



SIMILITUD DE CADENAS DE ADN MEDIANTE EL ALGORITMO DE ALINEAMIENTO NW

Grupo 21



Ignacio Gomis Lli
Lidia Montero Egidos
Sara Monzó Bravo
Paul Vargas Hurtado

UNIVERSITAT DE VALÈNCIA – ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA

Contenido

CP1: Análisis del algoritmo.....	2
1-Objetivo y utilidad del algoritmo	2
2-Descripción del algoritmo.....	3
3-Análisis de coste y almacenamiento en memoria	6
Coste computacional del algoritmo:.....	6
Almacenamiento en memoria:	6
4-Análisis de dependencia de datos.....	8
5-Propuestas de paralelización.....	9
Anexo: Código	11
Anexo: Tiempo dedicado.....	18
Anexo: Autoevaluación	19
Práctica 0.....	20
1-Tiempos de ejecución	20
2-Representación gráfica de los tiempos	21
3-Modificaciones en el código	22
4-Tiempo de trabajo	24
Anexo: Autoevaluación	25
CP2: Propuesta de implementación con OpenMP	26
1-Descripción del algoritmo paralelo	26
2-Implementación del algoritmo	29
3-Análisis del algoritmo.....	41
Coste computacional	41
Coste de memoria.....	41
Estimaciones	41
Anexo: Tiempo trabajado	43
Anexo: Autoevaluación	44
Anexo: Referencias	45

CP1: Análisis del algoritmo

1-Objetivo y utilidad del algoritmo

El objetivo del algoritmo Needleman-Wunsch, publicado en 1970, es obtener el mejor alineamiento de dos cadenas de proteínas o nucleótidos, (Needleman, S. and Wunsch, C. (1970)). Nosotros a partir de este algoritmo, obtendremos el porcentaje de similitud de este alineamiento.

Un ejemplo de esto sería si comparásemos AAAGT con AAGAG nos devolvería que la mejor similitud serían AA-AGT con AAGAG- con 4 coincidencias.

La principal utilidad de este algoritmo está en el campo de la bioinformática, para la comparación de muestras, (Wikipedia, 2018). Entre otras posibilidades, puede servir para ver la cercanía del ADN del ser humano a otros seres o comparar las mutaciones de ADN entre diversos individuos de una especie,

Entrada del algoritmo: Dos cadenas de texto en formato fasta (.fa o .fasta), selección de valores numéricos para coincidencias, fallos y huecos. Nosotros en el algoritmo fijaremos estos valores como (2,-2,-1)

Salida del algoritmo: En nuestro proyecto mostraremos el porcentaje de coincidencia entre cadenas, pero el algoritmo normalmente devuelve las dos cadenas con su mejor alineamiento.

2-Descripción del algoritmo

El algoritmo se basa en programación dinámica, al resolver el problema principal como un conjunto de subproblemas y luego escoger la mejor combinación como el resultado del problema principal, (Wikipedia, 2018).

Por tanto, inicialmente se construye una tabla como la siguiente:

		G	T	A	C	A	T	A
	0	-1	-2	-3	-4	-5	-6	-7
G	-1							
A	-2							
T	-3							
T	-4							
A	-5							
C	-6							
A	-7							

En la cual los valores de la primera fila y segunda fila son la suma del anterior con la penalización por hueco (Que establecimos como -1) y en la posición 0,0 asignamos un 0.

$$M[0,0]=0$$

$$M[i,0]=M[i-1,0]+\text{Hueco}$$

$$M[0,j]=M[0,j-1]+\text{Hueco}$$

Luego, se completa la matriz, para cada celda se realizan las siguientes operaciones:

$$M[i,j]=\text{Max}(M[i-1,j]+\text{Hueco}, M[i,j-1]+\text{Hueco}, M[i-1,j-1]+\text{Función coincidencia})$$

Función coincidencia= Si(Letra1=Letra2) Entonces (Valor coincidencia) Sino (Valor Fallo)

Nosotros hemos dado los valores siguientes: Valor coincidencia=2, valor fallo = -2

De una manera un poco más formal:

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + \delta(v_i, -) \\ S_{i,j-1} + \delta(-, w_j) \\ S_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

$$\delta(\alpha, \beta) = \begin{cases} 2 & \text{si } \alpha = \beta \\ -2 & \text{si } \alpha \neq \beta \text{ y } \alpha \neq '-' \text{ y } \beta \neq '-' \\ -1 & \text{si } \alpha = '-' \text{ ó } \beta = '-' \end{cases}$$

Además, se debe anotar en cada celda como se obtiene ese máximo, es decir, si viene de una diagonal, de lateral o arriba, y no solo una, sino que si hay empates se anotan varios. Estas direcciones sirven después para reconstruir la mejor cadena.

Tras hacer el cálculo tenemos la matriz así:

		G	T	A	C	A	T	A
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	2	1	0	-1	-2	-3	-4
A	-2	1	0	3	2	1	0	-1
T	-3	0	3	2	1	0	3	2
T	-4	-1	2	1	0	-1	2	1
A	-5	-2	1	4	3	2	1	4
C	-6	-3	0	3	6	5	4	3
A	-7	-4	-1	2	5	8	7	6

Finalmente, a partir de la última casilla, se obtienen todos los caminos generados al seguir las direcciones anteriormente anotadas. Entonces recorriendo, cada vez que se encuentra una diagonal, es una coincidencia y si encuentra una horizontal o una vertical, es un hueco (si es una lateral el hueco va en la segunda cadena, sino el hueco se coloca en la primera). Hay que señalar, que lo realmente importante este camino pase por el mayor valor de la matriz, (fuente requerida), sin embargo, al empezar desde la última celda, como esta depende de todas las celdas anteriores, tendrá un camino hasta la misma, y nos ahorra el tener que llevar un control de cuál es la casilla de valor máximo

Para nuestro ejemplo solo se genera un camino:

		G	T	A	C	A	T	A
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	2	1	0	-1	-2	-3	-4
A	-2	1	0	3	2	1	0	-1
T	-3	0	3	2	1	0	3	2
T	-4	-1	2	1	0	-1	2	1
A	-5	-2	1	4	3	2	1	4
C	-6	-3	0	3	6	5	4	3
A	-7	-4	-1	2	5	8	7	6

Por tanto, el alineamiento, que tiene 5 coincidencias de 7, es:

G-T-ACATA

GATTACA—

Al emplear el algoritmo utilizaremos cadenas reales de ADN obtenidas del centro nacional de información en biotecnología (National Center for Biotechnology Information,(n.d.).)

3-Análisis de coste y almacenamiento en memoria

Coste computacional del algoritmo:

Desglosado por funciones del programa:

Función	Lecturas Matriz	Escrituras Matriz	Lecturas Cadenas	Escrituras Cadenas	Operaciones
CargarFichero	0	0	$2*N$	$2*N$	0
InicializarMatriz	0	$4*(N+M+1)$	0	0	0
CalcularCasilla	3	2	4	0	19
CompletarMatriz	$3NM$	$2NM$	$4NM$	0	$19NM$
AuxGetRuta (caso mejor)	0	0	0	0	5
AuxGetRuta (caso esperado)	3	0	0	0	13
AuxGetRuta (caso peor)	3	0	0	0	15
GetRuta (Caso óptimo)	$3*\min(N,M)$	0	0	0	$13*\min(N,M)$
GetRuta (Caso peor, sin poda)	Problema NP-Completo				
Orden Esperado	$N*M$	$N*M$	$N*M$	N	$N*M$

Queremos señalar que tenemos preparadas dos versiones de CalcularCasilla, para probar cual tiene mejor rendimiento en el ordenador de laboratorio. La principal radica en que en una de las versiones no empleamos ningún salto condicional, lo cual nos genera algunas operaciones y lecturas adicionales, pero elimina las paradas de procesador por fallos de predicción. Por otro lado, al observar que la cantidad de operaciones y lecturas podía reducirse con dos operaciones de máximo entre dos valores, comprobaremos qué penaliza menos.

En segundo lugar, el problema de obtener la ruta hemos observado que se trata de un problema NP-Completo, cuyo coste escala de manera exponencial con la talla del problema. Los problemas de obtención de mejores rutas suelen ser NP-Completo (Por ejemplo, el problema del viajante) pero le hemos añadido una función de poda que limita la exploración a las cercanías de la diagonal, por lo que los valores esperados son cercanos al caso óptimo.

Almacenamiento en memoria:

Si consideramos despreciables los costes de memoria de variables auxiliares, Strings de entrada y contadores de bucles y nos limitamos a la matriz de datos, el consumo en memoria es el siguiente:

Dados dos String de tamaño N y M, la matriz tiene un tamaño de $N+1 \times M+1$, y estará compuesta de una estructura que almacena tres booleanos y un valor numérico. Este

valor lo establecemos actualmente como un Int, pero en un futuro, si se diera el caso de que superásemos el máximo valor para el entero, podría pasar a un Double.

