DESCRIPCION
===========

Implementación de algoritmos de desencriptación por fuerza bruta, para textos cifrados con los algoritmos blowfish o cast5 de la librería OpenSSL.

La desencriptación se realiza de 3 maneras: de forma serial, a travésés de hilos y memoria compartida utilizando librería OpenMP, y travéses de procesos y pasos de mensajes con la librería OpenMPI

PREREQUISITOS
=============

* openmpi-1.6.5: Open MPI lib
* openssl-1.0.1e: Open SSL crypto lib
* CUnit-2.1-2: C Unit Testing lib

nota: libopenmp viene incluida por defecto en el compilador gcc

INSTALACION
===========

* instalación usando make

```
shell$ make install
```

* instalación manual

Descomprimir todas las librerías ubicadas en el directorio libs

```
shell$ tar -zxvf CUnit-2.1-2.tar.gz
shell$ tar -zxvf openmpi-1.6.5.tar.gz
shell$ tar -zxvf openssl-1.0.1e.tar.gz
```

* openmpi-1.6.5:

```
shell$ cd libs/openmpi-1.6.5
shell$ ./configure
[...lots of output...]
shell$ sudo make all install
```

* CUnit-2.1-2:

```
shell$ cd libs/CUnit-2.1-2
shell$ ./configure
shell$ make
```

```
     shell$ make check
     shell$ sudo make install

* openssl-1.0.1e:

     shell$ cd libs/openssl-1.0.1
     shell$ ./config --prefix=/usr/local
     shell$ make
     shell$ make test
     shell$ sudo make install

nota: en caso de error, leer los archivos INSTALL ubicados en los
directorios de las librerías, e intentar reinstalar

COMPILACION
===========

* para compilar los códigos fuentes:

     shell$ make all

* se generan 5 ejecutables en la carpeta /bin

     * serial: la implementación serial de el algoritmo de fuerza
       bruta
     * omp: la implementación en hilos y memoria usando OpenMP
     * mpi: la implementación en procesos y paso de mensajes usando
       OpenMPI

     * encrypt: encriptador de archivos
     * decrypt: desencriptador de archivos

     * unit-test: test unitarios de los componentes del algoritmo

TEST
====

* para ejecutar los test:

     ## ejecutar todos los test
     shell$ make test

     ## únicamente test unitarios
     shell$ make test-unit

     ## únicamente test de aplicación
     shell$ make test-app
```

```
     ## únicamente test de utilidades encrypt y decrypt
     shell$ make test-utils

     nota: todos los test deberían terminar en "passed".


VARIABLES DE AMBIENTE
=====================

* CANT_KEYS:
     Cantidad de claves para utilizar en la desencriptación por
     fuerza bruta. Por defecto, si no se configura la cantidad de
     claves, se utilizan 10^8 claves en la desencriptación.

          shell$ export CANT_KEYS=[ cantidad de claves ]

* OMP_NUM_THREADS:
     Cantidad de hilos a utilizar por la implementación en OpenMP.
     Por defecto, si no se configura la cantidad de hilos, se
     utilizan 4 hilos
     para la desencriptación usando OpenMP. Es adecuado configurar la
     cantidad de hilos a utilizar igual al numero de cores físicos
     que posee el microprocesador, si se utiliza un numero mayor, se
     obtienen peores resultados de performance. Un numero menor no
     aprovecha todas las capacidades del microprocesador.

     shell$ export OMP_NUM_THREADS=[ cantidad de hilos ]

EJECUCION
=========

* encriptador
     El programa "encrypt" encripta un archivo, con una clave y un
     método determinado, y guarda el texto encryptado en un archivo
     de salida. Todos los parámetros son obligatorios. Las claves
     deben ser numéricas mayores a cero y menores a CANT_KEYS.

     uso: ./encrypt [INPUT FILE] [KEY CODE] [METHOD] [OUTPUT FILE]

     INPUT FILE: archivo para encriptar
     KEY CODE: clave numérica de encriptación ( entre 1 y CANT_KEYS )
     METHOD: método de cifrado ( blowfish o cast5 )
     OUTPUT FILE: archivo de salida

     Retorna 0 si la encriptación tuvo éxito y se genera el archivo
     de salida. En caso de error, retorna un valor distinto de cero,
     y se imprime en stdout la causa del error.
```

