UNIVERSIDADE ESTADUAL DE SANTA CRUZ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL EM CIÊNCIA E TECNOLOGIA

JULIO OLIVEIRA DA SILVA

SOLUÇÃO DE ALTO DESEMPENHO PARA RECONSTRUÇÃO DE ÁRVORES FILOGENÉTICAS USANDO O MÉTODO DE MÁXIMA VEROSSIMILHANÇA

PPGMC - UESC

ILHÉUS - BA
2016

JULIO OLIVEIRA DA SILVA

SOLUÇÃO DE ALTO DESEMPENHO PARA RECONSTRUÇÃO DE ÁRVORES FILOGENÉTICAS USANDO O MÉTODO DE MÁXIMA VEROSSIMILHANÇA

PPGMC - UESC

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional em Ciência e Tecnologia da Universidade Estadual de Santa Cruz como parte das exigências para a obtenção do Grau de Mestre em Modelagem Computacional em Ciência e Tecnologia.

Área de concentração: Bioinformática

Orientadora: Prof.ª Dra. Martha Ximena Torres Delgado

Coorientador: Prof. Dr. Esbel Tomás Valero Orellana

ILHÉUS - BA

2016

S586 Silva, Júlio Oliveira.
Solução de alto desempenho para reconstrução de árvores filogenéticas usando o método de máxima verossimilhança / Júlio Oliveira da Silva. – Ilhéus, BA: UESC, 2016.
xxi, 156f. : Il.

Orientadora: Martha Ximena Torres Delgado.
Dissertação (Mestrado) – Universidade Estadual de Santa Cruz. Programa de Pós-Graduação em Modelagem Computacional em Ciência e Tecnologia.
Inclui referências e apêndice.

1. Filogenia – Processamento de dados. 2. Computação de alto desempenho. 3. OpenMP (Interface de programas aplicativos). I. Título.

CDD 581.38

JULIO OLIVEIRA DA SILVA

SOLUÇÃO DE ALTO DESEMPENHO PARA RECONSTRUÇÃO DE ÁRVORES FILOGENÉTICAS USANDO O MÉTODO DE MÁXIMA VEROSSIMILHANÇA

PPGMC - UESC

Ilhéus - BA, 07 de Março de 2016

Comissão Examinadora

Prof.ª Dra. Martha Ximena Torres Delgado
UESC
(Orientadora)

Prof. Dr. Esbel Tomás Valero Orellana
UESC
(Co-orientador)

Prof. Dr. Angelo Amâncio Duarte
UEFS

Prof. Dr. Aprigio Augusto Lopes Bezerra
UESC

DEDICATÓRIA

À minha mãe Marinalva Oliveira, meu pai José Batista (in memoriam), toda a minha família por todo apoio, dedicação e compreensão e a todos que dedicam suas vidas ao progresso científico.

AGRADECIMENTOS

Ao Departamento de Ciências Exatas e Tecnológicas (DCET) da Universidade de Santa Cruz e todos os pesquisadores vinculados e, à FAPESB pelo apoio e oportunidade da realização do curso.

À professora e orientadora Martha Ximena, pela magnífica orientação, sapiência e paciência.

Ao professor e co-orientador Esbel Tomás por toda a colaboração e apoio.

À todos os professores do PPGMC-UESC pela excelência em capacitação e suporte aos possíveis futuros colegas de profissão. Aos funcionários do PPGMC pelo auxílio e suporte.

À minha mãe, pela dedicação e apoio, a quem devo minha formação profissional e sociocultural.

Aos amigos, familiares e colegas de curso pelo incentivo e apoio psicológico nos momentos difíceis.

Aos colegas e amigos do NBCGIB, em especial à Eduardo Almeida Costa e Carlos Magno Moreira Sales por todo suporte e ajuda.

PENSAMENTO

“A imaginação é mais poderosa que o conhecimento. Ela amplia a visão dilata a mente, desafia o impossível. Sem a imaginação o pensamento estagna.”

Albert Einstein (1875-1955)

SOLUÇÃO DE ALTO DESEMPENHO PARA RECONSTRUÇÃO DE ÁRVORES FILOGENÉTICAS USANDO O MÉTODO DE MÁXIMA VEROSSIMILHANÇA

RESUMO

Atualmente, a diversidade biológica é entendida como o resultado do processo de evolução das espécies. Desta forma, os seres vivos são entidades que se transformam ao longo de gerações sob influência do meio. O objetivo da Reconstrução de Árvores Filogenéticas (RAF) é hipoteticamente inferir a relação correta entre espécies contemporâneas através de dados de sequências (aminoácidos, nucleotídeos, códons, etc) de membros representativos de cada espécie (eleitos através do alinhamento de sequências) e, através disso, explicar a história evolutiva das espécies e/ou grupo de espécies. A RAF auxilia no desenvolvimento de vacinas, no estudo da biodiversidade, na produção de novas drogas para propósitos agrícolas e médicos, entre outros fins. Apesar da existência de muitos programas usados na RAF, a grande dificuldade está na limitação do poder de processamento para encontrar uma correlação filogenética em tempo hábil e prático, isto é, no menor tempo de processamento possível. Este trabalho objetiva melhorar o desempenho das implementações encontradas em um dos principais programas *open source* de RAF que usa o método de máxima verossimilhança, o PhyML. No intuito de melhor explorar o poder de processamento existente e permitir a RAF com maior número de espécies em menor tempo de processamento, através de estudos e identificação de gargalos no código do PhyML, propôs-se uma solução de alto desempenho com a implementação da técnica de processamento paralelo com o paradigma de memória compartilhada OpenMP (*Open Multi-Processing*), através da qual foi possível obter *speedup* de até 9 vezes tanto para sequências de nucleotídeos quanto para sequências de proteínas e eficiência de 60% a 80%.

Palavras-chave: Reconstrução de Árvores Filogenéticas. Método de máxima verossimilhança. Modelo evolutivo. Modelagem computacional. Computação de alto desempenho. OpenMP.

SOLUTION OF HIGH PERFORMANCE FOR THE RECONSTRUCTION OF PHYLOGENETIC TREES USING THE METHOD OF MAXIMUM LIKELIHOOD

ABSTRACT

Currently, biodiversity is understood as the result of the process of evolution of the species. Thus, living beings are entities that turn over generations under the influence of the medium. The purpose of Reconstruction of Phylogenetic Trees (RPT) is hypothetically infer the correct relationship between contemporary species through sequence data (amino acids, nucleotides, codons, and so on) of representative members of each species (elected by the sequence alignment) and thereby explain the evolutionary history of the species and/or group of species. The RPT supports in the development of vaccines, in the biodiversity studies, in the production of new drugs for medical and agricultural purposes, between other purposes. Although there are many programs used in the RPT, the great difficulty lies in limiting the processing power to find a phylogenetic correlation in a timely and practical, i.e., in the shortest possible processing time. This work aims to improve the performance of implementations found in one of the major open source programs RPT using the method of maximum likelihood, the PhyML. In order to better exploit the power of existing processing and allow the RPT with the highest number of species in reduced processing time, through studies and identification of bottlenecks in PhyML code, it was proposed a high-performance solutions to the implementation of technical with parallel processing shared memory paradigm OpenMP (*Open Multi-Processing*), through which it was possible to obtain speedup of up to 9 times for both sequences of nucleotides and sequences of proteins and efficiency of 60% to 80%.

Keywords: Reconstruction of Phylogenetic Trees. Maximum likelihood method. Evolutionary model. Computational modeling. High-performance computing. OpenMP.

LISTA DE TABELAS

Tabela 1 - Comparação de parâmetros dos modelos evolutivos de nucleotídeos. ...22

Tabela 2 - Dados de DNA utilizados como arquivos de entrada nas simulações do PhyML. 52

Tabela 3 - Tempo médio de execução de cada função gargalo medido com a estrutura timespec para arquivos de sequências de DNA. 63

Tabela 4 - Tempo médio de execução de cada função gargalo medido com a estrutura timespec para arquivos de sequências de proteínas. 65

LISTA DE FIGURAS

Figura 1 - Exemplo descritivo das características e nomenclaturas de uma árvore filogenética no contexto da filogenia. 8

Figura 2 - Representação de árvore filogenética sem raiz (a) e árvore filogenética com raiz (b). 9

Figura 3 - Uma operação NNI: cada ramo interno na árvore se liga a quatro subárvores ou vizinhos mais próximos (a, b, c, d). A troca de uma subárvore de um lado do galho interno com uma do outro lado constitui uma modificação topológica com NNI, nessa figura, têm-se duas operações NNI. 11

Figura 4 - Uma operação SPR: a subárvore representada pelo nó a é removida, e, em seguida, reinserida em um local diferente na subárvore restante. 11

Figura 5 - TBR: a árvore é dividida em duas subárvores com o corte de um galho interno. Um galho de uma das subárvores é então escolhido como a melhor inserção (análise das possíveis inserções) e reinserido novamente na subárvore restante. 12

Figura 6 - Matriz de alinhamento de sequências de DNA, na qual as linhas representam as espécies e as colunas representam os sítios (base nucleotídica, nesse caso, para determinada posição homóloga entre as espécies).....14

Figura 7 - Ilustração de várias substituições na mesma sequência, em que uma sequência ancestral de DNA divergiu em duas outras, as quais de forma independente acumularam substituições de nucleotídeos ao longo da linhagem, porém, apenas duas linhagens diferentes são visíveis.....15

Figura 8 - Árvore usada para demonstrar o calculo da função de verossimilhança. Em seu estágio final, possui três espécies u, w e s, com duas espécies ancestrais v e r e comprimentos de galhos trv, trs, tvu e tvw. 28

Figura 9 - Diagrama da análise de bootstrap: cada réplica leva a inferência de uma árvore e possibilita o calculo de bootstrap nos clados da árvore inicial para conferir sua confiabilidade.....36

Figura 10 - árvore modelo para o calculo de MV realizado pelo PhyML. 37

Figura 11 - Arquiteturas paralelas: (a) arquitetura de memória compartilhada e (b) arquitetura de memória distribuída.....44

Figura 12 - Modelo Fork-Join suportado pelo OpenMP.....46

Figura 13 - Variabilidade da quantidade de chamadas da função LK_Core sobre 20 simulações do PhyML com o mesmo arquivo de sequências e mesma configuração de parâmetros (por questões de melhor visibilidade da ilustração, os valores de quantidade de chamadas foram ordenados progressivamente). 50

Figura 14 - Gráfico de chamadas gerado pelo GProf para o programa PhyML com arquivo de entrada contendo sequências de DNA. 55

Figura 15 - Plano de perfil gerado pelo GProf para o programa PhyML com arquivo de entrada contendo sequências de DNA.....56

Figura 16 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando pequenos arquivos de sequências de DNA. 58

Figura 17 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando grandes arquivos de sequências de DNA. ...58

Figura 18 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando pequenos arquivos de sequências de DNA. 59

Figura 19 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando grandes arquivos de sequências de DNA.....59

Figura 20 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando pequenos arquivos de sequências de proteínas. 60

Figura 21 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando grandes arquivos de sequências de proteínas. 61

Figura 22 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando pequenos arquivos de sequências de proteínas. 61

Figura 23 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando pequenos arquivos de sequências de proteínas. 62

Figura 24 - Tempo cumulativo para pequenos arquivos de sequências de DNA. 64

Figura 25 – Tempo cumulativo para grandes arquivos de sequências de DNA. 64

Figura 26 - Tempo cumulativo para pequenos arquivos de sequências de proteínas. 65

Figura 27 - Tempo cumulativo para grandes arquivos de sequências de proteínas. 66

Figura 28 - Trecho da implementação do EPCC Microbenchmark usado para medir o overhead da diretiva “pragma omp for” em que (a) é a versão serial do benchmark e (b) é a versão paralela em relação à versão serial. 68

Figura 29 - Overheads ilustrativo de algumas diretivas OpenMP: o overhead das diretivas comuns e construtores é dado em microssegundos à media que se aumenta o número de threads. 70

Figura 30 - Overhead das principais diretivas previsíveis para o bloco de trabalho da função LK_Core com tempo médio de execução igual a 100 e 500 microssegundos para sequências de DNA. 73

Figura 31 - Overhead da principal diretiva previsível para o bloco de trabalho da função Pull_Scaling_Factors com tempo médio de execução variando entre 500 e 800 nanosegundos para sequências de DNA. 74

Figura 32 - Overhead da principal diretiva previsível para o bloco de trabalho da função Rate_Correction com tempo médio de excução variando entre 50 e 60 nanosegundos para sequências de DNA. 74

Figura 33 - Overhead das principais diretivas previsíveis para o bloco de trabalho da função LK_Core com tempo médio de execução com concentração em 110, 500 e 600 microssegundos para sequências de proteínas. 75

Figura 34 - Overhead da principal diretiva previsível para o bloco de trabalho da função Update_P_LK_AA com tempo médio de execução com concentração em 150, 500 e 600 microssegundos para sequências de proteínas. 75

Figura 35 - Overhead da principal diretiva previsível para o bloco de trabalho da função Pull_Scaling_Factors com tempo médio de execução variando entre 480 e 660 nanosegundos para sequências de proteínas. 76

Figura 36 - Overhead da principal diretiva previsível para o bloco de trabalho da função Rate_Correction com tempo médio de execução variando entre 50 e 68 nanosegundos para sequências de proteínas. 76

Figura 37 - Overhead da principal diretiva previsível para o bloco de trabalho da função PMat_Empirical com tempo médio de execução variando entre 12 e 15 microssegundos para sequências de proteínas. 77

Figura 38 - As três configurações de alternativas topológicas sobre um galho interno. n1, n2, n5 e n6 são quatro subárvores, e b1, b2, b4 e b5 são os quatro galhos conectados a raiz de n1, n2, n5 e n6, respectivamente. Estes tamanhos são os mesmos nas três configurações topológicas (a, b e c). U e V são subárvores da esquerda e direita, respectivamente, dos galhos internos, e, x, y e z são os tamanhos de galhos internos que maximizam a verossimilhança da filogenia correspondente. 78

Figura 39 - Algoritmo genérico do programa PhyML. “Bloco X” é um desvio para um código descrito fora do fluxo. As funções gargalos estão identificadas após o termo em negrito “Gargalo” ou “Gargalos”, permitindo identificar onde as funções gargalos são usadas dentro do algoritmo. 81

Figura 40 - Vetor de nós da topologia e nós vizinhos diretos de cada nó, usado para simular a estrutura topológica em árvore através da vetorização dos nós da árvore. 84

Figura 41 - Tabelas de bipartição usada pelo PhyML para encontrar nós folhas saindo em duas direções de cada vez. 85

Figura 42 - Galho b indica que a MV parcial deve ser calculada na subárvore do nó $\boldsymbol{n}$ em função dos nós $n1$ e $n2$. 90

Figura 43 - Ilustração da MV parcial calculada pela função *Update_P_LK_Nucl* através de chamadas por estratégias de pós-ordem, raiz com MV parcial atualizada para toda base de toda categoria e sítio (números meramente ilustrativos). 91

Figura 44 - Speedup da análise de desempenho da função LK_Core com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.7, 5.5 e 7.2. 105

Figura 45 - Eficiência da análise de desempenho da função LK_Core com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA.

Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 96%, 93.4%, 91.8% e 89.6%.....105

Figura 46 - Speedup da análise de desempenho da função LK_Core com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 2.0, 3.8, 5.5 e 7.1. 106

Figura 47 - Eficiência da análise de desempenho da função LK_Core com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 97.5%, 94.2%, 91.5% e 89.3%.....106

Figura 48 - Speedup da análise de desempenho da função Update_P_LK_Nucl com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.8, 3.4, 5.0 e 6.4. 107

Figura 49 - Eficiência da análise de desempenho da função Update_P_LK_Nucl com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 89.7%, 85.7%, 84% e 79.8%.....107

Figura 50 - Speedup da análise de desempenho da função Update_P_LK_Nucl com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.4, 5.0 e 6.2. 108

Figura 51 - Eficiência da análise de desempenho da função Update_P_LK_Nucl com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 92.8%, 86.1%, 83.1% e 77%. 108

Figura 52 - Speedup da análise de desempenho da função Find_Mutual_Direction com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.3, 2.1, 2.8 e 3.3. 109

Figura 53 - Eficiência da análise de desempenho da função Find_Mutual_Direction com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 63.7%, 52.6%, 46.3% e 40.8%. 109

Figura 54 - Speedup da análise de desempenho da função Find_Mutual_Direction com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.3, 2.3, 3.3 e 4.3. 110

Figura 55 - Eficiência da análise de desempenho da função Find_Mutual_Direction com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 66.9%, 57.2%, 54.6% e 53.4%. 110

Figura 56 - Speedup da análise de desempenho da função LK_Core com relação ao tempo médio de execução com pequenos arquivos de sequências de

proteínas. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.5, 5.0 e 6.4. 111

Figura 57 - Eficiência da análise de desempenho da função LK_Core com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 94.4%, 86.9%, 82.9% e 79.6%. 112

Figura 58 - Speedup da análise de desempenho da função LK_Core com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 2.0, 3.9, 6.3 e 8.4. 113

Figura 59 - Eficiência da análise de desempenho da função LK_Core com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 97.8%, 97.9%, 105.4% e 105.6%. 113

Figura 60 - Speedup da análise de desempenho da função Update_P_LK_AA com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.5, 5.1 e 6.5. 114

Figura 61 - Eficiência da análise de desempenho da função Update_P_LK_AA com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 92.9%, 86.5%, 84.3% e 81.2%. 114

Figura 62 - Speedup da análise de desempenho da função Update_P_LK_AA com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.6, 5.5 e 7.2. 115

Figura 63 - Eficiência da análise de desempenho da função Update_P_LK_AA com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 93.0%, 90.9%, 91.0% e 89.9%. 115

Figura 64 - Speedup da análise de desempenho da função PMat_Empirical com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.1, 1.1, 1.0 e 1.0. 116

Figura 65 - Eficiência da análise de desempenho da função PMat_Empirical com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 52.7%, 27.7%, 17.4% e 12.2%. 116

Figura 66 - Speedup da análise de desempenho da função PMat_Empirical com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 0.7, 1.1, 1.2 e 1.1. 117

Figura 67 - Eficiência da análise de desempenho da função PMat_Empirical com relação ao tempo médio de execução com grandes arquivos de sequências

de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 34.0%, 27.6%, 19.4% e 14.0%. 117

Figura 68 - Análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando até 8 threads. 119

Figura 69 - Speedup da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando até 8 threads. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.5, 5.0 e 6.3. 120

Figura 70 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 92.9%, 87.9%, 84.1% e 78.7%. 120

Figura 71 - Análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando 10 e 12 threads. 121

Figura 72 - Speedup da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando 10 e 12 threads. Linearidade: média de speedup com 10 e 12 threads respectivamente, 7.5 e 8.3. 121

Figura 73 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA com 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 74.5% e 69.4%. 122

Figura 74 - Análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando até 8 threads. Os maiores desvios padrão ocorreram com o arquivo 416_4150 sendo de 30% com 02 threads, seguido pelo arquivo 408_9563 com 19% para a versão serial. 123

Figura 75 - Speedup da análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando até 8 threads. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.8, 3.2, 4.2 e 5.0. 123

Figura 76 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de DNA com até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 90.2%, 79.9%, 69.9% e 62.1%. 123

Figura 77 - Análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando 10 e 12 threads. O maior desvio padrão ocorreu para versão serial com o arquivo 406_6016 e desvio padrão de 54% seguido pelo arquivo 626_4716 com desvio padrão de 11%. 124

Figura 78 - Speedup da análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando 10 e 12 threads. Linearidade: média de speedup com 10 e 12 threads respectivamente, 5.5 e 6.1. 125

Figura 79 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de DNA com 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 55.4% e 50.4%. 125

Figura 80 - Análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando até 8 threads. O maior desvio padrão ocorreu para a versão serial com o arquivo 50_1000 e desvio padrão de 19%. 126

Figura 81 - Speedup da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando até 8 threads. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.6, 2.7, 3.7 e 4.4. 126

Figura 82 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 88.4%, 82.9%, 79.0% e 75.0%. 126

Figura 83 - Análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando 10 e 12 threads. O Maior desvio padrão ocorreu para a versão serial com o arquivo 50_1000 e desvio padrão de 19%. 127

Figura 84 - Speedup da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando 10 e 12 threads. Linearidade: média de speedup com 10 e 12 threads respectivamente, 1.5 e 1.7. 127

Figura 85 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 18.5% e 14.3%. 128

Figura 86 - Análise de desempenho do PhyML com grandes arquivos de sequências de proteínas usando até 8 threads. Os maiores desvios padrão ocorreram com a versão serial sendo de 17% com o arquivo 77_11234 e de 15% com o arquivo 77_9918. 129

Figura 87 - Speedup da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com até 8 threads. Linearidade: média de speedup com 2, 4, 6 e 8 threads respectivamente, 1.8, 3.3, 4.7 e 6.0. 129

Figura 88 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 77.5%, 66.7%, 62.2% e 54.8%. 129

Figura 89 - Análise de desempenho do PhyML com grandes arquivos de sequências de proteínas usando 10 e 12 threads. Os maiores desvios padrão ocorreram com a versão serial sendo de 20% com o arquivo 77_11234 e de 17% com o arquivo 77_9918. 130

Figura 90 - Speedup da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com 10 e 12 threads. Linearidade: média de speedup com 10 e 12 threads respectivamente, 8.2 e 9.1. 130

Figura 91 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 65.4% e 60.0%. 131

Figura 92 - Estimativa de speedup para pequenos arquivos de sequências de DNA, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 77% à 80%. 133

Figura 93 - Estimativa de speedup para grandes arquivos de sequências de DNA, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 71% à 79%. 133

Figura 94 - Estimativa de speedup para pequenos arquivos de sequências de proteínas, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 71% à 91%. 134

Figura 95 - Estimativa de speedup para grandes arquivos de sequências de DNA, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 90% à 96%. 134

Figura 96 - Análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA. 138

Figura 97 - Speedup sobre a análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA. Linearidade: média de speedup OpenMP 5.6, MPI 5.4, Híbrida (2 processos) 2.4 e Híbrida (4 processos) 3.8. 138

Figura 98 - Eficiência sobre a análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência OpenMP 70.3%, MPI 67.2%, Híbrida (2 processos) 30.6% e Híbrida (4 processos) 47.6%.....139

Figura 99 - Análise de desempenho das versões paralelas do PhyML com grandes arquivos de sequências de DNA. 140

Figura 100 - Speedup sobre a análise de desempenho das versões paralelas do PhyML com grandes arquivos de sequências de DNA. Linearidade: média de speedup OpenMP 4.7, MPI 5.5, Híbrida (2 processos) 3.0 e Híbrida (4 processos) 4.1.....140

Figura 101 - Eficiência sobre a análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência OpenMP 58.8%, MPI 69.2%, Híbrida (2 processos) 37.0% e Híbrida (4 processos) 51.1%.....140

Figura 102 - Comportamento da versão MPI e Híbrida com o aumento do número de réplicas, pequeno arquivo de sequências de DNA (12_3768). 142

Figura 103 - Comportamento da versão MPI e Híbrida com o aumento do número de réplicas, pequeno arquivo de sequências de DNA (42_3768). 142

Figura 104 - Comportamento da versão MPI e Hibrida com o aumento do número de réplicas, grande arquivo de sequências de DNA (346_886). 143

Figura 105 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com pequenos arquivos de sequências de DNA.....144

Figura 106 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com grandes arquivos de sequências de DNA. Os maiores desvios padrão ocorreram com o arquivo 306_7062, sendo de 13% com 2 threads e de 12% para a versão serial. 145

Figura 107 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com pequenos arquivos de sequências de proteínas. Os maiores desvios padrão ocorreram com o arquivo 50_1000, sendo de 17% para a versão serial e 14% com 2 threads. 145

Figura 108 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com grandes arquivos de sequências de proteínas. Os maiores desvios padrão ocorreram com o arquivo 77_9918, sendo de 15% para a versão serial e 14% com 2 threads. 146

Figura 109 - Tempos de execução da versão MPI do PhyML simulada sobre arquitetura de memória distribuída e memória compartilhada com pequenos arquivos de sequências de DNA. 149

Figura 110 - Tempos de execução da versão MPI do PhyML simulada sobre arquitetura de memória distribuída e memória compartilhada com grandes arquivos de sequências de DNA. 149

Figura 111 - Comparação de speedup da versão MPI do PhyML simulada sobre uma arquitetura de memória distribuída e memória compartilhada com pequenos arquivos de sequências de DNA. 150

Figura 112 - Comparação de speedup da versão MPI do PhyML simulada sobre arquitetura de memória distribuída e memória compartilhada com grandes arquivos de sequências de DNA. 150

LISTA DE ABREVIATURAS E SIGLAS

BFGS	Broyden-Fletcher-Goldfarb-Shanno
DNA	DeoxyriboNucleic Acid
GProf	GNU Profiling
GTR	General Time Reversible
JC69	Jukes e Cantor 1969
MV	Máxima Verossimilhança
NJ	Neighbor Joining
NNI	Nearest-Neighbour Interchange
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
OTUs	Operational Taxonomic Units
RAF	Reconstrução de Árvores Filogenéticas
RaxML	Randomized Axelerated Maximum Likelihood
RNA	RiboNucleic Acid
SPR	Subtree Pruning and Regrafting
TBR	Tree Bisection and Reconnection
UPGMA	Unweighted Pair Grouping Method with Arithmetic Means

SUMÁRIO

RESUMO VII

ABSTRACT VIII

1 INTRODUÇÃO 1

1.1 Motivação 3

1.2 Objetivos 5

2 REVISÃO DE LITERATURA 7

2.1 Árvores Filogenéticas 7

2.1.1 Espaço de árvores e estratégias de busca no espaço 9

2.1.2 Representação de dados filogenéticos 12

2.1.3 Processos de Markov 14

2.1.4 Modelos Evolutivos 19

2.1.4.1 Jukes-Cantor e outros modelos evolutivos de DNA e proteínas 20

2.1.5 Taxas evolutivas de heterogeneidade 24

2.1.5.1 Distribuição discreta para taxas de mutação 25

2.1.5.2 Distribuição gama para taxas de mutação 26

2.1.6 Método de Máxima Verossimilhança 28

2.1.7 Medidas de incerteza com *bootstrap* 34

2.1.8 Heurística adotada pelo PhyML de acordo com a literatura 36

2.1.9 Versões de implementações do PhyML 40

2.1.9.1 Softwares alternativos que implementam a Máxima Verossimilhança 41

2.2 Paradigmas de paralelização 43

2.2.1 Paradigma de memória distribuída MPI 45

2.2.2 Paradigma de memória compartilhada OpenMP 46

3 METODOLOGIA 47

3.1 Ferramentas utilizadas 48

3.2 Etapas básicas do processo de paralelização do PhyML 49

3.3 Critérios usados na detecção de gargalos e análise de desempenho 49

4 DESENVOLVIMENTO 51

4.1 Sobre os dados de sequências usados nas simulações 51

4.2 Detecção de funções gargalos 53

4.2.1 Sobre a ferramenta de perfilamento GProf 53

4.2.2 Funções gargalos do PhyML 56

4.3 Viabilidade de paralização de funções gargalos 67

4.3.1 *Overhead* das diretivas OpenMP com a ferramenta EPCC Microbenchmark 67

4.3.2 Viabilidade de paralização das funções gargalos do PhyML 70

4.4 Heurísticas e estratégias de paralelização das funções gargalos 77
4.4.1 Heurística e localização dos pontos críticos do PhyML 78
4.4.2 Descrição das funções gargalos 82
4.4.2.1 Funções *Pull_Scaling_Factors* e *Rate_Correction* 82
4.4.2.2 Função *Find_Mutual_Direction* 83
4.4.2.3 Função *PMat_Empirical* 88
4.4.2.4 Função *Update_P_LK_Nucl* 89
4.4.2.5 Função *Udate_P_LK_AA* 93
4.4.2.6 Função *LK_Core* 94
4.4.3 Desenvolvimento dos códigos em paralelo 97
4.4.3.1 Paralelização da função *Find_Mutual_Direction* 97
4.4.3.2 Paralelização da função *PMat_Empirical* 98
4.4.3.3 Paralelização da função *Update_P_LK_Nucl* 99
4.4.3.4 Paralelização da função *Update_P_LK_AA* 100
4.4.3.5 Paralelização da função *LK_Core* 100
4.4.3.6 Paralelização das funções *Pull_Scaling_Factors* e *Rate_Correction* 101

5 RESULTADOS E DISCUSSÕES 102
5.1 Análise de desempenho da versão paralelizada com OpenMP 104
5.1.1 Análise de desempenho das funções gargalos do PhyML 104
5.1.2 Análise de desempenho do tempo total de execução do PhyML 118
5.2 Comparação entre versões paralelizadas 136
5.3 Comparação PhyML e PhyML com a biblioteca beagle (serial) 144

6 CONCLUSÕES E TRABALHOS FUTUROS 146

APÊNDICE A - Comparação da versão MPI do PhyML simulada sobre uma arquitetura de memória distribuída e memória compartilhada 149

REFERÊNCIAS 151

1 INTRODUÇÃO

A inferência filogenética é o processo que determina as relações evolutivas de um conjunto de espécies, usando sequências genéticas como dados de entrada e apresentando graficamente como resultado tais relações em forma de árvores filogenéticas. Essas árvores fornecem informações que auxiliam a desvendar os possíveis relacionamentos entre as espécies atuais e supor as histórias evolutivas das mesmas (SOUZA NETO, 2015, p. 1). Ainda de acordo com Souza Neto (2015, p. 1), uma vez que as informações das espécies extintas são insuficientes para produzir árvores com total grau de certeza sobre a reconstrução, deve-se considerar cada árvore filogenética como uma possibilidade hipotética.

As árvores filogenéticas são compostas de folhas, galhos e nós internos. De acordo com Yang (2006, p. 73) e Souza Neto (2015, p. 1), folhas são sequências genéticas que representam as espécies, isto é, cada folha na topologia é uma espécie. Galhos são as interligações entre os nós da árvore e representam o tempo de evolução entre eles. Já nós internos correspondem aos ancestrais hipotéticos. Dado um conjunto de sequências filogenéticas, encontrar uma árvore que melhor representa uma filogenia é um problema bastante complexo, pois além da computação exaustiva, o número de árvores a serem avaliadas cresce muito rapidamente à medida que se aumenta a quantidade de espécies em estudo (FELSENSTEIN, 2004, p. 19-32; YANG, 2006, p. 75-77).

De acordo com Felsenstein (2004), os principais métodos usados para Reconstrução de Árvores Filogenéticas (RAF) são: Distância, Máxima Parcimônia, Máxima Verossimilhança e Inferência Bayesiana. Cada um desses métodos possui suas técnicas e peculiaridades que explora a seu favor as hipóteses sobre o processo de evolução, como heurísticas próprias ou algoritmos estatísticos, para produzir a filogenia desejada (SOUZA NETO, 2015, p. 1). É importante ressaltar que a RAF, além da heurística de algum desses métodos, exige a utilização de sequências genéticas devidamente alinhadas e um modelo evolutivo de substituição de nucleotídeos ou aminoácidos. Os métodos citados serão descritos a seguir:

a) métodos de distância: segundo Souza Neto (2015, p. 1), esses métodos foram pioneiros no processo de RAF e ainda são muito utilizados devido à sua performance, embora já existam métodos mais exatos. Estes atuam

sobre uma matriz de distâncias (construída baseada em modelos evolutivos) aplicando os respectivos algoritmos para obter a melhor hipótese da filogenia. Ainda de acordo com Yang (2006, p. 81) e Souza Neto (2015, p. 1), dentre os métodos de distâncias mais conhecidos e utilizados, destacam-se: UPGMA (do inglês, *Unweighted Pair Grouping Method with Arithmetic Means*) (SNEATH; SOKAL, 1973) o NJ (*Neighbor Joining*) (SAITOU; NEI, 1987), o BioNJ (GASCUEL, 1997), o Weighbor (do inglês, *Weighted Neighbor Joining*) (BRUNO; SOCCI; HALPERN, 2000) e o FastMe (do inglês, *Fast Minimum Evolution*) (DESPER; GASCUEL, 2002);

b) método de máxima parcimônia: segundo Yang (2006, p. 82) e Souza Neto (2015, p. 2), este método busca uma árvore que traduz o menor número possível de mudanças ocorridas nos caracteres das sequências genéticas submetidas, e que, desta forma, ao se atribuir pesos na mudança de um caractere para outro, a hipótese da árvore mais parcimoniosa será aquela cujo somatório das mudanças informar o menor valor. Dentre os softwares que utilizam o método da máxima parcimônia, pode-se citar: DNAPARS, PROTPARS, PARS, MIX (FELSENSTEIN, 1993) e TNT (GOLOBOFF; FARRIS; NIXON, 2005);

c) método de máxima verossimilhança (MV): este método busca a árvore que hipoteticamente possui a maior probabilidade de acordo com os dados das sequências filogenéticos. Neste sentido, Souza Neto (2015, p. 2) afirma que as implementações atuais são baseadas em heurísticas que produzem árvores perto das ótimas, pois trabalham com algoritmos busca que auxiliam na exploração do espaço de topologias (quantidade de possíveis árvores a serem exploradas, obtida da análise combinatória em função das possíveis disposições das espécies na árvore). Dentre os algoritmos de busca pode-se citar: troca dos vizinhos mais próximos NNI (do inglês, *Nearest-Neighbour Interchange*), poda e inserção de sub-árvores SPR (do inglês, *Subtree Pruning and Regrafting*) ou bissecção e reconexão de árvores TBR (do inglês, *Tree Bisection and Reconnection*). PhyML (GUINDON et al., 2010; GUINDON; GASCUEL, 2003), FastDNAml (OLSEN et al., 1994), MorePhyML (CRISCUOLO, 2011), DNAML e

PROML (FELSENSTEIN, 1993) são exemplos de softwares que utilizam o método de máxima verossimilhança;

d) método de inferência bayesiana: ainda fundamentado no trabalho de Souza Neto (2015, p. 2), este método usa o suporte estatístico baseado no teorema de Bayes para a validação da árvore inferida. Analisando do ponto de vista computacional, este método utiliza um recurso numérico que é a amostragem de uma densidade de probabilidade através do método de Monte Carlo via Cadeias de Markov (MCMC). Os principais softwares de RAF que utilizam esse método são MrBayes (HUELSENBECK; RONQUIST, 2001) e o BAMBE (do inglês, *Bayesian Analysis in Molecular Biology and Evolution*) (LARGET; SIMON, 1999).

Dentre os métodos de RAF supracitados, o método de MV será amplamente utilizado no escopo desse trabalho por meio do programa PhyML. Apesar de possuir como principal heurística o método de MV, o PhyML faz uso de outros dois métodos de RAF como o método de distância, usado na construção da topologia inicial e o método da parcimônia usado estrategicamente no espaço de busca da árvore hipoteticamente ótima.

1.1 Motivação

Árvores filogenéticas detêm informações úteis para uma grande variedade de questões biológicas e sociais e podem ser usadas como ferramentas de auxílio na tomada de decisões em uma variedade de situações como eficácia de vacinas e drogas agrícolas e transplantes de órgãos, por exemplo. Segundo Amorim (2002), árvores filogenéticas podem auxiliar no controle e combate de parasitas responsáveis por doenças, no estudo epidemiológico, para criação de vacinas mais eficientes, na produção de novas drogas agrícolas e médicas, no estudo do desenvolvimento da biodiversidade, entre outros. Já Souza Neto (2015, p. 5), enaltece a sua importância afirmando que árvores filogenéticas são úteis também para fornecer subsídios importantes para decisões relativas a transplantes de órgãos ou de tecidos de outras espécies, entre outras utilidades (AMORIM, 2002; ZWICKL, 2006).

Segundo Guindon et al. (2005, 2010), a inferência filogenética usando métodos probabilísticos, incluindo MV e abordagem Bayesiana, tem possivelmente fornecido os resultados mais bem sucedidos das ultimas décadas, se mostrando acurados em uma variedade de estudos (GUINDON; GASCUEL, 2003; KUHNER; FELSENSTEIN, 1993; RANWEZ; GASCUEL, 2001). Estes métodos têm se tornando populares e são agora comumente considerados como as melhores abordagens, comparados a outros métodos como Distância e Parcimônia (GUINDON et al., 2005, 2010).

Em contrapartida, apesar dos notáveis avanços, a desvantagem da inferência com o método de MV é que este requer grande esforço computacional, tendendo a ser demasiadamente lento (CHOR; TULLER, 2005; GUINDON et al., 2005, 2010; GUINDON; GASCUEL, 2003; HORDIJK; GASCUEL, 2005). Esse alto custo de processamento esta atrelado principalmente ao espaço de busca da possível topologia ótima, aos exaustivos cálculos de MV vinculados à complexidade dos modelos evolutivos (sítios, categorias e otimização dos parâmetros do modelo), e a estratégia estatística utilizada para certificação da árvore inferida (*bootstrap*). Além disso, o aumento drástico e contínuo do tamanho de dados de sequências (GUINDON et al., 2005, 2010) tem aumentado ainda mais o tempo de processamento na inferência de filogenias usando MV.

Neste sentido, o PhyML, que realiza o processo de RAF através do método de MV, vem sendo aperfeiçoado gradativamente ao longo do tempo, com a adoção e aplicação de estratégias e heurísticas que possibilitem a minimização dos altos custos de processsmanto relacionados às características herdadas.

Com relação à redução de tempos de processamento na busca pela possível topologia ótima, foram implementadas duas heurísticas no PhyML: a NNI (GUINDON; GASCUEL, 2003) e a SPR (HORDIJK; GASCUEL, 2005), em que o PhyML utiliza uma topologia inicial como base e usa essas heurísticas para aplicar rearranjos na topologia, evitando computação desnecessária causada pela busca exaustiva em grande parte do espaço de topologias e, reduzindo consideravelmente o tempo de processamento.

Já no sentido da estratégia para certificação da topologia usando réplicas *bootstrap*, foi adotada a implementação de uma versão do PhyML paralelizada com o paradigma de memória distribuída MPI (do inglês, *Message Passing Interface*),

paralelizando as réplicas de *bootstrap*. Ambas as medidas reduzem o tempo de processamento do PhyML com relação à certificação da topologia (GUINDON et al., 2010).

Desta forma, nota-se que o núcleo dos cálculos de MV e a otimização dos parâmetros do modelo evolutivo são custosos (em termos de processamento) em função da quantidade de espécies e do tamanho das sequências genéticas que as representam e, além disso, tanto a quantidade de vezes a ser processado esse núcleo de cálculos quanto a realização de otimização dos parâmetros é imprevisível, pois ocorrem em função da quantidade de alteração topológicas com NNI ou SPR. Neste sentido, faz-se necessário uma análise sobre o núcleo dos cálculos de MV e o processo de otimização dos parâmetros do modelo evolutivo, com a intenção de implementar uma paralelização do PhyML com o paradigma de memória compartilhada com OpenMP, ainda não existente no PhyML, para, possivelmente, reduzir o tempo de processamento e aproveitar as característica de independência evolutiva sobre sítios e explorar de forma adequada o poder de processamento das modernas arquiteturas multi-cores.

1.2 Objetivos

O principal objetivo desse trabalho é estudar formas de acelerar o desempenho do PhyML 3.0 e, desta forma, fornecer uma versão de alto desempenho do software através da sua paralelização com o paradigma de memória compartilhada usando OpenMP. A intenção é estudar o processo de RAF usando o método de MV e as heurísticas adotadas pelo PhyML para identificar os pontos críticos do software e aplicar a paralelização para otimizar o seu desempenho. Por outro lado, deseja-se realizar a integração dessa versão com o paradigma de memória distribuída MPI, já em uso e, desta forma, oferecer alternativas de uso do PhyML com a versão OpenMP, MPI ou a versão híbrida (OpenMP+MPI), a critério da necessidade do usuário.

Os objetivos supracitados podem ser alcançados através dos objetivos específicos descritos a seguir:

a) fazer a paralelização do PhyML 3.0 com o paradigma de memória compartilhada usando OpenMP;
b) contribuir com a aprendizagem sobre os fundamentos e modelagem matemática do processo de RAF;
c) contribuir com a aprendizagem da modelagem matemática usada pelo método de máxima verossimilhança;
d) contribuir com a aprendizagem sobre o PhyML através da modelagem e heurísticas utilizadas por este software.

Este trabalho está organizado da seguinte forma: a Seção 2 apresenta a revisão de literatura, através de uma visão geral sobre árvores filogenéticas e métodos de RAF, porém enfatizando: o método de máxima verossimilhança e suas estratégias de evolução e o software PhyML e suas implementações, por serem ambos objetos de estudos deste trabalho. A Seção 2 aborda também o paradigma de memória distribuída MPI e o paradigma de memória compartilha OpenMP.

A Seção 3 apresenta a metodologia utilizada, com informações e características sobre: ferramentas usadas para identificar as funções gargalos e medir os respectivos tempos de execução e de apoio usadas para definir a viabilidade ou não de paralelização desses gargalos, bem como estratégias usadas para realizar de forma adequada a análise de desempenho.

Já a Seção 4 apresenta o desenvolvimento com a identificação dos gargalos encontrados no PhyML 3.0 para arquivos de sequências de DNA (do inglês, *Deoxyribonucleic Acid*) e proteínas. A Seção 4 apresenta ainda uma análise de viabilidade de paralelização desses gargalos, bem como uma explicação geral da heurística do PhyML através de um algoritmo com base nas peculiaridades encontradas no código fonte. Além disso, esta Seção aborda a origem e justificativa de escolha dos dados usados nas simulações, bem como explana detalhadamente em que consiste os gargalos do PhyML, apresentando algoritmos para cada um deles e abordando as estratégias usadas para realizar a paralelização dos gargalos, quando viáveis.

A Seção 5 apresenta os resultados e discussões da análise de desempenho com medidas de *speedup* e eficiência para as versões paralelizadas, considerando sequências de DNA e proteínas e uma correlação com as versões do PhyML: serial, com MPI, com OpenMP, com OpenMP e MPI (híbrida) e a versão com a biblioteca

beagle (biblioteca especilaizada em calculos de máxima verossimilhança). Por fim, a Seção 6 apresenta as conclusões e sugestões de propostas para a elaboração de trabalhos futuros.

2 REVISÃO DE LITERATURA

Essa Seção apresenta todas as características necessárias ao processo de inferência filogenética com o método de máxima verossimilhança, conceitos sobre os paradigmas de memória distribuída MPI e memória compartilhada OpenMP.

2.1 Árvores Filogenéticas

As árvores filogenéticas são representações gráficas, estruturadas na forma de uma árvore, que explicam a possível história evolutiva das espécies ou de grupos de espécies correlacionados. Essas árvores são construídas a partir de sequências de caracteres (tais como sequências de DNA, proteínas ou códons) obtidas de diversos organismos. De acordo com Amorim (2002), Hypólito (2005) e Ticona (2008), como a árvore filogenética é uma hipótese sobre o histórico de relações evolutivas, para a construção da árvore, é necessário utilizar caracteres que sejam indicadores confiáveis de ancestralidade comum. Para isso, deve-se utilizar características herdadas de um ancestral comum, podendo ser qualquer uma destas recebidas como herança (SOUZA NETO, 2015, p. 5).

Dada a sua utilidade, a compreensão de árvores filogenéticas exige o entendimento de algumas características e terminologias comumente utilizadas no contexto da filogenia (HYPÓLITO, 2005; TICONA, 2008), sendo estas:

a) nós: são pontos na árvore, que podem ser internos, representando hipoteticamente o ancestral comum, ou terminais, também conhecidos como folhas, que são os organismos estudados e se encontram na extremidade da árvore;

b) ramos/galhos: são linhas que ligam os nós e representam o tempo de evolução entre eles;
c) OTUs (do inglês, *Operational Taxonomic Units*): são espécies/organismos incluídos na análise, também conhecidos como táxons, dos quais se deseja inferir a história filogenética;
d) topologia: uma representação gráfica unindo as OTUs através de ramos e nós.

Essas características e terminologias são ilustradas na Figura 1. Com base nessa ilustração, percebe-se que, de cada nó interno ramificam-se exatamente dois ramos, explicitando que a relação evolutiva entre as espécies é representada por árvores binárias, isto é, no máximo duas ramificações saindo de cada nó interno.

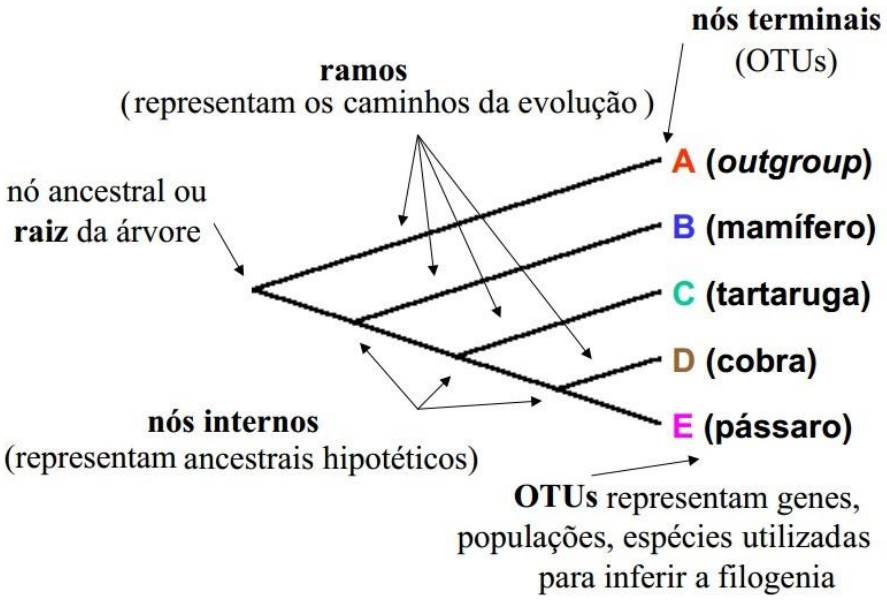

Figura 1 - Exemplo descritivo das características e nomenclaturas de uma árvore filogenética no contexto da filogenia.
Fonte: (MENDES, 2015).

Ainda de acordo com a árvore ilustrada na Figura 1, é relevante salientar e observar uma característica interessante, que todos os táxons são descendentes de um mesmo ancestral, o nó raiz. Neste contexto, cabe ressaltar que existem árvores com e sem raiz. Uma árvore enraizada implica em uma relação de ancestralidade que define uma direção de tempo de evolução a partir um ancestral comum. Por outro lado, existem também as árvores não enraizadas ou árvores sem raiz, em que

não existe relação de ancestralidade comum a todas as espécies. Considerando a, b, c e d OTUs, a Figura 2 (a) ilustra uma árvore sem raiz, enquanto a Figura 2 (b) ilustra uma árvore com raiz.

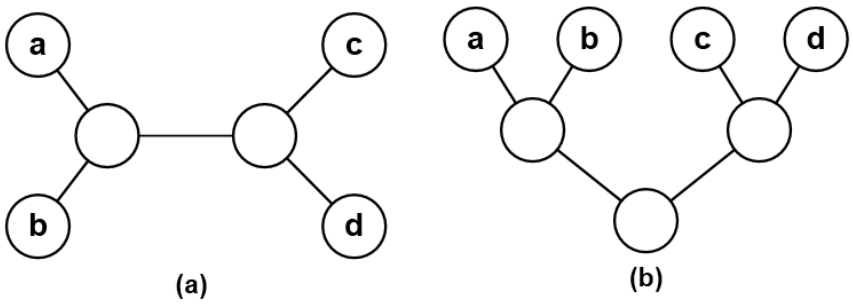

Figura 2 - Representação de árvore filogenética sem raiz (a) e árvore filogenética com raiz (b).

No contexto de árvores sem raiz, Souza Neto (2015, p. 6) afirma que é possível escolher um ponto qualquer da árvore onde se insira um nó raiz, e desta forma, transforma uma árvore sem raiz em uma árvore enraizada e que, a variação da quantidade de possíveis locais de inserção deste nó raiz possibilita gerar várias árvores enraizadas. Desta forma, a subseção seguinte aborda o espaço de possíveis árvores que pode ser gerado a partir dos mesmos dados de sequências e as estratégias usadas na filogenia para evitar a exploração exaustiva de todo o espaço de árvores.

2.1.1 Espaço de árvores e estratégias de busca no espaço

As árvores filogenéticas representam hipóteses da história evolutiva das espécies, porém, a inferência da árvore que se mostra mais adequada aos dados das sequências é uma tarefa difícil e exige alto custo computacional. Além disso, para um determinado conjunto de táxons, existe o que se chama de espaço de árvores, que é o conjunto de possibilidades hipotéticas de histórias evolutivas ou de árvores filogenéticas a ser analisado, o que torna este processo ainda mais complexo.

Neste sentido, de acordo com Hypólito (2005) e Souza Neto (2015, p. 7), dá-se o nome de topologia à árvore que apresenta somente as relações de parentesco entre as espécies, isto é, à forma como os nós internos são conectados uns com os outros e com as folhas, sem considerar os valores de comprimentos de galhos. Desta forma, quanto maior a quantidade de táxons, maior é o número de topologias a serem analisadas.

O número de topologias varia de acordo com o tipo de árvore (com ou sem raiz) e o número de táxons. De acordo com Felsenstein (2004) e Hypólito (2005), Seja X a quantidade de topologias com raiz, Y a quantidade de topologias sem raiz e s a quantidade de táxons, o número de topologias com raiz é obtido através da eq. (1), já o número de topologias sem raiz é obtido através da eq. (2), as quais estão descritas a seguir.

$$X = \frac{(2s-3)!}{2^{(s-2)}(s-2)!} \tag{1}$$

$$Y = \frac{(2s-5)!}{2^{(s-3)}(s-3)!} \tag{2}$$

No contexto da inviabilidade computacional de exploração exaustiva de todo o espaço de árvores (fatorial), principalmente quando a quantidade de espécies envolvidas na inferência é muito grande, de acordo com Souza Neto (2015, p. 38), existem estratégias usadas para evitar a exploração de todo o espaço e, ainda assim, possivelmente encontrar uma solução ótima ou próxima da ótima. Para isso, toma-se uma árvore original como modelo e aplica-se sobre esta, iterativamente, modificações topológicas para encontrar uma árvore considerada ótima. Neste sentido, as estratégias de modificações topológicas NNI, SPR e TBR são abordadas a seguir:

a) NNI: esta técnica troca subárvores vizinhas de pares diferentes de determinado galho interno, modificando a árvore inicial. Todas possíveis trocas de subárvore em relação a esse galho interno são testadas, isto é, as operações são efetuadas até que a melhor solução seja retornada (SOUZA NETO, 2015, p. 38). A ilustração da Figura 3 exemplifica tal técnica;

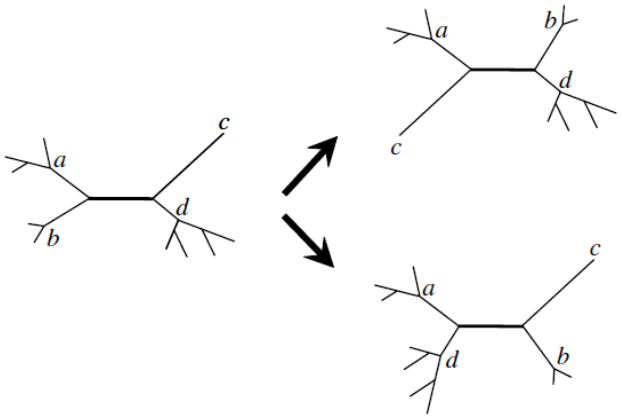

Figura 3 - Uma operação NNI: cada ramo interno na árvore se liga a quatro subárvores ou vizinhos mais próximos (a, b, c, d). A troca de uma subárvore de um lado do galho interno com uma do outro lado constitui uma modificação topológica com NNI, nessa figura, têm-se duas operações NNI.
Fonte: (YANG, 2006, p. 86).

b) SPR: uma subárvore é separada da solução inicial e, posteriormente, é reinserida em todas as possíveis posições da árvore restante. Esse processo é repetido para todas as subárvores da solução inicial até encontrar a melhor inserção SPR. Note que esta técnica permite avaliar um espaço maior de topologias do que a do NNI (SOUZA NETO, 2015, p. 39). A ilustração mostrada na Figura 4 apresenta uma operação com SPR;

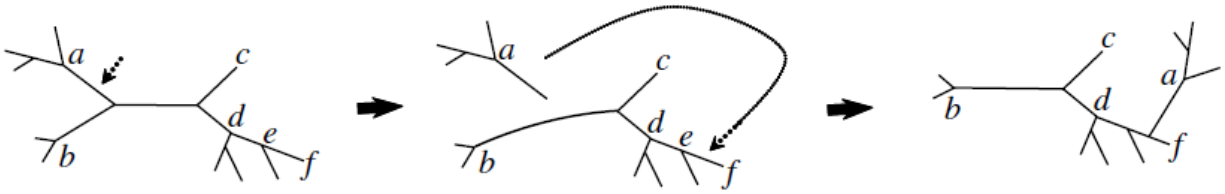

Figura 4 - Uma operação SPR: a subárvore representada pelo nó a é removida, e, em seguida, reinserida em um local diferente na subárvore restante.
Fonte: adaptado de (YANG, 2006, p. 86).

c) TBR: consiste em eliminar um ramo interno da árvore modelo, originando duas subárvores e, posteriormente, reconectar tais subárvores através da criação de um novo ramo, o qual reflete a melhor inserção dentre os galhos da subárvore, para isso, todas as subárvores e todas as

reconexões possíveis são examinadas, retornando a melhor árvore encontrada. O TBR explora um espaço maior de soluções do que o SPR. A Figura 5 a seguir ilustra uma operação com TBR.

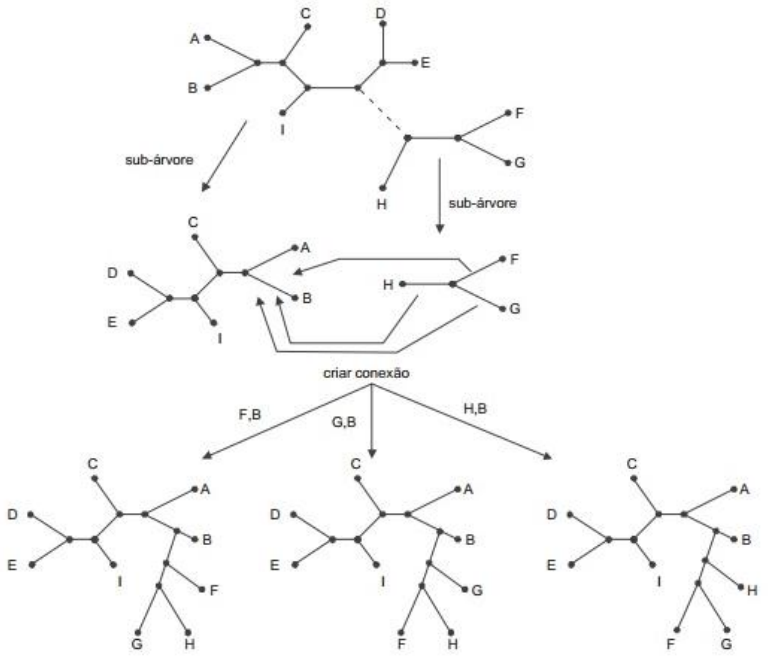

Figura 5 - TBR: a árvore é dividida em duas subárvores com o corte de um galho interno. Um galho de uma das subárvores é então escolhido como a melhor inserção (análise das possíveis inserções) e reinserido novamente na subárvore restante.
Fonte: (TICONA, 2008).

2.1.2 Representação de dados filogenéticos

Os estudos de biologia molecular tem produzido uma massiva quantidade de dados filogenéticos, os quais podem ser apresentados como sequências de diversos tipos. Dentre os principais tipos, destacam-se as sequências de DNA, RNA (do inglês, *RiboNucleic Acid*) e sequências de proteínas. A inferência filogenética tem como base esses três tipos de sequências, os quais representam o mesmo gene em cada um dos organismos.

