BUAP | Facultad de Ciencias de la Computación

BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

AGRUPACIÓN DE FRAGMENTOS DE SECUENCIAS DE ADN A PARTIR DE TÉCNICAS PARALELAS PARA TAREAS DE ENSAMBLAJE DE GENOMAS

TESIS

PRESENTADA PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

ROBERTO HERNÁNDEZ MUNIVE

ASESORES:

DR. IVAN OLMOS PINEDA

DR. JOSÉ ARTURO OLVERA LÓPEZ

PUEBLA, PUE., ABRIL 2016

DEDICATORIA

A la patria que me dio educación.

AGRADECIMIENTOS

A los asesores que me ayudaron en el desarrollo de este trabajo, a mi familia por su apoyo y a la Secretaría de Educación Pública por la beca que me ayudó a realizar este trabajo.

CONTENIDO

RESUMEN

El ensamble de cadenas de ADN se plantea como un problema de optimización combinatoria y se clasifica como NP-duro, en esta área las técnicas heurísticas se han implementado exitosamente, debido a que no son exhaustivas ni deterministas.

Algunas técnicas metaheurísticas que se han implementado para el ensamble de secuencias de ADN destacan por realizar un preprocesamiento, en el cual se identifican subgrupos de secuencias que tienen una alta probabilidad a ser alineadas entre sí. Este preprocesamiento se basa en el paradigma divide y vencerás, de tal forma que el costo computacional de encontrar las alineaciones de secuencias se reduzca sustancialmente.

En este sentido, en el presente trabajo de tesis se expone un trabajo en el que se utiliza un algoritmo de agrupación para pre-evaluar un conjunto de fragmentos de secuencias de ADN, las cuales a partir de un algoritmo de emparejamiento, determina subgrupos de fragmentos, los cuales tienen una alta probabilidad de ser alineados en una tarea de ensamble. Aunado a lo anterior, la tarea se lleva a cabo utilizando cómputo paralelo, de manera específica, utilizando tecnologías emergentes de paralelización como las tarjetas gráficas de propósito general, que hoy en día implementan un alto número de procesadores. Lo anterior consigue que la tarea de agrupación se lleve a cabo con menor tiempo de ejecución.

Como resultado final, en esta tesis se presenta un método que nos permite separar en grupos los fragmentos de secuencias de ADN, para ser procesados posteriormente por alguna técnica de ensamble de secuencias de ADN.

1. INTRODUCCIÓN

En 1665 Robert Hooke publicó uno de los principios fundamentales de la biología, donde describe que los organismos están formados por compartimentos individuales llamados “células” (esta fue la primera vez que se acuñó el término célula en un sentido biológico), actualmente se conoce que todos los seres vivos están formados por una o más unidades similares conocidas como células. En la naturaleza podemos encontrar una gran cantidad de células, cuya forma puede variar pero en el fondo comparten características en su ciclo de vida; nacer, comer, replicarse y morir.

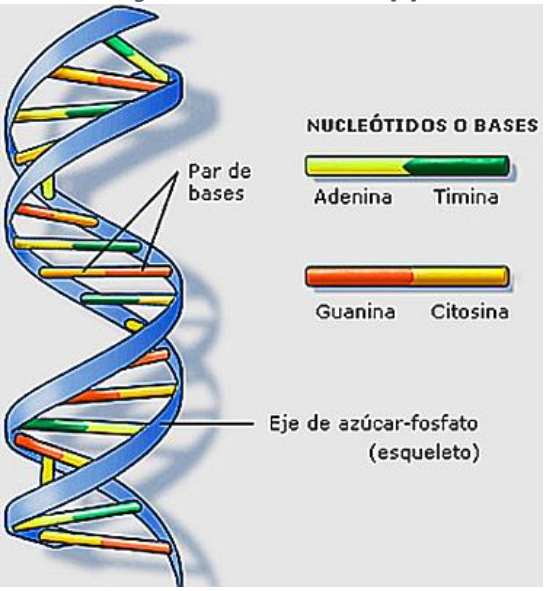
Las células carecen de un cerebro que les ayude a tomar decisiones, en su lugar cuentan con una compleja red de reacciones químicas que sintetizan o degradan materiales para enviar instrucciones desde comer hasta morir, el algoritmo que controla a las células todavía se encuentra más allá de nuestra comprensión.

Más tarde en 1839 los biólogos Schleiden y Schwann publicaron su obra donde hacen énfasis en la presencia de centros o núcleos en las células, descubrieron que diferentes organismos tienen diferente número de cromosomas, lo que sugirió que estos podrían ser los portadores de información específica de las especies. Por el año 1940 los biólogos comprendieron que los rasgos de una célula son inherentes a su información genética que a la vez recibió de su ascendencia, la información genética está organizada en los genes que se encuentran en los cromosomas.

El ADN fue descubierto en 1869 por Johann Friedrich cuando extrajo una sustancia de los núcleos de las células a la que nombró nucleína, a principios de 1900 era conocido que el ADN (nucleína) es una gran molécula consistente con cuatro tipos de bases: adenina (A), timina (T), guanina (G), citosina (C) . Aunque originalmente también se incluyó una quinta base llamada uracilo (U) cuya química es parecida a la Timina, pero fueron agrupados en dos clases conocidas como ácido desoxirribonucleico (ADN) que utiliza la timina y ácido ribonucleico (ARN) usa el uracilo [5].

El 1953 determinaron la estructura de doble hélice de la molécula de ADN, la cual define las características de los organismos y los hace únicos, en esta época ya se tenía conocimiento del puente de hidrógeno, una relación complementaria entre las bases; adenina-timina y guanina-citosina que se conoce como ley de Chargaff (solo es aplicable al ADN) y se define como la relación cuantitativa de los nucleótidos, la cantidad de adenina es igual a la de timina y la cantidad de guanina es igual a la de citosina, es decir: (A + G = C + T) como se muestra en la figura 1-1. Sin embargo existen diferencias en esta relación entre organismos eucariota con uno procariota.

Figura 1-1 Hélice de ADN [1].

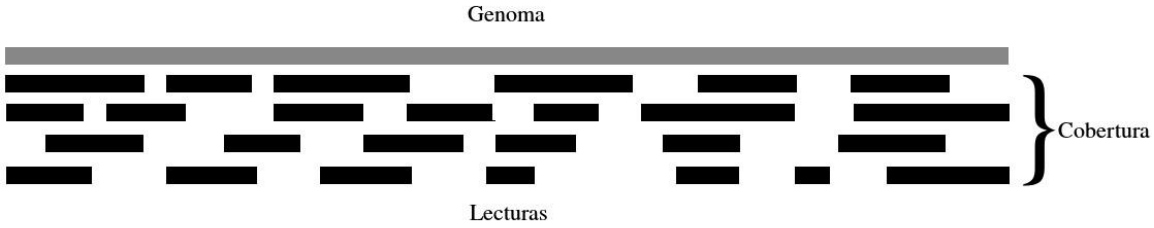


Gracias a la ley de Chargaff podemos expresar la secuencia de ADN completa con una hebra de la hélice y computacionalmente descrita por una cadena de caracteres formada por un alfabeto de cuatro letras {A, T, G, C} representando a los nucleótidos, de este modo un ejemplo de secuencia es ATTCCCCTAACCCTAACCCTA.

Podemos definir el problema de secuenciación de un genoma como la representación digital del ADN de un organismo. En este proceso se debe aislar el ADN y leerlo mediante un secuenciador para convertirlo en cadenas de texto para su procesamiento computacional. Actualmente los secuenciadores tienen la limitante de leer una ínfima cantidad de pares de bases (rango de comprendido en cientos de pares de bases), este proceso toma pequeños fragmentos (llamados lecturas) leídos de manera aleatoria que posteriormente serán utilizados en la tarea de ensamble, a este método se le conoce como secuenciación por perdigón ("*shotgun sequencing*") (ver figura 1-2).

El ensamble de secuencias se puede definir como el método para combinar los fragmentos de secuencias de ADN y esperar reconstruir el genoma original, esta tarea se realiza empalmando los fragmentos que tienen algún traslape para formar fragmentos más largos hasta formar el genoma original [5].

Figura 1-2 Shotgun sequencing.



Los métodos actuales de secuenciación, conocidos como NGS por sus siglas en inglés "Next Generation Sequencing", obtienen muchas lecturas en un tiempo menor que la tecnología anterior conocida como secuenciación de Sanger, al igual que el tiempo, también el costo de la secuenciación es menor. La longitud de los fragmentos leídos con un secuenciador NGS va desde los 35 a un poco más de 1000 pares de bases [4].

La secuenciación de escopeta (shotgun sequencing) es uno de los procesos a través de los cuales las secuencias son generadas, involucra un proceso físico aleatorio de lectura que devuelve pequeños fragmentos del genoma de interés, por la naturaleza del método es posible que ciertas partes de dicho genoma no sean leídas si no se realiza una cantidad mínima de lecturas.

Para evaluar la viabilidad teórica de la lectura de un conjunto de datos devueltos por el algoritmo de secuenciación de escopeta, Eric Lander y Mike Waterman desarrollaron un análisis estadístico basado en las estadísticas de Poisson y determinaron que si algunos eventos ocurren de manera uniforme al azar el número de eventos que ocurren dentro de un determinado intervalo de tiempo está representado por una distribución de poisson [6].

Dado una λ que representa el número de eventos que ocurrirán dentro de un intervalo dado de tiempo, la probabilidad de que exactamente *n* eventos ocurrirán dentro del mismo intervalo de tiempo está representada con la fórmula $f(n,\lambda) = \frac{\lambda^n e^{-\lambda}}{n!}$.

Dentro del contexto de secuenciación estamos interesados en la búsqueda de los intervalos que no contienen eventos, es decir *n=0*, debido a que estos representan brechas en el genoma. La cantidad de brechas que pueden esperarse en un número de segmentos de ADN contiguos puede obtenerse mediante los siguientes parámetros:

G- Longitud del genoma

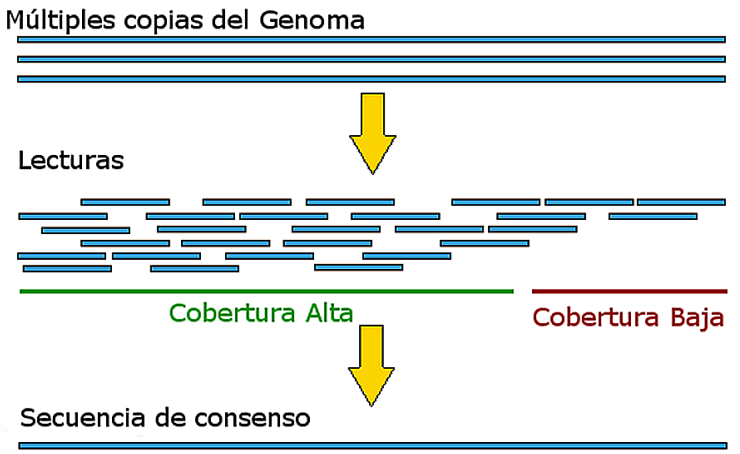
n - Número de secuencias leídas

L - Longitud de las secuencias

$C = nL/G$ - Redundancia en los datos

De esta forma podemos estimar la cantidad de lecturas que debemos realizar para obtener una buena cobertura del genoma.

