Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Zacatenco

Departamento de Ingeniería Eléctrica

Sección de Computación

Particionamiento paralelo eficiente del algoritmo con complejidad $O(n^4)$ para el problema de RNA

Tesis que presenta:

Ing. Gibran Jalil Cruz Villa

para obtener el Grado de:

Maestro en Ciencias

en la Especialidad de:

Ingeniería Eléctrica

Opción:

Computación

Director de la Tesis:

Dr. Arturo Díaz Pérez

Ciudad de México, México. Octubre 2005

Dedicatoria

A mis padres José y Graciela,
por una vida de constante
esfuerzo y superación.

A Hannah Itzel,
¡Bienvenida a la familia!
Que tu feliz llegada sea
la señal de lo hermoso que
puede ser el futuro.

Agradecimientos

A mis padres, por ayudarme a trazar cada una de las metas en mi vida, por mostrarme que todo el esfuerzo que requiere ser mejor persona vale la pena. A mi padre, por transmitirme los valores fundamentales en mi formación, esos que nunca pasarán de moda. A mi madre, por quitar de mi vida el miedo a ser una mejor persona.

A mis hermanos Edwin y Dulce, por ser los compañeros de toda una vida. A Edwin, por ser el amigo incondicional y completamente necesario para cualquier persona, por siempre estar dispuesto a ayudarme a crecer un poco más. A Dulce, por toda la alegría y cariño que me regalaste, por la inspiración que encontré en tu constancia y dedicación.

A Stephanie, por sumarte a mis sueños y permitirme ser parte de los tuyos. Gracias por todo el apoyo y amor brindados.

A esos amigos los cuales hicieron del camino compartido algo mejor. A Luis y Jesús, porque nueve años de amistad nunca serán en vano. A Jonathan, Daniel, Rogelio, Wilhem, Emmanuel y tantos otros no menos importantes.

A mi asesor, Dr. Arturo Díaz Pérez, por todo el conocimiento, apoyo y orientación indispensables para la realización de esta tesis.

A los doctores Carlos A. Coello Coello y Guillermo B. Morales Luna, por sus valiosas revisiones y aportaciones a este trabajo.

A Sofia Reza por todo el apoyo administrativo brindado, así como su disposición por ayudarnos. A todo el personal administrativo de la Biblioteca de Ingeniería Eléctrica por su apoyo en la búsqueda de material fundamental para este trabajo.

Al Consejo Nacional de Ciencia y Tecnología por el apoyo financiero indispensable para la realización de este posgrado.

Resumen

Una molécula de ácido ribonucleico (RNA) consiste en una larga secuencia de subunidades -ribonucleótidos- enlazadas entre ellas. La cadena de subunidades, llamadas bases, es conocida como estructura primaria, pero cuando las bases se enlazan entre sí, forman lo que se conoce como estructura secundaria. Se puede obtener una gran cantidad de estructuras secundarias a partir de una sola estructura primaria. El problema del RNA consiste en obtener, de entre todas las combinaciones posibles, aquella estructura secundaria que sea estable, es decir, la que optimiza la liberación de energía. El espacio de búsqueda de este problema crece de manera exponencial, por lo que utilizar un algoritmo de fuerza bruta es infactible. Se tienen dos algoritmos, propuestos por Kruskal y Sankoff, los cuales resuelven este problema en un tiempo computacional razonable, estos algoritmos son identificados por su complejidad computacional, $O(n^3)$ y $O(n^4)$. La diferencia entre estos dos algoritmos radica en el conjunto de soluciones que pueden explorar. A pesar de su complejidad polinomial, ambos algoritmos requieren tiempos de ejecución grandes cuando resuelven el problema del RNA para estructuras primarias largas. Por lo tanto, en esta tesis presentamos un algoritmo paralelo para el algoritmo $O(n^4)$, el cual reduce su tiempo de ejecución. Se analizaron las dependencias entre los datos a utilizar por el algoritmo $O(n^4)$ para dividir, de manera equitativa, el trabajo computacional en varios procesadores. El buen desempeño de un algoritmo paralelo se logra al reducir los tiempos de comunicación y sincronización.

Puede existir más de una estructura secundaria con la mínima liberación de energía, y en biología puede ser importante comparar éstas soluciones. Por lo tanto, un segundo aspecto considerado en esta tesis es, obtener todas las posibles soluciones a partir de una estructura primaria dada. Basados en las tablas de programación dinámica utilizadas en los algoritmos $O(n^3)$ y $O(n^4)$, se propusieron modificaciones en el proceso de llenado, de tal manera que se

almacenara la información requerida para rastrear todas las posibles soluciones.

Se presentan resultados experimentales, los cuales demuestran que los dos aspectos considerados en esta tesis fueron resueltos de manera adecuada. Para el algoritmo paralelo se presentan resultados que muestran un mejor desempeño comparado contra otras estrategias implementadas.

Abstract

A molecule of ribonucleic acid (RNA) is made up of a long chain of subunits -ribonucleotides- linked together. The chain of subunits, called bases, is recognized as the primary structure, but, when they form bonds with one another they create a secondary structure. Several secondary structures can be obtained from a single primary structure. The RNA problem is concerned about obtaining, among all possible combinations, the most stable secondary structure, that is, the one that optimizes the free energy. The search space for this problem grows in an exponential way, then a brute-force algorithm is unfeasible. Two algorithms proposed by Kruskal and Sankoff can solve the problem in a reasonable amount of computational time, they are identified by their computational complexity, $O(n^3)$ and $O(n^4)$. The difference between them is the subset of possible secondary structures they can explore. Despite their polynomial complexity, both algorithms require long running times when they solve very long primary sequences. Therefore, in this thesis we present a parallel algorithm for the $O(n^4)$ strategy which reduces execution time. We review data dependencies in the $O(n^4)$ algorithm to divide the computational effort among several processors. A good performance of the parallel algorithm is obtained by reducing synchronization and communication costs.

More than one secondary structure with the minimal energy can exist and, in biology, it might be important to compare several solutions. Then, a second issue considered in this thesis deals with obtaining all possible solutions for a single primary structure. Based on the dynamic-programming tables of $O(n^3)$ and $O(n^4)$ algorithm, we propose a modification in the filling process to record all necessary information which permits to track all possible solutions.

We present experimental results which demonstrate that both aspects are adequately

solved. For the parallel algorithm we provide results which improve performance as compared to other implementation strategies.

Índice general

Índice de figuras

Índice de cuadros

Índice de algoritmos

Introducción

Bajo condiciones normales, una cadena de ribonucleótidos se enrolla y dobla, las bases formarán enlaces entre ellas en patrones complejos. De esta manera, la molécula conforma ciclos. Ambas, la conformación y el patrón de enlaces, forman la *estructura secundaria* de la molécula de RNA. Dada la estructura primaria (cadena de ribonucleótidos) del RNA, se puede imaginar un vasto número de estructuras secundarias que se pueden formar apareando las cuatro posibles bases que la conforman (adenina - A, citosina - C, guanina - G y uracil - U) de la siguiente manera: A's con U's y C's con G's. De acuerdo a las leyes de la química termodinámica, de cualquier modo, *generalmente* sólo una estructura será estable, es decir, la estructura secundaria que optimice la energía liberada. Un método para deducir la estructura secundaria estable a partir de la estructura primaria es extremadamente útil. Encontrar el método más eficiente para este fin es el denominado: "Problema del RNA" [1]. Es muy importante estudiar este problema pues en biología existe una clara conexión entre estructura y funcionalidad. Una vez que se conoce la forma de la molécula su función se puede identificar.

El costo computacional para la solución de este problema combinatorio crece de manera exponencial. Cuando un problema requiere un tiempo computacional elevado, y sus características lo permiten, se buscan algoritmos que requieran un tiempo menor. Para el diseño de algoritmos con estas características, se utilizan algunas herramientas como la programación dinámica o las técnicas heurísticas, en las cuales se da solución al problema, ya sea dando una aproximación al óptimo o dividiendo el problema original en subproblemas, los cuales son resueltos en un tiempo menor. Otra alternativa es la paralelización de un algoritmo secuencial, de esta manera dos o más procesadores trabajan en conjunto para resolver el mismo problema y así reducir el tiempo total requerido. No existen técnicas generales para poder realizar la paralelización de un algoritmo, por lo que se debe hacer un estudio particular del

problema, el cual incluya el análisis de las estructuras de datos a utilizar y sus dependencias. El problema del RNA, por el consumo de tiempo que representa, es interesante para su estudio y sobre todo en la reducción del tiempo requerido para su solución. En esta tesis se realizará un estudio para la paralelización del algoritmo $O(n^4)$[1], el cual se basa en la estrategia denominada "Programación dinámica". Debido a la alta dependencia entre datos se requiere que la estrategia disminuya la necesidad de comunicación entre los procesadores. Existe un paquete computacional (Vienna RNA package) el cual es actualmente el sistema de software más citado para resolver el problema del RNA en paralelo. Su aritmética para el balance de carga es satisfactorio pero su diseño no optimiza las comunicaciones. En 1998, Chen et al. probaron experimentalmente que para secuencias de longitud 9212 las comunicaciones toman aproximadamente el 50 % del tiempo de CPU [2]. En el 2002 Mireya Tovar presenta una tesis en la cual se estudia un algoritmo de complejidad $O(n^3)$, así como tres estrategias de particionamiento para la paralelización del algoritmo. Además, se presenta una comparación de los rendimientos de las tres distintas versiones. Existe además el problema de la existencia de más de una estructura secundaria estable. Es decir, cuando más de una posible estructura secundaria logra la mínima liberación de energía, los sistemas actuales presentan sólo una de estas estructuras. Esta tesis incluirá el diseño de un algoritmo que permita encontrar todas las estructuras secundarias óptimas. Como se mencionó antes, este trabajo se enfocará en el algoritmo de complejidad $O(n^4)$ el cual, a diferencia del algoritmo con complejidad $O(n^3)$, permite encontrar estructuras secundarias más complejas, es decir, nos permite evaluar ciclos anidados (estructuras de orden ≤ 2, o sea ciclos que se encuentran dentro de otro ciclo) lo cual entrega más flexibilidad en la búsqueda de estructuras.

El objetivo principal de esta tesis es desarrollar una estrategia de particionamiento eficiente para el algoritmo paralelo de complejidad $O(n^4)$ del problema del RNA. Las preguntas que se atacaron en este trabajo son: ¿Cómo realizar de una manera eficiente la paralelización del algoritmo de complejidad $O(n^4)$ para el problema del RNA? y ¿Cómo almacenar todas las estructuras secundarias estables encontradas en el proceso de búsqueda?

El contestar estas preguntas es importante debido a la gran cantidad de tiempo que es requerido para la solución del problema del RNA, además de que en Biología Molecular es útil el tener todas las posibles estructuras secundarias estables, y sobre éstas tomar una

decisión.

El primer objetivo se logra disminuyendo el tiempo empleado en las comunicaciones entre los procesadores encargados del cálculo. Se estudiará el particionamiento de los datos de tal manera que se minimicen las comunicaciones entre procesadores, disminuyendo esto a su vez el tiempo de cómputo. Se presentará un análisis de los resultados obtenidos al modificar la estrategia de particionamiento. El otro objetivo importante de esta tesis es obtener todas las estructuras secundarias estables de una estructura primaria dada. Es decir, todas aquellas posibles estructuras que compartan la liberación de energía mínima. Para realizar esto se modificará el tipo de rastreo realizado para reconstruir la estructura secundaria, de modo que se puedan encontrar todas las estructuras secundarias que son estables a partir de una estructura primaria dada.

La metodología que se siguió para lograr los objetivos de este trabajo consistió de los siguientes pasos:

1. Se hizo una revisión de la literatura relacionada al problema del RNA y su implementación paralela.

2. Se construyó un prototipo funcional para la obtención preliminar de resultados y análisis de alcances reales del proyecto. Este prototipo trabaja de manera secuencial.

3. Se diseñó la estructura del sistema final, esto con la finalidad de tener una estructura formal sobre la cual se trabajó.

4. Se realizaron pruebas de cada una de las fases propuestas en el diseño del sistema.

5. Se realizó la unión de los componentes individuales antes desarrollados.

6. Se evaluaron y obtuvieron los resultados del sistema.

7. Se reportaron de manera detallada los resultados obtenidos.

Como resultado del trabajo desarrollado se obtuvo un esquema de particionamiento eficiente para el algoritmo de complejidad $O(n^4)$ propuesto por Sankoff para el problema del RNA optimizando las comunicaciones requeridas para resolver este problema de manera paralela. Se realizó la implementación de este esquema en lenguaje C utilizando la biblioteca

para paso de mensajes MPI. Se modificó la propuesta original para que de esta manera el sistema obtuviera todas las estructuras secundarias estables de una estructura primaria dada. Éstas se muestran mediante la representación de paréntesis, siendo de manera textual tanto la entrada como la salida de los datos.

