Universidad Autónoma Metropolitana
Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería
Licenciatura en Ingeniería en Computación

Reporte de proyecto tecnológico:

Implementación paralela del algoritmo para compendios de mensaje MD6

Que presenta:
José Alejandro Avilés Jiménez
204200337

Asesor:
M. en C. Oscar Alvarado Nava

Trimestre Lectivo:
14 I

México D.F. Abril 2014

Declaratoria

Yo, Oscar Alvarado Nava*, declaro que aprobé el contenido del presente Reporte de Proyecto de Integración* y *doy mi autorización para su* publicación en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Yo, José Alejandro Avilés Jiménez*, doy mi autorización* a la Coordinación de Servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para publicar el presente documento en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Resumen

En este documento se presenta la investigación, análisis, desarrollo y observaciones sobre el proyecto.

La investigación cuenta con el análisis y fundamentos de la propuesta del profesor investigador del MIT Ronald Rivest"*The MD6 Hash Algorithm*", que fue un candidato para la competencia NIST (*National Institute of Standards and Technology*) del nuevo algoritmo SHA3.

Una vez realizado la investigación y teniendo bases suficientes se desarrolló la sección del marco teórico indicando el funcionamiento y proceso por el que debe desarrollarse, cuenta con descripción funcional así como de notación técnica e imágenes que ilustran partes fundamentales del algoritmo dando el panorama a detalle sobe el proceso.

Dentro de la sección de pruebas y conclusiones muestra los resultados obtenidos así como de observaciones del algoritmo sobre le desempeño de este mismo pudiendo crear un análisis e identificar tendencias sobre rendimiento al incrementar el número de procesadores trabajando simultáneamente en el proceso.

También encontrara secciones donde se describe la instalación del proceso así como de un manual de cómo utilizarlo.

Tabla de Contenido.

Antecedentes

La criptografía es el arte o ciencia de cifrar y descifrar información mediante técnicas especiales y se emplea frecuentemente para permitir un intercambio de mensajes que solo puedan ser leídos por personas a las que van dirigidos y que poseen los medios para descifrarlos.

En una definición más moderna según Katz & Lindell nos mencionan que la criptografía es el estudio científico de técnicas para asegurar la información digital, las transacciones, y computación distribuida que puedan entrar en ataque interno o externo.

Históricamente, los principales consumidores de la criptografía eran militares y los organismos de inteligencia. Hoy, sin embargo, la criptografía está en todas partes. Los mecanismos de seguridad que se basan en la criptografía es una parte integral del sistema de casi cualquier computadora.

Funciones hash resistentes a colisión.

Una *función hash criptográfica* es un procedimiento determinista que toma un conjunto de bloques de datos y devuelve una cadena de bits de tamaño fijo (*valor hash criptográfico*), de forma que un cambio accidental o intencional de los datos va a cambiar el valor de hash. Los datos que se codifican se llama el "*mensaje*", y el valor de salida hash a veces se llama el resumen del mensaje o simplemente digesto.

Una *función hash criptográfica* ideal tiene 4 propiedades principales

1. El *valor hash* es fácil de calcular para cualquier mensaje.
2. No es factible encontrar el mensaje original a partir del *valor hash*.
3. No es factible modificar un mensaje y que su *valor hash* no cambie.
4. No es factible encontrar dos mensajes diferentes con el mismo *valor hash*.

Una función de digesto o "*hash*" es un algoritmo o programa que se utiliza como herramienta para dotar a un documento digital de integridad [2], esto es, poder detectar cualquier alteración posterior a su firma. Dicho algoritmo produce un número en función del documento, que es un resumen del mismo.
Si el algoritmo se aplica siempre al mismo documento produce el mismo número en cuestión, pero si el documento variase en tan solo un bit de información el número producido va a diferir completamente del anterior. El numero producido por este programa entonces determina al documento en forma univoca.

Estos algoritmos se encuentran categorizados dentro de una clase de funciones que se conocen como funciones de un solo sentido (*One way function*), ya que dado un documento es posible obtener siempre su digesto pero, por el contrario, es prácticamente imposible

deducir el documento original a partir del valor del digesto. Esto permite garantizar la unicidad e integridad del mensaje que se está por firmar.
Algunos algoritmos existentes sobre funciones hash criptográficas son:

- MD2 (*Message Digest Algorithm 2*) desarrollado por Ronald Rivest en 1989.
- MD4 (*Message-Digest Algorithm 4*) desarrollado por el Profesor Ronald Rivest del MIT en 1990
- MD5 (*Message-Digest Algorithm 5*) desarrollado por el Profesor Ronald Rivest del MIT en 1991.
- SHA1 (*Secure Hash Algorithm 1*) diseñado por la NSA(*National Security Agency*) y publicada por el NIST(*National Institute of Standards and Technology*) en el año de 1993.
- SHA-2 (*Secure Hash Algorithm 2*) desarrollado por el NIST(*National Institute of Standards and Technology*) en su primera publicación en el año 2001

Justificación

Actualmente la criptografía tiene muchas aplicaciones, por ejemplo, se utilizan en firmas digitales, *métodos de timestamping*, verificación de integridad de archivos.

El panorama para el diseño de las *funciones de hash* sin duda ha cambiado en las últimas dos décadas. Los avances en la tecnología han proporcionado menos restricciones, y más oportunidades. Los avances teóricos más recientes proporcionan herramientas tanto para el atacante como para el diseñador, de tal forma que el ataque hacia estos algoritmos se ve más vulnerable ya que se cuenta tanto con la tecnología como con el conocimiento para poder romper este tipo de cifrado. Lo que conlleva al diseñador a crear algoritmos más seguros y efectivos.

La principal diferencia de los sistemas criptográficos modernos respecto a los clásicos está en que su seguridad no se basa en el secreto del sistema, sino en la robustez de sus **operadores** (algoritmos empleados) y sus **protocolos** (forma de usar los operadores).

Al crear algoritmos más seguros muchas veces lleva a más tiempo de procesamiento y como el tiempo de ejecución de este tipo de algoritmos es proporcional al tamaño del mensaje o archivo mientras más grande sea el mensaje más tiempo de procesamiento necesitara, es ahí donde entra el análisis de ingeniería para crear algoritmos más rápidos.
El algoritmo a implementar en su descripción natural tiene la estructura de ser paralelizable lo que conlleva a poder implementarlo en sistemas que proveen de cómputo paralelo como lo soy hoy en día los sistemas *multi-core*, o bien en *clusters*.

Actualmente la mayoría de las empresas realizan *backups* de su información continuamente; ya que es información confidencial se exige el aseguramiento de la integridad de esta información; dado que es de grandes cantidades el procesamiento en este tipo algoritmos tarda mucho; con la implementación de este algoritmo de forma paralela reduciría considerablemente el tiempo de ejecución, generando una firma en menos tiempo y con mayor seguridad.

La propuesta del algoritmo a implementar fue presentada por Ronald Rivest Profesor Investigador del MIT en su última publicación en febrero del 2009 en su propuesta "*The MD6 Hash Algorithm*", que fue un candidato para la competencia NIST (*National Institute of Standards and Technology*) del nuevo algoritmo SHA3.

Es un algoritmo novedoso y actual en su género que promete mucho por lo cual el solo hecho de implementarlo y optimizarlo en base a las arquitecturas actuales es un gran reto por lo que requiere de tener buenos fundamentos en matemáticas y de conocimientos en disciplinas como lo son arquitectura de computadoras, sistemas operativos y distribuidos, análisis y diseño de algoritmos.
Por lo cual se requiere de un ingeniero en computación para poder desarrollar el proyecto.

Introducción.

MD6 pos sus siglas en ingles *Message-DigestAlgoritm 6* es una función hash criptográfica. Utiliza una estructura definida "*Merkletreelike*" que permite el procesamiento paralelo de funciones para entradas de gran tamaño.

El diseño de "*Merkletree*" se basa en las afirmaciones que hace Intel donde describe que el futuro de los procesadores tendrá decenas de núcleos. Con esta tendencia que marca Intel estas estructuras han de explotar el hardware y tener un mejor desempeño.

Para tener una idea clara se definirán se describe brevemente los siguientes conceptos

Función Hash Criptográfica.

Una *función hash criptográfica* es un procedimiento determinístico que toma un bloque arbitrario de datos y devuelve una cadena de bits de longitud fija, el valor hash(cifrado), de manera que un cambio intencional o accidental de los datos cambiara el valor hash. Los datos que se codifican se le denomina "*mensaje"* y el valor hash obtenido se nombra "*resumen de mensaje"* o "*resumen*".

Una función hash criptografía ideal tiene 4 propiedades principales:

- Es fácil calcular el valor hash para cualquier mensaje.
- No es viable encontrar el mensaje original partir de su resumen.
- No es factible modificar un mensaje sin cambiar su resumen.
- No es factible encontrar dos mensajes diferentes con el mismo valor hash.

Las funciones hash criptográficas tienen mucha aplicaciones dentro de la seguridad informática, en particular en firmas digitales, códigos de autenticación de mensaje MAC, y otra formas de autenticación. También son utilizadas como funciones hash ordinarias a los datos para generar algún índices en tablas, para huellas digitales, pata detectar duplicación de datos o archivos, para comprobar la integridad de los archivos ente otros.

Merkletreelike

En criptografía es un tipo de estructura de datos que contiene un árbol de resumen de información sobre un bloque de datos más grande, por ejemplo un archivo usado para verificar su contenido. Los arboles hash son una extensión de las listas hash.

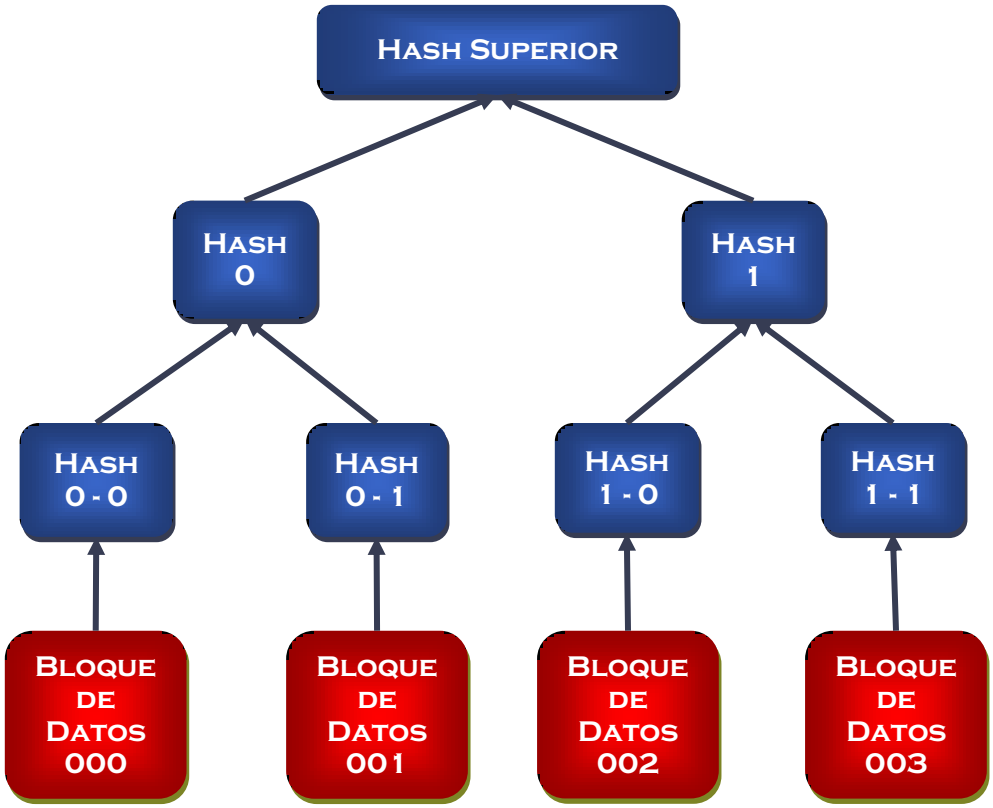


Ilustración 1.1. Árbol MerkleTree

La estructura de árbol permite asignar bloques de datos en los nodos inferiores del árbol los cuales serán calculado paralelamente, el cálculo se hace de abajo hacia arriba en donde el nodo raíz contendrá el valor hash final.
Existen algunas ventajas de este tipo de estructura como son la de ir calculando valores hash parciales mientras que otros árboles son cargados con los bloques de datos, otra ventaja es que si existe algún error en un cálculo por mala lectura se vuelve a calcular solo es parte del archivo, entre otras ventajas.

Descripción General Del Algoritmo MD6.

En esta parte del proyecto se describe y analiza las partes fundamentales del proyecto como los son términos a manejar y los bloques que conforman dicho algoritmo.

Características importantes del algoritmo son:

- Acepta entradas de mensaje de tamaño hasta 2^{64} -1 bits y produce una salida "resúmenes" de 1 a 512 bits.
- MD6 en su descripción natural tiene la característica de poderse implementar en paralelo, por lo que tendrá un buen desempeño bajo sistemas multi-core ejecutándose de manera concurrente.
- MD6 utiliza una función de compresión única, sin importar el tamaño del mensaje a "*digerir*", asigna bloques de datos de entrada de 4096 bits generando una salida parcial de bloques de 1024 bits haciendo una reducción de 4 veces el tamaño del bloque.
- El modo de operación estándar es basado en arboles, los datos entre la hojas son en un árbol de 4 nodos, y el valor *hash*es calculado en la raíz.
  Este modo de operación es altamente paralelizable.
- Dado que el modo estándar de MD6 requiere almacenamiento proporcional a la altura del árbol, hay un modo alternativo de almacenamiento variante al ajustar el parámetro opcional "L" que disminuye tanto los requerimientos de almacenamiento así también el paralelizamiento.
  Ajuste L=0 resultando en un modo de operación secuencial.
- Todos los valores intermedios pasados sobre el árbol son de 1024 bits; el valor final se obtiene de un truncamiento de la salida final de la función de compresión de la longitud deseada.
- La función de comprensión puede ser vista como una encriptación con una llave fija (o equivalentemente, como la aplicación de una permutación fija aleatoria del espacio de mensajes).

Especificaciones de MD6.

Esta sección provee las especificaciones detalladas del algoritmo MD6, tales como:

- Notación que utilizaremos en la descripción.
- Descripción de entradas y modificadores del algoritmo.
- Descripción de la salida.
- Modo de operación de cada uno de los bloques funcionales del algoritmo.

Notación.

Sea w el tamaño de la palabra, en bits, MD6 es definida en términos de la palabra por defecto w = 64 bits. Sin embargo, este diseño soporta implementaciones eficientes utilizando otros tamaños de palabra.

> En este documento una palabra siempre referirá a 64-bits (8-bytes). $w = 64$ bits

Definimos a W como el conjunto $\{0,1\}^w$, para todas la palabras *w-bits.*

Si A (o cualquier otra letra mayúscula) denota un arreglo de información. Entonces a (minúscula) denota el *i-esimo* elemento de A. Con excepción de la letra "W" que ya se definió en anteriormente.Definimos que tanto A*[i]* como A_i denotan el i-esimo elemento de A.Usaremos *'0'* como el índice origen de los arreglos.
La notación A[*i*...*j*] (o A*i..j*) denotan un *sub-arreglo de A* incluyendo los índices *i* y *j*.

MD6 se define en el sentido *big-endian,* indicando que el byte más significativo es el que se encuentra en la dirección más baja.

Operadores utilizados en el algoritmo son:

$\oplus$enota la operación "XOR" bit a bite entre palabras.

$\wedge$Denota la operación "AND" bit a bite entre palabras.

$\vee$Denota la operación "OR" bit a bite entre palabras.

$\neg x$Denota la negación bit abit de la palabra *x*.

$x<<b$La palabra *x* es desplazada b bits hacia la izquierda (entrando ceros).

$x>>>b$La palabra *x* es desplazada b bits hacia la derecha (entrando ceros).

$x<<<b$La palabra *x* es rotada b bits hacia la izquierda.

$x>>>b$La palabra *x* es rotada b bits hacia la derecha.

|| : Denota concatenación.

0x...: Denota una constante hexadecimal.

Entradas de MD6.

Esta sección describe las entradas del algoritmo *MD6* tanto obligatorias como opcionales.

M-El mensaje o archivo a ser *cifrado*. (Obligatorio)

d- Longitud del resumen del mensaje, en bits.

K- valor de la llave.

L- Modo de control.

r- Numero de rondas.

