UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

COMPUTACIÓN DE ALTO DESEMPEÑO

VNIVERSIDAD NACIONAL
AVTONOMA DE
MEXICO

PROYECTO FINAL

OpenMP

Grupo:

2

Integrantes:

Ferrusca Ortiz Jorge Luis
Hernández González Ricardo Omar
Torres Caballero Bruno

Profesor

M. En I. Oscar René Valdez Casillas

Semestre

2020-1

Fecha de Entrega

21 de Noviembre de 2019

- **Índice de contenido**:

- **Índice de gráficas**:

⇒ **Objetivo**:

Que los alumnos usen sus conocimientos en programación estructurada en lenguaje C, para desarrollar un programa que sea capaz de leer un archivo de texto plano, obtener todas las cadenas de caracteres. Deberá recibir una matriz no singular desde la línea de comandos que es con la que efectuará los cálculos para el cifrado.
El resultado obtenido o la cadena cifrada se deberá guardar en un archivo de salida.
Se compararán los resultados obtenidos en los programas en forma serie y en paralelo.

⇒ **Antescedentes**:

OpenMP

OpenMP es un Api para programación multiproceso o multihilo de memoria compartida, que se fundamenta en el modelo fork/join.
Está formado por un conjunto de directivas, funciones de librería y variables de entorno, donde las directivas le permiten al programador comunicarse con el compilador y las funciones de librería son las que asignan o preguntan por los parámetros paralelos que se van a usar.
OpenMP es uno de los programas para programación paralela más usados hoy en día que trabaja en C/C++ y Fortran.

Modelo Fork/Join

Cuando se tiene una tarea muy pesada, puede ser dividida en tareas independientes más sencillas (fork) que cuando cada una tiene una solución a la tarea, se combinan (join) para poder tener la solución general a la tarea principal.

En OpenMP cuando se inicia un programa se crea un hilo maestro o padre que cuando necesita ingresar a una región paralela crea los hilos necesarios para poder realizar la tarea. Después de que cada hilo termine de realizar sus tareas, se presenta una sincronización y se sigue con la ejecución del programa. Puede que alguno de los hilos que se crearon necesite hacer este mismo procedimiento lo cual se puede realizar sin ningún problema.

Hill
Este algoritmo data de 1929 cuando es dado a conocer por su creador, el matemático Lester S. Hill a través de un artículo publicado en el diario de Nueva York. Dicho algoritmo se basa en el uso de álgebra lineal, específicamente las reglas del álgebra de matrices con la intención de mejorar las técnicas de cifrado utilizadas entonces, objetivo que cumple satisfactoriamente y que permite trabajar de manera práctica con tres o más símbolos de manera simultánea. Así las reglas a seguir son las siguientes:

Regla	Indicaciones
1	Asignar un valor numérico a cada letra del alfabeto a utilizar iniciando en cero

2	La clave a utilizar debe constar de tantas letras como se desee siempre que sea posible colocar los equivalentes numéricas de cada una de ellas en una matriz de NxN
3	El Mcla se divide en diagramas, trigramas o Ngramas necesarios, tal que sus equivalentes numéricos sean colocados en matrices de Nx1
4	El criptograma se obtiene multiplicando las matrices K*Mcla, esto es, Cripto(Nx1) = K(NxN) * Mcla(Nx1)
5	El mensaje en claro se recupera llevando a cabo el proceso inverso, sólo que en este caso la multiplicación se realizará entre las matrices correspondientes a y Cripto

Cabe mencionar que todas las operaciones aritméticas se realizan en la forma módulo n, donde n corresponde al tamaño del alfabeto que se esté empleando.

⇒ **Desarrollo de contenido:**

⇒ **Código Serie:**

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

int** reserveMemoryMatrix(int rows, int columns);
void fillMatrix(int** matrix, int dimension);
void printMatrix(int** matrix,  int rows, int columns);
char* getAlphabet();
// Functions for read file and get a string clean for encrypt
char* readFile(char *filename);
void replaceAndremoveSpaces(char* string,char* alphabet);
int checkIfExist(char character, char* alphabet);
// Function for fill string only if it's necessary
char* completeText(char* string, int numberMissingCharacters);
// Functions for convert string to vectors of numbers
int** separateStringToVectors(char* stringToSeparate, int numberOfVectors, int
dimension, char* alphabet);
int* convertVectorToNumbers(char* vector, char* alphabet);