El presente documento de tesis está organizado en cinco capítulos de la siguiente manera. En el capítulo 1 se estudia a profundidad el problema del RNA y se explican los dos algoritmos de ($O(n^3)$ y $O(n^4)$) [1], haciéndose un estudio de sus principales diferencias. El capítulo 2 presenta los conceptos más importantes en la programación paralela, así como las herramientas más utilizadas en la actualidad para su programación. En el capítulo 3 se muestran los algoritmos desarrollados para la obtención de todas las estructuras secundarias estables así como los resultados que éstos presentaron. El capítulo 4 presenta el diseño e implementación de la paralelización del algoritmo con complejidad $O(n^4)$. Para finalizar, el capítulo 5 muestra los resultados obtenidos al implementar los algoritmos del capítulo 4, así como un análisis de los mismos.

Capítulo 1

El problema del RNA

En este capítulo se abordará el problema del RNA (Ácido Ribo-Nucleico) y se explicarán dos de los algoritmos más utilizados para su solución [1]. Además, se presentarán las restricciones existentes para la formación de una estructura secundaria a partir de su estructura primaria.

1.1. Descripción del problema

En todas las especies presentes en la actualidad en la Tierra la información genética es almacenada y transmitida gracias a los ácidos nucleicos, tanto en forma de DNA como de RNA. Pero además, esta información puede leerse y expresarse en forma de moléculas funcionales: las proteínas.

Bajo condiciones normales, una cadena de ribonucleótidos se enrolla y dobla y las bases formarán enlaces entre ellas en patrones complejos. De esta manera, la molécula conforma ciclos. Ambas, la conformación y el patrón de enlaces, son llamadas *estructura secundaria* de la molécula de RNA.

Una estructura primaria del RNA está formada de una sucesión de ribonucleótidos, cada uno de los cuales contiene una de cuatro posibles bases: adenina (A), citosina (C), guanina (G) y uracil (U). A partir de la estructura primaria, se puede imaginar un vasto número de estructuras secundarias que se pueden formar apareando A's con U's y C's con G's de diferentes maneras. A cada una de estas estructuras secundarias se asocia una cierta cantidad

de energía liberada, la cual puede ser calculada, entre otros factores, a partir del número de bases apareadas en la estructura. De acuerdo a las leyes de la química termodinámica *generalmente* sólo una estructura será estable, es decir, la estructura secundaria que optimice la energía liberada. Un método para obtener la estructura secundaria estable a partir de la estructura primaria es extremadamente útil. Encontrar el método más eficiente para este fin es el denominado: "Problema del RNA".

Es muy importante estudiar este problema pues en biología existe una clara conexión entre estructura y funcionalidad. Una vez que se conoce la forma de la molécula su función se puede identificar.

1.1.1. Restricciones de una estructura secundaria

Supongamos que una molécula es una secuencia $\mathbf{a} = a_1, ..., a_n$ donde cada a_i puede ser A, G, C o U. La secuencia $\mathbf{a}$ es llamada *estructura primaria*. Se usa $a_i \cdot a_j$, donde $i < j$, para indicar que a_i y a_j forman un par, o están apareadas entre sí de la manera Watson-Crick. Una estructura S para $\mathbf{a}$ es un conjunto de ese tipo de pares. Se requiere que los elementos de S satisfagan las siguientes restricciones:

1. Complementariedad de Watson-Crick: Si S contiene a la pareja $a_i \cdot a_j$, entonces a_i y a_j son respectivamente A y U, U y A, G y C o C y G.

2. No traslapamiento: Si S contiene a la pareja $a_i \cdot a_k$, entonces S no puede contener a ninguna pareja $a_i \cdot a_j$ con $j \neq k$ ni $a_j \cdot a_k$ con $j \neq i$.

3. No anudamientos: Si $i_1 < i_2 < i_3 < i_4$, entonces las parejas $a_{i_1} \cdot a_{i_3}$ y $a_{i_2} \cdot a_{i_4}$ no pueden existir en S de manera simultanea.

4. Sin vueltas cerradas: Si S contiene a la pareja $a_i \cdot a_j$, entonces $|j - i| \geq 4$.

De la segunda restricción se obtiene que cada a_i ocurre en exactamente un par o ningún par, y a_i se describe como apareada o no apareada respectivamente.

Como todas estas restricciones limitan las posibles formas de una estructura secundaria, podemos referirnos a éstas como "restricciones geométricas", aunque éstas deriven de una variedad de consideraciones estereoquímicas, mecánicas y termodinámicas.

Debido a que la secuencia **a** permanece constante, es conveniente referirnos a a_i como i, por ejemplo, $i \cdot j$ significa $a_i \cdot a_j$.

1.1.2. Tipos de representación de una estructura secundaria

Se han propuesto algunas formas de representar a una estructura secundaria a partir de su estructura primaria. En [3] se presentan algunas de las más conocidas. A continuación se describen estas maneras de representación.

- *Pictórica.* La estructura secundaria se representa uniendo las bases consecutivas, separadas de una manera equidistante, por medio de líneas curvadas, además de unir cada uno de los pares formados. La figura 1.1 presenta un ejemplo de este tipo de representación, en la cual se tiene una estructura de veintinueve bases y su estructura secundaria está definida por los pares:

$$S = \{1 \cdot 29, 2 \cdot 28, 3 \cdot 27, 4 \cdot 26, 5 \cdot 25, 6 \cdot 24, 8 \cdot 20, 9 \cdot 19, 12 \cdot 17\}$$

- *Círculo.* Propuesta por [4]. En ésta, las bases que conforman la estructura primaria se colocan equidistantes una de la otra a lo largo del perímetro de un círculo. Las bases apareadas se unen mediante arcos de círculo. La figura 1.2 presenta un ejemplo de este tipo de representación.

- *Árbol.* Cada par de la estructura secundaria se representa por medio del vértice de un grafo, y un arco lleva ventaja de un vértice a otro si los pares que éstos representan son exteriores o interiores del mismo ciclo. Existen dos grandes desventajas de esta representación, la primera es que pierde información acerca de las bases no apareadas que están en los ciclos y la segunda es la pérdida de orientación en la molécula, es decir, no se sabe donde empieza ni donde termina. La figura 1.3 presenta un ejemplo de este tipo de representación.

Existe otra manera de representación para una estructura secundaria, que es tal vez la más utilizada. En ésta se presenta una secuencia de paréntesis a cada uno de los cuales corresponde uno de los nucleótidos en la estructura primaria. Las bases apareadas se representan mediante los pares de paréntesis, uno abierto y uno cerrado, correspondientes a un par de bases.

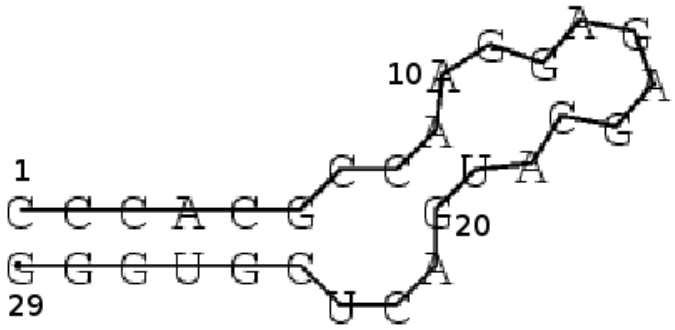

Figura 1.1: Ejemplo de representación pictórica

La representación por paréntesis fue la seleccionada para utilizarse en este trabajo, siendo ésta la más sencilla para llevar a una representación pictórica.

1.1.3. Ciclos

Cualquier estructura secundaria S puede ser descrita de manera única y natural, formada de subestructuras llamadas *ciclos, pares apilados* y *regiones aisladas*. En una estructura secundaria, cuando se forma un par (A con U o G con C) digamos $i \cdot j$, entonces decimos que éste determina el ciclo s desde i hasta j, el cual consiste de los símbolos $i, i+1, ..., j-1, j$. Los *ciclos* de S se denotan como $s_1, ..., s_i$. Si $i \cdot j$ es un par y $i < r < j$, se dice que $i \cdot j$ rodea r. De manera similar, $i \cdot j$ rodea a un par $p \cdot q$, si rodea tanto a p como a q respetando la restricción 3. Si ocurre que hubiese pares $p_1 \cdot q_1$, $p_2 \cdot q_2$, ..., $p_{k-1} \cdot q_{k-1}$ con extremos entre $i+1$ y $j-1$ inclusive, accesibles desde $i \cdot j$, es decir que no están rodeados por algún otro par diferente a $i \cdot j$ decimos que el ciclo s es de orden k o que es un $k - ciclo$.

A continuación se presentarán los seis tipos de subestructuras permitidas en una estructura secundaria.

1. Si S contiene a $i \cdot j$ y ninguno de los elementos $i+1, ..., j-1$ está apareado, el ciclo que se forma es denominado *horquilla*.

2. Si S contiene a $i \cdot j$, $i+1 \cdot j-1$, ..., $i+h \cdot j-h$, se dice que cada uno de estos pares (excepto el último) se *apila* al siguiente, es decir, son *pares apilados*.

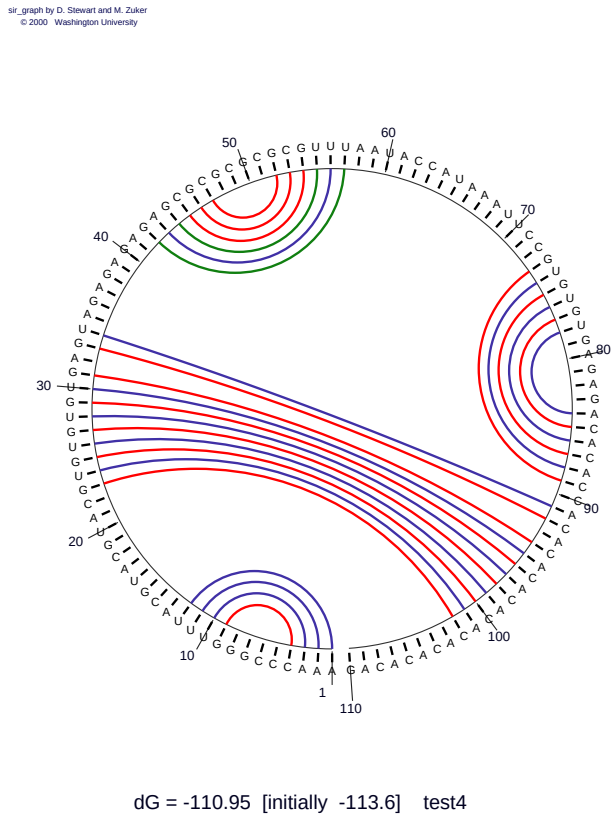

Figura 1.2: Ejemplo de representación por círculo

3. Si $i+1 < p < q < j-1$ y S contiene a $i \cdot j$ y a $p \cdot q$, pero los elementos entre i y p están no apareados al igual que los elementos entre q y j, entonces estas dos regiones no apareadas se denominan *ciclo interno*.

4. Si S contiene a $i \cdot j$ y a $(i+1) \cdot q$, y existen algunos elementos no apareados entre q y j, estos elementos forman un *bulto*.

5. Si S contiene a $i \cdot j$ y $i \cdot j$ rodea dos o más pares, los cuales no se rodean entre sí, entonces se forma un *ciclo múltiple*.

6. Si r no está apareado y no hay ningún par en S que lo rodee, entonces se dice que r está en una región aislada.

La figura 1.4 ilustra cada una de las subestructuras antes presentadas.

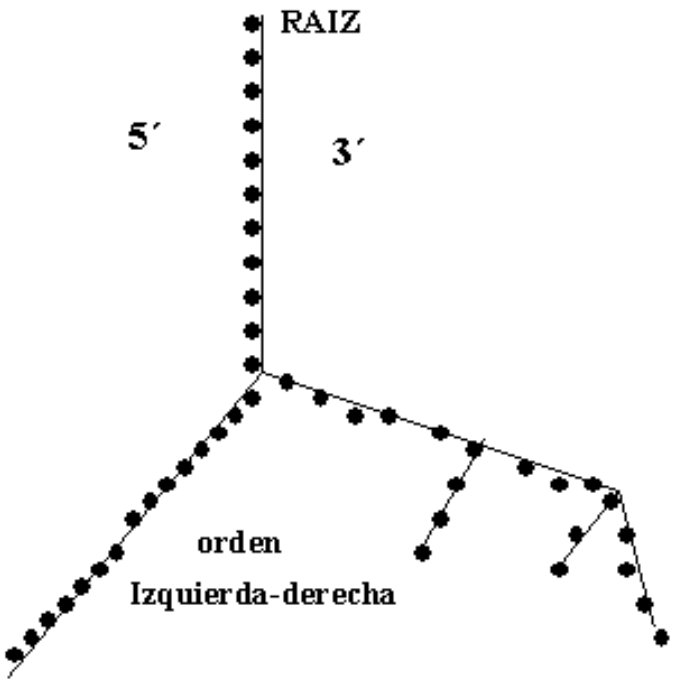

Figura 1.3: Ejemplo de representación de árbol

1.2. Algoritmos para resolver el problema del RNA

Sankoff y Kruskal [1] propusieron dos algoritmos para resolver el problema del RNA los cuales son de complejidad $O(n^3)$ y $O(n^4)$ y ambos utilizan "programación dinámica"[5] como estrategia de solución del problema. En esta sección se mostrarán los dos algoritmos propuestos para resolver el problema del RNA. El primero que se presentará tiene una complejidad $O(n^3)$, y se muestra en las ecuaciones 1.1 y 1.2. Inmediatamente después se presentará el algoritmo con complejidad $O(n^4)$ (ecuaciones 1.4, 1.5 y 1.6) y se discutirán sus principales diferencias.

