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

Estratégia Distribuída Híbrida em *Cluster Multicore* Heterogêneo para Alinhamento Múltiplo de Sequências Biológicas com o DIALIGN-TX

Emerson de Araújo Macedo

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

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

Coorientador
Prof. Dr. Gerson Henrique Pfitscher

Brasília
2010

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Maurício Ayala-Rincón

Banca examinadora composta por:

Profª. Drª. Alba Cristina Magalhães Alves de Melo (Orientadora) – CIC/UnB
Prof. Dr. Mário Antônio Ribeiro Dantas – UFSC
Prof. Dr. Ricardo Pezzuol Jacobi – CIC/UnB

CIP – Catalogação Internacional na Publicação

Emerson de Araújo Macedo.

Estratégia Distribuída Híbrida em *Cluster Multicore* Heterogêneo para Alinhamento Múltiplo de Sequências Biológicas com o DIALIGN-TX/ Emerson de Araújo Macedo. Brasília : UnB, 2010.
101 p. : il. ; 29,5 cm.

Tese (Mestre) – Universidade de Brasília, Brasília, 2010.

1. Bioinformática, 2. Alinhamento de sequências,
3. programação paralela, 4. Computação de alto desempenho

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

Estratégia Distribuída Híbrida em *Cluster Multicore* Heterogêneo para Alinhamento Múltiplo de Sequências Biológicas com o DIALIGN-TX

Emerson de Araújo Macedo

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Profª. Drª. Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Mário Antônio Ribeiro Dantas
UFSC

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Maurício Ayala-Rincón
Coordenador do Mestrado em Informática

Brasília, 25 de Outubro de 2010

Dedicatória

Little Princess Bunny, this is for you. It's ready because you believed in me all the way. Without your unconditional love, support and patience, it would never be ready. It's not my victory, it's our victory. You're my inspiration. I love you.

Agradecimentos

Agradeço a toda minha família, especialmente meu pai e minha mãe, por terem me ensinado o valor da humildade, perseverança e do amor a Deus.

Agradeço a Alba Cristina Melo, professora da Universidade de Brasília, por sua orientação motivante e consistente. Seu entusiasmo pela pesquisa acadêmica, seu planejamento metódico e coerente mantiveram meu foco no objetivo. Sua confiança em minha capacidade me fez acreditar até o fim e sua paciência inigualável tornou essa dissertação de mestrado uma realidade.

Agradeço ao amigo Gerson Henrique Pfitscher, professor da Universidade de Brasília, por sua coorientação objetiva, visão pragmática e pela inestimável ajuda na instalação e configuração do *cluster multicore* heterogêneo no Laboratório de Processamento de Alto Desempenho da Universidade de Brasília, sem o qual os testes de implementação não teriam sido possíveis.

Agradeço a Mário Antônio Ribeiro Dantas, professor da Universidade Federal de Santa Catarina, e a Ricardo Pezzuol Jacobi, professor da Universidade de Brasília, por terem aceitado participar da banca de defesa desta dissertação e pelas modificações sugeridas a este documento.

Agradeço à Secretaria de Pós-Graduação, em especial a Rosa Amariles, pelo profissionalismo e colaboração prestativa durante os intermináveis trâmites burocráticos ao longo de todos esses anos.

Resumo

O Alinhamento Múltiplo de Sequências (AMS) é um problema importante em Bioinformática, permitindo a interpretação de árvores filogenéticas, a identificação de domínios e padrões conservados e a predição de estruturas secundárias. Como o AMS é um problema NP-Difícil, heurísticas são utilizadas. O programa DIALIGN-TX implementa uma heurística iterativa para calcular o AMS em três fases. A fase 1 calcula todas as comparações par a par das sequências de entrada, exigindo a maior parcela do tempo de execução para o cálculo do AMS. Esta fase possui grande potencial para execução em paralelo, pois as comparações par a par são independentes entre si.

Os *clusters multicore* heterogêneos surgem da expansão gradual de ambientes compostos por *clusters multicore* homogêneos. Para explorar as características *multicore* e heterogênea desse sistema em *cluster*, é intuitivo que o emprego de um modelo de programação híbrido com trocas de mensagens e memória compartilhada seja mais apropriado, bem como de uma estratégia de alocação de tarefas que permita lidar com as diferentes capacidades de processamento de seus nós.

A presente dissertação propõe e avalia um estratégia distribuída híbrida para que a ferramenta DIALIGN-TX seja executada num *cluster multicore* heterogêneo. A estratégia proposta foi implementada em um *cluster multicore* heterogêneo com três nós com capacidades de processamento e velocidades de *clock* diferentes. Foi utilizado um modelo híbrido de programação com troca de mensagens para a comunicação entre os nós e memória compartilhada para comunicação entre os *cores* de um mesmo nó. Foram implementadas três novas estratégias de alocação de tarefas, chamadas *Hybrid Fixed* (HFixed), *Hybrid Self-Scheduling* (HSS) e *Hybrid Weighted Factoring* (HWF).

Os resultados obtidos mostraram que a solução proposta consegue reduzir de maneira bastante significativa o tempo de execução da fase 1 do AMS do DIALIGN-TX. Além disso, mostraram que a escolha de uma política de alocação de tarefas adequada é de fundamental importância para o desempenho da solução.

Palavras-chave: Bioinformática, Alinhamento de sequências, programação paralela, Computação de alto desempenho

Abstract

The Multiple Sequence Alignment (MSA) is an important problem in Bioinformatics, allowing interpretation of phylogenetic trees, identification of domains and conserved motifs and prediction of secondary structures. As the MSA is an NP-Hard problem, heuristics are used. The DIALIGN-TX program implements an iterative heuristic to calculate the MSA in three phases. Phase 1 calculates all pairwise comparisons of the input sequences, requiring the largest portion of execution time for the calculation of MSA. This phase has great potential for parallel execution, since its pairwise comparisons are independent from each other.

The heterogeneous multicore clusters arise from the gradual expansion of environments composed of homogeneous multicore clusters. To explore the multicore and heterogenous characteristics of that cluster system, it is intuitive that the use of a hybrid programming model with message passing and shared memory is more appropriate, as well as a task allocation strategy for addressing the different computation powers in its nodes.

This dissertation proposes and evaluates a hybrid distributed strategy that allows DIALIGN-TX to be executed in a heterogeneous multicore cluster. The proposed strategy was implemented in a heterogeneous multicore cluster with three nodes with diferent processing capabilities and clock speeds. A hybrid programming model with message passing for communication among nodes and shared memory for communication among cores of the same node was used. Moreover, three new strategies for task allocation were implemented: Hybrid Fixed (HFixed), Hybrid Self-Scheduling (HSS) and Hybrid Weighted Factoring (HWF).

The results showed that the proposed solution can reduce quite significantly the execution time of the first phase of the MSA of DIALIGN-TX. Furthermore, they also showed that choosing an appropriate task allocation centeringpolicy has fundamental importance for the performance of the solution.

Keywords: Bioinformatics, Sequence Alignment, Parallel Programming, High-Performance Computing

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

AMS	*Alinhamento Múltiplo de Sequências*
AWF	*Adaptive Weighted Factoring*
BLAST	*Basic Local Alignment Search Tool*
BLOSUM	*BLock SUbstitution Matrix*
CHAOS	*CHAin Of Sequences*
Clustal	*Cluster alignment program*
DIALIGN	*DIagonal ALIGNment*
FASTA	*FAST-All sequence alignment algorithm*
FASTP	*Versão do FASTA para alinhamento de Proteínas*
FSC	*Fixed-Size Chunking*
GGSEARCH	*Global-query Global-database similarity SEARCH tool*
GPU	*Graphics Processing Unit*
GSS	*Guided Self-Scheduling*
HFixed	*Hybrid Fixed*
HSS	*Hybrid Self-Scheduling*
HWF	*Hybrid Weight Factoring*
LAGAN	*Limited Area Global Alignment of Nucleotides*
LPAD	*Laboratório de Processamento de Alto Desempenho*
MPI	*Message Passing Interface*
MPICH	*Message Passing Interface CHameleon*
MSA	*Multiple Sequence Alignment*
NCBI	*National Center for Biotechnology Information*
NW	*Needleman-Wunch*
OpenMP	*Open specifications for Multi-Processing*

PAM	*Percent Accepted Mutation*
PC	*Personal Computer*
Pfam	*Protein families database*
PGAS	*Partitioned Global Address Sapce*
PIWI	*P-element Induced WImpy testes in Drosophila*
PTC	*Parallel T-Coffee*
SMP	*Symmetric Multi-Processor*
SS	*Self-Scheduling*
SSEARCH	*Sequence Similarity SEARCH tool*
SW	*Smith-Waterman*
TC	*T-Coffee*
T-COFFEE	*Tree based Consistency Objective Function For alignmEnt Evaluation*
TSS	*Trapezoidal Self-Scheduling*
UniProt	*Universal Protein Resource*
UniRef	*Uniprot Reference database*
WF	*Weight Factoring*

Capítulo 1

Introdução

A Bioinformática é uma área que envolve a participação de disciplinas de diversas áreas, entre elas Matemática e Ciência da Computação, para prover ferramentas estatísticas, algoritmos, bases de dados e interfaces de usuários que permitam aos biólogos realizarem tarefas, como o alinhamento de sequências biológicas, e obterem resultados que permitam novas inferências e deem novo sentido para os dados biológicos.

O alinhamento de sequências é uma das tarefas mais comuns em Bioinformática [63]. A busca em bases de dados biológicos e a predição de estruturas secundárias são exemplos são exemplos de procedimentos que dependem da comparação de sequências. Se a intenção é a busca por homólogos em uma base de dados, as sequências podem ser comparadas duas a duas (*pairwise*). Por outro lado, se o objetivo for visualizar o efeito da evolução em uma família inteira de proteínas, então múltiplas sequências devem ser alinhadas.

O alinhamento múltiplo de sequências biológicas (AMS) é um problema importante em Bioinformática, pois permite aos biólogos a interpretação de árvores filogenéticas, a identificação de domínios e padrões conservados e a predição de estruturas secundárias [63].

O AMS é um problema NP-difícil e, por isso, heurísticas são utilizadas. Dentre as heurísticas que tratam o problema do AMS, as abordagens progressiva e iterativa são as mais relevantes para esta dissertação.

A heurística progressiva possui três etapas distintas. Primeiro, calcula os alinhamentos par a par de todas as sequências biológicas. Segundo, constrói uma árvore binária para agrupar em suas folhas as sequências de acordo com as suas semelhanças [27]. E, finalmente, a árvore binária é utilizada como guia para alinhar as sequências, folha a folha, progressivamente. Um exemplo bem conhecido de ferramenta que implementa esta heurística é o programa ClustalW [41].

A heurística iterativa baseia-se na ideia de que a solução para um problema pode ser encontrada modificando-se uma solução sub-ótima que já exista [63]. Assim, calcula-se um alinhamento inicial para que, em seguida, seja refinado várias vezes até que não seja mais possível encontrar alinhamento melhor.

Um exemplo de método iterativo é a ferramenta DIALIGN. Apresen-

tado em 1996, com o nome de DIALIGN 1.0 [56, 57], este programa calcula o AMS começando pela comparação de todos os pares de sequências biológicas de entrada à procura de pares de segmentos com alto grau de semelhança (fragmentos). Em seguida, atribui um escore para cada fragmento encontrado. Finalmente, constrói o alinhamento final acrescentando cada fragmento encontrado de acordo com os maiores pesos recebidos. Desde a sua primeira versão, o DIALIGN tem sido alvo de aperfeiçoamento ao longo dos anos, como as modificações em sua função objetivo apresentadas no DIALIGN 2.0 [51], a incorporação de abordagens com âncoras [11] e uma versão paralela chamada DIALIGN-P, pois a obtenção de resultados no cálculo do AMS requer alto poder computacional [69]. Depois, sofreu uma reimplementação na versão DIALIGN-T [76] e, finalmente, sua última versão chama-se DIALIGN-TX [74].

O DIALIGN-TX incorpora características da heurística progressiva na fase de construção do AMS, incluindo a criação de uma árvore binária e a opção da formação do alinhamento pelo modo iterativo tradicional ou progressivamente. A nosso conhecimento, não existe versão paralela ou distribuída para o DIALIGN-TX.

Atualmente, a popularização dos processadores *multicore* em computadores pessoais tornou esta tecnologia atraente para a construção de ambientes computacionais de alto desempenho, como *clusters*. O site *TOP500 Supercomputers Sites* publica uma compilação semestral dos 500 sites com os sistemas computacionais instalados de maior capacidade de processamento do mundo [21]. Em sua mais recente lista de junho de 2010, observou-se que 99,76% dos *clusters* possuía processadores *multicore* em seus nós.

Em geral, a expansão de qualquer sistema computacional demanda uma substituição de equipamentos, porém os custos proibitivos de uma total atualização do ambiente admitem apenas substituições parciais por equipamentos mais modernos. Esta inclusão de equipamentos com arquiteturas diferentes das pré-existentes acaba permitindo o surgimento de ambientes *multicore* antes projetados para serem homogêneos e que, agora, tornam-se heterogêneos. É neste contexto que entram os *clusters multicore* heterogêneos, tentando-se descobrir técnicas e estratégias para tirar um melhor proveito de todo o seu poder computacional.

Adequando-se a este cenário de heterogeneidade em um *cluster multicore*, é natural inferir que uma solução adequada para este ambiente utilize um modelo de programação coerente com as suas características. Neste caso, um modelo híbrido de programação seria a escolha intuitiva, pois permite que os nós do *cluster* possam se comunicar via troca de mensagens, como em qualquer *cluster* homogêneo, e viabiliza a comunicação entre os *cores* de um mesmo nó via memória compartilhada, ou seja, todos os recursos de processamento de um *cluster multicore* heterogêneo, ou até homogêneo, serão melhor aproveitados.

Finalmente, utilizando um modelo híbrido de programação adequado para um ambiente em *cluster multicore* heterogêneo, pode-se escolher uma estratégia de alocação de tarefas para lidar com as diferentes capacidades de processamento de um sistema *multicore* heterogêneo. Políticas de alo-

cação de tarefas conhecidas, como *Fixed* [72], *Self-scheduling* [78] e *Weight factoring* [42] poderiam ser adaptadas para a execução distribuída em um *cluster multicore* heterogêneo.

A presente dissertação de mestrado tem por objetivo propor e avaliar uma estratégia distribuída híbrida para execução em um *Cluster Multicore* Heterogêneo para tratar o problema do alinhamento múltiplo de sequências biológicas utilizando a ferramenta DIALIGN-TX. As principais contribuições deste trabalho são:

- Solução distribuída MPI/OpenMP para execução do DIALIGN-TX em *clusters Multicore* heterogêneos: esta estratégia proposta nesta contribuição usa um modelo híbrido de programação com trocas de mensagens e memória compartilhada para a execução distribuída da ferramenta DIALIGN-TX de alinhamento de sequências biológicas, em um *cluster multicore* heterogêneo para reduzir seu tempo de execução em comparação com a versão original da ferramenta. Esta solução pode ser aplicada também em um *cluster multicore* homogêneo, pois uma de suas etapas identifica o poder computacional disponível no *cluster*, tanto a quantidade de nós quanto a quantidade de *cores*, usando estes dados para distribuir as tarefas entre os nós do *cluster multicore*.

- Solução OpenMP para execução do DIALIGN-TX em máquinas *multicore*: esta estratégia usa um modelo de programação com memória compartilhada através da biblioteca OpenMP para execução do DIALIGN-TX em computadores com processadores *multicore*. Com o baixo custo e a popularização destes processadores *multicore* é possível que qualquer biólogo tenha acesso a um computador pessoal com processador *multicore* e, assim, poderá utilizar a versão OpenMP do DIALIGN-TX com desempenho superior à versão original.

- Múltiplas estratégias de alocação de tarefas em *clusters multicore* heterogêneos: a última contribuição desta dissertação são as abordagens híbridas de alocação de tarefas utilizadas na solução distribuída híbrida. As políticas de alocação de tarefas bem conhecidas *Fixed* [72], *Self-Scheduling* [78] e *Weight Factoring* [42] foram adaptadas para o modelo híbrido de programação com trocas de mensagens e memória compartilhada para se adequarem as características *multicore* e heterogênea do *cluster* empregado na solução distribuída MPI/OpenMP. Os nomes das novas estratégias são: *Hybrid Fixed* (HFixed), *Hybrid Self-Scheduling* (HSS) e *Hybrid Weight Factoring* (HWF).

O presente documento está organizado em 7 capítulos. O Capítulo 2 discorre sobre a Comparação de pares de sequências biológicas e seus principais algoritmos e ferramentas utilizados. O Capítulo 3 aborda o problema do Alinhamento múltiplo de sequências (AMS) apresentando as heurísticas progressiva e iterativa utilizadas para tratá-lo e as ferramentas que as implementam, como o DIALIGN-TX. A seguir, no Capítulo 4, são apresentadas as estratégias paralelas para alinhamento de sequências que foram

adaptadas a partir de suas versões originais apresentadas no Capítulo 3. O Capítulo 5 detalha a proposta desta dissertação, descrevendo o projeto da Estratégia Distribuída Híbrida em *Cluster Multicore* Heterogêneo para calcular o AMS utilizando a ferramenta DIALIGN-TX. No Capítulo 6, são mostrados os resultados obtidos durante os testes com a execução *standalone* e distribuída da estratégia proposta e uma análise dos dados é fornecida para cada caso. E, finalmente, o Capítulo 7 apresenta a Conclusão e algumas ideias para trabalhos futuros.

Capítulo 2

Comparação de pares de sequências biológicas

Em Bioinformática, a comparação de pares de sequências é empregada na análise de bases de dados biológicos. Sequências de DNA e proteínas são os alvos mais comuns desta técnica, que permite a procura por uma sequência desejada, a identificação de ancestrais ou ainda evoluções funcionais entre espécies [32].

Uma das primeiras fases de um trabalho que envolva a análise de sequências em Bioinformática é a identificação do grau de similaridade existente entre as sequências biológicas envolvidas, já que identificar sequências similares frequentemente significa identificar estruturas e/ou funções similares.

Existem dois tipos básicos de alinhamentos: global e local. O alinhamento global é geralmente empregado quando as sequências são altamente semelhantes e o alinhamento de pontuação máxima é obtido considerando as sequências como um todo. O algoritmo exato de Needleman-Wunsch (NW) [62] é o exemplo mais conhecido.

No alinhamento local, o alinhamento máximo é obtido a partir de qualquer par de subsequências, sendo muito utilizado para identificar trechos altamente conservados entre dois genomas. O algoritmo exato de Smith-Waterman (SW) [73] é o seu exemplo mais conhecido.

Sejam S_1 e S_2 duas sequências biológicas de entrada. Simplificadamente, o alinhamento global entre S_1 e S_2 pode ser feito da seguinte forma: caso as sequências tenham tamanhos diferentes, serão inseridas lacunas (ou *gaps*) em cada uma para que passem a ter o mesmo tamanho. Em seguida, S_1 e S_2 serão dispostas uma acima da outra de modo a ressaltar as similaridades.

Agora, a partir dos alinhamentos possíveis entre S_1 e S_2, é preciso decidir qual deles representa a comparação mais adequada, ou seja, qual é o melhor. Para tanto, um critério objetivo que utilize uma função de pontuação para classificar cada alinhamento é utilizado. Assim, por exemplo, sejam os seguintes alinhamentos possíveis Al_1 e Al_2 entre $S_1 = ACTAGACGA$ e $S_2 = AACGTAG$:

a) Al_1:

S_1 =ACATCGC-AG

S_2 =A-A-CG-TAG

b) Al_2:

S_1 =A-CATCGCAG

S_2 =AAC-G-T-AG

Se forem considerados valores para cada coincidência(*match*), não coincidência(*mismatch*) e *gaps*, seria possível atribuir uma função de pontuação para cada alinhamento e inferir qual deles representaria a melhor escolha. Desta forma, sejam as seguintes pontuações:

- $gap = -1$;
- $match = 2$;
- $mismatch = -2$.

Para Al_1, ter-se-ia $4\ gaps{+}6\ matches{+}0\ mismatches$, somando-se 8 pontos no total. Agora Al_2, $4\ gaps + 4\ matches + 2\ mismatches$, obtendo-se 0 pontos. Neste caso, o melhor alinhamento seria a opção *(a)*.

A partir disso, surge o problema de como se obter o *alinhamento ótimo*, ou seja, obter o alinhamento com a pontuação máxima entre duas sequências a partir de um critério de pontuação bem definido.

Nas seções a seguir, os algoritmos NW e SW serão explicados em mais detalhes.

2.1 Algoritmo de Needleman-Wunsch

O algoritmo proposto por Needleman e Wunsch (NW) [62], faz a comparação entre os caracteres do par de sequências ao longo de toda sua extensão, testando cada uma das combinações possíveis para encontrar similaridades entre elas. Para isso, emprega a técnica de programação dinâmica para armazenar os melhores alinhamentos parciais obtidos e, a partir destes, determinar qual será o melhor alinhamento para solucionar a comparação como um todo. Como saída, o algoritmo fornece o alinhamento global ótimo.

2.2 Algoritmo de Smith-Waterman

Em contra-partida, Smith e Waterman (SW) [73] propuseram um alterações no algoritmo NW para identificar, numa comparação de duas sequências de DNA ou proteínas, os alinhamentos bem definidos de subsequências,

porém com quaisquer tamanhos, começando e terminando em qualquer posição. Desta forma, o algoritmo SW utiliza também a técnica de programação dinâmica para garantir que, dentre todas as comparações possíveis, um alinhamento ótimo local, e não global, seja encontrado [73].

O algoritmo SW se utiliza de um sistema de escores para marcar cada uma das similaridades positivas (*matches*) ou negativas (*mismatches*) encontradas entre cada par comparado. Penalidades também são empregadas caso um caractere seja comparado ao um espaço em branco durante um alinhamento, ou seja, um *gap* fora encontrado.

Para entender o funcionamento deste algoritmo, seja $x = \{A, G, A, T, A\}$ e $y = \{C, A, G, A, T\}$ exemplos de sequências de DNA. Considere também que o sistema de escores e penalidades empregue os seguintes escores e penalidade ao comparar dois caracteres de x e y:

- $match : +1$;
- $mismatch : -1$;
- $gap : -2$.

Para encontrar o alinhamento local ótimo, as sequências x e y são distribuídas numa matriz F da mesma forma que no algoritmo NW, com índices i e j para cada sequência sendo comparada. O valor da célula $F(i, j)$ corresponde ao escore do melhor alinhamento entre os dois segmentos x e y que se iniciam no índice i e terminam em j.

A primeira diferença entre os algoritmos SW e NW se mostra na possibilidade de cada célula de F assumir o valor 0 ao invés de valores negativos como no alinhamento global, significando que um novo alinhamento pode estar se iniciando a partir daquela célula. A outra diferença é que, neste procedimento, o alinhamento pode terminar em qualquer célula de F, diferente do global, que procura do final da matriz até seu início e percorre todas as possibilidades.

Os máximos escores obtidos em cada célula de F, segundo o algoritmo SW, são calculados utilizando a função

$$F(i,j) = max \begin{cases} 0, \\ F(i-1, j-1) \ + \ s(x_i, y_i), \\ F(i-1, j) \ - \ g, \\ F(i, j-1) \ - \ g \end{cases} \tag{2.1}$$

onde o maior valor, dentre as quatro possibilidades acima, seria considerado o máximo escore da comparação entre o caractere da posição i da sequência x e o caractere da posição j da sequência y em F. O valor de $s(x_i, y_j)$ será aquele adotado para *matches* ou *mismatches*, ou seja, $+1$ ou -1 neste exemplo. O valor de g corresponderá ao gap associado àquele par de caracteres sendo comparados. Neste ponto, caso o valor final obtido não seja igual a zero, um ponteiro será acrescentado à célula $F(i, j)$ para indicar qual das posições anteriores - $F(i-1, j-1)$, $F(i-1, j)$ ou $F(i, j-1)$ - foi considerada no cálculo no máximo escore de $F(i, j)$.

	x	A	G	A	T	A
y	0	0	0	0	0	0
C	0	0	0	0	0	0
A	0	↖1	0	↖1	0	↖1
G	0	0	↖2	0	0	0
A	0	↖1	0	↖3	0	↖1
T	0	0	0	0	**↖4**	0

Figura 2.1: Matriz de escores máximos obtida na comparação das sequências $x = \{A, G, A, T, A\}$ e $y = \{C, A, G, A, T\}$, utilizando o algoritmo SW. A célula com maior escore em toda a matriz está em cinza com bordas em negrito.

O início do processo se dá com o preenchimento da primeira linha e coluna, $F(i, 0)$ e $F(0, j)$, com o valor 0 (zero), já que este algoritmo não permite valores negativos como no alinhamento global. Em seguida, a comparação de cada par de caractere seguirá o cálculo da função F. Veja o caso do par $'AA'$ em $F(1, 2)$. Haverá um *match*, o valor de similaridade s será 1 e o valor de $F(i - 1, j - 1) = F(0, 1) = 0$. Como as outras três possibilidades para $F(1, 2)$ serão menores que 1, então o máximo escore obtido será 1, dado pelo *match* em $F(1 - 1, 2 - 1) + s(x_1, y_2) = F(0, 1) + 1 = 0 + 1 = 1$. Um ponteiro em $F(1, 2)$ indicará a célula $F(i - 1, j - 1)$ utilizada no cálculo do escore considerado máximo.

Seguindo o algoritmo em todas as células de F, teremos como resultado a matriz da figura 2.1.

Observe que a célula em destaque na figura 2.1, $F(4, 5)$, contém o maior escore obtido em toda a matriz. Depois que todas as células foram preenchidas, é a partir daquela contendo o maior escore de toda a matriz, que o algoritmo Smith-Waterman inicia o *traceback* [73], seguindo os ponteiros até encontrar uma célula que aponta para a anterior contendo o valor 0(zero) e, então, pára o *traceback*. O alinhamento local ótimo será dado pela subsequências de x e y correspondentes às células do *traceback*. A figura 2.2, ilustra o *traceback* nas células com bordas em negrito e os alinhamentos locais ótimos resultantes, $x' = \{A, G, A, T\}$ e $y' = \{A, G, A, T\}$, em cinza escuro.

O algoritmo SW é muito empregado em comparações de pares de sequências à procura por semelhanças, porém seus sucessivos cálculos demandam muito processamento e fazem com que seu uso seja recomendado para situações específicas, nas quais encontrar a solução ótima seja mais importante do que obter a solução rapidamente.

As ferramentas como o SSEARCH e o GGSEARCH implementam os

	x	A	G	A	T	A
y	0	0	0	0	0	0
C	0	0	0	0	0	0
A	0	↖1	0	↖1	0	↖1
G	0	0	↖2	0	0	0
A	0	↖1	0	↖3	0	↖1
T	0	0	0	0	↖4	0

Figura 2.2: Matriz F de escores máximos obtida na comparação das sequências $x = \{A, G, A, T, A\}$ e $y = \{C, A, G, A, T\}$, utilizando o algoritmo SW. A célula com maior escore (em cinza e bordas em negrito) marca o início do *traceback* (em negrito) para obtenção dos alinhamentos locais ótimos (em cinza escuro), neste exemplo, $x' = AGAT$ e $y' = AGAT$.

algoritmos SW e NW, respectivamente, e os utilizam na comparação de sequências biológicas [65, 23].

2.3 Heurísticas para comparação de sequências

Os algoritmos de alinhamentos de sequências através da programação dinâmica possuem alta demanda de processamento e memória devido à sua complexidade quadrática de tempo e espaço. Com isso, algumas propostas foram apresentadas para obter resultados satisfatórios porém com tempos de resposta muito melhores.

Os dois métodos que mais se destacaram foram o FASTA e o BLAST, sendo, em média, 50 vezes mais rápidos que os algoritmos de programação de dinâmica [59, 6]. Em seu desenvolvimento foram empregadas heurísticas que quase sempre funcionam para encontrar sequências relacionadas, porém não dão garantia de que o alinhamento encontrado seja ótimo, como ocorre na programação dinâmica [59].

2.3.1 FASTA

Em 1988, Lipman e Pearson [47], propuseram um método heurístico para realizar buscas em um banco de dados biológicos e procurar sequencialmente por similaridades entre uma *string* fornecida como entrada e aquelas contidas na base. Chamado de FASTA, este método permite a comparação entre sequências de proteínas e DNA.