Solo el mensaje *"M"*es obligatorio, las demás variables son opcionales y tienen un valor por *default* si no se les asigna otro.

Denotemos a *"H"* como la *función hash MD6;* subíndicespueden ser tomados como parámetros de la función.

Mensaje *M* a ser cifrado.

Está primer entrada obligatoria para MD6 es el mensaje o archivo a ser resumido, que es una secuencia de bits con una longitud *m* finita, donde:

$$0 \leq m < 2^{64}$$

El tamaño en bytes seria 2^{61} bytes.

Longitud del resumen *d.*

La segunda entrada de MD6 es la longitud deseada de la salida en bits, donde:

$$0 < d \leq 512$$

Cambiar el parámetro *"d"* debe dar lugar a una función hash totalmente diferente, no solo la salida ahora tiene una longitud diferente, pero su valor debe parecer no estar relacionado con los *valores hash* calculados por el mismo mensaje para otros valores de '*d'*.

Llave *k.*

A menudo es conveniente trabajar con una familia $\{H_{d,\ K}\}$ de funciones hash, no solo indexado por el tamaño a digerirsino también por una clave *'K'*extraídos de un conjunto finito.

El usuario puede proporcionar una llave *Ken keylen* bytes, para cualquier tamaño de llave, donde:

$$0 \leq keylen \leq 64$$

Es conveniente el uso de minúsculas en *k*para denotar el número máximo en 8 palabras de 64 bits para la llave, así que usamos *keylen*para representar el número actual de bytes en la llave suministrada.

Hay una función hash $\{H_{d,\ K}\}$ para cada combinación de longitud de digesto *d*y llave *k*. El valor por *default* cuando no se especifica la llave es de $k = null$de tamaño 0. $H_{d,}=H_{d,\ null}$

Dentro de MD6, la clave se rellena con ceros hasta que su longitud sea exactamente 64 bytes. El tamaño original de *keylen*en bites se conserva y una entrada auxiliar de la función de compresión de MD6.

El tamaño máximo de la llave (64 bytes) es bastante larga, lo que permite a la llave ser una concatenación se subcampos usados para diferentes propósitos (por ejemplo parte de una llave secreta, que forma parte un valor al azar) si se desea.

Si la llave desead es mayor a 512 bits, los primero pueden ser *hasheados* con MD6, utilizando, por ejemplo, $d = 512$ y $k = null;$el resultado puede ser suministrado a MD6 como la llave.

Modo De Control.*L*

El modo de operación estándar para MD6 es basado en un árbol jerárquico que se ilustra el la figura3.2.

Los datos del mensaje a *cifrar*se colocan en las hojas de un árbol cuaternario suficientemente largo. El cómputo se realiza de las hojas hacia la raíz. Cada nodo que no sea hoja del árbol corresponde a una ejecución de la función de compresión, que tiene $n = 64$ *palabras*de entrada y produce $c = 16$ *palabras*de salida. Los últimos *d bits*de la salida son producidos en la raíz y se toman como la salida de la función hash.

Existen casos donde el modo estándar de operación de MD6 requiere de mucha memoria; en estos casos, una variante de MD6 puede ser especificada para utilizar menos memoria (implicando que sea menos paralelizable).

Esta opción es ejercida opcionalmente con el *"parámetro de modo de operación"L.* Al variar *L,* MD6 varia suavemente entre un mínimo de memoria Merkle-Damgard-likecon un modo de operación secuencial ($L = 0$)y un modo altamente paralelizable basado en arboles con el modo de operación L = 64.

El modo de operación estándar de MD6 tiene $L = 64$para un funcionamiento totalmente jerárquico.En realidad cualquier valor de $L > 27$dará un hash jerárquico, $L = 64$es elegido como predeterminado para representar un valor "suficientemente largo" que el modo de operación secuencial nunca se invoca.

Numero de rondas *r.*

La función de compresión *f*tiene un parámetro *r*que controla el número de rondas, en términos generales, cada ronda corresponde a un ciclo de reloj en una implementación típica de hardware, o en 16 pasos para una implementación en software.

El valor por *default* de *r* es:

$$r = 40 + \lfloor d / 4 \rfloor$$

Así $H_{d,,\ K,\ L} = H_{d,,\ K,\ L,\ 40\ +\ |d/4|}$. Para*d = 160,*entonces MD6 tiene un valor por default de *r = 80rondas*, para *d = 512,* MD6, tiene un valor por default de *r = 168ronda.* Uno puede aumentar *r* para mayor seguridad, o disminuir *r* para aumentar su rendimiento, sacrificando seguridad para mejorar el desempeño.
Sin embargo cuando MD6 requiere modo codificado se requiere que $r \geq 80$. Esto proporciona protección de la llave, incluso cuando la salida deseada es corta (como podría ser una MAC). Así cuando MD6 tiene una clave vacía, el valor predeterminado de *r* es:

$$r = \max(80\ ,40\ +\lfloor\ d\ /\ 4\ \rfloor)$$

El parámetro de ronda *r* es expuesto en la API de MD6, así que puede ser variado explícitamente por el usuario. Esto puede ser de interés en el análisis de seguridad, o para aplicaciones con limitaciones de tiempo ajustado y reducción de las necesidades de seguridad. O bien uno podría aumentar por encimael valor por defecto de*r*para dar cabida a los diversos niveles de paranoia. Además, si se tiene una llave, pero no es secreta, entonces se podría especificar menos de 80 rondas si se desea.

Salida de MD6.

La salida de MD6 es una cadena de bits *D,* que tiene exactamente una longitud de *d* bits.

$$D = H_{d,\ K,\ L,\ r}\ (M);$$

*D*es el valor hash que se produjo mediante el mensaje *M. Es también llamado "messagedigest" por "resumen del mensaje"* MD.

En algunos contextos, la salida deMD6 puede ser definida para incluir otros parámetros. Por ejemplo en las firmas digitales, la *función hash* debe aplicarse por el remitente y también por el destinatario al mismo mensaje para comprobar la integridad. Estos cálculos deben producir el mismo resultado. Para que esto funcione, el receptor no solo debe conocer el mensaje *M*y la longitud del mensaje *d,*sino también los valores de los parámetros *K, L, r*con valores no predeterminados.
En estas aplicaciones, estos parámetros (excepto *K*) podría ser considerada como parte de la salida de la función hash. Por lo menos tiene que ser comunicado al receptor el valor del resumen *D*para verificar la integridad.

Modo De Operación De MD6.

Una función hash es típicamente construida a partir de una función de compresión, que asigna las entradas de longitud fija obteniendo cadenas de longitud fija más pequeñas. El *"modo de operación*" especifica como la función de compresión puede ser utilizada varias veces para habilitar entradas hash arbitrarias para producir una salida de longitud fija.

Para describir una *función hash,* uno necesita definir:

- El modo de operación,
- La función de compresión, y
- Varias constantes usadas en el procesamiento.

La función de compresión *f*toma entradas de longitud fija (*n = 89 palabras*) y produce salidas de longitud fija pero más cortas (*c = 16 palabras*).

$$f : W^{89} \rightarrow W^{16}$$

Recordando que *W* es el conjunto de todas las palabras binarias de longitud *w = 64 bits*. Por conveniencia llamaremos a un bloque-c de palabras *"chunk"*. El encadenamiento de variables producidas por MD6 son *"chunks"*.

Las*"89 palabras"*de entrada de la función de compresión*f* contienen15 palabras constates "*vectorQ",* 8 palabras para la llave *K,* una palabra U *"id Único",* una palabra de control *V,* y 64 palabras de datos del bloque B.

Puesto que *Q* es constante, la función de compresión *"efectiva"* o *"reducida"*$f_Q$asigna 74 palabras de entrada a 16 palabras de salida.

$$f : W^{74} \rightarrow W^{16}$$

Através de la relación:

$$f_q(x) = f_q(Q \,||\, x)$$

Por lo tanto, la función de compresión MD6 logra una compresión de 4 veces a partir de bloques de datos de 4 "*chunks"*ajustados exactamente dentro de un bloque de datos, y un *"chunk"*

Modo De Operación Jerárquico.

El modo de operación estándar es basado en arboles ver figura 3.1. Una implementación de este modo jerárquico requiere de almacenamiento por lo menos proporcional a la altura del árbol.

Dado que algunos dispositivos pequeños pueden no tener suficiente almacenamiento disponible, MD6 proporciona una alturalimitecon el parámetro "*L"*.Cuando la altura llega a L + 1, MD6 cambia el *operador de compresión paralelo* PAR, al *operador de compresión secuencial* SEQ.

El modo de operación de MD6 es opcionalmente parametrizado mediante un entero *"L"*tal que, $0 \leq L \leq 64$, que permite un transición fluida del modo predeterminado basado en el modo de operación dearboles jerárquico (*Para L grande*)hasta un modo de operación iterativo (*Para L = 0*). Cuando *L = 0,*MD6 trabaja de una manera similar a la del conocido método Merkle-damgard.

En nuestra descripción del modo de operación de MD6, MD6 representa a L *"paralelo"* fasessobre los datos, cada reducción del tamaño de los datos es de u factor de 4 y después realiza (si es necesario) un fase secuencial para finalizar.

Dado que el tamaño de entrada debe ser inferior a 2^{64} bits y la función de compresión final produce una salida de $2^{10} = 1024$ bits (antes de la truncación final a d bits), no serán mas de 27 fases paralelas (desde $27 = \log_4(264 = 210)$).
El valor por *default* de *L = 64*, ya que es mayor que 27, asegura que por defecto MD6 estará completamente jerárquico.

Gráficamente, MD6 crea una secuencia de arboles 4-ternarios de altura como máximo L, cada uno con 4^L hojas de *"chunks"*(con c = 16 palabras cada una), a continuación combina los valores producidos en sus raíces (si hay más de una) de una manera secuancial como *Merkle-Damgard-like*.

Si 4^L es mayorque el numero de bloques de 16-palabras en el mensaje de entrada, entonces solamente un árbol es creado y MD6 se convierte en un método puramente basado en arboles.
Por otra parte si L = 0, no se crean los arboles y la entrada está dividida en 48-palabras (*3-chunk*) bloques de datos que se combinan en una secuencia (*Merkle-Damgard-like*).En la actualidad hay solo tres *"chunks"*de datos en un bloque de datos, ya que un *chunk* es una cadena variable de la función de compresión previa en el nodo inmediatamente a la izquierda.
Para valores intermedios de L,negociaremos la altura del árbol (y por lo tanto los requisitos mínimos de memoria) en las oportunidades de paralelismo (y por lo tanto tal vez mayor velocidad).

La figura 3.4 indica el método top-level para el modo de operación de MD6, que se describe en una forma de abajo hacia arriba, nivel por nivel.

Primero todas la operaciones de compresión en el nivel *l =1*se llevan a cabo, a continuación todas la operaciones en el nivel *l = 2*se llevan a cabo y así sucesivamente.

MD6 es descrito de esta forma muy clara, una implementación practica, puede organizarse de una forma diferente (pero por supuesto, de manera que calcule la misma función). Por ejemplo, las operaciones en los diferentes niveles pueden ser mezclados, con una operación de compresión que se realiza tan pronto como sus entradas estén disponibles.

Cada operación de compresión se realiza de forma predeterminada con la operación PAR, que se describe en la figura 3.5. La operación PAR podrá llevarse a cabo en paralelo (de ahí su nombre). Teniendo en cuenta los datos del nivel *l – 1,* que producen los datos del nivel *l.* Que será en una cuarta parte del tamaño. Esto se repite hasta que se alcance un nivel en donde los datos restantes sean de apenas 16-palabras. Estos datos se truncan para convertirse en las *salida hash final*.

La figura 3.6 describe el modo de operación SEQ que es muy similar al funcionamiento *Merkle-Damgard*. Trabaja de forma secuencial a través de los datos de entrada en el último nivel y produce el resultado hash final.

Entradas de la Función de Compresión.

MD6 tiene formatos de entrada en la función de compresión *f* de la siguiente forma. Hay *n = 89 palabras,* formateadas de la siguiente manera con el tamaño por *default.*Ver la figura 3.7.Los primeros 4 *itemsQ, K, U, V*, son entradas auxiliares, mientras que el ultimo *ítem B*es de datos.

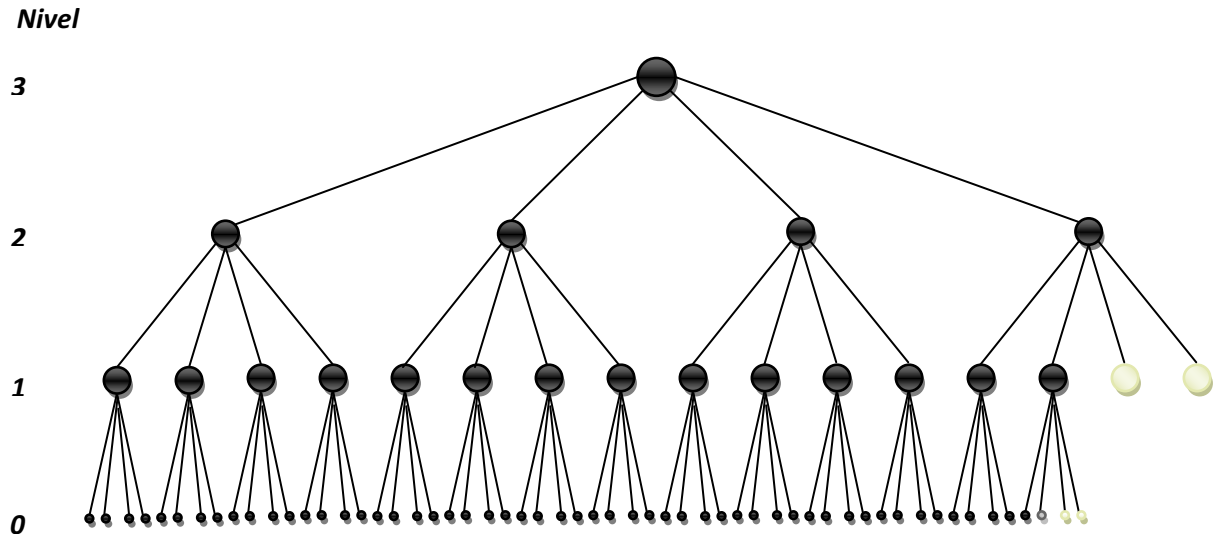


Ilustración 3.1 Estructura del modo de operación estándar de MD6

La figura 3.1Estructura del modo de operación estándar con *L = 64*. El cálculo procede de abajo hacia arriba, las entradas están en el nivel 0, y el valor hash final se emite en la raíz del árbol. Cada límite entre 2 nodos representa 16 palabras (128 bytes o 1024 bits) un*"chunk"*. Cada punto negro pequeño en el nivel 0 corresponde a un bloque de 16-palabras, un *"chunk"* del mensaje de entrada. El unto gris en el nivel 0 corresponde a un último bloque parcial (menos de 16 palabras) que se rellena con ceros hasta alcanzar el tamaño de 16 palabras. Un punto blanco (en cualquier nivel) corresponde a un bloque relleno de puros ceros. Cada punto negro mediano o grande arriba del nivel 0 corresponde a una aplicación de la función de compresión. El punto negro más grande representa la operación compresión final, esta es la raíz. El valor hash final se obtiene al truncar el valor calculado ahí.

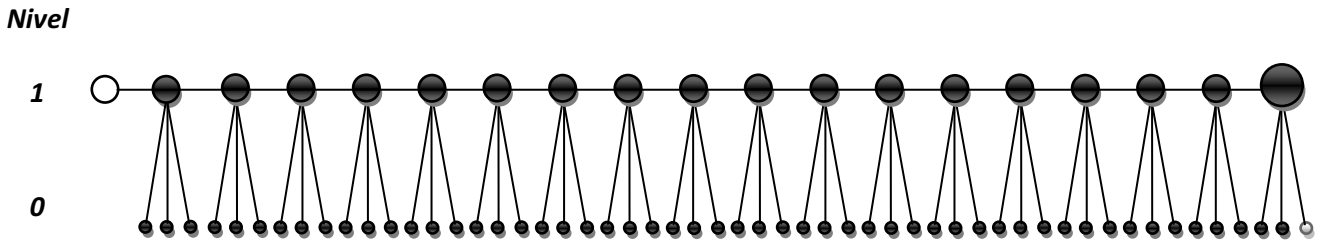


Ilustración 3.2 Estructura del modo de operación Secuencial de MD6

Figura 3.2 Estructura del modo de operación secuencial con *L = 0*. El computo procede únicamente de izquierda a derecha, El nivel 1 representa el procesamiento por *SEQ*.
La salida de la función hash es producida por el nodo que estamás a la derecha del nivel 1. Esto es similar al procesamiento estándar *Merkle-Damgard*. El círculo blanco de la izquierda en el nivel 1 es de 1024 bits todo ceros, es el vector de inicialización para el cálculo secuencial en ese nivel. Cada nodo tiene 4 entradas 1024-bits uno desde la izquierda y tres debajo, el "tamaño de bloque del mensaje" efectivo es de entonces 384 bytes, ya que 128 bytes de la entrada de 512 bytes de la función de compresión son utilizados para el"*encadenamiento de valores*".