int* multiplyVector(int** matrix, int *vector, int dimension, int module);

char* convertNumbersToStrign(int *vector, char* alphabet, int dimension);
char* encryptVector(int** matrix, char * alphabet, int dimension, int numberOfVectors);
void printFile(char* fileName, char* string);

int main(int argc, char *argv[]){
   int dimension, **nonSingularMatrix, *vector, lenghtAlphabet, **matrixOfVectors, module;
   char *stringToEncrypt, *alphabet, *textEncrypt;

   if(argc != 2){
      printf("numero de parametros incorrecto:  program <file>\n");
      return 0;
   }

   // Get Alphabet for encrypt
   alphabet = getAlphabet();
   lenghtAlphabet = strlen(alphabet);

   printf("Ingresa la dimension de la matriz: ");
   scanf("%d", &dimension);

   nonSingularMatrix = reserveMemoryMatrix(dimension, dimension);
   fillMatrix(nonSingularMatrix, dimension);

   // Reading from file, return string
```

```
    stringToEncrypt = readFile(argv[1]);

    // Removing blank spaces for string to encrypt, pass by value is used
    replaceAndremoveSpaces(stringToEncrypt,alphabet);

    // check if string is complete, otherwise it is filled
    module = strlen(stringToEncrypt) % dimension;
    if(module != 0){
        printf("Necesita completarse\n");
        int numberMissingCharacters = dimension - module;
        stringToEncrypt = completeText(stringToEncrypt, numberMissingCharacters);
    }

    // up to this point, ready string separate in vectors of numbers for multiply with non
singular matrix
    int numberOfVectors = strlen(stringToEncrypt)/dimension;
    matrixOfVectors = separateStringToVectors(stringToEncrypt, numberOfVectors,
dimension, alphabet);

    for (int i = 0; i < numberOfVectors; i++){
        matrixOfVectors[i] = multiplyVector(nonSingularMatrix, matrixOfVectors[i], dimension,
strlen(alphabet));
    }


    textEncrypt = encryptVector(matrixOfVectors, alphabet, dimension, numberOfVectors);

    printFile("salida.txt",textEncrypt);

    return 0;
}


char* getAlphabet(){
    // return "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz .,;-!?_";
    // return "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ”;
}

char* readFile(char *fileName){
    char *buffer = NULL;
    int string_size, read_size;
    FILE *file = fopen(fileName, "r");

    if (file){
        // Seek the last byte of the file
        fseek(file, 0, SEEK_END);
        // Offset from the first to the last byte, or in other words, filesize
        string_size = ftell(file);
        // go back to the start of the file
```

```
        rewind(file);

        // Allocate a string that can hold it all
        buffer = (char*) malloc(sizeof(char) * (string_size + 1) );

        // Read it all in one operation
        read_size = fread(buffer, sizeof(char), string_size, file);

        // fread doesn't set it so put a \0 in the last position
        // and buffer is now officially a string
        buffer[string_size] = '\0';

        if(string_size != read_size){
            // Something went wrong, throw away the memory and set
            // the buffer to NULL
            free(buffer);
            buffer = NULL;
        }

        // Always remember to close the file.
        fclose(file);
    }

    return buffer;
}

void printFile(char* fileName, char* string){
    FILE * file = fopen(fileName, "w");
    if(file){
        fputs(string,file);
        fclose(file);
    }
}

void replaceAndremoveSpaces(char* str, char* alphabet) {
    size_t str_len = strlen(str);
    char result[str_len];
    size_t p = 0;
    size_t i = 0;
    for (i = 0; i < str_len; ++i) {
        if (str[i] != ' ') {
            if(str[i] == 10 || str[i] == 11 || str[i] == 13 || checkIfExist(str[i], alphabet)){
                result[p] = str[i];
            }else{
                switch((unsigned int)str[i]){
                    case 165: //Ñ
                      result[p] = 'N';
                      break;
                    case 164: //ñ
                      result[p] = 'n';
```

