

*Uma abordagem MPI para transcrição e tradução de
DNA*

Elihofni Lima
Erick Grilo
Max Fratane

1 Introdução e descrição da aplicação

A transcrição do DNA é o processo através do qual o DNA serve de modelo para a síntese de RNA feita por um ser vivo. Apenas uma cadeia de DNA é usada nesse processo, ativada pela enzima RNA-polimerase. Em uma determinada região da molécula de DNA, ocorre a separação das hélices, onde uma delas forma o RNA através do encadeamento de nucleotídeos complementares. Em suma, essa é a fase responsável por parear as bases nitrogenadas do DNA com as do RNA: A do DNA com U do RNA, T do DNA com A do RNA, C do DNA com G do Rna e G do DNA com C do RNA.

Relembrando alguns conceitos de biologia, A (adenina), C (citossina) T (timina) e G (guanina) são os nucleotídeos que compõem o DNA, e as mesmas, com exceção da timina (que vira U, de uracila, que por sua vez é uma base nitrogenada), compõem o RNA. Cada trinca de 3 dessas bases nitrogenadas é chamada de códon. Um códon codifica um aminoácido, vide Morandini et al. (2013).

Motivado pelo tamanho que uma cadeia de DNA pode ter (em Venter et al. (2001), no mapeamento do genoma humano, por exemplo, foram encontradas aproximadamente 2.91 bilhões de pares de bases nitrogenadas) e pelo processo de transcrição ser uma tarefa repetitiva, a criação de um programa paralelo do tipo SPMD (Simple Program, Multiple Data) aparenta ser uma boa abordagem para a solução desta tarefa. Chibli (2008) nos dá uma abordagem usando MPI para solucionar problemas de comparação de *strings*, incluindo a transcrição de DNA para RNA e a identificação de aminoácidos, enquanto em Kleinjung et al. (2002) e Xue et al. (2014), o padrão MPI é utilizado para outro fim, que é o alinhamento de sequências de DNA (que visa encontrar similaridades no DNA que podem indicar relações evolucionárias entre diferentes indivíduos, ou seja, características semelhantes entre indivíduos de espécies diferentes.)


O objetivo é paralelizar a transcrição e a tradução de DNA, onde a transcrição é o processo responsável por traduzir uma cadeia de DNA para RNA e a tradução consiste em identificar o aminoácido que o códon em questão representa, a partir de códon de RNA (que foram obtidos a partir da transcrição de códon de DNA para RNA). Note que o processo só se inicia efetivamente quando na cadeia original de DNA dado na entrada é identificado um códon (uma cadeia de DNA iniciada pelos códon compostos por ATC, ACT ou ATT, que possui informações para a síntese de uma proteína). Enquanto um códon não é encontrado, não há nenhum processamento efetivo. Quando um códon é encontrado, o processo se inicia. Como estamos considerando nenhuma mutação genética, o tamanho do códon deve ser um número múltiplo de 3 para que o algoritmo funcione corretamente. A cadeia original pode ter qualquer tamanho.

2 A implementação sequencial

O método sequencial consiste em tratar toda a cadeia de entrada como uma única cadeia, oriunda da leitura de um arquivo texto onde se encontra tal cadeia. A partir daí, toda a cadeia do DNA é transcrita para RNA da seguinte forma: primeiramente, é identificada o local correto para iniciar a transcrição da cadeia (onde o códon se encontra). A cadeia então é subdividida em grupos de três bases nitrogenadas (um códon) e em seguida, a cada códon, sua transcrição (conversão de DNA para RNA) é feita e a identificação do aminoácido que aquele códon se refere é feita em seguida. Ao término da análise de um códon, o códon DNA original, a sua transcrição para códon RNA e o aminoácido que ele representa é escrito em uma linha do arquivo e esse procedimento se repete por todo o tamanho da cadeia de entrada. Caso o códon não possui 3 nucleotídeos (o tamanho do códon não é múltiplo de 3), tal códon é ignorado.