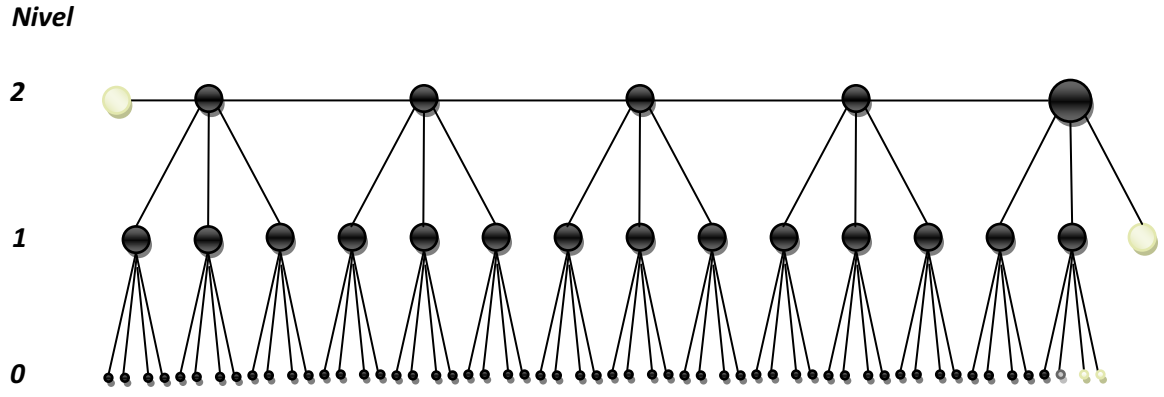


Ilustración 3.3 Estructura del modo de operación intermedio de MD6

Figura 3.3 Estructura del modo de operación con un modo intermedio de la operación *L = 1.* El computo procede de abajo hacia arriba y de izquierda a derecha, el nivel 2 representa el procesamiento secuencia. La salida de la función hash es producida por el nodo que esta más a la derecha del nivel 2. El circulo blanco a la izquierda del nivel 2 es el vector de inicialización con todos sus valore en cero para el cálculo secuencial en ese nivel.

Q – Un vector contante (que da una aproximación a la parte fraccionaria de la raíz cuadrada de 6) de longitud *q = 15 palabras*.

K – Una *"llave"*(que actúa como la etiqueta, llave secreta, etc.)de longitud *k = 8-palabras,* que contiene un llave prevista en *keylen* bytes

U–Una palabra "*id* de nodo único"

V– Una palabra "palabra de control"

B–Un bloque de datos de longitud *b = 64-palabras*

El Nodo Único ID U

El *"nodo único ID"* es una entrada auxiliar de una sola palabra para la función de compresión. En particularpara llevar a cabo la operación de la función de compresión únicamente se especifica, dando tanto el número de nivel para esta operación como el índice dentro de este nivel. Ver Figura 3.9

- *l* - Un byte dando el numero de nivel
- *i* - Siete bytes que indican la posición dentro del nivel, con la operación de la función de compresión mas a la izquierda (la primera) etiquetada como *i = 0.*

Por ejemplo, la primera operación de la función de compresión que se lleva a cabo es *U = (l, i) = (1, 0).* Un nodo *(l, i)*en el nivel *l, 1 < l ≤ L(producido por PAR)* tiene hijos en (*l-1,4i*), (*l-1,4i + 1*), (*l-1,4i + 2*), y (*l-1,4i + 3*). Un nodo *(l, i)*en el nivel *1<l = L + 1 (Producido por SEQ)* tiene hijos en *(l-1, 3i), (l-1, 3i + 1), y (l-1, 3i + 2).*

Palabra de Control *V*

La *palabra de control V*es una entrada auxiliar (una palabra) de la función de compresión que da los parámetros relevantes para el procesamiento. Vea la figura 3.8

Modo De Operación De MD6

Entradas:

M: Un mensaje *M* a ser procesado de longitud *m* en bits.

d : La longitud d (en bits) de la salida hash deseada $1 \leq d \leq 512$

K: Una llave arbitraria $k = 8$ *palabras*, que contiene una llave suministrada en *keylen* bytes, rellenándolo a la derecha con (*64 - keylen*) bytes en ceros.

L: EL parámetro de modo (numero de nivel máximo, o el número de fases paralelas).

r : Numero de rondas.

Salida:

D: El valor hash en *d bits* $D = H_{d,K,L,r}(M)$

Procedimiento:

Inicializar:

- Sea $l = 0$, $M_0 = M$ y $m_0 = m$

Bucle principal nivel por nivel:

- Sea $l = l + 1$.
- Sí $l = l + 1$, regresar $SEQ(M_{l-1}, d, K, L, r)$ como la salida de la función hash.
- Sea $M_l = PAR(M_{l-1}, d, K, L, r, l)$. Siendo $m_l$la longitud de M_l en bits.
- Si $m_l = cw$ *(*es decir, si M_l es de longitud *c palabras),* Regresar los últimos *d bits*de M_l como la salida de la función hash. De lo contrario regresar a la parte superior del bucle principal nivel-por-nivel.

Ilustración 3.4 Especificaciones del modo de operación de MD6

Figura 3.4 Modo de operación de MD6. Con la configuración por *default* de $L = 64$, la operación *SEQ* nunca es utilizada, La operación PAR es llamada continuamente para reducir el tamaño de entrada por un factor de b = c = 64 = 16 = 4 hasta quedar un bloque de 16 palabras (*un chunk*). Por otro lado, configurando L = 0 Se obtiene un modo totalmente secuencial de MD6.

Operación *PAR* de MD6

Entradas:

M_{l-1}: Un mensaje M a ser procesado de longitud m_{l-1} en bits.

d : La longitud d (en bits) de la salida hash deseada $1 \leq d \leq 512$

K: Una llave arbitraria $k = 8$ *palabras*, que contiene una llave suministrada en *keylen* bytes alineados a la derecha con (*64 - keylen*) con bytes en ceros.

L: EL parámetro de modo (numero de nivel máximo, o el número de fases paralelas).

r : Numero de rondas.

l : Un entero no negativo que indica el numero de nivel $1 \leq l \leq L$

Salida:

M_l: Un mensaje de longitud *ml* bits, donde $ml = 1024 * max(1, \lceil m_{l-1} / 4096 \rceil)$.

Procedimiento:

Inicializar:

- Sea Q un arreglo de longitud q = 15 palabras dando la parte fraccionaria de $\sqrt{6}$.
- Sea f la función de compresión a la que se le asignan 89 palabras de entrada (incluyendo las 64 palabras del bloque de datos B) obteniendo como salida un *chunk* C = 16 palabras utilizando r rondas para el procesamiento.

Reducción:

- Extender la entrada M_{l-1} si es necesario (y solo si es necesario) añadiendo bits en cero hasta que su longitud sea un múltiplo entero positivo de $b = 64$ *palabras* .Entonces M_{l-1} puede ser visto como un secuencia de $B_0, B_1, \ldots, B_{j-1}$ de bloque de *b palabras,* donde$j = max(1, \lceil m_{l-1} / bw \rceil)$.
- Para cada bloque de *b-palabras* B_i, i = 0, 1, …. , *j-1*, procesar $C_i$de la siguiente forma.

  - Sea p el numero de bits de relleno en B_i ; $0 \leq p \leq 4096$ (p solo puede ser distinto de cero para $i = j - 1$)
  - Sea $z = 1$ si $j = 1$, si no dejar que sea $z = 0$ ($Z = 1$ solo para el último bloque a comprimir en el computo total).
  - Sea v un palabra con los valores $r \,||\, L \,||\, z \,||\, p \,||\, keylen \,||\, d$Ver figura 3.8
  - Sea $U = l * 2^{56} + i$ un Nodo único ID, un valor único de una palabra para esta función de compresión.
  - Sea $C_i = f(Q \,||\, K \,||\, U \,||\, V \,||\, B_i)$. ($C_i$ tiene una longitud de c= 16 palabras).
- Regresar $Ml = C_0 \,||\, C_1 \,||\, \ldots \,||\, C_{j-1}$

Ilustración 3.5 Especificaciones *PAR*

Figura 3.5 El operador PAR es una operación de compresión paralela que produce el nivel l a partir del árbol partir del nivel $l - 1$. Con la configuración L = 64, esta rutina es usada varias veces en cada nivel del árbol para generar el siguiente nivel superior, hasta que el valor de la raíz es producido.

Operación SEQ (Opcional)

Entradas:

M_L: Un mensaje M de cierta longitud m_L en bits.
d : La longitud d (en bits) de la salida hash deseada $1 \leq d \leq 512$
K: Una llave arbitraria $k = 8$ *palabras* , que contiene una llave suministrada en *keylen* bytes alineados a la derecha con (*64 - keylen*) con bytes en ceros.
L: EL parámetro de modo (máxima altura del arbol).
r : Numero de rondas.
l : Un entero no negativo que indica el numero de nivel $1 \leq l \leq L$

Salida:

D: El valor hash en *d bits.*

Procedimiento:

Inicializar:

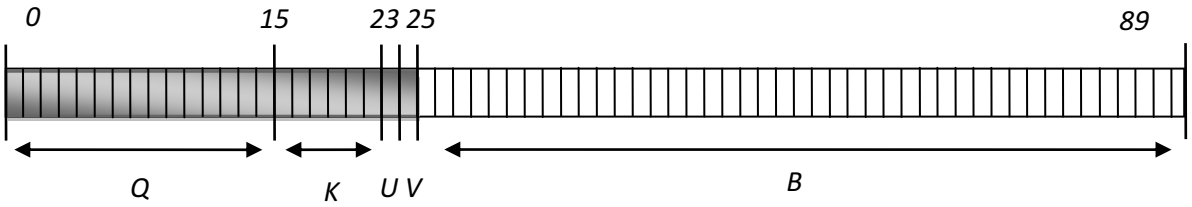
- Sea Q un arreglo de longitud q = 15 palabras dando la parte fraccionaria de $\sqrt{6}$.
- Sea f la función de compresión a la que se le asignan 89 palabras de entrada (incluyendo las 64 palabras del bloque de datos B) obteniendo como salida un *chunk* C = 16 palabras utilizando r rondas para el procesamiento.

Bucle principal:

- Sea C_{-1} un vector cero de longitud $c = 16$ *palabras* (Este es el "*IV*")
- Extender la entrada M_L si es necesario (y solo si es necesario) añadiendo bits en cero hasta que su longitud sea un múltiplo entero positivo de $(b - c) = 48$ *palabras* .Entonces M_L puede ser visto como un secuencia de $B_0, B_1, \ldots, B_{j-1}$ de bloques de $(b - c)$ *palabras,* donde $j = max(1, \lceil m_L / (b - c)w \rceil)$.
- Para cada bloque de $(b - c)$ *palabras* B_i, i = 0, 1, …. , *j-1*, en secuencia procesar $C_i$de la siguiente forma.
  - Sea p el numero de bits de relleno en B_i ; $0 \leq p \leq 3072$(p solo puede ser distinto de cero para $i = j - 1$)
  - Sea $z = 1$ si $i = j - 1$, si no dejar que sea $z = 0$ ($Z = 1$ solo para el último bloque a comprimir en el computo total).
  - Sea v un palabra con los valores $r \,||\, L \,||\, z \,||\, p \,||\, keylen \,||\, d$ver figura 3.8
  - Sea $U = l * 2^{56} + i$ un Nodo único ID, un valor único de una palabra para esta función de compresión.
  - Sea $C_i = f\,(Q \,||\, K \,||\, U \,||\, V \,||\, C_{i-1} \,||\, B_i)$. ($C_i$ tiene una longitud de c= 16 palabras).
- Regresar los últimos d bits de C_{j-1}como la salida de la función hash.

Ilustración 3.6 Especificaciones *SEQ*

Figura 3.6EL operador MD6 SEQ es una operación hash secuencial "Merkle-Damgard-Like", produciendo el valor hash de salida final. Con la configuración por *default* SEQ nunca es utilizado en el procesamiento.



Vector de entrada en la función de compresión

La entrada de la función de compresión contiene 89 palabras de 64bits: un vector constante *Q* de 15 palabras, una palabra de id de nodo único *U,* una palabra de la variable de control *V,* y 64 palabras del bloque de datos *B.*Los primeros cuatro ítems construyen la información auxiliar, mostrados en gris.

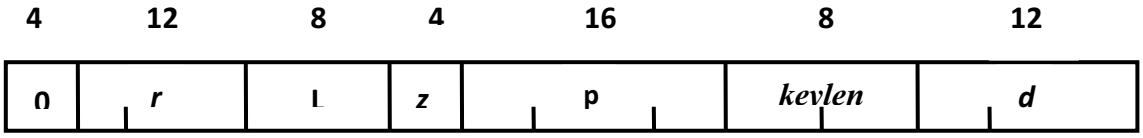


Ilustración 3.7 Palabra de control V

Figura 3.8Composición de la palabra de control *V*. Los 4 bits más significativos son ceros (reservados para uso futuro). El tamaño (en bits) para cada campo estádado por encima del campo.

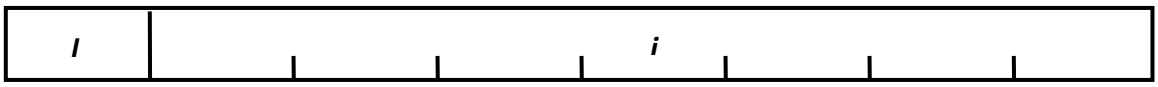


Ilustración 3.8 Palabra de nodo unico U

Figura 3.9Composición de la palabra *U* de nodo único. Los *l bytes* más significativosson el numero de nivel, El campo de 7 bytes *i* dan el índice del nodo en el nivel (*i =0,1*...).

- *r* – Numero de rondas en la función de compresión (12 bits).
- *L*-Parámetro modo (Nivel máximo).
- *z* – El valor 1 si es la operación de compresión final, de lo contrario 0 (4 bits). En las figuras 2.1, 2.3, y 2.2 Estas operaciones se indican en los círculos negros más grandes.
- *p* – El numero de bits de datos de relleno (rellenado con bits de cero) en el bloque de entra actual *B* (16 bits).
- *keylen* – La longitud original (en bytes) de la llave *k* suministrada (8 bits).
- *D* – La longitud deseada en bits del resumen de salida (12 bits).

Función De Compresión MD6

Esta sección describe el funcionamiento de la función de compresión de MD6.

La función de compresión *f* toma como entrada un *arreglo N* de longitud *n = 89 palabras*.Y como salida un *arreglo C* de longitud c = *16 palabras.*

Aquí *f* se describe teniendo una sola entrada *N de 89 palabras*, aunque también puede ser visto teniendo una *entrada auxiliar de 25 palabras(Q || Q || U || V)*seguidas de un *bloque de entrada de datos B de 64 palabras*.

La función de compresión *f* es calculada como se muestra en la figura 2.10.

La función de compresión puede ser vista como un conjunto N cifrado con una llave arbitraria S de longitud fija (que no es secreta), seguida de una operación de truncamiento que regresa solo las ultimas 16 palabras de $E_S(N)$. Ver figura 2.11. Aquí se determinala *"constante de ronda"S*del algoritmo de encriptación.