```
                    break;
                default:
                    result[p] = 'X';
            }
        }
        p++;
    }
  }
  if (p < str_len){
    str[p] = '\0';
  }
  for (i = 0; i < p; ++i) {
    str[i] = result[i];
  }
}

int checkIfExist(char character, char* alphabet){
  for (int j = 0; j < strlen(alphabet); j++){
    if (character == alphabet[j]){
      return 1;
    }
  }
  return 0;
}


char* completeText(char* string, int numberMissingCharacters){
  char *new_string = (char*) calloc (strlen(string)+numberMissingCharacters,
sizeof(char));

  char *complements = (char*) calloc (numberMissingCharacters, sizeof(char));

  for (int i = 0; i < numberMissingCharacters; i++)
    complements[i] = 'X';

  strcpy(new_string, string);
  strcat(new_string, complements);

  return new_string;
}

int** separateStringToVectors(char* stringToSeparate, int numberOfVectors, int
dimension, char* alphabet){
  char **arrayOfStrings;
  int **matrixOfNumbers;

  // Reserve memory for array containing string vectors
  arrayOfStrings = (char**) calloc(numberOfVectors, sizeof(char*));

  for (int i = 0; i < numberOfVectors; i++)
```

```
        arrayOfStrings[i] = (char*) calloc(dimension, sizeof(char));

    // Resever memory for matrix containing numbers vectors
    matrixOfNumbers = reserveMemoryMatrix(numberOfVectors, dimension);

    // Separate string each 'dimension' elements
    for (int i = 0; i < numberOfVectors; i++){
        for (int j = 0; j < dimension; j++){
            arrayOfStrings[i][j] = stringToSeparate[(i*dimension)+j];
        }
    }

    // Convert each string to numbers according to the alphabet
    for (int i = 0; i < numberOfVectors; i++)
        matrixOfNumbers[i] = convertVectorToNumbers(arrayOfStrings[i], alphabet);

    return matrixOfNumbers;

}

int* convertVectorToNumbers(char* vector, char* alphabet){
    int* vectorOFNumbers = (int*) calloc(strlen(vector), sizeof(int));

    for (int i = 0; i < strlen(vector); i++){
        for (int j = 0; j < strlen(alphabet); j++){
            if (vector[i] == alphabet[j]){
                vectorOFNumbers[i] = j;
            }
        }
    }

    return vectorOFNumbers;
}

char* encryptVector(int** matrix, char * alphabet, int dimension, int numberOfVectors){
    char* ev = (char*) calloc(dimension*numberOfVectors,sizeof(char));
    for(int i=0; i<numberOfVectors; i++){
        ev = strcat(ev, convertNumbersToStrign(matrix[i],alphabet,dimension));
    }
    return ev;
}

char* convertNumbersToStrign(int *vector, char* alphabet, int dimension){
    char* str = (char*) calloc(dimension,sizeof(char));
    int pos;
    for(int i = 0 ; i<dimension ; i++){
        str[i] = alphabet[vector[i]];
    }
    return str;
}
```

```
int* multiplyVector(int** matrix, int *vector, int dimension, int module){
    //Reserving memory for new vector
    int *resultVector = (int*) calloc(dimension, sizeof(int));
    int i, j, tmp;

    //Iterating the matrix and multiplying by the vector pointer
    for(i=0; i < dimension; i++) {
        for(j=0; j < dimension; j++) {
            *(resultVector + i) += (matrix[i][j] * (*(vector + j)));
        }
        // Apply module
        *(resultVector+i) = *(resultVector+i) % module;
    }

    return resultVector;
}

int** reserveMemoryMatrix(int rows, int columns){
    int **matrix = (int **) calloc(rows, sizeof(int*));

    for (int i = 0; i < rows; i++)
        matrix[i] = (int*) calloc(columns, sizeof(int));

    return matrix;
}

void fillMatrix(int** matrix, int dimension){
    for (int i = 0; i < dimension; i++){
        for (int j = 0; j < dimension; j++){
            printf("Ingresa el elemento [%d][%d] de la matriz: ",i,j);
            scanf("%d",&matrix[i][j]);
        }
    }
}

