http://journal.unoeste.br/index.php/ce/index
DOI: 10.5747/ce.2020.v12.n3.e328
ISSN on-line 2178-8332

Colloquium Exactarum

Submetido: 14/07/2020 Revisado:20/09/2020 Aceito: 25/09/2020

APRIMORAMENTOS DA SOLUÇÃO PARALELA BASEADA EM OPERAÇÕES COLETIVAS PARA O *BOOTSTRAP* DA RECONSTRUÇÃO DE ÁRVORES FILOGENÉTICAS NO *PHYML 3.0*

IMPROVEMENTS TO THE PARALLEL SOLUTION BASED ON COLLECTIVE OPERATIONS FOR THE PHYLOGENETIC TREE RECONSTRUCTION BOOTSTRAP IN PHYML 3.0

Martha Ximena Torres Delgado

Universidade Estadual de Santa Cruz – UESC, BA.
E-mail: mxtd2000@yahoo.com.br

RESUMO – A filogenética é um ramo de estudos que objetiva determinar as relações evolutivas entre grupos de espécies. Como produto, é obtida uma hipótese que relaciona a coleção de espécies analisadas, que normalmente é representada através de uma árvore filogenética. *PhyML* é um dos principais programas que realizam a Reconstrução de Árvores Filogenéticas. O *bootstrap* é um método estatístico utilizado para medir a confiança de um determinado conjunto de dados, que é usualmente aplicado na análise de árvores filogenéticas inferidas. No *PhyML*, esse método possui duas implementações paralelas MPI: com operações ponto-a-ponto e operações coletivas. A segunda versão é mais eficiente que a primeira, porém apresenta uma limitação no número de *bootstrap* a ser utilizado devido ao aumento no consumo de memória. Para solucionar esse problema foram desenvolvidas três soluções. O objetivo deste trabalho foi realizar a validação destas versões juntamente com testes de desempenho. A validação mostrou que as soluções propostas apresentam resultados equivalentes à versão ponto-a-ponto. Já nas simulações de desempenho, duas soluções se mostraram superiores à versão ponto-a-ponto, sendo que a melhor conseguiu ganhos de 28,46% e 39,64% para 32 e 64 processos, respectivamente. Portanto, os aprimoramentos permitem alternativas à versão ponto-a-ponto sem limitação de memória.
Palavras-chave: reconstrução de árvores filogenéticas; processamento paralelo; MPI.

ABSTRACT – Phylogenetics determines the evolutionary relationships between groups of species, through a phylogenetic tree. PhyML is among the main programs for the reconstruction of phylogenetic trees. Bootstrap is a statistical method used to measure the confidence of a

given data set, which is usually applied in the analysis of inferred phylogenetic trees. In PhyML this method has two MPI parallel implementations: with point-to-point operations and collective operations. The second version is more efficient than the first, however it has a limitation on the number of bootstrap to be used due to the increase in memory consumption. In order to solve this problem, three proposals were developed. The objectives of this work were to carry out the validation of these versions together with performance tests. The validation showed that the proposed solutions present results equivalent to the point-to-point version. In the performance simulations, two solutions were shown to be superior to the point-to-point version, with the best one achieving gains of 28.46% and 39.64% for 32 and 64 processes, respectively. Therefore, the enhancements allow alternatives to the point-to-point version without limiting memory.
Keywords: phylogenetic tree reconstruction; parallel processing; MPI.

1. INTRODUÇÃO

A filogenética é uma área de pesquisa da biologia evolutiva e, tem como objetivo determinar as relações genéticas entre um conjunto de espécies. Os estudos filogenéticos geralmente são desenvolvidos por meio de sequenciamento genético molecular das espécies que estão sendo estudadas e por fim, aplicação de um modelo evolutivo em cima desse sequenciamento, bem como um método de reconstrução de árvores filogenéticas. Como resultado tem-se uma hipótese com relação ao conjunto de espécies estudadas, que geralmente é exposto por meio de uma árvore filogenética (PASQUALIN, 2016).

A escolha da árvore filogenética ótima é um problema NP-completo, por isso, existem diversos métodos utilizados para Reconstrução de Árvores Filogenéticas (RAF), sendo que cada um possui suas técnicas e particularidades, como heurísticas próprias e algoritmos estatísticos. Segundo Felsenstein (2004), os principais métodos usados para RAF são: Distância, Inferência Bayesiana, Máxima Parcimônia e Máxima Verossimilhança.

A árvore filogenética reconstruída por meio de um método de RAF qualquer, pode ser vista como uma das possibilidades existentes no espaço de árvores e, portanto, é desejável utilizar alguma avaliação de confiabilidade na inferência obtida, o método de *bootstrap* é uma técnica tradicional para este fim.

Segundo Ticona (2008), este método consiste em gerar alterações aleatórias no conjunto de dados original, a fim de gerar um determinado número de conjuntos de dados. Após a geração das réplicas das sequências, cada uma destas é usada como entrada para o método de RAF em questão, que então infere uma árvore para cada réplica e, de acordo com Verli (2014), as árvores geradas são então comparadas e, para cada nó da árvore, é associado um valor de porcentagem indicando a frequência em que as duas sequências ligadas pelo nó aparecem juntas em diferentes árvores, de maneira que, quanto maior o valor da porcentagem, maior é a probabilidade de que as sequências foram posicionadas corretamente na filogenia. Essa porcentagem é conhecida como grau de suporte ou proporção de *bootstrap*.

Dentre os principais programas *open source* de RAF, está o *PhyML* (GUINDON; GASCUEL, 2003). O *PhyML* é um software que desempenha o processo de reconstrução

utilizando o método de máxima verossimilhança, que é um método que calcula a probabilidade de os dados de uma árvore específica representarem a realidade de acordo com um modelo evolutivo estabelecido. Este programa tem sido amplamente utilizado devido sua simplicidade, além de apresentar eficiência e exatidão. Ainda na versão 2.45, o *PhyML* já possuía uma versão paralela que foi desenvolvida utilizando MPI (*Message Passing Interface*), sendo que esta paralelização foi criada sobre a opção de *bootstrap*.

Até então, existem duas implementações MPI de alto desempenho que envolvem a paralelização do *bootstrap*. A primeira implementação se baseia no uso de operações de comunicação ponto-a-ponto do MPI. Já a segunda versão se trata da utilização de comunicação coletiva, que teve como objetivo reduzir a comunicação entre processos e aumentar a granularidade do algoritmo. Com base na análise de desempenho desenvolvida, esta versão apresenta menor tempo de execução (TORRES; DA SILVA, 2018).

