que causa o retorno em cascata da computação, podendo levá-la a retornar ao ponto inicial; *b)* Realização de *checkpoints* inúteis, isto é, aqueles que não geram um estado global consistente e, portanto, não permitem o avanço da linha de recuperação; *c)* Necessidade de armazenamento de grande quantidade de *checkpoints*. Como os *checkpoints* são tirados de maneira assíncrona, não há como garantir quais irão gerar um estado consistente e, portanto, faz-se necessário armazenar inúmeros. Como consequência, é necessário selecionar quais devem ser mantidos e quais devem ser eliminados. Este trabalho é realizado pelo coletor de lixo, que na abordagem de *checkpoint* não coordenado exige um projeto mais complexo e precisa ser acionado com maior frequência [2].

O efeito dominó ocorre após a fase de recuperação individual dos processos, quando o sistema retorna para um estado inconsistente, estado no qual uma mensagem consta como recebida por um processo, mas não consta como enviada pelo outro [2]. Esta situação pode ocorrer devido à não sincronização entre os processos no momento de realização do *checkpoint*, como é o caso na abordagem não coordenada. Buscando trazer o sistema a um estado consistente, o processo receptor é retornado para um *checkpoint* anterior ao mais recente. Este retorno pode desencadear sucessivos estados inconsistentes nos demais processos, que em resposta, continuarão retornando para *checkpoints* anteriores, e assim retrocedendo cada vez mais a execução, podendo levá-la de volta ao estado inicial.

Checkpoint Coordenado

O *checkpoint* coordenado com recuperação por retorno é a técnica de tolerância a falhas mais popular a nível de aplicação [28]. Na metodologia de *checkpoints* coordenados ocorre a sincronização de processos, isto é, os processos e seus canais de comunicação são bloqueados para que seja possível, simultaneamente, realizar o *checkpoint* de todos os processos, garantindo a gravação de um estado consistente [27]. Uma das principais vantagens dessa abordagem está na simplicidade de implementação.

Além disso, o fato de cada *checkpoint* pertencer a um estado global consistente torna esta metodologia imune ao efeito dominó, e também simplifica a implementação do coletor de lixo, que apenas necessita guardar o último *checkpoint*, ao invés de analisar quais *checkpoints* manter e quais deletar. A principal desvantagem de *checkpoints* coordenados está na latência gerada pela sincronização dos processos. Isto dificulta a escalabilidade, pois quanto maior for a quantidade de nós e, consequentemente, de processos a sincronizar, maior será a latência.

Checkpoint Induzido pela Comunicação

Por fim, o *checkpoint* induzido pela comunicação combina as estratégias dos dois outros tipos de *checkpoint* (coordenado e não coordenado), com o objetivo de usufruir das vantagens de cada um: evitando a ocorrência do efeito dominó e, ao mesmo tempo, possibilitando que alguns *checkpoints* sejam realizados de maneira independente [2]. Vale ressaltar que, apesar de alguns *checkpoints* serem realizados de maneira independente, outros ocorrem de maneira forçada para garantir alguns estados globalmente estáveis.

Nessa abordagem, a troca de mensagens segue uma metodologia *piggyback*, na qual, após receber uma mensagem do processo transmissor, o processo receptor envia uma mensagem de resposta não apenas com uma confirmação de recebimento, mas incluindo nela também algumas informações de protocolo. Essas informações são então utilizadas pelo processo transmissor para decidir se deve, ou não, ser forçado o *checkpoint*.

As principais desvantagens dessa abordagem são a baixa escalabilidade; o baixo desempenho em momentos de alta comunicação; e a necessidade de realizar muito mais *checkpoints* forçados do que independentes (podendo chegar até o dobro), retirando a flexibilidade da abordagem [27].

Log de Mensagens

A técnica de *log* de mensagens combina a utilização de *checkpoints*, que podem ser tirados de acordo com as técnicas mencionadas anteriormente, com os *logs*: arquivos que armazenam informações acerca de eventos não-determinísticos, tal como a recepção de mensagens [27]. Estas informações armazenadas são chamadas de determinantes.

O objetivo desta abordagem consiste em evitar o reenvio de mensagens durante a fase de recuperação. Por exemplo, na abordagem baseada em *checkpoints*, se um processo falhar logo após receber a mensagem de outro processo, será necessário retornar não apenas o processo que falhou, mas também o transmissor da mensagem para que a mensagem possa ser reenviada. Isso traz um *overhead* na recuperação do sistema. Porém, por meio do retorno baseado em *logs*, a mensagem não necessita ser reenviada, pois todas as informações contidas nela foram previamente armazenadas nos *logs*, tornando necessário apenas simular o recebimento da mensagem, reexecutando-a a partir das informações contidas no *log*, como apresentado na Figura 3.4. Dessa maneira, apenas o processo que falhou necessita ser retornado, fato que reduz o *overhead* na fase de recuperação. Outra vantagem dessa estratégia é que a reexecução, sem a dependência de eventos não-determinísticos, previne a ocorrência do efeito dominó. A desvantagem desta abordagem está na manutenção dos *logs*, característica que aumenta o seu custo computacional.

A recuperação baseada em *log* de mensagens pode ser realizada por meio de três mecanismos: pessimista, otimista e causal. A decisão de qual desses utilizar dependerá dos requisitos do sistema.

Pessimista

Na abordagem pessimista, o sistema assume que a falha pode ocorrer a qualquer momento, inclusive no breve período de tempo entre a recepção de um evento não determinístico e o seu armazenamento no *log*, situação com baixa probabilidade de ocorrência [27]. Por essa razão, após a ocorrência de um evento não determinístico, interrompe-se a execução da aplicação enquanto guardam-se as informações do evento, em um sistema de armazenamento estável. A principal vantagem desta abordagem é a possibilidade de armazenar apenas o último determinante do *log*. Já a principal desvantagem está no grande *overhead* gerado pela paralisação da execução para armazenamento das informações no *log*.

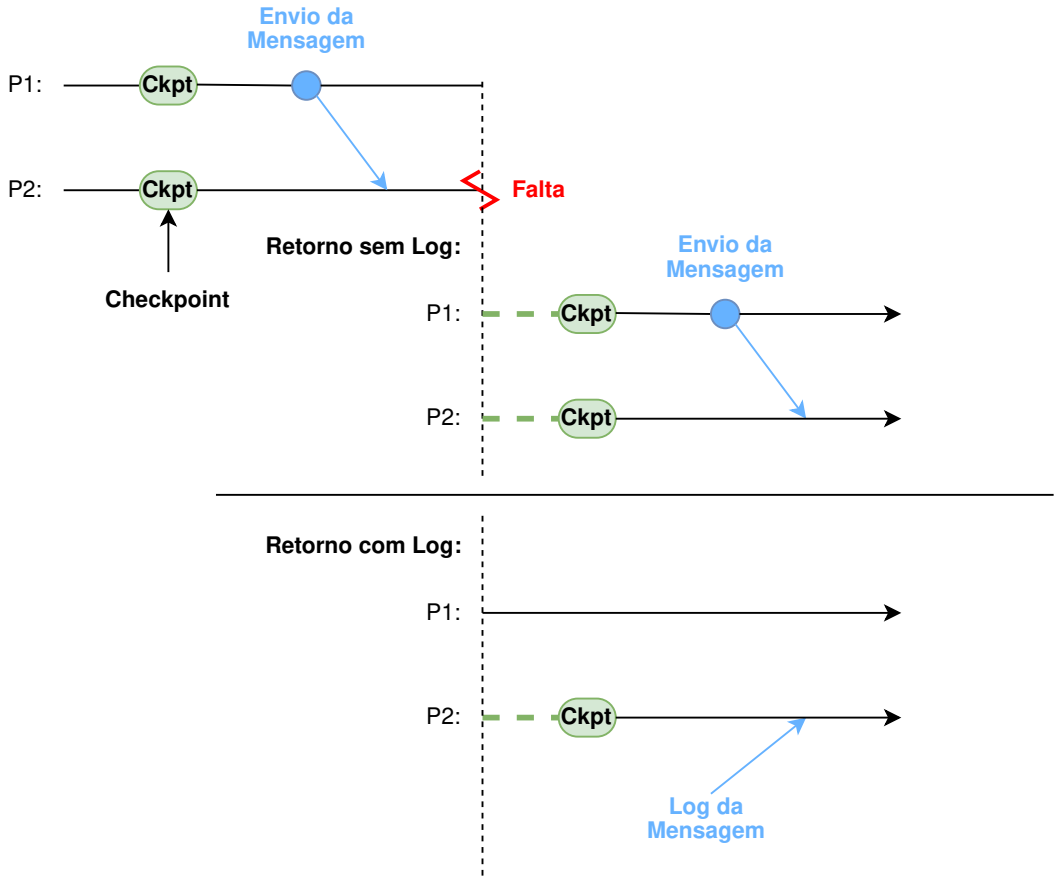


Figura 3.4: Retorno de aplicação com e sem a utilização do *log* de mensagens.

Otimista

A abordagem otimista, como o nome já diz, assume que não ocorrerão falhas entre a chegada de um evento não-determinístico e o seu armazenamento no *log*. Esta metodologia é chamada de assíncrona, pois permite a finalização da execução de um evento não-determinístico antes que o armazenamento de suas informações seja finalizado [27]. Além disso, cada processo possui um *log*, armazenado em dispositivo volátil, e as suas informações não são compartilhadas com demais processos.

Devido a essas decisões, esta abordagem reduz *overheads*, pois não exige a interrupção da execução da aplicação para armazenar o determinante no *log* e, como o armazenamento é realizado em dispositivo volátil, as informações são guardadas e resgatadas mais rapidamente. A principal desvantagem desta abordagem está na maior complexidade exigida para recuperação, que pode ser até impossibilitada caso haja falta de energia, comprometendo dados salvos nos dispositivos voláteis.

Causal

O mecanismo causal busca obter vantagens dos mecanismos otimista e pessimista. Assim como na abordagem otimista, nesta as informações são armazenadas de maneira assíncrona, em um dispositivo de armazenamento volátil. Entretanto, a informação é compartilhada com outros processos através da estratégia de *piggyback*, adicionando o determinante, ao final de uma mensagem que já seria trocada entre tais processos. Isto possibilita que o sistema tenha várias cópias de *logs*, aumentando as chances de a informação estar disponível, caso haja uma ou mais falhas [27]. A abordagem causal possibilita que os processos transfiram informações do *log*, de um dispositivo volátil, para outro estável, de acordo com a necessidade.

As principais vantagens dessa abordagem são a redução do *overhead*, da mesma maneira que na metodologia otimista e a possibilidade de guardar apenas o último determinante registrado no *log*. Em contrapartida, o sistema não suporta falhas em larga escala, caso não tenha sido realizada a transferência para um armazenamento estável. Além disso, as mensagens do sistema passam a ser muito grandes, devido ao *piggyback* dos determinantes. Por fim, o protocolo de recuperação é complexo, pois pode ser necessário obter informações de outros processos para realizar a recuperação de um processo.

Capítulo 4

Ferramenta MASA

O *MASA-CUDAlign* [29] [3] [4] [5] é uma ferramenta de comparação de sequências biológicas exata (como apresentado na Seção 2), isto é, que sempre produz o melhor resultado. Ele recebe como entrada duas sequências de *DNA* de interesse e retorna como saída o valor do escore ótimo, sua respectiva posição na matriz de similaridade e o alinhamento que gera tal resultado. Devido à grande extensão de algumas sequências biológicas, o processo de comparação pode demorar horas e até mesmo dias para ser concluído. Por essa razão, o *MASA-CUDAlign* utiliza em seu processamento um dispositivo de *hardware* robusto, que são as placas gráficas (GPUs).

4.1 MASA-CUDAlign 1.0

O *MASA-CUDAlign 1.0* foi a primeira versão desenvolvida por Sandes [29]. A seguir serão detalhadas algumas de suas mecânicas para dar uma visão geral acerca do funcionamento da ferramenta.

4.1.1 Paralelismo Externo

O paralelismo externo é aquele que ocorre entre os blocos [29]. Cada bloco é um conjunto de elementos da matriz M, agrupados a cada R linhas e C colunas. As dimensões do bloco não são arbitrárias, pois levam em consideração o valor dos parâmetros B, T e α. Os dois primeiros representam a quantidade de blocos e *threads* executados em uma chamada de *kernel*, e o terceiro é a quantidade de linhas que cada *thread* processará. O primeiro passo do paralelismo externo consiste em calcular as dimensões do bloco, com base nos parâmetros descritos e, em seguida, redimensionar a matriz de similaridade para que fique em termos de blocos ao invés de elementos. Em 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.

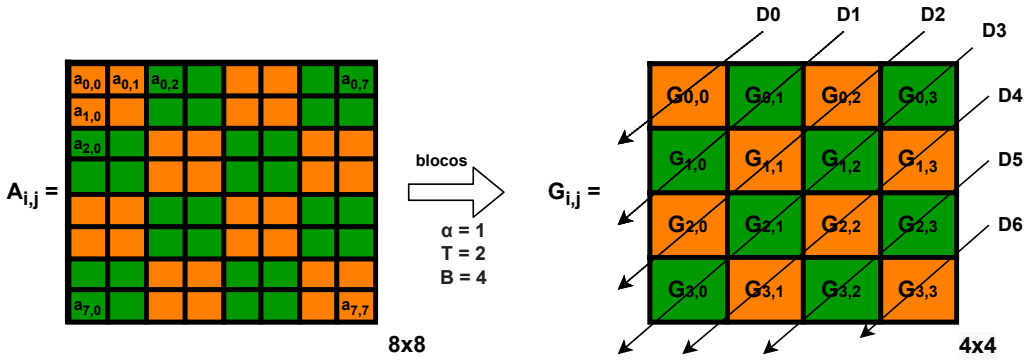


Figura 4.1: Redimensionamento da matriz para blocos e paralelismo externo.

Após o redimensionamento da matriz, os blocos são agrupados em antidiagonais, de acordo com a soma de suas coordenadas, respeitando a dependência de dados existente entre eles, conforme apresenta a seguinte equação: $D_k = \{G_{i,j} | i + j = k\}$. O paralelismo externo é então obtido por meio da técnica do *wavefront*, que se dá através do processamento simultâneo de todos os blocos de uma antidiagonal. Note, a partir da Figura 4.1, que a primeira antidiagonal processa apenas um bloco, porém, à medida que o processamento se aproxima da antidiagonal principal, a quantidade de blocos processados simultaneamente aumenta e, em seguida, volta a reduzir ao se distanciar da antidiagonal principal.

4.1.2 Paralelismo Interno

O paralelismo interno, por sua vez, se dá dentro do bloco. De maneira análoga ao paralelismo externo, no paralelismo interno as células do bloco são agrupadas em antidiagonais, de acordo com o valor da soma de suas coordenadas e com o parâmetro α, conforme a seguinte equação: $d_k = \{(i,j) | \lfloor \frac{i}{\alpha} \rfloor + j = k\}$. A Figura 4.2 apresenta o agrupamento de células em antidiagonais nas cores verde e laranja alternadas. Note que as células dos cantos e das diagonais d_0, d_1 e d_3 foram enumeradas para facilitar a verificação da equação.

O processamento das células do bloco é realizado da esquerda para a direita, utilizando a técnica do *wavefront*, que processa uma antidiagonal por vez. Para garantir que o cálculo de um elemento seja realizado sem dependência de dados pendente, enquanto determinada *thread* T_k processa a coluna j, a *thread* seguinte T_{k+1} estará processando a coluna anterior

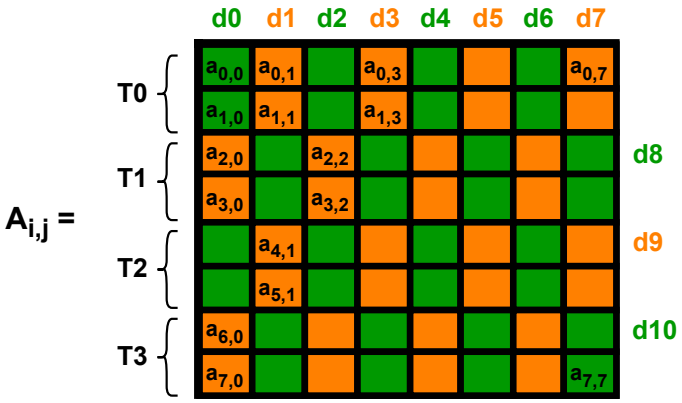


Figura 4.2: Paralelismo interno.

$j-1$ e assim sucessivamente. Ao final de cada diagonal realiza-se uma sincronização entre as *threads*, garantindo que o processamento da próxima antidiagonal não se inicie antes da finalização da atual.

4.1.3 Delegação de Células

As Seções 4.1.1 e 4.1.2 demonstram como o *MASA-CUDAlign* obtém o paralelismo externo e interno ao bloco, porém o grau de paralelismo varia de acordo com a diagonal processada. Isto porque as primeiras diagonais utilizam menos *threads* do que a quantidade total alocada ao bloco, devido à dependência de dados e, portanto, processam menos células simultaneamente. No paralelismo interno, este efeito é observado nas diagonais d_0, d_1 e d_2, que processam, respectivamente, duas, quatro e seis células simultaneamente, conforme é mostrado na Figura 4.2. As diagonais d_3 à d_7, por sua vez, processam oito células cada, pois são capazes de utilizar todas as quatro *threads*. Por fim, as diagonais d_8 à d_{10} processam seis, quatro e duas células, reduzindo novamente o grau de paralelismo.

Observando este efeito, o *MASA-CUDAlign 1.0* optou por utilizar uma abordagem denominada delegação de células. Nesta técnica, apenas as C primeiras diagonais do bloco são processadas, enquanto as demais ficam pendentes para a execução do bloco seguinte. As diagonais iniciais do bloco seguinte são então processadas juntamente com as diagonais pendentes do bloco anterior, formando como se fosse uma diagonal completa e assim maximizando o grau de paralelismo, conforme é mostrado na Figura 4.3.

O contorno azul da Figura 4.4 evidencia que, apesar de o bloco ser logicamente interpretado com um formato quadrado, na prática ele passa a ser processado como se fosse um paralelogramo. A construção deste paralelogramo não se restringe apenas a blocos de

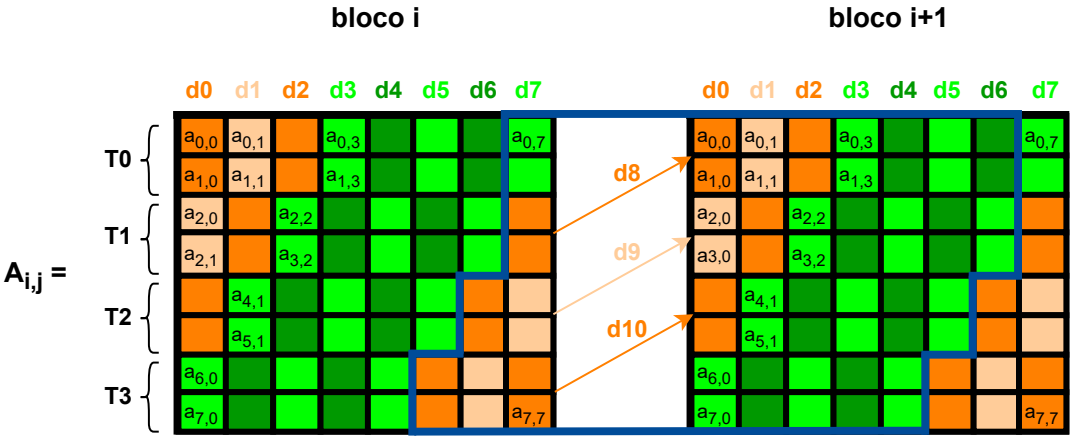


Figura 4.3: Delegação de células.

uma mesma linha, pois a delegação de células também pode ser realizada entre o bloco final de uma linha com o bloco inicial da linha seguinte, como é o caso dos blocos b_1 e b_2, da Figura 4.4.