Os dados das sequências são compostos por uma série de caracteres desenhados a partir de um alfabeto limitado. As sequências de DNA são compostas

de uma sucessão de nucleotídeos, que por sua vez são identificados por bases nitrogenadas, as quais se dividem em dois grupos: as purinas, compostas pelas bases Adenina e Guanina e, as pirimidinas, compostas pelas bases Citosina e Timina, representadas respectivamente por A, G, C e T. As sequências de RNA também são compostas por sucessão de nucleotídeos e possui os mesmos caracteres do DNA no grupo das purinas, já no grupo das pirimidinas o tipo Timina é substituído pelo tipo Uracila, desta forma, as sequências de RNA são representadas como A, G, C e U. Já as sequências de proteínas consistem de uma sucessão de aminoácidos, os quais podem assumir 20 estados diferentes, sendo representados por A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W e Y (ZWICKL, 2006).

Para inferir árvores filogenéticas, é necessário que as sequências sejam homólogas, isto é, que tenham ancestrais em comum. Neste sentido, quando o material genético é passado aos descendentes de um indivíduo, a sequência de caracteres sofre mutação. De acordo com Zwickl (2006), uma mutação pode ser a substituição (troca de um caractere por outro), a eliminação (deleção de uma quantidade específica de caracteres) ou a inserção de um caractere ou conjunto de caracteres.

Antes do processo de inferência, as sequências de dados devem ser melhoradas realizando o alinhamento dessas para cada um dos táxons obtidos. Pois, de acordo com Hypólito (2005), se há a necessidade de verificar se existe evidencias que um grupo com diferentes indivíduos divergem de um mesmo ancestral por mutação e seleção, basta comparar a partir do alinhamento, as sequências de caracteres que os representa, ou seja, o alinhamento de sequências simplesmente confere uma relação evolutiva em comum com as espécies consideradas. Esse processo é essencial para padronizar o comprimento das sequências que possivelmente sofreu alterações com as inserções e deleções na mutação.

De acordo com Souza Neto (2015, p. 10), as sequências em análise são alinhadas em forma de uma matriz, e serão utilizadas na reconstrução da árvore filogenética. A matriz é obtida através de técnicas de alinhamento de caracteres homólogos, agregando-os na maior quantidade de colunas possíveis, sendo que a quantidade de linhas determina a quantidade de espécies e a quantidade de colunas determina o comprimento das sequências. O comprimento das sequências

determina a quantidade de sítios do alinhamento, desta forma, dá-se o nome de sítio a cada coluna da matriz de alinhamento de sequências. A ilustração da Figura 6 mostra a matriz supracitada.

$$X = \{x_{ij}\} \quad \begin{array}{c} \\ Espécie\ 1 \\ Espécie\ 2 \\ Espécie\ 3 \\ . \\ . \\ . \\ Espécie\ n \end{array} \begin{array}{c} \begin{array}{cccccccc} S_1 & S_2 & S_3 & S_4 & . & . & . & S_n \end{array} \\ \begin{pmatrix} A & A & C & C & . & . & . & T \\ A & A & C & G & . & . & . & G \\ A & C & C & C & . & . & . & T \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ A & C & C & C & . & . & . & T \end{pmatrix} \end{array}$$

Figura 6 - Matriz de alinhamento de sequências de DNA, na qual as linhas representam as espécies e as colunas representam os sítios (base nucleotídica, nesse caso, para determinada posição homóloga entre as espécies).

Fonte: adaptado de (GONÇALVES, 2008).

Existem diversos softwares que realizam o processo de alinhamento de sequências genéticas, dentre os quais, pode-se citar Clustalw2 (LARKIN et al., 2007), BLAST (MCCLEAN, 2004), FASTA (BORDOLI, 2003), entre outros.

Nas subseções seguintes serão abordados os processos de Markov, bem como modelos evolutivos para sequências de DNA e taxas de heterogeneidade sobre sítios inseridas nestes modelos evolutivos.

2.1.3 Processos de Markov

Com a suposição de evolução independente de um sítio para outro, a inferência filogenética probabilística usa modelos evolutivos para calcular a probabilidade de mudança de um caractere para outro, considerando os caracteres encontrados em duas sequências quaisquer com determinado sítio.

Neste sentido, cabe enfatizar que um modelo evolutivo é uma modelagem matemática baseada em cadeias de Markov da transição de caracteres em uma provável mutação, em que o objetivo é modelar o processo de substituição, ou seja,

determinar dentro de uma mutação quais as probabilidades dos caracteres, substituírem uns aos outros.

No entanto, a mudança de um caractere para outro considerando duas sequências quaisquer com determinado sítio pode não ter ocorrido de forma imediata, isto é, pode haver estados de mutação que ocorreram durante o processo, mas que não são observáveis. Esse processo e algumas possíveis substituições ao longo da mutação sobre determinado sítio são ilustradas na Figura 7, em que a sequência original sofre mutação originando outras duas sequências, porém, nota-se que os estados intermediários não são observáveis nas sequências resultantes.

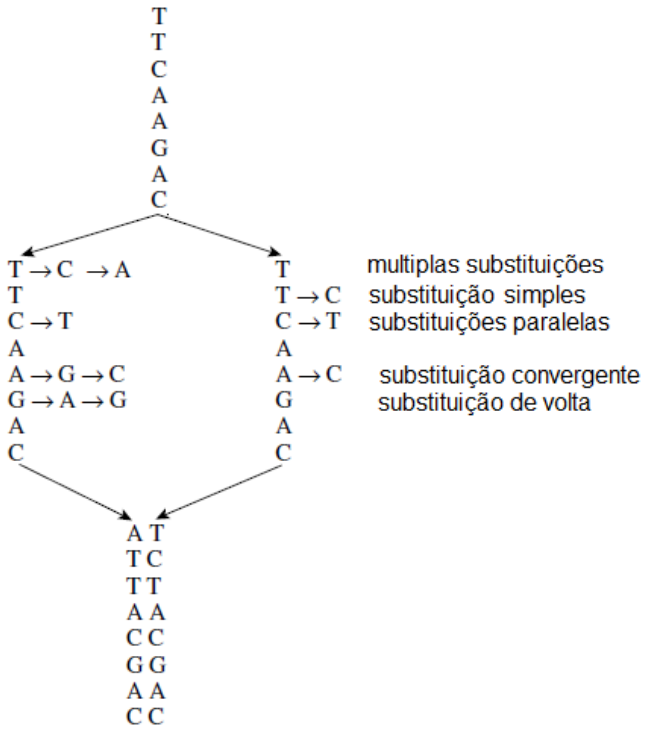

Figura 7 - Ilustração de várias substituições na mesma sequência, em que uma sequência ancestral de DNA divergiu em duas outras, as quais de forma independente acumularam substituições de nucleotídeos ao longo da linhagem, porém, apenas duas linhagens diferentes são visíveis.
Fonte: adaptado de (Yang, 2006, p.4).

Neste sentido, os processos de Markov são utilizados para mensurar de forma adequada a probabilidade de mudança de um estado para outro, isto é, considerando, de certa forma, os estados intermediários com mutações múltiplas ou silenciosas (substituição de volta). De acordo com Souza Neto (2015, p. 11) e Yang (2006, p. 4), um processo é considerado de Markov se dadas as probabilidades do

presente, as probabilidades futuras não dependerem do passado, sendo chamado de cadeias de Markov caso seu espaço de estados seja discreto ou enumerável em tempo contínuo. Supondo um processo de Markov com espaço de estados S discreto, com tempo definido nos reais positivos, e X a variável que assume valores em S, a característica principal deste processo implica que, dado um estado $X(t-h)=i$ em algum tempo $t-h$, a probabilidade de que $X(t)=j$, em um tempo futuro t, não depende dos valores de X antes do tempo $t-h$ (HYPÓLITO, 2005; SOUZA NETO, 2015, p. 11). Com uma suposição de homogeneidade sobre o processo de Markov, essa afirmação representa a probabilidade condicional de mudança de um caractere para outro e é formalmente representado pela eq. (3) (CYBIS, 2009; HYPÓLITO, 2005).

$$P(X(t)=j|X(t-h)=i) \tag{3}$$

A suposição de homogeneidade indica que a probabilidade condicional $P(X(t)=j|X(t-h)=i)$ é independente de t, e, portanto pode ser escrita como uma função de transição $p_{ij}(h)$, formalmente definida pela eq. (4) (SOUZA NETO, 2015, p. 11):

$$p_{ij}(s,t)=P[X(t)=j \mid X(s)=i] \qquad s \leq t \tag{4}$$

Desta forma, entender a origem da função de transição representada na eq. (4) é o ponto de partida para a justificativa da aplicação e resolução das cadeias de Markov. Neste sentido, considerando os conceitos de probabilidade total, Souza Neto (2015, p. 11) afirma que dados os eventos A, B_1 e B_2, tais que B_1 e B_2 são mutuamente exclusivos, a probabilidade de A pode ser obtida de acordo com a eq. (5):

$$P(A)=\sum_{\forall i} P(A \wedge B_i)=\sum_{\forall i} P(A \mid B_i)\, P(B_i) \tag{5}$$

Ainda de acordo com Souza Neto (2015, p. 11), condicionando $[X(t)=j|X(s)=i]$ a $[X(u)=r]$ para algum u na restrição $s \leq u \leq t$ e considerando $A=[X(t)=j \mid X(s)=i]$ e $B=[X(u)=r \mid X(s)=i]$, determina-se a eq. (6), conhecida

como a equação de Chapman-Kolmogorov (EWENS; GRANT, 2001; GRIMMENT; STIRZAKER, 1992):

$$p_{ij}(s,t) = \sum_{\forall r} P[X(t) = j, X(s) = i | X(u) = r, X(s) = i] P[X(u) = r | X(s) = i]$$

$$p_{ij}(s,t) = \sum_{\forall r} P[X(t) = j | X(u) = r] P[X(u) = r | X(s) = i]$$

$$p_{ij}(s,t) = \sum_{\forall r} p_{rj}\,(u,t) p_{ir}(s,u) = \sum_{\forall r} p_{ir}\,(s,u) p_{rj}(u,t) \tag{6}$$

A eq. (6) pode ainda ser reescrita na forma matricial, a qual está representada na eq. (7) juntamente com as condições pré-estabelecidas:

$$P(s,t) = P(s,u)P(u,t) \qquad s \leq u \leq t \tag{7}$$

Adicionando uma variação no instante t, condicionando os instantes como $s \leq t \leq t + \Delta t$, a eq. (7) da origem à eq. (8):

$$P(s, t + \Delta t) = P(s,t)P(t, t + \Delta t) \tag{8}$$

Subtraindo os dois termos da igualde por $P(s,t)$, e posteriormente colocando este termo em evidência no lado direto, encontra-se a eq. (9).

$$P(s, t + \Delta t) - P(s,t) = P(s,t)P(t, t + \Delta t) - P(s,t)$$

$$P(s, t + \Delta t) - P(s,t) = P(s,t)[P(t, t + \Delta t) - I] \tag{9}$$

Dividindo a eq. (9) pela variação Δt e tomando o limite com $\Delta t \to 0$, tem-se a uma equação diferencial matricial representada na eq. (10):

$$\lim_{\Delta t \to 0} \frac{P(s, t + \Delta t) - P(s,t)}{\Delta t} = P(s,t) \lim_{\Delta t \to 0} \frac{P(t, t + \Delta t) - I}{\Delta t}$$

$$\frac{\partial P(s,t)}{\partial t} = P(s,t)Q(t) \qquad s \leq t \tag{10}$$

Porém, sabe-se que a cadeia é homogênea, isto é, a função de transição independe dos valores absolutos de *s* e *t*, dependendo apenas da diferença $\tau = (t - s)$. Neste caso, reescrevendo a eq. (10) em função de τ tem-se a eq. (11):

$$\frac{\partial P(\tau)}{\partial \tau} = P(\tau)Q \tag{11}$$

Desta forma, de acordo com Hypólito (2005) e Souza Neto (2015, p. 12), considerando que a matriz de transição seja igual à matriz identidade, isto é, adotando as seguintes condições iniciais:

$$p_{ij}(0)\begin{cases}1 & se\ \ i = j\\ 0 & se\ \ i \neq j\end{cases}, \text{ou seja, } P(0) = I$$

tem-se uma única solução para a equação diferencial descrita na eq. (11), a qual é representada pela eq. (12):

$$P(\tau) = e^{Q\tau} \tag{12}$$

Nesse sentido, para este trabalho foram utilizadas cadeias de Markov de tempo contínuo, cuja caracterização é obter através da eq. (12), uma matriz de probabilidades de transição em determinado intervalo de tempo $P(\tau)$ através da matriz de taxas de transição Q, também conhecida como matriz geradora infinitesimal Q. Na matriz Q cada elemento representa a taxa infinitesimal de determinada mudança de estado do processo markoviano, em que $P(t)$ é a matriz de probabilidades de transição gerada a partir de $Q(t)$ e $P(j|i,t) \equiv P(X_t = j|X_0 = i)$ representa a probabilidade de transição do estado i para o estado j no tempo t (CYBIS, 2009).

De acordo Felsenstein (2004, p. 205-206) e Yang (2006, p.13-14), um procedimento realizando a decomposição da matriz Q em seus autovalores e autovetores leva à matriz P. Assim, a solução da eq. (12) é dada da seguinte forma:

$$P(t) = e^{Qt} = ADA^{-1} \tag{13}$$

onde D representa a matriz diagonal cujos elementos são os autovalores de Q, A é a matriz cujas colunas são os autovetores direitos de Q, A^{-1} é a inversa da matriz A e $P(t)$ é a matriz de probabilidade de transição no tempo t.

A soma dos elementos de uma determinada linha da matriz P é igual a 1, enquanto a soma dos elementos de cada linha da matriz Q é igual a zero, como

mencionado anteriormente (CYBIS, 2009). Ambas as afirmações são desmembradas e confirmadas matematicamente em (SOUZA NETO, 2015, p. 13).

Na subseção seguinte serão abordados os modelos que especificam de forma particular o processo de evolução temporal de cada sequência. Esses modelos são construídos basicamente utilizando a matriz geradora infinitesimal Q e a distribuição estacionária ou frequência de equilíbrio dos estados (caracteres). É importante destacar que além do conceito de probabilidade de transição em cadeias de markov, é de fundamental importância o conceito de distribuição estacionária, como pré-requisito ao entendimento desses modelos.

De acordo com Hypólito (2005), a distribuição estacionaria de um determinado estado, suponha j, é dada por πj e pode ser entendida como a probabilidade de observar o estado j em um ponto aleatório do tempo, quando não se tem nenhum conhecimento do estado inicial da cadeia. Assim, generalizando para o espaço de estados ou alfabeto de caracteres, Cybis (2009) apresenta um teorema que afirma que, para um processo de Markov com espaço de estados finito e com sua matriz geradora infinitesimal Q, existe um vetor estacionário p_0, o qual representa o espaço de estados.

Neste sentido, os modelos descritos a seguir assumem que a distribuição inicial dos processos é a própria distribuição estacionária, justificada pelo fato das sequências em estudo estarem evoluindo sob esse processo há bastante tempo, ou seja, no presente, as frequências dos estados ou caracteres já estariam muito próximas da distribuição estacionária (CYBIS, 2009).

2.1.4 Modelos Evolutivos

Todos os modelos evolutivos, tanto de DNA quanto de proteínas apresentam suas peculiaridades, apesar disso, na reconstrução filogenética, estes possuem uma característica em comum, que é a utilização de uma matriz de taxas infinitesimais e um vetor de distribuição estacionária. De acordo com Cybis (2009), independente do modelo evolutivo, considerando modelos evolutivos de DNA neste caso, a

representação de sua matriz de taxas infinitesimais pode ser padronizada como segue:

$$Q = \begin{pmatrix} -q_A & q_{A,G} & q_{A,C} & q_{A,T} \\ q_{G,A} & -q_G & q_{G,C} & q_{G,T} \\ q_{C,A} & q_{C,G} & -q_C & q_{C,T} \\ q_{T,A} & q_{T,G} & q_{T,C} & -q_T \end{pmatrix} \tag{14}$$

onde q_{ij} é a taxa instantânea de mutação da base i para a base j, e os elementos da diagonal principal são definidos de tal forma que a soma de todos os elementos da linha seja igual a zero, ou seja, $q_i = \sum_{i \neq j} q_{ij}$, para $i, j \in \Omega$. Note que essa é apenas uma representação de Q, pois cada modelo evolutivo possui sua matriz de taxas infinitesimais com taxas específicas de acordo com as características do modelo.

Como afirma Souza Neto (2015, p. 14), assim como a matriz de taxas infinitesimais, o vetor de distribuição estacionária também possui uma representação padrão que pode-se afirmar como genérica a todos os modelos evolutivos de nucleotídeos, conforme definida por Cybis (2009) e formulada a seguir:

$$p_0 = (\pi_A, \pi_G, \pi_C, \pi_T) \tag{15}$$

onde π_i é a proporção da base i do DNA e p_0 é a distribuição inicial do processo. Por questões de simplicidade, a subseção 2.1.4.1 a seguir apresenta um exemplo ilustrativo usando o modelo evolutivo de DNA mais simples e uma contextualização com outros modelos evolutivos de DNA e proteínas.

2.1.4.1 Jukes-Cantor e outros modelos evolutivos de DNA e proteínas

O modelo evolutivo mais simples para sequências de nucleotídeos foi proposto por Jukes e Cantor (1969), comumente denotado pela sigla JC69. Esse modelo assume que as substituições de bases de sequências de DNA ocorrem de maneira markoviana, que as probabilidades de mutação de um nucleotídeo i para um nucleotídeo j são todas idênticas e que todas as bases ocorrem com igual probabilidade (CYBIS, 2009; HYPÓLITO, 2005).

Desta forma, a matriz de taxas infinitesimais para as mutações de toda base i para toda base j, com espaço de estados $\Omega = \{A, G, C, T\}$ é definida pela eq. (16). Já a distribuição inicial de probabilidades das bases é definida na eq. (17). Note que a taxa total de mutação do processo é 3α (CYBIS, 2009; YANG, 2006, p. 6; HYPÓLITO, 2005).

$$Q_{JC69} = \begin{pmatrix} -3\alpha & \alpha & \alpha & \alpha \\ \alpha & -3\alpha & \alpha & \alpha \\ \alpha & \alpha & -3\alpha & \alpha \\ \alpha & \alpha & \alpha & -3\alpha \end{pmatrix} \tag{16}$$

$$p_0 = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right) \tag{17}$$

A matriz de probabilidades de transição $P = \{p_{ij}\}$ para o modelo JC69 é então obtida a partir da decomposição da matriz de taxas instantâneas $Q = \{q_{ij}\}$ mediante eq. (13), onde $Q = \{q_{ij}\}$ é a matriz definida na eq. (16). Desta maneira, a matriz de probabilidades de transição $P = \{p_{ij}\}$ é obtida através da resolução do sistema de equações representado na eq. (18), onde $\alpha_t = \frac{1}{4}(1 - e^{-4\alpha t})$ (CYBIS, 2009; YANG, 2006, p. 7; HYPÓLITO, 2005).

$$P_{JC69} = \begin{pmatrix} 1 - 3\alpha_t & \alpha_t & \alpha_t & \alpha_t \\ \alpha_t & 1 - 3\alpha_t & \alpha_t & \alpha_t \\ \alpha_t & \alpha_t & 1 - 3\alpha_t & \alpha_t \\ \alpha_t & \alpha_t & \alpha_t & 1 - 3\alpha_t \end{pmatrix} \tag{18}$$

De acordo com Cybis (2009) e Souza Neto (2015, p. 16), o modelo de JC69 foi um dos primeiros propostos na literatura para explicar a substituição de bases nucleotídicas, entretanto, apresenta hipóteses muito restritas que não explicam diversos dos fenômenos observados em sequências de DNA como as diferentes taxas para distintos tipos de mutações e diferentes proporções entre nucleotídeos. Neste sentido, será mostrada a seguir uma comparação dos parâmetros de alguns dos métodos mais abordados na literatura, para que se tenha noção das principais mudanças adotadas por cada um deles. Será tomado como modelo a matriz de taxas infinitesimais $Q = \{q_{ij}\}$, dada pela eq. (19) a seguir (CYBIS, 2009):

$$Q = \begin{pmatrix} h_1 & \alpha_2\pi_G & \alpha_4\pi_C & \alpha_6\pi_T \\ \alpha_1\pi_A & h_2 & \alpha_8\pi_C & \alpha_{10}\pi_T \\ \alpha_3\pi_A & \alpha_7\pi_G & h_3 & \alpha_{12}\pi_T \\ \alpha_5\pi_A & \alpha_9\pi_G & \alpha_{11}\pi_C & h_4 \end{pmatrix} \tag{19}$$

onde h_k é tal que a soma da linha k é zero, para todo $k \in \{1,2,3,4\}$.

A Tabela 1 mostra a comparação dos parâmetros de alguns modelos evolutivos de DNA.

Tabela 1 - Comparação de parâmetros dos modelos evolutivos de nucleotídeos.
Fonte: adaptado de (CYBIS, 2009).

Modelo Evolutivo	Distribuição $\boldsymbol{p_0}$	Taxa de mutação
Jukes Cantor (JC69)	$\left(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}\right)$	$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \alpha_6 = \alpha_7 = \alpha_8 = \alpha_9 = \alpha_{10} = \alpha_{11} = \alpha_{12}$
Kimura-2 Parâmetros (K80)	$\left(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}\right)$	$\alpha_1 = \alpha_2 = \alpha_{11} = \alpha_{12}$; $\alpha_3 = \alpha_4 = \alpha_5 = \alpha_6 = \alpha_7 = \alpha_8 = \alpha_9 = \alpha_{10}$
Kimura-3 Parâmetros (K81)	$\left(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}\right)$	$\alpha_1 = \alpha_2 = \alpha_{11} = \alpha_{12}$; $\alpha_3 = \alpha_4 = \alpha_9 = \alpha_{10}$; $\alpha_5 = \alpha_6 = \alpha_7 = \alpha_8$
Felsenstein 1981 (F81)	$(\pi_A, \pi_G, \pi_C, \pi_T)$	$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \alpha_6 = \alpha_7 = \alpha_8 = \alpha_9 = \alpha_{10} = \alpha_{11} = \alpha_{12}$
Tamura Ney 1993 (TN93)	$(\pi_A, \pi_G, \pi_C, \pi_T)$	$\alpha_1 = \alpha_2$; $\alpha_{11} = \alpha_{12}$; $\alpha_3 = \alpha_4 = \alpha_5 = \alpha_6 = \alpha_7 = \alpha_8 = \alpha_9 = \alpha_{10}$
Hasegawa, Kishino e Yano (HKY85)	$(\pi_A, \pi_G, \pi_C, \pi_T)$	$\alpha_1 = \alpha_2 = \alpha_{11} = \alpha_{12}$; $\alpha_3 = \alpha_4 = \alpha_5 = \alpha_6 = \alpha_7 = \alpha_8 = \alpha_9 = \alpha_{10}$
General Time Reversible (GTR)	$(\pi_A, \pi_G, \pi_C, \pi_T)$	$\alpha_1 = \alpha_2$; $\alpha_3 = \alpha_4$; $\alpha_5 = \alpha_6$; $\alpha_7 = \alpha_8$; $\alpha_9 = \alpha_{10}$; $\alpha_{11} = \alpha_{12}$

De acordo com os valores de parâmetros mostrados na Tabela 1, nota-se que os modelos evolutivos JC69, K80 e K81 possuem o mesmo vetor de distribuição inicial de probabilidades das bases, com valor igual para toda base do espaço, sendo este valor $^1/_4$. Já os modelos F81, TN93, HKY85 e GTR possuem o mesmo vetor de distribuição de probabilidades das bases, porém, com valores de distribuição diferentes, isto é, variando de base para base, com valor π_x, onde x é a base considerada no vetor de acordo com o espaço de bases.

Ainda de acordo com a Tabela 1, com relação às taxas de mutação da matriz de taxas infinitesimais, no modelo JC69 a taxa de mutação permanece a mesma,

independente das bases envolvidas na mutação. Já o modelo K80 adota duas taxas considerando as transições e transversões entre as bases envolvidas na mutação. O modelo K81 considera três taxas de mutação, duas taxas de transversões e uma de transições, neste caso, para as taxas de transversões, as duas taxas de transversões são originadas das combinações complementares das bases com as seguintes configurações $A \leftrightarrow C$ e $G \leftrightarrow T$ ou $A \leftrightarrow T$ e $G \leftrightarrow C$. Já o modelo F81 adota as mesmas taxas do modelo JC69 se diferenciando apenas na heterogeneidade do vetor de distribuição inicial de probabilidades das bases. O modelo TN93 assume três taxas, sendo duas originadas de transições e uma de transversões, para as taxas de transições, consideram-se mutações entre purinas e pirimidinas. O modelo HKY85 assume as mesmas duas taxas do K80 com transições e transversões, porém, com vetor de distribuição de probabilidades das bases não homogêneo. Por fim, o modelo GTR assume diferentes taxas para toda possível combinação do espaço de bases envolvidas na mutação.

De forma mais abrangente, incluindo os modelos já descritos na Tabela 1, a literatura trabalha com 7 modelos evolutivos de DNA, sendo eles: JC69, K80, F81, Felsenstein 1984 (F84), HKY85, TN93 e GTR. Por analogia, repete-se a metodologia usada com modelos evolutivos de DNA para trabalhar com modelos evolutivos de proteínas, com a diferença no tamanho do alfabeto que é de 20 caracteres. Neste sentido, na literatura, comumente se trabalha com 11 modelos evolutivos de proteínas sendo eles: LG, WAG, Dayhoff, JTT, Blosum62, mtREV, rtREV, cpREV, DCMut, VT e mtMAM.

Todos os modelos evolutivos discutidos até o momento determinam que os diferentes sítios nas sequências evoluem da mesma maneira e com a mesma taxa ao longo do sítio, porém, esta suposição pode ser irrealista em dados reais, desta forma, a subseção a seguir aborda os modelos com variação nas taxas evolutivas de heterogeneidades ao longo dos sítios.

2.1.5 Taxas evolutivas de heterogeneidade

Os modelos evolutivos apresentados na subseção 2.1.4 (Modelos Evolutivos) assumem a mesma taxa de evolução ao longo do sítio, o que significa que toda espécie do alinhamento sofrem evolução à mesma ‘velocidade’. Porém, de acordo com Cybis (2009), essa hipótese não tem fundamento biológico, pois os diferentes sítios de sequências de DNA podem estar submetidos a diferentes pressões evolutivas e de mutação, revelando que os sítios evoluem sobre condições particulares, em que alguns sítios são tão conservados ao ponto de não apresentarem substituições entre táxons relacionados, enquanto outros apresentam variação muito grande.

Neste sentido, ignorar essas características pode levar a grandes impactos na análise filogenética, pois estas podem aumentar o número de substituições, as quais não podem ser detectadas pela simples comparação dos dados das sequências (YANG, 2006, p. 110). Esse pressuposto levou à incorporação da heterogeneidade de taxas evolutivas sobre sítios, desencadeando um novo conjunto de modelos evolutivos, os quais proporcionam melhor ajuste dos dados observados e, consequentemente, reconstruções filogenéticas mais realistas (SOUZA NETO, 2015, p. 21).

De acordo ainda com Cybis (2009) e Souza Neto (2015, p. 21), esses modelos de heterogeneidade apenas informam como as diferentes taxas de mutação estão distribuídas entre os sítios, porém, não descrevem as características do processo de evolução das sequências em si e, por esta razão, há a necessidade de aliar o modelo com a matriz de taxas de mutação Q e o vetor de probabilidades iniciais p_0 abordados na subseção 2.1.4 (Modelos Evolutivos). No sentido da heterogeneidade, Yang (2006, p.110) afirma que uma abordagem sensata é usar uma distribuição estatística para modelar a variação da taxa e que, em geral, são usadas duas abordagens estatísticas, a distribuição discreta e a distribuição contínua gama. Ambas as distribuições são abordadas nas subseções seguintes.

2.1.5.1 Distribuição discreta para taxas de mutação

Este modelo divide os sítios das sequências de DNA em C categorias, com cada categoria assumindo taxa de mutação distinta. De acordo com Cybis (2009) e Yang (2006, p.110), a probabilidade de um determinado sítio pertencer a uma categoria l com taxa de mutação μ_l é definida por q_l, tal que estas probabilidades estão sujeitas às seguintes restrições $\sum_{l=1}^{C} q_l = 1$ e $\sum_{l=1}^{C} \mu_l q_l = 1$, em que a primeira restrição estima a frequência para 1 e a segunda fixa a taxa média de mutação da sequência, e é equivalente às restrições feitas à taxa de mutação geral dos modelos de substituição de bases abordados na subseção 2.1.4 (Modelos Evolutivos), para possibilitar a estimação dos comprimentos dos ramos.

Ainda de acordo com Cybis (2009) e Yang (2006, p. 110-111), o processo de evolução segue o formato da filogenia, em que através do processo de substituição, as taxas de mutação μ_l alongam ou encurtam os galhos para cada sítio. Desta forma, os sítios com taxa de mutação geral μ_l têm taxas de mutação de uma base para outra definida por $\mu_l Q$, tal que Q é a matriz de taxas instantâneas do modelo evolutivo compartilhada sobre todos os sítios. As restrições sobre as taxas de heterogeneidade anteriormente definidas, determinam que um modelo com C categorias de taxas de mutação possui $2C - 1$ parâmetros, além daqueles da matriz Q de taxas instantâneas (CYBIS, 2009; YANG, 2006, p. 110).

O fato é que em sequências de dados reais, as taxas de mutação de cada categoria para cada sítio são desconhecidas, assim, a probabilidade de observar os dados em algum local do sítio é determinada pela média ponderada sobre as taxas de cada categoria, o que leva o modelo a assumir que todos os sítios têm a mesma probabilidade de pertencer a uma determinada categoria, e indica que, apesar dos sítios evoluírem a taxas distintas, o processo é independente e identicamente distribuído.

As taxas de mutação u_l das categorias podem ser definidas previamente ou estimadas a partir dos dados das sequências. De acordo com Cybis (2009), a definição prévia das taxas acarreta um processo mais complicado, pois além da grande quantidade de parâmetros a serem estimados, exige sensibilidade e experiência do usuário. Já a estimação das taxas das categorias a partir dos dados

exige uma prévia definição do número C de categorias. Alguns estudos revelam que o número de categorias não deve exceder 3 ou 4 (YANG, 2006, p. 110).

Um caso especial do modelo com distribuição discreta para taxas de mutação é o modelo para sítios invariáveis, comumente representado pela simbologia $I+$ (por exemplo, JC69+I, HKY85+I, GTR+I, depende da matriz de transição escolhida). O modelo para sítios invariáveis assume dois grupos: o grupo dos sítios invariáveis com taxa de mutação $\mu_0 = 0$ ao longo do sítio e o outro segue o processo determinado por sua matriz de transição com taxa de mutação constante $\mu_1 = \frac{1}{1-p_0}$, tal que o parâmetro p_0 do modelo é a proporção de sítios invariáveis (CYBIS, 2009; YANG, 2006, p. 111).

De acordo com Cybis (2009), é importante ressaltar que, se um sítio possui bases diferentes ao longo das diversas sequências, ele não é constante e, portanto, não pode pertencer ao grupo de sítios invariantes. Entretanto, se o sítio não pertence ao grupo dos sítios invariantes, ainda assim, existe uma probabilidade de que ele seja constante, pois existem alguns trechos das sequências genéticas de DNA que são muito conservados, (geralmente, devido alguma função específica desempenhada pela sequência, que seria perdida caso houvesse alteração na base) de maneira que, em termos práticos, funcionam como sítios invariáveis.

2.1.5.2 Distribuição gama para taxas de mutação

Outra abordagem para variar as taxas de mutação ao longo dos sítios é assumir que as taxas seguem uma distribuição contínua. De acordo com Yang (2006, p. 111), dentre as várias opções analisadas, a distribuição gama se mostrou a mais amplamente utilizada. Formalmente, a distribuição gama é representada pela simbologia $+\Gamma$ (exemplos: GTR$+\Gamma$, HKY85$+\Gamma$, K80$+\Gamma$). Na justificativa de popularidade da distribuição gama, Cybis (2009) afirma que não há nenhuma razão biológica para a escolha dessa distribuição, mas que sua popularidade é decorrente de sua versatilidade.

Neste sentido, considere X uma variável aleatória, tal que $\mathrm{X} \sim \Gamma(a, b)$. De acordo com Cybis (2009) e Yang (2006, p. 112), a função de densidade de probabilidade da variável X é dada por:

$$f(x) = \begin{cases} \dfrac{b^a x^{a-1} e^{-bx}}{\Gamma(a)}, & x \geq 0 \\ 0, & caso\ contrário \end{cases} \tag{20}$$

onde a função gama $\Gamma(a)$ é definida por:

$$\Gamma(a) = \int_0^\infty t^{a-1} e^{-t}\, dt \tag{21}$$

note que a média é obtida com $\frac{a}{b}$ e a variância é obtida com $\frac{a}{b^2}$, sendo $a = b$, a média é 1, de acordo com as regras pré-estabelecidas na subseção anterior.

De forma análoga ao caso das taxas de distribuição discreta, não se sabe a taxa de mutação de cada sítio, e, para calcular a probabilidade dos dados, deve-se fazer uma média sobre todas as taxas de distribuição, porém, nesse caso, a distribuição é contínua, logo, esse resultado pode ser obtido mediante integração no intervalo de zero a infinito (CYBIS, 2009; YANG, 2006, p.112) ou equivalentemente e de forma menos custosa (em termos de processamento), aplicando o somatório sobre todas as categorias (metodologia adotada pelo software usado no estudo de caso) para obter um valor aproximado (FELSENSTEIN, 2004, p. 261).

De acordo com Yang (1996) o modelo que designa uma distribuição gama para as taxas de mutação possui a vantagem de explicar, através de apenas 1 parâmetro, a variação das taxas, além de possuir uma interpretação simples e maior apelo biológico, devido a sua característica contínua. No entanto, o custo computacional envolvido é muito grande a ponto de inviabilizar amostras com até 6 sequências (CYBIS, 2009).

Yang (1994) constata uma alternativa de qualidade para os dois modelos citados acima: o modelo da distribuição gama discretizada. Este apresenta uma fácil interpretação e uma boa aderência ao modelo de distribuição gama, além de um custo computacional compatível com o de taxas discretas. Definido pela simbologia $+\Gamma_d$ e utiliza C categorias, todas com probabilidade $1/C$, para aproximar a

distribuição gama. Na prática, o usuário deve escolher com quantas categorias deseja trabalhar, de maneira a estimar apenas um parâmetro (CYBIS, 2009).

Na subseção seguinte será abordada de forma mais aprofundada a integração das categorias aos modelos evolutivos e seu uso na inferência de filogenias usando o método de máxima verossimilhança, os demais métodos não serão abordados detalhadamente por não serem o foco de investigação deste estudo.

2.1.6 Método de Máxima Verossimilhança

A verossimilhança determina a probabilidade $\mathcal{P}(\mathcal{D}|\theta)$ de um conjunto de dados $\mathcal{D}$ (sequências genéticas devidamente alinhadas) ajustar-se ao modelo $\theta = \{\tau, \mathcal{B}, \mathcal{M}\}$, sendo que τ é uma topologia, $\mathcal{B}$ é o conjunto de comprimento dos ramos de τ e $\mathcal{M}$ é o modelo evolutivo de substituição de sequências genéticas. Neste sentido, o objetivo da MV é encontrar os parâmetros do modelo θ, de modo que a função de verossimilhança, definida como $L(\theta) = \mathcal{P}(\mathcal{D}|\theta)$, seja maximizada (SOUZA NETO, 2015, p. 40; TICONA, 2008).

Por questão de simplicidade e entendimento da estimação de MV, será adotado um exemplo meramente ilustrativo, o qual foi desenvolvido por (TICONA, 2008) usando a ilustração da Figura 8 a seguir.

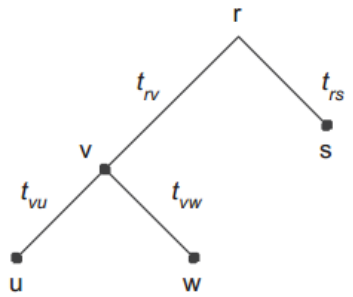

Figura 8 - Árvore usada para demonstrar o calculo da função de verossimilhança. Em seu estágio final, possui três espécies u, w e s, com duas espécies ancestrais v e r e comprimentos de galhos t_{rv}, t_{rs}, t_{vu} e t_{vw}.

Fonte: (TICONA, 2008).

Ainda no contexto da Figura 8, suponha que as três espécies *u*, *w* e *s* são representadas pelo conjunto de dados $\mathcal{D}$, que cada sequência do conjunto possui N_{sit} sítios, e que, u_j, w_j e s_j representam, respectivamente, os estados das espécies u, w e s no sítio j. Considere ainda a existência de um modelo de substituição de caracteres, que permite calcular as probabilidades de transição entre os estados e que a análise será realizada utilizando sequências de DNA com alfabeto de caracteres $\Omega = \{A, C, G, T\}$.

Sobre a premissa de que os sítios das sequências evoluem de forma idêntica e independente, Felsenstein (2004, p. 271) e Yang (2006, p. 100) definem que o cálculo da verossimilhança pode ser determinado através de um produto, conforme eq. (22) a seguir:

$$L = \prod_{j=1}^{N_{sit}} \mathcal{P}\left(\mathcal{D}^j, \theta\right) \tag{22}$$

onde $\mathcal{P}\left(\mathcal{D}^j, \theta\right)$ representa a verossimilhança no sítio *j,* por convenção, denotada a partir de agora como L_j.

Como os nós internos são desconhecidos, o valor de L_j será igual à soma das probabilidades de cada cenário possível, levando em consideração o espaço de estados (SOUZA NETO, 2015, p. 41). Neste sentido, Yang (2006, p. 100) define que a probabilidade de um determinado estado é função apenas do estado anterior (processo de Markov), indicando uma independência da ramificação da árvore, o que permite que, em relação à árvore da Figura 8, L_j possa ser expressa conforme eq. (23) a seguir:

$$L_j = \sum_{r_j \in \Omega} \sum_{v_j \in \Omega} \pi_{r_j} P_{r_j s_j}(t_{rs}) P_{r_j v_j}(t_{rv}) P_{v_j u_j}(t_{vu}) P_{v_j w_j}(t_{vw}) \tag{23}$$

onde: r_j e v_j representam os possíveis estados para os nós internos r e v, respectivamente; t_{ij} é o comprimento do galho que conecta os nós i e j; π_{r_j} é a frequência do nucleotídeo correspondente ao estado r_j no conjunto de sequências $\mathcal{D}$ e; $P_{x,y}(t)$ é a probabilidade da mudança do estado x para o estado y após um tempo

t (TICONA, 2008). Os termos $P_{x,y}(t)$ e π_{r_j} foram abordados na subseções 2.1.3 e 2.1.4.

O cálculo da verossimilhança pode ainda ser efetuado recursivamente empregando verossimilhanças condicionais de subárvores (TICONA, 2008; YANG, 2006, p. 102-103). Neste sentido, para o caso da Figura 8, a verossimilhança condicional da subárvore, cuja raiz é o nó r, denotada como $L_j^r(r_j)$, é a probabilidade dos eventos observados a partir de tal subárvore para os nós descendentes, dado que o estado do nó r no sítio j seja r_j. Logo, se o nó r tem os descendentes v e s, tem-se a verossimilhança definida recursivamente pela eq. (24) a seguir:

$$L_j^r(r_j) = \left[\sum_{v_j \in \Omega} P_{r_j,v_j}(t_{rv})\, L_j^v(v_j)\right] \times \left[\sum_{s_j \in \Omega} P_{r_j,s_j}(t_{rs})\, L_j^s(s_j)\right] \tag{24}$$

e para uma determinado nó folha a, no qual o estado a_j é fornecido por D, o termo $L_j^s(s_j)$ da eq. (24) assume valor 1 para a caractere encontrado na folha e 0 para os demais, tal comportamento é mostrado na eq. (25) a seguir (TICONA, 2008; YANG, 2006, p. 102):

$$L_j^a(x) = \begin{cases} 1, & se\ a_j = x \\ 0, & caso\ contrário \end{cases} \tag{25}$$

assim, usando essa propriedade, com relação à Figura 8, os termos recursivos $L_j^v(v_j)$ e $L_j^s(s_j)$ da eq. (24) são definidos respectivamente, como $L_j^v(v_j) = P_{v_j,u_j}(t_{vu})P_{v_j,w_j}(t_{vw})$ e $L_j^s(s_j) = P_{r_j,s_j}(t_{rs})$. De forma simplificada, a eq. (26) a seguir representa, recursivamente, o calculo de MV para o sítio j, em que o termo $L_j^r(r_j)$ é obtido através da eq. (24), com os termos recursivos $L_j^v(v_j)$ e $L_j^s(s_j)$ desmembrados pelas expressões supracitadas:

$$L_j = \sum_{r_j \in \Omega} \pi_{r_j} L_j^r(r_j) \tag{26}$$

note que o processo realizado na eq. (24) pode ser aplicado em qualquer nó interno de uma árvore e não somente em um nó raiz, mas que começando pela raiz permite a propagação dos cálculos até as folhas e, de certa forma, condiciona que os

cálculos sejam realizados cumulativamente das folhas para a raiz da árvore (YANG, 2006, p. 105).

Conforme eq. (22), o cálculo da verossimilhança total é realizado através do produto dos valores L_j de todos os sítios, porém, estes valores são muito pequenos de tal forma que costumam ocasionar erros de precisão numérica. Neste sentido, convenciona-se a utilização de logaritmos naturais por estes apresentarem resultados mais adequados (TICONA, 2008; YANG, 2006, p. 122). Desta forma, aplicando o logaritmo natural em ambos os lados da eq. (22), tem-se a eq. (27):

$$lnL = \sum_{j=1}^{N_{sit}} lnL_j \tag{27}$$

De acordo com a subseção 2.1.5 (Taxas evolutivas de heterogeneidade), o modelo M de substituição de sequências genéticas supõe que os sítios dos dados D evoluem a uma taxa constante, porém, como mostrado, essa suposição não tem fundamento biológico para dados reais. Neste sentido, as taxas de heterogeneidade sobre sítios serão incorporadas à função de verossimilhança por resultar em inferências mais realistas (SOUZA NETO, 2015, p. 43; TICONA, 2008).

A distribuição discreta ou taxas de heterogeneidades específicas por sítio são incorporadas ao modelo M através de um vetor $W = \left[w_1, w_2, \cdots, w_{N_{sit}}\right]^T$, no qual w_j representa a taxa de evolução correspondente ao sítio j. Nesse caso, o cálculo da verossimilhança é realizado da mesma forma descrita na eq. (24), porém, com comprimento de ramo t_{ij} sendo multiplicado por w_j (SOUZA NETO, 2015, p. 43; TICONA, 2008).

Já a inserção da taxa de heterogeneidade Gama no modelo, conforme subseção 2.1.5.2 (Distribuição gama para taxas de mutação), assume uma variável aleatória w_j obtida de uma distribuição Gama contínua (Γ). Assim, a verossimilhança para um sítio j é dada pela eq. (28) (SOUZA NETO, 2015, p. 43; TICONA, 2008):

$$L_j = \int_0^{\infty} \mathcal{P}\left(\mathcal{D}^{(j)} | \theta, w_j = x\right) f(x) dx \tag{28}$$

onde f é a função de densidade de probabilidade com distribuição Γ, e $\mathcal{P}\left(\mathcal{D}^{(j)} | \theta, w_j = x\right)$ é a verossimilhança do sítio j condicionado à taxa x. Como o

cálculo da integral é muito custoso computacionalmente, emprega-se uma função discreta Γ para aproximar tal valor, conforme eq. (29) (Yang, 1994):

$$L_j = \int_0^{\infty} \mathcal{P}\left(\mathcal{D}^{(j)}|\theta, w_j = x\right) f(x)dx \approx \sum_{k=1}^{N_{cat}} \rho_k \mathcal{P}\left(\mathcal{D}^{(j)}|\theta, w_j = x_k\right) \tag{29}$$

onde a distribuição Γ para as taxas dos sítios é discretizada em $k = 1 \cdots N_{cat}$ categorias, com x_k sendo a taxa de evolução da categoria k e ρ_k sendo a probabilidade da categoria k. A eq. (29) pode ainda ser reescrita da seguinte forma (SOUZA NETO, 2015, p. 44; TICONA, 2008):

$$L_j = \sum_{k=1}^{N_{cat}} \sum_{r_j \in \Omega} \rho_k \pi_{r_j} L_j^r(r_j x_k) \tag{30}$$

onde $L_j^r(r_j, w_j = x_k)$ é obtida da mesma forma que $L_j^r(r_j)$ da eq. (24), porém, multiplicando os comprimentos dos ramos t_{rv} e t_{rs} pela taxa da categoria x_i (SOUZA NETO, 2015, p. 44; TICONA, 2008).

Note que após obter a expressão da verossimilhança, a qual se encontra em função das probabilidades e dos comprimentos dos ramos, deve-se achar os valores de j (equações de verossimilhanças) que maximizem a verossimilhança (SOUZA NETO, 2015, p. 44; YANG, 2006, p. 128). Computar a expressão de MV para toda a topologia de uma só vez é praticamente inviável em função da grande quantidade de variáveis e parâmetros (galho, categoria, entre outros) do sistema. Neste sentido, a estratégia usada explora a ideia recursiva da eq. (24) para solucionar o sistema de equações para apenas um galho por vez e, da mesma forma, um parâmetros por vez. Note que os dois termos do lado direito da igualdade na eq. (24) podem ser resolvidos individualmente e, cada um está em função de apenas um galho.

Existem vários métodos numéricos utilizados na maximização da função de verossimilhança, dentre estes, de acordo com Morrison (2007) e Yang (2000), pode-se citar o método Newton-Raphson (YANG, 2006, p. 129,132), o método BFGS (do inglês, *Broyden-Fletcher-Goldfarb-Shanno*) (YANG, 2006, p. 133) e o método de Brent (BRENT, 1973; PRESS et al., 1992).

Para o método de Newton-Raphson, primeiro deriva-se L_j em função do número de ramos da árvore (os valores de j) e iguala-se a expressão resultante a

zero, em seguida, aplica-se o método de Newton-Raphson para achar os zeros do sistema. Desta forma, encontram-se os valores de j que ao serem substituídos na expressão L_j maximizam a verossimilhança da topologia (SOUZA NETO, 2015, p. 44; YANG, 2006, p. 133). O trabalho de (YANG, 2000) explana detalhadamente a aplicação do método de Newton-Raphson sobre a função de máxima verossimilhança.

O método de Newton-Raphson requer o gradiente e a Hessiana da função objetivo, impactando em alto custo computacional (SOUZA NETO, 2015, p. 44; YANG, 2006, p. 133). Neste sentido, visando reduzir o esforço computacional, manter a simplicidade do método de Newton e aumentar a gama de problemas que podem ser resolvidos, foram desenvolvidos os métodos quase-Newton, os quais aproximam a matriz hessiana usando apenas informações da derivada de primeira ordem, o BFGS é um método que se enquadra nessa categoria (YANG, 2006, p. 133). Desta forma, quando as Hessianas não estão disponíveis ou suas fatorações têm custo computacional proibitivo, o método BFGS e uma boa alternativa ao método de Newton.

Quando as primeiras derivadas são impossíveis ou dispendiosas de se calcular, uma abordagem alternativa consiste em utilizar um método livre de derivadas (YANG, 2006, p. 133). No contexto filogenético, o método de Brent é comumente citado como alternativa de uso de método livre de derivadas. Este é um método híbrido resultante da junção do método da bissecção, do método da secante e do método da interpolação quadrática inversa. Uma característica desejável desse método é que este não requer o uso de derivadas, bem como, não assume que a função seja diferenciável. Sobre o procedimento de funcionamento, este método usa inicialmente o método quadrático, se o resultado for um ponto fora do intervalo, usa-se o método da secante, se este também tiver como resultado um ponto fora do intervalo, usa-se o método da bissecção, o qual também exerce um controle sobre a velocidade real de convergência, se os métodos utilizados convergirem muito lentamente.

Independente do método utilizado, os cálculos de maximização são realizados para diversas topologias, permitindo a escolha daquela que proporcionar a maior verossimilhança dentre as testadas.

Em razão do vasto espaço de possíveis topologias (subseção 2.1.1), da fundamentação estatística e das premissas hipotéticas utilizadas, algumas estratégias são usadas para medir a confiabilidade de uma inferência. Neste sentido, a subseção seguinte aborda a estratégia de *bootstrap*.

2.1.7 Medidas de incerteza com *bootstrap*

Uma árvore filogenética resultante de um processo de reconstrução, independente do método de RAF utilizado pode ser vista como um ponto estimado do espaço de árvores e, portanto, pode gerar desconfiança a respeito da credibilidade do resultado da inferência. Neste sentido, é desejável aplicar procedimentos para cálculos de erros padrões, construção de intervalos de confiança ou realização de testes de significância que possibilite uma medida de confiabilidade às árvores inferidas. Existem alguns métodos corriqueiramente utilizados para avaliar a confiabilidade de árvores estimadas, neste contexto será abordado o método de *bootstrap*.

No contexto da filogenia, o *bootstrap* é uma técnica de reamostragem estatística empregada frequentemente na avaliação de clados[1] de uma árvore inferida. Felsestein foi o primeiro a sugerir tal técnica aplicada ao método de máxima verossimilhança (FELSENSTEIN, 1985), contudo, o *bootstrap* pode ser empregado em outros métodos de reconstrução filogenética como o método da máxima parcimônia ou o método de distância (SOUZA NETO, 2015, p. 45; YANG, 2006).

De acordo com Souza Neto (2015, p. 45), este método consiste em aplicar, determinado número de vezes, uma perturbação aleatória no conjunto de dados originais (sítios das sequências), de modo a originar árvores réplicas, que possibilite o cálculo de valores probabilísticos sobre os clados da árvore inicial. Baseia-se na suposição de que, pequenas modificações aleatórias nos dados não diminuirão a capacidade de encontrar os agrupamentos, se estes estiverem bem representados pelos dados. Neste sentido, grupos com baixos valores de *bootstrap* são aqueles

[1] clado: grupo monofilético ou grupo que contém todos os organismos descendentes do mesmo ancestral.

que deixaram de ser inferidos em muitas das réplicas, sugerindo baixo suporte dos dados.

De acordo com Ticona (2008) e Souza Neto (2015, p. 45), os conjuntos de sequências gerados pelo *bootstrap* (popularmente conhecido como réplicas) possuem o mesmo número de sítios que as sequências originais, porém, cada sítio das réplicas é escolhido aleatoriamente a partir dos dados originais. Desta forma, uma réplica pode ter várias cópias do mesmo sítio e não possuir cópia de algum sítio do alinhamento. Após a geração aleatória das réplicas das sequências, cada uma destas é empregada como entrada para o método de RAF considerado, o qual infere uma árvore para cada réplica.

Uma forma de processar as árvores inferidas a partir das réplicas é calculando uma árvore de consenso (SWOFFORD; SULLIVAN, 2009), em que cada árvore inferida a partir das réplicas permite calcular a proporção de cada clado da árvore inicial (inferida a partir dos dados originais) ser encontrado nas árvores das réplicas. Essa proporção é conhecida como proporção de *bootstrap* ou grau de suporte e mede a probabilidade de um clado ser recuperado no conjunto de réplicas (TICONA, 2008). A ilustração da Figura 9 mostra um diagrama do processo de análise de *bootstrap*.

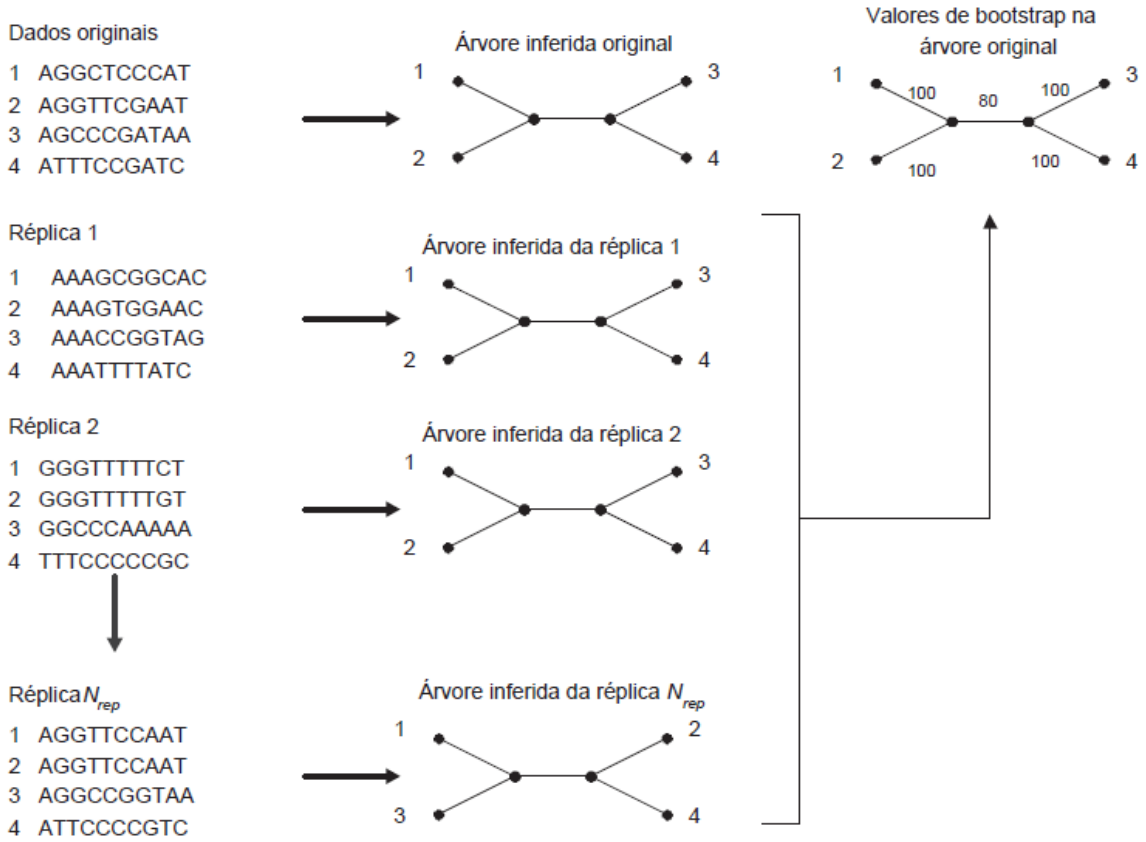

Figura 9 - Diagrama da análise de *bootstrap*: cada réplica leva a inferência de uma árvore e possibilita o calculo de *bootstrap* nos clados da árvore inicial para conferir sua confiabilidade.

Fonte: (TICONA, 2008).

De acordo com Ticona (2008), um inconveniente à aplicação da análise de *bootstrap* é a grande demanda de tempo necessária para realizar a análise, pois, para uma boa fundamentação estatística, a literatura recomenda uma grande quantidade de réplicas (500 a 1000), porém, em termos práticos, em alguns casos o tempo requerido de inferência de cada réplica pode ser inviável.

2.1.8 Heurística adotada pelo PhyML de acordo com a literatura

A heurística do PhyML aqui descrita é baseada nas duas publicações oficiais desse programa (GUINDON et al., 2010; GUINDON; GASCUEL 2003). O PhyML estima a máxima verossimilhança de alinhamentos de nucleotídeos e aminoácidos e, através de processos de Markov e estratégias de heterogeneidade sobre sítios,

implementa 8 modelos evolutivos de DNA e 12 modelos evolutivos de proteínas, podendo trabalhar com analises contendo até 4.000 sequências e até 2.000.000 caracteres por sequência, no entanto, sua zona de conforto esta situada entre 100 a 500 sequências e menos que 10.000 caracteres por sequência.

Para realizar os cálculos de máxima verossimilhanças de uma árvore, o PhyML utiliza a combinação de duas abordagens. Para entendê-las, considere a Figura 10 a seguir, em que na primeira abordagem, deseja-se calcular a MV do galho l_a e na segunda, deseja-se calcular a MV da subárvore U.

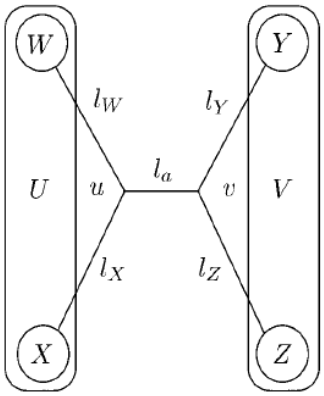

Figura 10 - árvore modelo para o calculo de MV realizado pelo PhyML.
Fonte: adaptado de (GUINDON; GASCUEL, 2003).