Internamente, la función de compresión tiene un bucle principal de *r rondas* (cada una consiste en *c = 16 pasos*) seguida de una operación de truncamiento el cual trunca el resultado final a *c = 16 palabras.*

El bucle principal se lleva a cabo en un total de *t = rc pasos,*donde cada paso calcula el valor de una palabra. Este bucle puede ser implementado mediante la carga de la entradaen las primera *n palabras* de un *arreglo A* de *longitud n + t,* entonces calcular cada una de las de las *t palabras restantes* a su vez, Ver figura 3.12. Equivalentemente este bucle puede ser implementado como un registro de corrimiento con retroceso no lineal con 89 palabras de estado. Ver figura 3.14

La operación de truncamiento simplemente devuelve las ultimas 16 palabras de *A*como salida de la función de compresión (que también llamamos "*la variable de cadena"*). La función de compresión siempre genera *c = 16 palabras (1024 bits)* por esta cadena de variable. Esta es al menos dos veces mayor que cualquier salida de la función hash de MD6, en línea con la estrategia de *"wide pipe".*

La función de compresión toma las"feedbacktap positions"t_0, t_1, t_2, t_3, t_4,cada uno en el *rango de 1 a n-1 = 88,* como parámetros. (Tome en cuenta la sobrecarga para leve para el símbolo *t*: cuando es subíndice se refiere a una posición de tap, cuando no es subíndice se refiere al número de pasos calculados r = *rc*)

La Función De Compresión *f* de MD6

Entradas:

N : Un arreglo *N[0 ... n -1*] de *n = 89 palabras. (*Esto consiste típicamente en 25 bloques de palabras de información auxiliar, seguidas de un bloque de datos B de 64 palabras, pero aquí todas las entradas son tratadas uniformemente*)*

r : Numero de rondas.

Salida:

C: Un arreglo *C[0 .... c-1]* de longitud *c = 16 palabras.*

Parámetros: *c, r,* t_0, t_1, t_2, t_3, t_4, r_i, l_i, S_i, *cuando* $1 \leq t_i \leq n$ *para todo* $0 \leq i \leq 4$, $0 \leq r_i, l_i \leq w/2$ *para todo i, y* S_i *es una palabra de w-bits para cada* $i, 0 \leq i \leq t$

Procedimiento:

Sea $t = rc$. (Cada ronda tiene $c = 16$ *pasos)*.

Sea *A[0 ... t + n -1]* un arreglo de *n + t palabras.*

Inicializar:

- *A[0 ... n -1]* es inicializado con la entrada *N[0 ... n -1]*

Bucle principal del cálculo:

for *i = 0* ***to*** *t + n -1: /* pasos*/*

$$x = S_{i-n} \oplus A_{i-n} \oplus A_{i-t0} \quad \textit{[Línea 1]}$$

$$x = S_{i-n} \oplus (A_{i-t1} \wedge A_{i-t2}) \oplus (A_{i-t3} \wedge A_{i-t4}) \quad \textit{[Línea 2]}$$

$$x = x \oplus (x \gg r_{i-n}) \quad \textit{[Línea 3]}$$

$$x = x \oplus (x \ll l_{i-n}) \quad \textit{[Línea 3]}$$

Truncamiento y salida:

Salida *A[t + n – c ... t + n - 1]* es el arreglo de salida *C[0 ... c-1]* de longitud *c = 16.*

Ilustración 3.9 Función de compresión f

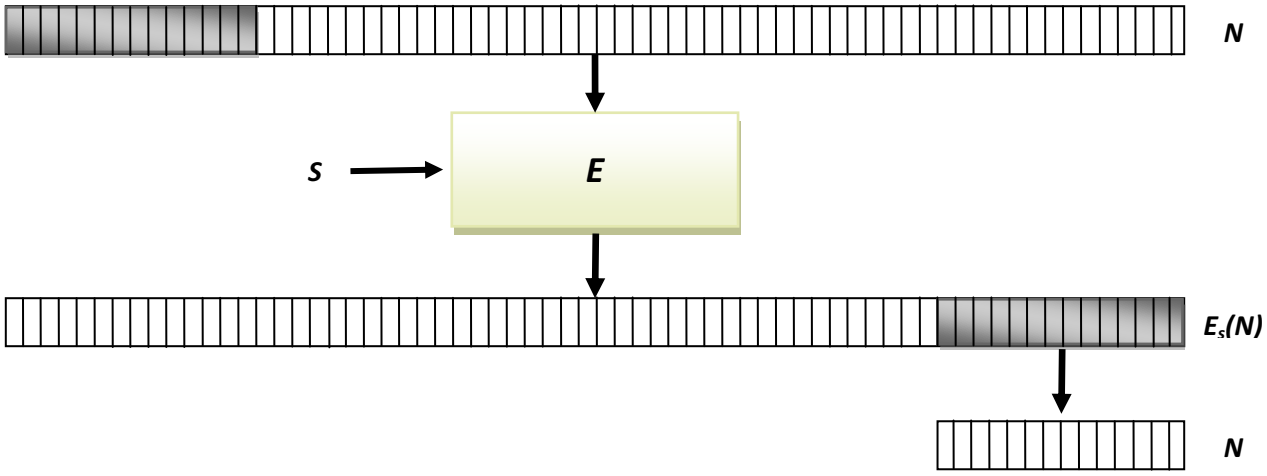


Ilustración 3.10 Operación de encriptación

Figura 3.11 La función de compresión *f* vista como una operación de encriptación seguida por una operación de truncamiento. La entrada N de 89 palabras (Con las primeras 15 palabras sombreadas indicando que son una constante) es encriptada (cifrada) bajo control de la *llave S* de la función$E_S(N)$. La *llave S*es arbitraria pero de longitud fija y publica. Las ultimas 16 palabras se $S_M(N)$ forman la compresión deseada de la salida C de la función de compresión.

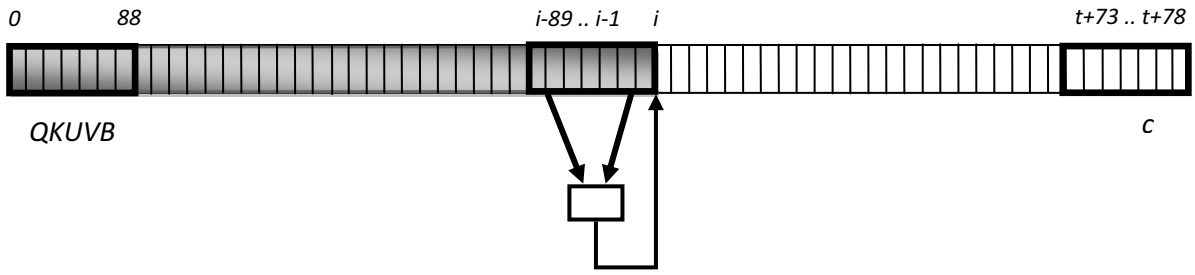


Ilustración 3.11 Función de retroalimentación

Figura 3.12 El bucle principal de la función de compresión. La parte del arreglo *A[0 ... 88]*es inicializado con la entrada de la función de compresión (*Q, K, U, V, B*) ver figura 2.7. Entonces para*i = 89, 90, ... , t + 88*, *A[i]*se calcula como función de retroalimentación de *A[i – 89, ... , i - 1]*puede ser vista como una *"ventana deslizante"*moviéndose de izquierda a derecha como la porción calculada (mostrada en gris) crece. Las ultimas 16 palabras para la ventana final *A[t, ... , t + 88]*, que son*A[t + 73, ... , t + 88],* constituye la salida C.

Constantes de MD6

Tap positions:

t_0	t_1	t_2	t_3	t_4
17	18	21	31	67

Cantidad de desplazamientos:

Los valor de r_{i-n}, l_{i-n} son la cantidad inicial de desplazamientos a la derecha y la cantidad posterior desplazamientos a la izquierda en el paso con el *índice i,* el *índice* $i - n$ se toma el modulo $c = 16$.

$(i - n) \mod 16$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_{i-n}	10	5	13	10	11	12	2	7	14	15	7	13	11	7	6	12
l_{i-n}	11	24	9	16	15	9	27	15	6	2	29	8	15	5	31	9

Constante de ronda:

Los valores S_{i-n} para $i \geq n$son determinados de la siguiente forma. Debido a que S_{i-n}es una constante dentro de una ronda, pero cambia de ronda a ronda, S_{i-n}se define en términos de una secuencia auxiliar S'_j. La constante S'_1 y S^* determinan la secuencia de completa.

$$S_{i-n} = S'_{[(i-n)\,/\,16]}$$

$$S'_0 = 0x0123456789abcdef$$

$$S^* = 0x7311c\ 28124225cfa0$$

$$S'_{j+1} = \left(S'_j \lll 1\right) \oplus (S'_j \wedge S^*)$$

Ilustración 3.12 Constantes de MD6

Además para cada i, $0 \leq i < t$, hay parámetros r_i, l_i denotando cantidades de desplazamientos fijos a la derecha y a la izquierda: se trata de valores diferentes de cero, $0 < r_i, l_i \leq w/2 = 32$. Finalmente existe un valor (de una palabra) de "*retroalimentación constante*" $S_i$para cada i, $0 \leq i < t$.

Las líneas1 y 2 en el bucle de la figura 2.10 son completamente paralelas bit a bit, sin mezclar posiciones entre diferentes bits. La línea 1 contiene el termino constanteS_i y los términos lineales A_{i-n} (el término*"fin de ronda")* y A_{i-t0}. *La* línea2 contiene *"termino cuadrático"* no lineal $(A_{i-t1} \wedge A_{i-t2})$ y $(A_{i-t3} \wedge A_{i-t4})$.

Tenga en cuenta no perder la información durante el cálculo de la función de compresión hasta el truncamiento final a una salida de longitud de *c palabras.* De hecho el paso de compresión básica es invertible, debido a la recurrencia exclusiva *A[i -n]* con los valores que solo dependen de los elementos de entrada de otra entrada *A[i -1, ... , i – n + 1].* Y porque aplicándole una operación XOR a un valor con el mismo pero desplazado es invertible (A esta operación se le conoce como el operador *"xorshift"*es conocida y estudiada en la literatura de números aleatorios). Así la asignación de dentro de *f* de las primeras *n-palabras* de *A*alas últimas n-palabras de A son invertibles, lo que corresponde a la asignación Π de la figura 2.15

En hardware, el paso de iteración básica puede ser implementado con un circuito de ancho constante, no hay multiplicaciones, sumas u otras operaciones que requieran de una propagación de acarreo u otros retrasos no constantes. Esto significa que una implementación en hardware puede ser extremadamente rápida. Tenga en cuenta que desde $c \leq min(t_0, t_1, t_2, t_3, t_4)$, no hay dependencia de un paso en la salida de otrodurante un mínimo de *c pasos*.Donde los pasos dentro de una ronda son independientes, todos los pasos dentro de una ronda pueden ser ejecutados simultáneamente. En general, una ronda solo requiere de un *ciclo de reloj,*y una función decompresión puede ser calculada en *r ciclos de reloj.*
No hay tablas de búsqueda de datos dependientes u otras operaciones en que el momento podría ser dependiente de los datos. Esto reduce la vulnerabilidad a los ataques *"side-channel" (ataques laterales)*.

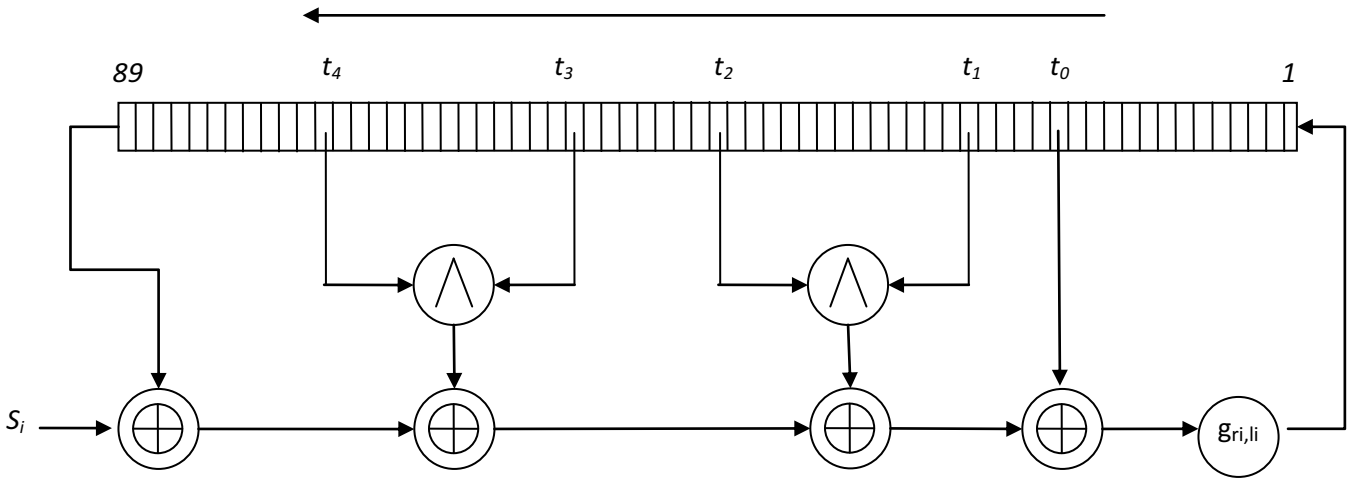


Ilustración 3.13 Registro de corrimiento con retroalimentación no lineal

Figura 3.14 El bucle de computo principal de la función de compresión es visto como un registro de desplazamiento con retroalimentación no lineal. Cada celda del registro de desplazamiento, cada línea, y cada *Si*representan una palabra de 64 bits, debido a que el registro de desplazamiento tiene una dirección de derecha a izquierda, es muy práctico que

la celda 1 este a la derecha y la celda 89 este hacia la izquierda; la celda marcada con 1 corresponde a la localidad *A*[*i-1]*en la figura 3.10, y la celda marcad con 89 corresponde a la localidad *A*[*i-89]*. La *tap positions* que se muestran son correctas para $t_0 = 17$ y para la posición $n = 89$, pero para $t_1 \ldots t_4$ son solo ilustrativos (por la facilidad de dibujo). La función $g_{ri,\ li}$ es la función descrita en la *ecuación 3.0*Cada palabra de retroalimentación de 64 bits depende de una palabra S*i*y seis palabras del registro

Pasos, rondas y rotaciones.

Los $t = rc$ *pasos* pueden ser considerados como una secuencia de rotaciones, cada uno de los cuales consta de de n = 89 pasos y con ello plenamente los ciclosdel registro de desplazamiento de retroalimentación de *n-palabras* mediante el cálculo de las siguientes *n palabras de estado.*
Tengamos en cuenta que una rotación consiste en $n/c = 89 / 16 = 59 / 16$ *rondas,* y 2 rotaciones que corresponden a casi 11 rondas.
Cada rotación produce un nuevo *vector de n-palabras,* puedes ver el cálculo como *rc / n* rotaciones, cada rotación produce un *vector de estado invertible de n-palabras*del vector de estado anterior de n palabras. La entrada es el primer vector de estado de *n palabras;* la salida son las ultimas c palabras de las ultimas n palabras del vector de estado. (Nota: no necesariamente *rc / n* no es necesariamente un entero).
La función de compresión también puede ser vista como r rondas de 16 pasos cada una calcula las siguientes *16 palabras de estado*.

Las constantes de retroalimentación se organizan de manera que refleje el punto de vista de la orientación de la ronda. La ronda constante *Si*sigue siendo la misma para todos los 16 pasos de una ronda y los cambios para la próxima ronda. El cambio de cantidades *r() y l()*varían dentro de una ronda pero después tienen el mismo patrón de variación en rondas sucesivas.

Para la implementación en software de MD6, es conveniente realizar 16 veces un bucle desenrollándolo, de modo que cada ronda es implementada como un bloque básico de código en una rama libre.

Difusión *Intra-palabra*vía*xorshifts*

En algunos métodos es necesariollevar a cabo difusión entre las diferentes posiciones de los bits dentro de una palabra. Esta difusión *intra-palabra* es proporcionada por el operador $g_{ri,\ li}$ implícita en las líneas 3-4 en el bucle de la figura 2.10.

$$g_{ri,li}(x) = \{\, y = x \oplus (x \gg ri);$$

$$\textbf{retrun}\ y \oplus (x \ll li) \qquad \textit{Ecuación 3.0}$$

$$\}$$

Aquí *"l" y r* son sobrecargadas; *ri* y *li*se refieren a la cantidad de corrimientos, mientras que *r* refiere al número de rondas y *l* es usada en el modo de operación y refiere al numero de nivel.
La función g es lineal e invertible (sin perdidas).
El operador $x = x \oplus (x \gg ri)$ y $x = x \oplus (x \ll li)$ son conocidos como operadores *"xorshifts"*.

Cantidad de desplazamientos

Los valores de desplazamiento están en el índice c = 16 (es decir, se repiten cada ronda).

Constantes de ronda.

Las constantes de ronda S_j' proporcionan cierta variabilidad entre rondas. Cada ronda *j* tiene su propia constante S_j'; los pasos dentro de una ronda todos usan la misma ronda constante.

