Algoritmo de nussinov para previsão de estruturas secundárias de RNA

Vítor Hugo Santos de Camargo ra116426

***Resumo*—A previsão de estruturas secundárias de RNA é uma interseção intrigante entre biologia e informática, evidenciando a importância da forma molecular do RNA em suas diversas funções celulares. Diferentemente do DNA, a estrutura do RNA é crucial para a sua função, tornando sua análise, especialmente devido à presença de pseudonós, um desafio. A bioinformática surge como uma solução, processando volumes massivos de dados. Ruth Nussinov, em 1978, lançou as bases com um algoritmo de programação dinâmica. Inspirando-se no algoritmo de programação dinâmica proposto por Ruth Nussinov em 1978, foram realizadas implementações paralelas específicas, utilizando abordagens de pthreads e MPI, buscando otimizar a previsão. Essas implementações não apenas demonstraram diferenças significativas no desempenho, mas também destacaram a vitalidade e a importância da evolução constante na bioinformática aplicada ao RNA."**

***Index Terms*—Nussinov, MPI, pthreads, RNA, DNA.**

I. Introdução

A Previsão de estruturas secundárias de RNA representa um fascinante cruzamento entre a biologia e a informática, refletindo a profunda interdependência entre a forma e a função no mundo molecular. O RNA, diferente de seu primo, o DNA, desempenha uma miríade de funções vitais na célula, desde a síntese de proteínas até a regulação genética. A forma específica ou estrutura que uma molécula de RNA adota é crucial para determinar sua função biológica. No entanto, decifrar essas estruturas é uma tarefa complexa devido à vasta combinatorialidade das possíveis configurações e à presença de pseudonós. Aqui é onde a bioinformática entra em cena. Com a capacidade de processar grandes volumes de dados e utilizar algoritmos sofisticados, a bioinformática tornou-se uma ferramenta indispensável na jornada de desvendar as intrincadas estruturas do RNA e entender seus múltiplos papéis na biologia celular.

II. Descrição

A. *Problema*

RNA, o primo menos famoso do DNA, é responsável por executar e transferir as mensagens genéticas estabelecidas pelo DNA para o seu corpo para tarefas como síntese de proteínas. O DNA codifica e age como um projeto para seus traços genéticos, enquanto os vários tipos de RNA convertem essa informação genética em proteínas úteis, transferindo os projetos de DNA para fora do núcleo e para o ribossomo. Tanto o DNA quanto o RNA são muito interessantes.

Formalmente, podemos definir o RNA como uma sequência S, composta por quatro bases (nucleotídeos); $S \in \{G, C, A, U\}$. Essas 4 bases vêm em pares, conhecidos como pares Watson-Crick, e se acoplam da seguinte forma: $G \leftrightarrow C$ e $A \leftrightarrow U$. Estes pares são os pares de base “fortes/estáveis”, mas, na realidade, também temos o acoplamento de pares menos estáveis, conhecidos como pares canônicos, nomeadamente $G \leftrightarrow U$.

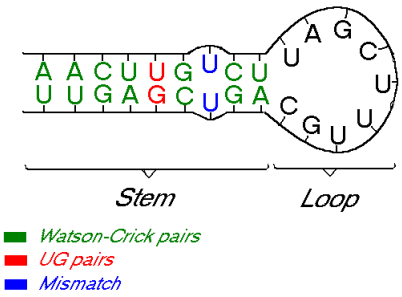


Figura 1: Molécula de RNA mostrando pares de nucleotídeos.

Alguns tipos de RNA podem formar estruturas secundárias ao se reemparelhar consigo mesmos. Essas estruturas secundárias mudam drasticamente as propriedades do RNA. Isso acontece porque, ao contrário do DNA, que tem uma estrutura de dupla hélice, o RNA tem uma estrutura de fita simples. Por causa disso, o RNA às vezes se dobra sobre si mesmo criando essas estruturas secundárias. O processo de dobramento também pode gerar padrões ou formas específicas na estrutura do RNA, como hélices, loops, saliências, junções e pseudonós. Estes podem ocorrer por causa da natureza da dobra, ou porque um nucleotídeo teve um emparelhamento incorreto. Por exemplo, na figura 1 U não pode se acoplar consigo mesmo, criando assim uma pequena saliência.