Para calcular a verossimilhança condicional do galho l_a no sítio corrente i, assume-se que a verossimilhança condicional das subárvores U e V ($L(i = h|U)$ e $L(i = h'|V)$, respectivamente) estão atualizadas para cada caractere do alfabeto. Desta forma a verossimilhança condicional do galho l_a é calculada de acordo com a eq. (31) a seguir:

$$L = \prod_i \sum_{h,h' \in \Omega} \pi_h L(i = h|U) L(i = h'|V)\, P_{hh'}(l_a) \tag{31}$$

note que h é a variação do alfabeto no nó u e h' é a variação do alfabeto no nó v. Equivalentemente, essa equação pode ser reescrita conforme eq. (32) a seguir:

$$L = \prod_i \left(\sum_{h \in \Omega} \pi_h L(i = h|U) \left(\sum_{h' \in \Omega} L(i = h'|V) P_{hh'}(l_a) \right) \right) \tag{32}$$

note que os somatórios dessa equação podem ser correlacionados com o calculo de subárvores apresentados na eq. (24), porém, com a diferença em estar usando a probabilidade sobre o galho ($\mathrm{P_{hh'}(l_a)}$) para juntar a MV das duas subárvores. É importante ressaltar que embora o termo de heterogeneidade e aplicação da função de log de máxima verossimilhança não estejam representados nessa equação, o PhyML os utiliza.

Já a verossimilhança condicional de determinada subárvore (nesse caso, U) para determinado sítio (nesse caso, i**)**, utilizada principalmente em processo de rearranjos, é calculada de acordo com a eq. (33) a seguir.

$$L(i = h|U) = \left[\sum_{g \in \Omega} L(i = g|w) P_{hg}(l_w)\right] \times \left[\sum_{g \in \Omega} L(i = g|x) P_{hg}(l_x)\right] \tag{33}$$

A eq. (33) encontrada nas publicações do PhyML apresenta uma terminologia diferente das abordadas nas equações anteriores. Neste sentido, por questão de simplicidade, reescrevendo cada termo da eq. (33) com a mesma terminologia da eq. (24), temos a equação eq. (34) a seguir.

$$L_{\mathrm{i}}^{\mathrm{u}}(\mathrm{u_i}) = \left[\sum_{\mathrm{w_i} \in \Omega} P_{\mathrm{u_i,w_i}}(l_{\mathrm{w}})\, L_{\mathrm{i}}^{\mathrm{w}}(\mathrm{w_i})\right] \times \left[\sum_{\mathrm{x_i} \in \Omega} P_{\mathrm{u_i,x_i}}(l_{\mathrm{x}})\, L_{\mathrm{i}}^{\mathrm{x}}(\mathrm{x_i})\right] \tag{34}$$

Após calcular a verossimilhança condicional das subárvore U e V, ajusta-se o tamanho de galho l_a através da eq. (32) e tem-se a verossimilhança total para o sítio i da árvore ilustrada na Figura 10. O uso combinado dessas duas equações e o armazenamento de valores de MV calculados em iterações anteriores permite o calculo e atualização de verossimilhanças locais e globais (de toda a árvore).

A heurística geral do PhyML é delineada como segue:

a) obtém a topologia modelo através de uma das seguintes estratégias: (I) uma topologia gerada dos dados com o algoritmo de distâncias BioNJ (GASCUEL, 1997); (II) uma topologia gerada dos dados com o algoritmo de parcimônia de Fitch (FITCH, 1971) ou; (III) topologias geradas aleatoriamente através do parâmetro de entrada RAND (usadas somente com as estratégias SPR e BEST);

b) usa-se a eq. (32) para calcular a MV de galhos e a eq. (34) para calcular a MV de subárvores, como dito anteriormente, o uso sucessivo e combinado dessas equações leva à MV de toda a árvore. A função de máxima verossimilhança é otimizada através do método de Brent e/ou BFGS (sendo que esta otimização ocorre com um parâmetro de cada vez);
c) utiliza refinamentos sucessivos com uma das seguintes estratégias de rearranjos de topologia: NNI, SPR e BEST (a estratégia BEST usa as duas estratégias anteriores e seleciona a melhor inferência resultante de ambas);
d) por fim, utiliza a estratégia de certificação de inferências, o *bootstrap*.

Os passos descritos nos itens b e c são aplicados sucessivamente para refinar a árvore até levar à convergência da inferência, isto é, até que modificações significativas deixem de acontecer. As equações eq. (32) e eq. (34) apresentam no PhyML a interessante característica de estimar localmente a MV de subárvores e galhos. Esta flexibilidade fornece uma grande economia de cálculos, pois permite que tanto na otimização de parâmetros quanto da topologia, ao invés de calcular a MV para cada mudança que venha a ocorrer, as mudanças sejam localmente verificadas se aumentam ou não a MV do galho ou subárvore, e, no final do processo, somente as otimizações que apresentam condições de otimalidade local são aplicados na árvore original, por fim, é calculada a MV de toda a topologia, reduzindo consideravelmente o tempo de processamento.

Ainda com relação ao passo descrito no item b, a abordagem NNI testa sucessivas movimentações à medida que calcula a MV localmente e, de acordo com os melhores escores vai armazenando os rearranjos que aumentam localmente a MV da árvore. Ao fim da iteração os melhores rearranjos são aplicados e a MV de toda a árvore é recalculada. Quando a aplicação dos rearranjos não aumenta a MV total da árvore inferida em relação à árvore anterior, um fator λ (última quantidade de rearranjos aplicada) vai sendo divido por dois e esse valor é a nova quantidade de rearranjos a ser aplicada à árvore antes dos rearranjos anteriores. Esse processo é conhecido como movimentos de volta à topologia anterior e continua sendo realizado até que se obtenha uma inferência com MV equivalente ou melhor que a MV da árvore anterior.

Da mesma forma, a abordagem SPR adota estratégias para evitar cálculos desnecessários. Para isso, usa-se um filtro de parcimônia para descartar movimentos menos promissores e, além disso, os rearranjos (quantidade limitada de rearranjos) que melhoram a MV e cálculos de MV sobre esses rearranjos são testados localmente. Somente após uma quantidade limitada de movimentações que melhoram a MV localmente, aplica-se de fato, estas movimentações e calcula a MV para toda a topologia. Após o procedimento com SPR, são aplicados alguns rearranjos com NNI para refinar ainda mais e finalizar o processo de inferência.

É importante enfatizar que o PhyML é um software estatístico que trabalha com otimizações de tamanhos de galhos, otimização de parâmetros do modelo evolutivo e otimizações topológicas. Além disso, o PhyML trabalha com geração de números aleatórios, como no caso de simulações de *bootstrap* e geração de topologias iniciais aleatórias quando não geradas pelo BioNJ. Todos esses fatores podem influenciar na variação do tempo de execução do PhyML e, consequentemente, na variação da quantidade de chamadas e tempo de execução das funções envolvidas no calculo de MV. Essas variações ocorrem mesmo quando simulado várias vezes com o mesmo arquivo de sequências e mesmos parâmetros de entrada.

2.1.9 Versões de implementações do PhyML

Atualmente, além da variedade de parâmetros de entrada, o PhyML apresenta três formas básicas de execução, de acordo com a forma de compilação. A primeira forma consiste de uma versão serial do PhyML, a segunda consiste da compilação com paradigma de memória distribuída MPI, fornecendo uma versão paralelizada sobre as réplicas *bootstrap*. Já a terceira forma consiste da compilação com suporte à biblioteca beagle (AYRES et al., 2012).

Beagle é uma API e biblioteca de alto desempenho para inferência filogenética estatística. A API fornece suporte ao núcleo dos cálculos de máxima verossimilhança em uma variedade de plataformas de hardware computacionais e explora as placas gráficas GPU (do inglês, *Graphics Processing Unit*) da Nvídia com

CUDA, OpenCL (do inglês, *Open Computing Language*) e CPU (do inglês, *Central Processing Units*) com instruções vetoriais e sistemas multi-core via OpenMP. De acordo com o manual do PhyML, com a implantação dessa API, este software tornou-se 10 vezes mais rápido na sua versão serial (resultados não mostrados na literatura) quando comparada com a versão serial sem o uso da API, porém, ainda não foi implementado no PhyML nenhum dos paradigmas de paralelização que esta biblioteca dá suporte. A estratégia dessa API é utilizar vetorização ao invés da estrutura em árvore para realizar os cálculos e, desta forma, evitar maiores custos computacionais com cópias de e para a memória. Apesar disso, a implantação dessa API no PhyML ainda não oferece suporte à certificação de topologias com a abordagem *bootstrap*, bem como não trabalha com a estratégia para computar a MV para sítios invariáveis, a proporção para sítios invariáveis.

2.1.9.1 Softwares alternativos que implementam a Máxima Verossimilhança

Considerando softwares de RAF que usam o método de MV no processo de inferência filogenética, esta subseção apresenta alguns dos mais conhecidos, usados e recentemente citados na literatura, bem como apresenta alguns dos últimos resultados comparativos do PhyML com outros softwares de RAF dessa categoria.

MorePhyML é um script criado e documentado por Criscuolo (2011), o qual adiciona a heurística de rearranjo de topologias TBR ao PhyML (versão 3.0 com SPR) e, adota algumas estratégias para possivelmente encontrar uma topologia mais acurada do que a resultante pelo PhyML.

No processo de inferência, o MorePhyML inicialmente usa o PhyML, começando com uma árvore gerada pelo BioNJ, aplicando o rearranjo da topologia com a heurística SPR e realizando o procedimento *bootstrap* com a heurística NNI. Após realizar esse procedimento com o PhyML, aplica-se sucessivos rearranjos com a heurística TBR sobre a topologia mais acurada resultante do PhyML.

Criscuolo (2011), compara o MorePhyML com os principais softwares que usam o método de MV no processo de inferência, o RaxML (do inglês, *Randomized*

Axelerated Maximum Likelihood, versão 7.0.4; STAMATAKIS, 2006), o GARLI (do inglês, *Genetic Algorithm for Rapid Likelihood Inference*, versão 1.0.695; ZWICKL, 2006) e o PhyML (versão 3.0; GUINDON et al., 2010). Os resultados das simulações revelaram que, na maioria das vezes, a heurística do MorePhyML obtém topologias mais acuradas do que as resultantes dos outros softwares usados nas comparações (PhyML 3.0 com SPR, RAxML e GARLI). Com relação ao PhyML, a melhoria na acurácia das topologias com o MorePhyML era previsível, pois há uma ampliação do espaço de busca (sobre as heurísticas NNI e SPR) pela aplicação de rearranjos com a heurística TBR, como explica Morrison (2007). Em contrapartida, essas simulações revelaram que o ganho em acurácia faz o MorePhyML perder em termos de tempos de execução, chegando a ser 8 vezes mais lento que o PhyML para sequências de proteínas e até 10 vezes mais lento com sequências de DNA. Com relação aos softwares RAxMX e GARLI, em termos de tempos de execução, essas comparações mostraram que o PhyML é ligeiramente mais rápido com sequências de DNA e equiparado com sequências de proteínas. Apesar das melhorias em acurácia do MorePhyML com relação ao PhyML, esse mesmo estudo revelou que ao iniciar o PhyML com mais de uma (várias) topologia inicial, reduz-se consideravelmente o percentual de árvores menos acuradas que o MorePhyML. Este estudo revelou ainda que outros softwares de inferência que usam a MV foram testados, porém, se mostraram muito lentos (como o PAUP (SWOFFORD, 2002), por exemplo) ou com acurácia muito ruim (como o FastTree (PRICE; DEHAL; ARKIN, 2010), por exemplo), mostrando que o PhyML realmente se encontra entre os melhores softwares de RAF usando MV. Todas as afirmações e informações aqui citadas podem ser conferidas com mais detalhes em (CRISCUOLO, 2011).

Já as simulações realizadas na versão mais atual do PhyML (versão 3.0), foram comparadas apenas com o software RAxML (versão 7.0), pois, de acordo com Guindon et al. (2010), o RAxML apresenta as características desejáveis para uma comparação justa e adequada, usando o modelo evolutivo GTR e rearranjos da topologia com a heurística SPR.

Segundo Guindon et al. (2010), nos resultados das simulações, o RAxML se mostrou ligeiramente mais rápido em conjuntos de dados de DNA, em contrapartida, as árvores estimadas pelo RAxML foram menos acuradas do que as estimadas pelo PhyML. Já para proteínas, o RAxML se mostrou mais lento do que o PhyML, porém

estimando topologias mais acuradas. Desta forma, uma análise genérica sobre a comparação do PhyML com outros softwares, pode-se concluir que não há um melhor software de RAF, mas softwares que se mostram vantajosos em algumas situações e desvantajosos em outras. Contudo, Guindon et al. (2010) afirma que em geral, os resultados mostram que o PhyML (versão 3.0) é rápido, preciso, estável e está pronto para uso.

2.2 Paradigmas de paralelização

O processamento paralelo é uma estratégia utilizada em computação para aproveitar de forma adequada os recursos tecnológicos disponíveis e resolver mais rapidamente problemas computacionais complexos, dividindo-os em subtarefas que serão alocadas em vários processadores para serem executados simultaneamente. Para isso, esses processadores se comunicam para que haja sincronização ou troca de informações (CALAZAN, 2013).

De acordo com Calazan (2013), o paralelismo pode ser introduzido de acordo com as diferentes arquiteturas disponíveis. Desta forma, pode-se explorar os núcleos presentes nas CPUs atuais utilizando a memória de forma compartilhada através do modelo de programação *Fork-Join* (conjunto de threads gerenciadas por uma destas). Esta arquitetura é denominada de multiprocessadores com memoria compartilhada (CHAPMAN; JOST; PAS, 2007). Existe ainda a arquitetura denominada de multicomputadores, em que os processadores usam a memória de forma distribuída através da interligação por uma rede de alta velocidade e colaboram entre si na computação paralela por meio de troca de mensagens (GROPP; LUSK, 1992). As caracterísricas das arquiteturas de memória compartilhada e distribuída estão ilustradas na Figura 11 a seguir.

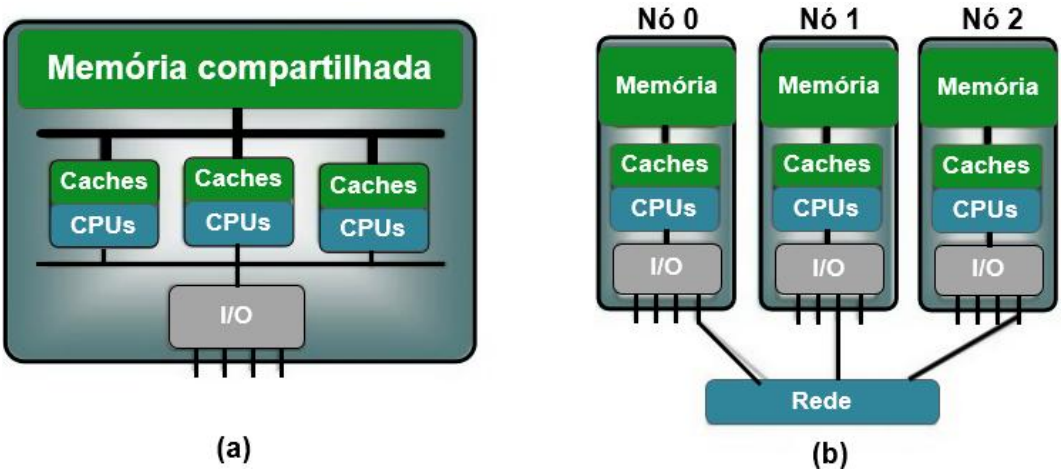

Figura 11 - Arquiteturas paralelas: (a) arquitetura de memória compartilhada e (b) arquitetura de memória distribuída.

Além dessas duas formas de utilização de arquiteturas (Figura 11), é possível ainda a utilização de um modelo híbrido que faz uso conjunto dessas duas arquiteturas, nesse caso, cada um dos nós da arquitetura de memória distribuída (Figura 11 - (b)) possui a estrura da arquitetura de memória compartilhada (Figura 11 - (a)), isto é, com mais de um núcleo de processamento. Por outro lado, as arquiteturas GPUs também apresentam grande potencial na área de processamento paralelo, mas não serão abordadas no escopo deste trabalho.

No sentido de paradigmas de paralelização, de acordo com Jeffers e Reinders (2015, p. 321) e Marzulo (2011) alternativas populares para paralelização são baseadas em tecnologias padrão, como OpenMP (DAGUM; MENON, 1998) e MPI (FORUM-MPI, 2012). O OpenMP segue o paradigma de memoria compartilhada; em contraste, o MPI apresenta, fundamentalmente, um modelo de passagem de mensagens, seu uso é mais adequado em arquiteturas distribuídas, como clusters de computadores. Já o modelo híbrido é a junção do paradigma de memória compartilhada com o paradigma de memória distribuída (OpenMP com MPI) (CALAZAN, 2013). As principais características dos paradigmas MPI e OpenMP são abordadas nas subseções seguintes.

2.2.1 Paradigma de memória distribuída MPI

Os modelos que exploram ambientes computacionais de memória distribuída são os que adotam o paradigma de troca de mensagens. Tais modelos são formados por rotinas de comunicação e sincronização de tarefas, sendo o MPI um exemplo. MPI é uma especificação de uma biblioteca de interface de troca de mensagens, isto é, um paradigma de programação paralela no qual os dados são movidos de um espaço de endereçamento de um processo para o espaço de endereçamento de outro através de operações de troca de mensagens (através de *sockets*, por exemplo) (RIBEIRO, 2011, p. 45).

O MPI é um paradigma adequado para paralelização de tarefas de granularidade grossa (KAMAL; WAGNER, 2010), isto é, as tarefas que exigem grande parte do tempo com processamento e pequena parte com comunicação (sincronização) com outros processos. O objetivo é o aumento de desempenho com a realização da menor troca possível de mensagens, visando anular e/ou reduzir o impacto do tempo despendido na distribuição e consolidação dos dados, no tempo total de execução. É importante destacar que todo o paralelismo em MPI é explicito, ou seja, de responsabilidade do programador, e implementado com a utilização dos construtores da biblioteca MPI.

MPI possibilita a implementação de programação paralela em memória distribuída em qualquer ambiente. Além de ter como alvo de sua utilização as plataformas de hardware de memórias distribuídas, MPI também pode ser utilizado em plataformas de memória compartilhada como as arquiteturas multiprocessadas como os computadores SMPs (do inglês, *Symmetric Multiprocessor*) e arquiteturas de acesso não uniforme à memória, NUMA (do inglês, *Non-Uniform Memory Access*). O paradigma MPI torna-se mais aplicável em plataformas híbridas, como cluster de máquinas NUMA (RIBEIRO, 2011, p. 45).

2.2.2 Paradigma de memória compartilhada OpenMP

O OpenMP é uma API para programação paralela em sistemas com memória compartilhada. Esta API consiste em um conjunto de diretivas de compilação, bibliotecas de funções e variáveis de ambiente para descrever tarefas paralelas, em que a intenção principal é prover ao programador uma interface simples e flexível para paralelizar aplicações (CALAZAN, 2013).

O OpenMP utiliza o modelo de programação *Fork-Join*, em que o programa começa de maneira semelhante a um processo simples, com um programa serial, denominado de thread inicial, quando essa thread encontra uma construção paralela do OpenMP (região paralela), cria-se um grupo de threads independentes e a thread inicial passa a ser denominada de thread mestre. Nessa região paralela esse grupo de threads trabalha a execução do código de maneira colaborativa onde podem compartilhar informações ou usar os dados de suas áreas privadas de memoria. No fim da região paralela, as threads são sincronizadas e finalizadas, permanecendo apenas a thread inicial (CALAZAN, 2013). A ilustração da Figura 12 exemplifica o modelo *Fork-join*.

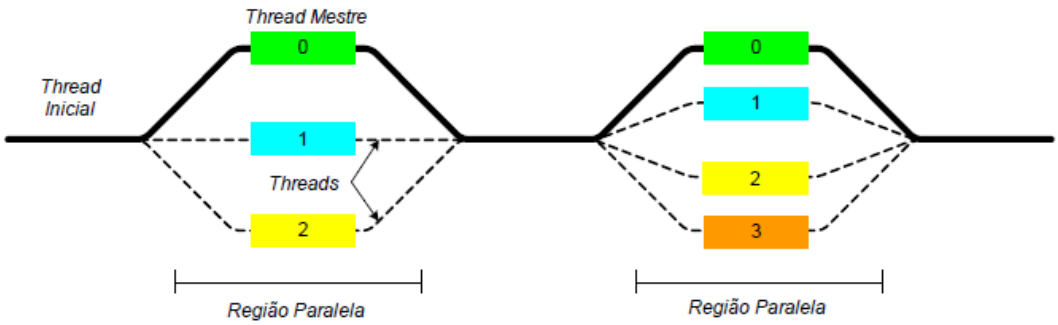

Figura 12 - Modelo Fork-Join suportado pelo OpenMP.
Fonte: (CALAZAN, 2013).

De acordo com Chapman, Jost e Pas, (2007, p. 23-30), as diretivas OpenMP se enquadram dentro de três requisitos básicos: (I) construtor paralelo - usa diretivas para criar e finalizar a região paralela com o grupo de threads; (II) construtores de divisão de trabalho - usam diretivas para definir o modo de divisão do trabalho entre as threads ao executar uma região paralela; e (III) construtores de sincronização -

usam diretivas para sincronizar os resultados de processamento das threads. No intuito de gerar códigos mais eficientes, pode-se fazer ainda uso combinado dos construtores (I) e (II). As diretivas de cada requisito não serão abordadas no escopo deste trabalho, no entanto, uma visão minuciosa sobre as características e sugestões de uso encontram-se no livro *Using OpenMP* (CHAPMAN; JOST; PAS, 2007) ou mesmo no site oficial do OpenMP (OPENMP, 2015).

De acordo com Ferreira (2012), em muitas situações, a chave para alcançar um bom desempenho para uma aplicação paralela é escolher a granularidade correta. A granularidade expressa a relação entre a quantidade de computação e a quantidade de comunicação em um algoritmo paralelo. Neste sentido, se a granularidade é demasiadamente fina, isto é, existe um grande número de pequenas tarefas, o desempenho pode ser comprometido ao sofrer com sobrecarga de comunicação.

De acordo com Jeffers e Reinders (2015, p. 322), o OpenMP é intrinsicamente de granularidade fina enquanto o MPI é de granularidade grossa. No entanto, o OpenMP pode assumir granularidade grossa através do paralelismo de tarefas ou granularidade fina através do paralelismo de dados (laços de repetição) (BAREKOVIC et al., 2011, p. 65; KAZUKI et al., 2000, p. 458). No sentido do paralelismo de dados, Muller et al. (2007, p. 191, 200, 222) e Jeffers e Reinders (2015, p. 321-322) afirmam que pode-se adotar estratégias de agrupamento para torna-la mais próxima possível da granularidade grossa. Isso pode ser obtido através da paralelização de laços mais externos ou no mais alto nível (externo) possível do código, no intuito de possivelmente garantir que a quantidade de processamento seja maior que a quantidade de comunicação, e desta forma, o algoritmo paralelo possa proporcionar melhor desempenho.

3 METODOLOGIA

Esta Seção aborda as ferrmentas de apoio utilizadas no processo de paralelização do PhyML, as etapas básicas adotadas e os critérios usados para encontrar os gargalos e realizar a análise de desempenho da versão paralela.

3.1 Ferramentas utilizadas

As ferramentas aqui mencionadas serão abordadas com abrangência na Seção 4 (**DESENVOLVIMENTO**). Gprof é uma ferramenta de perfilamento de aplicações e, nesse contexto, foi utilizada na detecção de gargalos do PhyML em virtude de dois motivos: (I) por ser uma ferramenta livre para uso e, (II) e por já ser a ferramenta padrão de perfilamento do pacote PhyML. No entanto, o GProf contabiliza os tempos de execução sem considerar o fluxo de execução (não contabiliza os tempos de execução das sub-rotinas dentro da função) e, além disso, notou-se que algumas funções do PhyML possuem tempos de execução variando em torno de nanosegundos enquanto o GProf possui precisão de microssegundos. Neste sentido, adotou-se a estratégia de computar manualmente o tempo de cada função gargalo individualmente (com base no ranque de funções gargalos gerado pelo GProf). O objetivo é mensurar de forma adequada e com valores mais precisos, a porcentagem de tempo cumulativo (de todas as vezes que a função foi chamada) e o tempo médio de execução de cada função em relação ao tempo total do PhyML. Para realizar esse processo foi utilizada a ferramenta timespec, que é uma estrutura pertencente ao arquivo de cabeçalho "*time.h*" da linguagem de programação C, a qual possui código livre.

EPCC Microbenchmark é uma ferramenta de benchmark utilizada na otimização de aplicações paralelizadas usando OpenMP. Neste contexto, ela foi utilizada para investigar se as diretivas OpenMP a serem inseridas na paralelização de uma função gargalo não geraria um *overhead* maior que o proprio tempo médio de execução da função, levando a inviabilidade de paralelização. O *overhead* para cada função gargalo é medido considerando as diretivas previstas à paralelização da função (previsão através de análise no próprio código fonte), o tempo médio de execução dessa função (usado para simular o tempo de execução da função na EPCC sobre as diretivas previstas) e a quantidade de threads desejada.

3.2 Etapas básicas do processo de paralelização do PhyML

Este trabalho se iniciou com um estudo geral sobre o processo de reconstrução de árvores filogenéticas, dando seguimento com um estudo mais direcionado ao método de máxima de verossimilhança e os modelos evolutivos utilizados. Em seguida foi realizado um estudo detalhado do programa PhyML, tanto em relação ao código fonte quanto em relação à literatura disponível. Por fim, foi realizado um levantamento teórico sobre a exploração de estratégias de paralelização usando ferramentas de apoio. Neste sentido, as etapas básicas utilizadas no processo de paralelização do PhyML foram: (a) encontrar as funções gargalos do PhyML através do Gprof, uma ferramenta de perfilamento; (b) medir os tempos seriais dessas funções usando a estrutura *timespec*; (c) identificar e estimar a viabilidade ou não de paralelização dos gargalos através da granularidade das funções e dos *overhead*s das diretivas OpenMP obtidos com a ferramenta de benchmark EPCC Microbenchmark; (d) paralelizar as funções gargalos viáveis; e, (e) medir o desempenho dos gargalos individualmente e do PhyML, paralelizados.

3.3 Critérios usados na detecção de gargalos e análise de desempenho

Conforme abordado na subseção 2.1.8 (Heurística adotada pelo PhyML de acordo com a literatura), o PhyML apresenta grande variabilidade na quantidade de chamadas de cada função e em seus tempos de execução, mesmo quando simulado com o mesmo arquivo de sequências e mesmos parâmetros de entrada. Para efeito ilustrativo, a Figura 13 mostra essa características para a função *LK_Core* do PhyML, para a qual foram realizadas 20 simulações do programa com o mesmo arquivo de sequências e as mesmas configurações de parâmetros de entrada.

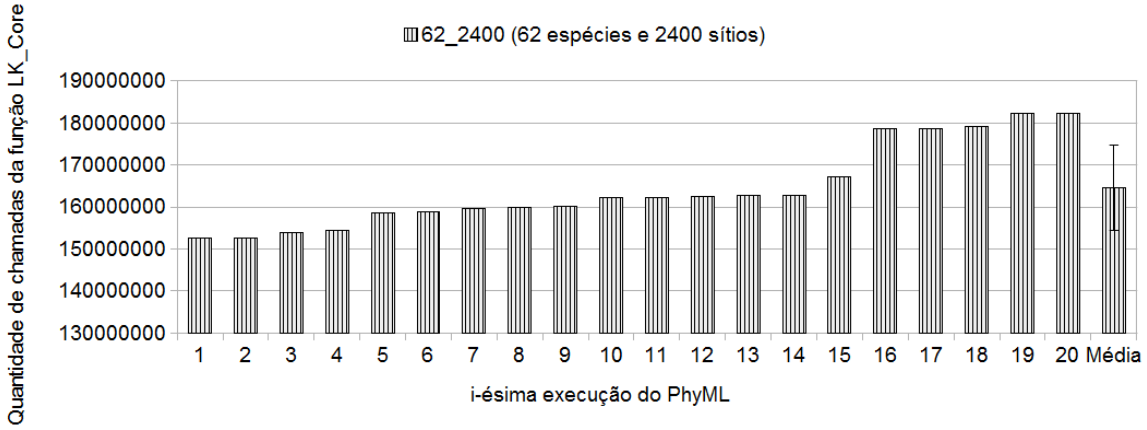

Figura 13 - Variabilidade da quantidade de chamadas da função *LK_Core* sobre 20 simulações do PhyML com o mesmo arquivo de sequências e mesma configuração de parâmetros (por questões de melhor visibilidade da ilustração, os valores de quantidade de chamadas foram ordenados progressivamente).

Essa característica indesejável pode dificultar a comparação de valores temporais na análise de desempenho do PhyML. Neste sentido, foi adotada a estratégia de realizar 20 simulações para todo critério desejado (serial e paralelo) e trabalhar com o valor da média e desvio padrão sobre os valores dessas 20 simulações. A ideia é manter visíveis as características de variabilidade de tempos de execução e quantidade de chamadas das funções gargalos e, possibilitar a comparação de tempos sequenciais e paralelos de forma adequada.

Além da análise de desempenho completa do PhyML e suas versões paralelas, adotou-se a estratégia de medir o desempenho de cada uma das funções gargalos individualmente com o intuito de descobrir a contribuição ou não dessas para o desempenho do programa.

4 DESENVOLVIMENTO

Esta Seção apresenta as funções gargalos do PhyML, a análise de viabilidade de paralelização e as estratégias de paralelização aplicadas às funções viáveis.

Os testes e análises aqui apresentadas foram realizados no computador de alto desempenho CACAU (Centro de Armazenamento de dados e Computação Avançada da Universidade Estadual de Santa Cruz-UESC) que se encontra no laboratório NBCGIB (Núcleo de Biologia Computacional e Gestão de Informações Biotecnológicas) na UESC. O cluster possui duas arquiteturas. A arquitetura de nós comuns do CACAU possui 20 nós de cálculo, totalizando 1,0 TeraFLOP. Cada nó possui 2 processadores Intel(R) Xeon(R) E5430 com 2.66GHz, QuadCore, 16 GB de memória RAM e 1 placa gigabit ethernet. Em dados gerais a arquitetura comum do CACAU totaliza 160 cores e 320 GB de memória RAM. O CACAU possui ainda uma arquitetura GPU, esta é composta de 3 nós de cálculos que são equipados com placas gráficas GPU, cada nó contendo 12 cores Intel Zion Xeon ES-2440 e 2 GPU NVÍDIA Tesla K20 em cada nó, totalizando 6 placas gráficas e 48 GB de memória RAM. A arquitetura GPU não será utilizada no desenvolvimento deste trabalho, apenas nas análises de desempenho apresentadas na Seção 5 (RESULTADOS E DISCUSSÕES).

4.1 Sobre os dados de sequências usados nas simulações

Os arquivos de sequências utilizados como parâmetro de entrada nas simulações do PhyML foram escolhidos baseados em três quesitos básicos, considerando a quantidade de espécies do arquivo (quantidade de sequências alinhadas) e a quantidade de sítios (tamanho das sequências alinhadas), sendo estes dados: (I) pequena quantidade de espécies (variando de um arquivo para outro) e mesma quantidade de sítios (em todos os arquivos); (II) pequena quantidade de espécies (mesma quantidade de espécies em todos os arquivos) variando a quantidade de sítios (de um arquivo para outro); (III) grande quantidade

de espécies e grande quantidade de sítios (variando ambas de um arquivo para outro), porém, com limites dentro da zona de conforto do PhyML. Com relação aos termos “pequenos arquivos de sequências” e “grandes arquivos de sequências” usados no agrupamento dos arquivos, levou-se em consideração as proximidades de tempos de execução. Alguns desses dados de DNA são mostrados na Tabela 2.

Tabela 2 - Dados de DNA utilizados como arquivos de entrada nas simulações do PhyML**.**

Arquivo de entrada	Quant. espécies	Quant. sítios	Quesito
12_3768	12	3768	I
20_3768	20	3768	I
29_3768	29	3768	I
42_3768	42	3768	I
55_3768	55	3768	I
62_0600	62	0600	II
62_1200	62	1200	II
62_1800	62	1800	II
62_2400	62	2400	II
62_3000	62	3000	II
62_3768	62	3768	II
218-2294	218	2294	III
406_6016	406	6016	III
500_1398	500	1398	III
626_4716	626	4716	III

Os arquivos contendo as sequências com as características descritas nos quesitos I e II foram obtidos apenas para sequências sintéticas de DNA, os quais foram utilizados principalmente para analisar o comportamento e escalabilidade das funções gargalos do PhyML considerando as duas situações: (I) aumentando a quantidade de espécies mantendo o tamanho de sequências fixo e (II) aumentando o tamanho das sequências mantendo a quantidade de espécies fixa, conforme mostrados na Tabela 2. No processo para obter os arquivos de sequências com tais características, é necessário seguir um procedimento cuidadoso para que a retirada de algumas sequências do arquivo não interfira no processo de RAF e cause deformações nos clados da topologia original. Essas sequências foram retiradas do

trabalho de (GONÇALVES, 2008), o qual descreve e realiza todo esse processo para respeitar as características dos clados ao obter tais sequências.

Já os arquivos de sequências do quesito III foram utilizados principalmente para analisar o desempenho do PhyML. Para este quesito, a ideia foi trabalhar com sequências já simuladas pelo PhyML com o objetivo de evitar eventuais contratempos em relação ao formato do arquivo aceito pelo programa entre outras particularidades. Esses arquivos foram retirados do site oficial do PhyML (ATGC, 2015a), para sequências de DNA (DNABIGDATA, 2015) e para sequências de proteínas (PROTEINDATA, 2015). De acordo com informações do site oficial, esses arquivos de sequências forma retirados do banco de dados de sequências filogenéticas TreeBASE (TREEBASE, 2015) e, usados anteriormente em baterias de testes (ATGC, 2015b) comparando estratégias do PhyML, como NNI e SPR, por exemplo, e o desempenho deste com relação à outros softwares de RAF.

4.2 Detecção de funções gargalos

Nesta subseção serão abordadas algumas características da ferramenta de perfilamento GProf, bem como a sua utilização para encontrar as funções gargalos do PhyML.

4.2.1 Sobre a ferramenta de perfilamento GProf

A detecção de gargalos em aplicações pode ser feita de diferentes maneiras, as mais comuns são: monitorar o comportamento de cada função computando o tempo de execução manualmente ou usar alguma ferramenta de monitoramento que trace um mapeamento de tempo de execução e quantidade de chamadas de cada função. Para aplicações que sejam extremamente grandes ou que seja complexo fazer a leitura do código, como é o caso do PhyML, o monitoramento manual é

tedioso, desta forma, é ideal o uso de uma ferramenta que possa automatizar esse processo.

Ferramentas de perfilamento permitem a leitura de onde o programa costuma gastar mais tempo, considerando o levantamento do fluxo de execução das funções e mensurando os tempos de execução de trechos do código, bem como a frequência de chamadas dessas funções. Existem várias ferramentas de perfilamento de aplicações como: Valgrind (SEWARD; NETHERCOTE; WEIDENDORFER, 2008) ferramenta gratuita e de código livre usada em análise dinâmica de aplicações, VisualVM (VISUALVM, 2015) ferramenta gratuita utilizada em aplicações que utilizam java, GProf (GPROF, 2015) ferramenta gratuita e de código livre usada em análise dinâmica de aplicações, VTune (VTUNE, 2015) ferramenta proprietária da Intel, entre outras. Dentre as ferramentas citadas, o Valgrind e VTune foram utilizados para testes de consumo de memória. Já o GProf será abordado de maneira detalhada em virtude tanto do PhyML já possui-lo como ferramenta padrão de perfilamento, quanto da sua utilização para encontrar as funções gargalos do PhyML.

O GProf (*GNU Profiling*) é uma ferramenta do GCC (GNU *Compiler Collection*) desenvolvida por Jay Fenlason e Richard Stallman e descrita no manual "*GNU GProf - the GNU Profiler*" (FENLASON; STALLMAN, 2000). Esta é uma ferramenta de análise dinâmica que coleta informações do programa em tempo de execução e constrói o perfil de tempo de execução do programa, ranqueando as funções que mais exigem tempo de execução. As funções mais custosas computacionalmente são candidatas a serem reescritas ou paralelizadas segundo algum paradigma visando uma melhora no desempenho geral do programa.

O perfilamento com o GProf permite a identificação da quantidade de funções existentes no código, a quantidade de chamadas e a porcentagem do tempo gasto para executar cada função. Como resultado do perfilamento, o GProf fornece dois tipos de informações, a chamada gráfica (*Call Graph*) e o plano de perfil (*Flat Profile*). A chamada gráfica mostra, para cada função, quais funções a chamaram, quais outras funções ela chamou e quantas vezes e, uma estimativa de quanto tempo foi gasto nas sub-rotinas de cada função, conforme ilustração da Figura 14.

```
348                 Call graph (explanation follows)
349
350
351  granularity: each sample hit covers 2 byte(s) for 0,00% of 3241,24 seconds
352
353  index % time    self  children    called     name
354                                                     <spontaneous>
355  [1]     100,0    0,00 3241,18                 main [1]
356                   0,00 2993,97       1/1           Simu_Loop [2]
357                   0,00  227,97       1/1           aLRT_From_String [26]
358                   0,00   12,32       1/1           Prepare_Tree_For_Lk [42]
359                   0,00    6,37       1/1           Dist_And_BioNJ [46]
360                   0,17    0,35       1/1           Compact_Data [59]
361                   0,00    0,03       1/1           Get_Seq [93]
362                   0,01    0,00       1/1           Check_Br_Lens [116]
363                   0,00    0,00       1/949276      Lk [3]
```

Figura 14 - Gráfico de chamadas gerado pelo GProf para o programa PhyML com arquivo de entrada contendo sequências de DNA.

Na Figura 14, as colunas ilustradas representam, respectivamente: (*index*) índice que identifica a função em análise; (*% time*) porcentagem de tempo gasto pela função e suas sub-rotinas; (*self*) o tempo gasto pela função sem considerar o fluxo das subrotinas ou tempo exclusivo; (*children*) o tempo gasto pelas sub-rotinas da função; (*called*) a quantidade de vezes que cada sub-rotina foi chamada dentro da função/total de chamadas em todo o programa e; (*name*) o nome da função e suas sub-rotinas. Note que a terceira coluna nomeada *self*, possui valores para algumas funções enquanto que, para outras funções isso não ocorre. Isso acontece em virtude do GProf possuir precisão de microssegundos enquanto o tempo exclusivo das funções mostradas na ilustração estão variando em torno de nanosegundos.

Já o plano de perfil mostra quanto tempo o programa gastou em cada função e quantas vezes a função foi chamada. A informação é clara e pode-se identificar facilmente, de acordo com o ranque, quais funções são os gargalos do programa, conforme ilustração da Figura 15.

```
1  Flat profile:
2
3  Each sample counts as 0.01 seconds.
4    %   cumulative   self              self     total
5   time   seconds   seconds    calls   s/call   s/call  name
6   39.48     13.36    13.36 24726870     0.00     0.00  Lk_Core
7   38.56     26.41    13.05    20211     0.00     0.00  Update_P_Lk_Nucl
8   15.46     31.64     5.23 24726870     0.00     0.00  Pull_Scaling_Factors
9    4.17     33.05     1.41 85197420     0.00     0.00  Rate_Correction
```

Figura 15 - Plano de perfil gerado pelo GProf para o programa PhyML com arquivo de entrada contendo sequências de DNA.

De forma análoga à ilustração anterior, na Figura 15, as colunas ilustradas representam, respectivamente: (*% time*) o percentual de tempo de execução exigido pela função durante a execução de todo o programa; (*cumulative seconds)* é a soma dos tempos cumulativos (de todas as vezes que a função foi chamada) de todas as funções listadas até o momento na terceira coluna (*self seconds*); (*self seconds*) o tempo exclusivo cumulativo da função (nesse caso, cumulativo em relação á todas as chamadas da função na mesma execução do programa, sem considerar o fluxo das subrotinas); (*calls*) a quantidade de vezes que a função foi chamada; (*self s/call*) o tempo gasto pela função para cada chamada; (*total s/call*) o tempo gasto pela função e suas sub-rotinas e; (*name*) a identificação da função.

A primeira, a terceira e a quarta coluna da Figura 15, é a base para determinar os gargalos de uma aplicação, considerando respectivamente, percentual de tempo de execução da função em relação ao tempo de execução do programa, o tempo exclusivo cumulativo de execução que ela consome e a quantidade de chamadas.

4.2.2 Funções gargalos do PhyML

Através do GProf, os gargalos do PhyML foram detectados com arquivos de sequências de DNA e proteínas, considerando, em termos de quantidade de espécies e tamanho das sequências, pequenos e grandes arquivos de dados de entrada. Foi realizada uma bateria de testes no CACAU (resultados não mostrados

em razão do grande volume de dados) com diferentes parâmetros de entrada do PhyML para possivelmente detectar alterações no ranque das funções gargalos. Dentre os parametros testados na bateria de testes, podem ser citados: a) com relação ao tipo de dados: 8 modelos evolutivos de DNA e 12 modelos evolutivos de proteínas; b) com relação à otimizações topológicas: NNI, SPR e BEST (e suas combinações com os modelos evolutivos); c) com relação à heterogeneidade de taxas: com taxas fornecidas pelo usuário e com taxas retiradas dos próprios dados e; d) com relação à variação ao longo do sítio: com e sem proporção para sítios invariáveis. A bateria de testes revelou que a variação nos parâmetros de entrada mantém o mesmo ranque de funções gargalos, mas que, quando as simulações correlacionam o tipo de dados: DNA e proteínas, esse ranque se altera.

Como afirmado na subseção anterior, os principais critéios de classificação das funções gargalos usados pelo GProf estão relacionados ao ranqueamento através do percentual de tempo de execução exigido pela função e do tempo exclusivo cumulativo. Desta forma, vale ressaltar que o percentual de tempo de execução é a porcentagem de tempo exclusivo (sem considerar o tempo de execução das subrotinas do fluxo de execução) que a função consumiu em relação ao tempo total de execução do programa. Já o tempo exclusivo cumulativo é o somatório dos tempos consumidos por determinada função (considerando todas as chamadas dessa função na mesma execução do programa) sem considerar o tempo de execução das subrotinas do fluxo de execução. Os ranques ilustrados nessa subseção consideram somente as 6 primeiras funções mais custosas computacionalmente, pois as demais apresentam tempo de execução cumulativo praticamente insignificante em relação a estas 6.

Para sequências de DNA, de acordo com os resultados do GProf, os tempos exclusivos cumulativos para pequenos arquivos de entrada são ilustrados na Figura 16, já os tempos exclusivos cumulativos para grandes arquivos de entrada são ilustrados na Figura 17.

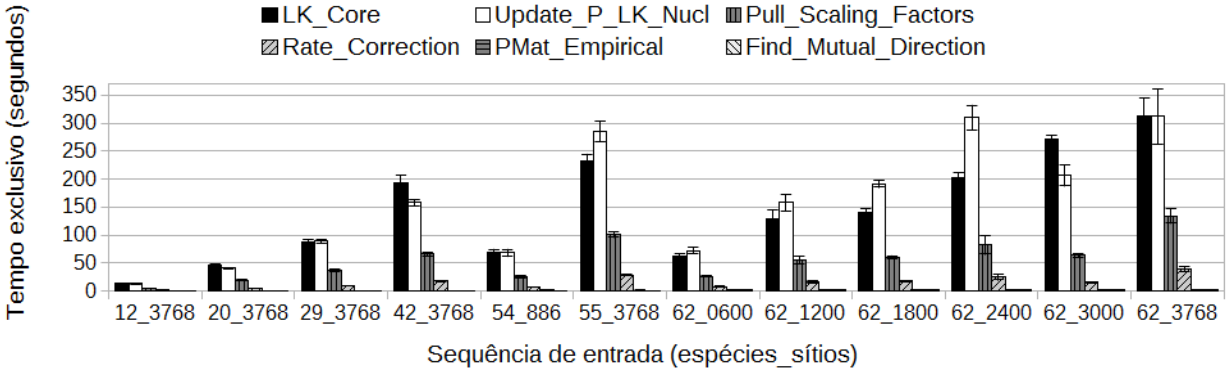

Figura 16 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando pequenos arquivos de sequências de DNA.

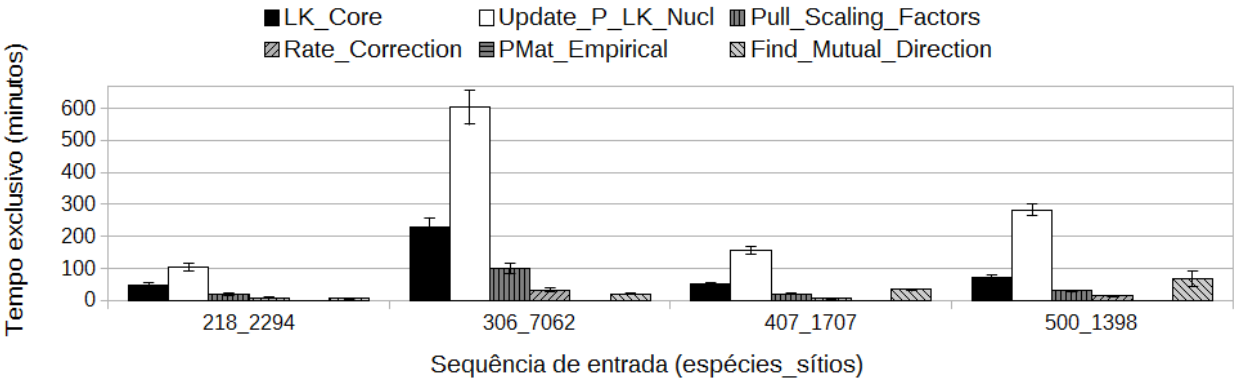

Figura 17 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando grandes arquivos de sequências de DNA.

Para pequenos e grandes arquivos de sequências de DNA, conforme ilustrações representadas na Figura 16 e Figura 17 acima, as funções *LK_Core*, *Update_P_LK_Nucl*, *Pull_Scaling_Factors* e a função *Rate_Correction* se mostram como os principais gargalos do PhyML, mas que, em alguns casos, com grandes arquivos de sequências, a função *Find_Mutual_Direction* se mostrou como um possível gargalo. Já a função *PMat_Empirical* é irrelevante com sequências de DNA, independente do tamanho do arquivo de entrada.

As ilustrações da Figura 18 e Figura 19 a seguir, mostram de acordo com o GProf, o percentual de tempo de execução de cada função para, respectivamente, pequenos e grandes arquivos de sequências de DNA. Através desses valores percentuais em relação ao tempo de execução total do PhyML é possível analisar de forma mais criteriosa a representatividade do impacto do tempo de execução de

cada função gargalo, tanto em relação ao tempo total de execução do PhyML, quanto em relação às demais funções gargalos.

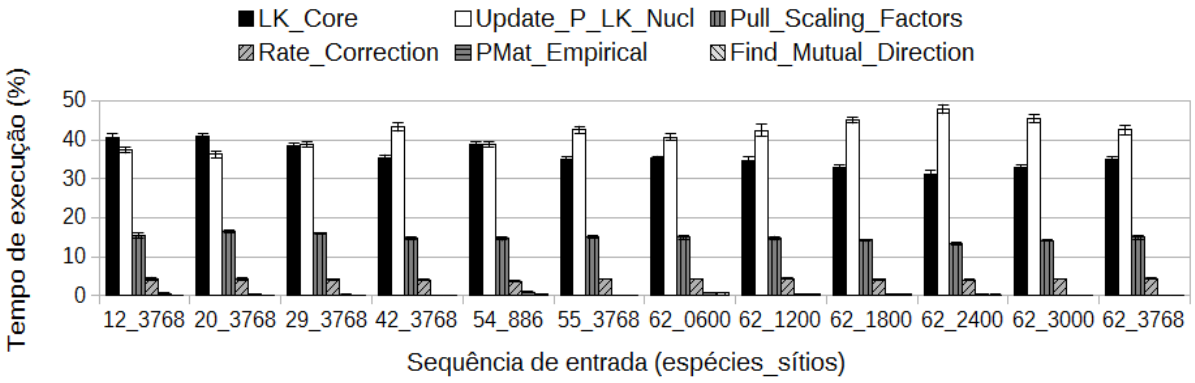

Figura 18 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando pequenos arquivos de sequências de DNA.

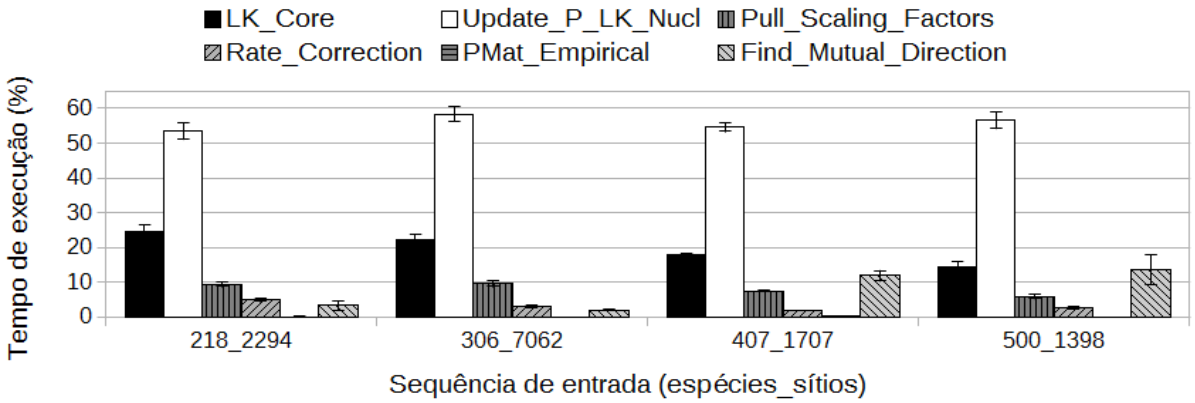

Figura 19 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando grandes arquivos de sequências de DNA.

Os valores ilustrados na Figura 18 acima revelam que para pequenos arquivos de sequências de DNA, as funções *LK_Core* e *Update_P_LK_Nucl* apresentaram o maior percentual de tempo de execução do PhyML, sendo que, na maioria dos casos, a função *Update_P_LK_Nucl* exigiu o maior percentual de tempo de execução em relação às demais. Em seguida, as funções *Pull_Scaling_Factors* e *Rate_Correction,* respectivamente, apresentaram o terceiro e quarto maior percentual de tempo de execução em relação às demais funções, enquanto as funções *Find_Mutual_Direction* e *PMat_Empirical* não mostraram percentual de

tempo de execução relevantes em relação às demais. Já com relação à grandes arquivos de sequências de DNA, Figura 19 acima, a função *Update_P_LK_Nucl* continua mantendo a maior representatividade, enquanto a função *LK_Core* perde importância em relação aos pequenos arquivos de sequências de DNA. Em contrapartida, a função *Find_Mutual_Direction* ganha importância em relação aos pequenos arquivos de sequências de DNA, chegando, em alguns casos, a consumir mais tempo do que a função *Pull_Scaling_Factors* (de 10 a 15 por cento do tempo total do PhyML). Da mesma forma ocorrida com pequenos arquivos de sequências de DNA, a função *PMat_Empirical* se mostrou irrelevante para grandes arquivos de sequências de DNA.

Já para sequências de proteínas, de acordo com o GProf, os tempos exclusivos cumulativos de cada função ranqueada são ilustrados na Figura 20 e Figura 21 a seguir.

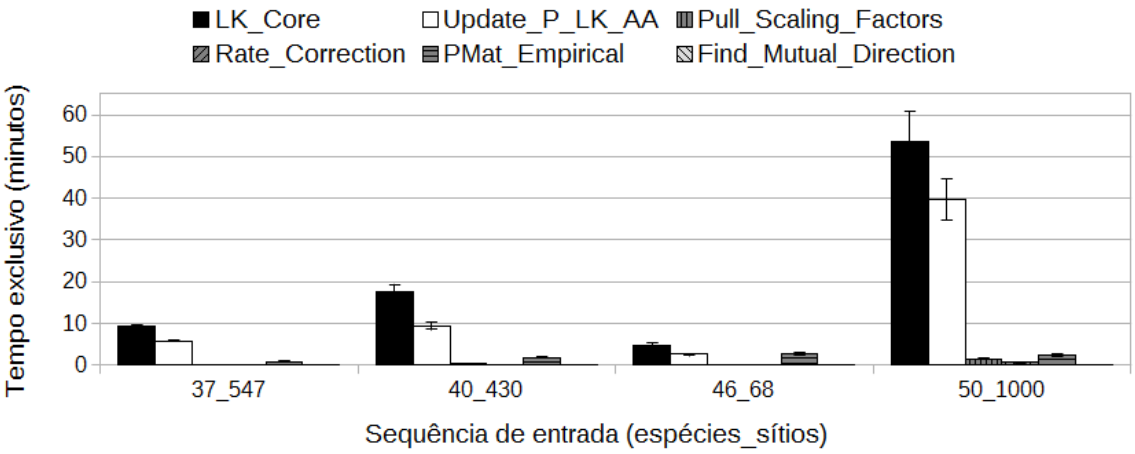

Figura 20 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando pequenos arquivos de sequências de proteínas.

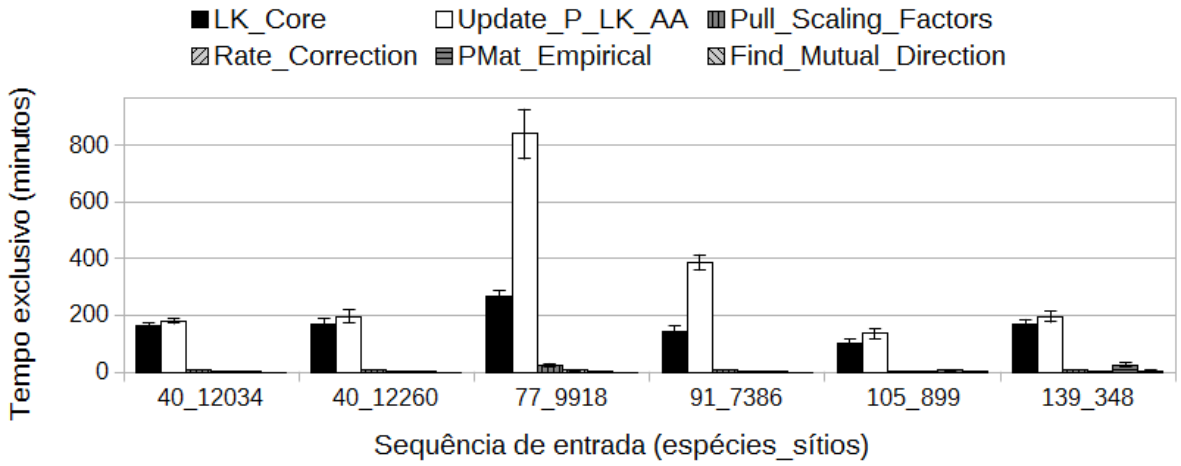

Figura 21 - Classificação das funções gargalos do PhyML de acordo com o tempo exclusivo cumulativo usando grandes arquivos de sequências de proteínas.

É possível notar através da Figura 20 e Figura 21, que as funções gargalos são: *LK_Core*, *Update_P_LK_AA*, *Pull_Scaling_Factors*, *Rate_Correction* e, diferentemente do ocorrido com sequências de DNA, a função *PMat_Empirical* mostra tanta relevância quanto as funções *Pull_Scaling_Factors* e *Rate_Correction* e, em muitos casos até mais relevância que estas. Já a função *Find_Mutual_Direction* se mostra irrelevante em termos de tempo de execução quando se tratando de dados de proteínas.

As ilustrações da Figura 22 e Figura 23 a seguir mostram o percentual de tempo de execução em relação ao tempo total do PhyML com, respectivamente, pequenos e grandes arquivos de sequências de proteínas.