Es posible estimar la energía liberada E por una estructura secundaria S con cierto grado de confianza, suponiendo que está puede ser expresada como una suma de términos donde cada término es asociado con cada uno de los ciclos s_r,

$$E(S) = e(s_1) + e(s_2) + \cdots + e(s_t)$$

donde S representa la estructura secundaria y $s_1, s_2, ..., s_t$ son los ciclos que la conforman. La función $e(s)$ determina el valor de energía del ciclo s [1].

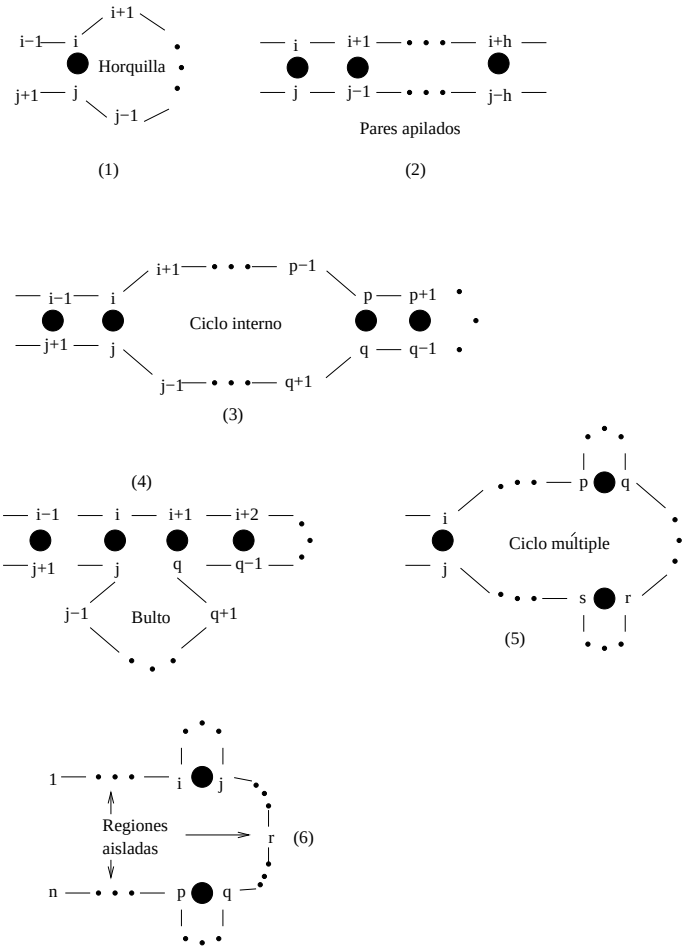

Figura 1.4: Tipos de subestructuras

La función $F(i,j)$ (ecuación 1.1) calcula la mínima energía a liberar por la secuencia $[i,j]$. Por lo que la mínima energía de toda la secuencia se encontrará al evaluar $F(1,n)$, siendo n la longitud de la estructura primaria. Esta función busca el valor mínimo entre los tres posibles casos a evaluar. En el primero, se viola una de las restricciones antes revisadas (vueltas cerradas), por lo que no le será asignada energía. El segundo caso se evalúa cuando i y j forman un par. Se dice que la cadena $[i,j]$ es *cerrada* si i está apareado con j. La función $C(i,j)$ [1] toma la energía asociada a este par $(e(s))$ y agrega la energía asociada a la subsecuencia $[i+1, j-1]$. El tercer y último caso busca el valor mínimo de la suma de las energías asociadas a la estructura $[i,j]$ dividida en dos subsecuencias, $[i,h]$ y $[h+1,j]$,

[1]Esta manera de definir funciones será la utilizada en esta tesis, de tal forma que se eviten confusiones, y las bases G y C queden completamente diferenciadas de las funciones $G(i,j)$ y $C(i,j)$.

tomando h todos los valores posibles en el intervalo $i \leq h < j$. En resumen $F(i,j)$ queda definida, por casos, como se muestra en la ecuación 1.1.

$$F(i,j) = min \begin{cases} 0 & j - i < 4 \\ C(i,j) & \text{si } [i,j] \text{ es cerrada} \\ \min\limits_{i \leq h < j} [F(i,h) + F(h+1,j)] & \text{otra forma} \end{cases} \tag{1.1}$$

donde

$$C(i,j) = e(s) + F(i+1, j-1) \tag{1.2}$$

Debido a que se deben favorecer los enlaces entre A's y U's ($A \cdot U$ o $U \cdot A$) sobre los enlaces entre G's y C's ($G \cdot C$ o $C \cdot G$), la función $e(s)$ (ecuación 1.3) entregará los siguientes valores:

$$e(s) = \begin{cases} -2 & \text{si } (i,j) \text{ es un par } A \cdot U \text{ o } U \cdot A \\ -1 & \text{si } (i,j) \text{ es un par } G \cdot C \text{ o } C \cdot G \\ 0 & \text{otro caso} \end{cases} \tag{1.3}$$

El algoritmo con complejidad $O(n^4)$ utiliza la función $G(i,j)$ para calcular cada valor de la tabla de programación dinámica. Esta función se define de la siguiente manera:

$$G(i,j) = min \begin{cases} G_1(i,j) \\ \min\limits_{i \leq h < j} [G(i,h) + G(h+1,j)] \end{cases} \tag{1.4}$$

Se observa que la ecuación 1.4 analiza dos casos de entre los cuales se seleccionará el valor mínimo. En el primer caso se buscan estructuras que contengan ciclos de orden 1 y 2. Esto se realiza al llamar a la función $G_1(i,j)$ (ec. 1.5). El segundo caso busca el valor mínimo de la suma de las energías asociadas a la estructura $[i,j]$ dividida en dos subsecuencias, $[i,h]$ y $[h+1,j]$, tomando h todos los valores posibles en el intervalo $i \leq h < j$.

La función $G_1(i,j)$ (ec. 1.5) es la encargada de buscar ciclos de orden 1 y 2. Al igual que $G(i,j)$ analiza dos casos de los cuales seleccionará el valor mínimo. El primer caso hace una llamada a la función $C(i,j)$ (ec. 1.6) que explora los ciclos de orden 1 y 2. El segundo caso busca entre el traslape de las dos tablas de programación dinámica (las usadas para $G(i,j)$ y $G_1(i,j)$) el valor mínimo de la suma de las energías asociadas a la estructura $[i,j]$ dividida

en dos subsecuencias, $[i,h]$ y $[h+1,j]$, tomando h todos los valores posibles en el intervalo $i \leq h < j$.

$$G_1(i,j) = min \begin{cases} C(i,j) \\ \underset{i \leq h < j}{min}[G_1(i,h) + G(h+1,j), G(i,h) + G_1(h+1,j)] \end{cases} \tag{1.5}$$

$C(i,j)$ (ec. 1.6) es el valor óptimo de la secuencia $[i,j]$.

$$C(i,j) = \begin{cases} min \begin{cases} f_1(i,j) \\ \underset{i<p_1<q_1<j}{min}[f_2(i,j,p_1,q_1) + C(p_1,q_1)] \\ \underset{i+1 \leq h < j-1}{min}[G_1(i+1,h) + G_1(h+1,j-1)] + e(i,j) \end{cases} \\ \qquad\qquad \text{si } [i,j] \text{ es cerrada,} \\ \qquad\qquad \text{de otra forma,} \\ indefinido \end{cases} \tag{1.6}$$

Si la secuencia $[i,j]$ es cerrada se toma el mínimo de tres casos a analizar, el primero hace una llamada a la función f_1, la cual es la encargada de evaluar los ciclos de orden 1. El segundo caso realiza una iteración de llamadas (de las cuales obtendrá el valor mínimo) a las funciones $C(p_1,q_1)$ y f_2, siendo ésta la encargada de evaluar los ciclos de orden 2. Se utilizarán los índices auxiliares p_1 y q_1 los cuales tomarán todos los posibles valores en el intervalo $i < p_1 < q_1 < j$, **siendo éste el caso que entrega un tiempo proporcional a $O(n^4)$ para la ejecución del algoritmo**. El tercer caso busca el valor mínimo de la suma de las energías asociadas a la estructura $[i+1,j-1]$ dividida en dos subsecuencias, $[i+1,h]$ y $[h+1,j-1]$, tomando h todos los valores posibles en el intervalo $i+1 \leq h < j-1$ y adicionando el valor de energía del par exterior $i \cdot j$ a esta suma. Normalmente f_1 y f_2 dependen de la identidad de las bases apareadas y del número de bases no apareadas.

1.2.1. Diferencia entre los algoritmos $O(n^3)$ y $O(n^4)$

Existen diferencias notables entre los dos algoritmos presentados por Kruskal y Sankoff. Las diferencias más importantes son:

- El algoritmo de complejidad $O(n^3)$ presenta buenos resultados para secuencias de tamaño corto y orden ≤ 2, pero conforme la longitud de un ciclo crece, la teoría termo-

dinámica sugiere que la energía liberada debe contener un término que sea proporcional a log(longitud del ciclo). Debido a que en el algoritmo $O(n^3)$ el término es proporcional a la longitud del ciclo para secuencias de longitud mayor se requiere un ajuste al algoritmo que cumpla con esta ley.

- El algoritmo de complejidad $O(n^4)$ presenta un buen análisis de ciclos múltiples (orden $k = 2$ y $k = 3$), además de poder analizar ciclos de longitud mayor que el de complejidad $O(n^3)$. Este algoritmo logra un mejor resultado al agregar una función de verificación dentro de estructuras encontradas (con lo que obtenemos una complejidad $O(n^4)$), permitiendo la linealidad en el análisis de energía liberada.

La función f_2 de la ecuación 1.6 permite que se puedan analizar ciclos múltiples dentro de una estructura, es decir, que se pueda manejar el valor que se asignará al ciclo interno de acuerdo a su longitud, bases apareadas, etc.

Otro problema es que la solución al problema del RNA puede no ser única, como se menciona con anterioridad. *Generalmente* sólo una estructura será estable, es decir, puede aparecer más de una estructura secundaria que presente la misma mínima energía liberada. En biología molecular es útil el tener todas las posibles estructuras secundarias estables, y sobre éstas tomar una decisión.

1.3. Estrategias para la solución del problema

El problema del RNA tiene un espacio de búsqueda que presenta un crecimiento exponencial, es decir, el número total de combinaciones posibles para una estructura de RNA crece de manera exponencial con respecto al número total de nucleótidos en la estructura primaria. Para evitar manejar un espacio de búsqueda tan grande, el cual es prácticamente intratable con el poder de cómputo actual, se han utilizado diversas técnicas para encontrar una estructura secundaria óptima sin realizar una búsqueda exhaustiva en todo el espacio de estructuras secundarias.

1.3.1. Programación dinámica

La técnica que utilizan los algoritmos con complejidad $O(n^3)$ y $O(n^4)$ [1] para resolver el problema del RNA es la llamada *Programación dinámica* [5]. La programación dinámica, al igual que el método *divide y vencerás*, resuelve un problema combinando las soluciones de subproblemas. ("Programación" en este contexto se refiere a un método tabular, no a escribir código para una computadora.) Los algoritmos de *divide y vencerás* particionan el problema original en subproblemas independientes, resolviendo estos subproblemas recursivamente, para después combinar sus soluciones y así resolver el problema original. La programación dinámica se distingue de los métodos de *divide y vencerás*, pues en ésta los subproblemas no son independientes, es decir, cuando los subproblemas comparten subsubproblemas. La programación dinámica resuelve estos subsubproblemas una sola vez, a diferencia de los métodos de *divide y vencerás*, y guarda su resultado en una tabla, evitando de esta manera que se resuelva este subsubproblema cada vez que es encontrado.

