Paralelização da Geração de *Constraints Lists* em Alinhamentos Múltiplos de Sequências Genéticas

Mario João Jr.[1,3], **Alexandre C. Sena**[2], **Vinod E.F. Rebello**[3]

[1] Laboratório Médico de Pesquisas Avançadas, UERJ – Rio de Janeiro – RJ – Brasil

[2]Instituto de Matemática e Estatística, UERJ – Rio de Janeiro – RJ – Brasil

[3]Instituto de Computação, UFF – Niterói – RJ – Brasil

junior@lampada.uerj.br, asena@ime.uerj.br, vinod@ic.uff.br

Abstract. *The alignment of multiple genetic sequences is a crucial step in addressing various problems in the field of Bioinformatics. Since computing optimal alignments is NP-hard, heuristics are commonly employed. Among them, Consistency-Based Alignment yields the best results on average but incurs the highest computational cost. This study presents a parallelization strategy for the core consistency generation steps of this heuristic, along with its most suitable OpenMP scheduling policy. The results obtained with community benchmarks demonstrate the excellent performance of the proposed parallelization, significantly reducing the execution time of consistency generation—from over 4 hours to just under 9 minutes—for alignments of scientifically relevant sizes.*

Resumo. *O Alinhamento Múltiplo de Sequências genéticas é uma etapa essencial na resolução de vários problemas da área de Bioinformática. Devido à sua complexidade exponencial, heurísticas são utilizadas. A que obtém os melhores resultados, mas possui o maior custo computacional, é o Alinhamento Baseado em Consistência. Este trabalho apresenta a paralelização da geração da consistência, fase fundamental para esta heurística, determinando qual a melhor política de escalonamento OpenMP a ser utilizada. Os resultados obtidos mostram o excelente desempenho da paralelização proposta, reduzindo o tempo de execução da geração da consistência, para alinhamentos de tamanhos cientificamente relevantes, de mais de 4 horas para menos 9 minutos.*

1. Introdução

Alinhamento Múltiplo de Sequências (MSA) é o nome dado ao processo de alinhamento de mais de duas sequências genéticas (DNA, RNA ou proteínas) para identificar as similaridades que refletem aspectos relacionados à evolução das espécies e às funcionalidades das estruturas orgânicas [Edgar and Batzoglou 2006]. O alinhamento múltiplo é um passo essencial em pesquisas na área de Bioinformática, como por exemplo: evolução das espécies [Thompson et al. 2011], análise de domínios [Thompson et al. 2011], inferência filogenética [Mirarab and Warnow 2011], e predição de funcionalidades das proteínas e/ou de suas estruturas [Notredame et al. 2000].

Devido à complexidade para encontrar o alinhamento múltiplo ótimo ser $O(l^n)$ [Wang and Jiang 1994], onde $n > 2$ é o número de sequências e l é o tamanho médio delas, heurísticas são utilizadas, sendo a mais comum o *Alinhamento Pro-*

gressivo [Feng and Doolittle 1987]. Essa heurística é a base para diversas ferramentas de alinhamento múltiplo, porém possui características que fazem com que decisões tomadas no início do processo possam não levar às melhores soluções. Para mitigar este problema e melhorar a qualidade do MSA gerado [João Jr et al. 2023], foram criadas heurísticas de *Alinhamento Baseado em Consistência*, fundamentadas nos *Anchor Points* descritos em [Gotoh 1990], que foram implementados, inicialmente, pelo T-Coffee [Notredame et al. 2000]. Essas heurísticas partem do princípio de que um alinhamento múltiplo deve ser consistente. Isto é, se três sequências A, B e C estão sendo alinhadas e o resíduo A_i está alinhado com o resíduo C_k e o resíduo B_j está alinhado com o resíduo C_k, então A_i deve estar alinhado com B_j. Assim, para alinhar duas sequências A e B, os métodos de Alinhamento Baseado em Consistência buscam evidências em um ou mais tipos de alinhamento par a par que envolvem A ou B e alguma outra sequência e que possam fortalecer o posicionamento dos resíduos no alinhamento final envolvendo A e B. Tendo como base o T-Coffee, uma das ferramentas estado da arte para Alinhamento Baseado em Consistência, para que essas evidências possam ser descobertas e utilizadas no alinhamento múltiplo, são necessárias duas etapas: a geração da *Constraints List* (CL) e o alinhamento de *profiles* utilizando a mesma.

A geração da *Constraints List* é a parte mais custosa ($O(l^2 \times n^3)$) do T-Coffee. Para o MSA-XFlow, uma ferramenta desenvolvida pelos autores, que realiza 40 alinhamentos múltiplos para um mesmo conjunto de sequências, o tempo da geração da *Constraints List* chega a representar 70% do total. Nesse contexto, o objetivo do presente trabalho é diminuir o tempo necessário para a geração da *Constraints List* por meio do uso de multiprocessamento. Para isso, essa geração foi paralelizada utilizando a biblioteca OpenMP [1]. Os resultados obtidos mostram claramente o bom desempenho da solução proposta, que não só conseguiu reduzir significativamente o tempo de execução, mas também que ela é escalável ao se aumentar o tamanho do problema e a quantidade de CPUs. Por exemplo, foi possível reduzir o tempo da geração de $\approx 4,3$ horas para apenas $8,8$ minutos, utilizando 32 CPUs, atingindo um *speedup* quase linear de $29,25$.