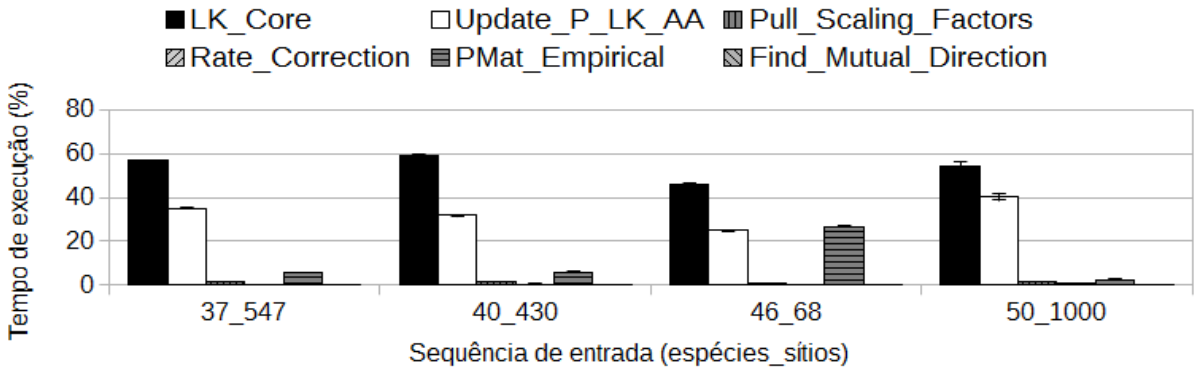

Figura 22 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando pequenos arquivos de sequências de proteínas.

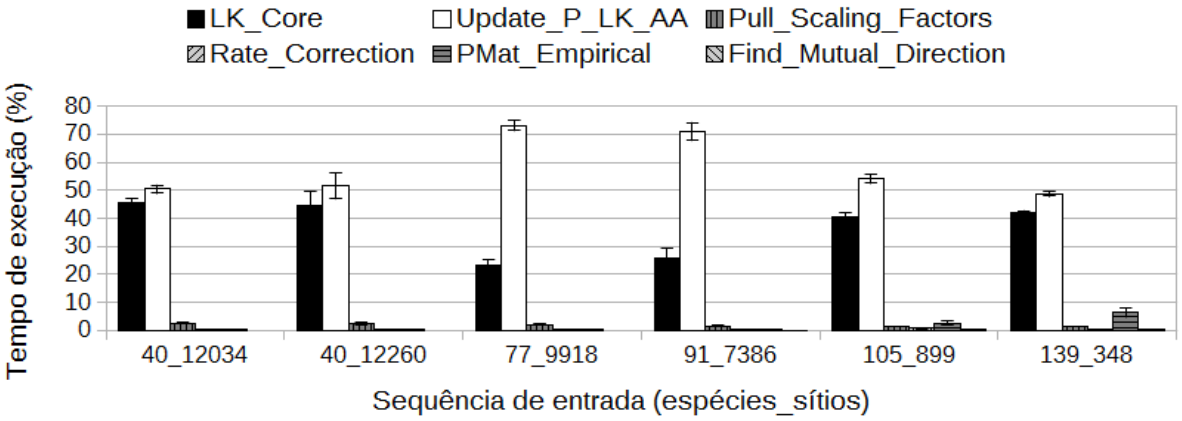

Figura 23 - Classificação das funções gargalos do PhyML de acordo com o percentual de tempo de execução usando pequenos arquivos de sequências de proteínas.

Da mesma forma ocorrida com sequências de DNA, com sequências de proteínas, as funções *LK_Core* e *Update_P_LK_AA* apresentaram o maior percentual de tempo de execução do PhyML, em que, para alguns casos a função *Update_P_LK_AA* chegou a consumir mais de 50% do tempo de execução, em seguida, as funções *PMat_Empirical, Pull_Scaling_Factors* e *Rate_Correction* complementaram o ranque de funções que mais consomem o tempo de execução. Com relação ao percetual de tempo de execução do PhyML com arquivos de sequências de proteínas, a função *Find_Mutual_Direction* mostra ser irrelevante.

Os tempos cumulativos resultantes do GProf ilustrados na Figura 16, Figura 17, Figura 20 e Figura 21 estão relacionados com o tempo exclusivo cumulativo, isto é, sem considerar o fluxo de execução. No entanto, para realizar análises mais direcionadas aos objetivos desse trabalho, há a necessidade de computar os tempos médios e cumulativos das funções já ranqueadas pelo Gprof, porém, considerando o fluxo de execução. Neste sentido, considere: (a) tempo cumulativo: somatória dos tempos de execuções de cada vez que a função foi chamada e (b) tempo médio: tempo cumulativo dividido pela quantidade de chamadas da função, significando o tempo de execução por chamada. As métricas de tempo médio serão úteis para a análise de viabilidade de paralelização das funções gargalos. Por outro lado, ambas as métricas, além de fornecerem valores mais precisos, poderão ser úteis posteriormente para análise de desempenho com relação ao comportmaneto individual de cada função.

Os tempos médios e cumulativos de cada função com diferentes arquivos de DNA e proteínas foram obtidos com a estrutura *timespec* usando precisão de até nanosegundos. Os valores ilustrados nessa subseção (4.2 Detecção de funções gargalos) para cada função são referentes à média sobre 20 execuções do mesmo arquivo de entrada usando os mesmo parâmetros de execução. A Tabela 3 a seguir mostra, de acordo com a estrutura *timespec*, os tempos médios das funções ranqueadas com arquivos de sequências de DNA.

Tabela 3 - Tempo médio de execução de cada função gargalo medido com a estrutura *timespec* para arquivos de sequências de DNA.

	Segundos					
	Update_P_ LK_Nucl	*LK_Core*	*Pull_Scaling _Factors*	*Rate_Cor-rection*	*PMat_Empi-rical*	*Find_Mutual_ Direction*
12_3768	2,46E-4	6,82E-4	5,66E-7	5,04E-8	6,02E-7	3,68E-5
20_3768	4,20E-4	1,18E-3	5,74E-7	5,06E-8	6,31E-7	2,11E-4
29_3768	5,30E-4	1,46E-3	5,71E-7	5,14E-8	6,85E-7	7,35E-4
42_3768	6,03E-4	1,61E-3	5,83E-7	5,24E-8	7,80E-7	2,92E-3
54_886	1,03E-4	2,56E-4	5,20E-7	5,13E-8	6,39E-7	7,72E-3
55_3768	6,46E-4	1,80E-3	5,89E-7	5,37E-8	8,06E-7	8,03E-3
62_0600	1,13E-4	3,00E-4	5,51E-7	5,42E-8	6,52E-7	1,25E-2
62_1200	2,26E-4	6,04E-4	5,94E-7	5,41E-8	6,94E-7	1,24E-2
62_1800	3,31E-4	9,14E-4	5,80E-7	5,43E-8	7,32E-7	1,24E-2
62_2400	4,43E-4	1,19E-3	5,79E-7	5,46E-8	7,46E-7	1,24E-2
62_3000	5,39E-4	1,50E-3	5,92E-7	5,42E-8	7,86E-7	1,24E-2
62_3768	6,77E-4	1,91E-3	5,95E-7	5,41E-8	8,03E-7	1,24E-2
218_2294	5,82E-4	1,61E-3	6,78E-7	6,15E-8	7,99E-7	1,26E+0
306_7062	2,05E-3	5,47E-3	6,65E-7	5,50E-8	1,03E-6	4,62E+0
346_897	1,62E-4	3,79E-4	4,27E-7	5,10E-8	7,35E-7	7,38E+0
407_1707	2,65E-4	6,58E-4	4,73E-7	5,20E-8	7,15E-7	1,38E+1
500_1398	3,58E-4	9,52E-4	6,04E-7	6,13E-8	7,55E-7	3,07E+1

Notoriamente (Tabela 3), os tempos médios de todas as funções, são relativamente baixos variando de microssegundos a milissegundos para as funções *LK_Core* e *Update_P_LK_Nucl*. Já para as funções *Pull_Scaling_Factors*, *Rate_Correction* e *PMat_Empirical*, o tempo médio varia em termos de nanosegundos. A função *Find_Mutual_Direction* mostra ser a função com maior tempo médio de execução por chamada, variando de microssegundos até segundos.

Já o tempo cumulativo de execução destas funções para, respectivamente, pequenos e grandes arquivos de sequências de DNA são ilustrados na Figura 24 e Figura 25 a seguir.

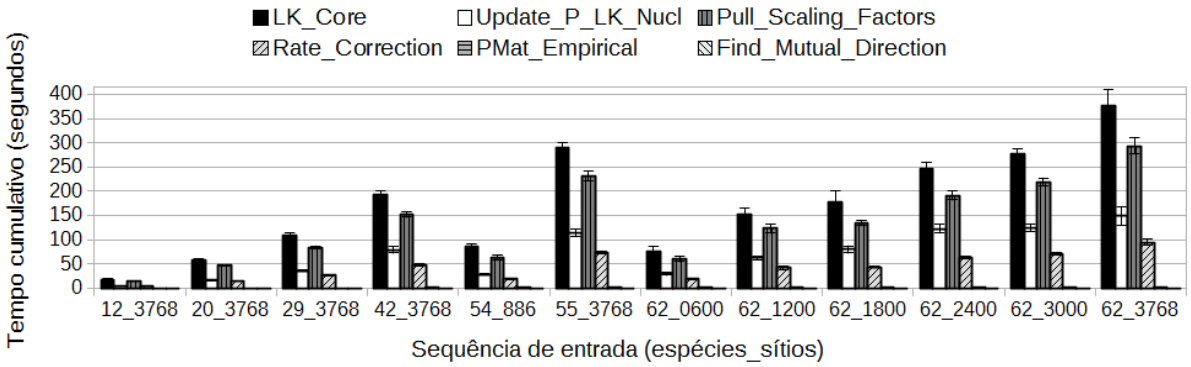

Figura 24 - Tempo cumulativo para pequenos arquivos de sequências de DNA.

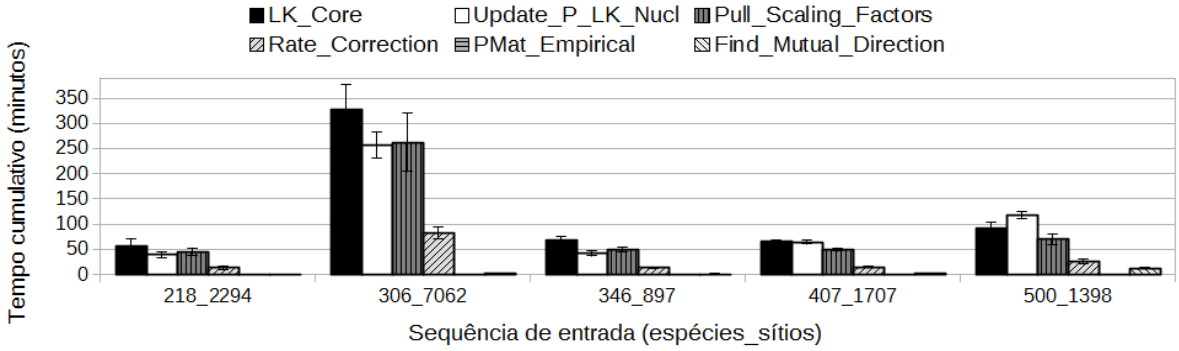

Figura 25 – Tempo cumulativo para grandes arquivos de sequências de DNA.

Os tempos cumulativos ilustrados na Figura 24 e Figura 25 acima indicam que apenas as funções *LK_Core*, *Update_P_LK_Nucl*, *Pull_Scaling_Factors* e *Rate_Correction* se mostraram custosas computacionalmente para arquivos de sequências de DNA. A função *PMat_Empirical* possui tempo cumulativo relativamente inferior em relação às demais. Já a função *Find_Mutual_Direction* mostra um tempo relativo para algumas sequências grandes de DNA.

Da mesma forma, usando arquivos de sequências de proteínas, os tempos médios e cumulativos (considerando os tempos das sub-rotinas) foram computados através da estrutura *timespec* para as funções ranqueadas pelo GProf com arquivos de sequências de proteínas. Neste sentido, a Tabela 4 a seguir mostra o tempo

médio para cada função ranqueada pelo GProf com arquivos de sequências de proteínas.

Tabela 4 - Tempo médio de execução de cada função gargalo medido com a estrutura *timespec* para arquivos de sequências de proteínas.

	Segundos					
	Update_P_ LK_AA	*LK_Core*	*Pull_Scaling_ Factors*	*Rate_Cor- rection*	*Find_Mutual_ Direction*	*PMat_Empi- rical*
37_547	1,04E-3	8,14E-4	5,53E-7	5,33E-8	1,80E-3	1,23E-5
40_12034	3,24E-2	2,74E-2	6,58E-7	5,41E-8	2,63E-3	1,51E-5
40_12260	3,32E-2	2,83E-2	6,60E-7	5,32E-8	2,67E-3	1,60E-5
40_430	1,00E-3	7,42E-4	5,46E-7	5,83E-8	2,49E-3	1,26E-5
46_68	1,55E-4	1,10E-4	4,82E-7	5,52E-8	4,32E-3	1,23E-5
50_1000	2,85E-3	2,11E-3	6,03E-7	5,80E-8	5,96E-3	1,31E-5
77_9918	3,00E-2	2,43E-2	6,47E-7	5,48E-8	2,76E-2	1,58E-5

Note que (de acordo com a Tabela 4), para sequências de proteínas, os tempos médios de execução das funções ranqueadas não chegam nem à unidade de segundos, variando em termos de: microssegundos a milissegundos para as funções *LK_Core* e *Update_P_LK_AA*; nanosegundos para as funções *Pull_Scaling_Factors* e *Rate_Correction*; microssegundos para a função *PMat_Empirical* e; milissegundos para a função *Find_Mutual_Direction*.

Já os tempos cumulativos das funções mostrados na Tabela 4, para, respectivamente, pequenos e grandes arquivos de sequências de proteínas, são ilustrados na Figura 26 e Figura 27 a seguir.

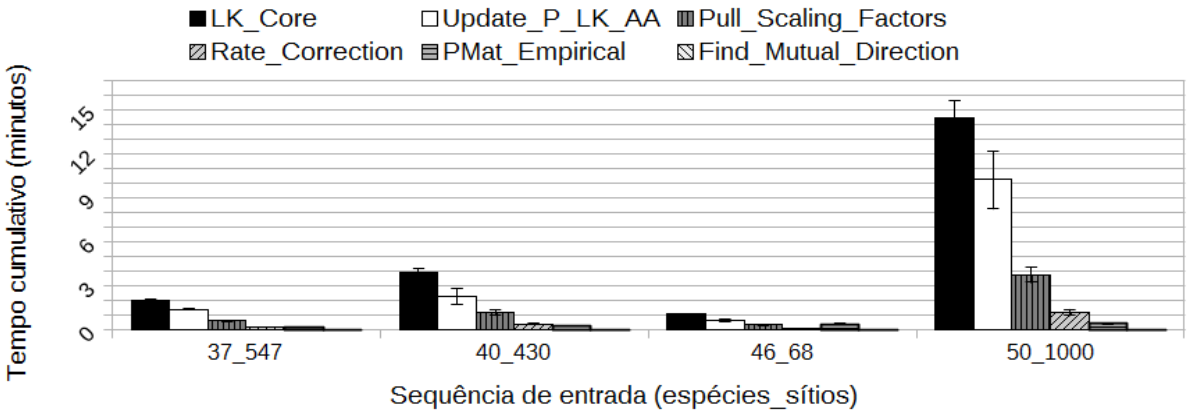

Figura 26 - Tempo cumulativo para pequenos arquivos de sequências de proteínas.

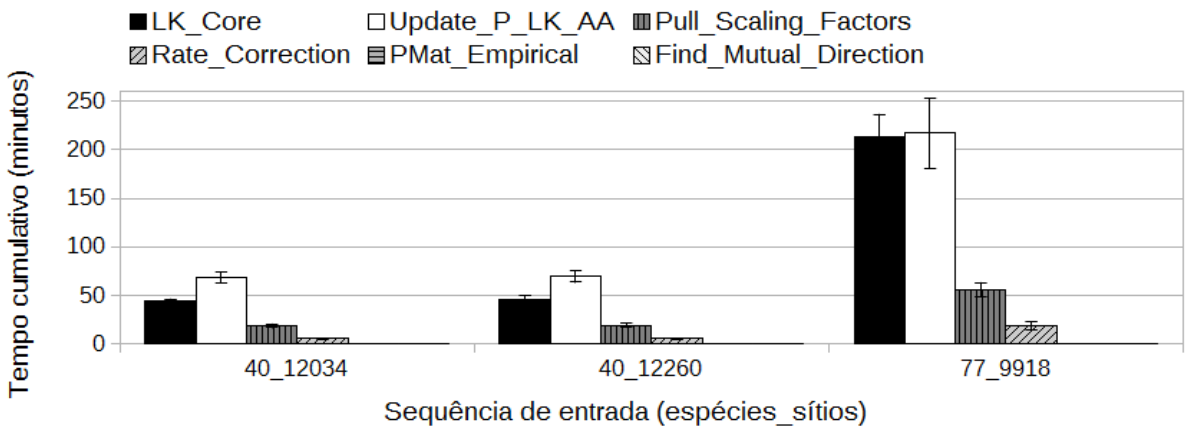

Figura 27 - Tempo cumulativo para grandes arquivos de sequências de proteínas.

Os tempos cumulativos mostrados na Figura 26 e Figura 27 ilustrados acima indicam que apenas as funções *LK_Core*, *Update_P_LK_AA*, *Pull_Scaling_Factors* e *Rate_Correction* se mostraram custosas computacionalmente para sequências de proteínas. Em casos raros a função *PMat_Empirical* possui um tempo de execução significativo, mas que não se mostra relevante quando relacionado ao tempo de execução das outras funções. Já a função *Find_Mutual_Direction* tem tempo de execução cumulativo baixíssimo e pode ser descartada como função gargalo para sequências de proteínas.

Note que apesar das funções apresentarem tempo médio de execução relativamente baixo tanto com arquivos de sequências de DNA quanto com arquivos de sequências de proteínas, a maioria destas se mostram custosas computacionalmente em termos de tempo cumulativo de execução ou mesmo em termos de percentual de tempo de execução do PhyML, indicando que o alto custo de processamento destas funções não se encontra exatamente no tempo de processamento do bloco, mas na exaustiva quantidade de vezes que essa função ou bloco de execução é chamado.

Estas análises sobre sequências de DNA e proteínas indicam que existem 5 funções gargalos no PhyML sendo elas: *LK_Core*, *Update_P_LK_Nucl, Update_P_LK_AA*, *Pull_Scaling_Factors* e *Rate_Correction.* A função *Find_Mutual_Direction* mostrou um comportamento inesperado com aumento de tempo cumulativo e percentual de tempo de execução para algumas sequências de DNA enquanto a função *PMat_Empirical* mostrou o mesmo comportamento para

sequências de proteínas. Por precaução, estas duas funções também serão estudadas de forma mais aprofundada para investigar a viabilidade ou não de paralelização, considerando *Find_Mutual_Direction* para sequências de DNA e *PMat_Empirical* para sequências de proteínas.

4.3 Viabilidade de paralização de funções gargalos

Nesta subseção serão abordadas algumas características da ferramenta de benchmark EPCC Microbenchmark, bem como a sua utilização como apoio na medição de *overhead*s de diretivas OpenMP e posterior análise de viabilidade de paralelização das funções gargalos.

4.3.1 *Overhead* das diretivas OpenMP com a ferramenta EPCC Microbenchmark

Em OpenMP, um tempo de execução está associado com a criação de regiões paralelas, com a divisão da carga de trabalho sobre as threads, com os tipos de sincronização associados e com dados privados das threads (CHAPMAN; JOST; PAS, 2007). Neste sentido, Bull (1999), Bull e O´Neill (2001) e Chapman, Jost e Pas (2007) afirmam que o *overhead* de uma aplicação OpenMP depende da estratégia de tradução do compilador, da arquitetura, das características das rotinas da biblioteca (em tempo de execução), da maneira que o compilador otimiza o código, das necessidades da aplicação em termos de diretivas de paralelização e do tempo de execução do bloco a ser paralelizado.

O EPCC Microbenchmark é uma ferramenta de benchmark utilizada na otimização de aplicações paralelizadas com OpenMP e foi criada para auxiliar programadores a estimar o tempo de execução relativo de diferentes construções usando OpenMP e, indicar a granularidade de tarefas, para as quais é desejável evitar *overhead*s excessivos com a geração e execução dessas tarefas em paralelo (BULL; O´NEILL, 2001; BULL; REID, 2012).

A técnica básica de funcionamento do EPCC é comparar o tempo médio de execução de um código serial com o tempo médio de execução do mesmo código em paralelo com uma diretiva OpenMP específica (BULL; O´NEILL, 2001). Para entender melhor o funcionamento desse benchmark, será explicado o seu trecho de código usada para medir o *overhead* da diretiva *"pragma omp for"*, para isso, considere a Figura 28 a seguir.

```
102 void refer() {
103     int j;
104     for (j = 0; j < innerreps; j++) {
105     delay(delaylength);
106     }
107 }
```

(a)

```
143 void testfor() {
144     int i, j;
145 #pragma omp parallel private(j)
146     {
147     for (j = 0; j < innerreps; j++) {
148 #pragma omp for
149         for (i = 0; i < nthreads; i++)
150             delay(delaylength);
151             }
152         }
153     }
154 }
```

(b)

Figura 28 - Trecho da implementação do EPCC Microbenchmark usado para medir o *overhead* da diretiva "*pragma omp for*" em que (a) é a versão serial do benchmark e (b) é a versão paralela em relação à versão serial.

Na Figura 28 são apresentados dois trechos de código implementados pelo benchmark EPCC. Ambos os trechos são utilizados para determinar o *overhead* da diretiva *"pragma omp for"*. O trecho (a) é o código serial utilizado como referência para a comparação de tempos de execução entre a versão serial e paralela. Já o trecho (b) é o código paralelo usado para medir o overhead da diretiva *"pragma omp for"*. No trecho (a) dois pontos são importantes; (I) laço de repetição da linha 104 - usado para obter a média de tempos de execução, em que a variável *innerreps* representa a quantidade de repetições usada para obter essa média e (II) função *delay(delaylength)* na linha 105 - é uma função de espera em que o tempo de espera *delaylength* é o tempo médio de execução da função que se deseja paralelizar com OpenMP na aplicação, o valor da variável *delaylength* é fornecido pelo usuário, o que permite simular no EPCC cada uma das funções que se deseja paralelizar através do seu tempo médio de execução. Já no trecho de código paralelo (b), ocorrem três diferenças relevantes em relação ao trecho serial (a), são elas: (I) linha 145 - criação da região paralela, note que esse trecho encontra-se fora

do laço de repetição que é responsável pela média de tempos para comparação com o trecho da versão serial (a), ou seja, não ocasiona interferências em relação à versão serial; (II) linha 148 - inserção da diretiva que se deseja medir o *overhead* (*pragma omp for*, por exemplo), nesse caso dentro do laço de repetição que contabiliza a média de tempo de execução (*innerreps*) e; por fim, (III) linha 149 - divisão do trabalho sobre *nthreads* (a quantidade de threads *nthreads* é fornecida pelo usuário) para que todas as threads executem o mesmo trecho da versão serial e permita uma comparação justa de tempos de execução. Desta forma, para um dado programa paralelo, seja T_p o tempo de execução do programa com p processadores e T_s o tempo de execução serial do mesmo programa, define-se o tempo de *overhead* do programa paralelo (T_o) de acordo com a expressão a seguir.

$$T_o = T_p - {}^{T_s}/_{p} \tag{35}$$

Da mesma forma que foi criado um trecho de código para mensurar o *overhead* da diretiva *"pragma omp for"* (mostrado na Figura 28), o benchmark EPCC implementa um trecho de código para cada diretiva OpenMP, considerando as diretivas dos três quesitos (sincronização, agendamento e paralelismo de tarefas). Cada trecho paralelo tem um trecho serial equivalente sendo usado como referência para medir o *overhead*. Através do tempo de espera *delaylength* (tempo médio de execução da função que se deseja paralelizar), o usuário pode simular no benchmark EPCC (estimativamente, já que não é o comportamento real e exato da função) qualquer função e analisar através do seu tempo médio de execução o *overhead* das diretivas de paralelização a serem utilizadas. Para cada tempo de entrada (*delaylength*), o benchmark EPCC fornece *overhed* para todas as diretivas implementadas e o usuário seleciona nos resultados as diretivas de seu interesse para analisar o *overhead*, considerando a medida de tempos médio de execução (*innerreps*) e o desvio padrão associado.

Apenas para efeito ilustrativo, pois não é de autoria deste trabalho, bem como tal simulação não foi realizada sobre condições específicas dessa pesquisa (como arquitetura e o tempo de espera, *delaylength*), a Figura 29 ilustra o comportamento do *overhead* de algumas diretivas quando aumentado o número de threads.

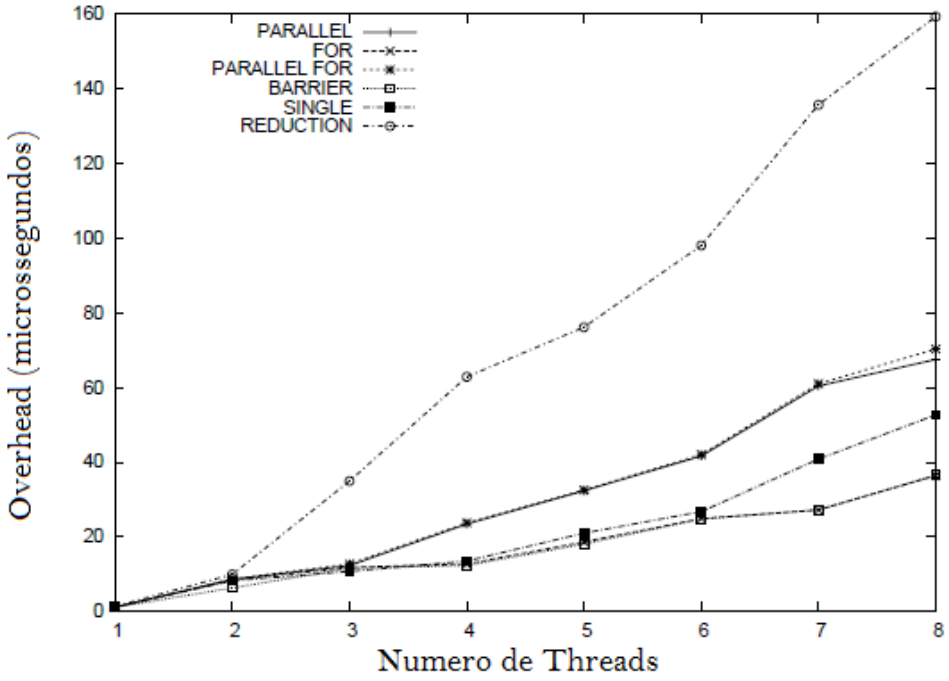

Figura 29 - *Overheads* ilustrativo de algumas diretivas OpenMP: o *overhead* das diretivas comuns e construtores é dado em microssegundos à media que se aumenta o número de threads.
Fonte: adaptado de (CHAPMAN; JOST; PAS, 2007).

Através da Figura 29, pode-se notar que é possível definir a viabilidade de paralelização ou não de uma função em relação ao *overhead* causado por determinada diretiva de paraleização, considernado o tempo médio de execução de uma função. Da mesma forma, pode-se notar que através dessa análise de *overhed* é possível decidir qual a diretiva mais adequada (menor *overhead* gerado) em termos de otimização para a utilização em alguma função, quando é possível trabalhar com alternativas.

4.3.2 Viabilidade de paralização das funções gargalos do PhyML

O tempo médio de execução de cada função gargalo tanto para sequências de DNA quanto para sequências de proteínas (Tabela 3 e Tabela 4, respectivamente - subseção 4.2.2 Funções gargalos do PhyML) na maioria dos casos, variou de nanosegundos à milissegundos. Neste sentido, com relação ao critério de

granularidade abordado anteriormente na subseção 2.2.2 (Paradigma de memória compartilhada OpenMP), foi feita uma análise geral do código de cada função gargalo para entender a sua essência e, desta forma, prever as possíveis diretivas que cada função exigiria na paralelização. A ideia é através dessa previsão e do tempo médio de execução de cada função gargalo (tempo de espera *delaylength* no EPCC), investigar através da ferramenta de benchmark EPCC se o *overhead* das diretivas de paralelização a serem utilizadas não excederia o tempo médio de execução da função a ser paralelizada. Um excedente do *overhead* (simulando a iserção da(s) diretiva(s) de paralelização no código) em relação ao tempo médio de execução do gargalo inviabiliza a paralelização dessa função, isto é, o tempo médio de inserção da diretiva é maior que o tempo médio de execução da própria função a ser paralelizada.

É importante salientar que essa é uma análise preliminar sobre as funções gargalos, isto é, uma análise geral que levou em consideração apenas a heurística de cada função, pois, ainda não se sabe da existência de possíveis fortes acoplamentos entre tarefas causados pelas estratégias de programação empregada na construção da função. Por este motivo, na maioria dos casos, *#pragma omp parallel for* foi a única diretiva de paralização previsível, com exceção da função *LK_Core*, para a qual foi possível perceber, de imediato, que esta necessitaria da cláusula *reduction*. Do ponto de vista de eficiência no uso das diretivas de paralelização, como já esperado, foi confirmado (resultados não mostrados) que para os tempos médios de execução de todas as funções gargalos consideradas, em se tratando de cláusulas relacionadas a balanço de carga, a cláusula *static* tem menor *overhead* do que as cláusulas *dynamic* e *guided*. Foi possível confirmar ainda que há um menor *overhead* em usar o construtor combinado *#pragma omp parallel for* (abrindo a região paralela e ao mesmo tempo distribuindo o trabalho), ao invés de usar o construtor paralelo fundamental *#pragma omp parallel* e depois o construtor de divisão de trabalho *#pragma omp for* (primeiro abrindo a região paralela para depois distribuir o trabalho). Por esta razão foi usado em todos os casos o construtor combinado *#pragma omp parallel for*. Além disso, a cláusula de divisão de trabalho *schedule* foi omitida, pois, por padrão, essa divisão é feita usando a cláusula *static*, que já possui o menor *overhead* sobre o tempo médio de execução das funções gargalos, como mencionado anteriormente.

Para medir o *overhead* das diretivas OpenMP com o benchmark EPCC é necessário usar o tempo médio de execução de cada função gargalo (*delaylength*). No entanto, foi testada uma grande quantidade de arquivos de sequências de entrada. Na maioria das funções, o seu tempo médio de execução varia de acordo com o tamanho do arquivo de sequências de entrada. Neste sentido, adotou-se duas estratégias para que o tempo médio de execução a ser usado no benchmark EPCC possa representar a maioria dos casos de tempos médios ocorridos para a função considerada. A primeira estratégia é usar os valores extremos do intervalo de tempos médios que a função gargalo assumiu, porém, essa estratégia somente é adequada quando a função não apresentar grande variabilidade em seus tempos médio de execução (se mantendo em termos de nanosegundos ou microssegundos ou milisegundos, por exemplo). Já a segunda estratégia está relacionada aos casos em que a função apresenta grande variabilidade de tempos médios de execução (variando de nanosegundos à milissegundos, por exemplo), pois nesses casos usar o intervalo não representa fielmente os tempos médios de execução da função, desta forma, adotou-se um parametro estatístico em que a ideia é usar os valores extremos do intervalo onde ocorre a maior concentração ou maior distribuição de tempos médios de execução. A intenção em adotar duas estratégias é cobrir todo o intervalo para funções gargalos que tenham tempo médio de execução variando em curto intervalo de tempo ou, no mínimo, abranger a área em que haja maior concentração ou distribuição de tempos médios de execução, isto é, o intervalo em que houve maior concentração de tempo médio de execução para determinada função.

As ilustrações do escopo deste trabalho que correlacionam o tempo médio de execução da função gargalo com o *overhead* de inserção de diretivas são resultantes de simulações com a ferramenta EPCC Microbenchmark. Neste sentido, a Figura 30 a seguir ilustra o *overhead* das diretivas *#pragma omp parallel for* juntamente com a cláusula *reduction* com tempo médio de execução (tempo de espera *delaylength* no EPCC) igual a 100 e 500 microssegundos, representando o tempo médio de execução da função *LK_Core* para sequências de DNA. Note que nos dois casos (100 e 500 microssegundos), somando os valores de *overhead* de ambas as diretivas obtêm-se um *overhead* menor que 12 microssegundos com até 8 threads, sendo este menor que os tempos médios de execução da função *LK_Core*

(100 e 500 microssegundos), indicando que não há *overhead* maior que o tempo médio de execução da função *LK_Core* e que pode haver ganho em sua paralelização.

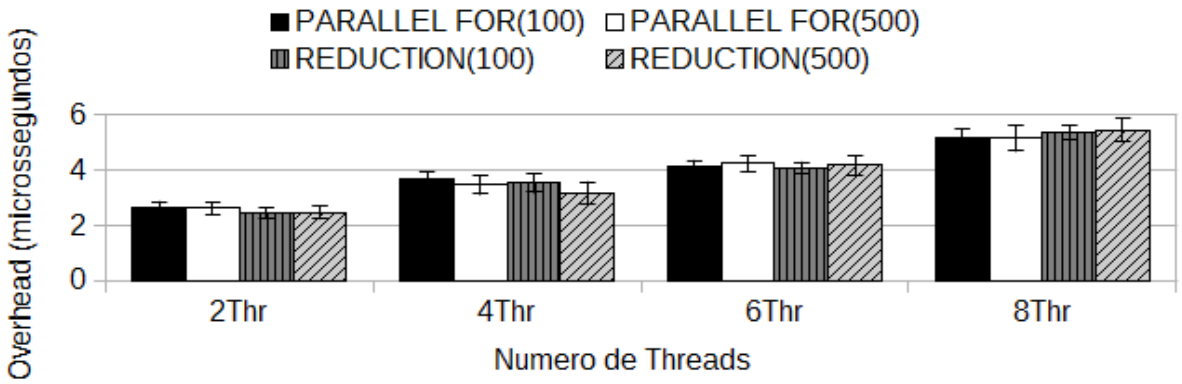

Figura 30 - *Overhead* das principais diretivas previsíveis para o bloco de trabalho da função *LK_Core* com tempo médio de execução igual a 100 e 500 microssegundos para sequências de DNA.

Ainda com relação à Figura 30, pode-se usar somente o *overhead* da diretiva *#pragma omp parallel for* com tempo médio de execução igual a 100 e 500 microssegundos para analisar também o overhead sobre as funções *Update_P_LK_Nucl* e *Find_Mutual_Direction*, pois ambas também apresentam esse mesmo tempo médio de execução e previsivelmente usarão somete essa diretiva. Nota-se que o *overhead* dessa diretiva é aproximadamente no máximo 6 microssegundos, indicando que, assim como a função *LK_Core*, há a possibilidade de ganho também na paralelização de ambas das funções *Update_P_LK_Nucl* e *Find_Mutual_Direction* com esse tempo médio de execução (tempo de espera *delaylength* no EPCC).

Para sequências de DNA, a função *Pull_Scaling_Factors* tem intervalo de de tempo médio variando entre entre 600 e 800 nanosegundos, ao utilizar esse tempo médio de execução para medir o *overhead* com a diretiva *#pragma omp parallel for*, obteve-se, conforme Figura 31, um *overhead* acima de 2 microssegundos, indicando a inviabilidade de paralelização desta função em razão do tempo médio de execução ser menor que o *overhead* de inserção da diretiva de paralelização.

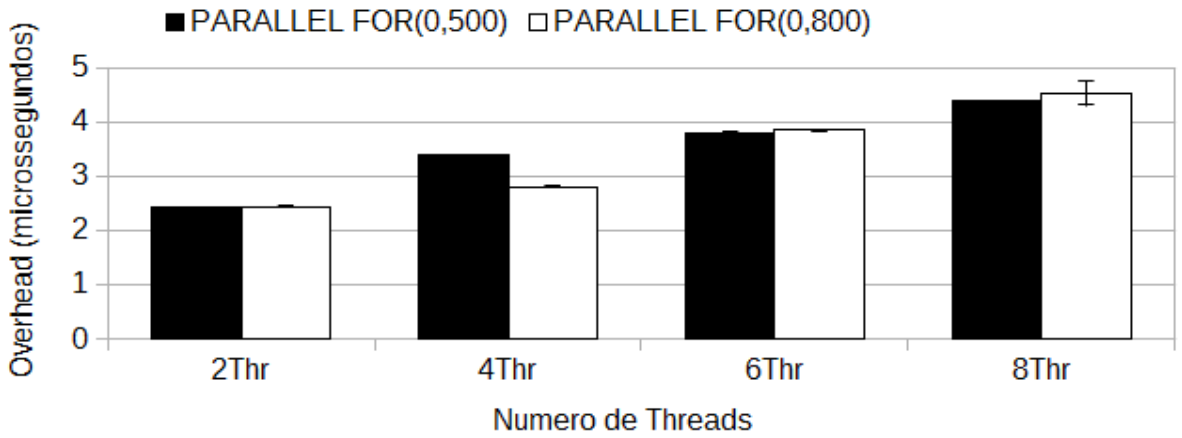

Figura 31 - *Overhead* da principal diretiva previsível para o bloco de trabalho da função *Pull_Scaling_Factors* com tempo médio de execução variando entre 500 e 800 nanosegundos para sequências de DNA.

Da mesma forma, a função *Rate_Correction* apresentou, para sequências de DNA, o intervalo de tempo médio de execução variando entre 50 e 60 nanosegundos, porém a diretiva de paralelização *#pragma omp parallel for* tem *overhead* acima de 2 microssegundos, inviabilizando também a paralelização desta função para sequências de DNA.

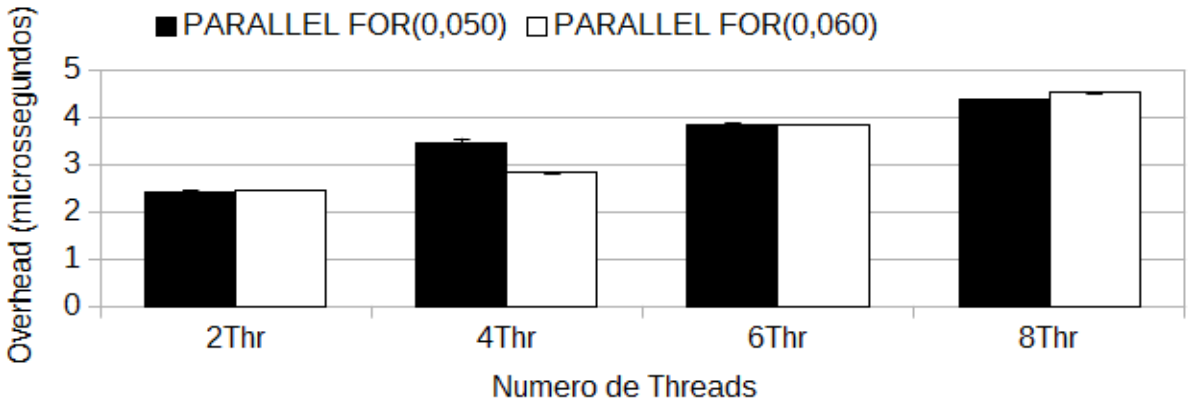

Figura 32 - *Overhead* da principal diretiva previsível para o bloco de trabalho da função *Rate_Correction* com tempo médio de excução variando entre 50 e 60 nanosegundos para sequências de DNA.

Para sequências de proteínas, a função *LK_Core* apresentou área de concentração de tempo médio de execução variando entre 110 e 600 microssegundos. De acordo com a Figura 33 a seguir, nota-se que somando os valores de *overhead* de inserção das duas diretivas previsíveis para a paralelização

desta função (*#pragma omp parallel for* com a cláusula *reduction*), esta não ultrapassaria um *overhead* de 12 microssegundos, indicando que, em relação ao *overhead* de inserção das diretivas no código desta função, pode-se ter um ganho na sua paralelização.

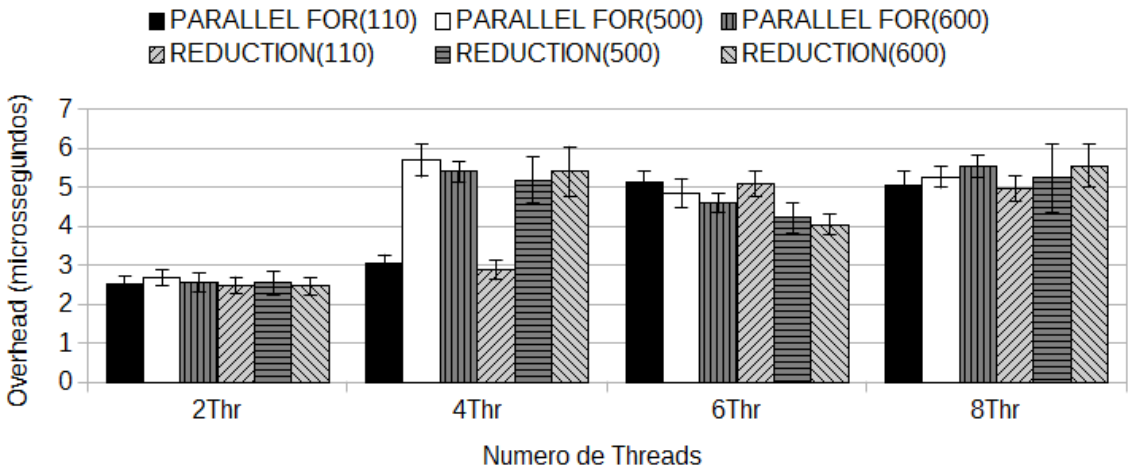

Figura 33 - *Overhead* das principais diretivas previsíveis para o bloco de trabalho da função *LK_Core* com tempo médio de execução com concentração em 110, 500 e 600 microssegundos para sequências de proteínas.

A função *Update_P_LK_AA* apresentou comportamento semelhante à função *LK_Core*, com intervalo de concentração de tempo médio de execução variando entre 150 e 600 microssegundos e o *overhead* da diretiva de paralelização *#pragma omp parallel for* chegando a no máximo 6 microssegundos, conforme Figura 34 .

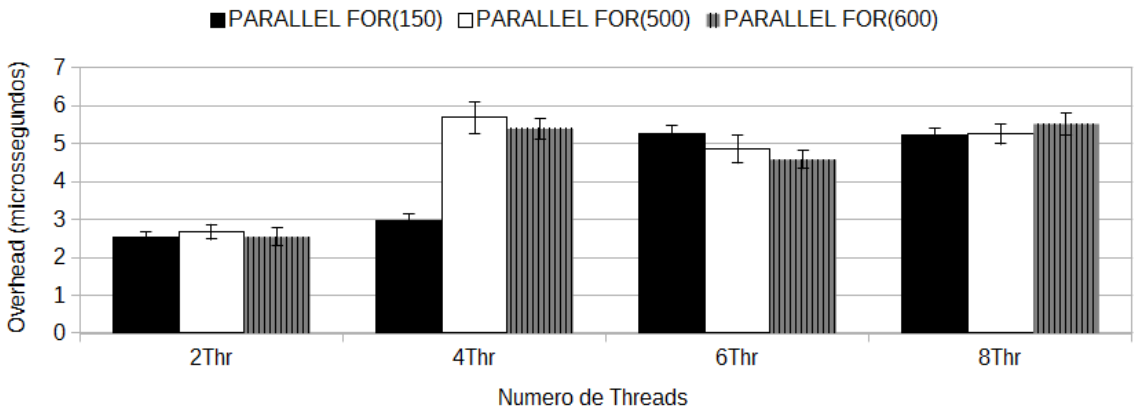

Figura 34 - *Overhead* da principal diretiva previsível para o bloco de trabalho da função *Update_P_LK_AA* com tempo médio de execução com

concentração em 150, 500 e 600 microssegundos para sequências de proteínas.

Da mesma forma ocorrida com sequências de DNA em que as funções *Pull_Scaling_Factors* e *Rate_Correction* apresentaram tempo médio de execução baixo em relação ao *overhead* da diretiva de paralelização *#pragma omp parallel for*, apresentaram o mesmo comportamento com sequências de proteínas, conforme Figura 35 e Figura 36 a seguir, indicando mais uma vez a inviabilidade de paraleliza-las.

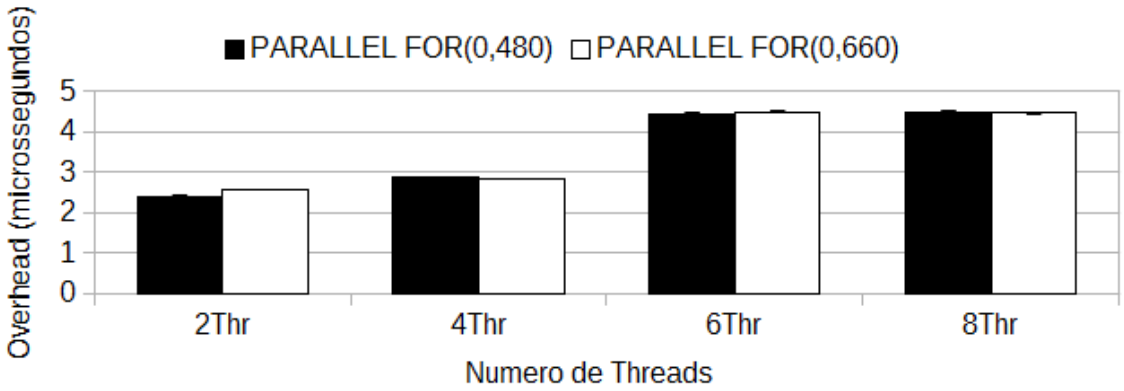

Figura 35 - *Overhead* da principal diretiva previsível para o bloco de trabalho da função *Pull_Scaling_Factors* com tempo médio de execução variando entre 480 e 660 nanosegundos para sequências de proteínas.

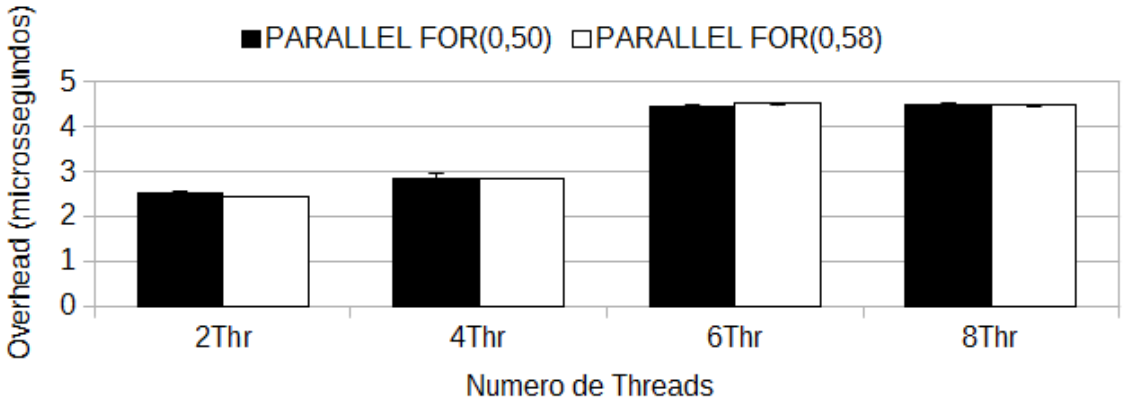

Figura 36 - *Overhead* da principal diretiva previsível para o bloco de trabalho da função *Rate_Correction* com tempo médio de execução variando entre 50 e 68 nanosegundos para sequências de proteínas.

Por fim, para sequências de proteínas, a função *PMat_Empirical* apresentou intervalo de tempo médio de execução variando entre 12 e 15 microssegundos, enquanto a diretiva *#pragma omp parallel for* previsivel à sua paralelização gera um *overhead* de 2 à 5 microssegundos, conforme Figura 37, indicando que em relação ao *overhead* de inserção da diretiva de paralelização, pode haver um ganho. É importante notar que a quantidade de microssegundos restantes a ser paralelizada em relação ao tempo médio de execução da função é pequena e que, nesse caso, o processo de sincronização pode levar a um *overhead* maior que o ganho.

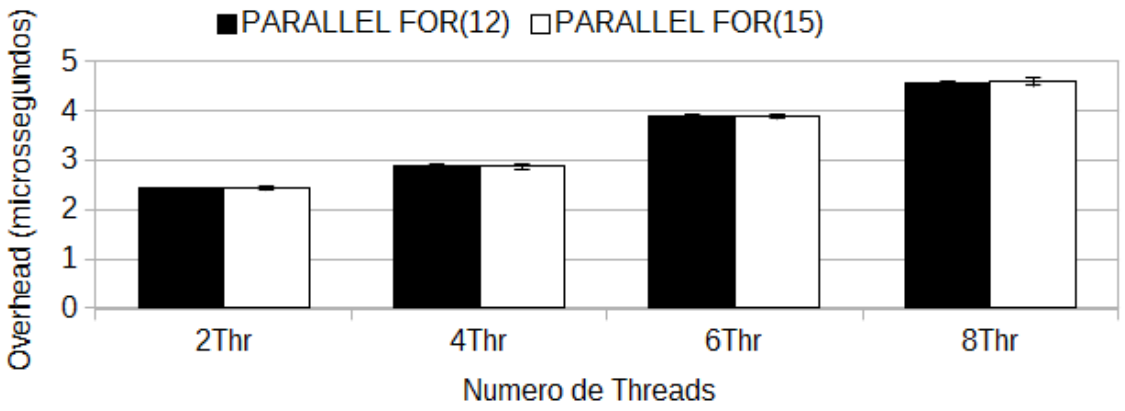

Figura 37 - *Overhead* da principal diretiva previsível para o bloco de trabalho da função *PMat_Empirical* com tempo médio de execução variando entre 12 e 15 microssegundos para sequências de proteínas.

Essa análise através do uso do benchmark EPCC Microbenchmark permitiu definir as funções a serem estudadas de forma mais detalhadas, são elas: *LK_Core* (DNA e proteínas), *Update_P_LK_Nucl* (DNA), *Update_P_LK_AA* (proteínas), *Find_Mutual_Direction* (DNA) *e PMat_Empirical* (proteínas).

4.4 Heurísticas e estratégias de paralelização das funções gargalos

Após ter encontrado as funções gargalos do PhyML (subseção anterior), iniciou-se o processo de investigar o que cada função gargalo representa dentro do código e a melhor forma de paraleliza-las, quando possível. Um requisito necessário para a realização desse processo foi a análise de todo o código do PhyML, com o

objetivo de correlacionar esse código com as heurísticas descritas nos artigos do PhyML (GUINDON et al., 2010; GUINDON; GASCUEL, 2003) e com os princípios gerais do cálculo de máxima verossimilhança e as estratégias de RAF abordadas na revisão de literatura (Seção 2). Essa correlação possibilitou o entendimento da função desempenhada por cada gargalo e facilitou estudos mais direcionados ao comportamento individual de cada um.

4.4.1 Heurística e localização dos pontos críticos do PhyML

A heurística aqui descrita supõe que a topologia inicial é construída com o método de distâncias usando o algoritmo BioNJ e que os rearranjos da topologia são efetuados através da estratégia NNI. Neste sentido, o PhyML inicialmente calcula uma matriz de distâncias com base nas sequências e modelo evolutivo e a utiliza com o método BioNJ para construir a topologia inicial. Usando a topologia inicial gerada pelo BioNJ, busca-se uma possível melhor árvore, otimizando todos os galhos da topologia (um por vez). Neste sentido, considerando a Figura 38, a otimização com NNI consiste em calcular o tamanho de determinado galho (nesse caso, considere o galho x) em função dos melhores valores de MV. Para isso, em algum galho interno qualquer (digamos x), todas as possíveis trocas dos galhos vizinhos ($b1$, $b2$, $b4$ e $b5$) são analisadas para se obter o galho (x) de melhor escore ou a possível subárvore de melhor troca, conforme ilustração da Figura 38.

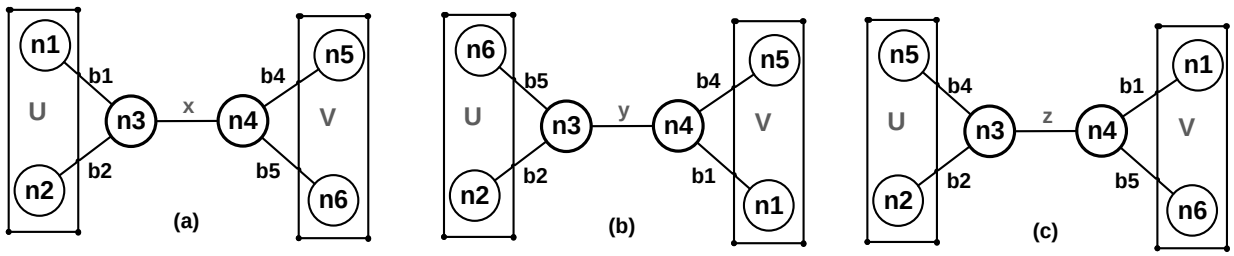

Figura 38 - As três configurações de alternativas topológicas sobre um galho interno. n1, n2, n5 e n6 são quatro subárvores, e b1, b2, b4 e b5 são os quatro galhos conectados a raiz de n1, n2, n5 e n6, respectivamente. Estes

tamanhos são os mesmos nas três configurações topológicas (a, b e c). U e V são subárvores da esquerda e direita, respectivamente, dos galhos internos, e, x, y e z são os tamanhos de galhos internos que maximizam a verossimilhança da filogenia correspondente.

Na ilustração da Figura 38, a topologia de melhor escore será a topologia em que x, y ou z possua a maior MV quando comparada às demais topologias. As análises de escores ou melhor troca com base no maior valor de MV são verificadas usando a eq. (32) sobre o galho x, porém, supondo que os valores de MV do restante da topologia (subárvores U e V, partindo de $n3$ e $n4$, respectivamente) encontram-se atualizados em função de aplicações sucessivas da eq. (34). Após obter uma lista de escores para cada galho interno, isto é, a lista com as possíveis melhores trocas para todas as subárvores da topologia, aplica-se, de fato, as melhores trocas em toda a topologia e atualiza a sua MV parcial (aplicações sucessivas da equação eq. (34)). Por fim, usando a topologia já atualizada para todo caractere dentro de cada categoria de cada sítio, obtém-se o novo valor total de MV da topologia (eq. (31)).

Em alguns casos, pode ocorrer que ao final do processo de rearranjos, o novo valor de MV seja menor que o valor obtido na execução anterior, isto é, a topologia atual é menos provável que a topologia da iteração anterior. Para reverter esse tipo de situação, usa-se a estratégia de desfazer movimentos NNI para reencontrar uma topologia com MV melhor ou equiparada à MV anterior. Neste caso, para não desfazer todos os movimentos NNI, usa-se um controlador λ (quantidade total de trocas realizadas na ultima iteração de rearranjos) que sucessivamente vai sendo dividido por 2 (nova quantidade de rearranjos a ser realizada sobre a topologia anterior) até que se encontre a topologia com MV mais próxima da máxima já encontrada. Em paralelo à otimizações da topologia, os parâmetros do modelo estão periodicamente sendo ré-estimados e os tamanhos de galhos otimizados. O processo iterativo de refinamento sobre a topologia corrente é realizado até levar à convergência, isto é, até que não haja mais trocas na topologia e a MV deixe de ser crescente, estabilizando os tamanhos de galho.

A Figura 39 a seguir ilustra um algoritmo genérico com a heurística do PhyML, do qual será desmembrada somente as partes necessárias ao entendimento das funções gargalo. Nos trechos onde alguma função gargalo é executada, esta é

identificada e listada após o termo em negrito “Gargalo” ou “Gargalos”. Por questões de simplicidade, considere “Bloco X” como um desvio para o Bloco X descrito à parte no algoritmo, o qual é executado e, ao seu término, a execução retorna ao ponto do algoritmo que havia chamado o “Bloco X”. Esses blocos têm por principal característica serem chamados em muitas partes do algoritmo.