Contudo, essa segunda versão possui uma limitação no número de *bootstrap* a ser utilizado devido ao aumento do tamanho dos vetores que devem acumular todas as informações produzidas por todos os processos que participam no cálculo. A fim de solucionar esse problema, foram desenvolvidas três soluções variantes da mesma utilizando estratégias como a criação de um laço adicional para contornar a limitação de consumo de memória, a utilização de operações de escrita paralela e geração de réplicas independentes em cada processo. Estas soluções foram validadas e tiveram seu desempenho analisado em relação à versão baseada em operações ponto-a-ponto.

As demais partes do artigo estão organizadas como segue: na seção 2, serão apresentados os trabalhos correlatos; na seção 3 apresenta-se a metodologia utilizada; na seção 4 é detalhado o desenvolvimento, na seção 5 mostra-se os resultados e na seção 6 apresentam-se as considerações finais.

2. TRABALHOS CORRELATOS

Como mencionado anteriormente, atualmente existem duas implementações paralelas do *bootstrap* do *PhyML 3.0*: a versão baseada em operações ponto-a-ponto que foi utilizada como referência para realizar as validações das demais soluções (GUINDON; GASCUEL, 2003); e a versão baseada em operações coletivas que serviu de ponto de partida para o desenvolvimento das soluções propostas (TORRES; DA SILVA, 2018).

2.1. Implementação baseada em operações ponto a ponto

Como mostrado na Figura 1, esse algoritmo possui um laço de repetição principal (linhas 5 a 34), cuja quantidade de iterações correspondem ao número de réplicas *bootstrap* a serem realizadas dividido pelo número de processos a serem utilizados, isto é, se há 100 réplicas *bootstrap* e 4 processos, o valor de *nbRep* será 25, e esse laço será executado 25 vezes.

No início de cada iteração, o algoritmo envia mensagens ponto-a-ponto: o processo 0 envia um vetor diferente para cada um dos outros processos, cujas posições são aleatoriamente embaralhadas e que serão utilizados para alterar os dados de entrada em cada processo. Uma vez que todos os processos tenham recebido seu respectivo vetor, cada processo procede ao cálculo da probabilidade e obtenção de uma árvore e suas estatísticas de forma paralela.

Figura 1. Solução baseada em operações ponto-a-ponto.

```
1  Procedimento Bootstrap_MPI(t_tree *tree)
2  /* número de bootstrap/número de processos */
3    nbRep = tree->mod->bootstrap/Global_numTask
4    nbElem = Global_numTask                          //número de processos
5    for replicate=0:nbRep então
6  /*Processo 0 envia mensagens ponto-a-ponto para cada
7  outro processo com o vetor de embaralhamento*/
8      if Global_myRank = 0 então
9        for i=0:nbElem então
10         MPI_Ssend(boot_data->wght para i+1)
11       fim for
12     else
13       MPI_Recv(boot_data->wght de 0)
14     fim if
15     BLOCO DE INSTRUÇÕES PARA O CÁLCULO DAS ÁRVORES
16     s = Write_Tree(boot_tree, NO)                    //armazena a árvore
17     t = mCalloc(T_MAX_LINE, sizeof(char))
18     Print_Fp_Out_Lines_MPI(boot_tree, replicate+1, t)  //armazena estatísticas da árvore
19   /*Os processos enviam para o processo 0 as árvores e suas estatísticas*/
20     if Global_myRank = 0 então
21       for i=1:nbElem então
22         MPI_Recv(bootStr de i)
23         MPI_Recv(bootStr de i)
24       fim for
25     else
26       MPI_Ssend(s para 0)
27       MPI_Ssend(t para 0)
28     fim if
29     for i=0:2*tree->n_otu-3 então
30       score_par[i] = tree->a_edges[i]->bip_score
31     fim for
32 /*Essa operação coletiva soma todos os valores de "score_par" de cada processo
33 para o vetor "score_tot" do processo 0*/
34     MPI_Reduce(score_par, score_tot, MPI_SUM, 0)
35   fim for
36   if Global_myRank = 0 então
37 /*armazena a informação final levando em conta todas as réplicas bootstrap*/
38     for i=0:2*tree->n_otu-3 então
39       tree->a_edges[i]->bip_score = score_tot[i]
40     fim for
41   fim if
42 fim Procedimento
```

Fonte: Dos autores.

Posteriormente, cada processo armazena suas informações da árvore e suas estatísticas em “s” e “t”, respectivamente (linhas 15 e 17). Em seguida, eles realizam a comunicação ponto-a-ponto novamente, embora desta vez cada processo envie os dados correspondentes a árvore calculada para o processo 0.

Assim, após realizar as iterações nbRep, o laço principal (linhas 5 a 34) termina e então o processo 0 armazena as informações finais levando em consideração todas as réplicas *bootstrap*, finalizando a árvore de consenso.

Em suma, essa solução apresenta um laço principal com *nbRep* iterações. Esse valor é diretamente proporcional ao número de réplicas e inversamente proporcional ao número de processos utilizados. O número de mensagens para cada iteração desse laço depende do número de processos participantes. Em cada iteração, os processos realizam o cálculo da árvore, produzem *P* mensagens no início e *2P* mensagens do mesmo tamanho no final (*P* = número de processos), em adição a operação coletiva *MPI_Reduce*, que é responsável por somar os valores da variável *“score_par”* de todos os processos e guardar em *“score_tot”* no processo 0.

2.2. Implementação baseada em operações coletivas

Como mostrado na Figura 2, a ideia dessa solução é fazer com que o processo 0 gere *nbRep* vetores diferentes (linhas 5 a 12), cujas posições são aleatoriamente embaralhadas, para cada processo e armazene eles em um vetor chamado “*novo_vetor_s*”.

Figura 2. Solução baseada em operações coletivas.

```
1  Procedimento Bootstrap_MPI(t_tree *tree)
2  /* número de bootstrap/número de processos */
3    nbRep = tree->mod->bootstrap/Global_numTask
4    nbElem = Global_numTask                    //número de processos
5    if Global_myRank = 0 então
6  //gera um conjunto de "nbRep" aleatoriamente para cada processo
7      for i=0:nbElem então
8        for k=0:nbRep então
9          novo_vetor_s[i*(nbRep)+z+k] = boot_data->wght[z]
10       fim for
11     fim for
12   fim if
13 /*operação coletiva que distribui para cada processo seu
14 respectivo conjunto de "nbRep" aleatório*/
15   MPI_Scatter(novo_vetor_s, novo_vetor, 0)
16   for k=0:nbRep então                        //loop local para calcular as árvores "nbRep"
17     BLOCO DE INSTRUÇÕES PARA O CÁLCULO DAS ÁRVORES
18     s = Write_Tree(boot_tree, NO)            //armazena a árvore
19     t = mCalloc(T_MAX_LINE, sizeof(char))
20     Print_Fp_Out_Lines_MPI(boot_tree, tree->io, k+1, t)   //armazena estatísticas da árvore
21     for ind=0:T_MAX_LINE então
22       novo_s[ind+k*T_MAX_LINE] = s[ind]        //armezena estatísticas das árvores
23       novo_t[ind+k*T_MAX_LINE] = t[ind]        //armazena as árvores
24     fim for
25     for i=0:2*tree->n_otu-3 então
26       score_par[i] = tree->a_edges[i]->bip_score
27     fim for
28   fim for
29   MPI_Reduce(score_par, score_tot, MPI_SUM, 0)
30 /*Cada processo envia para o processo 0 suas árvores e suas estatísticas*/
31   MPI_Gather(novo_s, bootStr_g, 0)
32   MPI_Gather(novo_t, bootStr_g, 0)
33   if Global_myRank = 0 então
34     for i=0:2*tree->n_otu-3 então
35       tree->a_edges[i]->bip_score = score_tot[i]
36     fim for
37   fim if
38 fim Procedimento
```

Fonte: dos autores.