Alguns trabalhos se propuseram a paralelizar ferramentas de alinhamento múltiplo baseadas em consistência e abordaram a paralelização da geração da consistência. Os autores do PTC, apresentado em [Zola et al. 2007], utilizam trocas de mensagens por MPI para paralelizar o T-Coffee [Notredame et al. 2000]. O PTC gerava as informações que compõem a *Primary Library* (estrutura inicial para armazenamento das evidências) de forma distribuída entre os nós, sem necessidade de exclusão mútua. Já o trabalho [Blazewicz et al. 2013] utilizou CPUs e GPUs para paralelizar o T-Coffee, gerando a *Primary Library* de forma particionada, por questões de armazenamento em GPU, e a remontando ao final. Dessa forma, também não foram utilizadas técnicas de exclusão mútua. Por sua vez, o QuickProbs2 [Gudyś and Deorowicz 2017] é uma ferramenta diferente de alinhamento múltiplo baseada em probabilidades obtidas de cadeias de Markov (HMM). Apesar de também ser um Alinhamento Baseado em Consistência, não há a utilização de *Constraints Lists*, não sendo possível a comparação com a paralelização proposta neste trabalho.

Assim, o trabalho proposto neste artigo apresenta a paralelização da parte mais custosa, em termos de tempo, da heurística de Alinhamento Baseado em Consistência:

[1] https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

a geração da consistência. Além de mostrar como se deram as alterações necessárias nos algoritmos baseados nos códigos-fonte do T-Coffee e do ProbCons [Do et al. 2005], também foi determinada a melhor política de escalonamento OpenMP a ser utilizada na paralelização em questão. A próxima seção aborda o Alinhamento Baseado em Consistência, como um aperfeiçoamento da heurística de Alinhamento Progressivo, e descreve a geração da *Constraints List*, sua principal estrutura de dados. A paralelização da geração desta estrutura de dados é detalhada na Seção 3. Em seguida, a Seção 4 analisa a maneira como a exclusão mútua foi implementada, compara as políticas OpenMP de escalonamento para o problema e apresenta os resultados de desempenho obtidos. Ao final, algumas conclusões e comentários sobre trabalhos futuros são apresentados.

2. Alinhamento Baseado em Consistência e Geração da *Constraints List*

A primeira e mais utilizada heurística para alinhamento múltiplo é o Alinhamento Progressivo, descrito originalmente no trabalho de Hogeweg e Hesper [Hogeweg and Hesper 1984] e consolidado por Feng e Doolittle [Feng and Doolittle 1987]. Neste último trabalho, os autores apresentam o método dividido em três etapas para realizar o alinhamento de múltiplas sequências genéticas baseado no alinhamento par a par das mesmas. Devido às suas características, o método pode inserir lacunas (*gaps*) erradas no início da sua última etapa, que não serão removidas durante o processo e permanecerão até o alinhamento final. Os erros cometidos pelo Alinhamento Progressivo tornam-se mais evidentes quando as sequências a serem alinhadas são pouco similares entre si.

Para melhorar a qualidade dos alinhamentos gerados pelo Alinhamento Progressivo, principalmente quando as cadeias possuem baixa similaridade, foi criada a heurística de Alinhamento Baseado em Consistência. Essa heurística tem como base os *Anchor Points* descritos por Gotoh [Gotoh 1990]. A ideia é que um alinhamento ótimo é consistente e essa consistência é indicada pelos *Anchor Points* que são resíduos em que há transitividade no alinhamento quando se tem como base o alinhamento par a par das sequências. Ou seja, se três sequências A, B e C estão alinhadas, e o resíduo A_i está alinhado com o resíduo C_k e o resíduo B_j está alinhado com o resíduo C_k, então A_i deve estar alinhado com B_j, indicando um *Anchor Point*. Notredame *et al.* desenvolveram a ferramenta referência para alinhamentos baseados em consistência, o T-Coffee [Notredame et al. 2000], e criou uma estrutura para armazenar os *Anchor Points*, chamando-a de *Constraints List*.

Uma *Constraints List* é uma lista de evidências indicando que, no alinhamento múltiplo final, um resíduo A_i deveria estar alinhado com o resíduo B_j. As evidências são representadas por tuplas contendo referências às sequências, aos resíduos envolvidos e um peso para a evidência. Neste trabalho, a *Constraints List* é implementada por uma estrutura de dados de lista ortogonal de três dimensões, conforme ilustrada em roxo, azul claro e azul escuro na Figura 1. Cada elemento da primeira dimensão da lista ortogonal (em azul claro) representa uma sequência (s_i). Cada elemento i da primeira dimensão da lista ortogonal é uma lista onde cada elemento representa um resíduo j de s_i (em azul escuro). Já a terceira dimensão da lista ortogonal é composta por listas de evidências relacionando o resíduo (r_{ij}) com outros resíduos de outras sequências (em roxo).