Existem várias abordagens para resolver o problema de prever estruturas secundárias de RNA, especialmente com os desenvolvimentos recentes no campo da bioinformática e ciência da computação. No entanto, em 1978, uma técnica simples, porém muito eficaz, foi proposta por Ruth Nussinov e colaboradores, que envolveu um algoritmo de programação dinâmica para prever essa estrutura secundária de uma sequência de RNA.

Formalizando o problema, temos:

- Uma molécula de RNA como uma string, seq, onde $seq \in \{G, C, A, U\}$ de comprimento $L = |seq|$.
- Uma estrutura secundária de RNA para seq é um conjunto P de pares de bases ordenadas, escritos como (i, j).
- $j - i > 3$ para não ter bases muito próximas umas das outras e evitar pseudonós.

O algoritmo de Nussinov resolve o problema de prever estruturas secundárias de RNA maximizando os pares de

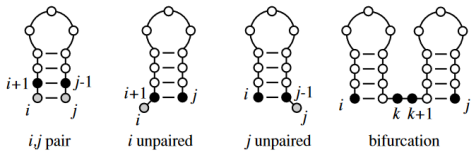


Figura 2: 4 possíveis passos das regras do algoritmo de Nussinov.

bases. Isso também "resolve"o problema de prever estruturas secundárias não cruzadas/pseudonós. Isso é alcançado atribuindo uma pontuação à nossa estrutura de entrada dentro de uma matriz $L \times L$, N_{ij}. Para fazer isso, para cada conjunto emparelhado de nucleotídeos, damos-lhe uma pontuação de +1, e para os outros, 0. Em seguida, tentamos maximizar as pontuações e voltar atrás nos nucleotídeos que maximizam nossa pontuação geral. Para maximizar nossos pares de bases, Nussinov afirma apenas 4 regras possíveis que podemos usar ao comparar nucleotídeos.

Os cálculos principais de Nussinov calculam as melhores estruturas secundárias para pequenas subsequências de nucleotídeos até que alcance as maiores.

O algoritmo:

- Adicione a posição não emparelhada i à melhor subestrutura da subsequência $i+1, j$.
- Adicione a posição não emparelhada j à melhor subestrutura da subsequência $i, j-1$.
- Adicione as bases emparelhadas $i-j$ à melhor subestrutura da subsequência $i+j, j-1$.
- Combine duas subestruturas ótimas i, k e $k+1, j$.

Seja:

$$\text{table}(i,j) = \max \begin{cases} \text{table}(i+1,j) \\ \text{table}(i,j-1) \\ \text{table}(i+1,j-1) + w(i,j) \\ \max\{\text{table}(i,k) + \text{table}(k+1,j)\} \end{cases}$$

onde w é a função de pontuação que avalia o par de sequências $seq[i]$ e $seq[j]$. Para o algoritmo de Nussinov, a função de pontuação retorna 1 se as sequências forem complementares (ou 'A' com 'T' ou 'G' com 'C'), e 0 caso contrário.

e seq: Sequência RNA de tamanho N. As entradas válidas são uma das sequintes:

B. *Estratégia de paralelização*

Através da análise da versão sequencial do algoritmo, definiu-se uma estratégia de paralelização que serviu como base para as duas versões subsequentes do algoritmo. O método realiza iterações calculando os elementos **table[i][j]** iniciando pelo canto inferior direito da matriz. Especificamente, a variável i inicia com o valor correspondente ao total de linhas menos um, e a coluna j começa em i + 1. Embora o processo abranja todas as linhas e colunas, o cálculo dos elementos de fato, efetua-se apenas nas posições situadas acima da diagonal principal da matriz.