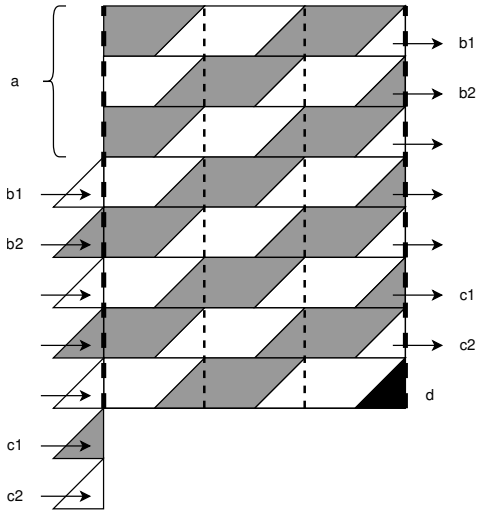


Figura 4.4: Delegação de células entre linhas.

4.1.4 Divisão de Fases

A técnica de delegação de células é extremamente benéfica para o desempenho da aplicação, na medida que aumenta consideravelmente o grau de paralelismo. Entretanto, a postergação do processamento dessas células gera um problema de dependência de dados entre os blocos da diagonal atual com a posterior. Além disso, o CUDA, plataforma

utilizada para executar o *MASA-CUDAlign 1.0*, pode executar os blocos em qualquer ordem e tanto em série quanto em paralelo. A consequência disso é que os blocos que dependem das células cujo processamento foi postergado, podem ser escolhidos para serem processados antes das células pendentes, provocando erros de cálculo na matriz de similaridade.

A Figura 4.5 apresenta três blocos, sendo o bloco 1 pertencente a uma diagonal e os blocos 2 e 3 pertencentes à diagonal seguinte. O bloco 1 será o primeiro a ser processado e delegará o processamento das células em amarelo para o bloco 3, que as processará juntamente com as suas células em azul. Devido à arbitrariedade na escolha dos blocos, o CUDA pode iniciar o processamento pelo bloco 2, cujas células em vermelho são dependentes das células delegadas (em amarelo). Neste cenário, o processamento será feito erroneamente, pois utilizará valores indefinidos para calcular os elementos da matriz de similaridade, preenchendo-a com valores incorretos.

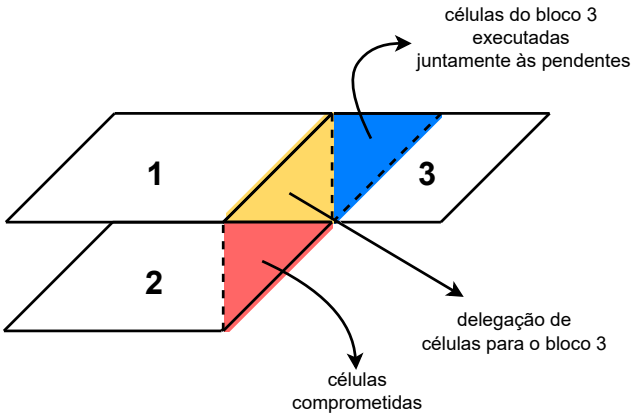


Figura 4.5: Dependência de dados criado pela delegação de células.

A fim de solucionar este problema, o *MASA-CUDAlign 1.0* dividiu o processamento da diagonal em duas fases: longa e curta. Na fase curta são processadas as $T-1$ diagonais, formadas pela agregação das células pendentes, com as diagonais iniciais do bloco seguinte. Esta fase é chamada de curta, pois normalmente esse conjunto de diagonais contém menos células do que no restante do bloco. A fase longa, por sua vez, encerra o processamento de todas as $C-(T-1)$ diagonais de um bloco. A Figura 4.6 evidencia tal separação de fases, na qual as regiões do tipo .1 referem-se à fase curta, enquantos as regiões enumeradas com .2 pertencem à fase longa.

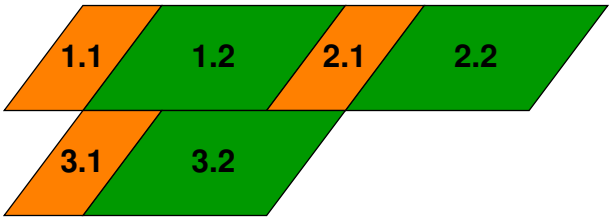


Figura 4.6: Divisão de fases.

4.2 MASA-CUDAlign 2.0 e 2.1

As versões 2.0 e 2.1 do *MASA-CUDAlign* diferenciam-se da versão 1.0, pois além de calcular o elemento de escore ótimo, também obtêm um dos alinhamentos que resultam nele, utilizando memória linear e tempo reduzido. A Figura 4.7 apresenta os seis estágios de execução do *MASA-CUDAlign 2.0*, que serão detalhados nas próximas seções.

Inicialmente, o Estágio 1 do *MASA-CUDAlign 2.0* reutiliza o algoritmo implementado no *MASA-CUDAlign 1.0* para realizar o cálculo da matriz de programação dinâmica. A nova funcionalidade desse estágio consiste no armazenamento, em memória ou em disco, de algumas linhas, chamadas de linhas especiais [3], que são relevantes pois serão utilizadas no Estágio 2 para recuperação do alinhamento.

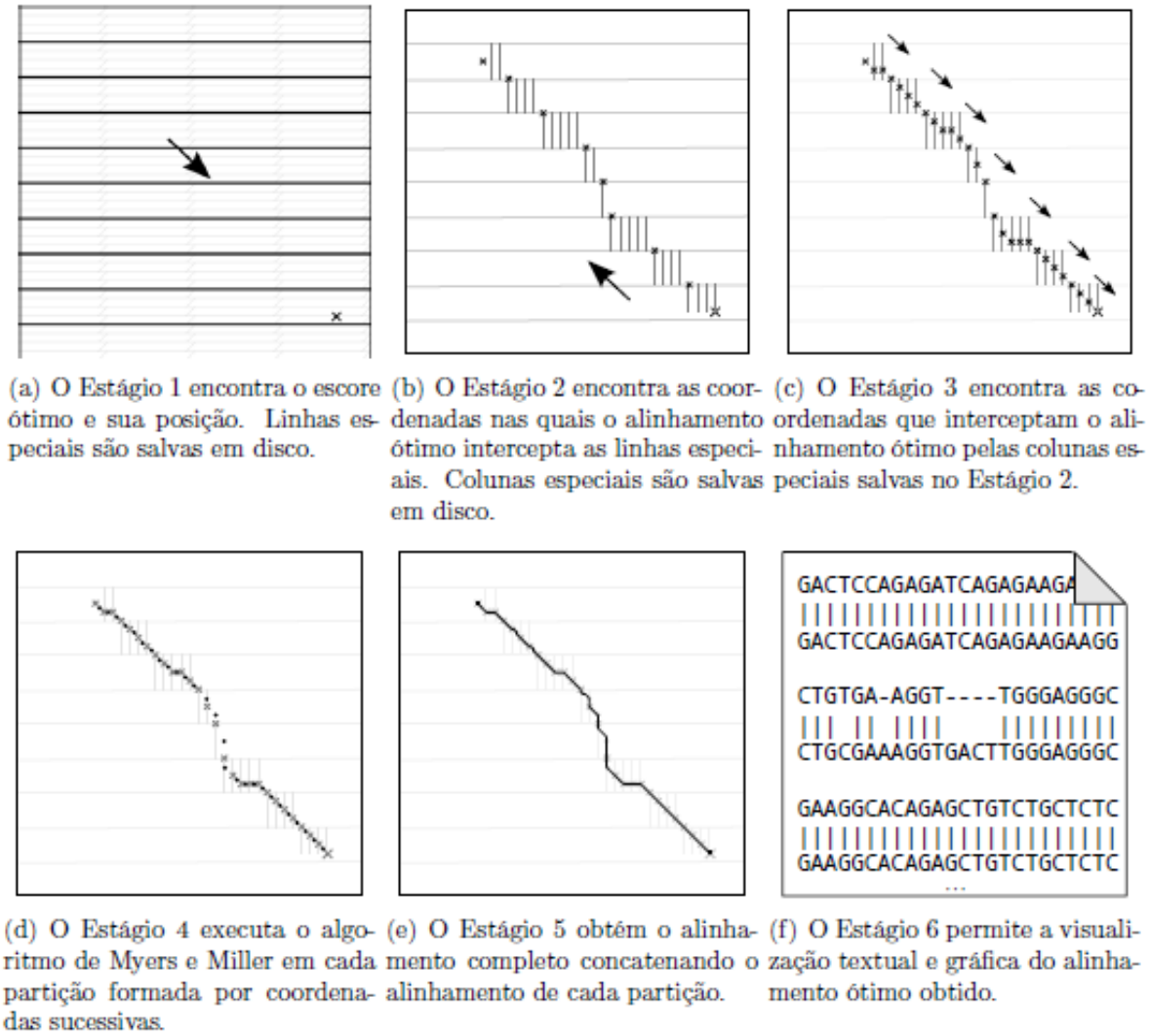
O objetivo do Estágio 2 consiste em recuperar os pontos do alinhamento que cruzam as linhas especiais salvas no Estágio 1, os chamados *crosspoints*. Para tanto, empregam-se duas técnicas: o *matching baseado em objetivo* e a *execução ortogonal*.

Matching Baseado em Objetivo

No algoritmo de *Myers-Miller*, processam-se as linhas de determinada região no sentido direto e inverso. Em seguida, compara-se a soma dos elementos de mesma coordenada, da linha direta com a inversa, e o ponto cuja soma resultar no maior escore é dito pertencente ao alinhamento. O Estágio 2 do *MASA-CUDAlign* opera de maneira similar, porém a obtenção do *traceback* se inicia pelo ponto de escore ótimo, obtido ao final do Estágio 1. A cada novo ponto obtido, atualiza-se o escore alvo, isto é, o valor do ponto que deve ser encontrado na próxima linha especial. A obtenção do escore-alvo permite que o processamento se encerre ao se encontrar o ponto de escore-alvo, ao invés de se processar a linha por completo.

Execução Ortogonal

A técnica da execução ortogonal troca o sentido de processamento da matriz, de horizontal para vertical. Assim, ao invés de processar todas as linhas até atingir a próxima



(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 *MASA-CUDAlign* [3].

linha especial, processam-se apenas as colunas necessárias para obter o ponto de escore-alvo. Isto é vantajoso, pois reduz expressivamente o processamento, uma vez que há muito menos células num fragmento de coluna, restrito por duas linhas especiais, do que em uma linha completa.

Estágios 3 à 6

Retornando aos estágios do *MASA-CUDAlign*, o Estágio 3 objetiva obter mais pontos do alinhamento. A técnica adotada neste estágio é a mesma do Estágio 2, porém com a diferença de que os pontos obtidos no Estágio 2 delimitam regiões compreendidas entre as linhas e colunas de dois *crosspoints* consecutivos. Essa propriedade permite a paralelização

do Estágio 3, uma vez que as regiões são independentes e, portanto, podem ser processadas simultaneamente.

O Estágio 4 adota o algoritmo de *Myers-Miller* (apresentado na Seção 2.2.4) para obter mais pontos do alinhamento, até que cada partição atinja um tamanho menor do que um valor de referência. A cada iteração, obtém-se cerca de o dobro de pontos da iteração anterior, de maneira que a sua duração depende da métrica de tamanho da partição. Da mesma forma que no Estágio 3, neste estágio as partições também são independentes e paralelizáveis.

O Estágio 4 traz uma adaptação ao algoritmo de *Myers-Miller* na qual, ao invés de sempre dividir a partição na linha central, o *MASA-CUDAlign* divide a partição no sentido de maior tamanho: linha ou coluna. Por meio dessa análise, as dimensões da partição permanecem mais próximas, reduzindo a quantidade de divisões necessárias para se atingir o tamanho desejado.

O Estágio 5 utiliza o algoritmo de *Needleman-Wunsch* para completar a recuperação do alinhamento ótimo, obtendo todos os pontos das lacunas de cada região. Para que o Estágio 5 seja mais eficiente, é necessário que o Estágio 4 reduza consideravelmente o tamanho das partições. Além disso, como o tamanho das partições é constante, a complexidade de uso da memória também é, proporcionando assim o uso de memória linear.

O Estágio 6 é o último do *MASA-CUDAlign 2.0*. O seu objetivo é representar, de maneira textual, o alinhamento ótimo para facilitar a análise e estudo mais detalhado do alinhamento.

4.2.1 Block Pruning

A versão 2.1 do *MASA-CUDAlign* introduz a técnica do Block Pruning (BP) [3], ou poda de blocos, que é utilizada no Estágio 1 com o intuito de acelerar o processamento da matriz de programação dinâmica. Tal otimização é alcançada através do descarte de blocos cujo escore é tão baixo que é matematicamente impossível de contribuírem para o alinhamento ótimo. O descarte desses blocos reduz a área de processamento, acelerando a computação.

A identificação de células *prunable*, isto é, células que podem ser descartadas, ocorre por meio da análise de suas coordenadas e de seu escore [3]. Cada célula possui uma distância Δ_i, que é dada pelo número de linhas entre a sua linha e a linha final da matriz. Analogamente, as células também possuem uma distância Δ_j entre a sua coluna e a última da matriz. Se o escore de determinada célula é $H_{(i,j)}$ e se o *match* é o cenário de maior pontuação possível, então o maior escore que tal célula pode atingir, ao final do alinhamento, ocorre no caso de um *match* perfeito entre as substrings S_0, conforme é

mostrado na Equação 4.1 e S_1, segundo a Equação 4.2 e seu valor é dado pela Equação 4.3, sendo *ma* a pontuação de *match*. Verifica-se então o valor do maior escore obtido até a linha i, o $best(i)$. Caso o escore máximo de uma célula seja menor do que o melhor escore encontrado até então, conforme mostra a Equação 4.4, então essa célula é descartável (conforme ilustra a Figura 4.8).

$$S_0[i, .., i + min(\Delta_i, \Delta_j)] \tag{4.1}$$

$$S_1[j, .., j + min(\Delta_i, \Delta_j)] \tag{4.2}$$

$$H_{max(i,j)} = H_{(i,j)} + min(\Delta_i, \Delta_j) \cdot ma \tag{4.3}$$

$$best(i) > H_{max(i,j)} \tag{4.4}$$

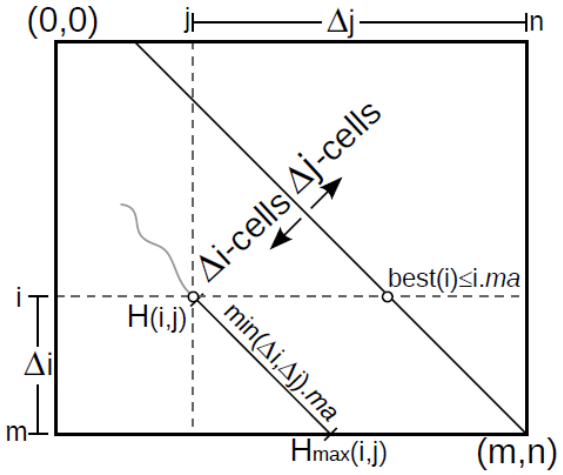


Figura 4.8: Ilustração do procedimento de *Pruning MASA-CUDAlign* [4].

4.3 MASA-CUDAlign 3.0

A utilização das técnicas do *wavefront* e do *Block Pruning* proporcionou uma melhora substancial no desempenho da ferramenta *MASA-CUDAlign*. Entretanto, a utilização de apenas uma GPU em seu processamento ainda limitava muito o potencial da ferramenta, que demandava tempo demasiado (cerca de 8h) para realizar comparações entre sequências mais extensas (acima de $33MB$) [4]. Esta situação motivou o desenvolvimento de uma

nova versão do *MASA-CUDAlign*, que introduz a execução multi-GPU, porém ainda sem *Block Pruning*.

Para tanto, a aplicação divide a matriz em conjuntos de colunas e distribui cada uma dessas partições entre as GPUs disponíveis [4]. Devido à possível heterogeneidade entre as GPUs, o tamanho das partições é estabelecido em função do poder computacional de cada uma delas. O objetivo dessa estratégia é que a distribuição seja balanceada, buscando ao máximo igualar a quantidade de linhas processadas por segundo, em cada GPU. Por esta razão, é necessário distribuir partições maiores às GPUs com maior poder computacional, e as menores às GPUs com menor poder computacional. Ao final do processamento de cada linha da partição, encontram-se as células da coluna de borda, que devem ser continuamente transferidas para a GPU posterior (à exceção da última).

Durante a execução da ferramenta, atribui-se a cada GPU um processo composto por três *threads* [4]: duas *threads* de comunicação, que transferem assincronamente as células entre GPUs, por meio de *sockets* (conforme a Figura 4.9); e a *thread* gerente, que é responsável por administrar a execução do *MASA-CUDAlign* e por gerenciar o fluxo de células entre o *MASA-CUDAlign* e as *threads* de comunicação.

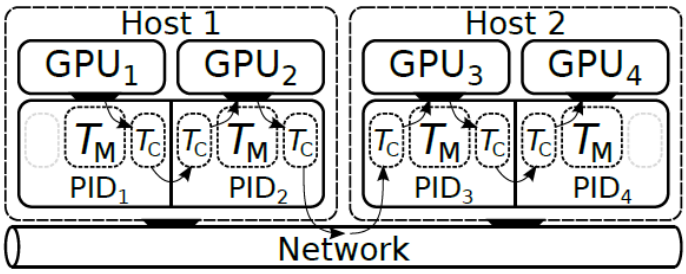


Figura 4.9: Ilustração das *threads* gerente e de comunicação [5].

As *threads* de comunicação podem ser de entrada ou de saída. As *threads* de entrada recebem dados da GPU anterior e entregam-nos à *thread* gerente, que irá utilizá-los para processar a matriz. Já as *threads* de saída recebem as células de borda, da *thread* gerente, e as enviam à GPU posterior. Cada *thread* associa-se a um *buffer* circular, que é responsável por armazenar os dados recebidos ou a serem enviados. A função desses *buffers* é trazer flexibilidade à comunicação, na medida em que desvinculam a comunicação do processamento, suavizando a latência envolvida na comunicação entre as GPUs, bem como em pequenas variações na qualidade da rede.

Os *buffers* são importantes indicadores da qualidade de distribuição de carga. Isto porque o enchimento ou esvaziamento do *buffer* indica que uma GPU está processando dados muito mais rapidamente do que a outra e que, portanto, esta deveria receber uma

partição de maior tamanho. Esta informação é relevante, pois sempre que os *buffers* enchem ou esvaziam, o processamento fica bloqueado, trazendo *overhead* para a aplicação.

4.4 MASA-CUDAlign 4.0

As versões anteriores do *MASA-CUDAlign* tinham o foco de otimizar o tempo de execução do Estágio 1, que é o mais demorado. Entretanto, a versão 4.0 da ferramenta tem por objetivo acelerar o processamento dos Estágios de 2 a 4, responsáveis pela recuperação do alinhamento ótimo. Para tanto, a premissa desta versão consiste em aproveitar o tempo ocioso das GPUs, entre os estágios, especulando resultados que futuramente serão capazes de acelerar a computação, por meio da técnica do *Incremental Speculative Traceback*.

Observou-se, a partir de vários experimentos, que os pontos de escore máximo de colunas intermediárias geralmente coincidem com os *crosspoints*. A técnica do *Incremental Speculative Traceback (IST)* se utiliza desta propriedade para especular *crosspoints*, no tempo em que as GPUs estariam ociosas, com o intuito de adiantar a computação.

Após finalizar o processamento do Estágio 1, determinada GPU_i iniciará o procedimento de especulação, buscando encontrar o próximo *crosspoint*. O procedimento é finalizado quando for identificado o melhor candidato a *crosspoint* ou quando a computação retornar à GPU em questão. Neste momento, a GPU que estiver executando o Estágio 2 verifica se o escore do candidato coincide com o escore-alvo. Caso coincida, tal GPU envia as coordenadas à GPU anterior, que dará continuidade ao processamento do Estágio 2. Por sua vez, se os escores não coincidirem, a GPU_i deverá descartar tal resultado e continuar buscando as coordenadas do próximo *crosspoint*. Tal técnica é apresentada na Figura 4.10.