La programación dinámica se utiliza de manera típica en problemas de optimización. En estos problemas suele existir más de una solución. Cada solución tiene un valor, del cual se desea encontrar el óptimo (máximo o mínimo). Se dice que esta solución es *una solución óptima* al problema, ya que puede existir más de una solución óptima al problema.

El desarrollo de un algoritmo de programación dinámica puede dividirse en una secuencia de cuatro pasos.

1. Caracterizar la estructura de una solución óptima.

2. Recursivamente definir el valor de una solución óptima.

3. Calcular el valor de una solución óptima de una manera *abajo-arriba*.

4. Construir la solución óptima rastreando la información calculada.

Los pasos 1-3 son la base de una solución por medio de programación dinámica a un problema. El paso 4 se utiliza sólo si se requiere saber una solución que entrega ese valor óptimo.

1.3.2. Técnicas heurísticas

Existen otras maneras de encontrar una solución óptima en un espacio de búsqueda sumamente grande. Algunas de ellas son las llamadas *técnicas heurísticas*.

Cuando se enfrenta un problema con espacio de búsqueda muy grande y las técnicas clásicas de búsqueda y optimización son insuficientes entonces se utilizan las técnicas heurísticas. La palabra "heurística" se deriva del griego *heuriskein*, que significa "encontrar" o "descubrir" [6].

El significado del término ha variado históricamente, algunos han usado el término como antónimo de "algorítmico" [7]. Las heurísticas fueron un área predominante en los orígenes de la inteligencia artificial.

Actualmente, el término suele usarse como un adjetivo, refiriéndose a cualquier técnica que mejore el desempeño en promedio de la solución de un problema.

Algunos ejemplos de técnicas heurísticas son los siguientes:

- Búsqueda Tabú
- Recocido simulado
- Escalando la colina
- Algoritmos genéticos

La **Búsqueda Tabú** [8] es realmente una meta-heurística, porque es un procedimiento que debe acoplarse a otra técnica, ya que no funciona por sí sola. La búsqueda tabú usa una "memoria" para guiar la búsqueda, de tal forma que algunas soluciones examinadas recientemente son "memorizadas" y se vuelven tabú (prohibidas) al tomar decisiones acerca del siguiente punto de búsqueda.

El **recocido simulado** [9] está basado en el enfriamiento de los cristales. El algoritmo requiere de una temperatura inicial, una final y una función de variación de la temperatura. Éste es una algoritmo probabilístico de búsqueda local.

La técnica **escalando la colina** se aplica a un punto a la vez (es decir, es una técnica local). A partir de un punto, se generan varios estados posibles y se selecciona el mejor de ellos. El algoritmo no tiene retroceso, ni lleva ningún tipo de registro histórico.

Los **algoritmos genéticos** [10] se derivaron de los conceptos de la evolución biológica. John H. Holland se interesó en los 1960s en estudiar los procesos lógicos involucrados en la adaptación. De tal forma, Holland vio el proceso de adaptación en términos de un formalismo en el que los programas de una población interactúan y mejoran en base a un cierto ambiente que determina lo apropiado de su comportamiento. El combinar variaciones aleatorias con un proceso de selección debía entonces conducir a un sistema adaptativo general.

1.3.3. Tamaño del espacio de búsqueda

Antes de continuar es conveniente presentar de manera simple el número total de parentizaciones a verificar si se realizara una búsqueda de manera exhaustiva. Denotemos el número de posibles parentizaciones de una estructura primaria de n bases como $P(n)$. Cuando $n \leq 4$, sólo se tiene una forma de estructura secundaria ya que no es posible realizar apareamientos entre las bases. Cuando $n > 4$ es posible dividir el número total de parentizaciones en una estructura secundaria como el producto de dos parentizaciones de subestructuras secundarias, y la división de estas subestructuras puede ocurrir entre el elemento k-ésimo y $(k+1)$-ésimo para cualquier $k = 1, 2, ..., n-1$. Además es importante considerar el caso en que **no todas las bases** se apareen. Es decir, en el problema de RNA pueden existir largas (sub)estructuras secundarias con un solo enlace. Esto añade un término constante a cada iteración de la sumatoria antes descrita. Con lo anterior se obtiene la recurrencia presentada en la ecuación 1.7.

$$P(n) = \begin{cases} 1 & \text{si } n = 1, 2, 3, 4. \\ \sum_{k=1}^{n-1} P(k)P(n-k) + P(2) & \text{si } n \geq 5 \end{cases} \tag{1.7}$$

Con lo que se obtiene una recurrencia del tipo

$$T(n) = \begin{cases} 1 & \text{si } n = 1, 2, 3, 4. \\ (n-5) + \sum_{k=1}^{n-1} T(k)T(n-k) & \text{si } n \geq 5 \end{cases} \tag{1.8}$$

La recurrencia 1.8 se puede resolver de manera simple mediante el método de sustitución, con lo cual obtenemos que el resultado para ésta es $\Omega(2^n)$. Debido a que la solución a esta recurrencia nos entrega un número de soluciones exponencial con respecto a n, se puede decir

que un método de "fuerza bruta" para resolver este problema es una estrategia ineficiente.

Cabe destacar que al utilizar programación dinámica para resolver el problema del RNA se obtienen pocos subproblemas, a diferencia de una búsqueda exhaustiva, ya que se tiene un problema por cada entrada (i, j) a la tabla de programación dinámica, es decir, para cada (i, j) que satisfaga $4 \leq i \leq j \leq n$, lo que entrega un espacio igual a $\Theta(n^2)$ para almacenar las soluciones en las tablas $F()$ y $C()$ para el algoritmo con complejidad $O(n^3)$ (ecuaciones 1.1 y 1.2) y $G()$, $G_1()$ y $C()$ para el algoritmo con complejidad $O(n^4)$ (ecuaciones 1.4, 1.5 y 1.6). Esto nos muestra que utilizar una estrategia de programación dinámica para resolver el problema del RNA es mucho más eficiente que enumerar todas las posibles estructuras secundarias y analizar cada una.

1.4. Trabajo relacionado

Dada la importancia del problema del RNA, existen diversos trabajos que abordan aspectos diferentes del problema. A continuación presentamos un breve resumen del trabajo relacionado.

1.4.1. Transcripción de DNA en RNA

Tanto el DNA como las proteínas son macromoléculas con papeles biológicos esenciales. Ambas son polímeros -que se pueden formar en condiciones prebióticas- de unidades repetitivas esenciales en la vida, nucleótidos y aminoácidos. La información genética está codificada en la secuencia de nucleótidos de los ácidos nucleicos (Adenina, Guanina, Citosina y Timidina para DNA, y Adenina, Guanina, Citosina y Uracilo para RNA). Además, la especificidad de unión entre los pares de bases (C-G, A-T, A-U) permite la replicación haciendo que una secuencia actúe de molde, en la generación de otra que mantiene el orden de la secuencia gracias a la complementariedad. De esta manera se transmite la información genética.

El mecanismo molecular de esta conversión es la traslación en la que una secuencia de nucleótidos es traducida en una secuencia de aminoácidos. El proceso tiene lugar en dos etapas: la transcripción de DNA a RNA y la traducción del RNA a proteína. La conversión de la información del DNA en información proteica no es directa. El DNA es primero transcrito

a RNA y éste traducido a proteína. El tipo particular de RNA formado, es llamado RNA mensajero (mRNA). Existen otros tipos de RNA, como el RNA de transferencia (tRNA) y el RNA ribosomal (rRNA). En una representación simbólica, la transcripción es tan sólo el cambio de la letra T de la secuencia de DNA por la letra U para obtener la secuencia de RNA. Es así un flujo unidireccional de expresión de la información desde el DNA al RNA y de éste a las proteínas. Este flujo, junto con el flujo de transmisión de la información de RNA a DNA, forman el dogma central de la biología molecular, enunciado por Crick en 1958.

1.4.2. Bases de datos de RNA

Las bases de datos para las estructuras secundarias del RNA contienen, dada una estructura primaria, la estructura bidimensional que se ha de formar y la posible función que desempeña. Estos repositorios son extremadamente útiles para dos casos específicos:

- Teniendo una estructura primaria, buscar la estructura secundaria que se formará (siempre y cuando ésta se encuentre previamente almacenada), de esta manera se evita el cálculo de la estructura secundaria estable, el cual puede ser muy costoso en tiempo.

- Teniendo la estructura secundaria, se pueden buscar las funciones que desempeña esa proteína con respecto a su forma. Para esto es requerido otro análisis denominado "medida de similitud", el cual permite, a partir de una estructura secundaria dada, caracterizar ésta con respecto a otra similar (cuando la misma formación no sea encontrada.)

Existe un repositorio extremadamente grande de estructuras de RNA llamado RNABase [11]. Éste ofrece acceso a una base de datos relacional que contiene todos los datos acerca de todas las estructuras de RNA disponibles públicamente. Este repositorio es actualizado diariamente con estructuras nuevas o revisadas que se ponen a disposición de manera pública. El sistema corre bajo ambiente Linux (Red Hat), utilizando Apache como demonio HTTP y PostgreSQL como manejador de base de datos.

1.4.3. Medidas de similitud del RNA

La anotación [12] de una estructura de RNA es el proceso de extraer los residuos de las conformaciones y las relaciones de los inter-residuos. Éstos se comparan con conformaciones permitidas. Para cada residuo (relación), una valor estadístico de peculiaridad es generado de tal manera que indique la desviación con respecto a la norma de residuos (relaciones) equivalentes observados.

El cálculo del valor de peculiaridad depende de métricas de distancia que fueron diseñadas para permitir la comparación de residuos y relaciones. Estos valores permiten evaluar una estructura dada y cuantitativamente identificar regiones que exhiben una conformación novedosa o inusual.

Existe un software de acceso en línea llamado MC-Annotate, el cual es un evaluador de estructuras de RNA. Éste, a partir de una estructura de RNA, haciendo uso de una base de datos de residuos de las conformaciones y relaciones binarias, obtiene el porcentaje de peculiaridad y permite su clasificación.

Esta herramienta es útil para comparar estructuras secundarias y analizar el grado en que difiere de otras estructuras previamente analizadas. Además permite analizar secciones de la estructura. Es decir, permite identificar formaciones inusuales dentro de la estructura de tal manera que se pueda tomar una decisión más acertada, ya sea como un error en la formación o como una nueva formación.

El problema del RNA es interesante debido a la complejidad que presenta, además de algunos problemas inherentes a él (el rastreo de la estructura secundaria, el tipo de representación a utilizar y la selección del algoritmo a implementar.) Resolver este problema es importante pues permite, a partir de una estrucutra primaria dada, encontrar la estructura secundaria estable y de esta manera deducir la función de la molécula que se forma. Por lo que este trabajo abordará este problema, así como la utilización de la paralelización como método para reducir el tiempo total a utilizar por el algoritmo con complejidad $O(n^4)$ el cual hace uso de la "programación dinámica" como herramienta de solución.

Capítulo 2

Programación paralela

En este capítulo se presentarán los conceptos más importantes de cómputo paralelo así como las herramientas existentes para este fin. Por su complejidad computacional, el problema del RNA es particularmente interesante. Se han estudiado diversas maneras de atacar el problema, una de ellas es la programación dinámica. Sin embargo, como se vió en el capítulo 1 el tiempo requerido para la su solución sigue siendo elevado. Una estrategia para reducir el tiempo total requerido es el uso de paralelismo.

Primero se presentan algunos conceptos fundamentales en el diseño de algoritmos paralelos. A continuación se mostrarán algunas arquitecturas utilizadas, diferentes esquemas de comunicación y finalizaremos abordando las herramientas más utilizadas para el cómputo paralelo así como ejemplos de algunas implementaciones existentes para resolver el problema del RNA haciendo uso de la paralelización.

2.1. Conceptos fundamentales de paralelismo

Existen dos conceptos básicos al hablar de cómputo paralelo: la computación paralela y en lo que consiste una computadora paralela. La computación paralela es el procesamiento de información que enfatiza la manipulación concurrente de elementos de datos pertenecientes a uno o más procesos resolviendo un problema común [13]. Una computadora paralela es una computadora con múltiples procesadores capaz de realizar cómputo paralelo [13]. Como definición alternativa podemos mencionar la siguiente: una computadora paralela es una

colección de elementos de procesamiento que se comunican y cooperan entre sí para resolver problemas grandes de manera rápida [14].

La utilización de una computadora paralela requiere de un algoritmo paralelo el cual explote las capacidades de ésta. La metodología para desarrollar un algoritmo paralelo difiere del desarrollo de un algoritmo secuencial debido a que se requiere tomar en cuenta aspectos propios de la arquitectura así como de los datos a utilizar en el problema.