Bloco A. Monta listas de direções saindo de cada nó interno para encontrar qualquer outro nó na topologia usando a direção correta entre as três possíveis saídas (**Gargalo:** *Find_Mutual_Direction*);

Bloco B. Otimiza todos os parâmetros do modelo (um por vez) usando o método BFGS ou o método de Brent para maximizar o valor do parâmetro;

Bloco C.I. Atualiza a matriz de probabilidades de transição $Ve^{D(t)}V^{-1}$ para o galho com V e V^{-1} já calculados (**Gargalo:** *PMat_Empirical*);

Bloco C.II. Atualiza a matriz de probabilidades de transição $Ve^{D(t)}V^{-1}$ para todo galho (recursivamente) com V e V^{-1} já calculados (**Gargalo:** *PMat_Empirical*);

Bloco D. Calcula a MV parcial de cada base para cada categoria de cada sítio para cada uma das subárvores da topologia (**Gargalos:** *Update_P_LK_Nucl* ou *Update_P_LK_AA*);

Bloco E. Usa a MV parcial (supostamente atualizada para as subárvores) para calcular a MV da topologia em função do calculo de MV de cada sítio do alinhamento sobre um galho da topologia (**Gargalos:** *LK_Core, Pull_Scaling_Factors* e *Rate_Correction*);

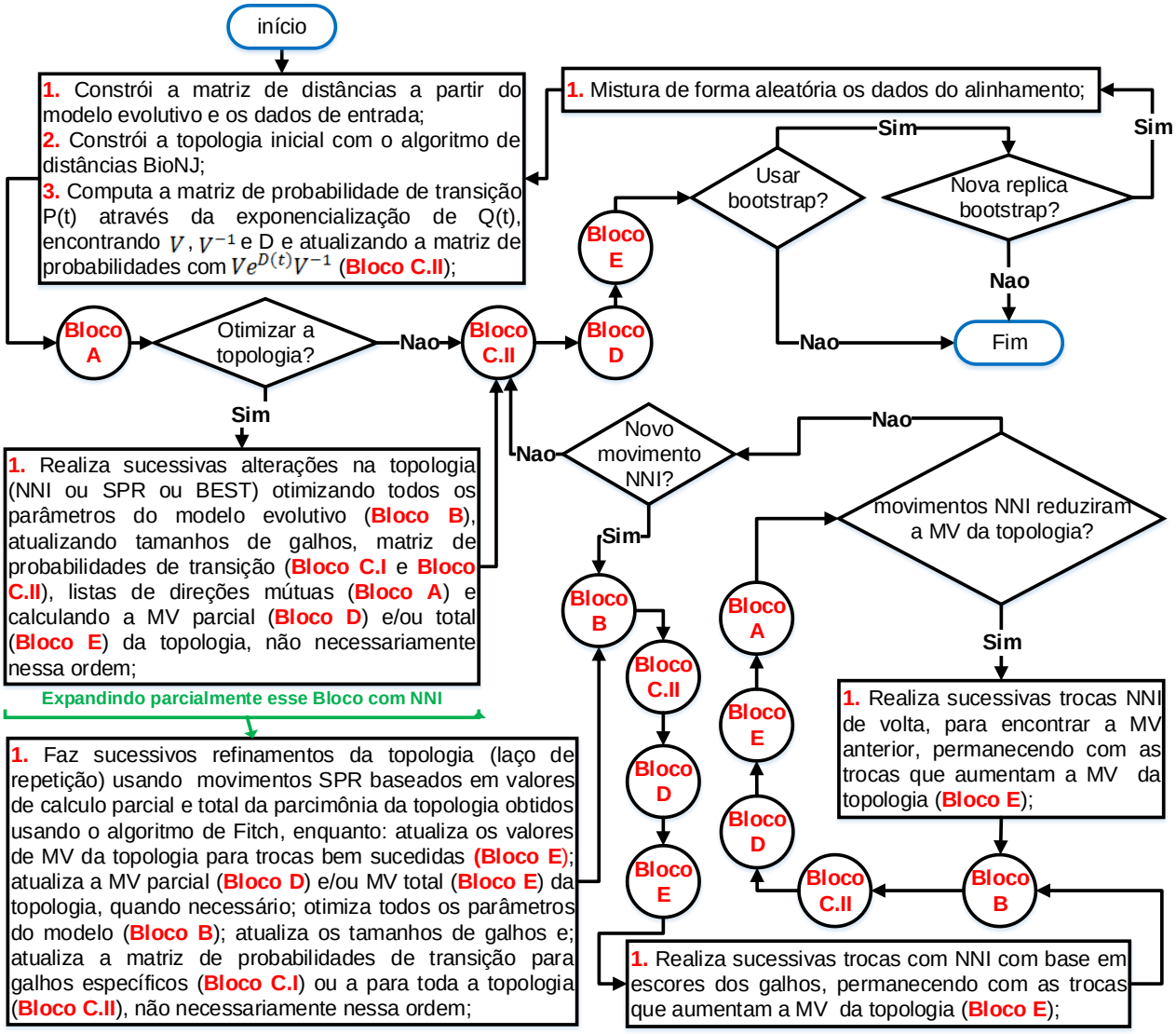

Figura 39 - Algoritmo genérico do programa PhyML. “Bloco X” é um desvio para um código descrito fora do fluxo. As funções gargalos estão identificadas após o termo em negrito “Gargalo” ou “Gargalos”, permitindo identificar onde as funções gargalos são usadas dentro do algoritmo.

Através da ilustração da Figura 39 pode-se notar os locais dentro da heurística do PhyML onde se encontram os gargalos de maior relevância. Todas estas funções serão explanadas de forma mais detalhadas a seguir, com exceção das funções *Pull_Scaling_Factors* e *Rate_Correction* em razão da paralelização com OpenMP ser inviável para estas.

4.4.2 Descrição das funções gargalos

Essa Subseção explana a descrição e algoritmos das funções gargalos. Por questões de simplicidades, nos algoritmos construídos será utilizado "←" como símbolo de atribuição, "*" como símbolo de multiplicação e "{Bloco de código}" como um bloco de código dependente da instrução/condição que o chamou.

4.4.2.1 Funções *Pull_Scaling_Factors* e *Rate_Correction*

Seja com relação a tamanhos de galhos ou MV, o PhyML realiza operações com valores reais muito pequenos, de tal forma que, dependendo da quantidade de operações realizadas com esses valores, estes podem ser reduzidos significativamente, possibilitando a ocorrência de erros por truncamento de valores, e, desta forma, alterando o resultado final dos cálculos de MV. Neste sentido, estas duas funções são usadas na intenção de melhorar a precisão desses valores através do uso de fatores de escala para possibilitar cálculos de maneira mais exata. A intenção é evitar os possíveis erros de truncamento ou perda de precisão numérica, mantendo este valor dentro dos limites mínimo e máximo que um número real pode ser representado e armazenado em um computador.

A justificativa do uso de fatores de escala para fazer a correção de valores de MV, é que, a melhoria na precisão do valor de MV das categorias para determinado galho deve ocorrer na mesma proporcionalidade para todas as categorias do sítio e que, de certa forma, afeta diretamente a MV do sítio. Neste sentido, o fator de escala atua como um controlador de proporcionalidade entre valores de MV de galhos.

A função *Pull_Scaling_Factors* obtém o fator de escala do sítio através dos fatores de escala de cada categoria, enquanto a função *Rate_Correction* aplica sucessivas correções usando esse fator. A função *Pull_Scaling_Factors* é chamada somente dentro da função *LK_Core*, enquanto a função *Rate_Correction* é chamada somente dentro da função *Pull_Scaling_Factors*. Ambas não serão desmembradas detalhadamente pelo fato de a paralelização com OpenMP ser inviável.

4.4.2.2 Função *Find_Mutual_Direction*

O PhyML simula a estrutura da topologia da árvore (nó, nó pai, nó filho esquerdo e nó filho direito) utilizando vetores[2], os quais são fáceis de acessar de forma indexada e permite uma redução de custos de acesso se comparados a processos de varredura recursiva na topologia em árvore. Para simular a topologia em árvore de forma vetorizada o PhyML mantém dois vetores: um vetor contendo os nós e outro contendo os galhos da topologia. Cada elemento do vetor de nós contém outro vetor de três posições que armazena os nós vizinhos direto do nó presente naquela posição. Essa estratégia de cada nó possuir um vetor com os três nós vizinhos diretos (links de acesso direto aos vizinhos) cria a possibilidade de sair de certo nó e chegar a qualquer outro sem a necessidade de usar a topologia em estrutura de árvore, isto é, usando apenas os caminhos criados com acessos aos nós vizinhos direto através de vetorização. Essa ideia está ilustrada na Figura 40, na qual a expressão “nós” representa o vetor de nós e os respectivos nós vizinhos diretos de cada nó da topologia.

[2] Estrutura de dados linear ou sequencial

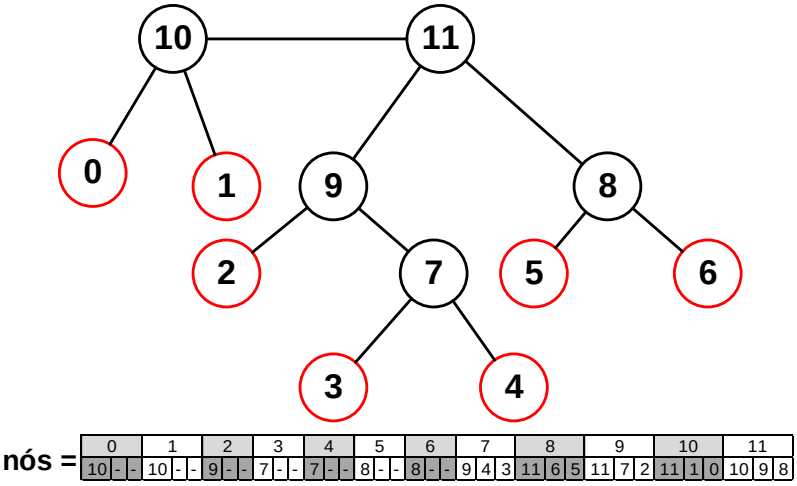

Figura 40 - Vetor de nós da topologia e nós vizinhos diretos de cada nó, usado para simular a estrutura topológica em árvore através da vetorização dos nós da árvore.

Com base na ideia de varredura sobre a topologia usando vetorização, cada nó da topologia contém o que se chama de tabela de bipartição, que é uma matriz de dimensão $3xM$ (sendo M no máximo a quantidade de espécies menos um). Cada linha dessa matriz é uma lista de bipartição e contém todos os nós folhas que podem ser alcançados a partir de dois galhos saindo do nó em questão. Desta forma, há no máximo 3 possíveis combinações (pares de galhos saindo do nó) para encontrar os nós folhas da topologia a partir do nó em questão, justificando 3 como a quantidade de linhas da matriz. A Figura 41 ilustra a ideia de listas de bipartição e as tabelas de bipartição para cada um dos nós internos de uma topologia exemplo.

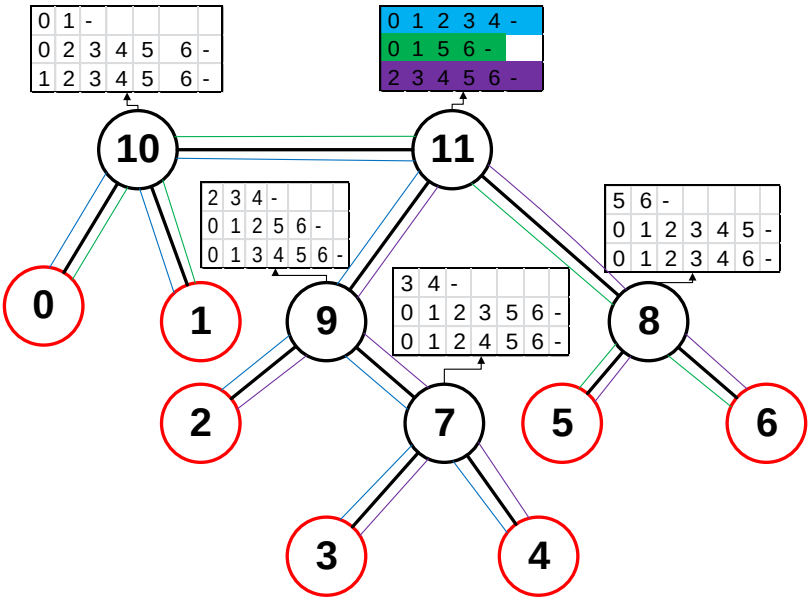

Figura 41 - Tabelas de bipartição usada pelo PhyML para encontrar nós folhas saindo em duas direções de cada vez.

Ainda de acordo com a Figura 41, para a tabela de bipartição do nó 11, pode-se observar a construção das listas de bipartição a partir das linhas nas cores azul (primeira lista de bipartição), verde (segunda lista de bipartição) e roxo (terceira lista de bipartição). Note que as tabelas de bipartição são definidas apenas para os nós internos, pois para nós folha existe apenas uma lista de bipartição (apenas um galho saindo daquele nó), a qual possui todos os outros nós folhas da topologia.

Usando as tabelas de bipartição (ilustradas na Figura 41) em conjunto com os vetores de nós e os respectivos nós vizinhos (ilustrados na Figura 40), pode-se sair de qualquer nó interno da topologia e chegar a qualquer outro (folha ou interno). O problema para concretizar esse percurso está em encontrar a direção correta (lista de bipartição a ser usada) que, saindo daquele nó interno leve à folha desejada ou mesmo a outro nó interno da topologia. De outra forma, as direções que deve-se seguir saindo de um nó interno para qualquer outro nó da topologia ainda não foram definidas.

Neste sentido, considere $n1$ e $n2$ dois nós internos de determinada topologia, os quais já possuem as respectivas tabelas de bipartição, mas as direções para sair de $n1$ e chegar em $n2$ (e vice-versa) ainda não foram definidas. A função *Find_Mutual_Direction* é responsável por encontrar e armazenar nos nós em questão, com base nas listas de bipartição de ambos os nós, as direções mútuas (de

$n1$ para $n2$ e de $n2$ para $n1$) entre dois nós internos da topologia, nesse caso, $n1$ e $n2$.

Para encontrar as direções mútuas entre pares de nós internos, a função *Find_Mutual_Direction* utiliza uma matriz de escores de dimensão $3x3$, na qual o índice da linha representa a lista de bipartição que deve-se usar para sair de $n1$ e chegar em $n2$ e o índice da coluna representa a lista de bipartição que deve-se usar para sair de $n2$ e chegar em $n1$. A estratégia para encontrar a lista/direção correta consiste em comparar cada elemento de cada uma das listas de bipartição de $n1$ com cada elemento das listas de bipartição de $n2$. À medida que se encontra um elemento de uma das listas de $n1$ em uma das listas de $n2$, o valor da célula da matriz de escores na posição linha e coluna (sendo linha e coluna respectivamente, lista de $n1$ que possui o elemento e, lista de n2 que possui o elemento) é incrementado. Ao final de varredura e comparação de cada elemento das listas de $n1$ com cada elemento das listas de $n2$, tem-se a matriz de escores preenchida. Os índices da célula da matriz que continua zerada após esse processo, representam as direções corretas que deve-se seguir em ambas as direções (lista de bipartição de $n1$ para $n2$ - índice linha, e, lista de bipartição de $n2$ para $n1$ - índice coluna). Para definir a direção, a estratégia é usar a característica de bipartição para isolar a única lista de bipartição que saindo de $n1$ não leva ao nó $n2$ e simultaneamente isolar a única lista de bipartição que saindo de $n2$ não leva ao nó $n1$, isto é, isolar as listas de bipartição que não possui intersecção com nenhum elemento. Após isolar as listas de bipartição que não possui intersecção de elementos, resta apenas o caminho que leva de $n1$ para $n2$ e vice versa.

A seguir é apresentado o algoritmo desta função. Por questões de simplicidade e compreensão do algoritmo, considere as seguintes afirmações e termos: o nó n_1 possui três listas de bipartição, uma saindo em cada direção a partir de n_1; $n_1\_tam\_list\_bip[i]$ é o tamanho da i-ésima lista de bipartição do nó n_1; $n_1\_Mat\_bip[n-k]$ é o k-ésimo elemento da n-ésima lista de bipartição do nó n_1; $escores[\mathrm{i}-\mathrm{j}]$ é uma matriz de dimensão $3x3$ que representa as possíveis direções que se pode tomar saindo de n_1 para n_2 (índice linha - i) e de n_2 para n_1 (índice coluna - j); $n_1\_Para\_n_2$ é a lista de bipartição que deve-se usar para sair de n_1 e chegar em n_2. Por analogia, segue os mesmo termos para o nó n_2.

Algoritmo: *Find_Mutual_Direction*
Entrada: n_1: nó, n_2: nó
Saída: direção/lista de bipartição que deve-se usar para sair de n_1 e chegar em n_2 e, a direção que deve-se usar para sair de n_2 e chegar em n_1

```
A. Para list_n1 = 1 até 3 - direções saindo de n1{
  1 Para list_n2 = 1 até 3 - direções saindo de n2{
    1.1 escores[ list_n1 − list_n2] ← 0;
    1.2 Para n1_elem = 1 até n1_tam_list_bip[list_n1]
      1.2.1 Para n2_elem = 1 até n2_tam_list_bip[list_n2]
        1.2.1.1 Se n1_Mat_bip[list_n1 − n1_elem] = n2_Mat_bip[list_n2 −
                n2_elem]{
                I.   escores[ list_n1 − list_n2] ← scores[ list_n1 − list_n2] + 1;
                II.   Volte ao Passo 1.2;
              }
    }
  }
B. Para list_n1 = 1 até 3 - direções saindo de n1{
   1  Para list_n2 = 1 até 3 - direções saindo de n2
    1.1 Se escores[ list_n1 − list_n2] = 0 - listas sem intersecção de
        elementos{
        I.    n1_Para_n2 ← list_n1;
        II.   n2_Para_n1 ← list_n2;
       }
  }
```

Note que a função *Find_Mutual_Direction* realiza o processo para apenas um par de nós, porém a função *Fill_Dir_Table*, faz sucessivas chamadas a esta função, passando em cada chamada uma combinação diferente de par de nós internos, de tal forma que todas as combinações possíveis de pares de nós internos são passados à função *Find_Mutual_Direction*. Com isso, define-se para todo par de nós internos, as direções corretas para sair de algum nó e encontrar qualquer outro, seja o outro um nó folha ou um nó interno.

A função *Find_Mutual_Direction* é executada sempre que uma nova topologia esteja sendo construída, como no caso de construção da topologia inicial ou construção de uma nova topologia para uso com réplicas *bootstrap*. Outro ponto da heurística onde se usa muitas chamadas à função *Find_Mutual_Direction* é quando há alguma alteração da topologia com algoritmos de otimização topológica como o NNI, o SPR ou o BEST.

4.4.2.3 Função *PMat_Empirical*

A MV de um galho l sobre o tamanho de galho t é obtida por $P(l) = e^{Q(t)} = VDV^{-1}$ (conforme eq. (12) e eq. (13) - subseção 2.1.3 Processos de Markov). Porém, informações encontradas no código do PhyML indica que este usa o desenvolvimento de Taylor sobre $e^{Q(t)}$, o qual fornece a expressão $P(l) = e^{Q(t)} = Ve^{D(t)}V^{-1}$. Essa expressão apresenta uma característica útil para computar a MV de qualquer galho usando a mesma matriz V e sua inversa V^{-1}, isto é, sem a necessidade de recalcular os autovalores e os respectivos autovetores direitos e a inversa da matriz de autovetores direitos de $Q(t)$ para cada galho da topologia. Como o PhyML necessita que cada galho tenha sua matriz $P(l)$, há a necessidade de computar $P(l) = Ve^{D(t)}V^{-1}$ para cada galho, porém, a característica da expressão de Taylor supracitada permite que V e V^{-1} seja computada apenas uma vez e, considerando o cálculo para um galho específico, basta atualizar a matriz diagonal $e^{D(t)}$ de acordo com o galho em questão e, por fim, realizar a operação completa $P(l) = Ve^{D(t)}V^{-1}$, reduzindo o custo computacional e o tempo de processamento.

A função *PMat_Empirical* tem por objetivo computar a matriz de probabilidades de transição $P(l)$ para o galho l recebido como parâmetro. Para isso usa $P(l) = Ve^{D(t)}V^{-1}$, aplicando a estratégia das matrizes já calculadas V e V^{-1} e apenas atualizando a matriz diagonal de autovalores $e^{D(t)}$ em função do galho l. Esta função possui, em alguns casos, tempo cumulativo de execução significativo, em razão das operações de multiplicações de matrizes realizadas para obter $P(l)$ aliado à quantidade de vezes que as operações precisam ser realizadas. A operação de multiplicação de matrizes é aplicada sobre três matrizes de dimensão nxn. Para sequências de DNA, n é igual a 4, representando a quantidade de bases. Já para sequências de proteínas, n é igual a 20, o que aliada a quantidade de vezes, justifica o fato da função *PMat_Empirical* se apresentar como gargalo para algumas sequências de proteínas, mas não se apresentar como gargalo para sequências de DNA. A seguir, é apresentado o algoritmo para esta função.

Algoritmo: *PMat_Empirical*
Entrada: l: galho, V: matriz de autovetores direitos, V^{-1}: matriz inversa de V, D: matriz diagonal de autovalores de $Q(t)$
Saída: $P(l)$: matriz de probabilidade de transição do galho l

A. $P(l) \leftarrow 0$;
B. $\exp\_D_t \leftarrow e^{D(t)}$;
C. $\text{Aux} \leftarrow \exp\_D_t * V$;
D. $P(l) \leftarrow \text{Aux} * V^{-1}$;
E. Retorna $P(l)$.

Esta função é chamada sempre que há a necessidade de atualizar a matriz de probabilidades de transição para determinado galho para o qual a taxa de transição sofreu alguma alteração.

4.4.2.4 Função *Update_P_LK_Nucl*

A função *Update_P_LK_Nucl* tem por objetivo calcular ou atualizar a MV parcial[3] de uma subárvore dentro de uma topologia. Esta função recebe como parâmetros um galho e um nó, por convenção, nomeados aqui de b e n, respectivamente. O galho b e o nó n indicam que a subárvore a ser calculada a MV parcial se encontra no lado oposto a esse galho, em relação ao nó n. A MV parcial do nó n é calculada em função: dos nós $n1$ e $n2$; das respectivas MV parciais dos nós $n1$ e $n2$ (calculadas em alguma iteração anterior) e das matrizes de probabilidades de transição dos galhos $b1$ e $b2$. A Figura 42 ilustra algumas dessas características, na qual pode-se notar o galho b e a subárvore na qual será calculada a MV.

[3] Considere MV parcial de um galho como a verossimilhança de toda a subárvore abaixo daquele galho.

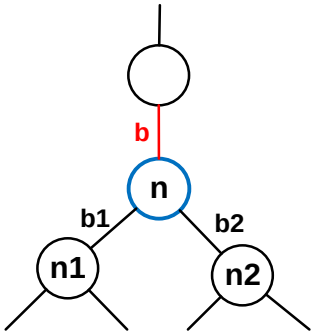

Figura 42 - Galho b indica que a MV parcial deve ser calculada na subárvore do nó n em função dos nós $n1$ e $n2$.

Para o calculo de MV parcial, a função *Update_P_LK_Nucl* implementa de forma adaptada a equação de probabilidade condicional de Felsenstein (conforme eq. (34)) na qual a probabilidade de determinada base ocorrendo no nó n (considerando a respectiva categoria e sítio) é o produtório da MV calculada de n para $n1$ sobre o galho $b1$ pela MV calculada de n para $n2$ sobre o galho $b2$. Já a MV calculada de n para $n1$ sobre o galho $b1$ ocorre fixando uma base no nó n e variando as possíveis bases em $n1$, ou seja, é a somatória do: valor de probabilidade de transição da base fixada em n para a base corrente em $n1$ sobre o galho $b1$ (conforme matriz de probabilidade de transição do galho $b1$) multiplicado pela MV parcial da base corrente em $n1$ que foi calculada em alguma iteração anterior (considerando categoria e sítio corrente para a matriz de probabilidade de $b1$ e a MV de $n1$). Por analogia, segue de forma semelhante o calculo de MV de n para $n2$ sobre o galho $b2$.

Esta função calcula e armazena o valor de MV parcial para cada uma das bases variando no nó n em cada uma das categorias e para cada um dos sítios. Assim, o nó n possui a MV parcial de toda a subárvore abaixo dele, para toda base, categoria e sítio. O cálculo de MV para a subárvore acima da subárvore corrente (nó n) ocorrerá de forma direta (de forma análoga ao feito na iteração corrente para o nó n com a MV parcial dos nós $n1$ e $n2$), ou seja, simplesmente usando os valores atualizados de MV desta subárvore obtidos de forma cumulativa das subárvores abaixo dessa.

A função *Update_P_LK_Nucl* calcula a MV parcial de apenas uma subárvore, porém, na maioria das vezes, através de chamadas recursivas feitas por funções de varredura da topologia, é utilizada para computar a MV parcial de toda a árvore. O

PhyML implementa duas estratégias de varredura em topologia, varredura em pós-ordem e varredura em pré-ordem. Quando a MV da topologia é calculada através de chamadas recursivas usando estratégia em pós-ordem, ao término dos cálculos de MV parcial, a raiz ou o nó vizinho ao primeiro nó da topologia (para topologia enraizada ou topologia sem raiz, respectivamente) possui a MV da árvore para cada sítio (atualizada para toda a topologia, considerando cada base dentro de cada categoria de cada sítio).

A estratégia de varredura pós-ordem, mencionada anteriormente, está ilustrada na Figura 43, através da qual pode-se notar que ao final dos cálculos, todo nó possui a MV parcial para cada base dentro de cada categoria de cada sítio, considere que os nós Esp1, Esp2 e Esp3, de forma análoga à Esp4, possuem MV_Esp1, MV_Esp2 e MV_Esp3, respectivamente. Essa ilustração considera 4 espécies com sequências de nucleotídeos, com 2 sítios, usando 4 categorias por sítio (valores meramente ilustrativos).

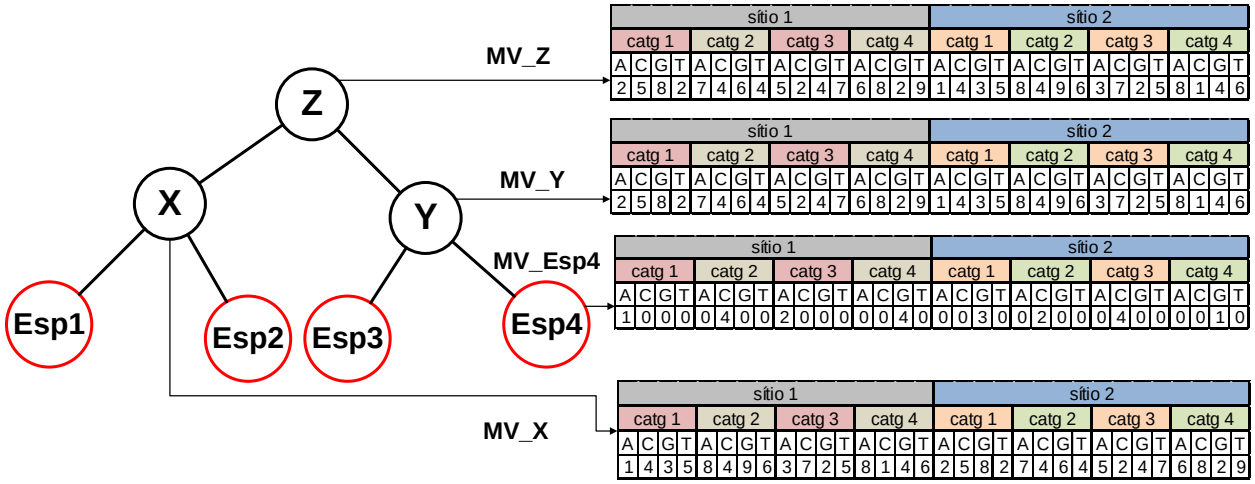

Figura 43 - Ilustração da MV parcial calculada pela função *Update_P_LK_Nucl* através de chamadas por estratégias de pós-ordem, raiz com MV parcial atualizada para toda base de toda categoria e sítio (números meramente ilustrativos).

A seguir é apresentado o algoritmo para essa função, por questões de simplicidade e compreensão do algoritmo, considere os seguintes termos: NT a quantidade de tipos de bases de nucleotídeos a serem processadas; $P\_Mat_1[X-Y]$ o valor de probabilidade de transição da mudança da base X presente no nó n

para a base Y presente no nó n_1 sobre o galho b_1; $\mathrm{n}_1\_Amb[X]$ o valor de MV parcial do nó n_1 com a base X, se n_1 for uma folha ambígua e a base X pertencer ao conjunto ambíguo; $\mathrm{MV\_n}_1[X]$ como a MV parcial do nó n_1 com a base X, sendo n_1 um nó interno – o valor $\mathrm{MV\_n}_1[X]$ foi calculado em alguma iteração anterior; $\mathrm{MV\_n\_n}_1$ como a MV parcial do nó n para o nó n_1 em determinada base - calculado no algoritmo; MV_n como a MV parcial de toda a subárvore até o nó n, com a base corrente no nó n. Por analogia, segue os mesmo termos para $\mathrm{n}_2\_Amb[X]$, $\mathrm{MV\_n}_2[X]$; $\mathrm{MV\_n\_n}_2$ e $P\_Mat_2[X-Y]$ para o nó n_2. Considere ainda que em todos esses subprocessos, a base em questão está sendo processada/usando/atualizando valores dentro de determinada categoria que se encontra dentro de determinado sítio.

Algoritmo: *Update_P_LK_Nucl*
Entrada: T: topologia, b: galho, n: nó
Saída: T: topologia com MV parcial da subárvore (que saindo do nó n, se encontra oposta ao galho b) calculada/atualizada para cada base ocorrida na raiz da subárvore (nó n) dentro de cada uma das categorias definidas em cada um dos sítios.

Seja n_1 e n_2 nós vizinhos do nó n e, b_1 e b_2 galhos que interligam n a n_1 e, n a n_2, respectivamente, a MV do nó n (subárvore saindo do galho b) será calculada em função dos nós n_1 e n_2 e dos galhos b_1 e b_2.

A. Obtém de n_1 e n_2, as informações, como soma de fator de escala e MV parcial ($\mathrm{MV\_n}_1$ e $\mathrm{MV\_n}_2$) para cada possível base. Matriz de probabilidades de transição para os galhos b_1 e b_2 (P_Mat_1 e P_Mat_2) e vetores de MV parcial quando os nós n_1 e/ou n_2 são folhas ambíguas (n_1_Amb e/ou n_2_Amb) necessárias para o calculo de MV;
B. Para $sit = 1$ até a quantidade de sítios do alinhamento{
  1 Para $catg = 1$ até a quantidade de categorias definida para os sítios{
    1.1 Para $n\_nt = 1$ até a quantidade NT variando no nó n{
      1.1.1 $\mathrm{MV\_n\_n}_1 \leftarrow \mathrm{MV\_n\_n}_2 \leftarrow 0$;
      1.1.2 Se n_1 é folha não ambígua{
        estado $\leftarrow \mathrm{n}_1\_estado\_folha$;
        $\mathrm{MV\_n\_n}_1 \leftarrow \mathrm{P\_Mat}_1[n\_nt - estado]$;
      }
      1.1.3 Se n_1 é folha ambígua
        I. Para $n_1\_nt = 1$ até a quantidade NT variando no nó n_1
        $\mathrm{MV\_n\_n}_1 \leftarrow \mathrm{MV\_n\_n}_1 + \mathrm{P\_Mat}_1[n\_nt - \mathrm{n}_1\_nt] * \mathrm{n}_1\_Amb[\mathrm{n}_1\_nt]$;

1.1.4 Se n_1 é nó interno

I. Para $n_1\_nt = 1$ até a quantidade NT variando no nó n_1

$\text{MV\_n\_n}_1 \leftarrow \text{MV\_n\_n}_1 + \text{P\_Mat}_1[n\_nt - \text{n}_1\_nt] * \text{MV\_n}_1[\text{n}_1\_nt];$

1.1.5 Se n_2 é folha não ambígua {

estado $\leftarrow$ $\text{n}_2\_estado\_folha$;

$\text{MV\_n\_n}_2 \leftarrow \text{P\_Mat}_2[n\_nt - estado];$

}

1.1.6 Se n_2 é folha ambígua

I. Para $n_2\_nt = 1$ até a quantidade NT variando no nó n_2

$\text{MV\_n\_n}_2 \leftarrow \text{MV\_n\_n}_2 + \text{P\_Mat}_2[n\_nt - \text{n}_2\_nt] * \text{n}_2\_Amb[\text{n}_2\_nt];$

1.1.7 Se n_2 é nó interno

I. Para $n_2\_nt = 1$ até a quantidade NT variando no nó n_2

$\text{MV\_n\_n}_2 \leftarrow \text{MV\_n\_n}_2 + \text{P\_Mat}_2[n\_nt - \text{n}_2\_nt] * \text{MV\_n}_2[\text{n}_2\_nt];$

1.1.8 Define a MV parcial de n_nt no nó n

$MV\_n = \text{MV\_n\_n}_1 * \text{MV\_n\_n}_2;$

}

}

2 Define o fator de escala do nó n na categoria corrente ($catg$) como a soma dos fatores de escala também na categoria corrente ($catg$) dos nós n_1 e n_2;

3 Aplica o fator de escala para cada base;

}

4.4.2.5 Função *Udate_P_LK_AA*

De forma análoga à função *Update_P_LK_Nucl*, a função *Udate_P_LK_AA* calcula o MV parcial para sequências de aminoácidos. A justificativa para usar funções diferentes com finalidades idênticas é que os passos 1.1.3 (I), 1.1.4 (I), 1.1.6 (I) e 1.1.7 (I) do algoritmo da subseção 4.4.2.4 (Função *Update_P_LK_Nucl*) foram construídos no código do PhyML usando variáveis e não um laço de repetição como no algoritmo. Desta forma, a função *Update_P_LK_Nucl* trabalha com apenas quatro variáveis, representado respectivamente a MV da variação de A, C, G e T do nó n para o nó $n1$ e as mesmas variáveis do nó n para o nó $n2$, com a mesma significância. Já a função *Udate_P_LK_AA* utiliza vinte variáveis para representar a variação dos aminoácidos. Neste sentido, será omitido o algoritmo da função

Udate_P_LK_AA, pois, por analogia, pode-se usar o algoritmo da função *Update_P_LK_Nucl* para a compreensão desta.

4.4.2.6 Função *LK_Core*

A função *LK_Core* é o núcleo dos cálculos de MV. Esta recebe um galho da topologia e um sítio como parâmetros e calcula a MV do sítio por intermédio do galho corrente, assumindo que as subárvores em ambos os lados do galho recebido encontram-se com as MV parciais atualizadas para cada caractere dentro de cada categoria do sítio. Como visto na subseção 4.4.2.4 (Função *Update_P_LK_Nucl*), a MV parcial de toda a topologia é atualizada pela função *Update_P_LK_Nucl* para sequências de DNA ou pela função *Update_P_LK_AA* para sequências de proteínas (conforme subseção 4.4.2.5 Função *Update_P_LK_AA*).

Para a realização dos cálculos de MV, a função *LK_Core* usa de forma adaptada a estratégia descrita por Adachi e Hasegawa, formulada sobre a eq. (32), na qual a MV total do sítio é a somatória das MVs das categorias. Já a MV de determinada categoria, é calculada em função da somatória sobre a probabilidade de transição de todas as possíveis combinações de variação de caracteres em ambos os lados do galho e categoria corrente. Note que em concordância com a eq. (32), o cálculo da probabilidade de transição sobre a variação de caracteres utiliza: a frequência do caractere fixado em um dos lados do galho, as MV parciais dos caracteres em ambos os lados do galho (já calculadas) e o valor de probabilidade de transição de um caractere para outro no galho, categoria e sítio corrente. A matriz de probabilidade de transição para o galho corrente possui valores de transição para cada combinação de caracteres e para cada categoria.

É importante ressaltar que na função *LK_Core*, o tamanho do galho corrente pode ou não já ter passado por sucessivas otimizações e que, a matriz de probabilidade de transição é constituída de valores de probabilidade de transição de um caractere para qualquer outro sobre o galho e categoria corrente, os quais foram obtidos com o método BFGS ou pelo método de Brent antes de chegar a essa função.

A função *LK_Core* usa a estratégia de aplicação de fatores de escala na MV de cada categoria e, ao final aplica também na MV do sítio com objetivo de evitar perda de precisão nos cálculos de MV. Ao final do calculo de MV do sítio, é aplicado o logaritmo sobre esse valor de MV, conforme explicitado na eq. (27), e, em função dessa característica, a MV do sítio é inserida ou incrementada na MV da topologia como somatória e não como produtório, em contraste com a eq. (32) e, por isso adaptada, e em concordância com a eq. (27).

A seguir é apresentado o algoritmo desta função. Por questões de simplicidade e compreensão, considere as seguintes afirmações e termos: b_Lado_esq é o caractere corrente no lado esquerdo do galho b (recebido como parâmetro); b_Lado_dir índice do caractere que se encontra no lado direito do galho b; MV_Catg_Sit é a MV da categoria e sítio corrente; $b\_MatProbTrans[X-Y]$ valor da probabilidade de transição da mudança do caractere X (lado esquerdo do galho b) para o caractere Y (lado direito do galho b) sobre o galho b no sítio e categoria corrente; $folha$ é o estado do caractere na folha (lado direito do galho b) para um galho externo; $freq$ é a frequência da folha ($folha$) encontrada no lado direito do galho b; b_MV_folha é a MV do estado corrente encontrado na folha ambígua; $b\_esq\_MV\_Catg\_Sit$ é a MV da categoria e sítio corrente na subárvore esquerda do galho b, calculada pela função *Update_P_LK_Nucl* ou pela função *Update_P_LK_AA*; $T\_MV\_Catg\_Sit[X]$ é a MV da categoria X do sítio corrente sendo armazenada na topologia inserida na MV da topologia T; MV_Sit é a MV do sítio corrente; $MV\_Escalar\_Catg\_Sit[X]$ é a MV com fator de escala da categoria X do sítio corrente; $Densid\_Gama[X]$ é taxa de densidade de probabilidade da categoria X.

Algoritmo: LK_Core
$Entrada$: T: topologia, b: galho corrente, $sít$: sítio corrente
$Saída$: T: com a MV do galho b atualizada para o sítio corrente em cada uma das categorias e caracteres e a MV do sítio adicionada a MV da topologia

A. Para $catg = 1$ até a quantidade de categorias{
  1 $MV\_Catg\_Sit \leftarrow 0$;
  2 Se b é galho externo e b_dir_folha é uma folha não ambígua{
    2.1 $soma \leftarrow 0$;
    2.2 $folha \leftarrow b\_dir\_folha$;
    2.3 Para $b\_Lado\_esq = 1$ até a quantidade de caracteres

2.3.1 $soma \leftarrow soma + b\_MatProbTrans[b\_Lado\_esq - folha] * b\_esq\_MV\_Catg\_Sit[b\_Lado\_esq];$

2.4 $MV\_Catg\_Sit \leftarrow MV\_Catg\_Sit + soma * freq[folha];$

}

3 Se b é galho externo e b_dir_folha é uma folha ambígua

3.1 Para $b\_Lado\_dir = 1$ até a quantidade de caracteres{

3.1.1 $soma \leftarrow 0;$

3.1.2 Se $b\_MV\_folha[b\_Lado\_dir] > 0$ {

3.1.2.1 Para $b\_Lado\_esq = 1$ até a quantidade de caracteres

I. $soma \leftarrow soma + b\_MatProbTrans[b\_Lado\_esq - b\_Lado\_dir] * b\_esq\_MV\_Catg\_Sit[b\_Lado\_esq];$

3.1.2.2 $MV\_Catg\_Sit \leftarrow MV\_Catg\_Sit + soma * freq[b\_Lado\_dir] * b\_dir\_MV\_Catg\_Sit[b\_Lado\_dir];$

}

4 Se b é galho interno

4.1 Para $b\_Lado\_dir = 1$ até a quantidade de caracteres{

4.1.1 $soma \leftarrow 0;$

4.1.2 Se $b\_dir\_MV\_Catg\_Sit[b\_Lado\_dir] > 0$ {

4.1.2.1 Para $b\_Lado\_esq = 1$ até a quantidade de caracteres

I. $soma \leftarrow soma + b\_MatProbTrans[b\_Lado\_esq - b\_Lado\_dir] * b\_esq\_MV\_Catg\_Sit[b\_Lado\_esq];$

4.1.2.2 $MV\_Catg\_Sit \leftarrow MV\_Catg\_Sit + soma * freq[b\_Lado\_dir] * b\_dir\_MV\_Catg\_Sit[b\_Lado\_dir];$

}

}

5 $T\_MV\_Catg\_Sit[Catg] \leftarrow MV\_Catg\_Sit;$

}

B. Faz chamada à função *Pull_Scaling_Factors* para definir e aplicar o fator de escala do sítio;

C. $MV\_Sit \leftarrow 0;$

D. Para $catg = 1$ até a quantidade de categorias

1. $MV\_Sit \leftarrow MV\_Escalar\_Catg\_Sit[catg] * Densid\_Gama[catg];$

E. Se o sítio corrente ($Sít$) for invariável, aplica o a proporção para sítios invariáveis. A proporção para sítios invariáveis é a frequência dos dados para o sítio corrente com aplicação de fatores de escala;

F. Aplica o logaritmo na MV do sítio $Log\_MV\_Sit \leftarrow \log(MV\_Sit);$

G. Aplica no valor Log_MV_Sit o fator de escala do sítio;

H. Multiplica Log_MV_Sit do sítio $Sít$ pela quantidade de vezes que este aparece nas sequências e adiciona o resultado na MV da topologia $T;$

É notório que a função *LK_Core* calcula a MV para apenas um sítio sobre o galho corrente, porém, a função que faz chamadas à função *LK_Core* (função *LK*), faz chamada para cada um dos sítios do alinhamento para que, a MV da topologia

sobre o galho em questão seja calculada para todos os sítio do alinhamento e, desta forma, com a suposição de que em ambos os lados do galho a MV parcial encontra se atualizada, obtém-se a MV de toda a Topologia. Por outro lado, a função *LK* é chamada sucessivas vezes para recalcular a MV de determinado galho quando há otimização da topologia (movimentos NNI, SPR ou BEST), ou mesmo quando há otimização de parâmetros do modelo, justificando o alto custo computacional da função *LK_Core* em função da grande quantidade de vezes que é chamada.

4.4.3 Desenvolvimento dos códigos em paralelo

Os dados obtidos do OpenMP MicroBenchmark Suite (subseção 4.3 Viabilidade de paralização de funções gargalos) indicaram que pode-se ter ganho com a paralelização das seguintes funções: *LK_Core, Update_P_LK_Nucl*, *Update_P_LK_AA*, *Find_Mutual_Direction* e *PMat_Empirical*. No entanto, notou-se que todos esses gargalos possuem tempo médio de execução inferior a 1 segundo, Neste sentido, no intuito de paralelizar o gargalo de forma que este tenha a maior granularidade possível, adotou-se a estratégia de paralelizar o gargalo o mais externamente quanto possível e, caso exista um gargalo dentro de outro, somente o gargalo mais externo será paralelizado e, desta forma, o gargalo mais interno será paralelizado de forma indireta. Neste sentido, a seguir são explanadas as estratégias de paralelização dessas funções gargalos.

4.4.3.1 Paralelização da função *Find_Mutual_Direction*

A função *Find_Mutual_Direction* é composta de estruturas de repetição aninhadas, porém, as estruturas de repetição mais externas são pequenas variando de um a três. Nesse caso, encontrar dentro da função uma estratégia que possibilite melhorar o desempenho tornou-se uma tarefa quase impraticável, de tal forma que, a estratégia foi analisar se nas chamadas dessa função existem estruturas

adequadas à paralelização. Como descrito na subseção 4.4.2.2 (Função *Find_Mutual_Direction*), essa função encontra as direções mútuas apenas entre um par de nós, porém na função *Fill_Dir_Table*, que chama a função *Find_Mutual_Direction,* existem duas estruturas de repetição aninhadas, usadas para fazer sucessivas chamadas à função *Find_Mutual_Direction*, combinando cada nó interno com todos os outros ainda não processados. Neste sentido, essas duas estruturas aninhadas foram paralelizadas usando o conjunto de diretivas *"#pragma omp for private(lista)"*, em que *lista* é a variável de controle do segundo laço de repetição.

4.4.3.2 Paralelização da função *PMat_Empirical*

Conforme pôde ser visto no algoritmo da subseção 4.4.2.3 (Função *PMat_Empirical*), esta função possui quatro blocos que, em termos de forte acoplamento entre tarefas podem ser paralelizados. Esses blocos estão relacionados a operações com matrizes. Existem quatro blocos, os quais trabalham com matrizes de dimensão igual à quantidade de possíveis caracteres, sendo, portanto, de dimensão 4 para DNA e de dimensão 20 para proteínas. Cada um dos blocos foi paralelizado individualmente em função da quantidade de linhas da matriz. Neste caso, como há mais de um bloco a ser paralelizado, antes do "Passo A" do algoritmo descrito na subseção 4.4.2.3 (Função *PMat_Empirical*), foi criada a região paralela com a diretiva *"#pragma omp parallel"* e foi fechada somente depois do ultimo bloco paralelizado "Passo D", com o objetivo de evitar maiores custos computacionais ao sucessivamente abrir e fechar a regiões paralelas. Já os blocos (Passo A, Passo B, Passo C e Passo D) foram paralelizados com a diretiva *"#pragma omp for private (lista)"*, em que *"lista"* são as variáveis privadas do bloco. Nessa função, a *"lista"* é composta pelas variáveis de controle de repetição internas, como o índice da coluna da matriz.

4.4.3.3 Paralelização da função *Update_P_LK_Nucl*

Com a possível indicação de ganho na paralelização desta função, foi investigado sobre análise minuciosa do código, qual o escopo dentro desta função que costuma ser mais custoso e que é viável a paralelização, ou mesmo se existe uma estrutura nas chamadas dessa função que seja plausível de paralelização.

Com base no algoritmo desta função, descrito na subseção 4.4.2.4 (Função *Update_P_LK_Nucl*), nota-se que o bloco de maior custo encontra-se no Passo B do algoritmo, onde ocorre o calculo de MV parcial em função dos sítios, o qual se apresentou como melhor bloco para a paralelização. Uma paralelização externa sobre as chamadas da função *Update_P_LK_Nucl* é inviável em razão destas ocorrem de forma recursiva por funções de varredura da topologia. Neste sentido, o bloco de código do Passo B foi adotado para aplicar a medida de paralelização, pois em termos de granularidade, o Passo B (calculo sobre sítios) é mais custoso que os cálculos mais internos, os quais ocorrem no Passo 1 e Passo 1.1 (calculo sobre categorias e bases, respectivamente), já que o Passo B é o laço de repetição mais externo. Além disso, a paralelização sobre sítios aloca mais blocos de trabalho para cada thread do que sobre categorias ou bases, em que os laços de repetição são significativamente menores.

Ainda de acordo com o Passo B do algoritmo da subseção 4.4.2.4 (Função *Update_P_LK_Nucl*), pode-se notar que os cálculos de MV sobre sítios são realizados de maneira independente, isto é, sem forte acoplamento entre tarefas, a não ser pela reutilização de algumas variáveis no código original. A paralelização foi aplicada diretamente no laço de repetição sobre os sítios (Passo B) com o conjunto de diretivas de paralelização "#pragma *omp parallel for private(lista)*", em que "*lista*" são variáveis locais da função que estão sendo reutilizadas e que precisam ser privadas de cada thread para evitar erros de calculo ou forte acoplamento entre tarefas.

4.4.3.4 Paralelização da função *Update_P_LK_AA*

Para a função *Update_P_LK_AA*, o processo foi análogo ao da função *Update_P_LK_Nucl*, pois ambas possuem o mesmo algoritmo, escritos individualmente para cada função. Neste sentido, a paralelização ocorreu dentro da própria função sobre a quantidade de sítios a ser calculada a MV parcial (análogo ao Passo B do algoritmo da subseção 4.4.2.4 Função *Update_P_LK_Nucl*), com o conjunto de diretivas "#pragma *omp parallel for private(lista)*", e, da mesma forma que a função anterior, os cálculos são feitos para cada sítio de forma independente, porém, algumas variáveis tiveram que ser privatizadas (índice do laço de repetição das categorias, variáveis que armazenam a MV para cada caractere do alfabeto(DNA), variável que armazenam valores de escala, entre outras) para evitar forte acoplamento entre tarefas nos cálculos de MV.

4.4.3.5 Paralelização da função *LK_Core*

Conforme pôde ser observado no algoritmo da subseção 4.4.2.6 (Função *LK_Core*), o corpo da função *LK_Core* realiza cálculos de MV para apenas um sítio recebido como parâmetro. Porém, conforme mencionado anteriormente, a estratégia é usar a característica de granularidade do paradigma OpenMP para tentar otimizar o ganho com a paralelização em função do processamento do bloco de maior custo computacional. Neste sentido, buscou-se explorar a paralelização de *LK_Core* de forma externa, isto é, a paralelização em função das chamadas de *LK_Core*. A função *LK* é responsável por fazer as chamadas à *LK_Core* passando o próximo sítio a ser calculada a MV. De acordo com a heurística apresentada e obtida a confirmação através do código fonte do PhyML, os sítios evoluem independentemente, desta forma, de imediato, a paralelização sobre as chamadas de *LK_Core* mostrou-se viável.

Apesar da viabilidade de paralelização das chamadas de *LK_Core*, esta apresentou forte acoplamento entre tarefas, pois, após o final dos cálculos para cada sítio, há a necessidade de adicionar a MV do sítio à MV da topologia, forçando a comunicação das threads para sincronizar o valor de MV da topologia. Por outro lado, o PhyML mantém estruturas globais que são usadas em todo o código, a função *LK_Core* estava usando dois vetores da estrutura, os quais possuem tamanhos limitados, isto é, o tamanho dos vetores é a quantidade de categorias. Um dos vetores estava sendo usado para armazenar a MV das categorias do sítio, já o outro estava sendo usado para armazenar as somas de fatores de escala das categorias para o sítio corrente. Porém, ao realizar a paralelização de *LK_Core* em função dos sítios, esses vetores estavam sendo usando simultaneamente por várias threads e, desta forma, realizando os cálculos de MV com base em valores de outros sítios. Neste sentido, o código do PhyML foi alterado dentro da função *LK_Core* de tal forma que os vetores passaram a ser definidos localmente na função e, portanto, privados de cada thread, evitando possíveis usos simultâneos que causariam erros de calculo ou mal funcionamento do programa PhyML. É importante destacar que esses vetores tem apenas quatro posições (tamanho 4) o que não reflete em aumento execissivo do uso de memória, pois isto poderia interferir no desempenho.

O laço de repetição da função *LK* que faz chamadas à função *LK_Core* foi paralelizado com o conjunto de diretivas "#pragma *omp parallel for reduction(+:MV_Top) private(lista)*", em que "*MV_Top"* é a variável que cumulativamente vai armazenando o valor de MV de cada sítio e "*lista"* é a lista de variáveis que devem ser privadas de cada thread. Por convenção, a partir deste momento, considere que toda referencia à função *LK_Core* incluindo também o laço de repetição da função *LK* que a chama, pois este laço é exclusivo apenas para fazer chamadas a essa função.

4.4.3.6 Paralelização das funções *Pull_Scaling_Factors* e *Rate_Correction*

Estas duas funções se mostraram como gargalos em termos de tempo cumulativo, porém o baixo tempo médio de execução de ambas as inviabilizaram de

serem paralelizadas em razão do *overhead* das diretivas OpenMP, no entanto a função *Rate_Correction* é chamada somente dentro da função *Pull_Scaling_Factors,* enquanto esta é chamada somente pela função *LK_Core*, e *LK_Core* já possui suas chamadas paralelizadas em função dos sítios. Neste sentido, de certa forma, tanto a função *Pull_Scaling_Factors* quanto a função *Rate_Correction* se encontram paralelizadas de forma indireta através da paralelização das chamadas da função *LK_Core*.

5 RESULTADOS E DISCUSSÕES

Os testes, simulações e análises de desempenho aqui apresentadas foram realizadas usando as duas arquiteturas do CACAU. Para simulações com até 8 threads e tempos seriais relacionados a estas simulações, foi utilizada a arquitetura comum do CACAU, já simulações com 10 e 12 threads aqui apresetadas bem como os tempos seriais que estejam relacionados às simulações com 10 e 12 threads foram simulados sobre a arquitetura GPU. As características de ambas as arquiteturas foram apresentadas noinício da seção anterior (4 DESENVOLVIMENTO).

Nas simulações foi utilizado o compilador GCC versão 5.3.0, sistema operacional linux OpenSuze x86 versão 2.6.18, OpenMP versão 4.0 e MPI versão openmpi 1.6.5. Quanto à estratégia de rearranjos de topologia, foi utilizada a opção BEST (por ser uma estratégia que retorna a árvore mais verossímil entre as estratégias NNI e SPR e, consequentemente é a estratégia que mais consome tempo na execução) tanto para sequências de DNA quanto para sequências de proteínas. Já com relação ao modelo evolutivo, para dados de DNA foi utilizado o modelo GTR (por ser o modelo evolutivo mais realístico, de acordo com a literatura), enquanto para dados de proteínas foi utilizado o modelo evolutivo LG (modelo padrão adotado nas execuções do PhyML).

Já com relação às análises comparativas e avaliação da aplicação paralela (e suas versões) em relação à versão serial, serão consideradas duas métricas: o *speedup* e a eficiência.

O *speedup* fornece um indicador para o aumento de velocidade em razão da utilização de uma arquitetura paralela, sendo este a razão entre o tempo de processamento da versão serial e o tempo de processamento da versão paralela sobre p processadores, isto é:

$$S_p = \frac{T_s}{T_p} \tag{36}$$

Em que T_s é o tempo de execução do programa na versão serial, T_p é o tempo de execução do programa paralelo sobre p processadores e S_p (*speedup)* é o ganho de desempenho. Com relação ao ganho *(S_p)*, o caso geral ocorre quando $1 < S_p < p$ (sublinear). Porém, podem ocorrer outros casos como $S_p < 1$ (*slowdown*), indicando que a versão serial é mais eficiente que a versão paralela, $S_p = p$ (linear), representando a situação ideal em que não há sobrecarga e, por fim, $S_p > p$ (supra linear), porém, esses dois últimos, quase sempre são casos hipotéticos, pois a maioria das soluções paralelas introduz alguma sobrecarga produto da distribuição de carga e da comunicação entre processos (ALMEIDA; HONDA; LIMA, 2012; FERREIRA, 2012).

Já a medida de eficiência de um programa paralelo, informalmente, pode se dizer que é a fração de tempo que os processadores usam durante a execução da computação efetiva, representando a qualidade da aplicação através do grau de aproveitamento dos recursos e é obtido através da razão entre o *speedup* e o número de processadores utilizados, isto é:

$$E_p = \frac{S_p}{p} \tag{37}$$

em que S_p é o *speedup*, p é o número de processadores utilizados e E_p é a medida de eficiência. O ideal é que $E_p = 1$ (linear), indicando que o ganho é proporcional ao número de processadores utilizados, no entanto, em casos gerais encontra-se $1/p < E_p < 1$ (sub linear) (FERREIRA, 2012; JUNIOR, 2003). Pode-se ainda encontrar valores extremistas para eficiência como $E_p < 1/p$ (*slowdown*) e $E_p > 1$ (supra linear) (ALMEIDA; HONDA; LIMA, 2012; FERREIRA, 2012).

5.1 Análise de desempenho da versão paralelizada com OpenMP

Esta subseção apresenta a análise de desempenho da versão do PhyML paralelizada com OpenMP. Essa análise (tempos de execução, *speedup* e eficiência) foi realizada tanto em relação às funções paralelizadas (por função) quanto em relação ao PhyML como um todo, com resultados de simulações sobre conjunto de dados de DNA e proteínas na arquitetura de nós comuns do CACAU.

5.1.1 Análise de desempenho das funções gargalos do PhyML

As simulações realizadas no PhyML paralelizado com OpenMP e descritas nesta subseção (especificamente) tiveram por objetivo identificar se, individualmente, cada função paralelizada estaria colaborando para um melhor desempenho do PhyML e, caso a função paralelizada não esteja cooperando em termos de ganho, a paralelização seria então removida da função. Neste sentido, as funções gargalos paralelizadas foram simuladas com sequências de DNA e proteínas em que os valores de *speedup* e eficiência em relação ao tempo médio de execução de cada função foram analisados. Ainda com relação aos valores de tempos médios de execução usados como referência para obter os valores de *speedup* e eficiência, para cada arquivo de sequências trabalhou-se com a média e desvio padrão sobre 20 execuções para cada arquivo de entrada.