Coste de celda con Int: 8 Bytes

Coste de celda con Double: 16 Bytes

(Valores obtenidos con la función Sizeof (Struct Celda))

Por tanto, el tamaño de la matriz es de:

$$8 \cdot (N+1) \cdot (M+1) = (8N+8) \cdot (M+1) = 8NM + 8N + 8M + 8 \text{ Bytes}$$

Por ejemplo, si ponemos $N=M=10^7$ (máximo valor puesto en el enunciado) nuestro coste será: $800.000.160.000.008 \text{ Bytes} = 800\text{TB}$

Este valor es ridículamente grande, por tanto tenemos la siguiente formula, que nos calcula el tamaño máximo de N y M para que la matriz sea computable, según su memoria:

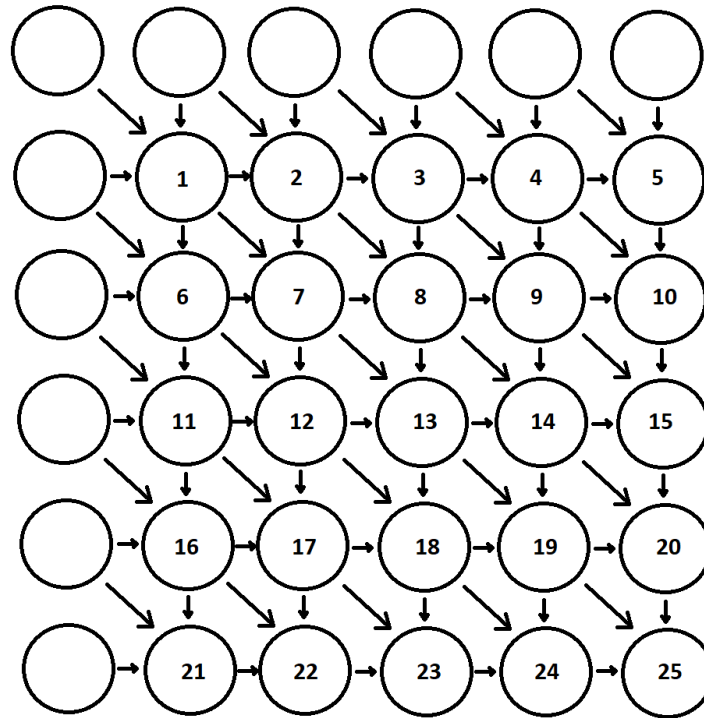
$$N=M < \text{SQRT}((\text{Memoria-Margen})/8)$$

Siendo Margen, de tamaño suficiente para incluir dos String de Tamaño N y M y a lo sumo 1KB para variables temporales.

Al poder configurar la entrada de parámetros, se podría estudiar una cadena más larga, con varias ejecuciones distintas sobre diversas partes de la cadena, al tratar el problema total como un subconjunto de cadenas se puede obtener una aproximación al resultado real.

4-Análisis de dependencia de datos

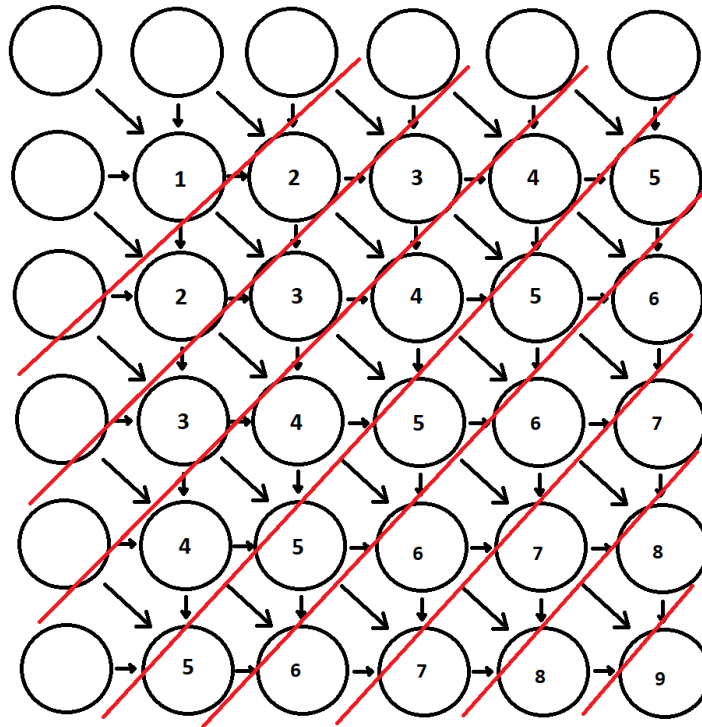
Como se puede observar, la dependencia de datos de una celda con índices $[i,j]$ son las celdas $[i-1,j]$, $[i,j-1]$ y $[i-1,j-1]$, así como el índice $[i-1]$ de una de las cadenas y el índice $[j-1]$ de la otra cadena, sin embargo, las cadenas solo resultarán leídas, mientras que las celdas se escriben iterativamente, y solo una vez cada una de ellas, causando que una vez terminada una sección esta pueda ser leída sin riesgo.



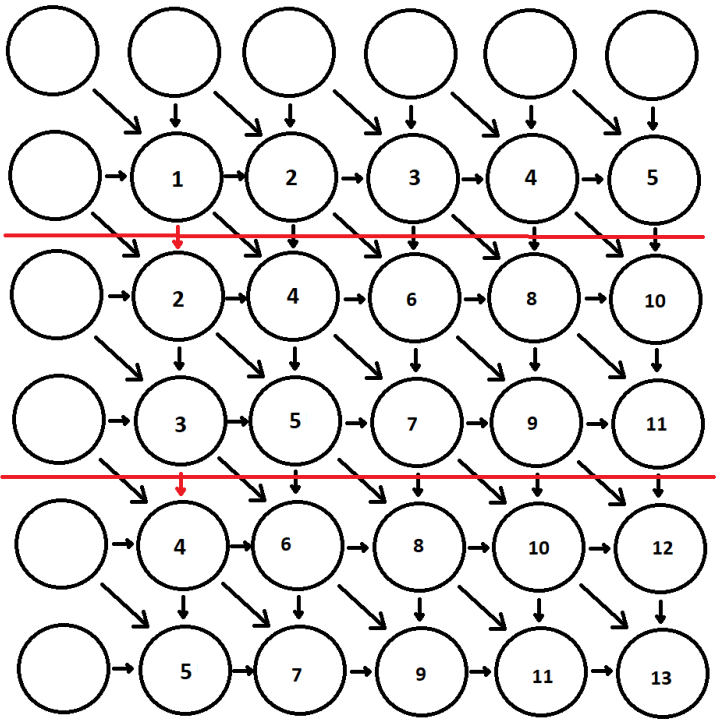
5-Propuestas de paralelización

Esencialmente, al tratarse de una matriz, la primera estrategia de paralelización que podemos pensar es por paralelismo de datos, pero al tener que generarla por partes podemos considerar que también la trataremos una parte de paralelismo por flujo de datos, siendo un.

En primer lugar, la paralelización ideal seguiría un esquema como el siguiente:



Sin embargo, la alternativa que proponemos, para tener fragmentada la matriz en tantas particiones como procesadores disponibles, sería usando el siguiente esquema (O la trasposición de esta misma matriz) de tal manera que cada procesador se encargue de una cantidad igual de celdas, a excepción del primero que calculará las que queden sin emparejar en grupos del tamaño de partición. Esta carga no se le puede asignar al último porque entonces podría adelantar al anterior, al tener una menor cantidad de celdas. El siguiente hilo sería lanzado al finalizar la primera columna de la sección.



Anexo: Código

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>

/*Struct con el valor de la celda y las direcciones
 Si las direcciones son verdadero, entonces ha heredado de esa
 direccion
*/
struct Celda
{
    int score;
    short dir; //Array de booleanos
};

char* CargarFichero(char*,unsigned,unsigned);
struct Celda** inicializarMatriz(unsigned, unsigned);
void CompletarMatriz(char*,char*,struct Celda**);
void CalcularCasilla(unsigned, unsigned, bool, struct Celda**);
unsigned GetRuta(struct Celda**,unsigned,unsigned);
int AuxGetRuta(struct Celda**, unsigned, unsigned, int, unsigned*);
/*Maximo entre dos valores*/
unsigned maxU(unsigned arg1, unsigned arg2)
{
    if(arg1>arg2){
        return arg1;
    }
    else
        return arg2;
}
int maxI(int arg1, int arg2)
{
    if(arg1>arg2){
        return arg1;
    }
    else
        return arg2;
}
/*Funcion para pasar a mayuscula una cadena. Obtenida de:
https://stackoverflow.com/questions/35181913/convert-char-to-
uppercase-in-c
*/
void Mayus(char * temp) {
    char * name;
    name = strtok(temp,":");
    char *s = name;
    while (*s) {
        *s = toupper((unsigned char) *s);
        s++;
    }
}

/**
 * Main Lleva control del orden de ejecucion de las funciones del
 algoritmo y la entrada de parametros
 * @author Lidia y Nacho
 * @date 8/2/2018
 */
```