O FASTA aperfeiçoou a versão anterior apresentada em 1985, FASTP [46], que efetua buscas por similaridades apenas entre sequências de proteínas, ao introduzir maior capacidade de reconhecer sequências distantes (*sensitivity*) e ignorar *matches* entre aquelas não relacionadas (*selectivity*).

```
>EM000111 |acc=EM000111|descr=seq exemplo |len=320
ACAAGATGCCATTGTCCCCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCC
CCTGGAGGGTGGCCCCACCACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGC
CTCCTGACTTTCCTCGCTTTGAGTGGACCTCCCAGGCCAGTGCCGGGCCCCTCATAGGAGAGG
AAGCTCGGGAGGTGGCCAGGAAGGCGCACCCCCCCAGCAATCCGCGCGCCGGGACAGAATGCC
CTGCAGGAACTTCTTCTGGTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAG
TTTAATTACAGACCTGAA
```

Figura 2.3: Exemplo de sequência em formato FASTA.

A sequência, ou arquivo de sequências, submetida ao programa, deve estar no *formato FASTA*, hoje amplamente utilizado até por outras ferramentas de busca, como o BLAST [61]. A estrutura deste formato consiste de uma primeira linha descritiva, sempre identificada com um caractere de maior que $>$ e as linhas seguintes consideradas dados da sequência. A figura 2.3 ilustra um exemplo.

Em geral, o FASTA opera da seguinte forma: primeiro, realiza-se uma busca na base de dados e na sequência de entrada procurando-se por k-tuplas, ou seja, palavras ou padrões de comprimento definido e igual a k. Depois, executa-se um alinhamento local entre os padrões encontrados em todas as sequências, utilizando-se de uma matriz de pesos, como a BLOSUM50 [32]. A busca resultante é consideravelmente mais rápida quando comparada ao algoritmo SW [73]. Isso porque as comparações não serão realizadas entre cada resíduo individualmente, mas entre os padrões encontrados.

Uma diferença importante a observar é o fato do FASTA não se preocupar em encontrar um alinhamento ótimo, como é feito o Smith-Waterman, porém seus resultados muito rápidos o tornam uma opção bem mais viável para pesquisas rotineiras deste tipo em base de dados biológicos [59]. Atualmente, é possível utilizar este método através da Internet [65, 23]

2.3.2 BLAST

Em 1990, um outro método foi proposto por Altschul et al. [2] para submeter uma sequência de caracteres à uma busca por similaridades estatisticamente significantes [6] num banco de dados biológicos. Com o nome de *Basic Local Alignment Search Tool* - BLAST, tal ferramenta apresentou vantagens, como maior rapidez nas respostas que o FASTA, que logo a tornaram uma das mais utilizadas até o hoje.

A sensibilidade do BLAST é maior quando a busca por similaridades envolver sequências de proteínas. Ele apresenta a vantagem de tradução automática de uma sequência de nucleotídeos para os quadros proteicos e

otimização de pesquisas em base de dados contendo proteínas, sendo que o FASTA precisaria de seis passos separados para executar a mesma tarefa [61, 6].

Similar ao FASTA, o algoritmo empregado no BLAST acelera o processo de alinhamento local de sequências, aplicando heurísticas divididas em 3 (três) camadas, conforme descritas em [6]:

- *Seeding*: a ideia central desta heurística é que alinhamentos significativos têm palavras em comum. Uma palavra é composta por um número determinado de letras, por exemplo 3 para proteínas e 11 para ácidos nucleicos. O BLAST, então, localiza todas as palavras comuns (*word hits*) e, em seguida, usa-as como sementes (*seeds*) para o alinhamento, eliminando grande parte do espaço de busca;

- *Extension*: nesta heurística, uma vez que o espaço de busca tenho sido "semeado", ou seja, as palavras comuns tenham sido encontradas, os alinhamentos são gerados estendendo-se cada palavra nas duas direções a partir da semente;

- *Evaluation*: na última camada heurística, os alinhamentos gerados a partir das sementes estendidas são avaliados procurando-se os alinhamentos que são estatisticamente significativos. Para isso, determina-se um limite S e elimina-se aqueles alinhamentos com escores abaixo do limite S. Os alinhamentos com os mais alto escores (*High-scoring Segments Pairs*) dentro do limite S serão avaliados quanto ao seu valor estatístico e incluídos no alinhamento final quando possível.

É possível testar facilmente o processo acima numa ferramenta BLAST submetendo uma sequência de entrada ao servidor web dedicado e disponível no NCBI, *National Center for Biotechnology Information*, cujo acesso é possível pela Internet através do endereço em [61]. Este sistema é o mais

Programa	Banco de dados	Seqüência de Entrada
BLASTN	Nucleotídeo	Nucleotídeo
BLASTP	Proteína	Proteína
BLASTX	Proteína	Nucleotídeo convertido para proteína
TBLASTN	Nucleotídeo convertido para proteína	Proteína
TBALSTX	Nucleotídeo convertido para proteína	Nucleotídeo convertido para proteína

Figura 2.4: Conjunto de programas BLAST.

utilizado para pesquisas em bases de dados biológicos e disponibiliza todas as versões da ferramenta NCBI-BLAST, desde a original até as mais recentes [59, 6], veja a figura 2.4.

Capítulo 3

Alinhamento múltiplo de sequências biológicas

3.1 Definição do problema

Na comparação de um par de sequências biológicas, pode-se identificar a melhor correspondência entre os pares de resíduos encontrando-se um alinhamento ótimo, através de algoritmos exatos, NW e SW, que empregam programação dinâmica (seções 2.1 e 2.2). Utilizar-se dessa mesma ideia numa quantidade ainda maior de sequências é uma inferência lógica, contudo apresenta limitações computacionais que dificultam sua aplicação prática.

Seja o exemplo a seguir. Conforme visto nas seções 2.1 e 2.2, utilizando-se o método de programação dinâmica para encontrar o alinhamento de 2 sequências biológicas n_1 e n_2 de tamanhos t_1 e t_2, respectivamente, o número de comparações necessárias para preencher a matriz de escores é igual ao produto dos tamanhos das sequências, ou seja, $t_1 \times t_2$. Agora, segundo [59], estender esta análise para três sequências implicaria em substituir a matriz de escores por um gradeado em forma de cubo, onde seriam preenchidos os escores calculados via programação dinâmica. Então, para 3 sequências biológicas com tamanhos de 400 resíduos, o número de comparações seria igual a $t_1 \times t_2 \times t_3 = \ 400^3 = 1.6 \times 10^5$. Um alinhamento de 3 sequências com este número de comparações e usando um método de programação dinâmica é considerado realizável. Porém, caso a quantidade de sequências aumente, o número de passos e memória necessários tornam-se muito altos para aplicações práticas, conforme mostrado na tabela 3.1, baseada num exemplo de [59].

Assim, seja S um conjunto contendo n sequências biológicas de tamanho máximo t. Chama-se *Alinhamento Múltiplo de Sequências*, AMS, o problema computacional de alinhar simultaneamente as n sequências de S, quando $n > 2$.

O AMS é um problema NP-Difícil [80]. Sua demanda de espaço em memória e o tempo de execução crescem exponencialmente de acordo com a quantidade e tamanho das sequências de entrada.

# de sequências (n)	# de comparações 400^n
2	$400^2 = 1.6 \times 10^5$
3	$400^3 = 6.4 \times 10^7$
4	$400^4 \approx 2.6 \times 10^{10}$
5	$400^5 \approx 1.0 \times 10^{13}$

Tabela 3.1: Quadro sobre o aumento exponencial da quantidade de comparações exigidas num alinhamento global exato com programação dinâmica para $n > 2$ sequências biológicas de tamanho $t = 400$ resíduos, baseado em [59]. Sendo a complexidade $O(t^n)$, a cada nova sequência considerada para um novo alinhamento (coluna da esquerda), aumenta-se a quantidade de comparações em $t = 400$ vezes (coluna da direita).

Ainda assim, conforme visto, é possível obter um AMS ótimo para pequenos valores de $n > 2$ e tamanhos t reduzidos [71]. O algoritmo de Carrillo-Lipman [15] apresentou uma forma de obter um alinhamento ótimo para $n = 3$ ainda através de programação dinâmica, porém reduzindo-se o espaço de busca dos alinhamentos. Não obstante, para valores de $n > 3$, seu custo computacional o tornou uma opção impraticável para aplicação direta.

O programa MSA [45] propôs uma extensão do método exato mostrado na seção 2.1, implementando uma versão do algoritmo Carrillo-Lipman. A partir disso, foi possível alcançar um alinhamento ótimo para $3 \leq n \leq 8$ sequências biológicas de tamanhos em torno de $200 \leq t \leq 300$ resíduos.

Porém, matematicamente, o programa MSA não garante encontrar um resultado ótimo, pois seu procedimento básico emprega na verdade uma *heurística* [24] [45] para calcular a melhor aproximação do resultado exato com um custo computacional realizável para $n > 2$ [63].

Dentre as heurísticas que lidam com o problema do AMS, são relevantes para este trabalho as abordagens *progressiva* e *iterativa*. Estas serão apresentadas a seguir.

3.2 Heurística Progressiva

Em geral, as ferramentas que implementam uma heurística progressiva de alinhamento múltiplo se baseiam no método apresentando em [24]. Em essência, este método obtém o AMS através de processo gradual, onde um alinhamento inicial das sequências mais relacionadas é calculado e, posteriormente, uma-a-uma, as menos relacionadas são adicionadas ao alinhamento.

Especificamente, o algoritmo básico desta abordagem possui as seguintes etapas:

a. Primeiro, são calculados todos os alinhamentos dos pares formados a partir das $n > 2$ sequências biológicas de entrada, utilizando-se programação dinâmica.

Depois, constrói-se uma matriz de distâncias, D, onde cada elemento $D_{i,j}$ (fórmula 3.1, originalmente apresentada em [24]) representa a distância na qual foi convertido o escore $S_{i,j}$ correspondente alinhamento do par S_i e S_j de sequências.

$$D_{i,j} = -ln \frac{S_{i,j} - S_{rand}}{S_{max} - S_{rand}} \times 100 \tag{3.1}$$

$S_{i,j}$: corresponde ao escore do alinhamento global ótimo do par S_i e S_j.
S_{rand}: é o escore esperado do alinhamento entre um par aleatório de sequências em S.
S_{max}: é a média dos escores obtidos ao se alinhar as sequências do par $S_{i,j}$ com elas mesmas, ou seja, $S_{max} = \frac{(S_{i,i}+S_{j,j})}{2}$.

b. Na etapa seguinte, constrói-se uma árvore binária, chamada *árvore-guia*, para agrupar as sequências de acordo com as suas semelhanças [27]. Suas folhas correspondem às sequências de entrada, os nós internos à alinhamentos parciais (ou agrupamentos) e o nó raiz ao alinhamento final.

c. Finalmente, o alinhamento final começa ser construído a partir da primeira folha adicionada à árvore-guia, seguindo-se para outras folhas ou nós internos, sejam eles sequências ou alinhamentos parciais.
Este passo deve ser executado quantas vezes forem necessárias até que não existam mais sequências de S ou agrupamentos fora do AMS final. Deve-se ressaltar que os nós devem ser alinhados na mesma ordem em que foram inseridos na árvore-guia, ou seja, a partir dos nós mais semelhantes em direção àqueles menos semelhantes.

Para n sequências biológicas de tamanho t, o procedimento acima implicará numa complexidade $O(n \times t^2)$.

Um problema conhecido desta abordagem é a propagação de erros. Caso um erro seja cometido nos alinhamentos iniciais das sequências mais semelhantes, então este será propagado para os próximos alinhamentos e influenciará no resultado final.

Dois exemplos de ferramentas que seguem esta abordagem são o ClustalW [41] e o T-Coffee [64].

3.2.1 ClustalW

O pacote de programas $Clustal$ [40], implementa a heurística progressiva em seus 3 estágios [24]. Do mesmo modo, também possui suas limitações, como a dependência da qualidade dos alinhamentos iniciais e a dificuldade de escolha dos melhores parâmetros de alinhamento [59] [41].

Em sua versão mais recente, ClustalW [41], foram apresentadas algumas modificações visando aprimorar a qualidade dos alinhamentos obtidos.

No primeiro estágio de alinhamento dos pares de sequências, o ClustalW disponibilizou a nova opção de empregar um algoritmo global exato, mais lento, porém que garante um resultado ótimo a cada par de sequências alinhado. Na versão original, uma algoritmo mais rápido e de menor acurácia era o único método de alinhamento desta fase [40] [41].

No segundo estágio de agrupamento das sequências, outra modificação foi feita. Agora, o ClustalW construirá a árvore-guia identificando os pares mais próximos a partir das diferentes distâncias contidas na matriz do estágio anterior. A versão original do programa utiliza um método mais simples que considerava constante a taxa de mudanças ao longo da árvore [40].

No terceiro estágio, seguindo a ordem da árvore binária gerada, o alinhamento começa a ser construído a partir das folhas em direção à raiz. De acordo com árvore-guia, são atribuídos pesos às sequências. Os alinhamentos parciais de sequências estreitamente relacionadas recebem pesos menores, pois devem possuir duplicação de resíduos. Àqueles, onde as sequências são muito divergentes, atribuem-se pesos maiores.

Finalmente, são utilizadas duas funções de penalidades para lacunas (*gaps*), uma para abertura de lacuna e outra para extensão. Dois tipos principais de matrizes de pesos são utilizadas, a PAM e BLOSUM, sendo esta a opção padrão [41].

3.2.2 T-Coffee

Outra forma de aplicar a heurística progressiva para alinhar três ou mais sequências biológicas foi descrita em [64] num procedimento bifásico, chamado T-Coffee (TC).

Na primeira fase, o algoritmo TC constrói uma biblioteca de restrições composta por uma lista de restrições obtidas nas comparações par a par das sequências de entrada. Os métodos empregados nestes alinhamentos podem ser tanto globais quanto locais [64].

Cada restrição criada na biblioteca de restrições segue a forma $\{x_i, y_j, w\}$, onde os dois primeiros elementos representam respectivamente, o resíduo x da i-ésima sequência de entrada e o resíduo y da j-ésima sequência. O valor de w responde pelo peso atribuído àquela restrição.

Para evitar entradas duplicadas na biblioteca de restrições, os pares de resíduos que aparecerem em mais de uma restrição, serão combinados (*merge*) numa única restrição e seus pesos serão somados [84].

O TC pode estender sua biblioteca de restrições para conter dados indiretos sobre as próprias restrições. Por exemplo, sejam as restrições $\{x_i, y_j, w_1\}$ e $\{y_j, z_k, w_2\}$. É possível inserir na biblioteca de restrições uma restrição indireta $\{x_i, z_k, w_3\}$, que não tenha qualquer relação com o alinhamento entre a i-ésima e k-ésima sequências biológicas de entrada, porém w_3 seja uma função dos pesos de restrições anteriores, neste caso, w_1 e w_2.

Quando concluída, a biblioteca de restrições torna-se uma tabela tridimensional de pesquisa (*lookup table*), onde existe uma lista de restrições

associadas a cada sequência e a cada resíduo. As linhas da tabela são indexadas pelas sequências e as colunas pelos resíduos. Cada elemento $[x, y]$ da tabela guarda todas as restrições do tipo $\{x_i, y_j, w\}$. Com isso, obtém-se um mesmo custo de acesso às restrições associadas a qualquer resíduo em qualquer sequência [84].

Na segunda fase, o alinhamento progressivo é calculado. Seguindo as características desta abordagem, uma árvore-guia é construída primeiro a fim determinar a ordem dos alinhamentos parciais. Em princípio, o TC emprega um algoritmo de programação de dinâmica [33] para obter alinhamentos parciais exatos, porém outros métodos podem ser usados [84].

Para calcular a correspondente matriz de programação dinâmica em cada alinhamento parcial, a biblioteca de restrições é consultada. Sejam dois alinhamentos parciais Al_1 e Al_2 em nós internos da árvore-guia e que serão combinados. Cada par $[i, j]$ da matriz de escores será obtido a partir da combinação de todas as restrições contidas na biblioteca de restrições relacionadas com a coluna i em Al_1 e a coluna j em Al_2.

Através deste procedimento de consulta à tabela de restrições a cada gradação na montagem do alinhamento final, o TC propõe obter resultados mais precisos e evitar a possível propagação de erros iniciais que é inerente da abordagem progressiva. Ao permitir que diferentes métodos de alinhamentos (globais ou locais) sejam empregados internamente, o TC oferece flexibilidade no cálculo do AMS.

Contudo, para isso, o TC exige consideráveis demandas de espaço em memória e tempo de execução, as quais restringem sua aplicação para $n \leq 100$ sequências biológicas. Seja t o tamanho médio nas $n > 2$ sequências de entrada. Em seu comportamento padrão, o TC gera a biblioteca de restrições a partir de alinhamentos par a par que exigem $\frac{n(n-1)}{2}$ comparações. O número de restrições armazenadas implica um tamanho em memória proporcional a $O(t \times n^2)$. A segunda fase progressiva apresenta computação intensiva, pois exige o cálculo de $n - 1$ alinhamentos múltiplos parciais seguidos de constante consulta à tabela de restrições.

A seção 4.4 mostrará uma implementação alternativa de alta performance viabiliza o emprego do TC para valores de $n > 100$ sequências.

3.3 Heurística Iterativa

Em [34] foi proposta uma abordagem diferente para calcular o AMS e evitar os erros iniciais que podem acontecer na heurística progressiva. Com o nome de heurística iterativa, esta abordagem baseia-se na ideia de que uma solução para um determinado problema pode ser encontrada a partir de uma solução sub-ótima já existente [63]. Assim, os métodos esta que implementarem esta heurística irão realinhar subconjuntos de sequências repetidas vezes e inclui-las num alinhamento global que não restem mais sequências sem alinhamento.

Nas seções seguintes, será apresentado o programa DIALIGN e algu-

mas de suas aplicações ao utilizar a heurística iterativa no cálculo de um alinhamento múltiplo de sequências biológicas.

3.3.1 DIALIGN

O DIALIGN, proposto em [56], apresenta uma abordagem distinta daquela empregada nos algoritmos NW e SW (seções 2.1 e 2.2), pois seu alinhamento apoia-se na procura por semelhanças entre trechos de mesmo tamanho dentro das sequências, ao invés de comparar seus resíduos individualmente.

Estes trechos são ditos ininterruptos porque não possuem *gaps*, ou seja, são regiões ininterruptas de resíduos, também chamadas de segmentos [57]. Cada par de segmentos comparado recebe o nome de *fragmento*, ou *diagonal*, pois teria uma aparência em forma de diagonal em matrizes de comparação de pares de sequências [56, 51, 52].

Com isso, pode-se dizer que o DIALIGN faz comparações entre *segmentos* das sequências, não aplica penalidades para os *gaps* existentes e, por fim, procura pelo melhor alinhamento dos *fragmentos* obtidos. Esse ponto de vista mostra-se adequado para o alinhamento tanto de pares ($N = 2$) de sequências biológicas, quanto para o alinhamento múltiplo ($N \geq 3$), desde que estas compartilhem semelhanças locais sem precisarem ter qualquer relacionamento global entre si [57].

Uma visão geral do processo básico de alinhamento, considerando um par de sequências biológicas de entrada, pode ser resumido nos seguintes passos:

- Identificar e reunir num conjunto todas as diagonais, permitindo que tenham tamanhos diferentes e *mismatches*, e que os *gaps* existentes não sejam penalizados;
- Determinar o significado de cada diagonal do conjunto, ou seja, o quão semelhantes são os segmentos que cada uma representa;
- Descartar todas as diagonais que não atendam a um determinado critério, chamado de *consistência* [56];
- Dentre as diagonais consistentes que restarem, escolher um subconjunto contendo apenas as diagonais mais significativas, ou seja, que representem melhor as semelhanças de seus segmentos. Este subconjunto será o alinhamento final.

O DIALIGN descarta os trechos de baixa similaridade dentro das sequências ao ignorar as diagonais pouco significativas, como visto acima. Fazendo isso, consegue encontrar e alinhar corretamente pequenas regiões semelhantes, mesmo que as sequências sejam muito longas e que, por todas as suas extensões, sejam aparentemente diferentes [56].

A seguir será apresentado em detalhes o funcionamento do algoritmo de alinhamento de pares de sequências empregado pelo DIALIGN.

Inicialmente, é necessário esclarecer o conceito de *consistência*, pois um conjunto de diagonais somente fará parte do alinhamento final se atendem a este critério [57].

Figura 3.1: Conjunto consistente de diagonais

Seja S um conjunto de sequências biológicas e N o número de elementos de S. Para $N = 2$ sequências de entrada, quaisquer diagonais D_1 e D_2 pertencendo ao alinhamento são ditas *consistentes* se $D_1 \ll D_2$ ou $D_2 \ll D_1$, onde "$D_1 \ll D_2$" significa que as posições das sequências alinhadas em D_1 precedem necessariamente aquelas alinhadas em D_2 [56].

A figura 3.1 mostra um conjunto consistente de diagonais num alinhamento de pares e múltiplo de sequências [51]. Em (*a*), vê-se como as diagonais devem estar ordenadas ($D_1 \ll D2$) para que o conjunto seja considerado consistente num alinhamento de pares de sequências. E, neste caso, a saída do programa (*b*) compreende o alinhamento existente (*maiúsculo*) e também mostra aqueles resíduos que não estiveram envolvidos em nenhuma das diagonais selecionadas (*minúsculo*). Num alinhamento múltiplo, (*c*) mostra o conjunto consistente de diagonais e, em (*d*), a saída do programa com alinhamento múltiplo encontrado.

Diante disso, é possível acompanhar como o DIALIGN obtém o alinhamento de um par de sequências biológicas de entrada, $N = 2$, através de seu comportamento básico descrito abaixo:

a. Inicialmente, as duas sequências são distribuídas numa matriz, sendo uma ao longo do eixo x e a outra ao longo do eixo y prosseguindo com a identificação de todas as possíveis diagonais.

b. Depois, mede-se a importância de cada diagonal, ou seja, avalia-se o quão semelhantes são os segmentos que cada uma representa. E isso é feito atribuindo-se um peso (escore) para cada diagonal.

c. Para determinar o peso é necessário definir, primeiro, a quantidade $P(l, m)$. Esta representa a probabilidade de uma *determinada* diagonal de comprimento l conter pelo menos m *matches* [56], a saber:

$$P(l, m) = \sum_{i=m}^{l} \binom{l}{i} p^i (1-p)^{l-i} \tag{3.2}$$

sendo que, dentro da matriz usada para comparar as sequências de entrada, a quantidade p indica a probabilidade de cada célula representar um *match*, onde pode-se assumir uma distribuição uniforme e considerar valores como $p = 0.25$ ao comparar sequências de DNA e $p = 0.05$ no caso de proteínas [56].

d. Em seguida, define-se $E(l, m)$ como o logaritmo negativo da probabilidade $P(l, m)$ [56]:

$$E(l, m) := -ln(P(l, m)) \tag{3.3}$$

e. E a partir de $E(l, m)$ tem-se a definição do peso de uma diagonal D qualquer:

$$w(D) := \begin{cases} E(l, m), & se\ E(l, m) > T \\ \\ 0, & caso\ contrario. \end{cases} \tag{3.4}$$

onde quanto maior o valor de $w(D)$ maior será a chance dos pontos que formam a diagonal D representarem *matches* entre seus segmentos e não apenas serem casos isolados e aleatórios. Isso permite que a importância de diagonais de tamanhos diferentes seja comparada, pois pesos são atribuídos tanto às diagonais pequenas com alto índice de *matches* quanto às maiores e com baixas incidências de *matches* [56]. O usuário ainda pode definir um limite T e evitar que possíveis diagonais pequenas e aleatórias sejam consideradas nos cálculos e provoquem "*ruídos*"no resultados [56, 57, 51].

f. Neste ponto, avalia-se cada conjunto possível de diagonais que atendam ao critério de *consistência*, por exemplo $D_1, ..., D_k$. Depois, calcula-se a soma dos pesos dessas diagonais, o qual corresponde ao escore deste conjunto, segundo a fórmula 3.5:

$$Sc = \sum_{i=1}^{k} w(D_i) \tag{3.5}$$

Com isso, pode-se definir que o *alinhamento máximo* é aquele conjunto *consistente* de diagonais cujo escore é *máximo*, isto é, foi o maior escore encontrado entre todos os conjuntos avaliados [56].

g. Para efetuar esta avaliação, determinar os escores desses conjuntos possíveis e encontrar o *alinhamento máximo* das duas sequências, utiliza-se o recurso da programação dinâmica para alinhar as diagonais. Especificamente, se S e T forem as duas sequências biológicas de entrada e seus tamanhos forem representados por L_S e L_T, então para cada par (i, j) com $1 \leq i \leq L_S$ e $1 \leq j \leq L_T$ determina-se todos os inteiros $k \geq 0$ com $k \leq min(i-1, j-1)$, a fim de que a diagonal $(s_{i-k}, t_{j-k}; ...; s_i, t_j)$ que se inicia em $(i-k, j-k)$ e termina na posição (i, j) tenha um peso $w(D)$ positivo. Em seguida, para todas as posições (i, j), define-se um $score(i, j)$ para o *alinhamento máximo* das diagonais $D_1, ..., D_k$ que representam os segmentos de prefixos em $(s_1 s_2 s_3 ... s_i)$ e $(t_1 t_2 t_3 ... t_j)$. E, para o caso de existirem muitos alinhamentos com escore máximo, convenciona-se que os dados da última diagonal alinhada, neste caso D_k, sejam armazenados e obtidos através da função $prec(i, j) := D_k$, que representa última diagonal alinhada, isto é, a diagonal *precedente*.

h. Agora, considerando as diagonais consistentes já calculadas, para cada $D = (s_{i-k}, t_{j-k}; ...; s_i, t_j)$ com peso positivo, $w(D) > 0$, define-se o peso máximo acumulado, $\sigma(D)$, como sendo a soma dos escores de todas as diagonais anteriores incluindo D, conforme a relação abaixo [56]:

$$\sigma(D) = score(i-k, j-k-1) + w(D) \tag{3.6}$$

A diagonal que imediatamente precede D é dada pela função $\pi(D)$ [56]:

$$\pi(D) = prec(i-k-1, j-k-1) \tag{3.7}$$

A partir dessas duas equações, é possível calcular recursivamente o escore para a diagonal D [56]:

$$score(i, j) = max\{score(i, j-1),\ score(i-1, j),\ \sigma(D_{i,j})\} \tag{3.8}$$

onde D, neste caso, é a maior diagonal terminando no ponto (i, j) e que atende à seguinte relação [56]:

$$\sigma(D_{i,j}) = max\{\sigma(D) :\ D\ termina\ no\ ponto\ (i, j)\} \tag{3.9}$$

Do mesmo modo, segue a equação de recorrência $prec$ dada por [56]:

$$prec(i.j) := \begin{cases} prec(i, j-1), & se\ score(i,j) = score(i,j-1), \\ prec(i-1, j), & se\ score(i, j-1) < \\ & \quad score(i,j) = score(i-1,j), \\ D_{i,j}, & se\ score(i-1,j),\ score(i-1,j) < \\ & \quad score(i,j) = \sigma(D_{i,j}). \end{cases} \tag{3.10}$$

i. Finalmente, utilizando-se de programação dinâmica para calcular os valores de $score(i,j)$ e $prec(i,j)$, chega-se à matriz de similaridades onde o *alinhamento máximo* é obtido percorrendo-se o caminho inverso (*backtracking* [56]), fazendo-se:
$D_1 := prec(L_S, L_T)$
$D_{i+1} := \pi(D_i)$, desde que $\pi(D_i)$ esteja definido [56].

Este procedimento garante o alinhamento de um par de sequências biológicas de entrada. O mesmo, porém, aplicado num alinhamento múltiplo aumentaria exponencialmente a complexidade computacional em relação ao número N de sequências de entrada [56].

Isso ocorre porque, para $N \geq 3$ sequências, um ponto qualquer dentro de uma diagonal seria avaliado considerando simultaneamente todos os N segmentos que ela representa - e não apenas um par deles. Por exemplo, $score$ e $prec$ (equações 3.8 e 3.10) teriam N parâmetros ao invés de somente i e j, tornando os cálculos cada vez mais demorados à medida que se aumentasse o número de sequências de entrada.

Por este motivo, foi adotada no DIALIGN uma estratégia gananciosa, *greedy* [56], a fim de tratar o problema do alinhamento múltiplo de sequências, a saber:

a. Para $N \geq 3$, todos os pares de sequências são comparados aplicando-se o procedimento básico para $N = 2$ descrito acima. Desta forma, serão obtidos $\frac{N(N-1)}{2}$ alinhamentos ótimos [56] cujas diagonais encontradas serão agrupadas num conjunto, por exemplo, D_1. Neste momento, D_1 ainda não está consistente, então será calculado para cada diagonal de D_1 o peso de sobreposição, w_s, que reflete tanto o peso da diagonal quanto seu grau de sobreposição com outras diagonais, a fim de reconhecer semelhanças que ocorram em mais de duas sequências [56, 51].

b. Então, numa abordagem gananciosa, as diagonais de D_1 serão ordenadas de forma decrescente de acordo com seus escores (pesos) de sobreposição w_s.

c. E, em seguida, escolhendo-se a partir da primeira diagonal de D_1 com maior escore w_s, uma a uma, as diagonais de D_1 serão inseridas num outro conjunto *consistente*, D_2, observando que somente serão aceitas aquelas que estiverem *consistentes* com as diagonais que já existirem em D_2. O alinhamento múltiplo, então, estará definido em D_2 [56, 51].

d. Nesta etapa, a heurística iterativa é aplicada pelo DIALIGN ao procurar realinhar aqueles trechos das sequências que ainda não foram alinhados em D_2, repetindo-se os três passos acima para cada novo grupo de diagonais restantes em D_1 e que sejam *consistentes* com aquelas já pertencentes a D_2, parando somente quando não houver diagonais consistentes e com pesos positivos, $w_s > 0$, para serem incluídas no alinhamento múltiplo.