Para sequências de DNA foram analisadas as funções *LK_Core*, *Update_P_LK_Nucl* e a função *Find_Mutual_Direction*. Já para sequências de proteínas foram analisadas as funções *LK_Core*, *Update_P_LK_AA* e a função *PMat_Empirical*. Nesta subseção não serão discutidos de forma minuciosa os valores de *speedup* e eficiência, pois o objetivo é apenas identificar se a função colabora ou não com um melhor desempenho do programa, e os valores de *speedup* e eficiência por si só já são intuitivos para essa análise preliminar. Em

contrapartida à análise superficial desses valores em relação ao desempenho individual das funções, uma análise detalhada dos valores de *speedup* e eficiência será realizada na análise de desempenho do PhyML como um todo.

Com relação aos tempos médios de execução da função *LK_Core*, a Figura 44 e Figura 45 a seguir ilustram, respectivamente, o *speedup* e a eficiência com pequenos arquivos de sequências de DNA.

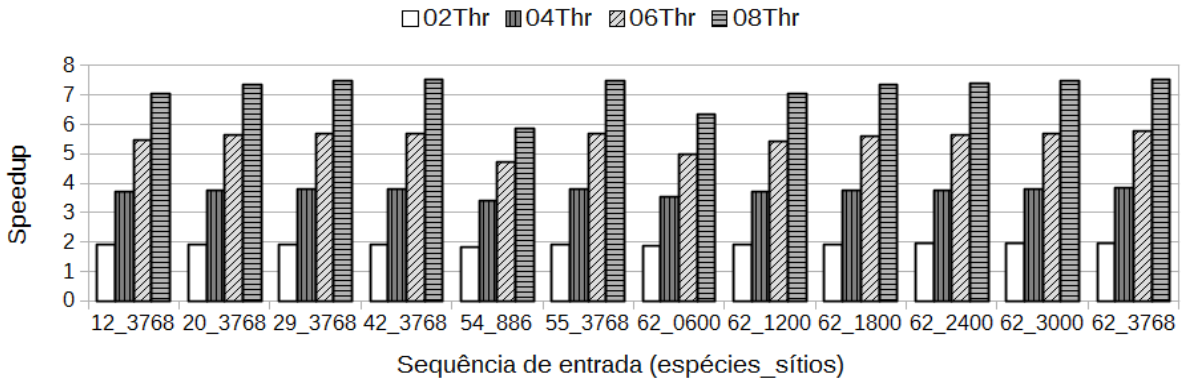

Figura 44 - *Speedup* da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.7, 5.5 e 7.2.

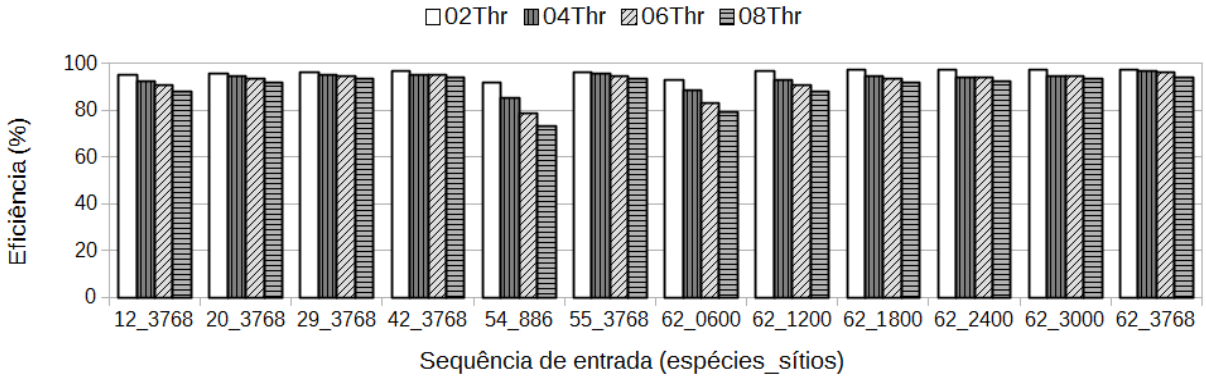

Figura 45 - Eficiência da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 96%, 93.4%, 91.8% e 89.6%.

Nota-se (Figura 44) que a função *LK_Core* quando simulada com pequenos arquivos de sequências de DNA apresenta ganho com a paralelização, sendo este ganho próximo do linear com 2, 4, 6 e 8 threads (conforme valores na legenda da

Figura 44). Além disso, a qualidade da paralelização dessa função é excelente, com média de eficiência variando de 89.6% à 96% (conforme Figura 45). Ainda com relação à função *LK_Core*, a Figura 46 e Figura 47 a seguir ilustram, respectivamente, o *speedup* e a eficiência com grandes arquivos de sequências de DNA.

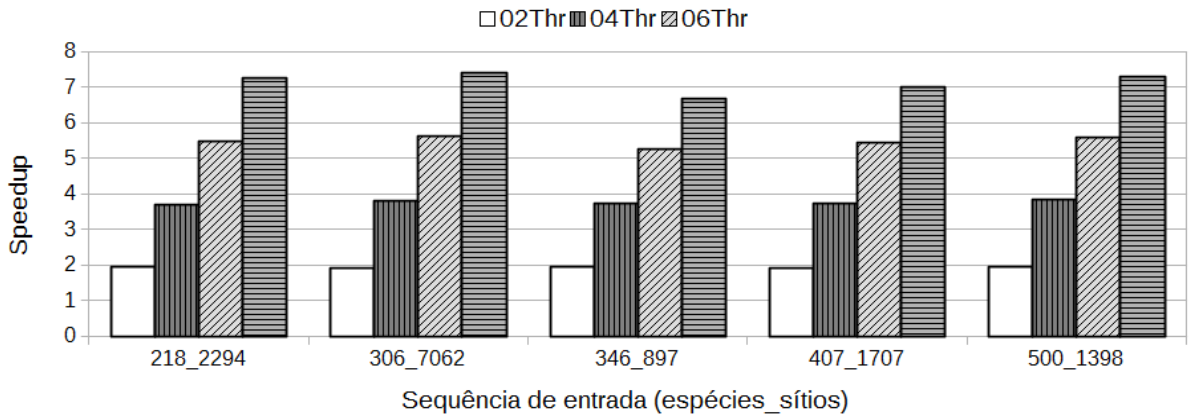

Figura 46 - *Speedup* da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 2.0, 3.8, 5.5 e 7.1.

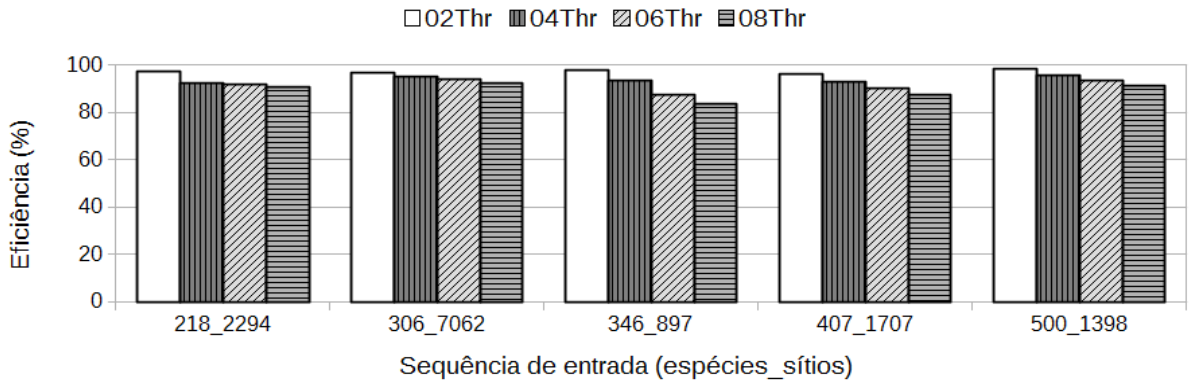

Figura 47 - Eficiência da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 97.5%, 94.2%, 91.5% e 89.3%.

O desempenho da função *LK_Core* com grandes aquivos de sequências de DNA manteve o mesmo comportamento apresentado com os pequenos arquivos, com *speedup* médio linear usando 2 threads e *speedup* próximo do linear com 4, 6 e 8 threads (Figura 46) e uma excelente qualidade da paralelização, com média de

eficiência variando de 89.3% à 97.5% (conforme Figura 47), indicando que esta função, em ambos os casos (pequenos e grandes arquivos de sequências), de fato contribui com um melhor desempenho do PhyML com dados de DNA.

Já com relação à análise de desempenho da função *Update_P_LK_Nucl*, a Figura 48 e Figura 49 a seguir ilustram, respectivamente, o *speedup* e a eficiência com pequenos arquivos de sequências de DNA.

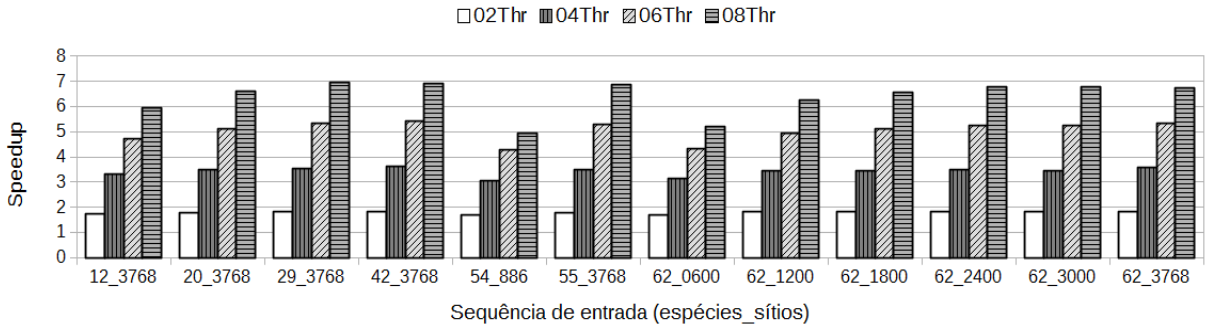

Figura 48 - *Speedup* da análise de desempenho da função *Update_P_LK_Nucl* com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.8, 3.4, 5.0 e 6.4.

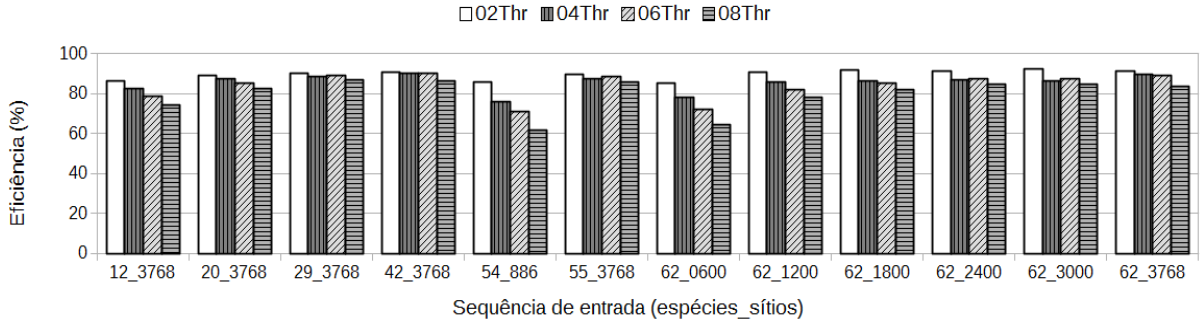

Figura 49 - Eficiência da análise de desempenho da função *Update_P_LK_Nucl* com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 89.7%, 85.7%, 84% e 79.8%.

De acordo com a Figura 48 e Figura 49, nota-se que com pequenos arquivos de sequências de DNA, a função *Update_P_LK_Nucl* apresenta um execlente desempenho com *speedup* próximo do linear (conforme valores da média de *speedup* na legenda da Figura 48) bem como uma ótima qualidade na paralelização, com eficiência média variando de 79.8% à 89.7%. Já com relação à análise de

desempenho da função *Update_P_LK_Nucl* com grandes arquivos de sequências de DNA, a Figura 50 e Figura 51 a seguir ilustram, respectivamente, o *speedup* e a eficiência.

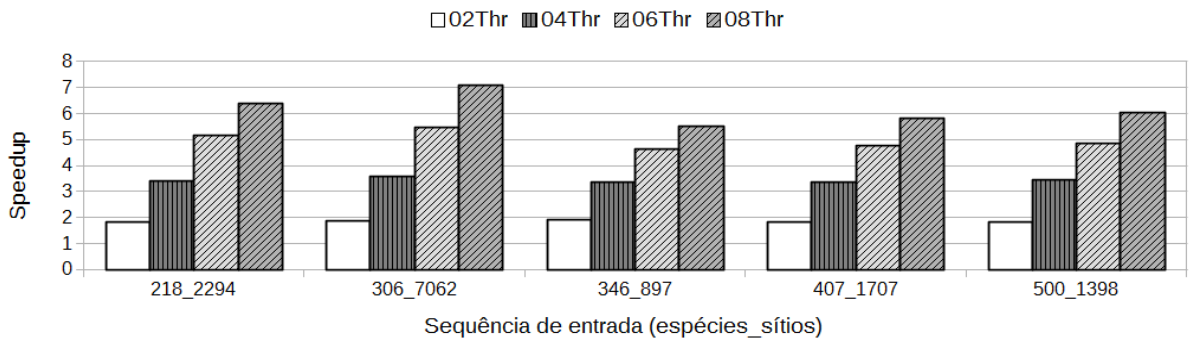

Figura 50 - *Speedup* da análise de desempenho da função *Update_P_LK_Nucl* com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.4, 5.0 e 6.2.

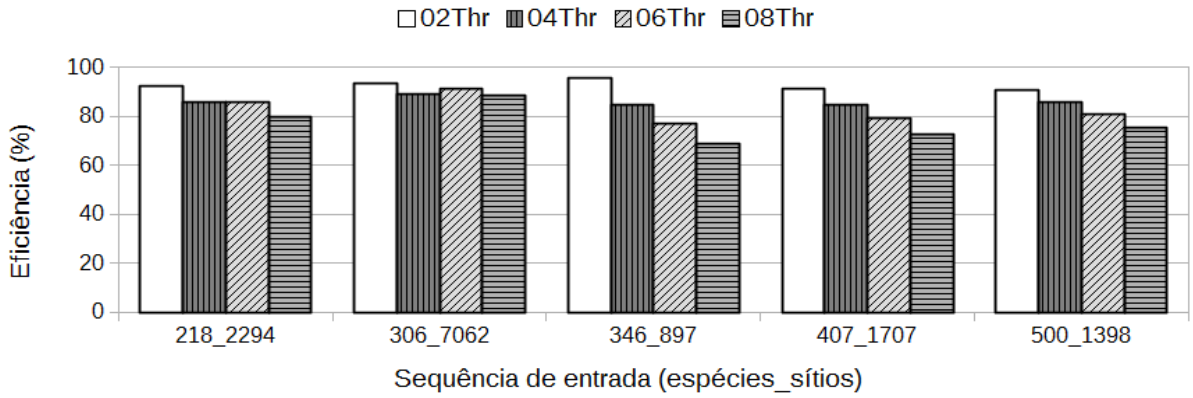

Figura 51 - Eficiência da análise de desempenho da função *Update_P_LK_Nucl* com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 92.8%, 86.1%, 83.1% e 77%.

A análise de desempenho da função *Update_P_LK_Nucl* com grandes arquivos de sequências de DNA (Figura 50 e Figura 51) revelou que o ganho continua próximo do linear (conforme valores da média de *speedup* na legenda da Figura 50) e que a qualidade se mantem ótima também com grandes dados de DNA (com eficiência média variando de 77% à 92.8%), ou seja, a função *Update_P_LK_Nucl* contribui para um melhor desempenho do PhyML tanto com pequenos quanto com grandes arquivos de sequências de DNA.

Considerando ainda dados de DNA, para a análise de desempenho da função *Find_Mutual_Direction*, a Figura 52 e Figura 53 a seguir ilustram, respectivamente, o *speedup* e a eficiência com pequenos arquivos de sequências de DNA.

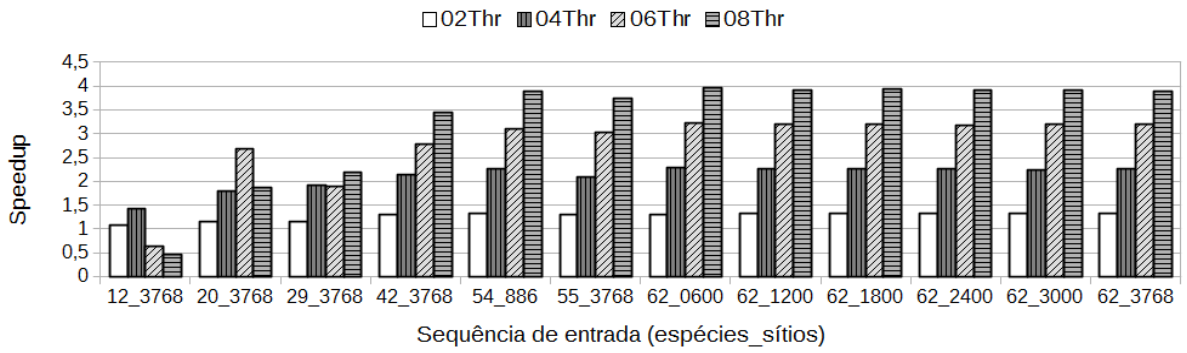

Figura 52 - *Speedup* da análise de desempenho da função *Find_Mutual_Direction* com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.3, 2.1, 2.8 e 3.3.

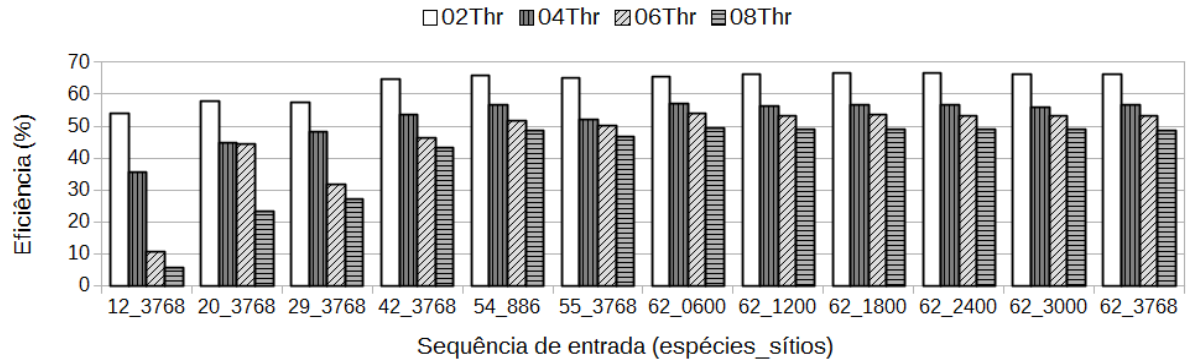

Figura 53 - Eficiência da análise de desempenho da função *Find_Mutual_Direction* com relação ao tempo médio de execução com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 63.7%, 52.6%, 46.3% e 40.8%.

De acordo com a Figura 52 e Figura 53, com pequenos dados de DNA, a função *Find_Mutual_Direction* contribui com o desempenho do PhyML, porém de forma não tão relevante (com média de *speedup*, no máximo 3.3), além disso, a eficiência da paralelização indica uma qualidade não muito boa, com média de eficiência variando de 40.8% à 63.7%, porém, ainda que mínimo, existe um ganho a ser avaliado. Já com relação à análise de desempenho da função

Find_Mutual_Direction com grandes arquivos de sequências de DNA, a Figura 54 e Figura 55 a seguir ilustram, respectivamente, o *speedup* e a eficiência.

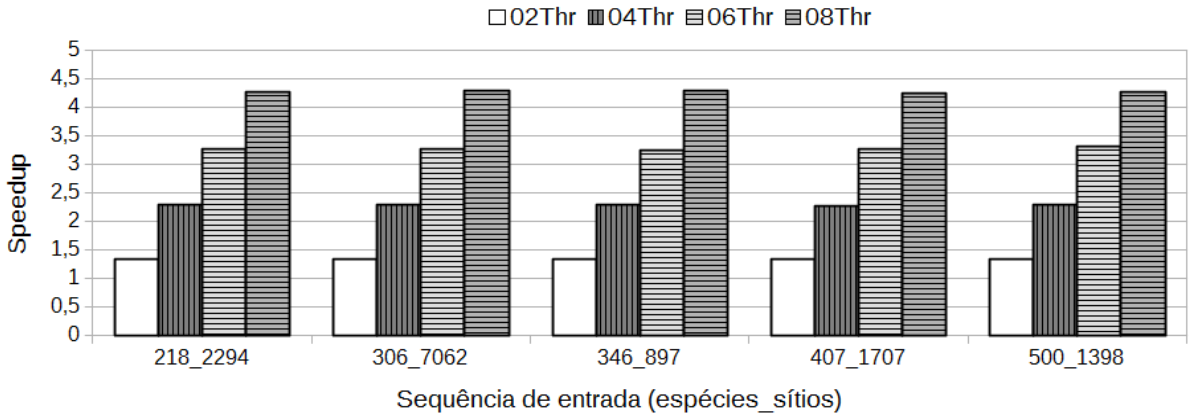

Figura 54 - *Speedup* da análise de desempenho da função *Find_Mutual_Direction* com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.3, 2.3, 3.3 e 4.3.

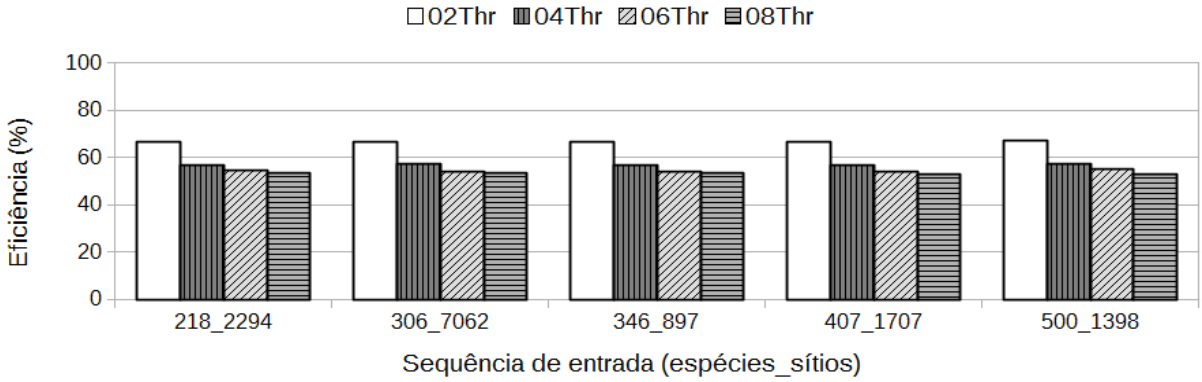

Figura 55 - Eficiência da análise de desempenho da função *Find_Mutual_Direction* com relação ao tempo médio de execução com grandes arquivos de sequências de DNA. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 66.9%, 57.2%, 54.6% e 53.4%.

Para ambos os casos (pequenos e grandes dados de DNA), a análise de desempenho (valores de *speedup* e eficiência) da função *Find_Mutual_Direction* mostrou que esta contribui para o desempnho do PhyML, porém de forma não tão significativa se comparada às outras duas funções analisadas anteriormente. Isso pode ter ocorrido em razão da variabilidade do tamanho (não previsível) dos laços de execução internos (apresentados no algoritmo dessa função). No entanto, é necessário enfatizar o ponto de vista de que durante a execução do PhyML, os

recursos já foram alocados para as demais funções paralelizadas e que, apesar da paralelização sobre esta função não ser tão eficiente, utilizar recursos que já estão alocados e encontram-se ociosos no momento de execução desta função é uma opção favorável à melhora de desempenho do PhyML, portanto, esta função se manterá paralelizada.

Com relação à análise de desempnho das funções paralelizadas com dados de proteínas, a função *LK_Core*, com pequenos dados de proteínas apresentou os seguintes valores de *speedup* e eficiência, respectivamente, ilustrados na Figura 56 e Figura 57 a seguir.

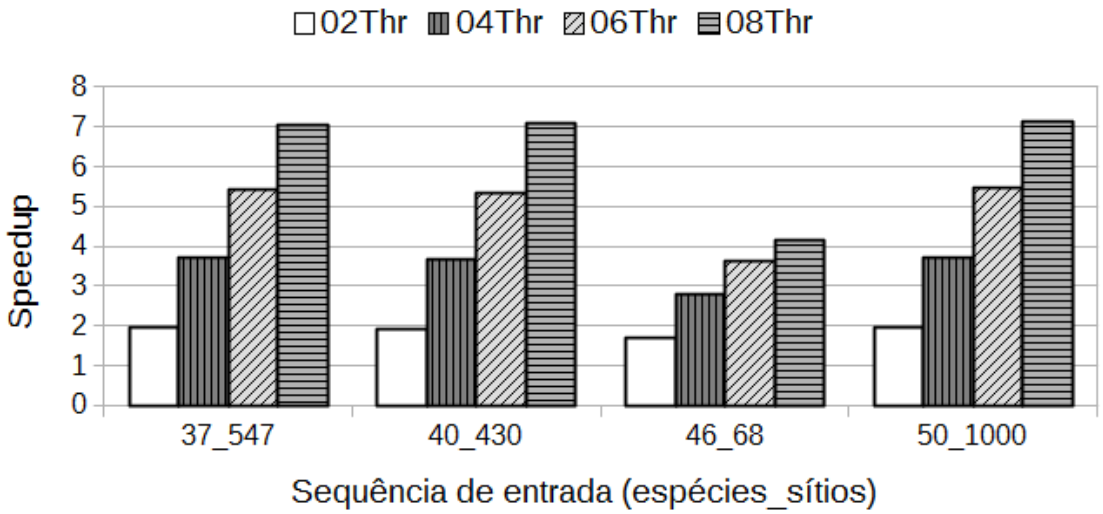

Figura 56 - *Speedup* da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.5, 5.0 e 6.4.

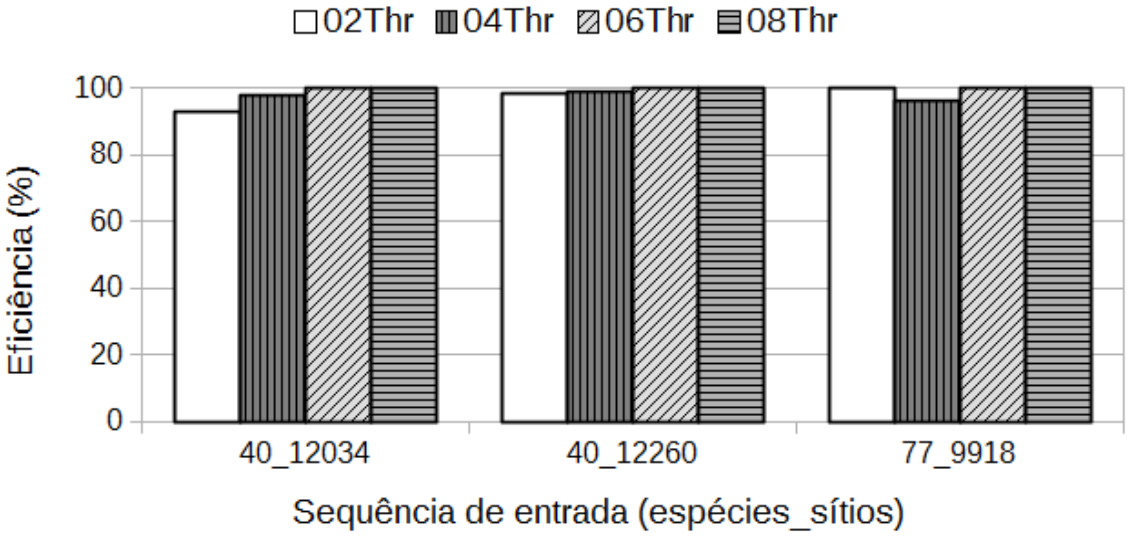

Figura 57 - Eficiência da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 94.4%, 86.9%, 82.9% e 79.6%.

Nota-se (Figura 56) que a função *LK_Core* quando simulada com pequenos arquivos de sequências de proteínas é proxima do linear com 2, 4, 6 e 8 threads (conforme valores da média de *speedup* na legenda da Figura 56) e que a qualidade da paralelização dessa função é excelente, com média de eficiência variando de 79.6% à 94.4% (Figura 57). Já com relação aos grandes dados de proteínas, a Figura 58 e Figura 59 apresentam, respectivamente o *speedup* e a eficiência da função *LK_Core*.

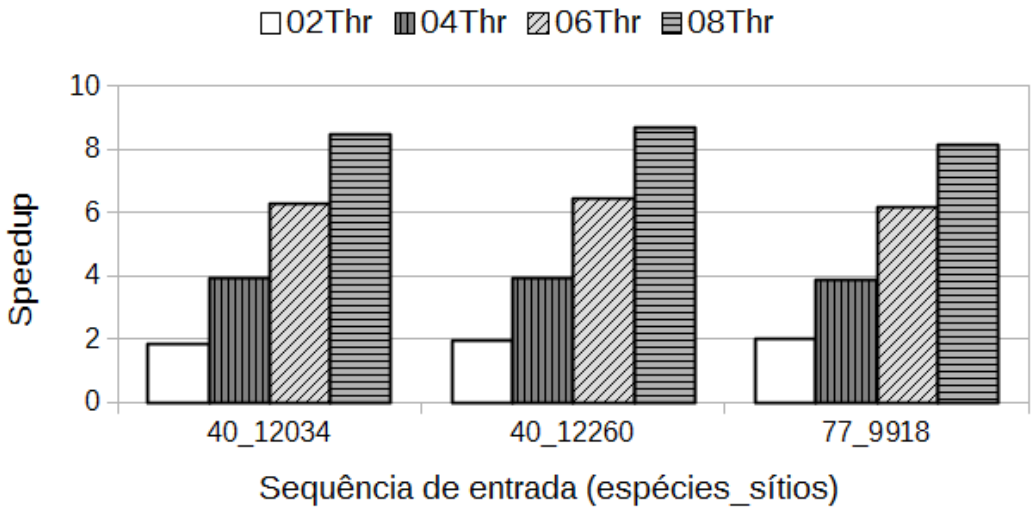

Figura 58 - *Speedup* da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 2.0, 3.9, 6.3 e 8.4.

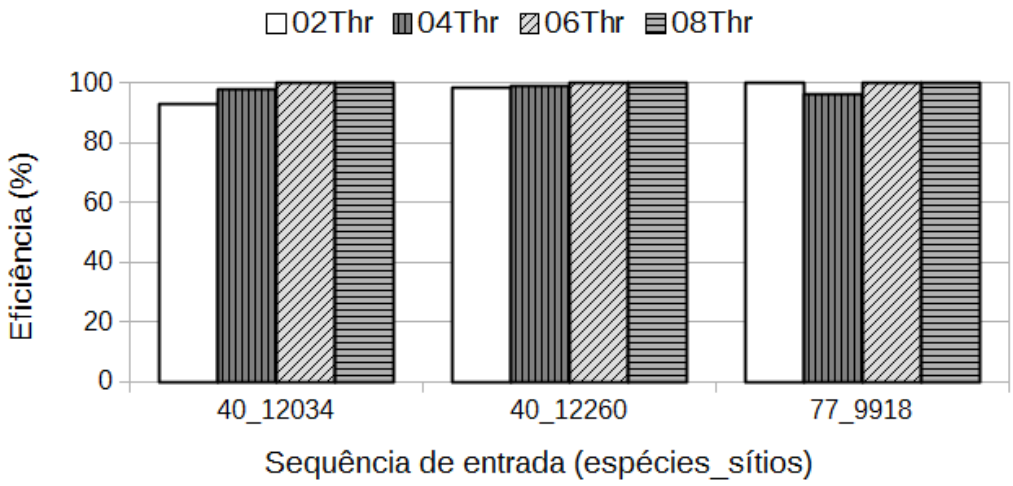

Figura 59 - Eficiência da análise de desempenho da função *LK_Core* com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 97.8%, 97.9%, 105.4% e 105.6%.

O desempenho da função *LK_Core* com grandes aquivos de sequências de proteínas obteve um comportamento ainda melhor do que o apresentado com os pequenos arquivos, com *speedup* linear ou supra linear (Figura 58) e qualidade da paralelização excelente, com média de eficiência variando de 97.8% à 105.6% (Figura 59), indicando que esta função, não somente contribui com um melhor

desempenho do PhyML com dados de proteínas, como também fornece uma paralelização de excelente qualidade.

Já com relação à análise de desempenho da função *Update_P_LK_AA*, específica para dados de proteínas, a Figura 73 e Figura 74 a seguir ilustram, respectivamente, o *speedup* e a eficiência com pequenos arquivos de sequências de proteínas.

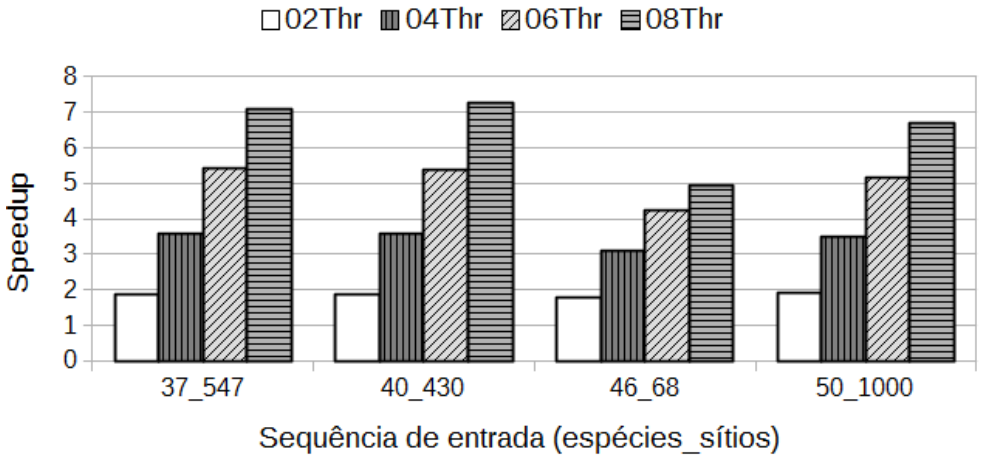

Figura 60 - *Speedup* da análise de desempenho da função *Update_P_LK_AA* com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.5, 5.1 e 6.5.

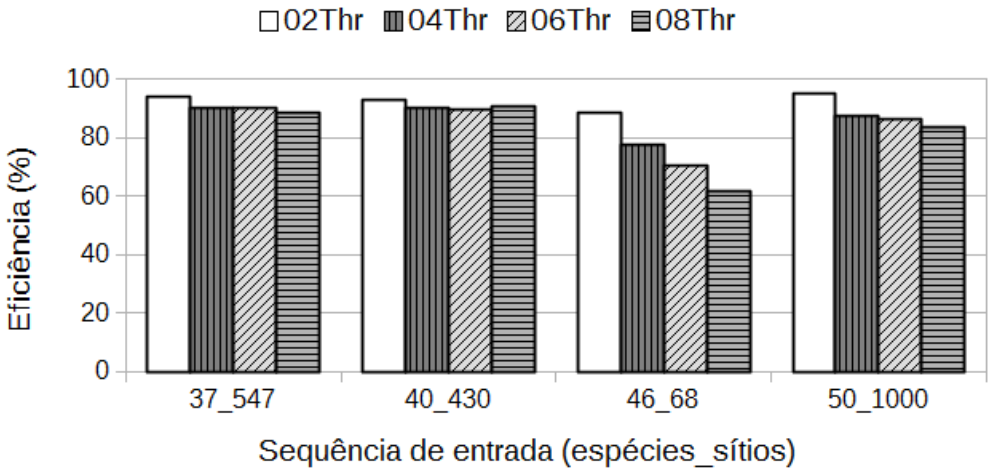

Figura 61 - Eficiência da análise de desempenho da função *Update_P_LK_AA* com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 92.9%, 86.5%, 84.3% e 81.2%.

De acordo com a Figura 73 e Figura 74, nota-se que com pequenos arquivos de sequências de proteínas, a função *Update_P_LK_AA* apresenta um execlente desempenho com *speedup* próximo do linear bem como uma excelente qualidade na paralelização, com média de eficiência variando de 81.2% à 92.9%. Com relação à análise de desempenho da função *Update_P_LK_AA* com grandes arquivos de sequências de proteínas, a Figura 75 e Figura 76 a seguir ilustram, respectivamente, o *speedup* e a eficiência.

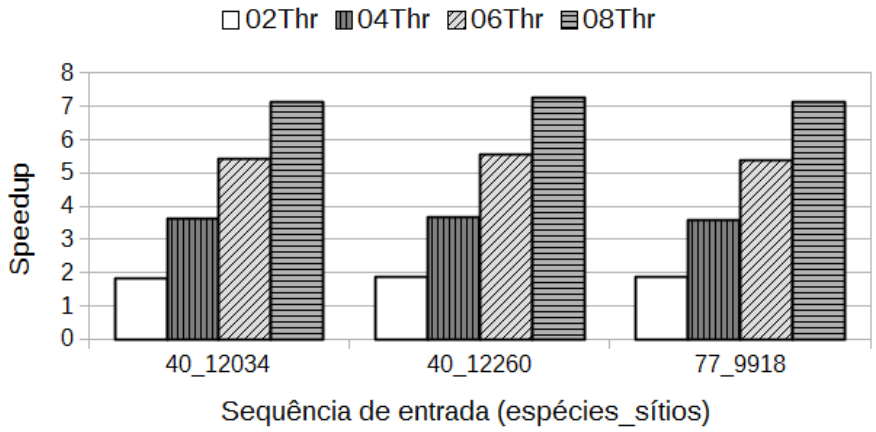

Figura 62 - *Speedup* da análise de desempenho da função *Update_P_LK_AA* com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.6, 5.5 e 7.2.

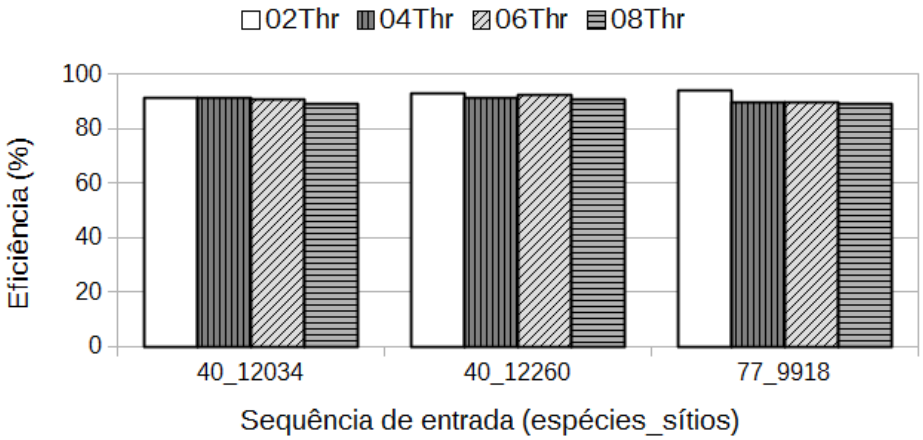

Figura 63 - Eficiência da análise de desempenho da função *Update_P_LK_AA* com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 93.0%, 90.9%, 91.0% e 89.9%.

Com grandes dados de proteínas, a função *Update_P_LK_AA* apresentou melhor desempenho do que com pequenos dados, e, da mesma forma, manteve um *speedup* próximo do linear (Figura 75), já a qualidade também se mostrou excelente com média de eficiência variando de 89.9% à 91% (Figura 76). Desta forma, a função *Update_P_LK_AA* contribui para um melhor desempenho do PhyML tanto com pequenos quanto com grandes dados de proteínas.

Por fim, será analisado o desempenho da função *PMat_Empirical*. A Figura 77 e Figura 78 a seguir ilustram, respectivamente, o *speedup* e a eficiência para pequenos dados de proteínas.

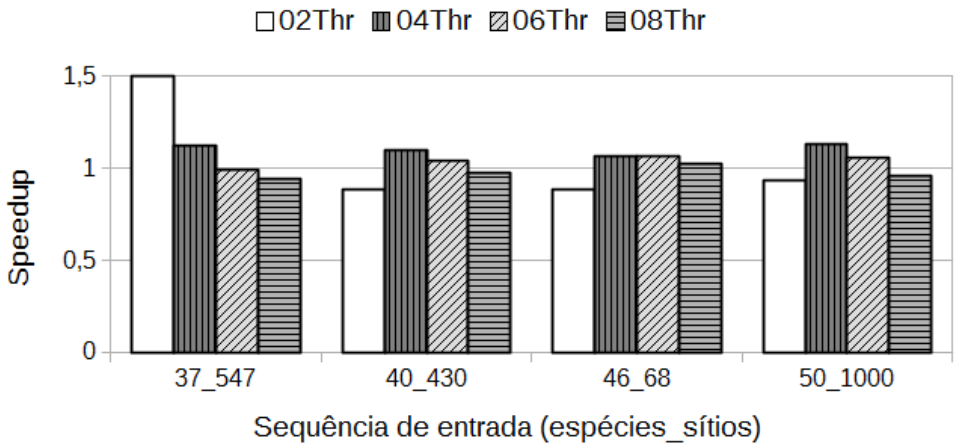

Figura 64 - *Speedup* da análise de desempenho da função *PMat_Empirical* com relação ao tempo médio de execução com pequenos arquivos de sequências de proteínas. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.1, 1.1, 1.0 e 1.0.

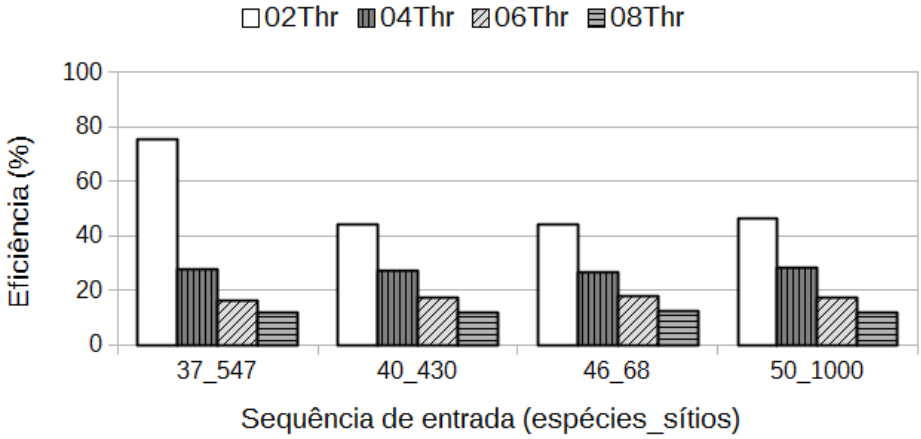

Figura 65 - Eficiência da análise de desempenho da função *PMat_Empirical* com relação ao tempo médio de execução com pequenos arquivos de

sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 52.7%, 27.7%, 17.4% e 12.2%.

A análise de desempenho da função *PMat_Empirical* com pequenos dados de proteínas revelou que esta não apresenta ganho relevante, com *speedup* em torno de 1. Além disso, a qualidade da paralelização é ruim, com média de eficiência variando de 12.2% à 52.7%. Já com relação à análise de desempenho dessa função com grandes arquivos de sequências de proteínas, a Figura 66 e Figura 67 a seguir ilustram, respectivamente, o *speedup* e a eficiência.

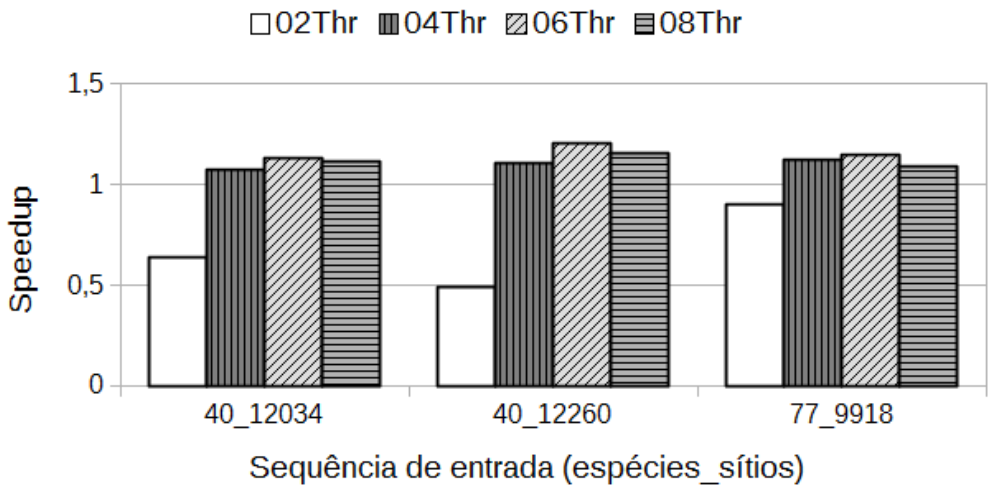

Figura 66 - *Speedup* da análise de desempenho da função *PMat_Empirical* com relação ao tempo médio de execução com grandes arquivos de sequências de proteínas. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 0.7, 1.1, 1.2 e 1.1.

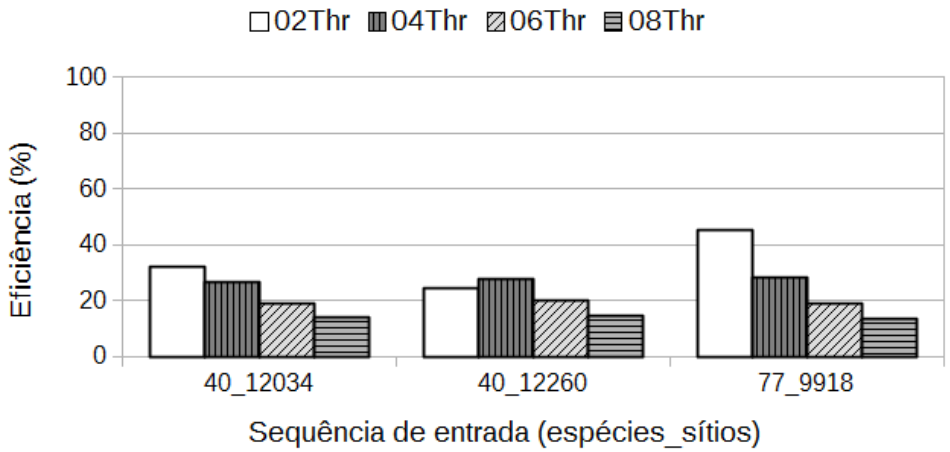

Figura 67 - Eficiência da análise de desempenho da função *PMat_Empirical* com relação ao tempo médio de execução com grandes arquivos de

sequências de proteínas. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 34.0%, 27.6%, 19.4% e 14.0%.

Com relação aos grandes dados (Figura 66 e Figura 67), o desempenho da função *PMat_Empirical* é ainda pior, tanto em relação ao *speedup* quanto eficiência. Ou seja, diferentemente das demais funções, a função *PMat_Empirical* não apresentou ganhos significativos, chegando a ter desempenho pior que o da versão serial, em alguns casos. Isso ocorreu em razão desta função apresentar tempo médio de execução muito próximo do *overhead* das diretivas de paralelização utilizadas (conforme Figura 37 da subseção 4.3 Viabilidade de paralização de funções gargalos), em que pequenas interferências relacionadas ao balanço de carga ou à comunicação/sincronização podem ter acarretado em tais perdas de desempenho, em outras palavras, restaram poucos microssegundos a serem paralelizados.

A análise de desempenho com relação aos tempos médios de cada função permitiu identificar que, das funções gargalos, a função *PMat_Empirical* não pode ser paralelizada em razão de apresentar perda de desempenho e, desta forma, contribuir negativamente para o desempenho do PhyML. Neste sentido, a análise de desempenho do programa PhyML será realizada com as funções *LK_Core*, *Update_P_LK_Nucl* e *Find_Mutual_Direction* paralelizadas para arquivos de sequências de DNA e, com as funções *LK_Core* e *Update_P_LK_AA* paralelizadas para sequências de proteínas.

5.1.2 Análise de desempenho do tempo total de execução do PhyML

Esta subseção apresenta a análise de desempenho do PhyML paralelizado com OpenMP com resultados tanto para sequências de DNA quanto para sequências de proteínas. As relações de ganhos serão discutidas com relação aos valores de *speedup* e eficiência, os quais serão medidos para pequenos e grandes arquivos de DNA e proteínas. Ainda com relação aos valores de eficiência, como em casos gerais ocorre de esta ser uma eficiência sub linear, a avaliação de eficiência nesta subseção, especificamente, será realizada de forma empírica, pois não há um fator que permita a comparação do quão boa ou ruim é a porcentagem de eficiência

sub linear obtida. Desta forma, uma eficiência de até 60% será considerada ruim, de 60% até 75% será considerada aceitável, de 75% à 85% será considerda ótima e acima de 85% será considerada excelente. Da mesma forma descrita na subseção anterior, os valores referentes aos tempos de execução ilustrados nas figuras desta subseção (5.1.2 Análise de desempenho do tempo total de execução do PhyML) são referentes à média e desvio padrão sobre 20 execuções do mesmo arquivo de sequências para cada arquivo de entrada. Ainda no escopo desta subseção, considere que os valores das simulações referentes a 10 e 12 threads foram medidos sobre a arquitetura das placas gráficas GPU, inclusive os tempos da versão serial a que estes valores são comparados.

A Figura 68, Figura 69 e Figura 70 ilustram a análise de desempenho do PhyML para pequenos arquivos de sequências de DNA usando até 8 threads, representando respectivamente, o tempo de execução do PhyML, o *speedup* e a eficiência.

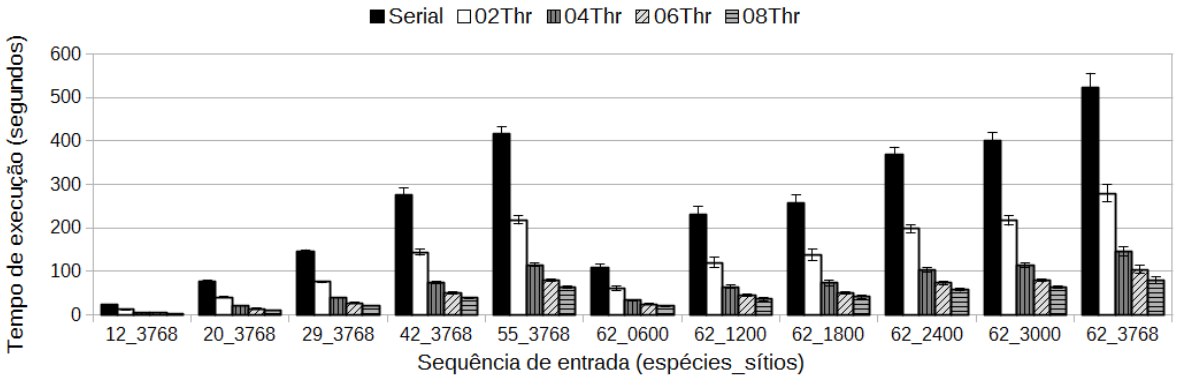

Figura 68 - Análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando até 8 threads.

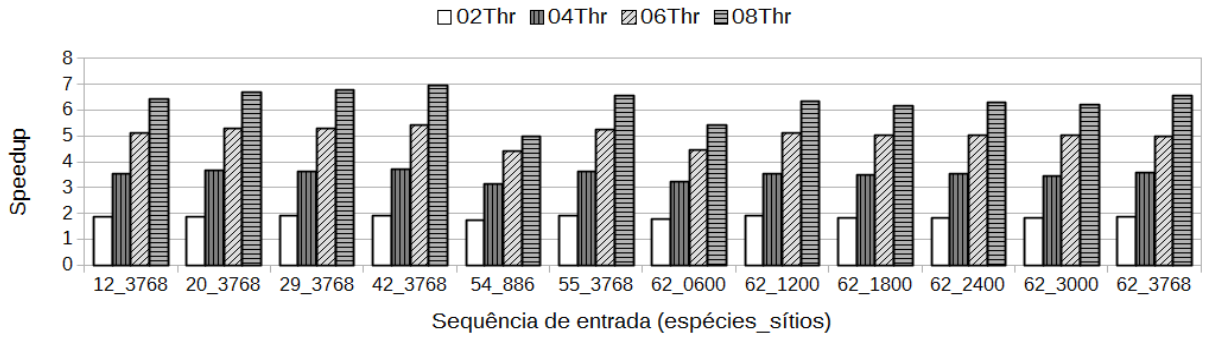

Figura 69 - *Speedup* da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando até 8 threads. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.9, 3.5, 5.0 e 6.3.

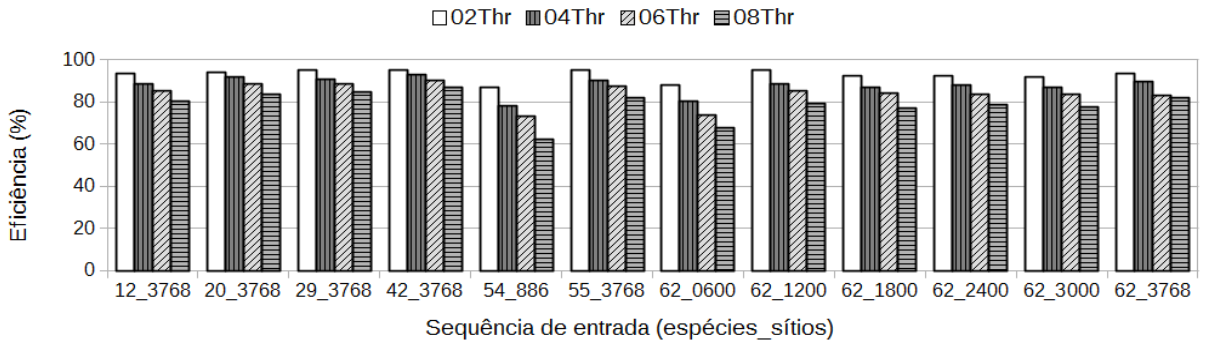

Figura 70 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 92.9%, 87.9%, 84.1% e 78.7%.

A ilustração da Figura 68 fornece o tempo de execução, através da qual é possível notar a existência de uma melhora do desempenho das simulações paralelas em relação à simulação da versão serial, porém, não é possível somente com estas informações quantificar com exatidão o ganho obtido. Em contrapartida, a ilustração da Figura 69 quantifica com base nas informações da Figura 68 quantas vezes as versões paralelas executaram mais rápido que a versão serial, através da qual é possivel notar que com 8 threads, obteve-se média de *speedup* igual a 6.3, indicando que, na maioria dos casos, a versão paralela do PhyML com 8 threads e pequenos arquivos de sequências de DNA chega a ser mais de 6 vezes mais rápida do que a versão serial. Ainda com relação à Figura 69, note que em nenhum dos casos (arquivo de sequências e quantidade de threads) houve *speedup* linear ($s_p = p$), pois, como foi justificado na subseção 4.3 (Viabilidade de paralização de funções gargalos), há *overhead* na inserção das diretivas de paralelização, bem como pode

existir *overhead* com comunicação e sincronização. Além disso, somente as partes mais custosas do PhyML foram paralelizadas, restando trechos na versão paralela que continuam executando de forma serial, contribuindo para o *speedup* não linear. Apesar disso, em todos os casos ilustrados na Figura 69, o *speedup* foi próximo do linear, conforme valores de média de *speedup* mostrados na legenda. Já a Figura 70 ilustra a eficiência no uso dos recursos, nesse caso, pode-se notar que a pior eficiência ocorreu quando usado 8 threads (média de eficiência igual a 78.7%), mas que, apesar disso, a qualidade da paralelização está de ótima a excelente, com média de eficiência variando de 78.7% à 92.9%. Já a Figura 71, Figura 72 e Figura 73, a seguir, ilustram a análise de desempenho do PhyML para pequenos arquivos de sequências de DNA, mas usando 10 e 12 threads. Da mesma forma, as ilustrações a seguir, representam respectivamente, o tempo de execução do PhyML (medidos sobre a arquitetura das placas gráficas GPU), o *speedup* e a eficiência.