```

* @param Nombre o ruta de Fichero1 (Necesario)
* @param Nombre o ruta de Fichero2 (Necesario)
* @param Tamano de cadena de fichero1 (Opcional o ambos ficheros si
falta siguiente parametro)
* @param Tamano de cadena de Fichero2 (Opcional)
* @param Inicio de cadena de Fichero1 (Opcional o ambos ficheros si
falta siguiente parametro)
* @param Inicio de cadena de Fichero2 (Opcional)
*/
int main( int argc, char *argv[] )
{
    unsigned T1,T2,I1,I2;
    switch(argc)
    {
        case 3: //Ambos desde el principio hasta lo que coja
            T1=100;T2=100;I1=0;I2=0;

            break;
        case 4: //Ambos con Tamaño arg[3]
            T1=atoi(argv[3]);T2=T1;I1=0;I2=0;
            break;
        case 5: //Cada uno con Tamaño arg[3] y arg[4]
            T1=atoi(argv[3]);T2=atoi(argv[4]);I1=0;I2=0;
            break;
        case 6: //Cada uno con Tamaño arg[3] y arg[4] empezando desde
arg[5]
            T1=atoi(argv[3]);T2=atoi(argv[4]);I1=atoi(argv[5]);I2=I1;
            break;
        case 7: //Cada uno con Tamaño arg[3] y arg[4] empezando desde
arg[5] y arg[6]
            T1=atoi(argv[3]);T2=atoi(argv[4]);I1=atoi(argv[5]);I2=atoi(argv[6]);
            break;
        default: //Instrucciones de uso
            printf("Error en introduccion de datos:\n Se pueden introducir
entre 2 y 6 argumentos:\n2 argumentos:\n Fichero_1 Fichero_2\n");
            printf("3 argumentos:\n Fichero_1 Fichero_2
Tamano_maximo\n");
            printf("4 argumentos:\n Fichero_1 Fichero_2
TamanoMax_Cadenal TamanoMax_Cadena2\n");
            printf("5 argumentos:\n Fichero_1 Fichero_2
TamanoMax_Cadenal TamanoMax_Cadena2 Inicio_Cadenas\n");
            printf("6 argumentos:\n Fichero_1 Fichero_2
TamanoMax_Cadenal TamanoMax_Cadena2 Inicio_C1 Inicio_C2\n");
            printf("Tamano e inicio escalados: 1:100\n");
    }
    if(argc >= 3 && argc <=7)
    {
        printf("%s comparado con %s\n",argv[1],argv[2]);
        printf("Tamanos: %d %d\n",T1*100,T2*100);
        printf("Puntos de inicio: %d %d\n",I1*100,I2*100);
        struct timespec t1,t2,t3,t4;
        double total;
        //t1 = time(0);
        clock_gettime(CLOCK_REALTIME, &t1);
        char* string1=CargarFichero(argv[1],T1,I1);
        char* string2=CargarFichero(argv[2],T2,I2);
        //
        printf("Fin construccion cadenas\n");
        struct Celda **Matriz;
        if(strlen(string1)==0 || strlen(string2)==0)

```

```

    {
        printf("Una cadena esta vacia");
        exit(2);
    }
    //
printf("Lectura correcta \n");
Matriz=inicializarMatriz(strlen(string1),strlen(string2));
//t2 = time(0);
clock_gettime(CLOCK_REALTIME, &t2);
//
printf("Matriz iniciada\n");
CompletarMatriz(string1,string2,Matriz);
//t3 = time(0);
clock_gettime(CLOCK_REALTIME, &t3);

//
printf("Matriz completa\n");
int resultado=
GetRuta(Matriz,strlen(string1),strlen(string2));
//
printf("Ruta calculada\n");
//t4 = time(0);

clock_gettime(CLOCK_REALTIME, &t4);

total = ( t2.tv_sec - t1.tv_sec )*1000
        + ( t2.tv_nsec - t1.tv_nsec ) / (float)(1000000);
printf("InicIALIZado:      %lf\n", total );

total = (t3.tv_sec - t2.tv_sec)*1000
        + (t3.tv_nsec - t2.tv_nsec) / (float)(1000000);
printf("Creacion de matriz: %lf\n", total );

total = (t4.tv_sec - t3.tv_sec)*1000
        + (t4.tv_nsec - t3.tv_nsec) / (float)(1000000);
printf("Backtracking:      %lf\n", total );

total = (t4.tv_sec - t1.tv_sec)*1000
        + (t4.tv_nsec - t1.tv_nsec) / (float)(1000000);
printf("Total:              %lf\n", total );
printf("Coincidencia(porc): %d\n",
100*resultado/maxU(strlen(string1),strlen(string2)));
}
else
{
    exit(3);
}
printf("Fin");
exit(0);
return 0;
}

/**
 * CargarFichero funcion que guarda en un string el contenido de un
 * fichero .fasta
 * @author Lidia y Nacho
 * @date 8/2/2018
 * @param NombreFichero nombre del fichero, incluida extension y ruta
 * relativa
 * @out string con el contenido en mayusculas

```

```

*/
char* CargarFichero(char* NombreFichero,unsigned tamano,unsigned
inicio)
{
    tamano*=100;
    inicio*=100;
    FILE *archivo;
    unsigned i;
    char caracteres[100];
    char *cadena=malloc(tamano);    //<--Origen error
    strcpy (cadena, "");
    archivo = fopen(NombreFichero,"r");
    if (archivo == NULL)
    {
        printf("%s no existe",NombreFichero);
        exit(1);
    }
    else
    {
        fgets(caracteres,100,archivo); //Primera linea
        for(i=0;i<inicio;i++)
            fgets(caracteres,100,archivo);
        i=tamano;
        while (feof(archivo) == 0 && strlen(cadena)<i) //Hasta fin de
archivo o memoria
        {
            fgets(caracteres,100,archivo);
            strcat(cadena, caracteres);
        }
        Mayus(cadena);
        return cadena;
    }
}

// inicializarMatriz funcion que crea la matriz de tamano r c e
inicializa la primera fila y columna con valores negativos
descendentes.
// @author Sara
// @date 12/2/2018
// @param unsigned r rows
// @param unsigned c cols
// @return arr matriz con los valores negativos
struct Celda** inicializarMatriz(unsigned r, unsigned c)
{
    unsigned i;
    struct Celda **arr =(struct Celda **)malloc(r*c* sizeof(struct
Celda));
    for (i = 0; i <= r; ++i)
        arr[i] = (struct Celda *)malloc(c * sizeof(struct Celda));
    //Casos base posicion:  r = 0, c = 0
    arr[0][0].score = 0;
    arr[0][0].dir = 0;

    for(i = 1 ; i<=r; i++)
    {
        arr[i][0].score = -i;
        arr[i][0].dir=0;
    }

    for(i = 1 ; i<=c; i++)

```

```

    {
        arr[0][i].score = -i;
        arr[0][i].dir=0;
    }
    return arr;
}

/**
 * CompletarMatriz funcion que calcula el algoritmo Needleman-Wunsch
 para una matriz
 * @author Nacho
 * @date 7/2/2018
 * @param string1 Cadena de texto 1
 * @param string2 Cadena de texto 2
 * @param matrix Matriz de Celdas, su tamano debe ser el de las
 cadenas de texto +1
 */
void CompletarMatriz(char* string1,char* string2,struct Celda**
matrix)
{
    unsigned i;
    unsigned j;
    unsigned size1=strlen(string1);
    unsigned size2=strlen(string2);
    for(i=1;i<=size1;++i)
        for(j=1;j<=size2;++j)
            //El argumento de calcular casilla es cierto si ambos
strings coinciden o uno de ellos es N
            //Recordar que el tamano de la matriz es 1 mayor que los
strings, y estos se alinean con el final.
            {
                CalcularCasilla(i, j, (string1[i-1]==string2[j-
1]||string1[i-1]=='N'||string2[j-1]=='N'), matrix);
            }
        return;
}

/**
 * CalcularCasilla funcion que calcula el contenido de una casilla de
la matriz para el algoritmo Needleman-Wunsch
 * @author Nacho
 * @date 6/2/2018
 * @param i Indice de fila (No puede ser 0)
 * @param j Indice de columna (No puede ser 0)
 * @param igual Comparativa (Char==Char)
 * @param matrix Matriz de structs sobre la que se opera. In/Out
 */
void CalcularCasilla(unsigned i, unsigned j, bool igual, struct Celda
**matrix)
{
    // Constantes Match 2, -2 mismatch, -1 Hueco
    int A = matrix[i-1][j].score - 1;
    int B = matrix[i][j-1].score - 1;
    int C = matrix[i-1][j-1].score+ (igual*4)-2; //C= arg +
(argB*(Match-Fallo))+Fallo

    //Calculo de la direcci3n como un array de booleanos
    int D=0;
    D += (A>=B && A>=C)<<1; //Vertical en la posicion

```



```

    D += (B>=A && B>=C); //Horizontal en 2a posicion
    D += (C>=B && C>=A)<<2; //Diagonal en 3a posicion
    matrix[i][j].dir = D;

    matrix[i][j].score=maxI(maxI(A,B),C);

}

/**
 * GetRuta Funcion de backtraking para saber cual es el mayor indice
de coincidencias. Aumenta la cuenta si encuentra diagonales.
 * @author Paul
 * @author Nacho en la optimización
 * @date 12/2/2018
 * @date 14/2/2018 en optimizacion
 * @param matrix Matriz de structs sobre la que se opera. In/Out
 * @param i Indice de fila inicial (Debe ser la ultima)
 * @param j Indice de columna inicial (Debe ser la ultima)
 */
unsigned GetRuta(struct Celda** matrix, unsigned i, unsigned j)
{
    unsigned maximo = 0;
    //Reinicializado de los scores, ahora mediremos la distancia a la
esquina 0,0
    //El valor -1 representa dato desconocido
    unsigned x,y;
    for(x=0;x<=i;x++)
        for(y=0;y<=j;y++)
            matrix[x][y].score=-1;