Figura 1-3 Secuenciación de un genoma [2].



1.1 CONCEPTOS BÁSICOS

La reconstrucción de un genoma mediante los fragmentos de ADN puede definirse como encontrar una secuencia de longitud más corta *S*, dada una colección *F* de fragmentos de secuencias y un error $0 <= e <= 1$, tal que para todo fragmento *A* en *F*, existe una subcadena *B* de S tal que: $d(A, B) \leq e\ |A|$. Donde $d(A, B)$ se define como la distancia (Damerau-Levenshtein) de edición entre *A* y *B* que se refiere a la cantidad mínima de inserciones, eliminaciones, sustituciones y transpuestas para transformar la cadena *A* en la cadena *B*.

Este problema se puede modelar de tres formas; súper cadena común más corta (SCC conocido también como SCS "Shortest common Superstring"), reconstrucción y *multicontig* (múltiples regiones cubiertas contiguamente).

Definición 1: Dado un alfabeto *Σ* y un conjunto de k cadenas, $P = \{s_1, s_2, s_3, \ldots, s_k\}$ contenido en *Σ+*, una súper cadena de P es una cadena que contiene cada cadena en P como subcadena. Es decir que los elementos que pertenecen a las secuencias ensambladas cubran lo más posible a los elementos de la secuencia de referencia, las secuencias ensambladas deben ser subcadenas de la

secuencia original y la unión de las subcadenas contenga la mayor cantidad de elementos de la cadena original.

Definición 2: Para el conjunto $P, S * (P)$ denota la súper cadena más corta de *P*.

Sabemos que el problema de la súper cadena común más corta es un problema NP-Difícil, por lo tanto, el problema de ensamble de secuencias de ADN es NP-difícil. Este no es el único problema a resolver en la tarea de ensamble, aunado a este, se presentan errores de lectura en el secuenciador o un proceso biológico conocido como poliploidía y se define como el fenómeno en el cual se crean células con más de dos juegos de la misma o distintas especies, esto ocurre por ejemplo cuando un organismo hereda características del padre y de la madre, entonces habrá secuencias del genoma repetidas pero con pequeñas variaciones, estas premisas involucran un aumento en la complejidad del problema ya que puede haber variantes de una misma lectura, por ejemplo:

Error de lectura en la secuenciación

Lectura "correcta"	Lectura 1	Lectura 2
TGAT**T**GGGAGCATG	TGAT**A**GGGAGCATG	TGAT**T**GGGAGCATG

Diferencia de lectura causada por poliploidía, donde existen las dos cadenas de manera biológica.

Lectura 1	Lectura 2
TGAT**T**GGGAGCATG	TGAT**A**GGGAGCATG

Es importante notar que los errores de lectura son ocasionados por el secuenciador, mientras que las lecturas con un caso de poliploidía son lecturas correctas de diferentes cromosomas, en ambos casos debemos valorar que los patrones a encontrar no deben ser exactamente iguales.

Otro problema a considerar vuelve a tener relación con la tecnología de secuenciación, dado que estos entregan múltiples cadenas leídas de forma aleatoria, existirán fragmentos de cadenas repetidas, a estos se les conoce como repeticiones y deberán ser tratados para disminuir los huecos en el genoma, conocidos como 'gaps'. Se han desarrollado dos estrategias para reducir la probabilidad de que aparezcan fragmentos que sean repeticiones.

La primera estrategia consiste en tratar de obtener las lecturas con la mayor longitud que el secuenciador sea capaz de obtener, de este modo, cuanto mayor sea la longitud de las lecturas, menor será la posibilidad de que aparezca en otra posición dentro del genoma a reconstruir, esta opción es poco viable para los secuenciadores NGS actuales ya que las secuencias que producen son de longitud corta.

La segunda estrategia utiliza información de una lectura cercana, esto se hace obteniendo lecturas en pares, con una distancia k de caracteres, a estas lecturas se les llama lecturas por pares *mate pair*

reads. Con esta técnica reducimos la posibilidad de que estos pares aparezcan en una repetición a medida de que la distancia k entre los pares aumenta. El inconveniente de este método recae en que se necesita un método adicional para construir el camino entre los pares de secuencias, incluso este camino puede contener errores que pueden crecer en relación con la distancia entre el par de lecturas, además no siempre existe la posibilidad de realizar las lecturas de esta forma.

1.2 MOTIVACIÓN

Con base en las descripciones anteriores podemos apreciar que el problema de ensamblar secuencias de ADN presenta una complejidad exponencial. Para aproximar una solución, se ha recurrido al desarrollo de heurísticas que se han implementado exitosamente debido a que presentan las características de no ser exhaustivas ni deterministas, lo cual reduce la complejidad computacional empleada, sin embargo, debido a la cantidad elevada de combinaciones necesarias (tareas altamente repetitivas), por los algoritmos que se utilizan en este tipo de técnicas, el tiempo de cómputo suele ser muy elevado [8].

Dentro de las tecnologías comerciales más asequibles para desempeñar este tipo de tarea tenemos a las tarjetas gráficas de propósito general (GPGPU) las cuales explotan las características del cálculo acelerado ofreciendo un rendimiento sin precedentes mediante el traslado de las partes de la aplicación con mayor carga computacional a la GPU y dejando el resto del procesamiento a la CPU principal. Por otro lado tenemos que el uso de algoritmos de secuenciación, como el mencionado anteriormente, representan sus datos mediante una lista en donde sus elementos no guardan dependencias mutuas, lo cual, al momento de computarse se traduce a que estos pueden dividirse por bloques y trabajar con todos los datos de manera simultánea, de manera sucinta, podemos realizar una clasificación de manera paralela.

En este sentido, surge la necesidad de diseñar diferentes tipos de herramientas computacionales, como son las dedicadas a procesar fragmentos de secuencias de ADN para su ensamble. Dado que esta tarea es computacionalmente cara, se requiere explorar técnicas que permitan reducir los tiempos de respuesta para el ensamble de secuencias, por lo que se tienen que explorar técnicas heurísticas, así como paralelas, que permitan apoyar en la reducción de los tiempos de procesamiento [16].

En este trabajo de tesis se propone un método para agrupar las lecturas utilizando información de empalmes entre fragmentos para generar grupos con alta probabilidad de formar fragmentos de mayor longitud en la tarea de ensamble, de modo que se alimente a un ensamblador de secuencias con los grupos generados para reducir el espacio de búsqueda, se aprovecha la independencia de los datos para utilizar técnicas de procesamiento paralelo de instrucciones y simples y múltiples datos (SIMD) y se aterrizan en la arquitectura CUDA de tarjetas gráficas de propósito general.

1.3 OBJETIVO GENERAL

Diseñar una metodología para agrupar fragmentos de cadenas de ADN que tengan una alta probabilidad a ser alineadas, a través del procesamiento paralelo de los datos utilizando tecnologías como las tarjetas gráficas de propósito general (GPGPU), así como métricas de distancia que evalúen a los candidatos, con la finalidad de contribuir en la disminución de los tiempos de ejecución global.

1.4 OBJETIVOS ESPECÍFICOS

1. Comparar las técnicas de agrupamiento para cadenas de caracteres, identificando los algoritmos más representativos para la tarea.
2. Seleccionar las métricas de distancia que permitan determinar si dos cadenas de caracteres tienen probabilidad de alinearse.
3. Diseñar un algoritmo de clusterización paralelo basado en las métricas de distancia diseñadas, utilizando la tecnología de paralelización basada en GPU's.
4. Codificar el algoritmo de clusterización seleccionado, así como la métrica seleccionada.
5. Evaluación de resultados obtenidos.

1.5 PROPUESTA DE SOLUCIÓN

En este trabajo de tesis se enfoca en realizar un preprocesamiento de las lecturas del secuenciador para alimentar un ensamblador de secuencias de ADN. La solución propuesta implementa un algoritmo de búsqueda inexacta de patrones que agrupa los fragmentos de secuencia de ADN que tengan cierta cantidad de caracteres empalmados, en la fase final se crean grupos con alta probabilidad de ensamblarse y crear una secuencia más larga conocida como *contig*.

Dentro de este proceso se utilizan algoritmos para la búsqueda inexacta de patrones, al realizar estas búsquedas se comparan los *N* fragmentos contra los restantes *N-1* fragmentos, como estos datos son independientes entre ellos, se puede paralelizar la tarea y de esta forma realizamos *K* búsquedas simultáneas para disminuir el espacio de búsqueda en el que tendrá que trabajar el ensamblador de secuencias de ADN. El ensamblador implementado para empalmar los grupos es de tipo ensamble mapeado.

Como resultados finales se compara la salida del genoma ensamblado a partir de las lecturas agrupadas por el algoritmo propuesto, se verifica la calidad del ensamble y el tiempo que le toma ensamblar la secuencia completa**.**

1.6 ORGANIZACIÓN DE LA TESIS

En esta sección se presenta la organización de este documento, dividido en siete capítulos que se organizan de la siguiente forma.

1. Introducción: En este capítulo se describe el problema, los objetivos, la motivación y la solución propuesta al problema**.**
2. Marco teórico: Se plantean los conceptos base que se usan en este trabajo. Se hace una introducción a la teoría de alineación de cadenas y se describen las técnicas de paralelismo empleadas.
3. Trabajos previos: Descripción de los algoritmos utilizados y las técnicas existentes que fueron adaptadas para resolver el problema.
4. Algoritmo propuesto: Análisis del algoritmo y su funcionamiento.
5. Experimentación: Experimentos propuestos.
6. Resultados obtenidos: Análisis de los resultados y descripción de los datos.
7. Conclusiones y trabajo futuro

2. MARCO TEÓRICO

Dentro de este capítulo se exponen los conceptos fundamentales utilizados en los ensambles de ADN.

En primera instancia debemos definir la búsqueda de subcadenas debido a que es un concepto fundamental para el desarrollo de este trabajo así como los algoritmos más representativos de esta área. En segundo término se presentará de manera somera los algoritmos de ensamble de secuencias de ADN.

2.1 CONCEPTOS BÁSICOS DE BÚSQUEDA DE SUBCADENAS

Este tipo de algoritmos es conocido como algoritmos de patrones de un texto, algoritmos de emparejamiento de secuencias, algoritmos de casamiento de secuencias o por su nombre en inglés "string matching". El objetivo de estos algoritmos busca una subcadena llamada patrón, en otras palabras se busca todas las ocurrencias de un patrón *p* de longitud *m* dentro de un texto *t* de longitud *n*, donde se debe cumplir la condición n ≥ m. Existen cinco clasificaciones para agrupar los algoritmos.