Una metodología para el diseño y construcción de un algoritmo paralelo consiste de cuatro etapas típicas: descomposición, asignamiento, orquestación y mapeo.

La descomposición divide el cómputo en tareas las cuales a su vez serán divididas en procesos. Las tareas pueden ser estáticas o dinámicas, el número de tareas disponibles puede variar con el tiempo. La descomposición identifica concurrencia. El objetivo de la descomposición es tener suficientes tareas para mantener a lo procesadores ocupados, pero no en exceso. "El número de tareas disponibles en un momento dado es una cota superior a la aceleración alcanzable."

El asignamiento es el mecanismo que especifica como dividir el trabajo entre procesos. Junto con la descomposición es conocido como: **Particionamiento**. Los objetivos principales del asignamiento es el balanceo de carga así como la reducción de costos de comunicación y sincronización.

La orquestación es el paso en el cual se decide cuales procesos se comunican entre sí y de qué manera harán el envío de información. A diferencia de los dos pasos anteriores, en la orquestación la arquitectura, el modelo de programación y el lenguaje de programación juegan un papel muy importante. En ésta se requieren mecanismos para intercambiar datos con otros procesadores (comunicación), sincronización entre todos los procesadores, así como tomar decisiones acerca de la organización de las estructuras de datos, mezclar o dividir mensajes y traslapar cómputo con comunicaciones. Los objetivos principales de la orquestación son promover la localidad de referencias a datos, una calendarización adecuada de tareas para evitar tiempos latentes y la reducción del trabajo adicional para controlar el paralelismo.

En el mapeo cada proceso se asigna a un procesador de manera que se trata de maximizar la utilización de los procesadores y minimizar los costos de comunicación y sincronización.

La figura 2.1 presenta un esquema representativo de la función de cada uno de los pasos antes descritos.

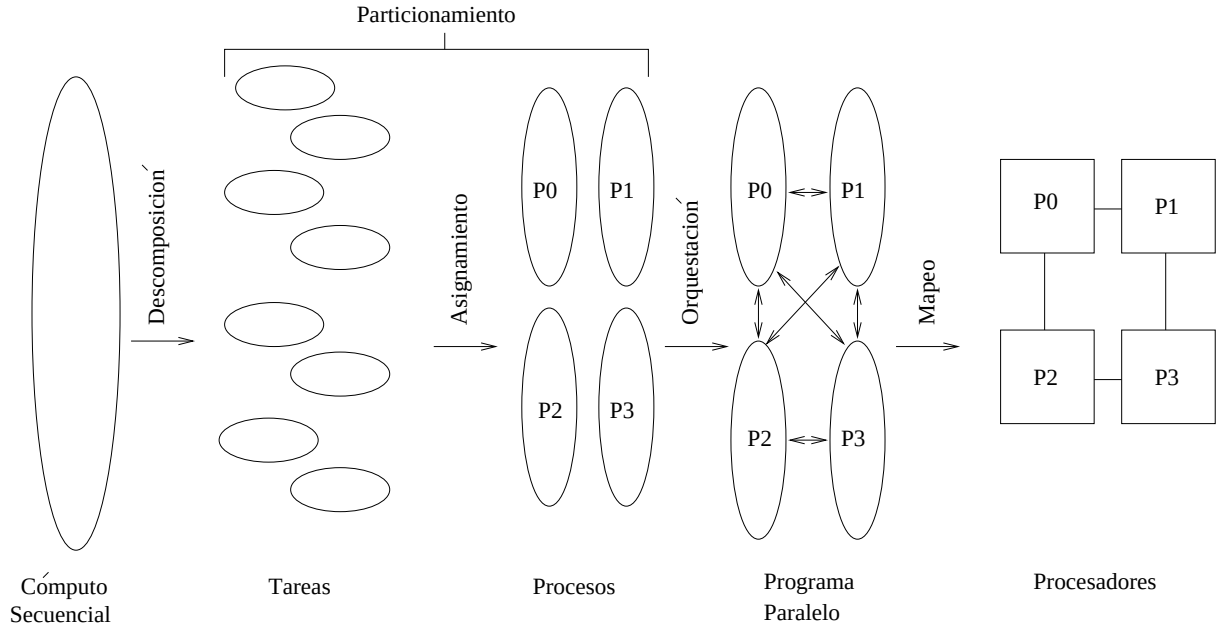

Figura 2.1: Pasos para la creación de un programa paralelo

2.2. Arquitecturas paralelas

Para un diseño correcto de nuestro algoritmo paralelo es necesario conocer la arquitectura paralela a utilizar. Existen distintos tipos de arquitecturas paralelas por lo que esta sección realizará un breve estudio sobre las más importantes en la actualidad. Existen diversas clasificaciones de computadoras paralelas. De manera simple la figura 2.3 sintetiza en un solo esquema la arquitectura de una computadora paralela.

En [15] se presenta un análisis de las tendencias arquitecturales que muestran actualmente las computadoras de alto rendimiento.

Las computadoras secuenciales tradicionales siguen el modelo introducido por John von Neumann. La figura 2.2 presenta el esquema de una computadora con arquitectura SISD (*Single instruction stream, single data stream*).

En la actualidad se utilizan sólo dos tipos de arquitecturas paralelas: las de memoria compartida y las de memoria distribuida. La figura 2.4 muestra un diagrama donde se presenta

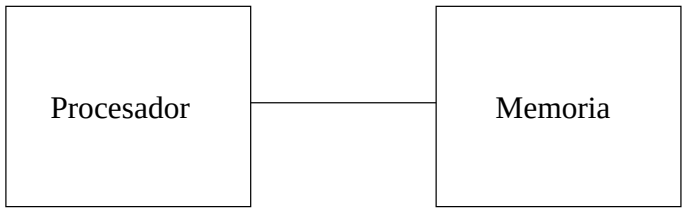

Figura 2.2: Arquitectura SISD

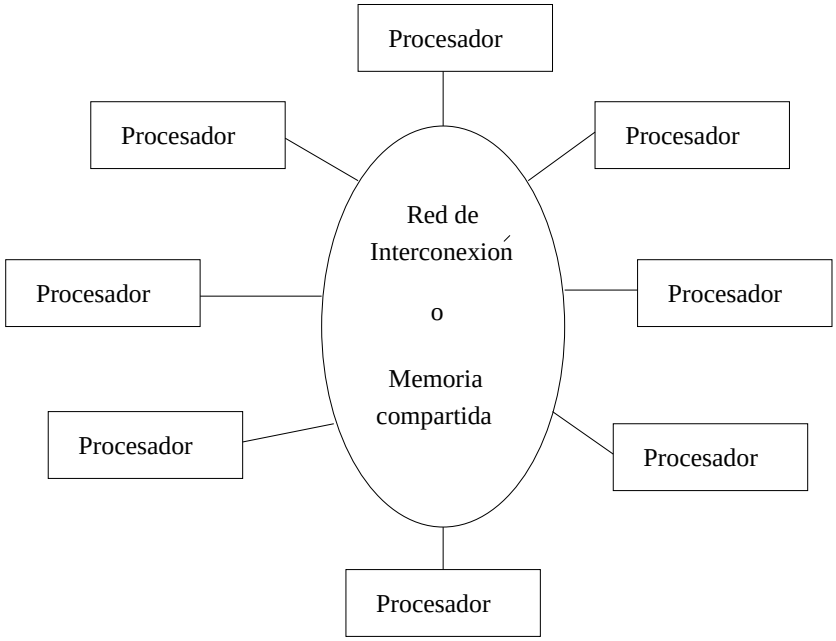

Figura 2.3: Arquitectura de una computadora paralela

esta clasificación así como las arquitecturas más representativas de cada caso.

2.2.1. Arquitecturas de memoria compartida

Las computadoras multi-CPU consisten de un número de procesadores, cada uno de los cuales es capaz de ejecutar un programa independiente. Los multiprocesadores son computadoras multi-CPU con memoria compartida. En un multiprocesador con acceso a memoria uniforme (UMA, por sus siglas en inglés) la memoria compartida es centralizada. En un multiprocesador con acceso a memoria no uniforme (NUMA, por sus siglas en inglés) la memoria compartida es distribuida. En las arquitecturas SMP, al igual que en las UMA, la memoria es centralizada, pero en éstas se tiene un conjunto de procesadores idénticos trabajando

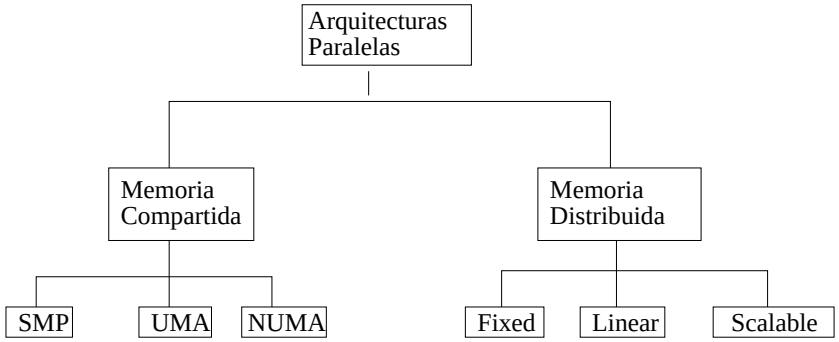

Figura 2.4: Tipos de arquitecturas paralelas

simultáneamente.

Multiprocesadores con acceso a memoria uniforme

El patrón más simple de intercomunicación entre procesadores asume que todos los procesadores trabajan simultáneamente a través de un dispositivo de interconexión hacia la memoria compartida. Existen diversos mecanismos para implementar esta interconexión incluyendo un canal común hacia la memoria global. La figura 2.5 nos muestra un diagrama del modelo de multiprocesador UMA.

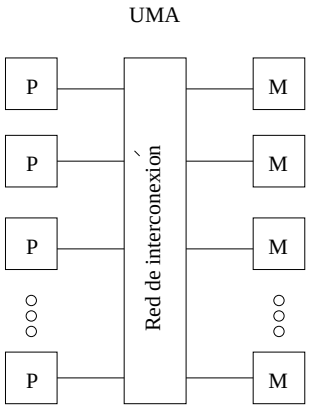

Figura 2.5: Modelo de un multiprocesador con acceso a memoria uniforme

Multiprocesadores con acceso a memoria no uniforme

Al igual que los multiprocesadores UMA, los multiprocesadores NUMA se caracterizan por compartir el espacio de direcciones, pero a diferencia de los multiprocesadores UMA, en

éstos la memoria es distribuida. Cada procesador tiene acceso directo a cierta memoria (memoria local), y el espacio de direcciones compartido es formado combinando estas memorias locales. El tiempo requerido para acceder a cierta localidad de memoria en un multiprocesador NUMA depende de si la dirección es parte de su memoria local o no. La figura 2.6 nos muestra un ejemplo del modelo de multiprocesador NUMA.

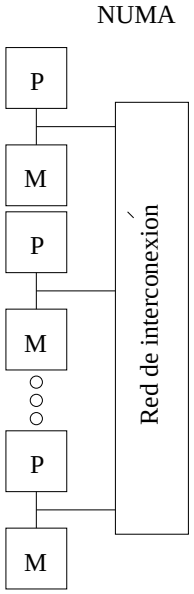

Figura 2.6: Modelo de un multiprocesador con acceso a memoria no uniforme

2.2.2. Arquitecturas de memoria distribuida

Otra arquitectura multi-CPU sin memoria compartida es la multicomputadora. Cada procesador tiene su propia memoria local y la interacción entre procesos ocurre por medio del paso de mensajes. En la figura 2.7 podemos observar que una multicomputadora puede ser vista como un conjunto de computadoras von Neumann, cada una de las cuales está constituida por una unidad de procesamiento central que ejecuta un programa que desempeña una secuencia de operaciones de lectura y escritura en una memoria adjunta. Todos estos nodos (computadoras von Neumann) se encuentran ligados entre sí por medio de un dispositivo de interconexión. Las arquitecturas *Fixed* son aquellas donde las conexiones entre las computadoras no pueden crecer de manera directa, es decir, se tiene que realizar una reestructuración de la arquitectura de interconexión para poder añadir más nodos. En una

arquitectura *lineal* no es necesaria la reestructuración de la arquitectura de interconexión utilizada para añadir nodos individuales de procesamiento, pero sí se tiene una cantidad máxima de nodos de procesamiento a conectar. La arquitectura *Scalable*, a diferencia de las dos anteriores, está diseñada para poder crecer de una manera controlada, es decir, esta arquitectura permite añadir nodos de procesamiento controlando su interconexión a dos niveles, el primero es la interconexión de nodos en "grupos" y el segundo es la conexión entre dispositivos de comunicación lo cual permite la comunicación entre los distintos grupos. Cabe destacar que la mayoría de las supercomputadoras presentadas en [16] entran en esta última categoría.