Com esta abordagem, o DIALIGN obtém o alinhamento múltiplo de $N \geq 3$ sequências biológicas de entrada [56].

Os passos descritos nesta seção compõem a versão do programa chamada de DIALIGN 1.0 e apresentada em [56, 57]. Estes procedimentos são o cerne das versões seguintes do DIALIGN que serão abordadas nas seções 3.3.2 e 3.3.4.

O DIALIGN possui restrições quanto à complexidade de espaço e tempo que influenciam seu desempenho mesmo em versões mais recentes. O trabalho publicado em [58] comparou os alinhamentos gerados pelo DIALIGN 2.0 (seção 3.3.2) e outros métodos, mostrando-se que o DIALIGN foi superior quanto à qualidade dos alinhamentos, porém seu tempo de execução foi maior que os dos outros métodos [11].

Assunto de propostas como [51, 11, 69, 76, 74, 8], a otimização do desempenho do DIALIGN também é objeto deste trabalho. Nas seções seguintes serão apresentadas as outras versões do DIALIGN e suas propostas para aperfeiçoar esta ferramenta que serviu de fundamento para esta dissertação.

3.3.2 DIALIGN 2.0

Como visto na figura 3.1, o DIALIGN 1.0 é um método flexível capaz de produzir um alinhamento múltiplo *global* e identificar que as sequências biológicas de entrada estão relacionadas através de semelhanças distribuídas por todas as suas extensões. Ou ainda,combinar num único alinhamento múltiplo *local*, as diferentes regiões semelhantes que pertencerem a sequências distintas, ignorando os trechos que não estiverem relacionados [51].

No entanto, a escolha pelo DIALIGN 1.0 requer uma avaliação quanto ao custo-benefício, pois ainda que seja recomendado, originalmente, para o alinhamento de sequências biológicas de tamanho limitado [56, 11], esta versão apresenta elevadas demandas de espaço em memória e tempo de execução [56, 11].

Caso o DIALIGN 1.0 seja empregado no alinhamento de um par ($N = 2$) de sequências biológicas de comprimento máximo igual a L, o espaço necessário para armazenar a matriz decorrente da equação 3.8 é proporcional ao produto dos comprimentos ($L \times L$), isto representa uma complexidade de espaço $O(L^2)$ [53]. Do mesmo modo, seria preciso um tempo de execução proporcional a $O(L^3)$ para encontrar uma alinhamento ótimo, considerando

todas as $O(L^3)$ diagonais possíveis dentro da matriz de comparação [51]. Porém, restringindo-se o tamanho da diagonais para um determinado limite máximo T_{max}, por exemplo, essa complexidade é reduzida e o tempo de execução torna-se proporcional a $O(L^2)$ [51].

Agora, se esta versão for utilizada num alinhamento múltiplo ($N \geq 3$), a complexidade de tempo do DIALIGN 1.0 para concluir todo o procedimento será ainda maior, como na fórmula 3.11 apresentada em [51]. É importante observar a influência da quantidade n_a, que representa a quantidade média de diagonais obtidas na primeira fase da heurística iterativa, onde são calculados todos os alinhamentos ótimos dos pares de sequências formados a partir das $N \geq 3$ sequências de entrada [56, 51].

$$[N^2 \times L^2] + [N^4 \times n_a^2] + [N^2 \times n_a \times log(N^2 \times n_a)] + [N^2 \times n_a \times N^2 \times L] \quad (3.11)$$

Foi neste contexto que, em [55, 51], foi apresentada uma atualização à versão inicial do programa, o DIALIGN 2.0. Para reduzir o custo computacional descrito na equação 3.11, introduziu-se em [55] uma nova função de pesos $w_2(D)$ para cada diagonal D e dada pela fórmula 3.12. O objetivo é efetuar uma seleção mais eficiente das diagonais que serão consideradas no alinhamento múltiplo, diminuindo-se a quantidade média n_a de diagonais presente na equação de complexidade do tempo (fórmula 3.11) e, por conseguinte, reduzindo o tempo de execução total do programa [51].

$$w_2(D) = -log(P_2(l_D, s_D)) \quad (3.12)$$

No DIALIGN 2.0, a função $P_2(l_D, s_D)$ denota a probabilidade de se encontrar *qualquer* diagonal de comprimento l_D e cuja soma dos valores de suas similaridades individuais seja no mínimo s_D dentro da matriz de comparação de duas sequências biológicas de entrada.

$$P_2(l_D, s_D) \approx l_1 \times l_2 \times P(l, m) \quad (3.13)$$

A partir da equação 3.13 [55], observa-se que a probabilidade depende também dos comprimentos das duas sequências sendo comparadas, neste caso l_1 e l_2. Se uma constante K for definida como $K := logl_1 + logl_2$, tem-se a seguinte aproximação:

$$\begin{aligned} w_2(D) &= -logP_2(l_D, s_D) \approx -log[l_1 \times l_2 \times P(l, m)] \\ &= -logP(l, m) - logl_1 - logl_2 \\ &= -logP(l, m) - (logl_1 + logl_2) \\ &= w(D) - K, \end{aligned} \quad (3.14)$$

concluindo-se que a nova função w_2 pode ser obtida simplesmente subtraindo-se a constante K da antiga função de pesos w.

De acordo com as fórmulas 3.3 e 3.4, o DIALIGN 1.0 tende a compor os alinhamentos a partir de pequenas diagonais, cujos tamanhos geralmente não superam um valor mínimo fixado de 7 resíduos [55, 51].

Sequências:

```
ASE-Fly       RRNARERNRVKQVNNGFALLREKIPEEVSEAFEAQGAGRGASKKLSKVETLRMAVEYIRSL
TFE3-Human    KKDNHNLIERRRRFNINDRIKELGTLIPKSSDPEMRWNKGTILKASVDYIRKL
MYC-Chicken   KRRTHNVLERQRRNELKLSFFALRDQIPEVANNEKAPKVVILKKATEYVLSI
```

Diagonais selecionadas:

```
  1 RRNARERNRVKQVNNGFALLREKIPEE   (ASE-Fly)
  4 NHNLIERRRRFNINDRIKELGTLIPKS   (TFE3-Human)

 46 SKVETLRMAVEYIRSL   (ASE-Fly)
 38 NKGTILKASVDYIRKL   (TFE3-Human)

  3 NARERNRVKQVNNGFALLREKIPE   (ASE-Fly)
  6 NVLERQRRNELKLSFFALRDQIPE   (MYC-Chicken)

 42 SKKLSKVETLRMAVEYIRSL   (ASE-Fly)
 33 NEKAPKVVILKKATEYVLSI   (MYC-Chicken)

  1 KKDNHNLIERRRR   (TFE3-Human)
  1 KRRTHNVLERQRR   (MYC-Chicken)

 23 LGTLIPKSSDPE   (TFE3-Human)
 23 LRDQIPEVANNE   (MYC-Chicken)

 39 KGTILKASVDYIRKL   (TFE3-Human)
 38 KVVILKKATEYVLSI   (MYC-Chicken)
```

Alinhamento Resultante:

```
ASE-Fly      ---RRNARERNRVKQVNNGFALLREKIPEEvseafeaqgagrgaSKKL-SKVETLRMAVEYIRSL
TFE3-Human   KKDNHNLIERRRRFNINDRIKELGTLIPKSSD-----------PEmrwNKGTILKASVDYIRKL
MYC-Chicken KRRTHNVLERQRRNELKLSFFALRDQIPEVAN-----------NEKA-PKVVILKKATEYVLSI
```

Figura 3.2: Alinhamento de 3 sequências *helix-loop-helix* construídas pelo DIALIGN 2.0. O programa selecionou um conjunto *consistente* de 7 diagonais, ou pares de segmentos. Os números à esquerda representam a primeira posição do respectivo segmento dentro da sequência. Os resíduos não alinhados são mostrados em letras minúsculas [55].

Como exemplo, considere-se utilizar o DIALIGN 1.0 para alinhar as sequências biológicas ASE-Fly, TFE3-Human e MYC-Chicken da figura 3.2 ao invés do DIALIGN 2.0. A primeira diagonal poderia ter sido dividida em 3 outras diagonais, conforme na figura 3.3 [55]. Dessa forma, o alinhamento seria composto de vários fragmentos menores e não pelas 7 longas diagonais selecionadas pelo DIALIGN 2.0 e mostradas em 3.2 [51].

Se uma diagonal D, com alta taxa de similaridades igualmente distribuída entre os pares de resíduos individuais que a compõem, for dividida em n diagonais $D_1, ..., D_n$, o peso $w(D)$ deveria ser consideravelmente maior que a soma dos pesos $\sum_i w(D_i)$. Entretanto, isso não acontece com o DIALIGN 1.0, pois a soma dos pesos das diagonais menores (equações 3.2, 3.3,

```
Diagonal selecionada no DIALIGN 2.0:

1 RRNARERNRVKQVNNGAALLREKIPEE (ASE-Fly)
4 NHNLIERRRRFNINDRIKELGTLIPKS (TFE3-Human)


A mesma diagonal dividida em 3 no DIALIGN 1.0:

(diagonal 1)   (diagonal 2)   (diagonal 3)
  RRNARERN       RVKQVNN       GAALLREKIPEE  (ASE-Fly)
  NHNLIERR       RRFNIND       RIKELGTLIPKS  (TFE3-Human)
```

Figura 3.3: Uma diagonal consistente com mais de 7 resíduos selecionada pelo DIALIGN 2.0 não será reconhecida no DIALIGN 1.0, que a dividiu em 3 outras menores [55].

3.4, 3.5) tende a ser maior que o peso da diagonal que os originou. Assim, os alinhamentos produzidos pelo DIALIGN 1.0 tendem a ser formados a partir de muitas diagonais pequenas em detrimento de poucas diagonais grandes, descartando potenciais diagonais significativas [55, 51].

Foi com essa redução de diagonais consistentes consideradas no alinhamento múltiplo, através das equações 3.12 e 3.13, que a versão 2.0 conseguiu melhorar o desempenho do DIALIGN em relação a sua versão original, especificamente pela redução dos valores de n_a na equação 3.11 de complexidade de tempo. Os resultados de uma comparação mais detalhada foram expostos em [55, 51].

3.3.3 CHAOS-DIALIGN

Devido à sua versatilidade e à qualidade de seus alinhamentos, o DIALIGN foi usado como ferramenta de alinhamento em outros trabalhos envolvendo longas sequências de DNA, como em [36, 35, 26]. Entretanto, seu desempenho cai quando as sequências são muito extensas, mostrando-se muito lento nestes casos [11].

Conforme apresentado na seção 3.3.2, o algoritmo básico de alinhamento de pares de sequências do DIALIGN possui um tempo de execução que depende dos tamanhos das sequências. Se S e T forem duas sequências biológicas de entrada, a complexidade de tempo para se obter um alinhamento ótimo será proporcional ao produto de seus comprimentos ($|S| \times |T|$).

Em [11], foi apresentada uma modificação ao programa DIALIGN que implementa uma opção de linha de comando permitindo que, durante sua execução, o alinhamento empregue a abordagem com *âncoras* a fim de diminuir o espaço de busca e, consequentemente, reduzir o tempo de execução. Também foi introduzida uma nova ferramenta chamada CHAOS, propondo a rápida identificação das âncoras que serão usadas como entradas pelo DIALIGN.

A partir de então, a combinação CHAOS-DIALIGN foi aplicada no ali-

nhamento de sequências biológicas de extensos comprimentos, como as longas sequências genômicas, onde o DIALIGN começou a ser aplicado no alinhamento de sequências biológicas de extensos comprimentos [11, 1].

O CHAOS (*CHAin Of Sequences*) é uma ferramenta para alinhamento local de pares de sequências biológicas. A saída do CHAOS é utilizada pelo DIALIGN numa abordagem por âncoras, ou *anchor points*, [11] a fim de reduzir o tempo de execução do DIALIGN.

A saída produzida pelo CHAOS é uma cadeia de alinhamentos locais com o *score máximo*, que pode ser utilizada como *âncora* por outras ferramentas que possuem procedimentos de alinhamento global, como é o caso do LAGAN [9] e o DIALIGN [51].

No DIALIGN, o CHAOS é empregado tanto no alinhamento de pares de sequências biológicas [10], quanto no alinhamento múltiplo [11]. Se forem $N = 2$ sequências de entrada, executa-se o CHAOS para identificar um conjunto de alinhamentos locais, chamado A, por exemplo.

Em seguida, um algoritmo baseado no *problema da subsequência crescente máxima* [37] é utilizado para encontrar em A a cadeia de alinhamentos locais que possua um *score máximo*) e que será considerada como ponto de âncora para o procedimento de alinhamento, conforme apresentado em [10]. Se k for número de alinhamentos contidos em A, então a complexidade de tempo deste algoritmo dada por $O(k\ log\ k)$.

Agora, para $N \geq 3$ sequências, primeiro executa-se o CHAOS para cada par de sequências formado a partir das N entradas e, ao final de todas as comparações, obtém-se um conjunto B contendo j alinhamentos locais. A partir deste ponto, estes alinhamentos serão considerados potenciais pontos de âncoras para o alinhamento múltiplo.

Porém, seguindo-se a mesma ideia apresentada na seção 3.3.1, é preciso garantir a *consistência* entre as potênciais âncoras contidas em B. Deste modo, no passo seguinte, atribui-se um escore a cada elemento de B e, em seguida, ordena-se o conjunto de modo descrente, usando-se o mesmo *algoritmo greedy* da seção 3.3.1 de modo que as potências âncoras com maior escore apareçam primeiro.

No último passo, inicia-se a validação das potenciais âncoras de B, ou seja, a partir da âncora de maior escore verifica-se sua consistência quanto às outras já aceitas e, caso esteja inconsistente, esta será removida de B. Como resultado final, restará em B apenas um conjunto consistente de âncoras ao passo que seus elementos formarão um único alinhamento múltiplo [11, 1].

Um desafio enfrentado ao usar esta ferramenta é que quanto mais âncoras forem encontradas, maior será a redução do espaço de busca e maior será o ganho em tempo de execução no DIALIGN. Por outro lado, muitas âncoras irão restringir excessivamente o espaço de busca e induzir uma diminuição na qualidade (*sensibilidade*) do alinhamento [11].

Desta forma, o principal desafio da abordagem de alinhamento com âncoras é encontrar um equilíbrio entre performance e sensibilidade, isto é, encontrar o máximo de pontos de âncoras possível e ainda chegar a um

alinhamento ótimo ou próximo deste [11].

A ferramenta CHAOS-DIALIGN foi disponibilizada num servidor de internet com uma interface web a fim de facilitar seu acesso à comunidade científica [12].

As características vistas até aqui, incluindo o recurso de âncoras, estão disponíveis via linha de comando no DIALIGN versão 2.2. O endereço para download está disponível em [54].

3.3.4 DIALIGN-TX

O DIALIGN 2.2 e versões anteriores possuem dois pontos fracos: sua função objetivo e seu procedimento ganancioso para construção do AMS [76]. Na comparação entre pares de sequências, a função objetivo usada para atribuir um escore a cada alinhamento local (fragmento) encontrado, apresenta uma tendência em favorecer alinhamentos locais isolados sobre outros possíveis alinhamentos globais.

No cálculo do AMS, o procedimento ganancioso adotado pelo DIALIGN 2.2 pode incluir fragmentos enganosos nos conjuntos parciais que formarão o alinhamento final. Isso acontece porque fragmentos pouco significativos receberam altos escores durante as comparações par a par e, por isso, foram incluídos no AMS em formação. Como a abordagem gananciosa desta fase não permite que estes fragmentos aceitos sejam removidos posteriormente, um único fragmento enganoso pode prejudicar a qualidade final do alinhamento [76, 74].

Neste contexto, uma nova implementação do DIALIGN foi apresentada e propôs novas heurísticas para lidar com esses vieses da versão 2.2. Chamada de DIALIGN-T [76], esta versão da ferramenta possui duas heurísticas principais: uma para amenizar o efeito da função objetivo e outra para calibrar o AMS durante sua construção.

Na primeira heurística, um algoritmo de encadeamento de fragmentos foi implementado para reconhecer e favorecer cadeias de fragmentos locais com baixos escores [76]. Dessa forma, o DIALIGN-T irá utilizar estas cadeias de fragmentos para contrabalançar os fragmentos isolados que a função objetivo tenha favorecido.

Na segunda, modificou-se o procedimento ganancioso utilizado no cálculo do AMS das versões anteriores. Agora, antes de ordenar os fragmentos obtidos e incluir na construção do AMS aqueles com os maiores escores, todos os fragmentos terão seus escores ajustados da seguinte forma: multiplica-se o escore de cada fragmento pelo quadrado do escore total do respectivo par de sequências dividido pelo escore total de todos os alinhamentos par a par. Assim:

$$w'(f) = w(f).\left(\frac{w(S_i, S_j)}{W}\right)^2 \tag{3.15}$$

onde, as sequências biológicas de entrada são representadas por $S_1, ..., S_n$. f corresponde ao fragmento obtido na comparação entre as sequências S_i e

S_j. $w(S_i, S_j)$ é o escore total do alinhamento par a par obtido entre S_i e S_j, ou seja, a soma de todos os escores de uma cadeia ótima de fragmentos [76]. O valor de W representa a soma total de escores de todos os alinhamentos par a par, calculado com a seguinte equação:

$$W = \sum_{1 \leq i \leq j \leq n} w(S_i, S_j) \tag{3.16}$$

Com esta implementação da abordagem gananciosa, o DIALIGN-T leva em consideração a semelhança existente entre o par de sequências sendo comparado e evita a inclusão de fragmentos pouco significativos no AMS e fornece um alinhamento múltiplo final com qualidade superior àquele obtido com o DIALIGN 2.2, conforme mostrado nos resultados de [76].

Contudo, em 2008, uma nova publicação apontou que o AMS calculado pelo DIALIGN-T através de seu procedimento ganancioso ainda era vulnerável ao efeito negativo de fragmentos aleatórios enganosos incluídos no AMS final [74].

Neste contexto, foi apresentada a última versão do DIALIGN. Chamada de DIALIGN-TX, esta versão introduziu uma nova heurística implementando uma combinação do método ganancioso do DIALIGN-T com uma adaptação da abordagem progressiva semelhante àquela vista na seção 3.2.

Em [74], foi apresentada uma descrição geral do algoritmo que calcula o AMS no DIALIGN-TX para um conjunto de sequências de entrada $s_1, \cdots, s_k$. A figura 3.4 mostra alguns trechos dessa descrição onde são identificadas três fases no cálculo do AMS.

A fase 1 do AMS no DIALIGN-TX compreende o cálculo de todas as comparações par a par para um determinado conjunto de entrada, por exemplo $s_1, \cdots, s_k$ (figura 3.4), como nas versões anteriores do DIALIGN. A saída desta fase é um conjunto F contendo todos os fragmentos possíveis, também chamados de segmentos ou diagonais (seção 3.3.1).

Na fase 2, pode-se observar as novas características introduzidas no DIALIGN-TX, como a construção de uma árvore-guia e seu emprego no cálculo do AMS com o método progressivo. O AMS progressivo é chamado de A_0. Um outro AMS também é calculado, A_1, utilizando o método ganancioso original do DIALIGN.

A última fase é responsável pelo processo iterativo característico do DIALIGN (seção 3.3.1) que irá acrescentar ao AMS (A_0 e A_1) os fragmentos adicionais que ainda pode ser encontrados a comparação par a par das sequências de entrada.

Os resultados apresentados em [74] apontaram que os alinhamentos múltiplos calculados pelo DIALIGN-TX 1.0 possuíam qualidade superior às versões anteriores. Porém, este avanço teve um custo reportado: aumento no consumo de recursos computacionais, como tempo de CPU e memória.

Nesta dissertação, todas as menções feitas ao DIALIGN-TX referem-se à verão 1.0.2, disponível para download em [75].

Algoritmo: DIALIGN-TX ($s_i, \ldots, s_k$) - Cálculo do AMS

```
/* Fase 1: Encontrar todos os fragmentos (cálculo das comparações par a par)* /
    F ← Ø
    para todo s_i, s_j tal que i < j
        faça | F ← F ∪ PAIRWISE_ALIGNMENT(s_i, s_j, Ø)
    fim do laço para todo
    ⋮
/* Fase 2 */
    Inicia cálculo de A1 utilizando o método ganancioso original do DIALIGN
    Inicia cálculo de A0 utilizando o novo método progressivo do DIALIGN-TX
    Construir árvore-guia
    Atualiza A0 utilizando a árvore-guia e o método ganancioso original do DIALIGN
    ⋮
/* Fase 3 */
    enquanto houver fragmentos
        Atualize A1 utilizando o método ganancioso original do DIALIGN
    enquanto houver fragmentos
        Atualize A0 utilizando o método ganancioso original do DIALIGN
    ⋮
    se peso(A_0) > peso(A_1) então
        RETORNE ← A_0
    senão
        RETORNE ← A_1
    fim da condição se
```

Figura 3.4: Descrição geral em pseudo-código do algoritmo para o cálculo do AMS no DIALIGN-TX para um conjunto de sequências de entrada $s_1, \cdots, s_k$. O alinhamento A_0 é calculado com a nova abordagem *progressiva* introduzida no DIALIGN-TX. O alinhamento A_1 também é calculado utilizando o método ganancioso empregado nas versões anteriores do DIALIGN. O algoritmo retorna o alinhamento que receber o maior peso (escore). A saída da sub-rotina *PAIRWISE_ALIGNMENT* é uma cadeia de fragmentos, que é equivalente ao alinhamento par a par do DIALIGN [74].

Capítulo 4

Estratégias paralelas para alinhamento de sequências

A pesquisa e a análise do grande volume de sequências existentes em bases de dados biológicos, contendo cada vez mais sequências da ordem de centenas de milhares ou até milhões de bases, têm apresentado relevantes demandas computacionais e chamado a atenção da comunidade científica, como em [83, 19, 49].

A necessidade de intenso processamento para realização do alinhamento múltiplo, principalmente à medida que se lida com um grande número de sequências, torna bastante interessante a resolução deste problema em arquitetura paralela, como tem sido mostrado nos trabalhos [83] [50] [69] [8], detalhados a seguir.

4.1 DIALIGN-P

O DIALIGN [56, 51] produz alinhamentos de alta *sensibilidade* e abrangência para grandes sequências biológicas [66], porém consome elevado tempo de execução.

O DIALIGN-P [69] é uma versão paralela do DIALIGN, que visa reduzir o tempo de execução deste algoritmo no alinhamento múltiplo de sequências biológicas de duas formas: na primeira, distribui os alinhamentos par a par do primeiro passo do AMS entre diversos processadores; na segunda, utiliza heurísticas para calcular o alinhamento de sequências genômicas de tamanho médio de 1 MB.

A solução adotada no DIALIGN-P possui duas estratégias diferentes para distribuir os cálculos entre vários processadores [69]. Na primeira, todos alinhamentos ótimos par a par são calculados em paralelo. O trabalho é dividido entre os processadores de acordo com a carga estimada utilizando um algoritmo *greedy*. A carga é estimada como o resultado do produto dos comprimentos dos pares de sequências. Em seguida, os alinhamentos produzidos são ordenados de forma decrescente com base no escore e o primeiro processador livre receberá o par de maior escore e, assim, sucessivamente.

Na segunda estratégia, uma abordagem paralela é combinada com um

procedimento de alinhamento ancorado, semelhante ao visto na seção 3.3.3. A ferramenta de alinhamento local CHAOS irá gerar a cadeia de alinhamentos locais de escores ótimos para cada par de sequências alinhando.

O DIALIGN-P irá selecionar dentro da cadeia de fragmentos fornecida pelo CHAOS, quais âncoras servirão de ponto de corte para dividir a longa sequência de entrada em pequenas sub-sequências. Estas serão alinhadas de forma independente em tarefas distintas distribuídas entre os processadores.

A biblioteca de troca de mensagens MPI [29] [30] foi utilizada na implementação do DIALIGN-P. Os testes foram efetuados num *cluster* contendo 32 máquinas equipadas com dois processadores Intel Pentium III com 650 MHz e 1 GB de memória RAM cada uma. A rede de interconexão dos nós foi construída num switch $Myrinet$ de 1.28 GBit/s. O sistema operacional empregado foi uma distribuição Linux com suporte para SMP (*Symmetric Multi-Processor*).

Nos testes apresentados em [69], foram alinhados conjuntos de sequências biológicas contendo 20, 55 e 100 proteínas com tamanho médio de 381 resíduos. Os ganhos alcançados em relação ao DIALIGN original, utilizando-se a abordagem do DIALIGN-P rodando em 64 processadores foram de 94,82%, 90,80% e 75.90%, respectivamente. No alinhamento de sequências genômicas, usando-se 3 sequências sintéticas com entrada e a ferramentas CHAOS (seção 3.3.3), o tempo de execução foi reduzido em 97,5%.

4.2 ClustalW-MPI

Em [44], foi apresentada uma implementação paralela do programa ClustalW (seção 3.2.1) com o objetivo de reduzir seu tempo total de execução.

Todos os estágios do algoritmo sequencial foram paralelizados com o auxílio da biblioteca MPI. A arquitetura sugerida foi um *cluster* de estações de trabalho com memória distribuída, contudo pode ser executado também em computadores paralelos [44].

No estágio inicial, utiliza-se uma estratégia de escalonamento que define um tamanho fixo às tarefas encarregadas pelo alinhamento dos pares de sequências e pelo cálculo de matriz de distâncias para, em seguida, distribuí-las nos processadores disponíveis via MPI.

No próximo estágio, o código-fonte do ClustalW foi alterado para otimizar o tempo de construção da árvore-guia, reduzindo a complexidade de tempo para $O(n^2)$. Outra alteração, foi a implementação em MPI de uma busca paralela a fim de encontrar as sequências com o mais alto grau de divergência.

Na construção progressiva do alinhamento final, diferentes pontos de otimização foram encontrados no último estágio. No algoritmo de programação dinâmica, responsável pela adição das sequências restantes ao alinhamento final, foi aplicado o paradigma de recursão paralela e, ainda, foram paralelizados os cálculos em cada um de seus passos de avanço e retrocesso. Noutro ponto, paralelizou-se o alinhamento dos nós mais externos

da árvore-guia estimando-se um ganho (*speedup*) máximo de $\frac{n}{log\ n}$.

O *cluster* empregado nos testes de ganho era composto de oito PCs interligados por uma rede FastEthernet padrão, sendo cada um equipado com dois processadores Pentium III de 800 MHz. As sequências biológicas utilizadas tinham tamanhos em torno de 1100 resíduos, numa quantidade total de 500 sequências. No primeiro estágio, utilizando-se todos os 16 processadores com 80 sequências para cada um, observou-se um *speedup* de 15.8. No último estágio, a abordagem mista para o alinhamento progressivo obteve uma *speedup* de 4.3 em relação a versão original.

4.3 MT-ClustalW

Outra estratégia de paralelização do algoritmo ClustalW foi apresentada em [18]: o programa MT-ClustalW. Trata-se de uma modificação do ClustalW-SMP [18] [17] que emprega uma técnica de múltiplas *threads* (*multithreading*) para aumentar o grau de paralelismo e permitir seu emprego em arquiteturas tanto SMP (*Symmetric Multi-Processor*) quanto com mais de um *core* (*multicore*).

Todos os estágios do algoritmo ClustalW foram divididos para execução em um conjunto de 2, 4 e 8 *threads*, utilizando-se a linguagem de programação `C`, a biblioteca de *threads* `pthread` com o objeto `mutex` sendo utilizado para sincronização.

Um gargalo, observado na execução do segundo estágio, mostrou que a construção da árvore-guia demandaria uma modificação no código-fonte para forçar sua distribuição entre as *threads* e garantir sua sincronização [18].

Uma função paralela foi implementada para conter os dois laços `for` mais internos dentro de um aninhamento de quatro níveis implementados no segundo estágio do ClustalW.