Para criar a *Constraints List* são necessárias duas etapas. Na primeira etapa, as

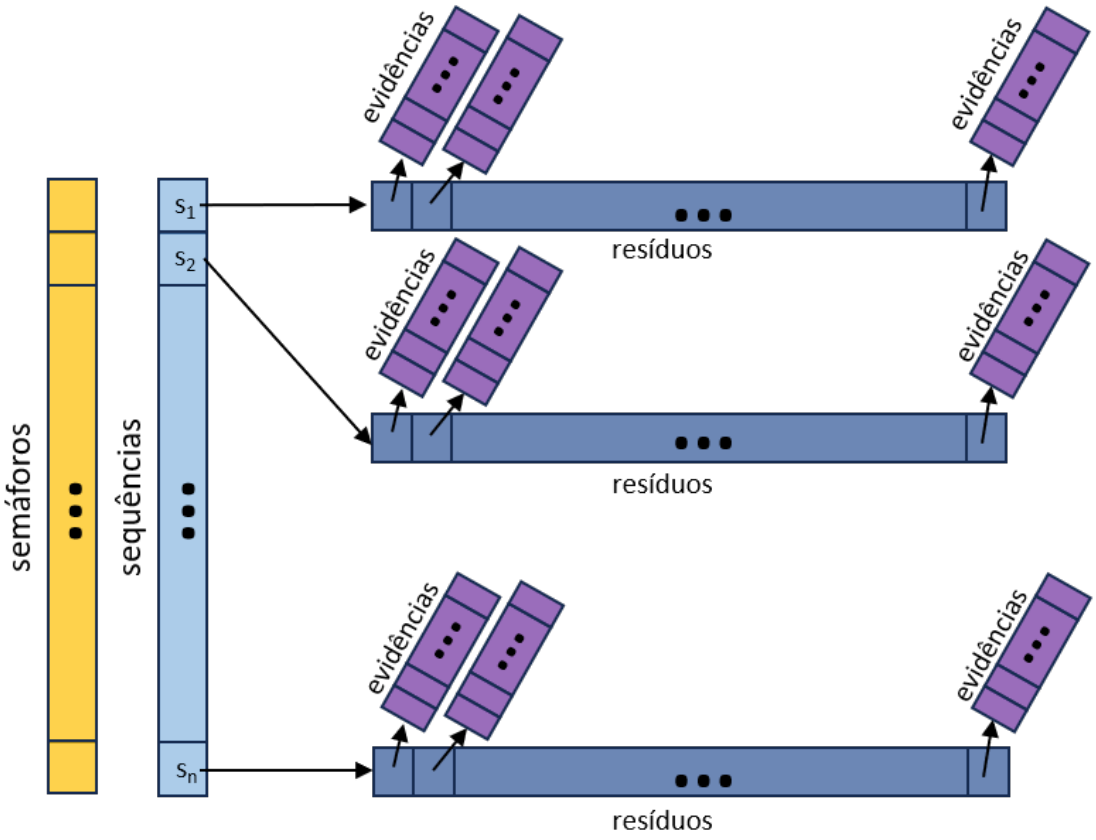

Figura 1. Representação da implementação da *Constraints List*

sequências são alinhadas par a par (Algoritmo 1). Para tal, são realizados $\frac{nseq \times (nseq-1)}{2}$ alinhamentos par a par independentes. Neste trabalho, cada alinhamento é realizado pelo algoritmo de Viterbi [Viterbi 1967] que utiliza HMM, adaptado do código-fonte do ProbCons [Do et al. 2005]. A probabilidade de alinhamento de cada par de resíduos, dada pelo HMM, é utilizada como peso. A conexão (aresta) entre cada par de resíduos alinhados é armazenada numa estrutura de dados similar à *Constraints List*, denominada *Primary Library*, juntamente com seu peso.

Algoritmo 1: Primeira etapa da geração da *Constraints List*

```
1 Seja s o conjunto de sequências e nseq o seu tamanho;
2 para i entre 1 e nseq-1 faça
3 |   para j entre i+1 e nseq faça
4 |   |   Viterbi (s_i, s_j)
```

Como a *Primary Library* é indexada em sua primeira dimensão pela sequência (lista em azul claro da Figura 1), duas inserções são feitas: uma para a aresta da primeira para a segunda sequência e outra para a aresta da segunda para a primeira. A *Primary Library* é armazenada conforme ilustrada em azul e roxo na Figura 1 e dará origem à *Constraints List*. Por exemplo, caso o alinhamento par a par das sequências proteicas $s_1 =$ GWYA e $s_2 =$ GWWRGVC seja:

```
GWY----A
GW-WRGVC
```

Seis tuplas seriam inseridas na *Primary Library*: (1, 1, 2, 1, p_1), (2, 1, 1, 1, p_1), (1, 2, 2, 2, p_2), (2, 2, 1, 2, p_2), (1, 4, 2, 7, p_3) e (2, 7, 1, 4, p_3). Sendo o primeiro par de tuplas referente ao alinhamento dos resíduos G, o segundo par referente aos resíduos W e o último, referente ao alinhamento de A com C. Cabe observar que os pesos são iguais para cada par e que os índices são relativos às sequências originais, e não ao alinhamento par a par. A *Primary Library* é a única estrutura compartilhada por todos os alinhamentos independentes e será transformada na *Constraints List* pelo Algoritmo 2.