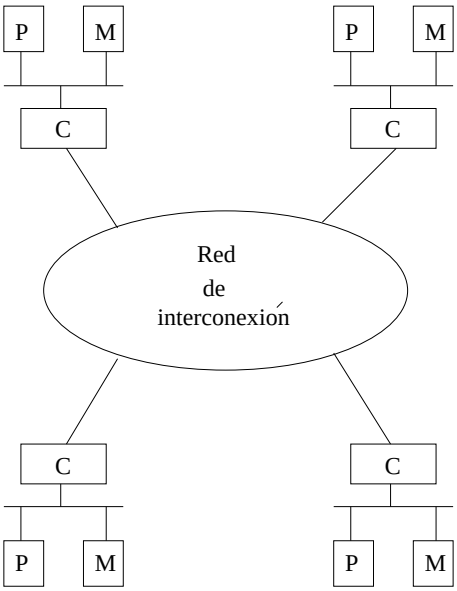

Figura 2.7: Organización de una multicomputadora

2.3. Nuevas tendencias en arquitecturas paralelas

A continuación se presenta con más detalle cada uno de los tipos de computadora paralela presentados en [15].

2.3.1. Computadoras SIMD con memoria distribuida

Una primera arquitectura a analizar es la de las computadoras SIMD (*Single instruction Multiple Data*) con memoria distribuida. Las computadoras de este tipo son conocidas como máquinas con *arreglo de procesadores*. En estas computadoras todos los procesadores ejecutan la misma instrucción al mismo tiempo (pero con diferentes datos), por lo que no se requiere sincronización entre los procesadores. Esto simplifica de gran manera el diseño de estos sistemas. Un *procesador de control* decide qué instrucción se deberá realizar en el arreglo de procesadores. Todas las computadoras de este tipo en la actualidad utilizan un procesador que se comunica con este procesador de control. Las operaciones que no pueden ser realizadas por el arreglo de procesadores ni por el procesador de control son realizadas por este procesador. Es posible excluir procesadores del arreglo al realizar una operación, esto deja a los procesadores excluidos en estado de "listo", y al no realizar operaciones automáticamente se reduce el rendimiento total. Otro factor que reduce el rendimiento de estos sistemas es la necesidad de un procesador por datos alojados en la memoria de otro procesador diferente. Por estas razones, las computadoras SIMD con memoria distribuida son utilizadas en trabajos de cómputo paralelo especializado. Generalmente son utilizadas en procesamiento digital de señales y manipulación digital de imágenes. Un ejemplo de computadora con esta arquitectura es la "Quadrics Appemille".

2.3.2. Computadoras MIMD con memoria compartida

A diferencia de las computadoras SIMD con memoria distribuida, en este tipo de computadoras los procesadores no sólo están enlazados entre sí, sino que cada uno tiene acceso al dispositivo de memoria que se utilice, además de manejar procesadores vectoriales para su arquitectura. En estas arquitecturas el crecimiento del ancho de banda de acceso a la memoria debería crecer de manera lineal junto con el número de procesadores a utilizar. Desafortunadamente, una interconexión completa es muy costosa, creciendo con $O(n^2)$ cuando el número de procesadores crece con $O(n)$. Debido a esto se han utilizado diferentes alternativas de conexión. Algunas, como la conexión "crossbar" sí utilizan n^2 conexiones. Las redes$-\Omega$ utilizan $n \log_2 n$ interconexiones, mientras que las redes de "canal central" utilizan

sólo una conexión. Esta última es la menos costosa de implementar, pero lógicamente la contención en el canal de comunicación a la memoria genera que estas arquitecturas reduzcan su rendimiento.

En la actualidad todas las multicomputadoras con procesadores vectoriales utilizan conexiones "crossbar". Esto es factible pues el número de procesadores a conectar es aún pequeño (máximo 32 procesadores). En estos sistemas MIMD diferentes tareas se pueden ejecutar en diferentes procesadores de manera simultánea. Existen algunas computadoras comerciales bajo esta arquitectura como la "IBM eServer p690", la "SGI Origin3000", y la "Cenju-4".

2.3.3. Computadoras MIMD con memoria distribuida

Este tipo de arquitecturas es sin duda la de más crecimiento en las computadoras de alto rendimiento. La ventaja de esta arquitectura sobre su semejante de memoria compartida es clara: el problema del ancho de banda se elimina debido a que el ancho de banda se escala de manera automática junto con el número de procesadores. Por supuesto, esta arquitectura presenta una desventaja sobre la de memoria compartida: La comunicación entre los procesadores es mucho más lenta que en memoria compartida, por lo tanto, la sobrecarga que genera la sincronización en tareas de comunicación es varios órdenes mayor que la generada en memoria compartida.

Algunas topologías utilizadas para esta arquitectura son: mallas bidimensionales y tridimensionales, "árboles anchos" e hipercubos, aunque actualmente muchas de estas arquitecturas utilizan topologías "crossbar". Un problema común de estas arquitecturas es la gran diferencia entre la capacidad de comunicación y la capacidad de cómputo. Algunas computadoras que utilizan esta arquitectura son las "n-cube", además de la "Connection Machine 5 (CM5)".

2.3.4. Clusters

La adopción de los clusters, colecciones de PC's conectadas en una red local, se incrementó desde la introducción del primer cluster Beowulf en 1994. Su atractivo reside en el (potencialmente) bajo costo tanto de su implementación. Una variación que cabe destacar

es la utilización de nodos con más de un procesador interconectados en un cluster. Esto permite no sólo mejorar el rendimiento y mantener bajos costos (cuando se utilizan sólo un par de procesadores por nodo) sino también disminuir el consumo total de potencia. Desafortunadamente, el gran problema que presentan estas arquitecturas es la alta latencia en las comunicaciones. Debido a esto se utilizan redes de alto rendimiento como las que ofrecen Myrinet (10+10 Gbit/s), Infiniband (20Gb/s y 60Gb/s) y SCI (200-1,000 Mbyte/s). La velocidad de comunicaciones que ofrecen este tipo de redes es comparable con las que ofrecen arquitecturas de sistemas paralelos integrados, como los presentados anteriormente. Esta arquitectura está tomando mucho impulso por sus bajos costos. Ejemplos importantes de clusters son: MareNostrum (ubicada en el Centro de Supercómputo en Barcelona, logrando un máximo de 27910 GFlops), Thunder (ubicada en el Laboratorio Nacional de Lawrence, Livermore, logrando un máximo de 19940 GFlops) y el "System X" (ubicado en la Universidad de Virginia, logrando un máximo de 12250 GFlops.)

2.3.5. El proyecto "Blue Gene"

La arquitectura desarrollada por este proyecto consiste en la unión de circuitos integrados ASIC ("an application-specific integrated circuit") y memoria SDRAM-DDR. Estos circuitos integrados ASIC son un sistema completo de cómputo (incluyendo la interfaz de red y memoria). Los circuitos ASIC son interconectados en "tarjetas de cómputo", las cuales a su vez se unen en "nodos de cómputo", para finalmente unirse en gabinetes. Este proceso de "escalamiento de capacidad" puede resumirse de la siguiente manera:

- Cada circuito ASIC contiene 2 procesadores RISC (Reduced Instruction Set Computer - Computadora con Conjunto de Instrucciones Reducido) de bajo consumo de potencia.
- Cada tarjeta de cómputo contiene 2 circuitos ASIC.
- Cada nodo de cómputo contiene 16 tarjetas de cómputo.
- Cada gabinete contiene 32 tarjetas de cómputo.

El proyecto "Blue Gene" aún se encuentra en desarrollo por lo que su capacidad total de cómputo sólo ha sido calculada de manera teórica. Se puede notar que las multicompu-

tadoras (utilizando éstas el esquema de memoria distribuida), debido a su gran capacidad de escalamiento se están utilizando cada vez más, además de su notable reducción en los costos totales del sistema. Cabe mencionar que el 50 % de los diez sistemas de cómputo más poderosos en la actualidad utilizan la arquitectura del proyecto "Blue Gene". Un ejemplo de esta supercomputadora es la denominada "BlueGene/L", la cual logra un máximo de 136800 GFlops utilizando 65536 procesadores *PowerPC 440* a 700 MHz, por lo que su máximo teórico es de 183500 GFlops.

2.4. Diferentes esquemas de comunicación en programas paralelos

Como se explicó en la sección anterior existen dos tipos básicos de arquitecturas paralelas: las multicomputadoras y las multiprocesadores. Estas últimas son sistemas de memoria compartida donde cualquier dirección de memoria es accesible por cualquier procesador. Para cada una de las arquitecturas de computadoras paralelas existen alternativas de programación paralela.

La programación para memoria compartida utilizada en los multiprocesadores presenta las siguientes alternativas:

- Uso de un nuevo lenguaje de programación: *SISAL, Haskell, etc.*
- Modificación de un lenguaje de programación secuencial: *C*, HPF, Fortran 90, Concurrent Pascal.*
- Usar bibliotecas de funciones con un lenguaje secuencial: *Pthreads, P4, Parmacs.*

El modelo de programación para esta arquitectura utiliza un solo espacio de direccionamiento, lo que significa que la comunicación es implícita. Por lo anterior, son necesarios mecanismos de sincronización en el acceso a los datos para mantener la coherencia de los mismos. Estos mecanismos pueden ser implícitos o explícitos.

Ya que en las multicomputadoras la memoria está distribuida entre los procesadores, es decir, un procesador tiene acceso sólo a su memoria local, su programación es más compleja.

Como en estas arquitecturas no es posible referirse a datos compartidos el único mecanismo para intercambiar información es *el paso de mensajes*. Este mecanismo consiste de dos primitivas básicas:

- send: Utilizada para enviar datos de un proceso a otro.
- receive: Utilizada para recibir datos de un proceso.

El envío de un mensaje de un proceso a otro involucra el movimiento de información de un espacio de direccionamiento a otro, por lo que esta operación toma un tiempo en completarse. La figura 2.8 presenta una representación gráfica del paso de mensajes.

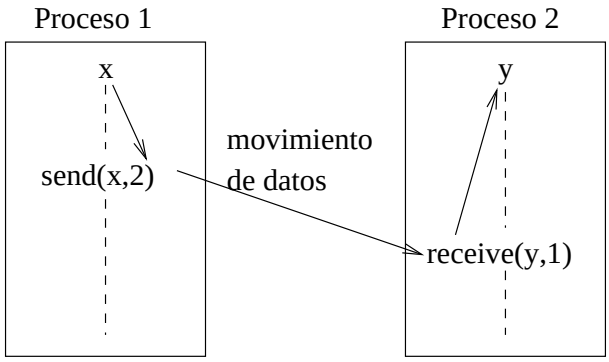

Figura 2.8: Paso de mensajes

El envío de un mensaje involucra el uso de buffers donde se almacenarán temporalmente los mensajes, el uso de etiquetas para la identificación de mensajes, el nombramiento de procesos, así como su sincronización.

Existen diferentes esquemas de comunicación de datos como la comunicación Maestro-Esclavo y la comunicación local. Es necesario conocer y analizar las ventajas y desventas de cada uno de estos esquemas para la utilización eficiente de los recursos.

2.4.1. Comunicación Maestro-Esclavo

El paso de mensajes puede ser fácilmente utilizado en trabajo computacional en el cual el trabajo total pueda ser dividido en tareas, y éstas se puedan realizar independientemente. En el esquema de comunicación *maestro-esclavo* la mayoría de los procesos se dedican a

realizar las tareas y algunos procesos (generalmente uno) se dedica a manejar y coordinar estas tareas. El proceso que coordina este mecanismo es llamado "maestro" y los demás procesos son llamados "esclavos". Este esquema también se conocido como *comunicación global*, debido a que el proceso "maestro" concentra toda la información y la reparte entre los "esclavos". Esto permite que la programación de este esquema no se complique, en lo que a manejo de datos se refiere, y permita una fácil comprensión de lo programado. La figura 2.9 presenta un esquema de la manera en que se maneja la información en el esquema de comunicación *maestro-esclavo*. La principal desventaja de este esquema de comunicación es el "cuello de botella" que genera la comunicación centralizada con el proceso maestro. A pesar de ser un esquema sencillo de programación, cuando el número de procesadores es grande, esta desventaja puede ocasionar un mal desempeño del algoritmo.

