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

Disk-Assited Parallel A*

Estratégia de Movimentação de Dados Memória-Disco para o Alinhamento Múltiplo Ótimo de Sequências

Cainã F. B. Razzolini

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifácio

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora) — CIC/UnB
Prof.ª Dr.ª Maria Emília Machado Telles Walter — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB

CIP — Catalogação Internacional na Publicação

Razzolini, Cainã F. B..

Disk-Assited Parallel A*: Estratégia de Movimentação de Dados Memória-Disco para o Alinhamento Múltiplo Ótimo de Sequências / Cainã F. B. Razzolini. Brasília : UnB, 2016.
67 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. alinhamento múltiplo de sequências, 2. A* paralelo,
3. gerenciamento de Memória-Disco

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

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

Disk-Assited Parallel A*

Estratégia de Movimentação de Dados Memória-Disco para o Alinhamento Múltiplo Ótimo de Sequências

Cainã F. B. Razzolini

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof.ª Dr.ª Maria Emília Machado Telles Walter
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Rodrigo Bonifácio
Coordenador do Bacharelado em Ciência da Computação

Brasília, 11 de julho de 2016

Dedicatória

Agradecimentos

Resumo

O alinhamento múltiplo de sequências (MSA) é uma operação muito frequente em Bioinformática, sendo executada dezenas de milhares de vezes por dia em todo o mundo. Várias estratégias paralelas foram propostas para acelerar a produção de resultados do MSA com A*, dentre elas o *Parallel* A^* que, apesar de obter resultados mais rápido, requer grande quantidade de memória, o que inviabiliza o uso desse tipo de algoritmo para conjuntos com um número razoável de sequências com padrões complexos. O presente trabalho de graduação propõe e avalia uma estratégia de movimentação de dados entre memória e disco para o *Parallel* A^* com o intuito de permitir a solução de instâncias do MSA que exijam mais memória que o disponível no ambiente de execução. Os resultados experimentais obtidos em 2 ambientes de testes mostram que a estratégia proposta permitiu a solução de instâncias do MSA que não eram finalizadas nos ambientes de testes por uso excessivo de memória, contudo, com um incremento considerável no tempo de execução.

Palavras-chave: alinhamento múltiplo de sequências, A* paralelo, gerenciamento de Memória-Disco

Abstract

Multiple Sequence Alignment (MSA) is a common operation in Bioinformatics, executed tens of thousands of times daily all over the world. Many parallel strategies were proposed for accelerating the production of results of the *Multiple Sequence Alignment* (MSA) with A*, amongst them the *Parallel* A^* (PA*) which, despite obtaining results faster, demands a great amount of memory, becoming unfeasible for some MSA instances. This undergraduate project proposes and evaluates a strategy of data movement between memory and disk for the *Parallel* A^*, in order to allow the solution of MSA instances which demand more memory than the available on the execution environment. Experimental results obtained in 2 test environments have showed that the strategy allowed the solution of MSA instances that could not be completed on the environments used due to the excessive use of memory. This was achieved, however, with a considerable increment on the execution time.

Keywords: multiple sequence alignment, Parallel A*, Disk-Memory management

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

BFS *Best-first Search.* 14, 17

DAPA* *Disk-Assisted Parallel A^*.* ix, x, xii, 38–40, 45, 46, 50–52

DDD *Delayed Duplicate Detection.* viii, 17–19, 21

DFBnB *Depth-first branch-and-bound.* 17

DFID *Depth-first iterativedeepening.* 17

DFS *Depth-first Search.* 17

FA* *Frontier A^*.* 21

FA*-DDD *Frontier A^* with Delayed Duplicate Detection.* 21, 24, 29

FPGA *Field Programmable Gate Array.* 28

GPU *Graphics Processecing Unit.* 28

HBDDD *Hash-based Delayed Duplicate Detection.* 19

IDA* *Iterative-deepening-A^*.* 17

MIMD *Multiple Intructions Multiple Data.* 27

MPI *Message Passing Interface.* 28

MSA *Multiple Sequence Alignment.* vi, vii, 2, 9

PA* *Parallel A^*.* vi–ix, xii, 2, 3, 29–31, 33, 34, 37, 38, 45, 46, 50, 52

PE2A* *Parallel External Partial Expansion A^*.* viii, 19–21

PEA* *Partial Expansion A^*.* 19

PFA*-DDD *Parallel Frontier A^* with Delayed Duplicate Detection.* viii, 21, 24, 26

SIMD *Single Instruction Multiple Data.* 27

SMP *Symmetric Multiprocessors.* 27, 28

SP *Sum-of-Pairs.* viii, x, 6, 7, 9, 10, 31

STL *Standard Template Library.* 30

WSP *Weighted Sum-of-Pairs.* viii, 7, 9

Capítulo 1

Introdução

A Bioinformática é uma área multidisciplinar que visa o desenvolvimento de ferramentas e algoritmos para auxiliar os biólogos na análise de dados, com objetivo de determinar a função ou estrutura das sequências biológicas e inferir informações sobre sua evolução nos organismos.

A evolução das técnicas de sequenciamento genômico gerou um crescimento quase exponencial no número de projetos desse tipo e um crescimento exponencial na quantidade de dados nos bancos de dados genômicos [18]. Dado esse panorama, existe uma quantidade significativa de sequências que devem ser comparadas entre si, o que gera a necessidade de ferramentas cada vez mais rápidas e precisa.

A comparação de sequências biológicas pode ser feita entre duas sequências, chamado alinhamento entre pares, ou mais de duas, chamado alinhamento múltiplo (MSA). O uso de mais do que duas sequências pode facilitar a identificação de semelhanças e diferenças entre o conjunto de sequências como um todo, podendo tornar essas claras ou até óbvias [26]. Um alinhamento múltiplo pode também fornecer a biólogos uma visão de como a evolução atuou sobre organismos.

Obter o alinhamento múltiplo ótimo de um conjunto de sequências é um problema de alta complexidade computacional, que foi provado NP-Difícil [39]. Por isso, muitas soluções heurísticas para o problema do alinhamento múltiplo foram propostas. No entanto, as soluções heurísticas não garantem que o alinhamento múltiplo ótimo das sequências seja obtido, retornando geralmente um resultado aproximado.

Para a obtenção do alinhamento múltiplo ótimo, existe um algoritmo simples que é uma generalização do algoritmo ótimo de programação dinâmica para obtenção do alinhamento par a par [26]. No entanto, esse algoritmo possui complexidade exponencial, o que impossibilita o seu uso para um número relativamente pequeno que sequências.

O espaço de busca do alinhamento múltiplo de sequências pode ser modelado como um grafo, dessa forma, tornando-o equivalente ao problema de encontrar o caminho de

menor custo nesse grafo [26]. O algoritmo mais amplamente conhecido para busca do caminho de menor custo é chamado de A* [32].

Este algoritmo avalia os nós através do cálculo do custo até chegar ao nó e uma estimativa do custo do caminho envolvendo o nó em questão até o nó final. O A * foi amplamente utilizado para a solução do problema do alinhamento múltiplo ótimo de sequências e, atualmente, os resultados mais rápidos são obtidos com a utilização de alguma variante dele.

1.1 Motivação

Sundfeld at al. [35] propuseram uma implementação do algoritmo *Parallel A** em máquinas *multicore* para solução exata do alinhamento múltiplo de sequências, conseguindo *speedups* chegando até 4.77× em relação a algumas soluções no estado da arte [27]. Contudo, a solução proposta possui um grande limitante no consumo de memória.

1.2 Objetivo

O presente trabalho de graduação tem por objetivo propor, implementar e avaliar uma estratégia de movimentação de dados entre memória e disco para o *Parallel A** proposto [35], que permita a solução de instâncias do MSA com sequências maiores ou com mais sequências que as originalmente possíveis.

Para tanto, foi introduzido o conceito de região, uma estrutura contendo todas as estruturas de dado por onde um nó é armazenado durante a execução da estratégia. Uma região trata um subconjunto dos nós do espaço de busca e cada *thread* é responsável por várias regiões, sendo que, a região de maior prioridade de uma *thread* é denominada como região ativa. As regiões não ativa são candidatas a serem transferidas da memória RAM para o disco.

Adicionalmente, foi projetado um modelo de movimentação de dados memória-disco, visando possibilitar a execução de instâncias do problema do alinhamento múltiplo de sequências que o algoritmo usado como base desse projeto não conseguia finalizar por requerer muita memória.

1.3 Estrutura do Documento

O presente documento está organizado da seguinte maneira. O Capítulo 2 apresenta o problema do alinhamento múltiplo de sequências biológicas, o algoritmo exato simples

para sua resolução e uma técnica de limitação do espaço de busca do problema. O Capítulo 3 apresenta o algoritmo A*, um algoritmo de busca de caminho com menor custo utilizado para resolver o problema do alinhamento múltiplo de sequências e algumas variações do A* que obtém um melhor desempenho. O Capítulo 4 apresenta a implementação do *Parallel A**, uma implementação do A* utilizando múltiplas *threads* e que serviu de base para este trabalho. O Capítulo 5 descreve a estratégia proposta para permitir a solução de instâncias mais complexas do que as que o *Parallel A** é capaz de resolver. O Capítulo 6 apresenta os resultados experimentais. Finalmente, o Capítulo 7 apresenta as conclusões e os trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

A comparação de sequências biológicas é um problema fundamental da Bioinformática, que visa organizar as sequências de maneira a explicitar regiões de similaridade entre elas. Essas similaridades são importantes pois permitem transferir informações de estrutura e/ou funcionalidade de uma sequência cujos atributos já são conhecidos para as outras [9].

Durante o processo de evolução, sequências acumulam inserções, deleções e substituições, logo, uma comparação deve primeiramente encontrar uma configuração para as sequências de forma a revelar ao máximo as similaridades e diferenças entre elas, a que chamamos Alinhamento.

Métodos de comparação de sequências geram um alinhamento para as sequências e um escore como resultado, escore este que é utilizado para decidir quão boa é a solução. Um alinhamento pode ser feito entre apenas duas sequências, definido como um alinhamento em pares, ou entre mais que duas, chamando então de alinhamento múltiplo.

2.1 Sequências

Sequências biológicas são cadeias ordenadas de substâncias químicas [9]. RNA, DNA e proteínas são os tipos de sequências exploradas em Bioinformática. DNA e RNA são formados por nucleotídeos, um tipo de molécula orgânica, que são representadas pelos alfabetos $\Sigma = \{A, T, C, G\}$ e $\Sigma = \{A, U, C, G\}$, respectivamente. Proteínas são sequências de aminoácidos, um tipo de composto orgânico, definidas no alfabeto
$\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$.

Dado um alfabeto $\Sigma = \{a_1, ..., a_n\}$ de n caracteres, definimos uma sequência de k caracteres a_i como sendo um elemento do produto de caracteres do alfabeto, conforme a Equação 2.1.

$$a^k = \prod_{i=1}^{k} a_i, a_i \in \Sigma \tag{2.1}$$

2.2 Alinhamentos

Um conjunto S de sequências pode ser definido como: $S = (a_1, ..., a_n)$ onde a_i é uma sequência. Seja $\beta = \{-\} \cup \{a_1, ..., a_n\}$, o alfabeto obtido da união do alfabeto Σ com o símbolo representando um espaço em branco "$-$", chamado "gap". Pode-se então definir um alinhamento múltiplo de sequências para o conjunto $S = (S_1, ..., S_n)$ como sendo um elemento do conjunto $\bar{S} = (\bar{S_1} \times ... \times \bar{S_n})$ onde $\bar{S_i}$ é obtido inserindo *gaps* em cada posição aonde as demais sequências possuíam um caractere não branco [6], tal que $|\bar{S_1}| = |\bar{S_2}| \cdots = |\bar{S_n}|$. $\bar{S}$ é definido como o espaço de busca.

Um alinhamento representa um conjunto de inserções, deleções e substituições sobre as sequências originais. Os métodos de pontuação (*scoring*) são então responsáveis por definir qual entre os possíveis alinhamentos é o melhor. A Figura 2.1 ilustra um exemplo de alinhamento múltiplo entre 5 sequências de aminoácidos.

```
AALAPELLVD-AK-SS-----TTSIFFPSA
VKESRHFQIDYDE-EG-----NCSLTISEV
SPNQQRISVVWND-DD-----SSTLTIYNA
IAQFRKEKETFKEKDTYKLFKNGTLKIKHL
SPLNGK--VT-NE-GT-----TSTLTMNPV
```

Figura 2.1: Alinhamento Múltiplo de Sequências.

2.3 Caminhos

A um conjunto de N sequências $S_1, ..., S_n$, associamos um arranjo $L(S_1, ..., S_n)$ em espaço n-dimensional, que corresponde aos hipercubos n-dimensionais obtidos do produto cartesiano das n sequências. Esses cubos unidos formam um paralelepípedo n-dimensional, que representa o espaço de busca do problema. O vértice correspondente aos primeiros caracteres de todas as sequências é chamado origem e o vértice mais distante desse é chamado final [6].

Um caminho $\gamma(S_1, ..., S_n)$ entre as sequências $S_1, ..., S_n$ é uma linha "quebrada" conectada que liga o vértice origem ao final, diminuindo a sua distância até o vértice final a cada passo. Cada caminho γ corresponde a um possível alinhamento. A Figura 2.2 mostra um exemplo de um caminho e do alinhamento correspondente a ele.

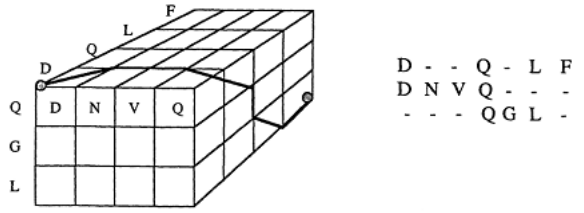

Figura 2.2: Alinhamento entre 3 sequências (Fonte: [6]).

A projeção de um caminho em um plano formado por um par de sequências S_i e S_j, denominado $p_{ij}(\gamma(S_1, ..., S_n))$, representa o alinhamento entre as sequências S_i e S_j [6], exemplificado pela Figura 2.3.

```
D - - Q L F
D N V Q - -
```

Figura 2.3: Alinhamento par-a-par projetado (Fonte: [6]).

2.4 Escores do MSA

Um dos aspectos mais importantes de um alinhamento múltiplo de sequências é a função de cálculo do escore de um alinhamento e não existe hoje consenso sobre qual a função mais indicada [9].

2.4.1 *Sum-of-Pairs* (SP)

Um método muito utilizado para calcular o escore de um alinhamento é o *Sum-of-Pairs* (SP). Com o SP, inicialmente calcula-se o valor do escore para cada par de alinhamentos, ou seja:

$$SP(m_i) = \sum_{k<l} S(m_i^k, m_i^l) \tag{2.2}$$

Onde é uma função que calcula o escore de uma posição dos alinhamentos e m_i^k e m_i^l são os caracteres dos alinhamentos k e l na posição i. Essa função pode calcular o escore com base em uma matriz de substituições, tais como PAM [7] e BLOSUM [16], formuladas com base em dados biológicos com relevância estatística, que atribuem um valor para cada combinação de *matches* e *mismatches*, ou com métodos mais simples, como atribuindo um valores únicos definidos *matches*, *mismatches* e *gaps*, assim como

ilustrado na Figura 2.4. A Figura 2.5 ilustra a matriz PAM250 para os 20 aminoácidos enconstrados na natureza.

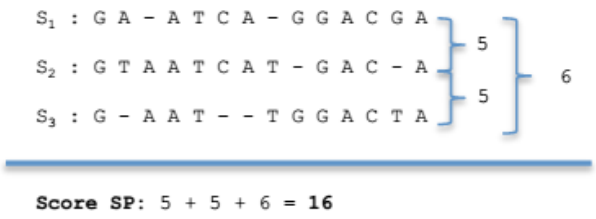

Figura 2.4: Exemplo do SP (Fonte: [35]).