4.5 Dynamic-MultiBP

A versão mais recente do *MASA-CUDAlign*, o *dynamic-MultiBP*, foi desenvolvida para tratar dois problemas. O primeiro está relacionado à utilização da técnica de *Pruning* que altera a área de processamento da matriz de similaridade, deixando-a com um formato irregular, somente conhecido ao final da execução [6]. Esta característica motivou a implementação da redistribuição dinâmica da carga de trabalho (quantidade de colunas) entre as GPUs. O segundo problema reside no *overhead* proveniente da redistribuição dinâmica, que não é desprezível, uma vez que exige a interrupção completa da aplicação durante o seu cálculo.

Para solucionar tais problemas, a versão *Dynamic-MultiBP* divide a execução em ciclos e, em cada um deles, é processada uma porção da matriz. O parâmetro *breakpoints*

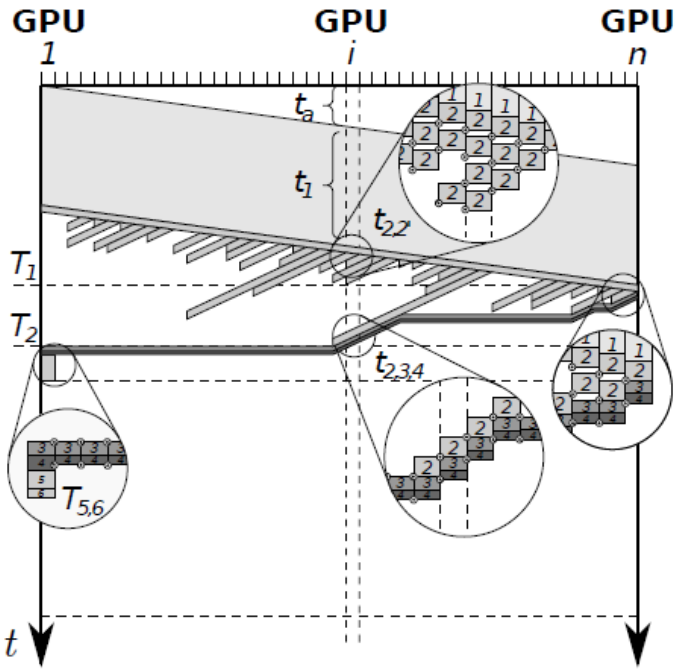


Figura 4.10: Técnica do *Incremental Speculative Traceback* [5].

define a quantidade de vezes que a execução será interrompida para se iniciar um novo ciclo e, portanto, a quantidade total de iterações na qual a execução será dividida sempre será igual à quantidade de *breakpoints* + 1. Na Figura 4.11 a matriz possui apenas um *breakpoint*, que divide a matriz em duas regiões. Ao final de cada ciclo são utilizadas métricas para reavaliar a distribuição de carga e alterá-la, se necessário. No caso apresentado na Figura 4.11, inicialmente as GPUs receberam colunas na proporção 2:2:2 e, posteriormente, isso foi alterado para 2:1:3.

4.5.1 Visão Geral

As soluções *Static-MultiBP* e *Dynamic-MultiBP* utilizam dos mesmos quatro módulos: A) *Executor*: módulo responsável pelo processamento da matriz e onde atua o descarte de blocos; B) *Decision-Maker*: avalia qual é a melhor estratégia a ser utilizada - *Static* ou *Dynamic*, de acordo com parâmetros de entrada; C) *Controller*: módulo que administra a computação da aplicação, enviando comandos para a execução; D) *Monitor*: intermedia a comunicação entre os módulos *Controller* e *Executor*. A diferença entre as versões está no fato de que a *Static* não utiliza a redistribuição dinâmica da carga de trabalho e, portanto, não executa a aplicação em múltiplas iterações, somente em uma única.

A Figura 4.12 apresenta o funcionamento simplificado do projeto de soluções MultiBP, que opera da seguinte maneira: (A) inicialmente o *Controller* lê o arquivo de configura-

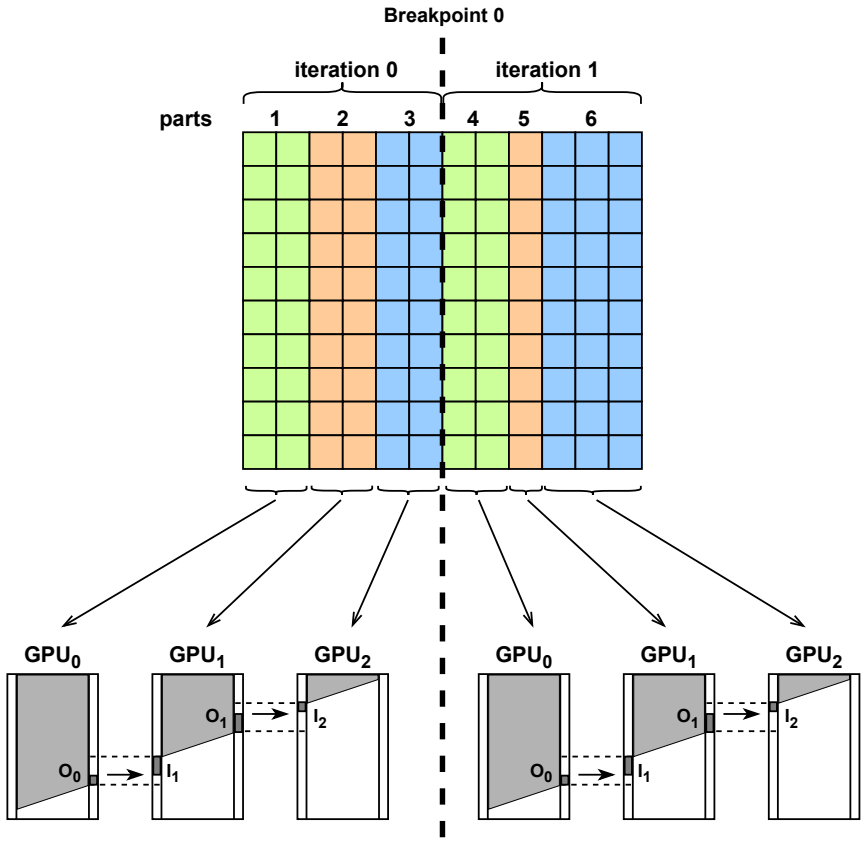


Figura 4.11: Distribuição de colunas entre as GPUs ativas.

ções, que contém informações acerca das GPUs ativas; (B) Se o módulo *Decision* estiver ativo, o *Controller* envia a ele informações acerca das sequências; (C) O *Decision Maker* executa seu algoritmo e retorna a estratégia mais adequada a ser utilizada - *Static* ou *Dynamic*. Caso seja o *Static*, o número de *breakpoints* é definido como 0 e, portanto, a execução é realizada em uma única iteração; (D) O *Controller* então elabora e envia um comando para cada módulo *Monitor*; (E) Tais comandos, que definem os parâmetros de execução, são enviados aos módulos *Monitor*; (G,H) Cada GPU processa uma partição de colunas da matriz, enviando as células da coluna de borda para a GPU posterior e aplicando as técnicas de *Pruning* e de troca do melhor escore para reduzir a área de processamento. (I) O último *Monitor* verifica a existência dos arquivos de controle, que são escritos pelo *MASA-CUDAlign* quando este atinge 80% e 100% da execução. O objetivo desses arquivos é sinalizar ao *Controller* de que ele já pode ler os arquivos de *log* dos *buffers* (no caso do primeiro arquivo), ou que a execução foi finalizada e que uma nova iteração pode ser iniciada (no caso do segundo arquivo); (J) Após a sinalização de que

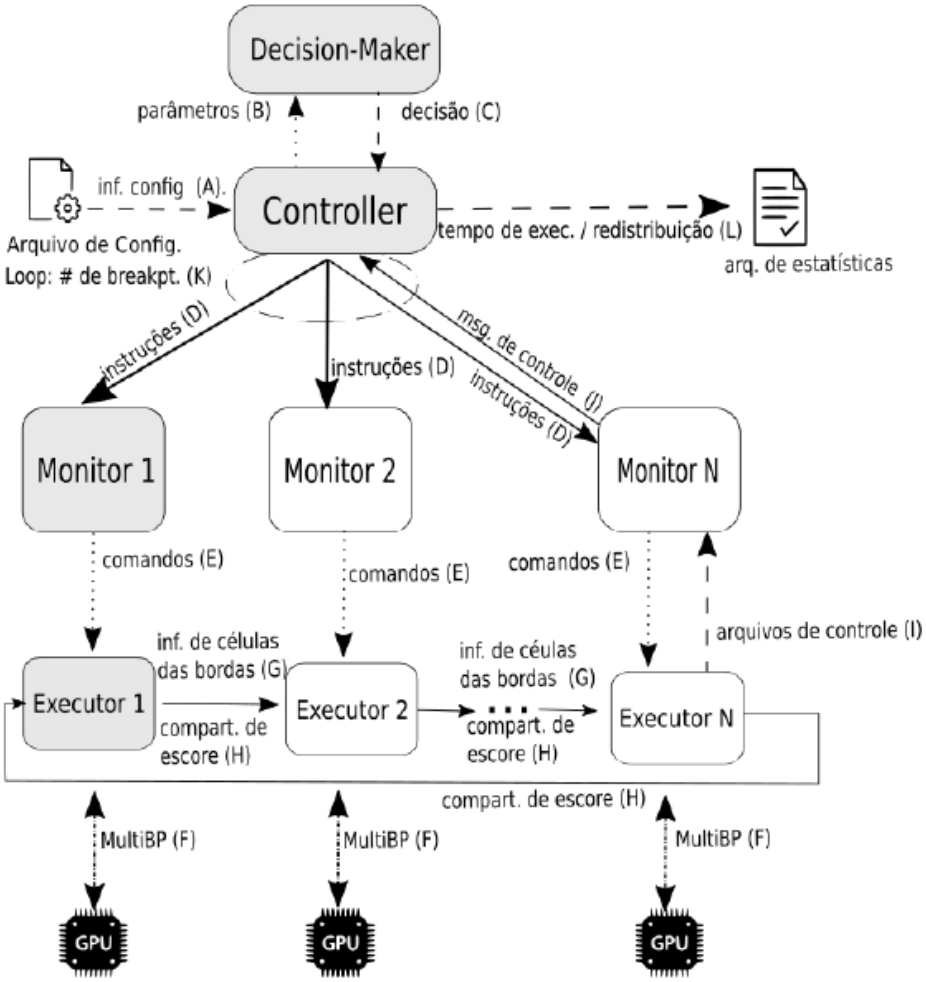


Figura 4.12: Esquemático do projeto Dynamic-MultiBP [6].

o arquivo de término de execução da iteração foi criado, o *Controller* atualiza a iteração corrente e define os novos comandos para a próxima iteração.

4.5.2 Módulo Executor

O módulo *Executor* tem o objetivo de executar o *MASA-CUDAlign*, de acordo com a linha de comando encaminhada pelo *Monitor*. Durante a sua execução, a métrica do melhor escore é trocada entre as GPUs, por meio de *threads* assíncronas, que seguem a estrutura de uma topologia em anel, ou seja, cada GPU envia o melhor escore para a próxima, à exceção da última GPU, que o envia à primeira. O compartilhamento da métrica de melhor escore é fundamental para ampliar a área de poda nas regiões recebidas por cada GPU, melhorando o desempenho da aplicação.

Vale ressaltar que, devido à dependência de dados na matriz, cada *Executor* deve enviar as suas células de borda, referentes à última coluna, para a GPU posterior. No caso da última GPU, ela armazenará toda a última coluna da iteração em um arquivo. Na iteração seguinte, esta coluna será lida pela primeira GPU do grupo para dar prosseguimento ao processamento da matriz.

Se por um lado o descarte de blocos aumenta o desempenho da aplicação, por outro ele desbalanceia a distribuição de carga entre as GPUs, uma vez que a poda proporciona um formato irregular na matriz, que somente é conhecido ao final da execução. Isto faz com que cada GPU tenha que processar mais ou menos células, a depender da área de poda em cada região e, como consequência, os *buffers* ficarão mais frequentemente cheios ou vazios, condição que intensifica o *overhead*. Por esta razão, ao final de cada iteração são analisados os *buffers* de *i/o* para recalcular a distribuição de carga. Para tanto, quando o *Executor* da última GPU termina de processar uma determinada quantidade de diagonais, ele sinaliza ao *Monitor* que os *buffers* já podem ser analisados.

4.5.3 Módulo Controller

O módulo *Controller* tem a função de gerir a execução do *MASA-CUDAlign-MultiBP*. A conexão é estabelecida entre o *Controller* com cada um dos módulos *Monitor* via *socket*. Em seguida, o *Controller* identifica a iteração corrente e elabora os comandos a serem enviados a cada *Monitor* (GPU), que iniciarão os módulos *Executor*, de acordo com o comando recebido.

Após o processamento de determinada quantidade de diagonais, o último *Monitor* enviará uma mensagem informando ao *Controller* que os parâmetros de retenção dos *buffer* já podem ser analisados. Estes parâmetros são armazenados durante a execução, em arquivos de *log* e é a partir deles que será feito o cálculo da redistribuição de carga. No *MASA-CUDAlign*, o parâmetro *split* é que define a quantidade de regiões na qual a matriz será dividida, bem como as suas proporções. Dessa maneira, reajustes de carga são realizados por meio de atualizações no *split*.

Logo após o cálculo dos novos valores de *split*, o *Controller* permanece esperando pela sinalização do último *Monitor* de que a iteração foi concluída. Somente após esta confirmação é que o *Controller* enviará os comandos referentes à iteração seguinte.

4.5.4 Módulo Monitor

O módulo *Monitor* serve como um intermediário, pois recebe o comando do *Controller* e o utiliza para iniciar o *Executor*. Além disso, o último *Monitor* também deve monitorar os arquivos de controle, criados pelo *Executor*. Quando o arquivo de análise dos *buffers*

é criado, o *Monitor* deve sinalizar ao *Controller*, que calculará os novos valores de *split*. Analogamente, assim que uma iteração é finalizada, o último *Executor* cria um arquivo de controle, identificado pelo *Monitor* que envia um sinal ao *Controller* informando que os comandos para a próxima iteração já podem ser enviados.

Capítulo 5

Trabalhos Relacionados

Neste capítulo são apresentados os trabalhos relacionados à área de tolerância a falhas para ambientes paralelos, com foco em aplicações de Bioinformática. Ao final do capítulo, um quadro comparativo contendo as diversas propostas é apresentado e discutido.

5.1 Múltiplas Tarefas Assíncronas (Paul et al., 2019 e 2020)

Em 2019, Paul *et al.* [30] criaram uma Application Programming Interface (API) com o intuito de dar suporte a múltiplas técnicas de tolerância a falhas para aplicações que operam segundo o modelo de múltiplas tarefas assíncronas (Assynchronous Many-Tasks (AMT)). Esta API funciona como extensão da biblioteca Habanero C/C++ (HClib) [31], utilizada para executar aplicações AMT. A limitação da API é que ela apenas opera com aplicações que utilizam um único nodo.

Nas aplicações do tipo AMT, uma grande tarefa é subdividida em várias tarefas menores, que devem receber suas respectivas entradas: dados provenientes de tarefas anteriores. Esta abordagem assume que as tarefas não são bloqueadas, nem esperam por dados de outras tarefas durante a sua execução e portanto, para que uma tarefa seja iniciada suas entradas já devem estar previamente disponíveis.

Para implementação das técnicas de tolerância a falhas, os autores utilizam de duas primitivas implementadas no C++11: *promise* e *future*, que são utilizadas em conjunto para verificar alterações em variáveis comunicadas entre tarefas e assim proporcionar tanto sincronização das tarefas quanto a detecção de falhas.

O trabalho implementa as seguintes técnicas de tolerância a falhas: *Replay*- técnica que reexecuta apenas as tarefas que tenham falhado; *Replication* - técnica na qual a tarefa é replicada em várias unidades de processamento e, ao final da execução, compara-se o valor

de cada uma das tarefas, assumindo que o valor mais frequente é o correto e descartando o resultado das demais instâncias; Algortihm-Based Fault Tolerance (ABFT)- técnica na qual são realizados cálculos matemáticos específicos da aplicação, como checksum, para verificar se o retorno da função está adequado; e Checkpoint/Restart (C/R)- técnica em que são salvos estados do sistema após a checagem de erros de um conjunto de tarefas (Seção 3.3.2). Em caso de falha, reinicia-se a execução a partir do estado do último *checkpoint* válido.

O artigo apresenta uma comparação do tempo de execução de algumas aplicações, como o algoritmo de comparação de sequências biológicas *Smith-Waterman* (Seção 2.2.2), em cenários sem falhas, utilizando diferentes técnicas de tolerância a falhas. Os resultados mostram que a técnica do *replay* introduz pouco *overhead* às aplicações, sempre com *overhead* abaixo de 10%. Já a técnica de replicação apresentou altos *overheads*, uma vez que a quantidade de recursos computacionais é limitada e repartida entre as tarefas principal e sua réplica, trazendo um *overhead* (tempo de execução) próximo a 100% na maioria das aplicações.

Em 2020, Paul *et. al.* [28] publicaram um novo artigo acerca da mesma API, adicionando a funcionalidade de operar em múltiplos nodos. A abordagem assume que as tarefas se comunicam por Message Passing Interface (MPI). A comunicação entre os nodos é estabelecida através da implementação User-Level Fault Mitigation (ULFM) [32], do protocolo de comunicação integrado ao MPI, em conjunto com o framework de tolerância a falhas Fenix [33]. Em [28] são utilizadas as mesmas técnicas de tolerância a falhas de [30], porém o ambiente de execução foi um cluster constituído por nove nodos, cada um contendo dois processadores de 16 *cores*.

Assim como [30], no artigo [28] os autores compararam as aplicações, em um cenário sem falhas, utilizando as versões original e tolerante a falhas das aplicações, porém com diferentes quantidades de nodos: dois, quatro e nove. Em tais experimentos a quantidade de nodos pouco influenciou no tempo de execução sem falhas das aplicações. As técnicas de tolerância a falhas empregadas foram: *replay*, que apresentou *overhead* de cerca de 2% para todas as quantidades de nodos; e replicação, que novamente apresentou *overheads* de cerca de 100%.

Outro experimento consistiu em provocar uma falha do tipo *fail/stop* em um dos processos e falhas de *software* com 1% de chance nos demais. O intuito deste experimento foi mostrar que a aplicação é capaz de suportar falhas de ambos os tipos, porém os autores não avaliam quantitativamente o *overhead*, que varia de 60% à 100%, segundo gráfico apresentado no artigo. O motivo de os *overheads* não terem apresentado grandes mudanças em relação ao cenário sem falhas se deve ao fato de que foi utilizado um *spare node* para substituir o nodo que falhou.

5.2 SEDAR (Montezanti et al., 2020)

Em 2020, Montezanti *et al.* propuseram o Soft Errors Detection and Automatic Recovery (SEDAR) [34], uma biblioteca que implementa mecanismos de tolerância a falhas para aplicações que utilizam GPUs em seu processamento, suportando múltiplas falhas transientes do tipo Silent Data Corruption (SDC) e erros de *timeout*. O SEDAR adota a estratégia de replicação em conjunto com o *checkpointing*, seja a nível de aplicação ou a nível de usuário, para proporcionar tolerância a falhas. A comunicação entre as GPUs é estabelecida através do MPI.

Na técnica de replicação, cada processo é composto por duas *threads*: a principal e a replicada. Ambas são executadas no mesmo *core* e portanto compartilham a memória cache, o que agiliza o carregamento de dados, uma vez que não é necessário carregá-los da memória principal. Ao final do processamento, o resultado de ambas as *threads* é comparado a fim de verificar se houve a falha. Caso o valor de saída das *threads* seja diferente, a tarefa será executada uma terceira vez e então será iniciado um mecanismo de votação para definir o valor de saída correto. Além disso, para reduzir a propagação da falha, as *threads* são sincronizadas antes do envio de cada mensagem com o intuito de comparar o conteúdo das mensagens e assim verificar a ocorrência da falha. Vale ressaltar que o procedimento de sincronização tem um tempo limite (*timeout*) para que a outra *thread* atinja o mesmo ponto de execução. Decorrido este tempo de *timeout* sem resposta, considera-se que houve uma falha na *thread* atrasada.