Em seguida, a segunda etapa é a transformação da *Primary Library* em *Constraints List*, ilustrada no Algoritmo 2, adaptado do código-fonte do T-Coffee [Notredame et al. 2000]. Para cada aresta da *Primary Library*, é verificada a existência de outras arestas que envolvam o mesmo resíduo, mas em outras sequências (Linhas 2 a 9). Se a soma dos pesos dessas arestas ultrapassar um valor pré-definido (LIMITE), a aresta inicial é mantida na *Constraints List* tendo como peso a média dos pesos das arestas. Caso contrário, a mesma será removida da *Constraints List* (Linhas 10 a 16).

Algoritmo 2: Segunda etapa da Geração da *Constraints List*

```
1  Seja pesos[0 até nseq-1] uma lista ortogonal de duas dimensões;
2  para cada sequência s_1 faça
3      para cada resíduo r_1 de s_1 faça
4          para cada aresta a de s_1 envolvendo r_1 faça
5              peso = n = 0;
6              para cada aresta b de a.s_2 envolvendo a.r_2 faça
7                  peso += b.peso;
8                  n++;
9              se n ≠ 0 então adiciona a aresta a com peso/n à lista pesos[s_1];
10 para cada sequência s_1 faça
11     para cada resíduo r_1 de s_1 faça
12         para cada aresta a de s_1 envolvendo r_1 faça
13             se novo peso de a em pesos[s_1] for maior que LIMITE então
14                 altera o peso de a na lista auxiliar de arestas;
15             senão
16                 remove a da lista auxiliar de arestas;
```

Ao término do Algoritmo 2, as arestas passam a ser interpretadas como evidências de que os resíduos envolvidos possuem grande chance de estarem alinhados. Com isso, a *Primary Library* é totalmente transformada na *Constraints List*, que será salva em arquivo. A transformação de uma estrutura de dados em outra, ao invés da criação de uma nova estrutura, foi necessária para economizar memória.

3. Paralelização da Geração da *Constraints List*

Cada uma das etapas da geração da *Constraints List* foi paralelizada separadamente. Na primeira etapa, devido à independência na execução de cada um dos alinhamentos, estes foram disparados em paralelo. O paralelismo para tal foi implementado utilizando-se a

diretiva OpenMP `parallel for` no laço da linha 2 do Algoritmo 1. Cabe ressaltar que, devido ao valor inicial do índice do laço da linha 3 ($i + 1$), cada *thread* OpenMP disparada irá executar um número menor de iterações em relação à iteração anterior.

Apesar dos alinhamentos serem independentes, existe um trecho de código, adicionado ao Algoritmo de Viterbi, que pode causar condições de corrida: a inclusão das informações na *Primary Library*. A solução clássica para tal problema seria a inclusão de um semáforo de exclusão mútua para impedir que duas *threads* alterassem a *Primary Library* simultaneamente (estrutura ilustrada em azul e roxo na Figura 1). Dessa forma, apenas uma *thread* seria capaz de realizar a inserção em toda a *Primary Library*. Porém, tal implementação limita o paralelismo pois permite que apenas uma *thread* acesse a região crítica por vez. Uma avaliação dessa limitação será apresentada na Subseção 4.1.

Como se trata de uma lista ortogonal, indexada na primeira dimensão pela sequência (ilustrada em azul claro na Figura 1), a solução adotada foi a criação de um vetor de semáforos (ilustrado em amarelo na Figura 1), um semáforo para cada sequência. Reduzindo dessa maneira a área onde faz-se necessária a exclusão mútua (i.e., a cada uma das listas da segunda dimensão), e não a lista estrutura de dados como um todo, permitindo que várias *threads* possam acessar o vetor de sequências simultaneamente. É importante observar que a exclusão mútua só se faz necessária devido à segunda inserção realizada para cada par de resíduos. Se essa não fosse necessária, cada *thread*, e apenas ela, faria a inserção em uma única lista da segunda dimensão da *Primary Library*.

Já no Algoritmo 2 a paralelização foi realizada por meio da mesma diretiva OpenMP `parallel for` nos dois laços mais externos (linhas 2 e 10), uma vez que não há dependência de dados. Nesse algoritmo, os únicos locais onde seria possível ocorrer condições de corrida seriam nos acessos à *Primary Library*. Porém, como cada *thread* altera apenas as arestas (tornando-as evidências) pertencentes à lista de uma sequência (listas em azul escuro na Figura 1), a possibilidade de ocorrer condições de corrida é eliminada.

4. Análise Experimental

Para analisar o desempenho da geração paralela da *Constraints List*, foram usadas sequências de proteínas da família PF00005 [Hung et al. 1998] pertencentes ao *benchmark* PFAM [Finn et al. 2014]. A PF00005 é a maior família de proteínas presente em muitas bactérias completamente sequenciadas e possui uma significativa quantidade de sequências disponíveis. Desta família, foram selecionadas sequências pertencentes a um dos três grupos: tamanhos menores que 100 resíduos; entre 200, inclusive, e 300 resíduos; e entre 400, inclusive, e 500 resíduos. Esses tamanhos escolhidos são bastante significativos, uma vez que, segundo o UniProtKB [The UniProt Consortium 2024], cerca de 81% das sequências proteicas contidas em seu banco de dados possuem tamanhos menores ou iguais a 500 resíduos.