Em seguida, a operação coletiva *MPI_Scatter* é executada para distribuir os *nbRep* vetores para cada processo e armazená-los no vetor "*novo_vetor*". Logo após, cada processo entra em um laço de repetição (linhas 15 a 27) que calcula localmente cada árvore para cada iteração armazenando suas informações nos vetores "*novo_s*" e "*novo_t*" (linhas 20 a 23), além de salvar as informações parciais das árvores calculadas em "*score_par*" (linhas 24 a 26). Quando o cálculo local das *nbRep* réplicas chega ao fim, a operação coletiva MPI_Reduce é executada para coletar a informação total de todas as réplicas. Por fim, é executada a operação coletiva *MPI_Gather* duas vezes para armazenar as informações das árvores e suas estatísticas de todas as réplicas no processo 0.

Segundo Torres e Da Silva (2018), essa versão apresentou desempenho superior a versão baseada em operações ponto-a-ponto em todos os testes de avaliação de desempenho que foram realizados, os quais consistiram de simulações com 18 arquivos de dados de sequências com 8, 16, 32 e 64 processos, mostrando que a versão baseada em operações coletivas foi mais eficiente que a versão ponto-a-ponto.

No entanto, como foi mencionado na introdução deste trabalho, essa solução apresenta uma limitação no número de réplicas (*bootstrap)* que pode ser utilizado. Isso se deve à quantidade de memória necessária para alocar os vetores que devem acumular todos os dados produzidos por todos os processos participantes do cálculo.

Ticona (2008), aponta que para uma boa fundamentação estatística, é recomendada uma grande quantidade de réplicas (1000 ou mais), porém, em termos práticos, isto demanda grande quantidade de tempo para realizar a análise.

3. MATERIAS E MÉTODOS

A execução deste trabalho foi realizada 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. A arquitetura de nós comuns do CACAU possui 20 nós de cálculo. Cada nó possui 8 processadores Intel® Xeon® E5430 com 2.66GHz, QuadCore, 16 GB de memória RAM e 1 placa gigabit ethernet. Em dados gerais o cluster totaliza 160 processadores e 320 GB de memória RAM. Além disto, o

sistema de arquivos utilizado é o NFS e os nós também são interconectados através de uma rede *InfiniBand*.

A implementação MPI utilizada nas simulações deste trabalho foi o Open MPI 3.1.2.

Além disso, foi utilizado o software *Ktreedist* (SORIA-CARRASCO *et al*., 2007), para realizar a validação das soluções. Esse programa calcula um *K-score* que mede as diferenças gerais no comprimento dos ramos e na topologia relativos a duas árvores filogenéticas depois de escalar uma das árvores para ter uma divergência global que seja mais similar possível à outra árvore. Valores altos de *K-score* indicam pobre correspondência entre a árvore comparada e a árvore referência. Este programa também calcula a distância Robinson-Foulds (RF) (ROBINSON; FOULD, 1981) entre as árvores. A distância RF é o número de partições que estão presentes em uma árvore, mas não na outra e vice-versa. O *Ktredist* fornece a distância RF normalizada. Isto leva a um valor entre 0% e 100%, que pode ser interpretado como a porcentagem na diferença da topologia entre a árvore referência e a árvore comparada.

Inicialmente foi avaliado o comportamento das soluções proposta1 e 2, que serão descritas na próxima seção. Os experimentos foram testados em 4 (4 cores/1 nó), 8 (8 cores/1 nó), 16 (16 cores/2 nós), 32 (32 cores/4 nós) e 64 (64 cores/8 nós) processos, para cada quantidade de processos foram testadas três sequências de entrada e para cada arquivo de entrada foram testadas as soluções proposta1 e 2, juntamente com a solução original do *PhyML* que utiliza operações ponto a ponto, descrita na seção 2.1.

Inicialmente, as sequências de entrada são alinhadas em forma de uma matriz que 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.

Uma das sequências de entrada, a data84, contém dados gerados de maneira simulada. As outras duas, *nucleic_M2764* (57 espécies/803 sítios) e *proteic_M1989* (18 espécies/173 sítios), são dados que consistem em sequências reais de *DNA* e proteína, sendo que tanto a sequência artificial quanto as reais foram retiradas do site oficial do *PhyML* (ATGC, [2. Além disso, todas as simulações foram executadas utilizando 10240 réplicas, segundo Pattengale *et al.* (2009) não existe um valor fixo para o número apropriado de réplicas, mas um critério conservador poderia ser a adoção de vários milhares de réplicas.

Depois de analisar os resultados preliminares, iniciou-se a realização dos experimentos em 32 (32 cores/4 nós) e 64 (64 cores/8 nós) processos. Os parâmetros de entrada utilizados no *PhyML 3.0* para realizar as validações e análises de performance foram -a 0.5 (distribuição gamma), -b 10240 (número de *bootstrap*), -s SPR (técnica heurística de modificação topológica), -o tlr (otimização da topologia da árvore [t], comprimento dos ramos [l] e taxa de substituição [r]) e -c 4 (número de categorias de taxa de substituição). O modelo evolutivo utilizado para as sequências de DNA foi -m HKY85 e -m LG para as sequências de proteína, os mesmos utilizados no *benchmark* publicado no site oficial do *PhyML* (ATGC, [2019]) de onde foram retiradas as sequências de entrada.

Foram escolhidas 20 sequências de entrada de tamanho médio. De acordo com informações do site oficial, esses arquivos de sequências foram retirados do banco de dados de sequências filogenéticas TreeBASE (TREEBASE, 2019). As Tabelas 1 e 2 mostram os dados utilizados, bem como a quantidade de espécies e sítios de cada um.

Uma vez escolhidas as sequências de entrada, foi realizada a validação das

implementações com base nos resultados fornecidos pelo programa *Ktreedist* e o desempenho foi analisado com base no tempo de execução fornecido pelo próprio *PHyML*.

Tabela 1. Dados de DNA utilizados como arquivos de entrada nas simulações do PhyML.