Na técnica de *checkpointing*, por sua vez, os *checkpoints* podem ser tirados a nível de sistema ou a nível de usuário. O *checkpoint* a nível de sistema é realizado de maneira síncrona pela biblioteca Distributed MultiThreaded Checkpointing (DMTCP) [35]. Apesar de síncronos, alguns *checkpoints* podem estar corrompidos devido a erros silenciosos, tornando necessário salvar uma cadeia de recuperação de *checkpoints*. Após a falha, retorna-se a computação para o último *checkpoint* salvo e, caso o erro se repita na reexecução, assume-se que este *checkpoint* está corrompido e retorna-se a computação para o *checkpoint* anterior. Esta abordagem é repetida sucessivas vezes até que a execução se estabilize.

Os *checkpoints* a nível de usuário também são realizados de maneira assíncrona e são comparados a cada sincronização para envio de mensagens. Caso sejam iguais, significa que não houve falhas e portanto o *checkpoint* atual é válido. Neste caso, o *checkpoint* anterior é descartado. Caso sejam diferentes, significa que houve falha e então reinicia-se a execução a partir do *checkpoint* anterior. Esta abordagem é vantajosa, pois evita que sejam armazenados *checkpoints* inválidos.

Os experimentos foram realizados em um cluster constituído por dois nodos, cada um com 8 *cores*. Foram selecionadas 3 aplicações para os testes: um algoritmo de multiplica-

ção de matrizes; o método de Jacobi; e o algoritmo de comparação de sequências biológicas *Smith-Waterman* (Seção 2.2.2). Cada aplicação foi submetida a vários cenários de falha e diferentes abordagens de tolerância a falhas. No caso do algoritmo de *Smith-Waterman*, a técnica de um único *checkpoint* a nível de usuário foi aquela que trouxe menor *overhead* para a aplicação, sendo cerca de 1% em um cenário sem falhas e de 6% para um cenários com falha. As demais técnicas apresentaram *overheads* similares para os cenários sem falha, porém se mostraram pouco eficientes em relação ao tempo de recuperação, apresentando *overheads*, em cenários de falha, de cerca de 50% em muitos casos e chegando até 100% em alguns casos.

5.3 FT-RAxML-NG (Hubner et al., 2020)

Em 2020, Hubner *et al.* [36] desenvolveram um mecanismo de tolerância a falhas para atuar em sua aplicação de bioinformática. O Randomized Axelerated Maximum Likelihood (RAxML) é uma ferramenta de inferência filogenética, que obtém a árvore filogenética, estrutura que reconstrói a ancestralidade evolutiva de um conjunto de genomas. A aplicação adota uma estratégia de paralelização com alta dependência de dados, na qual cada *core* executa um processo. A comunicação entre os processos é proporcionada através da interface de comunicação MPI.

O FT-RAxML-NG é a versão tolerante a falhas da ferramenta RAxML-NG. Seu mecanismo de tolerância a falhas utiliza a estratégia de *checkpoint* coordenado, metodologia na qual os processos sincronizam-se para realizar o *checkpoint*, que é armazenado em disco. O mecanismo de tolerância a falhas da aplicação ainda adota uma estratégia de *mini-checkpoints*: que são *checkpoints* locais, armazenados em memória e replicados entre os processos toda vez que sofrem algum tipo de atualização. A diferença entre os *checkpoints* e os *mini-checkpoints* não é claramente explicitada, porém deduz-se que os *checkpoints* são utilizados para armazenar parâmetros globais, ou seja, comuns a todos os processos, enquanto os *mini-checkpoints* armazenam parâmetros locais, específicos de cada processo.

A detecção da falha no FT-RAxML-NG é alcançada através do ULFM [32], implementação do MPI que introduz chamadas para dar suporte a falhas. O ULFM detecta a falha a partir dos chamados *heartbeat signals*, sinais que os processos trocam entre si, em intervalos regulares de tempo, para verificar se estão vivos. Caso um dos processos passe muito tempo sem enviar um desses sinais, até um determinado *timeout*, os processos observadores, aqueles que se comunicam com este processo, irão alegar que o mesmo falhou. Para garantir que este sinal será enviado em intervalos regulares, a aplicação adotou as *heartbeat threads*, ou seja, threads responsáveis apenas por enviar os *heartbeat signals* aos demais processos, em intervalos regulares de tempo. Vale ressaltar que este

tipo de metodologia abre margem para a detecção de falso-positivos: detecção de falhas que não ocorreram de fato. Para minimizar a probabilidade da detecção de falso-positivos, o FT-RAxML-NG realizou diversos experimentos para dimensionar experimentalmente o melhor tempo de timeout e a frequência de acionamento das *heartbeat threads*.

Após a detecção da falha, o ULFM sinaliza, a um dos processos restantes, a ocorrência da falha e este processo, por sua vez, irá propagar a informação aos demais processos através de uma chamada MPI dedicada. Em seguida, os processos remanescentes irão entrar em consenso acerca dos processos ativos, elegerão um novo comunicador e reestabelecerão a comunicação entre si. O FT-RAxML-NG assumirá então o controle da execução carregando, do *checkpoint* e do *mini-checkpoint*, o valor dos últimos parâmetros salvos, necessários à execução. Vale ressaltar que os *checkpoints* são idênticos em todos os nodos, enquanto os *mini-checkpoints* são específicos de cada nodo, porém são replicados em mais de uma máquina. Na fase de recuperação, ambos os *checkpoints* são carregados de uma das máquinas que não falhou e que possuía uma cópia deles. Também é importante destacar que o FT-RAxML-NG não utiliza *spare nodes*, nodos reserva utilizados para substituir aqueles que falharam e portanto, após a falha, deve ser feita a redistribuição de carga entre os processos remanescentes.

Os experimentos foram conduzidos em dois supercomputadores: O ForHLR II, que contém 1178 nodos, cada um equipado com dois processadores; e o SuperMUC-NG, constituído por 6336 nodos, cada um com dois processadores. Na seção de análise de resultados, os autores executam dois experimentos que têm por objetivo analisar e mensurar as fontes de *overhead*, através da comparação das versões tolerante a falhas e original do RAxML. No primeiro experimento é realizada a comparação entre as versões tolerante a falhas (FT-RAxML-NG) e original (RAxML-NG) do RAxML, ambos executados utilizando o OpenMPI v4.0. Neste caso, o *overhead* da versão tolerante a falhas foi pequena (de 0% a 4%). Já no segundo experimento, comparou-se a versão tolerante a falhas do RAxML, utilizando o ULFM, frente à versão original (novamente executada no OpenMPI v4.0) e o *overhead* do FT-RAxML-NG variou de 60 a 70%. Como a principal diferença entre os experimentos foi a interface de comunicação utilizada, os autores utilizaram este resultado para justificar que o ULFM é responsável pela maior parte do *overhead*.

Tal resultado parece ser coerente, entretanto é importante destacar dois fatores que não foram esclarecidos no texto: 1) No primeiro experimento, ambas as versões foram executadas utilizando o OpenMPI v4.0, enquanto no segundo conjunto de experimentos, a versão tolerante a falhas foi executada utilizando o ULFM e a versão original foi executada no OpenMPI v4.0, ou seja, as condições de comparação não foram exatamente as mesmas; 2) Além disso, deve-se destacar que o FT-RAxML-NG executado no primeiro conjunto de experimentos (utilizando o OpenMPI v4.0) apenas diferencia-se da versão original

na medida que implementa os *mini-checkpoints* e portanto nenhuma das operações de detecção da falha está ativa, uma vez que não há os mecanismos disponibilizados pelo ULFM. Dessa maneira, caso ocorresse uma falha, a aplicação seria incapaz de detectar e tratá-la. Tal condição favorece os resultados e conclusões dos autores, pois apresenta um *overhead* baixo, porém que não leva em consideração uma das principais fontes de *overhead* de qualquer mecanismo de tolerância a falhas: a detecção da falha.

5.4 Framework em Instâncias *Spot* (Brum et al., 2021)

Em 2021, Brum *et. al.* [37] elaboraram um *framework* para lidar com cenários de revogação de instâncias *spot*, na nuvem da Amazon Web Service (AWS). A aplicação selecionada para este projeto consiste na ferramenta de comparação de sequências biológicas, *MASA-CUDAlign*, que é baseada no algoritmo de *Smith Waterman* e que implementa a técnica de *checkpointing* para proporcionar tolerância a falhas em uma única GPU (Capítulo 4).

Conforme explicado na Seção 4.2, o *MASA-CUDAlign* salva, no Estágio 1, algumas linhas especiais para garantir o processamento do Estágio 2. Estas linhas especiais também podem ser utilizadas como *checkpoints* na execução com uma GPU, uma vez que, antes de se iniciar uma comparação, o *MASA-CUDAlign* verifica quais linhas especiais foram salvas e inicia sua execução da última delas.

O *framework*, por sua vez, é responsável por detectar a falha e reiniciar a aplicação e é dividido em dois módulos: *Controller* e *Worker*. O módulo *Worker* monitora o estado da máquina virtual (Virtual Machine (VM)) e comunica ao *Controller* a mudança de estado (*active* ou *revoked*). O *Controller*, por sua vez, tem a função de escolher a máquina virtual que executará a aplicação, inicializá-la no *EC2*, serviço de instâncias da *AWS*, e migrar a aplicação para outra *VM*, em caso de revogação.

A revogação pode ser detectada de três maneiras. Na primeira, o módulo *Worker* permanece esperando por uma mensagem proveniente da *AWS* informando que a instância será revogada. Já a segunda maneira envolve a utilização do Software Development Kit (SDK) da *AWS* para *python*, o *boto3*, que monitora o estado da *VM*. Por fim, caso a conexão Secure Shell Protocol (SSH), estabelecida entre o *Controller* e *Worker* seja interrompida e não seja possível reestabelecê-la por 3 tentativas consecutivas, o *Controller* infere que a instância foi revogada.

O projeto do *framework* adota um algoritmo guloso, que busca sempre executar a instância mais barata que garanta o fim da execução antes do *deadline* proposto. Dessa maneira, quando ocorre a revogação, o *framework* dará preferência a instâncias *spot*, que são mais baratas, porém podem ser revogadas e, apenas em casos de muitas revogações,

selecionará uma instância *on demand* para finalizar a execução. As instâncias *on demand* são irrevogáveis, porém são cerca de 3 vezes mais caras que as *spot* [38] [39]. Após selecionar a nova *VM*, o *framework* irá inicializá-la e, por fim, reiniciar a aplicação do último *checkpoint* completamente salvo.

Os experimentos foram realizados utilizando apenas uma instância GPU *Spot* da *AWS* e a ferramenta *MASA-CUDAlign*. O primeiro experimento consistiu em comparar o tempo de execução da aplicação com e sem o *framework*. Neste caso, os resultados demonstraram um *overhead*, em média, abaixo de 3%. Já no experimento seguinte, verificou-se o *overhead* do *framework* em um cenário de falhas, que foi, em média, cerca de 13%. Vale ressaltar que tal *overhead* se dá devido à: seleção da nova *VM*, inicialização da mesma e reexecução da aplicação.

5.5 Tolerância a Falhas em Aplicações IoT (Mudassar et al., 2022)

Em 2022, Mudassar *et al.* [40] propuseram um mecanismo de tolerância a falhas para aplicações de Internet of Things (IoT), atuando no modelo de computação chamado *edge computing*. Os autores se aproveitam da grande disponibilidade de máquinas presentes neste modelo de computação para utilizar a técnica de replicação, aliada ao *checkpoint* assíncrono.

Na técnica desenvolvida, os nodos são organizados em grupos, formados por um nodo organizador e os demais membros do grupo. O nodo organizador tem a função de atribuir tarefas aos demais membros, bem como realizar a detecção, tratamento e recuperação da falha. O nodo organizador é composto por três módulos: o gerenciador de recursos, que distribui as tarefas aos nodos de acordo com sua disponibilidade e capacidade de processamento; o módulo detector de falhas, que utiliza *heartbeats* enviados aos nodos para verificar se os mesmos estão vivos; e o gerenciador de falhas, que realiza os procedimentos de recuperação da aplicação após a falha e também indica nodos alternativos para a reexecução da tarefa.

Os membros do grupo, por sua vez, possuem dois módulos: um gerenciador de *checkpoints*, que cria e salva os *checkpoints* em seu próprio nodo; e um gerenciador de replicações, que replica o *checkpoint* armazenado localmente, em outros nodos.

Como mencionado anteriormente, o *checkpoint* é realizado de maneira assíncrona e portanto, cada processo tira o *checkpoint* no momento que julgar mais oportuno. Além disso, os *checkpoints* são realizados de maneira incremental e portanto eles armazenam apenas as diferenças entre o *checkpoint* anterior e o atual.

Já a técnica de replicação opera tanto em termos de tarefas quanto de *checkpoints*, ou seja, os nodos reserva (*spare*) tanto executarão as tarefas realizadas nos nodos ativos quanto armazenarão os *checkpoints* deles. Os nodos escolhidos para replicação são selecionados pelo módulo de tolerância a falhas do nodo organizador, que prioriza nodos ociosos, pertencentes ao mesmo grupo do nodo que falhou, ou nodos de outro grupo.

Diante de uma falha, o módulo de tolerância a falhas do nodo organizador a detectará através dos sinais de *heartbeats* e então selecionará um novo nodo para assumir o trabalho daquele que falhou. Vale ressaltar que este trabalho parte do pressuposto de que os nodos que falharam não retornam à execução e portanto a replicação se torna mais difícil à medida que ocorrem múltiplas falhas, uma vez que a disponibilidade de nodos reduz-se. Em seguida, o organizador carrega, no nodo substituto, uma cópia (de outro nodo) do último *checkpoint* válido.

Um dos experimentos realizados pelos autores consistiu em avaliar o *delay* da aplicação em um cenário com diferentes quantidades de falhas. No pior dos cenários, com falhas em 20% das tarefas, o *delay* foi de apenas um segundo. O artigo não informa o quanto isso representa em termos percentuais ou o tempo total da aplicação em um cenário sem falhas, o que dificulta mensurar a representatividade deste atraso. Vale ressaltar que este atraso, aparentemente pequeno, se deve principalmente à técnica da replicação, visto que após a falha, outro nodo assume a execução quase que de imediato, reduzindo consideravelmente o *overhead*. Entretanto, o custo desta eficiência se mostra na necessidade de se ter nodos adicionais (*spare*) no sistema.

5.6 Discussão

A Tabela 5.1 apresenta as principais características dos artigos analisados neste capítulo. Conforme pode ser visualizado na primeira coluna da Tabela 5.1, os artigos selecionados foram publicados recentemente, entre os anos de 2020 e 2022. A coluna *Técnica* apresenta as técnicas de tolerância a falhas utilizadas nos trabalhos. Note que metade dos trabalhos analisados utiliza alguma versão da ferramenta MPI, tanto para proporcionar a comunicação entre múltiplos nodos, quanto para auxiliar na detecção e tratamento da falha. Ainda nesta coluna, verifica-se a ampla utilização das técnicas de *checkpointing* e replicação. Os tipos de falha tratados nos artigos consistiram, majoritariamente, em falhas do tipo *Fail/Stop*, porém alguns trataram de falhas de software (terceira coluna).

Os artigos foram selecionados priorizando aplicações de tolerância a falhas recentes e que atuassem na área de Bioinformática. Porém, isto não foi um fator limitante, uma vez que há, na lista de trabalhos relacionados, soluções genéricas e até mesmo aquelas que tratam de aplicações na área de IoT. Quanto à abrangência do conteúdo dos *check-*

points (coluna 5), temos estratégias genéricas, que salvam todo o estado dos processos, e específicas de aplicação, que usam o conhecimento da aplicação para salvar somente as informações necessárias para a recuperação.

Por fim, a maioria dos trabalhos analisados implementa mecanismos para tolerância a uma única ou múltiplas falhas. A única exceção refere-se ao artigo de Brum *et al.* [37], que também utiliza a ferramenta *MASA-CUDAlign* com tolerância a falhas para uma única GPU.

Dentre os trabalhos que suportam múltiplas falhas, três usam mecanismos do MPI [28],[34],[36], e o quinto [40] usa replicação em conjunto com *checkpoint/recovery*.

Artigo	Ano	Técnica	Tipo de Falha	Aplicação	Estratégia	Suporta Falhas
[28]	2020	Replay, Replicação, ABFT*, Checkpoint/ Recuperação, MPI-ULFM+ Fenix	Software, Fail/Stop	Smith Waterman, Stencil, Conjugate Gradient, Cholesky	Genérica	Simples/ Múltiplas
[34]	2020	Replicação, Checkpoint / Recuperação, rMPI, RedMPI	Software	Smith-Waterman, Jacobi, MM	Genérica	Simples/ Múltiplas
[36]	2021	Checkpoint / Recuperação, MPI-ULFM	Fail/Stop	RAxML	Específica de Aplicação	Simples/ Múltiplas
[37]	2021	Checkpoint / Recuperação	Revogação (Fail/Stop)	Smith Waterman	Específica de Aplicação	Simples
[40]	2022	Replicação + Checkpoint / Recuperação	Fail/Stop	IoT	Genérica	Simples/ Múltiplas
Nosso Trabalho	2025	Checkpoint / Recuperação	Fail/Stop	Smith Waterman	Específica de Aplicação	Simples/ Múltiplas

Tabela 5.1: Artigos que implementam tolerância a falhas.

Diante do exposto, nesta dissertação é proposto um mecanismo *application-specific* de tolerância a falhas, que não usa versões tolerantes a falhas do MPI (Tabela 5.1). Essa decisão foi tomada porque a aplicação alvo (*MASA-CUDAlign*) não usa MPI, e realiza a comunicação diretamente com *sockets*. Com isso, a solução proposta pode ser aportada tanto para soluções que usam MPI como para as que não usam.

Além disso, optou-se por não usar replicação, pois a solução deve funcionar tanto em ambiente de *cluster* local como de nuvem. Nesses dois casos, a alocação de recursos adicionais para execução replicada de tarefas acrescenta um custo considerável em termos de recursos alocados à aplicação. Ao invés disso, optamos por utilizar a técnica do *checkpointing*, pois esta é utilizada em todos os trabalhos analisados, bem como para se aproveitar de uma funcionalidade já provida pelo *MASA-CUDAlign*, que é o armazenamento da última coluna da iteração em disco.

Por último, optou-se por tratar tanto falhas únicas quanto falhas múltiplas, já que o último tipo de falha ocorre em *clusters* quando uma parte da rede de interconexão se torna indisponível. Assim sendo, após a revisão da literatura realizada, notou-se que esta é a primeira proposta de mecanismo de tolerância a falhas *application-specific* para aplicações paralelas que não utilizam MPI, suportando falhas múltiplas, conforme será descrito em detalhes no próximo capítulo.

Capítulo 6

FT-CUDAlign: Mecanismo de Tolerância a Falhas no MASA-CUDAlign

A presente dissertação de mestrado tem por objetivo propor um mecanismo de tolerância a falhas para a ferramenta de comparação de sequências biológicas *MASA-CUDAlign*, mais especificamente na sua versão mais recente: o *dynamic-MultiBP* (Seção 4.5). Para tanto, foi adotada uma estratégia específica de aplicação (*application-specific*), que se aproveita do atual funcionamento do *dynamic-MultiBP*, no qual há o armazenamento das colunas de borda, de cada iteração, para utilização como *checkpoints* globais coordenados.

O mecanismo de tolerância a falhas deste trabalho utiliza módulos da aplicação *MASA-CUDAlign* para identificar falhas, a depender do momento em que esta ocorre. Entretanto, apenas o módulo *Controller* (Seção 4.5) trata a falha e, portanto, a premissa deste projeto é de que não ocorrem falhas no mestre do *cluster* (máquina que executa o *Controller*).

6.1 Visão Geral

Para o projeto do *CUDAlign* Multi GPU tolerante a falhas (FT-CUDAlign), optou-se por usar a última coluna gravada ao final de cada iteração (Seção 4.5) como *checkpoint* coordenado. Caso ocorra falha na execução de uma iteração, o *FT-CUDAlign* usa o último *checkpoint* e re-executa o *MultiBP* a partir da iteração precedente. A Figura 6.1 apresenta a visão geral do *FT-CUDAlign* a cada iteração.