2.1.1 Clasificación de algoritmos de emparejamiento

1. **Fuerza bruta:** Este algoritmo es el más simple y el más complejo computacionalmente, lo que hace es recorrer el patrón de izquierda a derecha comparando con las subcadenas de la misma longitud que inician en cada carácter del texto.

2. **Búsqueda con autómata determinista:** Leen todos los caracteres del texto de uno en uno y modifican en cada paso algunas variables que permiten identificar las posibles ocurrencias. A este tipo pertenecen los algoritmos de Knuth-Morris-Pratt, Shift-Or, Shift-And, Myers.

3. **Búsqueda en una ventana a lo largo del texto:** Buscar el patrón dentro de una ventana que se desliza por el texto. Para cada posición de la ventana se busca de derecha a izquierda un sufijo de la ventana que corresponda a un sufijo del patrón. Algunos algoritmos de este tipo son; Boyer-Moore, Boyer-Moore-Haspool y Sunday Quick Search. Este tipo de algoritmos no trabajan bien cuando la longitud del patrón es corta y existe alta posibilidad de encontrarlo dentro del texto.

4. **Búsqueda de derecha a izquierda en una ventana:** La búsqueda se realiza de derecha a izquierda dentro de la ventana, pero en este esquema se busca el sufijo más largo en la ventana que sea subcadena del patrón. Ejemplos de este tipo de algoritmos son BDM, BNDM y BOM. Al igual que el anterior, este tipo de algoritmos para patrones de longitud corta no funcionan bien.

5. **Esquemas basados en funciones de dispersión:** Utilizan una función de dispersión sobre un patrón de longitud n y van recorriendo el texto de n en n elementos y calculado su valor de dispersión para compararlo con el del patrón, algoritmos como *Rabin-Karp* utilizan este método.

Existen adaptaciones de estos algoritmos para extender la capacidad de búsqueda al utilizar un conjunto de patrones $P = \{p_1, p_2, p_3, \dots\ p_n\}$, a este tipo de búsquedas se les conoce como búsqueda múltiple de subcadenas (*multiple string matching*) [12] [13].

2.1.2 Métricas de distancia de edición

Una operación de edición es una operación en cadenas, se trata de una relación binaria simétrica en el conjunto de todas las cadenas consideradas. Dado un conjunto de operaciones de edición $O = \{O_1, O_2, O_3, \dots, O_m\}$, la distancia de edición entre una cadena *X* y *Y* es el número mínimo de operaciones *O* necesarias para transformar *Y* en *X* [15].

Las principales operaciones de edición son:

1. **Edición de carácter:** Inserción o eliminación de un carácter.
2. **Reemplazar un carácter:** Cambiar un carácter por otro en la misma posición.
3. **Intercambio:** Intercambiar caracteres adyacentes.
4. **Mover una subcadena:** Transformar $X = x_1 \dots\ x_n$ en la cadena $x_1 \dots x_{i-1} x_j \dots x_{k-1} x_i \dots x_{j-1}\ x_k \dots x_n$.
5. **Copia una subcadena:** Transformar $X = x_1 \dots\ x_n$ en $x_1 \dots x_{i-1} x_j \dots x_{k-1} x_i \dots x_n$.
6. **Eliminación de una subcadena:** Eliminar una subcadena mientras una copia permanezca en la cadena.

Las operaciones listadas se utilizan en diversas métricas que consideran distintas operaciones, algunas de las más conocidas son:

Levenshtein: Es una métrica obtenida de *O* que considera solo reemplazos de carácter e intercambios, es decir los puntos *1* y *2* listados. Dadas dos cadenas *X*, *Y* de longitud *m* y *n* la distancia se calcula como $\min\{d_H(X^*, Y^*)\}$, donde X^* y Y^*son cadenas de longitud *k*, $k \geq \max\{m, n\}$ sobre un alfabeto $A^* = A\ \cup \{*\}$, la distancia es el número mínimo de operaciones 1 o 2 para transformar *Y* en *X.* La similitud de Levenshtein se calcula por $1 - \frac{d_L(X,Y)}{\max(m,n)}$.

Damerau-Levenshtein: Es la métrica de edición en *W(A)* obtenido de *O* consistente solo de reemplazo de caracteres, inserciones, eliminaciones y trasposiciones. A diferencia de la métrica de Levenshtein donde la trasposición ocupa dos operaciones (una inserción y una eliminación), esta utiliza solo una.

2.2 ENSAMBLAJE DE SECUENCIAS DE ADN

El problema de ensamblar un genoma es análogo a tomar varias copias de un libro, pasarlas por un triturador, tomar los fragmentos resultantes y tratar de reconstruir una sola copia del libro.

La tarea de ensamble conlleva el proceso:

Secuenciar el ADN: En este paso se realiza una serie de lecturas por las máquinas llamadas secuenciadores, estas lecturas son pequeños fragmentos del ADN.

Encontrar solapes: Para ensamblar la secuencia, el primer paso es encontrar los empalmes entre los fragmentos de ADN obtenidos. Con esta información se puede crear un mapa para ensamblar secuencias más largas.

Formar *contigs:* Un *contig* es una secuencia continua de ADN que ha sido ensamblada mediante los fragmentos de ADN empalmado, de este modo se van creando fragmentos más grandes del ADN.

Ensamblar *scaffolds*: Un *scaffold* es una secuencia no redundante formada por la unión de uno o más *contigs*, a diferencia de estos el *scaffold* no requiere que exista un solape entre los *contigs* para ensamblar secuencias más largas. Este paso es opcional en el ensamble y se puede considerar un postprocesamiento que mejora la calidad del ensamble. Las secuencias formadas con *scaffolds* pueden contener gaps entre ellas, no así en los *contigs* [1].

2.3 CÓMPUTO PARALELO

La eficiencia de una computadora para resolver un problema depende del algoritmo utilizado, en teoría las máquinas paralelas ofrecen un poder de cómputo superior pero adaptar la solución a este paradigma para explotar todas sus capacidades no siempre es fácil. El mejor algoritmo secuencial para resolver un problema dado no conlleva al mejor algoritmo paralelo, para llegar a su homónimo se deben explotar las fuentes del paralelismo del problema.

La concurrencia busca ejecutar simultáneamente todas las tareas que se puedan mientras no exista dependencia de datos entre ellas, se basa en una red de intercambio de información que es asimétrica con intercambios asíncronos. En este paradigma los problemas de asignación de tareas y la administración de la interconexión son los problemas más destacados. Las tareas que se deben paralelizar deben ser definidas de forma explícita.

En el paralelismo las tareas se organizan de forma regular como una tabla, deben ser lo más parecido posible entre ellas, la red de interconexión es síncrona y simétrica. Con este precedente

encontramos que muchas veces se debe realizar un tratamiento previo para espaciar y transformar las tareas de forma simétrica, este es el mayor reto a resolver en este paradigma.

La granularidad de paralelismo es definido como el tamaño promedio de las tareas elementales, se define por su número de instrucciones, palabras en memoria utilizadas y su tiempo de ejecución. Se puede definir como la cantidad de cómputo que una tarea realiza antes de que necesite comunicarse con otra tarea, los granos pueden crecer agrupando tareas o decrecer si se desagrupan, se establece una relación entre el tiempo de cómputo con respecto al número de eventos de comunicación. Granularidad grande significa que se tiene menos comunicación, pero que potenciales tareas concurrentes sean agrupadas y ejecutadas secuencialmente. La determinación del grano ideal para una aplicación dada es importante para definir la mejor relación entre paralelismo y comunicación. Un paralelismo de grano fuerte se utiliza en arquitecturas que ejecutan múltiples instrucciones en múltiples datos MIMD (*Multiple Instruction, Multiple Data*), mientras que el grano fino es utilizado en arquitecturas de una instrucción y múltiples datos SIMD (*Single Instruction, Multiple Data*).

El grado del paralelismo se define como el número de tareas que se pueden ejecutar de manera paralela durante la ejecución de una aplicación, corresponde a una medida del número de operaciones que se pueden ejecutar simultáneamente y se refleja en el número de procesadores a utilizar. El grado puede variar a través de las distintas partes de un programa, este tiene un máximo, mínimo y promedio, la cota superior es el número máximo de procesadores que podrían utilizarse sin afectar la eficacia que podría ser mediocre si muchos procesadores pasan inactivo mucho tiempo en la ejecución.

Los programas paralelos son más difíciles de codificar que los secuenciales debido a que la concurrencia presenta nuevos retos como las condiciones de carrera, la sincronización y comunicación de las tareas. La máxima aceleración posible de un algoritmo paralelo obedece lo que se conoce como la ley de Amdahl, llamada así por Gene Amdahl, la cual nos dice que "*la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente*". Formalmente se define por $T_m = T_a\,((1 - F_m) + \frac{F_m}{A_m})$, donde F_m es la fracción de tiempo que el sistema utiliza el subsistema mejorado, A_m es el factor de mejora que se ha introducido en el subsistema mejorado, T_a es el tiempo de ejecución anterior y T_m es el nuevo tiempo de ejecución. [17]

2.4 TARJETAS GRÁFICAS

Este tipo de dispositivos respectan a un coprocesador dedicado al procesamiento de gráficos y operaciones de coma flotante, surge con el objetivo de liberar carga de trabajo al procesador central en aplicaciones con una elevada demanda de gráficos, tales como juegos y procesamiento de

imágenes o vídeo. De esta forma, mientras una gran parte de lo relacionado a gráficos se procesa en la unidad procesadora de gráficos (GPU), el procesador central (CPU) se dedica a otro tipo de tareas o cálculos.

En estas tarjetas se implementan operaciones que han sido optimizadas para el procesamiento gráfico, a las cuales se les conoce como primitivas, por ejemplificar una de estas se puede mencionar el suavizado de bordes. Actualmente se dispone de gran cantidad de primitivas no solo para el procesamiento de gráficos, también para el cómputo de propósito general.

En la actualidad se puede utilizar una GPU para realizar cómputo de aplicaciones de diverso tipo como: ingeniería, ciencia, medios digitales entre otros. Existen dos grandes fabricantes de estos dispositivos: AMD y nVidia.

nVidia desarrolló tecnología para procesar los datos gráficos de forma paralela, posteriormente se implementa para procesar cómputo general, se crea un conjunto de herramientas llamada CUDA que significa *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo), se le conoce como núcleo CUDA a los procesadores de esta plataforma. En la actualidad las tarjetas comerciales de bajo costo implementan cerca de 1600 núcleos CUDA [3].

AMD aprovecha el poder de cómputo de la GPU mediante su tecnología llamada *Stream processing*, que es como denomina a los procesadores, la plataforma de desarrollo se conoce como OpenCL (estándar de programación paralela de sistemas heterogéneos), la diferencia con CUDA es que los núcleos CUDA consisten de un procesador capaz de realizar solo una operación a la vez, los núcleos Stream tiene cinco procesadores y pueden desempeñar simultáneamente hasta 4 operaciones simples (como suma/multiplicación), pero solo para vectores. En operaciones con escalares, solo uno de los cinco núcleos hace el trabajo. La diferencia en rendimiento no está dada por el número de núcleos, los núcleos CUDA son mejores para operaciones de punto flotante. Las GPU de AMD son utilizadas en operaciones con enteros y son implementadas en tareas como criptografía.