void printMatrix(int** matrix, int rows, int columns){
    for (int i = 0; i < rows; i++){
        for (int j = 0; j < columns; j++){
            printf("%d ",matrix[i][j]);
        }
        printf("\n");
    }
}
```

⇒ **Código Paralelo:**

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

int** reserveMemoryMatrix(int rows, int columns);
void fillMatrix(int** matrix, int dimension);
void printMatrix(int** matrix,  int rows, int columns);
char* getAlphabet();
// Functions for read file and get a string clean for encrypt
char* readFile(char *filename);
void replaceAndremoveSpaces(char* string,char* alphabet);
int checkIfExist(char character, char* alphabet);
// Function for fill string only if it's necessary
char* completeText(char* string, int numberMissingCharacters);
// Functions for convert string to vectors of numbers
int** separateStringToVectors(char* stringToSeparate, int numberOfVectors, int
dimension, char* alphabet);
int* convertVectorToNumbers(char* vector, char* alphabet);
int* multiplyVector(int** matrix, int *vector, int dimension, int module);
char* convertNumbersToStrign(int *vector, char* alphabet, int dimension);
char* encryptVector(int** matrix, char * alphabet, int dimension, int numberOfVectors);
void printFile(char* fileName, char* string);

int main(int argc, char *argv[]){
   int dimension, **nonSingularMatrix, *vector, lenghtAlphabet, **matrixOfVectors, module;
   char *stringToEncrypt, *alphabet, *textEncrypt;
   int i;

   if(argc != 2){
      printf("numero de parametros incorrecto:  program <file>\n");
      return 0;
   }

   #pragma omp parallel
   {
      #pragma omp single
      {
         alphabet = getAlphabet();
         lenghtAlphabet = strlen(alphabet);
      }

      #pragma omp single
      {
         printf("Ingresa la dimension de la matriz: ");
         scanf("%d", &dimension);
      }

      #pragma omp single nowait
```

```
    {
        stringToEncrypt = readFile(argv[1]);
    }
}

#pragma omp parallel sections
{
    #pragma omp section
    {
        nonSingularMatrix = reserveMemoryMatrix(dimension, dimension);
    }

    #pragma omp section
    {
        // Removing blank spaces for string to encrypt, pass by value is used
        replaceAndremoveSpaces(stringToEncrypt,alphabet);
    }
}

#pragma omp parallel sections
{
    #pragma omp section
    fillMatrix(nonSingularMatrix, dimension);

    #pragma omp section
    {
        module = strlen(stringToEncrypt) % dimension;
        if(module != 0){
            // printf("Necesita completarse\n");
            int numberMissingCharacters = dimension - module;
            stringToEncrypt = completeText(stringToEncrypt, numberMissingCharacters);
        }
    }
}

int numberOfVectors = strlen(stringToEncrypt)/dimension;

matrixOfVectors = separateStringToVectors(stringToEncrypt, numberOfVectors,
dimension, alphabet);

#pragma omp parallel for private(i) shared(nonSingularMatrix, matrixOfVectors)
firstprivate(dimension, alphabet)
for (i = 0; i < numberOfVectors; i++)
    matrixOfVectors[i] = multiplyVector(nonSingularMatrix, matrixOfVectors[i], dimension,
strlen(alphabet));

textEncrypt = encryptVector(matrixOfVectors, alphabet, dimension, numberOfVectors);

printFile("salida.txt",textEncrypt);

printf("Texto cifrado en archivo salida.txt\n");
```

```
    return 0;
}


char* getAlphabet(){
    // return "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz .,;-!?_";
    // return "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
}

char* readFile(char *fileName){
    char *buffer = NULL;
    int string_size, read_size;
    FILE *file = fopen(fileName, "r");

    if (file){
        // Seek the last byte of the file
        fseek(file, 0, SEEK_END);
        // Offset from the first to the last byte, or in other words, filesize
        string_size = ftell(file);
        // go back to the start of the file
        rewind(file);

        // Allocate a string that can hold it all
        buffer = (char*) malloc(sizeof(char) * (string_size + 1) );

        // Read it all in one operation
        read_size = fread(buffer, sizeof(char), string_size, file);

        // fread doesn't set it so put a \0 in the last position
        // and buffer is now officially a string
        buffer[string_size] = '\0';

        if(string_size != read_size){
            // Something went wrong, throw away the memory and set
            // the buffer to NULL
            free(buffer);
            buffer = NULL;
        }

        // Always remember to close the file.
        fclose(file);
    }

    return buffer;
}

int** reserveMemoryMatrix(int rows, int columns){
    int **matrix  = (int **) calloc(rows, sizeof(int*));
    int i;
```

```
    for (i = 0; i < rows; i++)
        matrix[i] = (int*) calloc(columns, sizeof(int));

    return matrix;
}

void replaceAndremoveSpaces(char* str, char* alphabet) {
    size_t str_len = strlen(str);
    char result[str_len];
    size_t p = 0;
    size_t i = 0;
    for (i = 0; i < str_len; ++i) {
        if (str[i] != ' ') {
            if(str[i] == 10 || str[i] == 11 || str[i] == 13 || checkIfExist(str[i], alphabet)){
                result[p] = str[i];
            }else{
                switch((unsigned int)str[i]){
                    case 165: //Ñ
                      result[p] = 'N';
                      break;
                    case 164: //ñ
                      result[p] = 'n';
                      break;
                    default:
                      result[p] = 'X';
                }
            }
            p++;
        }
    }

    if (p < str_len){
        str[p] = '\0';
    }

    for (i = 0; i < p; ++i) {
        str[i] = result[i];
    }

}

int checkIfExist(char character, char* alphabet){
    for (int j = 0; j < strlen(alphabet); j++){
        if (character == alphabet[j]){
            return 1;
        }
    }
    return 0;
}
```