O mecanismo de tolerância a falhas inicia sua atuação na etapa de detecção pré-execução, que ocorre antes do processamento da parte da matriz de programação dinâmica na iteração corrente. Caso ocorra uma falha nesta etapa, a GPU que falhou é

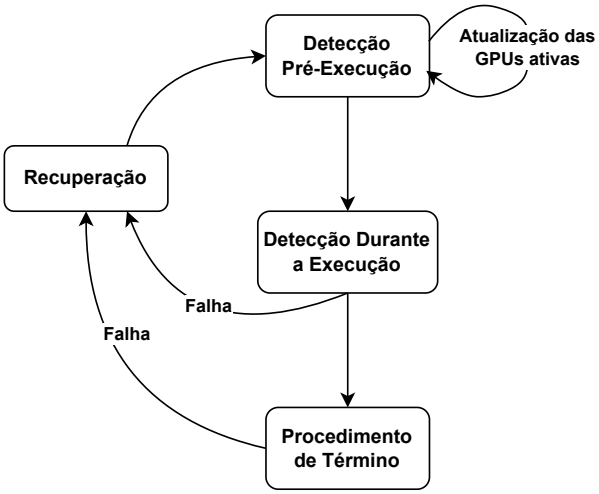


Figura 6.1: Visão geral do *FT-CUDAlign* a cada iteração.

retirada da lista de GPUs ativas e a execução da iteração corrente é iniciada com as GPUs remanescentes, desde que haja pelo menos duas.

A etapa seguinte consiste na detecção durante a execução da iteração, que ocorre no processamento da matriz de programação dinâmica. Diante de uma falha, inicia-se o protocolo de recuperação e, em seguida, reinicia-se a computação da iteração atual somente com as GPUs ativas no momento, conforme ilustrado na Figura 6.1. Após finalizar o processamento da parte da matriz de programação dinâmica na iteração corrente, executa-se o procedimento de término, no qual são realizados protocolos de sincronização com o intuito de detectar falhas após o processamento da matriz, garantindo a finalização da iteração atual para que se possa dar início à próxima.

6.2 Diagrama de Sequências

A Figura 6.2 apresenta o diagrama de sequências de uma execução do *FT-CUDAlign* utilizando três GPUs. Após a fase de detecção pré-execução (Seção 6.3.1), inicia-se efetivamente a etapa de execução do *CUDAlign* (Seção 6.3.2). Primeiramente, o *Controller* envia comandos aos módulos *Monitor* para darem início à execução do *CUDAlign*. Cada GPU processa normalmente sua região da matriz de programação dinâmica até o momento em que ocorre uma falha em uma das GPUs (na Figura 6.2, a falha ocorre na GPU 2). Neste momento, inicia-se a etapa de detecção, na qual as GPUs adjacentes à GPU que falhou (no exemplo, são as GPUs 1 e 3) detectam a falha e sinalizam ao *Controller*.

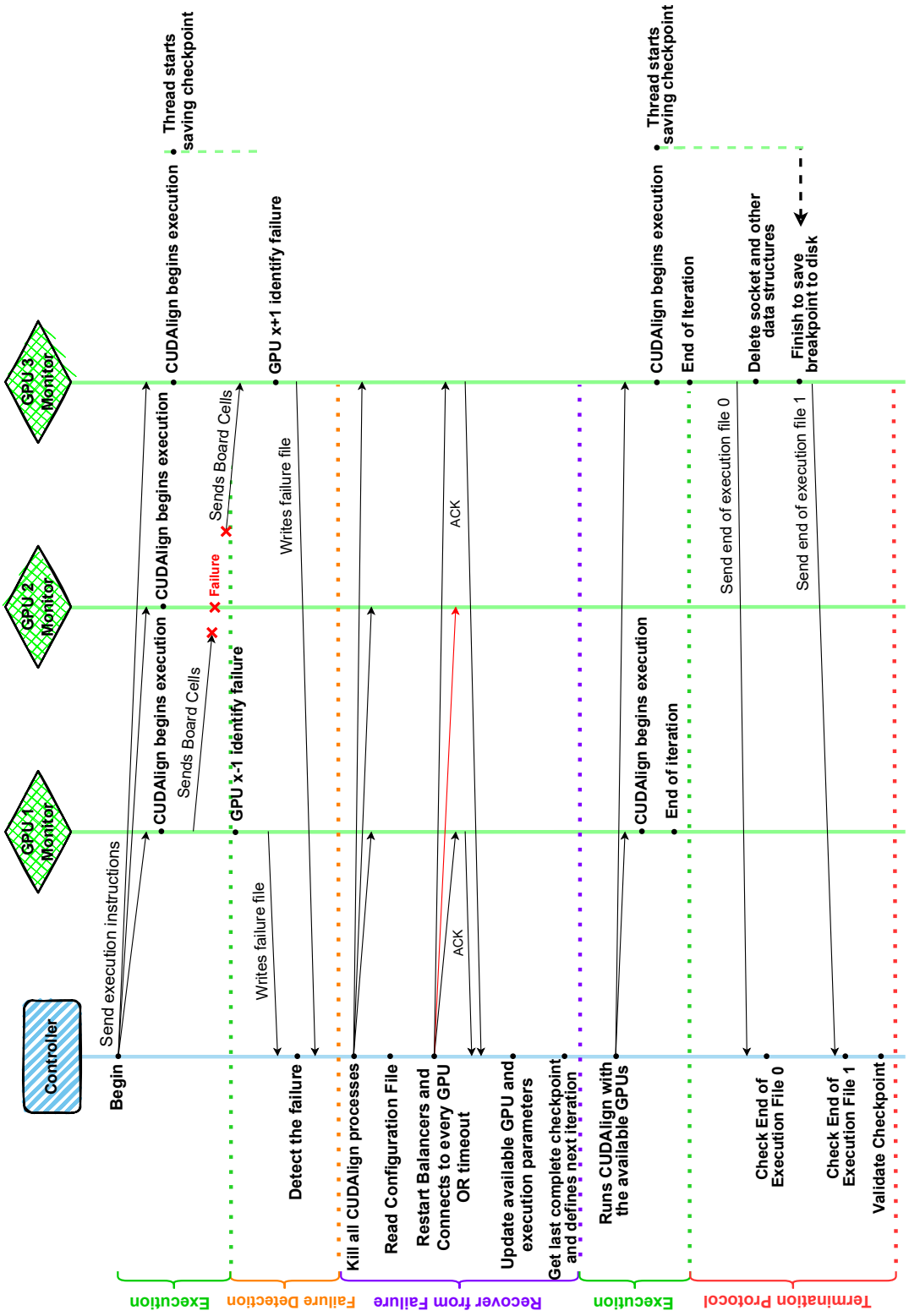


Figura 6.2: Fluxo do mecanismo de tolerância a falhas.

Assim que o *Controller* recebe o sinal de falha, inicia-se a etapa de recuperação, detalhada na Seção 6.4.

Após a fase de recuperação, inicia-se uma nova etapa de execução, na qual o *Controller* envia comandos para reexecutar o CUDAlign, do último *checkpoint* válido. A execução segue, sem falhas, até o final da iteração, no qual são realizados os protocolos de término (Seção 6.5), que finalizam a iteração atual para que uma nova iteração possa ser executada.

6.3 Detecção da Falha

A primeira etapa necessária para desenvolver o mecanismo de tolerância a falhas consiste em detectar a falha. Neste projeto, a detecção da falha é alcançada por meio da análise do retorno de funções de *socket* antes e durante o processamento da matriz de programação dinâmica.

6.3.1 Detecção Pré Execução

A execução da versão *multiBP* do *MASA-CUDAlign* se inicia buscando a conexão do módulo *Controller* com os *Monitores* [12]. Para tanto, inicialmente são listados os endereços *IP* de todas as GPUs inseridas no arquivo de configurações. Em seguida, utiliza-se da função *connect* para o estabelecimento da conexão entre os módulos.

Caso não seja possível estabelecer a comunicação com qualquer um dos módulos *Monitor*, a função *connect* retorna um valor negativo. O *Controller* repete a chamada da função *connect* por até n vezes (neste trabalho definido empiricamente como três), enquanto seu retorno for um valor negativo. Caso, em uma dessas chamadas, a função *connect* seja bem sucedida, a comunicação é estabelecida e a execução da aplicação prossegue normalmente. Por outro lado, se as três tentativas forem mal sucedidas, o *Controller* retira a respectiva GPU da lista de máquinas disponíveis e atualiza outros parâmetros de execução. O *Controller* retorna à sua execução normal, enviando novos comandos aos *Monitors* das GPUs ativas.

6.3.2 Detecção Durante a Execução

Dando prosseguimento ao fluxo de execução, os módulos *Monitor* inicializam o *CUDAlign*, com base nos comandos recebidos pelo *Controller* (Figura 4.12). Em seguida, os processos *CUDAlign* estabelecem comunicação entre si, via *socket*, e iniciam o processamento da matriz de programação dinâmica. Neste momento, o mecanismo de tolerância a falhas avalia o retorno das funções de troca de células *send*, *recv* e *select*, com o intuito de detectar falhas. Analogamente ao procedimento adotado na detecção pré-execução, caso

o retorno dessas funções não coincida, por três tentativas seguidas, com o valor esperado, a falha é sinalizada ao *Controller*.

A utilização de *sockets* para detecção da falha implica que apenas as GPUs envolvidas na comunicação podem identificá-las. No *MASA-CUDAlign*, cada GPU apenas se comunica com as GPUs adjacentes e, portanto, apenas estas estão aptas a detectar a falha.

Assim como mencionado na Seção 4.5, a última GPU do *MASA-CUDAlign* salva a última coluna da iteração em disco. Na versão sem tolerância a falhas, esta coluna é utilizada, na iteração posterior, para dar continuidade ao processamento da matriz. No projeto do *FT-CUDAlign*, esta coluna é utilizada como *checkpoint* e, portanto, caso uma falha seja identificada, o *Controller* inicia a etapa de recuperação e identifica a última coluna (*checkpoint*) completamente salva em disco. A execução será reiniciada a partir deste *checkpoint* e todo o processamento posterior a ele será descartado. Vale ressaltar que, após a falha, a GPU que falhou pode, ou não, retornar, mantendo ou reduzindo a quantidade de GPUs ativas no escopo para executar as próximas iterações. Para tanto, busca-se a conexão com todas as GPUs listadas, porém são necessárias pelo menos duas GPUs ativas para dar prosseguimento à execução da aplicação.

6.4 Recuperação da Falha

Uma vez identificada a falha, o *Controller* inicia a etapa de recuperação, detalhada na Figura 6.3. Primeiramente, o *Controller* envia comandos a todas as GPUs para terminarem seus processos, *CUDAlign* e *Monitor*, e então lê o arquivo de configurações, com o intuito de atualizar suas informações acerca das GPUs disponíveis para a execução da próxima iteração. Em seguida, o *Controller* reinicia os *Monitor* e busca reestabelecer a comunicação com cada um deles. Caso a conexão não seja estabelecida após três tentativas, o *Controller* assume que a GPU em questão não está ativa e atualiza a lista de GPUs, retirando a respectiva e atualizando alguns outros parâmetros de execução.

6.4.1 Atualização do *Split*

O *split* é um vetor de parâmetros que define a quantidade de partes na qual a matriz de similaridade será dividida, bem como o tamanho de cada uma de suas partições. No exemplo apresentado pela Figura 4.11, foram selecionadas três GPUs e apenas um *breakpoint*, que divide a matriz em duas regiões, resultando no total de seis partições, cada uma contendo duas colunas.

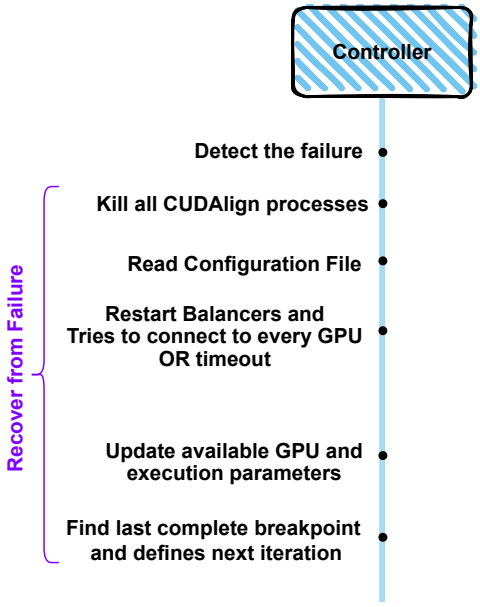


Figura 6.3: Etapas de recuperação do *Controller*.

Em um cenário de falha sem retorno é necessário recalcular os valores de *split* de maneira a reduzir a quantidade de partes na qual a execução será dividida, bem como para redistribuir a carga de trabalho entre as GPUs ativas.

A Figura 6.4 apresenta um cenário que exemplifica a necessidade de recalcular o parâmetro *split*. A matriz foi inicialmente dividida em nove partes, pois sua execução foi ramificada em três iterações, utilizando três GPUs em cada. Na situação descrita pela Figura 6.4, a primeira iteração é completamente finalizada, porém, durante a execução da segunda iteração, a terceira GPU falha e não retorna. O *Controller* então aciona o protocolo de recuperação da falha que tem como última ação a atualização do parâmetro *split*. Como a quantidade de iterações permaneceu constante em três, porém a quantidade de GPUs foi reduzida para dois, a nova quantidade total de partes processadas será de sete: três partes provenientes da primeira iteração, que foi completamente concluída, e quatro partes provenientes da execução das duas iterações seguintes, utilizando duas GPUs em cada.

A Figura 6.4 ainda apresenta dois cenários: com e sem o recálculo do parâmetro *split*. Caso o *split* não seja recalculado, o tamanho das partições permanecerá o mesmo e a carga de trabalho da GPU que falhou não será redistribuída entre as GPUs ativas, de maneira que, ao final da execução, a região cinza não será processada. Já no cenário de recálculo do *split*, o *Controller* obtém o valor total da carga de trabalho em cada iteração, que

no exemplo em questão é seis, e então redistribui essa carga igualmente entre as GPUs restantes. Dessa maneira, garante-se que a matriz será completamente processada. Note que, na última iteração, a própria aplicação optou por recalcular o valor do *split*, porém ainda mantendo o mesmo valor de carga de trabalho por iteração.

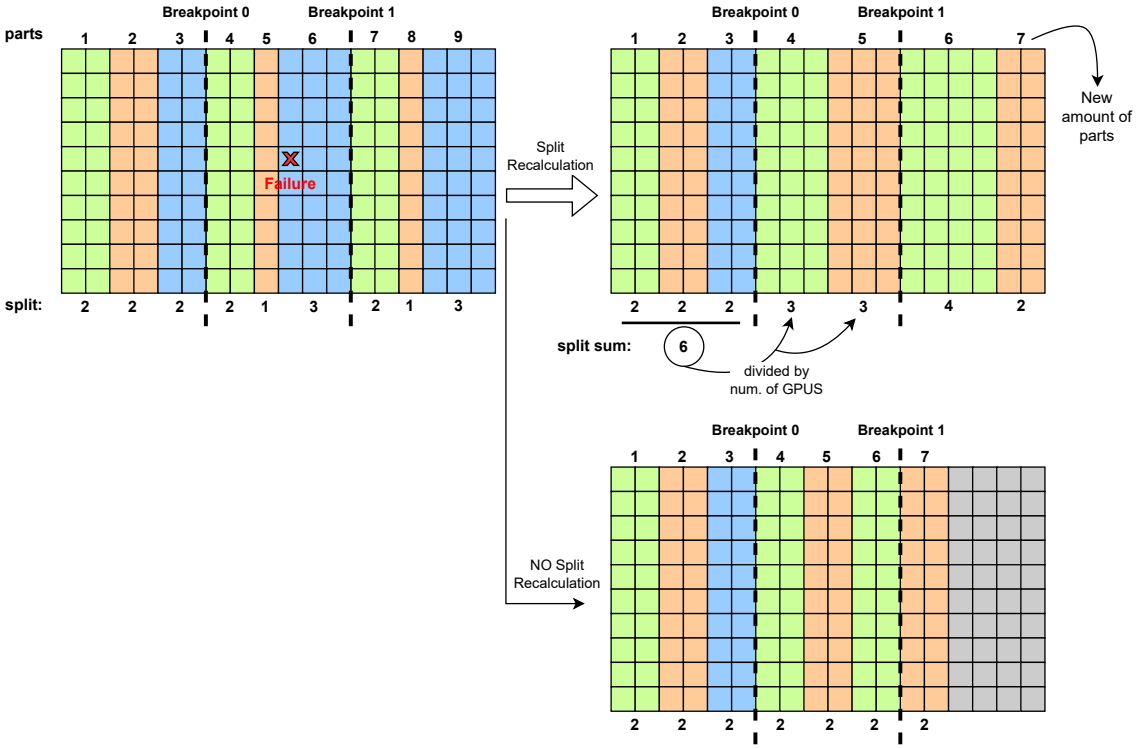


Figura 6.4: Recalculando o parâmetro split.

6.4.2 Definição da Próxima Iteração

Para identificar a próxima iteração a ser processada, o *Controller* verifica se o *checkpoint* foi totalmente salvo. Essa verificação é necessária pois a existência de *buffer cache* em memória nos sistemas de arquivos faz com que as escritas em arquivos sejam postergadas e realizadas de maneira assíncrona. Assim sendo, escritas assumidas pelo programador como feitas totalmente em disco podem estar ainda parcialmente ou totalmente em memória [41]. Para tanto, o *Controller* compara os tamanhos do *checkpoint* e de uma coluna da matriz. Caso sejam iguais, significa que o *checkpoint* foi completamente escrito em disco. Caso contrário, o *chekpoint* está incompleto e por isso será necessário reprocessar a última iteração por completo.

6.5 Protocolos de Término

Os protocolos de término têm a função de detectar a falha após o processamento da matriz de programação dinâmica em cada iteração, bem como verificar se a iteração atual foi concluída com sucesso para só então dar início ao processamento da próxima.

6.5.1 Sincronização de Destruição do *Socket*

Após a finalização do processamento da matriz de programação dinâmica na iteração, o *MASA-CUDAlign* ainda realiza algumas operações antes de ser encerrado, tais como: finalizar escrita do arquivo de estatísticas, terminar de salvar *checkpoint* em disco e destruir estruturas de dados (*buffer*, *socket*, dentre outras).

Na versão *dynamic-MultiBP*, os *sockets* são destruídos logo após o término do envio de todas as células. O mesmo não é válido para a versão *FT-CUDAlign*, uma vez que a destruição do *socket* impossibilita a detecção de falhas a partir do término do processamento da matriz até a finalização completa da iteração. Para solucionar este problema, o protocolo de sincronização de destruição de *socket* atua, primeiramente, postergando a deleção dos *sockets*.

Além disso, devido à dependência de dados entre as partições, cada GPU finaliza sua execução em um momento diferente. Esta condição possibilita um cenário no qual a penúltima GPU finaliza completamente seu processo *MASA-CUDAlign* antes que a última sequer tenha terminado de processar a sua partição da matriz. Neste cenário, a falha jamais será identificada, visto que a única GPU com a qual a última se comunica é a penúltima que, na situação descrita, já finalizou seu processo e que, portanto, é incapaz de detectar a falha.

Por esta razão, a segunda forma de atuação deste protocolo, apresentada na Figura 6.5, ocorre assim que uma GPU termina de processar sua região da matriz e de enviar células à GPU seguinte. Neste cenário, ao invés de destruir seu *socket* de escrita e finalizar seu processo *CUDAlign*, esta GPU permanecerá esperando por uma mensagem proveniente do *socket* de leitura da GPU seguinte informando que a mesma já finalizou sua execução. Só então o *socket* de escrita será destruído e o processo finalizado.