* desencriptador

El programa "decrypt" desencripta un archivo, con una clave y método determinado, y guarda el texto desencriptado en un archivo de salida. Todos los parámetros son obligatorios. Las claves deben ser numéricas mayores a cero y menores a CANT_KEYS.

uso: ./decrypt [INPUT FILE] [KEY CODE] [METHOD] [OUTPUT FILE]

INPUT FILE: archivo para desencriptar
KEY CODE: clave numérica de desencriptación
METHOD: método de cifrado (blowfish o cast5)
OUTPUT FILE: archivo de salida

Retorna 0 si la encriptación tuvo éxito y se genera el archivo de salida.
En caso de error, retorna un valor distinto de cero, y se imprime en stdout la causa del error.

* algoritmos de fuerza bruta

Los programas "serial", "omp", "mpi", intentan desencriptar un archivo con todas las claves disponibles. Cada uno implementa una forma diferente de dividir el trabajo a realizar.

uso: ./serial ENCRYPTED_FILE
uso: ./omp ENCRYPTED_FILE
uso: mpirun [-np NUM_PROCESS] mpi ENCRYPTED_FILE

ENCRYPTED_FILE: archivo encriptado
NUM_PROCES: procesos utilizados por la implementación mpi

Estos archivos desencriptan el primer bloque (8 bytes) del archivo de entrada,
y buscan la palabra "Frase" en los primeros 5 caracteres.

La implementación serial prueba las claves una a la vez. en caso de tener éxito detiene las iteraciones
La implementación en hilos divide el trabajo entre los hilos creados usando memoria compartida.
La implementación en procesos divide el trabajo entre los procesos disponibles usando paso de mensajes.

En caso de éxito, retornan 0, y se generan 2 archivos de salida:

* report: contiene información del resultado (clave, método de cifrado, tiempo aproximado)
* key: contiene la clave que tuvo éxito en la desencriptación.

De no tener éxito, retornan un valor distinto de 0

DESARROLLO DEL PROYECTO
==========

El desarrollo del proyecto empieza analizando los requerimientos del mismo, en el cual se pedía la utilización de la librería OpenSSL para desencriptar por fuerza bruta un archivo. El mismo estaba encriptado con los method Blowfish o Cast5, con una clave de 16 bytes con una sintaxis predeterminada. La desencriptación ddebía implementarse de forma secuencial o en paralelo, usando las librerías OpenMP y OpenMPI.

El primer paso fue explorar las librerías a utilizar. Para ello, se descargaron e instalaron las librerías
y se estudio su funcionamiento, y se realizaron códigos ejemplos simples para probar su funcionamiento. Estos ejemplos pueden encontrarse en el directorio examples, donde se encuentran ejemplos de uso de la libcrypto de OpenSSL, OpenMP y OpenMPI. También se estudio la forma de compilar y ejecutar estos programas. Durante esta actividad de investigación, fueron almacenándose las documentaciones consultadas. Las mismas se encuentran en el directorio doc.

Una vez comprendida las herramientas a utilizar, se definió la infraestructura a utilizar para el desarrollo del proyecto. En primera instancia, se definió la estructura del proyecto en directorios:

```
/
----/bin            // Todos los ejecutables
----/obj            // Todos los codigos objetos
----/doc            // Documentación
----/examples       // Códigos de ejemplo
----/input          // Archivos de entrada para los algoritmos
----/output         // Archivos de salida de los algoritmos
----/libs           // Librerías utilizadas
----/scripts        // Bash Scripts
----/include        // Todos los archivos de cabecera
----/src            // Todos los códigos fuente
```

Una vez definida la estructura del proyecto, se decidió utilizar una herramienta de versionamiento de código. Se utilizo Mercurial, y se creo un repositorio local para el proyecto. Esto facilito la recuperación de archivos en caso de errores durante el desarrollo. Se puede visualizar el desarrollo total del proyecto usando:

```
shell$ hg history
```

Luego de un análisis mas detallado de los requerimientos, se define una primera arquitectura del proyecto. Se identificaron actividades comunes a todas las implementaciones:

* Lectura de parámetros, variables de ambiente y generación de resultados
* Lectura/Escritura de archivos del File System.
* Generación de claves utilizadas en la encriptación/desencriptación
* Ejecución de la encriptación/desencriptación

La arquitectura se dividió en componentes, los cuales implementan las funciones que realizan las actividades comunes. La división en componentes permitió simplificar el desarrollo y evolución del software, permitiendo la reutilización de código. Los componentes definidos fueron los siguientes:

* commons: provee funciones de inicialización de variables y generación de resultados
* keygen: provee funciones para la generación de claves de 16 bytes
* fs: provee funciones para la lectura/escritura de archivos
* encryptor: provee funciones para la encriptación/desencriptación con blowfish y cast5

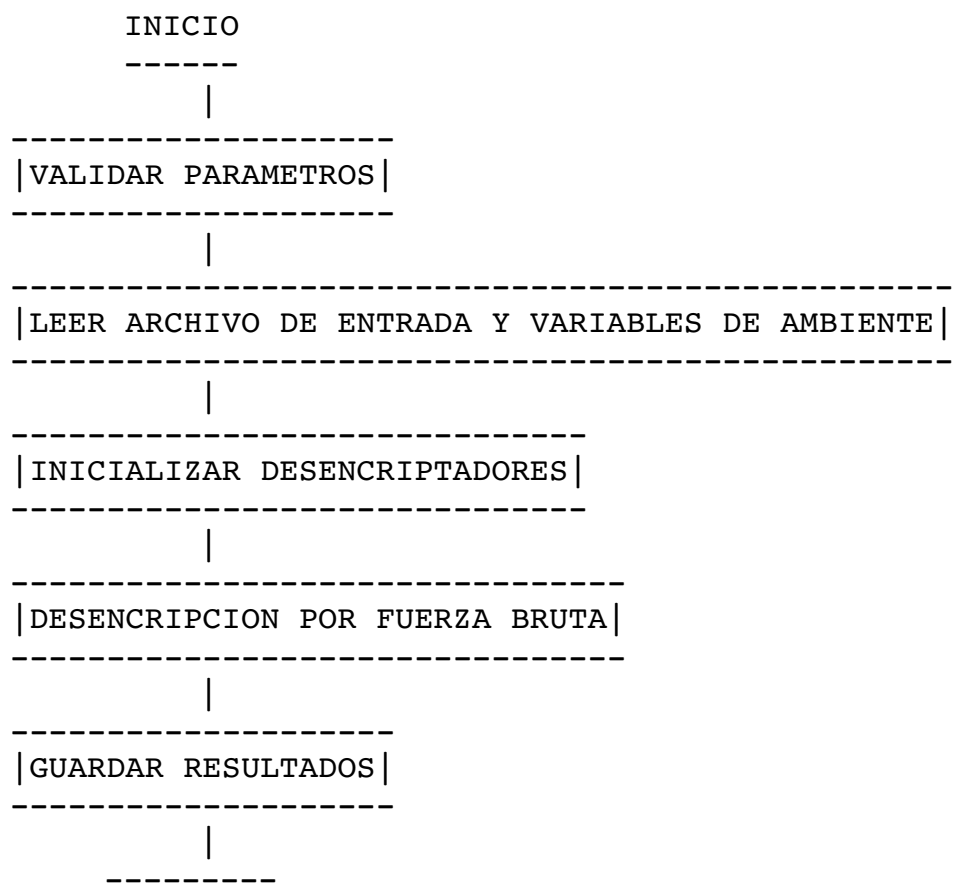
Para el desarrollo de los componentes, se decidió utilizar la técnica TDD (Test Driven Development), para garantizar el funcionamiento unitario de cada componente y mejorar la calidad final del software. Por ello, se uso la librería CUnit para la creación y ejecución de test unitarios de componentes. Cada componente fue testeado individualmente durante su desarrollo antes de realizar la integración de los mismos en una sola aplicación.

Para la compilación del código y ejecución de tests, se utilizo la herramienta make, lo cual facilita la ejecución de tareas repetitivas y de limpieza. Para ello, se genero un Makefile que se encuentra en la carpeta raíz del proyecto.