	-	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
-	0	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
C	12	05	17	19	20	19	20	21	22	22	22	20	21	22	22	19	23	19	21	17	25
S	12	17	14	16	16	16	16	16	17	17	18	18	17	17	19	18	20	18	20	20	19
T	12	19	16	14	17	16	17	17	17	17	18	18	18	17	18	17	19	17	20	20	22
P	12	20	16	17	11	16	18	18	18	18	17	17	17	18	19	19	20	18	22	22	23
A	12	19	16	16	16	15	16	17	17	17	17	18	19	18	18	18	19	17	21	20	23
G	12	20	16	17	18	16	12	17	16	17	18	19	20	19	20	20	21	18	22	22	24
N	12	21	16	17	18	17	17	15	15	16	16	15	17	16	19	19	20	19	21	19	21
D	12	22	17	17	18	17	16	15	13	14	15	16	18	17	20	19	21	19	23	21	24
E	12	22	17	17	18	17	17	16	14	13	15	16	18	17	19	19	20	19	22	21	24
Q	12	22	18	18	17	17	18	16	15	15	13	14	16	16	18	19	19	19	22	21	22
H	12	20	18	18	17	18	19	15	16	16	14	16	15	17	19	19	19	19	19	17	20
R	12	21	17	18	17	19	20	17	18	18	16	15	11	14	17	19	20	19	21	21	15
K	12	22	17	17	18	18	19	16	17	17	16	17	14	12	17	19	20	19	22	21	20
M	12	22	19	18	19	18	20	19	20	19	18	19	17	17	11	15	13	15	17	19	21
I	12	19	18	17	19	18	20	19	19	19	19	19	19	19	15	12	15	13	16	18	22
L	12	23	20	19	20	19	21	20	21	20	19	19	20	20	13	15	11	15	15	18	19
V	12	19	18	17	18	17	18	19	19	19	19	19	19	19	15	13	15	13	18	19	23
F	12	21	20	20	22	21	22	21	23	22	22	19	21	22	17	16	15	18	8	10	17
Y	12	17	20	20	22	20	22	19	21	21	21	17	21	21	19	18	18	19	10	07	17
W	12	25	19	22	23	23	24	21	24	24	22	20	15	20	21	22	19	23	17	17	00

Figura 2.5: PAM250 (Fonte: [7]).

O SP assume que cada sequência descende das N-1 outras sequências ao invés de um ancestral comum, logo considera-se que ele não possui uma boa justificativa probabilística [9]. Isso acarreta na contagem múltipla de eventos evolucionários, agravando o problema com o crescimento do número de sequências consideradas. O modelo de *Weighted Sum-of-Pairs* [2] foi proposto para levar em consideração essas limitações.

2.4.2 *Weighted Sum-of-Pairs* (WSP)

Altschul, Carroll e Lipman [2] propuseram duas formas de raciocínio sobre a atribuição de pesos a pares de sequências. Para a primeira forma, dada uma árvore evolucionária que correlaciona as sequências, existe um único caminho que liga o par de folhas p, o qual definimos como o caminho p,que usa todas as arestas por onde passa.

Uma mutação e em uma aresta da árvore contribui para a distância entre todas os pares de sequências que usam e. De forma a contar todas as mutações igualmente, independente de em que aresta ocorrem, precisamos que a soma dos pesos de todos os pares usando a

aresta e seja a unidade. Essa condição implica em um conjunto de equações lineares, uma para cada aresta, que correlacionam os pesos ainda desconhecidos.

Seja a ordem de um nó na árvore definida como o número de arestas que se conectam àquele nó. Seja p p um caminho que passe por nós de ordem $a_1, \cdots, a_k$. Definimos o peso do par p conforme a Equação 2.3:

$$W_p = 1/\prod_{i=1}^{k}(a_1 - 1) \tag{2.3}$$

Por exemplo, se um caminho p passa por um nó de ordem 3 e um de ordem 4, o peso de p é de 1/6. Essa forma de definição de pesos no entanto possui a limitação de depender exclusivamente da topologia da árvore evolucionária relacionando as sequências, não considerando o comprimento das arestas. Portando, se o comprimento de uma aresta se aproxima de zero, o peso continua inalterado, mas se a aresta deixa de fazer parte, os pesos aumentam drasticamente. Esse problema pode ser visualizado na árvore evolucionária da Figura 2.6.

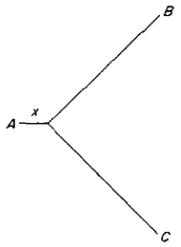

Figura 2.6: Árvore evolucionária (Fonte: [2]).

Intuitivamente, conforme o comprimento x da sequência A diminui, o melhor alinhamento será definido pelos alinhamentos $A-B$ e $A-C$, contudo para esse raciocínio todos os 3 pares possuem pesos iguais, independente do valor de x.

O segundo raciocínio [2] para definição de pesos considera que todos os pares de sequências são fontes de informação igualmente relevantes sobre a homologia entre as sequências. Se um par p é muito próximo, ele fornece pouca informação sobre as variações de caracteres em diferentes posições e mais informação sobre o alinhamento em si, contudo não se assume que as informações carregadas pelos pares são independentes.

Se caminhos p e q se sobrepõem pela maior parte de seus comprimentos, eles carregam quase a mesma informação. Por isso deve-se definir um conjunto de coeficientes de correlação entre os pares.

Sejam l_p, l_q os comprimentos dos caminhos p e q e l_{pq} o comprimento da sobreposição entre os caminhos, o coeficiente de correlação ρ_{pq} é dado pela Equação 2.4:

$$\rho_{pq} = l_{pq}^2 / l_p l_q \tag{2.4}$$

Com isso, define-se uma matriz M possuindo colunas e linhas indexadas pelos pares de sequências p e q e possuindo o valor de ρ_{pq}. Assim, como apenas a magnitude relativa interessa, os pesos podem ser calculados como sendo a soma das linhas da matriz M^{-1}. A estrutura dessa matriz permite que os pesos sejam calculados em tempo linear [1].

Este raciocínio gera um resultado mais intuitivo que o primeiro. No caso apresentado pela Figura 2.6, conforme o comprimento x se aproxima de zero, os pesos dos alinhamentos $A - B$ e $A - C$ tendem à unidade e o peso do alinhamento $B - C$ tende a zero.

Weights	Pairs	α,*m*,*L*	α,α,*m*,*L*	α,α′,*m*,*L*	α,*m*,*m*′,*L*
Unweighted	α, *m*	11	5	5	1
	α, *L*	22	0	0	37
	m, *L*	11	44	44	8
Rationale 1	α, *m*	11	11	11	11
	α, *L*	22	22	9	9
	m, *L*	11	11	25	25
Rationale 2	α, *m*	11	11	5	5
	α, *L*	22	22	9	9
	m, *L*	11	11	31	31

Symbol	Protein	NBRF code
α	Human hemoglobin α	HAHU
m	Human myoglobin	MYHU
L	Broad bean leghemoglobin	GPVF
α′	Bovine hemoglobin α	HABO
m′	Bovine myoglobin	MYBO

Figura 2.7: Comparação dos escores obtidos pelos 3 métodos (Fonte: [2]).

A Figura 2.7 mostra uma comparação entre os resultados de alinhamentos múltiplos gerados pela ferramenta MSA [23] utilizando SP, WSP raciocínio 1 e WSP raciocínio 2. A árvore filogenética necessária para o WSP foi estimada usando método de *neighbor joining* [33]. Sendo assim, podemos calcular o score $sc(A)$ do alinhamento múltiplo A como sendo:

$$sc(A) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} w_{i,j} \cdot c(A_{i,j}) \tag{2.5}$$

Onde $w_{i,j}$ e $c(A_{i,j})$ são respectivamente, o peso e escore do alinhamento par a par das sequências i e j.

2.5 Algoritmo exato simples

O problema do alinhamento múltiplo de sequências pode ser resolvido por um algoritmo exato conforme mostrado a seguir para um alinhamento de 3 sequências, podendo ser expandido para o número de sequências desejado.

Sejam S_1, S_2 e S_3 sequências de tamanhos n_1, n_2 e n_3, respectivamente, e $D(i, j, k)$ o escore do alinhamento ótimo usando SP para as sequências $S_1[1, .., i]$, $S_2[1, .., j]$, $S_3[1, .., k]$. Considere que o escore para *match*, *mismatch* e *gap* são *smatch*, *smsis* e *sspace*, respectivamente. O algoritmo para calcular o alinhamento múltiplo é mostrado no Algoritmo 1.

Para obter o alinhamento ótimo das três sequências , um cubo é utilizado, tal que cada célula (i, j, k) que não se encontra na fronteira, ou seja, $i \neq 0$,$j \neq 0$ e $k \neq 0$, possui sete vizinhos que devem ser considerados para determinar $D(i, j, k)$.

Para as fronteiras, considere $D'_{1,2}(i, j)$ como sendo o valor de SP para as subsequências $S_1[1, .., i]$ e $S_2[1, .., j]$. Definimos $D'_{1,3}(i, j)$ e $D'_{2,3}(i, j)$ analogamente [13] e chegamos a:

$$
\begin{aligned}
D(i, j, 0) &= D'_{1,2}(i, j) + (i + j) + sspace \\
D(i, 0, k) &= D'_{1,3}(i, k) + (i + k) + sspace \\
D(0, j, k) &= D'_{2,3}(j, k) + (j + K) + sspace \\
D(0, 0, 0) &= 0
\end{aligned}
$$

O algoritmo assume que a linha 0 e a coluna 0 são inicializadas com 0. Para cada sequência um laço é adicionado, nesse caso possuindo 3 laços. Uma vez dentro do laço, inicia-se o cálculo pelos *matches* e *mismatches*, linhas 4 a 20, e em seguida são consultadas as 7 células vizinhas, linhas 22 a 28. Ao final, $D(i, j, k)$ recebe o menor desses valores.

Considerando k sequências de tamanho n, o algoritmo constrói um matriz N-dimensional e exige um tempo de $\Theta(n^k)$ [13]. Esse problema foi provado ser NP-Difícil [39].

2.6 Método Carrilo-Lipman

Carrilo e Lipman [6] mostraram a possibilidade de se limitar o espaço de busca onde se encontra o alinhamento ótimo usando limites que permitem que algumas células da matriz de programação dinâmica computadas com o algoritmo não sejam calculadas. Isso foi feito utilizando-se o alinhamento exato dos pares de sequências e um escore gerado por um método de alinhamento múltiplo heurístico, que não garante a obtenção de um alinhamento ótimo, para definir um limite superior para o alinhamento [26].

Algoritmo 1 Algoritmo de Alinhamento Exato Simples

```
1: for i = 1 → n1 do
2:     for j = 1 → n2 do
3:         for k = 1 → n3 do
4:             if S1(i) = S2(j) then
5:                 cij = smatch
6:             else
7:                 cij = smis
8:             end if
9:             if S1(i) = S3(j) then
10:                cik = smatch
11:            else
12:                cik = smis
13:            end if
14:            if S2(i) = S3(j) then
15:                cjk = smatch
16:            else
17:                cjk = smis
18:            end if
19:            d1 = D(i − 1, j − 1, k − 1) + cij + cik + cjk
20:            d2 = D(i − 1, j − 1, k) + cij + sspace
21:            d3 = D(i − 1, j, k − 1) + cik + sspace
22:            d4 = D(i, j − 1, k − 1) + cjk + sspace
23:            d5 = D(i − 1, j, k) + 2 ∗ sspace
24:            d6 = D(i, j − 1, k) + 2 ∗ sspace
25:            d7 = D(i, j, k − 1) + 2 ∗ sspace
26:
27:            D(i, j, k) = Min[d1, d2, d3, d4, d5, d6, d7]
28:        end for
29:    end for
30: end for
```

2.6.1 Limites em alinhamentos múltiplos

Segundo Carrilo e Lipman [6], algumas observações podem ser feitas sobre o problema de determinar o alinhamento ótimo $\gamma^*(S_1, ..., S_n), n > 2$, que acarretam na diminuição do número de computações necessárias para a obtenção da solução ótima.

Seja μ_{ij} uma uma medida de escore de um par de sequências S_i e S_j, então a medida M de um caminho N-dimensional γ é dada pela Equação 2.6:

$$M(\gamma) = \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma)) \tag{2.6}$$

Onde p_{ij} é a projeção de γ no plano determinado pelas sequências S_i e S_j. Considere também um caminho N-dimensional γ^e, que será chamando γ-estimado. Dado que γ^* é um caminho ótimo, tem-se as Inequações 2.7 a 2.11:

$$M(\gamma^e) - M(\gamma^*) >= 0 \tag{2.7}$$

$$\sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^*)) >= 0 \tag{2.8}$$

Denota-se γ_{ij}^* como sendo o caminho ótimo para a medida μ_{ij} das sequências S_i e S_j. Dado $\mu_{ij}(p_{ij}(\gamma^*)) < \mu_{ij}(\gamma_{ij}^*)$, temos que a Inequação 2.9:

$$\sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{\substack{i<j \\ (i,j)\neq(k,l)}}^{N} \mu_{ij}(\gamma^*) >= \mu_{kl}(p_{kl}(\gamma^*)) \tag{2.9}$$

Assim define-se U_{kl} conforme a Equação 2.10:

$$\sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{\substack{i<j \\ (i,j)\neq(k,l)}}^{N} \mu_{ij}(\gamma^*) \tag{2.10}$$

Sendo U_{kl} um limite superior para a medida das projeções de qualquer caminho ótimo N-dimensional nos planos definidos pelas sequências S_k e S_l. Logo, para encontrar um alinhamento ótimo γ^* consideram-se apenas os caminhos γ em $L(S_1, ..., S_n)$ que satisfaçam $\mu_{kl}(p_{kl}(\gamma)) = U_{kl}$. Dado isso, define-se X como o conjunto que contém os caminhos γ que satisfazem a propriedade descrita pela Equação 2.11:

$$X = \bigcap_{i<j}^{n} X_{ij} \tag{2.11}$$

Assim, para encontrar um alinhamento ótimo γ^* é suficiente considerar apenas os caminhos em X, o que significa aplicar o algoritmo de programação dinâmica apenas em uma sub-região Y de $L(S_1, ..., S_n)$.

2.6.2 Simplificação da fórmulas de Carrilo-Lipman

Trabalhos posteriores simplificaram as equações do método Carrilo-Lipman [12] [24], conforme mostrado abaixo.

Seja L o limite inferior da soma de pares [12]:

$$L = \sum_{i<j} d(S_i, S_j) \cdot scale(S_i, S_j) \tag{2.12}$$

Onde $scale(S_i, S_j)$ é o peso atribuído às sequências S_i e S_j e $d(S_i, S_j)$ é o custo do alinhamento ótimo em par de sequências. Dado que L possui o valor do escore para o alinhamento ótimo, não existe outro alinhamento das sequências S_i e S_j que resulte em um escore inferior a L, logo L limita inferiormente o escore dos alinhamentos.

O escore do alinhamento múltiplo ótimo é obtido pela soma dos escores do alinhamento de pares de sequências do alinhamento múltiplo. Sendo assim, o escore do alinhamento múltiplo ótimo é menor ou igual à soma dos escores dos alinhamentos ótimos de pares.

Seja U um limite superior, A^o o alinhamento múltiplo ótimo, $c(A^0)$ o custo desse alinhamento e $c(A_i^0, A_j^o)$) o custo do par de sequências i e j do alinhamento A^o. Podemos então definir a inequação:

$$U - L >= \sum_{i<j} [scale(S_i, S_j) \cdot (c(A_{i,j}^o) - d(S_i, S_j))] \tag{2.13}$$

Logo, para quaisquer par de sequências p e q:

$$U - L >= \sum_{i<j} [scale(S_i, S_j) \cdot (c(A_{i,j}^o) - d(S_i, S_j))] >= [scale(S_p, S_q) \cdot (c(A_{p,q}^o) - d(S_p, S_q)) \tag{2.14}$$

Assim, rearranjando a inequação obtemos:

$$scale(S_i, S_j) \cdot c(A_{i,j}^o) <= scale(S_i, S_j) \cdot d(S_i, S_j) + U - L \tag{2.15}$$

Consideremos agora um alinhamento heurístico A^h. Temos que $c(A^o) <= c(A^h)$, logo $c(A^h)$ é um limite superior para o valor do escore do alinhamento ótimo, então $U - L$ pode ser obtido por:

$$U - L = scale(S_i, S_j) \sum_{i<j} [c(A_{i,j}^h) - d(S_i, S_j)] \tag{2.16}$$

Capítulo 3

Algoritmo A*

3.1 Algoritmo A*

O A* [14] é um algoritmo de busca de caminho do tipo *Best-first Search* (BFS) cujo resultado é o caminho de menor custo entre um vértice inicial N_i e um vértice final N_f,utilizando heurísticas para limitar o espaço de busca. Para o funcionamento do A*, é necessária uma função de custo que defina a ordem na qual os nós deverão ser visitados para encontrar o caminho ótimo.

A Equação 3.1 mostra essa função, onde $g(N)$ é o custo da origem até o nó N, $h(N)$ é o custo estimado do nó N até o nó de destino e $f(N)$ é o custo estimado do caminho ótimo entre o nó inicial e o nó final que passa por N.

$$f(N) = g(N) + h(N) \tag{3.1}$$

O algoritmo faz uso de duas listas, uma para nós em aberto, que serão expandidos, e outra para nós fechados, que já o foram. As listas são inicializadas com o nó de origem e vazia, respectivamente.