    AuxGetRuta(matrix, i, j, 0, &maximo);

    return maximo;
}

int AuxGetRuta(struct Celda** matrix, unsigned i, unsigned j, int
cont, unsigned *maximo)
{
    int A=-1,B=-1,C=-1;
    //Poda, impide repetición de celdas si ya ha calculado el camino
    if(matrix[i][j].score>=0)
    {
        return matrix[i][j].score;
    }
    //Poda, impide recorrer un nuevo camino si no hay posibilidad de
superar la mejor marca
    if((cont+i<*maximo || cont+j<*maximo)&&(*maximo>0))
    {
        matrix[i][j].score=0;
        return 0;
    }
    //Caso base
    if(i == 0 || j == 0)
    {
        *maximo = maxI(cont, *maximo);
        matrix[i][j].score=0;;
        return 0;
    }
    else
    {

```

```
if(matrix[i][j].dir>3)
{
    //En las diagonales se añade distancia si existen
    A=AuxGetRuta(matrix, i - 1, j - 1, cont + 1, maximo)+1;
}

if(matrix[i][j].dir==2||matrix[i][j].dir==3||matrix[i][j].dir==6||matrix[i][j].dir==7)
{
    B=AuxGetRuta(matrix, i - 1, j, cont, maximo);
}

if(matrix[i][j].dir==1||matrix[i][j].dir==3||matrix[i][j].dir==5||matrix[i][j].dir==7)
{
    C=AuxGetRuta(matrix, i, j - 1, cont, maximo);
}

matrix[i][j].score=maxI(maxI(A,B),C);
return matrix[i][j].score;
}
```

Anexo: Tiempo dedicado

	Tiempo Total (Horas)	Días	Personas	Trabajo diario medio (Minutos)
Estudio del algoritmo	4	2	2	60
Implementación de código	20	7	4	43
Redacción de documentación	10	5	3	40
Total:	34	14	Variable	48

Anexo: Autoevaluación

Proyecto: Similitud de cadenas de ADN mediante el algoritmo de
alineamiento NW

Autores: Ignacio Gomis Lli
Lidia Montero Egidos
Sara Monzó Bravo
Paul Vargas Hurtado

Concepto evaluado	Valor
Compleitud: Todos los apartados incluidos, incluido tiempo dedicado (0-1)	1
Claridad de redacción: redacción, ortografía, organización (0-1)	0.9
Objetivo y utilidad del algoritmo (0-1)	0.8
Descripción del algoritmo: Descripción y en lenguaje matemático (0-1)	1
Estructura y tipo de datos, tamaño de memoria usado por el algoritmo (0-1)	0.8
Análisis del coste del algoritmo: Operaciones y accesos a memoria (0-1)	0.9
Análisis de dependencia de datos, se indica claramente que datos dependen de otros (0-1)	1
Propuestas de paralelización: Descripción de la distribución/organización de los datos y tareas que se pueden realizar en paralelo (0-1)	0.9
Referencias al final del documento (Al menos 5, 3 fuentes/revistas contrastadas y referenciadas en el texto con estilo Harvard) (0-1)	0.4
Formato del documento (Tipo de letra, PDF o DOC sin comprimir)	0.9
TOTAL (Sobre 10)	8.6

En caso de plagio o contenido sin referenciar, no se aceptará el documento para su evaluación

¿PLAGIO?	
----------	--

Práctica 0

1-Tiempos de ejecución

Hemos realizado las pruebas secuenciales, tanto sin optimizar como optimizando con la opción O2. Los resultados están presentados a continuación, los cuales son la media de 5 ejecuciones (a excepción del tamaño máximo que es media de 3 ejecuciones):

Sin optimizar	100^2	400^2	900^2	1400^2	2300^2	3200^2	12400^2	90000^2
Tiempo(μs)	751	7717.2	30120.4	87027	205275.2	413013.2	5595340.6	267597797.3
Operaciones	190000	760000	1710000	2660000	4370000	6080000	23560000	171000000
Segundos	0.000751	0.0077172	0.0301204	0.087027	0.2052752	0.4130132	5.5953406	267.5977973
MOPS	252.996	98.481	56.772	30.565	21.288	14.721	4.211	0.639

Optimizado	100^2	400^2	900^2	1400^2	2300^2	3200^2	12400^2	90000^2
Tiempo(μs)	185.4	2301.4	9889	30167.4	68609.6	164137.6	2151090.4	108709251.3
Operaciones	190000	760000	1710000	2660000	4370000	6080000	23560000	171000000
Segundos	0.0001854	0.0023014	0.009889	0.0301674	0.0686096	0.1641376	2.1510904	108.7092513
MOPS	1024.811	330.234	172.919	88.175	63.694	37.042	10.953	1.573

O0 vs O2	100^2	400^2	900^2	1400^2	2300^2	3200^2	12400^2	90000^2
Speed-Up	4.051	3.353	3.046	2.885	2.992	2.516	2.601	2.462

Hay que señalar que estos datos únicamente corresponden al apartado de cálculo de la matriz, ya que es la parte que estamos estudiando del algoritmo.

Aquí podemos observar como las operaciones por segundo decrecen conforme aumenta la talla del problema, lo cual lleva a pensar que la pérdida de rendimiento se debe a los cambios en la memoria del procesador, al necesitar realizar más cambios entre niveles de cache o memoria principal.

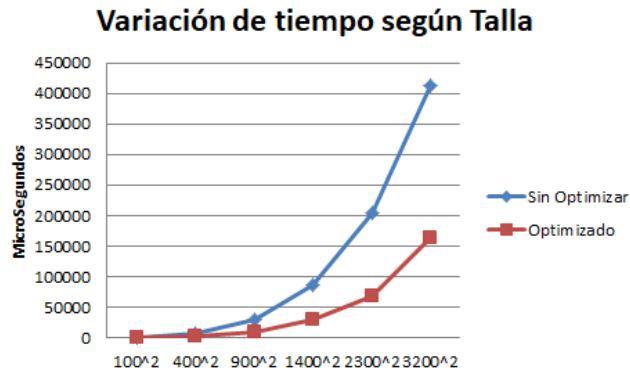
Los tamaños han sido seleccionados teniendo en cuenta los tamaños de la memoria del computador. Los hemos calculado mediante una tabla en Excel, de la cual incluimos un fragmento, ya que para obtener los valores hemos creado toda la secuencia:

Argumento	Num. Celdas	Bytes	MB	GB	
1	10000	80000	0.07629395	7.4506E-05	<- Contenido en un núcleo de nivel L2
2	40000	320000	0.30517578	0.00029802	
3	90000	720000	0.68664551	0.00067055	
4	160000	1280000	1.22070313	0.00119209	<- Contenido en 6 núcleos de nivel L2
9	810000	6480000	6.17980957	0.00603497	
					<- Prueba intermedia 1
14	1960000	15680000	14.9536133	0.01460314	<- Contenido en un núcleo de nivel L3
23	5290000	42320000	40.3594971	0.03941357	
					<- Prueba intermedia 2

32	10240000	81920000	78.125	0.07629395	<- Contenido en 6 L3
124	153760000	1230080000	1173.0957	1.14560127	<- Prueba intermedia 3
900	8100000000	6.48E+10	61798.0957	60.3497028	<- Prueba máxima
926	8574760000	6.8598E+10	65420.2271	63.8869405	Tope de memoria principal
927	8593290000	6.8746E+10	65561.5997	64.0249997	

2-Representación gráfica de los tiempos

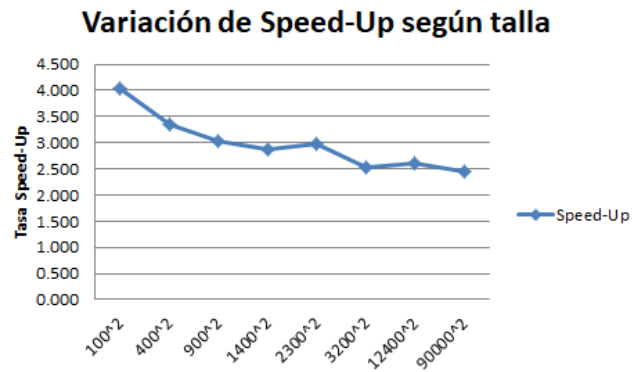
En primer lugar presentamos la gráfica de evolución de los tiempos conforme aumenta la talla, aquí se puede observar como tiene un comportamiento de orden N^2 , exactamente como esperábamos en el análisis del problema



Como los valores mayores no se pueden visualizar correctamente en esta escala, añadimos la misma gráfica pero con una escala logarítmica. Cabe señalar que estos dos últimos valores no son tan próximos entre sí como los anteriores, rompiendo la escala, por tanto no se puede apreciar correctamente la curva que se genera.



Finalmente, mostramos la gráfica del speed-Up al optimizar con O2, en la cual llama la atención su irregularidad. Nuestra explicación ante ello es debido a que los puntos donde la optimización funciona peor son algunos saltos que comparados con la anterior tienen que cambiar de caches un nivel más que en el anterior.



3-Modificaciones en el código

El código ha sido mantenido todo a excepción de las funciones para obtener el tiempo. El main por tanto ha quedado así:

```
int main( int argc, char *argv[] )
{
    unsigned T1,T2,I1,I2;
    switch(argc)
    {
        case 3: //Ambos desde el principio hasta lo que coja
            T1=100;T2=100;I1=0;I2=0;

            break;
        case 4: //Ambos con Tamaño arg[3]
            T1=atoi(argv[3]);T2=T1;I1=0;I2=0;
            break;
        case 5: //Cada uno con Tamaño arg[3] y arg[4]
            T1=atoi(argv[3]);T2=atoi(argv[4]);I1=0;I2=0;
            break;
        case 6: //Cada uno con Tamaño arg[3] y arg[4] empezando desde
            arg[5]
                T1=atoi(argv[3]);T2=atoi(argv[4]);I1=atoi(argv[5]);I2=I1;
            break;
        case 7: //Cada uno con Tamaño arg[3] y arg[4] empezando desde
            arg[5] y arg[6]
                T1=atoi(argv[3]);T2=atoi(argv[4]);I1=atoi(argv[5]);I2=atoi(argv[6]);
            break;
        default: //Instrucciones de uso
            printf("Error en introduccion de datos:\n Se pueden introducir
entre 2 y 6 argumentos:\n2 argumentos:\n    Fichero_1 Fichero_2\n");
            printf("3 argumentos:\n    Fichero_1 Fichero_2
Tamano_maximo\n");
            printf("4 argumentos:\n    Fichero_1 Fichero_2
TamanoMax_Cadenal TamanoMax_Cadena2\n");
    }
```

```

printf("5 argumentos:\n Fichero_1 Fichero_2
TamanoMax_Cadena1 TamanoMax_Cadena2 Inicio_Cadenas\n");
printf("6 argumentos:\n Fichero_1 Fichero_2
TamanoMax_Cadena1 TamanoMax_Cadena2 Inicio_C1 Inicio_C2\n");
printf("Tamano e inicio escalados: 1:100\n");
}
if(argc >= 3 && argc <=7)
{
printf("%s comparado con %s\n",argv[1],argv[2]);
printf("Tamanos: %d %d\n",T1*100,T2*100);
printf("Puntos de inicio: %d %d\n",I1*100,I2*100);
struct timeval t1,t2,t3,t4;
double total;
gettimeofday(&t1, NULL);
char* string1=CargarFichero(argv[1],T1,I1);
char* string2=CargarFichero(argv[2],T2,I2);
//printf("Fin construccion cadenas\n");
struct Celda **Matriz;
if(strlen(string1)==0 || strlen(string2)==0)
{
printf("Una cadena esta vacia");
exit(2);
}
//printf("Lectura correcta \n");
Matriz=InicializarMatriz(strlen(string1),strlen(string2));
gettimeofday(&t2, NULL);
//printf("Matriz iniciada\n");
CompletarMatriz(string1,string2,Matriz);
gettimeofday(&t3, NULL);

//printf("Matriz completa\n");
int resultado=
GetRuta(Matriz,strlen(string1),strlen(string2));
//printf("Ruta calculada\n");

gettimeofday(&t4, NULL);

total = ((t2.tv_sec * 1000000 + t2.tv_usec)-(t1.tv_sec *
1000000 + t1.tv_usec));
printf("Inicializado: %lf\n", total );

total = ((t3.tv_sec * 1000000 + t3.tv_usec)-(t2.tv_sec *
1000000 + t2.tv_usec));
printf("Creacion de matriz: %lf\n", total );

total = ((t4.tv_sec * 1000000 + t4.tv_usec)-(t3.tv_sec *
1000000 + t3.tv_usec));
printf("Backtracking: %lf\n", total );

total = ((t4.tv_sec * 1000000 + t4.tv_usec)-(t1.tv_sec *
1000000 + t1.tv_usec));
printf("Total: %lf\n", total );
printf("Coincidencia(porc): %d\n",
100*resultado/maxU(strlen(string1),strlen(string2)));
}
else
{
exit(3);
}
printf("Fin");
exit(0);

```



```
    return 0;  
}
```

El resto del código se puede consultar [aquí](#)

4-Tiempo de trabajo

Tiempo de trabajo en el laboratorio: 2 horas, 3 personas.

Tiempo de elaboración de documentación: 2 horas, 4 personas.

	Tiempo Total (Horas)	Días	Personas	Trabajo diario medio (Minutos)
Trabajo de laboratorio	2	1	3	120
Realización de documentación	2	1	4	120
Total:	4	2	Variable	240

Anexo: Autoevaluación

Proyecto: Similitud de cadenas de ADN mediante el algoritmo de alineamiento NW

Autores: Ignacio Gomis Lli
Lidia Montero Egidos
Sara Monzó Bravo
Paul Vargas Hurtado

Concepto evaluado	Valor
Compleitud de estudio experimental (0-1)	0.9
Claridad de redacción y presentación de los datos. (0-1)	0.9
Código del algoritmo secuencial definitivo. (0-2)	1.8
Representación de los datos mediante tablas y/o gráficas. (0-2)	1
Compleitud de las medidas utilizadas en el estudio experimental. (0-1)	0.8
Estudio experimental de tiempo de ejecución y velocidad. (0-2)	1.8
Justificación de los datos experimentales obtenidos en función del algoritmo y la arquitectura del ordenador. (0-1)	1.5
TOTAL (Sobre 10)	8.7

En caso de plagio o contenido sin referenciar, no se aceptará el documento para su evaluación

¿PLAGIO?	
----------	--

CP2: Propuesta de implementación con OpenMP

1-Descripción del algoritmo paralelo

Al analizar el algoritmo, observamos que de los paradigmas de programación paralela explicados en clase y detallados en la bibliografía recomendada: Rauber y Rünger (2010) el más recomendable sería el paralelismo de datos, ya que trabajamos sobre una matriz de gran tamaño y la misma tarea con la misma carga computacional en cada sección.

Por otro lado, en la sección de la búsqueda de la mejor coincidencia podría emplearse el paradigma de divide y vencerás, pero el fragmento recursivo, además de que no es el objetivo de estudio de este proyecto, tiene un fuerte componente secuencial para impedir repeticiones de caminos, por lo tanto hemos considerado que paralelizarlo solo resultaría negativo para el resultado.

Finalmente, para la configuración inicial de la matriz, hemos decidido aplicar el paradigma del maestro-esclavo, ya que lanzaremos un hilo independiente para cada fichero de entrada, tanto para leerlo como para inicializar las fila y columna 0 de la matriz, con la expectativa de que reducir a la mitad el tiempo de inicializado del programa. Este último caso es anecdótico, ya que es lo que menos coste temporal tiene en el algoritmo, pero consideramos que merecía la pena para recortar tiempos en llamadas con tamaños de matriz muy grandes.

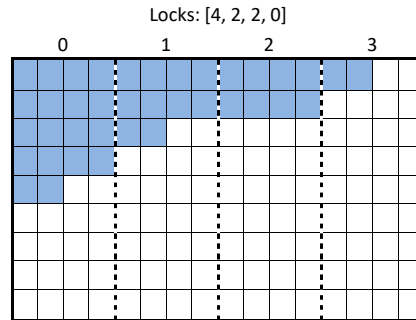
Dadas las dependencias del algoritmo descritas durante el [CP1](#) podemos observar dos posibles soluciones, paralelizar antidiagonales, o paralelizar bloques asincrónamente, es decir, cada uno esperando a que los datos para avanzar estén disponibles.

Nos decidimos por la segunda opción porque paralelizar las antidiagonales, para garantizar resultados correctos requeriría muchos cambios de bloques de memoria o una solución asíncrona similar a la de los bloques.

Desde un principio nuestro código fue desarrollado con la intención de poder repartir el trabajo en múltiples bloques, así que para esta entrega, lo más destacable a variar del algoritmo será la función anteriormente llamada CompletarMatriz, la cual únicamente organizaba el orden de cálculo de CalcularCasilla. Por tanto aquí reemplazaremos CompletarMatriz por un método que emplea OpenMP y un sistema de locks para ordenar las llamadas a CalcularCasilla.

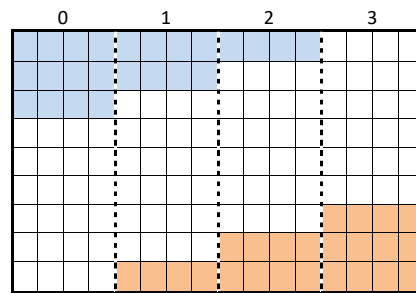
También modificaremos el Main para poder llamar a la función con el algoritmo secuencial o el algoritmo con paralelización mediante OpenMP, dejándolo preparado para ampliarlo en un futuro para MPI.

Para clarificar el método que estamos proponiendo, adjuntamos el siguiente diagrama:



Tenemos en la matriz repartida en 4 bloques, delimitados por las líneas discontinuas y habiendo calculado ya las casillas coloreadas. Se puede observar que hay adjunto un vector que indica el número de filas completadas por cada bloque. Cuando un hilo completa una fila incrementa en uno su indicador, y luego comprueba si puede comenzar la siguiente iteración, comprobando que sus filas terminadas sean menores que las del bloque anterior, que en caso contrario, simplemente dormirá esperando el momento indicado. En este caso, el hilo 2 estaría esperando a que el hilo 1 completase su fila.

Claramente, el principal problema de esto son los tiempos de espera de los procesadores ya que el último hilo no podrá operar hasta que todos los anteriores hayan completado una fila, y de la misma manera el primer hilo quedará a la espera hasta que todos los posteriores finalicen. En la siguiente imagen podemos ver en azul, en el caso ideal, las celdas que estarían completas antes de la puesta en marcha del hilo 3 y en rojo las celdas que faltarían por hacer al finalizar el hilo 0, haciendo esto del orden de 24 unidades de tiempo que no tenemos máximo paralelismo, en este ejemplo.



Sin embargo, la manera de minimizar este efecto sería el asignar múltiples bloques a un mismo hilo, ordenados de la manera indicada. La imagen adjunta, con la misma leyenda que la anterior, muestra que el tiempo de paro de procesadores resulta menor, 12 unidades de tiempo.

	0	1	2	3	0	1	2	3
0								
1								
2								
3								
0								
1								
2								
3								

Este trenzado reducirá los tiempos de inactividad, pero incrementará el número de dependencias de temporales en el programa, así que tendremos que encontrar el valor que mejor se adapte. Hay que señalar, que si el trenzado llega a tamaños muy pequeños, la solución acaba siendo la misma que trabajar por diagonales.

Entonces, la directiva de OMP que emplearemos, será `#pragma parallel`, acompañada de `get_thread_num()` y `get_max_threads()`, ajustando la cantidad de los hilos mediante la variable de entorno. La idea es tener el máximo control posible del problema, asignando nosotros a los hilos indicados la carga de trabajo deseada y el momento indicado de trabajar.

Por otro lado, la lectura y actualización de los datos de la matriz es controlada por un lock, para evitar problemas de concurrencia.

2-Implementacion del algoritmo

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>
#include <omp.h>

/*Struct con el valor de la celda y las direcciones
 Si las direcciones son verdadero, entonces ha heredado de esa
 direccion
*/
struct Celda
{
    int score;
    char dir; //Array de booleanos
};

void ImprimirInstruccionesDeUso();
char* CargarFichero(char*,unsigned,unsigned);
struct Celda** inicializarMatriz(unsigned, unsigned,unsigned);
void CompletarMatrizSecuencial(char*,char*,struct Celda**);
void CompletarMatrizOmp(char*,char*,struct Celda**,unsigned);
unsigned* AsignarVector(unsigned,unsigned);
void CalcularSubMatriz(struct
Celda**,unsigned,unsigned,char*,char*,unsigned*,unsigned);
void CalcularCasilla(unsigned, unsigned, bool, struct Celda**);
unsigned GetRuta(struct Celda**,unsigned,unsigned);
int AuxGetRuta(struct Celda**, unsigned, unsigned, int, unsigned*);

//Funciones auxiliares

/*Maximo entre dos valores*/
unsigned maxU(unsigned arg1, unsigned arg2)
{
    if(arg1>arg2){
        return arg1;
    }
    else
        return arg2;
}
int maxI(int arg1, int arg2)
{
    if(arg1>arg2){
        return arg1;
    }
    else
        return arg2;
}

/*Funcion para pasar a mayuscula una cadena. Obtenida de:
https://stackoverflow.com/questions/35181913/converting-char-to-
uppercase-in-c
*/
void Mayus(char * temp) {
    char * name;
    name = strtok(temp,":");
    char *s = name;
    while (*s) {
```

```

    *s = toupper((unsigned char) *s);
    s++;
}

/**
 * CompletarMatriz funcion que gestiona los hilos para realizar el
 algoritmo Needleman-Wunsch en multiples procesadores
 * @author Nacho
 * @date 28/2/2018
 * @param string1 Cadena de texto 1
 * @param string2 Cadena de texto 2
 * @param matrix Matriz de Celdas, su tamano debe ser el de las
 cadenas de texto +1
 * @param sobrecarga Cantidad de bloques que realiza un hilo. A mayor
 cantidad menores tiempos, pero su valor maximo deberia ser la
 longitud del string.
 */
void CompletarMatrizOmp(char* string1, char* string2, struct Celda**
matrix, unsigned sobrecarga)
{
    unsigned i;
    unsigned p=omp_get_max_threads();
    unsigned *posiciones=AsignarVector(strlen(string2), p*sobrecarga);
    unsigned *locks =(unsigned *)malloc(p*sobrecarga*
sizeof(unsigned));
    for (i = 0; i < p*sobrecarga; ++i)
    {
        locks[i] = 0;
    }

    #pragma omp parallel private(i)
    shared(matrix, posiciones, string1, string2, locks, p, sobrecarga)
    {
        unsigned id=omp_get_thread_num();
        for(i=0; i<sobrecarga; i++)
            if(id==0 && i==0)
                CalcularSubMatriz (matrix,
1, posiciones[0], string1, string2, locks, 0);
            else
                CalcularSubMatriz (matrix, posiciones[id-
1+p*i], posiciones[id+p*i], string1, string2, locks, (id+p*i));
    }
}

/**
 * AsignarVector funcion que calcula las posiciones finales de cada
 bloque
 * @author Lidia
 * @date 4/3/2018
 * @param tamano Tamano del ancho de la matriz
 * @param p Numero de procesadores
 * @return final Vector dinamico de posiciones
 */
unsigned* AsignarVector(unsigned tamano, unsigned p)
{
    unsigned a, l, i;
    unsigned *final =(unsigned *)malloc(p*sizeof(unsigned));
    l=tamano/p;
    //Garantizar un minimo avance
    if(l==0)
        l=1;
}