Seguindo a figura 4.1, a *thread* principal inicia a função paralela chamando um sinal de início, `signal(start)`, e espera por um sinal de resposta para executa o próximo laço. Uma variável chamada `thread_num` é usada para armazenar o número máximo de *threads* que podem ser criadas [18].

Enquanto aguarda pelo sinal de início, a função paralela executa outras tarefas. Com a chegada do sinal, ela irá decrementar de um o valor de `thread_num` e enviar um sinal (`signal(thread_num)`) para a *thread* principal. Em seguida, a função paralela continua a executar as tarefas que ainda existirem, esperando por um sinal de início. Se não existirem mais tarefas, sua execução será encerrada [18].

A partir do sinal de resposta da função paralela, a *thread* principal irá retornar para os laços seguintes somente depois que todas as threads paralelas completarem sua execução. Tão logo todos os laços estejam concluídos, a *thread* principal continuará a execução [18].

Um array de funções paralelas foi implementado no MT-ClustalW para

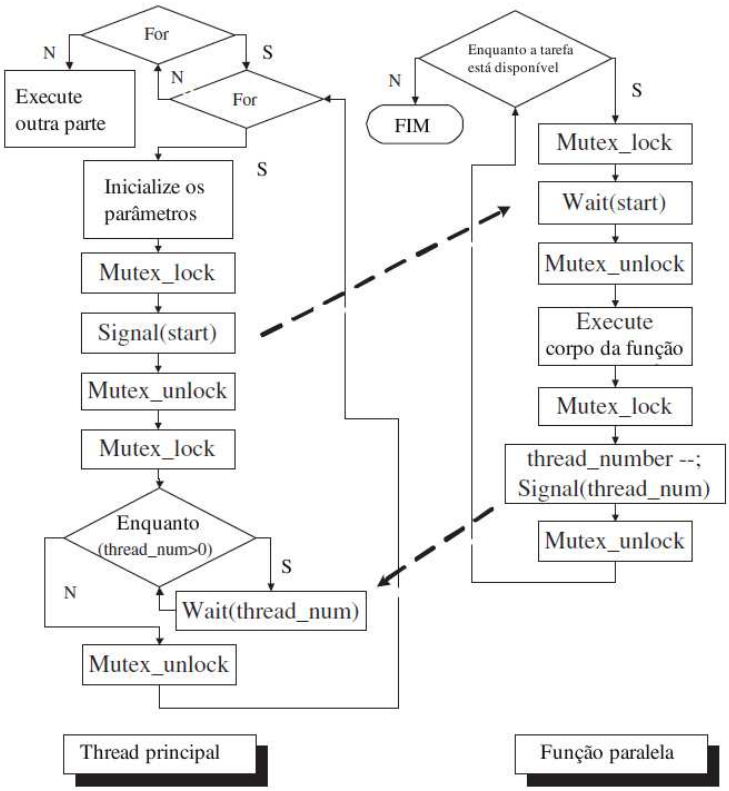

Figura 4.1: Fluxograma da sincronização de *threads* implementada no MT-ClustalW [18].

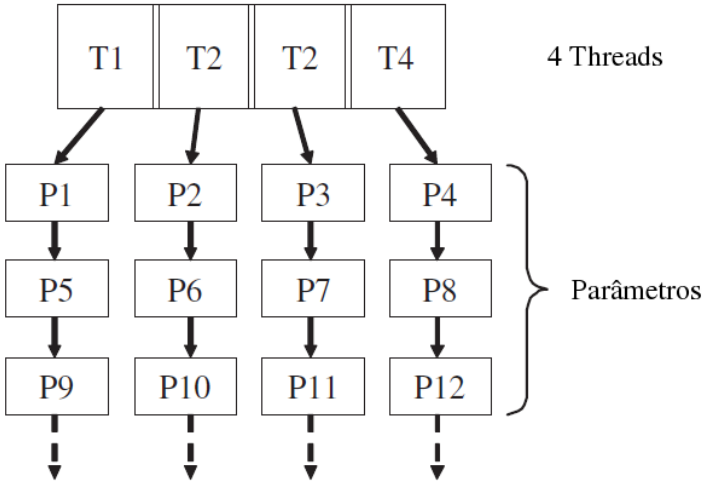

Figura 4.2: Distribuição de carga do MT-ClustalW. Todas as cargas são distribuídas para todas as funções *thread* como parâmetros [18].

diminuir a sobrecarga de criação da biblioteca `pthread`, como na figura 4.2. O tamanho máximo do array é determinado pelo usuário. Toda a carga é distribuída igualmente entre todas as *threads* e absorvidas pelas funções paralelas para as tarefas sejam executadas.

Durante os testes, o MT-ClustalW foi capaz de alinhar 16 conjuntos sequências biológicas diferentes, onde a quantidade variou entre 200, 400, 600 e 800 sequências e seus comprimentos também oscilaram em valores entre 200, 400, 600 e 800 resíduos. O equipamento utilizado foi uma máquina com processador *multicore* Intel Pentium-D Dual-Core (2 *cores*) de 2.8 GHz e RAM de 2 GB. O sistema operacional empregado foi o MS Windows XP Pro Service Pack 2 sem qualquer aplicação instalada. Nos resultados obtidos, conseguiu-se uma redução de cerca 55% do tempo de execução na comparação de 600 sequências de tamanho médio de 600 resíduos utilizando-se 4 *threads*, conforme apresentado em [18].

4.4 Parallel T-Coffee

Com o intuito de aumentar a capacidade de processamento do programa T-Coffee (TC, seção 3.2.2), uma versão paralela que utiliza MPI foi implementada. Os resultados foram apresentados em [84] com o nome de Parallel T-Coffee (PTC).

Em sua programação, o PTC baseia-se na versão 3.79 do TC, mantendo as funcionalidades originais e alterando somente as linhas de código que serão responsáveis pela execução paralela. Os estágios para geração da biblioteca de restrições e alinhamento progressivo do TC foram paralelizados.

O paradigma de troca de mensagens foi implementado numa arquitetura mestre-escravo, onde os mecanismos de distribuição de memória escolhidos empregaram primitivas de comunicação unilateral da biblioteca MPI-2.

Em geral, O PTC executa três etapas essenciais. A primeira começa por um processo chamado *inicialização*, onde o mestre recupera as sequências biológicas de entrada - local ou remotamente - efetua o *parse* e, finalmente, as distribui para os escravos. Neste ponto, a matriz de distâncias (seção 3.2.2) é calculada em paralelo por tarefas independentes executadas em cada escravo. No escalonamento destas tarefas, cada escravo processa uma parte da árvore de distâncias, mede o tempo exigido e o envia para o mestre, que irá redistribuir os cálculos subsequentes de acordo com a eficiência de cada escravo [84].

Na segunda etapa está a geração da biblioteca de restrições. Esta, conforme visto na seção 3.2.2, é a que mais consome tempo e espaço em memória durante a execução do TC. Para tentar superar isso, o PTC distribui as demandas desta etapa entre os escravos, através dos seguintes passos:

a. Gerar todas as restrições. O mestre irá distribuir para os escravos as tarefas de alinhamentos par a par de acordo com a eficiência conhecida ou por tarefas concluídas. Com isso o número de mensagens entre mestre e escravos é reduzido. Tão logo um escravo termine seus cálculos, deverá construir sua lista de restrições e armazená-la em sua memória local. Esta lista também chamada de biblioteca de restrições.

b. Agrupar e reavaliar os pesos das restrições. Para eliminar entradas duplicadas na lista de restrições que está distribuída entre os escravos, considera-se que a junção parcial das listas de restrições é associativa, pois trata-se tão somente da soma de seus pesos. Desse modo, localmente, as restrições são combinadas (*merge*) usando-se o processo original visto na seção 3.2.2. Em seguida, uma ordenação paralela é utilizada para agrupar as restrições que estejam repetidas em outros escravos. Cada restrição é atribuída a um repositório local que é determinado de acordo com o número total de processadores, incluindo o mestre [84]. Por sua vez, finalmente, cada processador irá eliminar as repetições contidas em seus repositórios locais combinando-as em uma única entrada, conforme o algoritmo original do TC.

c. Converter a biblioteca de restrições numa tabela de consulta (*lookup table*). Inicia-se pela divisão da tabela entre os processadores. Para n sequências de entrada e p processadores, cada um destes irá receber $\frac{n}{p}$ linhas da tabela (seção 3.2.2). Em todas as linhas recebidas, cada processador irá armazená-las num array e criar dois vetores de indexação para guardar os deslocamentos (*offsets*) de linhas e colunas respectivamente. Estes vetores serão trocados entre todos os escravos para permitir que cada um deles possa calcular o endereço exato, local ou remoto, de um determinada entrada na tabela. Para acessar os dados de uma parte remota da tabela, tão logo seja identificada, a linha inteira é recuperada, numa operação de busca antecipada (*prefetching*) [84], e armazenada localmente a fim de otimizar futuras consultas semelhantes. Cada parte recuperada antecipadamente é colocada numa *cache LRU* de tamanho especificado pelo usuário.

A terceira etapa implementa o alinhamento progressivo. O TC consome muito mais tempo neste ponto do que outras ferramentas desta abordagem, como o ClustalW [84]. Então, considerou-se que por se tratar de um alinhamento orientado por uma árvore binária, sua execução em paralelo poderia ser reduzida a um problema de escalonamento de grafo acíclico dirigido.

As relações de precedência impostas pela árvore binária fazem com que o caso ótimo, isto é, se a árvore estiver perfeitamente balanceada, seja limitado a uma estimativa de $\frac{n}{log\ n}$ para tarefas de tamanhos similares.

Assim, inicialmente, cada vértice do grafo é atribuído àquele processador que mais cedo irá começar a rodar a tarefa, ou seja, possui o menor tempo estimado para começar uma execução. As prioridades das tarefas e a lista de escravos são atualizadas toda vez que uma tarefa é concluída e outra precisa ser escalonada [84].

O PTC executa as tarefas de menor tamanho primeiro e também gerencia uma lista de processadores escravos que descreve a ordem de atribuição de tarefas. As tarefas de maior prioridade são atribuídas aos processadores que estiverem no começo da lista. Enquanto o número de tarefas (alinhamentos parciais) for maior que o número de escravos, estes serão escolhidos

de acordo com os dados sobre o desempenho de cada um obtidos na inicialização.

Quando o número de escravos for maior do que o número de tarefas, o mestre aplicará outro critério na alocação. Se uma tarefa a ser escalonada depende de outra que acabou de ser calculada pelo escravo solicitante, então ela é atribuída a este. Caso contrário, a tarefa será atribuída ao processador escravo que contenha a maior biblioteca de restrições em sua memória cache dentre todos os processadores solicitantes.

O PTC foi avaliado durante o alinhamento múltiplo de três conjuntos distintos de sequências biológicas, aqui chamados de S_1, S_2 e S_3. O primeiro, S_1, continha 349 sequências de comprimento máximo igual a 140 resíduos. Em S_2, 554 sequências com no máximo 331 resíduos. E, S_3, 1048 sequências de 523 resíduos no máximo.

A arquitetura empregada nos testes do PTC compreende máquinas contendo processadores duais Intel Xeon de 3GHz, cada uma equipada com 2GB de RAM e sistema operacional Linux. A cada passo para medir os tempos de execução, foram utilizados 16, 24, 32, 48, 64 e 80 processadores, onde cada um usou 768 MB de RAM. O *cluster* foi montado numa rede *FastEthernet*, utilizando para comunicação uma implementação da biblioteca MPI, chamada `mpich2-1.0.4p1`. O PTC foi compilado com pacote `GCC-3.4.4` [84].

A partir dos dados apresentados em [84], os ganhos de desempenho do PTC variaram de acordo com o número de processadores utilizados a cada medição. Por exemplo, ao alinhar S_1, o tempo de execução caiu de quase 12 minutos (702 segundos) para pouco mais de 4 minutos (243 segundos) utilizando-se 16 e 80 processadores, respectivamente. Num conjunto maior, S_3, o PTC obteve uma redução de aproximadamente 5 horas no cálculo do AMS ao aumentar de 16 para 80 o número de processadores do *cluster* sendo empregados.

4.5 Quadro Comparativo

Um resumo das soluções paralelas apresentadas neste capítulo é apresentado na tabela 4.1. Os alinhamentos realizados podem ser globais ou locais. O número de processadores e o paradigma de comunicação foram fornecidos nas respectivas publicações, sendo considerados para este caso os valores mais proeminentes, como o maior número de processadores usado e o modelo de programação predominantemente implementado.

Observe-se na tabela 4.1 que todas ferramentas deste capítulo realizam um alinhamento global das sequências de entrada, sendo que o programa PTC também encontra alinhamentos locais. A quantidade de processadores empregados variou bastante. Enquanto o MT-ClustalW concentrou seus testes numa única máquina *multicore* com 2 *cores*, utilizando o paradigma de comunicação via memória compartilhada com programação de múltiplas *threads*, as outras ferramentas foram executadas em *clusters* homogêneos

Programas	Tipo de Alinhamento	# de CPUs	Paradigma de Comunicação	Ambiente Paralelo
DIALIGN-P	Global	96	MPI	Homogêneo
ClustalW-MPI	Global	16	MPI	Homogêneo
MT-ClustalW	Global	2	*Multithreading*	Homogêneo
Parallel T-Coffee	Global/Local	80	MPI	Homogêneo

Tabela 4.1: Quadro comparativo das estratégias paralelas.

com 96, 16 e 80 processadores, todos se comunicando pelo mesmo paradigma de comunicação: troca de mensagens implementada com uma biblioteca MPI.

Capítulo 5

Projeto da Estratégia Distribuída Híbrida em *Cluster Multicore* Heterogêneo para AMS com o DIALIGN-TX

Em 2004, a descrição da ferramenta DIALIGN-P mostrou que é possível reduzir o tempo de cálculo do AMS no DIALIGN. Baseando-se no DIALIGN 2.2, o DIALIGN-P atua na primeira fase do AMS distribuindo os cálculos dos comparações par a par das sequências de entrada entre diversos processadores homogêneos existentes, conforme visto na seção 4.1.

Em 2005, uma reimplementação do DIALIGN 2.2 foi apresentada sob o nome de DIALIGN-T [76]. Descrevendo um conjunto de alterações nos algoritmos originais, esta versão produziu alinhamentos de qualidade superior para sequências biológicas relacionadas local ou globalmente.

Em 2008, novas heurísticas foram acrescentadas ao DIALIGN-T para aprimorar o AMS obtido quando as sequências estiverem globalmente relacionadas [74]. Com o nome de DIALIGN-TX, este novo método além de utilizar numa árvore binária (*árvore-guia*), também aplica duas novas abordagens para construir o AMS, uma *gananciosa* e outra *progressiva*.

Atualmente, a ferramenta DIALIGN-P está obsoleta pois não acompanhou as modificações introduzidas no DIALIGN-T e DIALIGN-TX. Ao nosso conhecimento, não existe versão atual distribuída do DIALIGN-TX em ambientes híbridos multicore.

5.1 Decisões de projeto

O *cluster multicore* homogêneo é uma arquitetura já estabelecida na construção de um ambientes computacionais de alto desempenho [21]. Por outro lado, um *cluster multicore* heterogêneo não costuma ser projetado, ele surge da necessidade de atualização ou expansão de um *cluster multicore* homogêneo, quando são adicionados computadores com processadores *multicore* diferentes dos que originalmente compunham o cluster. Pode-se inferir,

ainda, que as aplicações projetadas para a execução naquela arquitetura homogênea não conseguirão explorar de modo eficiente o poder computacional agora disponível num ambiente heterogêneo. Assim, escolheu-se o *cluster multicore* heterogêneo como ambiente alvo para uma uma estratégia para que seja capaz de adaptar a execução originalmente sequencial de uma aplicação para uma execução distribuída nesta arquitetura.

Para a execução distribuída em um *cluster multicore* heterogêneo foi escolhido um modelo de programação híbrido com troca de mensagens e memória compartilhada, pois estes se mostram adequados à utilização dos diferentes modos de comunicação existentes em um *cluster multicore* heterogêneo.

A aplicação escolhida para ter sua execução distribuída num *cluster multicore* heterogêneo foi o DIALIGN-TX, a versão mais recente do programa DIALIGN (seção 3.3.1). O DIALIGN-TX mostrou-se uma ferramenta bastante útil para o alinhamento múltiplo de sequências [70, 43], apresentando, em vários casos, resultados superiores às demais ferramentas [74]. Além disso, a fase 1 do AMS no DIALIGN, apresenta alto grau de paralelismo (seções 3.3.1 e 4.1), o que o torna bastante adequado para execução distribuída em *clusters multicore* heterogêneos.

Considerando que um *cluster multicore* heterogêneo emerge da adição gradativa de computadores com diferentes arquiteturas *multicore*, decidiu-se pela implementação de um programa capaz de recuperar dados do *cluster* que serão relevantes para a execução distribuída. Este programa foi chamado de *profiler*.

Finalmente, considerando a heterogeneidade do ambiente alvo e a execução distribuída da aplicação, optou-se pela adaptação de três estratégias de alocação de tarefas: *Self-Scheduling* [78], *Fixed* [72] e *Weight Factoring* [42].

5.2 Definição de *cluster multicore* heterogêneo

Nesta dissertação, um *cluster multicore* heterogêneo é um *cluster* de computadores, onde todos os seus nós possuem processadores *multicore* e, pelo menos, um nó possui um processador *multicore* com quantidade de *cores* diferente [16, 13].

Seja o *cluster multicore* da figura 5.1 composto por i nós. Existem dois nós com quantidades diferentes de *cores*. O nó n_2 possui uma quantidade k de *cores* e o nó n_i possui j *cores*. Esta condição é suficiente para que este *cluster multicore* seja considerado um *cluster multicore* heterogêneo.

Um *cluster multicore* heterogêneo como o da figura 5.1 possui uma característica híbrida quando à forma de comunicação física. Esta pode ser realizada por dois modos diferentes: a troca de mensagens entre os nós e a memória compartilhada entre os *cores* [16].

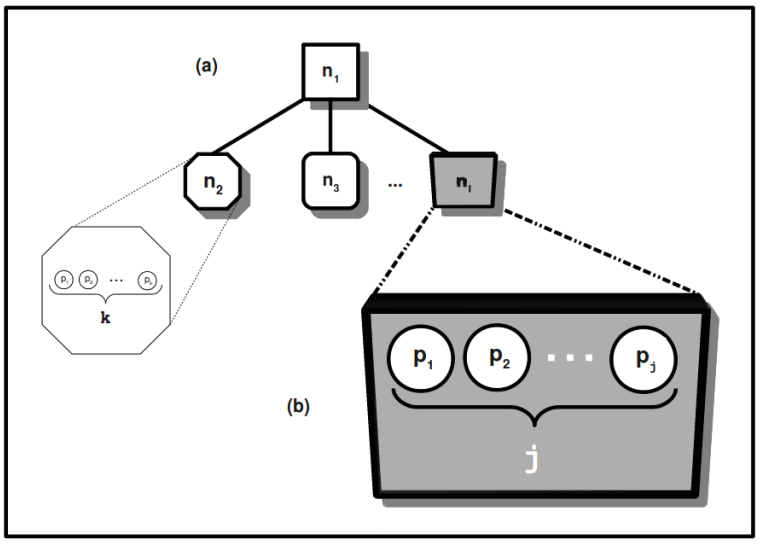

Figura 5.1: *Cluster multicore* heterogêneo.

5.3 Visão geral da solução

A estratégia distribuída para execução do DIALIGN-TX funciona da maneira descrita a seguir (figura 5.2). Primeiramente, o usuário solicita a execução do DIALIGN-TX via linha de comando. Em seguida, o nó mestre determina o poder computacional dos nós do *cluster*, que irá recuperar os dados sobre cada nó do *cluster*, inclusive ele mesmo. Então, o nó mestre armazena os dados do *cluster* no arquivo local de configuração.

Agora, tanto o nó mestre quanto os nós escravos farão a aquisição das sequências biológicas que serão usadas no cálculo do AMS. Essas sequências encontram-se em um diretório compartilhado via NFS.

Em seguida, o nó mestre inicia a geração de tarefas. A fase 1 do AMS no DIALIGN-TX realiza a comparação par a par de todas as sequências de entrada (seções 3.3.1 e 3.3.4). Com base nisso, o nó mestre organizará todas as comparações par a par em tarefas e *chunks*. Uma tarefa corresponde a uma comparação par a par. Um *chunk* corresponde a um conjunto de tarefas. Uma unidade de trabalho executada por cada *core*, então, é igual a uma tarefa. Ao final dessa etapa, será gerada uma lista de tarefas.

Depois, passa-se à etapa de distribuição de tarefas. O nó mestre consulta os dados de configuração e desempenho do nós do *cluster* e define quantas tarefas irão compor cada *chunk*. O mestre também irá calcular um valor de peso que será atribuído a cada nó de acordo com o seu desempenho. Em seguida, o nó mestre consulta a política de alocação de tarefas e determina como as tarefas serão agrupadas em *chunks* e distribuídas aos nós do *cluster*.

O nó mestre irá iniciar a distribuição das tarefas de acordo com a política de alocação. Todos os nós do *cluster* executarão suas tarefas e notificarão o nó mestre. Se existirem tarefas pendentes, o mestre fará novamente a distribuição das mesmas. Quando a lista de tarefas estiver vazia, passa-se

à etapa de coleta de resultados. O mestre solicita os resultados, que serão enviados pelos escravos.

Na coleta de resultados, o nó mestre irá organizar os resultados enviados pelos nós escravos e preencher suas estruturas em memória que serão utilizadas nas fases seguintes do AMS do DIALIGN-TX.

Finalmente, o nó mestre seguirá com a execução sequencial das fases 2 e 3 do AMS do DIALIGN-TX e enviará o alinhamento final para o usuário.

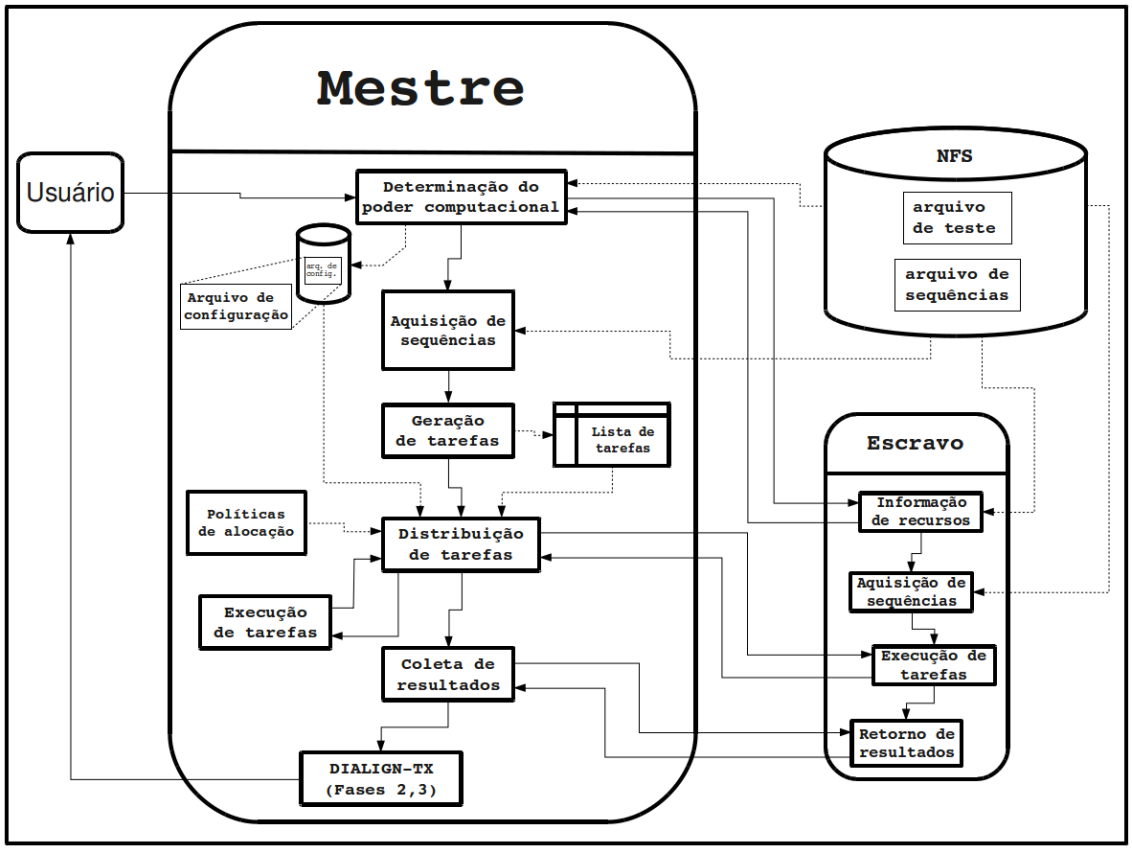

Figura 5.2: Visão geral da estratégia proposta.

5.3.1 Determinação do poder computacional

A primeira etapa da execução distribuída começa com a determinação do poder computacional do *cluster multicore* heterogêneo (figura 5.2). O usuário solicita a execução do programa via linha de comando e o nó mestre, então, verifica a existência do arquivo local contendo as configurações do *cluster*. Se o arquivo local existir, passa-se à próxima etapa. Se o arquivo não existir, o nó mestre executa o módulo `profiler` para obter e armazenar os dados do *cluster*.

O módulo `profiler` inicia sua execução pela obtenção do nome do seu nó (*hostname*) e do número de *cores* que este possui. A seguir, o `profiler` realiza o teste de desempenho, inicialmente, para o nó mestre.

O teste de desempenho do nó mestre começa pela leitura do arquivo de teste que está disponível em um diretório compartilhado e contém um conjunto de sequências biológicas. A seguir, o `profiler` executa o módulo responsável pela execução distribuída da fase 1 do DIALIGN-TX com memória compartilhada, fornecendo como entrada o conteúdo do arquivo de teste. O `profiler`, então, registra o tempo de execução deste módulo e encerra o teste de desempenho do nó mestre.

A seguir, passa-se ao teste de desempenho dos escravos. O `profiler` inicia o ambiente de troca de mensagens. Depois, o processo mestre envia uma mensagem para todos os escravos solicitando o nome do nó escravo, o número de *cores* que este possui e o tempo de execução do teste de desempenho.

A seguir, cada processo escravo obtém seu *hostname* e número de *cores* e, logo após, inicia sua execução local do teste de desempenho do mesmo modo como foi realizado pelo nó mestre, registrando ao final seu tempo de execução. Com o fim do teste de desempenho, cada processo escravo envia para o processo mestre os dados solicitados (nome, número de *cores* e o tempo de execução do teste desempenho).

Após o processo mestre receber todas as respostas do processos escravos, o `profiler` irá armazenar num arquivo local de configuração (figura 5.2) os dados que serão utilizados posteriormente pela solução distribuída, concluindo a etapa de determinação do poder computacional.

5.3.2 Aquisição de sequências

A próxima etapa caracteriza-se pela aquisição das sequências biológicas que serão utilizadas como entrada para o cálculo do AMS no DIALIGN-TX. Inicia-se pela leitura de um arquivo de sequências armazenado em um diretório compartilhado.

Utilizando o ambiente de troca de mensagens iniciado durante a determinação do poder computacional, todos os processos, mestre e escravos, fazem a leitura deste arquivo de sequências e preenchem suas estruturas internas que serão utilizadas em sua etapa seguinte.

5.3.3 Geração de tarefas

O processo mestre inicia a geração de tarefas pela identificação da menor unidade de trabalho que pode ser atribuída para execução em um *core* de qualquer nó do *cluster multicore* heterogêneo. Esta unidade de trabalho é chamada de tarefa e corresponde a uma comparação par a par da fase 1 do AMS no DIALIGN-TX.

Diante disso, o processo mestre pode calcular a quantidade de trabalho total da fase 1 do AMS no DIALIGN-TX, ou seja, o número total de tarefas. Tal como nas versões anteriores do DIALIGN [56, 57], a fase 1 do AMS no DIALIGN-TX executa as $\frac{n(n-1)}{2}$ comparações par a par possíveis para um conjunto de entrada contendo n sequências biológicas [76, 74]. Assim, a

quantidade total de tarefas que deverão ser distribuídas entre os *cores* do *cluster multicore* heterogêneo é igual $\frac{n(n-1)}{2}$.

Na solução proposta, uma tarefa é a menor unidade trabalho que um *core* irá executar por vez no *cluster multicore* heterogêneo. No entanto, é importante observar que vários *cores* executam, ao mesmo tempo, tarefas diferentes, como em [16, 82, 81, 68].

A figura 5.3 mostra um trecho de pseudo-código contendo a fase 1 do AMS no DIALIGN-TX [74]. Observe que as tarefas definidas nesta etapa da execução distribuída correspondem a cada iteração responsável pela execução do método $PAIRWISE\_ALIGNMENT(s_i,\ s_j)$, onde s_i e s_j são a i-ésima e j-ésima sequências biológicas de entrada, sendo $i < j$.