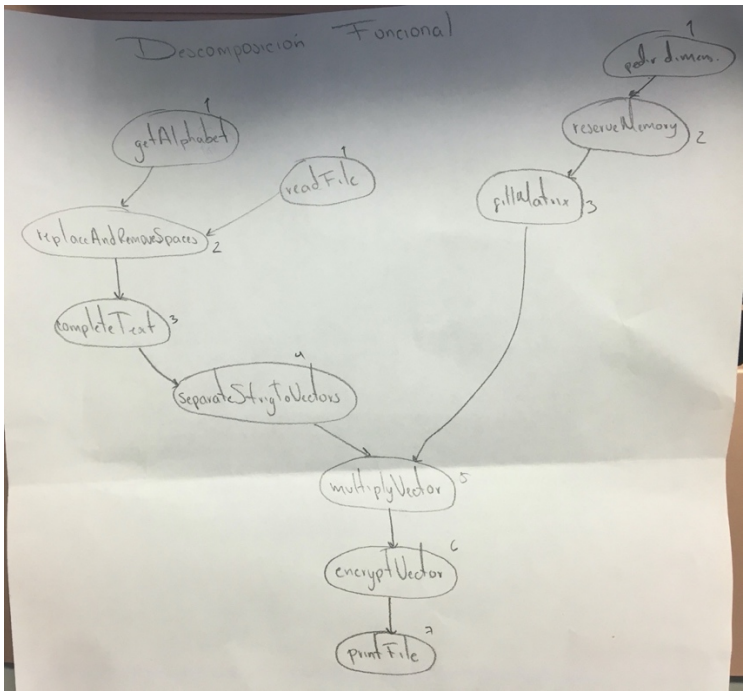
```
void fillMatrix(int** matrix, int dimension){
    int i,j;

    for (i = 0; i < dimension; i++){
        for (j = 0; j < dimension; j++){
            printf("Ingresa el elemento [%d][%d] de la matriz: ",i,j);
            scanf("%d",&matrix[i][j]);
        }
    }
}

char* completeText(char* string, int numberMissingCharacters){
    char *new_string = (char*) calloc (strlen(string)+numberMissingCharacters,
sizeof(char));

    char *complements = (char*) calloc (numberMissingCharacters, sizeof(char));

    for (int i = 0; i < numberMissingCharacters; i++)
        complements[i] = 'X';

    strcpy(new_string, string);
    strcat(new_string, complements);

    return new_string;
}

int** separateStringToVectors(char* stringToSeparate, int numberOfVectors, int
dimension, char* alphabet){
    char **arrayOfStrings;
    int **matrixOfNumbers;
    int i,j;

    arrayOfStrings = (char**) calloc(numberOfVectors, sizeof(char*));

    #pragma omp parallel for shared(arrayOfStrings, dimension) private(i)
    for (i = 0; i < numberOfVectors; i++)
        arrayOfStrings[i] = (char*) calloc(dimension, sizeof(char));

    // Resever memory for matrix containing numbers vectors
    matrixOfNumbers = reserveMemoryMatrix(numberOfVectors, dimension);

    // Separate string each 'dimension' elements
    #pragma omp parallel for private(i,j) shared(arrayOfStrings, stringToSeparate,
dimension)
    for (i = 0; i < numberOfVectors; i++){
        for (j = 0; j < dimension; j++){
            arrayOfStrings[i][j] = stringToSeparate[(i*dimension)+j];
        }
    }

    // Convert each string to numbers according to the alphabet
```

```
    #pragma omp parallel for private(i) shared(matrixOfNumbers,arrayOfStrings, alphabet)
    for (i = 0; i < numberOfVectors; i++)
        matrixOfNumbers[i] = convertVectorToNumbers(arrayOfStrings[i], alphabet);


    return matrixOfNumbers;

}

void printFile(char* fileName, char* string){
    FILE * file = fopen(fileName, "w");
    if(file){
        fputs(string,file);
        fclose(file);
    }
}


int* convertVectorToNumbers(char* vector, char* alphabet){
    int* vectorOFNumbers = (int*) calloc(strlen(vector), sizeof(int));

    for (int i = 0; i < strlen(vector); i++){
        for (int j = 0; j < strlen(alphabet); j++){
            if (vector[i] == alphabet[j]){
                vectorOFNumbers[i] = j;
            }
        }
    }

    return vectorOFNumbers;
}

char* encryptVector(int** matrix, char * alphabet, int dimension, int numberOfVectors){
    char* ev = (char*) calloc(dimension*numberOfVectors,sizeof(char));
    for(int i=0; i<numberOfVectors; i++){
        ev = strcat(ev, convertNumbersToStrign(matrix[i],alphabet,dimension));
    }
    return ev;
}

char* convertNumbersToStrign(int *vector, char* alphabet, int dimension){
    char* str = (char*) calloc(dimension,sizeof(char));
    int pos;
    for(int i = 0 ; i<dimension ; i++){
        str[i] = alphabet[vector[i]];
    }
    return str;
}

int* multiplyVector(int** matrix, int *vector, int dimension, int module){
    //Reserving memory for new vector
```

```
    int *resultVector = (int*) calloc(dimension, sizeof(int));
    int i, j, tmp;

    //Iterating the matrix and multiplying by the vector pointer
    for(i=0; i < dimension; i++) {
        for(j=0; j < dimension; j++) {
            *(resultVector + i) += (matrix[i][j] * (*(vector + j)));
        }
        // Apply module
        *(resultVector+i) = *(resultVector+i) % module;
    }

    return resultVector;
}

void printMatrix(int** matrix, int rows, int columns){
    for (int i = 0; i < rows; i++){
        for (int j = 0; j < columns; j++){
            printf("%d ",matrix[i][j]);
        }
        printf("\n");
    }
}
```