Arquivo de entrada	Quant. Espécies	Quant. Sítios
nucleic_M2564	142	1130
nucleic_M2565	51	1951
nucleic_M2566	73	918
nucleic_M2567	64	1044
nucleic_M2572	191	990
nucleic_M2599	66	1137
nucleic_M2600	69	814
nucleic_M2655	50	872
nucleic_M2902	51	908
nucleic_M2974	56	827

Fonte: (ATGC, [2019]).

Tabela 2. Dados de Proteínas utilizados como arquivos de entrada nas simulações do PhyML.

Arquivo de entrada	Quant. Espécies	Quant. Sítios
proteic_M1372	23	394
proteic_M1373	23	392
proteic_M1375	23	530
proteic_M1376	36	484
proteic_M1384	11	391
proteic_M2344	7	232
proteic_M2345	7	232
proteic_M2637	19	380
proteic_M2638	18	378
proteic_M2954	5	1006

Fonte: (ATGC, [2019]).

4. DESENVOLVIMENTO

A seguir serão apresentadas as três soluções propostas para melhorar o desempenho da paralização do *bootstrap* do *PHyML 3.0*.

4.1. Solução de proposta1

Como a Figura 3 ilustra, a ideia por trás dessa versão se baseou em envolver a maior parte do algoritmo baseado em operações coletivas em um laço adicional (linhas 5 a 33), cujo número de iterações é dado pela divisão *(n°_de_bootstrap/limit)* (linha 4), sendo que "*limit*" é fornecido na entrada da aplicação e representa a quantidade de *bootstrap* que serão processados a cada iteração desse laço. Esta alteração no código permitiu manter os vetores que provocavam erros de memória com um tamanho que não causasse problemas. Portanto, as demais soluções propostas também incluem esta mudança, implicando em aumento de um parâmetro para realizar as execuções do programa. Este parâmetro pode-se ajustar a qualquer tamanho de memória sendo necessário fazer um teste preliminar com diferentes valores de *limit*, esta parte será melhor detalhada na seção de resultados e discussões.

Figura 3. Solução proposta1.

```
1  Procedimento Bootstrap_MPI(t_tree *tree)
2    nbElem = Global_numTask                    //número de processos
3    nbRep = limit/Global_numTask
4    divAll = tree->mod->bootstrap/limit
5    for b=0:divAll então                       //laço externo adicionado
6      if Global_myRank = 0 então
7    //gera um conjunto de "nbRep" aleatoriamente para cada processo
8        for i=0:nbElem então
9          for k=0:nbRep então
10           novo_vetor_s[i*(nbRep)+z+k] = boot_data->wght[z]
11         fim for
12       fim for
13     fim if
14   /*operação coletiva que distribui para cada processo seu
15   respectivo conjunto de "nbRep" aleatório*/
16     MPI_Scatter(novo_vetor_s, novo_vetor, 0)
17     for k=0:nbRep então                        //loop local para calcular as árvores "nbRep"
18       BLOCO DE INSTRUÇÕES PARA O CÁLCULO DAS ÁRVORES
19       s = Write_Tree(boot_tree, NO)            //armazena a árvore
20       t = mCalloc(T_MAX_LINE, sizeof(char))
21       Print_Fp_Out_Lines_MPI(boot_tree, tree->io, k+1, t)   //armazena estatísticas da árvore
22       for ind=0:T_MAX_LINE então
23         novo_s[ind+k*T_MAX_LINE] = s[ind]      //armazena estatísticas das árvores
24         novo_t[ind+k*T_MAX_LINE] = t[ind]      //armazena as árvores
25       fim for
26       for i=0:2*tree->n_otu-3 então
27         score_par[i] = tree->a_edges[i]->bip_score
28       fim for
29     fim for
30     MPI_Reduce(score_par, score_tot, MPI_SUM, 0)
31   //Cada processo envia para o processo 0 suas árvores e suas estatísticas
32     MPI_Gather(novo_s, bootStr_g, 0)
33     MPI_Gather(novo_t, bootStr_g, 0)
34   fim for
35   if Global_myRank = 0 então
36     for i=0:2*tree->n_otu-3 então
37       tree->a_edges[i]->bip_score = score_tot[i]
38     fim for
39   fim if
40 fim Procedimento
```

Fonte: dos autores.

4.2. Solução de proposta2

Esta versão é uma modificação da solução proposta1, isto é, também faz uso do laço de repetição adicional para manter os vetores em um tamanho que evite problemas de memória. Como mostrado na Figura 4, a modificação realizada foi a substituição das operações coletivas *MPI_Gather*, que eram responsáveis por reunir os dados calculados por todos os processos no processo 0, por operações de escrita paralela em arquivo disponíveis no padrão MPI (linhas 34 a 37).

Figura 4. Solução proposta2.

```
1  Procedimento Bootstrap_MPI(t_tree *tree)
2    nbElem = Global_numTask                    //número de processos
3    nbRep = limit/Global_numTask
4    divAll = tree->mod->bootstrap/limit
5    for b=0:divAll então                       //laço externo adicionado
6      if Global_myRank = 0 então
7    //gera um conjunto de "nbRep" aleatoriamente para cada processo
8        for i=0:nbElem então
9          for k=0:nbRep então
10           novo_vetor_s[i*(nbRep)+z+k] = boot_data->wght[z]
11         fim for
12       fim for
13     fim if
14   /*operação coletiva que distribui para cada processo seu
15   respectivo conjunto de "nbRep" aleatório*/
16     MPI_Scatter(novo_vetor_s, novo_vetor, 0)
17     for k=0:nbRep então                      //loop local para calcular as árvores "nbRep"
18       BLOCO DE INSTRUÇÕES PARA O CÁLCULO DAS ÁRVORES
19       s = Write_Tree(boot_tree, NO)                    //armazena a árvore
20       t = mCalloc(T_MAX_LINE, sizeof(char))
21       Print_Fp_Out_Lines_MPI(boot_tree, tree->io, k+1, t)   //armazena estatísticas da árvore
22       for ind=0:T_MAX_LINE então
23         novo_s[ind+k*T_MAX_LINE] = s[ind]              //armezena estatísticas das árvores
24         novo_t[ind+k*T_MAX_LINE] = t[ind]              //armazena as árvores
25       fim for
26       for i=0:2*tree->n_otu-3 então
27         score_par[i] = tree->a_edges[i]->bip_score
28       fim for
29     fim for
30     MPI_Reduce(score_par, score_tot, MPI_SUM, 0)
31     //define em qual parte dos arquivos cada processo escreverá
32     MPI_File_set_view(fh_tree)
33     MPI_File_set_view(fh_stats)
34     //escrita paralela das árvore e suas estatísticas
35     for k=0:nbRep
36       MPI_File_write(fh_tree, novo_s[k*T_MAX_LINE])
37       MPI_File_write(fh_stats, novo_t[k*T_MAX_LINE])
38     fim for
39   fim for
40   if Global_myRank = 0 então
41     for i=0:2*tree->n_otu-3 então
42       tree->a_edges[i]->bip_score = score_tot[i]
43     fim for
44   fim if
45 fim Procedimento
```

Fonte: dos autores.