Um ponto crucial para a estratégia de paralelização é reconhecer que os elementos em uma dada diagonal são independentes entre si, dependendo apenas dos elementos das diagonais anteriores, ou seja, das diagonais situadas abaixo da diagonal atual. Sendo importante então alguma maneira de sincronização antes dos cálculos inciarem em uma diagonal acima.

O algoritmo de Nussinov apresenta uma complexidade temporal de $O(n^3)$. Esta complexidade advém da estrutura de seus loops: há dois loops principais, referentes às variáveis i e j, e aninhado dentro destes há um terceiro loop, associado à variável k. A combinação destes três loops, cada um percorrendo a sequência, resulta na natureza cúbica da complexidade do algoritmo.

1) ***PTHREADS****:* Nesta implementação, a função principal 'kernel_nussinov' é executada individualmente por cada thread criada. Embora grande parte do código se mantenha semelhante à versão sequencial, há um detalhe crucial no loop 'j'. Na versão original, 'j' inicia em 'i + 1' e incrementa-se em 1 a cada iteração. Contudo, na versão paralelizada utilizando pthreads, ele inicia em 'i + 1 + id da thread' e seu incremento é determinado pelo número total de threads.

Ao criar as threads, cada uma é atribuída com um ID único passado através de uma estrutura do tipo 'thread_args'. Esta identificação desempenha um papel essencial na distribuição das tarefas: uma diagonal que, na versão sequencial, era processada por uma única thread, agora é compartilhada entre todas as threads criadas.

A divisão do trabalho é feita no loop 'j'. Cada thread processa elementos da diagonal atual baseando-se em seu ID. Por exemplo, considerando a criação de 4 threads:
- A thread de ID 0 cuidará dos elementos table[i][0], table[i][4], table[i][8], e assim por diante.
- A thread de ID 1 se encarregará dos elementos table[i][1], table[i][5], table[i][9], seguindo a mesma ideia.

Devido a essa divisão de trabalho, foi necessário inserir uma barreira para garantir que todas as threads completem o processamento da diagonal atual antes de avançar para a próxima.

O tipo de paralelismo aqui feito é o paralelismo de dados, onde cada thread executa a mesma tarefa, porém, com dados diferentes da tabela.

2) ***MPI****:* Esta implementação adota uma estratégia muito semelhante à versão com pthreads. Aqui, após a inicialização do ambiente MPI com 'MPI_Init', cada processo começa a efetuar cálculos na matriz através da função 'kernel_nussinov'. Cada um destes processos utiliza um identificador único denominado "rank", obtido de seu comunicador.

Assim como no código com pthreads, onde o início era determinado por 'i + 1 + id da thread', na versão MPI é determinado por 'i + 1 + rank'. O incremento, que antes era baseado no número de threads, agora é determinado pelo número de processos.

A divisão de trabalho se mantém conceitualmente similar à versão com threads: no loop j, cada processo trabalha em elementos específicos da diagonal atual, baseado em seu rank. Por exemplo, ao considerarmos quatro processos:
- O processo de rank 0 trabalha com os elementos table[i][0], table[i][4], table[i][8] e assim sucessivamente.

- O processo de rank 1 lida com os elementos table[i][1], table[i][5], table[i][9], e assim por diante.

A principal distinção nesta versão MPI é a abordagem de sincronização. Em vez de utilizar uma barreira explícita como no pthreads, emprega-se um método de "broadcast". Na versão com pthreads, cada thread tinha acesso imediato às atualizações na matriz, uma vez que compartilhavam a mesma memória. Contudo, em MPI, cada processo opera em sua própria região de memória. Isso torna crucial que, após atualizações em sua porção da matriz, cada processo compartilhe essas atualizações com todos os outros processos antes de proceder para uma outra diagonal.

Tal como na versão anterior, a estratégia adotada é um paralelismo de dados: cada processo executa a mesma operação, mas em segmentos diferentes da matriz.

III. Análise

A. Configuração utilizada

- **Sistema operacional:** Lubuntu 22.04.2 LTS.
- **Processador:** Ryzen 5 5600x (6 núcleos físicos e simultâneos multithread, totalizando 12).
- **Memória:** 16 gigas de memória RAM DDR4 3600Mhz.