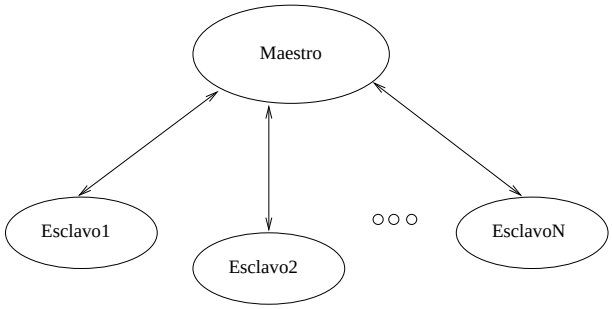

Figura 2.9: Esquema de comunicación maestro-esclavo

2.4.2. Comunicación local

El esquema de comunicación conocido como *Programa Único Múltiples Datos* (**SPMD** por sus siglas en inglés) o *comunicación local* se diferencía del esquema maestro-esclavo en que no existen diferentes programas a ejecutarse, es decir no existe un programa maestro y programas esclavos. En el esquema SPMD sólo se realiza un programa el cual ejecutará cada uno de los procesos a utilizar. En este esquema se supone que todos los procesos tienen la misma jerarquía y la comunicación puede realizarse directamente entre ellos sin necesidad de un proceso manejador. Lo que varía entre los procesos es la sección de datos que manejan, es decir, cada proceso trabajará con datos diferentes. Este esquema generalmente es más eficiente que el esquema *maestro-esclavo*, pero su programación resulta más complicada debido

a que los procesos deben sincronizarse entre ellos para el envío y recepción de información, esto al no existir un proceso maestro que los sincronice.

La figura 2.10 presenta un esquema de la manera en que se maneja la información en el esquema de comunicación *SPMD*. La principal desventaja de este esquema de comunicación se encuentra en la fase de orquestación, ya que se debe realizar un estudio más detallado acerca de las comunicaciones a realizar entre procesos y de la manera en que se llevarán a cabo las comunicaciones. Sin embargo, si se implementa este esquema de manera eficiente presenta un mejor rendimiento que el anterior.

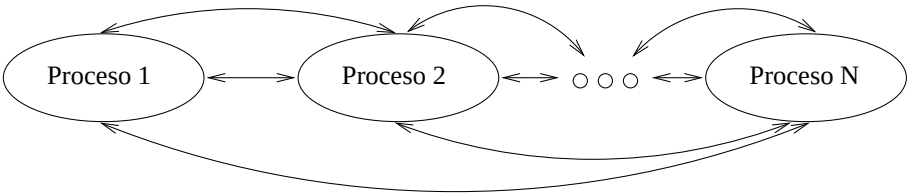

Figura 2.10: Esquema de comunicación local

2.5. Lenguaje C y bibliotecas para programación paralela

El lenguaje C [17] es un lenguaje de programación de propósito general que ofrece como ventajas economía de expresión, control de flujo y estructuras de datos modernos y un rico conjunto de operadores. Además, C no es un lenguaje de "muy alto nivel" ni "grande", y no está especializado en alguna área particular de aplicación. Pero su ausencia de restricciones y su generalidad lo hacen más conveniente y efectivo para la resolución de nuestro problema que otros lenguajes supuestamente más poderosos. El lenguaje C no está ligado a ningún hardware o sistema en particular y es fácil escribir programas que se ejecutarán sin cambios en cualquier máquina que maneje C.

El lenguaje C en su versión ANSI C no está diseñado para aprovechar las ventajas que presentan las arquitecturas paralelas, por lo que es necesario hacer uso de bibliotecas agregadas para la explotación eficiente de todos los recursos de procesamiento disponibles. Existen bibliotecas para programación en C en arquitecturas de memoria compartida como

de memoria distribuida. A continuación se presentarán dos de las bibliotecas más utilizadas para este propósito.

2.5.1. Pthreads

La biblioteca de pthreads [18] es una biblioteca que cumple los estándares POSIX y que nos permite trabajar con distintos hilos de ejecución (threads) al mismo tiempo. Esta biblioteca es la más utilizada para la programación en arquitecturas de memoria compartida utilizando lenguaje C.

La diferencia entre un hilo y un proceso es que los procesos no comparten memoria entre sí, a no ser que se haya declarado explícitamente usando alguno de los mecanismos de IPC (InterProcess Communication) de Unix, mientras que los hilos sí comparten la memoria entre ellos. Además, para crear hilos se usan las funciones de la biblioteca pthread o de cualquier otra que soporte hilos mientras que para crear procesos se utiliza la llamada al sistema **fork()**, que se encuentra en todos los sistemas Unix.

Ya que pthreads es una biblioteca POSIX, se podrán portar los programas hechos con ella a cualquier sistema operativo POSIX que soporte hilos.

Creación y manipulación de hilos

Para crear un hilo se utiliza la función **pthread_create()** de la biblioteca y de la estructura de datos **pthread_t** que identifica cada hilo diferenciándolo de los demás y que contiene todos sus datos.

Cuando se realiza el llamado a la función **pthread_create**, se crea el hilo y comienza su ejecución. Se tienen dos opciones en el momento en que el hilo finaliza su ejecución: esperar a que terminen otros hilos con los cuales realiza un trabajo cooperativo, en el caso de que se desee recoger algún resultado, o simplemente indicar que cuando termine la ejecución de la función el hilo se destruya y libere sus recursos. Estos últimos se conocen como "Hilos desconectados" y se especifican como un atributo cuando son creados.

La figura 2.11 presenta gráficamente la manera en que se son creados y manejados los hilos en el sistema operativo.

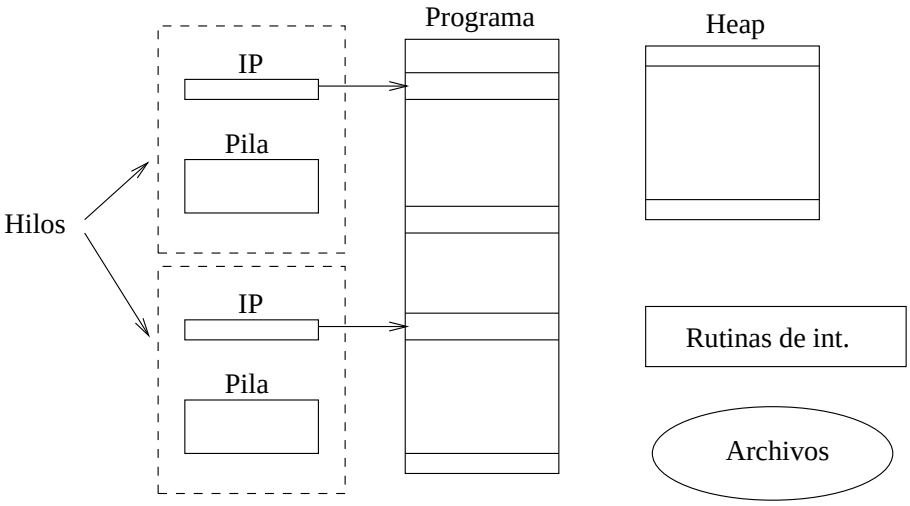

Figura 2.11: Manejo de hilos

Acceso a datos compartidos

Debido a que esta biblioteca se utiliza para arquitecturas de memoria compartida es necesario tomar en cuenta que el acceso a datos compartidos tiene que realizarse de acuerdo a las siguientes reglas:

- Las lecturas múltiples del valor de una variable no causa conflicto
- La escritura a una variable compartida tiene que hacerse con exclusión mutua

El acceso a secciones críticas se puede controlar mediante una variable compartida. La biblioteca pthreads ofrece primitivas de mayor nivel como variables de exclusión mutua (**pthread_mutex**) y variables de condición.

2.5.2. MPI

Para utilizar una arquitectura de memoria distribuida existe una biblioteca de funciones llamada MPI (Message Passing Interface)[19]. Esta biblioteca está disponible para diversos lenguajes de programación incluyendo C, y como su nombre lo indica es una biblioteca que nos ofrece funciones para programación con paso de mensajes.

MPI permite a programas con diferente espacio de memoria sincronizarse y transferir datos del espacio de memoria de un proceso hacia el espacio de memoria de otro enviando y

recibiendo mensajes [20]. En el paso de mensajes básico, los procesos coordinan sus actividades enviando y recibiendo mensajes de una manera explícita. En el nivel más básico MPI provee esta funcionalidad utilizando dos funciones básicas:

- **MPI_Send**: Esta función envía datos del proceso que manda a llamar a otro proceso determinado.
- **MPI_Recv**: Esta función recibe datos de un proceso especificado y los almacena en el espacio de memoria del proceso que llama a la función.

La versión actual de MPI asume que los procesos son generados estáticamente, es decir, el número de procesos a utilizar para la resolución de un problema es determinado al iniciar la ejecución del programa y no se añadirán procesos adicionales durante la ejecución. A cada proceso se le asigna un identificador único en el intervalo de $0, 1, ..., p-1$, donde p es el número total de procesos.

Comunicación bloqueante y no bloqueante

Es necesario saber qué hacer si el arribo de un mensaje se retrasa. Es decir, qué pasa si el proceso 1 ejecuta la función **MPI_Recv** esperando un mensaje del proceso 0, pero éste no ejecutará la función **MPI_Send** para el proceso 1 durante un tiempo significativo.

Muchos sistemas proveen una alternativa, que son las operaciones de recepción **no bloqueantes**. En MPI es llamada **MPI_Irecv**. La *I* proviene de *inmediato*. Esto es, el proceso regresa inmediatamente de la llamada a la función. De esta manera, el proceso que realiza la llamada puede realizar trabajo útil (que no dependa de los datos a recibir) y regresar luego a verificar si ya se recibió el mensaje.

El uso de directivas no bloqueantes puede mejorar dramáticamente el desempeño en programas de paso de mensajes. Si un nodo en un sistema paralelo tiene la capacidad de manejar simultáneamente los cálculos y las comunicaciones el costo de tiempo por comunicaciones se puede reducir. Por lo tanto, esta característica debe ser usada cuando el algoritmo a programar lo permita.

Cabe destacar que al utilizar la biblioteca MPI de funciones para programación con paso de mensajes, no sólo se puede utilizar el programa en una arquitectura de memoria distribuida

sino que de una manera transparente se puede ejecutar en una arquitectura de memoria compartida. Esto a diferencia del uso de una biblioteca de funciones para programación con memoria compartida.

Con esto se puede observar que existe una gran cantidad de opciones la explotación de una arquitectura paralela, así como una gran variedad de estas últimas. Por lo que una parte importante en el desarrollo de una algoritmo paralelo es la correcta selección de la combinación de estas tecnologías para una implementación más eficiente.

Capítulo 3

Obtención de las estructuras secundarias estables

Este capítulo describe la manera en que se solucionó el primer objetivo de este trabajo: encontrar todas las estructuras secundarias estables. Este objetivo se alcanzó modificando el tipo de rastreo de la solución que presenta la liberación óptima de energía. Se presentan los algoritmos para la solución de este problema en las dos complejidades descritas en [1] ($O(n^3)$ y $O(n^4)$). Además se presentan los resultados que se obtuvieron al implementar los algoritmos presentados en este capítulo.

3.1. Diseño de los algoritmos

El primer punto a presentar en este trabajo es, a partir de una estructura primaria dada, la obtención de todas las estructuras secundarias estables que cumplan con las restricciones presentadas en el capítulo 1. Esto es importante debido a que en el espacio de búsqueda del problema se puede presentar más de una estructura estable, y haciendo un análisis de cada una de las estructuras obtenidas (lo cual no está dentro de los alcances de este trabajo) se puede tomar la decisión sobre cuál estructura es la que realmente se formará.

La manera de resolver este problema es analizando el modo de recuperación de la estructura secundaria óptima, además del modo en que se llenan las tablas de rastreo para la programación dinámica, de tal manera que se puedan recuperar todos los casos que produ-

cen cada valor mínimo en la tabla de programación dinámica. Esto se realizó modificando el rastreo que normalmente se utiliza al reconstruir la solución óptima [5], debido a que en este proceso de reconstrucción sólo se obtiene una solución.

3.1.1. Algoritmos de rastreo y reconstrucción para el algoritmo $O(n^3)$

El rastreo que originalmente se propone en [1] para el algoritmo $O(n^3)$ almacena la posición que genera el valor mínimo para la ecuación 1.1, descartando el posible caso de que ocurra una repetición del valor mínimo para diferentes posiciones. Con esta información se puede realizar el rastreo de una solución óptima siguiendo de una manera recursiva los lugares de división de la estructura y finalizando cada caso con el apareamiento de las bases en las posiciones indicadas por el rastro.

El algoritmo 3.1 presenta la modificación al rastreo para almacenar más de una estructura secundaria estable. Esto se realiza en el cálculo de la función **F()** presentada en la ecuación 1.1 del algoritmo propuesto en [1]. Este algoritmo también hace uso de la ecuación 1.2 para realizar el cálculo de la función **C()**. La línea 12 de este algoritmo realiza el rastreo de los pares que aparecen en la estructura primaria. La principal modificación la presentan las líneas 18 y 21, en la línea 18 se almacenan **todas** las posiciones h en la cuales se encontró un mínimo para la subsecuencia $[i, j]$ y en la línea 21 se almacena esta lista de posiciones para la subsecuencia en $PointerF[i][j]$. Con este algoritmo se almacenarán todas las posiciones en las cuales se encuentran los valores mínimos para la subsecuencia $[i, j]$ y pueden ser recuperadas en un proceso posterior de reconstrucción.