Tais operações permitem que todos os processos escrevam seus resultados no mesmo arquivo simultaneamente. Para evitar que os dados escritos por cada processo sejam sobrescritos pelos outros processos, é delegado a cada processo um trecho diferente do arquivo (linhas 31 e 32) e, desta forma, cada processo escreve seus resultados em uma "região" diferente do arquivo de saída. A alteração realizada teve como objetivo reduzir o *overhead* gerado pela comunicação entre processos e, com isso, aprimorar o desempenho da aplicação.

4.3. Solução de proposta3

Assim como a solução anterior, esta versão também é uma modificação da solução proposta1. Como apresenta a Figura 5, desta vez, a modificação realizada foi a remoção da operação coletiva *MPI_Scatter*, que era responsável por distribuir os *nbRep* vetores diferentes (que eram todos gerados aleatoriamente no processo 0) para cada processo, armazenando-os no vetor "*novo_vetor*".

Para isso, foi feito com que cada processo gere seus próprios *nbRep* vetores aleatoriamente independentes dos outros e salve-os diretamente no vetor "*novo_vetor*" (linhas 7 a 9), sem necessidade de comunicação entre processos.

Devido à dificuldade de gerar números verdadeiramente aleatórios, a maior parte dos algoritmos geradores de números aleatórios geram números que são apenas "pseudoaleatórios".

Figura 5. Solução proposta3.

```
Procedimento Bootstrap_MPI(t_tree *tree)
  nbElem = Global_numTask                    //número de processos
  nbRep = limit/Global_numTask
  divAll = tree->mod->bootstrap/limit
  for b=0:divAll então                       //laço externo adicionado
/*gera um conjunto de "nbRep" aleatoriamente independente
para cada processo*/
    for k=0:nbRep então
      novo_vetor[z+k] = boot_data->wght[z]
    fim for
    for k=0:nbRep então                      //loop local para calcular as árvores "nbRep"
      BLOCO DE INSTRUÇÕES PARA O CÁLCULO DAS ÁRVORES
      s = Write_Tree(boot_tree, NO)          //armazena a árvore
      t = mCalloc(T_MAX_LINE, sizeof(char))
      Print_Fp_Out_Lines_MPI(boot_tree, tree->io, k+1, t)   //armazena estatísticas da árvore
      for ind=0:T_MAX_LINE então
        novo_s[ind+k*T_MAX_LINE] = s[ind]    //armezena estatísticas das árvores
        novo_t[ind+k*T_MAX_LINE] = t[ind]    //armazena as árvores
      fim for
      for i=0:2*tree->n_otu-3 então
        score_par[i] = tree->a_edges[i]->bip_score
      fim for
    fim for
    MPI_Reduce(score_par, score_tot, MPI_SUM, 0)
  //Cada processo envia para o processo 0 suas árvores e suas estatísticas
    MPI_Gather(novo_s, bootStr_g, 0)
    MPI_Gather(novo_t, bootStr_g, 0)
  fim for
  if Global_myRank = 0 então
    for i=0:2*tree->n_otu-3 então
      tree->a_edges[i]->bip_score = score_tot[i]
    fim for
  fim if
fim Procedimento
```

Fonte: dos autores.

A cada nova execução da função utilizada para gerar os números aleatórios, ela utiliza um valor *r* para calcular um novo número randômico, este valor r é produzido na execução anterior da função e armazenado em uma memória interna dela própria. Porém, para um mesmo valor *r* inicial (chamado de "semente"), a sequência de números gerada pela função será sempre a mesma. Esse é o caso do algoritmo que o *PhyML* utiliza para embaralhar aleatoriamente os *nbRep* vetores de cada processo.

Essa versão original do *PhyML* utiliza o horário atual (momento da execução) em segundos como semente para a geração dos números, isto evita que a semente utilizada em duas execuções seja a mesma, a menos que essas duas execuções sejam realizadas exatamente na mesma hora. Essa estratégia funciona porque todos os números são gerados apenas no processo 0.

No entanto, ao gerar os números em processos diferentes, pode acontecer de dois ou mais processos estarem sincronizados, fazendo com que a semente seja igual para estes e a sequência de números gerada seja idêntica. Para evitar este problema, foi preciso garantir que os processos tivessem sementes diferentes entre si e a cada execução. Nesta versão, a semente de cada processo é obtida da seguinte maneira: *(nº_de_processos x id_do_processo)+(horário_atual).*

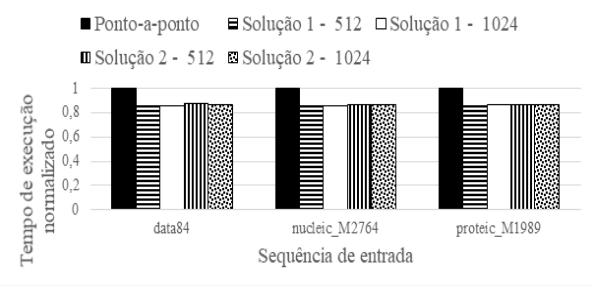
5. RESULTADOS E DISCUSSÕES

Para facilitar a visualização dos resultados apresentados nas subseções 5.1 e 5.3, os tempos de execução de cada sequência de entrada foram normalizados de maneira que, para cada sequência, os tempos de execução foram divididos pelo maior tempo para essa sequência. Assim, os valores apresentados nos gráficos a seguir variam entre 0 e 1, sendo 1 o maior tempo de execução obtido para o arquivo de sequência correspondente.

5.1. Resultados preliminares

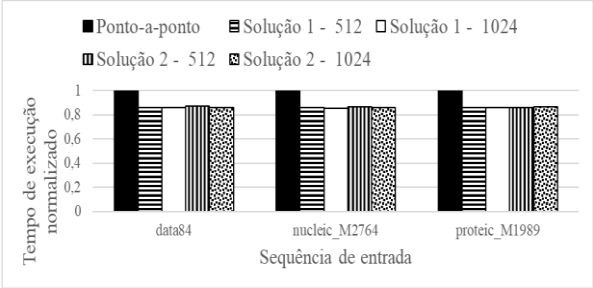
Para cada solução proposta foram utilizados dois valores para a variável *limit* (ver seção 4.1): 512 e 1024. Sendo que 1024 é aproximadamente o valor máximo de *bootstrap* que era possível utilizar na versão baseada em operações coletivas original sem provocar problemas de memória no CACAU. As Figuras 6 até 10 mostram o desempenho correspondente para 4, 8, 16, 32 e 64 processos respectivamente.