B. Obtenção dos tempos de execução

Para determinar os tempos de execução, utilizamos comandos providos pela biblioteca Polybench. Estes comandos são:

- **polybench_start_instruments**: Inicia a medição do tempo de execução.
- **polybench_stop_instruments**: Encerra a medição do tempo de execução.
- **polybench_print_instruments**: Exibe na tela o tempo total de execução obtido ao final da execução.

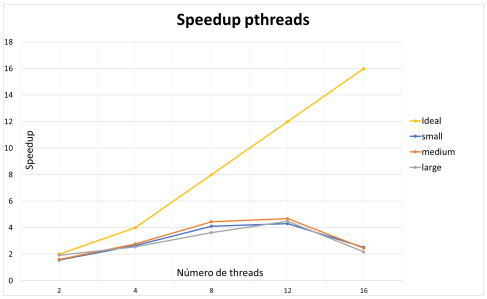
Cabe destacar que a medição dos tempos de execução, nas três versões paralelizadas do algoritmo, abrange desde a primeira até a última linha de código. Isso significa que até mesmo as operações sequenciais — como a alocação das matrizes **table** e **seq** e a criação de threads ou processos — estão inclusas no tempo final apresentado.

As entradas para cada algoritmo são small, medium e large e, possuindo tempos de execução que varial entre 3, 6 e 9 minutos, respectivamente. Cada algoritmo foi executado para cada entrada 5 vezes, variando o número de threads ou processos criados em cada teste, minimizando assim inconsistências nos resultados finais. Os valores de threads ou processos utilizados foram valores múltiplos de 2 (2, 4, 8, 12, 16).

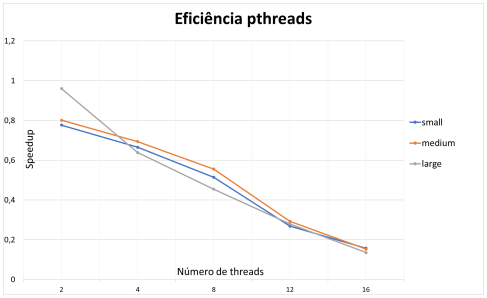
C. Resultados

Antes dos resultados, é importante deinifir os conceitos de speedup e eficiência para um melhor entendimento dos mesmos

O speedup, no contexto de computação paralela, refere-se à melhoria no desempenho alcançada ao executar um programa em um sistema de múltiplos processadores ou threads em relação ao desempenho em um único processador ou thread. É uma métrica crucial para avaliar a eficiência de algoritmos e programas paralelos.



(a) Speedup pthreads



(b) Eficiência pthreads

Figura 3: Speedup e eficiência para pthreads.

A fórmula para calcular o speedup S pode ser dada como:

$$S = \frac{T_1}{T_p} \tag{1}$$

Onde:

- T_1 é o tempo necessário para resolver um problema usando um único processador ou thread.
- T_p é o tempo necessário para resolver o mesmo problema usando p processadores ou threads.

A eficiência é outra métrica importante em computação paralela e é usada para avaliar o desempenho relativo dos processadores em um sistema paralelo. Em termos simples, a eficiência representa quão bem os recursos computacionais são utilizados.

A fórmula para calcular a eficiência E é:

$$E = \frac{S}{p} = \frac{T_1}{p \times T_p} \tag{2}$$

nessa equação, p é o número de processadores ou threads e S é o speedup. A eficiência pode variar de 0 a 1 (ou 0% a 100%). Uma eficiência de 1 (ou 100%) indica que todos os processadores estão sendo usados de maneira ideal, enquanto valores menores indicam que algum recurso está sendo subutilizado.

*1) **PTHREADS**:* A figura 3 mostra os gráficos tanto para o speedup quanto para a eficiência do código utilizando pthreads, para as entradas small, medium e large, variando o número de threads no eixo x.