Se debe mencionar que la plataforma CUDA cuenta con más tiempo de desarrollo, lo que ha permitido que se difunda y se mejore, cuenta con más soporte debido a que existe una comunidad de desarrolladores más grande y con el paso del tiempo se han ido abriendo algunos paquetes como software libre este aún no es completamente libre.

3. TRABAJO PREVIO

3.1 ALGORTIMO DE GENE MYERS

La alineación aproximada de cadenas es la técnica de encontrar todas las ubicaciones en las que una consulta de tamaño *m* empalma en una subcadena de longitud *n* con *k* o menos diferencias. Estos algoritmos computan una representación de bits de un conjunto de estado actual para un autómata con una diferencia *k* para la consulta dada. Para determinar el valor de *k* se utiliza una métrica de edición, en este algoritmo se utiliza la distancia de Damerau-Levenshtein ya que es una variante del algoritmo original para tomar trasposiciones como una sola operación [7] [9] [11].

Se asume que la consulta es $P = p_1 p_2 \dots p_m$ y el texto donde se busca es $T = t_1 t_2 \dots t_n$, dado valor de umbral positivo $k \geq$. Las operaciones que se realizan son binarias y se representan por los símbolos de la tabla 3-1.

Tabla 3-1 Operaciones utilizadas por el algoritmo de Myers

Símbolo	**Operación**
|	Disyunción
&	Conjunción
~	Complemento
^	Disyunción exclusiva
+	Suma
≪	Corrimiento a la izquierda

Algoritmo 3-1 Algoritmo de Myers.

Inicialización:

$TC = 0^m;\ VP = 1^m;\ VN = 0^m;\ Distancia = m$

Actualización:

$Para\ j = 1\ hasta\ n$

$TC = PM_{T_j} \mid (\left(\left((\sim \mathrm{TC}) \& PM_{T_j}\right) \ll 1\right) \ \&\ PM_{T_{j-1}})$

D0 = (((TC & VP) + VP) ^ VP) | TC |VN

$\mathrm{HP} = \mathrm{VN} \mid \sim (\mathrm{D0} \mid \mathrm{VP})$

$HN = D0 \ \& \ VP$

$Si\ ((HP \ \& \ 10^{m-1}) \neq 0^m)$

$\mathrm{Distancia} = \mathrm{Distancia} + 1$

$Si\ ((HN \ \& \ 10^{m-1}) \neq 0^m)$

$\mathrm{Distancia} = \mathrm{Distancia} - 1$

$Si\ Distancia \leq k$

$Retorna\ coincidencia\ en\ el\ índice\ j$

$VP = (HN \ll 1) \mid \sim (D0 \mid (HP \ll 1))$

$VN = D0 \ \& \ (HP \ll 1)$

$fin\ para$

Ejemplo:

Sea el texto $T = G\boldsymbol{ACTTAA}CGT$ y el patrón $P = \boldsymbol{ACTTA}$, tenemos los valores $m = 5, n = 9, k = 1$, como se realizan operaciones con vectores de bits se debe codificar el alfabeto en binario utilizando m bits, por lo tanto para el ejemplo tenemos $A = 10001,\ G = 00000,\ C = 01000,\ T = 00110$ respecto al patrón ***ACTTA***, es decir, se crea un vector de bits de la longitud del patrón para cada símbolo del alfabeto, se introduce *1* si posición *j* aparece el símbolo y *0* en caso contrario.

El primer paso es inicializar las variables *TC, VP, VN y distancia*. En seguida se itera recorriendo los caracteres del texto reemplazando su valor con la codificación indicada para ejecutar las operaciones mencionadas.

Tabla 3-2 Traza del algoritmo de Myers

j	PM[T[j]]	PM[T[j-1]]	TC	VP	VN	D0	HP	HN	distancia
-	-	-	00000	11111	00000	-	-	-	5
1	10001	00000	10001	11110	00000	11111	00000	11111	4
2	00010	10001	00010	11101	00010	11110	00001	11110	3
3	01100	00010	01100	11001	00100	11110	00010	11100	2
4	10001	01100	10001	00010	00000	10111	00100	10001	1

Se inician las iteraciones en el segundo elemento del texto y se consulta el elemento anterior, en caso de que el valor de HN sea distinto de cero se decrementa el valor de la distancias porque se ha conseguido un empalme, para este ejemplo no es necesario recorrer todos los valores del texto pues se descubre un empalme de $m - k$ símbolos en la posición 4, es decir, en la cuarta iteración como se muestra en la tabla 3-2.

La ventaja de estos algoritmos está basada en el hecho de computar una instrucción con w bits (normalmente 32 o 64 en las computadoras actuales), si $m \leq w$ el costo computacional es $O(n)$ ya que existe un número constante de operaciones por carácter, el costo general es $O([m/w]/n)$.

3.2 PARALELISMO EN CUDA

En comparación con el procesamiento tradicional con CPU el cómputo en GPU's es un concepto nuevo, de hecho los GPU's son nuevos en comparación con los procesadores tradicionales. CUDA son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo), hace referencia a un conjunto de herramientas de desarrollo creadas por nVidia que permite la codificación de algoritmos en las tarjetas gráficas de mismo fabricante.

La arquitectura CUDA se encuentra organizada en una serie de multiprocesadores o *streaming multiprocessors* (SM) agrupados en bloques, donde cada bloque puede contener una gran cantidad de hilos de ejecución (hasta 1024 hilos de ejecución por bloque). Al conjunto de bloques se les denomina *grid*; la capacidad de este varía según las especificaciones de las tarjetas. Las GPU cuentan con una memoria dinámica de acceso aleatorio *Dynamic Random Access Memory* (DRAM) de doble tasa de transferencia de datos gráficos *Graphics Double Data Rate* (GDDR) que funciona como una memoria global accesible de manera única por sus propios procesadores.

Este modelo fue diseñado para crear aplicaciones que escalen su paralelismo de forma transparente para incrementar el número de núcleos computacionales, en este diseño destacan tres puntos, que además son clave para la arquitectura; jerarquía de grupos de hilos, memoria compartida y la barrera de sincronización. Los hilos pueden acceder a distintas jerarquías de memoria, algunas de las cuales son compartidas.

Figura 3-1 Memoria privada para cada hilo [3].

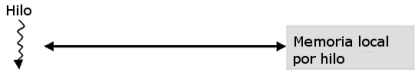


Figura 3-2 Memoria compartida por bloque [3].

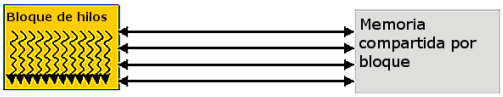
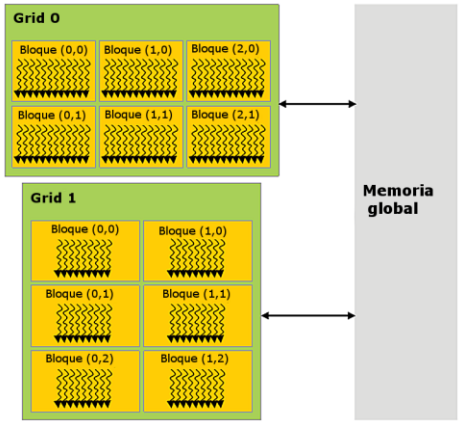
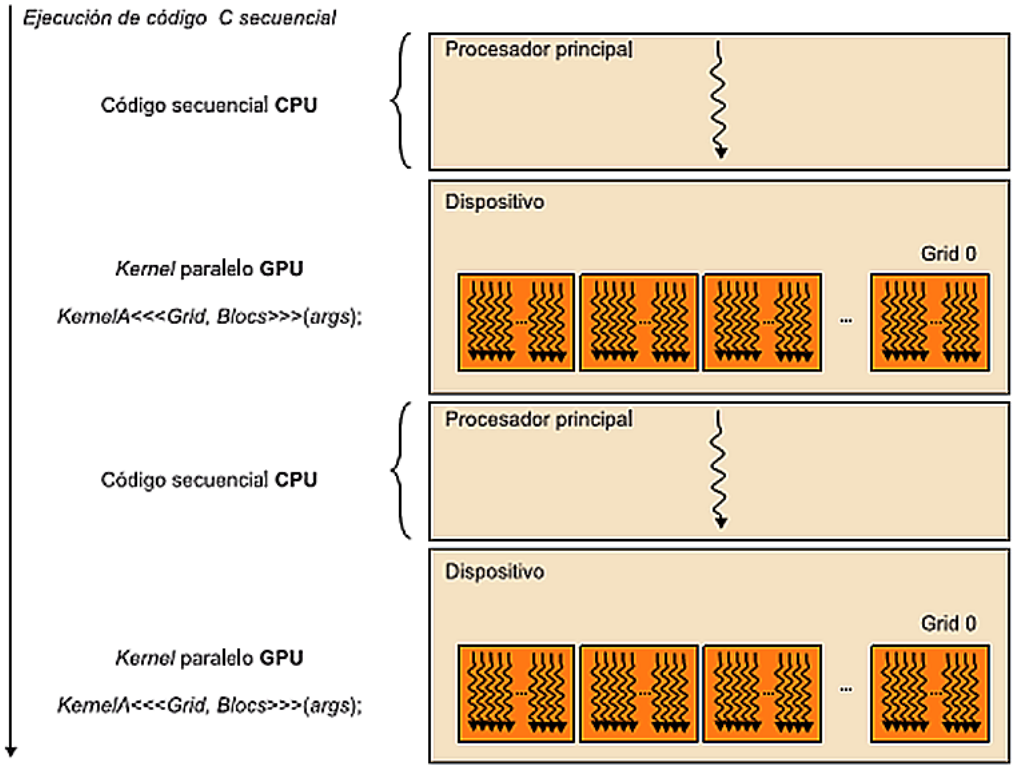


Figura 3-3 Memoria Global [3].



El modelo de programación es distinto al paradigma tradicional, la tarjeta gráfica se ve cómo un dispositivo externo de procesamiento comúnmente conocido como *device*, este opera como una especie de coprocesador de procesador principal, al cual se le conoce como *host*. Desde el punto de vista del programador, el sistema está compuesto por un procesador principal (*host*) y por uno o más dispositivos (*devices*) que pertenecen al modelo SIMD que está pensado para explotar el paralelismo, para esto se crea a una función conocida como *kernel* donde se describen las instrucciones que se ejecutarán en el *device*.

Figura 3-4 Flujo de ejecución de un programa escrito en lenguaje C [3].



La ejecución de un programa inicia en el *host*, cuando se llama la función *kernel* la ejecución se traslada al *device* donde se genera un elevado número de hilos, el conjunto de todos estos hilos que se generan se denominan *grid*. Se pueden llamar tantas funciones *kernel* como sean necesarias, al momento de ejecutar aplicaciones que demanden mucho tiempo de cómputo es recomendable desactivar la interfaz gráfica, ya que el controlador del *device* puede terminar las aplicaciones que afecten la tasa de refresco de la pantalla.