Algoritmo: DIALIGN-TX ($s_i, \ldots, s_k$) - Cálculo da Fase 1 do AMS

```
/* Fase 1: Encontrar todos os fragmentos (cálculo das comparações par a par)* /
    F ← Ø
    para todo s_i, s_j tal que i< j
        faça | F ← F ∪ PAIRWISE_ALIGNMENT(s_i, s_j, Ø)
    fim do laço para todo
    ⋮
```

Figura 5.3: Descrição geral em pseudo-código do algoritmo para calcular a fase 1 do AMS no DIALIGN-TX. Um conjunto de sequências biológicas $s_1, \cdots, s_k$ é fornecido como entrada. ***F*** é o conjunto de todos os fragmentos possíveis. A cada iteração do laço ***para todo...faça*** um par de sequências s_i, s_j é fornecido como entrada para o método *PAIRWISE_ALIGNMENT*. Sua saída é uma cadeia de fragmentos, que é equivalente ao alinhamento par a par do DIALIGN [74].

Desta forma, o processo mestre cria uma `lista de tarefas` contendo as $\frac{n(n-1)}{2}$ combinações $(i,\ j)$ possíveis para as iterações do laço ***para todo*** da fase 1 do AMS da figura 5.3. Cada entrada desta lista possui dois campos, sendo o primeiro para armazenar uma iteração $(i,\ j)$ do laço de comparação de pares de sequências e o segundo para assinalar se a tarefa correspondente foi atribuída ou não.

Na geração de tarefas, o processo mestre atualiza somente os primeiros campos da lista com todas as combinações de tarefas. O segundo campo somente será atualizado na etapa de distribuição de tarefas.

5.3.4 Distribuição de tarefas

O processo mestre passa à distribuição de tarefas e inicia a consulta ao arquivo local de configuração para obter os dados *cluster multicore* hetero-

gêneo, como o número de *cores* em cada nó e o seu tempo de execução no teste de desempenho (figura 5.2).

Utilizando a informação sobre o número de *cores* em cada nó, o processo mestre consulta a lista de tarefas e define a quantidade mínima de tarefas que será atribuída a cada nó como sendo igual ao número de *cores* existentes naquele nó. Essa quantidade mínima de tarefas é chamada de *chunk*.

O *chunk* é o menor conjunto de tarefas que pode ser atribuído a um nó específico do *cluster multicore* heterogêneo. Entretanto, devido à característica heterogênea do *cluster multicore*, dois nós podem receber um *chunk* cada um, porém contendo quantidades diferentes de tarefas.

Por exemplo, sejam os nós n_2 e n_i da figura 5.1. Durante a distribuição de tarefas, o nó mestre atribuirá, no mínimo, um *chunk* para cada para nó. Porém, o *chunk* do nó n_2 irá conter `k` tarefas (comparações par a par) diferentes, enquanto que o *chunk* do nó n_i conterá `j` tarefas.

A seguir, utilizando a informação sobre o tempo de execução de cada nó no teste de desempenho, o processo mestre calcula um peso de desempenho para cada nó e, depois, consulta a política de alocação de tarefas para definir a quantidade de tarefas que será atribuída a cada nó e como elas deverão ser distribuídas.

O processo de atribuição de pesos é realizado pelo nó mestre em dois passos. Primeiro, a partir dos dados de configuração, o processo mestre calcula a média aritmética dos tempos de execução obtidos pelos nós no teste de desempenho. Segundo, a cada nó, atribui um peso correspondente ao resultado da divisão entre a média aritmética resultante e o tempo de execução do nó. Os pesos obtidos com valores não inteiros são arredondados para uma casa decimal.

O peso de desempenho será utilizado na aplicação da estratégia de alocação de tarefas HWF, conforme descrito na seção 5.4.

Finalmente, o processo mestre distribui os *chunks* de tarefas para os processos escravos via troca de mensagens. A cada conjunto de tarefas distribuído, o processo mestre irá atualizar o segundo campo da lista de tarefas correspondente à atribuição daquela tarefa. O processo mestre continuará enviando mensagens, enquanto existirem tarefas pendentes na lista de tarefas.

Quando não existirem mais tarefas pendentes, o processo mestre encerra a distribuição de tarefas e passa à coleta dos resultados.

5.3.5 Execução de tarefas

A execução de tarefas é uma etapa que acontece em todos os nós do *cluster multicore* heterogêneo. Sua função é calcular as comparações par a par da fase 1 do AMS que são atribuídas pelo processo mestre na forma de tarefas e, assim que terminar, informar a conclusão para o processo mestre.

Porém existem algumas diferenças entre a execução que acontece em um nó escravo e aquela em um nó mestre. Em um nó escravo, após a aquisição das sequências biológicas de entrada, o processo escravo recebe uma

mensagem do processo mestre informando as sequências biológicas que deverão ser processadas (uma ou mais tarefas). Em seguida, o processo escravo inicia a execução distribuída com memória compartilhada da fase 1 do AMS do DIALIGN-TX, executando todas as comparações par a par (tarefas). Quando terminar a fase 1, o processo escravo envia uma mensagem para o processo mestre informando sua conclusão.

Algoritmo: DIALIGN-TX ($s_i, \ldots, s_k$) - Cálculo da Fase 1 do AMS
Execução distribuída com memória compartilhada

```
/* Fase 1: Encontrar todos os fragmentos (cálculo das comparações par a par)* /
    F ← Ø
    para todo s_i, s_j tal que i < j
        faça em paralelo | F ← F ∪ PAIRWISE_ALIGNMENT(s_i, s_j, Ø)
    fim do laço para todo
    ⋮
```

Figura 5.4: Descrição geral em pseudo-código do algoritmo para calcular a fase 1 do AMS no DIALIGN-TX em paralelo. Um conjunto de sequências biológicas $s_1, \cdots, s_k$ é fornecido como entrada. F é o conjunto de todos os fragmentos possíveis. Dentro do laço ***para todo...faça*** o método *PAIRWISE_ALIGNMENT* é executado em paralelo. A saída dessa fase é uma cadeia de fragmentos, que é equivalente ao alinhamento par a par do DIALIGN [74].

No nó mestre, a execução de tarefas é realizada em paralelo com a distribuição de tarefas, logo após a etapa de geração de tarefas. Como o processo mestre já tem à disposição, localmente, os dados do *cluster* e as sequências biológicas necessários, então ele inicia a execução distribuída com memória compartilhada da fase 1 do AMS do DIALIGN-TX em uma nova *thread*. Assim, quando terminar a execução das tarefas da fase 1, o processo mestre terá acesso local aos seus resultados.

5.3.6 Coleta e retorno de resultados

O processo mestre inicia a coleta de resultados pelo preenchimento de suas estruturas de dados com os seus resultados. Em seguida, envia uma mensagem para todos os processos escravos solicitando que enviem seus resultados.

Os processos escravos, então, iniciam a etapa de retorno dos resultados, enviando-os em mensagens para o processo mestre. Dependendo do tamanho do conjunto de resultados, os processos escravos usam mais de uma mensagem para enviar seus resultados para o processo mestre.

A cada mensagem que chegar, o processo mestre irá armazenar, temporariamente, os resultados até todos que os processos escravos terminem o

envio de seus resultados. Ao término, o processo mestre irá organizar os resultados recebidos e atualizar suas estruturas de dados em memórias.

5.3.7 Finalização

Finalmente, o nó mestre seguirá com a execução sequencial das fases 2 e 3 do AMS do DIALIGN-TX (figura 3.4) e enviará o alinhamento final para o usuário.

5.4 Políticas de alocação de tarefas

A seção 5.3.4 mostrou que a solução proposta possui uma etapa responsável pela distribuição de tarefas entre os nós do *cluster multicore* heterogêneo. Nesta etapa, o nó mestre consulta uma política de alocação de tarefas para decidir como distribuí-las de modo a balancear a carga do sistema.

Em geral, este problema de alocação de tarefas para um modelo de aplicação mestre/escravo em um *cluster* de computadores pode ser representado pela função $A(T, N)$, onde T tarefas precisam ser distribuídas entre os N nós do *cluster*.

Normalmente, considera-se que uma tarefa é a menor unidade de trabalho que pode ser atribuída a um determinado nó, inferindo-se que cada nó irá resolver apenas uma tarefa por vez [72]. Porém, ao considerar a adoção da função $A(T, N)$ para elaborar uma estratégia de alocação e distribuir as tarefas em um *cluster multicore* heterogêneo, observou-se a necessidade de adaptá-la ao aspecto *multicore* do ambiente e ao modelo híbrido de programação utilizado (seção 5.5), pois, agora, cada nó possui mais de um *core* e é capaz de processar mais de uma unidade de trabalho por vez.

Nesta dissertação, conforme visto na seção 5.3, a menor unidade de trabalho será uma comparação par a par entre duas sequências biológicas na fase 1 do AMS do DIALIGN-TX. Esta unidade de trabalho será chamada de tarefa. O nó mestre irá atribuir cada tarefa pendente a um *core* disponível, que irá executá-la dentro de um nó do *cluster multicore* heterogêneo.

A fim de explorar a capacidade total de processamento dessas tarefas em um *cluster multicore* heterogêneo, decidiu-se pelo emprego de uma estratégia de alocação de tarefas que permita o balanceamento da carga do sistema e tire proveito da característica *multicore* de cada nó do *cluster*.

Neste contexto, várias políticas de alocação de tarefas podem ser representadas usando-se a função genérica $A(T, N)$. Nesta dissertação, foram escolhidas as políticas: *Fixed* [72], *Self-Scheduling* [78] e *Weight Factoring* [42]. A adaptação destas políticas e de suas funções de alocação ao modelo híbrido de programação por troca de mensagens e memória compartilhada, resultou nas seguintes estratégias: *Hybrid Fixed* (HFixed), *Hybrid Self-Scheduling* (HSS) e *Hybrid Weight Factoring* (HWF).

5.4.1 Hybrid Fixed (HFixed)

A estratégia *Hybrid Fixed* (HFixed) baseia-se na politica *Fixed* [72], onde o número de tarefas é dividido pelo número de processadores. A estratégia HFixed utiliza a equação de alocação 5.1, onde T representa o número de tarefas e N o número de nós. A diferença na abordagem HFixed está no contexto de distribuição das tarefas, onde cada nó possui mais de um *core* e, com isso, mais de uma tarefa pode ser executada ao mesmo tempo. Na estratégia *Fixed*, ao contrário, infere-se que se existem N nós, então existem N unidades de processamento.

$$A(T, N) = \frac{T}{N} \tag{5.1}$$

Assim como na estratégia *Fixed*, a HFixed é indicada para sistemas com recursos dedicados e aplicações que, em tempo de execução, apresentem um comportamento uniforme. Por isso, a função de alocação da estratégia HFixed apresenta o valor constante representado na equação 5.1.

No entanto, a distribuição de uma carga constante entre os nós de um *cluster multicore* heterogêneo não considera aspectos do sistema que surgirão dinamicamente, como é o caso do nó mestre que, nesta dissertação, além de participar na execução das tarefas, terá a carga adicional dos recursos compartilhados que serão processados para manter a aplicação principal rodando.

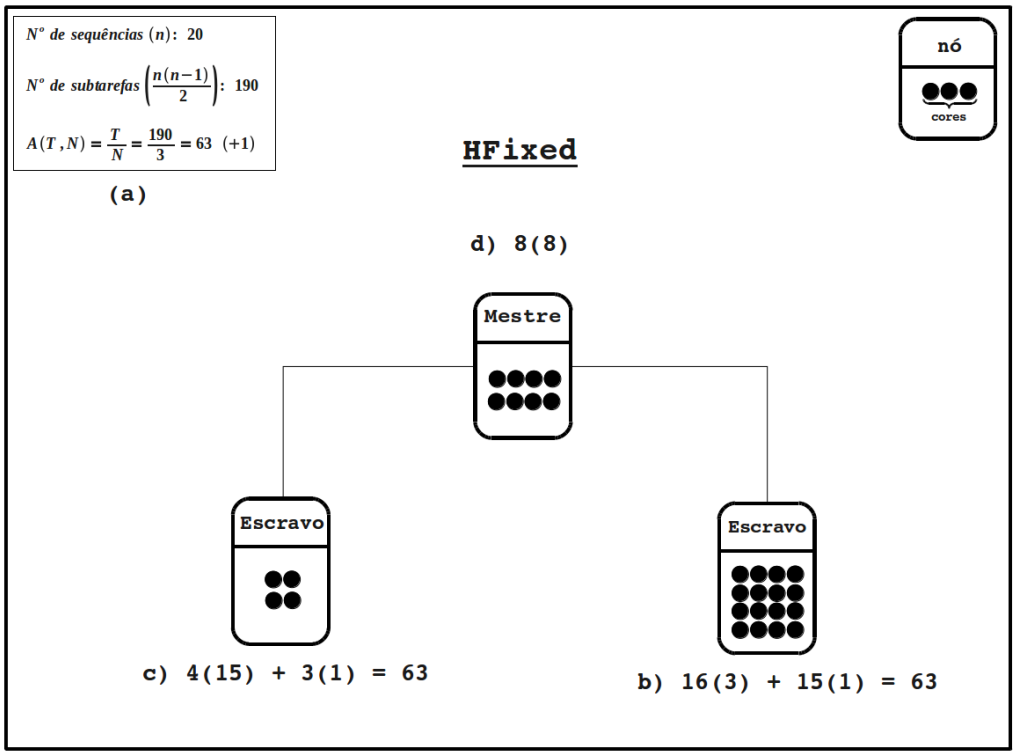

Figura 5.5: Exemplo de funcionamento da estratégia HFixed.

A figura 5.5 mostra um exemplo de execução da etapa de distribuição de

tarefas (seção 5.3.4), utilizando a estratégia HFixed para alocar as comparações par a par da fase 1 do AMS do DIALIGN-TX num *cluster multicore* heterogêneo.

Na figura 5.5 (a), pode-se observar que o conjunto de sequências biológicas de entrada contém 20 sequências ($n = 20$) e, consequentemente, a quantidade de tarefas pendentes é igual ao número de comparações par a par entre as 20 sequências de entrada, ou seja, $\frac{n(n-1)}{2} = 190$ tarefas (seção 5.3.3). A quantidade de tarefas que será atribuída a um determinado nó é chamada de *chunk*. Para obter o valor do *chunk* na abordagem HFixed com as 190 tarefas pendentes, utiliza-se a equação de alocação 5.1, $A(T, N) = \frac{190}{3} = 63$ tarefas. Como esta divisão não é exata, a tarefa restante é representada como (+1). Assim, o nó mestre ao aplicar esta estratégia irá distribuir um *chunk* de 63 tarefas para cada escravo. A tarefa restante será adicionada ao *chunk* que irá para o nó mais rápido (mestre). Um cenário possível da distribuição e execução deste *chunk* de tarefas esta disposto nos itens (b), (c) e (d) da figura 5.5.

Na figura 5.5 (b), 48 tarefas ocuparão os 16 *cores* com o processamento de 3 tarefas cada (`16(3)`). Porém restarão 15 tarefas que serão executadas pelos primeiros *cores* ociosos daquele escravo (`15(1)`). Observe-se que 1 *core* ficará ocioso, pois o número de tarefas pendentes é maior que o número de *cores*.

Do mesmo modo, na figura 5.5 (c), os 4 *cores* executarão 60 tarefas e os primeiros 3 *cores* livres irão processar as 3 tarefas restantes. Observe-se que, novamente, 1 *core* ficará ocioso.

Finalmente, na figura 5.5 (d), a tarefa restante (+1) obtida na equação de alocação do item (a) da figura 5.5 será adicionada ao *chunk* atribuído ao nó mestre este executará 64 tarefas (`8(8)`). Neste caso, todos os *cores* estarão ocupados processando 8 tarefas cada e nenhum ficará ocioso.

5.4.2 Hybrid Self-Scheduling (HSS)

A estratégia *Hybrid Self-Scheduling* (HSS) baseia-se na política *Self- Scheduling* (SS) apresentada em [78]. Se o nó mestre aplicar a abordagem HSS na etapa de distribuição de tarefas, estas serão alocadas à medida que os nós escravos solicitarem. Para descobrir o tamanho do *chunk* utilizado a cada distribuição de tarefas, o nó mestre utiliza a equação de alocação 5.2, onde $Cores_{N_i}$ representa a quantidade total de *cores* em um nó N_i do *cluster multicore* heterogêneo:

$$A(T, N) = Cores_{N_i} \tag{5.2}$$

Um exemplo de distribuição de tarefas utilizando a estratégia HSS é mostrado na figura 5.6. Semelhante à figura 5.5, em (a), observa-se que o número de sequências é igual 20 e a quantidade total de tarefas é igual a 190. Inicialmente, o *chunk* utilizado será no máximo igual à quantidade de tarefas solicitadas pelos nós escravos, ou seja, 28 tarefas para 28 *cores*.

Por isso, em (b), o nó $Escravo_2$ recebe 16 tarefas, uma para cada *core*. Em (c), o nó $Escravo_1$ recebe 4 tarefas, cada *core* irá processar uma delas. E, finalmente, em (d), o nó mestre recebe 8 tarefas para seus 8 *cores*.

Note-se que ao utilizar a abordagem HSS, o nó mestre distribuirá no máximo a quantidade exata de tarefas correspondente ao número de *cores* existente em um nó totalmente ocioso. Por exemplo, o nó $Escravo_2$ enviará uma mensagem para nó mestre informando que terminou seu processamento. O nó mestre entende com essa mensagem que todos os 16 *cores* do nó $Escravo_2$ estão ociosos e, por isso, enviará, no máximo, 16 tarefas para nó $Escravo_2$, uma para cada *core*. Isso ratifica que o *chunk* definido pelo nó mestre ao empregar a abordagem HSS é igual ao número de *cores* do nó, conforme a equação de alocação 5.2.

Assim, o funcionamento da estratégia HSS é uma adaptação da política SS à característica *multicore* do *cluster* heterogêneo. Enquanto a política SS distribui a cada processador exatamente exatamente uma unidade de trabalho, a estratégia HSS distribui a cada nó *multicore* o número de unidades de trabalho que ele é capaz de processar. Do ponto de vista do mestre que executa a estratégia HSS, não existe alocação de uma tarefa diretamente a um *core* de um determinado nó. O que existe é a alocação de tarefas para os nós escravos na mesma quantidade de *cores* que estes possuem.

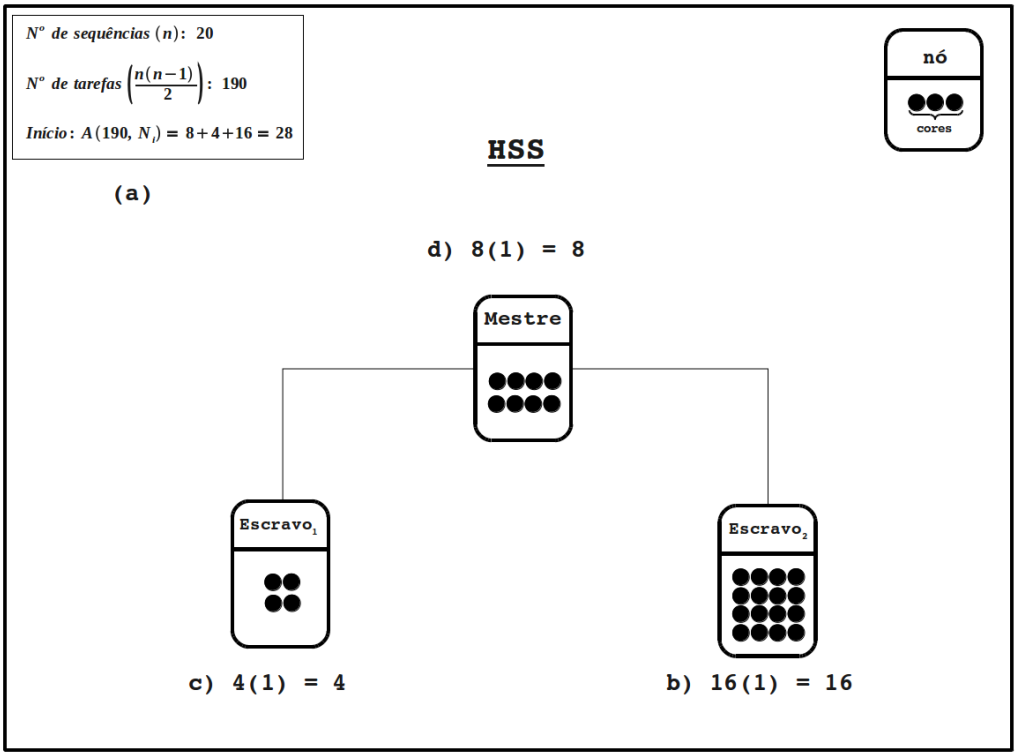

Figura 5.6: Exemplo de funcionamento da estratégia HSS.

5.4.3 *Hybrid Weight Factoring* (HWF)

A terceira política de alocação de tarefas adaptada para execução em um *cluster multicore* heterogêneo foi a *Weight Factoring* (WF), apresentada em [42]. A política WF foi concebida para considerar a diversidade de ambientes computacionais como um *cluster* heterogêneo. Baseando-se em outra estratégia chamada *Factoring* [28], a WF propõe que uma constante peso (W) seja associada a cada processador do *cluster* heterogêneo para refletir a capacidade de processamento de um determinado nó. Aqui, observe-se a associação entre nó e processador: um nó possui apenas um processador, porém nós diferentes podem ter processadores com velocidades diferentes. Deste modo, utilizando-se a WF, os *chunks* de tarefas são distribuídos aos processadores de acordo com o peso W de cada um.

Devido à sua abordagem direcionada para *clusters* heterogêneos, esta estratégia foi adaptada para considerar a característica *multicore* dos processadores atuais que estar presentes em *clusters* heterogêneos. Neste contexto, propõe-se a estratégia de alocação de tarefas *Hybrid Weight Factoring* (HWF), que ao invés de considerar a velocidade de apenas uma unidade de processamento dentro de um nó específico, avalia a velocidade de processamento total de um nó, ou seja, aplica um peso W para o nó *multicore* de acordo com a sua velocidade de processamento em relação aos outros nós do *cluster multicore* heterogêneo.

Nesta dissertação, o teste para avaliar a velocidade de processamento de cada nó é realizado pelo programa *profiler* durante a determinação do poder computacional vista na seção 5.3.1. Quando o nó mestre consulta o arquivo de configurações e a política de alocação de tarefas durante a etapa de distribuição de tarefas (seção 5.3.4), um valor de peso W_{n_i} é associado a cada nó n_i do *cluster multicore* heterogêneo.

A estratégia HWF utiliza a equação de alocação 5.3, onde T é o número de tarefas e N o número de nós do *cluster multicore* heterogêneo. A execução do HWF é realizada em estágios, onde a cada estágio s calcula-se o valor do *chunk* de tarefas que será atribuído a um nó n_i específico, utilizando-se a equação 5.3, adaptada de [42].

$$A(s, T, N, n_i) = max\left(\left\lceil \frac{W_{n_i} \times T}{N \times 2^{estagio(s)}} \right\rceil, 1\right) \tag{5.3}$$

A figura 5.7 mostra um exemplo de distribuição de tarefas em um *cluster multicore* heterogêneo, utilizando-se a estratégia HWF. Em (a), observa-se que o número de tarefas pendentes é igual a 190 comparações par a par para as 20 sequências biológicas de entrada. No estágio 1 da execução do HWF, inicia-se a fatoração do número total de tarefas, dividindo-o por 2. O resultado está representado por 95_1 tarefas pendentes tratadas no estágio 1 da estratégia HWF e 95_2 pendentes a serem tratadas nos estágios seguintes. Estas tarefas pendentes serão alocadas em cada estágio pelo nó mestre de acordo com o peso de cada nó do *cluster multicore* heterogêneo.

Deste modo, começando-se pelo nó mestre, o peso recebido foi $\frac{3}{2}$. O nó

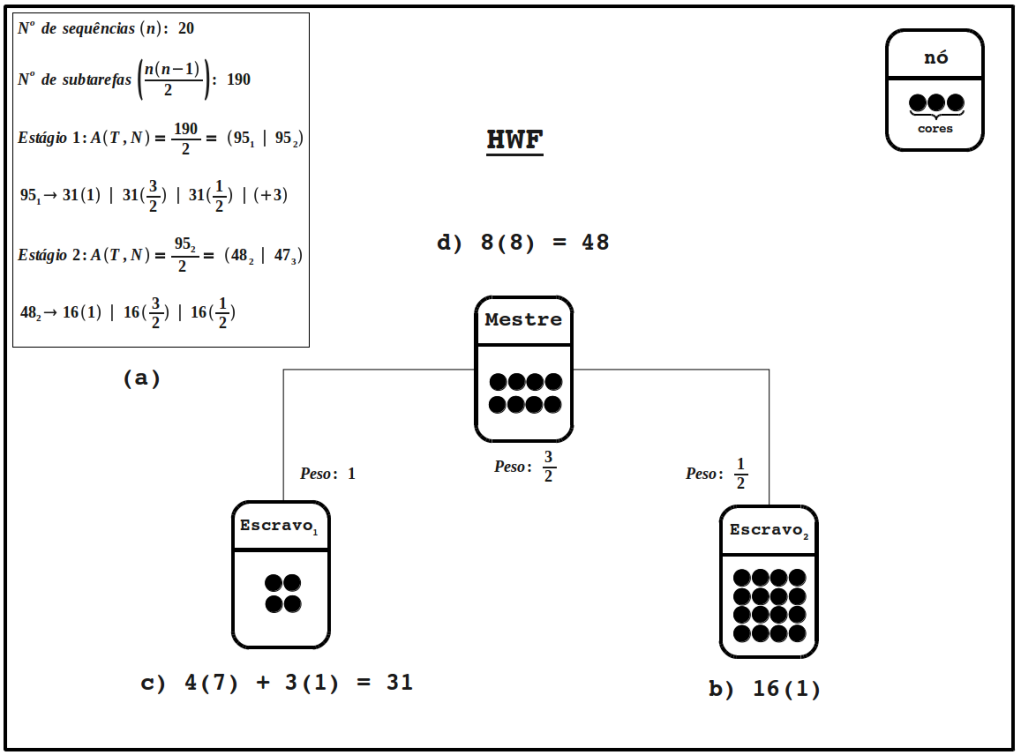

Figura 5.7: Exemplo de funcionamento da estratégia HWF.

$Escravo_1$ recebeu peso 1 e o nó $Escravo_2$ recebeu $\frac{1}{2}$. Em seguida, tem-se o cálculo dos *chunks* de tarefas que serão atribuídos a cada nó. Primeiro, divide-se o valor total de tarefas do estágio 1 (95_1) pelo número de nós (3). O resultado inteiro desta divisão (31) é adotado como o *chunk* do estágio 1 e o resto é descartado.

Depois, o nó mestre irá calcular o número de tarefas que será alocado a cada nó escravo n_i a partir do peso de W_{n_i}. Em (b), o nó $Escravo_2$ recebeu o peso $\frac{1}{2}$ correspondente à sua velocidade de processamento, então o mestre calculará uma quantidade $31 \cdot \left(\frac{1}{2}\right) = 15$ tarefas para o nó $Escravo_2$. Do mesmo modo, em (c), o nó mestre calculará o número de tarefas para o nó $Escravo_1$, $31 \cdot (1) = 31$ tarefas. E, finalmente, em (d), o nó mestre estimará que o número de tarefas que ele mesmo deverá executar será de $31 \cdot \left(\frac{3}{2}\right) = 46$ tarefas.

Porém, antes de iniciar a distribuição dos *chunks* de tarefas, o nó mestre verifica se todas as 95_1 tarefas do estágio 1 foram agrupadas nos devidos *chunks*. E, para isso, o nó mestre soma todas as tarefas de todos os *chunks* e depois os subtrai do número de tarefas do estágio 1, assim: $95_1 - (\ 15_{Escravo_2} + 31_{Escravo_1} + 46_{Mestre}) = 3$. Este resultado maior que 0 (zero) significa a divisão dos *chunks* não foi exata e que ainda existem tarefas a serem alocadas (por exemplo, `(+3)` tarefas pendentes no estágio do item (a) da figura 5.7). Neste contexto, o nó mestre deverá aplicar o

seguinte critério:

a - Se um *chunk* possuir um número de tarefas menor do que o número de *cores* do nó para o qual ele será alocado, então o nó mestre irá colocar as tarefas pendentes neste *chunk* até igualar o número de tarefas com o número de *cores*;

b - Em seguida, se ainda existir alguma tarefa fora dos *chunks*, então o nó mestre irá colocá-las no *chunk* alocado para o nó n_i com o maior peso W_{n_i}.