```

```

//Asignacion
a=1;
for(i=0;i<p-1;++i)
{
    final[i]=a;
    a+=1;
}
final[p-1]=tamano;
//Para evitar accidentes
for(i=0;i<p;++i)
{
    if(final[i]>tamano)
        final[i]=tamano;
}
return final;
}
/**
 * CalcularSubMatriz rellena la matriz a partes, entre dos columnas
objetivo
 * @author Paul
 * @date 01/03/2018
 * @param matrix Matriz sobre la que se opera
 * @param c1 indice de la columna que calcularemos
 * @param c2 indice de la columna que calcularemos
 * @param string1 Cadena de texto
 * @param string2 Cadena de texto
 * @param locks Array de cerrojos
 * @param id Identificador del hilo que esta ejecutando esta funcion
 */
void CalcularSubMatriz(struct Celda** matrix, unsigned c1, unsigned
c2, char* string1, char* string2, unsigned* locks, unsigned id)
{
    int tiempo = 500;
    unsigned sizel=strlen(string1);
    unsigned i;
    unsigned j;

    for( i = 1; i <= sizel; ++i)
    {
        for( j = c1; j <= c2; ++j)
        {
            while( id > 0 && locks[id - 1] <= locks[id] )
            {
                usleep(tiempo);
            }

            CalcularCasilla(i, j, string1[i - 1] == string2[j-1] ||
string1[i - 1] == 'N' || string2[j-1] == 'N', matrix);
        }
        locks[id]++;
    }

    return;
}

/**
 * Main Lleva control del orden de ejecucion de las funciones del
algoritmo y la entrada de parametros
 * @author Lidia y Nacho

```



```

* @date 8/2/2018
* @date 3/3/2018 añadido OMP
* @param Nombre o ruta de Fichero1 (Necesario)
* @param Nombre o ruta de Fichero2 (Necesario)
* @param Tamano de cadena de fichero1 (Opcional o ambos ficheros si
falta siguiente parametro)
* @param Tamano de cadena de Fichero2 (Opcional)
* @param Inicio de cadena de Fichero1 (Opcional o ambos ficheros si
falta siguiente parametro)
* @param Inicio de cadena de Fichero2 (Opcional)
*/
int main( int argc, char *argv[] )
{
    unsigned T1,T2,I1,I2, sobrecarga, modo;
    char* nombrel;
    char* nombre2;
    bool lecturaCorrecta=1;
    switch(atoi(argv[1]))
    {
        case 2: //OpenMP
        {
            modo=2;
            sobrecarga=atoi(argv[2]);
            nombrel=argv[3];
            nombre2=argv[4];
            switch(argc)
            {
                case 5: //Ambos desde el principio hasta lo que coja
                    T1=1000;T2=1000;I1=0;I2=0;
                    break;
                case 6: //Ambos con Tamano argv[5]
                    T1=atoi(argv[5]);T2=T1;I1=0;I2=0;
                    break;
                case 7: //Cada uno con Tamano argv[5] y argv[6]
                    T1=atoi(argv[5]);T2=atoi(argv[6]);I1=0;I2=0;
                    break;
                case 8: //Cada uno con Tamano argv[5] y argv[6]
                    empezando desde argv[7]
                    T1=atoi(argv[5]);T2=atoi(argv[6]);I1=atoi(argv[7]);I2=I1;
                    break;
                case 9: //Cada uno con Tamano argv[5] y argv[6]
                    empezando desde argv[7] y argv[8]
                    T1=atoi(argv[5]);T2=atoi(argv[6]);I1=atoi(argv[7]);I2=atoi(argv[8]);
                    break;
                default: //Instrucciones de uso
                    ImprimirInstruccionesDeUso();
                    lecturaCorrecta=0;
                    break;
            }
        }
        break;
        case 3: //MPI
        {

        }
        break;
        case 1:
        {

```

```

modo=1;
nombre1=argv[2];
nombre2=argv[3];
switch(argc)
{
    case 4: //Ambos desde el principio hasta lo que coja
        T1=1000;T2=1000;I1=0;I2=0;
        break;
    case 5: //Ambos con Tamano arg[3]
        T1=atoi(argv[4]);T2=T1;I1=0;I2=0;
        break;
    case 6: //Cada uno con Tamano arg[3] y arg[4]
        T1=atoi(argv[4]);T2=atoi(argv[5]);I1=0;I2=0;
        break;
    case 7: //Cada uno con Tamano arg[3] y arg[4]
        empezando desde arg[5]
        T1=atoi(argv[4]);T2=atoi(argv[5]);I1=atoi(argv[6]);I2=I1;
        break;
    case 8: //Cada uno con Tamano arg[3] y arg[4]
        empezando desde arg[5] y arg[6]
        T1=atoi(argv[4]);T2=atoi(argv[5]);I1=atoi(argv[6]);I2=atoi(argv[7]);
        break;
    default: //Instrucciones de uso
        ImprimirInstruccionesDeUso();
        lecturaCorrecta=0;
}
}
break;
default:
{
    lecturaCorrecta=0;
    ImprimirInstruccionesDeUso();
}
break;
}

if(lecturaCorrecta==1)
{
    printf("%s comparado con %s\n",nombre1,nombre2);
    printf("Tamanos: %d %d\n",T1*100,T2*100);
    printf("Puntos de inicio: %d %d\n",I1*100,I2*100);
    struct timeval t1,t2,t3,t4;
    double total;

    gettimeofday(&t1, NULL);

    char* string1;
    char* string2;
    switch(modo)
    {
        case 1: //Secuencial
        {
            string1=CargarFichero(nombre1,T1,I1);
            string2=CargarFichero(nombre2,T2,I2);
        }
        break;
        case 2: //omp
        {
            #pragma omp parallel sections