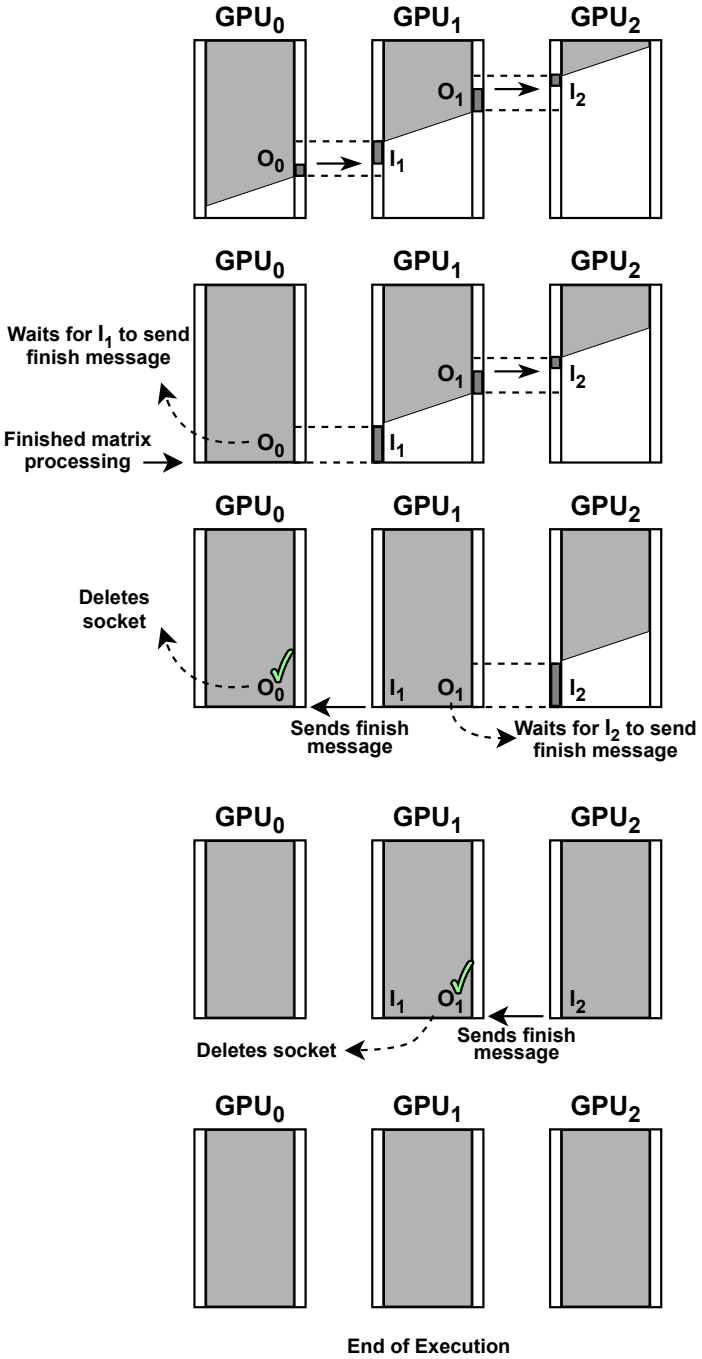


Figura 6.5: Protocolo de sincronização de destruição dos *sockets*.

6.5.2 Sincronização de Término da Execução

A última GPU do *MASA-CUDAlign* escreve, ao final de sua execução, um arquivo utilizado para sinalizar ao *Controller* que a iteração atual já foi completamente finalizada e que os comandos para a próxima iteração já podem ser enviados. Somente após a recepção deste arquivo é que o *Controller* validará o último *checkpoint* para verificar se a iteração corrente foi completamente finalizada. Por isso, a escrita deste arquivo deve ser a última ação do *MASA-CUDAlign*, ocorrendo até mesmo após a deleção do *socket*. Nesse contexto, caso uma falha ocorra entre a deleção do *socket* e a escrita do arquivo, o programa entra em *deadlock*, pois o *Controller* ficará indefinidamente esperando pelo sinal do *MASA-CUDAlign*, que nunca chegará devido à falha na última GPU.

Para solucionar este problema, modificou-se o código do *MASA-CUDAlign* de maneira a criar dois arquivos para sinalizar o término da execução: um antes e o outro após a deleção do *socket*. Assim que o *Controller* identifica o primeiro arquivo, ele espera pela criação do segundo arquivo até um determinado *timeout* (definido de maneira empírica). Caso o segundo arquivo de término de execução não seja criado dentro do tempo estabelecido, o *Controller* assume que ocorreu uma falha e então ativa o protocolo de recuperação. A Figura 6.6 apresenta tal interação entre o *Controller* e a última GPU do *MASA-CUDAlign*.

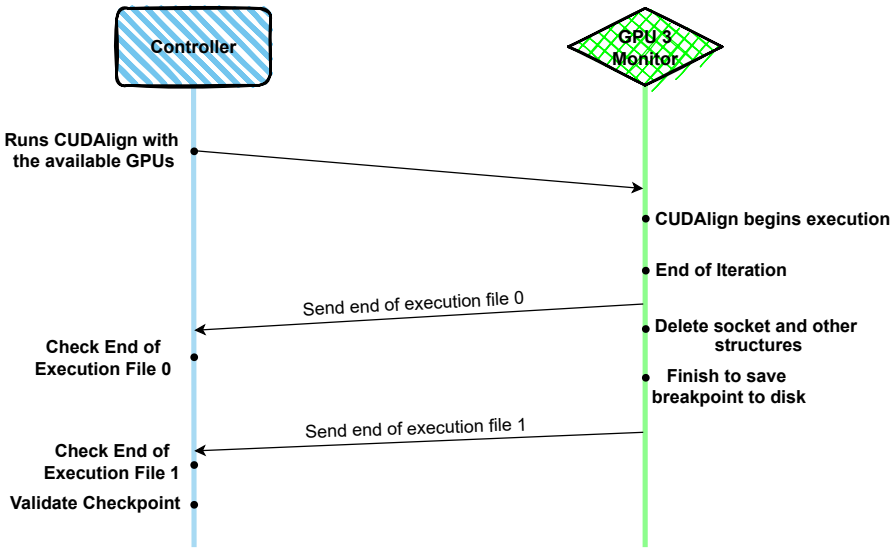


Figura 6.6: Protocolo de sincronização de término da execução.

Capítulo 7

Resultados Experimentais

Os resultados analisados neste capítulo tem três objetivos, os quais são: 1) Analisar o impacto do mecanismo de tolerância a falhas no tempo de execução da ferramenta *MASA-CUDAlign-MultiBP*; 2) Analisar como diferentes cenários de falha impactam no tempo de execução do *FT-CUDAlign*; 3) Verificar o comportamento do *FT-CUDAlign* em quantidade superior de GPUs, provocando falhas únicas ou múltiplas.

7.1 Sequências Selecionadas

As sequências selecionadas para realização dos experimentos estão apresentadas na Tabela 7.1, na qual são fornecidas informações acerca delas, tais como: identificação, nome dos organismos aos quais pertencem e tamanho, em quantidade de pares de bases (*Base Pairs* - BP). Devido a restrições de tempo e custo, apenas as quatro primeiras sequências foram selecionadas para realização da maioria dos experimentos, organizadas em ordem crescente de tamanhos: $5MBP$, $10MBP$, $23MBP$, $47MBP$.

Note que o tamanho de cada sequência é aproximadamente o dobro da anterior. Já as duas últimas sequências, os cromossomos 19 e 20, por serem as maiores dentre as selecionadas ($62MBP$ e $65MBP$, respectivamente), foram utilizadas nos experimentos com oito nodos. 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 [42].

7.2 Ambientes de Execução

Os experimentos foram desenvolvidos em dois ambientes distintos: na nuvem da AWS e no LAboratório de sistemas Integrados e COncorrentes (LAICO), da UnB.

Seq.	Id	Organismos	Tamanho
5M	AE016879.1	*Bacillus anthracis str. Ames*	5.227.293
	AE017225.1	*Bacillus anthracis str. Sterne*	5.228.663
10M	NC_014318.1	*Amycolatopsis mediterranei U32*	10.236.715
	NC_017186.1	*Amycolatopsis mediterranei S699*	10.236.779
23M	NT_033779.4	*Drosophila melanogaster chr. 2L*	23.011.544
	NT_037436.3	*Drosophila melanogaster chr. 3L*	24.543.557
chr21	NC_000021.9	*Homo sapiens chromosome 21*	46.709.983
	NC_006488.4	*Pan troglodytes chromosome 21*	33.445.071
chr19	NC_000019.10	*Homo sapiens chromosome 19*	59.455.082
	NC_006486.4	*Pan troglodytes chromossome 19*	62.185.004
chr20	NC_000020.11	*Homo sapiens chromosome 20*	65.364.869
	NC_006487.4	*Pan troglodytes chromosome 20*	67.483.737

Tabela 7.1: Quadro de informações das sequências selecionadas para experimentos.

7.2.1 Nuvem - AWS

O ambiente construído na *AWS* foi um *cluster*, constituído por um nodo mestre e uma quantidade variável de nodos trabalhadores, todos conectados por uma rede *VPC (Virtual Private Cloud)*. O *website* da *AWS* apenas informa que a velocidade da rede pode chegar até $10Mb/s$, porém não há maiores especificações do valor exato. A instância selecionada para a execução dos experimentos pertence à chamada família *g*, da *AWS*, cujas máquinas são especializadas em processamento gráfico e que, portanto, possuem GPUs robustas. Mais especificamente, a instância selecionada foi a *g5.xlarge*, cujas especificações, retiradas de [43], estão apresentadas na Tabela 7.2.

Os recursos utilizados neste *cluster* são disponibilizados pelo *ParallelCluster*, ferramenta de código aberto utilizada para criação e gerenciamento de *clusters* na *AWS* [44]. Para utilizá-lo é necessário preencher um arquivo de configurações contendo informações acerca dos recursos a serem utilizados, tais como: tipo e quantidade de instâncias, espaço de armazenamento, rede, dentre outros. O *AWS-ParallelCluster* automatiza a criação do *cluster* conforme as especificações indicadas. Para mais detalhes sobre a ferramenta *AWS ParallelCluster* na execução de aplicações de comparação de sequências biológicas, consultar [13].

Por se tratar de um ambiente homogêneo, todos os nodos alocados ao *cluster* foram da instância *g5.xlarge*, e a carga de trabalho foi distribuída igualmente entre as GPUs, *split* $333 : 333 : 334$, no caso de 3 GPUs. É importante destacar que a escolha de utilizar a AWS como provedor de serviços de computação em nuvem se deve à participação dos autores no programa de concessão de créditos na nuvem, relativo ao projeto CNPQ/AWS 421828/2022-6.

Instância	GPUs	Memória da GPU	Performance da Rede
g5.xlarge	1	24 GB	até 10 GB/s
	Núcleos	**Clock**	**Arquitetura**
	9216	1710 MHz	Ampere

Tabela 7.2: Especificações da instância g5.xlarge da *AWS*.

7.2.2 LAICO

Assim como na *AWS*, o ambiente do *LAICO* também forma um *cluster* que, no caso do *LAICO*, é constituído por um nodo mestre e dois nodos trabalhadores, entretanto, *no LAICO*, as GPUs são heterogêneas. A Tabela 7.3 apresenta as especificações destas três GPUs.

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

Tabela 7.3: Especificações das GPUs do LAICO.

A GPU1 foi selecionada para ser o mestre do *cluster*, enquanto as GPUs 2 e 3 serviram como nós trabalhadores. Neste contexto, a GPU1 executa tanto o módulo *Controller*, quanto o *Monitor*, enquanto as GPUs 2 e 3 executam um *Monitor* cada. Devido às diferentes especificações entre as GPUs, foi utilizada a técnica de *profiling* para comparar o desempenho das GPUs na execução do *MASA-CUDAlign*. Inicialmente, realizou-se a comparação das sequências de $5MBP$ (Tabela 7.1) utilizando apenas uma das GPUs por vez. Os tempos de execução obtidos foram: $GPU2(226.06s) > GPU1(161.32s) > GPU3(149.29s)$. Em seguida, selecionou-se a GPU mais lenta (GPU2) como referência e obteve-se, como resultado, que a GPU1 foi cerca de 28% mais rápida, enquanto a GPU3 foi cerca de 33% mais rápida.

Dada tal comparação de desempenho, optou-se por redistribuir a carga de trabalho de maneira proporcional à diferença de desempenho obtida da técnica de *profiling*, com *split* 244 : 267 : 389, respectivamente para as GPUs 1, 2 e 3. Em seguida, conectou-se as GPUs por meio da rede local da universidade, com velocidade de transferência de $94Mb/s$.

7.3 Experimentos

Os experimentos foram divididos em três categorias, conforme apresentado na Figura 7.1. A primeira categoria tinha por objetivo comparar as versões original (*MASA-CUDAlign-MultiBP*) e tolerante a falhas (*FT-CUDAlign*) do *MASA-CUDAlign*, sob um

cenário sem falhas, de maneira a mensurar o impacto do mecanismo de tolerância a falhas no desempenho da aplicação. Para tanto, foram utilizadas as quatro primeiras sequências da Tabela 7.1; três nodos, sendo um mestre e dois trabalhadores; quantidade variável de *breakpoints*: de um a três; e alternando entre ambas as versões da aplicação.

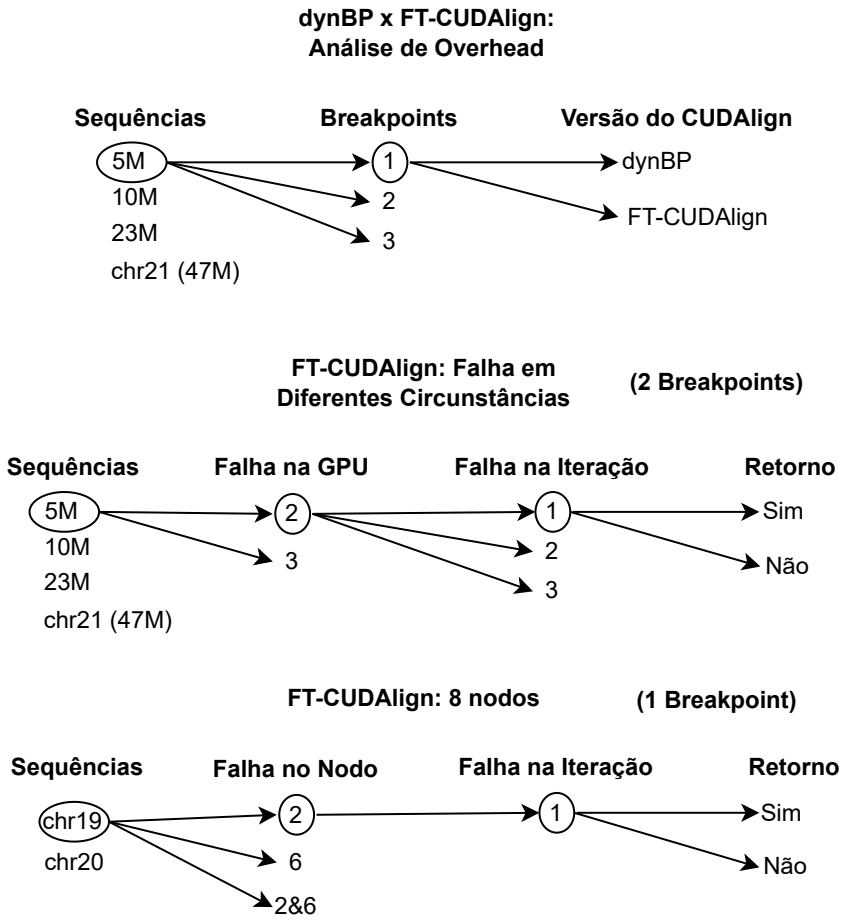


Figura 7.1: Escopo de experimentos.

Já a segunda bateria de experimentos tinha o intuito de avaliar o impacto de diferentes cenários de falha, no tempo de execução. Assim, a falha poderia ser provocada nas GPUs 2 ou 3, e em qualquer uma das três iterações. O mecanismo de tolerância a falhas parte da hipótese de que a máquina que executa o módulo *Controller* não falha e, por essa razão, os experimentos não incluíram o cenário de falha na GPU1. Além disso, também foi adicionada a condição de retorno, ou seja, se a GPU em que ocorreu a falha retornaria, ou não, ao escopo de GPUs ativas para processar as demais iterações.

A última bateria de experimentos tinha o objetivo de avaliar se o mecanismo de tolerância a falhas seria eficaz de operar em uma quantidade superior de máquinas (oito), sob um cenário de falhas simples e múltiplas. Os experimentos realizados foram análogos à segunda bateria, porém utilizando as maiores sequências da Tabela 7.1, e incluindo o cenário de falha múltipla nos nodos 2 e 6.

A falha foi simulada via *software*, matando os processos *CUDAlign* e *Monitor* em uma das GPUs. Para obter resultados mais facilmente comparáveis, optou-se por padronizar o momento de ocorrência da falha, de maneira que esta sempre fosse provocada ao atingir metade do processamento da iteração.

7.3.1 Cenário Sem Falhas

A Figura 7.2 apresenta os resultados na nuvem da *AWS* onde, no eixo y é ilustrado o *overhead* percentual da versão tolerante a falhas frente à versão original do *Dynamic-MultiBP*; e no eixo x, a quantidade de *breakpoints* em que a matriz será dividida. Analisando a Figura 7.2, é possível verificar que as sequências menores (5M e 10M) apresentam maior *overhead* percentual do que as sequências maiores (23M e chr21), nas quais o *overhead* é aproximadamente constante. Além disso, à medida que se aumenta a quantidade de *breakpoints*, o *overhead* também tende a aumentar.

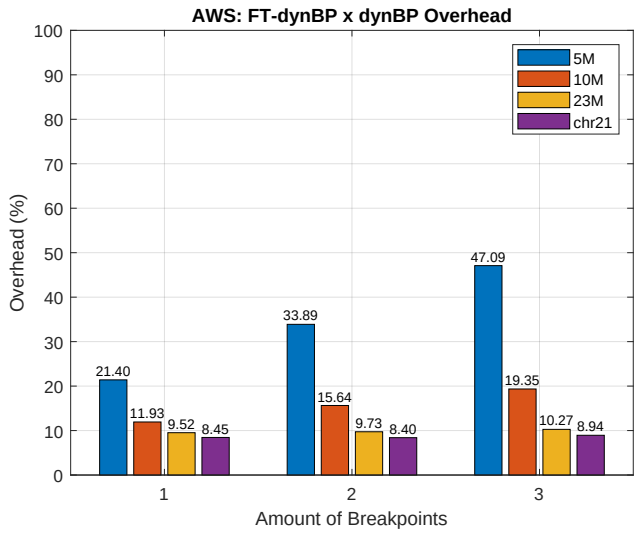


Figura 7.2: AWS: *overhead* do *FT-dynBP* frente à versão original.

Para entender corretamente os resultados descritos acima, primeiramente é necessário analisar as fontes de *overhead* do mecanismo de tolerância a falhas. A primeira fonte de atraso reside na identificação da falha. Conforme descrito na Seção 6.3.2, a detecção da falha é feita por meio da análise do retorno das funções de troca de células. A avalia-

ção do retorno de tais funções provoca um *overhead* inerente à comunicação, de maneira que, quanto maior for o par de sequências, mais células serão comunicadas, mais retornos serão avaliados e, consequentemente, maior será o *overhead*. Ou seja, o atraso proveniente da identificação da falha é proporcional ao tamanho do par de sequências a serem comparados.

A segunda fonte de *overhead* é proveniente dos protocolos de Destruição de *Socket* e Sincronização de Término da Execução, Seções 6.5.1 e 6.5.2. O protocolo de destruição de *socket* faz com que a GPU que já terminou de processar e enviar células da sua partição não encerre seu processo de imediato, mas aguarde por uma mensagem proveniente da GPU seguinte informando que a mesma já encerrou seu processamento, para só então encerrar seus processos. Tal sincronização é realizada com o intuito de garantir a detecção da falha pós processamento da matriz, porém provoca *overhead* para a aplicação na medida em que obriga as GPUs a esperarem umas pelas outras.