O primeiro passo do algoritmo é **find**, no qual se encontra o nó com o menor valor para a função de custo f na lista de abertos denominado N_k.

Em seguida, vem o passo **stop** no qual se testa se o N_k é o nó final ou se ele não existe,ou seja, a lista de nós abertos está vazia. Se algum dos testes for positivo, o algoritmo é encerrado. Caso contrário, segue-se para a próxima etapa.

Segue o passo de **expand and reconcile**, onde o nó N_k é retirado da lista de abertos e colocado na lista de fechados e todos os vizinhos de N_k, que não estejam na lista de fechados ou cujo o valor de $g(n)$ seja menor do que quando adicionado na lista de fechados, são adicionados à lista de abertos. O algoritmo então retorna ao passo de **find**.

Quando uma das condições de parada é cumprida o programa finaliza. Caso a lista de abertos esteja vazia, conclui-se que não existe um caminho entre os vértices inicial e final. Caso contrário, o nó na lista de abertos com o menor valor para f, $n_e \in N_f$, possui um valor de $g(N_k)$ igual ao custo do caminho ótimo de N_i a N_f. Uma função de *backtrack* sobre a lista de fechados pode retornar o caminho ótimo obtido.

O Algoritmo 2 a seguir apresenta o pseudocódigo para o A*.

Algoritmo 2 A-Star(N_i,N_f)

```
1: OpenList ← N_i
2: while OpenList.lowest_f() ∉ N_f and OpenList ≠ ∅ do
3:     current ← OpenList.get_lowest_f()
4:     if current.g <= ClosedList.find_g(current) then
5:         ClosedList ← ClosedList ∪ current
6:         for neighbor in neigh(current) do
7:             if neighbor.g < OpenList.find_g(neighbor) and
8:                             neighbor.g < ClosedList.find_g(neighbor) then
9:                 OpenList ← OpenList ∪ neighbor
10:            end if
11:        end for
12:    end if
13: end while
14: if OpenList = ∅ then
15:    n_e ← NULL
16: else
17:    n_e ← OpenList.get_lowest_f()
18: end if
19: return n_e
```

O exemplo mais clássico de utilização do algoritmo A* é na determinação da menor distância por estrada entre duas cidades, conhecendo-se a distância terrestre entre cada cidade e suas vizinhas e a distância aérea de todas as cidades à cidade alvo.

Como por exemplo considere o grafo representado pela Figura 3.1 como sendo um mapa da Romênia [32].

Para calcular a distância entre Arad e Bucharest utilizando o algoritmo A* usamos a distância terrestre entre as cidades como custo entre os nós e a distância aérea até Bucareste como valor de $h(N)$. A *OpenList* é inicializada com o nó de Arad e na fase de **expand and reconcile** recebe os nós de Sibiu, Timisoara e Zerind. Na próxima iteração, $F(N_{Sibiu}) = 140+253 = 393$, $F(N_{Zerind}) = 75+374 = 449$ e $F(N_{Timisoara}) = 118+329 = 447$, logo Sibiu é passada para a *ClosedList* e seus vizinhos, Fagaras, Rimnicu, Arad e Oradea, são adicionados a *OpenList*.

Na Figura 3.2, cada árvore representa uma iteração do algoritmo, sendo que as folhas das árvores representam os nós dentro da *OpenList* ao final daquela iteração. Quando o nó

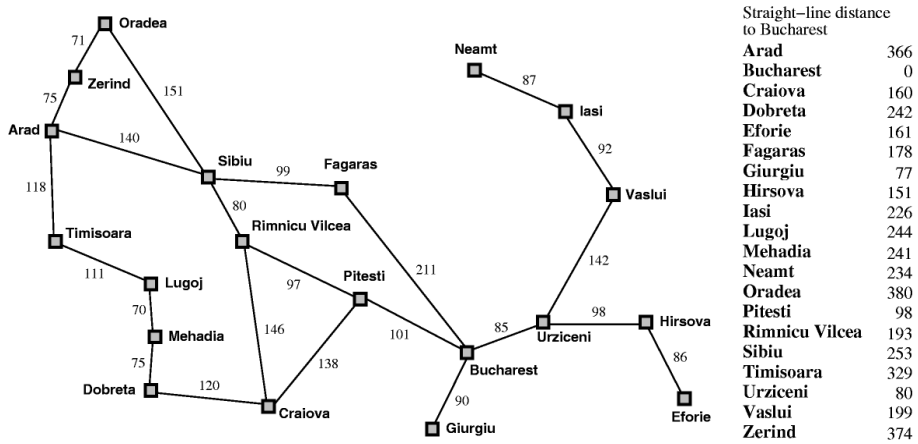

Figura 3.1: Mapa da Romênia (Fonte: [32]).

de Bucharest é adicionado, na próxima etapa de *find* o nó escolhido com certeza será ele, pois $h(N_{Bucareste}) = 0$. Com isso na etapa de **stop**, o algoritmo cumpre um dos requisitos de parada e encerra, nesse caso, retornando o valor de 449. Por fim, também é possível recuperar o caminho tomado para se conseguir esse resultado usando um algoritmo de *backtraking* na *ClosedList* se for necessário.

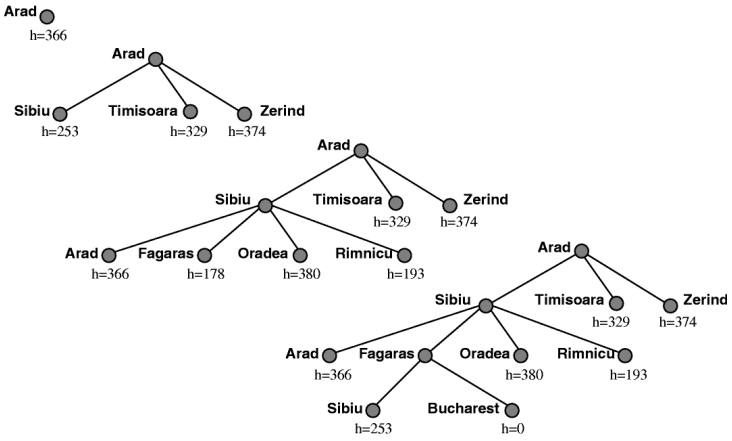

Figura 3.2: Iterações do A* (Fonte: [32]).

3.1.1 A* para o Alinhamento Múltiplo de Sequências

Conforme descrito no Capítulo 2, o espaço de busca do problema do alinhamento múltiplo de sequências pode ser representado por um paralelepípedo N-dimensional, e um alinhamento pode ser visto como um caminho entre a origem e o final. Um alinhamento ótimo é então o caminho de menor custo entres esses vértices. Assim, é possível encon-

trar o alinhamento múltiplo ótimo de sequências utilizando o A* para encontrar o melhor caminho.

Hart, Nilsson e Raphael [14] demonstraram que se $h(N)$ é um limite inferior qualquer, então A* é admissível, isto é, ele nunca superestima o custo para se alcançar o objetivo, e gera um resultado ótimo. Spouge [34] demonstrou que o limite inferior do método Carrillo-Lipman (Seção 2.6) é uma heurística admissível para o algoritmo A*. Esta heurística é também conhecida como $h_{2,all}$.

3.1.2 Limitação do A*

O algoritmo A*, assim como algoritmos do tipo BFS em geral, armazenam todos os estados que geram ou em uma lista de nós abertos ou fechados. Problemas onde a expansão de nós ocorre de maneira relativamente eficiente, como no caso do alinhamento múltiplo, costumam exaurir a memória disponível em minutos, abortando o algoritmo [21].

Em alguns casos, esse problema pode ser contornado utilizando algoritmos de *Depth-first Search* (DFS), tais como *Depth-first branch-and-bound* (DFBnB), *Depth-first iterati-vedeepening* (DFID) ou *Iterative-deepening-A** (IDA*) [19], que requerem uma quantidade de memória linear à profundidade da busca.

Contudo, como DFS não é capaz de detectar a maioria dos nós duplicados, pode gerar exponencialmente mais nós que uma BFS. Por exemplo, em um problema de espaço retangular em *grid*, *Breadth-first Search* gera $O(r^2)$ nós em uma distância de raio r, enquanto uma DFS gera $O(3^r)$ nós, já que cada nó possui quatro vizinhos, dos quais um é seu pai [21].

Dessa forma, para detectar todas as duplicatas, é preciso armazenar uma vasta quantidade de nós. Nesse intuito podem ser utilizados discos com centenas de gigabytes que custam menos de um dólar por gigabyte, chegando a ser cem vezes mais barato que memória. Entretanto, o disco não é um substituto direto de memória dado que a latência de disco é de aproximadamente 5 milissegundos, ou seja, cerca de 100.000 mil vezes maior que a latência de memória.

3.2 Adaptações do A*

3.2.1 *Delayed Duplicate Detection* (DDD)

Uma solução para o problema de espaço para BFS é a *Delayed Duplicate Detection* (DDD) [31]. Em vez de verificar a duplicidade de nós assim que um no é gerado, ele é simplesmente adicionando a lista de nós abertos. Ao final de um nível de busca, os nós na lista de abertos são ordenados por sua representação de estado, agrupando nós

duplicados em posições adjacentes e então eliminando-os. Então, é feito um *merge* da lista de nós abertos ordenada com a lista de nós fechados ordenada, eliminando os nós que também estão na lista de fechados da lista de nós abertos, e produzindo uma nova lista de nós fechados ordenada. A implementação [31] foi feita utilizando memória, contando com a memória virtual para gerenciar as ocorrências de *overflow* para o disco. Se o espaço do problema for significantemente maior que a capacidade da memória contudo, a complexidade assintótica de I/O desse algoritmo é de pelo menos o tamanho do espaço do problema vezes a profundidade da busca em largura.

A vantagem do DDD, contudo é permitir o armazenamento das listas de nós em disco. Seja uma lista de nós armazenada em um único arquivo. O arquivo é lido sequencialmente, expandindo-se nós com custo *bestf* e anexando-os no fim do arquivo. Quando o arquivo inteiro tiver sido lido, os nós adicionados durante a iteração corrente são inclusos e a lista é ordenada utilizando-se algoritmos de ordenação para arquivos em disco [11]. Finalmente, o arquivo ordenado é acessado sequencialmente, removendo-se nós duplicados e por fim a próxima iteração é iniciada.

Para implementar essa estratégia de maneira eficiente, inicia-se com uma lista ordenada em um arquivo de entrada, apenas com o nó inicial. Lê-se o arquivo sequencialmente, expandindo nós de melhor custo (*bestf*). Se um nó filho possui custo *bestf* ou menor, ele é expandido recursivamente, assim como o são seus filhos de custo igual ou menor e assim por diante. Todos os novos nós gerados são anexados a um *buffer* em memória. Quando o *buffer* estiver cheio, os nós são ordenados e as duplicatas são eliminadas, mantendo apenas a ocorrência com menor valor de g. Os nós remanescentes são então salvos em um arquivo de saída. Continua-se a ler o arquivo de entrada, gerando vários arquivos de saída, cada um ordenado e não possuindo duplicatas entre seus próprios nós, mas com duplicatas em outros arquivos de saída. Realiza-se então um *merge* entre todos os arquivos de saída e o arquivo de entrada em um único arquivo sem duplicatas usando um *multi-way merge* [21]. Para tal, uma *heap* é inicializada com o primeiro nó de cada arquivo e a cada ciclo, o nó no topo da *heap* é retirado, escrito no novo arquivo de saída e substituído na *heap* pelo próximo nó do seu arquivo de origem. Durante o processo, os nós duplicados chegam juntos ao topo da *heap*, onde o *merge* mantém apenas aquele com menor valor de g. O processo continua até que todos os arquivos tenham sido totalmente percorridos. Então inicia-se a próxima iteração da busca, usando o arquivo de saída do *merge* como arquivo de entrada. O algoritmo alterna fases de expansão e de *sort-merge* até que uma solução ótima seja encontrada [21].

O algoritmo de *sort-merge* descrito acima pode exigir até o dobro de espaço em disco que o necessário. Quando um nó é lido para a memória, não é mais necessário em disco, contudo a maioria dos sistemas operacionais não permite a remoção de parte do conteúdo

do arquivo enquanto esse está sendo lido. Se os nós em cada arquivo estão uniformemente distribuídos sobre os espaço de estados, é possível que a remoção só ocorra quando a fase de *sort-merge* tenha sido quase completada, momento no qual esses nós, excluindo-se suas duplicatas, terão sido escritos no novo arquivo de entrada. Para contornar esse problema, cada um dos arquivos descritos acima é quebrado em uma sequência de segmentos de arquivo menores, de forma a possibilitar a remoção deles assim que forem lidos para a memória.

A complexidade de I/O desse algoritmo é da ordem do tamanho do espaço de busca vezes o número de iterações de expansão. Para buscas em largura, isso não é pior que tamanho do espaço vezes a sua profundidade, que pode ser reduzido realizando-se as iterações de expansões iniciais sem realizar *sort-merge* até que a memória esteja cheia pela primeira vez [21].

3.2.2 *Parallel External Partial Expansion A** (PE2A*)

O PEA* [15] é uma variante do A* que reduz a necessidade de memória para a solução do problema do alinhamento múltiplo ao armazenar em sua lista de nós abertos, apenas os nós julgados promissores. Um nó i, filho de um nó j, é julgado promissor se $f(i) <= C + F(j)$, aonde C é um valor de corte não-negativo pré-definido e $F(j)$ é inicialmente igual a $f(j)$, e ao fim de cada etapa de expansão, se o nó j possui nós filhos não promissores, o valor de $F(j)$ é atualizado para o menor valor de $f(n)$ entre todos os filhos não promissores de j. Assim, se o nó j possui apenas filhos não promissores, então nenhum nó é adicionado a lista de nós abertos e apenas o valor de $F(j)$ é atualizado, diminuindo ainda mais a prioridade de j. O pseudocódigo para o PEA* é mostrado no Algoritmo 3.

O HBDDD é uma forma eficiente de DDD que evita o *overhead* da ordenação em disco e é capaz de processar duplicatas em tempo linear. Uma função de *hash* é utilizada para escolher o arquivo de destino de um nó. Assim sendo, nós duplicados serão direcionados ao mesmo arquivo e, dessa forma, no momento da remoção de duplicatas, apenas nós em um mesmo arquivo precisam estar em memória. Esse fato também possibilita a paralelização da busca, uma vez que cada arquivo pode ser atribuído a uma *thread* distinta [22].

O *Parallel External Partial Expansion A** (PE2A*) [15] é uma combinação do PEA* e do HBDDD. O PE2A* funciona em dua fases: expansão e *merge*. A fase de expansão mapeia nós para *buckets* usando uma função de *hash*. Cada *bucket* possui três arquivos relacionados a si, um arquivo de nós fronteiriços, ainda a serem expandidos, um arquivo de nós fechados que foram expandidos e um arquivo de nós novos, que serão expandidos posteriormente.

Algoritmo 3 PEA* [15]

1: $g(s) \leftarrow 0$
2: $F(s) \leftarrow g(s) + h(s)$
3: $OPEN \leftarrow \{s\}$
4: $CLOSED \leftarrow \emptyset$
5: **while** $OPEN \neq \emptyset$ **do**
6: $\quad n \leftarrow min(F(n_i)), n_i \in OPEN$
7: $\quad OPEN \leftarrow OPEN - \{n\}$
8: $\quad$ **if** $n \in T$ **then**
9: $\quad\quad$ **return**
10: $\quad$ **end if**
11: $\quad SUCC_{<=C} \leftarrow \{n_j \mid n_j \in succ(n), f(n_j) <= F(n) + C\}$
12: $\quad SUCC_{>C} \leftarrow \{n_k \mid n_k \in succ(n), f(n_k) > F(n) + C\}$
13: $\quad$ **for each** $n_l \in SUCC_{<=C}$ **do**
14: $\quad\quad$ **if** $n_l \notin OPEN \cup CLOSED$ **then**
15: $\quad\quad\quad g(n_l) = g(n) + c(n, n_l)$
16: $\quad\quad\quad F(n_l) = g(n_l) = h(n_l)$
17: $\quad\quad\quad OPEN \leftarrow OPEN \cup \{n_l\}$
18: $\quad\quad$ **else if** $n \in OPEN$ **and** $g(n) + c(n, n_l) < g(n_l)$ **then**
19: $\quad\quad\quad g(n_l) = g(n) + c(n, n_l)$
20: $\quad\quad\quad F(n_l) = g(n_l) = h(n_l)$
21: $\quad\quad$ **else if** $n \in CLOSED$ **and** $g(n) + c(n, n_l) < g(n_l)$ **then**
22: $\quad\quad\quad g(n_l) = g(n) + c(n, n_l)$
23: $\quad\quad\quad F(n_l) = g(n_l) = h(n_l)$
24: $\quad\quad\quad CLOSED \leftarrow CLOSED - \{n_l\}$
25: $\quad\quad\quad OPEN \leftarrow OPEN \cup \{n_l\}$
26: $\quad\quad$ **end if**
27: $\quad$ **end for**
28: $\quad$ **if** $SUCC_{>C} = \emptyset$ **then**
29: $\quad\quad CLOSED \leftarrow CLOSED \cup \{n\}$
30: $\quad$ **else**
31: $\quad\quad F(n) \leftarrow min(f(n_m)), n_m \in SUCC_{>C}$
32: $\quad\quad OPEN \leftarrow OPEN \cup \{n\}$
33: $\quad$ **end if**
34: **end while**

O algoritmo expande o conjunto de nós fronteiriços dentro do limite f atual e mantém controle do menor valor de f dentre os nós que ultrapassa o limite, para que sirva como limite na próxima iteração. O pseudocódigo para o PE2A* é mostrado no algoritmo 4

O algoritmo inicia colocando um nó inicial no *bucket* definido pela função de *hash* utilizada. O limite mínimo é definido como o valor f do nó inicial. Todos os *buckets* com nós com f menor ou igual ao limite mínimo são divididos entre as *threads* para serem expandidos.