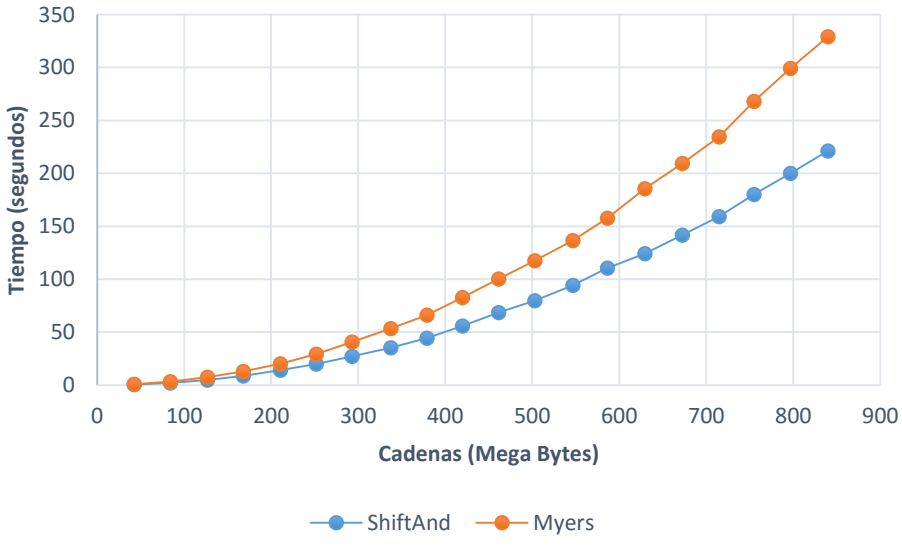
4. ALGORITMO PROPUESTO

En este capítulo se describe el algoritmo propuesto para agrupar las secuencias de ADN. A partir de un fichero de entrada conteniendo los fragmentos de genoma secuenciados, el algoritmo se encarga de calcular el espacio requerido por la tarjeta gráfica al igual que la cantidad de hilos de ejecución que se necesitan para procesar determinado número de fragmentos. Una vez que los datos han sido cargados a la tarjeta gráfica, se buscan los empalmes y se forman los grupos creando un grafo con los empalmes que mantiene cada fragmento, se retorna una matriz que representa el grafo al procesador para que forme los grupos en un archivo de salida.

4.1 ALINEACIÓN DE CADENAS

El principal recurso de este trabajo es el algoritmo de Gene Myers, esto se debe a dos características muy importantes; la complejidad algorítmica y la búsqueda aproximada de patrones. En comparación con otras técnicas de búsqueda de patrones exactos la complejidad del algoritmo es mayor, sin embargo es necesario considerar una búsqueda de aproximada por dos razones; los errores de secuenciación son por naturaleza estocásticos y pueden cambiar una o más bases dentro de la lectura, de manera similar existe la poliploidía que es un fenómeno natural dentro de los genomas. En una inspección visual de las lecturas no es posible distinguir estos errores pero nos afectarán de igual manera, como existen pequeñas variaciones en las lecturas se toma en cuenta que dentro de los empalmes a encontrar no pueden ser completamente idénticos sino que se debe manejar un pequeño error, es por eso que en este trabajo se hace uso del citado algoritmo.

Figura 4-1 Comparación entre los algoritmos de emparejamiento Shift-AND y Myers.



En la figura 4-1 se muestra la diferencia de comportamiento temporal que tiene el algoritmo *Myers* con *Shift-AND* al buscar empalmes en un conjunto de fragmentos de ADN. El algoritmo *Shift-AND* no es capaz de encontrar patrones aproximados a las cadenas, por este motivo se optó por utilizar el algoritmo de *Myers*. Este recurso se utiliza para encontrar empalmes en el prefijo o sufijo de un fragmento, de este modo se comparan los $n-1$ fragmentos de forma rápida y precisa. El tamaño de los empalmes se determina como argumento del programa, esto tiene relación directa con el tamaño de la lectura.

4.2 GENERACIÓN DE GRUPOS

Para generar los grupos se sigue una técnica simple pero efectiva, se genera una matriz de adyacencia, tan pronto como sea encontrado un empalme con otro fragmento se agrega su posición en un vector de adyacencias, el tamaño de este vector se encuentra restringido debido a que no es viable formar una matriz cuadrada, el costo de esta es exponencial, por citar un ejemplo: el tamaño que utiliza un dato del tipo *int* en C es de 4 bytes, suponiendo que se requiere agrupar lecturas de un organismo pequeño cuyo número de fragmentos son 13,000, tenemos que crear una matriz de dimensiones 13000 x 13000. El costo espacial de esta matriz es de 6, 760, 000, 000 bytes, un número que supera por mucho el límite de las tarjetas actuales.

Sabemos de antemano que no es factible encontrar lecturas donde todos y cada uno de sus elementos se encuentren relacionados entre ellos, podemos iniciar por el hecho de que no se necesita comparar un fragmento consigo mismo. En cambio se propone utilizar un límite que nos permita utilizar el espacio de una manera más eficiente, durante la secuenciación de un genoma se utiliza la ecuación de Lander – Waterman para determinar la cantidad de lecturas que se deben realizar de tal modo que se obtenga una un aproximado al genoma completo, por lo regular se utiliza una redundancia que implica leer 10 veces la longitud del genoma. Con estos datos se puede estimar que el genoma tendrá una cobertura de 10 en un caso ideal, los empalmes mínimos en este caso serían 40 si las lecturas tuvieran la misma longitud y los mismos empalmes.

Para computar el resultado final se traslada la matriz resultante a la memoria del *host*, el resultado obtenido hasta este punto es una matriz de adyacencia, para encontrar las conexiones entre estos puntos se realiza una búsqueda en profundidad, la cual formará los fragmentos por grupos y no repetirá ningún elemento, aunque si hay dos lecturas idénticas no es posible discernir diferencia alguna entre estas y se tratan como datos distintos.

Como resultado final se obtiene un determinado número de grupos con fragmentos que comparten n símbolos en el prefijo o sufijo, lo que los hace candidatos a formar un *contig* en la tarea de ensamble.

4.3 ALGORITMO PROPUESTO

El algoritmo se compone de dos funciones principales, la primera encuentra los empalmes entre los fragmentos mediante el algoritmo de *Myers* y los enlaza en un grafo que es retornado al procesador, el cual realiza la segunda función, una búsqueda en profundidad en el grafo para formar los grupos y enviarlos a un fichero de salida. A continuación se describe el primer algoritmo.

Algoritmo 4-1 Algoritmo propuesto.

```
Agrupa (Fragmentos, máximo, nodos, identificador, k, pb) {

i = 0;
pref = prefijo (Fragmentos [identificador], pb);
suf = sufijo (Fragmentos [identificador], pb);

        MIENTRAS (i < máximo) {
                SI (Myers (pref, Fragmentos [i], k) < Fragmentos
        [i].longitud() AND i != identificador )
                nodos[identificador].agrega(i);
            SINO SI (Myers (suf, Fragmentos [i], k) < Fragmentos [i].longitud()
        AND i !=  identificador)
                nodos[identificador].agrega(i);
            i++;
        }
```

Este algoritmo tiene como entradas:

1. **Fragmentos:** Un arreglo de cadenas que contiene los fragmentos del genoma.
2. **Máximo:** Es el número de cadenas contenidas en la variable *Fragmentos*.
3. **Nodos:** Es un arreglo de listas que contiene los elementos que se empalman con las cadenas.
4. **Identificador:** Es el identificador del proceso que busca los empalmes de un fragmento.
5. **K:** Es la diferencia en distancia de edición permitida entre el empalme buscado.
6. **PB:** Es la longitud del empalme buscado.

Las funciones *prefijo* y *sufijo* retornan una cadena con el prefijo o sufijo de longitud *pb*, estos elementos son los patrones que se buscarán en la función *Myers*, la cual recibe el patrón buscado (prefijo o sufijo según sea el caso), la cadena donde se buscará el patrón y la distancia de edición permitida, retorna un valor entero menor a la longitud de la cadena en donde se busca en caso de que el patrón sea encontrado y un número mayor en caso contrario. La función busca empalmes con todos los demás elementos, agregándolos a la lista correspondiente si hay una coincidencia.

Para recorrer todos los fragmentos debe haber un identificador de hilo de ejecución para cada uno, pudiendo ser el mismo hilo con dos identificadores distintos en el caso de que el número de fragmentos sea mayor al número de hilos soportados por la tarjeta gráfica, los identificadores de los hilos son asignados por ésta aunque pueden ser re-ordenados para calcular empalmes de otro fragmento.

Algoritmo 4-2 Búsqueda en profundidad.

```
BúsquedaEnProfundidad (Nodos, Fragmentos) {
        visitados[] = Máximo*[-1];
        PARA CADA nodo EN Nodos{
                SI (visitados[nodo] == -1){
                        visitados[nodo] = nodo;
                        siguientes.push(nodo);
                        MIENTRAS (siguientes.longitud() > 0 ){
                                PARA CADA vértice EN nodo.vértices()  {
                                        SI (visitados[vértice] == -1){
                                                visitados[vértice] = vértice;
                                                siguientes.push(nodo);
                                         }
                                }
                                Imprime (Fragmento[nodo]);
                                siguientes.pop();
                        }

                }
        }

}
```

En la función *BúsquedaEnProfundidad* realiza una impresión de los elementos del grafo que se encuentran unidos, es decir que tenemos elementos de forman grafos que no tienen vértices en común (grafos disjuntos o conjuntos disjuntos) y se utiliza la búsqueda en profundidad para ordenar los fragmentos en grupos que son escritos en un fichero como una lista de cadena de caracteres. La entrada de la función es la lista de *Nodos* y la lista de *Fragmentos* a la que apuntan los nodos, la salida es un archivo de texto con la lista de fragmentos separados en grupos.

4.4 EJEMPLO DEL ALGORITMO PROPUESTO

Dado un genoma del cual supondremos ha sido secuenciado cubriendo la mayor parte de estructura, pero que ha faltado un pequeño fragmento:

Tabla 4-1 Genoma secuenciado.

GTAGTCATGTTGAAAAACTTACGAGTAAATTACGTTGTCGAGGGCGTGCAAGTAGCGCAACCCGTGACA
AGCGCAAATTCGGAAGTAT**ACGCCAA**TCTACCGCCTCCCGTACCCGCGGAGACGTATCAAACCGACGAA
GATTACGAGGAAGATGACGGAGGGTGGGC

Nota: En este genoma se supone no se obtuvo la lectura de la parte marcada en azul.

Como resultado esperado de ensamble de las lecturas de este genoma se debe obtener dos *contigs* ya que no se ha logrado leer la secuencia completa.

Tabla 4-2 Contigs leídos.