Figura 6. Resultados preliminares com 4 processos.



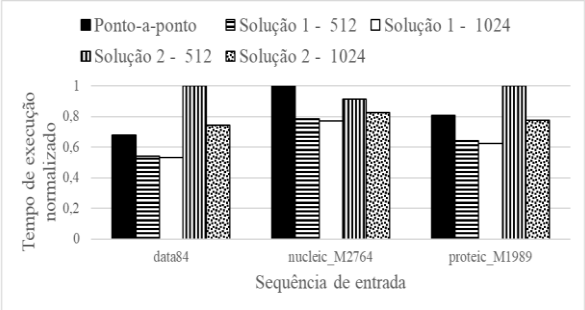
Fonte: dados de pesquisa.

Figura 7. Resultados preliminares com 8 processos.



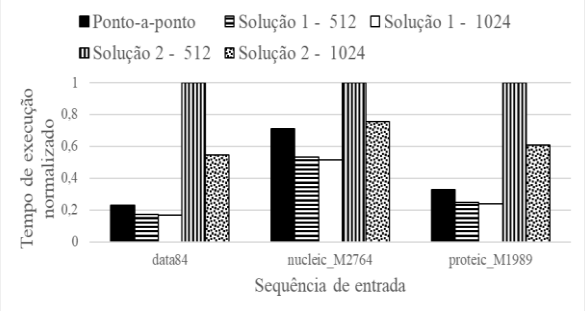
Fonte: dados de pesquisa.

Figura 8. Resultados preliminares com 16 processos.



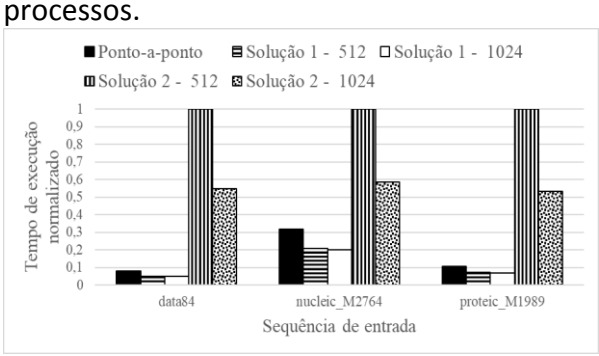
Fonte: dados de pesquisa.

Figura 9. Resultados preliminares com 32 processos.



Fonte: dados de pesquisa.

Figura 10. Resultados preliminares com 64 processos.



Fonte: dados de pesquisa.

Os dados apontam que, no geral, a versão com o melhor desempenho foi a solução proposta1 utilizando 1024 para a variável *limit*, fornecendo resultados equivalentes ou superiores às demais soluções para todas as quantidades de processos utilizados. O pequeno ganho de desempenho em relação à mesma solução utilizando o valor de 512 para *limit* se deve ao maior grau de granularidade da primeira em relação à última. Isto é, a solução proposta1 (1024) realiza mais iterações entre cada operação de comunicação entre processos e, com isso, gera menos *overhead.*

A solução proposta2 apresenta resultados semelhantes à solução proposta1 e superiores à versão ponto-a-ponto para até 8 processos, contudo a partir de 16 processos os valores começam a se distanciar chegando a resultados muito piores que a versão ponto-a-ponto. Além disso, a diferença na granularidade entre as execuções utilizando 512 e 1024 para *limit* se torna ainda mais evidente do que na solução proposta1, uma vez que ao aumentar o número de iterações, também é aumentado o número de operações de escrita paralela, essas que estão limitadas pelo sistema de arquivos utilizado pela máquina onde foram efetuadas as simulações. Tal comportamento e limitação serão discutidos na subseção 5.3.

Visto que os resultados preliminares apresentaram melhores tempos de execução utilizando 1024 para a variável *limit* em ambas soluções desenvolvidas durante o projeto devido ao maior grau de granularidade, as demais simulações neste

trabalho foram realizadas atribuindo somente este valor para a variável.

5.2. Validação das soluções propostas

Com o objetivo de verificar a corretude das implementações das soluções propostas, uma comparação foi feita entre as árvores fornecidas pela versão baseada em operações ponto-a-ponto e as árvores construídas pelas soluções propostas.

A Tabela 3 mostra os valores de *K-score* e RF fornecidos pelo programa *Ktreedist* como mencionado na seção 3.

Esses resultados indicam que as soluções propostas proporcionam resultados equivalentes à versão ponto-a-ponto, apresentando um *K-score* máximo de 0.093 na solução proposta1 e distâncias RF que chegaram a 0% e no geral se mantiveram iguais ou menores a 20%. Benchmarks realizados com o *PhyML 3.0* utilizando a técnica de SPR e publicados no site oficial do *PhyML* (ATGC, 2019) apresentam valores médios de distância RF de 15% para sequências de *DNA* e 14% para sequências de proteína.

Tabela 3. K-score e distância RF normalizada entre a versão baseada em operações ponto-a-ponto e as soluções propostas para 32 processos.

Arquivos de sequência	Solução proposta 1		Solução proposta 2		Solução proposta 3	
	K-score	RF %	K-score	RF %	K-score	RF %
nucleic_M2564	0,093	27%	0,062	13%	0,069	16%
nucleic_M2565	0,036	13%	0,012	8%	0,036	12%
nucleic_M2566	0,048	11%	0,033	10%	0,030	4%
nucleic_M2567	0,001	10%	0,008	12%	0,001	10%
proteic_M1372	0,037	45%	0,001	0%	0,001	0%
proteic_M1373	0,001	20%	0,002	15%	0,001	20%
proteic_M1375	0,088	20%	0,003	0%	0,087	20%
proteic_M1376	0,003	0%	0,060	12%	0,040	9%

Fonte: dados da pesquisa.

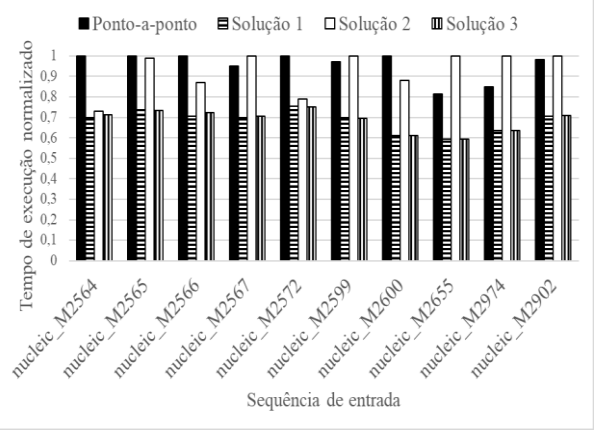
5.3. Análise de desempenho

As Figuras 11 e 12 mostram os tempos de execução normalizados para as três soluções propostas junto com a solução ponto-a-ponto utilizando 32 processos. Sendo que a Figura 11 apresenta os dados de DNA e a Figura 12 os dados de proteína.