Na figura 5.7, as 3 tarefas pendentes do estágio 1 foram distribuídas nos *chunks* da seguinte forma: em (b), 1 tarefa foi para o *chunk* do $Escravo_2$ para completar a alocação de 16 tarefas para os 16 *cores* existentes. Depois, em (c), as 2 tarefas restantes formam para o nó mestre que possui o valor de peso W_{Mestre} maior entre os três nós do *cluster multicore* heterogêneo.

Seguindo-se para o estágio 2 no item (a) da figura 5.7, note-se o comportamento fatorial no qual se baseia o WHF, onde aplica-se o mesmo procedimento do estágio 1, porém o número de tarefas pendentes é igual 95_2. Assim, aplica-se a equação de alocação no estágio 2 para definir quantidade de tarefas pendentes que será tratadas neste estágio. Neste caso, a divisão não foi exata como no estágio 1, então o nó mestre irá tratar 48_2 tarefas pendentes no estágio 2 e 47_3 tarefas pendentes no estágio seguinte. Semelhante ao realizado no estágio 1, o nó mestre irá calcular os *chunks* que serão atribuídos a cada nó, dividindo-se as 48_2 tarefas pendentes pelos números de nós. Note-se que o resultado desta divisão foi exato e igual a 16. O nó mestre irá, em seguida, aplicar os pesos de desempenho de cada nó para definir o *chunk* de tarefas que será alocado em um nó específico (ver a última linha do item (a) da figura 5.7), assim:

- $Escravo_1$: $16 \cdot (1) = 16$ tarefas;

- $Mestre$: $16 \cdot \left(\frac{3}{2}\right) = 24$ tarefas;

- $Escravo_2$: $16 \cdot \left(\frac{1}{2}\right) = 8$ tarefas.

Assim, tem-se os valores dos *chunks* que serão alocados a cada nó, porém é necessário verificar se alguma tarefa pendente não foi colocada em algum *chunk*. Como no estágio 1, o nó mestre irá calcular: $48_2 - (\ 16_{Escravo_2} + 16_{Escravo_1} + 16_{Mestre}) = 0$. Como este resultado é igual a 0 (zero), o nó mestre assume que todas as tarefas pendentes do estágio 2 foram colocadas em seus *chunks* e, em seguida, inicia a distribuição das tarefas, enviando os *chunks* para o respectivos nós do *cluster multicore* heterogêneo.

O procedimento descrito para os estágios 1 e 2 acima prossegue pelos estágios seguintes até que o nó mestre não possua mais tarefas pendentes para serem alocadas.

5.5 O Modelo Híbrido de Programação

A ideia de utilizar dois paradigmas de comunicação num modelo híbrido de programação já foi utilizada em aplicações científicas reais, como em [20, 38, 39, 48], e em *clusters* SMP compostos de processadores tradicionais (com somente um *core*), como no trabalho apresentado em [22]. Recentemente, o modelo híbrido foi considerado para também para aproveitar os recursos de ambientes compostos de processadores *multicore*, como *clusters multicore* homogêneos e heterogêneos, como em [68, 16, 82, 81].

Para a comunicação via troca de mensagens, o padrão estabelecido na computação de alto desempenho é o emprego de uma biblioteca MPI para comunicação entre os nós do *cluster*, seja este homogêneo ou heterogêneo. Para a comunicação via memória compartilhada, a especificação OpenMP tem sido usada como interface para a construção de aplicações *multithread* devido à sua facilidade de uso, padronização e portabilidade.

Em 2009, o trabalho apresentado em [68] trouxe uma taxonomia para os modelos de programação em plataformas híbridas. A solução híbrida adotada nesta dissertação é um exemplo do modelo chamado *Masteronly* citado em [68]. Neste modelo, somente um processo MPI é usado em cada nó e nenhuma chamada MPI é efetuada entre os *cores* do mesmo nó.

Assim, nesta dissertação, o modelo híbrido de programação consiste na utilização de dois paradigmas de comunicação, troca de mensagens e memória compartilhada, para a explorar a capacidade de processamento de um *cluster multicore* heterogêneo durante a execução distribuída da ferramenta de alinhamento múltiplo de sequências DIALIGN-TX.

5.5.1 Trocas de mensagens com MPI

A visão geral da solução apresentada na seção 5.3 mostrou que o nó mestre troca mensagens com os nós escravos para descobrir os recursos do *cluster multicore* heterogêneo, distribuir tarefas e coletar resultados.

Esta comunicação foi construída utilizando-se uma interface de programação com troca de mensagens baseada na biblioteca MPI. Um processo mestre é criado para controlar a comunicação entre os nós do *cluster*. Os processos escravos somente se comunicam com o processo mestre para responder solicitações do processo mestre ou enviar os resultados de seus cálculos.

Para compor a solução híbrida com OpenMP, o processo mestre irá criar um processo escravo para cada nó do *cluster multicore* heterogêneo. Cada processo escravo irá receber e executar as tarefas enviadas pelo nó mestre distribuindo-as entre diversas *threads* e, quando terminar os seus cálculos, informará a conclusão para o processo mestre.

A adaptação do programa DIALIGN-TX para uma execução distribuída com troca de mensagens com MPI, exigiu a inclusão de rotinas para gerenciamento do ambiente MPI, bem como métodos para viabilizar a comunicação entre os processos mestre e escravos. Eis as principais rotinas utilizadas na

programação da comunicação entre mestre e escravos:

a) `MPI_Init( &argc, &argv)`: responsável por inicializar o ambiente de execução MPI;

b) `MPI_Comm_size( comm, &numprocs )`: recupera em `numprocs` o tamanho do grupo associado a um comunicador (`comm`); no caso, foi adotado o valor *default* `MPI_COMM_WORLD`); através de `numprocs` o processo mestre conhece quanto nós existem no *cluster multicore* heterogêneo;

c) `MPI_Comm_rank( comm, &my_rank )`: recupera o *rank* do processo que fez a chamada a esta rotina dentro de um mesmo comunicador (neste caso desta solução,
`MPI_COMM_WORLD`);

d) `MPI_Get_processor_name( &processorName )`: recupera o nome do processador onde o processo (mestre ou escravo) está sendo executado - este método foi utilizado na construção do programa *profiler* usado para a determinação do poder computacional (seção 5.3.1);

e) `MPI_Isend( lista de parâmetros )`: rotina para envio não blocante de mensagens, utilizada em toda comunicação realizada entre os processos mestre e escravo. Na `lista de parâmetros`, os dados mais comuns são as iterações das comparações par a par (tarefas) enviadas pelo processo mestre e os fragmentos resultantes encontrados pelos nós escravos;

f) `MPI_Irecv( lista de parâmetros )`: rotina para recepção não blocante de mensagens. Utilizada principalmente pelos escravos na espera por tarefas a serem executadas. Foi empregada também no processo mestre para esperar o resultados dos cálculos dos processos escravos;

g) `MPI_Wait( &request, &status )`: esta rotina bloqueia a execução que uma operação de envio ou recepção de mensagens não blocantes (itens (e) e (f)) tenha se completado;

h) `MPI_Finalize()`: rotina responsável pelo encerramento do ambiente de execução MPI. No programa principal do DIALIGN-TX, uma chamada a esta rotina foi colocada logo após o encerramento da fase 1 do AMS, pois nenhuma outra rotina MPI foi usada depois deste ponto.

Seguindo-se o modelo híbrido *Masteronly* [68], os processos MPI são responsáveis pela criação das *threads* OpenMP em cada nó *multicore*. O processo MPI em cada nó é também a *thread* principal responsável pela comunicação com os outros processos MPI, antes e depois do cálculos a serem realizados. Assim, não há comunicação entre o processo mestre e os processos escravos durante a execução das tarefas. O processo mestre envia as tarefas para os processos escravos e estes somente enviam mensagens para o mestre quando solicitados ou tiverem terminado seus cálculos.

5.5.2 Memória compartilhada com OpenMP

A solução com memória compartilhada empregou a biblioteca OpenMP na execução *standalone* do DIALIGN-TX com OpenMP e na programação dos nós escravos da estratégia híbrida MPI/OpenMP.

O passo inicial foi adaptar o código original do DIALIGN-TX para que suportasse a execução com múltiplas *threads* através da biblioteca OpenMP. Esta etapa serviu para validar a comunicação entre os *cores* existentes em cada nó. Eis o que foi feito:

a) O arquivo `Makefile` fornecido com o programa DIALIGN-TX foi alterado para suportar a compilação com suporte ao OpenMP. A *flag* `-fopenmp` foi adicionada ao final das linhas de compilação e ligação correspondentes;

b) Na solução *standalone* do DIALIGN com OpenMP, a principal alteração foi realizada no arquivo contendo a a função `find_all_diags( lista de parâmetros )`, onde acrescentou-se as diretivas OpenMP `#pragma omp parallel for private` no laço `for` mais externo responsável pelo cálculo das comparações par a par entre todas as sequências de entrada;

c) Cláusulas OpenMP foram acrescentadas à diretiva `#pragma omp parallel for`, porém duas merecem nota: `num_threads`, determina quantas *threads* serão criadas paralelamente e receberão as iterações do laço *for*); a segunda, `schedule`, determina o tamanho do *chunk* (número de iterações do laço) que será distribuído entre as *threads* e como será o escalonamento entre elas). Na solução *standalone* do DIALIGN-TX com OpenMP, as opções utilizadas foram *dynamic* para um escalonamento dinâmico e um valor de *chunk* igual a 2 (o valor do *chunk* foi obtido empiricamente);

d) Na solução híbrida do DIALIGN-TX com MPI/OpenMP, a função original do DIALIGN-TX `find_all_diags( lista de parâmetros )` foi alterada substituindo-se o laço responsável pela comparações par a par por um um time de *threads* criado com a diretiva `#pragma omp sections` onde a principal cláusula utilizada foi a `PRIVATE( lista de parâmetros )`, pois sua lista de parâmetros continha índices das sequências que deveriam ser comparadas par a par em cada *thread* criada;

e) Para execução da solução *standalone* do DIALIGN-TX com OpenMP, foi acrescentada uma opção de linha comando para permitir que o usuário escolha quantas *threads* deverão ser criadas durante o cálculo da fase 1 do AMS. O usuário precisa passar a opção `-k` seguida do número de *threads* desejado.

Durante a execução OpenMP da solução distribuída do DIALIGN-TX com MPI/OpenMP não há qualquer comunicação entre os nós do *cluster*.

Assim, os recursos de memória compartilhada entre as *threads* ficam dedicados à execução das tarefas solicitadas pelo nó mestre. Tão logo a seção OpenMP é encerrada em um nó escravo, o time de *threads* é fechado e a execução segue com a *thread* principal responsável pela comunicação MPI com o processo mestre. De modo semelhante, no nó mestre, quando a execução OpenMP é encerrada, o time de *threads* também é fechado porém o nó mestre pode ainda possuir tarefas pendentes, então o processo principal pode enviar novas mensagens MPI para os escravos e reiniciar sua própria seção OpenMP para execução de novas tarefas. Quando não existirem mais tarefas pendentes, primeiro o ambiente OpenMP é encerrado com o fechamento do time de *threads* e depois o ambiente MPI com a chamada à rotina `MPI_Finalize()`.

Capítulo 6

Resultados Experimentais

6.1 Descrição do Ambiente

O ambiente de testes foi construído no Laboratório de Processamento de Alto Desempenho (LPAD), localizado no Centro Internacional de Física da Matéria Condensada da Universidade de Brasília (UnB).

Foram utilizados 3 (três) computadores dispostos numa arquitetura em *cluster*, conectados por um roteador pré-existente no LPAD e sem acesso à internet. A representação do ambiente pode ser vista na figura 6.1.

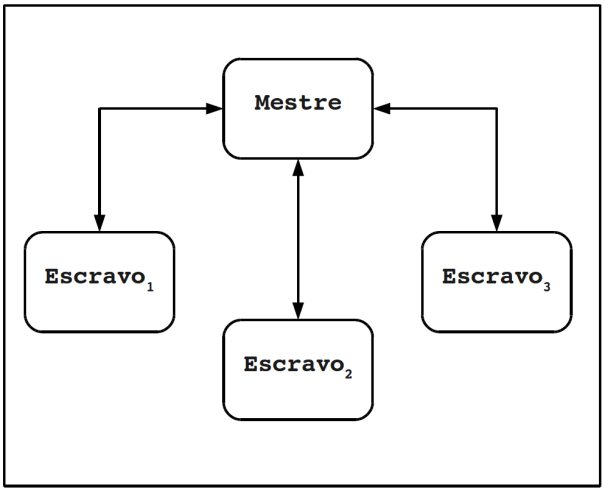

Figura 6.1: Modelo mestre-escravo em *cluster*.

Conforme mostra a tabela 6.1, os três nós possuem características de hardware diferentes entre si. O primeiro nó, chamado de node1, possui um processador *multicore* com 8 (oito) *cores* Intel Xeon de 3.2 GHz e cache L2 de 2048 KB cada. A memória principal do node1 tem capacidade de 4 GB.

O segundo, node2, trata-se de um processador *multicore* com 16 *cores* do tipo Intel Itanium 2 de 1.6 GHz e cache L3 de 6 MB compartilhada entre todos eles. A memória principal do node2 tem capacidade de 16 GB.

O node3 tem apenas 4 *cores* do tipo Intel Xeon de 3.6 GHz e cache L2 de 2048 KB. Sua memória principal tem capacidade de 4 GB.

Quanto à arquitetura, os nós node1 e node3 possuem conjunto de instruções de 32 bits e uma extensão que suporta 64 bits (X86_64). Já o node2, Itanium 2, tem um conjunto IA-64 exclusivamente de 64 bits. Note que o *clock* dos processadores que compõem o node2 é bem mais baixo que o *clock* dos processadores dos outros nós (tabela 6.1) .

O sistema operacional escolhido para instalação em cada máquina foi o Fedora Linux. A versão mais recente disponível à época era de número 12. Uma ressalva deve ser feita para o Itanium 2, onde somente foi possível instalar a versão 9 do Fedora, por esta suportar a arquitetura IA-64. O Fedora Linux foi utilizado, principalmente, pelo suporte e distribuição gratuita para arquiteturas IA-64 e facilidade de instalação e configuração de um *cluster* sem acesso à internet.

	Nós do cluster		
Processador	node1	node2	node3
Fabricante	Intel	Intel	Intel
Modelo	Xeon	Itanium 2	Xeon
Arquitetura	X86_64	IA-64	X86_64
Clock (GHz)	3.2	1.6	3.6
# *Cores*	8	16	4
Cache	2048 KB (L2)	6 MB (L3)	2048 KB (L2)
Memória (GB)	4	16	4
Disco Rígido (GB)	120	180	120
Sistema Operacional	Fedora 12	Fedora 9 (IA-64)	Fedora 12
Compilador	gcc 4.4	gcc 4.3.1	gcc 4.4

Tabela 6.1: Dados de hardware e software instalados em cada máquina do *cluster* heterogêneo.

Para que fosse realizada a comparação com o DIALIGN-TX original, o mesmo foi obtido via internet em [75] e instalado em todos os nós.

Conforme decisão de projeto, o modelo de programação para comunicação entre os nós do *cluster* foi o de troca de mensagens. A biblioteca escolhida foi a MPI, por ser um padrão já estabelecido na computação de alto desempenho. A implementação MPICH2 [4] foi a primeira escolha tendo em vista o suporte à versão mais recente da biblioteca MPI-2.

A instalação do MPICH2 em cada máquina foi realizada, porém, a comunicação entre os nós não funcionou durante os testes iniciais de ambiente. Essa tentativa sem sucesso confirmou o que estava previsto na documentação do MPICH2: até a conclusão desta dissertação, o MPICH2 ainda não suportava *clusters* com arquiteturas heterogêneas [4], sendo recomendado o uso da versão anterior, MPICH1 [3, 4].

A partir de então, foi instalado o pacote `mpich-1.2.7p1.tar.gz` que

implementa a MPI-1. O endereço para download usado à época foi [3].

Para a comunicação entre os *cores* dentro de um mesmo nó, foi utilizada a biblioteca OpenMP 3.0, já suportada no compilador `gcc` desde a sua versão 4.3.1 [31]. O sistema operacional Fedora instalado nas máquinas já possuía uma versão do `gcc` com suporte ao OpenMP.

A característica híbrida da estratégia elaborada neste trabalho advém do emprego simultâneo destas duas bibliotecas no modelo de programação mestre-escravo adotado. O nó mestre irá se comunicar com os nós escravos via mensagens MPI para distribuir tarefas e coletar resultados. No processamento dentro de cada nó, o OpenMP foi empregado na programação da comunicação via memória compartilhada.

Neste ponto, o *cluster* heterogêneo *multicore* estava configurado e prontos para o início dos testes.

6.2 Execução *standalone*

Os primeiros testes com o *cluster multicore* heterogêneo foram realizados em cada nó separadamente. Esta etapa foi chamada de execução *standalone* e seu objetivo principal foi obter dados absolutos de desempenho sobre cada máquina.

O primeiro teste foi executar o DIALIGN-TX em cada nó do *cluster* e registrar os tempos de execução para o cálculo do AMS. Para isso, era necessário definir um conjunto de sequências biológicas para fornecer como entrada para o programa.

O grupo de sequências escolhido foi recuperado na base de dados de proteínas *Pfam* [25] através da página para download em [14]. Neste endereço, foi possível baixar o arquivo `PF02171.full_length_sequences.fasta.gz` que contém 766 sequências biológicas dispostas num arquivo em formato `fasta` (seção 2.3.1).

Como esse arquivo, originalmente, contém um conjunto diversificado de sequências quanto ao seus tamanhos, optou-se por utilizar um subconjunto formado pelas últimas 324 sequências do arquivo e, com isso, obter uma tamanho médio calculado de 1029 caracteres.

Note-se que, deste ponto em diante, este subconjunto de sequências, passou a ser referenciado como `324(1029)-PIWI` e foi utilizado em todos os testes com a ferramenta DIALIGN-TX, inclusive nos preliminares de validação dos alinhamentos múltiplos computados.

Para registrar os tempos de execução, foi escolhida a instrução `\time`, disponível nos sistemas operacionais *GNU/Linux*. Esta instrução, usada com a opção `-o`, permitiu a leitura dos tempos de execução percebidos a partir do momento em que se pressionasse a tecla `<ENTER>` até o retorno do *prompt* de comando e, em seguida, direcioná-los para um arquivo específico pré-estabelecido.

6.2.1 Execução sequencial

Os tempos de execução para os nós node1, node2 e node3 foram registrados na tabela 6.2. Observe-se que em cada máquina foi utilizado somente um *core* durante todos os cálculos da fase 1 do AMS no DIALIGN-TX (seção 5.3.3, figura 5.3) e do AMS completo (seção 3.3.4 figura 3.4), fornecendo como entrada o conjunto de sequências biológicas `324(1029)-PIWI`. Note-se, também, que a diferença de arquitetura possibilitou que o nó contendo o processador *Itanium 2* (node2) tivesse um percentual de tempo de execução sequencial menor durante a fase 1 do AMS no DIALIGN-TX. Na última coluna, vê-se que o node2 gastou somente 38,76% para executar a fase 1, enquanto que os outros nós precisaram gastar mais de 50% de seu tempo nesta fase.

	Tempos de execução sequencial (segundos)		
Nós do *cluster*	DIALIGN-TX (AMS completo)	DIALIGN-TX (AMS: Fase 1)	Percentual da Fase 1
node1	9452,00	4986,00	52,75%
node2	21488,00	8328,00	38,76%
node3	8610,00	4436,00	51,52%

Tabela 6.2: Tempos de execução do primeiro AMS calculado com o DIALIGN-TX original para o conjunto de sequências de entrada (`324(1029)-PIWI`). Também foram registrados os tempos de execução somente para a primeira fase do AMS. Na última coluna está registrado o percentual corresponde ao tempo de execução demandado pela fase 1 do AMS no DIALIGN-TX em cada nó do *cluster*.

Tendo em vista o tempo consumido durante a computação sequencial das comparações par a par (tabela 6.2), nota-se que a distribuição dos cálculos da fase 1 do AMS terá impacto somente em 38,76% a 52,75% do tempo total necessário para uma execução sequencial do DIALIGN-TX encontrar o AMS completo, rodando nestas máquinas do *cluster multicore* heterogêneo.

Neste ponto, foram escolhidos outros conjuntos de sequências biológicas que seriam utilizados em definitivo, juntamente com o `324(1029)-PIWI`, como entrada para o programa em todos os testes subsequentes.

Os arquivos foram selecionados na base de dados *UniRef* [77] com o intuito de fornecer diferentes cargas de trabalho para os testes com a estratégia proposta. São eles:

a) `uniref_100x600.fasta`: composto por 100 sequências de tamanho médio 600 resíduos;

b) `uniref_200x800.fasta`: composto por 200 sequências de tamanho médio 800 resíduos;

c) `uniref_300x1000.fasta`: composto por 300 sequências de tamanho médio 1000 resíduos;

d) `uniref_400x1200.fasta`: composto por 400 sequências de tamanho médio 1200 resíduos.

DIALIGN-TX (sequencial)	Tempos de execução (segundos)		
# Sequências (Tamanho Médio)	node3	node1	node2
100(600)	198,49	222,57	322,55
200(800)	1251,25	1413,24	2255,44
300(1000)	4596,00	5187,00	8956,00
324(1029) - PIWI	4436,00	4986,00	8328,00
400(1200)	9494,00	10724,00	18545,00

Tabela 6.3: Tempos de execução sequencial do DIALIGN-TX original para a fase 1 do AMS. A carga é representada pela quantidade de sequências de entrada. Os tempos de execução foram obtidos com somente 1 *core* em cada nó do *cluster*

.

A principal diferença entre os conjuntos acima e o `324(1029)-PIWI` está nos tamanhos das sequências constituintes. Enquanto os tamanhos daquelas sequências possuem uma variação de no máximo 50 resíduos para mais ou para menos, no `324(1029)-PIWI`, existem 178 sequências com tamanhos variando entre 500 e 999 caracteres, 142 sequências com tamanhos entre 1000 e 1999 resíduos e, ainda, 4 com mais de 3000 resíduos.

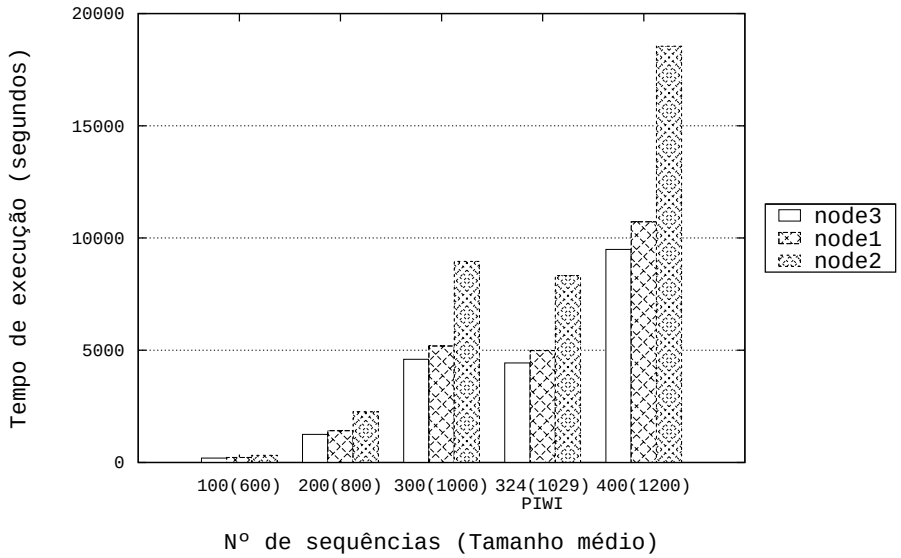

Figura 6.2: Tempos de execução sequencial do DIALIGN-TX original referentes à fase 1 do AMS. A carga é representada pela quantidade de sequências de entrada. Os tempos de execução foram obtidos em cada nó do *cluster* individualmente: node3 (4 *cores*), node1 (8 *cores*) e node2 (16 *cores*).

Então, apesar de seu tamanho médio ser 1029 caracteres, o conjunto `324(1029)-PIWI` representa uma carga bem diversificada se comparado ao conjunto contido no arquivo `uniref_300x1000.fasta`.

Em seguida, foram realizados testes com os conjuntos definitivos de sequências biológicas, executando-se a versão sequencial do DIALIGN-TX e medindo-se o tempo necessário para se calcular todas as comparações par a par. Os primeiros resultados podem ser lidos na tabela 6.3 e comparados no gráfico da figura 6.2.

Em cada máquina, na linha de comando, foi executado o seguinte comando:

```
$ ./dialign-tx ../conf /path/seqFile.fasta
```

No gráfico da figura 6.2, pode-se observar que o node2 teve o pior tempo de execução sequencial para computar a fase1 do AMS para todos os conjuntos de entrada fornecidos. Isto já era esperado baseando-se nos resultados do teste inicial na tabela 6.2. Do mesmo modo, os outros nós obtiveram tempos mais próximos em relação aos obtidos pelo node2, sendo que o node3 sempre conseguiu os melhores tempos de execução sequencial.

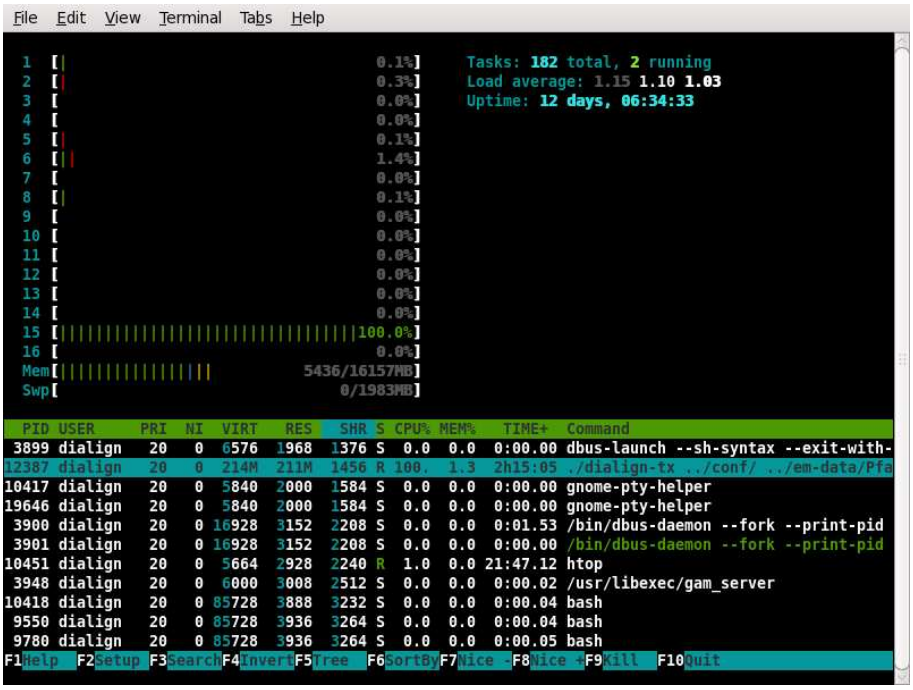

Figura 6.3: Visualização da execução sequencial do DIALIGN-TX original para a fase 1 do AMS no node2 (16 *cores*). A ferramenta utilizada foi o `htop` [60]. Na linha de comando selecionada, lê-se que após 2 horas e 15 minutos o `dialign-tx` continua sendo executado somente no *core* 15 que permanece em 100% de uso.

Os valores absolutos da tabela 6.2 mostram outro lado da influência da arquitetura em cada *core* existente nos nós. O node2, por exemplo, foi acima

de duas vezes mais lento que outros nós para calcular o AMS completo e acima de uma vez e meia mais lento para calcular a fase 1 do AMS. Isso, provavelmente, pode ser explicado por causa da grande diferença entre o *clock* dos processadores do node2 (`1.6 GHz`) e os *clocks* dos processadores dos demais nós (`3.2 GHz e 3.6 GHz`). Como a comparação par a par possui elevada demanda de CPU, essa diferença de velocidade causa grande impacto sobre o tempo de execução.

O processo de execução do DIALIGN-TX durante os testes foi acompanhado com a visualização fornecida pelo programa `htop` [60]. A figura 6.3 mostra uma instância do `htop` durante a execução sequencial do DIALIGN-TX no node2 (16 *cores*). Nela, a linha de comando selecionada mostra que o `dialign-tx` está há 2 (duas) horas e 15 minutos executando a fase 1 do AMS para o conjunto sequências `324(1029)-PIWI`, mantendo o *core* número 15 em 100%.

6.2.2 Execução com OpenMP

Para a realização dos testes com o OpenMP, foi necessário apenas rodar a versão do DIALIGN-TX com OpenMP em cada máquina separadamente e, em seguida, comparar o AMS resultante com aquele obtido na execução sequencial. Feitos os ajustes necessários, descritos na seção 5.5.2, esta versão estava pronta para os testes com os conjuntos definitivos de sequências.