Uma *thread* de expansão prossegue expandindo os nós com f menor que o limite

mínimo, gerando dois conjuntos de sucessores para cada nó expandido n: Nós com $f <= F(n) + C$ e nós com $f > F(n) + C$. Nós sucessores cujos valores de f não ultrapassem o limite mínimo são recursivamente expandidos. Nós que ultrapassem o limite mínimo, mas que não ultrapassam $F(n) + C$ são anexados aos arquivos que representam a fronteira da busca e necessitam de detecção de duplicatas na próxima fase de *merge*. Nós parcialmente expandidos, que não possuem nenhum sucessor eliminado, são anexados aos arquivos que coletivamente representam os nós fechados. Nós com sucessores eliminados têm seu F atualizado e são anexados a lista de fronteira. Como PE2A* é estritamente *best-first*, ele pode ser encerrado assim que um nó alvo for alcançado.

Durante a fase de *merge*, cada *thread* começa por ler a lista de nós fechados do *bucket* para uma *hash-table*. Assim como o A*-DDD [22], o PE2A* exige no mínimo de espaço suficiente em memória para armazenar a maior lista de fechados entre todos os *buckets* que precisam de *merge*. A seguir, todos nós fronteiriços gerados na última iteração são analisados com a lista de nós fechados em busca de duplicatas, escrevendo no arquivo da lista de nós abertos da próxima iteração aqueles que não possuírem duplicatas, e para os demais apenas o nó com menor valor de g é salvo na lista de abertos.

O PE2A* foi capaz de resolver 80 das 82 instâncias do *BAliBASE Reference Set 1 benchmark* [37], com a heurística $h_{all,2}$ e $h_{all,3}$, para as duas instâncias mais difíceis [15].

3.2.3 *Parallel Frontier A* with Delayed Duplicate Detection* (PFA*-DDD)

*Frontier A** (FA*) [20] e DDD [31] são duas estratégias de como lidar com as limitações de memória do algoritmo A*. FA* diminui a quantidade necessária de memória para o algoritmo, reduzindo significantemente o número de estados expandidos. DDD, por sua vez, diminui a quantidade de memória requerida por cada nó, melhora a localidade das referências e permite o uso do disco invés de memória [21].

Korf [20] afirma em sua apresentação do algoritmo FA* que a união das duas estratégias era não trivial, uma vez que possibilitaria uma situação de *leak-back*, na qual um nó já expandido o seria novamente [21]. Niewiadomski, Amaral e Holte [27] apresentaram uma implementação que evita o problema de *leak-back*, a qual chamaram *Frontier A* with Delayed Duplicate Detection* (FA*-DDD), bem como uma versão paralelizada deste, o PFA*-DDD.

Dado um grafo $G(V, E)$ ponderado e direcionado, um par de vértices $s, t \in V$, e uma função heurística h admissível e consistente para t, os algoritmos computam o caminho de custo mínimo de s a t. As listas *Open*, contendo os nós abertos, *ClosedIn*, de arestas

Algoritmo 4 PE2A* [15]

```
1: function SEARCH(initial)
2:     bound ← f(initial); bucket ← hash(initial)
3:     write(OpenFile(bucket), initial)
4:     while ∃bucket ∈ Buckets : min_f(bucket) <= bound do
5:         for each bucket ∈ Buckets : min_f(bucket) <= bound do
6:             ThreadExpand(bucket)
7:         end for
8:         if incumbent then
9:             Break
10:        end if
11:        for each bucket ∈ Buckets : NeedsMerge(bucket) do
12:            ThreadMerge(bucket)
13:        end for
14:        bound ← min_f(Buckets)
15:    end while
16: end function

17: function THREADEXPAND(bucket)
18:    for each state ∈ Read(OpenFile(bucket)) do
19:        if F(state) <= bound then
20:            RecurExpand(state)
21:        else
22:            append(NextFile(bucket), state)
23:        end if
24:    end for
25: end function
```

originado em nós não-fechados e chegando a nós fechados, e *ClosedOut*, de arestas indo de nós fechados para nós abertos.

Os algoritmos iniciam com *Open* contendo o nó inicial s e as listas *ClosedIn* e *ClosedOut* vazias. Enquanto $Open \neq \emptyset$ o algoritmo realiza uma etapa de busca. Esta etapa de busca ocorre da seguinte maneira: Verifica-se se t está em *Open* e se o v de t é $fmin$. Se for, finaliza o algoritmo e retorna g de t como sendo o custo mínimo de um caminho de s à t. Caso contrário, expande-se todos os vértices v em *Open* tal que $f(v) = fmin$, coletando os vértices gerados e sues valores de f e g. Atualiza-se *Open*, *ClosedIn* e *ClosedOut* para que fiquem consistentes, marcando os vértices expandidos como fechados e cada vértice distinto gerado, não-aberto e não-fechado, sendo marcado com aberto.

Para atualizar *Open* retiram-se os vértices expandidos e adicionam-se os vértices gerados que não estavam em *Open* antes da atualização nem receberam uma aresta de um vértice de *ClosedIn* antes da atualização. Se um vértice em *Open*, após a atualização,

Algoritmo 4 PE2A* [15] (continuação)

```
26: function RecurExpand(n)
27:     if IsGoal(n) then
28:         incubent ← n
29:         return
30:     end if
31:     SUCC_p ← {n_p | n_p ∈ succ(n), f(n_p <= F(n) + C)}
32:     SUCC_q ← {n_q | n_q ∈ succ(n), f(n_q > F(n) + C)}
33:     for each succ ∈ SUCC_p do
34:         if f(succ) <= bound then
35:             RecurExpand(succ)
36:         else
37:             append(NextFile(hash(succ)), succ)
38:         end if
39:         if SUCC_q = ∅ then
40:             append(ClosedFile(hash(n)), n)
41:         else
42:             F(n) ← min_f(n_q), q ∈ SUCC_q
43:             append(NextFile(hash(n)), n)
44:         end if
45:     end for
46: end function

47: function ThreadMerge(bucket)
48:     Closed ← read(ClosedFile(bucket))
49:     Open ← ∅
50:     for each n ∈ NextFilebucket do
51:         if n ∉ Closed ∪ Open or g(n) < g(Closed ∪ Open[n]) then
52:             Open ← (Open − Open[n]) ∪ {n}
53:         end if
54:     end for
55:     write(OpenFile(bucket), Open)
56:     write(ClosedFile(bucket), Closed)
57: end function
```

possui uma duplicata, são definidos f e g como sendo os valores mínimos desses dentre todas as duplicatas. Para atualizar *ClosedIn*, removem-se as arestas que começam nos vértices expandidos e adiciona-se as arestas que chegam a um vértice expandido, desde que não estejam em *ClosedOut* antes da atualização. Para atualizar *ClosedOut*, removem-se as arestas chegando a vértices expandidos e adicionam-se as arestas originadas em nós expandidos, desde que não estivessem em *ClosedIn* antes da atualização. Se *Open* estiver vazia ao fim do passo de busca, finaliza-se a execução.

Para que o problema de *leak-back* acontecesse, um vértice gerado que fosse fechado ou que estivesse para ser fechado, teria que ser adicionado à lista *Open* durante a atualização dessa. Isso não ocorre pois, durante a atualização da lista *Open*, o uso da lista *ClosedIn* pré atualização filtra vértices gerados que estejam fechados, enquanto que o uso da lista *ClosedOut* filtra os vértices prestes a serem fechados.

Quando G é não-direcionado ou bidirecionado o algoritmo pode ser simplificado eliminando *ClosedOut*, pois com G não-direcionado, as listas *ClosedIn* e *ClosedOut* são equivalentes e se for bidirecionado *ClosedOut* é equivalente à transposta de *ClosedIn*.

FA*-DDD sequencial

O algoritmo FA*-DDD [27] computa um conjunto conciso de registros $\mathcal{X}_d$ para valores crescentes de d. Cada $\mathcal{X}_d$ representa a informação de *Open*, *ClosedIn* e *ClosedOut* depois de d passos de busca. Seja $fmin_d$ o valor f mínimo dentre todos os registros de $\mathcal{X}_d$, $\mathcal{X}_d^{fmin_d}$ o conjunto de registros de $\mathcal{X}_d$ cujo valor de f é igual a $fmin_d$. Para $d = 0$, $\mathcal{X}_d = \{(s, 0, h(s), \emptyset, \emptyset)\}$. Para $d >= 0$, o algoritmo computa $\mathcal{X}_{d+1}$ da seguinte maneira: se $\mathcal{X}_d^{fmin_d}$ contém um registro cujo valor v é t, então a execução termina e o valor g do registro é retornado como o custo mínimo de um caminho até t. Senão, o algoritmo expande os registros em $\mathcal{X}_d^{fmin_d}$ para computar $\mathcal{Y}_d$, o conjunto registros gerados pelas expansões. Em seguida, $\mathcal{X}_d$ é reconciliada a $\mathcal{Y}_d$ para computar $\mathcal{X}_{d+1}$. Se $\mathcal{X}_{d+1}$ for vazio ou $fmin_d = \infty$, a execução termina e ∞ é retornado como o custo mínimo de um caminho de s até t [27].

PFA*-DDD

O PFA*-DDD [27] é uma versão paralela do algoritmo sequencial que distribui o trabalho desse entre n *workstations*. Sejam $v_{min}, v_{max} \in V$ tais que $v_{min} \leq u \leq v_{max}, \forall u \in V$. Uma lista de intervalo-n é uma lista de $n+1$ vértices onde v_{min} é o primeiro elemento, o último vértice é v_{max} e cada vértice é maior ou igual ao vértice anterior.

Durante a computação de $\mathcal{X}_d$, registros são atribuídos a uma *workstation* de acordo com uma lista de intervalo-n Λ_d. Todos os registros atribuídos a uma *workstation* i tem um valor v tal que $\Lambda_d[i] < v \leq \Lambda_d[i+1]$. $\mathcal{X}_{d,i}$ representa os registros na *workstation* i no começo da iteração d.

Para $d = 0$, $\mathcal{X}_{0,0} = \{(0, s, h(0), \emptyset, \emptyset)\}$ e $\mathcal{X}_{0,i} = \emptyset, \forall i \neq 0$. Seja $fmin_{d,i}$ o valor mínimo de f na *workstation* i e $fmin_d$ o valor mínimo de f entre todas elas. $\mathcal{X}_{d,i}^{fmin_d}$ é o conjunto de registros em $\mathcal{X}_{d,i}$ cujo valor de f é $fmin_d$.

O algoritmo prossegue expandindo os registros nos diferentes $\mathcal{X}_{d,i}^{fmin_d}$. Para garantir um balanceamento apropriado, os registros devem ser redistribuídos entre as *workstations*, logo, na **fase 1** todas as estações compartilham o tamanho de seus $\mathcal{X}_{d,i}^{fmin_d}$ para obter o número total de registros a serem expandidos. A *Workstation* i pode então determinar os índices inicial e final que ela deve expandir. Se algum registro que deverá ser expandido pela *workstation* i não estiver em sua memória local, ele é transferido para essa.

Na **fase 2** *workstation* i expande os registros de cada $\mathcal{X}_{d,i}^{fmin_d}$ atribuído a ela para gerar $\mathcal{Y}_{d,i}$, o conjunto dos registros gerados pelas expansões. $\mathcal{Y}_{d,i}$ pode conter duplicatas, por isso, a *workstation* i ordena os registros e reduz $\mathcal{Y}_{d,i}$.

Se durante a expansão, uma *workstation* i encontra um registro r cujo valor v é t, ela dispara uma *flag* de término e salva o valor de g de r. Ao final da fase 2, se qualquer *workstation* disparou a *flag* de término, a execução é encerrada e o valor de g armazenado é retornado.

O próximo passo consiste de reconciliar os registros nos diferentes $\mathcal{X}_{d,i}$ com os registros nos $\mathcal{Y}_{d,i}$. A **fase 3** decide entre duas estratégias de reconciliação. Na opção A, a mesma partição de $\mathcal{X}_d$ é utilizada os registros em cada $\mathcal{Y}_{d,i}$ são mandados para suas *workstations* de origem conforme determinado pela partição. A estratégia B depende de uma técnica de amostragem para redistribuir os registros em $\mathcal{X}_d$ e $\mathcal{Y}_d$. Para computar "passo de amostragem", todas as *workstations* comunicam o tamanho de seus $\mathcal{X}_{d,i}$ e $\mathcal{Y}_{d,i}$ para a *workstation* 0. Uma vez que o "passo de amostragem" tenha sido transmitido, as *workstations* enviam uma lista de amostras de suas $\mathcal{X}_{d,i}$ e $\mathcal{Y}_{d,i}$ para a *workstation* 0. A *workstation* 0 então calcula uma nova lista de intervalos Λ_{d+1} e a transmite para as demais. A escolha entre as estratégias A e B é baseada na estimativa de trabalho máximo de cada *workstation* para cada estratégia.

O objetivo da reconciliação na **fase 4** é obter $\mathcal{X}_{d+1,i}$ em cada *workstation* de forma que:

$$\bigcup_{0 \leq i < n} \mathcal{X}_{d+1,i} = \mathcal{X}_{d+1} \tag{3.2}$$

$\mathcal{X}_{d+1}$ possui um registro para cada vértice em $\mathcal{X}_d$ e $\mathcal{Y}_d$ que não estava em $\mathcal{X}_d^{fmin_*}$. Se a estratégia A foi escolhida para a fase 4, todos os registros são reconciliados localmente.

Na **fase 5**, cada *workstation* i deleta $\mathcal{X}_{d,i}$, incrementa d e prossegue para a fase 1.

Dicionários e Filas de prioridade

As estratégias de reconciliação utilizadas pelo algoritmo paralelo PFA*-DDD necessitam de formas eficientes para encontrar um registro representando um vértice $v \in \mathcal{X}_{d,i}$, de se encontrar um registro com um determinado valor f e para eliminar de $\mathcal{X}_{d,i}$ todos os registros com valor $f = fmin$. A solução encontrada foi manter duas representações de $\mathcal{X}_{d,i}$, um dicionário indexado pelo vértice e uma fila de prioridade ordenada pelo valor f, conforme mostrado na Figura 3.3.

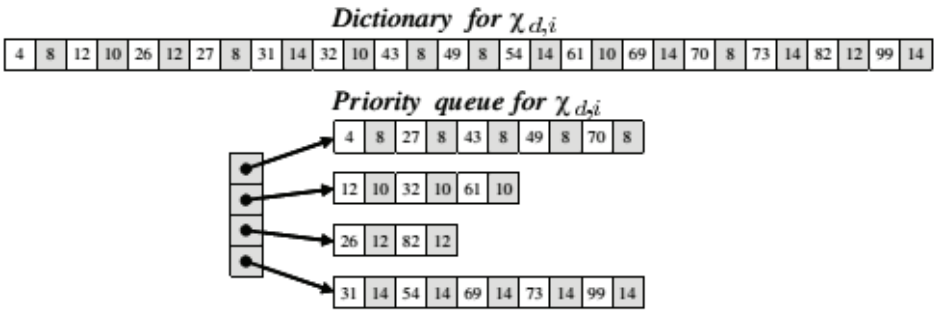

Figura 3.3: Dicionário e fila de prioridade para $\mathcal{X}_{d,i}$, Assumindo que um registro consiste apenas do vértice (branco) e do valor f (cinza) (Fonte: [27]).