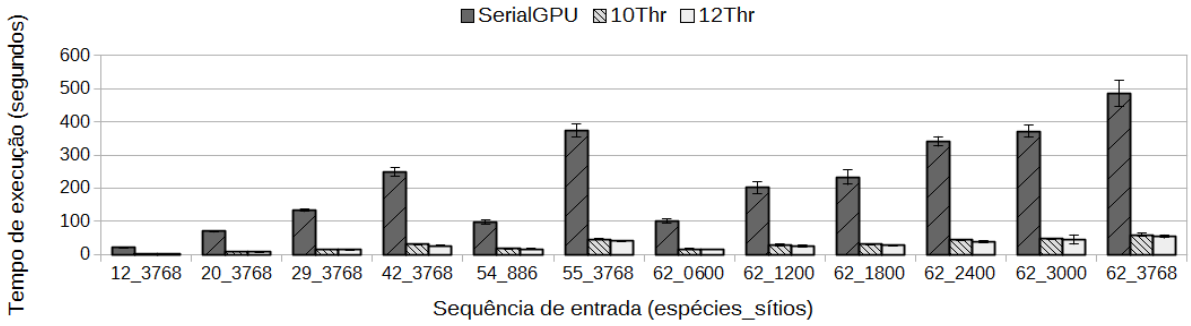

Figura 71 - Análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando 10 e 12 threads.

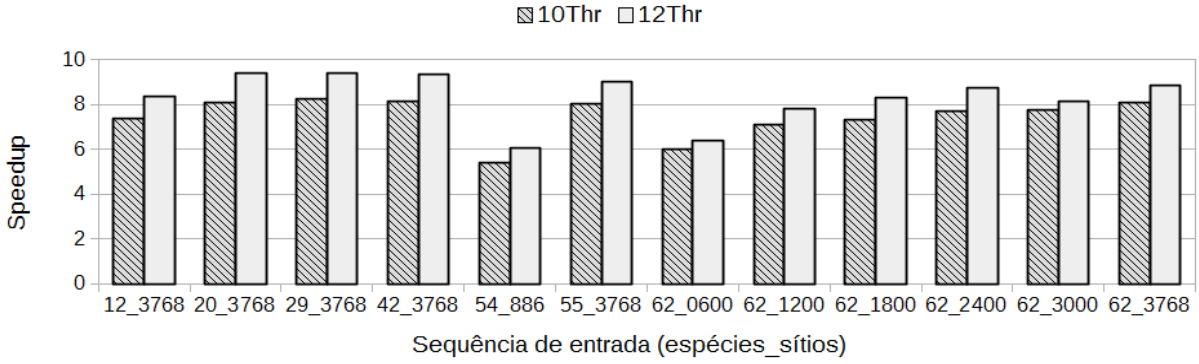

Figura 72 - *Speedup* da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA usando 10 e 12 threads. Linearidade: média de *speedup* com 10 e 12 threads respectivamente, 7.5 e 8.3.

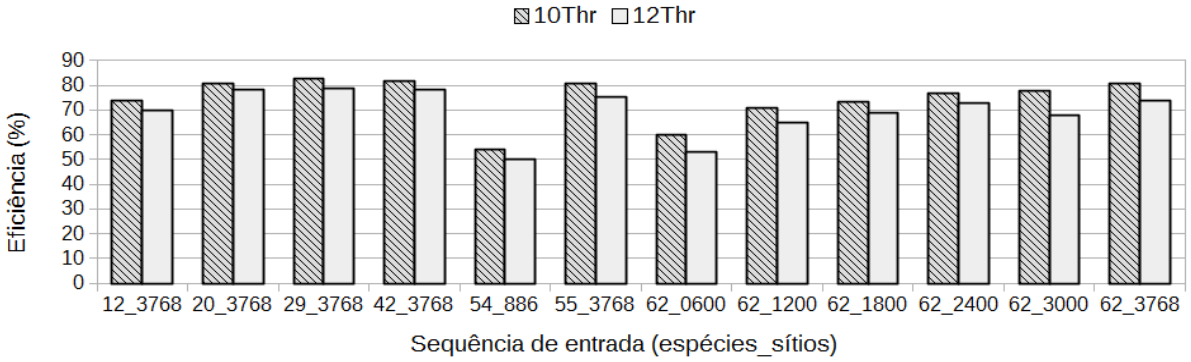

Figura 73 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de DNA com 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 74.5% e 69.4%.

Usando 10 threads, com pequenos arquivos de sequências de DNA, conforme ilustração da Figura 72, a media de *speedup* é igual a 7.5, indicando que a versão paralela, na maioria dos casos, é mais de 7 vezes mais rápida que a versão serial, enquanto com 12 threads, a versão paralela tem média de *speedup* igual a 8.3, isto é, mais de 8 vezes mais rápida que a versão serial. Da mesma forma, não houve casos de *speedup* linear, pelas mesmas razões explicitadas anteriormente, e neste caso, considerando um maior *overhead* de sincronização em razão da maior quantidade de threads envolvidas no processamento. Já a eficiência ilustrada na Figura 73, indicou que há menor eficiência quando usando 12 threads do que quando usando 10 threads, indicando uma saturação em relação à quantidade de processadores utilizados, apesar disso, a eficiência ainda é aceitável com média de 74.5% e 69.4%, para, respectivamente, 10 e 12 threads. Esses mesmo testes foram realizados para grandes arquivos de sequências de DNA. Da mesma forma, a Figura 74, Figura 75 e Figura 76 ilustram a análise de desempenho do PhyML para grandes arquivos de sequências de DNA usando até 8 threads, assim, representando respectivamente, o tempo de execução do PhyML, o *speedup* e a eficiência.

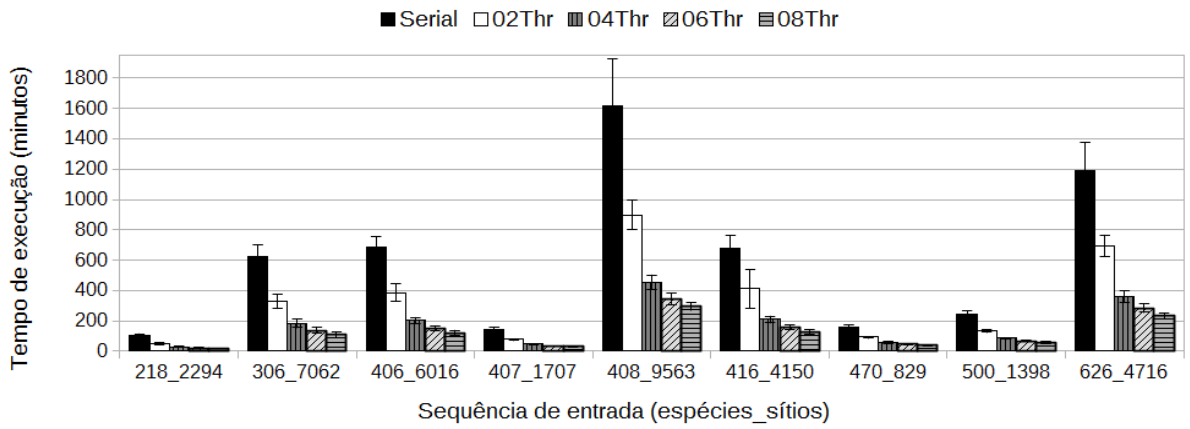

Figura 74 - Análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando até 8 threads. Os maiores desvios padrão ocorreram com o arquivo 416_4150 sendo de 30% com 02 threads, seguido pelo arquivo 408_9563 com 19% para a versão serial.

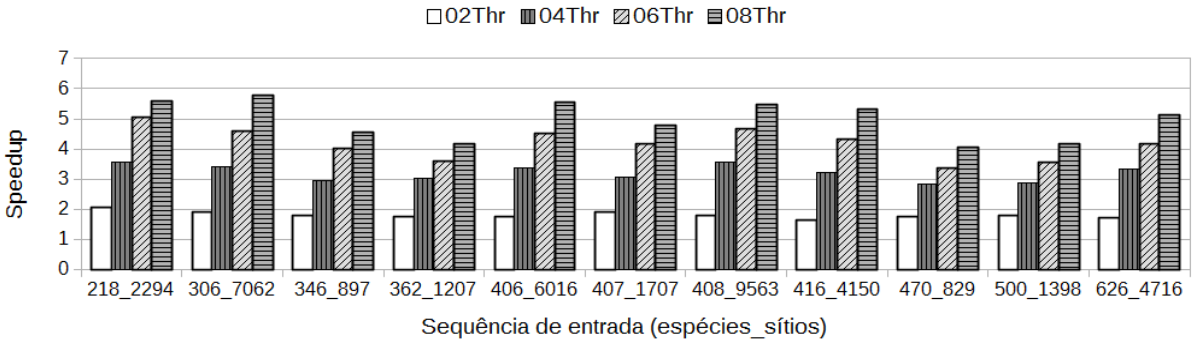

Figura 75 - *Speedup* da análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando até 8 threads. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.8, 3.2, 4.2 e 5.0.

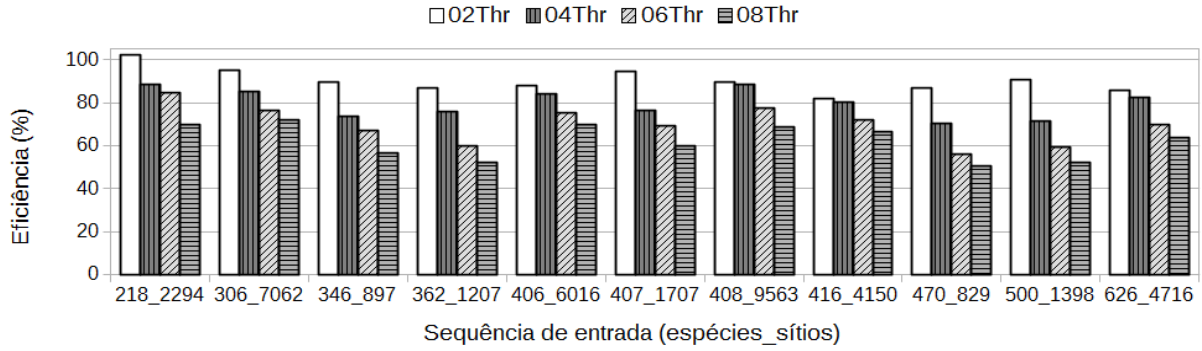

Figura 76 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de DNA com até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 90.2%, 79.9%, 69.9% e 62.1%.

Com grandes arquivos de sequências de DNA, o *speedup* foi, na maioria dos casos, menor quando comparado com os resultados para pequenos arquivos de sequências de DNA, no entanto, a versão paralela com 8 threads, na maioria dos casos, ainda conseguiu ser em média 5 vezes mais rápida que a versão serial. Para ter uma melhor noção do ganho, considerando o caso do arquivo de sequência "408_9563" que tem tempo de execução serial aproximadamente igual a 1600 minutos (Figura 74), que equivale a aproximadamente 26,6 horas, enquanto a versão paralela com 8 threads tem tempo de execução aproximadamente igual a 300 minutos (Figura 74), aproximadamente o equivalente a 5 horas, um tempo relativamente menor. Apesar de haver uma redução no *speedup*, a eficiência (Figura 76), se mostrou de aceitável a boa, com média variando de 62.1% à 90.2%.

Ainda em relação às simulações com grandes arquivos de sequências de DNA, a Figura 77, Figura 78 e Figura 79 ilustram a análise de desempenho do PhyML com 10 e 12 threads. Neste sentido, as ilustrações a seguir, representam respectivamente, o tempo de execução do PhyML, o *speedup* e a eficiência.

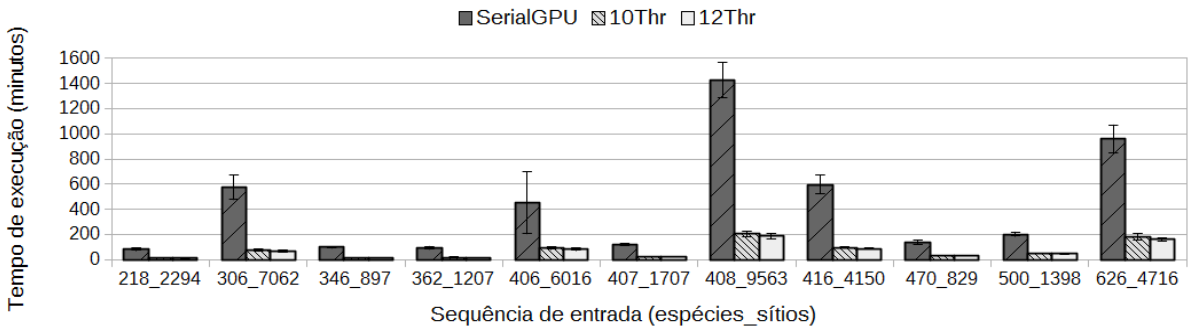

Figura 77 - Análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando 10 e 12 threads. O maior desvio padrão ocorreu para versão serial com o arquivo 406_6016 e desvio padrão de 54% seguido pelo arquivo 626_4716 com desvio padrão de 11%.

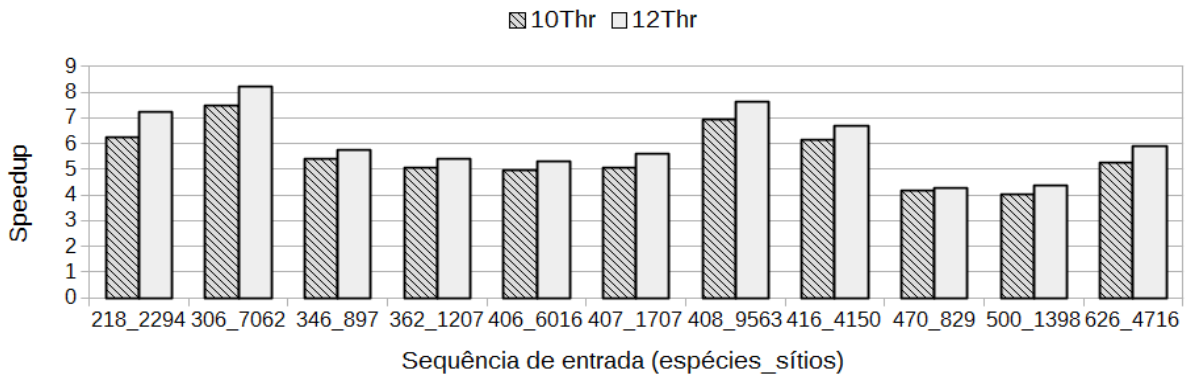

Figura 78 - *Speedup* da análise de desempenho do PhyML com grandes arquivos de sequências de DNA usando 10 e 12 threads. Linearidade: média de *speedup* com 10 e 12 threads respectivamente, 5.5 e 6.1.

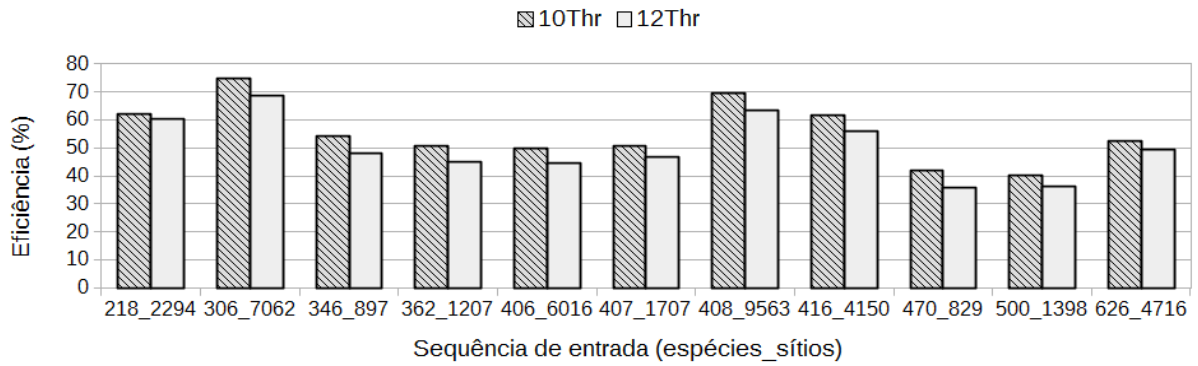

Figura 79 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de DNA com 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 55.4% e 50.4%.

Usando 10 e 12 threads com grandes arquivos de sequências de DNA, o desempenho foi pior do que com pequenos arquivos, a média de *speedup* se manteve superior a 5 (Figura 78), enquanto a qualidade foi ruim, com média de eficiência de 55.4% e 50.4% com 10 e 12 threads, respectivamente (Figura 79). Essa perda de desempenho está diretamente vinculada à saturação da aplicação paralela em relaçao à quantidade de processadores utilizados.

A análise de desempenho do PhyML também foi aplicada para pequenos e grandes arquivos de sequências de proteínas. Neste sentido, a Figura 80 ilustra o tempo de execução do PhyML usando até 8 threads com pequenos arquivos de sequências de proteínas. Já as ilustrações da Figura 81 e da Figura 82 representam, respectivamente, o *speedup* e eficiência dessas simulações.

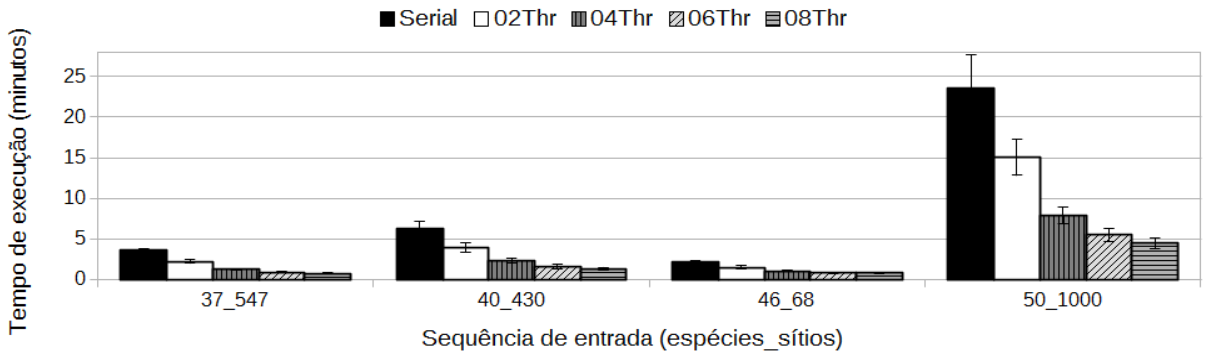

Figura 80 - Análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando até 8 threads. O maior desvio padrão ocorreu para a versão serial com o arquivo 50_1000 e desvio padrão de 19%.

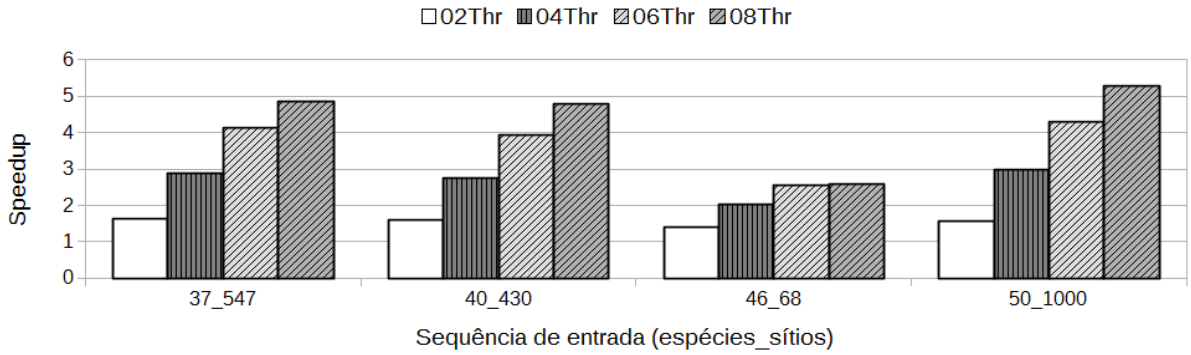

Figura 81 - *Speedup* da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando até 8 threads. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.6, 2.7, 3.7 e 4.4.

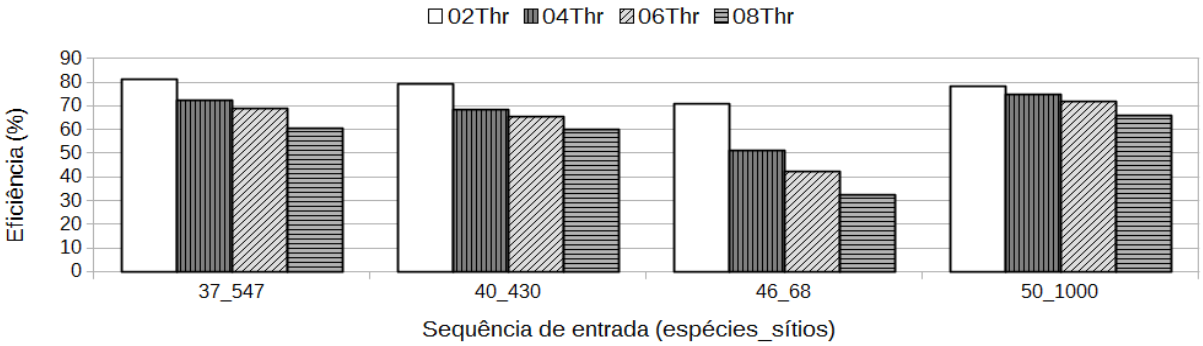

Figura 82 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 77.5%, 66.7%, 62.2% e 54.8%.

Ainda usando pequenos arquivos de sequências de proteínas, a ilustração da Figura 83 apresenta o tempo de execução do PhyML usando 10 e 12 threads, enquanto as ilustrações da Figura 84 e da Figura 85, apresentam respectivamente, o *speedup* e a eficiência destas simulações. É importante destacar que os resultados com 10 e 12 threads são referentes às simulações na GPU e, portanto, esses valores comparativos são referentes aos tempos de execução da versão serial obtidos também nesta arquitetura.

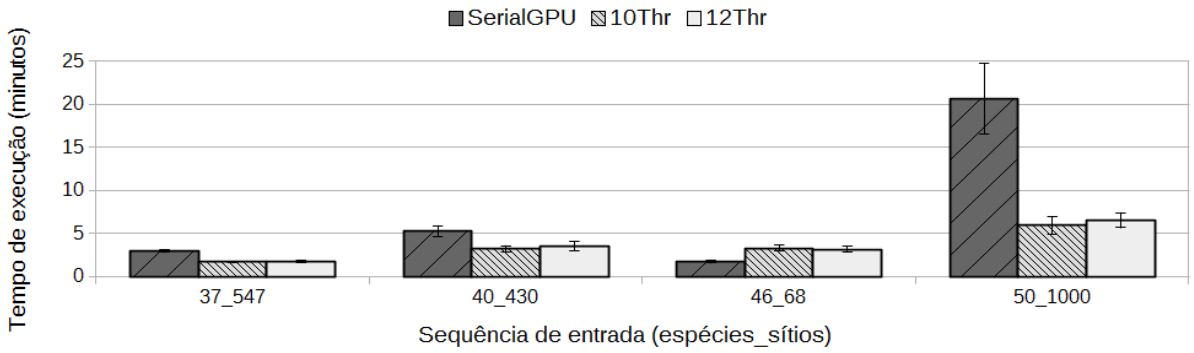

Figura 83 - Análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando 10 e 12 threads. O Maior desvio padrão ocorreu para a versão serial com o arquivo 50_1000 e desvio padrão de 19%.

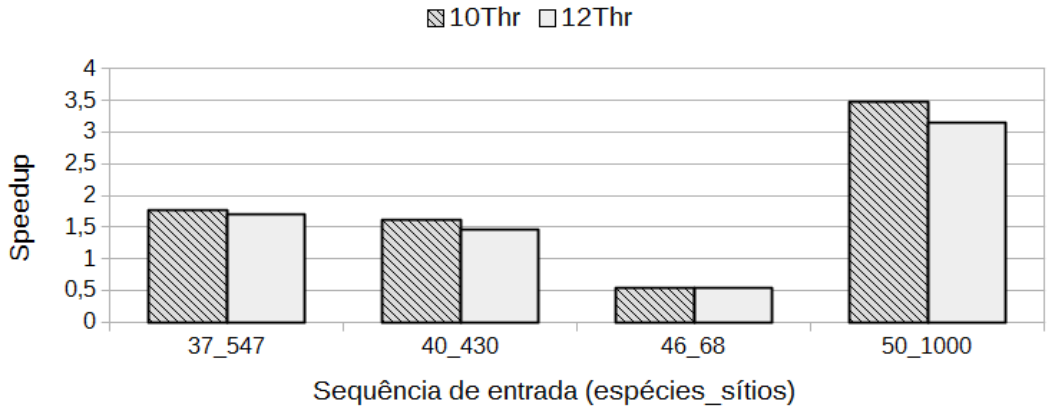

Figura 84 - *Speedup* da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando 10 e 12 threads. Linearidade: média de *speedup* com 10 e 12 threads respectivamente, 1.5 e 1.7.

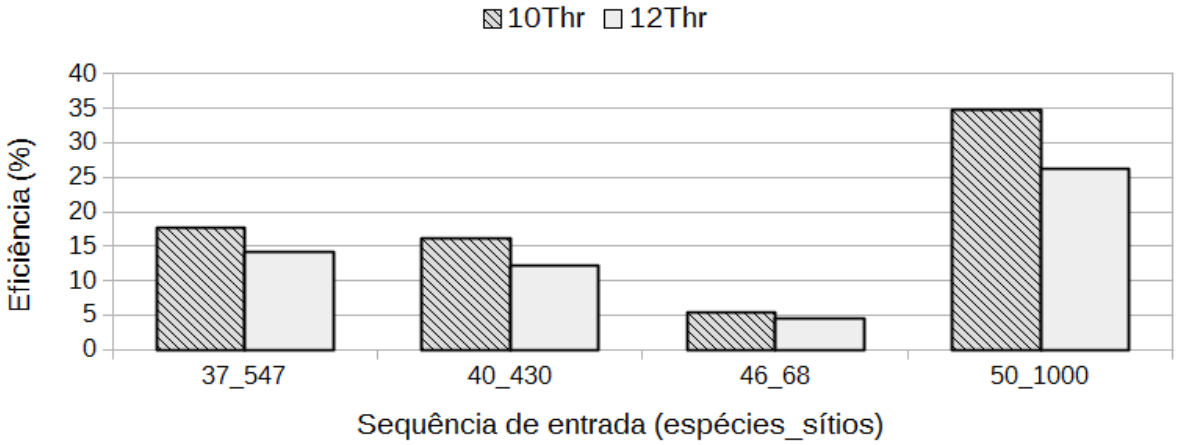

Figura 85 - Eficiência da análise de desempenho do PhyML com pequenos arquivos de sequências de proteínas usando 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 18.5% e 14.3%.

As simulações sobre pequenos arquivos de proteínas indicaram *speedup* crescente com até 8 threads, porém, com 10 e 12 threads os *speedup* sofreu uma redução brusca (Figura 84), o mesmo ocorreu em termos de eficiência (Figura 85). O melhor *speedup* ocorreu com 8 threads, com média de speedup igual a 4.7 vezes mais rápida que a versão serial. Já a eficiência com até 8 threads, ficou na média de 54.8%, uma qualidade abaixo do esperado (empiricamente, no mínimo 60%). Uma análise com grandes arquivos de sequências de proteínas podem revelar novos indicadores.

Já com relação a grandes arquivos de sequências de proteínas, a Figura 86 a seguir ilustra o tempo de execução do PhyML usando até 8 threads, enquanto as ilustrações da Figura 87 e da Figura 88 representam, respectivamente, o *speedup* e eficiência dessas simulações.

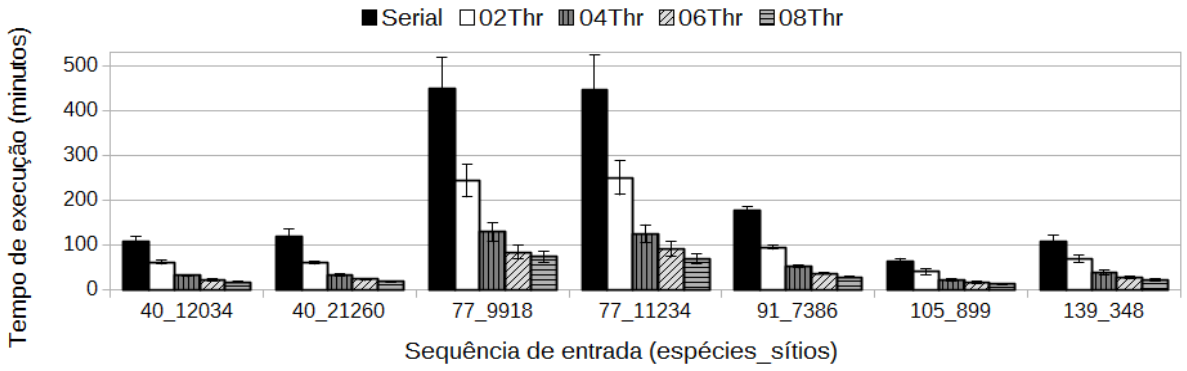

Figura 86 - Análise de desempenho do PhyML com grandes arquivos de sequências de proteínas usando até 8 threads. Os maiores desvios padrão ocorreram com a versão serial sendo de 17% com o arquivo 77_11234 e de 15% com o arquivo 77_9918.

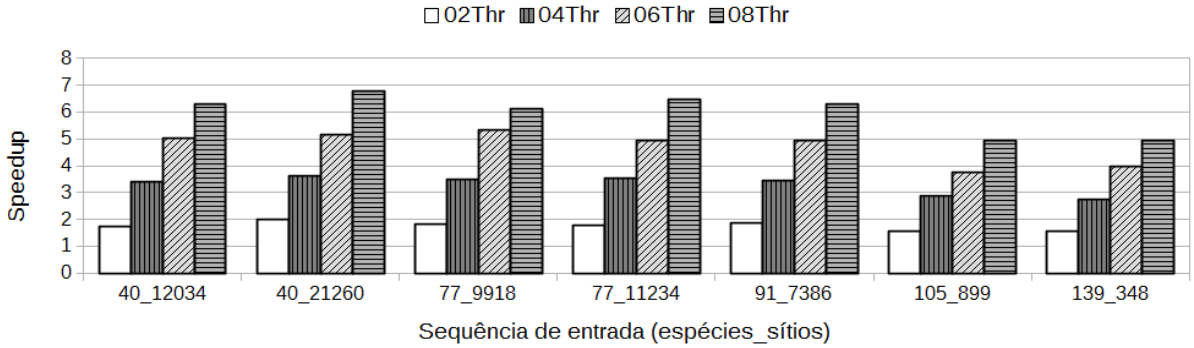

Figura 87 - *Speedup* da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com até 8 threads. Linearidade: média de *speedup* com 2, 4, 6 e 8 threads respectivamente, 1.8, 3.3, 4.7 e 6.0.

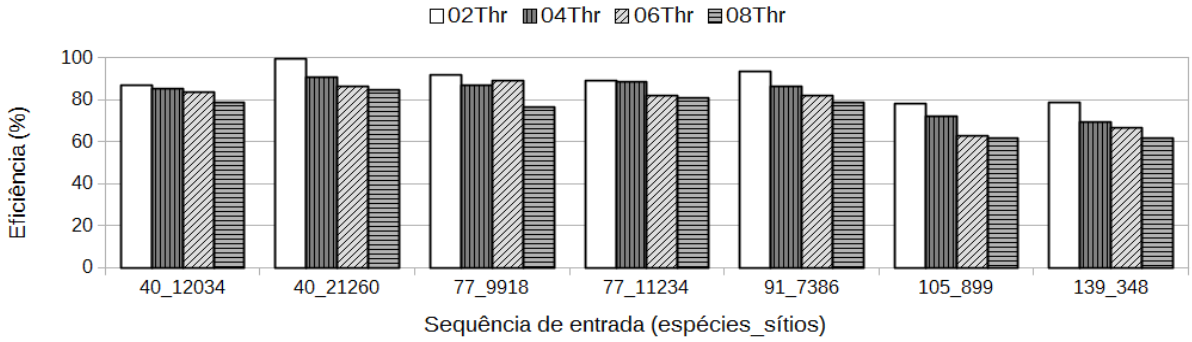

Figura 88 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com até 8 threads. Linearidade: média de eficiência com 2, 4, 6 e 8 threads respectivamente, 88.4%, 82.9%, 79.0% e 75.0%.

Da mesma forma, usando grandes arquivos de sequências de proteínas, a ilustração da Figura 89 apresenta o tempo de execução do PhyML usando 10 e 12 threads, enquanto as ilustrações da Figura 90 e da Figura 91, apresentam respectivamente, o *speedup* e a eficiência destas simulações sobre a arquitetura GPU.

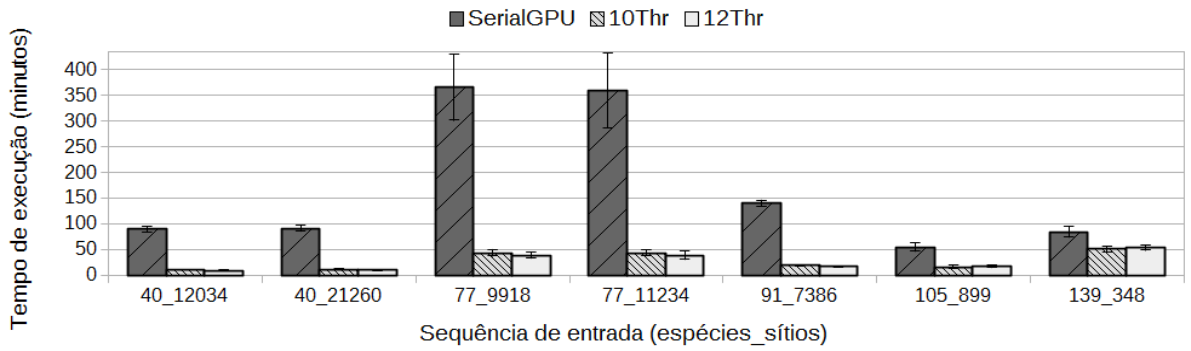

Figura 89 - Análise de desempenho do PhyML com grandes arquivos de sequências de proteínas usando 10 e 12 threads. Os maiores desvios padrão ocorreram com a versão serial sendo de 20% com o arquivo 77_11234 e de 17% com o arquivo 77_9918.

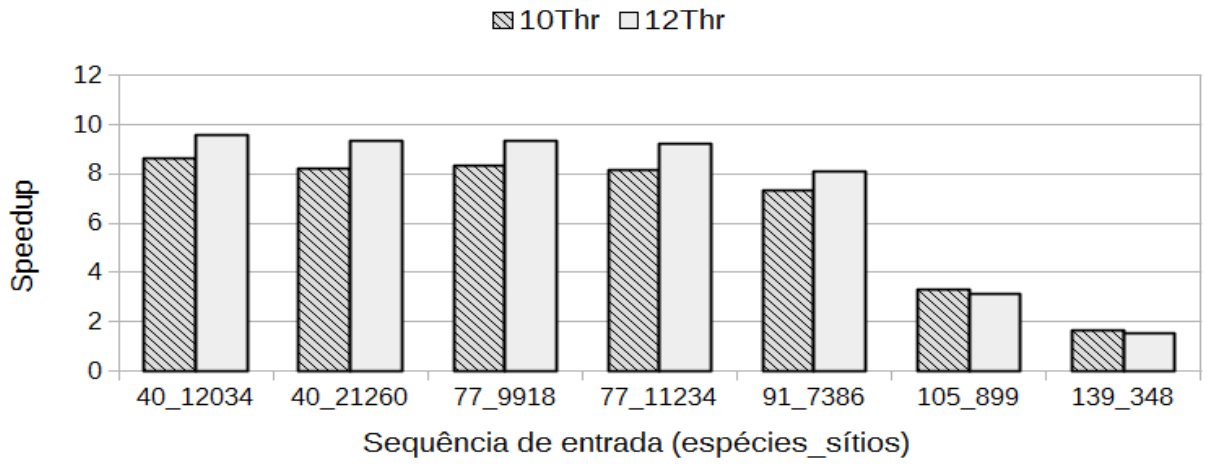

Figura 90 - *Speedup* da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com 10 e 12 threads. Linearidade: média de *speedup* com 10 e 12 threads respectivamente, 8.2 e 9.1.

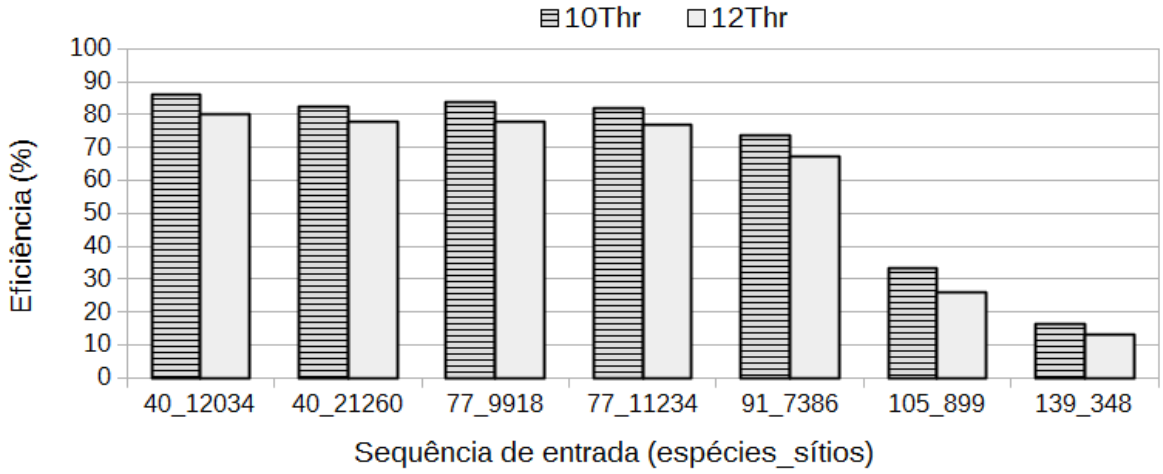

Figura 91 - Eficiência da análise de desempenho do PhyML com grandes arquivos de sequências de proteínas com 10 e 12 threads. Linearidade: média de eficiência com 10 e 12 threads respectivamente, 65.4% e 60.0%.

Com grandes arquivos de sequências de proteínas, houve uma melhora nas medidas de *speedup* e eficiência, com 8 threads a média de *speedup* foi de 6, enquanto com 10 e 12 threads a media foi de, respectivamente 8.2 e 9.1, sendo que os índices de eficiência acompanharam os valores de *speedup*, com média variando de 54.8% à 77.5% com até 8 threads e de 65.4% e 60.0% com 10 e 12 threads, respectivamente. Nesses casos, os valores de eficiência são aceitáveis, variando em torno de 60%.

No contexto geral das simulações dessa subseção, notou-se a ocorrência de algumas peculiaridades nas correlações de resultados, dentre as quais é importante destacar: (I) melhor desempenho do PhyML com pequenos arquivos em comparação com grandes arquivos de sequências de DNA; (II) melhor desempenho do PhyML com grandes arquivos em comparação com pequenos arquivos de sequências de proteínas; (III) melhor desempenho do PhyML com pequenos arquivos de sequências de DNA em comparação com pequenos arquivos de sequências de proteínas e; (IV) melhor desempenho do PhyML com grandes arquivos de sequências de proteínas em comparação com grandes arquivos de sequências de DNA.

Justificar a ocorrência de tais peculiaridades em alguns casos, principalmente quando se tratando de tipos de dados diferentes (Casos III e IV), é uma tarefa difícil, pois além de não existir uma proporcionalidade no tamanho dos arquivos de sequências para prover uma comparação justa, existe também a ocorrência de

fatores externos às funções paralelizadas, como a variação na quantidade de otimizações, que pode influenciar no tempo de execução e, consequentemente, na medida de desempenho.

Uma forma encontrada para justificar tais ocorrências foi analisar a estimativa de *speedup*, conhecida como lei de Amdahl (CHAPMAN; JOST; PAS, 2007, p. 33). A lei de Amdahl descreve como encontrar a máxima melhoria esperada (máximo *speedup*) de um sistema quando apenas parte dele é otimizado. Essa estimativa é calculada com relação à fração de tempo da parte paralelizada do algorítmo, nesse caso, com relação ao percentual de tempo de execução (fornecido pelo GProf) das funções consideradas na análise desempenho do PhyML. Como há mais de uma função paralelizada, a ideia é estimar o *speedup* com base na soma dos percentuais de tempos de execução de todas as funções paralelizadas (obtendo a contribuição total das funções paralelizadas). Neste sentido, o *speedup* estimado, de acordo com a lei de Amdahl, foi obtido conforme a eq. (38) a seguir:

$$T_p = CP_s + \frac{CP_p}{P} \tag{38}$$

em que CP_s é fração de tempo de execução (soma de percentuais de tempos de execução) das partes estritamente seriais do algorítmo, CP_p é fração de tempo de execução (soma de percentuais de tempos de execução) das partes estritamente paralelas do algorítmo e P é a quantidade de processadores usados nas simulações.

Em razão da grande quantidade de arquivos de sequências de DNA e proteínas em uso, para ambos os casos, o *speedup* estimado foi calculado considerando os valores limites (mínimo e máximo) do intervalo da distribuição de somas de percentuais de tempos de execução das funções consideradas na paralelização, no qual a intenção de usar o intervalo é generalizar a estimativa de *speedup* para a maioria dos arquivos de entrada. Neste sentido, para sequências de DNA, foram consideradas as somas de percentuais de tempos de execução das funções *LK_Core*, *Update_P_LK_Nucl* e *Find_Mutual_Direction*, unicas consideradas na paralelização para o tipo DNA. Já para sequências de proteínas, foram consideradas as somas de percentuais de tempos de execução das funções *LK_Core* e *Update_P_LK_AA.* É importante destacar que esses valores de *speedup* são referidos como estimados em razão dos valores de percentuais de tempos de execução serem oriundos do GProf, o qual fornece percentual de tempo de

execução exclusivo para cada função, isto é, desconsiderando os tempos do fluxo de execução das subrotinas.

A Figura 92 e Figura 93 a seguir ilustram a estimativa de *speedup* para, respectivamente, pequenos e grandes arquivos de sequências de DNA.

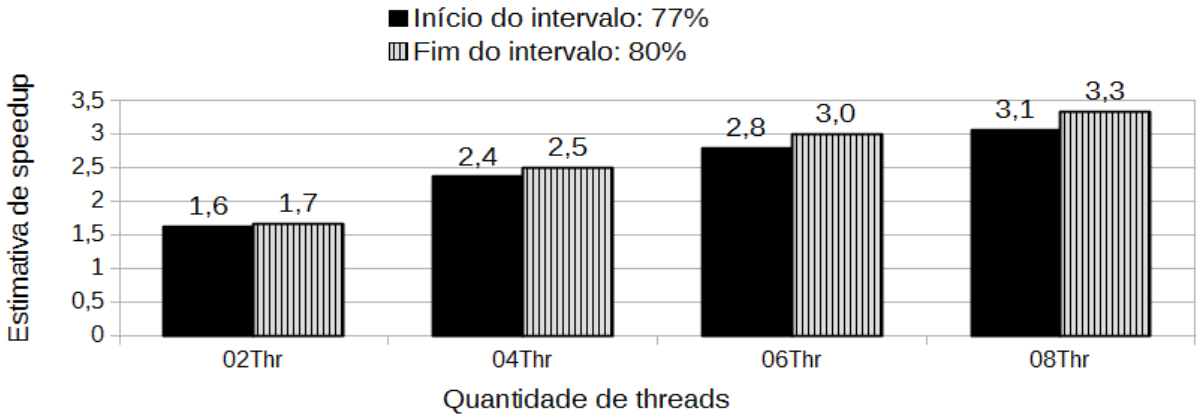

Figura 92 - Estimativa de *speedup* para pequenos arquivos de sequências de DNA, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 77% à 80%.

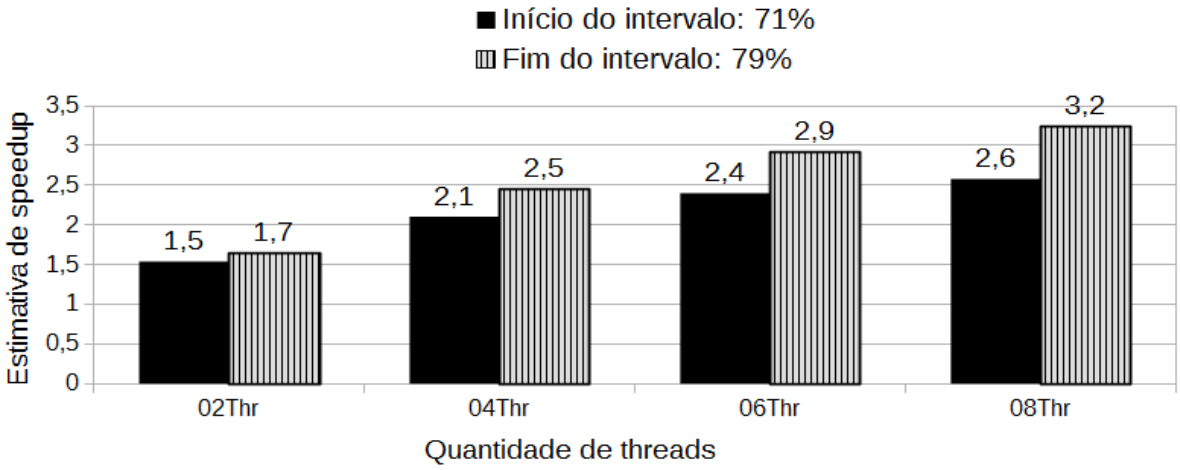

Figura 93 - Estimativa de *speedup* para grandes arquivos de sequências de DNA, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 71% à 79%.

Note que além de maiores valores de soma de percentual de tempo de execução das funções (Figura 92 e Figura 93), existe uma ligeira vantagem da estimativa de desempenho do PhyML com pequenos arquivos em relação aos grandes arquivos de sequências de DNA. Neste sentido, a melhora do desempenho

do PhyML com pequenos arquivos de sequencias de DNA nas simulações (caso I) está relacionada ao percentual de tempo de execução das funções paralelizadas, que na maioria dos casos é maior para pequenos arquivos do que para grandes arquivos de sequências de DNA.

Esta mesma metodologia foi aplicada para arquivos de sequências de proteínas. Neste sentido, a Figura 94 e Figura 95 a seguir ilustram a estimativa de *speedup* para, respectivamente, pequenos e grandes arquivos de sequências de proteínas.

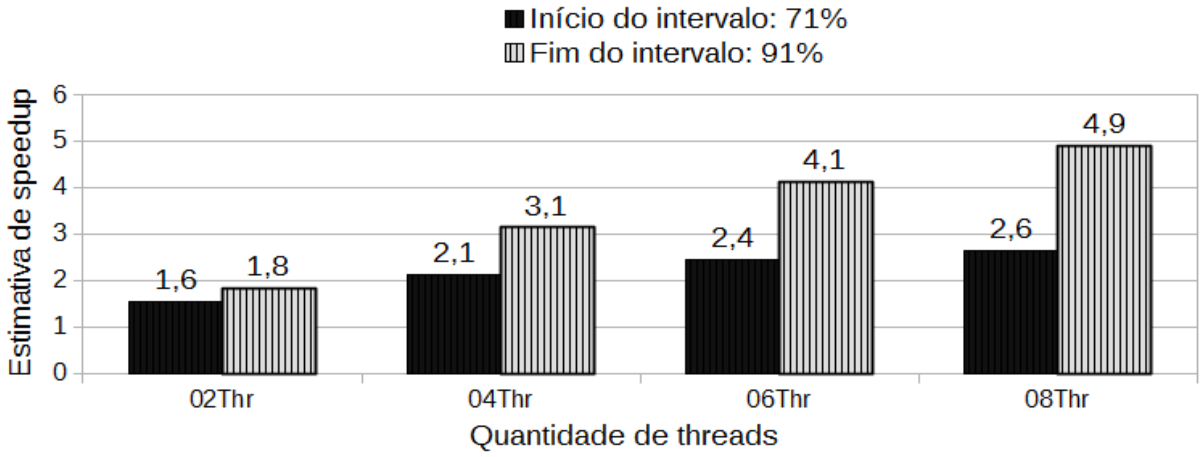

Figura 94 - Estimativa de *speedup* para pequenos arquivos de sequências de proteínas, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 71% à 91%.

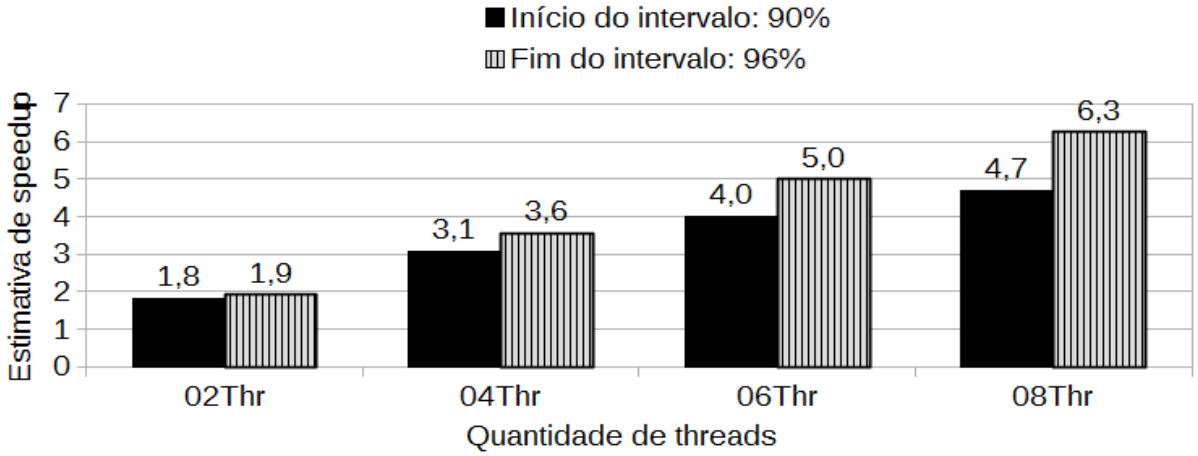

Figura 95 - Estimativa de *speedup* para grandes arquivos de sequências de DNA, para os quais as funções paralelizadas apresentam percentual de tempo de execução no intervalo de 90% à 96%.

Notavelmente, há maiores valores de soma de percentual de tempo de execução das funções (Figura 94 e Figura 95) para grandes arquivos de sequencias em comparação a pequenos arquivos de sequências de proteínas. Da mesa forma, é notável uma melhora da estimativa de desempenho do PhyML com grandes arquivos em relação aos pequenos arquivos de sequências de proteínas. Neste sentido, a justificativa da melhora do desempenho do PhyML com grandes arquivos de sequencias de proteínas nas simulações (caso II) está relacionada ao percentual de tempo de execução das funções paralelizadas, que na maioria dos casos, é significativamente maior para grandes arquivos do que para pequenos arquivos de sequências de DNA.

A comparação de desempenho com pequenos arquivos de sequências revelaram melhor desempenho do PhyML para dados de DNA, no entanto as estimativas indicam o contrário (Figura 92 e Figura 94). Isso pode ter ocorrido pelo fato dos pequenos arquivos de sequências de proteínas possuírem pequena quantidade tanto de sítios quanto de espécies, enquanto os de DNA, apesar de possuírem pequena quantidade de espécies, possui uma quantidade maior de sítios.

Já com relação ao caso IV, em que há melhor desempenho para grandes arquivos de sequências de proteínas em relação a grandes arquivos de sequências de DNA, uma situação curiosa é que para grandes arquivos de sequências de DNA, a função *Find_Mutual_Direction* que em alguns casos chega a ter *speedup* de até 4, esta paralelizada e possivelmente apresentaria um diferencial no *speedup* em relação às simulações com sequencias de proteínas, no entanto, note que a estimativa de ganho é maior para sequências de proteínas, pois os extremos de somas de percentual de tempos de execução é relativamente maior para os dados de proteínas com 90% e 96%, enquanto para DNA é de 71% e 79% (Figura 93 e Figura 95).

Ainda com relação à variação nos valores de desvio padrão, notou-se que, na maioria dos casos, obteve-se um desvio padrão pequeno, variando em torno de 2% a 10%, porém, para alguns arquivos de sequências ocorreram desvios padrão de até 50%, possivelmente causados por instabilidades na arquitetura onde ocorreram as simulações.

5.2 Comparação entre versões paralelizadas

Esta subseção apresenta os resultados de simulações comparativas das existentes versões paralelas do PhyML (OpenMP, MPI e OpenMP+MPI) usando réplicas de *bootstrap* e arquivos de sequências de DNA. Para a quantidade de réplicas, foram adotadas 100 réplicas como valor padrão em todas as simulações. A versão do PhyML com OpenMP foi simulada usando 8 threads, a versão MPI foi simulada usando 8 processos, já a versão híbrida (OpenMP+MPI) foi simulada de duas maneiras uma usando 4 processos e 2 threads por processo e a outra usando 2 processos e 4 threads por processo. É importante destacar que tanto a versão MPI quanto a versão híbrida (nos dois casos: 2 ou 4 processos), os processos se encontravam distribuídos no mesmo nó de calculo do cluster. No entanto, o fato do paradigma MPI ser apropriado para arquiteturas distribuídas e no caso dessas simulações estarem sendo realizadas sobre uma arquitetura de memoria compartilhada (mesmo nó), foi realizado uma simulação com a versão MPI distribuídas sobre 8 processadores de 8 nós e esses resultados comparados com os da simulação com MPI realizada sobre apenas um nó. Os resultados comparativos dessas simulações mostram que a versão MPI não é prejudicada (não há perda de desempenho) em razão do seu uso em uma arquitetura compartilhada, na verdade, existe uma melhora no desempenho da versão MPI quando usando essa arquitetura, conforme resultados do Apêndice (APÊNDICE A - Comparação da versão MPI do PhyML simulada sobre uma arquitetura de memória distribuída e memória compartilha). Além disso, é necessário destacar que ao usar 8 nós (mas apenas um processo por nó) para realizar a simulação de forma distribuída, existem 7 processadores ociosos em cada nó (48 processadores ociosos no total) e que estes processadores também devem ser levados em consideração em análises de eficiência (já que estão impossibilitados de serem usados por outros processos), o que de fato prejudicaria a eficiência da versão MPI. Desta forma, cada versão (MPI, OpenMP e Híbrida) usou apenas 8 processadores (apenas um nó de cálculo), como forma de realizar uma comparação justa entre versões, isto é usando a mesma quantidade de recursos.

Antes de apresentar os resultados das simulações, é necessário abordar o algoritmo MPI do PhyML, para que os resultados das simulações da versão híbrida possa fazer sentido ou sejam mais intuitivos. No PhyML, o MPI é usado para fazer a distribuição de réplicas bootstrap para os processadores envolvidos no processo de paralelização. Neste sentido, o algoritmo é construído da forma como segue.

Entrada: sequências alinhadas
Saída: árvore boot e escore da árvore

```
A. Rank 0: Para i=0 até N° réplicas/processos{
   1 Rank 0: Para k=1 até a quantidade de ranks-1:
      1.1 Rank 0: envia ao Rank k a réplica i[aleatória para
      cada Rank];
   2 Rank K: recebe a réplica i[diferente para cada Rank];
   3 Cada rank: calcula a MV da topologia;
   4 Rank 0: Para k=1 até a quantidade de ranks-1:
      4.1 Rank 0: recebe do Rank k a árvore processada
      i[diferente de cada Rank];
   5 Rank k: envia a árvore processada i;
   6 Cada rank: faz redução do escore de sua árvore;
}
```

No do algoritmo descrito acima, observa-se que cada processo recebe, processa e envia de volta a i-ésima réplica, isto é, há uma distribuição e sincronização para cada nova réplica a ser processada, indicando que se a quantidade de réplicas é muito grande, poderá haver um alto custo computacional relacionado à comunicação. Note que a paralelização com OpenMP está inserida justamente no interior do algoritmo paralelizado com MPI, mais precisamente no “Passo 3”, onde ocorre o calculo de MV, ou seja, a versão híbrida utiliza tanto a paralelização sobre as réplicas bootstrap (MPI) quanto a paralelização sobre os cálculos de MV (OpenMP).

A Figura 96, Figura 97 e Figura 98 ilustradas a seguir, representam respectivamente, o tempo de execução das versões paralelas do PhyML com pequenos arquivos de sequências de DNA, o *speedup* e a eficiência dessas simulações.