Para a execução da aplicação sequencial, usa-se:

- Compilar: 'gcc main-seq.c transcription.c io.c -o dna-seq'
- Executar: './dna-seq'



```
erick@erick-SS50CA: ~/trabalho-sd
erick@erick-SS50CA:~/trabalho-sd$ ./main
Insira o nome do arquivo de entrada (até 32 caracteres):
dna1.txt

Processo Mestre: CISTRON INICIA NO INDICE 2
TAMANHO DO CISTRON 60

-RESULTADOS:-
DNA RNA AMINO
UUU Phe AAA
UUG Leu AAC
CCG Pro GGC
CAU His GTA
CGU Arg GCA
UUU Phe AAA
UUG Leu AAC
CCG Pro GGC
CAU His GTA
CGU Arg GCA
UUU Phe AAA
UUG Leu AAC
CCG Pro GGC
CAU His GTA
CGU Arg GCA
UUU Phe AAA
UUG Leu AAC
CCG Pro GGC
CAU His GTA
CGU Arg GCA
UUU Phe AAA
UUG Leu AAC
CCG Pro GGC
CAU His GTA
CGU Arg GCA
UUU Phe AAA
UUG Leu AAC
CCG Pro GGC
CAU His GTA
CGU Arg GCA
Tempo Total: 0.001205s
erick@erick-SS50CA:~/trabalho-sd$
```

Figura 1: Exemplo de uma execução da aplicação sequencial, indicando onde o cístoron é encontrado na cadeia origina, seu tamanho e o tempo total de execução.

3 A implementação paralela

O método paralelo consiste em paralelizar todo o processo de transcrição de DNA em RNA e o de tradução do códon para um aminoácido. Dessa forma, o trabalho é dividido em n tarefas (1 tarefa mestre, $n-1$ tarefas escravo) onde a tarefa mestre é a responsável por repartir a sequência de DNA de entrada para as demais tarefas, ficando com a primeira parte da sequência. Após o envio dessas partes da entrada para cada uma das tarefas escravo, a tarefa mestre faz a transcrição da sua parcela da sequência de DNA para RNA, ao mesmo tempo que as tarefas escravo (após receberem a mensagem que contém os dados da tarefa mestre) também iniciam esse processo. A tarefa mestre então realiza a identificação dos aminoácidos que ela transcreveu e fica aguardando as demais tarefas escravo realizarem a parcela do seu processamento (transcrever e identificar os aminoácidos que lhe foram enviadas).

Quando todas as tarefas escravo terminarem o seu processamento, elas enviam de volta os resultados para a tarefa mestre (os códons transcritos e os aminoácidos identificados), que por sua vez imprime na tela o RNA transcrito e os aminoácidos identificados e escreve em um arquivo tais dados.

```
erick@erick-S550CA:~/trabalho-sd$ erick@erick-S550CA:~/trabalho-sd$ nprun -np 4 dna-mpi
dna2.txt

Processo Mestre: INICIOU
Processo Mestre: LEITURA NO ARQUIVO CONCLUIDA
Processo Mestre: CISTRON INICIA NO INDICE 10
Processo Mestre: TAMANHO DA CADEIA LIDA = 131
Processo Mestre: TAMANHO DO CISTRON = 122
Processo Mestre: TOTAL DE CODONS = 40
Processo Mestre: 10 CODONS POR PROCESSO
Processo Mestre: ENVIOU CODONS DNA PARA PROCESSO #1
Processo Mestre: ENVIOU CODONS DNA PARA PROCESSO #2
Processo Mestre: ENVIOU CODONS DNA PARA PROCESSO #3
Processo Mestre: TRANSCRICAO PARCIAL CONCLUIDA
Processo #2: INICIOU
Processo #2: RECEBEU PARTE DA CADEIA DE DNA DO MESTRE
Processo #2: TRANSCRICAO PARCIAL CONCLUIDA
Processo #2: IDENTIFICACAO DE AMINOACIDOS PARCIAL CONCLUIDA
Processo #2: ENVIU RESULTADOS PARA O PROCESSO MESTRE
Processo #2: FINALIZOU
Processo #3: INICIOU
Processo #1: INICIOU
Processo Mestre: IDENTIFICACAO DE AMINOACIDOS PARCIAL CONCLUIDA
Processo Mestre: RECEBEU RNAs DO PROCESSO #1
Processo #1: RECEBEU PARTE DA CADEIA DE DNA DO MESTRE
Processo #1: TRANSCRICAO PARCIAL CONCLUIDA
Processo #1: IDENTIFICACAO DE AMINOACIDOS PARCIAL CONCLUIDA
Processo #1: ENVIU RESULTADOS PARA O PROCESSO MESTRE
Processo #1: FINALIZOU
Processo #3: RECEBEU PARTE DA CADEIA DE DNA DO MESTRE
Processo #3: TRANSCRICAO PARCIAL CONCLUIDA
Processo #3: IDENTIFICACAO DE AMINOACIDOS PARCIAL CONCLUIDA
Processo #3: ENVIU RESULTADOS PARA O PROCESSO MESTRE
Processo #3: FINALIZOU
Processo Mestre: RECEBEU RNAs DO PROCESSO #2
Processo Mestre: RECEBEU RNAs DO PROCESSO #3
Processo Mestre: TRANSCRICAO TOTAL CONCLUIDA
Processo Mestre: RECEBEU AMINOS DO PROCESSO #1
Processo Mestre: RECEBEU AMINOS DO PROCESSO #2
Processo Mestre: RECEBEU AMINOS DO PROCESSO #3
```