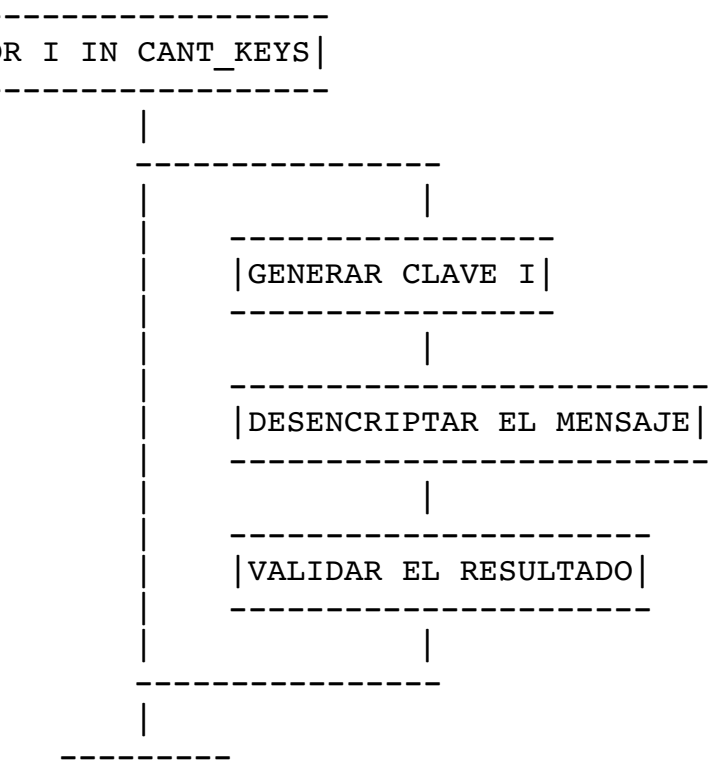
Una vez finalizado el desarrollo de los componentes requeridos, se paso a la integración de los mismos. Para ello, se desarrollaron dos utilidades: encrypt y decrypt. Estos usan los componentes desarrollados para realizar la encriptación y desencriptación de un archivo. Se desarrollo un test en un script de bash para validar si el funcionamiento era correcto. El test encripta un archivo con una clave y método determinado, para luego desencriptarlo con la misma clave, y valida si el archivo desencriptado es igual al archivo de entrada original.

En esta instancia, se detectaron algunos problemas de funcionamiento. Por ello, se modificaron y extendieron algunos componentes. Estas utilidades luego se utilizaron para testear el funcionamiento de las aplicaciones principales.

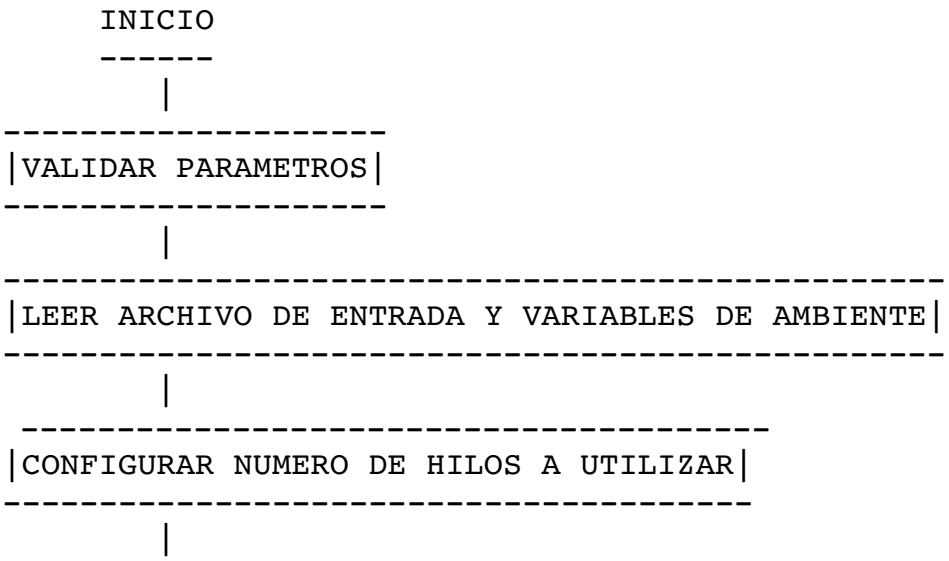
A continuación, se desarrollo la primera implementación del algoritmo de fuerza bruta, la cual realiza la desencriptación de forma secuencial. El flujo de la implementación serial es el siguiente:

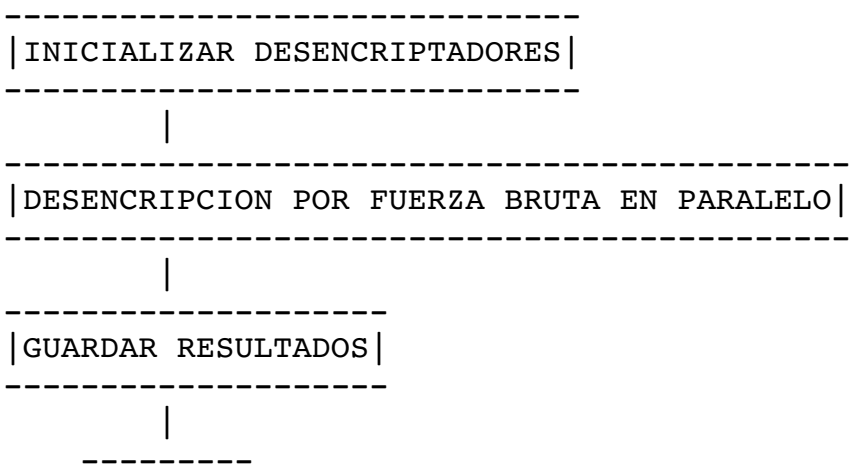


La desencriptación por fuerza bruta se realiza de manera iterativa, generando y probando las claves una a la vez,
en caso de obtener un resultado favorable, el ciclo se detiene:

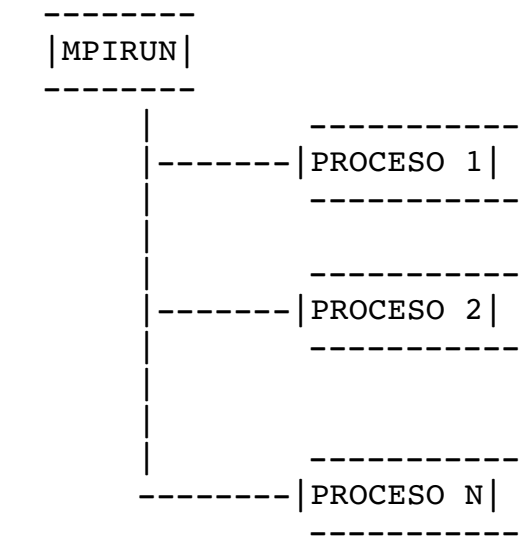


La implementación con OpenMP mediante hilos y memoria compartida, tiene el mismo flujo, pero se agrega una etapa
en donde se configura la cantidad de hilos a utilizar en la región paralela y se calcula el trabajo a realizar por
los hilos participantes. El trabajo a realizar se divide automáticamente entre los hilos, y se utiliza un
parallel for para ejecutar la desencriptación por fuerza bruta:

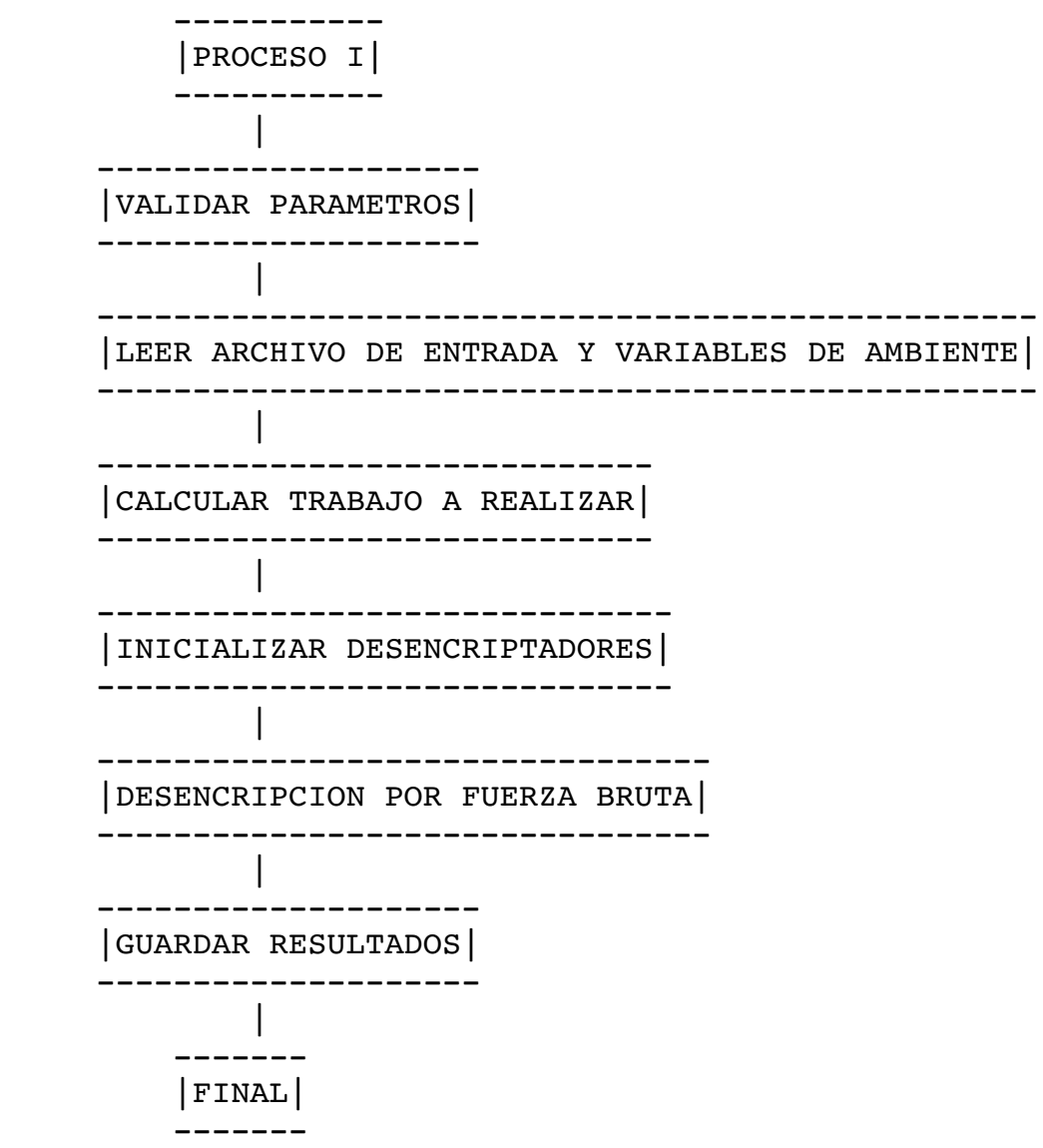




La implementación con OpenMPI mediante procesos y paso de mensajes realiza el trabajo de forma diferente. En esta,
la utilidad mpirun es la encargada de crear los procesos que van a ejecutar el trabajo. Cada proceso es una copia del
programa original. Cada uno selecciona una parte del trabajo a realizar y lo ejecuta de la misma manera que en la
implementación secuencial. Al finalizar, el proceso que tuvo éxito es el encargado de generar los resultados. La
cantidad de trabajo a realizar por cada proceso se divide de manera equitativa entre todo los procesos disponibles.
El flujo del programa es el siguiente:



Cada proceso iniciado por mpirun realiza el trabajo de la misma manera que sus procesos hermanos, seleccionando
su porción de trabajo a través del id de procesos.



