Uma abordagem MPI para transcrição e tradução de DNA

Erick Grilo, Max Fratane, Elihofni Lima

10 de Junho de 2017

1 Introdução

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 cadeias, 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 (citosina) 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 códons. Um códon codifica um aminoácido.

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.Em Chibli (2008), por exemplo, uma abordagem usando MPI para solucionar problemas de comparação de *strings*, incluindo a transcrição de DNA para RNA e a identificação de aminoáciods, 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) . Logo, o objetivo do programa 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, a partir de códons de RNA (que foram obtidos a partir da trascrição de códons de DNA para RNA), identificar o aminoácido que o códon em questão representa. Como estamos considerando nenhuma mutação genética, a cadeia de entrada deve ser múltipla de 3 para que o algoritmo (tanto paralelo quanto sequencial) funcionem corretamente.

2 O método 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: a cada três nucleotídeos (um códon), sua transcrição (conversão de DNA para RNA) é feita e, em seguida, a identificação do aminoácido que aquele códon se refere. Caso o códon não possui 3 nucleotídeos, tal codon é ignorado. Ao término do processamento, a cadeia de RNA é escrita no arquivo final, junto com cada códon transcrito e o respectivo aminoácido que ele representa.

3 O método paralelo

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 divido 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.

4 A implementação paralela

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 fomra prática e eficiente.

A nossa implementação consiste no uso do ambiente MPI para paralelizar o processo: Ao iniciar o ambiente MPI, o ambiente de execução paralela é inicalizado. Em seguida, as tarefas são criadas e inicializadas no comunicador padrão (MPI_COMM_WORLD) e, então, existem duas rotinas possíveis para serm 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.

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 trascription.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-cadeis 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.

5 Código fonte da aplicação sequencial

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "io.h"
#include "transcription.h"
#define TAMANHO_CODON 3
```

```
char** sequencial(char* str, int size, char* fileName){
    char** finalArray = split(str, 3);

    int i;
        /*if(size%3){
                printf("Quantidade de bases nitrogenadas do DNA Ã¡invlidas. O Ã
                exit(1);
        }*/
    for(i = 0; i < (size/3); i++){
        char* c = finalArray[i];
        char* a = transcription(c, 3);
        char* b = aminoacids(a, 3);

        char str[11];
                strcpy(str, a);
                strcat(str, " ");
                strcat(str, b);
                strcat(str, " ");
                strcat(str, c);
                strcat(str, "\n");

        escreverAppend(str, fileName);
    }

    return finalArray;
}

int main(){
        char *in, *chain;
        printf("Insira o nome do arquivo de entrada: \n");
        scanf("%s", in);
        chain = ler(in);
        int length = strlen(chain);
        printf("%d \n",length);
        printf("%s \n",chain);
        sequencial(chain,length,"saida.txt");

        return 0;
}
```

5.1 *Helpers* que também são utilizados no código da aplicação paralela

5.1.1 Entrada e saída

```
#include "io.h"

static int tamanho(FILE* f){
        fseek (f, 0, SEEK_END);
        int length = ftell (f);
        fseek (f, 0, SEEK_SET);

        return length;
}

char* ler(char* caminho){
        FILE* f = fopen(caminho, "r");
        if(!f)exit(1);

        char* string = calloc(tamanho(f),sizeof(char));

        fscanf(f, "%s",         string);

        fclose(f);

        return string;
}

void escrever(char* str, char* caminho){
        FILE* f = fopen(caminho, "w");
        if(!f)exit(1);
        fprintf(f, "%s", str);

        fclose(f);
}

void escreverAppend(char* str, char* caminho){
        FILE* f = fopen(caminho, "a");
        if(!f)exit(1);
        fprintf(f, "%s", str);

        fclose(f);
}
```

5.1.2 Funções responsáveis pela transcrição do DNA para RNA e a identificação do aminoácido correspondente

```
#include "transcription.h"

void initialize(char *array,int size){
    int i;
    for(i=0;i<size;i++)array[i] = '\0';
}

/**
* BIO: CODONS
* funcao que quebra uma string em substrings
* de tamanho 3.
* str - String recebida pela thread.
* Exemplo: AACCGCTCA -> AAC CGC UCA
*/
char** split(char* str, int subStringSize){
    int size = strlen(str);
    int finalArraySize = size / subStringSize;

    char temp[subStringSize + 1];
    temp[subStringSize] = '\0';


    /*
     * Array que armazena as substrigns.
     */
    char** finalArray = calloc(finalArraySize, sizeof(char*));
    int i = 0;
    for (i = 0; i < finalArraySize; i++) {
        finalArray[i] = calloc((subStringSize + 1), sizeof(char));
    }

    /*
     * Iterador do array final.
     */
    int j = 0;

    /*
     * Iterador da string tempor?ria.
     */
    i = 0;
    /*
     * Iterador do string de entrada.
     */
    int k = 0;

    for(i = 0; k < size; i++, k++){
        temp[i] = str[k];
```

```
        if(i == (subStringSize - 1)){
            //Reseto o iterador da string tempor?ria.
            i = -1;

            //atribuo a string de tamanho 3 para o vetor final.
            int l = 0;
            for(l = 0; l < subStringSize; l++){
                finalArray[j][l] = temp[l];
            }