```

```

shared(nombre1,nombre2,T1,I1,T2,I2)
{
    #pragma omp section
    string1=CargarFichero(nombre1,T1,I1);
    #pragma omp section
    string2=CargarFichero(nombre2,T2,I2);
}
break;

default:
{
    exit(4);
}
break;

}

struct Celda **Matriz;
if(strlen(string1)==0 || strlen(string2)==0)
{
    printf("Una cadena esta vacia");
    exit(2);
}

Matriz=inicializarMatriz(strlen(string1),strlen(string2),modo);
gettimeofday(&t2, NULL);
if(modo==1)
    CompletarMatrizSecuencial(string1,string2,Matriz);
if(modo==2)
    CompletarMatrizOmp(string1,string2,Matriz,sobrecarga);
gettimeofday(&t3, NULL);
int resultado=
GetRuta(Matriz,strlen(string1),strlen(string2));

gettimeofday(&t4, NULL);

total = ((t2.tv_sec * 1000000 + t2.tv_usec)-(t1.tv_sec *
1000000 + t1.tv_usec));
printf("Inicializado:      %lf\n", total );

total = ((t3.tv_sec * 1000000 + t3.tv_usec)-(t2.tv_sec *
1000000 + t2.tv_usec));
printf("Creacion de matriz: %lf\n", total );

total = ((t4.tv_sec * 1000000 + t4.tv_usec)-(t3.tv_sec *
1000000 + t3.tv_usec));
printf("Backtracking:      %lf\n", total );

total = ((t4.tv_sec * 1000000 + t4.tv_usec)-(t1.tv_sec *
1000000 + t1.tv_usec));
printf("Total:              %lf\n", total );
printf("Coincidencia(porc): %d\n",
100*resultado/maxU(strlen(string1),strlen(string2)));
}
else
{
    exit(3);
}
printf("Fin");
exit(0);

```

```

    return 0;
}

/**
 * ImprimirInstruccionesDeUso imprime la informacion de uso de este
 * programa, asi como los autores.
 * @author Nacho
 * @date 3/03/2018
 */
void ImprimirInstruccionesDeUso()
{
    printf("Programa para el calculo de porcentaje de similitud de dos
cadenas de Adn\n");
    printf("Autores:\n");
    printf("\tIgnacio Gomis Lli\n");
    printf("\tLidia Montero Egidos\n");
    printf("\tSara Monzo Bravo\n");
    printf("\tPaul Vargas Hurtado\n");
    printf("Asignatura: Arquitectura de procesadores, 3ro Ingenieria
Informatica\n");
    printf("Profesorado:\n");
    printf("\tJose Manuel Claver Iborra\n");
    printf("\tAdria Gimenez Pastor\n");
    printf("Universidad de Valencia, ETSE\n");
    printf("Curso 2017/2018\n");
    printf("\n");
    printf("Instrucciones de uso:\n");
    printf("-Los ficheros deben estar escritos en formato fasta\n");
    printf("-Los argumentos de tamaño tienen un escalado 1:100, es
decir, en caso de que\n");
    printf("se introduzca un 1 nos referimos a la posición 100 del
fichero\n");
    printf("\n");
    printf("Existen distintos modos de ejecución, se determina cual se
emplea según el\n");
    printf("primer argumento\n");
    printf("-Ejecución secuencial, primer argumento 1\n");
    printf("3 argumentos:\n 1 Bloques_Por_Hilo Fichero_1
Fichero_2\n");
    printf("4 argumentos:\n 1 Fichero_1 Fichero_2 Tamaño_maximo\n");
    printf("5 argumentos:\n 1 Fichero_1 Fichero_2 TamañoMax_Cadena1
TamañoMax_Cadena2\n");
    printf("6 argumentos:\n 1 Fichero_1 Fichero_2 TamañoMax_Cadena1
TamañoMax_Cadena2 Inicio_Cadenas\n");
    printf("7 argumentos:\n 1 Fichero_1 Fichero_2 TamañoMax_Cadena1
TamañoMax_Cadena2 Inicio_C1 Inicio_C2\n");
    printf("\n");
    printf("-Ejecución con OMP, segundo argumento 2, requiere opción
de compilado -fopenmp\n");
    printf("4 argumentos:\n 2 Bloques_Por_Hilo Fichero_1
Fichero_2\n");
    printf("5 argumentos:\n 2 Bloques_Por_Hilo Fichero_1 Fichero_2
Tamaño_maximo\n");
    printf("6 argumentos:\n 2 Bloques_Por_Hilo Fichero_1 Fichero_2
TamañoMax_Cadena1 TamañoMax_Cadena2\n");
    printf("7 argumentos:\n 2 Bloques_Por_Hilo Fichero_1 Fichero_2
TamañoMax_Cadena1 TamañoMax_Cadena2 Inicio_Cadenas\n");
    printf("8 argumentos:\n 2 Bloques_Por_Hilo Fichero_1 Fichero_2
TamañoMax_Cadena1 TamañoMax_Cadena2 Inicio_C1 Inicio_C2\n");
}

```

```

/**
 * CargarFichero funcion que guarda en un string el contenido de un
 fichero .fasta
 * @author Lidia y Nacho
 * @date 8/2/2018
 * @param NombreFichero nombre del fichero, incluida extension y ruta
 relativa
 * @out string con el contenido en mayusculas
 */
char* CargarFichero(char* NombreFichero,unsigned tamano,unsigned
inicio)
{
    tamano*=100;
    inicio*=100;
    FILE *archivo;
    unsigned i;
    char caracteres[100];
    char *cadena=malloc(tamano);    //<--Origen error
    strcpy (cadena, "");
    archivo = fopen(NombreFichero,"r");
    if (archivo == NULL)
    {
        printf("%s no existe",NombreFichero);
        exit(1);
    }
    else
    {
        fgets(caracteres,100,archivo); //Primera linea
        for(i=0;i<inicio;i++)
            fgets(caracteres,100,archivo);
        i=tamano;
        while (feof(archivo) == 0 && strlen(cadena)<i) //Hasta fin de
archivo o memoria
        {
            fgets(caracteres,100,archivo);
            strcat(cadena, caracteres);
        }
        Mayus(cadena);
        return cadena;
    }
}

// inicializarMatriz funcion que crea la matriz de tamano r c e
inicializa la primera fila y columna con valores negativos
descendentes.
// @author Sara
// @date 12/2/2018
// @param unsigned r rows
// @param unsigned c cols
// @return arr matriz con los valores negativos
struct Celda** inicializarMatriz(unsigned r, unsigned c, unsigned m)
{
    unsigned i;
    struct Celda **arr =(struct Celda **)malloc(r*c* sizeof(struct
Celda));
    for (i = 0; i <= r; ++i)
        arr[i] = (struct Celda *)malloc(c * sizeof(struct Celda));
    //Casos base posicion:  r = 0, c = 0
    arr[0][0].score = 0;

```

```

arr[0][0].dir = 0;
switch(m)
{
    case 1: //Secuencial
    {
        for(i = 1 ; i<=r; i++)
        {
            arr[i][0].score = -i;
            arr[i][0].dir=0;
        }
        for(i = 1 ; i<=c; i++)
        {
            arr[0][i].score = -i;
            arr[0][i].dir=0;
        }
    }
    case 2: //Omp
    {
        #pragma omp parallel sections shared(arr) private (i)
        {
            #pragma omp section
            {
                for(i = 1 ; i<=r; i++)
                {
                    arr[i][0].score = -i;
                    arr[i][0].dir=0;
                }
            }
            #pragma omp section
            {
                for(i = 1 ; i<=c; i++)
                {
                    arr[0][i].score = -i;
                    arr[0][i].dir=0;
                }
            }
        }
    }
    break;
default:
{
    exit(4);
}
break;
}

return arr;
}

/**
 * CompletarMatriz funcion que calcula el algoritmo Needleman-Wunsch
 para una matriz
 * @author Nacho
 * @date 7/2/2018
 * @param string1 Cadena de texto 1
 * @param string2 Cadena de texto 2
 * @param matrix Matriz de Celdas, su tamano debe ser el de las
 cadenas de texto +1
 */
void CompletarMatrizSecuencial(char* string1,char* string2,struct