Das milhares de sequências que compõem estes grupos, foram selecionadas amostras com 100 e 500 sequências para avaliar o impacto não apenas dos tamanhos das sequências, mas também da sua quantidade, no desempenho da geração da *Constraints List*. Esta combinação de quantidades e tamanhos de sequências permite avaliar a eficácia da proposta em relação a diferentes tamanhos de problemas de alinhamento múltiplo de proteínas, mesmo com as limitações impostas pelos custos de utilização de infraestrutura

em nuvem. Os experimentos foram executados em uma instância de 32 CPUs da família C7g da AWS EC2 equipada com processadores AWS Graviton 3 com clock de 2,6 GHz, com 2 GiB de memória alocada para cada CPU utilizada. Esta família foi escolhida por apresentar a melhor relação custo/desempenho de acordo com a AWS[2]. Para todos os experimentos, os tempos considerados são a média de três execuções do algoritmo. Apesar do número reduzido de execuções, por questões de custo, as variações nos tempos são pequenas, com o coeficiente de variação chegando no máximo a $3,74\%$.

Para melhor analisar os resultados obtidos, esta seção está dividida em três partes: primeiramente, a Subseção 4.1 mostra a comparação entre a exclusão mútua usual em toda a estrutura de dados referente à *Constraints List* e a maneira como foi realizada neste trabalho. Em seguida, Subseção 4.2, é realizada uma análise do uso das políticas de escalonamento do OpenMP e seus desempenhos. Por fim, são apresentados os *speedups* obtidos pela paralelização da geração da *Constraints List* na Subseção 4.3.

4.1. Mutex $\times$ Vetor de Semáforos

Conforme detalhadamente explicado na Seção 2, a inserção na *Primary Library*, que após a execução do Algoritmo 2 será transformada em *Constraints List*, é realizada duas vezes para cada par de resíduos alinhados durante a execução do algoritmo de Viterbi para alinhamento par a par de sequências. Ao se executar esses alinhamentos em paralelo, podem ocorrer condições de corrida nesta inserção. Assim, a solução clássica para tal é a criação de uma região crítica delimitada pela utilização de um semáforo de exclusão mútua (Mutex) para cada acesso à *Primary Library*, representada na Figura 1 em azul e roxo, de maneira que apenas uma *thread* por vez acesse toda a *Primary Library*.

Ao observar o local onde as duas inserções são realizadas na *Primary Library* (uma das listas indexadas pelo vetor de sequências, em azul claro na Figura 1), percebe-se que a primeira será sempre executada na sequência indexada pelo laço da linha 2 do Algoritmo 1. Tais inserções serão sempre realizadas pela mesma *thread*, uma vez que, pela forma como a paralelização do algoritmo foi feita, cada sequência s_1 é alinhada às outras na mesma *thread*. Assim, as condições de corrida só vão ocorrer quando uma *thread* estiver realizando a primeira inserção na mesma sequência que uma outra *thread* estiver realizando a segunda inserção.

Para uma versão da paralelização da geração da consistência usando Mutex, a Figura 2, gerada pelo *profiler* Linaro Map [Linaro Limited 2025], mostra a porcentagem de utilização das 32 CPUs, representada por uma sequência de finas barras (uma para cada intervalo de tempo), durante a execução da geração da consistência para o exemplo com 500 sequências, com tamanhos de até 100 resíduos. A primeira parte, com pequenas barras verde claro e maioria cinza escuro, é referente à utilização das 32 CPUs durante a execução do Algoritmo 1. As barras em cinza escuro e claro, sobre as barras verde claro, representam a porcentagem de cada intervalo em que as 32 *threads* ficaram, respectivamente, executando operações de sincronização do OpenMP e aguardando o término dessas operações.

Após o Algoritmo 1, as barras grandes em verde claro indicam que todas as 32 CPUs permaneceram com taxa de utilização perto de 100% até próximo do final do Al-

[2]https://aws.amazon.com/ec2/instance-types/ - A família C8g só ficou disponível depois que a avaliação de desempenho tinha começado.

goritmo 2. No fim da execução (E/S), as barras em laranja e verde escuro representam o tempo que a *thread* principal levou para escrever a *Constraints List* em disco. A ociosidade durante a execução do Algoritmo 1, representada pelas barras em cinza escuro na Figura 2, torna ineficiente a utilização da técnica de exclusão mútua, utilizando um único semáforo para toda a estrutura ortogonal quando o tamanho das sequências é pequeno (menores do que 200 resíduos).

Figura 2. Uso de CPU para o exemplo com 500 sequências de até 100 resíduos com exclusão mútua no acesso a toda a estrutura ortogonal

Para otimizar o desempenho da execução paralela do Algoritmo 1, foi utilizado um vetor de semáforos, um semáforo para cada sequência (em amarelo na Figura 1), ao invés de um único semáforo para toda a estrutura ortogonal. A cada inserção, apenas a área referente à segunda dimensão (em azul escuro na Figura 1), relativa à sequência onde a mesma está ocorrendo, é protegida pelo respectivo semáforo. Tal melhoria apresentou, nos experimentos realizados, um tempo de execução quase três vezes menor para o exemplo ilustrado na Figura 2 e tempos de execução $\approx 4\%$ menores para exemplos com sequências com mais de 200 resíduos.