O protocolo de término de execução, por sua vez, provoca *overhead* devido à espera pela criação dos dois arquivos: um antes e outro após a destruição do *socket*. Tal *overhead* é aproximadamente constante em termos absolutos, pois as etapas necessárias em tal sincronização são apenas a escrita do primeiro arquivo e a espera pela escrita do segundo arquivo. Entretanto, tal protocolo é repetido a cada iteração e, portanto, quanto mais iterações, maior será o *overhead* proveniente dele. É por esta razão que, na Figura 7.2, o *overhead* cresce à medida que se incrementa a quantidade de *breakpoints*. Este efeito é mais evidente em sequências menores, nas quais o tempo de atraso proveniente deste protocolo é percentualmente mais significativo. Entretanto, isto não constitui um problema, uma vez que o mecanismo de tolerância a falhas foi desenvolvido para lidar com sequências mais extensas. Além disso, apesar de o *overhead* nas sequências menores (5M e 10M) ser mais significativo em termos percentuais, em termos absolutos esta diferença é pouco significativa. Isto pode ser verificado na Figura 7.3, na qual as barras de mesma cor representam, respectivamente, os tempos de execução na versão original e tolerante a falhas do *MASA-CUDAlign-MultiBP*.

A Figura 7.4 mostra que, tanto nos resultados obtidos no *LAICO* quanto naqueles obtidos na *AWS*, o *overhead* cresce com o número de *breakpoints*. Porém, é possível notar que os *overheads* no *LAICO* foram significativamente menores que os da *AWS*. A principal diferença reside no fato de que os tempos de execução, entre as versões original e tolerante a falhas do *MASA-CUDAlign-MultiBP*, obtidos no *LAICO* foram muito mais próximos, resultando em *overheads* percentuais pouco significativos. Tal diferença entre os tempos de execução foi tão baixa que, em dois casos, a versão tolerante a falhas chegou a ser inclusive mais rápida que a original, devido a flutuações no tempo entre execuções. Tais cenários estão representados na Figura 7.5, mais especificamente nas comparações

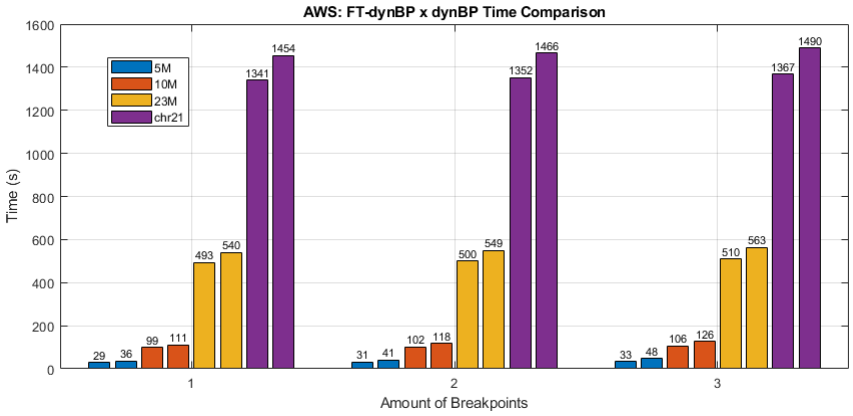


Figura 7.3: AWS: Comparação dos tempos de execução entre as versões original e tolerante à falhas.

do cromossomo 21, utilizando um e três *breakpoints*.

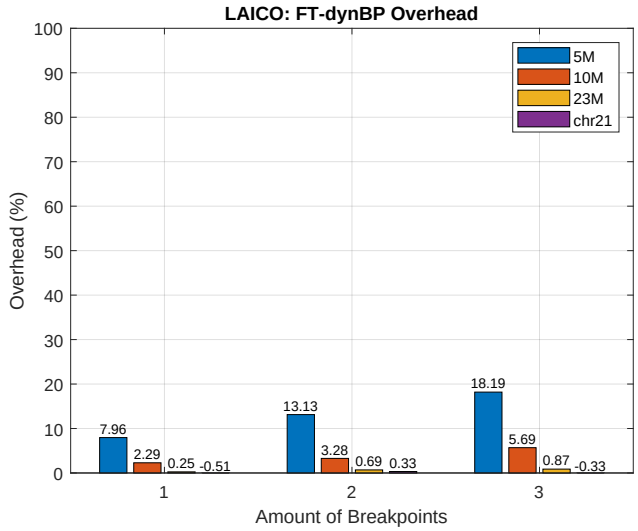


Figura 7.4: LAICO: *overhead* do *FT-dynBP* frente à versão original.

7.3.2 Cenário com Falhas em Diferentes Circunstâncias

A Figura 7.6 apresenta quatro gráficos de barra. Cada um desses gráficos representa o resultado obtido, na *AWS*, da comparação dos quatro primeiros pares de sequências especificados na Tabela 7.1. O eixo y mede o tempo de execução do *MASA-CUDAlign*, na versão *FT-CUDAlign*, enquanto o eixo x indica em qual iteração ocorreu a falha. Além disso, as cores das barras representam diferentes cenários de falha:

- **Barra Azul:** Falha na GPU2 com retorno;

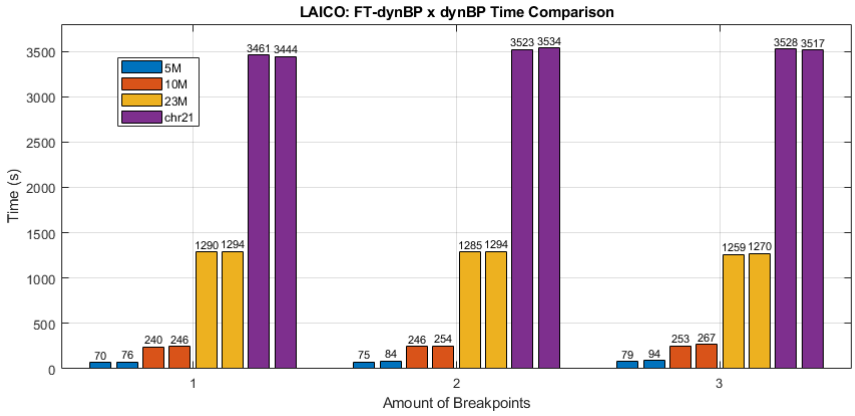
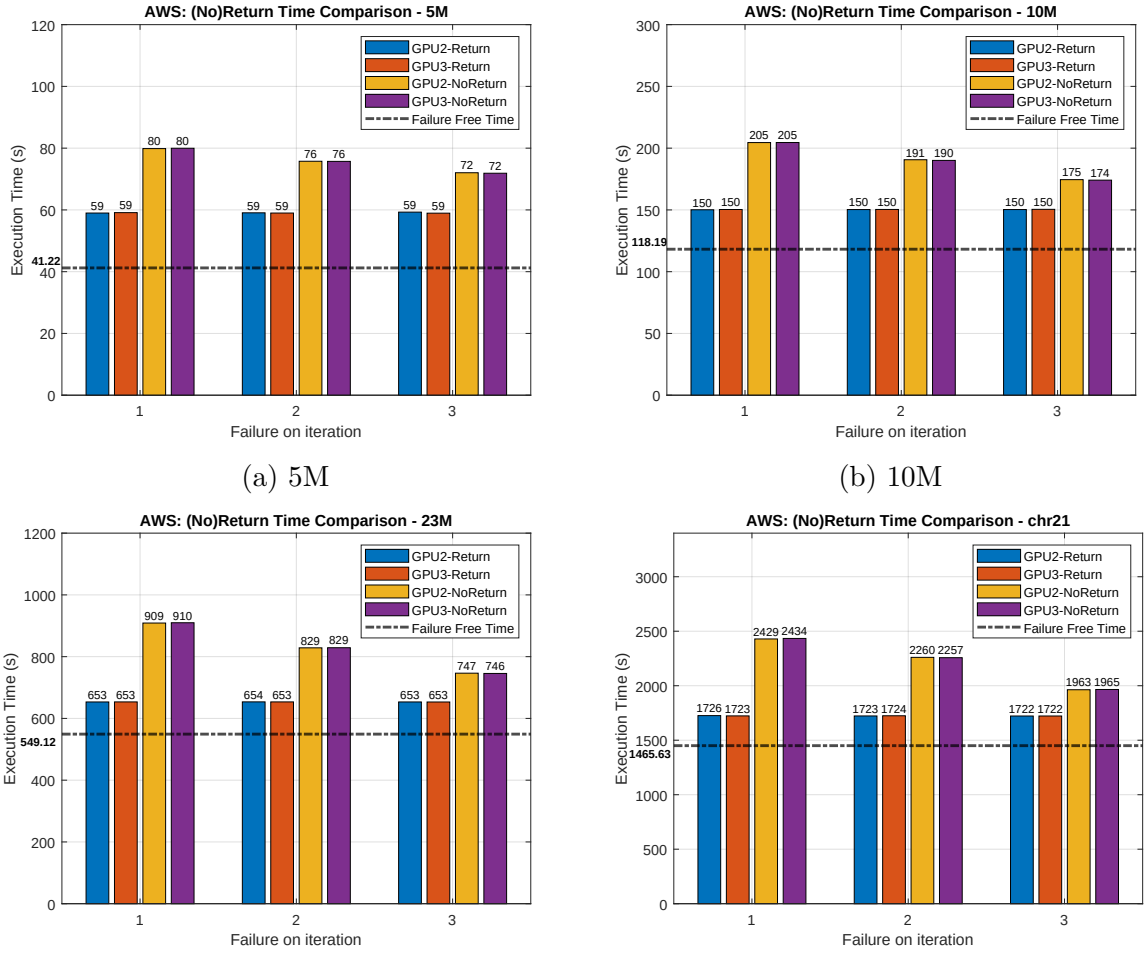


Figura 7.5: LAICO: Comparação dos tempos de execução entre as versões original e tolerante à falhas.



(a) 5M

(b) 10M

(c) 23M

(d) 47M

Figura 7.6: Tempos de execução em diferentes cenários de falha na AWS.

- **Barra Amarela:** Falha na GPU3 com retorno;
- **Barra Laranja:** Falha na GPU2 sem retorno;
- **Barra Roxa:** Falha na GPU3 sem retorno;
- **Linha Tracejada:** Execução sem falha.

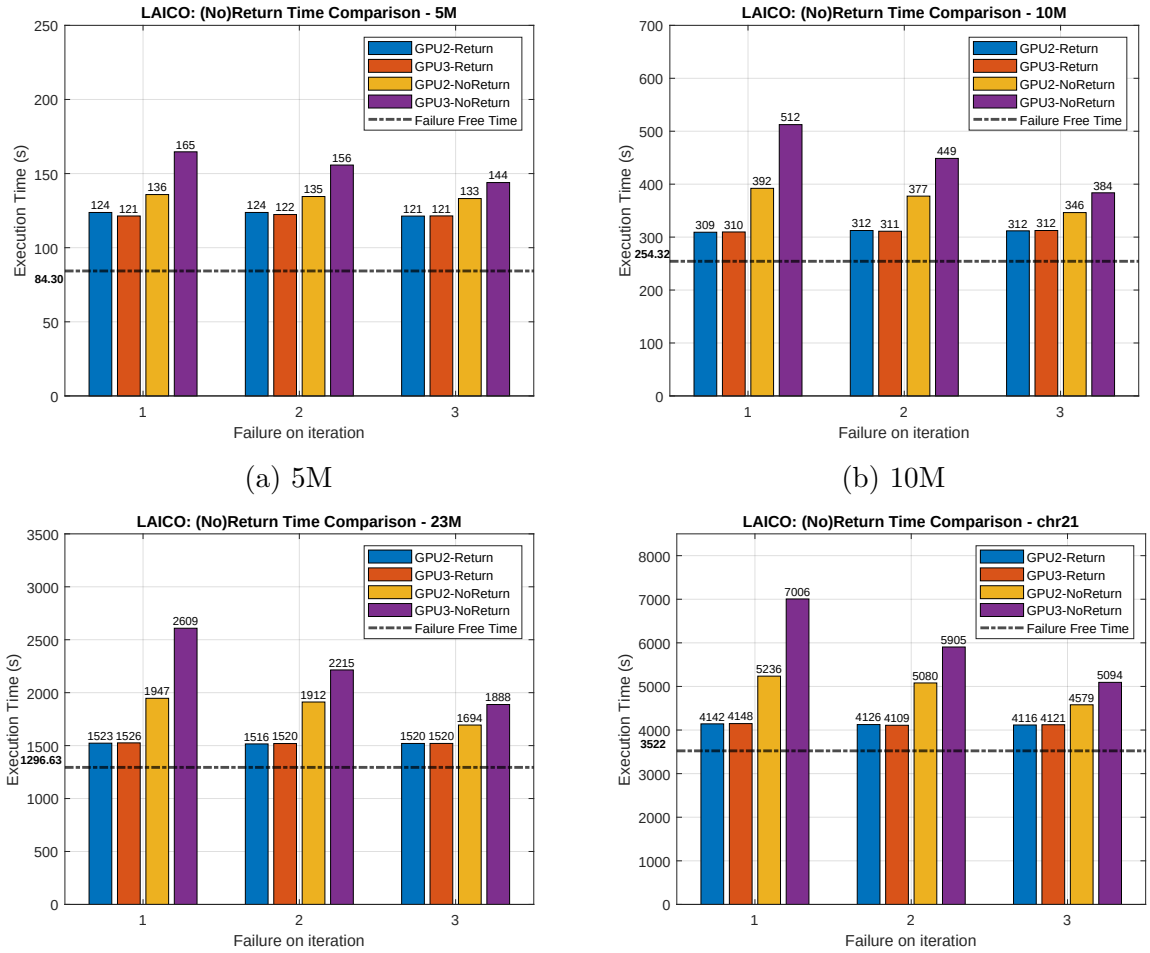
Na Figura 7.6 é possível observar que, em um cenário de falha seguida pelo retorno da GPU, tanto a iteração, quanto a GPU em que ocorreu a falha não impactam no tempo de execução. Isto porque, como a GPU retorna, a quantidade e a capacidade das GPUs disponíveis permanecerá constante.

Analisando o cenário de falhas sem retorno na *AWS*, percebe-se que a GPU em que ocorre a falha continua não impactando no tempo de execução, pois o ambiente é homogêneo. O mesmo não pode ser dito acerca da iteração na qual ocorre a falha, pois uma falha, sem retorno, no início da execução fará com que mais iterações sejam processadas com uma menor quantidade de GPUs. A Figura 7.7 apresenta os mesmos resultados obtidos no *LAICO*.

Analisando o cenário de falha com retorno, verifica-se resultados semelhantes aos obtidos na *AWS*, com a diferença que os tempos de execução no *LAICO* são maiores, uma vez que o poder de processamento das GPUs locais é inferior ao daquelas utilizadas na *AWS*. Ao avaliar o cenário sem retorno, por sua vez, observa-se que o tempo de execução muda consideravelmente de acordo com a GPU em que ocorre a falha. Isto se deve ao fato de que o *LAICO* possui um ambiente heterogêneo. Conforme descrito na Seção 7.2.2, a GPU3 se mostrou a mais rápida do escopo, e cerca de 33% mais rápida que a GPU2. Este efeito justifica a razão pela qual uma execução com falha na GPU3 consome mais tempo do que uma execução com falha na GPU2.

7.3.3 *Overhead* com Falhas Simples e Múltiplas - 8 nodos

Os experimentos desta seção tinham como objetivo verificar se o mecanismo de tolerância a falhas funcionaria para uma quantidade superior de GPUs com uma ou múltiplas falhas simultâneas. Para tanto, selecionou-se novamente a instância $g5.xlarge$, porém utilizando oito nodos, ao invés de três. Devido à maior capacidade de processamento, optou-se por comparar as duas últimas sequências da Tabela 7.1, que são as maiores do escopo ($62MBP$ e $65MBP$, respectivamente). A Figura 7.8 apresenta os resultados deste experimento, com as falhas sendo provocadas nos nodos 2 e 6. As duas falhas simultâneas também foram provocadas em outros conjuntos de nodos (1 e 5), porém tais resultados não foram apresentados na Figura 7.8, pois os tempos de execução foram muito próximos àqueles obtidos com a falha nos nodos 2 e 6.



(a) 5M (b) 10M

(c) 23M (d) 47M

Figura 7.7: Tempos de execução em diferentes cenários de falha no LAICO.

As barras em azul mostram o tempo de execução sem falhas. Já as barras em laranja e roxo sinalizam a falha no nodo 2, sob os respectivos cenários: com uma falha e retorno; com uma falha e sem retorno. As barras amarela e verde sinalizam a falha no nodo 6 sob os mesmos cenários. A barra em ciano, por sua vez, indica a falha simultânea nos nodos 2 e 6.

Os resultados deste experimento mostram que o mecanismo de tolerância a falhas se mostrou eficaz ao operar em quantidades superiores de máquinas, sob cenários de falhas simples e múltiplas. Os efeitos observados na Figura 7.8 mostram que, assim como os resultados obtidos na Seção 7.3.2, os tempos de execução aumentam de maneira crescente de acordo com os cenários: sem falha, falha em um nodo e retorno; falha em um nodo e sem retorno; falha em dois nodos sem retorno. Além disso, observou-se que os tempos de execução para comparar o cromossomo 19, em cada cenário, foram menores que os do cromossomo 20. Isto porque o tamanho da matriz formada pelo cromossomo 19 é um pouco menor do que o do cromossomo 20.

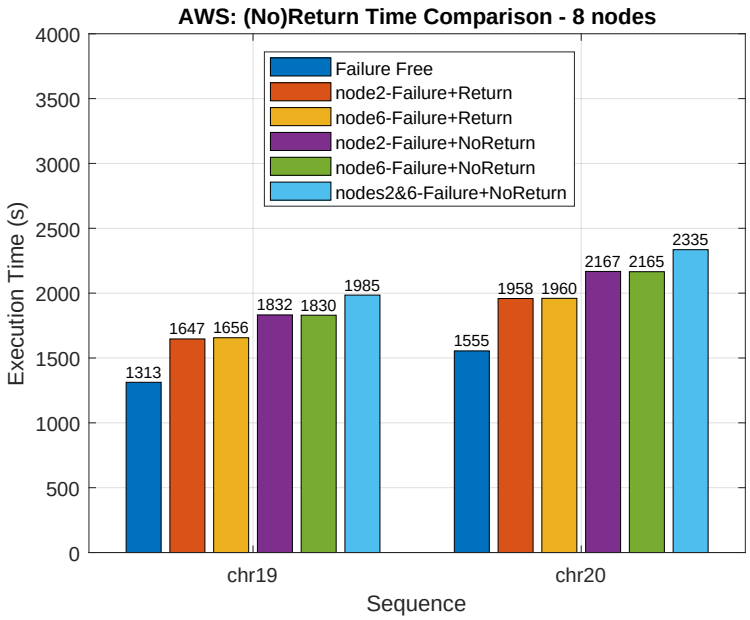


Figura 7.8: AWS: atraso em cenário de falhas para oito nodos.

7.4 Análise de Contribuição de *Overheads*

Os resultados a seguir foram obtidos a partir da análise das execuções sem falhas e das execuções com falha e retorno, provenientes dos experimentos anteriores. O intuito desta análise é avaliar as contribuições das principais fontes de *overhead* da versão tolerante a falhas (*FT-CUDAlign*), em um cenário de falhas simples.

Para comparar tais cenários, inicialmente foi necessário identificar as fontes de *overhead* em um cenário de falha. As três principais fontes de *overhead* elencadas foram: identificação, recuperação e reexecução, conforme é apresentado na Equação 7.1.

$$overhead_{failure} = identification + recovery + reexecution \tag{7.1}$$

O tempo de identificação refere-se ao intervalo entre a ocorrência e a detecção da falha. Conforme mencionado na Seção 6.3.2, para evitar uma falsa identificação de falha devido à latência de comunicação na rede, antes de se identificar a falha é necessário realizar a análises do retorno das chamadas *send* e *recv*, com intervalos de x segundos entre elas. Nos nossos testes, esses valores foram empiricamente definidos como três análises e dois segundos. Por esta razão, o tempo de identificação é aproximadamente constante em seis segundos. A escolha do tempo de intervalo entre as tentativas

O *overhead* de recuperação, por sua vez, refere-se ao tempo que o *Controller* demora, a partir da detecção da falha, para reiniciar a execução da aplicação. Conforme apresentado

na Figura 6.3, as etapas de recuperação são sempre as mesmas e independem do tamanho das sequências, tornando este *overhead* aproximadamente constante.