```

```

Celda** matrix)
{
    unsigned i;
    unsigned j;
    unsigned size1=strlen(string1);
    unsigned size2=strlen(string2);
    for(i=1;i<=size1;++i)
        for(j=1;j<=size2;++j)
            //El argumento de calcular casilla es cierto si ambos
            strings coinciden o uno de ellos es N
            //Recordar que el tamaño de la matriz es 1 mayor que los
            strings, y estos se alinean con el final.
            {
                CalcularCasilla(i, j, (string1[i-1]==string2[j-
1]||string1[i-1]=='N'||string2[j-1]=='N'), matrix);
            }
    return;
}

/**
 * CalcularCasilla funcion que calcula el contenido de una casilla de
 * la matriz para el algoritmo Needleman-Wunsch
 * @author Nacho
 * @date 6/2/2018
 * @param i Indice de fila (No puede ser 0)
 * @param j Indice de columna (No puede ser 0)
 * @param igual Comparativa (Char==Char)
 * @param matrix Matriz de structs sobre la que se opera. In/Out
 */
void CalcularCasilla(unsigned i, unsigned j, bool igual, struct Celda
**matrix)
{
    // Constantes Match 2, -2 mismatch, -1 Hueco
    int A = matrix[i-1][j].score - 1;
    int B = matrix[i][j-1].score - 1;
    int C = matrix[i-1][j-1].score+ (igual*4)-2; //C= arg +
    (argB*(Match-Fallo))+Fallo

    //Calculo de la dirección como un array de booleanos
    int D=0;
    D += (A>=B && A>=C)<<1; //Vertical en la posición
    D += (B>=A && B>=C); //Horizontal en 2a posición
    D += (C>=B && C>=A)<<2; //Diagonal en 3a posición
    matrix[i][j].dir = D;

    matrix[i][j].score=maxI(maxI(A,B),C);
}

/**
 * GetRuta Funcion de backtracking para saber cual es el mayor indice
 * de coincidencias. Aumenta la cuenta si encuentra diagonales.
 * @author Paul
 * @author Nacho en la optimización
 * @date 12/2/2018
 * @date 14/2/2018 en optimización
 * @param matrix Matriz de structs sobre la que se opera. In/Out
 * @param i Indice de fila inicial (Debe ser la última)
 * @param j Indice de columna inicial (Debe ser la última)
 */

```

```

unsigned GetRuta(struct Celda** matrix, unsigned i, unsigned j)
{
    unsigned maximo = 0;
    //Reinicializado de los scores, ahora mediremos la distancia a la
    esquina 0,0
    //El valor -1 representa dato desconocido
    unsigned x,y;
    for(x=0;x<=i;x++)
        for(y=0;y<=j;y++)
            matrix[x][y].score=-1;

    AuxGetRuta(matrix, i, j, 0, &maximo);

    return maximo;
}

int AuxGetRuta(struct Celda** matrix, unsigned i, unsigned j, int
cont, unsigned *maximo)
{
    int A=-1,B=-1,C=-1;
    //Poda, impide repetición de celdas si ya ha calculado el camino
    if(matrix[i][j].score>=0)
    {
        return matrix[i][j].score;
    }
    //Poda, impide recorrer un nuevo camino si no hay posibilidad de
    superar la mejor marca
    if((cont+i<*maximo || cont+j<*maximo)&&(*maximo>0))
    {
        matrix[i][j].score=0;
        return 0;
    }
    //Caso base
    if(i == 0 || j == 0)
    {
        *maximo = maxI(cont, *maximo);
        matrix[i][j].score=0;;
        return 0;
    }
    else
    {
        if(matrix[i][j].dir>3)
        {
            //En las diagonales se añade distancia si existen
            A=AuxGetRuta(matrix, i - 1, j - 1, cont + 1, maximo)+1;
        }

        if(matrix[i][j].dir==2||matrix[i][j].dir==3||matrix[i][j].dir==6||matr
ix[i][j].dir==7)
        {
            B=AuxGetRuta(matrix, i - 1, j, cont, maximo);
        }

        if(matrix[i][j].dir==1||matrix[i][j].dir==3||matrix[i][j].dir==5||matr
ix[i][j].dir==7)
        {
            C=AuxGetRuta(matrix, i, j - 1, cont, maximo);
        }
    }
}

```



```
matrix[i][j].score=maxI(maxI(A,B),C);  
return matrix[i][j].score;  
}
```

3-Análisis del algoritmo

Coste computacional

Reduciendo la [tabla anterior](#) a los cambios realizados para operar con OMP, y siendo S el número de veces que un hilo consulta el cerrojo y L el número de bloques en los que se ha partido la matriz.

Función	Lecturas Matriz	Escrituras Matriz	Lecturas Cadenas	Escrituras Cadenas	Operaciones
CalcularCasilla	3	2	4	0	19
AsignarVector	0	0	0	L	$3+2L$
CompletarSubmatriz (Propias)	0	0	S	N	2S
CompletarMatriz(Total)	3NM	2NM	$4NM+LS$	$L(N+1)$	$19NM+2LS+2L+3$
Orden Esperado	$N*M$	$N*M$	$N*M+L*S$	$L*N$	$N*M+L*S$

Como podemos comparar con la tabla anterior, los sobrecostes, la inicialización y el control de los hilos han incrementado las operaciones totales de la función Completar Matriz, aunque la cota superior sigue estando delimitada por $N*M$, nos encontramos con $L*S$, que aunque L será conocido y de tamaño deseado, S resultará impredecible, y en algún caso podría ser muy elevado, aunque suponemos que el coste vendrá más marcado por las operaciones de calcular casillas.

Coste de memoria

Tras realizar las pruebas con un único procesador hemos comprobado que el empleo de Int para las celdas es suficiente, sin necesidad de pasar a usar Doubles, por lo tanto el tamaño de la matriz es el mismo que calculamos en el [CPI](#).

La paralelización nos añade al coste de memoria dos vectores de Unsigneds con la cantidad de hilos*bloquesPorHilo, lo cual podemos considerar despreciable en tamaños menores o incluso del coste de una fila adicional en la matriz, es decir, para un problema de P procesadores con Q bloques por hilo, tendría un peso total de $P*Q*4$ bytes.

Estimaciones

Debido a la estructura de esta paralelización, asumiendo que los procesadores fuesen igual de rápidos entre sí y la carga estuviera perfectamente distribuida y la matriz tuviera un gran tamaño, nos encontraríamos con que este problema sería escalable, ya que mientras los hilos no tuviesen que detenerse, todos podrían funcionar concurrentemente durante la mayor parte del algoritmo. En este caso los speed-ups podrían tender a P y la eficiencia a 1, ya que el código secuencial es únicamente la configuración de los hilos.

Sin embargo, eso es el caso ideal, en la práctica nos encontraremos con que los sobrecostes pueden resultar muy elevados si no usamos un trenzado lo bastante alto o si por el contrario el trenzado resulta tan minucioso que los hilos se encuentren más tiempo comprobando que pueden avanzar que calculándose, además de que si el ancho del bloque es menor que el de un bloque en la memoria de cache, se podrían generar demasiadas comunicaciones entre las memorias. Es por ello que esperamos que si manteniendo el tamaño y los hilos, variando el entrelazado, el speed-up realizará una curva que alcanzará un máximo en un punto intermedio.

Por el contrario, si mantenemos fijo el entrelazado e hilos y aumentamos el tamaño, los sobrecostes de inicio y final aumentan de manera lineal y finalmente si aumentamos el número de hilos, el sobrecoste por inicio y final también aumenta.

Por tanto, podemos suponer que la diferencia entre los valores ideales teorizados en un inicio será ocasionada únicamente por estos sobrecostes, los cuales podrían comprometer que nuestro algoritmo resulte escalable, así como que obtengamos eficiencias más bajas de lo esperado.

Anexo: Tiempo trabajado

	Tiempo Total (Horas)	Días	Personas	Trabajo diario medio (Minutos)
Estudio de la paralización	3	2	1	90
Implementación de código	8	4	4	30
Redacción de documentación	10	3	3	65
Total:	21	9	Variable	140

Anexo: Autoevaluación

Proyecto: Similitud de cadenas de ADN mediante el algoritmo de alineamiento NW

Autores: Ignacio Gomis Lli
Lidia Montero Egidos
Sara Monzó Bravo
Paul Vargas Hurtado

Concepto evaluado	Valor
Complejidad (0-1)	1
Claridad de redacción (0-1)	1
Referencia al análisis de dependencias en el CP1 (0-0.5)	0.5
Discusión de paradigmas de programación aplicables al caso. (0-1)	0.75
Análisis del coste computacional del algoritmo paralelo (0-1)	0.75
Propuesta de directivas de OpenMP y organización del código para la paralelización del algoritmo (0-2)	2
Estimación analítica de aceleración, eficiencia y escalabilidad (0-1)	0.5
Nuevas referencias (0-0.5)	0.5
Formato del documento (0-0.5)	1
TOTAL (Sobre 10)	8

En caso de plagio o contenido sin referenciar, no se aceptará el documento para su evaluación

¿PLAGIO?	
----------	--

Anexo: Referencias

Wikipedia (7/2/2018). Wikipedia. [online]. Available from https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm [10/2/2018]

National Center for Biotechnology Information (n.d.). NCBI.[online]. Available from <https://www.ncbi.nlm.nih.gov/genome/?term=homo+sapien> [10/2/2018]

Needleman, Saul B. and Wunsch, Christian D. (1970). "A general method applicable to the search for similarities in the amino acid sequence of two proteins", in Journal of Molecular Biology. Vol. 48 No.3, pp .443–53.

Chetan (11/08/2008) Technology Blog. [online]. Available from <http://technology66.blogspot.com.es/2008/08/sequence-alignment-techniques.html> [10/02/2018]

Rauber, Thomas and R nger, Gudula (2010). Parallel programming for multicore and cluster systems, Berlin, Springer-Verlag.