4.2. Análise do Uso das Políticas de Escalonamento OpenMP

O OpenMP oferece três políticas de escalonamento para *threads* disparadas por meio da diretiva OpenMP `parallel for`. São elas: `static`, `guided` e `dynamic`. Todas podem ou não fazer uso do parâmetro *chunk*. Na política `static`, a determinação de que *thread* irá executar que iteração do *loop* é feita antes mesmo das *threads* começarem a ser disparadas. Nela, o número de iterações $niter$ é dividido entre as $nthreads$ *threads*, sendo as primeiras delas responsáveis pelas primeiras $\lceil niter/nthreads \rceil$ iterações, o que se repete para as $niter \mod nthreads$ primeiras *threads*. Cada uma das *threads* restantes é responsável por $\lfloor niter/nthreads \rfloor$ iterações, distribuídas da mesma forma. Quando o parâmetro *chunk* é utilizado, ele determina quais iterações serão executadas por cada *thread*, intercalando as iterações de acordo com o valor do *chunk*.

Por sua vez, quando as políticas `dynamic` e `guided` são utilizadas, a atribuição das iterações do *loop* às *threads* é realizada à medida que as *threads* são disparadas ou terminam alguma iteração. Na política `dynamic`, o número de iterações atribuídas a cada *thread* é determinado pelo parâmetro *chunk* e, caso o mesmo não seja utilizado, o número de iterações atribuídas é um. Tal política apresenta sobrecarga para escolher qual iteração é atribuída a cada *thread*, porém é indicada quando há desbalanceamento entre o tempo de execução de cada *thread*. Já a política `guided` irá atribuir a cada *thread*, um número de iterações proporcional ao número de iterações ainda não atribuídas dividido pelo número de *threads*. Dessa forma, a política inicialmente associa um número maior de iterações por *thread*, seguido de intercalações dinâmicas com números que vão diminuindo até atingir um valor mínimo (o parâmetro *chunk* ou um).

Neste trabalho foram avaliadas quatro configurações envolvendo as políticas de escalonamento do OpenMP: as políticas `dynamic` (**Dynamic1**) e `guided` (**Guided1**)

com suas configurações padrão, ou seja, com *chunk* igual a um; e a política `static` com sua configuração padrão (**Static**) e também com *chunk* igual a um (**Static1**).

A Figura 3 mostra a porcentagem de utilização de 32 CPUs da mesma maneira que a Figura 2, porém utilizando o vetor de semáforos. Ela está dividida em quatro sub-figuras, cada uma representando uma das quatro configurações analisadas: **Dynamic1**, **Static1**, **Static** e **Guided1**. As sub-figuras possuem seus tamanhos proporcionais aos seus tempos de execução e são divididas em três partes por linhas verticais pretas. A primeira parte é referente à execução do Algoritmo 1, a segunda ao Algoritmo 2 e a terceira à execução sequencial da criação, em disco, do arquivo correspondente a *Constraints List*. Neste exemplo, foram utilizadas 500 sequências com tamanhos entre 200 e 300 resíduos.

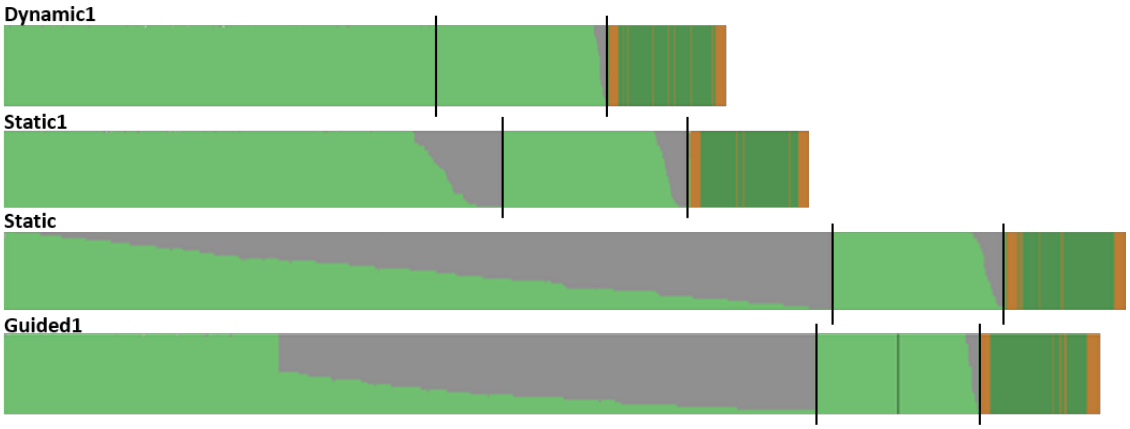

Figura 3. Uso de CPU para cada configuração das políticas OpenMP avaliadas para o exemplo de 500 sequências entre 200 e 300 resíduos com 32 *threads*

A primeira observação a ser feita em relação às execuções utilizando as diferentes políticas é que, por ser executada sequencialmente, o tempo para a geração do arquivo correspondente a *Constraints List* é o mesmo, independente da política escolhida. Também é possível perceber por que a configuração com a política `dynamic` e *chunk* igual a um (sub-figura **Dynamic1**) apresentou o menor tempo de execução: durante quase todo o tempo de execução dos Algoritmos 1 e 2 todas as 32 CPUs apresentaram próximo de 100% de utilização.