A combinação dessas estruturas possui como vantagens: durante a expansão, os registros $\mathcal{X}_{d,i}$ que têm um valor f mínimo são encontrados em $O(1)$ na fila de prioridade. Como os registros são ordenados nos dicionários, a computação das amostras é trivial e o número de transferências de dados entre as *workstations* após a repartição é reduzido. Além disso, se a estratégia A foi escolhida, a busca por registros que representam um mesmo vértice no dicionário tem custo de tempo de $log(k)$, onde k é o número de registros no dicionário. A eliminação de registros com valor valor $f = fmin$ na fila de prioridades também pode ser feita em tempo constante e, por fim, a travessia de $\mathcal{X}_{d,i}^{fmin_d}$ durante a reconciliação é sequencial, melhorando a localidade.

Capítulo 4

A* Paralelo para Ambientes de Alto Desempenho

4.1 Conceitos Básicos

Computação de alto desempenho é um campo de estudo da Ciência da Computação que começou a ser explorado no início da década de 1960, quando pela primeira vez cogitou-se a possibilidade de se dividir problemas em subproblemas menores que pudessem ser resolvidos simultaneamente [8].

Na década de 1970 houve a criação e popularização dos supercomputadores vetoriais, projetados para executar instruções sobre um vetor de dados. Sendo assim, um exemplo de *Single Instruction Multiple Data* (SIMD) da arquitetura de Flynn [10], definida como uma instrução aplicada simultaneamente à vários dados distintos. Estes supercomputadores vetoriais alcançaram uma velocidade de processamento significativamente superior a das arquiteturas convencionais da época.

O início da década de 1980 trouxe a popularização dos *Symmetric Multiprocessors* (SMP), que se encaixam na categoria *Multiple Intructions Multiple Data* (MIMD) da arquitetura de Flynn, definida como diversas instruções aplicadas a diferentes dados. Isso permitia que vários processadores fossem utilizados, cada um executando suas instruções sobre um conjunto de dados específico. Essas máquinas em geral utilizavam o paradigma de memória compartilhada para trocar dados entre processadores.

Próximo ao final da década de 1980 contudo, as limitações inerentes aos SMP ficaram claras. Conforme o número de processadores empregados era aumentado, mais rapidamente o meio de comunicação entre processadores, o barramento, era saturado [8].

Visando contornar esse problema, foram propostos os sistemas de computação distribuída, nos quais computadores relativamente completos eram conectados por uma rede,

de forma a gerar um sistema integrado. Esses sistemas empregavam o paradigma de troca de mensagens para realizar a comunicação entre computadores.

Nessa categoria estão os *clusters* de computadores, definidos como sistemas de computação distribuída que utilizam *hardware* e *software* disponíveis no mercado (*commodity components*) [29]. Os *clusters* se disseminaram rapidamente possuírem um grande apelo comercial, resultante do seu baixo custo e facilidade de programação.

Enquanto os *clusters* eram desenvolvidos, arquiteturas específicas também eram disseminadas. Essa arquiteturas foram desenvolvidas de forma a responder a necessidades específica, sendo assim otimizadas para certos problemas, recebendo por isso a alcunha de aceleradores. Exemplos típicos de aceleradores são as FPGAs (*Field Programmable Gate Arrays*) e GPUs (*Graphics Processecing Units*).

Atualmente é comum encontrar sistemas híbridos, como por exemplo um *cluster* de computadores *multicore* (SMP) no qual um ou mais *mulicores* estão conectados a aceleradores, tais como FPGAs ou GPUs.

Os *Clusters* de computadores são conjuntos de computadores conectados através de *hardware* e *software* especializado, apresentado como uma imagem única ao usuários [38]. Essa imagem única é uma abstração que permite ao usuário considerar o sistema como sendo apenas um computador.

Em um *cluster*, cada nó de processamento é um computador quase completo, possuindo processador, memória e disco rígidos próprios e geralmente sem monitores, teclados ou *mouse* [17]. Esses nós são interligados por uma rede, em geral de alta velocidade utilizando componentes disponíveis no mercado.

Clusters possuem vantagens como:

- Potencial de escalabilidade, alto desempenho e alta disponibilidade;
- Relativa facilidade na adição e remoção de componentes;
- Relativa facilidade de programação

Atualmente, o padrão MPI (*Message Passing Interface*) [36] é amplamente utilizado por *clusters*. O MPI permite a troca de mensagens de forma síncrona ou assíncrona entre processos em computadores distintos, bem como oferece as primitivas de *broadcast* "*one to all*"e "*all to all*", entre outras.

Apesar de os *clusters* SMPs (ou *clusters de multicores*) poderem ser programados exclusivamente com troca de mensagens, uma maneira mais eficiente consiste em utilizar memória compartilhada para comunicação entre núcleos de um processador, utilizando OpenMP [28] ou POSIX *threads* [30] (*pthreads*) , e entre os computadores por troca de mensagem, com MPI.

4.2 *Parallel A**

Conforme explorado no Capítulo 3, o algoritmo A* é uma opção utilizada para calcular o alinhamento múltiplo exato entre sequências, contudo a memória requerida inviabiliza a execução do algoritmo para instâncias com muitas sequências ou cujas sequências sejam muito compridas. Essa limitação afeta ainda mais implementações feitas em paralelo, uma vez que cada *thread* empregada possui um espaço de memória próprio.

Nesta seção, apresentaremos o algoritmo *Parallel A**, que foi utilizado no presente trabalho de graduação. O *Parallel A** [35] tem por objetivo computar o alinhamento ótimo entre múltiplas sequências utilizando várias *threads* em ambiente de memória compartilhada. As seções 4.2.1, 4.2.2 e 4.2.3 apresentam, respectivamente, o projeto da *OpenList*, o uso de *templates* e a estratégia de paralelização empregada no *Parallel A**.

4.2.1 Implementação da *OpenList*

A implementação da *OpenList* é um dos pontos de maior impacto na memória e no tempo de execução do algoritmo. Ela é fundamental em duas operações do algoritmo: **find** e **dequeue**.

- **Operação dequeue:** decide qual o nó a ser expandido, retirando-o da *OpenList*;
- **Operação find:** verifica se um nó já está presente na *OpenList*, verificando se um nó de maior prioridade já foi encontrado

Estruturas de dados convencionais não se adequam bem às duas operações simultaneamente. Uma fila de prioridade pode manter os nós ordenados de acordo com sua prioridade, tornando simples a operação de *dequeue*, contudo não possui uma operação otimizada para encontrar um nó dentro dela. Uma *hashtable* pode ser utilizada para armazenar e encontrar nós já expandidos, contudo ela não possui operação que retorne o nó de maior prioridade.

O principal problema se deve ao fato de, apesar das duas operações serem aplicadas sobre os mesmos dados, os nós, elas se baseiam em campos distintos desses. A primeira precisa ordenar e encontrar nós de acordo com sua prioridade. Já a segunda precisa encontrar nós de acordo com suas coordenadas, que identificam o nó unicamente.

Uma forma comum de lidar com essa necessidade é utilizar ambas as estruturas simultaneamente. Uma fila de prioridade é utilizada para a operação de *dequeue* e um dicionário, que pode ser implementado com um *hashtable* ou um *map*, para a operação de *find*. Essa abordagem foi utilizada pelo FA*-DDD, conforme explicado na seção 3.2.3, e pode ser visualizada na Seção 3.2.3

Essa estratégia atende às necessidades do algoritmo A*, contudo possui um claro *overhead* pois cada nó é armazenado duas vezes em memória e cada operação de *dequeue* realizada acarreta na execução de uma operação de busca no dicionário.

Visando suprir as necessidades do A* e, ao mesmo tempo, evitar o *overhead* da implementação acima, Sundfeld at al. [35] propuseram para o *Parallel A** uma implementação da *OpenList* utilizando um *multiindex* [4].

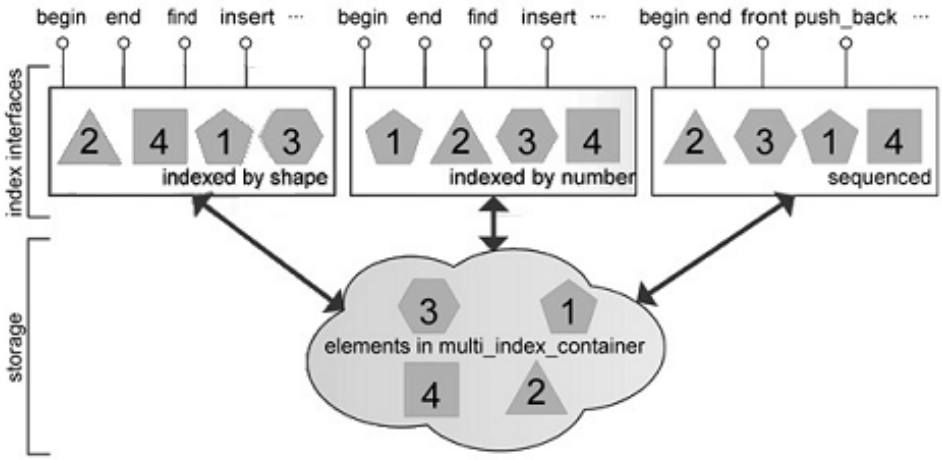

Figura 4.1: Exemplo de multiindex com 3 indíces: a forma, o número e a ordem em que foram inseridos (Fonte: [4]).

A Figura 4.1 ilustra um exemplo no qual deseja-se armazenar objetos com um formato e número específicos, bem como a ordem na qual foram inseridos. A ideia do *multitindex* é permitir que o usuário defina os campos a serem utilizados como índices, tornando o armazenamento em memória dos objetos invisível mas garantindo não sejam armazenadas duplicatas.

Para a *OpenList* foram utilizados dois índices, a prioridade e as coordenadas. Isso permitiu evitar o *overhead* da abordagem descrita anteriormente e dependendo da implementação do *multiindex*, pode permitir que as buscas e inserções em ambos os índices sejam feitos em tempo $O(log(n))$.

A Tabela 4.1 mostra uma comparação feita entre uma implementação usando *multitindex* e uma utilizando fila de prioridade e um *map*, ambos da *Standard Template Library* (STL), biblioteca padrão e disponível para programas C++. O *multiindex* foi implementado usando a *Libboost*, uma biblioteca popular e disponível na maioria dos sistemas operacionais atuais [35]. Pode-se constatar na tabela que a implementação com *multiindex* executou mais rápido e ocupou menos espaço em memória RAM.

Tabela 4.1: Comparação entre as implementações baseadas em Dicionários e Multiindex (Fonte: [35]).

Número de sequências	Tipo	Tempo (s)	RAM (GB)	*Page faults* (k)
3	Dicionário	41.05	1.93	126.80
	Multiindex	31.61	1.86	122.28
4	Dicionário	73.29	2.91	177.91
	Multiindex	67.02	2.34	155.43

4.2.2 *Templates*

O número de sequências em um alinhamento múltiplo é uma variável de fundamental importância que influencia o comportamento do programa de diversas maneiras, tais como: o número de vizinhos de cada nó, que é igual a $s^k - 1$, onde k é o número de sequências, o cálculo do valor do SP depende dos $\frac{k*(k-1)}{2}$ pares possíveis e o número de coordenadas de cada nós depende do número de sequências, por exemplo, para $k = 3$, as coordenadas do nó inicial são $(0, 0, 0)$, já para $k = 4$, são $(0, 0, 0, 0)$.

Isso implica que o espaço em memória necessário para armazenar um nó varia com o número de sequências. Uma solução para isso seria utilizar uma área de memória variável através de alocação dinâmica. Desta forma, um nó conteria os valores f e g, necessários para o algoritmo, e um campo c, que é um ponteiro para a localização em memória alocada para as coordenadas. Uma segunda alternativa seria utilizar um *array* dentro do nó, o que tornaria necessário uma estrutura diferente para cada número de sequências que um usuário fornecesse para o alinhamento. A Figura 4.2 ilustra essas possibilidades.

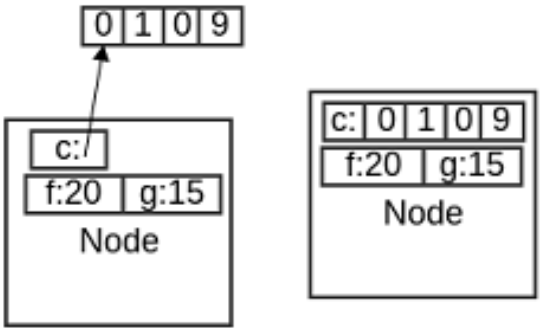

Figura 4.2: Duas possíveis implementações para representar um nó (Node) na memória: à esquerda utiliza alocação dinâmica para as coordenadas e à direita um *array* de tamanho fixo (Fonte: [35]).

O algoritmo *Parallel* A^* faz uso intenso de memória, ocorrendo *page faults* frequentemente durante a busca por informações de um nó. Além disso, quando se faz uso das coordenadas também podem ocorrer *page faults* uma vez que as regiões de memória são

alocadas em momentos diferentes, não havendo garantia de que estarão na mesmo página. Dado isso, manter as coordenadas dentro do nó é preferível pois melhora a localidade do dados. Sendo assim, para utilizar essa abordagem e flexibilizar o código de maneira a permitir a variação do número de sequências, Sundfeld at al. [35] propuseram o uso de *templates*.

O *Template* é um recurso disponível em várias linguagens de programação que permite a especificação de classes e funções com tipos genéricos, o que permite que sejam utilizados para vários tipos de dados diferentes, sem necessitar que sejam reescritas. O uso de *templates* na implementação do A* já havia sido utilizado e foi citado como um dos fatores que mais influenciaram o seu desempenho [5]. Os *Templates* são base de um paradigma de programação conhecido como metaprogramação.

```
1  template < int N >
2  struct Coordenadas {
3      int c[N];
4  };
5
6  int main()
7  {
8      Coordenadas<3> c3;
9      Coordenadas<5> c5;
10
11     c5.c[0] = c5.c[1] = c5.c[2] = c5.c[3] = c5.c[4] = 0;
12     c3.c[0] = c3.c[1] = c3.c[2] = 0;
13     return 0;
14 }
```

Figura 4.3: Código utilizando programação genérica para definir um *array* de tamanhos distintos (Fonte: [35]).

A Figura 4.3 ilustra um código onde é criada uma especificação genérica de uma estrutura que armazena um *array* de números inteiros. Na função *main*, são criadas duas estruturas to tipo *Coordenadas*, porém o parâmetro *template* passado para cada é diferente, sendo $< 3 >$ para a primeira e $< 5 >$ para a segunda, gerando estruturas com *arrays* de comprimento 3 e 5, respectivamente.

Neste caso o compilador gera duas estruturas diferentes em memória, denominadas **especializações**. Caso uma nova variável do tipo *Coordenadas* seja criada, com comprimento diferente das demais, basta adicionar uma linha na função *main* e o compilador irá criar a especialização correspondente na compilação, sem que tenha sido necessário alterar nada na estrutura de *Coordenadas*.

Os *Templates* também podem ser usados em classes, e, assim como para estruturas, cada especialização da classe gera uma nova estrutura na memória, contendo também as funções que estão implementadas dentro da classe. Isto implica mais uma otimização, pois as funções que dependem do número de sequências são geradas pelo compilador e diferentes otimizações podem ser aplicadas para cada caso.

A Tabela 4.2 demonstra uma comparação do desempenho e do consumo de memória entre uma implementação feita com alocação dinâmica e uma feita com *templates*. Os resultados indicaram que o uso de *templates* reduziu o tempo e o consumo de memória e ao mesmo tempo diminui a quantidade de *page faults* entre $46,44\%$ e $50,06\%$.

Tabela 4.2: Comparação entre alocação dinâmica e uso de templates (Fonte: [35]).

Número de sequências	Tipo	Tempo (s)	RAM (GB)	*Page faults* (k)
3	Memória Dinâmica	31.61	1.86	122.28
	Templates	18.62	0.86	56.80
4	Memória Dinâmica	67.02	2.34	155.43
	Templates	41.12	1.16	77.81
5	Memória Dinâmica	621.32	9.96	653.41
	Templates	401.53	4.89	320.79

4.2.3 Paralelização

A estratégia de paralelização escolhida para o *Parallel A** foi executar as etapas iniciais do algoritmo A* em paralelo, o que quer dizer que cada *thread* usada possuirá sua própria *OpenList*, e, na fase de reconciliação, avaliar se um dado nó permanece na *thread* na qual está ou se deverá ser executado por outra *thread*.