Independente do tamanho da entrada, o speedup observado está distante do ideal, contudo, permanece no território do aceitável. Nota-se que, até 8 threads, há um crescimento consistente do speedup. Esta tendência sugere uma distribuição eficiente de carga entre as threads e uma gestão eficaz das mesmas. Contudo, conforme esperado, ao incrementar o número

de threads, em especial quando nos aproximamos do limite total de processadores disponíveis, observa-se um atenuamento no crescimento do speedup ou até mesmo uma diminuição. Tal fenômeno pode ser justificado pela sobrecarga gerada pelo gerenciamento, comunicação e sincronização dessas threads.

As threads podem entrar em competição por recursos como cache ou memória. Isso pode resultar em conflitos e atrasos, manifestando-se em fenômenos conhecidos como cache-misses, contention (quando há concorrência pelo mesmo recurso) e thrashing (situação em que a disputa por recursos conduz a um desempenho substancialmente degradado).

Um padrão evidente nos testes é que, com o aumento tanto do número de threads quanto do tamanho da entrada, a porcentagem de cache-misses também aumenta. Enquanto para a entrada 'small', essa porcentagem oscilou entre 30% e 34%, para a entrada 'medium' essa taxa subiu para uma média de aproximadamente 39%. Mais impactante ainda, a entrada 'large' registrou um pico de 49% de cache-misses com 16 threads.

Adicionalmente, é claro que, conforme o tamanho da entrada e o número de threads crescem, o número de instruções por ciclo diminui. Isso indica que, mesmo com um volume de trabalho computacional ampliado, a eficiência geral é comprometida.

Um último ponto a ser observado é a substancial elevação no número de context-switches à medida que as entradas e threads aumentam. Por exemplo, na entrada 'large' com apenas 2 threads, o número de context-switches rondou os 4 mil. Com 16 threads, esse número saltou para 200 mil.

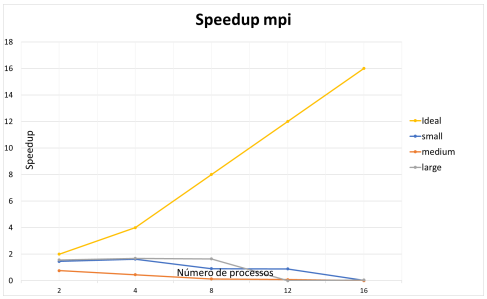
A eficiência do pthreads demonstra um comportamento bem esperado com base no speedup, com destaque para a entrada large, se situando bem próximo do valor 1 (ideal) para 2 threads, provavelmente devido ao fato de um balanceamento entre pouco gerenciamento devido à poucas threads com a vantagem de ter mais de um fluxo trabalhando.

*2) **MPI**:* Na figura 4 estão os gráficos tanto para o speedup quanto para a eficiência do código utilizando mpi, para as entradas small, medium e large, variando o número de processos no eixo x.

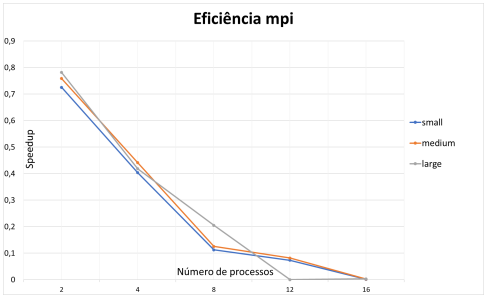
Novamente, independentemente do tamanho da entrada, o speedup observado, quando utilizando MPI, mostra-se distante do ideal. Além disso, é interessante notar que, a partir de 4 processos, o speedup começa a se afastar de um valor aceitável. Em contraste com os resultados obtidos com pthreads, onde o speedup cresce até um certo ponto, com MPI, o comportamento é distinto. Apesar do speedup para 2 processos bem próximo do ideal, observa-se uma queda gradual à medida que o número de processos aumenta, independentemente do tamanho da entrada.

Essa tendência pode ser explicada pelo overhead associado à comunicação entre os processos. Após cada processo calcular seus elementos designados, há a necessidade de realizar um broadcast antes de iniciar o cálculo de uma nova diagonal. Esta etapa de comunicação tem um custo significativo, uma vez que todos os processos devem compartilhar suas versões atualizadas da matriz com todos os outros processos.