La siguiente recurrencia genera estas constantes:

$$S_0' = 0x0123456789abcdef$$
$$S^* = 0x7311c2812425cfa0$$
$$S_{j+1}' = \left(S_{j+1}' \lll 1\right) \oplus \left(S_j' \wedge S^*\right)$$

Esta relación de recurrencia es uno a uno de S_j' se puede determinar S_{j+1}', y viceversa.

Representación Alternativa De La Función de Compresión.

Hay otra forma útil de representar la función de compresión. Dado que la parte *Q* 15-palbars es una entrada constante fija, podemos definir la función f_Q como:

$$f_Q(x) = f(Q \,||\, x)$$

donde f_Q asigna entradas de longitud $n - q = 74$ *palabras* hacia salida de longitud $c = 16$ *palabras*.Esta representación de *f* es útil en las pruebas de seguridad de la función de compresión.

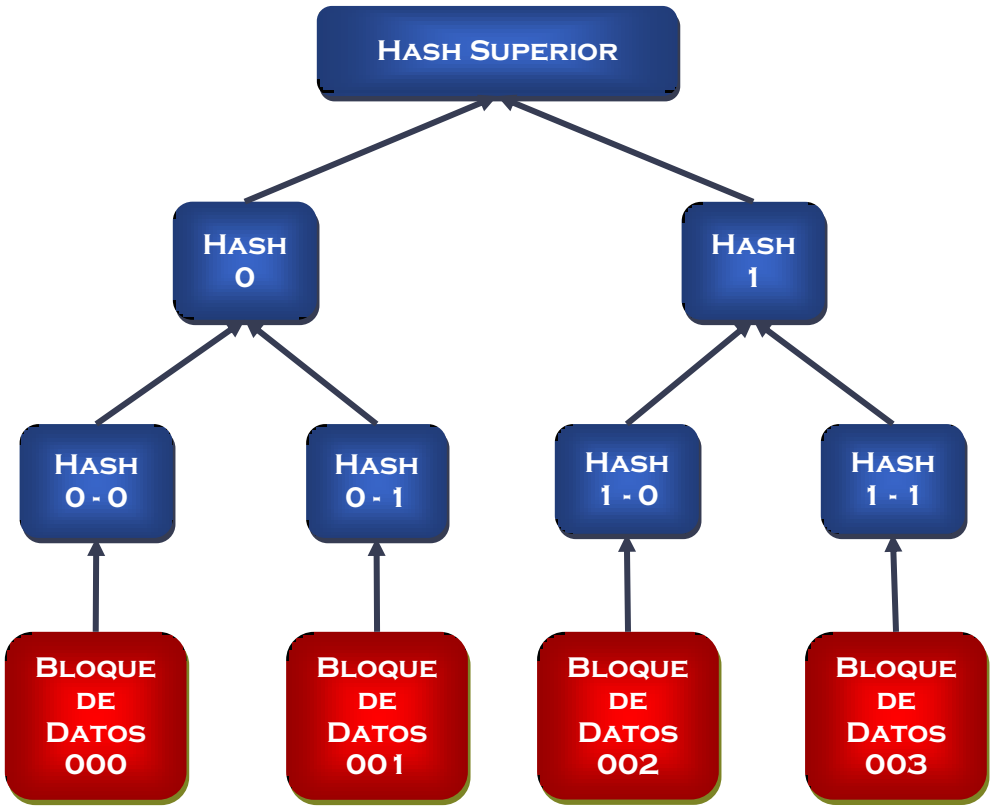
Analisis y Pruebas del Proyecto.

Esta sección del documento es una bitácora de análisis, desarrollo y construcción de la "*Implementación Paralela Del Algoritmo Para Compendios De Mensaje MD6*" para poderse ejecutar bajo sistemas multicore teniendo la posibilidad de procesarse concurrentemente, mostrando la concepción del análisis y diseño de la implementación así como de las particularidades presentadas en el desarrollo y haciendo una relación diseño – implementación, que de una perspectiva de cómo se fue diseñando e implementando en base a su descripción.

Para esta parte del proyecto se inicia retomando el diseño de la implementación secuencial y analizando los bloques que se van paralelizar.

La identificación del proceso que se efectuara paralelamente se transmitirá a la aplicación sobre la implementación cilk que es un lenguaje de programación de propósito general diseñado para la programación multihilo.
Se presentan decisiones y como se aplica cierta lógica en los módulos para llevar a cabo su tarea bajo este modelo.

Estructura general del procesamiento del algoritmo



La estructura de árbol permite asignar bloques de datos en los nodos inferiores del árbol los cuales serán calculado paralelamente, el cálculo se hace de abajo hacia arriba en donde el nodo raíz contendrá el valor hash final.

Análisis Diseño –Implementación

En esta sección se mostraran las partes que llevaron al diseño del algoritmo de forma concurrente.

Como bien el proceso consiste en generar el digesto para un archivo con n bytes, este mismo se leerá y se almacenara en memoria para poderse iniciar la compresión.
Como recordaremos de la teoría la base de procesamiento del algoritmo se basa en la estructura de árbol (merkletree) como se muestra a continuación.

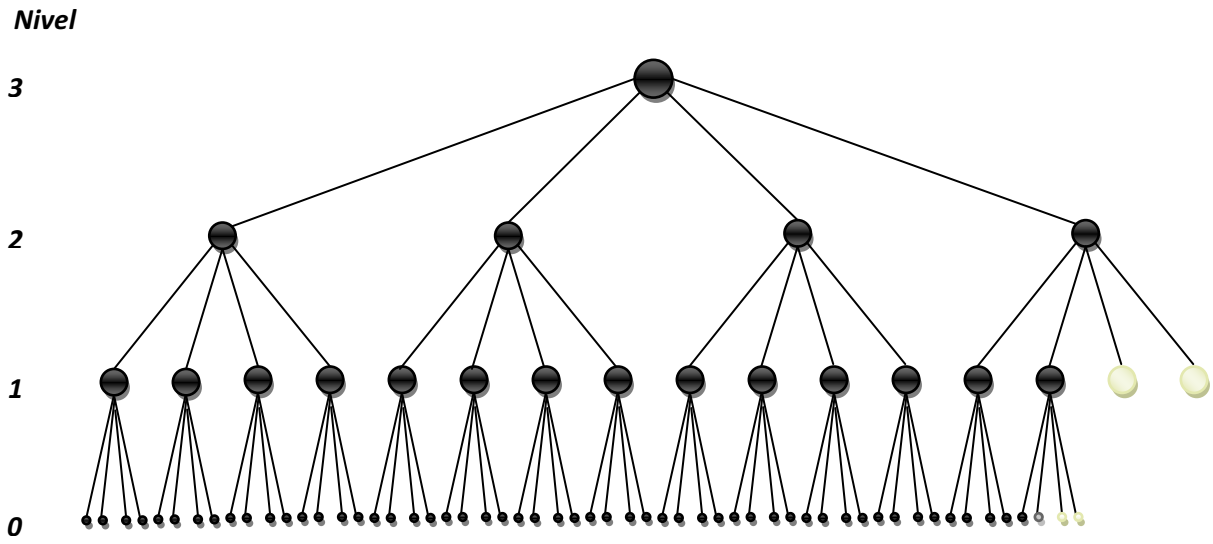
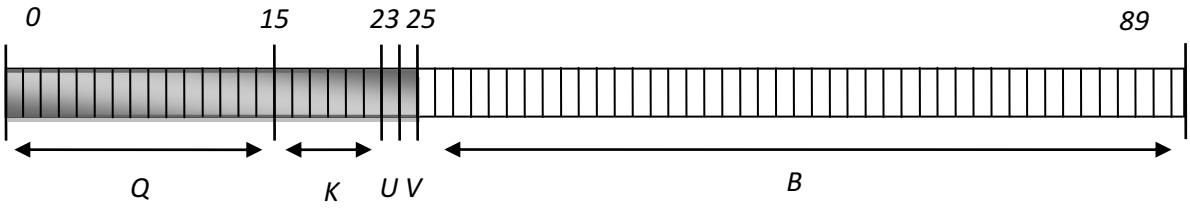


Ilustración: Estructura del modo de operación estándar de MD6

Basado en esto tenemos.

- La longitud del mensaje de entrada
- En base a lo longitud del mensaje se puede calcular el número de niveles del árbol
- Como cada nivel superior es la longitud del nivel reducido en 4; se conoce la longitud para cada nivel permitiendo generar las estructuras donde se cargara la información, donde se guardara los digestos parciales en donde se almacenara el mensaje final.

El bloque de entrada se compone de la siguiente forma



Vector de entrada en la función de compresión

una función que nos empaquetara la entrada completa según sus variables.

- Por lo que nuestra función principal de compresión leerá 512 bytes crenado su bloque completo y aplicando la operación de digesto.
- Nuestro bloque de entrada se encuentra almacenado en memoria por el apuntador B. con una longitud SIZE; teniendo esto como premisa se diseñó la función *hash_segment* bajo la siguiente estructura.

```
cilk void hash_segment(int ell, size_t lo, size_t hi){
  if ((hi-lo)==512) /* tamaño de bloque de entrada */
    compress_block(ell,lo);
  else
  { size_t mid = (lo+hi)/2;
    spawn hash_segment(ell,lo,mid);
    spawn hash_segment(ell,mid,hi);
    sync;
  }
}
```

- Una función recursiva que tiene como parámetros de entrada:

  ell: número de nivel de ejecución.
  lo: límite inferior del bloque de procesamiento
  hi: límite superior del bloque de procesamiento

Aplicando el operador de minimización para la recursión mientras el bloque de procesamiento no sea igual a 512 bytes (64 palabras) ; se dividirá en bloques de 2 aplicando la misma función para los bloques por separado.

Análisis del algoritmo

Se tiene la siguiente definición

$$t(n) = \begin{Bmatrix} 1 \; si \; n = 1 \\ 2t\left(\frac{n}{2}\right) si \; n > 2 \end{Bmatrix}$$

n	T(n)
1	1
2	2
4	4
8	8
16	16

Por lo que el orden del algoritmo es:

O(t(n)) = n

Para el cálculo del orden del algoritmo en paralelo se tienen los siguientes factores:

Suponiendo un caso ideal se tendría que O(t(n)) = n/p

En un programa paralelo la estimación del tiempo es más compleja.
Se tiene como factor el número de procesadores, la manera que están conectados entre sí (topología) y los módulos de memoria (memoria compartida o distribuida).

Dificultades:

- No siempre es fácil determinar el orden en que los procesadores empiezan y terminan.
- A lo largo de la ejecución de un programa paralelo hay puntos de sincronización de los que no siempre podemos determinar su duración al no saber en el orden en que van a llegar los distintos procesos a estos puntos.

Pruebas

Para la ejecución de pruebas se ocupó un equipo con las siguientes características

Ubuntu

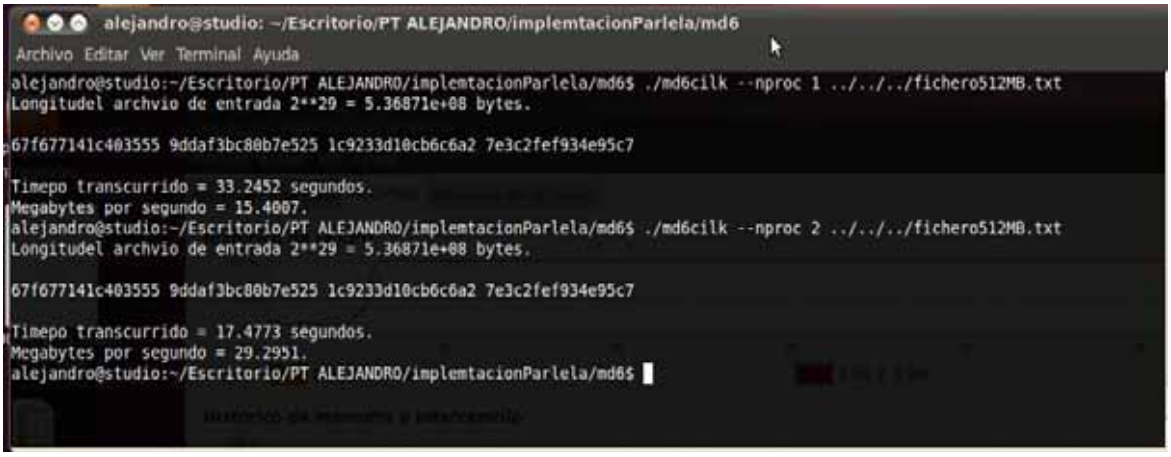
Versión 10.04 (lucid)
Núcleo Linux 2.6.32-21-generic
GNOME 2.30.2

Hardware

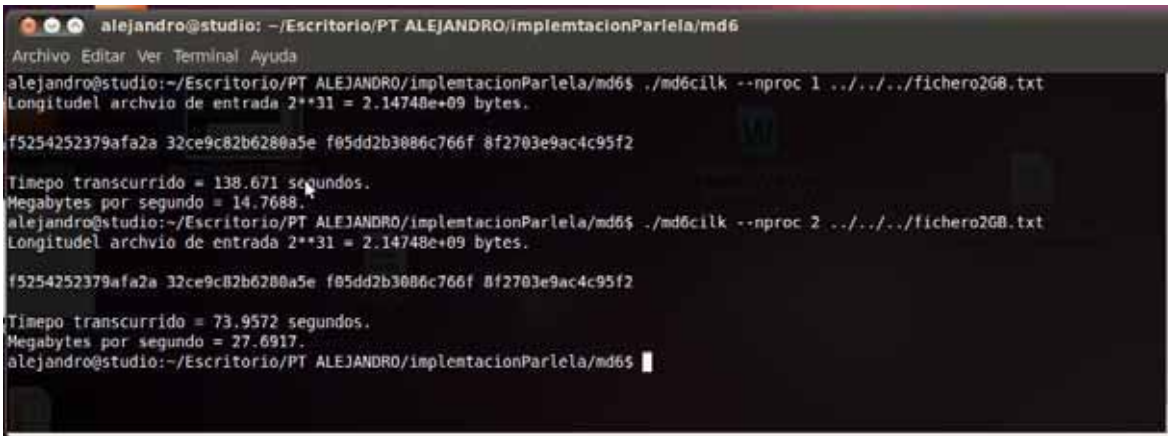
Memoria: 2.9 GiB
Procesador 0: Intel(R) Core(TM)2 Duo CPU T5850 @ 2.16GHz
Procesador 1: Intel(R) Core(TM)2 Duo CPU T5850 @ 2.16GHz

Se relazaron pruebas con archivos de 512MB y 2GB para 1 y 2 procesadores cada uno.

Prueba 1: Archivo de entrada de 512MB



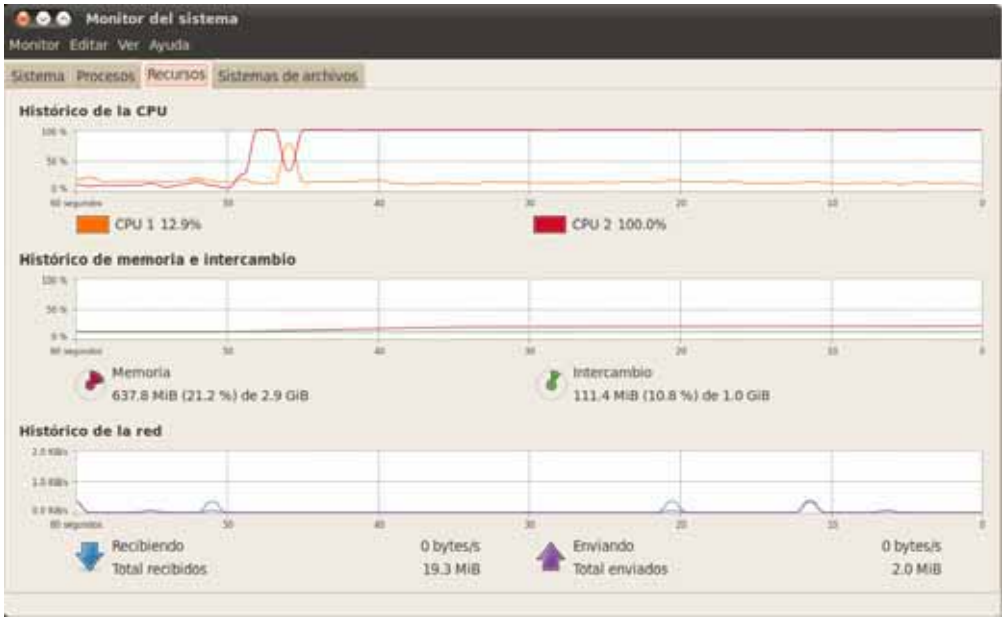
Prueba 2: Archivo de entrada de 2GB



Comportamiento de los hilos de ejecución en el monitor del sistema.