            //Incremento o index atual do vetor final.
            j++;
        }
    }

    return finalArray;
}

/**
* BIO: ÇÂTRANSRIO [DNA -> RNA]
* Traduz uma sequencia em DNA para RNA.
* str - String recebida pela thread..
* Exemplo: AACCGCTCA -> AACCGCUCA
*/
char* transcription(char *chain, int size){
    int i;
    char *rna = calloc((size+1), sizeof(char)),k;
    //initialize(rna,size);
    for(i=0;i<size;i++){
        k = chain[i];
        if(k == 'A') rna[i] = 'U';
        else if(k == 'T') rna[i] = 'A';
        else if(k == 'C') rna[i] = 'G';
        else if(k == 'G') rna[i] = 'C';
    }
    return rna;
}
/**
* AMINOACIDOS
* funcao que recebe uma string de tamanho 3
* e traduz para um nome.
* Exemplo: AAC CGC UCA -> Nome1 Nome2, Nome3
*/
char* aminoacids(char *in, int size){
    //ãno gosto de switch e case, ád no mesmo:
    if(in[0] == 'U' && in[1] == 'U'){
        if(in[2] == 'U' || in[2] == 'C') return "Phe";
        else return  "Leu";
```

```
    }
    else if(in[0] == 'C' && in[1] == 'U') return  "Leu";
    else if(in[0] == 'A' && in[1] == 'U'){
        if(in[2] == 'U' || in[2] == 'C') return  "Leu";
        else return  "Met";
    }
    else if(in[0] == 'G' && in[1] == 'U') return  "Val";
        //fim da primeira coluna
    else if(in[0] == 'U' && in[1] == 'C') return  "Ser";
    else if(in[0] == 'C' && in[1] == 'C') return  "Pro";
    else if(in[0] == 'A' && in[1] == 'C') return  "Thr";
    else if(in[0] == 'G' && in[1] == 'C') return  "Ala";
        //fim da segunda coluna
    else if(in[0] == 'U' && in[1] == 'A'){
        if(in[2] == 'U' || in[2] == 'C') return  "Tyr";
            //S\N: toda vez que isso aparecer significa que na tabela estava um
        else return  "S/N";
    }
    else if(in[0] == 'C' && in[1] == 'A'){
        if(in[2] == 'U' || in[2] == 'C') return  "His";
        else return  "GluN";
    }
    else if(in[0] == 'A' && in[1] == 'A'){
        if(in[2] == 'U' || in[2] == 'C') return  "AspN";
        else return  "Lys";
    }
    else if(in[0] == 'G' && in[1] == 'A'){
        if(in[2] == 'U' || in[2] == 'C') return  "Asp";
        else return  "Glu";
    }
        //fim da terceira coluna
    else if(in[0] == 'U' && in[1] == 'G'){
        if(in[2] == 'U' || in[2] == 'C') return  "Cys";
        if(in[2] == 'G') return "Tryp";
        else return "S/N";
    }
    else if(in[0] == 'C' && in[1] == 'G') return "Arg";
    else if(in[0] == 'A' && in[1] == 'G'){
        if(in[2] == 'U' || in[2] == 'C') return "Ser";
        else return "Arg";
    }
    else if(in[0] == 'G' && in[1] == 'G') return "Gly";
    //se ãno entrou em nenhum caso ávlido, o nome de retorno é
sem sentido:
    return "S/N";
}

char** translate(char* str, int size){
    char* transc = transcription(str, size);
```

```
    char** finalArray = split(transc, 3);

    int i;
    for(i = 0; i < (size/3); i++){
        char* c = finalArray[i];
        char* aminoacid = aminoacids(c, 3);

        int l = 0;
        for(l = 0; l < 3; l++){
            finalArray[i][l] = aminoacid[l];
        }
    }

    return finalArray;
}

/**
 * Procura pelo íincio de um íCstron (A partir desse ponto pode-se çcomear a çãtr
 * @param cadeiaDNA Cadeia Original do arquivo
 * @return Cadeia de DNA pronta para çãtranscrio
 */
char* getCistron(char* cadeiaDNA) {
    int tamanho = strlen(cadeiaDNA);
    int i;
    for(i=0;i<tamanho;i++){
        if(cadeiaDNA[i] == 'A') {
            if((cadeiaDNA[i+1] == 'T' && (cadeiaDNA[i+2] == 'T' || cadeiaDNA[i+
                printf(COR_VERDE "\nProcesso Mestre: CISTRON INICIA NO INDICE %
                char* cadeiaCistron = malloc((tamanho - (i+3)) * sizeof(char));
                strncpy(cadeiaCistron, cadeiaDNA + i+3, (tamanho - (i+3)));
                return cadeiaCistron;
            }
        }
    }
    return NULL;
}
```

6 Código fonte da aplicação paralela

7 Descrição dos experimentos computacionais

deu ruim aqui br

Referências

Chibli, E. A. (2008), *A Multiprocessor Parallel Approach to Bit-parallel Approximate String Matching*.

Kleinjung, J., Douglas, N. & Heringa, J. (2002), 'Parallelized multiple alignment', *Bioinformatics* **18**(9), 1270–1271.

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.