Observando a utilização das CPUs durante a execução do Algoritmo 2, percebe-se que, em todas as configurações, há queda na utilização das CPUs próximo do final da execução (barras cinzas na Figura 3), sendo esta mais acentuada nas duas configurações em que a política `static` foi utilizada. Apesar das diferenças de desempenho entre as configurações, ao final da execução do Algoritmo 2, o real impacto das diferentes políticas de escalonamento testadas está na execução do Algoritmo 1.

É importante ressaltar que a cada iteração do *loop* da linha 2 do Algoritmo 1, o número de comparações par a par diminui, ou seja, o trabalho a ser realizado por cada iteração do *loop* da linha 2 diminui. Tendo em vista essa observação, percebe-se o motivo do aumento acentuado da ociosidade exibido na sub-figura **Static** da Figura 3. Como a política `static`, sem o parâmetro *chunk*, divide as iterações, atribuindo as primeiras à primeira *thread*, seguindo assim para as próximas, tem-se que as iterações com o maior trabalho a ser realizado serão atribuídas à primeira *thread*. Desta forma, a última *thread*

terá o menor trabalho a ser executado e terminará antes de todas, ficando ociosa até o término da primeira *thread*. O mesmo ocorre com outras *threads*, seguindo do final para o início. Quando se adiciona o parâmetro *chunk* igual a um à política `static` (subfigura **Static1**), cada *thread* só executa uma iteração do *loop* da linha 2 a cada vez que é escalonada. Com isso, as iterações são executadas de forma intercalada pelas *threads*, mantendo o trabalho equilibrado por mais tempo ao longo da execução do Algoritmo 1. Mas, mesmo assim, o desbalanceamento causado pelo escalonamento estático ainda se faz presente ao final desse algoritmo, como esperado.

A configuração **Guided1** apresenta um comportamento intermediário entre as duas configurações baseadas na política `static`. Isso ocorre devido à divisão inicial utilizada pela configuração. Nela, a quantidade de trabalho dividida inicialmente diminui gradativamente entre as *threads* ao longo da execução. Porém, ainda assim, as primeiras *threads* disparadas serão responsáveis pelas primeiras iterações, deixando as últimas ociosas aguardando pelo término das primeiras. Por fim, pelo fato da política `dynamic` escalonar dinamicamente as *threads* e ao desbalanceamento de trabalho entre as iterações do *loop* da linha 2, o uso do parâmetro *chunk* igual a um fez com que a configuração **Dynamic1** obtivesse o melhor desempenho entre todas as configurações, obtendo também *speedups* próximos de lineares, apesar da sobrecarga inerente a essa política.

4.3. *Speedups*

Os *speedups* obtidos, baseados nas médias dos tempos de três execuções dos experimentos utilizados com 16 e 32 CPUs, são exibidos na Tabela 1. A única razão para não apresentar os *speedups* com 2, 4 e 8 CPUs é o aumento considerável do custo para realizar esses experimentos na nuvem computacional. Nesta tabela, os tempos de referência levam em consideração toda a execução da Geração da *Constraints List*, inclusive a parte sequencial executada após o Algoritmo 2 para a geração do arquivo em disco. É possível observar que os números apresentados estão em consonância com os padrões apresentados no exemplo ilustrado na Figura 3.

Tabela 1. *Speedups* totais para 16 e 32 CPUs

		16 CPUs				32 CPUs			
Seqs.	Tamanho	Dynamic1	Static1	Static	Guided1	Dynamic1	Static1	Static	Guided1
100	até 100	11,52	10,01	6,73	6,74	17,24	14,04	10,16	10,20
500		14,11	13,38	9,48	9,41	24,72	22,63	16,92	16,92
100	200 a 300	13,89	12,36	7,21	7,05	23,68	19,78	11,48	11,26
500		14,49	13,83	8,43	8,59	26,06	23,63	15,84	15,89
100	400 a 500	15,05	13,49	7,36	8,36	27,43	23,55	12,57	14,20
500		15,46	15,01	8,68	8,72	29,25	28,10	16,78	16,64

Pelos motivos abordados na seção anterior, a configuração **Dynamic1** apresenta os melhores *speedups* com eficiência igual a $91,4\%$ para o maior experimento executado (500 sequências entre 400 e 500 resíduos), o que reduziu o seu tempo de execução de mais de quatro horas para um pouco menos de 9 minutos. Esse desempenho é bastante consistente ficando muito próximo do ideal.

As configurações **Static** e **Guided** apresentaram desempenho muito abaixo do ideal, em função do desbalanceamento explicado na subseção anterior. Por sua vez, apesar do bom desempenho da configuração **Static1**, especialmente para 500 sequências

com tamanhos entre 400 e 500, ele fica muito aquém do desempenho alcançado pela configuração **Dynamic1**. Vale a pena ressaltar que a ligeira perda de eficiência da execução com 32 CPUs em relação a 16 CPUs se deve, principalmente, ao tempo de gravação da *Constraints List* em disco, que por ser constante causa um impacto maior nos tempos de execução com 32 CPUs que são menores do que com 16 CPUs.

Para avaliar o ganho de desempenho sem considerar a parte sequêncial da geração da consistência, a Tabela 2 apresenta os *speedups* considerando-se apenas os tempos de execução dos Algoritmos 1 e 2, ou seja, sem considerar o tempo da parte final que grava a *Constraints List* em disco (E/S na Figura 2).