Em relação à solução ponto-a-ponto e para 32 processos, a solução 1 teve um ganho de desempenho médio de 28,46%, a solução 2 de 12,13% e a solução 3 de 27,98%.

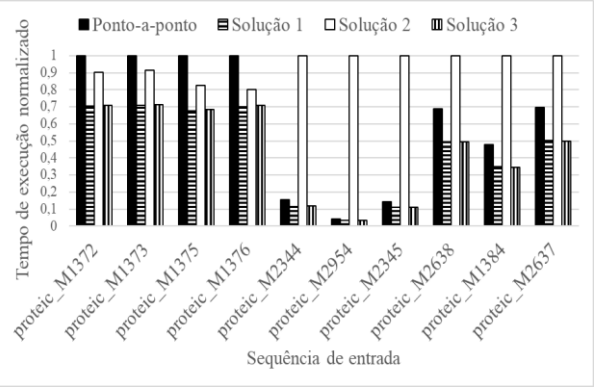
As Figuras 13 e 14 mostram os tempos de execução normalizados para as três soluções propostas junto com a solução ponto-a-ponto utilizando 64 processos. Sendo que a Figura 13 apresenta os dados de DNA e a Figura 14 os dados de proteína.

Figura 11. Tempos de execução normalizados da versão ponto-a-ponto e das três soluções propostas para as sequências de DNA com 32 processos.



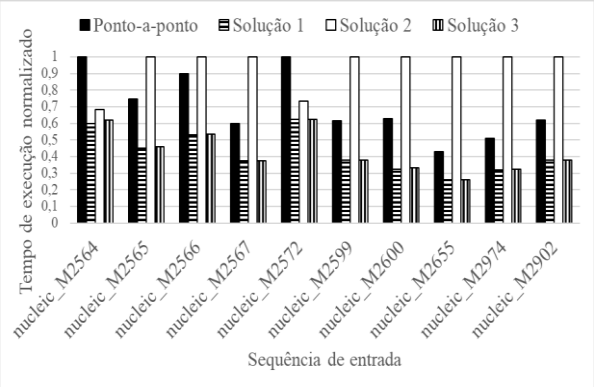
Fonte: dados de pesquisa.

Figura 12. Tempos de execução normalizados da versão ponto-a-ponto e das três soluções propostas para as sequências de proteína com 32 processos.



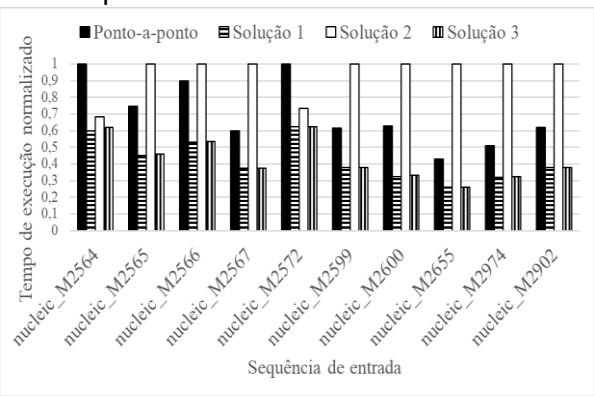
Fonte: dados de pesquisa.

Figura 13. Tempos de execução normalizados da versão ponto-a-ponto e das três soluções propostas para as sequências de DNA com 64 processos.



Fonte: dados de pesquisa.

Figura 14. Tempos de execução normalizados da versão ponto-a-ponto e das três soluções propostas para as sequências de proteína com 64 processos.



Fonte: dados de pesquisa.

Para 64 processos, Figuras 13 e 14, a solução proposta1 apresentou um aumento de performance médio de 39,64% e a solução proposta3 de 38,83%, enquanto a solução proposta2 mostrou uma perda de desempenho média de 13,47%, em relação à versão ponto-a-ponto. Com isso é possível observar que as soluções proposta1 e 3 sempre apresentam resultados equivalentes entre si e superiores às demais soluções, com a solução proposta1 se mostrando levemente superior.

A solução proposta2, que utiliza operações de escrita paralela, sempre apresenta uma perda de desempenho que é proporcional ao número de processos utilizados. Para os testes com 32 processos, Figuras 11 e 12, a perda foi de em média duas horas, e para 64, Figuras 13 e 14, a perda foi de quatro horas em média, em relação às demais soluções propostas. Este problema faz com que esta solução apresente desempenho semelhante ou inferior à versão ponto-a-ponto em sequências de entrada menores (i.e., menos espécies e/ou sítios), enquanto que com sequências maiores como a *nucleic_M2564*, ela fornece resultados superiores à solução ponto-a-ponto e relativamente mais próximos das demais versões.

A causa para essa condição pode ser explicada pela limitação do sistema de arquivos da máquina onde foram realizadas as simulações, o NFS que não é um sistema de arquivos paralelo, mas sim um sistema de arquivos distribuído. Ele foi projetado para permitir que processos em múltiplos computadores acessem um conjunto comum de arquivos, porém ele não foi projetado para dar a vários processos acesso concorrente e eficiente ao mesmo arquivo. Ainda que seja possível realizar operações em arquivos de forma paralela, como foi visto, a perda no desempenho é considerável. Latham *et al.* (2004) discutem alguns impactos dos sistemas de arquivos na escalabilidade do MPI-IO, comparando a performance de operações do mesmo em diversos sistemas de arquivos e mostraram como sistemas propriamente paralelos apresentam melhor desempenho que o NFS. Avila *et al.* (2004) apresentam uma proposta de aprimoramento e avaliação de performance do projeto NFSP (*NFS Parallel*), que visa melhorar a performance paralela dentro de um sistema NFS padrão, e afirmam ter obtido uma melhora de cerca de 5 vezes no desempenho das operações de escrita e o dobro do desempenho nas operações de leitura em relação à implementação anterior. Neste trabalho foi utilizado o NFS, pois a intenção era desenvolver uma versão genérica que pudesse rodar em qualquer *cluster*, sendo que com sistema de arquivos paralelos essa caraterística seria perdida.

Por outro lado, os nós do CACAU são interconectados por meio de uma rede

InfiniBand, que se trata de um barramento serial projetado para oferecer altas taxas de transferência e baixa latência. Esse barramento consegue transferir múltiplos canais de dados ao mesmo tempo com um sinal multiplexado, oferecendo escalabilidade em sistemas paralelos. Isto permitiu que as operações coletivas utilizadas na solução proposta1 pudessem operar com o desempenho apresentado e fez com que a remoção da operação coletiva *MPI_Scatter* na solução proposta3 não fornecesse mudanças significativas de desempenho em relação à solução proposta1. Ismail *et al.* (2013) desenvolvem uma análise de desempenho com operações coletivas do MPI em redes de interconexão *InfiniBand* e *Gigabit Ethernet*, apresentando resultados superiores e consistentes para vários tamanhos de mensagens com a rede *InfiniBand*, principalmente nas operações *MPI_Scatter* e *MPI_Gather*.