2 hilos de ejecución

1 hilo de ejecución.



Tiempos de ejecución

Archivo de 512 MB	
Procesadores	Tiempo Ejecución
1	33.24
2	17.47

Archivo de 2 GB	
Procesadores	Tiempo Ejecución
1	138.67
2	73.95

Conclusiones.

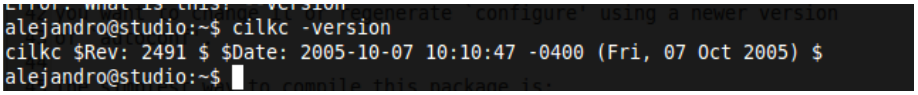
Con resultados obtenidos en las pruebas y con el equipo disponible se consideró que:

- Ya que es un algoritmo recursivo y que la función de compresión realiza el digesto para la entrada indicada, se pueden ejecutar bloques simultáneos.
- EL proceso empieza a ejecutarse una vez que se tiene en memoria la información, por lo que una de las desventajas es la memoria que se requiere para archivos grandes.
- Como se muestra en la figura del árbol, la ejecuciones pueden irse ejecutando de forma paralela y si se tiene un número significativo de procesadores el tiempo de ejecución se reduce
- Existirá un número finito N de procesadores para el cual el tiempo se mantenga constante esto debido a diferentes variables como lo es el tiempo de comunicación entre estos, y el número de fases pararlas que se puedan ejecutar a la vez.
- Para los tiempos de ejecución solo se pudo realizar en una máquina de 2 nucleas por lo que no se hizo un análisis más veraz en cuanto a número de procesadores
  Para el caso de dos núcleos fue un factor de la mitad de tiempo.

Manual de instalación de CILK y ejecución de MD6

Para la instalación de la extensión del lenguaje C CILK.

- Descomprimir el archivo cilk-5.4.6.tar.gz incluido en las fuentes o descargar de la página http://supertech.csail.mit.edu/cilk/
- Una vez generado el directorio ingresar y seguir los pasos
  - ./configure
  - make
  - make install
- Para validar la instalación correcta ejecutar cilkc –version
- Imprimiendo en pantalla la fecha y versión de esta algo similar a



Ejecución de proceso MD6

Una vez instalado los paquetes de cilk.

- Colocarse en el directorio donde se encuentran los fuentes
- Ejecutar el comando make que compilara los archivos y dependencias
- Se creara el archivo md6cilk que es el que se puede lanzar como proceso
- Lanzar el proceso de la siguinete forma
- $ ./md6cilk (--nprocnumProcesadores) [nombre del archivo]
- Si se omite el modificador –nproc por default tomara 1 procesador
- Al finalizar mostrar el digesto para el archivo.

Resumen

- make
- ./md6cilk (--nprocnumProcesadores) [nombre del archivo]

Bibliografía.

- Ronald L. Rivest; “The MD6 Hash Algoritm Report” http://groups.csail.mit.edu/cis/md6/docs/20090221md6report.

- Criptographic Hash AlgoritmCompetition http://csrc.nist.gov/groups/ST/hash/sha3/index.html

- Jonathan Katz & Yehuda Lindell. “Introduction To Modern Cryptography” Chapman& Hall, 1tedition 2007

Apéndice de notaciones.

Variable	*Default*	Uso
A	-	Vector de palabras utilizado en la definición de f
a	*rc + n*	Longitud del vector A utilizado en la definición de de f
B	-	Porción del bloque de datos de una entrada de la función de compresión
b	*64*	Numero de palabras en el arreglo B
c	*16*	Numero de palabras en la salida de la función de compresión
d	-	Numero de bits en la salida final de MD6 ($1 \leq d \leq 512$)
f	-	Función de compresión de MD6 que mapea W^n a W^c
f_Q	-	*f* sin prefijo suministrado, mapea W^{n-q} a W^c : $f_Q(x) = f(Q \,\|\|\, x)$
g	-	Función de difusión intra-palabra
K	*0*	La llave (una entrada de la función *f*)
k	*8*	Numero de palabras de entrada de compresión para la llave K
keylen	*0*	La longitud en bytes de la llave suministrada; $0 \leq keylen \leq kw\backslash 8$
l	-	Numero de nivel de un nodo de compresión.
l_i	-	Cantidad de desplazamientos a la izquierda para el paso *i* de la función de compresión
L	*64*	Parámetro de modo (numero de nivel máximo)
M	-	Mensaje de entrada a ser cifrado
m	-	Longitud del mensaje M de entrada en bits
N	-	Bloque de entrad en la función de compresión
n	*89*	Longitud de N en palabras
p	-	Numero de bits de relleno en el bloque de datos B
Q	-	Una aproximación a $\sqrt{6}$
q	*15*	Longitud de *Q*, en palabras
r	*40 + |d/4|*	Numero de rondas de 16 pasos en el cálculo de la función de compresión.
r_i	-	Cantidad de desplazamientos a la derecha para el paso *i* de la función de compresión
S_i	-	Una constante utilizada en el paso *i* en la función de compresión
S'_i	-	Secuencia de valores auxiliares utilizada para definir la secuencia S_i
t	*Rc*	Numero de pasos a calcular en la función de compresión
t_0	*17*	Primer posición tap
t_1	*18*	Segunda posición tap
t_2	*21*	Tercera posición tap
t_3	*31*	Cuarta posición tap
t_4	*67*	Quinta posición tap
t_5	*N*	Ultima posición tap
U	-	Nodo de id único (una palabra)

u	*1*	Longitud de *U*, en palabras
V	-	Palabra de control de la función de compresión
v	*1*	Longitud de V, en palabras
W	$\{0,1\}^w$	Conjunto de todas las palabras de w-bits
w	*64*	Numero de bits en un palabra
z	-	Bandera de un bit en V indicando la compresión final

SUMARIO CILK

Cilk es un lenguaje para implementación de algoritmos multiproceso. La filosofía detrás de Cilk es que el programador debe concentrarse en la estructuración del programa para exponer paralelismo y exploten la localidad, dejando el sistema de ejecución con la responsabilidad de la planificación del cálculo para ejecutar eficientemente en una plataforma determinada. Así, el sistema de ejecución Cilk se encarga de los detalles como el equilibrio de carga, la paginación, y protocolos de comunicación. A diferencia de otros lenguajes multi-hilo, Cilk es algorítmico, en la que el sistema de ejecución utiliza un planificador que permite la ejecución de programas para calcular con precisión, basada en medidas de complejidad abstracta.

Componentes de CILK

- Sistema de ejecución cilk
- Compilador cilk

Cilk corre bajos sistemas que soporten el estándar POSIX pthreads, en particular cilk correo bajos sistemas GNU.
En general cilk podría correr bajo sistemas basados con compilador GCC, POSIX pthread y MAKE de GNU estén disponibles

Programación en CILK

El lenguaje básico de cilk es simple, consiste en lenguaje C con la adición de 3 palabras clave para indicar el paralelismo y sincronización. Un programa en cilk cuando se ejecuta en un procesador tiene la misma semántica que un programa escrito en C, el resultado es que las palabras claves cilk son borradas.

Ejemplo

Programa que calcula la serie de Fibonacci recursivamente, escrito en leguaje c y con cilk.

```c
#include <stdlib.h>
#include <stdio.h>

int fib (int n){
    if (n<2) return (n);
    else{
        int x, y;
        x = fib (n-1);
        y = fib (n-2);
        return (x+y);
    }
}

int main (int argc, char *argv[]){

    int n, result;
    n = atoi(argv[1]);
    result = fib (n);
    printf ("Result: %d\n", result);

    return 0;
}
```

```c
#include <stdlib.h>
#include <stdio.h>

cilk int fib (int n){

    if (n<2) return n;
    else{
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
cilk int main (int argc, char *argv[]){

    int n, result;
    n = atoi(argv[1]);
    result = spawn fib(n);
    sync;
    printf ("Result: %d\n", result);

    return 0;
}
```

Nótese como los programas son muy similares, la única diferencia es la inclusión de las palabras claves cilk, spawn y sync.

Palabras clave

Cilk:Identifica un procedimiento *cilk*, que es la versión paralela de la función C. Un procedimiento *cilk* puede generar sub-procedimientos en paralelo y sincronizarlos a su finalización.

Spawn: Crea un hilo de ejecución en el procedimiento indicado como paralelo mediante la palabra cilk, continuando con la ejecución del hilo padre.

Sync: Se puede utilizar para esperar a que todos los procedimientos hijos generados antes de que el procedimiento actual regrese. Normalmente se utiliza para bloquear la ejecución del procedimiento actual hasta que todos los valores devueltos estén disponibles.

Compilación y ejecución de programas CILK

Para poder ejecutar programas en cilk es necesario tener instalado cilk, el cilk entiende programas con extensión cilk y acepta en general los mismos argumentos que el compilador de gcc.

Por ejemplo para compilar y ejecutar el código de ejemplo de Fibonacci puedes ejecutar el comando

$ cilkc -O2 fib.cilk -o fib

El cual producirá el archivo ejecutable fib, también podrás usar muchas de la opciones que tiene el compilador gcc, para ejecutar el programa deberás poner

> fib --nproc 4 30

Donde –nproc 4 indica que se ejecutara en un sistema con 4 procesadores y correrá en paralelo.

Durante el desarrollo del programa, es muy útil para recopilar datos de sobre el rendimiento de un programa cilk las banderas *-cilk-profile and -cilk-span.*

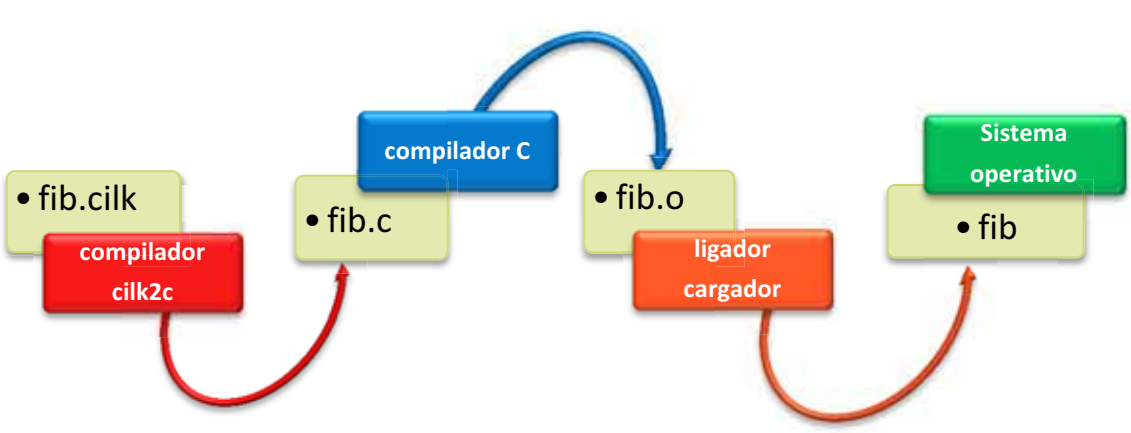
La bandrea *-cilk-profile* instruye a cilk para recopilar datos acerca de cuánto tiempo pasa cada procesador trabajando, como se producen las migraciones de muchos hilos, la cantidad de memoria asignada, etc.

La bandera *-cilk-span*permite la medición del span (Ruta crítica), span es el tiempo de ejecución ideal de su programa en número infinito de procesadores.

Para imprimir la información del perfil y el tiempo, puede usar la instrucción –stats X, donde X puede ser un número del 1 al 6