A execução do DIALIGN-TX com OpenMP foi realizada em cada nó do *cluster multicore* heterogêneo, executando-se a seguinte linha de comando:

```
$ \time -o timeFile.out ./openMP-dialign-tx -kN ../conf ../path/ seqFile.fasta
```

A opção `-kN` foi acrescentada ao programa e o valor de `N` corresponde à quantidade de *threads* que serão criadas durante a execução. Note-se que todos os *cores* de cada nó foram usados e que foi criada um *thread* por core. A tabela 6.4 e o gráfico da figura 6.4 apresentam os resultados obtidos com a execução *standalone* do DIALIGN-TX com OpenMP para os cinco conjuntos de entrada fornecidos.

DIALIGN-TX com OpenMP	Tempos de execução (segundos)		
# Sequências (Tamanho Médio)	node3	node1	node2
100(600)	77,39	43,02	44,06
200(800)	487,44	271,07	290,44
300(1000)	1872,32	1054,83	1149,66
324(1029) - PIWI	1725,31	698,07	1071,60
400(1200)	3716,00	2074,30	2373,06

Tabela 6.4: Tempos de execução *standalone* do DIALIGN-TX com OpenMP para a fase 1 do AMS. A carga é representada pela quantidade de sequências de entrada. Os tempos de execução foram obtidos em cada nó do *cluster* individualmente: node3 (4 *cores*), node1 (8 *cores*) e node2 (16 *cores*).

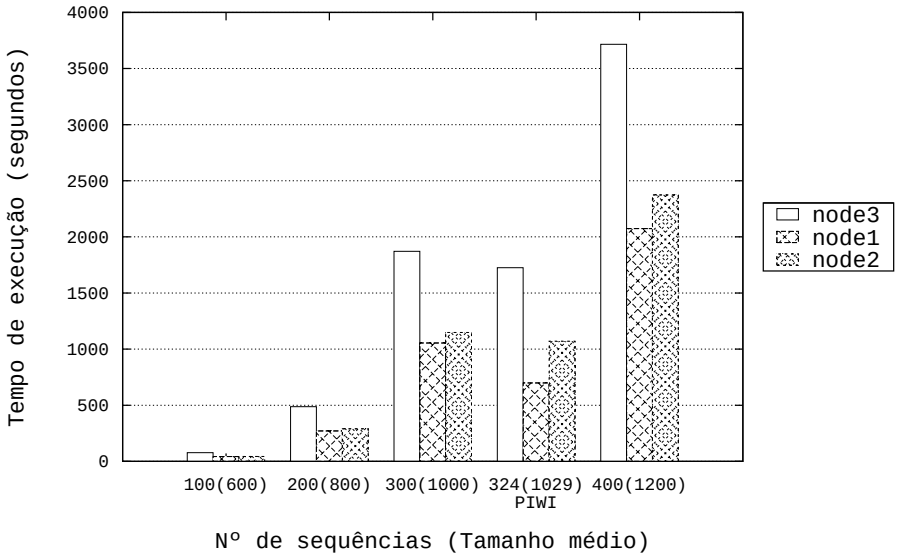

Figura 6.4: Tempos de execução *standalone* do DIALIGN-TX com OpenMP referentes à fase 1 do AMS. A carga é representada pela quantidade de sequências de entrada. Os tempos de execução foram obtidos em cada nó do *cluster* individualmente: node3 (4 *cores*), node1 (8 *cores*) e node2 (16 *cores*).

Durante os testes executados, seguiu-se a decisão de se criar tantas *threads* quantos forem os *cores* disponíveis em um nó. Assim, na execução do DIALIGN-TX com OpenMP foram criadas 8 *threads* no node1, 16 *threads* no node2 e 4 no node3.

Pelo gráfico da figura 6.4 pode-se visualizar que a distribuição dos cálculos das comparações par a par entre as *threads* conseguiu reduzir o tempo de execução da fase 1 do DIALIGN-TX. Pode-se ver ainda, que o node3 com apenas 4 *cores* e, consequentemente, 4 *threads* teve o pior tempo de execução com OpenMP para todos os conjuntos de entrada. O node2, mesmo com 16 *threads*, não conseguiu melhor resultado que o node1 com apenas 8 *threads*. Novamente, explorar a arquitetura *multicore* do node2 com OpenMP reduziu o tempo de execução deste em relação ao node3, mas não foi suficiente para obter tempos melhores que os obtidos com a arquitetura *multicore* do node1.

Observando-se a tabelas 6.3 e 6.4 e os gráficos 6.2 e 6.4, pode-se perceber que as alterações realizadas na ferramenta DIALIGN-TX (seção 5.5.2) atingiram o resultado esperado de reduzir o tempo de execução necessário para se calcular a fase 1 do AMS, distribuindo-se as iterações do laço responsável pelas comparações par a par (seção 5.3.3, figura 5.3) entre as diversas *threads* criadas.

Na figura 6.5, observa-se que as várias *threads* criadas foram associadas aos *cores* disponíveis no node2. A linha selecionada mostra a linha de comando `./openMP-dialign-tx -k16` sendo executada a 9 minutos e 40

segundos. Neste momento, pode-se ver no gráfico que os 8 primeiros *cores* já não estão sendo totalmente aproveitados, enquanto que os 8 últimos permanecem em 100% de utilização.

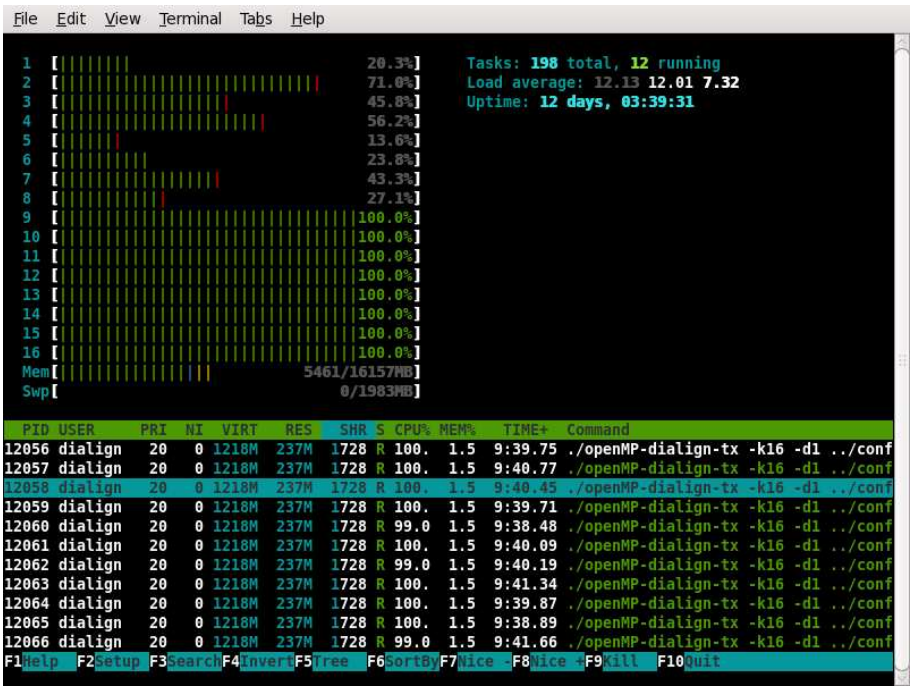

Figura 6.5: Visualização da execução do DIALIGN-TX com OpenMP referente à fase 1 do AMS no node2. Sem a cláusula `schedule(dynamic, 2)` para balancear a distribuição das iterações entres as *threads*, após 09 minutos e 30 segundos alguns dos *cores* já estão ociosos, enquanto outros permanecem em 100%.

Por último, a figura 6.6 mostra o comportamento esperado durante a execução do DIALIGN-TX com OpenMP em um nó do *cluster*. Conforme descrito na seção 5.5.2, acrescentando-se a cláusula `schedule(dynamic, 2)` à diretiva `#pragma omp for parallel` foi possível dividir as iterações responsáveis pelas comparações par a par e atingir o objetivo de manter os *cores* disponíveis em 100% de utilização pela maior quantidade de tempo de execução possível.

A linha selecionada `./openMP-dialign-tx -k16` mostra que após 17 minutos e 07 segundos calculando a fase 1 do AMS, todos os *cores* do node2 permanecem em 100%. Vale observar que 22 segundos depois do registro desta imagem, todos os *cores* haviam concluído seus cálculos e o programa estava encerrado, com um tempo total de execução de 17 minutos e 29 segundos para o cálculo da fase 1 do AMS.

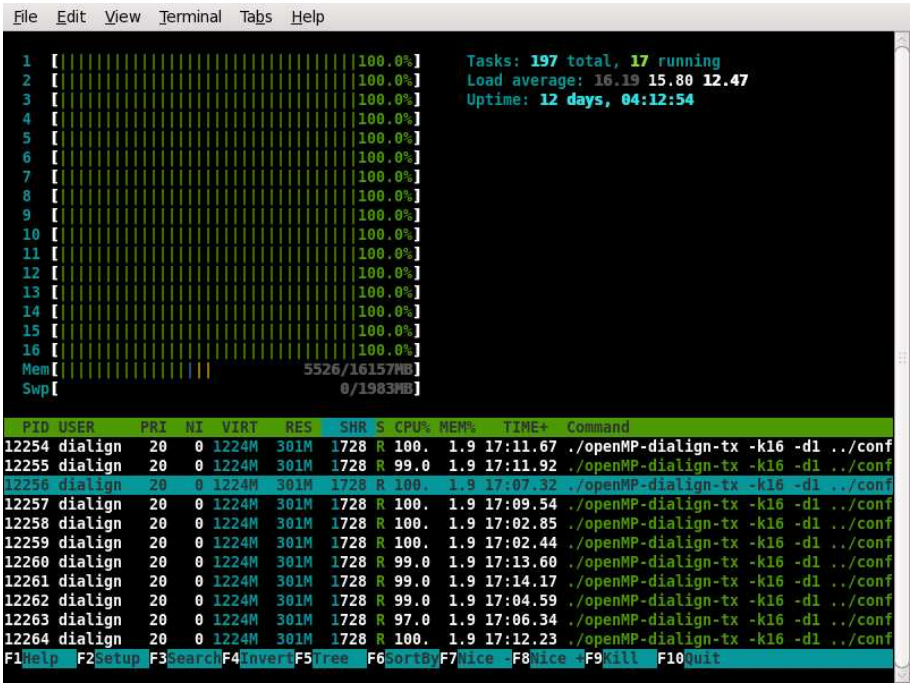

Figura 6.6: Visualização da execução balanceada do DIALIGN-TX com OpenMP para a fase 1 do AMS. A linha indica mostra após 17 minutos de execução, próximo do término, todos os 16 *cores* do node 2 continuam em 100%.

6.3 Execução distribuída no *cluster multicore* heterogêneo

Uma vez concluídos os testes com o execução *standalone*, compôs-se o ambiente mestre/escravo no *cluster multicore* heterogêneo para a execução distribuída do DIALIGN-TX com MPI/OpenMP, conforme descrito na seções 5.3 e 5.5.1 e 6.1.

Durante os testes, os 28 *cores* disponíveis nos nós node1, node2 e node3 foram utilizados a 100% de sua capacidade de processamento pela maior quantidade de tempo possível, de acordo com a estratégia de alocação de tarefas definida. Os resultados obtidos com estas estratégias adaptadas para o *cluster multicore* heterogêneo, apresentadas na seção 5.4, foram registrados como HSS, HFixed e HWF nas tabelas e gráficos desta seção.

Os testes foram agrupados de acordo com o nó definido como mestre e, depois, ordenados conforme a quantidade de *cores* do mestre, sendo anotados os tempos de execução, calculados os *speedups* e construídos os respectivos gráficos. Desta forma, as seções seguintes apresentam o três casos: node3 (4 *cores*) atuando como mestre, depois o node1 (8 *cores*) como mestre e, finalmente, o node2 (16 *cores*).

6.3.1 Caso 1: node3 atuando como nó mestre

No primeiro caso, a tabela 6.5 e a figura 6.7 mostram os tempos de execução obtidos para cada conjunto de sequências biológicas fornecido como entrada, quando o node3 foi definido como nó mestre. Pode-se notar que mesmo aumentando a carga do sistema (primeira coluna da tabela 6.5), a estratégia HWF obteve sempre o menor tempo de execução, enquanto a estratégia HSS obteve os piores tempos. Este comportamento era esperado considerando-se as características das políticas de alocação de tarefas que foram usadas como referência na implementações das estratégias propostas (seção 5.4).

Observe-se ainda que, conforme visto na seção 6.2.1, a maior diversidade de tamanhos das sequências do conjunto `324(1029)-PIWI` fornece uma carga menor para o sistema do que o conjunto `300(1000)`, mesmo que o tamanho médio e o número de sequências tenha aumentado. Com isso, os tempos de execução diminuíram em cada estratégia da tabela 6.5 e da figura 6.7, enquanto a quantidade e o tamanho médio das sequências aumentou do conjunto `300(1000)` para o `324(1029)-PIWI`.

DIALIGN-TX com MPI/OpenMP	Tempos de execução (segundos)		
# Sequências (Tamanho Médio)	HSS	HFixed	HWF
100(600)	29,67	19,23	11,95
200(800)	129,38	88,53	49,83
300(1000)	535,93	376,21	209,38
324(1029) - PIWI	469,85	257,29	160,93
400(1200)	950,32	695,28	367,21

Tabela 6.5: Tempos de execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node3 (4 *cores*). Escravos: node1 (8 *cores*) e node2 (12 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os tempos de execução foram obtidos para cada algoritmo de alocação de tarefas implementado: *Hybrid Self-Scheduling* (HSS), *Hybrid Fixed* (HFixed) e *Hybrid Weight Factoring* (HWF).

A tabela 6.6 e a figura 6.8 apresentam os *speedups* calculados para o caso do node3 como mestre na execução distribuída do DIALIGN-TX com MPI/OpenMP. Para o cálculo do *speedup*, os tempos de execução da estratégia HSS da tabela 6.5 foram utilizados como referência. Como exemplo, eis uma descrição de os valores de *speedup* da primeira linha da tabela 6.6 para o conjunto de entrada `100(600)` foram calculados, por exemplo, como abaixo:

- HSS: $Speedup = 29,67 \div 29,67 = 1,00$
- HFixed: $Speedup = 29,67 \div 19,23 = 1,54$
- HWF: $Speedup = 29,67 \div 11,95 = 2,48$

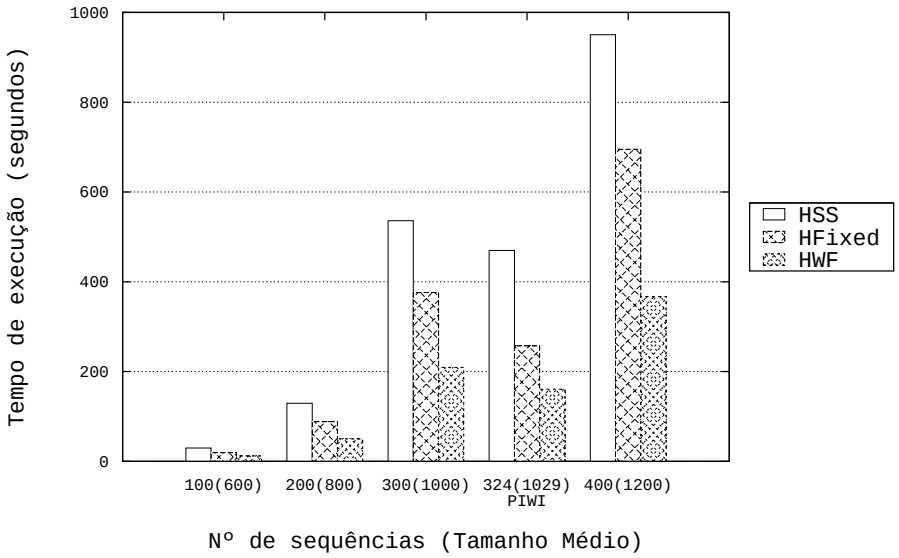

Figura 6.7: Tempos de execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node3 (4 *cores*). Escravos: node1 (8 *cores*) e node2 (16 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os tempos de execução foram obtidos para cada algoritmo de alocação de tarefas implementado: *Hybrid Self-Scheduling* (HSS), *Hybrid Fixed* e *Hybrid Weight Factoring* (HWF).

Esse procedimento também foi adotado na construção das tabelas e gráficos de *speedup* para os casos seguintes onde o node1 e o node2 atuam como mestre (seções 6.3.2 e 6.3.3).

DIALIGN-TX com MPI/OpenMP	*Speedup*		
# Sequências (Tamanho Médio)	HSS	HFixed	HWF
100(600)	1,00	1,54	2,48
200(800)	1,00	1,46	2,60
300(1000)	1,00	1,42	2,56
324(1029) - PIWI	1,00	1,83	2,92
400(1200)	1,00	1,37	2,59

Tabela 6.6: *Speedup* obtido na execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node3 (4 *cores*). Escravos: node1 (8 *cores*) e node2 (16 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os valores indicados foram obtidos calculando-se a razão entre os tempos de execução para cada algoritmo de alocação implementado, considerando o *Hybrid Self-Scheduling* (HSS) como referência.

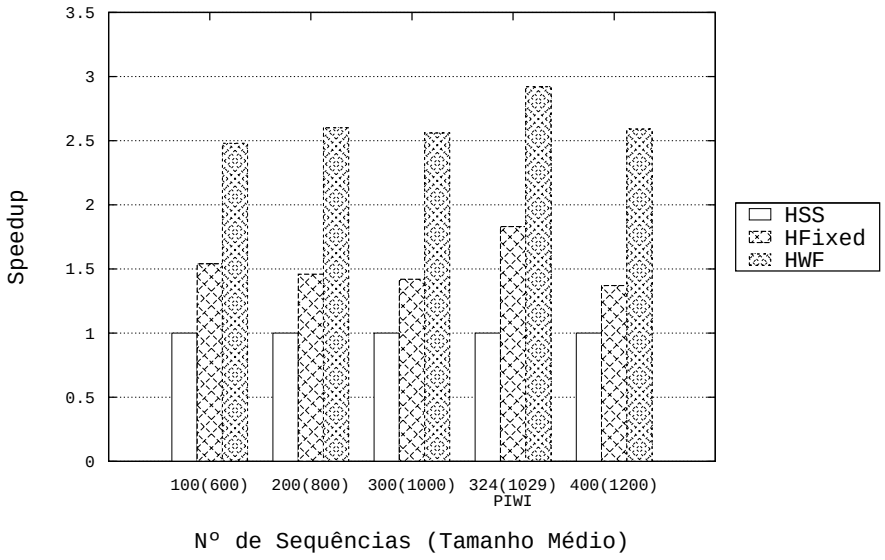

Figura 6.8: *Speedup* obtido na execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node3 (4 *cores*). Escravos: node1 (8 *cores*) e node2 (16 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os valores indicados foram obtidos calculando-se a razão entre os tempos de execução para cada algoritmo de alocação implementado, considerando o *Hybrid Self-Scheduling* (HSS) como referência.

Considerando-se a tabela 6.6 e o gráfico da figura 6.8, observa-se que a estratégia HWF obteve os maiores valores de *speedup*, variando de 2,48 2,92, mesmo aumentando-se a carga do sistema. Este comportamento era esperado, pois a abordagem HWF é uma adaptação de uma política de alocação de tarefas voltada para *clusters heterogêneos* (HF), conforme visto na seção 5.4.

6.3.2 Caso 2: node1 atuando como nó mestre

A seguir, o node1 foi definido como mestre e os mesmos conjuntos de entrada do primeiro caso foram submetidos como entrada para o cálculo da fase 1 do AMS no DIALIGN-TX com MPI/OpenMP.

Os tempos da tabela 6.7 e da figura 6.9 mostram que, quando o node1 for usado como mestre neste *cluster multicore* heterogêneo, a fase 1 do AMS será calculada num tempo de execução menor do que quando o node3 for adotado como mestre, considerando-se que a mesma estratégia de alocação de tarefas (seção 5.4) seja empregada nos dois casos.

Os *speedups* calculados para este caso estão representados nos dados da tabela 6.8 e no gráfico da figura 6.10. A distribuição de tarefas utilizando a estratégia HWF obteve os melhores *speedups* em relação às outras dispo-

DIALIGN-TX com MPI/OpenMP	Tempos de execução (segundos)		
# Sequências (Tamanho Médio)	HSS	HFixed	HWF
100(600)	25,63	15,74	10,02
200(800)	114,38	73,28	42,17
300(1000)	412,01	271,03	153,94
324(1029) - PIWI	339,54	182,29	108,39
400(1200)	803,55	564,38	297,64

Tabela 6.7: Tempos de execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node1 (8 *cores*). Escravos: node2 (16 *cores*) e node3 (4 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os tempos de execução foram obtidos para cada algoritmo de alocação de tarefas implementado: *Hybrid Self-Scheduling* (HSS), *Hybrid Fixed* (HFixed) e *Hybrid Weight Factoring* (HWF).

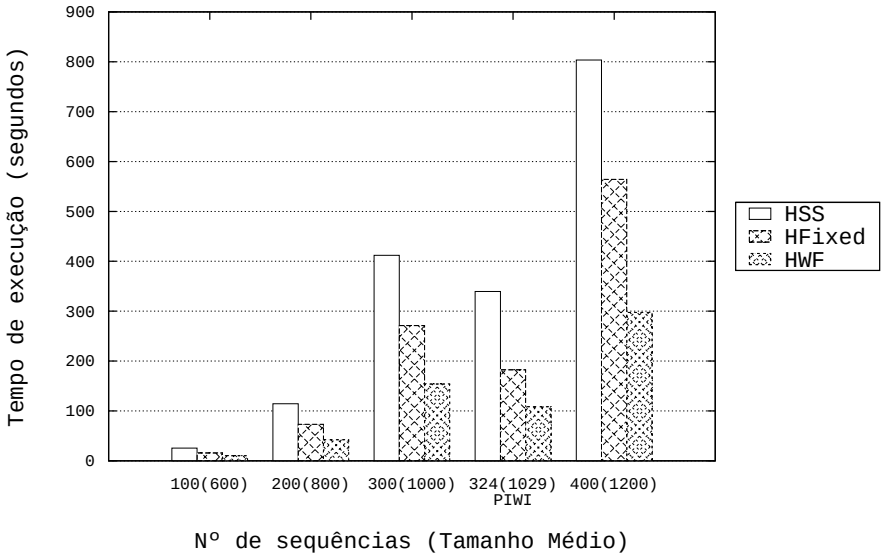

Figura 6.9: Tempos de execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node1 (8 *cores*). Escravos: node2 (16 *cores*) e node3 (4 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os tempos de execução foram obtidos para cada algoritmo de alocação de tarefas implementado: *Hybrid Self-Scheduling* (HSS), *Hybrid Fixed* (HFixed) e *Hybrid Weight Factoring* (HWF).

níveis. Entre as estratégias HFixed e HSS, a escolha da primeira resultou em *speedups* maiores que a segunda, o que significa que os custos de comunicação da estratégia HSS afetam o desempenho do DIALIGN-TX com MPI/OpenMP quando esta é aplicada na distribuição de tarefas.

Observe-se na figura 6.10 que o emprego do node1 como mestre, tendo

clock mais rápido e maior quantidade de *cores* que o node3, influenciou o desempenho da solução, gerando *speedups* maiores para cada estratégia de alocação adotada.

DIALIGN-TX com MPI/OpenMP	*Speedup*		
# Sequências (Tamanho Médio)	HSS	HFixed	HWF
100(600)	1,00	1,63	2,56
200(800)	1,00	1,56	2,71
300(1000)	1,00	1,52	2,68
324(1029) - PIWI	1,00	1,86	3,13
400(1200)	1,00	1,42	2,70

Tabela 6.8: *Speedup* obtido na execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node1 (8 *cores*). Escravos: node2 (16 *cores*) e node3 (4 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os valores indicados foram obtidos calculando-se a razão entre os tempos de execução para cada algoritmo de alocação implementado, considerando o *Hybrid Self-Scheduling* (HSS) como referência.

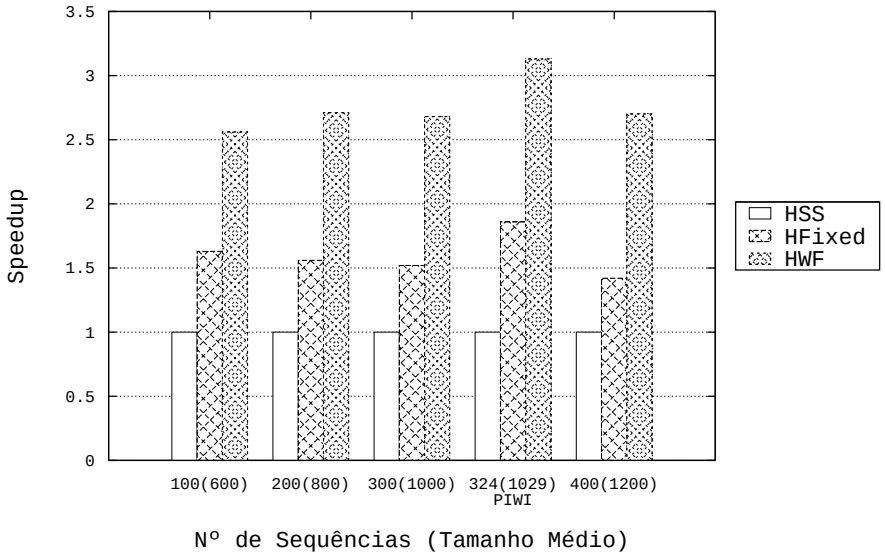

Figura 6.10: *Speedup* obtido na execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node1 (8 *cores*). Escravos: node2 (16 *cores*) e node3 (4 *cores*). A carga é representada pela quantidade de sequências de entrada. Os valores indicados foram obtidos calculando-se a razão entre os tempos de execução para cada algoritmo de alocação de tarefas implementado, considerando o *Hybrid Self-Scheduling* (HSS) como referência.

6.3.3 Caso 3: node2 atuando como nó mestre

No último caso, o node2 foi configurado para atuar como nó mestre do *cluster multicore* heterogêneo. Mesmo composto por 16 *cores* e uma arquitetura de 64 *bits*, o *clock* de `1.6 GHz` contribuiu, como aconteceu na seção 6.2.1, para que este caso apresentasse os piores tempos de execução na tabela 6.9 e no gráfico da figura 6.11).

Por outro lado, os tempos de execução da tabela 6.9 mostram que entre as três estratégias de alocação de tarefas empregadas nos testes deste caso, a HWF continuou obtendo os melhores resultados. Em relação aos casos anteriores (seções 6.3.1 e 6.3.2), as três estratégias apresentaram resultados piores, se comparadas as mesmas abordagens em cada caso.

DIALIGN-TX com MPI/OpenMP	Tempos de execução (segundos)		
# Sequências (Tamanho Médio)	HSS	HFixed	HWF
100(600)	31,24	21,75	13,34
200(800)	154,28	109,24	67,15
300(1000)	601,36	452,39	257,94
324(1029) - PIWI	553,39	352,19	217,22
400(1200)	988,26	771,11	433,08

Tabela 6.9: Tempos de execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node2 (16 *cores*). Escravos: node1 (8 *cores*) e node3 (4 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os tempos de execução foram obtidos para cada algoritmo de alocação de tarefas implementado: *Hybrid Self-Scheduling* (HSS), *Hybrid Fixed* (HFixed) e *Hybrid Weight Factoring* (HWF).

Para o *speedup*, a tabela 6.10 e o gráfico da figura 6.12 confirmam que node2 atuando como nó mestre não foi a melhor opção para execução distribuída do DIALIGN-TX com MPI/OpenMP neste *cluster multicore* heterogêneo. Ainda assim, os *speedups* alcançados com as estratégias HWF e HFixed também foram superiores aos da estratégia HSS para o node2 como mestre. Porém, acompanhando os tempos de execução obtidos, as três abordagens conseguiram *speedups* inferiores aqueles alcançados quando os nós node3 e node1 atuavam como nó mestre.