Para realizar essa avaliação, foi usada uma função de *hash* $hs(n) = x$, sendo n as coordenadas do nó e $0 \leq x \leq t$, com t sendo o número de total de *threads*. A Figura 4.4 ilustra a arquitetura paralela utilizada, com cada nó possuindo sua *OpenList* e uma lista interna, utilizada para receber os nós de outras *threads* após a avaliação.

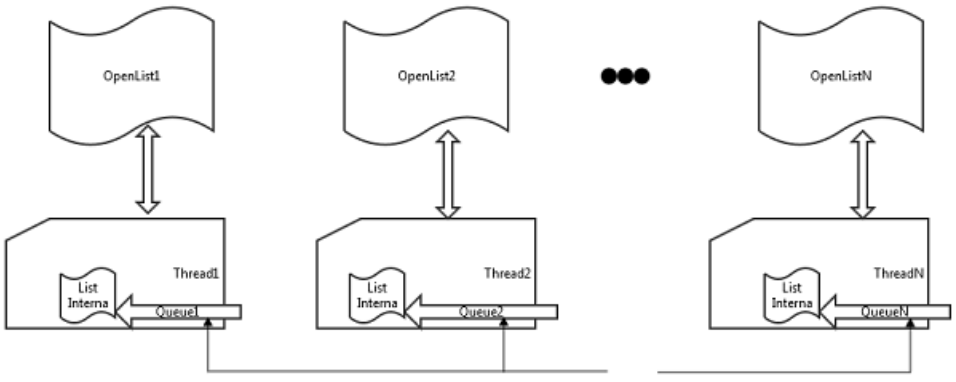

Figura 4.4: Arquitetura Paralela proposta para o A* paralelo (Fonte: [35]).

Para calcular a função de *hash* ,primeiramente , deve-se limitar a imagem da função. Para tal, define-se a Equação 4.1 [35], responsável por distribuir o nó n, utilizando a função de *hash* H.

$$H_{hashthread}(n) = (H(n)\ SHIFT\ H_{shift})MOD(t) \tag{4.1}$$

Onde H_{shift} é uma constante pré-definida e t é o número de *threads*. Isto quer dizer que, de todos os números que podem ser obtidos pelo *hash* H, apenas alguns bits de mais alta ordem são considerados, descartando-se o restante do número, de forma que $0 \leq Z_{hash}(n) \leq t$. Nesta situação, caso $H_{shift} \longrightarrow \infty$, chega-se ao caso em que todos os nós deverão ser processados pela *thread* 0, resultando em algoritmo igual ou pior que o serial. Por outro lado, se $H_{shift}0$ e $t \longrightarrow \infty$, chega-se ao outro extremo do paralelismo, com o *overhead* de comunicação chegando ao seu valor máximo.

No *Parallel* A^* foram analisadas quatro funções de *hash* [35]. A primeira utiliza curvas *Z-Order* [25], que mapeiam espaços multidimensionais para uma dimensão preservando a localidade dos dados, isto é, dois pontos próximos no espaço multidimensional estarão próximos na função *Z-Order*. Essas curvas contudo mapeiam sobre todos os inteiros, mas a função de *hash* necessária para o algoritmo deve mapear apenas sobre as t *threads*, por isso define-se a função 4.2.

$$Z_{full\ hash}(n) = (Z_{ord}(n)\ SHIFT\ H_{shift})MOD(t) \tag{4.2}$$

Onde $Z_{ord}(n)$ é a função que obtém a curva *Z-Order* das dimensões n. A segunda função de *hash* proposta foi a função 4.3.

$$S_{full\ sum}(n) = \sum_{i=0}^{k} n[i] \tag{4.3}$$

Na Equação 4.3, k é o número de sequências. Coordenadas próximas obterão somas próximas e, dessa forma, é possível aplicar a função $H_{hashthread}(n)$, de forma que apenas os bits de mais alta ordem sejam utilizados como *hash*.

Ambas as funções apresentadas utilizam todas as coordenadas de um nó para cálculo *hash*, por isso, imaginando que essa característica pudesse influenciar no desempenho, foram propostas versões das funções 4.2 e 4.3 que consideram apenas as duas primeiras coordenadas, $Z_{part\ hash(n)}$ e $S_{part\ sum(n)}$, respectivamente.

Para avaliar o desempenho das funções de *hash*, utilizou-se ambientes com 2 e 4 *threads*, bem como variou-se o valor de H_{shift}, tal que $0 \leq h \leq 12$, com $h = H_{shift}$.

A Figura 4.5 ilustra o desempenho das quatro funções de *hash* propostas. É possível notar as funções parciais e *full* não são muito diferentes. Contudo conforme H_{shift} cresce, as funções parciais perdem desempenho , o que não acontece com as funções *full*. Também notou-se que as funções de soma podem ter melhor desempenho que as curvas *Z-Order*, mas para $H_{shift} > 7$, esse desempenho piora consideravelmente, e para $H_{shift} \geq 10$

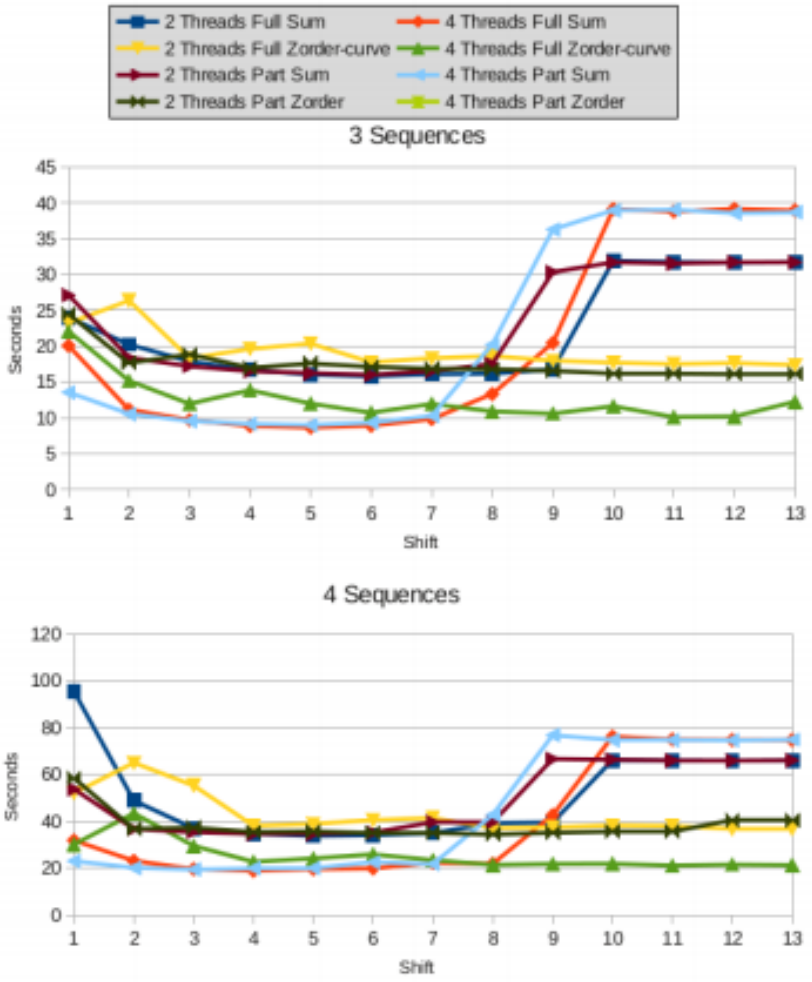

Figura 4.5: Gráfico comparativo para as funções de hash (Fonte: [35]).

o desempenho se mantém constante devido ao fato de a soma dos comprimentos das sequências utilizadas ser menor que 1024, o que faz com que para $H_{shift} \geq 10$, o resultado da função seja sempre 0.

4.2.4 Resultados experimentais

As funções $Z_{full\ hash}$ e $S_{full\ sum}$ foram escolhidas para uso no algoritmo. Testes foram feitos utilizando o conjunto *2myr* do Balibase [3], considerado um dos mais difíceis da primeira referência, em uma estação multiprocessada com 32 *cores*.

O experimento variou o número de *threads* utilizadas para analisar a escalabilidade da solução. As Figuras 4.6 a 4.7 ilustram os resultados para $Z_{full\ hash}$ e $S_{full\ sum}$, respectivamente.

A solução mostrou boa escalabilidade, melhorando seu desempenho a cada *core* adicionado, obtendo um *speedup* de $18x$ no melhor caso. A função de *hash Z-Order* demonstrou

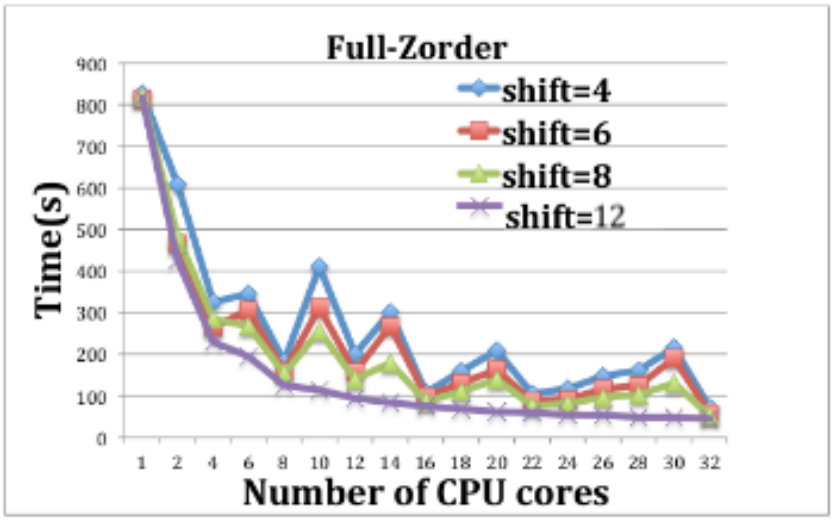

Figura 4.6: Testes *Zorder* para até 32 *cores* (Fonte: [35]).

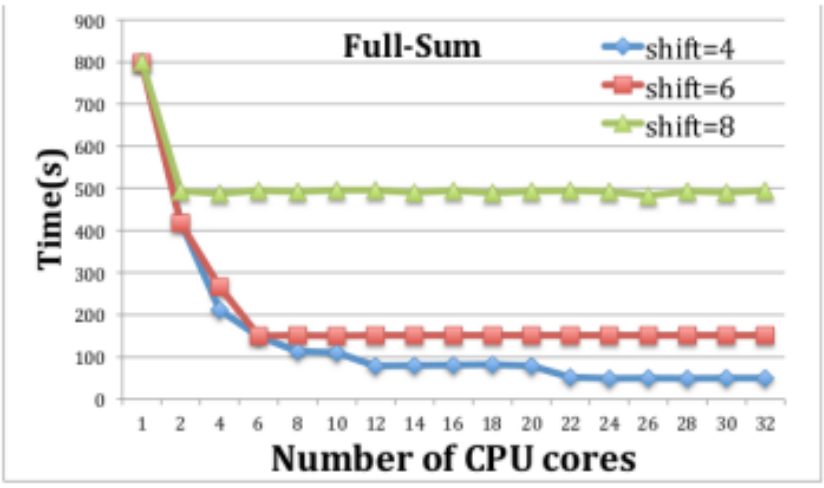

Figura 4.7: Testes $Fsum$ para até 32 *cores* (Fonte: [35]).

um desempenho mais estável para valores de H_{shift} maiores, mas para $4 \leq H_{shift} \leq 6$, o desempenho das duas foi muito próximo.

Com esses resultados, decidiu-se usar a função *hash Z-Order* e definir $H_{shift} = 12$ para um novo experimento. Como entrada, utilizaram 4 das 6 instâncias mais difíceis da referência 1 do Balibase [3]. Os resultados para as instâncias *2myr_ref1* e *2ack_ref1* podem ser vistos na Tabela 4.3. Esses testes foram realizados usando estações com 16 *cores*. Já para a instância *arp*, que pode ser vista na Tabela 4.4, uma quantidade muito maior de memória foi necessária e por isso foi utilizado uma estação com 32 *cores*, com processadores mais lentos, mas com uma quantidade de memória maior disponível.

Tabela 4.3: Solução das instâncias *2myr* e *2ack*, referência 1 do Balibase (Fonte: [35]).

Instância	Tempo(s)	RAM(GB)	*Page faults*(k)
2myr_ref1	547.99	76.9	5042
2ack_ref1	244.42	31.1	2040

Tabela 4.4: Solução da instância *arp*, referência 1 do Balibase (Fonte: [35]).

Instância	Tempo(min:s)	RAM(GB)	*Page faults*(k)
arp_ref1	146:27.698	532	35571

Como pode ser visto nas tabelas 4.3 e 4.4, o *Parallel* A^* necessita de quantidade imensa de RAM, podendo chegar a 532 GB na comparação da instância *arp_ref1* (Tabela 4.4). Isso invalida o uso dessa ferramenta em máquinas *desktops* encontradas atualmente no mercado, cuja capacidade média de memória RAM chega até 16 GB.

Capítulo 5

Disk-Assited Parallel A*

5.1 Visão Geral

A estratégia proposta para lidar com a limitação de memória do alinhamento múltiplo de sequências usando *Parallel A** [35] visa manter a uso da memória próximo a um *threshold* e sempre abaixo de um limite, transferindo nós da *ClosedList* de memória para o disco se necessário. Além disso, como existe a possibilidade de que os nós retirados de memória sejam necessários ao funcionamento do programa, a estratégia também deverá garantir que quando o *Parallel A** necessitar, esses dados sejam transferidos para a memória.

O limite máximo de ocupação de memória RAM deverá ser informado pelo usuário e representa a quantidade de memória RAM mínima disponível no sistema, em qualquer instante, durante o funcionamento do programa, para que ele opere corretamente. O valor deverá ser informado no momento da execução do programa.

Testes empíricos sugeriram que a *ClosedList* é responsável pela maior parte da ocupação da memória durante o funcionamento do *Parallel A**. Logo a estratégia proposta terá como objetivo transferir nós da *ClosedList* entre memória e disco para possibilitar o funcionamento do *Parallel A** para instâncias maiores que as possíveis anteriormente.

Para permitir a movimentação de nós da *ClosedList* da memória RAM para o disco, o *Disk-Assisted Parallel A** incluiu o conceito de *região* à implementação original do *Parallel A**, com o intuito de facilitar a diferenciação dos nós necessários para o processamento de cada etapa do algoritmo e os nós que podem ser transferidos para o disco, liberando espaço na memória RAM. Esse conceito é detalhado na Seção 5.2.

Com o conceito proposto de regiões, o *Disk-Assisted Parallel A** adicionou uma etapa de movimentação de dados entre memória e disco à iteração de cada *thread*, conforme descrito na Seção 5.3.3.

A Seção 5.3 explica em detalhe cada um dos módulos representados na Figura 5.2. A seguir, o conceito de região é detalhado.

5.2 Regiões por *thread*

Uma região corresponde a um subconjunto de nós do espaço de busca que serão processados sempre por uma mesma *thread*, podendo essa *thread* ser responsável por várias regiões. A inclusão do conceito de regiões visa permitir a separação de subconjuntos de nós da *ClosedList* que são necessários para o processamento do algoritmo em uma dada iteração, e que por isso devem estar em memória RAM, daqueles que não são necessários e que podem então ser transferidos para o disco, liberando espaço em RAM. Além disso, quanto maior o número de regiões, menor será o número de nós mapeados para cada região, o que diminui o tamanho individual de cada *ClosedList* e diminuindo o tempo gasto em operações de transferência de disco para memória RAM e de memória RAM para disco.

No código original (Capítulo 4), o conceito de regiões existia implicitamente, sendo o número de regiões igual ao número de *threads*, de forma que cada *thread* executava sobre sua própria região. Na implementação do *Disk-Assisted Parallel A** foi adicionada a variável regiões por *thread* (RPT), significando que cada *thread* T é responsável por RPT regiões, cada uma com suas *ClosedList* e *OpenList*.

A definição da região à qual o nó pertence é feita aplicando-se a função de *hash* sobre a posição do nó, restringindo-se a imagem da função ao produto do número de *threads* (t) pelo número de regiões por *thread* (r), conforme representado pela Equação 5.1.

$$H_{hashthread}(n) = (H(n)\ SHIFT\ H_{shift}) MOD (t \cdot r) \tag{5.1}$$

Define-se ainda a *região ativa* de uma *thread* como sendo a região sob responsabilidade da *thread* com o nó de maior prioridade. A região ativa é atualizada a cada iteração da *thread*.