> fib --nproc 4 --stats 1 30

Proceso de Compilación



Código Fuente

md6_mode.c

```
#include <assert.h>
#include <stdio.h>
#include <string.h>

#include "md6.h"

/* MD6 constantes indepenndientes del modo de operacion de ( md6.h) */
#define w md6_w     /* # de bits en una palabra              (64) */
#define n md6_n     /* # de palabras en la entrada           (89) */
#define c md6_c     /* # de palabras para la salida          (16) */

/* MD6 constantes para le modo de operacion                       */
#define q md6_q     /* # de palabras en Q                    (15) */
#define k md6_k     /* # de palabras en key (aka salt)       (8)  */
#define u md6_u     /* # de palabras en nodo unico ID        (1)  */
#define v md6_v     /* # palbras en control                  (1)  */
#define b md6_b     /* # palbaras en los datos de compreison (64) */

/* macros utiles */
#ifndef min
#define min(a,b) ((a)<(b)? (a) : (b))
#endif
#ifndef max
#define max(a,b) ((a)>(b)? (a) : (b))
#endif


int md6_default_r( int d ){
  return 40 + (d/4);
}


/* 15 64-bit words */
static const md6_word Q[15] =
  {
    0x7311c2812425cfa0ULL,
    0x6432286434aac8e7ULL,
    0xb60450e9ef68b7c1ULL,
    0xe8fb23908d9f06f1ULL,
    0xdd2e76cba691e5bfULL,
    0x0cd0d63b2c30bc41ULL,
    0x1f8ccf6823058f8aULL,
    0x54e5ed5b88e3775dULL,
    0x4ad12aae0a6d6031ULL,
    0x3e7f16bb88222e0dULL,
    0x8af8671d3fb50c2cULL,
    0x995ad1178bd25c31ULL,
    0xc878c1dd04c4b633ULL,
    0x3b72066c7a1552acULL,
```

```
    0x0d6f3522631effcbULL,
  };



int md6_byte_order = 0;
/*
** 0 = desconosido
** 1 = little-endian
** 2 = big-endian
*/
#define MD6_LITTLE_ENDIAN (md6_byte_order == 1)
#define MD6_BIG_ENDIAN    (md6_byte_order == 2)

void md6_detect_byte_order( void ){
  md6_word x = 1 | (((md6_word)2)<<(w-8));
  unsigned char *cp = (unsigned char *)&x;
  if ( *cp == 1 )        md6_byte_order = 1;      /* little-endian */
  else if ( *cp == 2 )   md6_byte_order = 2;      /* big-endian    */
  else                   md6_byte_order = 0;      /* desconocido       */
}

md6_word md6_byte_reverse( md6_word x ){
    #define mask8  ((md6_word)0x00ff00ff00ff00ffULL)
    #define mask16 ((md6_word)0x0000ffff0000ffffULL)
    #if (w==64)
      x = (x << 32) | (x >> 32);
    #endif
    #if (w >= 32)
      x = ((x & mask16) << 16) | ((x & ~mask16) >> 16);
    #endif
    #if (w >= 16)
      x = ((x & mask8) << 8) | ((x & ~mask8) >> 8);
    #endif
      return x;
}

void md6_reverse_little_endian( md6_word *x, int count ){
  int i;
  if (MD6_LITTLE_ENDIAN)
    for (i=0;i<count;i++)
      x[i] = md6_byte_reverse(x[i]);
}

// agrega una cadena de bits a otra
void append_bits( unsigned char *dest, unsigned int destlen,
          unsigned char *src,  unsigned int srclen ){
  int i, di, accumlen;
  uint16_t accum;
  int srcbytes;

  if (srclen == 0) return;

  /* Iniciliza accum, accumlen, y di */
  accum = 0;
  accumlen = 0;
  if (destlen%8 != 0)
```

```
    { accumlen = destlen%8;
      accum = dest[destlen/8];
      accum = accum >> (8-accumlen);
    }
  di = destlen/8;


  srcbytes = (srclen+7)/8;
  for (i=0;i<srcbytes;i++)
    {
      if (i != srcbytes-1)
    { accum = (accum << 8) ^ src[i];
      accumlen += 8;
    }
      else
    { int newbits = ((srclen%8 == 0) ? 8 : (srclen%8));
      accum = (accum << newbits) | (src[i] >> (8-newbits));
      accumlen += newbits;
    }
      while ( ( (i != srcbytes-1) & (accumlen >= 8) ) ||
          ( (i == srcbytes-1) & (accumlen > 0) ) )
    { int numbits = min(8,accumlen);
      unsigned char bits;
      bits = accum >> (accumlen - numbits);
      bits = bits << (8-numbits);
      bits &= (0xff00 >> numbits);
      dest[di++] = bits;
      accumlen -= numbits;
    }
    }
}


//inicializa la estrcutura para el claculo
int md6_full_init( md6_state *st,       /* estructura principal */
           int d,                       /* longitud de slaida */
           unsigned char *key,        /* llave */
           int keylen,     /* longitud de la llave en bytes */
           int L,           /* modificador */
           int r                        /* numero de rondas */
           ){

  if (st == NULL) return MD6_NULLSTATE;
  if ( (key != NULL) && ((keylen < 0) || (keylen > k*(w/8))) )
    return MD6_BADKEYLEN;
  if ( d < 1 || d > 512 || d > w*c/2 ) return MD6_BADHASHLEN;

  md6_detect_byte_order();
  memset(st,0,sizeof(md6_state));  /* setea estructura en cero */
  st->d = d;
  if (key != NULL && keylen > 0)
    { memcpy(st->K,key,keylen);
      st->keylen = keylen;
      md6_reverse_little_endian(st->K,k);
    }
  else
    st->keylen = 0;
```

```
  if ( (L<0) | (L>255) ) return MD6_BAD_L;
  st->L = L;
  if ( (r<0) | (r>255) ) return MD6_BAD_r;
  st->r = r;
  st->initialized = 1;
  st->top = 1;
  if (L==0) st->bits[1] = c*w;
  compression_hook = NULL;
  return MD6_SUCCESS;
}

/* Estado de inicilizacion con parametros pro default
**
*/

int md6_init( md6_state *st,
          int d
          ){

return md6_full_init(st,
               d,
               NULL,
               0,
               md6_default_L,
               md6_default_r(d)
               );
}


/* comprime un bloqe en el nivel ell
** entradas:
**     st         estructra del estado eactual de MD6
**     ell        numero de nivel
**     z          bandera de compresion final
** Salida:
**     C   digesto
*/

int md6_compress_block( md6_word *C,
            md6_state *st,
            int ell,
            int z
            ){
  int p, err;

  if ( st == NULL) return MD6_NULLSTATE;
  if ( st->initialized == 0 ) return MD6_STATENOTINIT;
  if ( ell < 0 ) return MD6_STACKUNDERFLOW;
  if ( ell >= md6_max_stack_height-1 ) return MD6_STACKOVERFLOW;

  st->compression_calls++;

  if (ell==1)
    { if (ell<(st->L + 1))
    md6_reverse_little_endian(&(st->B[ell][0]),b);
      else
    md6_reverse_little_endian(&(st->B[ell][c]),b-c);
```

```
    }

  p = b*w - st->bits[ell];

  err =
    md6_standard_compress(
      C,
      Q,
      st->K,
      ell, st->i_for_level[ell],
      st->r, st->L, z, p, st->keylen, st->d,
      st->B[ell]
              );
  if (err) return err;

  st->bits[ell] = 0;
  st->i_for_level[ell]++;

  memset(&(st->B[ell][0]),0,b*sizeof(md6_word));
  return MD6_SUCCESS;
}


int md6_process( md6_state *st,
         int ell,
         int final ){
  int err, z, next_level;
  md6_word C[c];

  if ( st == NULL) return MD6_NULLSTATE;
  if ( st->initialized == 0 ) return MD6_STATENOTINIT;

  if (!final)
    {
      if ( st->bits[ell] < b*w )
    return MD6_SUCCESS;
    }
  else
    { if ( ell == st->top )
    { if (ell == (st->L + 1))
        { if ( st->bits[ell]==c*w && st->i_for_level[ell]>0 )
        return MD6_SUCCESS;
        }
           else {
         if ( ell>1 && st->bits[ell]==c*w)
         return MD6_SUCCESS;
         }
    }

    }

  z = 0; if (final && (ell == st->top)) z = 1;
  if ((err = md6_compress_block(C,st,ell,z)))
      return err;
  if (z==1)
      memcpy( st->hashval, C, md6_c*(w/8) );
```

```
  next_level = min(ell+1,st->L+1);
  if (next_level == st->L + 1 && st->i_for_level[next_level]==0)
    st->bits[next_level] += c*w;
  /* copia C en el siguiente nivel */
  memcpy((char *)st->B[next_level] + st->bits[next_level]/8,
     C,
     c*(w/8));
  st->bits[next_level] += c*w;
  if (next_level > st->top) st->top = next_level;

  return md6_process(st,next_level,final);
}

// Actuliza la cadena del calculo
int md6_update( md6_state *st,
        unsigned char *data,
        uint64_t databitlen ){
  unsigned int j, portion_size;
  int err;

  /* check that input values are sensible */
  if ( st == NULL ) return MD6_NULLSTATE;
  if ( st->initialized == 0 ) return MD6_STATENOTINIT;
  if ( data == NULL ) return MD6_NULLDATA;

  j = 0; /* j = numero d ebits procesados hasta la actualizacion */
  while (j<databitlen){
      portion_size = min(databitlen-j,
             (unsigned int)(b*w-(st->bits[1])));

      if ((portion_size % 8 == 0) &&
      (st->bits[1] % 8 == 0) &&
      (j % 8 == 0))
    {
      memcpy((char *)st->B[1] + st->bits[1]/8,
         &(data[j/8]),
         portion_size/8);
    }
      else
    { append_bits((unsigned char *)st->B[1],
            st->bits[1],
            &(data[j/8]),
            portion_size);
    }
      j += portion_size;
      st->bits[1] += portion_size;
      st->bits_processed += portion_size;

      /* comprime el bloque level-1 si esta lleno */
      if (st->bits[1] == b*w && j<databitlen)
    { if ((err=md6_process(st,
                1,
                0
                )))
        return err;
    }
    }
```

```
  return MD6_SUCCESS;
}


//convierte el valor del digesto en hexadecimal
int md6_compute_hex_hashval( md6_state *st ){ int i;
  static unsigned char hex_digits[] = "0123456789abcdef";

  if ( st == NULL ) return MD6_NULLSTATE;

  for (i=0;i<((st->d+7)/8);i++)
    { st->hexhashval[2*i]
    = hex_digits[ ((st->hashval[i])>>4) & 0xf ];
      st->hexhashval[2*i+1]
    = hex_digits[ (st->hashval[i]) & 0xf ];
    }

  /*  agerga ceros a la cadena al final*/
  st->hexhashval[(st->d+3)/4] = 0;
  return MD6_SUCCESS;
}

// extrae los ultimos d bits de la cadena
void trim_hashval(md6_state *st){
  int full_or_partial_bytes = (st->d+7)/8;
  int bits = st->d % 8;
  int i;

  /* mouve los bytes mas relevantesal incio */
  for ( i=0; i<full_or_partial_bytes; i++ )
    st->hashval[i] = st->hashval[c*(w/8)-full_or_partial_bytes+i];

  /* agrega bytes de ceros */
  for ( i=full_or_partial_bytes; i<c*(w/8); i++ )
    st->hashval[i] = 0;

  if (bits>0)
    { for ( i=0; i<full_or_partial_bytes; i++ )
    { st->hashval[i] = (st->hashval[i] << (8-bits));
      if ( (i+1) < c*(w/8) )
        st->hashval[i] |= (st->hashval[i+1] >> bits);
    }
    }
}

//procesamiento final
/* Do final processing to produce md6 hash value
** Entrada:
**     st              estructura principal
** Salida:
**     mesnaje digesto
*/
int md6_final( md6_state *st , unsigned char *hashval){
  int ell, err;

  /* check that input values are sensible */
  if ( st->initialized == 0 ) return MD6_STATENOTINIT;
```

```
  if ( st->finalized == 1 ) return MD6_SUCCESS;

  if (st->top == 1) ell = 1;
  else for (ell=1; ell<=st->top; ell++)
     if (st->bits[ell]>0) break;
  err = md6_process(st,ell,1);
  if (err) return err;

  md6_reverse_little_endian( (md6_word*)st->hashval, c );
  if (hashval != NULL) memcpy( hashval, st->hashval, (st->d+7)/8 );
  trim_hashval( st );
  md6_compute_hex_hashval( st );

  st->finalized = 1;
  return MD6_SUCCESS;
}


int md6_full_hash( int d,                          /* longitud de salida */
           unsigned char *data,                         /* datos */
           uint64_t databitlen,  /* longitud del los dtaos en bits */
           unsigned char *key,                              /* llave */
           int keylen,                 /* (in bytes) OK to give 0 */
           int L,                                  /* modificador */
           int r,                             /* numero de rondas */
           unsigned char *hashval                          /* salida */
           ){
  md6_state st;
  int err;

  err = md6_full_init(&st,d,key,keylen,L,r);
  if (err) return err;
  err = md6_update(&st,data,databitlen);
  if (err) return err;
  md6_final(&st,hashval);
  if (err) return err;
  return MD6_SUCCESS;
}

int md6_hash( int d,
              unsigned char *data,
          uint64_t databitlen,
          unsigned char *hashval
         )
{ int err;

  err = md6_full_hash(d,data,databitlen,
              NULL,0,md6_default_L,md6_default_r(d),hashval);
  if (err) return err;
  return MD6_SUCCESS;
}
```

md6_compress.c

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

#include "md6.h"

/* maco utilis para maximo y minimo */
#ifndef min
#define min(a,b) ((a)<(b)? (a) : (b))
#endif
#ifndef max
#define max(a,b) ((a)>(b)? (a) : (b))
#endif

/*  definicion de varibales principales segun la defincion*/
#define   w   md6_w  /* # de bits en una palabra (64) */
#define   n   md6_n  /* # de palbras en la funcion de compresion (89) */
#define   c   md6_c  /* # de palbaras en la salida (16) */
#define   b   md6_b  /* # de palbaras axiliares en le mensaje*/
#define   v   md6_v  /* # de palbras para la palbra de control (1) */
#define   u   md6_u  /* # de palabras para el nodo de procesamiento (1)
*/
#define   k   md6_k  /* # de palabras para la llave (8) */
#define   q   md6_q  /* # de palabras para la constante (15) */


/* "Tap positions" */

#if (n==89)
#define  t0   17
#define  t1   18
#define  t2   21
#define  t3   31
#define  t4   67
#define  t5   89
#endif

#define RL00 loop_body(10,11)
#define RL01 loop_body( 5,24)
#define RL02 loop_body(13, 9)
#define RL03 loop_body(10,16)
#define RL04 loop_body(11,15)
#define RL05 loop_body(12, 9)
#define RL06 loop_body( 2,27)
#define RL07 loop_body( 7,15)
#define RL08 loop_body(14, 6)
#define RL09 loop_body(15, 2)
#define RL10 loop_body( 7,29)
#define RL11 loop_body(13, 8)
#define RL12 loop_body(11,15)
#define RL13 loop_body( 7, 5)
#define RL14 loop_body( 6,31)
#define RL15 loop_body(12, 9)
```

```
const md6_word S0 = (md6_word)0x0123456789abcdefULL;
const md6_word Smask = (md6_word)0x7311c2812425cfa0ULL;


/* funcion de compresion segun el numero de rondas
    en esta parte procesa el arreglo A
*/


void md6_main_compression_loop( md6_word* A , int r ){

  md6_word x, S;
  int i,j;

  S = S0;
  for (j = 0, i = n; j<r*c; j+=c)
    {

/* ********************************************************************* */
#define loop_body(rs,ls)                        \
      x = S;                                    \
      x ^= A[i-t5];                             \
      x ^= A[i-t0];                             \
      x ^= ( A[i-t1] & A[i-t2] );               \
      x ^= ( A[i-t3] & A[i-t4] );               \
      x ^= (x >> rs);                           \
      A[i++] = x ^ (x << ls);
/* ********************************************************************* */

      /*
      ** bucle c=16 times. (Una "ronda" por calculo.)
      ** La cantidad de desplazamientos estan en la s macros RLnn
      */
      RL00 RL01 RL02 RL03 RL04 RL05 RL06 RL07
      RL08 RL09 RL10 RL11 RL12 RL13 RL14 RL15

      /* Constante de ronda para cada una. */
      S = (S << 1) ^ (S >> (w-1)) ^ (S & Smask);

    }
}

// comprime n palbras de  entrada alamcenando la salida final en C

int md6_compress( md6_word *C,
          md6_word *N,
          int r,
          md6_word *A
         ){

  md6_word* A_as_given = A;

  /* valida las entradas */
  if ( N == NULL) return MD6_NULL_N;
  if ( C == NULL) return MD6_NULL_C;
  if ( r<0 || r > md6_max_r) return MD6_BAD_r;
```

```
  if ( A == NULL) A = calloc(r*c+n,sizeof(md6_word));
  if ( A == NULL) return MD6_OUT_OF_MEMORY;

  memcpy( A, N, n*sizeof(md6_word) );    /*  copia entrada N a auxiliar A
*/

  md6_main_compression_loop( A, r );          /* procesamiento */

  memcpy( C, A+(r-1)*c+n, c*sizeof(md6_word) ); /* copia ultimo nivel en
la variable salida C */

  if ( A_as_given == NULL ){
     memset(A,0,(r*c+n)*sizeof(md6_word));
      free(A);
  }

  return MD6_SUCCESS;
}

//construye la palabra de control
/* ConstruConstruct control word V for given inputs.
 entradas:
   r = numero de rondas
   L = parametrizacion de la altura del arbol
   z = 1 si el la ultima compresion
   p = numeor de bits de relleno en el bloque de compresion number of pad
bits in a block to be compressed
   keylen = numeor d ebytes para la llave
   d = lngitud de salida
 Regeresa:
   V = palaba de control
*/

md6_control_word md6_make_control_word( int r,
                    int L,
                    int z,
                    int p,
                    int keylen,
                    int d
                    ){
  md6_control_word V;
  V = ( (((md6_control_word) 0) << 60) |           /* tamano  4 bits */
    (((md6_control_word) r) << 48) |           /* tamano 12 bits */
    (((md6_control_word) L) << 40) |           /* tamano  8 bits */
    (((md6_control_word) z) << 36) |           /* tamano  4 bits */
    (((md6_control_word) p) << 20) |           /* tamano 16 bits */
    (((md6_control_word) keylen) << 12 ) |     /* tamano  8 bits */
        (((md6_control_word) d)) );                /* tamano 12 bits */
  return V;
}

// para la identificacion de los nodos
/* crea un  "nodo unico" U basado en el ivel y posicon
** dentro dle nivel;
** Entardas:
**    dest = address of where nodeID U should be placed
**    ell = numero de nievl
```

```
**    i = indice
** Returns
**    U =  nodo construido
*/

md6_nodeID md6_make_nodeID( int ell,   /* numero de nivel */
                  int i    /* indice dentro de el nivel */
                ){
  md6_nodeID U;
  U = ( (((md6_nodeID) ell) << 56) |
    ((md6_nodeID) i) );
  return U;
}

//ensamla los componentes de entrada
//empaqueta los datso para la compresion en N
void md6_pack( md6_word*N,
           const md6_word* Q,
           md6_word* K,
           int ell, int i,
           int r, int L, int z, int p, int keylen, int d,
           md6_word* B ){
  int j;
  int ni;
  md6_nodeID U;
  md6_control_word V;

  ni = 0;

  for (j=0;j<q;j++) N[ni++] = Q[j];

  for (j=0;j<k;j++) N[ni++] = K[j];

  U = md6_make_nodeID(ell,i);

  memcpy((unsigned char *)&N[ni],
     &U,
     min(u*(w/8),sizeof(md6_nodeID)));
  ni += u;

  V = md6_make_control_word(
           r,L,z,p,keylen,d);

  memcpy((unsigned char *)&N[ni],
     &V,
     min(v*(w/8),sizeof(md6_control_word)));
  ni += v;

  memcpy(N+ni,B,b*sizeof(md6_word));
}

//Compresion estandar
/* procesa el bloque usnado la etradas estandar
** entradas:
**     Q              contante
**     K              llave
**     ell            numero de nivel
```

```
**      i               indice dentro dle nivel
**      r               numeor de rondas
**      L               parametro de niveles
**      z               1 si es la ultima compresion
**      p               numero de nits d erelleno
**      keylen          bytes para la llave
**      d               longitud de salida deseada
**      B               bloque (64-palbaras) entrada
** Se modifica:
**      C               salida de 16
*/
int md6_standard_compress( md6_word* C,
                const md6_word* Q,
                md6_word* K,
                int ell, int i,
                int r, int L, int z, int p, int keylen, int d,
                md6_word* B
                ){
  md6_word N[md6_n];
  md6_word A[r*c+n];