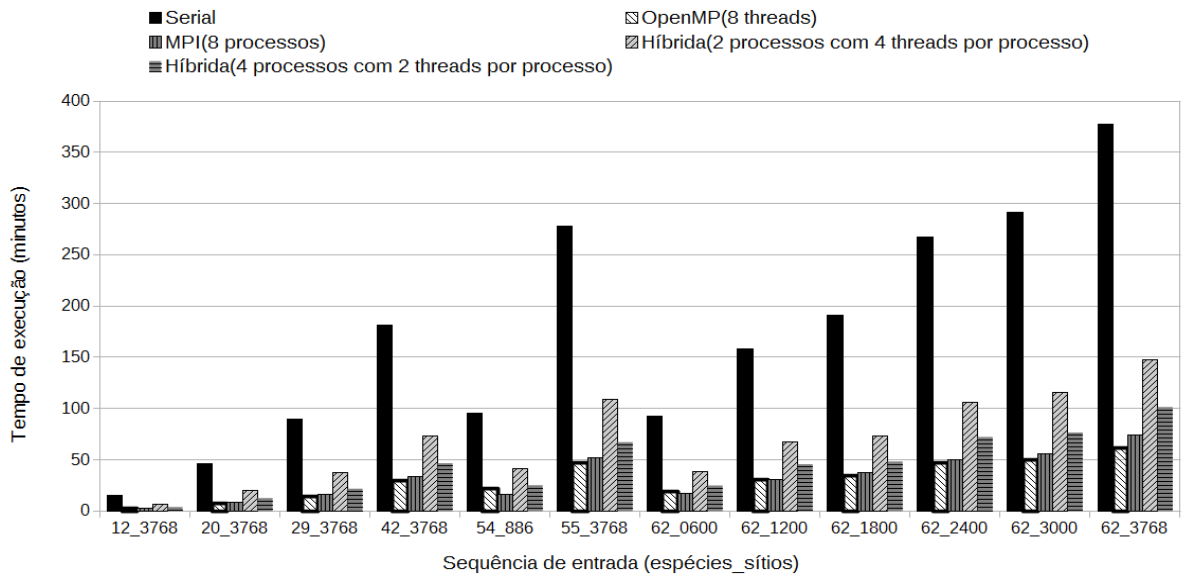

Figura 96 - Análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA.

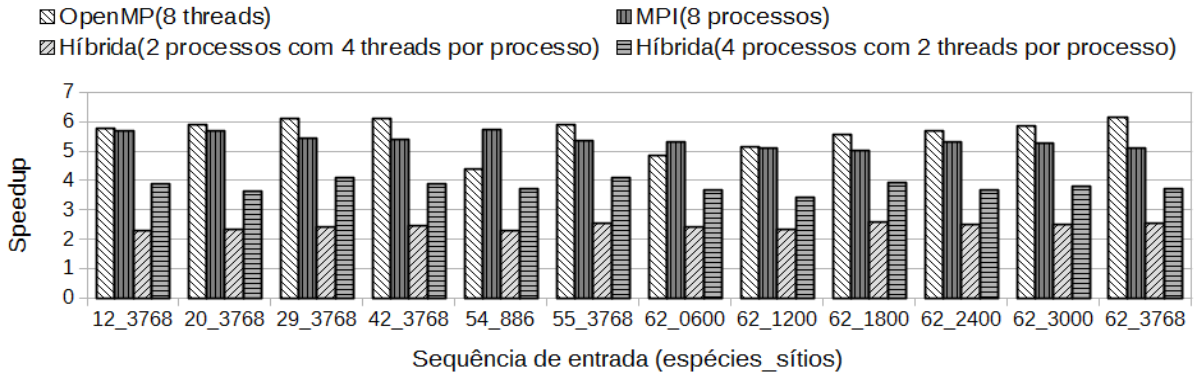

Figura 97 - *Speedup* sobre a análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA. Linearidade: média de *speedup* OpenMP 5.6, MPI 5.4, Híbrida (2 processos) 2.4 e Híbrida (4 processos) 3.8.

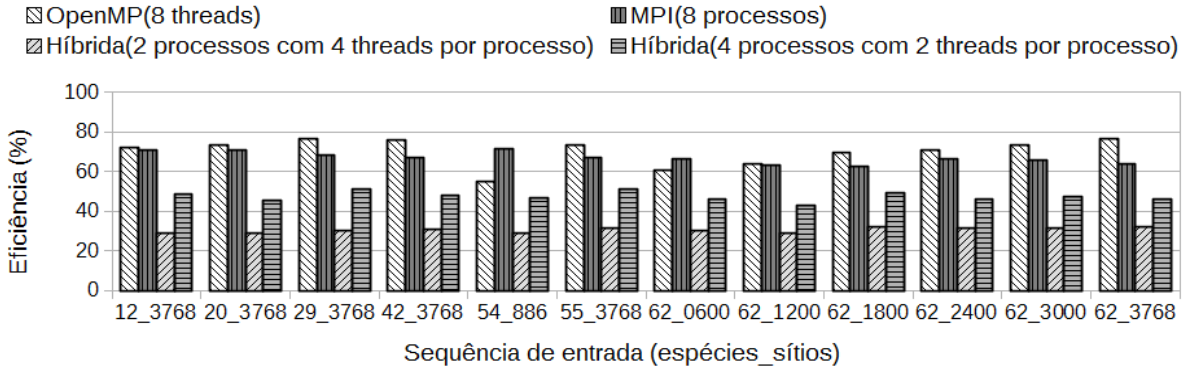

Figura 98 - Eficiência sobre a análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência OpenMP 70.3%, MPI 67.2%, Híbrida (2 processos) 30.6% e Híbrida (4 processos) 47.6%.

O *speedup* das versões paralelas com OpenMP e MPI com pequenos arquivos de sequências de DNA, apresentaram média de *speedup* respectivamente iguais a 5.6 e 5.4, se mostrando praticamente equivalentes em termos de desempenho. Neste sentido, como a versão híbrida engloba essas duas paralelizações, esperava-se que esta apresentasse um comportamento parecido, isto é, com desempenho similar às versões OpenMP e MPI quando usando a mesma quantidade de recursos, no entanto, esta apresentou *speedup* significativamente menor que o dessas versões, com média de *speedup* para a versão híbrida com 2 e 4 processos, respectivamente, 2.4 e 3.8. Já em termos de eficiência, estas simulações mostraram melhores índices para a paralelização com OpenMP, seguido pela paralelização com MPI, para as quais, obteve-se média de eficiência de 70.3% para a versão OpenMP e de 67.2% para a versão MPI. Neste caso, pode-se dizer que a versão OpenMP tem uma eficiência boa, pois apresenta melhor eficiência que a versão já consolidada MPI. Nesse mesmo sentido, a versão híbrida apresentou eficiência muito ruim, com média de eficiência para a versão híbrida com 2 e 4 processos, respectivamente, de 30% e 50%.

A Figura 99, Figura 100 e Figura 101 ilustradas a seguir, representam respectivamente, o tempo de execução das versões paralelas do PhyML, o *speedup* e a eficiência dessas simulações com grandes arquivos de sequências de DNA.

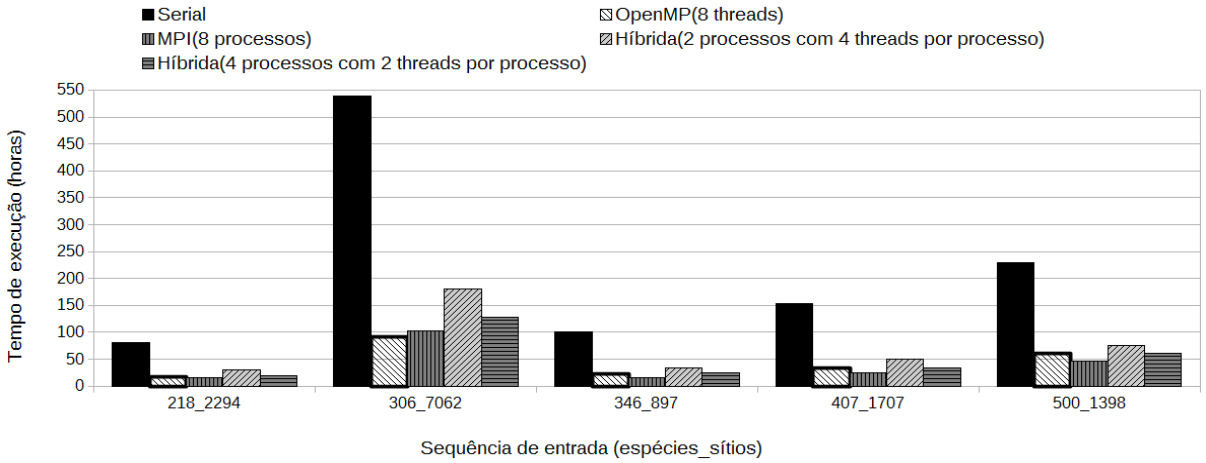

Figura 99 - Análise de desempenho das versões paralelas do PhyML com grandes arquivos de sequências de DNA.

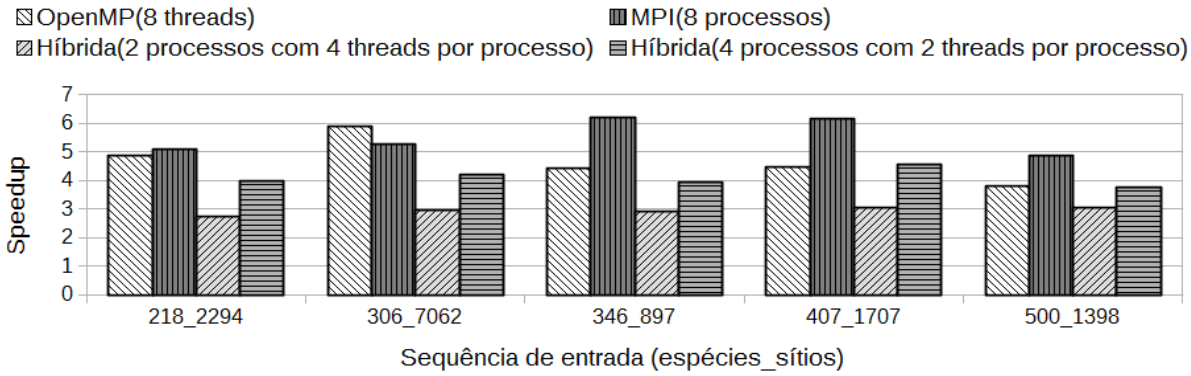

Figura 100 - *Speedup* sobre a análise de desempenho das versões paralelas do PhyML com grandes arquivos de sequências de DNA. Linearidade: média de *speedup* OpenMP 4.7, MPI 5.5, Híbrida (2 processos) 3.0 e Híbrida (4 processos) 4.1.

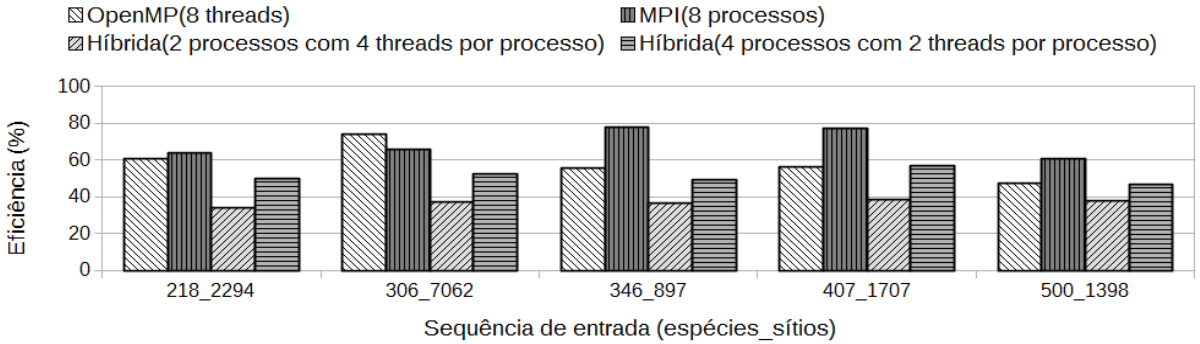

Figura 101 - Eficiência sobre a análise de desempenho das versões paralelas do PhyML com pequenos arquivos de sequências de DNA. Linearidade: média de eficiência OpenMP 58.8%, MPI 69.2%, Híbrida (2 processos) 37.0% e Híbrida (4 processos) 51.1%.

Já com grandes arquivos de sequências de DNA, a versões paralelas com OpenMP e MPI, obtiveram média de *speedup*, respectivamente, 4.7 e 5.5. Note que houve uma ligeira vantagem para a versão paralela com MPI. Já a versão híbrida, da mesma forma ocorrida com pequenos arquivos de sequências, apresentou *speedup* significativamente menor que o dessas versões, com média de *speedup* para a versão híbrida com 2 e 4 processos, respectivamente, 3.0 e 4.1 (Figura 100). Com relação à eficiência, as versões paralelas com OpenMP e MPI obtiveram respectivamente, média de eficiência de 58.8% e 69.2% (assim como a média de *speedup*, com uma ligeira vantagem (10.4%) para a versão MPI). Note que apesar da versão OpenMP possuir média de *speedup* e eficiência menores que a versão MPI, os valores não são discrepantes, isto é, a versão OpenMP pode ser considerada tão eficiente quanto a versão MPI. Já a versão híbrida com 2 processos e 4 processos, respectivamente, apresentou eficiência de 37.0% e 50.1%, uma eficiência ruim se comparada com a da versão MPI. Da mesma forma, a versão híbrida não apresentou um comportamento desejável, com perda de eficiência em relação às outras versões paralelas.

É possível notar ainda uma situação curiosa com relação à versão híbrida; que esta apresenta melhor desempenho com mais processos do que com mais threads, enquanto as versões MPI e OpenMP, estas apresentaram desempenho semelhantes. Neste contexto, esperava-se que a versão híbrida apresentasse desempenho equivalente ou melhor do que as versões distribuída e compartilhada (já que a híbrida é a junção de ambas, e estas se mostraram praticamente equivalentes), no entanto, os resultados mostraram-se contrários às expectativas. No sentido de encontrar e justificar essa ineficiência e comportamento da versão híbrida, foi realizada uma investigação, a qual revelou que que os pontos impactantes nessa ineficiência encontram-se no algoritmo da versão distribuída MPI (descrito no início dessa subseção) aliado à quantidade de réplicas a serem processadas, isto é, a quantidade de comunicações realizadas.

Para entender o impacto que a estratégia de comunicação MPI causa no desempenho da versão híbrida, foram realizadas algumas simulações, nesse caso, usando 4 nós completos (32 processadores) da arquitetura comum do CACAU.

Neste sentido, considere as ilustrações da Figura 102, Figura 103 e Figura 104 a seguir.

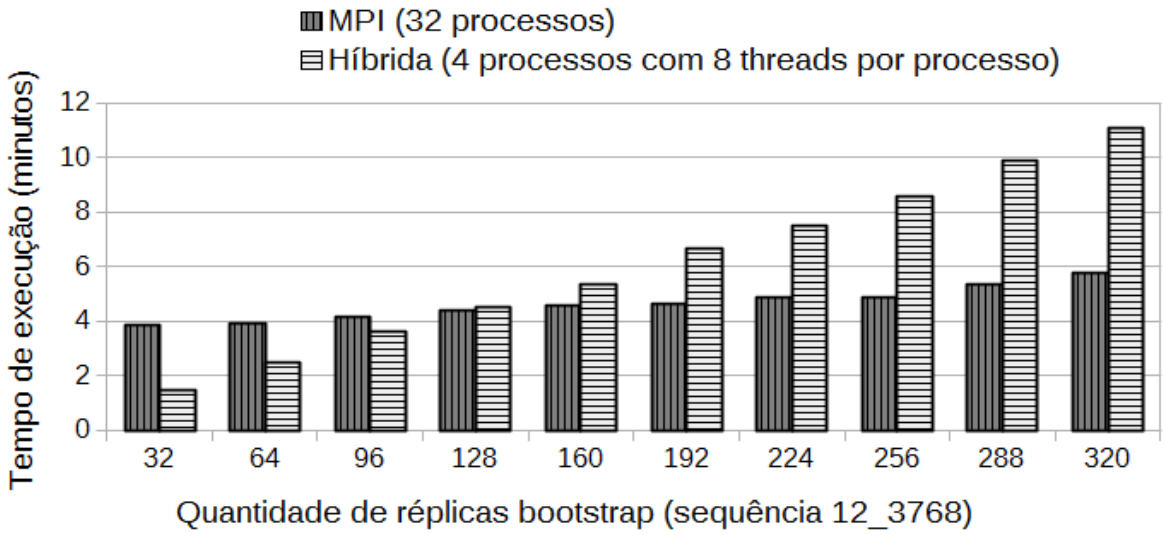

Figura 102 - Comportamento da versão MPI e Híbrida com o aumento do número de réplicas, pequeno arquivo de sequências de DNA (12_3768).

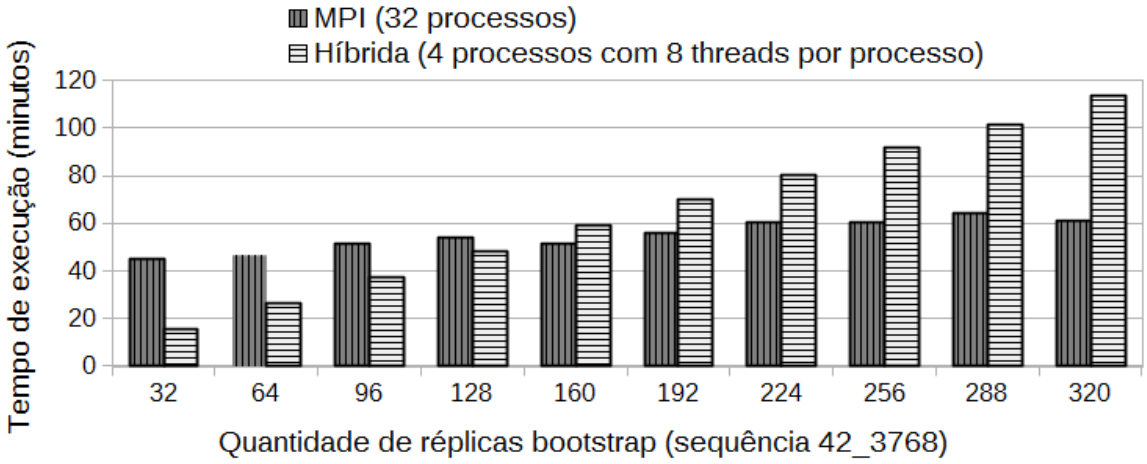

Figura 103 - Comportamento da versão MPI e Híbrida com o aumento do número de réplicas, pequeno arquivo de sequências de DNA (42_3768).

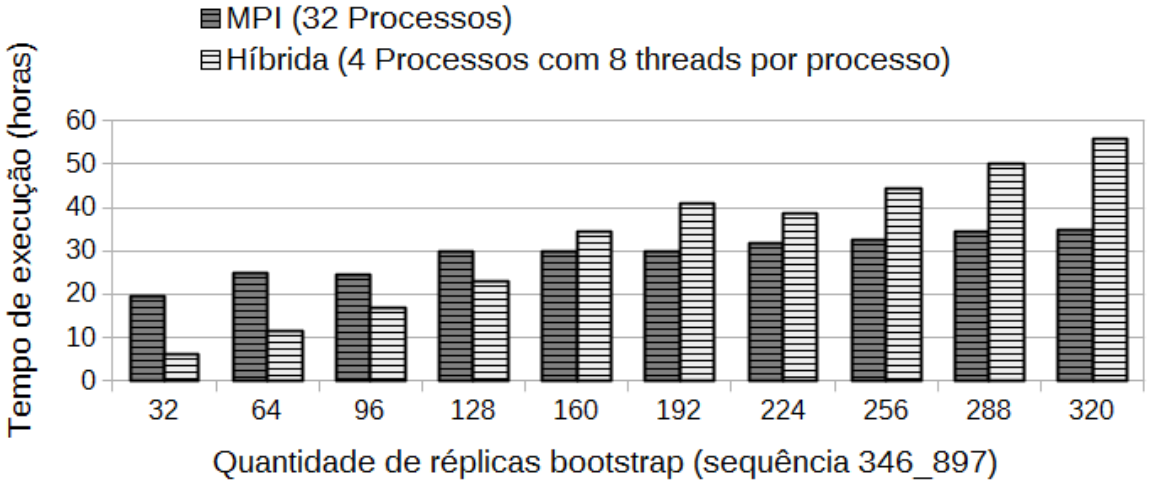

Figura 104 - Comportamento da versão MPI e Hibrida com o aumento do número de réplicas, grande arquivo de sequências de DNA (346_886).

Através das ilustrações da Figura 102, Figura 103 e Figura 104, é possível verificar que quando a quantidade de réplicas é pequena (até 128 réplicas com essa quantidade de processos e threads), a versão híbrida possui melhor desempenho do que a versão MPI, já quando a quantidade de réplicas passa desse valor, o desempenho da versão híbrida vai reduzindo em relação à versão MPI à medida que se aumenta a quantidade de réplicas, tanto com pequenos arquivos (Figura 102, Figura 103) quanto com grandes arquivos de sequências de DNA (Figura 104). Isso ocorre em razão da quantidade de comunicações realizadas no algoritmo MPI. Por exemplo, considerando o cenário das simulações (Figura 102, Figura 103 e Figura 104), isto é, com a versão MPI usando 32 processos e a versão híbrida usando 4 processo e 8 threads por processos, quando considerado apenas 32 réplicas ocorrem 32 comunicações em paralelo para a versão MPI apenas uma vez, já com a versão híbrida, ocorrem 4 comunicações em paralelo 8 vezes. Da mesma forma, quando usando 320 réplicas tem-se 32 comunicações em paralelo ocorrendo por 10 vezes para versão MPI e 4 comunicações em paralelo ocorrendo 80 vezes para a versão híbrida. De fato, nota-se que a quantidade de comunicações aliada à influência da arquitetura (comunicação distribuída e compartilhada) sobre a paralelização é a responsável pela ineficiência da versão híbrida frente à versão MPI.

Isso mostra que a estratégia de comunicação usada na paralelização distribuída MPI não colabora com a versão híbrida, mas que uma reformulação na

estratégia da troca de mensagens, distribuindo todas as réplicas apenas uma vez e realizando a sincronização somente quando todas as réplicas forem processadas, pode haver melhor desempenho e eficiência com a versão híbrida.

5.3 Comparação PhyML e PhyML com a biblioteca beagle (serial)

Esta subseção apresenta um análise comparativa do PhyML usando a biblioteca beagle (abordada no início da subseção 2.1.9 Versões de implementações do PhyML) em relação à versão paralela com OpenMP, tanto para sequências de DNA quanto para sequências de proteínas. A Figura 105 e Figura 106 ilustram respectivamente, os tempos de execução dessas versões com pequenos e grandes arquivos de sequências de DNA.

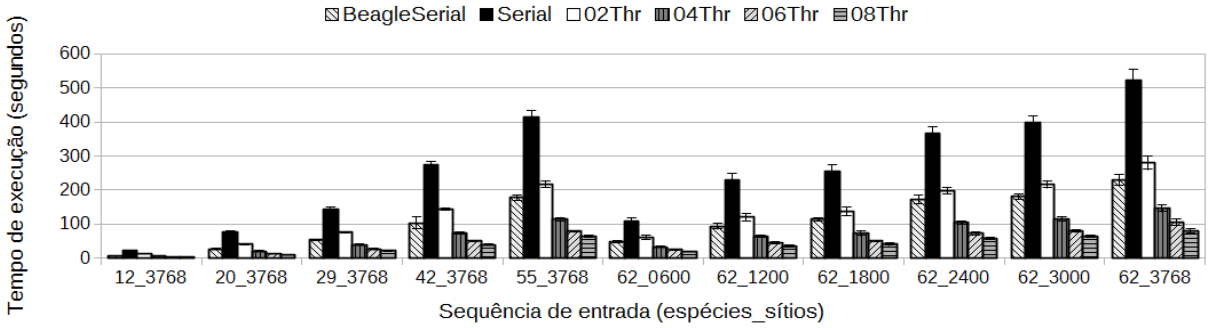

Figura 105 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com pequenos arquivos de sequências de DNA.

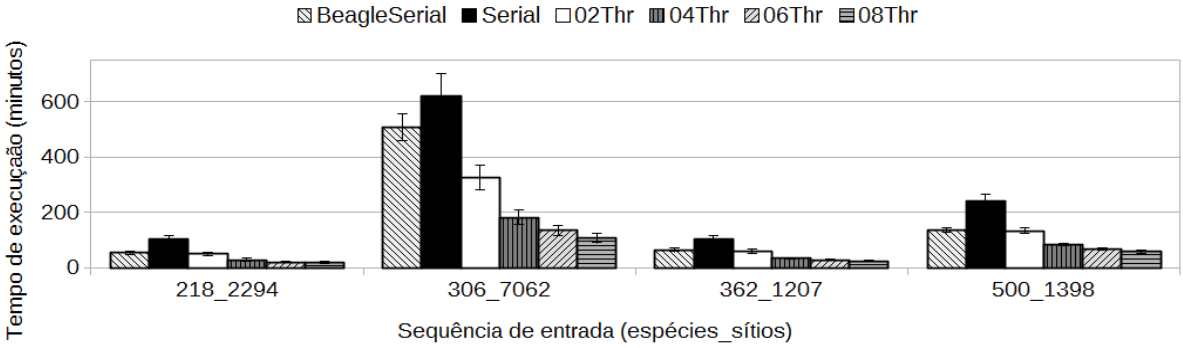

Figura 106 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com grandes arquivos de sequências de DNA. Os maiores desvios padrão ocorreram com o arquivo 306_7062, sendo de 13% com 2 threads e de 12% para a versão serial.

Os resultados mostrados nas duas ilustrações supracitadas (Figura 105 e Figura 106) indicam que, de fato a, a versão serial do PhyML com a biblioteca beagle é consideravelmente rápida em relação à versão serial do PhyML sem o uso da biblioteca, chegando a ser mais rápida do que com 2 threads em alguns casos para pequenos arquivo de DNA, mas que na maioria dos casos, para arquivos grandes, a versão paralela possui tempo de execução menor ou no mínimo igual. Já para pequenos e grandes arquivos de sequências de proteínas ilustrados na Figura 107 e Figura 108, respectivamente, a seguir, a versão paralela mantém o tempo de execução menor que a versão usando a biblioteca.

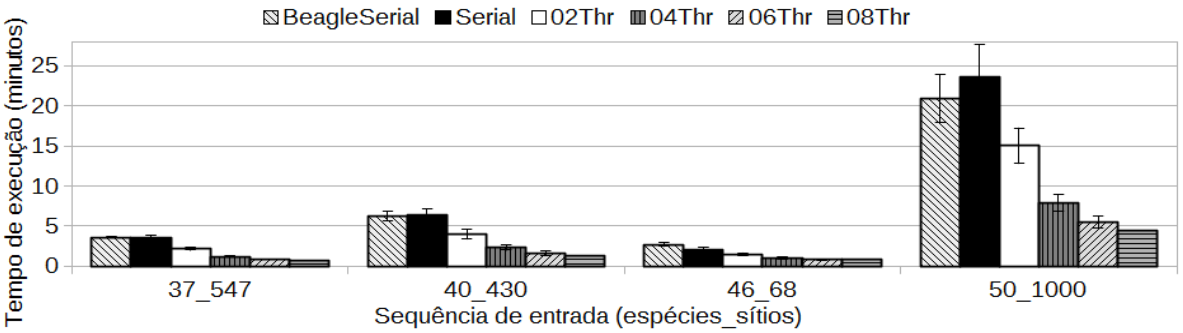

Figura 107 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com pequenos arquivos de sequências de proteínas. Os maiores desvios padrão ocorreram com o arquivo 50_1000, sendo de 17% para a versão serial e 14% com 2 threads.

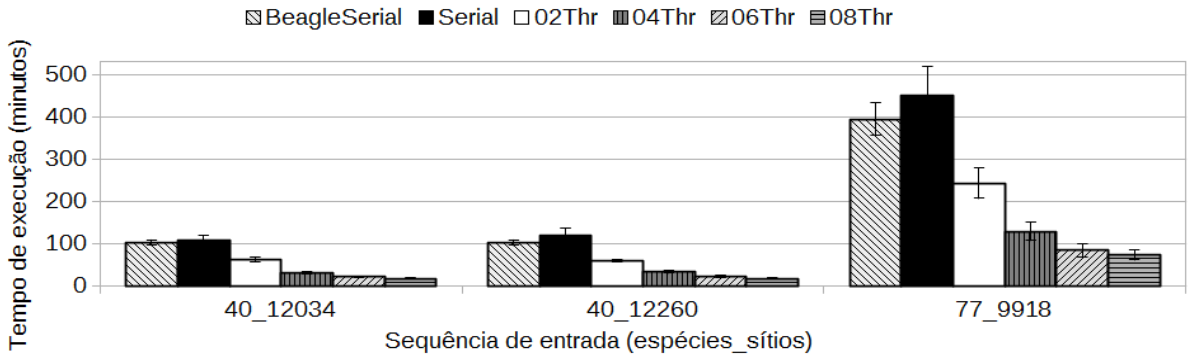

Figura 108 - Comparação da versão do PhyML com a versão beagle e a versão paralela do PhyML com OpenMP com grandes arquivos de sequências de proteínas. Os maiores desvios padrão ocorreram com o arquivo 77_9918, sendo de 15% para a versão serial e 14% com 2 threads.

É importante ressaltar que apesar de em alguns casos a versão do PhyML com a biblioteca possuir menor tempo de execução do que a versão paralela com 2 threads, esta versão não possui suporte ao *bootstrap* e proporção para sítios invariáveis, o que pode descaracterizar o seu uso a depender da necessidade do usuário.

6 CONCLUSÕES E TRABALHOS FUTUROS

Diante do alto custo de processamento envolvido na reconstrução filogenética, sendo este um dos principais gargalos da bioinformática, foi realizado um estudo detalhado do código fonte do programa de RAF PhyML 3.0, bem como um mapeamento das heurísticas e estratégias adotadas por este software, no intuito de descobrir os principais gargalos existentes e aplicar uma paralelização para melhorar o seu desempenho. Neste sentido, foi aplicada a paralelização com o paradigma de memória compartilhada no software PhyML 3.0, o qual utiliza o método de máxima verossimilhança em suas inferências. A análise de desempenho foi simulada sobre as versões paralelas do PhyML com sequências de DNA e proteínas.

A paralelização do PhyML com OpenMP fornece uma versão inédita desse software para o usuário quando este não necessitar do uso de réplicas *bootstrap*. Com esta paralelização, o usuário pode usufruir de uma versão do PhyML que

chega a ser até 9 vezes mais rápida que a versão serial tanto com sequências de DNA quanto sequências de proteínas. Para simulações sem *bootstrap*, a eficiência da versão paralela com OpenMP chegou a alcançar até 90%, representado um excelente uso dos recursos.

Já quando o usuário necessitar do uso de réplicas *bootstrap*, assim como a versão MPI, já consolidada no PhyML, a versão OpenMP também é uma alternativa viável, chegando a ser até 6 vezes mais rápida que a versão serial para sequências de DNA. Em muitos casos a versão OpenMP se mostrou mais rápida que a versão paralela com MPI. A eficiência dessa versão chegou a alcançar até 75%, um valor ótimo e equiparado aop da versão já consolidada MPI.

A versão híbrida do PhyML (MPI+OpenMP) não forneceu desempenhos equiparado ou melhor em relação ao resultantes das versões MPI ou OpenMP, no entanto, revelou um ponto do software (algoritmo de distribuição das réplicas bootstrap) que pode ser investigado para obter uma eficiência ainda melhor no processo de reconstrução.

Já com relação ao uso do PhyML usando a biblioteca beagle, aparentemente este possui um potencial a ser explorado, no entanto, ainda não é estável, pois ainda não possui suporte à proporção para sítio invariáveis e análises *bootstrap* e, nesse caso, a versão paralela se mantém como melhor opção de uso.

É importante ressaltar ainda que, na maioria dos casos, a eficiência das simulações da versão OpenMP se manteve acima de 60%, chagando a mais de 80% em muitos casos, indicando que o processadores ficam ocupados na maior parte do tempo em que estão alocados.

Em se tratando de RAF utilizando computadores de alto desempenho, este projeto de pesquisa possui impacto no âmbito nacional e internacional, visto que o PhyML é software *open source*. Com isto, pode-se afirmar que muito ainda pode ser feito para aperfeiçoar as pesquisas nesta linha de trabalho e melhorar o desempenho e eficiência do PhyML. Sendo assim, para trabalhos futuros, com base nos estudos realizados neste trabalho, propõe-se:

a) otimizar a estratégia de comunicação MPI, reduzindo a quantidade de mensagens, para que a versão híbrida possa colaborar com a eficiência do PhyML;

b) aplicar uma paralelização com CUDA/GPU ou OpenCL sobre os cálculos dos caracteres e categorias, os quais possuem custo computacional baixíssimo para ser realizados com o paradigma de memória compartilhada OpenMP, mas que aparentemente a exploração da paralelização de granularidade fina com GPU ou OpenCL forneceriam melhores resultados. Além disso, as funções *Pull_Scaling_Factors* e *Rate_Correction* se mostraram como gargalos para tempos cumulativos, mas o fato destas possuírem tempo médio de execução muito baixo inviabilizaram a paralelização pelo *overhead* das diretivas OpenMP, neste caso, uma investigação com os paradigmas supracitados podem fornecer melhor desempenho a essas funções e colaborar para um desempenho ainda melhor do PhyML;
c) implementar o suporte para proporção de sítios invariáveis e réplicas *bootstrap* com o uso da biblioteca beagle, a qual aparentemente fornece técnicas potencias de paralelização ainda não exploradas em conjunto com o PhyML;
d) implementar no PhyML através da biblioteca beagle as técnicas de paralelização que esta biblioteca já dá suporte como OpenMP, CUDA/GPU, OpenCL e vetorização AVX (do inglês, *Advanced Vector Extensions*).

APÊNDICE A - Comparação da versão MPI do PhyML simulada sobre uma arquitetura de memória distribuída e memória compartilhada

Comparação da versão MPI do PhyML simulada sobre uma arquitetura de memória distribuída (8 nós do cluster - 8 processadores) e memória compartilha (apenas um nó do cluster - 8 processadores) com 100 réplicas bootstrap, com pequenos e grandes arquivos de sequências de DNA. Ambos os casos mostram que a versão MPI do PhyML possui melhor desempenho quando simulada sobre uma arquitetura de memória compartilhada (mesmo nó).

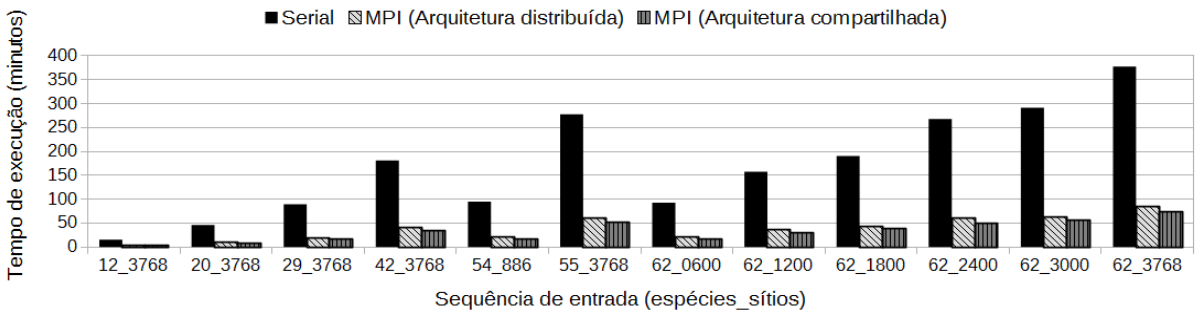

Figura 109 - Tempos de execução da versão MPI do PhyML simulada sobre arquitetura de memória distribuída e memória compartilhada com pequenos arquivos de sequências de DNA.

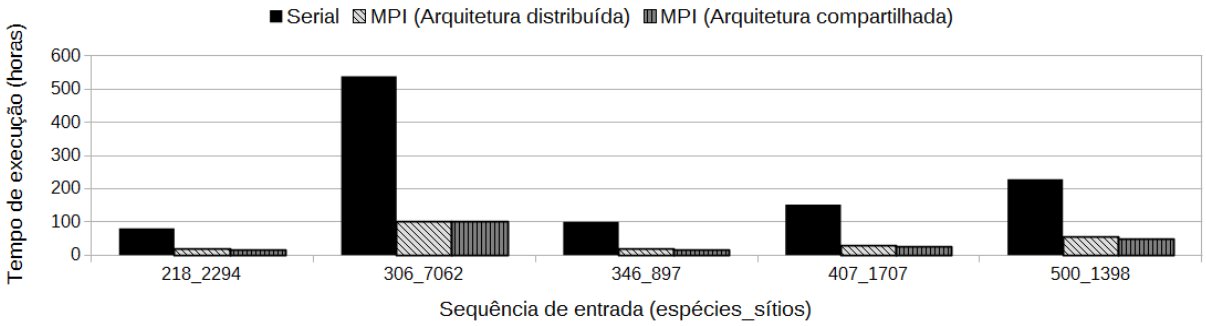

Figura 110 - Tempos de execução da versão MPI do PhyML simulada sobre arquitetura de memória distribuída e memória compartilhada com grandes arquivos de sequências de DNA.

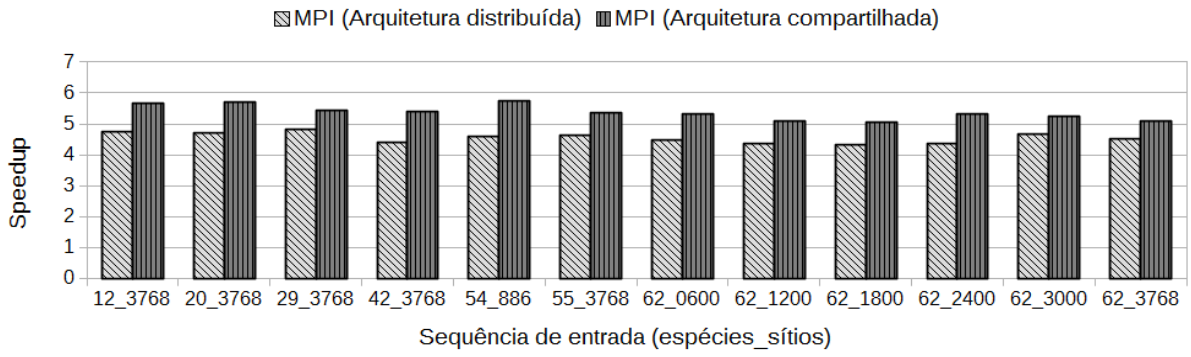

Figura 111 - Comparação de *speedup* da versão MPI do PhyML simulada sobre uma arquitetura de memória distribuída e memória compartilhada com pequenos arquivos de sequências de DNA.

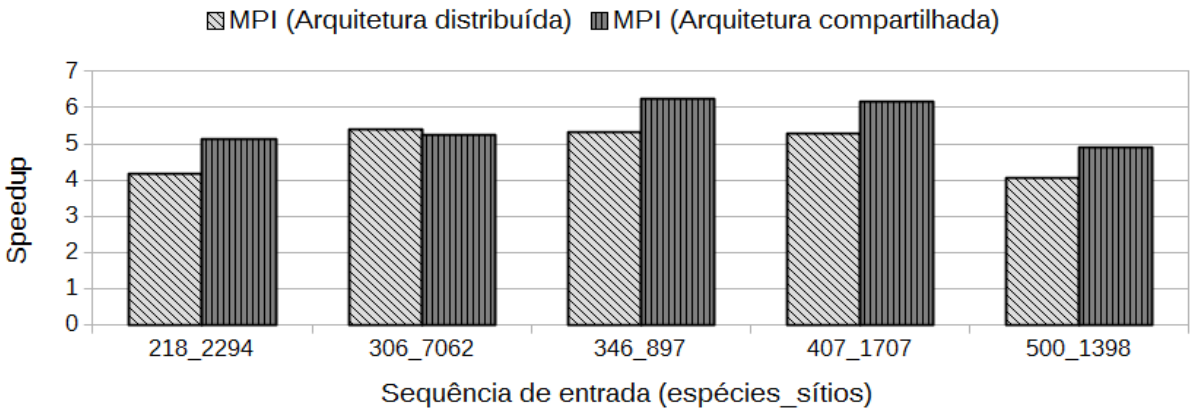

Figura 112 - Comparação de *speedup* da versão MPI do PhyML simulada sobre arquitetura de memória distribuída e memória compartilhada com grandes arquivos de sequências de DNA.

REFERÊNCIAS

ALMEIDA, Paulo R. X.; HONDA, Marcelo O.; LIMA, Daniel A. C. Desenvolvimento de algoritmos paralelos para processamento de imagens médicas. **XIII Congresso Brasileiro de Informática em Saúde-CBIS**, Curitiba, 2012.

AMORIM, Dalton S. **Fundamentos de sistemática filogenética**. 1. ed. Ribeirão Preto São Paulo: Holos, 2002. 156 p.

ANISIMOVA, Maria; GASCUEL, Olivier. Approximate Likelihood-Ratio Test for Branches: A Fast, Accurate, and Powerful Alternative. **Oxford University Press: Systematic Biology,** v. 55, n. 4, p.539-552, Aug 2006.

ATGC. ATGC: South of France bioinformatics platform - PhyML 3.0: new algorithms, methods and utilities. France. 2010. Disponível em: <http://www.atgc-montpellier.fr/phyml/>. Acesso em: 27 nov. 2015a.

ATGC. ATGC: South of France bioinformatics platform - PhyML 3.0 Benchmarks. France. 2010. Disponível em: <http://www.atgc-montpellier.fr/phyml/benchmarks/>. Acesso em: 27 nov. 2015b.

AYRES, Daniel L. et al. BEAGLE: An Application Programming Interface and High-Performance Computing Library for Statistical Phylogenetics. **Oxford University Press: Systematic Biology,** v. 61, n. 1, p.170-173, Jan 2012.

BAREKOVIC, Mladen et al. **Architeture of Computing Systems - ARCS**. Italy: Springer, 2011. 271 p.

BAZINET, Adam L.; ZWICKL, Derrick J.; CUMMINGS, Michael P. A Gateway for Phylogenetic Analysis Powered by Grid Computing Featuring GARLI 2.0. **Oxford University Press: Systematic Biology**, v. 63, n. 5, p. 812-818, Apr 2014.

BORDOLI, Lorenza. Similarity Searches on Sequence Databases: BLAST, FASTA. Swiss Institute of Bioinformatics EMBnet. EMBnet Course. German: Basel. 2003.

BRENT, Richard. P. **Algorithms for minimization without derivatives**. Englewood Cliffs, New Jersey: Prentice-Hall. 1973. 195 p.

BRUNO, William. J., SOCCI, Nicholas. D., HALPERN, Aaron. L. Weighted neighbor joining: a likelihood- based approach to distance-based phylogeny reconstruction. **Molecular Biology and Evolution**, v. 17, n. 1, p. 189-197, Jan 2000.

BULL, J. Mark; FIONA, Reid. A microbenchmark suite for OpenMP tasks . **Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP'12),** Heidelberg, v.7312, p. 271-274, 2012.

BULL, J. Mark. Measuring Synchronisation and Scheduling Overheads in OpenMP. **Proceedings of the First European Workshop on OpenMP**, p. 99-105, 1999.

BULL, J. Mark; O´NEILL, Darragh. A Microbenchmark Suite for OpenMP 2.0. **SIGARCH Computer Architecture News**, New York, v. 29, n. 5, p. 41-48, 2001.

CALAZAN, Rogério M. **Otimização por Enxame de Partículas em Arquiteturas Paralelas de Alto Desempenho**. 2013. 138 f. Dissertação (Mestrado em Engenharia Eletrônica) - Centro de Tecnologia e Ciências, Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro. Rio de Janeiro, 2013.

CHAPMAN, Barbara; JOST, Gabriele; PAS, Ruud V. D. **Using OpenMP:** Portable Shared Memory Parallel Programming. Cambridge, Massachusetts: MIT Press, 2007. 348 p.

CHOR, Benny; TULLER, Tamir. Maximum likelihood of evolutionary trees is hard. In: MIYANO, Satoru et al. **Research in Computational Molecular Biology**: Proceedings of 9th Annual International Conference. 1. ed. Cambridge, Massachusetts: Springer-Verlag, 2005, v. 3500, p. 296-310.

CRISCUOLO, Alexis. morePhyML: improving the phylogenetic tree space exploration with PhyML 3. **Molecular Phylogenetics and Evolution**, v. 61, n. 3, p. 944-948, Dec 2011.

CYBIS, Gabriela B. **Teste da Razão de Verossimilhança e seu poder em Árvores Filogenéticas**. 2009. 255 f. Dissertação (Mestrado em Matemática) - Instituto de Matemática, Universidade Federal do Rio Grande do Sul. Porto Alegre, 2009.

DAGUM, Leonardo; MENON, Ramesh. OpenMP: An Industry-Standard API for Shared-Memory Programming. **Computational Science & Engineering, IEEE**, v. 5, n. 5, p. 46-55, Jun/Mar 1998.

DESPER, Richard, GASCUEL, Olivier. Fast and accurate phylogeny reconstruction algorithms based on the minimum-evolution principle. **Journal of Computational Biology**, v. 9, n. 5, p. 687-705, 2002.

DNABIGDATA. DNA benchmark. Disponível em: <http://www.atgc-montpellier.fr/phyml/benchmarks/data/big_nucleic/>. Acesso em: 27 nov. 2015.

EWENS, Warren J.; GRANT, Gregory R. **Statistical Methods in Bioinformatics:** An Introduction. 2. ed. New York: Springer-Verlag, 2001. 476 p.

FELSENSTEIN, Joseph. Confidence limits on phylogenies: an approach using the bootstrap. **Evolution**, v. 39, n. 4, p. 783-791, Jul 1985.

FELSENSTEIN, Joseph. **Inferring Phylogenies**. Sunderland, Massachusetts: Sinauer Associates, 2004. 664 p.

FELSENSTEIN, Joseph. PHYLIP (phylogeny inference package) version 3.6a2. Distributed by the author, Department of Genetics, University of Washington, Seattle, 1993.

FENLASON, Jay; STALLMAN, Richard. GNU gprof: The gnu Profile. Free Software Foundation, 2002.

FERREIRA, Elias B. **Processamento Paralelo Aplicado a Métodos Filogenéticos**. 2012. 100 f. Dissertação (Mestrado em Computação) - Instituto de Informática, Universidade Federal de Goiás. Montes Claros, Goiás, 2012.

FORUM-MPI. MPI: A Message-Passing Interface Standard - version 3.0. University of Tennessee, Knoxville, Tennessee, Sep 2012.

GASCUEL, Olivier. BIONJ: An Improved Version of the NJ Algorithm Based on a Simple Model of Sequence Data. **Molecular Biology and Evolution**, v. 14, n. 7, p. 685-695, Jul 1997.

GOLOBOFF, Pablo A.; FARRIS, James. S.; NIXON, Kevin. TNT: Tree analysis using New Technology. **Systematic Biology**, v. 54, n. 1, p. 176-178, Feb 2005.

GONÇALVES, Glauber D. **Estudo de técnicas para melhorar o desempenho da reconstrução de árvores filogenéticas por análise bayesiana**. 2008. 73 f. Dissertação (Mestrado em Modelagem Matemática) - Universidade Estadual de Santa Cruz, Ilhéus-Bahia. 2008.

GPROF. GNU GProf. Disponível em: <https://sourceware.org/binutils/docs/gprof/>. Acesso em: 27 nov. 2015.

GRIMMENT, Geoffrey. R.; STIRZAKER, David. R. **Probability and Random Processes**. 2. ed. New York: Oxford University Press, 1992. p. 541.

GROPP, William; LUSK, Ewing. **A Test Implementation of the MPI Draft Message-Passing Standard**. 1992. 66 f. Master (office of Scientific computing) - Mathematics and computer Science Division, Washington, US, 1992.

GUINDON, Stéphane et al. PHYML Online - a web server for fast maximum likelihood-based phylogenetic inference. **Nucleic Acids Research**, v. 33, p. W557-W559, Jul 2005.

GUINDON, Stéphane et al. New Algorithms and Methods to Estimate Maximum-Likelihood Phylogenies : Assessing the Performance of PhyML 3.0. **Systematic Biology**, v. 59 n. 3, p. 307-321, Mar 2010.

GUINDON, Stéphane; GASCUEL, Olivier. A Simple , Fast , and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood. **Systematic Biology**, v. 52 n. 5, p. 696-704, Oct 2003.

HORDIJK, Wim; GASCUEL, Olivier. Improving the efficiency of SPR moves in phylogenetic tree search methods based on maximum likelihood. **Bioinformatics**, v. 21 n. 24, p. 4338-4347, Oct 2005.

HUELSENBECK, John P.; RONQUIST, Fredrik. MrBayes: Bayesian inference of phylogenetic trees. **Bioinformatics**, v. 17 n. 8, p. 754-755, Aug 2001.

HYPÓLITO, Elizabeth B. **Uma resposta bayesiana ao Paradoxo de Suzuki**. 2005. 71 f. Dissertação (Mestrado em Estatística) - Instituto de Matemática, Universidade Federal do Rio de Janeiro. Rio de Janeiro, 2005.

JEFFERS, Jim; REINDERS, James. **High Performance Parallelism Pearls:** Multicore and Many-core Programming Approaches Volume Two. 1. ed. Burlington, Massachusetts: Morgan Kaufmann, 2015. 592 p.

JUKES, Thomas H.; CANTOR, Charles R. Evolution of protein molecules. **Academic Press**, New York, p. 21-132, 1969.

JUNIOR, Francisco H. C. **Programação paralela eficiente e de alto nível sobre arquiteturas distribuídas**. 2003. 257 f. Tese (Doutorado em Ciência da Computação) - Centro de Informática, Universidade Federal de Pernambuco. Pernambuco, 2003.

KAMAL, Humaira; WAGNER, Alan. FG-MPI: Fine-grain MPI for Multicore and Clusters. **IEEE**. 2010.

KAZUKI, Valero et al. **Performance Computing:** third international symposium. Tokyo, Japan: Springer, 2000. 593 p.

KUHNER, Mary K.; FELSENSTEIN, Joseph. A Simulation Comparison of Phylogeny Algorithms under Equal and Unequal Evolutionary Rates. **Molecular Biology and Evolution**, v. 11, n. 3, p. 459-468, May 1994.

LARGET, Bret; SIMON, Donald L. Markov chain Monte Carlo algorithms for the Bayesian analysis of phylogenetic trees. **Molecular Biology and Evolution**, v. 16, n. 6, p. 750-759, 1999.

LARKIN, Mark. A. et al. Clustal W and Clustal X version 2.0. **Bioinformatics**, v. 23, n. 21, p. 2947-2948, Sep 2007.

MARZULO, Leandro A. J. **Explorando linhas de execução paralelas com programação orientada por fluxo de dados**. 2011. 140 f. Tese (Doutorado em Engenharia de sistemas e Computação) - Universidade Federal do Rio de Janeiro. Rio de Janeiro. 2011.

MCCLEAN, Phyl. BLAST: Basic Local Alignment Search Tool. Sep 2004.

MENDES, Rodrigo. Reconstrução Filogenética. Disponível em: <http://rodrigomendes.tripod.com/sitebuildercontent/sitebuilderfiles/RMseminario.pdf >. Acesso em: 08 Dez 2015.

MORRISON, David A. Increasing the efficiency of searches for the maximum likelihood tree in a phylogenetic analysis of up to 150 nucleotide sequences. **Systematic Biology**, v. 56, n. 6, p. 988-1010, Oct 2007.

MULLER, Matthias S. et al. **OpenMP Shared Memory Parallel Programming**. Germany, Berlin Heidelberg: Springer-Verlag, 2007. 447 p.

OLSEN, Gary J. et al. fastDNAml: a tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. **Computer applications in the biosciences**, v. 10, n. 1, p. 41-48, Dec 1994.

OPENMP. The OpenMP API specification for parallel programming. Disponível em: <http://openmp.org/wp/openmp-specifications/>. Acesso em: 23 dez. 2015.

PRESS, William H. et al. **Numerical Recipes in C:** The Art of Scientific Computing. 3. ed. New York, Cambridge: Cambridge University Press, 1992. 925 p.

PRICE, Morgan N.; DEHAL, Paramvir S.; ARKIN, Adam P. FastTree 2-approximately maximum-likelihood trees for large alignments. **PLoS One**, v. 5, n. 3, p. 9490, Mar 2010.

PROTEINDATA. Protein benchmark. Disponível em: <http://www.atgc-montpellier.fr/phyml/benchmarks/data/proteic/>. Acesso em: 27 nov. 2015.

RANWEZ, Vincent; GASCUEL, Olivier. Quartet-based phylogenetic inference: improvements and limits. **Molecular Biology and Evolution**, v. 18, n. 6, p. 1103-11016, Jun 2001.

RIBEIRO, Neumar S. **Explorando programação híbrida no contexto de clusters de máquinas numa**. 2011. 81 f. Dissertação (Mestrado em m Ciência da Computação) - Faculdade de Informática, PUCRS. Portpo Alegre. 2011.

SAITOU, Naruya; NEI, Masatoshi. The Neighbor-Joining Method: A New Method for Reconstructing Plylogenetic Trees. **Molecular Biology and Evolution**, v. 4, n. 4, p. 406-425, Jul 1987.

SEWARD, Julian; NETHERCOTE, Nicholas; WEIDENDORFER, Josef. **Valgrind 33 - Advanced Debugging and Profiling for GNU/Linux applications**. 2. ed. Network theory Ltd, Mar 2008. 573 p.

SNEATH, Peter. H. A.; SOKAL, Robert R. **Numerical taxonomy: the principles and practice of numerical classification**. 2 ed. San Francisco: W. H. Freeman, 1973. 573 p.

SOUZA NETO, Manoel A. **Desenvolvimento de servidor web de alto desempenho para soluções de reconstrução de árvores filogenéticas:** IgrafuWeb. 2015. 133 f. Dissertação (Mestrado em Modelagem Computacional em Ciência e Tecnologia) - Universidade Estadual de Santa Cruz. Ilhéus, Bahia. 2015.

STAMATAKIS, Alexandros. RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies. **Bioinformatics**, v. 30, n. 9, Jan 2014.

STAMATAKIS, Alexandros. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. **Bioinformatics**, v. 22, n. 21, p. 2688-2690, Aug 2006.

SWOFFORD, David L. PAUP*: Phylogenetic Analysis Using Parsimony. Sunderland, Massachusetts: Laboratory of Molecular Systematics Smithsonian Institution. 2002. 140 p.

SWOFFORD, David L.; SULLIVAN, Jack. Phylogeny inference based on parsimony and other methods using paup*. In: LEMEY, Phylippe; SALEMI, Marco; VADAMME, Anne-Mieke. (Ed.). **The Phylogenetic Handbook:** A practical Approach to DNA and Protein Phylogeny. 2. ed. Cambridge: Cambridge University Press, cap. 7, p. 160-206, Apr 2009.

TICONA, Waldo G. C. **Algoritmos evolutivos multi-objetivo para a reconstrução de árvores filogenéticas**. 2008. 134 f. Tese (Doutorado em Ciências na área de Ciência da Computação e Matemática Computacional) -
Instituto de ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos. 2008.

TREEBASE. TreeBASE: A Database of Phylogenetic Knowledge. Disponível em: <http://treebase.org/treebase-web/home.html>. Acesso em: 27 nov. 2015.

VISUALVM. VisualVM 1.3.8: All-in-One Java Troubleshooting Tool. Disponível em: <https://visualvm.java.net/>. Acesso em: 27 nov. 2015.

VTUNE. Intel VTune Amplifier 2016: Performance Profiler. Disponível em: <https://software.intel.com/en-us/intel-vtune-amplifier-xe>. Acesso em: 27 nov. 2015.

YANG, Ziheng. Among-site rate variation and its impact on phylogenetic analyses. **TREE**, v. 11, n. 9, p. 367-372, 1996.

YANG, Ziheng. **Computational Molecular Evolution**. 1. ed. Oxford University Press, Dez 2006. 376 p.

YANG, Ziheng. Maximum Likelihood Estimation on Large Phylogenies and Analysis of Adaptive Evolution in Human Influenza Virus A. **Journal of Molecular Evolution**, v. 51, n. 5, p. 423-432, Nov 2000.

YANG, Ziheng. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: Approximate methods. **Journal of Molecular evolution**, v. 39, n. 3, p. 306–314, 1994.

ZWICKL, Derrick J. **Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion**. 2006. 115 f. Thesis (Doctor of Philosophy) - Faculty of the Graduate School, University of Texas at Austin, Austin, Texas, 2006.