(a) Início da execução, mostrando o *status* dos processos, tamanho total da cadeia de entrada, tamanho do cístron que será processado, o total de códons, a quantidade de códons enviados para cada processo escravo e parte do resultado final.

```

erick@erick-SS50CA: ~/trabalho-sd
GCG CCG Pro
GTA CAU His
GCA CGU Arg
AAA UUU Phe
AAC UUG Leu
GCG CCG Pro
GTA CAU His
GCA CGU Arg
AAA UUU Phe
AAC UUG Leu
GCG CCG Pro
GTA CAU His
GCA CGU Arg
AAA UUU Phe
AAC UUG Leu
GCG CCG Pro
GTA CAU His
GCA CGU Arg
AAA UUU Phe
AAC UUG Leu
GCG CCG Pro
GTA CAU His
GCA CGU Arg
AAA UUU Phe
AAC UUG Leu
GCG CCG Pro
GTA CAU His
GCA CGU Arg
AAA UUU Phe
AAC UUG Leu
GCG CCG Pro
GTA CAU His
GCA CGU Arg
GCA CGU Arg
Processo Mestre: ESCRITA DE RESULTADOS NO ARQUIVO CONCLUIDA
Tempo total: 0.00075s
erick@erick-SS50CA:~/trabalho-sd$

```

(b) Fim da execução, mostrando a confirmação do recebimento dos dados processados pela tarefa mestre, os dados processados (que foram escritos em um arquivo e o tempo total de execução

Figura 2: Exemplo de uma execução da aplicação paralela, no ambiente MPI com dois processos, um mestre e um escravo

4 A implementação paralela

4.1 Ferramentas

O ambiente escolhido para a paralelização da implementação foi o MPI (*Message Passing Interface*, um padrão de comunicação de dados em computação

paralela cujo objetivo busca efetuar a troca de mensagens entre processos ou threads, dependendo da implementação) de forma prática e eficiente. A implementação do padrão MPI utilizada é a OpenMPI (Gabriel et al. 2004) escrito em C e a linguagem utilizada é C.

A nossa implementação consiste no uso do ambiente MPI para paralelizar o processo: Ao iniciar o ambiente MPI, o ambiente de execução paralela é inicializado. Em seguida, as tarefas são criadas e inicializadas no comunicador padrão (MPI_COMM_WORLD) e, então, existem duas rotinas possíveis para serem executadas por cada uma das tarefas, que é a rotina destinada para a tarefa mestre (identificada pelo rank 0) e outra para as demais tarefas (cujo identificadores variam de 1 a N-1, N sendo o número de processos).

No padrão MPI na implementação OpenMPI, as principais características são:

- **Transparência:** O padrão MPI fornece transparência de acesso (Fica transparente ao usuário como as máquinas acessam um dado recurso), transparência de concorrência: no MPI, chamadas síncronas podem ser feitas usando os comandos *MPI_Send* (envio de mensagens) e *MPI_Recv* (recebimento de mensagens) cuja implementação é transparente ao usuário. Vale notar que o MPI também fornece chamadas assíncronas porém o padrão MPI não fornece transparência de localização, de migração nem de relocalização (para fazer uso de várias máquinas com uma aplicação MPI, é necessário ter um arquivo com os endereços IP das máquinas onde os processos MPI irão executar. Ou seja, é necessário saber a localização física das máquinas e as máquinas não podem se movimentar) e nem transparência de falha: a tolerância a falhas em MPI é zero (em uma aplicação MPI, se um processo falha, todo o *job* (a aplicação) MPI falha (é abortada)).
- **Comunicação:** O padrão MPI utiliza a ideia de comunicação orientada à mensagens. O padrão MPI foi criado para resolver problemas que a comunicação via *sockets* não tratavam muito bem, como o seu uso em protocolos proprietários de alta velocidade. No padrão MPI, a comunicação se dá em um grupo de processos conhecidos chamado de comunicador (tal grupo é designado pela diretiva MPI_COMM_WORLD, que é o chamado comunicador universal (padrão) dos processos de uma aplicação MPI. Tal comunicador contém todos os processos MPI e é criado na inicialização do ambiente MPI por meio da diretiva MPI_INIT. Usando o MPI_COMM_WORLD, todos os processos envolvidos na aplicação MPI conhecem todos os processos, podendo qualquer processo se comunicar com um outro processo dentro do mesmo comunicador. Vale notar que no geral, é possível uma aplicação MPI possuir mais de um comunicador, onde um processo pode pertencer à mais de um comunicador, podendo ter diferentes *ranks* em diferentes comunicadores.

A troca de mensagens entre diferentes processos usa o identificador do grupo do processo e o rank do processo para identificar fonte ou desti-

natário de uma dada mensagem. Por exemplo, na chamada:

MPI.Recv (buf, count, datatype, src, tag, comm,status)

que é uma rotina de recebimento bloqueante do *MPI*, *buf* é a variável que contém o dado, *count* é a quantidade de dados que está sendo enviada na mensagem, *datatype* é o tipo de dado que está sendo enviado, *src* é o rank do processo que está enviando (a fonte) a mensagem, *tag* é o índice da mensagem, *comm* é o comunicador ao qual o processo pertence e *status* é uma variável do tipo *MPI_Status* que vai armazenar o status do sistema após tal operação.

Existem quatro modelos de comunicação fornecidos pelo MPI: padrão, síncrona, *bufferizada* e assíncrona. Tais modos diferem no modo que a mensagem é enviada. O envio padrão é dado pelo uso da rotina *MPI_Send*, que envia de uma forma bloqueante, bloqueando o processo até que a mensagem seja enviada para o destino. O envio síncrono é dado pela rotina *MPI_SSend*, que possui apenas uma diferença frente ao *MPI_Send*: enquanto *MPI_send* não aguarda o receptor receber a mensagem para desbloquear, *MPI_SSend* aguarda a mensagem chegar no destinatário para desbloquear sua rotina. Já o envio *bufferizado* copia a mensagem para um buffer de sistema para transmissão posterior se necessário. Ao usar esse tipo de chamada, o programador deve especificar o tamanho do buffer à priori usando a diretiva *MPI_BUFFER_ATTACH(buffer,size)*, onde *buffer* é o bufer e *size* é o tamanho do buffer e o envio assíncrono consiste em enviar a mensagem imediatamente, jogando-a no comunicador e (literalmente) torcer para que o processo receptor receba a mensagem, onde tal processo pode ou não receber a mensagem.

É utilizado o envio padrão (bloqueante) na nossa aplicação por ser o mais simples e necessário para a aplicação.

- Nomeação: A nível lógico, o padrão MPI usa o conceito de *ranks* para identificar seus processos. Na implementação OpenMPI, os processos possuem *ranks* que são enumerados de zero à N-1 (N sendo o número total de processos da aplicação MPI) e os *ranks* são os identificadores de um processo dentro de um dado comunicador. Por padrão, costuma-se usar (em abordagens do tipo coordenador-trabalhador) o rank 0 como sendo o coordenador e os demais ranks são os processos operários.
- Sincronização O MPI possui uma rotina específica chamada *MPI_Barrier(MPI_Comm communicator)* que sincroniza todos os processos que fazem parte do comunicador *communicator*, criando uma espécie de barreira onde a execução dos processos só continua quando todos eles atingirem o mesmo ponto. Tal sincronização é transparente ao programador que usa o padrão MPI.
- Tolerância a falhas MPI não tolera falhas: quedas de processos e/ou de mensagens trocadas entre processos ocasionam a interrupção do funcionamento da aplicação MPI como um todo.