A Figura 5.1 ilustra a estrutura de um *thread* e suas regiões r_1 a r_t no *Disk-Assisted Parallel A**.

5.2.1 HoldingList

Outra adição do *Disk-Assisted Parallel A** foi a criação de uma *HoldingList* às regiões. Essa lista será usada para armazenar os nós de uma região não ativa que forem expandidos no módulo de expansão. O objetivo da adição dessa lista é mitigar o tempo gasto na transferência de nós para a *OpenList*, que precisa procurar pelo nó na *ClosedList* e na própria *Openlist* pois um nó não pode existir simultaneamente na *OpenList* e na *ClosedList* da mesma região e nem haver duplicatas dentro da própria *OpenList*.

A *HoldingList* não considera a *ClosedList* no momento da inserção de nós, ela apenas garante que não existem duplicatas dentro dela mesma, mantendo apenas a versão de maior prioridade de cada nó adicionado. Além disso, a *HoldingList* também mantem

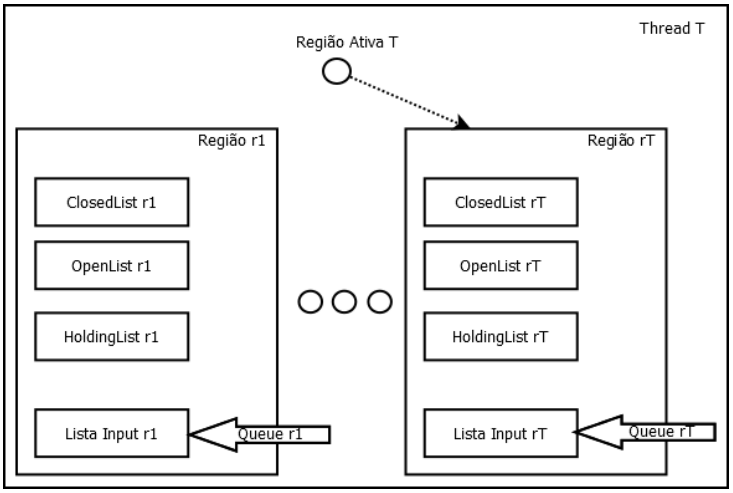

Figura 5.1: Estrutura de uma *thread* no DAPA*.

controle do nó de maior prioridade presente em sua lista, o que será usado no módulo de Atualização (Seção 5.3.2) para determinar a região ativa.

5.3 Descrição dos Módulos

O funcionamento do *Disk-Assisted Parallel A** pode ser dividido em cinco módulos, executados em sequência por cada *thread*, conforme ilustrado na Figura 5.2.

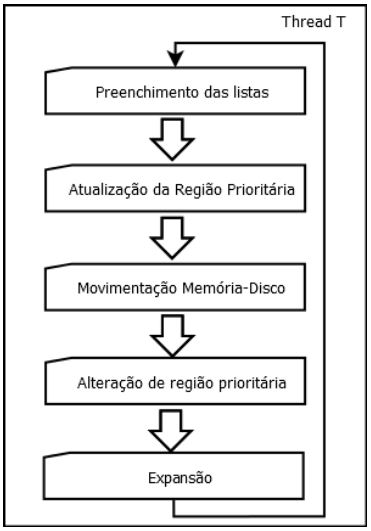

Figura 5.2: Visão geral do funcionamento de uma *thread* do *Disk-Assisted Parallel A**.

5.3.1 Preenchimento das Listas

Nesse módulo (Figura 5.2) é feita a transferência dos nós oriundos do módulo de expansão da *thread* na iteração anterior ou de outras *threads*. Esses nós expandidos foram escritos na *queue* da *thread* e devem ser transferidos para a *OpenList*, caso a região em questão seja a região ativa, ou para a *HoldingList*, caso contrário.

Como a *queue* de uma região é acessada por outras *threads*, os nós presentes na *queue* são transferidos dela para uma lista local, acessada apenas pela *thread* responsável, para liberá-la o mais rápido possível.

Para a região ativa, os nós são transferidos da lista interna para a *Openlist*, desde que o nó não esteja na *ClosedList*, ou se o valor de prioridade for melhor do que o nó na *ClosedList*, caso no qual o nó na *ClosedList* é descartado. Na Figura 5.3, esse procedimento é ilustrado para a região r_t.

Para as regiões não ativas, os nós são transferidos da lista interna para a *HoldingList*. A *HoldingList* no entanto nó mantém apenas uma cópia de cada nó. Logo se um nó que já estiver presente na lista for adicionado, apenas o nó com maior prioridade é mantido, descartando-se o outro. Na Figura 5.3, esse funcionamento é ilustrado pela região r_1.

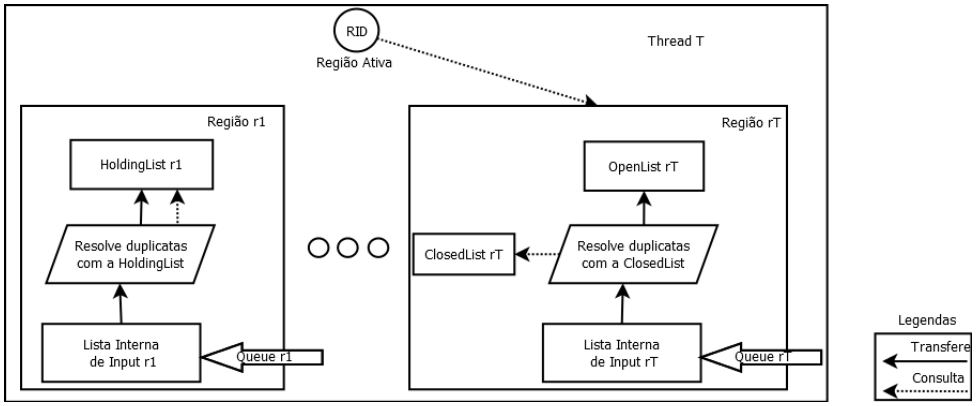

Figura 5.3: Módulo de preenchimento das Listas.

5.3.2 Atualização da região ativa

Nesse módulo, a *thread* acessa todas as regiões sob sua responsabilidade e identifica a região com o nó de maior prioridade dentre todas as regiões e a região, com a *ClosedList* em memória RAM, cujo nó mais prioritário tem a menor prioridade dentre os nós mais prioritários das outras regiões.

A região correspondente ao nó de maior prioridade passará a ser a região ativa da *thread* no módulo de Alteração da Região Prioritária (Seção 5.3.4). A outra região identificada é a candidata a ser transferida para disco no módulo de Movimentação (Seção 5.3.3).

5.3.3 Movimentação Memória-Disco

Após a atualização de sua região ativa, uma *thread* pode identificar os nós necessários ao prosseguimento da iteração e as regiões cuja *ClosedLists* podem ser transferidas para disco.

A *ClosedList* da região ativa deve estar em memória RAM para que a iteração da *thread* prossiga, logo, caso essa *ClosedList* tenha sido transferida para disco em uma iteração anterior, a *thread* a recupera do disco e a coloca em memória.

As *ClosedLists* da demais regiões sob responsabilidade da *thread* não serão mais acessadas durante a iteração corrente, logo, podem ser transferidas para o disco sem impedir o prosseguimento do algoritmo.

A Figura 5.4 ilustra as movimentações de dados da *ClosedList*. A seta 1 identifica a transferência da *ClosedList* de uma região não ativa da memória RAM para o disco. A seta 2 ilustra a transferência da *ClosedList* da região ativa identificada no módulo de Atualização (Seção 5.3.2) de disco para a memória RAM.

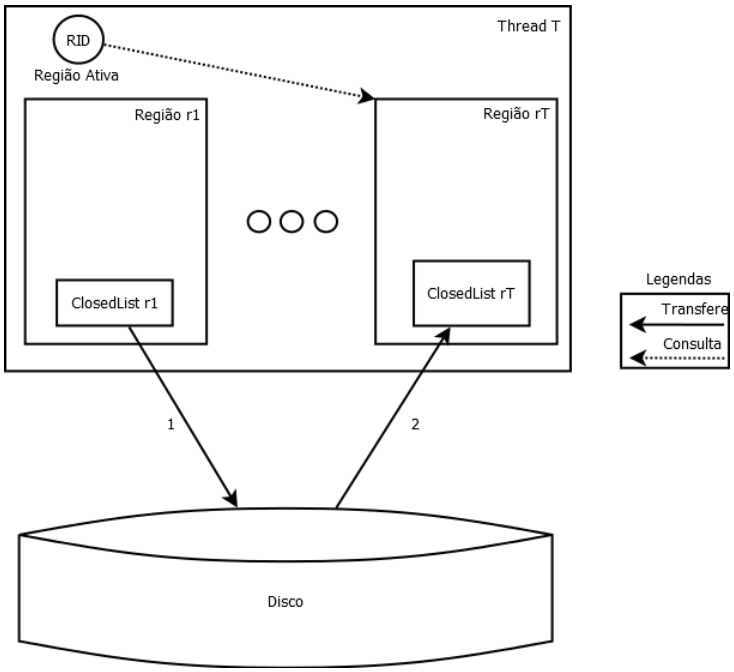

Figura 5.4: Módulo de movimentação de dados Memória-Disco.

5.3.4 Alteração da Região Ativa

Quando a região prioritária identificada no módulo de atualização (Seção 5.3.2) for diferente da atual região prioritária, é necessário transferir os nós que estiverem na *HoldingList* para a *OpenList*. Durante essa transferência a *ClosedList* é consultada para que um nó

só seja transferido caso não esteja na *ClosedList* ou se a sua prioridade for melhor do que a prioridade do nó presente na *ClosedList*.

A Figura 5.5 ilustra essa movimentação da *HoldingList* para a *OpenList*. A seta 1 indica a retirada do nó mais prioritário da *HoldingList*. A seta 2 ilustra a consulta realizada à *ClosedList* para verificar se um nó já foi aberto anteriormente. A seta 3 representa a inserção do nó na *OpenList*.

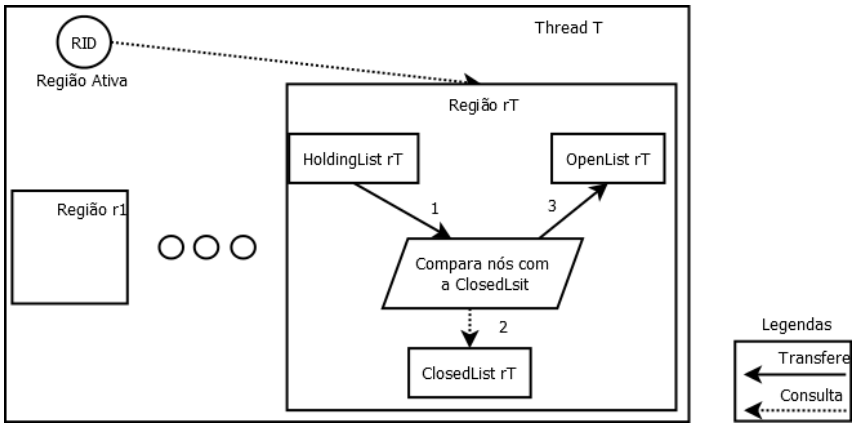

Figura 5.5: Movimentação de dados necessária quando houver troca de região ativa.

5.3.5 Expansão

O funcionamento do módulo de Expansão é ilustrado pela Figura 5.6. Nesse módulo, a *thread* atua apenas sobre a região ativa. Inicialmente, o nó de maior prioridade (*current*) é retirado da *Openlist*. Caso o nó retirado seja o nó final, o algoritmo segue para o procedimento de finalização, no qual as outras *threads* deverão confirmar se esse nó é a solução. Caso contrário, o nó é adicionado à *ClosedList*. Por fim, os nós vizinhos de *current* são adicionados às *queues* da respectivas regiões e a iteração é encerrada, seguindo para a próxima iteração.

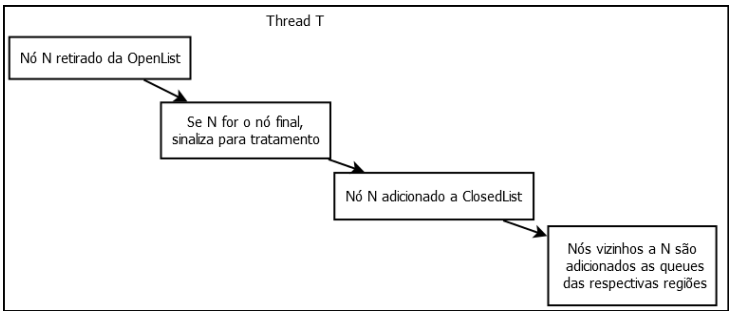

Figura 5.6: Módulo de expansão de nós.

Capítulo 6

Resultados Experimentais

Neste capítulo são apresentados os testes realizados com a implementação do *Disk-Assisted Parallel A** (Capítulo 5). Inicialmente foram feitos testes para comparar o desempenho da nova implementação em relação ao *Parallel A** (Capítulo 4). Em seguida, é feita a comparação de desempenho do *Disk-Assisted Parallel A** em duas máquinas *desktop* distintas. Após isso, são descritos testes feitos para analisar o impacto do limite de memória estabelecido sobre o desempenho da estratégia. Por fim são apresentados os resultados da execução do *Disk-Assisted Parallel A** sobre um conjunto de entrada sobre o qual o *Parallel A** excedia a capacidade de memória RAM da máquina utilizada e não conseguia finalizar sua execução.

6.1 Ambiente de Testes e Sequências utilizadas

Foram utilizadas duas máquinas para a execução dos testes. A Tabela 6.1 apresenta as configurações de *Hardware* e Sistema Operacional de cada máquina.

Tabela 6.1: Configuração das máquinas utilizadas.

Ambiente	*Hardware*	*SO*
I	Processador Intel Core i7-3770K, 3.50 GHz, 8GB RAM	Ubuntu 15.10 LTS
II	Processador Intel Core i7-3770, 3.40 GHz, 8GB RAM	Ubuntu 12.04 LTS

Para a realização dos testes foram utilizadas os conjuntos de sequências apresentadas na Tabela 6.2. Esses conjuntos fazem parte da base *Balibase* 1 [37]. A tabela mostra o nome do conjunto, o número de sequências e o tamanho da maior e menor sequência de cada conjunto.

Tabela 6.2: Sequências utilizadas nos experimentos.

Sequência	Número de sequências	Tamanho da menor sequência	Tamanho da maior sequência	Memória RAM necessária(MB)
1dlc	4	568	590	182
1gdoA	4	235	265	246
1wit	5	90	106	407
2ack	5	453	482	7063
1hvA	5	137	199	11278

6.2 Dados Experimentais e Análise dos Resultados

6.2.1 Comparação no tempo de execução e ocupação de memória do *Parallel A** e do *Disk-Assisted Parallel A**

Neste primeiro teste, foi realizada a comparação no tempo de execução e ocupação de memória do algoritmo PA* e do algoritmo proposto DAPA*, sem ativação da movimentação de dados memória-disco, para 4 conjuntos de sequências, 1*gdoA*, 1*dlc*, 1*wit* e 2*ack*. Esse teste foi realizado no ambiente *II* utilizando 8 *threads* e, para o *Disk-Assisted Parallel A**, 64 regiões por *thread*. A Tabela 6.3 mostra os tempo de execução dos dois algoritmos.

Tabela 6.3: Comparação entre *Parallel A** e *Disk-Assisted Parallel A**.

Sequência	Tempo de execução PA* (mm:ss)	Tempo de execução DAPA* (mm:ss)	Ocupação Máxima de memória RAM PA*(MB)	Ocupação Máxima de memória RAM DAPA*(MB)
1dlc	00:03,805	00:04,577	190,128	187,133
1gdoA	00:05,477	00:06,308	252,389	251,148
1wit	00:14,684	00:15,079	416,673	412,003
2ack	08:17,103	07:15,739	7358,924	7232,617

Pode ser notado que quanto maior é o consumo de memória para um conjunto de sequências, melhor é o desempenho do DAPA* em comparação ao PA*. Para o conjunto 2*ack* o DAPA* já possui tempo de execução 12% menor que o do PA*. Em todas as comparações mostradas na Tabela 6.3, o DAPA* consome menos memória RAM.

6.2.2 Teste de variação no número de *threads* e regiões por *thread*

No segundo teste, utilizando 3 sequências, 1*gdoA*, 1*dlc* e 1*wit*, o *DAPA** foi executado variando-se o número de *threads* e o número de regiões por *thread*. As Figuras 6.1 a 6.9

mostram os resultados dessas execuções. O eixo vertical representa o tempo de execução enquanto que o eixo horizontal representa o número de regiões por *thread*.