TIEMPOS DE EJECUCION
==========

Para comparar la performance de las diferentes implementaciones de los algoritmos de
fuerza bruta, se realizo una medición del tiempo (en segundos) ocupado por cada algoritmo, variando
la cantidad de claves utilizadas en la desencriptación. Los datos obtenidos se presentan
en la siguiente tabla:

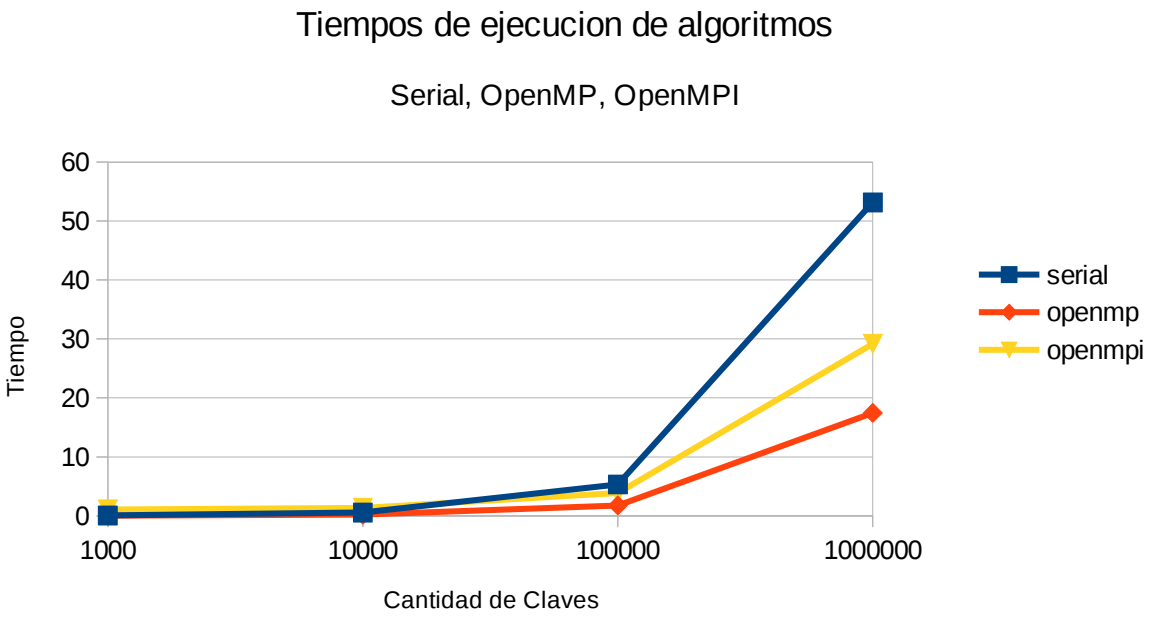
Tiempos de Ejecución			
claves	serial	openMP	openMPI
1000	0.061	0.024	1.072
10000	0.539	0.198	1.339
100000	5.321	1.75	3.864
1000000	53.175	17.456	29.167

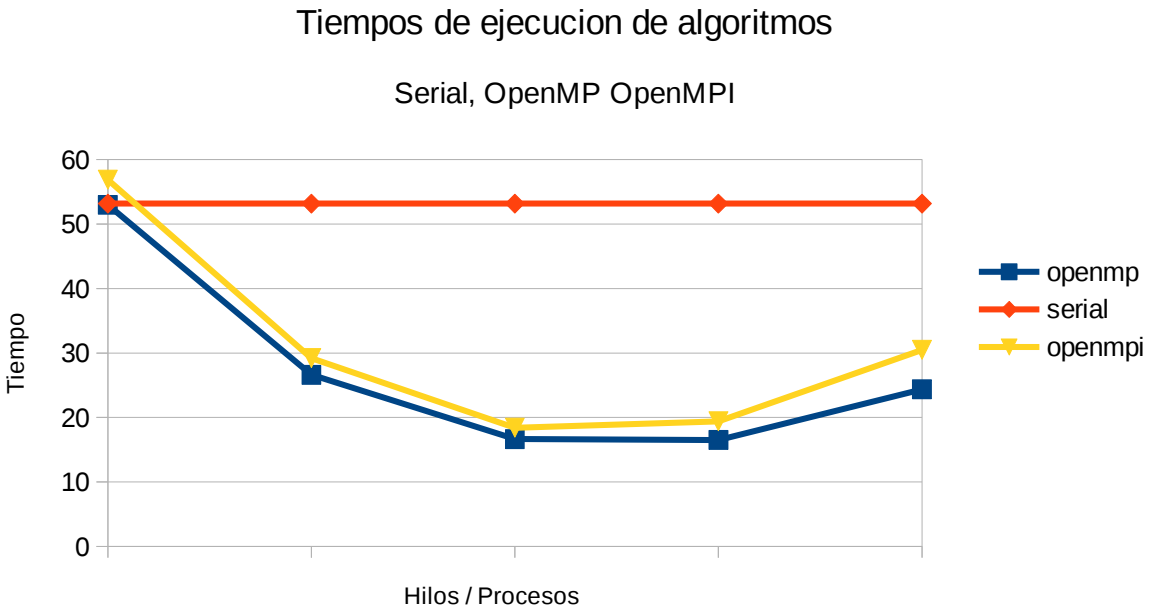
Queda evidente que con un numero mayor de claves, la implementación serial tiene un peor desempeño, comparado con las implementaciones en paralelo.