Por fim, o *overhead* de reexecução se refere ao tempo necessário para recalcular toda a região da matriz que é perdida após a falha. Nos experimentos em questão, fixou-se a quantidade de *breakpoints* em dois e, portanto, a matriz será dividida em três partes de mesmo tamanho, conforme ilustra a Figura 7.9. Além disso, foi estipulado que a falha sempre seria provocada ao atingir 50% do processamento de uma iteração e, portanto, o percentual de matriz que deverá ser reprocessado após a falha é cerca de 16.66%. Note que, como a região a ser reprocessada é um valor percentual referente ao tamanho total da matriz, o *overhead* de reexecução é proporcional ao tamanho das sequências selecionadas para comparação.

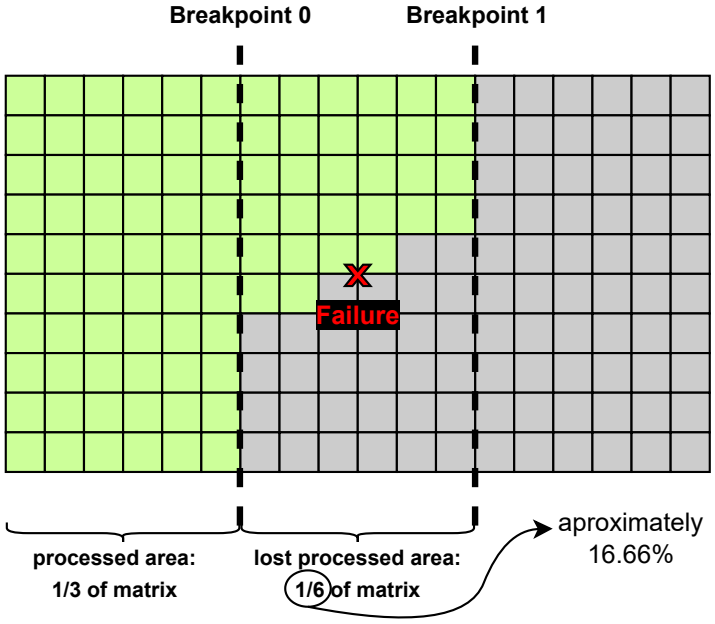


Figura 7.9: Detalhamento da falha.

A Figura 7.10 apresenta uma descrição percentual do tempo de execução em um cenário de falha com retorno, evidenciando assim a contribuição de cada uma dessas fontes de *overhead* e da efetiva execução da ferramenta *MASA-CUDAlign-MultiBP*, no tempo de execução total.

Na Figura 7.10, a barra amarela representa o tempo de execução da ferramenta *MASA-CUDAlign-MultiBP*, que se manteve aproximadamente constante em todas as sequências, em aproximadamente 80% do tempo total da execução com falha e retorno. Já a barra

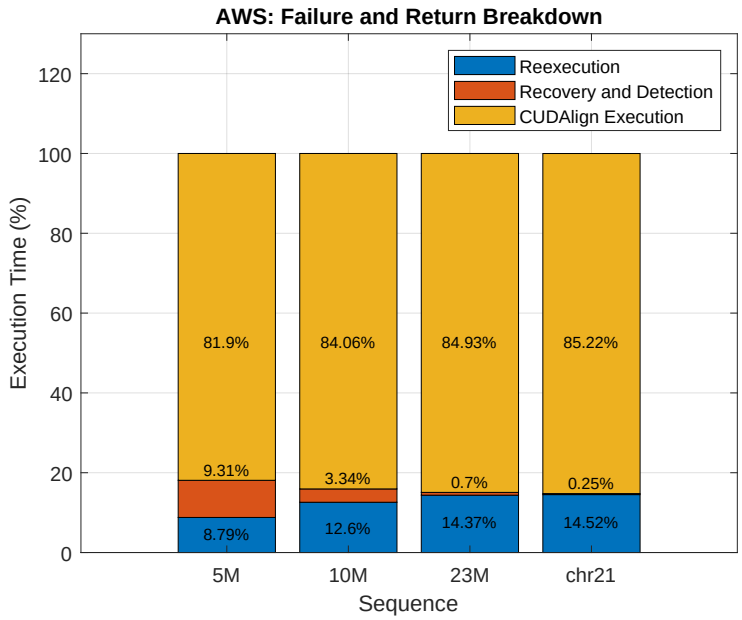


Figura 7.10: LAICO: descrição percentual de atividades no tempo de execução em um cenário de falha com retorno.

em laranja representa o percentual referente aos *overheads* de identificação e recuperação combinados. Finalmente, a barra azul apresenta o *overhead* de reexecução.

Analisando a Figura 7.10, nota-se que à medida que se aumenta o tamanho da sequência, o *overhead* referente à identificação e recuperação reduz-se expressivamente, enquanto o *overhead* de reexecução se torna mais significativo. Isto porque o tempo de identificação e recuperação é aproximadamente constante e seu efeito apenas se torna considerável em sequências menores, nas quais o tempo de reexecução é menos significativo devido ao tamanho reduzido da matriz. Este fenômeno também evidencia um ponto crítico no mecanismo de tolerância a falhas: tanto o tamanho da iteração quanto o momento em que a falha ocorre impactam profundamente no *overhead*. Isto porque, se a falha ocorrer a uma mesma porcentagem de processamento da iteração, a área de reexecução e, portanto, o *overhead* será maior, quanto maior for a área total da iteração. Analogamente, quanto mais ao final da iteração a falha ocorrer, maior será a área de reprocessamento, resultando novamente em maior *overhead* de reexecução.

Capítulo 8

Conclusão e Trabalhos Futuros

A presente dissertação propôs, implementou e avaliou um mecanismo de tolerância a falhas multi-GPU para a ferramenta *MASA-CUDAlign-MultiBP*. O desenvolvimento do projeto foi dividido em três principais etapas: 1) Detecção da falha, que é possível por meio da análise do retorno das funções de comunicação entre *sockets*, seja durante a troca de dados ou no estabelecimento da comunicação entre as GPUs; 2) Recuperação: esta etapa utiliza de colunas que já seriam salvas em disco para execução da próxima iteração, como *checkpoints*; 3) Protocolos de Término: tais protocolos são necessários para proporcionar a detecção da falha após o processamento da matriz de programação dinâmica na iteração e garantir que uma nova iteração não será iniciada antes de se verificar que a anterior foi concluída com sucesso.

Os experimentos foram divididos em três baterias: a primeira tinha o intuito de analisar o impacto do mecanismo de tolerância a falhas no tempo de execução da aplicação; a segunda tinha por objetivo analisar o impacto da falha em diferentes cenários; e a terceira buscava avaliar o funcionamento do *FT-CUDAlign* em uma quantidade superior de GPUs, submetidas a falhas simples e múltiplas. Para realizar os experimentos, foram selecionadas sequências de diferentes tamanhos, de maneira a verificar seu impacto no tempo de execução da aplicação.

Os experimentos mostraram que o *overhead* da versão tolerante a falhas (*FT-CUDAlign*) do *MASA-CUDAlign-MultiBP* frente à versão original é percentualmente mais significativo em sequências menores (5M e 10M), do que nas maiores (23M e chr21). Ainda assim, isto não constitui um problema visto que, apesar de percentualmente o *overhead* ser significativo, em termos absolutos sua influência não se mostra expressiva. Além disso, o foco do *FT-CUDAlign* é tratar falhas no processamento de sequências grandes, que apresentaram *overhead* percentual reduzido. Na nuvem da *AWS*, o *overhead* das sequências grandes permaneceu sempre abaixo de 11%, enquanto no *LAICO*, os *overheads* foram significativamente menores, sempre permanecendo abaixo de 1% para as mesmas sequências,

em um cenário sem falhas.

Já os experimentos referentes ao cenário com falhas em diferentes circunstâncias mostram o impacto do momento da falha no *overhead* da aplicação. Isto porque o prejuízo da falha cresce à medida que a falha se aproxima ao final da iteração, devido ao tamanho da área de processamento perdida. Além disso, em um cenário de falha sem retorno, a falha é mais prejudicial nas iterações iniciais e é suavizada à medida que se alcança as iterações finais, caso no qual há menor área de processamento pendente para ser executada com menos GPUs. Por fim, a quantidade de iterações também impacta no *overhead* da falha uma vez que, quanto mais iterações, menor será o tamanho de cada uma delas, causando menor *overhead* de reexecução. Por outro lado, quanto mais iterações, maior será o *overhead* proveniente da sincronização e da interrupção da execução.

A última bateria de experimentos evidenciou que o mecanismo de tolerância a falhas foi eficaz ao lidar com uma quantidade superior de GPUs, bem como em tratar falhas simples e múltiplas. Assim sendo, conclui-se que os objetivos desta dissertação foram atingidos e que o *FT-CUDAlign* é um mecanismo de tolerância a falhas leve e eficaz para o *MASA-CUDAlign-MultiBP*.

8.1 Trabalhos Futuros

A hipótese adotada neste trabalho é de que o mestre do *cluster* (máquina que executa o módulo *Controller*) não falha. Um dos possíveis trabalhos futuros consiste em tornar o módulo *Controller* tolerante a falhas, de maneira a tornar a aplicação como um todo tolerante a falhas. Para tanto, poderia ser adotada uma estratégia de replicação (ativa ou passiva) no mestre do *cluster*. A escolha entre a replicação ativa ou passiva depende da análise de *overhead* de cada uma dessas alternativas no contexto do *FT-CUDAlign*.

Outra possibilidade de trabalho futuro consiste em armazenar algumas linhas durante a execução do *FT-CUDAlign*, assim como é feito com as linhas especiais das versões anteriores do *MASA-CUDAlign*, de maneira a viabilizar menor retorno da computação por meio da recuperação horizontal. Com isso, é possível ter *mini-checkpoints* no interior de cada iteração e a recuperação seria feita em dois momentos: (a) global, como é feita atualmente; e (b) local, avançando a execução em cada GPU até o último *mini-checkpoint* válido. Como cada GPU teria *mini-checkpoints* em linhas diferentes da matriz, devido ao processamento *wavefront*, é necessário um protocolo que garanta o retorno a um *checkpoint* válido para todos os *mini-checkpoints* de todas as GPUs.

Uma terceira ideia de trabalho futuro consiste em adiantar a execução da iteração seguinte a partir das GPUs que já finalizaram a execução da iteração atual. Para tanto, iniciaria-se outro processo *CUDAlign* na GPU que já finalizou sua execução, que daria

início ao processamento da iteração seguinte. Nesse caso, haveria GPUs processando a iteração i e GPUs processando a iteração $i+1$. Ambos os processamentos coexistiriam até que a execução da iteração i fosse finalizada, após a etapa de sincronização de término da execução da iteração. Com esse modelo de execução, o *MASA-CUDAlign-MultiBP* avançaria mais rápido, porém a recuperação de falhas se tornaria bem mais complexa.

Referências

[1] Tröguer, Peter: *Dependable systems: Definitions and metrics*, 2011. Dependable Systems Course. viii, 15

[2] Elnozahy, Elmootazbellah Nabil, Lorenzo Alvisi, Yi Min Wang e David B Johnson: *A survey of rollback-recovery protocols in message-passing systems*. ACM Computing Surveys (CSUR), 34(3):375–408, 2002. viii, 18, 19

[3] Sandes, Edans Flavius de O e Alba Cristina MA de Melo: *Smith-waterman alignment of huge sequences with gpu in linear space*. Em *2011 IEEE International Parallel & Distributed Processing Symposium*, páginas 1199–1211. IEEE, 2011. viii, 23, 28, 29, 30

[4] Sandes, Edans FO, Guillermo Miranda, ACMA Melo, Xavier Martorell e Eduard Ayguade: *Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters. 14th ieee*. Em *ACM International Symposium on Cluster, Cloud and Grid Computing*, volume 4, páginas 62–63, 2014. viii, 2, 23, 31, 32

[5] Oliveira Sandes, Edans Flavius de, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro e Alba Cristina Magalhaes Melo: *Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters*. IEEE Transactions on Parallel and Distributed Systems, 27(10):2838–2850, 2016. viii, 2, 23, 32, 34

[6] Figueirêdo Júnior, Marco Antônio Caldas de: *Comparação paralela de sequências biológicas em múltiplas gpus com descarte de blocos e estratégias de distribuição de carga*. 2021. viii, 33, 36

[7] Luscombe, Nicholas M, Dov Greenbaum e Mark Gerstein: *What is bioinformatics? a proposed definition and overview of the field*. Methods of information in medicine, 40(04):346–358, 2001. 1, 4

[8] Durbin, Richard, Sean R Eddy, Anders Krogh e Graeme Mitchison: *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998. 1, 4, 5

[9] Sandes, Edans Flavius De Oliveira, Azzedine Boukerche e Alba Cristina Magalhaes Alves De Melo: *Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification*. 48:63:1–63:36, 2016. 1

[10] Ino, Fumihiko, Yuma Munekawa e Kenichi Hagihara: *Sequence homology search using fine grained cycle sharing of idle gpus*. IEEE Transactions on Parallel and Distributed Systems, 23(4):751–759, 2011. 2

[11] Liu, Yongchao, Adrianto Wirawan e Bertil Schmidt: *Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions*. BMC bioinformatics, 14:1–10, 2013. 2

[12] Figueiredo, Marco, Joao Paulo Navarro, Edans FO Sandes, George Teodoro e Alba CMA Melo: *Parallel fine-grained comparison of long dna sequences in homogeneous and heterogeneous gpu platforms with pruning*. IEEE Transactions on Parallel and Distributed Systems, 32(12):3053–3065, 2021. 2, 52

[13] Sousa, Walisson P, Filipe M Soares, Rafaela C Brum, Marco Figueiredo, Alba CMA Melo, Maria Clicia S de Castro e Cristiana Bentes: *Biological sequence comparison on cloud-based gpu environment*. Em *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*, páginas 239–263. Springer, 2023. 2, 60

[14] Needleman, Saul B e Christian D Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. Journal of molecular biology, 48(3):443–453, 1970. 6, 8

[15] Smith, Temple F, Michael S Waterman *et al.*: *Identification of common molecular subsequences*. Journal of molecular biology, 147(1):195–197, 1981. 8

[16] Gotoh, Osamu: *An improved algorithm for matching biological sequences*. Journal of molecular biology, 162(3):705–708, 1982. 8

[17] Myers, Eugene W e Webb Miller: *Optimal alignments in linear space*. Bioinformatics, 4(1):11–17, 1988. 10

[18] Hirschberg, Daniel S.: *A linear space algorithm for computing maximal common subsequences*. Communications of the ACM, 18(6):341–343, 1975. 10

[19] Egwutuoha, Ifeanyi P, David Levy, Bran Selic e Shiping Chen: *A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems*. The Journal of Supercomputing, 65:1302–1326, 2013. 13, 14, 16

[20] IEEE: *Ieee standard classification for software anomalies*. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), páginas 1–23, 2010. 13

[21] Nelson, Victor P.: *Fault-tolerant computing: Fundamental concepts*. Computer, 23(7):19–25, 1990. 13, 14

[22] Mukwevho, Mukosi Abraham e Turgay Celik: *Toward a smart cloud: A review of fault-tolerance methods in cloud systems*. IEEE Transactions on Services Computing, 14(2):589–605, 2018. 15, 16

[23] Huang, Yennun, Chandra Kintala, Nick Kolettis e N Dudley Fulton: *Software rejuvenation: Analysis, module and applications*. Em *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers*, páginas 381–390. IEEE, 1995. 15

[24] Psaier, Harald e Schahram Dustdar: *A survey on self-healing systems: approaches and systems*. Computing, 91:43–73, 2011. 16

[25] Ledmi, Abdeldjalil, Hakim Bendjenna e Sofiane Mounine Hemam: *Fault tolerance in distributed systems: A survey*. Em *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, páginas 1–5. IEEE, 2018. 17

[26] Felber, Pascal e André Schiper: *Optimistic active replication*. Em *Proceedings 21st International Conference on Distributed Computing Systems*, páginas 333–341. IEEE, 2001. 17

[27] Maloney, Andrew e Andrzej Goscinski: *A survey and review of the current state of rollback-recovery for cluster systems*. Concurrency and Computation: Practice and Experience, 21(12):1632–1666, 2009. 17, 19, 20, 21, 22

[28] Paul, Sri Raj, Akihiro Hayashi, Matthew Whitlock, Seonmyeong Bak, Keita Teranishi, Jackson Mayo, Max Grossman e Vivek Sarkar: *Integrating inter-node communication with a resilient asynchronous many-task runtime system*. Em *2020 Workshop on Exascale MPI (ExaMPI)*, páginas 41–51. IEEE, 2020. 19, 40, 47

[29] Sandes, Edans Flavius O e Alba Cristina MA de Melo: *Cudalign: using gpu to accelerate the comparison of megabase genomic sequences*. Em *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, páginas 137–146, 2010. 23

[30] Paul, Sri Raj, Akihiro Hayashi, Nicole Slattengren, Hemanth Kolla, Matthew Whitlock, Seonmyeong Bak, Keita Teranishi, Jackson Mayo e Vivek Sarkar: *Enabling resilience in asynchronous many-task programming models*. Em *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25*, páginas 346–360. Springer, 2019. 39, 40

[31] Grossman, Max, Vivek Kumar, Nick Vrvilo, Zoran Budimlic e Vivek Sarkar: *A pluggable framework for composable hpc scheduling libraries*. Em *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, páginas 723–732. IEEE, 2017. 39

[32] Bland, Wesley, Aurelien Bouteiller, Thomas Herault, George Bosilca e Jack Dongarra: *Post-failure recovery of mpi communication capability: Design and rationale*. The International Journal of High Performance Computing Applications, 27(3):244–254, 2013. 40, 42

[33] Teranishi, Keita, Marc Gamell, Rob Van der Wijngarrt e Manish Parashar: *Fenix a portable flexible fault tolerance programming framework for mpi applications*. Relatório Técnico, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018. 40

[34] Montezanti, Diego, Enzo Rucci, Armando De Giusti, Marcelo Naiouf, Dolores Rexachs e Emilio Luque: *Soft errors detection and automatic recovery based on replication combined with different levels of checkpointing*. Future Generation Computer Systems, 113:240–254, 2020. 41, 47

[35] Ansel, Jason, Kapil Arya e Gene Cooperman: *Dmtcp: Transparent checkpointing for cluster computations and the desktop*. Em *2009 IEEE International Symposium on Parallel Distributed Processing*, páginas 1–12, 2009. 41

[36] Hübner, Lukas, Alexey M Kozlov, Demian Hespe, Peter Sanders e Alexandros Stamatakis: *Exploring parallel mpi fault tolerance mechanisms for phylogenetic inference with raxml-ng*. Bioinformatics, 37(22):4056–4063, 2021. 42, 47

[37] Brum, Rafaela C, Walisson P Sousa, Alba CMA Melo, Cristiana Bentes, Maria Clicia S de Castro e Lúcia Maria de A Drummond: *A fault tolerant and deadline constrained sequence alignment application on cloud-based spot gpu instances*. Em *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*, páginas 317–333. Springer, 2021. 44, 47

[38] (AWS), Amazon Web Service: *Ec2 on demand pricing*. https://aws.amazon.com/pt/ec2/pricing/on-demand/, Acesso em 10/02/2025. 45

[39] (AWS), Amazon Web Service: *Ec2 spot pricing*. https://aws.amazon.com/pt/ec2/spot/pricing/, Acesso em 10/02/2025. 45

[40] Mudassar, Muhammad, Yanlong Zhai e Liao Lejian: *Adaptive fault-tolerant strategy for latency-aware iot application executing in edge computing environment*. IEEE Internet of Things Journal, 9(15):13250–13262, 2022. 45, 47

[41] Tanenbaum, Andrew S e Herbert Bos: *Modern operating systems*. Pearson Education, Inc., 2015. 55

[42] NCBI: *Ncbi main page*. https://www.ncbi.nlm.nih.gov/, Acesso em 01/02/2024. 59

[43] AWS: *Aws g5 instance specifications*. https://aws.amazon.com/pt/ec2/instance-types/g5/, Acesso em 01/02/2024. 60

[44] AWS: *Aws parallelcluster user guide*. https://docs.aws.amazon.com/pdfs/parallelcluster/latest/ug/aws-parallelcluster-ug.pdf, Acesso em 22/08/2024. 60