a) Justificación del tipo de descomposición utilizado.

Se decidió usar Descomposición Funcional, ya que todo lo teniamos en funciones y una funcion dependia del resultado de otra y así consecutivamente. Al analizar nuestras funciones, observamos la siguiente distribución:



Entre las ventajas que encontramos al hacerlo así, es que nuestro programa podía hacer operaciones independientes al mismo tiempo, sin necesidad de estar esperando a que una operación que no tiene nada que ver con otra se complete y con esto podamos perder tiempo y rendimiento.

Para realizar el código se pensó en el manejo de excepciones, como lo es completar el texto a cifrar para que sea divisible entre el numero de la dimension de la matriz no singular, quitar espacios en blanco, etc. Entonces lo primero que se hace es pedir la dimensión al usuario, posteriormente se reserva memoria para la matriz y se piden los datos al usuario. Se genera el alfabeto, el cual es una funcion donde retorna un arreglo de caracteres, ahì podríamos modificarlo. Se lee el archivo donde viene el texto a cifrar y se checan los siguientes casos, que el texto no tenga espacios ni caracteres que no esten en el alfabeto, si el modulo de la longitud del texto con la dimension de la matriz no singular es diferente de 0, quiere decir que no esta completo, por lo tanto tenemos que completar el texto, en este caso se completo con ‘X’. Luego separamos el texto en vectores de numeros, cada vector tendrá la dimensión de la matriz. Por último calculamos la multiplicación entre la matriz y cada uno de los vectores. Al final solo queda convertir esos vectores resultado en texto mediante el alfabeto e imprimirlos en un archivo.

b) Tipo de operaciones utilizadas en la paralelización y Justificación del tipo de operación utilizada.