En la siguiente tabla, se muestra como varían los tiempos de ejecución de los algoritmos, a medida que se aumenta la cantidad de trabajadores disponibles. En el caso de la implementación serial, siempre hay un trabajador. Para OpenMP los trabajadores son los hilos que ejecutan la sección paralela del código, y para OpenMPI los trabajadores son los procesos utilizados para la ejecución.

Tiempos de Ejecución			
hilos/procesos	openMP	serial	openMPI
1	52.977	53.175	56.866
2	26.611	53.175	29.194
4	16.646	53.175	18.396
8	16.501	53.175	19.398
16	24.372	53.175	30.479

El mejor desempeño se logra utilizando una cantidad de trabajadores equivalente a la cantidad de núcleos disponibles en el microprocesador. Al utilizar una cantidad superior, los tiempos de ejecución aumentan.


Tiempos de ejecucion de algoritmos
Serial, OpenMP, OpenMPI
60
50
40
30
20
10
0
Tiempo
1000
10000
100000
1000000
Cantidad de Claves
serial
openmp
openmpi


Tiempos de ejecucion de algoritmos
Serial, OpenMP OpenMPI
60
50
40
30
20
10
0
Tiempo
Hilos / Procesos
openmp
serial
openmpi

PRUEBA REAL
==========

Se midió el tiempo ocupado para desencriptar uno de los archivos brindados para pruebas. Se utilizo la implementación con OpenMP y OpenMPI para probar la totalidad de las claves, utilizando 4 hilos de ejecución y 4 procesos, respectivamente. Los resultados son los siguientes:

* Desencriptación usando OpenMP
    real   26m55.728s
    user   107m41.000s
    sys 0m0.716s

* Desencriptación usando MPI
    real   28m39.777s
    user   113m37.130s
    sys 0m3.668s

RESULTADOS
==========

* Oliver1:
    Frase: Nunca falta alguien que sobra.
    Método: blowfish.
    Clave: 46712343.

* Oliver2:
    Frase: El saber no ocupa lugar.
    Método: cast5.
    Clave: 49999913.

* Navarro1:
    Frase: Quien mal anda, mal acaba.
    Método: blowfish.
    Clave: 99921223.

* Navarro2:
    Frase: En el reino del revés, los gatos no dicen miau, dicen yes
           porque estudian mucho ingles.
    Método: cast5.
    Clave: 42904536.

CONCLUSIONES

- Organización del proyecto

  El analisis de los requerimientos, la investigacion de las herramientas a utilizar, la definicion de una arquitectura y estructura del proyecto de desarrollo ayudan en la construccion del software

- Uso de Herramientas

  El uso de herramientas para la automatizacion de tareas como ser testing, compilacion, ejecucion y versionamiento implican un costo inicial, pero facilitan la evolucion del desarrollo y permiten ahorrar tiempo.

- Uso de Librerias

  En comparacion con el uso de hilos a traves de pthread y precesos con forks, la utilizacion de estas librerias simplifican el trabajo a realizar y disminuyen la posibilidad de errores, aunque se pierde un poco el control que se logra utilizando las librerias pthread y fork/

- Optimización de Recursos

  Maximizar la utilización de los recursos de hardware disponibles a través de programas paralelos implica una disminución en los tiempos de ejecución y un aumento en la performance de las aplicaciones