Contig 1	**Contig 2**
GTAGTCATGTTGAAAAACTTACGAGTAAATTACG TTGTCGAGGGCGTGCAAGTAGCGCAACCCGTGA CAAGCGCAAATTCGGAAGTAT	TCTACCGCCTCCCGTACCCGCGGAGACGTATCAA ACCGACGAAGATTACGAGGAAGATGACGGAGG GTGGGC

Como resultado de las lecturas se obtiene la siguiente lista de fragmentos donde se resaltan los empalmes, los prefijos marcados en color verde, los sufijos en color rojo y en negrita los empalmes que se encuentran por traslapados por un sufijo y prefijo de otras cadenas, las partes sin marcar no tienen ningún traslape, lo que nos indica que esas secciones fueron leídas únicamente una vez. Se selecciona el tamaño del empalme deseado, para este ejemplo se utiliza un empalme de longitud 6, se debe comparar el prefijo y sufijo de cada cadena con el resto de las cadenas completas.

Tabla 4-3 Fragmentos leídos.

1	GTAGTCATGTTGAAAAACTT**ACGAGT**
2	**ACGAGT**AAATTACGTTGTCG**AGGGCG**
3	**AGGGCG**TGCAAGTAGCGCAA**CCCGTG**
4	**CCCGTGACAAGCGCAAATTCGGAAG**
5	**CGTGACAAGCGCAAATTCGGAAG**TAT
6	TCTA**CCGCCTCCCGTACCCGCGGAGAC**
7	**CCGCCTCCCGTACCCGCGGAGACGTA**
8	**GACGTA**TCAAACCGACGAAGA**TTACGA**
9	**TTACGA**GGAAGATGACGGAGGGTGGGC

La tabla 4-3 muestra la cantidad de hilos de ejecución necesarios para poder procesar los fragmentos de manera paralela. Cada uno de los nueve hilos se mapea para trabajar con una cadena y buscar los traslapes en los fragmentos restantes. Se toman dos subcadenas (prefijo y sufijo) de cada fragmento para compararlos con el resto de fragmentos, de este modo se construye la siguiente tabla utilizando el algoritmo Myers.

Tabla 4-4 Resultados del algoritmo Myers por fragmento.

Fragmento	**Ubicación del empalme encontrado**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	-	**5**	27	25	26	27	26	27	27
2	**24**	-	**5**	25	26	27	26	27	27
3	26	**25**	-	**5**	26	27	26	27	27
4	26	25	**26**	-	**23**	27	26	27	27
5	26	25	27	**7**	-	27	26	27	27
6	26	25	27	25	26	-	**22**	27	27
7	26	25	27	25	26	**9**	-	**5**	27
8	26	25	27	25	26	27	**25**	-	**5**
9	26	25	27	25	26	27	26	**26**	-

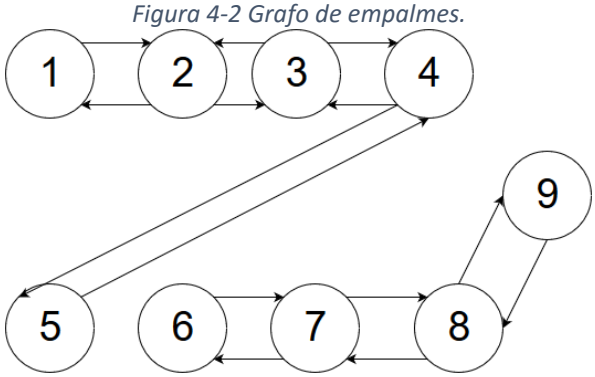
El cálculo de los traslapes se almacena en una lista para cada hilo, donde se agrega la incidencia de traslapes al final. En la tabla 4-4 se muestran los resultados de una búsqueda exhaustiva, los resultados donde se encuentran empalmes en el prefijo o sufijo son los que contienen la ubicación del empalme dentro de los índices de la cadena.

Tabla 4-5 Lista de adyacencia con los empalmes encontrados.

Hilos	**Lista de empalmes**	
1	2	-
2	1	3
3	2	4
4	3	5
5	4	-
6	7	-
7	6	8
8	7	9
9	8	-

La tabla 4-5 muestra la lista que representa el grafo de empalmes construido mediante el algoritmo de Myers, podemos apreciar que no hay relación entre el par de nodos $(5, 6)$, así como la carencia de un arco entre los nodos $\{1, 2, 3, 4, \ 5\}$ y $\{6, 7, 8, 9\}$, esto se resume en la figura 4-2. Este resultado es enviado al procesador para que efectúe la búsqueda en profundidad y almacene en un fichero las cadenas correspondientes a cada grupo.

Figura 4-2 Grafo de empalmes.



En la figura 4-2 tenemos el número de grupos formados y los elementos pertenecientes a cada grupo, con la búsqueda en profundidad podemos ubicar estos elementos e imprimirlos en un fichero.

Tabla 4-6 Grupos impresos por la búsqueda en profundidad.

Grupo 1		**Grupo 2**	
1	GTAGTCATGTTGAAAAACTTACGAGT	6	TCTACCGCCTCCCGTACCCGCGGAGAC
2	ACGAGTAAATTACGTTGTCGAGGGCG	7	CCGCCTCCCGTACCCGCGGAGACGTA
3	AGGGCGTGCAAGTAGCGCAACCCGTG	8	GACGTATCAAACCGACGAAGATTACGA
4	CCCGTGACAAGCGCAAATTCGGAAG	9	TTACGAGGAAGATGACGGAGGGTGGGC
5	CGTGACAAGCGCAAATTCGGAAGTAT		

La tabla 4-6 contiene los grupos formados por el algoritmo propuesto, este resultado puede ser ingresado en un ensamblador de secuencias de ADN y debe formar los contigs indicados en la tabla 4-2.

5. EXPERIMENTACIÓN

Para este trabajo se diseñó un experimento utilizando genomas de virus y bacterias disponibles en la web, se simulan las lecturas de un secuenciador para crear lecturas sintéticas pseudo aleatorias mediante un programa que fue diseñado para guardar un registro de las secciones del genoma que han sido cubiertas de tal manera que se forme determinado número de *contigs*.

El experimento consiste en fragmentar un genoma y agrupar los fragmentos resultantes tal que exista el mismo número de grupos que *contigs* leídos. Los genomas utilizados para los experimentos tienen una longitud pequeña en comparación con otros organismos, con el objetivo de evitar algún sesgo se seleccionaron de forma aleatoria ocho genomas de virus y bacterias. El motivo para seleccionar genomas pequeños se debe al tiempo que toma para agrupar, algunos de estos requieren hasta 1 minuto de procesamiento, dada la cantidad de repeticiones del experimento tomó cerca de 5 horas obtener los resultados presentados en la sección 5.1, de haber utilizado genomas más grandes la cantidad de tiempo requerido para procesar se incrementa de forma considerable.

Tabla 5-5-1 Genomas utilizados para la experimentación.

Genoma	Características Extraídas		
	Longitud	Número de lecturas	Cobertura del genoma (promedio)
Abalone herpesvirus victoria	211519	12691	0.9999
Adoxophyes orana granulovirus	99658	5979	0.9997
Adoxophyes orana nucleopolyhedrovirus	111725	6703	0.9996
African swine fever virus benin	182285	10937	0.9999
Feline coronavirus	29215	1752	0.9998
Shrimp white spot syndrome virus	307288	18437	0.9999
Trichoplusia ni ascovirus 2c	174060	10443	0.9999
Vaccinia virus GLV-1h68	203058	12183	0.9998

Nota: las longitudes se encuentras medidas en el número de pares de bases (pb), los genomas aquí presentados fueron obtenidos de European Nucleotide Archive, ENA www.ebi.ac.uk y son de libre acceso.

El número de lecturas necesarias se fue determinado con la ayuda de la ecuación de Eric Lander y Mike Waterman $C = \frac{LN}{G}$, con la intención de leer la secuencia completa como se ha mencionado anteriormente.

Se realizaron seis repeticiones para cada genoma con distintas longitudes de prefijos y sufijos con la finalidad de observar la correlación que mantiene con las características extraídas. La máxima longitud permitida para cada fragmento es de 200 pares de bases, pudiendo ser menor debido al tipo de simulación (secuenciación por perdigón). Las longitudes de los prefijos y sufijos se determinaron mediante el tamaño del fragmento, se intentan hallar empalmes desde el 7.5% hasta un 27.5% de su longitud, se realizan repeticiones con las siguientes longitudes de empalme: 15, 17, 23, 29, 40 y 55.

Para cada ejecución del experimento se generan las lecturas de cada genoma, se obtiene el número de *contigs* que fueron leídos y se buscan los grupos con los empalmes mencionados. El parámetro por defecto para indicar la diferencia de similitud entre las cadenas para el algoritmo de Myers es cero, dado que estamos realizando lecturas sin error y no existe la poliploidía en el fichero del genoma. Posteriormente se compara el número de grupos formados con el número de *contigs* leídos cuyo valor debe ser cercano o igual dependiendo de la longitud de los empalmes.

5.1 Resultados obtenidos

Al agrupar los fragmentos resultantes del simulador de secuencias de los ocho genomas, para cada una de las seis repeticiones del experimento, con los tamaños de prefijos y sufijos mencionados anteriormente, se obtuvieron los resultados presentados en las siguientes tablas. Los grupos formados con la configuración de longitud en prefijos y sufijos de mayor relevancia para cada genoma en cada ejecución del experimento se encuentran marcados en negrita.

Tabla 5-2 Grupos encontrados en la primera ejecución.

Número	Genoma	Longitud de empalme (pb)						Contigs leídos
		15	17	23	29	40	55	
1	Abalone herpesvirus victoria	2	2	**2**	4	16	33	2
2	Adoxophyes orana granulovirus	2	**2**	3	5	5	7	1
3	Adoxophyes orana nucleopolyhedrovirus	1	2	2	**2**	4	9	2
4	African swine fever virus benin	1	**2**	3	4	8	23	2
5	Feline coronavirus	1	1	1	1	**1**	3	1
6	Shrimp white spot syndrome virus	1	1	**1**	3	8	35	1
7	Trichoplusia ni ascovirus 2c	1	2	**2**	3	8	13	2
8	Vaccinia virus GLV-1h68	1	1	**4**	4	6	15	3

En el resultado de la tabla 6-1, corresponde a la primera ejecución y podemos apreciar que no en todos los genomas se obtuvo el número de grupos que es equivalente al número de *contigs*, en el último genoma se leyeron 3 *contigs* pero al agrupar los fragmentos de las lecturas simuladas se obtuvieron 4 grupos, un grupo más, de forma similar para el segundo genoma se obtuvieron 2 grupos, mientras que solo se leyó un *contig*. Para los genomas restantes se obtuvo el número de lecturas esperadas con búsquedas de diferentes longitudes en los empalmes.