  /* validacionde la entradas */
  if ( (C == NULL) ) return MD6_NULL_C;
  if ( (B == NULL) ) return MD6_NULL_B;
  if ( (r<0) | (r>md6_max_r) ) return MD6_BAD_r;
  if ( (L<0) | (L>255) ) return MD6_BAD_L;
  if ( (ell < 0) || (ell > 255) ) return MD6_BAD_ELL;
  if ( (p < 0) || (p > b*w ) ) return MD6_BAD_p;
  if ( (d <= 0) || (d > c*w/2) ) return MD6_BADHASHLEN;
  if ( (K == NULL) ) return MD6_NULL_K;
  if ( (Q == NULL) ) return MD6_NULL_Q;

  /* empaqueta las entradas en N */
  md6_pack(N,Q,K,ell,i,r,L,z,p,keylen,d,B);

  /* llama a la compresion de prueba si se define para checar
fucnionamiento  */
  if (compression_hook != NULL)
    compression_hook(C,Q,K,ell,i,r,L,z,p,keylen,d,B);

  return md6_compress(C,N,r,A);
}
```

md6.h

```
#ifndef MD6_H_INCLUDED
#define MD6_H_INCLUDED

#include <inttypes.h>

#define   md6_w    64


#if (md6_w==64)                    /* palabra standar md6 */
typedef uint64_t md6_word;
#define PR_MD6_WORD "%.16" PRIx64

#elif (md6_w==32)                  /* variante de 32 */
typedef uint32_t md6_word;
#define PR_MD6_WORD "%.8" PRIx32

#elif (md6 w==16)                  /* variante de 16 */
typedef uint16_t md6_word;
#define PR_MD6_WORD "%.4" PRIx16

#elif (md6_w==8)                   /* variante de 8 */
typedef uint8_t md6_word;
#define PR_MD6_WORD "%.2" PRIx8

#endif


#define md6_n      89    /* tamaño del bloque de entrada, en palabras  */
#define md6_c      16    /* tamaño del bloque de salida, en palabras   */
#define md6_max_r 255    /* numero maximo de rondas */

extern int md6_default_r( int d );    /* valor de numeor de rondas
estnadar */

void md6_main_compression_loop( md6_word *A,          /* arreglo d
etrabajo */
                int r                 /* numero de rondas */
                );

int md6_compress( md6_word *C,                                /* salida D
*/
          md6_word *N,                                /* entrada  */
          int r,                             /* nuemro de rondas */
          md6_word *A /* Arreglo detrabajo */
                );


typedef uint64_t md6_control_word;                    /* (r,L,z,p,d) */
md6_control_word md6_make_control_word( int r,          /* nuemro de
rondas */
                    int L,          /* numero de fases */
                    int z,          /* bandera de procesamiento final */
                    int p,          /* bits de relleno */
                    int keylen,     /* longitud de bytes en la llave */
```

```
                    int d           /* longitud de digesto */
                    );

typedef uint64_t md6_nodeID;                                    /* (ell,i) */
md6_nodeID md6_make_nodeID( int ell,                    /* numero de nivel
*/
                int i    /* indice dntro del nivel */
                );

void md6_pack( md6_word* N,                                      /* output
*/
           const md6_word* Q,                          /* constante */
           md6_word* K,                                      /* llave */
           int ell, int i,                                   /* para el
ndo */
           int r, int L, int z, int p, int keylen, int d,   /*
modificadores */
           md6_word* B                                       /* entrada */
           );

int md6_standard_compress(
        md6_word *C,                                        /* salida */
    const md6_word *Q,                          /* constante */
    md6_word *K,                                        /* llave */
    int ell, int i,                                     /* para el nodo */
    int r, int L, int z, int p, int keylen, int d,  /* modificadores */
    md6_word* B                                         /* entrada */
               );

// MD6 modo deopreaicon definido en md6_mode.c

/* MD6 constantes relacionadas al modo de operacion                  */

#define md6_q 15         /* # de palbaras para la constante Q          */
#define md6_k  8         /* # de palbaras por la llave                 */
#define md6_u (64/md6_w) /* # de palabras par el nodo                  */
#define md6_v (64/md6_w) /* # de palabras para la palabra de control   */
#define md6_b 64         /* # de palbars para el bloque de compresion  */

#define md6_default_L 64    /* modificador de niveles */

//determina el numero maximo de bits que pueden ser procesados con el
modificador L
#define md6_max_stack_height 29



//md6_state es la estructura para las variables principales del
procesamiento
typedef struct {

  int d;           /* longitud de digesto. 1 <= d <= 512.      */
  int hashbitlen;  /* varible segun la definicionde NIST       */

  unsigned char hashval[ (md6_c/2)*(md6_w/8) ];
      /* contiene le valor del digesto despues de la llamada ala funcion
md6_final */
```

```
            /* ejemplo  hashval[64];  (asume d<=c/2.)
*/

  unsigned char hexhashval[(md6_c*(md6_w/8))+1];

  int initialized;         /* cero, uno despues de que se llama a
md6_init */
  uint64_t bits_processed;                /* bits procesados */
  uint64_t compression_calls;    /* numero de fucniones de compresion
realizadas*/
  int finalized;          /* cero, y uno que se llama md6_final */

  md6_word K[ md6_k ];

  int keylen;

  int L;

  int r;

  int top;

  md6_word B[ md6_max_stack_height ][ md6_b ];

  unsigned int bits[ md6_max_stack_height ];
  // bits[ell] =     nuemro de bits listos en B[ell]

  uint64_t i_for_level[ md6_max_stack_height ];

} md6_state;

//MD6 declaracion de funcione sprincipales
//estan definidas en  md6_mode.c

extern int md6_init( md6_state *st,             /* inicializa
estructura*/
             int d                          /* longitud resumen */
             );

extern int md6_full_init( md6_state *st,        /*inicializa estructura*/
              int d,                    /* longitud resumen */
              unsigned char *key,                 /* llave */
              int keylen,       /* longitud llave en bytes */
              int L,                  /* modo d eopreacion */
              int r                     /* numero de rondas */
              );

extern int md6_update( md6_state *st,             /* eststus */
               unsigned char *data,           /* datos */
               uint64_t databitlen       /* longitud de datos */
               );

extern int md6_final( md6_state *st,            /* estatus */
              unsigned char *hashval       /* salida  */
              );

//funciones de procesamiento
```

```
extern int md6_hash( int d,                          /* longitud de salida */
             unsigned char *data,          /* bloque de entrada */
             uint64_t databitlen, /* numero de bits del bloque */
             unsigned char *hashval                  /* salida */
             );

extern int md6_full_hash( int d,                     /* longitud de salida
*/
              unsigned char *data,/* bloque de entrada */
              uint64_t databitlen,/* numero de bits del bloque */
              unsigned char *key,       /* llave */
              int keylen,       /* longitud de llave en bytes */
              int L,     /* modificador */
              int r,                    /* numero d erondas */
              unsigned char *hashval             /* salida */
              );


/* MD6 codigos de retorno segub API de NIST
** SUCCESS, FAIL, y BADHASHLEN
*/

/* SUCCESS:  */
#define MD6_SUCCESS 0

/* ERROR CODES: */
#define MD6_FAIL 1           /* cuqlquir prblema
*/
#define MD6_BADHASHLEN 2     /* hashbitlen<1 or >512 bits
*/
#define MD6_NULLSTATE 3      /* estado nulo de MD6
*/
#define MD6_BADKEYLEN 4      /* longitu de la llave <0 or >512 bits
*/
#define MD6_STATENOTINIT 5   /* estado no se inicializo
*/
#define MD6_STACKUNDERFLOW 6 /* */
#define MD6_STACKOVERFLOW 7  /* mensajemuy largo                    */
#define MD6_NULLDATA 8       /* apuntador nulo
*/
#define MD6_NULL_N 9         /* compresion: N es nulo
*/
#define MD6_NULL_B 10        /* compresion estandar: apuntado a B nulo
*/
#define MD6_BAD_ELL 11       /* compresion estandar: ell no esta en
{0,255}     */
#define MD6_BAD_p 12         /* compresion estandar: relleno inavlido
*/
#define MD6_NULL_K 13        /* compresion estandar: K es nulo
*/
#define MD6 NULL Q 14        /* compresion estandar: Q es nulo
*/
#define MD6_NULL_C 15        /* compresion estandar: C es nulo
*/
#define MD6_BAD_L 16         /* compresion estandar: L <0 o > 255
*/
```

```
                              /* md6_init: L<0 o L>255
*/
#define MD6_BAD_r 17          /* compress: r<0 o r>255
*/
                              /* md6_init: r<0 o r>255
*/
#define MD6_OUT_OF_MEMORY 18 /* compress: fallo de memoria
*/


/* debugeo y pruebas.
*/

void (* compression_hook)(md6_word *C,
              const md6_word *Q,
              md6_word *K,
              int ell,
              int i,
              int r,
              int L,
              int z,
              int p,
              int keylen,
              int d,
              md6_word *N
              );

#endif
```

md6.cilk

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <time.h>
#include <math.h>
#include "md6.h"

int zk;     /* numero de niveles */
int zn;     /* 2**zn es el numero de bytes de entrada en el nivel 0,
donde zn=7+2*zk */
char * A[30];   /* arreglo de apuntadaores para cada nivel    */
                /* usandose desde A[0...zk]
*/
                /* A[0] apunta a la  entrada
*/
        /* A[i+1] apunta a la version comprimida de  A[i]        */
        /* A[zk] contien el resultado final (de tamaño 128 bytes)
*/
uint64_t N[30]; /* tamaño de cada arreglo; desciend en potencias de 4
*/
                /* N[i] = 2**(zn-2*i)                                  */

md6_word Q[15] =   { /* sqrt(6)-2, en hexadecimal */
    0x7311c2812425cfa0ULL, 0x6432286434aac8e7ULL, 0xb60450e9ef68b7c1ULL,
    0xe8fb23908d9f06f1ULL, 0xdd2e76cba691e5bfULL, 0x0cd0d63b2c30bc41ULL,
    0x1f8ccf6823058f8aULL, 0x54e5ed5b88e3775dULL, 0x4ad12aae0a6d6031ULL,
    0x3e7f16bb88222e0dULL, 0x8af8671d3fb50c2cULL, 0x995ad1178bd25c31ULL,
    0xc878c1dd04c4b633ULL, 0x3b72066c7a1552acULL, 0x0d6f3522631effcbULL,
  };

md6_word K[8];  /* llave cero */

#define w  md6_w
#define c  md6_c
#define n  md6_n
#define b  md6_b
#define u  md6_u
#define v  md6_v
#define k  md6_k
#define q  md6_q

/* invierte la cadena*/
md6_word md6_byte_reverse( md6_word x ){
    #define mask8  ((md6_word)0x00ff00ff00ff00ffULL)
    #define mask16 ((md6_word)0x0000ffff0000ffffULL)
      x = (x << 32) | (x >> 32);
      x = ((x & mask16) << 16) | ((x & ~mask16) >> 16);
      x = ((x & mask8) << 8) | ((x & ~mask8) >> 8);
      return x;
}

/* incializa las variables para poder ejecutarse*/
```

```
void setup(int kk,char *nombreArchivo){

   FILE* file;
   //char chr;
   size_t size;
   int i;
   md6_word *B;
   zk = kk;
   zn = 7 + 2*zk;
   size = ((size_t)1)<<zn;
   for (i=0;i<=zk;i++)
     {
       N[i] = size;
       A[i] = malloc(N[i]);
       assert(A[i]);
       size = size/4;
     }

    file=fopen(nombreArchivo, "rb"); // abro el archivo de solo lectura.

   /* coloaca en  A[0] los bytes del archivo de entrada */
   for (i=0;i<N[0];i++){
     //A[0][i] = 0x11 * (1 + (i%7));
    //fread(A[0],1,1,file);
    //if(feof(file) == 0)
    //  chr = fgetc(file);
    //else
    //  chr = '0';
    //A[0][i] = chr;
    A[0][i] = fgetc(file);
   }
    fclose(file);

   B = (md6_word *)(A[0]);
   for (i=0;i<N[0]/8;i++)
     B[i] = md6_byte_reverse(B[i]);

}

/* funcion de compresion */

void compress_block(int ell, size_t lo){
/* comprime un bloque indicado por el nivel ell, incio en el byte de
posicion lo
   Guarda en el nivel ell+1, posicion del byte lo/4
   llama a la funcion  md6_standard_compress        */
   int err;
   int d = 256;
   int r = 104;
   int L = 64;
   int z = ((ell==zk-1)?1:0);
   int p = 0;
   int keylen = 0;
   err =
     md6_standard_compress((md6_word *)&(A[ell+1][lo/4]), // destino
               Q,                                 // constante
               K,                                 // llave
```

```
                ell+1, lo/512,              // bloque
                            r,L,z,p,keylen,d,          //modificadores
                (md6_word *)&(A[ell][lo])     // bloque de entrada
               ) ;
   if (err) { printf("Error %d\n",err); fflush(stdout); }
}



/* se aplica la funcion hash al bloque inidcado y se alamacenqa en el
nivel ell+1
   el bloque de dtaos a procesar es a[ell][lo..hi-1]
   Esta parte es la que se definio recursivamnte para poder aplicar el
paraleleismo
   nota: se debe explicar en el reporte del PT muy bien
*/
cilk void hash_segment(int ell, size_t lo, size_t hi){
  if ((hi-lo)==512) /* tamaño de bloque de entrada */
    compress_block(ell,lo);
  else
  { size_t mid = (lo+hi)/2;
    spawn hash_segment(ell,lo,mid);
    spawn hash_segment(ell,mid,hi);
    sync;
  }
}


int getLevels(char *fileName){

    float tamanio = 0;
    int zkk = 0;
    FILE* file;

    //printf("Obteniendo numero de niveles\n");

    file=fopen(fileName, "rb"); // abro el archivo de solo lectura.
    fseek(file, 0L, SEEK_END); // me ubico en el final del archivo.
    tamanio=ftell(file);       // obtengo su tamanio en BYTES.
    fclose(file);              // cierro el archivo.

    zkk = ceilf((log2f(tamanio)-7)/2);

    //printf("\nzk: %i\n",zkk);

    return zkk;
}

float getNumOfBytes(char *nombreArchivo){

    float tamanio = 0;
    FILE* file;

    //printf("Obteniedno numero de bytes\n");

    file=fopen(nombreArchivo, "rb"); // abro el archivo de solo lectura.
```

```
    fseek(file, 0L, SEEK_END); // me ubico en el final del archivo.
    tamanio=ftell(file);       // obtengo su tamanio en BYTES.
    fclose(file);              // cierro el archivo.

    //printf("\ntamanio: %f\n",tamanio);
    //tamanio = tamanio/1024/1024/1024;
    //printf("\ntamanio: %f\n",tamanio);

    return tamanio;
}

/*
    Indicar en reporte el sumario de CILK y sus palabras reservadas como
aplican
*/

cilk int main(int argc, char **argv)
{ int ell;
  int i;
  Cilk_time start_time, elapsed_time;
  double seconds;

  // funcuin setup que inicializa variables
  zk = getLevels(argv[1]);
  setup(zk,argv[1]);
  printf("Longitudel archvio de entrada 2**%d = %g
bytes.\n",7+2*zk,(double)N[0]);

  //incializa timer de procesamiento
  start_time = Cilk_get_wall_time();
  for (ell=0;ell<zk;ell++){
      spawn hash_segment(ell,0,N[ell]);
      sync;
  }

  /* imprime el resultado , tiempo de jecucion , y velocidad de
procesamiento */
  printf("\n");
  for (i=96;i<128;i+=8)
    printf( PR_MD6_WORD " ",*(md6_word *)&(A[zk][i]));
  elapsed_time = Cilk_get_wall_time() - start_time;
  seconds = Cilk_wall_time_to_sec(elapsed_time);
  printf("\n\nTimepo transcurrido = %g segundos.\n",seconds);
  printf("Megabytes por segundo = %g.\n", (N[0] / (1024*1024)) /
seconds);

  return 0;
}
```

makefile

```
#para quitar el error de ejecución ejecutar en SO
#export MALLOC_CHECK_=0

CC = gcc
CILKC = cilkc
CFLAGS = -Wall -O3

all:md6cilk

md6cilk:md6cilk.o md6_compress.o
	$(CILKC) $^ -o $@

md6cilk.o:md6.cilk
	$(CILKC) $(CFLAGS) -c $^ -o $@

md6_compress.o:md6_compress.c
	$(CC) $(CFLAGS) -c $^ -o $@

md6_mode.o:md6_mode.c
	$(CC) $(CFLAGS) -c $^ -o $@

clean:
	-rm -f *.o md6sum md6cilk
```