Entre las que se usaron fueron el for, private y share: estas se usaron para poder paralelizar los ciclos for y poder dividir el número de iteraciones entre el número de hilos.

```
#pragma omp parallel for shared(arrayOfStrings, dimension) private(i)
for (i = 0; i < numberOfVectors; i++)
    arrayOfStrings[i] = (char*) calloc(dimension, sizeof(char));
```

single, no wait: estas se ocuparon para que solo un hilo se ocupara de una función.

```
#pragma omp single nowait
{
    stringToEncrypt = readFile(argv[1]);
}
```

sections: se ocupa para poder utilizar descomposición funcional y hacer que varias funciones independientes entre sí se ejecuten al mismo tiempo.

```
#pragma omp parallel sections
{
    #pragma omp section
    fillMatrix(nonSingularMatrix, dimension);

    #pragma omp section
    {
        module = strlen(stringToEncrypt) % dimension;
        if(module != 0){
            // printf("Necesita completarse\n");
            int numberMissingCharacters = dimension - module;
            stringToEncrypt = completeText(stringToEncrypt, numberMissingCharacters);
        }
    }
}
```

c) Escalabilidad y Portabilidad del programa.

El programa sí es escalable, ya que hicimos todo genérico, como por ejemplo el módulo siempre será la longitud del alfabeto, la matriz y la dimensión de esta será elegida por el usuario y durante todo el programa se maneja eso.

⇒ **Análisis de los programas.**

a) Tiempo de ejecución del programa serial con diferentes tamaños del mensaje

tiempo	tamaño del texto
0.003491	25
0.006583	5580
0.096825	100614
1.080255	503110
4.01	1038397

b) Calculo de SpeedUp y gráficas

procesos	speedUp
2	0.07806527
50	0.16028066
100	0.24264062
150	0.4038599
200	0.66387279
500	1.77041989

c) Análisis de los resultados:

Observamos que el speed up va en aumento, aunque idealmente se debería de ver mas curva la gráfica

d) Manual de usuario:

Para compilar el programa se debe de usar gcc con la bandera -fopenmp de la siguiente manera:

gcc hill_paralelo.c -fopenmp -o hill_paralelo

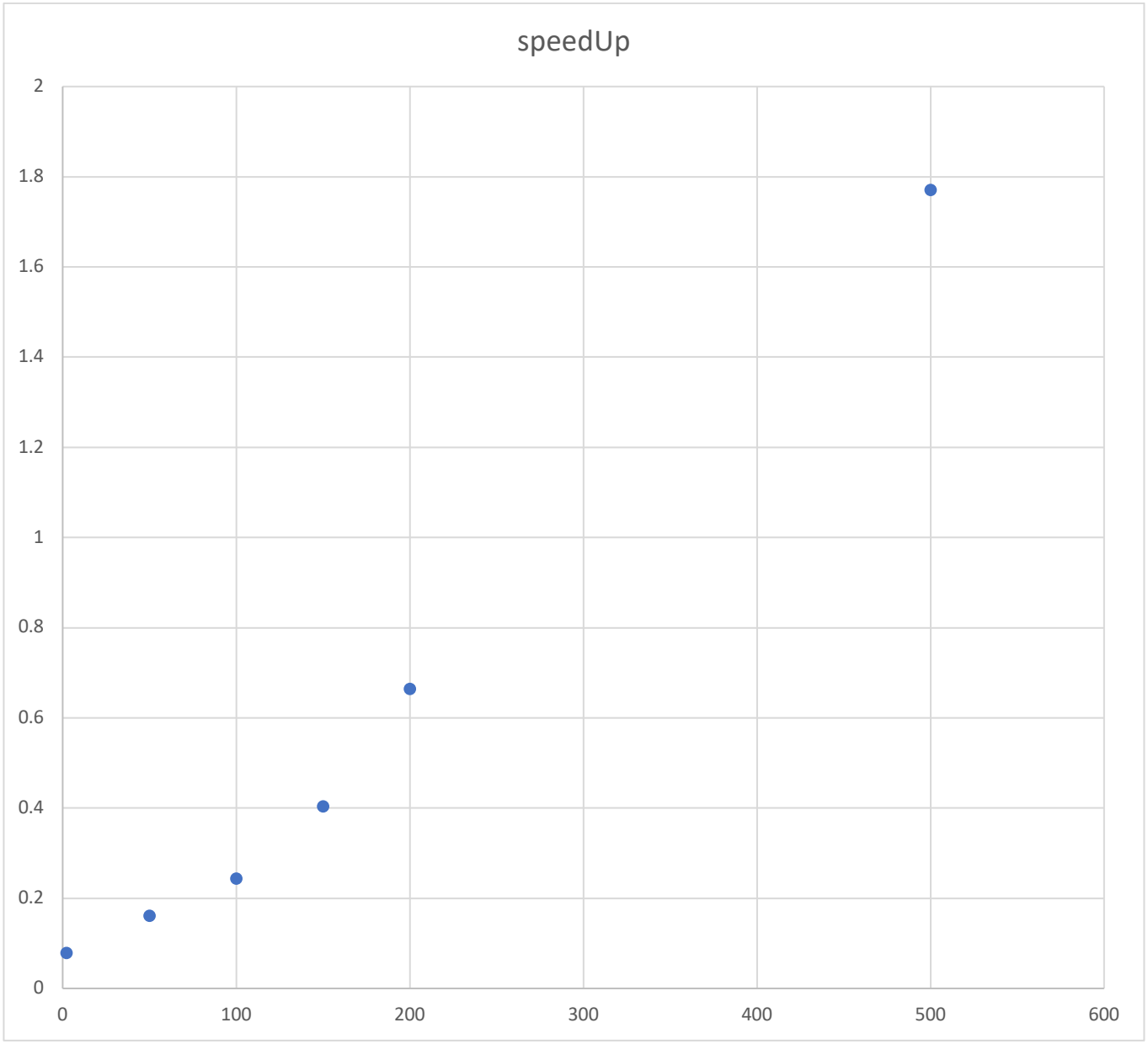
La bandera -o unicamente será para indicar el nombre del ejecutable.