El segundo genoma en esta ejecución requiere de una longitud de empalme menor a 15 pares de bases, mientras que el último necesita una longitud entre 18 y 22. Los genomas que se pueden agrupar de la misma manera con una longitud de empalme distinta tienen mayor relevancia dado que al encontrar empalmes más grandes existen mejores posibilidades para que esos fragmentos formen un *contig* durante la tarea de ensamble.

Tabla 5-3 Grupos encontrados en la segunda ejecución.

Número	Genoma	Longitud de empalme (pb)						Contigs leídos
		15	17	23	29	40	55	
1	Abalone herpesvirus victoria	1	1	3	**3**	10	19	2
2	Adoxophyes orana granulovirus	1	**3**	4	5	6	10	2
3	Adoxophyes orana nucleopolyhedrovirus	**1**	2	3	4	7	13	1
4	African swine fever virus benin	**1**	5	7	8	11	21	3
5	Feline coronavirus	3	3	3	**3**	4	4	3
6	Shrimp white spot syndrome virus	1	**1**	3	4	11	38	1
7	Trichoplusia ni ascovirus 2c	1	**1**	2	5	6	13	1
8	Vaccinia virus GLV-1h68	1	3	**5**	6	14	24	4

Los resultados de la segunda ejecución corresponden a la tabla 6-2, donde los datos de los genomas 1, 2, 4 y 8 no formaron el mismo número de grupos que de *contigs* para las longitudes de empalme utilizadas, en los genomas restantes se logró formar la misma cantidad de grupos que *contigs* leídos. Para el primer y último genoma la longitud de empalme necesaria para encontrar la misma cantidad de grupos como contigs leídos se encuentra en un número mayor a 17 y menor que 23 pares de bases, en el segundo genoma la longitud de empalme debe ser de 16 pares de bases, de igual forma para el genoma 4 se necesita una longitud de 16 pares de bases.

Tabla 5-4 Grupos encontrados en la tercera ejecución.

Número	Genoma	Longitud de empalme (pb)						Contigs leídos
		15	17	23	29	40	55	
1	Abalone herpesvirus victoria	2	2	2	**3**	12	29	4
2	Adoxophyes orana granulovirus	1	**3**	3	3	5	13	2
3	Adoxophyes orana nucleopolyhedrovirus	1	1	1	**1**	4	9	1
4	African swine fever virus benin	1	**1**	4	5	6	20	2
5	Feline coronavirus	**2**	2	3	3	4	8	1
6	Shrimp white spot syndrome virus	1	**2**	3	3	10	37	2
7	Trichoplusia ni ascovirus 2c	1	1	**2**	5	12	23	2
8	Vaccinia virus GLV-1h68	2	**2**	6	7	9	25	3

De los *contigs* leídos en la tabla 6-3, solo el número de grupos formados en los genomas 3, 6 y 7 corresponden con el número de *contigs* leídos, sin embargo los demás genomas se agruparon con una diferencia de un grupo respecto a los *contigs* leídos.

Tabla 5-5 Grupos encontrados en la cuarta ejecución.

Número	Genoma	Longitud de empalme (pb)						Contigs leídos
		15	17	23	29	40	55	
1	Abalone herpesvirus victoria	1	1	2	**4**	15	32	6
2	Adoxophyes orana granulovirus	**2**	5	5	5	12	24	2
3	Adoxophyes orana nucleopolyhedrovirus	1	**3**	5	7	11	19	3
4	African swine fever virus benin	6	**11**	16	18	23	35	9
5	Feline coronavirus	**4**	4	4	5	5	8	2
6	Shrimp white spot syndrome virus	3	4	**7**	10	19	43	7
7	Trichoplusia ni ascovirus 2c	2	**3**	7	15	22	41	4
8	Vaccinia virus GLV-1h68	1	4	**6**	8	16	45	6

Los resultados de la cuarta ejecución se presentan en la tabla 6-4, en esta tabla se presenta una mayor diferencia en el número de grupos respecto al número de *contigs* leídos. En los genomas 1, 4 y 5 hay dos grupos de diferencia respecto a los de referencia. El genoma 7 tiene una diferencia de un grupo y el resto de genomas se agrupó en un número igual a los *contigs* correspondientes.

Tabla 5-6 Grupos encontrados en la quinta ejecución.

Número	Genoma	Longitud de empalme (pb)						Contigs leídos
		15	17	23	29	40	55	
1	Abalone herpesvirus victoria	1	1	1	**1**	10	23	1
2	Adoxophyes orana granulovirus	2	**2**	3	6	9	11	2
3	Adoxophyes orana nucleopolyhedrovirus	**1**	2	2	4	10	20	1
4	African swine fever virus benin	**2**	3	6	7	10	18	2
5	Feline coronavirus	2	2	**2**	3	3	4	2
6	Shrimp white spot syndrome virus	1	1	2	**5**	19	42	4
7	Trichoplusia ni ascovirus 2c	1	1	**2**	6	3	10	2
8	Vaccinia virus GLV-1h68	**1**	2	5	6	15	36	1

Para la quinta ejecución, el único genoma agrupado con diferencia de un grupo es el genoma 6. En donde se espera que se formen cuatro grupos como dado que es el número de *contigs* leídos, mientras que el número de grupos es 5 para un tamaño de empalme de una longitud de 29, mientras que el número de grupos correcto se formará entre 24 y 28 pares de bases, el resto de los genomas se agrupa en el número de *contigs* leídos.

Tabla 5-7 Grupos encontrados en la sexta ejecución.

Número	Genoma	Longitud de empalme (pb)						Contigs leídos
		15	17	23	29	40	55	
1	Abalone herpesvirus victoria	1	1	2	**2**	8	26	2
2	Adoxophyes orana granulovirus	1	**2**	4	4	4	15	2
3	Adoxophyes orana nucleopolyhedrovirus	1	1	1	**2**	5	16	2
4	African swine fever virus benin	1	**3**	5	5	11	22	2
5	Feline coronavirus	1	1	1	**1**	2	4	1
6	Shrimp white spot syndrome virus	1	1	4	**6**	14	40	6
7	Trichoplusia ni ascovirus 2c	1	1	1	**4**	8	14	3
8	Vaccinia virus GLV-1h68	2	2	3	**4**	7	22	4

En el sexto resultado se presenta en la tabla 6-6, se obtienen dos valores distintos a los *contigs* leídos para la configuración de prefijos y sufijos buscados, estos valores se encuentran en los genomas 4 y 7, en los cuales se obtuvieron los valores 3 y 4, mientras que los *contigs* leídos fueron 2 y 3 respectivamente. Se forma un grupo de más para los genomas indicados, en cambio, en los genomas restantes se obtuvo el mismo número de grupos que de *contigs*.

En los resultados obtenidos se ha observado el tiempo de ejecución en relación al número de hilos utilizados para el cálculo de los ocho genomas del experimento. El equipo de cómputo utilizado para esta parte del experimento cuenta con las especificaciones presentadas en la tabla 5-8.

Tabla 5-8 Características del equipo de pruebas.

Componente	Descripción	
Procesador	Intel Core i7 3770K (3.9 GHz)	
Memoria	16 GB de memoria DDR3 (1600 MHz)	
Tarjeta de vídeo	GeForce GTX 970	
	Núcleos CUDA	1664
	Frecuencia de reloj	1178 Mhz
	Memoria	GDDR5 4GB

Figura 5-1 Tiempos de ejecución para el genoma Abalone herpesvirus victoria.

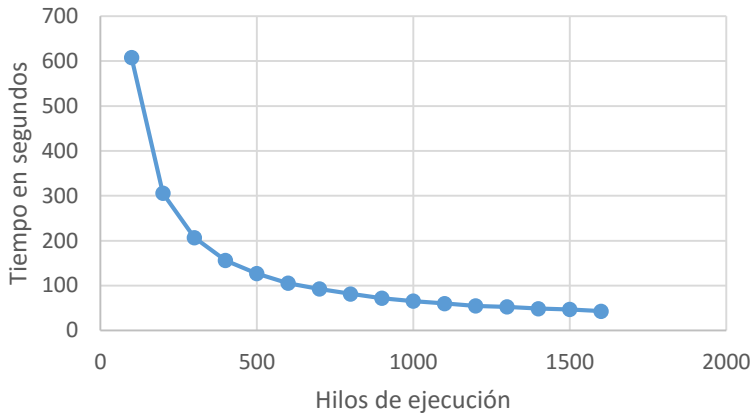


Figura 5-2 Tiempos de ejecución para el genoma Adoxophyes orana granulovirus.

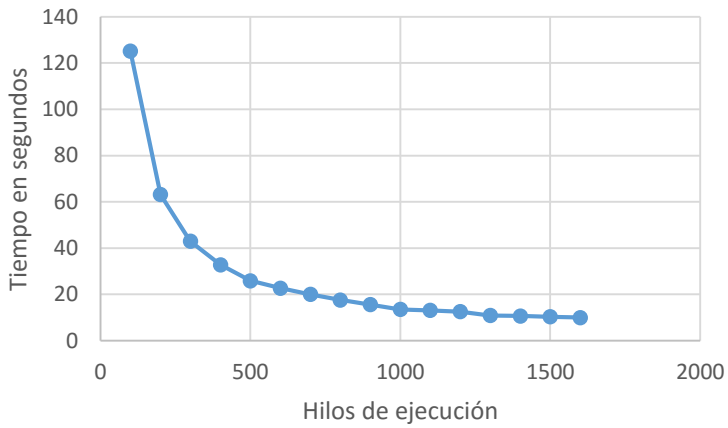


Figura 5-3 Tiempos de ejecución para el genoma Adoxophyes orana nucleopolyhedrovirus.

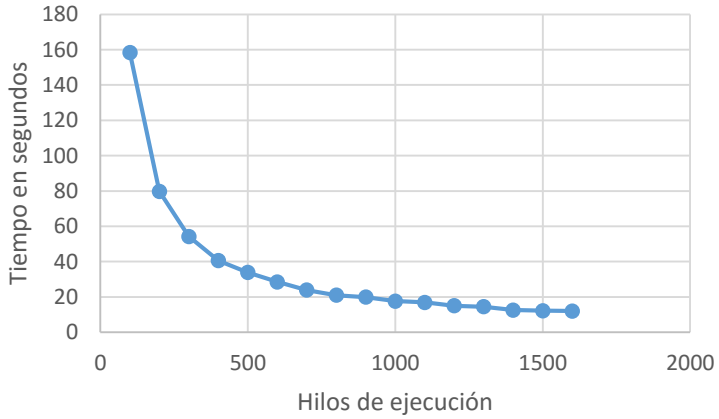


Figura 5-4 Tiempos de ejecución para el genoma African swine fever virus benin

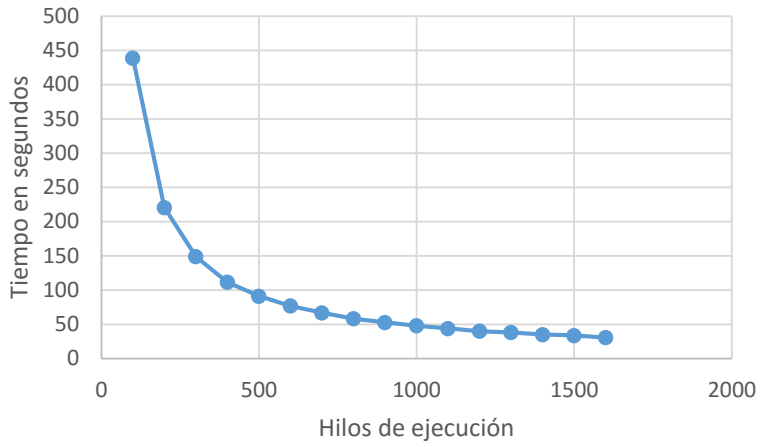


Figura 5-5 Tiempos de ejecución para el genoma Feline coronavirus.