Um aspecto relevante é a taxa de cache-misses, que em todas as entradas e variações do número de processos, mostrou-



(a) Speedup mpi



(b) Eficiência mpi

Figura 4: Speedup e eficiência para mpi.

se entre 7% a 10% menor em comparação aos testes realizados com pthreads. Isso pode ser atribuído ao isolamento de memória proporcionado pelo MPI, onde cada processo opera em seu próprio espaço de memória, reduzindo conflitos de acesso.

Assim como observado com Pthreads, à medida que o tamanho da entrada e o número de processos aumentam no MPI, o número de instruções por ciclo tende a diminuir. Isso sugere que, mesmo quando há mais trabalho computacional, a eficiência total do processamento é afetada.

Por fim, vale destacar o número de context-switches em MPI, que se mostrou significativamente menor em comparação ao observado com Pthreads, particularmente em entradas de maior tamanho.

A figura 4 mostra a eficiência para o código paralelizado utilizando mpi para as entradas small, medium e large, variando o número de processos.

O comportamento da eficiência para MPI é notavelmente diferente do pthreads, revelando fatores interessantes. Fatores como o cache-misses citado anteriormente, onde são bem menores que versões como pthreads e sequencial com certeza influênciam de forma positiva.

IV. Conclusão

Ao longo deste relatório, a paralelização do algoritmo de Nussinov foi explorada em detalhe utilizando diferentes abordagens: pthreads e MPI. O principal objetivo dessas técnicas de paralelização é acelerar a previsão de estruturas secundárias de RNA, um problema fundamental em bioinformática.

Os resultados obtidos através das duas abordagens paralelas mostraram um aumento notável no desempenho quando comparados à versão sequencial do algoritmo. Contudo, cada abordagem tem suas próprias nuances e características:

- A abordagem com **pthreads** demonstrou ser eficiente até certo ponto, alcançando um speedup e eficiência notáveis com um número menor de threads. No entanto, ao se aproximar do limite do número de processadores disponíveis, a eficiência começou a decair, provavelmente devido ao overhead associado à gestão e sincronização das threads.
- A abordagem com **MPI** mostrou um comportamento diferente. Mesmo que o speedup inicial para dois processos fosse significativamente alto, houve uma tendência de queda à medida que mais processos eram introduzidos. Isso pode ser atribuído ao custo de comunicação entre processos, que se torna mais significativo à medida que mais processos são adicionados.

É evidente que não existe uma abordagem única que seja ideal para todos os cenários. A escolha da técnica de paralelização dependerá do tamanho do problema, da arquitetura do sistema e de outros fatores específicos do ambiente. Além disso, o balanço entre computação e comunicação é um fator crítico na determinação do desempenho em ambientes paralelos.

Para esse algoritmo, por exemplo a abordagem utilizando threads foi bem mais eficiente que à utilizando processos, porém, pode nem sempre ser o caso, principalmente quando o mpi é utilizadm em clusters, o que não foi o caso devido à limitação de utilização de uma máquina só.

Em conclusão, a paralelização do algoritmo de Nussinov, através das técnicas discutidas, apresenta uma promissora solução para acelerar a previsão de estruturas secundárias de RNA. A medida que os conjuntos de dados na bioinformática crescem, a eficiência desses algoritmos se tornará cada vez mais crucial, tornando as abordagens de paralelização ainda mais relevantes no futuro da pesquisa em bioinformática.

REFERÊNCIAS

[1] W. Bielecki, P. Błaszyński, and M. Pałkowski, "3d tiled code generation for nussinov's algorithm," *Appl. Sci.*, vol. 12, p. 5898, 2022. Academic Editors: Nasro Min-Allah and Ubaid Abbasi.

[2] T. Yuki and L.-N. Pouchet, *PolyBench 4.2.1 (pre-release)*. Inria / LIP / ENS Lyon and Ohio State University, 2016. Available at https://web.cs.ucla.edu/~pouchet/software/polybench/.