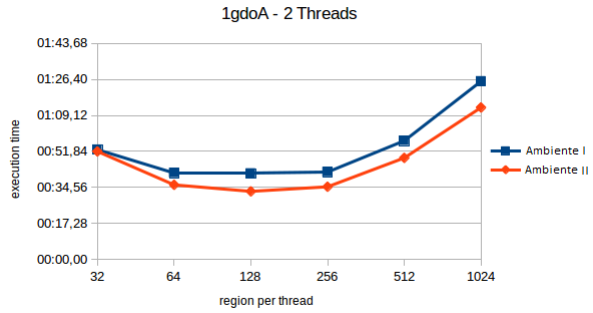

Figura 6.1: Gráfico comparativo da sequência 1gdoA para 2 *threads*.

Figura 6.2: Gráfico comparativo da sequência 1gdoA para 4 *threads*.

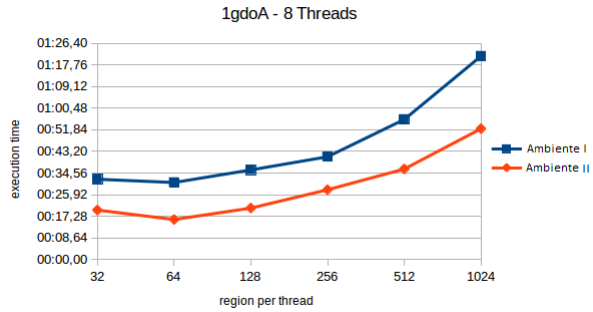

Figura 6.3: Gráfico comparativo da sequência 1gdoA para 8 *threads*.

As Figuras 6.1 a 6.3 apresentam o conjunto 1*gdoA*. O desempenho nos dois ambiente com 2 e 8 *threads* é bastante similar. Para 4 *threads*, houve uma redução abrupta de desempenho para 64, 256 e 1024 regiões por *thread* no ambiente *I*. Para o conjunto 1*gdoA*, a melhor configuração encontrada foi utilizando 8 *threads* e 64 regiões por *thread*, em ambos os ambientes.

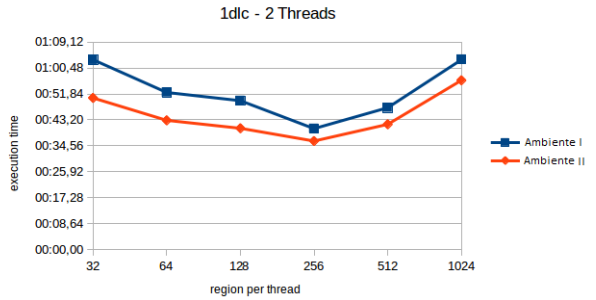

Figura 6.4: Gráfico comparativo da sequência 1dlc para 2 *threads*.

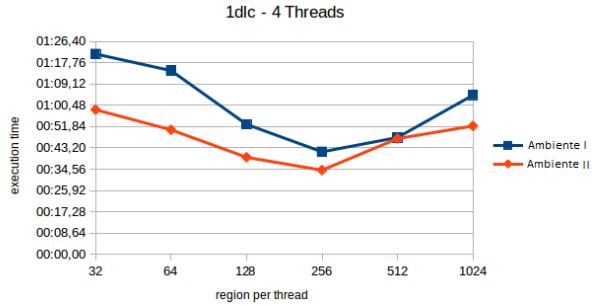

Figura 6.5: Gráfico comparativo da sequência 1dlc para 4 *threads*.

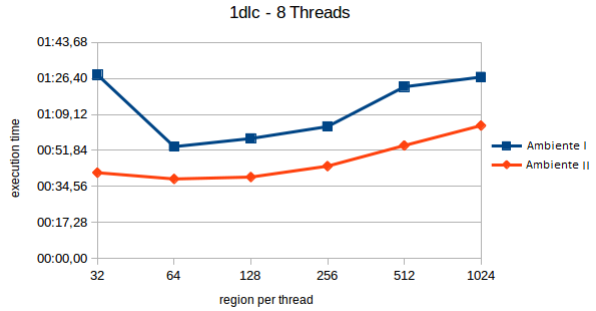

Figura 6.6: Gráfico comparativo da sequência 1dlc para 8 *threads*.

As Figuras 6.4 a 6.6 apresentam os resultados dos testes com o conjunto 1*dlc*. Para esse conjunto, ambos os ambientes apresentaram comportamentos bastante parecidos, com a configuração de 8 *threads* e 64 regiões por *thread* obtendo o melhor resultado.

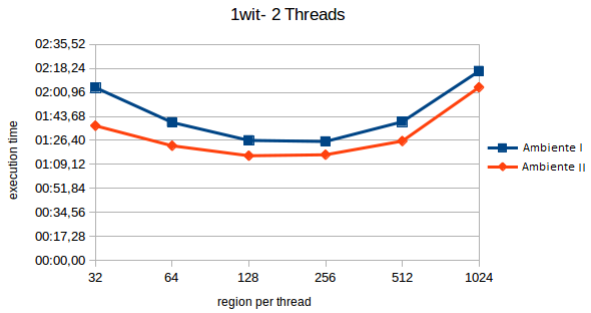

Figura 6.7: Gráfico comparativo da sequência 1wit para 2 *threads*.

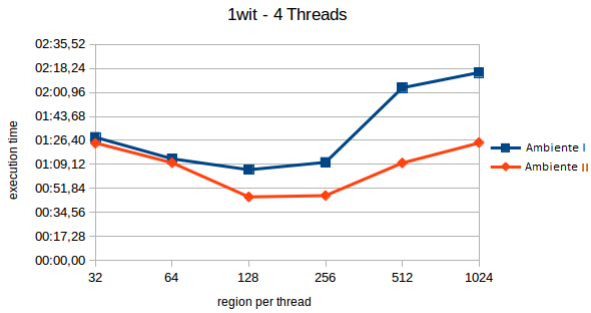

Figura 6.8: Gráfico comparativo da sequência 1wit para 4 *threads*.

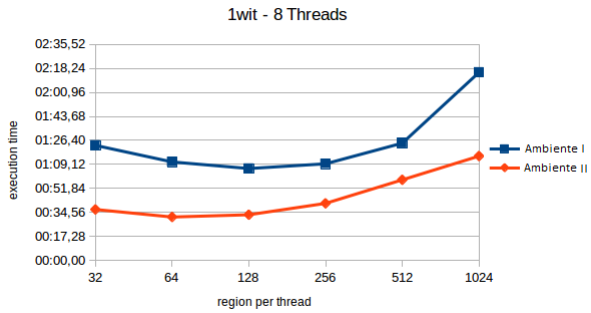

Figura 6.9: Gráfico comparativo da sequência 1wit para 8 *threads*.

As Figuras 6.7 a 6.9 apresentam os resultados para o conjunto 1*wit*. Pra esse conjunto, o ambiente I obteve o melhor resultado utilizando 8 *threads* e 128 regiões por *thread*, já o ambiente II, obteve utilizando 8 *threads* e 64 regiões por *thread*.

6.2.3 Teste de variação do limite de memória para a *Closed List*

O terceiro teste consistiu na execução do programa como valores decrescentes de limite de ocupação da *ClosedList* e foi realizado na máquina II. A Tabela 6.4 mostra os resultados do teste. Foi utilizada a sequência $1wit$, com 8 *threads* e 64 regiões por *thread* para a realização do teste. O valor de *threshold* utilizado foi de 90%, o que significa, uma vez que a ocupação de memória da *ClosedList* atinja 90% do limite informado, o DAPA* inicia a movimentação de dados memória-disco.

Tabela 6.4: Teste de variação no limite de memória.

Limite de Memória(MB)	Tempo de execução	Ocupação max da *ClosedList*(MB)	Ocupação máxima de memória(MB)
140	00:15.159	70,589	413,460
70	00:32.772	63,298	399,596
56	02:18.066	51,391	379,288
35	07:16.305	32,941	315,956
17,5	13:05.584	17,613	272,836

A Tabela 6.4 mostra que, conforme esperado, a redução no limite de memória provoca um aumento grande no tempo de execução pois as operações de disco passam a ser utilizadas.

6.2.4 Teste com sequência excedendo o limite de memória

O teste final visava testar o funcionamento do *Disk-Assisted Parallel A** para uma sequência que que exigisse mais memória que o disponível na máquina. Para tal foi escolhida o conjunto $1hvA$, que exigia do algoritmo *Parallel A** aproximadamente 11 GB.

Inicialmente, foi utilizada a configuração que obteve os melhores resultados no ambiente I no experimento 2 (Seção 6.2.2), com 8 *threads* e 64 regiões por *thread*, contudo, com esta configuração não foi possível obter o resultado. Tendo em vista esse resultado, outros testes foram realizados utilizando mais regiões por *thread* e *thresholds* menores.

Finalmente, utilizando 8 *threads*, 256 regiões por *thread*, limite da *ClosedList* de 1 GB e *Thresolhod* de 85% a execução encontrou o escore com sucesso, com um tempo de execução maior do que o esperado, ultrapassando 7 horas. O tempo de execução e memória máxima ocupada podem ser vistos na Tabela 6.5.

Tabela 6.5: Resultado do teste do DAPA*.

Tempo de execução(hh:mm:ss)	Memória máxima ocupada(MB)
7:14:27.782	6376,4

Capítulo 7

Conclusão e Trabalhos Futuros

O presente trabalho de graduação propôs, implementou e analisou a estratégia *Disk-Assisted Parallel A**, que visa permitir a solução de instâncias do problema do alinhamento múltiplo ótimo de sequências biológicas que o algoritmo *Parallel A** não conseguia resolver por uso excessivo de memória RAM.

Os resultados do experimento 1 (Seção 6.2.1) indicam que o desempenho do *Disk-Assisted Parallel A** comparado ao do *Parallel A** depende da quantidade de memória necessária do conjunto de sequências utilizado, por exemplo, para a instância $1dlc$, que requer aproximadamente 182 MB de memória RAM, o DAPA* levou 20% mais tempo para finalizar. Já para a instância $2ack$, que requer aproximadamente 7 GB de memória RAM, o DAPA* foi 12% mais rápido que o PA*.

Os resultados do experimento 4 (Seção 6.2.4) mostraram que o *Disk-Assisted Parallel A** executou para o conjunto $1hvA$, que necessitava de aproximadamente 11 GB no PA*, ocupando no máximo 6 GB, o que permitiu sua finalização na máquina II, que dispunha de 8 GB de RAM.

Como trabalhos futuros sugerimos:

- Paralelização da escrita de *ClosedLists* em disco, de maneira a reduzir o tempo necessário para finalização de uma iteração do DAPA*;
- Adição de um *threshold* inferior, de forma que, uma vez que o *threshold* superior tenha sido atingido, as *threads* transfiram para disco dados o suficiente para diminuir a ocupação de memória RAM à esse segundo *threshold*, evitando que o algoritmo fique constantemente tendo que ler e escrever no disco quando estiver trabalhando na borda do *threshold*;
- Estudo do impacto do uso de outros algoritmos de *hash* e valores de *shift* no desempenho do *Disk-Assisted Parallel A**, uma vez que uma função que atribua nós

próximos espacialmente a mesma região poderia diminuir o número de trocas de região ativa, diminuindo o número de operações de leitura do disco para a memória;

- Estudo mais detalhado do impacto no desempenho do algoritmo da variação no número de *threads* e no número de regiões por *thread* para conjuntos de sequências com alto custo de memória.

Referências

[1] Stephen F Altschul. Leaf pairs and tree dissections. *SIAM journal on discrete mathematics*, 2(3):293–299, 1989. 9

[2] Stephen F Altschul, Raymond J Carroll, e David J Lipman. Weights for data related by a tree. *Journal of molecular biology*, 207(4):647–653, 1989. 7, 8, 9

[3] Anne Bahr, Julie D Thompson, J-C Thierry, e Olivier Poch. Balibase (benchmark alignment database): enhancements for repeats, transmembrane sequences and circular permutations. *Nucleic Acids Research*, 29(1):323–326, 2001. 35, 36

[4] Boost. Boost Multi-index Containers Library. `http://www.boost.org/doc/libs/1_55_0/libs/multi_index/doc/tutorial/index.html`. Acessado em 02/12/2015. 30

[5] Ethan A Burns, Matthew Hatem, Michael J Leighton, e Wheeler Ruml. Implementing fast heuristic search code. In *SOCS*, 2012. 32

[6] Humberto Carrillo e David J Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988. 5, 6, 10, 11

[7] Margaret O Dayhoff e Robert M Schwartz. A model of evolutionary change in proteins. In *In Atlas of protein sequence and structure*, 1978. 6, 7

[8] Jack Dongarra. Trends in high performance computing: a historical overview and examination of future developments. *Circuits and Devices Magazine, IEEE*, 22(1):22–27, 2006. 27

[9] Richard Durbin, Sean R Eddy, Anders Krogh, e Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998. 4, 6, 7

[10] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972. 27

[11] Hector Garcia-Molina, Jeffrey D Ullman, e Jennifer Widom. *Database system implementation*, volume 654. Prentice Hall Upper Saddle River, NJ:, 2000. 18

[12] Sandeep K Gupta, John D Kececioglu, e Alejandro A Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2(3):459–472, 1995. 12

[13] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997. 10

[14] Peter E Hart, Nils J Nilsson, e Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968. 14, 17

[15] Matthew Hatem e Wheeler Ruml. External memory best-first search for multiple sequence alignment. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013. 19, 20, 21, 22, 23

[16] Steven Henikoff e Jorja G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992. 6

[17] Kai Hwang e Zhiwei Xu. *Scalable parallel computing: technology, architecture, programming*. McGraw-Hill, Inc., 1998. 28

[18] Andrew D Johnson e Christopher J O'Donnell. An open access database of genome-wide association results. *BMC medical genetics*, 10(1):6, 2009. 1

[19] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985. 17

[20] Richard E Korf. Divide-and-conquer bidirectional search: first results. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 1184–1189. Morgan Kaufmann Publishers Inc., 1999. 21

[21] Richard E Korf. Best-first frontier search with delayed duplicate detection. In *Proceedings of the 19th national conference on Artifical intelligence*, pages 650–657. AAAI Press, 2004. 17, 18, 19, 21

[22] Richard E Korf. Linear-time disk-based implicit graph search. *Journal of the ACM (JACM)*, 55(6):26, 2008. 19, 21

[23] David J Lipman, Stephen F Altschul, e John D Kececioglu. A tool for multiple sequence alignment. *Proceedings of the National Academy of Sciences*, 86(12):4412–4415, 1989. 9

[24] Shingo Masuno, Tsutomu Maruyama, Yoshiki Yamaguchi, e Akihiko Konagaya. An FPGA Implementation of Multiple Sequence Alignment Based on Carrillo-Lipman Method. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 489–492. IEEE, 2007. 12

[25] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966. 34

[26] David Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2nd edition, 2013. 1, 2, 10

[27] Robert Niewiadomski, José Nelson Amaral, e Robert C Holte. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *proceedings of the 21st national conference on Artificial intelligence-Volume 2*, pages 1039–1044. AAAI Press, 2006. 2, 21, 24, 26

[28] OpenMP. `http://openmp.org`. Acessado em 29/11/2015. 28

[29] Gregory F Pfister. *In search of clusters*. Prentice-Hall, Inc., 1998. 28

[30] POSIX Threads. Ieee std 1003.1c™-1995 ieee standard for information technology–portable operating system interface (posix®) - system application program interface (api) amendment 2: Threads extension (c language). `http://standards.ieee.org/findstds/interps/1003-1c-95_int/`. Acessado em 29/11/2015. 28

[31] Andrew W Roscoe. *A classical mind: essays in honour of CAR Hoare*. Prentice Hall International (UK) Ltd., 1994. 17, 18, 21

[32] Stuart Russell e Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010. 2, 15, 16

[33] Naruya Saitou e Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987. 9

[34] John L Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM Journal on Applied Mathematics*, 49(5):1552–1566, 1989. 17

[35] Daniel Sundfeld, George Teodoro, Alba Cristina Melo, et al. Parallel a-star multiple sequence alignment with locality-sensitive hash functions. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on*, pages 342–347. IEEE, 2015. 2, 7, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38

[36] The Message Passing Interface Standard. `http://www.mpi-forum.org/`. Acessado em 29/11/2015. 28

[37] Julie D Thompson, Frédéric Plewniak, e Olivier Poch. Balibase: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics*, 15(1):87–88, 1999. 21, 44

[38] Alex Vrenios. *Linux cluster architecture*. Sams, 2002. 28

[39] Lusheng Wang e Tao Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994. 1, 10