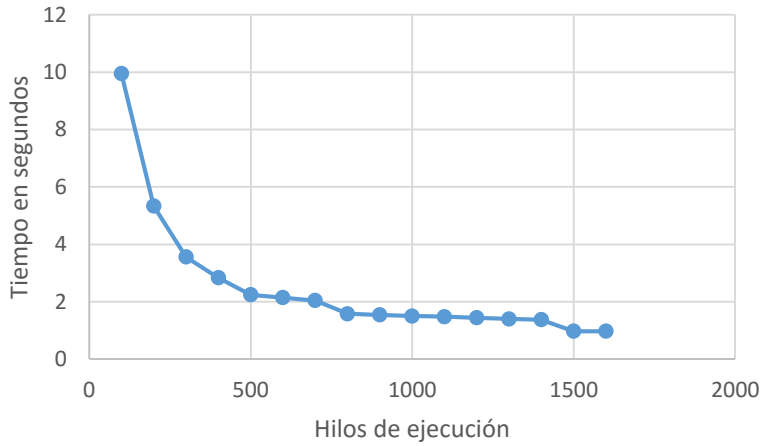


Figura 5-6 Tiempos de ejecución para el genoma Shrimp white spot syndrome virus.

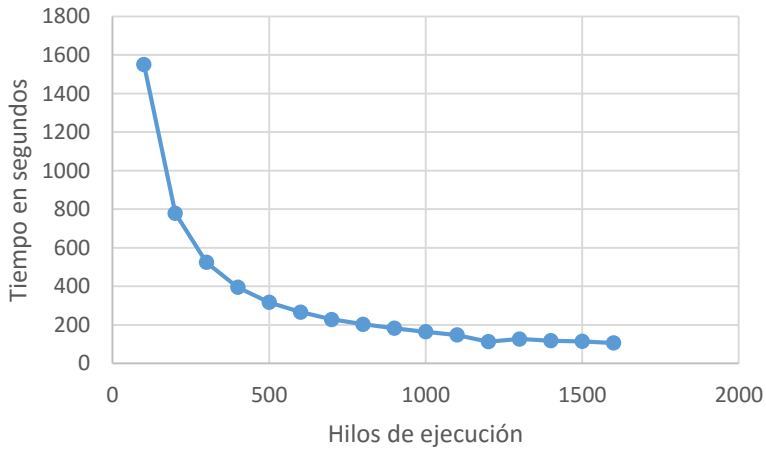


Figura 5-7 Tiempos de ejecución para el genoma Trichoplusia ni ascovirus 2c.

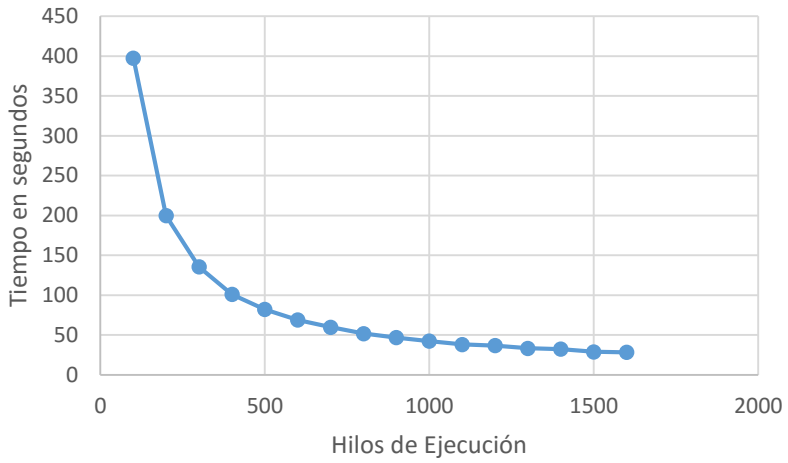
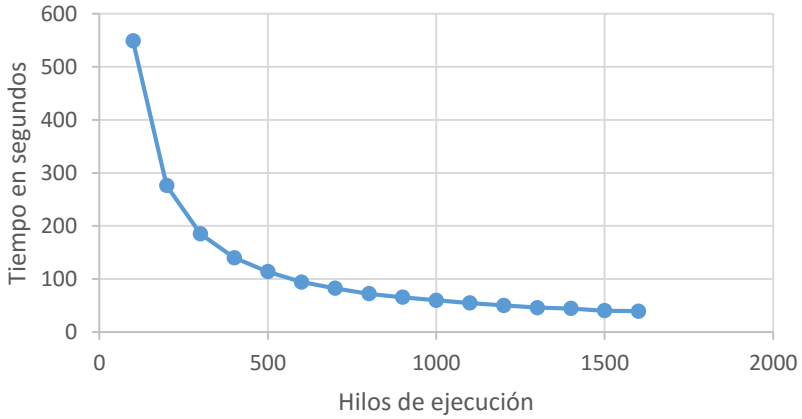


Figura 5-8 Tiempos de ejecución para el genoma Vaccinia virus GLV-1h68.



En las figuras 5-1 a 5-8 se presentan las gráficas del comportamiento temporal de la propuesta de algoritmo de este trabajo, con un número de hilos de ejecución en el rango [100, 1600], se obtiene una función decreciente, a medida que se incrementa la cantidad de hilos se reduce el tiempo necesario para completar las tareas, obedeciendo la ley de Amdahl el tiempo se reduce al mínimo necesario para ejecutar las tareas que no se pueden realizar en paralelo.

6. CONCLUSIONES Y TRABAJO FUTURO

En los resultados obtenidos se logra formar subconjuntos de fragmentos de secuencias de ADN con probabilidad elevada de ensamblarse en *contigs*, dentro de estos el número de grupos con valor más parecido al número de *contigs* leídos y con mayor empalme se encuentra en negrita, esto se debe a que al existir un mayor número de bases traslapadas será más probable que estos grupos se alineen para formar un *contig* en una posterior tarea de ensamble. De forma similar sucede para los casos donde no aparece una coincidencia entre el número de grupos formados con el de los *contigs* leídos. Al reducir el número fragmentos pertenecientes al conjunto inicial para dividirlo en subgrupos se resta complejidad a la tarea, beneficiando el tiempo de procesamiento requerido para formar el ensamble de los *contigs*.

Los resultados obtenidos nos señalan que para genomas más grandes se debe aumentar la redundancia deseada y realizar esto no garantizará de ninguna forma que se cubra el genoma de manera completa, ni que se tenga una buena cobertura del genoma completo para la tarea de ensamble.

Otro resultado que no es evidente se encuentra relacionado con la capacidad de cómputo actual, en el trabajo presentado se utilizó esta tecnología para encontrar traslapes en los fragmentos, está tarea es parte importante de algunas técnicas de ensamble de secuencias, con estos datos podemos afirmar que se cuenta con la capacidad de cálculo suficiente para lograr ensamblar los genomas con ayuda de las tarjetas gráficas.

En adición a este trabajo se puede explorar realizar la búsqueda de grupos con otras técnicas, aplicando transformaciones en las cadenas como la transformada de Burrows – Wheeler o utilizado *k-mers* para utilizar grafos de De Bruijn. Para optimiza el uso de memoria se puede codificar los símbolos utilizados para representar cada base y de este modo procesar genomas más grandes.

LISTA DE FIGURAS

LISTA DE TABLAS Y ALGORITMOS

REFERENCIAS

[1] Vera V., Francisco, & González B., Jesús A. (2014). LPS: una estrategia de ensambles de secuencias cortas de ADN (Tesis de maestría). Instituto Nacional de Astrofísica, Óptica y Electrónica. Tonantzintla Pue., México.

[2] Seemann, T. (2015). WGS in public health microbiology. [Presentación] http://pt.slideshare.net/torstenseemann/wgs-in-public-health-microbiology-mduvidrl-seminar-wed-17-jun-2015, University of Melbourne.

[3] Programming Guide : CUDA Toolkit Documentation. (2016). [online] Docs.nvidia.com. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ [Accessed 8 Feb. 2016].

[4] Pareek, C., Smoczynski, R. and Tretyn, A. (2011). Sequencing technologies and genome sequencing. Journal of Applied Genetics, 52(4), pp.413-435.

[5] Jones, N. and Pevzner, P. (2004). An introduction to bioinformatics algorithms. Cambridge, MA: MIT Press.

[6] Masoudi-Nejad, A., Narimani, Z. and Hosseinkhan, N. (2013). Next Generation Sequencing and Sequence Assembly. SpringerBriefs in Systems Biology.

[7] Baeza-Yates and G. Navarro, R. (1999). Faster Approximate String Matching. Algorithmica, 23(2), pp.127-158.

[8] Minetti, G., Alba, E. and Leguizamón, M. (2011). Problema de ensamblado de fragmentos de ADN resuelto mediante metaheurísticas y paralelismo. Doctor en ciencias de la computación. Universidad Nacional de San Luis.

[9] Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM, 46(3), pp.395-415.

[10] DanishAli, S. and Farooqui, Z. (2013). Approximate Multiple Pattern String Matching using Bit Parallelism: A Review. International Journal of Computer Applications, 74(19), pp.47-51.

[11] Hyyrö, H. (2005). Bit-parallel approximate string matching algorithms with transposition. Journal of Discrete Algorithms, 3(2-4), pp.215-229.

[12] Saikrishna, V. and Ray, S. (2013). Improved Approximate Multiple-Pattern String Matching using Consecutive N-Grams. International Journal of Computer Applications, 81(2), pp.26-31.

[13] TARHIO, J. and PELTOLA, H. (1997). String Matching in the DNA Alphabet. Softw: Pract. Exper., 27(7), pp.851-861.

[14] Blanco Diez, A. and Martín Ayuso, V. (2016). Implementación de algoritmos de ensamblaje de genomas en sistemas de memoria compartida y memoria distribuida. Maestría. Universidad Politécnica de Madrid.

[15] Chen, D. and Cheng, X. (2002). Pattern recognition and string matching. Dordrecht: Kluwer Academic Publishers.

[16] Enrique A., Gabriel Luque, Sami Khuri. (2005). Assembling DNA Fragments with Parallel Algorithms. IEEE.

[17] Hennessy, J., Patterson, D. and Asanović, K. (2012). Computer architecture. Waltham, MA: Morgan Kaufmann.