Para ejecutar se debe de pasarle como argumento el nombre del archivo a cifrar:

./hill_paralelo archivo.txt

Para descifrar:
./hill_paralelo salida.txt

⇒ **Gráfica de Speed Up**



- **Conclusiones:**

Ricardo: La realización de este proyecto la encontramos un poco complicado, ya que a veces queriamos paralelizar cosas y no funcionaba o tronaba el programa. En general estuvo interesante hacer este código en serie y posteriormente pasarlo a paralelo. A pesar de encontrarnos con varios problemas al momento de paralelizar, creemos que nuestra solución es correcta, tal vez y pueda optimizarce un poco más pero sería cosa de checar a fondo el código.

Ferrusca: La implementación de este programa considero que fue más extensa que la del proyecto en MPI, dado que si bien no era tampoco tan complejo, sí debiamos tener especial cuidado en las operaciones con vectores y matrices. En lo que respecta a la programación paralela, me parece que openMP provee una interfaz más sencilla a la hora de paralelizar los programas respecto de openMPI, los cuales si bien son diferentes y se usan en distintos casos, creo que el manejo de openMP fue un poco más sencillo.

Bruno: este proyecto tuvo varias complejidades, uno que tuvimos al momento de crear el programa serie, eran problemas con la memoria, ya que nos marcaba violaciones al segmento, e igual al momento de paralelizar el programa, tuvimos dificultades al dividir los ciclos, nos topamos con problemas de que no se guardaban los datos en las posiciones correctas. En este proyecto se notó una mejoría en el rendimiento del programa en paralelo con respecto al programa serie, que para un texto en un archivo de 2 megas, tardaba aproximadamente 5 segundos.

- **Referencias:**

Marcela Guzmán Caicedo y Felipe Epia Realpe. (2016). OpenMP. 20/11/2019, de Universidad Nacional de Colombia Sitio web: http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/tutoriales/openmp/index.html

Javier Cuenca. (2015). Programación con OpenMP. 20/11/2019, de Facultad de Informática. Universidad de Murcia Sitio web: http://www.ditec.um.es/~javiercm/curso_psba/sesion_03_openmp/PSBA_OpenMP.pdf

Creado por: Contreras M. Daniel, Flores F. Armando y Reséndiz J. Omar. (2011). Hill. 20/11/2019, de UNAM Sitio web: https://unamcriptografia.wordpress.com/2011/10/05/hill/