4.2 Descrição da implementação

A tarefa mestre então inicia sua execução: primeiro ela efetua a leitura do arquivo (utilizando a função *ler*, disponível em *io.c*), obtendo a cadeia de DNA inserida como entrada e, em seguida, dividindo a mesma em pedaços menores de sub-cadeias de DNA (por exemplo, para uma cadeia de 60 nucleotídeos, como cada códon possui tamanho 3, nessa etapa a tarefa mestre dividiria a cadeia em 20 sub-cadeias de tamanho 3 cada, utilizando a rotina *split*, encontrada em *transcription.c*) e em seguida, envia as subcadeias para as demais tarefas (mantendo a primeira porção para si) por meio da chamada da função *MPI_Send* (primeiramente enviando a quantidade de códons que a tarefa irá receber, em seguida enviando os códons), que faz o envio bloqueante síncrono de mensagens (isso significa que a tarefa que está enviando retorna a execução assim que a mensagem foi enviada, o que não implica que a mesma foi recebida pela tarefa receptora).

Após o envio dos dados necessários (quantidade de códons e os códons em si), a tarefa mestre começa a efetuar o processamento na parte dos códons que com ela ficou: primeiro, ela divide a sub-cadeia de DNA que restou em outras sub-cadeias de tamanho 3 (o tamanho de um códon) e, para cada uma dessas sub-cadeias, faz a transcrição de DNA para RNA, utilizando a função *transcription* (encontrada em *transcription.c*) e em seguida identificando o aminoácido referente ao códon transcrito, fazendo uso da rotina *transcription* (que pode ser encontrada em *transcription.c*).

Nesse momento, as demais tarefas escravo também estão fazendo o mesmo: após o recebimento das mensagens que foram enviadas pela tarefa mestre (recebimento feito por meio da rotina *MPI_Recv*), cada uma das tarefas escravo divide a cadeia de DNA recebida em sub-cadeias de tamanho 3 (da mesma forma que a mestre), e, para cada sub-cadeia de DNA gerada a partir da cadeia que a tarefa recebeu, ela faz a transcrição de DNA e a tradução do códon para o aminoácido que ele identifica (da mesma forma que a tarefa mestre). Ao término do processo, elas enviam o resultado do processamento (que seria todos os códons transcritos e cada um dos seus respectivos aminoácidos) para a tarefa mestre

A mestre então, para cada tarefa escravo, ela recebe os dados da tarefa escravo (por meio da rotina *MPI_Recv*) e, para cada conjunto de dados recebidos (de cada tarefa), ela vai armazenando as cadeias de RNA transcritos e os seus respectivos aminoácidos. Ao fim da recepção (todas as tarefas escravo concluíram o envio, a tarefa mestre então imprime na tela o resultado do processamento: para cada códon, imprime na tela o códon de DNA original da entrada, a cadeia de RNA transcrita referente à tal códon e o aminoácido que tal cadeia de RNA identifica. Em seguida, escreve o mesmo que ela imprimiu em um arquivo de saída. Ao fim da execução do ambiente MPI, é pego o tempo de finalização (a fim de ver quanto tempo o processamento levou) e o ambiente é finalizado pelo processo *main* (o mesmo onde o ambiente é inicializado) por meio da rotina

MPI_Finalize.

Para executar a aplicação paralela (em uma única máquina), usa-se:

- Compilar: ‘mpicc main-mpi.c transcription.c io.c -o dna-mpi’
- Executar (x processos): ‘mpirun -np x dna-mpi’

5 Descrição dos experimentos computacionais

A ideia inicial seria efetuar a medição de desempenho em diversas máquinas. Porém, como não foi possível, a máquina física onde o código foi executado é um computador com 8 GB de memória RAM DDR3, processador Intel i7-5500U (8 núcleos) @ 2.40 GHz, com o S.O Ubuntu 16.04 LTS 64 bit. Foram executados 2 casos de teste: uma entrada com a cadeia de DNA com 131 bases nitrogenadas, com um códon de tamanho 122 e uma entrada com um códon contendo 2521 bases nitrogenadas (observe que a quantidade de cótons na entrada pode diferir do tamanho da cadeia de DNA de entrada original).