6. CONSIDERAÇÕES FINAIS

Diante do problema da limitação do número de *bootstrap* da versão baseada em operações coletivas, foram propostas três soluções variantes da mesma, utilizando estratégias como a inclusão de operações de escrita paralela e geração de réplicas independentes em cada processo. Nesse sentido, foram executadas simulações com objetivo de realizar uma validação e análise de desempenho das soluções propostas em relação a versão baseada em operações ponto-a-ponto.

Os testes de validação utilizando o software *Ktreedist* revelaram que as soluções propostas contornam o problema da limitação do número de *bootstrap* e fornecem árvores equivalentes às árvores utilizadas como referência que foram geradas pela versão ponto-a-ponto, tornando seguro o uso das soluções desenvolvidas para a reconstrução de árvores filogenéticas, mesmo com valores altos de *bootstrap*.

Durante a análise de performance realizada, foi observado que a solução proposta1 apresentou os melhores tempos de execução e consistência entre todos os testes realizados. A solução proposta3 entregou resultados semelhantes à solução proposta1, enquanto que a solução proposta2 demonstrou quedas no desempenho ao aumentar o número de processos devido à limitação do sistema de arquivos utilizado.

Os resultados obtidos neste trabalho fornecem uma alternativa superior à versão baseada em operações ponto-a-ponto para a reconstrução de árvores filogenéticas com o *PhyML 3.0*, além de permitir a utilização de valores altos de *bootstrap* no processo de RAF e/ou a execução em máquinas com quantidade memória limitada. Neste sentido, para trabalhos futuros recomenda-se a exploração das operações paralelas em arquivo do MPI-IO na paralelização do *bootstrap* do PhyML 3.0 dentro de sistemas de arquivos paralelos e analisar seu desempenho.

AGRADECIMENTOS

Agradecemos ao NBCGIB pela infraestrutura e à FAPESB pelo apoio financeiro.

REFERÊNCIAS

ATGC: South of France bioinformatics platform - PhyML 3.0 Benchmarks. Disponível em: http://www.atgc-montpellier.fr/phyml/benchmarks. Acesso em: 06 jul. 2020.

AVILA, R. B.; NAVAUX, P. O. A.; LOMBARD, P.; LEBRE, A.; DENNEULIN, Y. Performance evaluation of a prototype distributed NFS server *In*: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING. 16., 2004, Foz do Iguaçu. **Anais** […]. Foz do Iguaçu: SBAC-PAD, 2004, p. 100-105. https://doi.org/10.1109/SBAC-PAD.2004.33

FELSENSTEIN, J. **Inferring Phylogenies**. Sunderland, Massachusetts: Sinauer Associates, 2004. 664 p.

GUINDON, S.; GASCUEL, O. A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood. **Systematic Biology**, v. 52, n. 5, p. 696-704, 2003. https://doi.org/10.1080/10635150390235520

ISMAIL, R; HAMID, N.; OTHMAN, M.; LATIP, R. Performance analysis of message passing interface collective communication on Intel Xeon quad-core Gigabit Ethernet and InfiniBand clusters. **Journal of Computer Science**, v. 9, p. 455-462, jan. 2013. https://doi.org/10.3844/jcssp.2013.455.462

LATHAM, R.; ROSS, R.; THAKUR, R. The Impact of File Systems on MPI-IO Scalability. *In*: KRANZLMÜLLER, D.; KACSUK, P.; DONGARRA, J. (eds.) **Recent Advances in Parallel Virtual Machine and Message Passing Interface.** EuroPVM/MPI 2004. Lecture Notes in Computer Science, v. 3241 Springer, Berlin: Heidelberg, 2004, p. 87-96. Disponível em: https://link.springer.com/chapter/10.1007/978-3-540-30218-6_18 /. Acesso em: 06 jul. 2020. https://doi.org/10.1007/978-3-540-30218-6_18

PASQUALIN, D. G. **SFREEMAP mapeamento estocástico de árvores filogenéticas livre de simulações.** 2016. Dissertação (Mestrado em Ciências Exatas na área de Informática) - Universidade Federal de Paraná, Curitiba, 2016. Disponível em: https://acervodigital.ufpr.br/bitstream/handle/1884/43801/R%20-%20D%20-%20DIEGO%20GIOVANE%20PASQUALIN.pdf?sequence=3&isAllowed=y. Acesso em : 02 fev. 2021.

PATTENGALE, N. D.; ALIPOUR M.; BININDA-EMOND, O. R. P.; MORET B. M. E.; STAMATAKIS, A. How Many Bootstrap Replicates Are Necessary?. *In*: BATZOGLOU, S. (ed.). **Research in Computational Molecular Biology.** RECOMB 2009. Lecture Notes in Computer Science, v. 5541. Springer, Berlin, Heidelberg, 2009. https://doi.org/10.1007/978-3-642-02008-7_13

ROBINSON, D. F.;FOULDS, L. R. Comparison of phylogenetic trees. **Mathematical Biosciense**s, v. 53, n. 1, p. 131-147, 1981. https://doi.org/10.1016/0025-5564(81)90043-2

SORIA-CARRASCO, V.; TALAVERA, G.; IGEA, J.; CASTRESANA, J. The K tree score: quantification of differences in the relative branch length and topology of phylogenetic trees. **Bioinformatics**, v. 23, n. 21, p. 2954-2956, 2007. https://doi.org/10.1093/bioinformatics/btm466

TICONA, W. G. C. **Algoritmos evolutivos multi-objetivo para a reconstrução de árvores filogenéticas.** 2008. 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. Disponível em: https://www.teses.usp.br/teses/disponiveis/55/55134/tde-02042008-142554/publico/tese_waldo_corregida.pdf. Acesso em : 06 jul. 2020.

TREEBASE: A Database of Phylogenetic Knowledge. [2019]. Disponível em: http://treebase.org/treebase-web/home.html. Acesso em: 06 jul. 2020.

TORRES, M.; DA SILVA J. O. Parallel Solution based on collective communication operations for phylogenetic bootstrapping in PhyML 3.0. *In*: ALVES, R. (eds) **Advances in Bioinformatics and Computational Biology**. BSB 2018. Lecture Notes in Computer Science, v. 11228. Springer, Cham, 2018. https://doi.org/10.1007/978-3-030-01722-4_13

VERLI, H.. **Bioinformática:** da biologia à flexibilidade molecular. São Paulo: SBBq, 2014. 282 p. Disponível em: http http://www.gradadm.ifsc.usp.br/dados/20171/7600011-3/Bioinformatica_1.1.pdf. Acesso em: 06 jul. 2020.