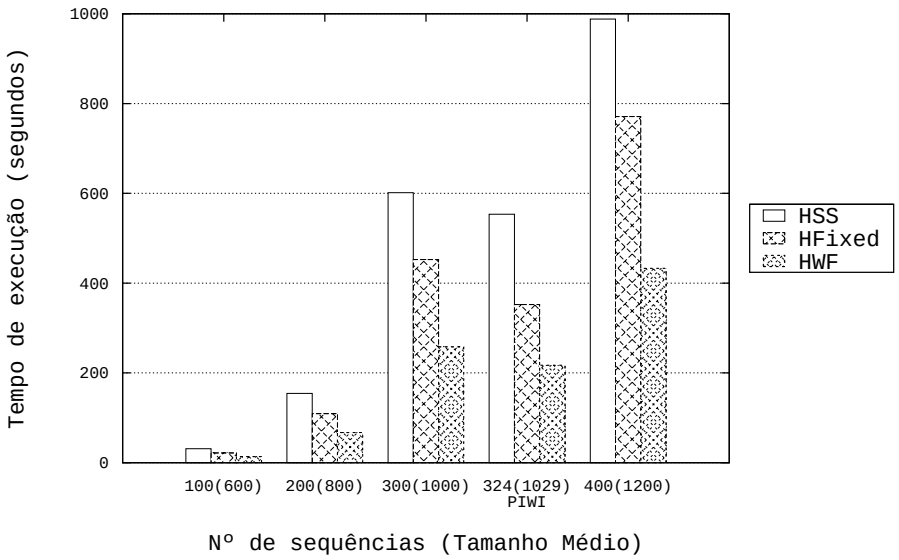

Figura 6.11: Tempos de execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node2 (16 *cores*). Escravos: node1 (8 *cores*) e node3 (4 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os tempos de execução foram obtidos para cada algoritmo de alocação de tarefas implementado: *Hybrid Self-Scheduling* (HSS), *Hybrid Fixed* (HFixed) e *Hybrid Weight Factoring* (HWF).

DIALIGN-TX com MPI/OpenMP	*Speedup*		
# Sequências (Tamanho Médio)	HSS	HFixed	HWF
100(600)	1,00	1,44	2,34
200(800)	1,00	1,41	2,30
300(1000)	1,00	1,33	2,33
324(1029) - PIWI	1,00	1,57	2,55
400(1200)	1,00	1,28	2,28

Tabela 6.10: *Speedup* obtido na execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node2 (16 *cores*). Escravos: node1 (8 *cores*) e node3 (4 *cores*). Total: 28 *cores*. A carga é representada pelo número e tamanho das sequências de entrada. Os valores indicados foram obtidos calculando-se a razão entre os tempos de execução para cada algoritmo de alocação implementado, considerando o *Hybrid Self-Scheduling* (HSS) como referência.

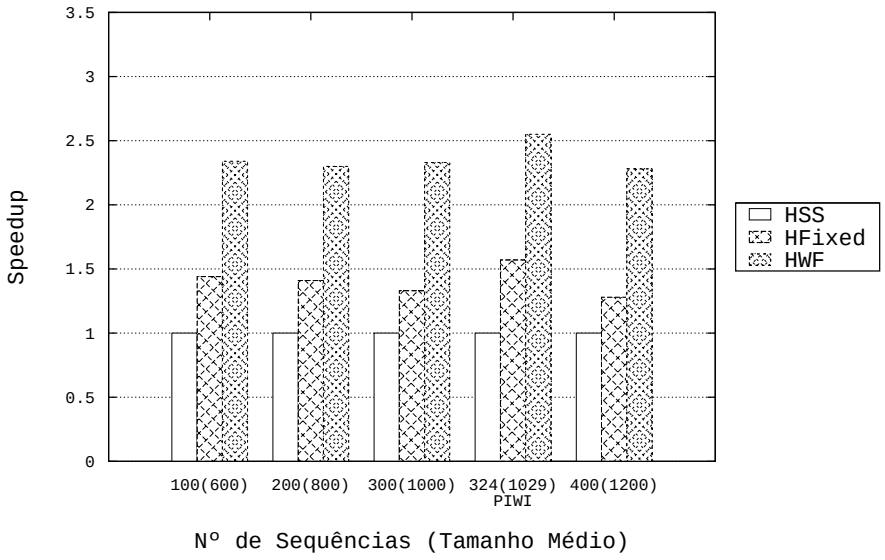

Figura 6.12: *Speedup* obtido na execução distribuída do DIALIGN-TX com MPI/OpenMP. Fase 1 do alinhamento múltiplo de sequências. Mestre: node2 (16 *cores*). Escravos: node1 (8 *cores*) e node3 (4 *cores*). A carga é representada pela quantidade de sequências de entrada. Os valores indicados foram obtidos calculando-se a razão entre os tempos de execução para cada algoritmo de alocação de tarefas implementado, considerando o *Hybrid Self-Scheduling* (HSS) como referência.

O baixo desempenho desta configuração do node2 como mestre era esperado a partir do momento em que se verificou que o *clock* dos *cores* do node2 era inferior àqueles dos nós node1 e node3. Conforme visto na seção 5.3, o nó mestre executa mais etapas da solução distribuída que os outros nós escravos, por exemplo, a geração e distribuição de tarefas, a coleta de resultados, além de consultas a lista de tarefas, arquivos de configuração e políticas de alocação. E em acréscimo, o nó mestre ainda participa do cálculo das comparações par a par da fase 1 do AMS.

Assim, a partir dos resultados mostrados nesta seção sobre a execução distribuída do DIALIGN-TX com MPI/OpenMP, entende-se que, além da estratégia de alocação de tarefas, a escolha de um dos nós do *cluster multicore* heterogêneo para atuar como mestre tem impacto no desempenho final da solução. Não se deve somente levar em consideração o número de *cores* existentes no nó, mas, também, suas velocidades representadas nos valores de seus *clocks*. Isso se deve ao fato de que o papel desempenhado pelo nó mestre durante a execução distribuída do DIALIGN-TX com MPI/OpenMP demanda uma quantidade maior de processamento sequencial do que aquele desempenhado pelos nós escravos, como visto na seção 5.5.

6.4 Análise dos resultados

6.4.1 Resultados da solução com MPI/OpenMP

Quando um conjunto de entrada contendo 300 sequências biológicas de tamanho médio de 1000 resíduos (caracteres) foi submetido à versão original do DIALIGN-TX, o melhor tempo de execução serial para computar a fase 1 do AMS foi obtido pelo node3 com 1 hora, 16 minutos e 36 segundos. Em contra-partida, a execução distribuída do DIALIGN-TX com MPI/OpenMP obteve um tempo de execução de 2 minutos e 34 segundos para calcular a fase 1 do AMS, tendo o node1 como mestre e executando a estratégia de alocação de tarefas HWF para o mesmo conjunto de entrada.

Aumentando-se o tamanho do conjunto de entrada para 400 sequências biológicas de tamanho médio de 1200 resíduos, o node3 obteve o melhor resultado executando a versão original DIALIGN-TX em apenas um *core*, com 2 horas, 38 minutos e 14 segundos. E, novamente, a execução distribuída do DIALIGN-TX com MPI/OpenMP conseguiu reduzir o tempo de execução para 4 minutos e 58 segundos, definindo o node1 como mestre e aplicando a estratégia de alocação de tarefas HWF.

A escolha da estratégia de alocação de tarefas também influenciou na redução do tempo de execução do DIALIGN-TX. Por exemplo, a execução do DIALIGN-TX original no node3 obteve o melhor tempo de execução serial entre os três nós, com 1 hora, 13 minutos e 56 segundos para calcular a fase 1 do AMS para um conjunto de entrada com 324 sequências de tamanho médio de 1029 resíduos. Usando o mesmo conjunto de entrada e definindo-se a estratégia HSS de alocação de tarefas para a execução distribuída do DIALIGN-TX com MPI/OpenMP, o node1 como mestre obteve um tempo de execução de 5 minutos e 40 segundos. Porém, mudando-se a estratégia de alocação para HWF, o node1 atuando como mestre obteve um tempo de execução de 1 minuto e 48 segundos para calcular a fase 1 do AMS no DIALIGN-TX com MPI/OpenMP.

6.4.2 Resultados da solução *standalone* com OpenMP

A execução *standalone* do DIALIGN-TX com OpenMP mostrou considerável redução do tempo de execução, explorando a quantidade de *cores* disponíveis no computador.

Como exemplo, considere-se o caso do conjunto de entrada contendo 400 sequências biológicas de tamanho médio de 1200 resíduos. Conforme visto na seção 6.2.1, executando-se a versão original do DIALIGN-TX no node3, utilizando apenas um de seus *cores* de 3.6 GHz, foram necessárias 2 horas, 57 minutos e 4 segundos para calcular todas as comparações par-a-par da fase 1 do AMS. Porém, executando-se o DIALIGN-TX com OpenMP e criando-se 4 *threads*, uma para cada um de seus *cores*, reduziu-se este tempo de execução para 1 hora, 1 minuto e 56 segundos (seção 6.2.2).

Com um número de *cores* maior do que o node3, o node1 conseguiu re-

duzir ainda mais seu tempo de execução. Enquanto o tempo de execução da fase 1 do AMS no DIALIGN-TX original, rodando em um de seus *cores* de 3.2 GHz, foi de 2 horas, 57 minutos e 4 segundos para o mesmo conjunto de entrada, a execução do DIALIGN-TX com OpenMP levou 34 minutos e 34,3 segundos para realizar o mesmo cálculo da fase 1 do AMS, criando-se 8 *threads*, uma para cada um dos *cores* disponíveis (seção 6.2.2).

No último caso, foi-se observada a maior redução do tempo execução entre a versão original do DIALIGN-TX e a versão com OpenMP. O node2 possui 16 *cores* de *clocks* de 1.6 GHz. Assim, utilizando-se apenas um desses *cores*, a fase 1 do AMS no DIALIGN-TX original levou 5 horas, 9 minutos e 5 segundos para ser calculada, sendo o mesmo conjunto de entrada de 400 sequencias biológicas e tamanho médio de 1200 resíduos. Porém, quando todos os 16 *cores* foram utilizados, executando-se o DIALIGN-TX com OpenMP e empregando 16 threads, seu tempo de execução caiu para 39 minutos e 33,06 segundos.

Esta grande redução do tempo de execução obtida no caso do node2, bem como aquelas obtidas nos casos anteriores do node1 e node3, são evidências de que é possível obter desempenhos melhores no cálculo do AMS com o DIALIGN-TX, mesmo para computadores pessoais, desde que estes disponham de processadores *multicore* e seja empregada uma versão do DIALIGN-TX com OpenMP para explorar tais recursos.

Capítulo 7

Conclusão e Trabalhos Futuros

A presente dissertação de mestrado propôs e avaliou uma Estratégia Distribuída Híbrida em *Cluster Multicore* Heterogêneo para Alinhamento Múltiplo de Sequências Biológicas (AMS) com o DIALIGN-TX. A estratégia proposta foi concebida utilizando-se um ambiente computacional composto por um *cluster multicore* heterogêneo com nós de capacidades de processamento e velocidades de *clock* diferentes.

Adotou-se um modelo híbrido de programação com trocas de mensagens, para a comunicação entre os nós, e memória compartilhada, para a comunicação entre os *cores* dentro de um mesmo nó.

Para determinar o poder computacional do *cluster multicore* heterogêneo, implementou-se um módulo chamado *profiler* para efetuar a troca de mensagens entre os nós e obter os dados de configuração que foram utilizados na execução da solução distribuída.

Os resultados obtidos com um *cluster multicore* heterogêneo, composto por 3 nós de capacidades computacionais distintas e totalizando 28 *cores*, mostraram que o tempo de execução da fase 1 do DIALIGN-TX foi bastante reduzido com a solução distribuída híbrida proposta, variando-se tanto o tamanho do conjunto de entrada quanto a estratégia de alocação de tarefas.

Neste contexto, podemos dizer que a solução proposta inova ao apresentar, como contribuição principal, uma abordagem para o cálculo do AMS com DIALIGN-TX utilizando um modelo híbrido de programação em *cluster multicore* heterogêneo, pois a nosso conhecimento não existe solução paralela ou distribuída para o DIALIGN-TX.

Uma contribuição secundária é apresentada ao identificar-se, durante a implementação da estratégia distribuída híbrida, uma solução para a execução do DIALIGN-TX com memória compartilhada em computadores com processadores *multicore*, a qual foi implementada utilizando-se a biblioteca OpenMP. Os resultados obtidos (seções 6.2.2 e 6.4.2) mostraram que esta contribuição também é capaz de reduzir bastante o tempo de execução da fase 1 do AMS do DIALIGN-TX, explorando-se a quantidade de *cores* disponíveis. Por exemplo, é possível aplicá-la numa estação de trabalho equipada com processadores *multicore* e usada para alinhamento múltiplo de sequências em uma bancada de laboratório de Bioinformática. Também, neste

caso, a nosso conhecimento, não existe ainda uma solução do DIALIGN-TX que empregue memória compartilhada com OpenMP.

A última contribuição desta solução são as três novas estratégias de alocação de tarefas que foram implementadas, chamadas de *Hybrid Fixed* (HFixed), *Hybrid Self-Scheduling* (HSS) e *Hybrid Weight Factoring* (HWF). Estas estratégias foram baseadas nas políticas de alocação *Fixed* [72], *Self-Scheduling* [78] e *Weight Factoring* [42], adaptadas com um modelo híbrido de programação e executadas para alocação de tarefas em *cluster multicore* heterogêneo durante o alinhamento múltiplo de sequências biológicas com o DIALIGN-TX. Os resultados obtidos no seção 6.3 mostraram que a escolha da política de alocação de tarefas adequada é de fundamental importância para o desempenho da solução híbrida principal. A nosso conhecimento, não existe uma solução distribuída do DIALIGN-TX que empregue estratégias híbridas semelhantes.

Assim, o conjunto de contribuições apresentado nesta dissertação fornece recursos suficientes para que se possa reduzir de forma considerável o tempo de execução da fase 1 do cálculo do alinhamento múltiplo de sequências do DIALIGN-TX, aplicando um modelo de programação híbrido e disponibilizando novas estratégias híbridas de alocação de tarefas para uma solução distribuída com o DIALIGN-TX em *cluster multicore* heterogêneo.

Como trabalhos futuros, sugere-se:

- Estender o modelo híbrido de programação da estratégia distribuída proposta para que seja compatível o modelo *Overlapping communication with computation* apresentado em [68]. Assim, seria possível avaliar uma solução distribuída híbrida com trocas de mensagens entre *threads* para evitar que os *cores* fiquem ociosos durante a comunicação entre os nós do *cluster multicore* heterogêneo;

- Adaptação de outra ferramenta bem conhecida de alinhamento múltiplo de sequências, como o ClustalW, para a estratégia distribuída híbrida proposto;

- Adaptação e implementação de outras políticas de alocação de tarefas para o modelo proposto, como *Trapezoidal Self-Scheduling* (TSS) [79], *Guided Self-Scheduling* (GSS) [67], *Fixed-Size Chunking* (FSC) e *Adaptive Weighted Factoring* (AWF) [5];

- Adaptação da solução proposta para uma arquitetura heterogênea *multicore* com GPU (*Graphics Processing Unit*);

- Estender a contribuição secundária desta dissertação, adaptando-se as fases 2 e 3 do alinhamento múltiplo de sequências do DIALIGN-TX para o modelo com memória compartilha utilizando-se OpenMP;

- Adaptação da solução proposta para estudar seu comportamento em diferentes redes de interconexão;

- Adaptação da solução proposta para utilizar novos paradigmas de sistemas de arquivos distribuídos;
- Adaptação da solução proposta para uma abordagem utilizando o modelo de programação paralela PGAS [7].

Referências

[1] S. Abdeddaïm and B. Morgenstern. Speeding up the dialign multiple alignment program by using the 'greedy alignment of biological sequences library' (gabios-lib). In *JOBIM '00: First International Conference on Computational Biology, Biology, Informatics, and Mathematics*, volume 2006 of *Lecture Notes in Computer Science*, pages 1–11, 2001.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, oct 1990.

[3] Argonne National Laboratory. MPICH, set 2010. ftp://ftp.mcs.anl.gov/pub/mpi/mpich-1.2.7p1.tar.gz.

[4] Argonne National Laboratory. MPICH2 Home Page, set 2010. http://www.mcs.anl.gov/research/projects/mpich2.

[5] I. Banicescu and V. Velusamy. Performance of scheduling scientific applications with adaptive weighted factoring. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, pages 84–84, Los Alamitos, CA, apr 2001. IEEE Computer Society.

[6] J. Bedell, I. Korf, and M. Yandell. *BLAST*. O'Reilly, jul 2003.

[7] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick. Hybrid PGAS Runtime Support for Multicore Nodes. In *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS10)*, New York, Oct 2010.

[8] A. Boukerche, J. M. Correa, A. C. M. A. de Melo, and R. P. Jacobi. A hardware accelerator for the fast retrieval of dialign biological sequence alignments in linear space. *IEEE Trans. Computers*, 59(6):808–821, 2010.

[9] M. Brudno, C. B. Do, G. M. Cooper, M. F. Kim, E. Davydov, N. C. S. Program, E. D. Green, A. Sidow, and S. Batzoglou. LAGAN and Multi-LAGAN: Efficient tools for large-scale multiple alignment of genomic dna. *Genome Research*, 13:721–731, 2003.

[10] M. Brudno and B. Morgenstern. Fast and sensitive alignment of large genomic sequences. *Computational Systems Bioinformatics Conference, International IEEE Computer Society*, 0:138, 2002.

[11] M. Brudno, M. C. nad Berthold Göttegens, S. Batzoglou, and B. Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, 4:66, 2003.

[12] M. Brudno, R. Steinkamp, and B. Morgenstern. The CHAOS/DIALIGN WWW server for multiple alignment of genomic sequences. *Nucleic Acids Research*, 32(Web-Server-Issue):41–44, 2004.

[13] T. W. Burger. Intel Multi–Core processors: Quick reference guide, aug 2005.

[14] J. F. R. Campus. Pfam: Family: Piwi, set 2010. http://pfam.janelia.org/family?acc=PF02171#tabview=tab2.

[15] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, oct 1988.

[16] L. Chai, Q. Gao, and D. K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *CCGRID*, pages 471–478. IEEE Computer Society, 2007.

[17] K. Chaichoompu and S. Kittitornkun. Multithreaded clustaiw with improved optimization for intel multi-core processor. In *ISCIT*, pages 590–594. IEEE, sep 2006.

[18] K. Chaichoompu, S. Kittitornkun, and S. Tongsima. MT-clustalW: multithreading multiple sequence alignment. In *IPDPS*, page 8pp. IEEE, apr 2006.

[19] P. Chain, S. Kurtz, E. Ohlebusch, and T. Slezak. An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges. *Briefings in Bioinformatics*, 4(2):105–123, 2003.

[20] S. Dong and G. E. Karniadakis. Dual-level parallelism for deterministic and stochastic CFD problems. In *SC2002: Proceedings of the IEEE ACM SC 2002 Conference, November 16–22, 2002, Baltimore, MD, USA*. IEEE Computer Society Press, 2002.

[21] J. J. Dongarra, H. W. Meuer, and E. Strohmaier. TOP500 supercomputer sites, 1999. Último acesso em 01/09/2010.

[22] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-openMP parallelization models on SMP clusters. In *IPDPS*. IEEE Computer Society, 2004.

[23] EMBL-EBI. FASTA similiraty search, jun 2009. http://www.ebi.ac.uk/Tools/fasta33/index.html.

[24] D.-F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J. Mol. Evol.*, 25:351–360, 1987.

[25] R. Finn, J. Mistry, J. Tate, P. Coggill, A. Heger, J. Pollington, O. Gavin, P. Gunesekaran, G. Ceric, K. Forslund, L. Holm, E. Sonnhammer, S. Eddy, and A. Bateman. The pfam protein families database. *Nucleic Acids Research*, 38, set 2010.

[26] J. P. Fitch, S. N. Gardner, T. A. Kuczmarski, S. Kurtz, R. Myers, L. L. Ott, T. R. Slezak, E. A. Vitalis, A. T. Zemla, and P. M. McCready. Rapid development of nucleic acid diagnostics. In *Proceedings of the IEEE*, volume 90, pages 1708–1721, nov 2002.

[27] W. M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155(760):279–284, jan 1967.

[28] S. Flynn Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Comm. of the ACM*, 35(8):90, aug 1992.

[29] M. P. I. Forum. MPI: A message-passing interface standard. *Techical report CS-94-230*, 1994.

[30] M. P. I. Forum. MPI-2: Extensions to the message-passing interface. *Techical report*, 1997.

[31] GCC team. GCC Home page, set 2010. http://gcc.gnu.org/news.html.

[32] C. Gibas and P. Jambeck. *Developing Bioinformatics Computer Skills*. O'Reilly, apr 2001.

[33] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[34] O. Gotoh. Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments. *J. Mol. Biol.*, 264:823–838, 1996.

[35] B. Göttgens, L. M. Barton, M. A. Chapman, A. M. Sinclair, B. Knudsen, D. Grafham, J. G. Gilbert, J. Rogers, D. R. Bentley, and A. R. Grenn. Transcriptional regulation of the stem cell leukemia gene (SCL) - comparative analysis of five vertebrate SCL analysis of vertebrate SCL loci. *Genomic Research*, 12:749–759, 2002.

[36] B. Göttgens, L. M. Barton, J. G. Gilbert, A. J. Bench, M.-J. Sanchez, S. Bahn, S. Mistry, D. Grafham, A. McMurray, M. Vaudin, E. Amaya, D. R. Bentley, and A. R. Green. Analysis of vertebrate SCL loci identifies conserved enhancers. *Nature Biotechnology*, 18:181–186, 2000.

[37] D. Gusfield. Algorithms on strings, trees, and sequences: Computer science and computational biology. *SIGACT News*, 28(4):41–60, 1997.

[38] Y. He and C. H. Q. Ding. MPI and openMP paradigms on cluster of SMP architectures: the vacancy tracking algorithm for multi-dimensional array transposition. In *SC'2002 Conference CD*, Baltimore, MD, nov 2002. IEEE/ACM SIGARCH.

[39] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In ACM, editor, *SC2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000*, pages 50–50, pub-ACM:adr and pub-IEEE:adr, 2000. ACM Press and IEEE Computer Society Press.

[40] D. G. Higgins and P. M. Sharp. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73:237–244, 1988.

[41] D. G. Higgins, J. D. Thompson, and T. J. Gibson. ClustalW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix. *Nucleic Acids Research*, 22(22):4673–4680, jan 1994.

[42] S. F. Hummel, J. P. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *SPAA*, pages 318–328, 1996.

[43] J. Kim and S. Sinha. Towards realistic benchmarks for multiple alignments of non-coding sequences. *BMC Bioinformatics*, 11:54, 2010.

[44] K.-B. Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[45] D. J. Lipman, S. F. Altschul, and J. D. Keceioglu. A tool for multiple sequence alignment. In *Proc. Natl. Acad. Sci. USA*, volume 86, pages 4412–4415, jun 1989.

[46] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Sci.*, 227:1435–1441, 1985.

[47] D. J. Lipman and W. R. Pearson. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci.*, volume 85, pages 24444–24448, 1988.

[48] R. D. Loft, S. J. Thomas, and J. M. Dennis. Terascale spectral element dynamical core for atmospheric general circulation models. In *SC2001: High Performance Networking and Computing. Denver, CO, November 10–16, 2001*. ACM Press and IEEE Computer Society Press, 2001.

[49] W. Miller. Comparison of genomic DNA sequences: Solved and unsolved problems. *BIOINF: Bioinformatics*, 17, 2001.

[50] M. M. Monwar and S. Rezaei. An efficient parallel processing approach for multiple biological sequence alignment. *IAENG International Journal of Computer Science*, 33(2):32–36, 2007.

[51] B. Morgenstern. DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15(3):211–218, 1999.

[52] B. Morgenstern. A space-efficient algorithm for aligning large genomic sequences. *Bioinformatics*, 16(10):948–949, 2000.

[53] B. Morgenstern. A simple and space-efficient fragment-chaining algorithm for alignment of dna and protein sequences. *Applied Mathematics Letters*, 15(1):11–16, 2002.

[54] B. Morgenstern and S. Abdeddaim. DIALIGN Download page, set 2010. http://bibiserv.techfak.uni-bielefeld.de/download/tools/DIALIGN_221.html.

[55] B. Morgenstern, W. R. Atchley, K. Hahn, and A. Dress. Segment-based scores for pairwise and multiple sequence alignments. In J. Glasgow, T. Littlejohn, F. Major, R. Lathrop, D. Sankoff, and C. Sensen, editors, *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology, ISMB'98 (Montréal, Canada, June 28 - July 1, 1998)*, pages 115–121, Menlo Park, 1998. AAAI Press.

[56] B. Morgenstern, A. Dress, and T. Werner. Multiple dna and protein sequence alignment based on segment-to-segment comparison. In *Proceedings of the Nation Academy of Science*, volume 93, pages 12098–12103, USA, October 1996.

[57] B. Morgenstern, K. Frech, A. Dress, and T. Werner. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, 1998.

[58] B. Morgenstern, O. Rinner, S. Abdeddaïm, D. Haase, K. F. X. Mayer, A. W. M. Dress, and H.-W. Mewes. Exon discovery by genomic sequence alignment. *Bioinformatics*, 18(6):777–787, 2002.

[59] D. W. Mount. *Bioninformatics — Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, Cold Spring Harbor, New York, sep 2004.

[60] H. Muhammad. htop – an interactive process-viewer for linux, set 2010. http://htop.sourceforge.net/index.php?page=main.

[61] NCBI. NCBI/BLAST home, jun 2009. http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[62] S. B. Needleman and C. D. Wunsch. A general method applicable to the search of similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[63] C. Notredame. Recent progresses in multiple sequence alingment: a survey. *Pharmacogenomics*, 3:131–144, 2002.

[64] C. Notredame, D. G. Higgins, and J. Heringa. T-coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of molecular biology*, 302(1):205–217, sep 2000.

[65] W. R. Pearson. FASTA sequence comparison at the u. of virginia, jun 2009. http://fasta.bioch.virginia.edu/fasta_www2/fasta_list2.shtml.

[66] D. A. Pollard, C. M. Bergman, J. Stoye, S. E. Celniker, and M. B. Eisen. Benchmarking tools for the alignment of functional noncoding DNA. *BMC Bioinformatics*, 5, 2004.

[67] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Computers*, 36(12):1425–1439, 1987.

[68] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/openMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 2009*, pages 427–436. IEEE Computer Society, 2009.

[69] M. Schmollinger, K. Nieselt, M. Kaufmann, and B. Morgenstern. DIALIGN P: Fast pair-wise and multiple sequence alignment using parallel processors. *BMC Bioinformatics*, 5:128, 2004.

[70] F. Schreiber, G. Wörheide, and B. Morgenstern. Orthoselect: a web server for selecting orthologous gene alignments from EST sequences. *Nucleic Acids Research*, 37(Web-Server-Issue):185–188, 2009.

[71] J. C. Setubal and J. Meidanis. *Introduction to Computional Molecular Biology*. PWS Publishing Company, 1997.

[72] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, University of California, 2001. Chair-Berman, Francine.

[73] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[74] A. R. Subramanian, M. Kaufmann, and B. Morgenstern. DIALIGN-TX: greedy and progressive approaches for segment-based multiple sequence alignment. *Algorithms for Molecular Biology*, 3:6, 2008.

[75] A. R. Subramanian, M. Kaufmann, and B. Morgenstern. DIALIGN-TX: download, set 2010. http://dialign-tx.gobics.de/download.

[76] A. R. Subramanian, J. Weyer-Menkhoff, M. Kaufmann, and B. Morgenstern. DIALIGN-T: An improved algorithm for segment-based multiple sequence alignment. *BMC Bioinformatics*, 6:66, 2005.

[77] B. E. Suzek, H. Huang, P. B. McGarvey, R. Mazumder, and C. H. Wu. Uniref: comprehensive and non-redundant uniprot reference clusters. *Bioinformatics*, 23(10):1282–1288, 2007. Web site: http://www.uniprot.org/help/uniref.

[78] P. Tang and P.-C. Yew. Processor self-scheduling for multiple-nested parallel loops. *1986 ICPP*, pages 528–535, August, 1986.

[79] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Trans. Parallel Distrib. Syst*, 4(1):87–98, 1993.

[80] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[81] C.-C. Wu, L.-T. Huang, L. F. Lai, and M.-L. Chen. Enhanced parallel loop self-scheduling for heterogeneous multi-core cluster systems. In *ISPAN*, pages 568–573. IEEE Computer Society, 2009.

[82] C.-C. Wu, L. F. Lai, and P.-H. Chiu. Parallel loop self-scheduling for heterogeneous cluster systems with multi-core computers. In *APSCC*, pages 251–256. IEEE, 2008.

[83] Yap, Frieder, and Martino. Parallel computation in biological sequence analysis. *IEEETPDS: IEEE Transactions on Parallel and Distributed Systems*, 9, 1998.

[84] J. Zola, X. Yang, A. Rospondek, and S. Aluru. Parallel T-Coffee: A parallel multiple sequence aligner. In *Proceedings of the ISCA on Parallel and Distributed Computing Systems-PDCS*, pages 248–253, sep 2007.