Tabela 2. *Speedups* apenas dos trechos paralelos para 16 e 32 CPUs

		16 CPUs				32 CPUs			
No, Seq,	Tamanho	Dynamic1	Static1	Static	Guided1	Dynamic1	Static1	Static	Guided1
100	até 100	15,52	13,02	7,88	8,03	29,34	20,88	13,06	13,40
500		15,83	14,97	10,24	10,12	30,75	27,23	19,51	19,56
100	200 a 300	15,76	13,76	7,65	7,48	30,26	23,76	12,70	12,48
500		16,00	10,01	8,92	9,09	31,45	28,11	17,68	17,84
100	400 a 500	15,80	14,01	7,51	8,56	30,23	25,36	13,04	14,82
500		15,86	15,38	8,80	8,85	30,80	29,47	17,25	17,12

Na Tabela 2 é possível perceber o quão próximo do *speedup* linear a paralelização dos dois algoritmos ficou. Em particular, ao se observar o experimento com 500 sequências entre 200 e 300 resíduos, tem-se que, para o mesmo, o *speedup* linear foi atingido para 16 CPUs e ficou em $31,45$ para 32 CPUs. Esses resultados destacam claramente a eficiência da paralelização proposta neste trabalho.

5. Conclusões e Trabalhos Futuros

O Alinhamento Baseado em Consistência é uma das principais heurísticas para o problema de alinhamento múltiplo de sequências genéticas. Devido ao seu longo tempo de execução para problemas contemporâneos, a paralelização faz-se fundamental. Uma vez que a geração da consistência é a parte mais custosa dessas heurísticas, este trabalho propôs, implementou e avaliou a paralelização desta etapa, com a geração da *Constraints List*, bem como possíveis políticas de escalonamento OpenMP a serem utilizadas. Os resultados mostraram claramente uma redução significativa no tempo de execução, assim como sua escalabilidade, especialmente quando foram utilizadas 32 CPUs. O uso desta solução permite a execução de instâncias grandes do problema em tempos aceitáveis. A próxima etapa é adicionar ao MSA-XFlow, ferramenta de alinhamento múltiplo desenvolvida pelos autores, uma versão paralela do alinhamento de *profiles*, momento em que a *Constraints List* é efetivamente utilizada para gerar o alinhamento final.

Agradecimentos

Os autores agradecem o apoio do CNPq através dos projetos Universal 404087/2021-3 e CNPq/AWS 421828/2022-6.

Referências

Blazewicz, J., Frohmberg, W., Kierzynka, M., and Wojciechowski, P. (2013). G-MSA - A GPU-based, fast and accurate algorithm for multiple sequence alignment. *Journal of Parallel and Distributed Computing*, 73(1):32–41.

Do, C. B., Mahabhashyam, M. S. P., Brudno, M., and Batzoglou, S. (2005). ProbCons: Probabilistic consistency-based multiple sequence alignment. *Genome research*, 15(2):330–40.

Edgar, R. C. and Batzoglou, S. (2006). Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368–373.

Feng, D.-F. and Doolittle, R. F. (1987). Progressive Sequence Alignment as a Prerequisite to Correct Phylogenetic Trees. *Journal of Molecular Evolution*, 25:351–360.

Finn, R. D., Bateman, A., Clements, J., Coggill, P., Eberhardt, R. Y., Eddy, S. R., Heger, A., Hetherington, K., Holm, L., Mistry, J., Sonnhammer, E. L., Tate, J., and Punta, M. (2014). Pfam: the protein families database. *Nucleic acids research*, 42.

Gotoh, O. (1990). Consistency of optimal sequence alignments. *Bulletin of Mathematical Biology*, 52(4):509–525.

Gudyś, A. and Deorowicz, S. (2017). QuickProbs 2: Towards rapid construction of high-quality alignments of large protein families. *Scientific Reports*, 7.

Hogeweg, P. and Hesper, B. (1984). The alignment of sets of sequences and the construction of phyletic trees: An integrated method. *Journal of molecular evolution*, 20:175–86.

Hung, L.-W., Wang, I. X., Nikaido, K., Liu, P.-Q., Ames, G. F.-L., and Kim, S.-H. (1998). Crystal structure of the ATP-binding subunit of an ABC transporter. *Nature*, 396(6712):703–707.

João Jr, M., Sena, A. C., and Rebello, V. E. F. (2023). On closing the inopportune gap with consistency transformation and iterative refinement. *PLoS ONE*, 18(7):1–24.

Linaro Limited (2025). Linaro forge user guide. Version 25.0.1.

Mirarab, S. and Warnow, T. (2011). FastSP: linear time calculation of alignment accuracy. *Bioinformatics*, 27(23):3250–3258.

Notredame, C., Higgins, D. G., and Heringa, J. (2000). T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205 – 217.

The UniProt Consortium (2024). UniProt: The Universal Protein Knowledgebase in 2025. *Nucleic Acids Research*, 53(D1):D609–D617.

Thompson, J. D., Linard, B., Lecompte, O., and Poch, O. (2011). A comprehensive benchmark study of multiple sequence alignment methods: Current challenges and future perspectives. *PLoS ONE*, 6(3).

Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269.

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

Zola, J., Yang, X., Rospondek, S., and Aluru, S. (2007). Parallel T-Coffee: A parallel multiple sequence aligner. *ISCA International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 248–253.