Nº processos	1ª execução	2ª execução	3ª execução	Média
1 processo	0.001601s	0.001495s	0,000170s	0.001657s

Tabela 1: Tabela da execução com entrada de um códon com 60 nucleotídeos para a aplicação sequencial

Nº processos	1ª execução	2ª execução	3ª execução	Média
1 processo	0.000384s	0.000669s	0.000779s	0.000611s
2 processos	0.000430s	0.000581s	0.000417s	0.000476s
3 processos	0.000592s	0.000389s	0.000520s	0.000500
4 processos	0.000878s	0.000730s	0.000920s	0.000842s
5 processos	0.000592s	0.000389s	0.000520s	0.000600s
10 processos	0,004734s	0,004441s	0,015296s	0,008157s

Tabela 2: Tabela da execução com entrada de um códon com 122 bases nitrogenadas para a aplicação paralela

Para a primeira bateria de testes, temos que o resultado da aplicação sequencial foi bem satisfatório: em todos os casos onde o número de tarefas é menor do número de núcleos, o desempenho da aplicação usando MPI foi melhor do que a sequencial, pois cada processo roda em um diferente núcleo. Vale notar que quando o número de processos ultrapassa o número de núcleos da máquina, o tempo da aplicação paralela é pior do que todos os demais testes dela e também é pior que o tempo da aplicação sequencial. Uma possível razão é o enfileiramento de mensagens entre os processos (onde para os processos serem executados eles passam a disputar os núcleos da CPU por não ter o suficiente para todos executarem em paralelo simultaneamente, nesse caso).

Nº processos	1ª execução	2ª execução	3ª execução	Média
1 processo	0.021487s	0.026066s	0.028954s	0.025502s

Tabela 3: Tabela da execução com entrada de um cístron com 2521 bases nitrogenadas para a aplicação sequencial

Nº processos	1ª execução	2ª execução	3ª execução	Média
1 processo	0.006514s	0.005006s	0.006639s	0.005891s
2 processos	0.006936s	0.005423s	0.005735s	0.006031s
3 processos	0.011321s	0.013001s	0.011141s	0.011821s
4 processos	0.010910s	0.010860s	0.014518s	0.012096s
6 processos	0.035180s	0.034607s	0.035032s	0.034940s
15 processos	0.041788s	0.041012s	0.038704s	0.040501s
41 procesos	0.013867s	0.014117s	0.021192s	0.016392s

Tabela 4: Tabela da execução com entrada de um cístron com 2521 bases nitrogenadas para a aplicação paralela

Para a segunda bateria de testes, o resultado se mantém: para todos os casos de teste onde a aplicação MPI é executada com menos processos que o número de núcleos de processamento da máquina usada, o desempenho da aplicação paralela usando MPI é melhor do que o desempenho da aplicação sequencial. Surpreendentemente, o resultado da aplicação ao executar com 41 processos (número primo inserido de propósito para fins de teste) foi melhor do que o executado com 6 e com 15 processos, respectivamente. Uma possível explicação pode ser que algum recurso que facilitasse a execução desta aplicação poderia estar sendo usado pelo SO da máquina (ou por uma outra aplicação em execução no plano de fundo) e ao executar a aplicação com 41 processos, já não estava mais.

6 Conclusões

Baseados nos dados acima apresentados, somos levados à concluir que o uso do paradigma de programação baseada em troca de mensagens é bem útil para aplicações que necessitam de processamento paralelo para ter um melhor desempenho. O OpenMPI oferece uma implementação robusta do MPI, fornecendo um ambiente de programação usando troca de mensagens de forma eficiente e transparente ao programador que invoca o MPI.

Referências

- Chibli, E. A. (2008), *A Multiprocessor Parallel Approach to Bit-parallel Approximate String Matching*.
- Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A. et al. (2004), Open mpi: Goals, concept, and design of a next generation mpi implementa-

- tion, *in* ‘European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting’, Springer, pp. 97–104.
- Kleijnung, J., Douglas, N. & Heringa, J. (2002), ‘Parallelized multiple alignment’, *Bioinformatics* **18**(9), 1270–1271.
- Morandini, C., Bellinello, L. C. & Constantino, C. (2013), *Coleção Objetivo - Apostila Resumo Biologia 1*.
- Venter, J. C., Adams, M. D., Myers, E. W., Li, P. W., Mural, R. J., Sutton, G. G., Smith, H. O., Yandell, M., Evans, C. A., Holt, R. A. et al. (2001), ‘The sequence of the human genome’, *science* **291**(5507), 1304–1351.
- Xue, Q., Xie, J., Shu, J., Zhang, H., Dai, D., Wu, X. & Zhang, W. (2014), A parallel algorithm for dna sequences alignment based on mpi, *in* ‘Information Science, Electronics and Electrical Engineering (ISEEE), 2014 International Conference on’, Vol. 2, IEEE, pp. 786–789.