Podemos observar que al finalizar el algoritmo 3.1 se obtiene en la esquina superior derecha de la tabla de programación dinámica para **F**, $MatrizF[0][n-1]$, el valor óptimo de liberación de energía del algoritmo con complejidad $O(n^3)$. El algoritmo 3.2 presenta la forma en que se puede reconstruir el total de estructuras secundarias estables y almacenarlas en *cadena*. Este algoritmo recursivo verifica los dos posibles casos de rastreo, cuando (i, j) forma un par (línea 9) y cuando no lo es (línea 13). Cuando se cumple el primer caso el algoritmo marca las posiciones i, j (de la cadena actual) como un par mediante la notación

Algoritmo 3.1 Cálculo de la función F almacenando todas las posibles estructuras secundarias estables para el algoritmo $O(n^3)$

1: **procedure** $FuncionF(i, j)$
2: $i \leftarrow$inicio de la subsecuencia
3: $j \leftarrow$fin de la subsecuencia
4: $MatrizF \leftarrow$tabla de programación dinámica para $F()$ (ecuación 1.1)
5: $MatrizC \leftarrow$tabla de programación dinámica para $C()$ (ecuación 1.2)
6: $PointerF \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizF$
7: $PointerC \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizC$
8: $ListaMinimos \leftarrow$ lista donde se almacenará la posición de los mínimos encontrados

Require: $j - i \geq 4$

9: $PointerC[i][j] \Leftarrow 0$
10: **if** el (i, j) forma un par **then**
11: Obtener valor $C(i, j)$ para $MatrizC[i][j]$
12: $PointerC[i][j] \Leftarrow 1$
13: **else**
14: **for** $h \Leftarrow i$ **to** $j - 1$ **do**
15: $MatrizF[i][h] = FuncionF(i, h)$
16: $MatrizF[h + 1][j] = FuncionF(h + 1, j)$
17: **if** $(MatrizF[i][h] + MatrizF[h + 1][j])$ es un mínimo **then**
18: agregar h a $ListaMinimos$
19: **end if**
20: **end for**
21: Guardar todos los elementos de $ListaMinimos$ en $PointerF[i][j]$
22: **end if**
23: **return** Valor mínimo encontrado
24: **endprocedure**

de paréntesis. La línea 13 se ejecuta en caso contrario por lo que se continúa el rastreo de todas las soluciones iterando sobre cada una de las posiciones almacenadas en $PointerF[i][j]$. La línea 18 del algoritmo se ejecuta cuando se encuentra más de una solución al problema, por lo que se realiza una copia de la reconstrucción obtenida hasta el momento y se continúa un nuevo rastreo en otra cadena con el valor $ActualF$ como nuevo punto de división.

3.1.2. Algoritmos de rastreo y reconstrucción para el algoritmo $O(n^4)$

Para la reconstrucción de una estructura secundaria estable utilizando el algoritmo $O(n^4)$ el rastreo que originalmente se propone en [1] almacena la posición que genera el valor mínimo para cada una de las ecuaciones 1.4, 1.5 y 1.6 descartando el posible caso de que ocurra una repetición del valor mínimo para diferentes posiciones. A partir de esta información se puede realizar el rastreo de una solución óptima siguiendo de manera recursiva los lugares de división de la estructura primaria, comenzando con el rastreo en $G()$ y finalizando cada búsqueda generada con el apareamiento de las bases en las posiciones indicadas por el rastro en $C()$.

Al utilizar el algoritmo de complejidad $O(n^4)$, presentado en las ecuaciones 1.4, 1.5 y 1.6, el proceso para almacenar el rastreo de todas las posibles soluciones óptimas es más complejo. Debido a que existe un mayor número de pasos de verificación de valores mínimos y la repetición de este valor mínimo en diferentes subestructuras es lo que se busca, se tiene que llevar un control de todas estas repeticiones, para en un proceso posterior reconstruirlas. El algoritmo 3.3 presenta el cálculo de la función G, en el cual se almacenarán todas las posibles estructuras secundarias estables derivadas de esta función. Este algoritmo busca dos posibles valores mínimos, el primero se busca en la línea 9, el segundo se busca iterando h (línea 10) y cada uno de los mínimos se agrega en la línea 14. De esta manera se podrán rastrear **todos** los valores mínimos encontrados almacenando sus posiciones en $PointerG[i][j]$.

El algoritmo 3.4 presenta el cálculo y almacenamiento de los valores para la función G_1. Este algoritmo, al igual que el algoritmo 3.3, busca dos posibles valores mínimos (líneas 9 y 15), siendo el segundo valor sujeto de repeticiones las cuales se almacenarán (línea 16). Al

Algoritmo 3.2 Reconstrucción de todas las estructuras secundarias estables para el algoritmo $O(n^3)$

1: **procedure** $Reconstruye(i, j, num)$
2: $i \leftarrow$inicio de la subcadena
3: $j \leftarrow$fin de la subcadena
4: $num \leftarrow$número de cadena a llenar
5: $cadena \leftarrow$arreglo de cadenas a llenar
6: $PointerF \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizF$
7: $PointerC \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizC$
8: $ActualF \leftarrow$elemento actual de $PointerF[i][j]$
Require: $j - i \geq 4$
9: **if** $PointerC[i][j] \neq 0$ **then**
10: $cadena[num][i] \Leftarrow'($
11: $cadena[num][j] \Leftarrow')'$
12: **else**
13: **for** $ActualF$ $\Leftarrow$primer elemento en $PointerF[i][j]$ **to** último elemento en $PointerF[i][j]$ **do**
14: **if** es el primer elemento **then**
15: $Reconstruye(i, ActualF, num)$
16: $Reconstruye(ActualF + 1, j, num)$
17: **else**
18: Realizar una copia de la reconstrucción actual en $cadena[num + 1]$
19: $Reconstruye(i, ActualF, num + 1)$
20: $Reconstruye(ActualF + 1, j, num + 1)$
21: **end if**
22: **end for**
23: **end if**
24: **endprocedure**

Algoritmo 3.3 Cálculo de la función G almacenando todas las posibles estructuras secundarias estables para el algoritmo $O(n^4)$

1: **procedure** $FuncionG(i, j)$
2: $i \leftarrow$inicio de la subsecuencia
3: $j \leftarrow$fin de la subsecuencia
4: $MatrizG \leftarrow$tabla de programación dinámica para $G()$ (ecuación 1.4)
5: $MatrizG_1 \leftarrow$tabla de programación dinámica para $G_1()$ (ecuación 1.5)
6: $PointerG \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizG$
7: $PointerG_1 \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizG_1$
8: $ListaMinimos \leftarrow$ lista donde se almacenará la posición de los mínimos encontrados

Require: $j - i \geq 4$

9: $MatrizG_1[i][j] = FuncionG_1(i, j)$
10: **for** $h \Leftarrow i$ **to** $j - 1$ **do**
11: $MatrizG[i][h] = FuncionG(i, h)$
12: $MatrizG[h + 1][j] = FuncionG(h + 1, j)$
13: **if** $(MatrizG[i][h] + MatrizG[h + 1][j])$ es un mínimo **then**
14: agregar h a $ListaMinimos$
15: **end if**
16: **end for**
17: **if** $MatrizG_1[i][j] <$ mínimo encontrado en $(MatrizG[i][h] + MatrizG[h + 1][j])$ **then**
18: Indicar en $PointerG[i][j]$ que se rastree en $PointerG_1[i][j]$
19: **else**
20: Guardar todos los elementos de $ListaMinimos$ en $PointerG[i][j]$
21: **end if**
22: **return** Valor mínimo encontrado
23: **endprocedure**

final (línea 19) se verificará cuál es el mínimo entre estos dos valores y se almacenará, ya sea la lista de posiciones que nos entregan el mismo valor mínimo (línea 22), o una marca que nos indique el valor mínimo en el primer caso (línea 20).

El algoritmo 3.5 realiza el cálculo de la función C del algoritmo con complejidad $O(n^4)$. En éste se calculará el valor mínimo y se almacenarán **todas** las posiciones que lo producen. Las líneas 6 y 7 presentan dos ciclos anidados, los cuales nos llevan a una complejidad $O(n^4)$. Éstos buscan el posible valor mínimo en ciclos de orden 2 y todas las posiciones que lo generen, almacenando cada una de ellas (línea 18). Se realiza lo mismo para el segundo caso (líneas 17 y 18). El último caso (líneas 21 y 22) al realizar sólo una evaluación no requiere almacenar más que una marca, indicando un caso de valor mínimo.

Para la reconstrucción de todas las estructuras estables se utiliza una versión extendida del algoritmo 3.2 en la cual se varían las verificaciones de $PointerF$ por las de $PointerG$ y $PointerG_1$. Lo anterior se presenta en los algoritmos 3.6, 3.7 y 3.8. El algoritmo 3.6 se encarga de iniciar la reconstrucción de las estructuras secundarias. Su principal función es la redirección recursiva hacia el rastreo en otra posición (líneas 7, 9 y 10). El algoritmo 3.7 es llamado por el algoritmo 3.6 y al igual que éste su principal función es redireccionar el rastreo hacia el caso adecuado. Una diferencia importante es que en las líneas 13 y 21 se verifica la existencia de más de una estructura secundaria estable. El algoritmo 3.8 es el encargado de llenar las cadenas con los pares de la estructura secundaria. Éste verifica cuál de los tres posibles casos es el que se está analizando y realiza el llenado de la estructura (mediante paréntesis) y/o la redirección correspondiente (líneas 7, 10 y 18).

De esta manera podemos obtener todas las estructuras secundarias estables que nos entreguen los algoritmos $O(n^3)$ y $O(n^4)$. La siguiente sección presenta los resultados prácticos de la implementación de los algoritmos presentados.

3.2. Construcción de la estructura secundaria estable

En esta sección se presentan los resultados obtenidos al realizar la programación de los algoritmos presentados en este capítulo. Cabe mencionar que la salida de nuestro programa se realiza mediante la representación por paréntesis, por lo que se llevó a cabo su transformación

Algoritmo 3.4 Cálculo de la función G_1 almacenando todas las posibles estructuras secundarias estables para el algoritmo $O(n^4)$

1: **procedure** $FuncionG_1(i, j)$
2: $i \leftarrow$inicio de la subsecuencia
3: $j \leftarrow$fin de la subsecuencia
4: $MatrizC \leftarrow$tabla de programación dinámica para $C()$ (ecuación 1.6)
5: $MatrizG_1 \leftarrow$tabla de programación dinámica para $G_1()$
6: $PointerG \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizG$
7: $PointerG_1 \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizG_1$
8: $ListaMinimos \leftarrow$ lista donde se almacenará la posición de los mínimos encontrados

Require: $j - i \geq 4$

9: $MatrizC[i][j] = FuncionC(i, j) + e(i, j)$
10: **for** $h \Leftarrow i$ **to** $j - 1$ **do**
11: $MatrizG[i][h] = FuncionG(i, h)$
12: $MatrizG[h + 1][j] = FuncionG(h + 1, j)$
13: $MatrizG_1[i][h] = FuncionG_1(i, h)$
14: $MatrizG_1[h + 1][j] = FuncionG_1(h + 1, j)$
15: **if** $(MatrizG[i][h] + MatrizG_1[h + 1][j])$ o $(MatrizG_1[i][h] + MatrizG[h + 1][j])$ es un mínimo **then**
16: agregar h a $ListaMinimos$ e indicar que caso produjo ese mínimo
17: **end if**
18: **end for**
19: **if** $MatrizC[i][j]$ $<$ mínimo encontrado en $(MatrizG[i][h] + MatrizG_1[h + 1][j], MatrizG_1[i][h] + MatrizG[h + 1][j])$ **then**
20: Indicar en $PointerG_1[i][j]$ que se rastree en $PointerC[i][j]$
21: **else**
22: Guardar todos los elementos de $ListaMinimos$ en $PointerG[i][j]$ y $PointerG_1[i][j]$ e indicar el caso que produjo el mínimo
23: **end if**
24: **return** Valor mínimo encontrado
25: **endprocedure**

Algoritmo 3.5 Cálculo de la función C almacenando todas las posibles estructuras secundarias estables para el algoritmo $O(n^4)$

1: **procedure** $FuncionC(i, j)$
2: $i, j \leftarrow$inicio y fin de la subsecuencia
3: $MatrizC, MatrizG_1 \leftarrow$tabla de programación dinámica para $C()$ y $G_1()$
4: $PointerC \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizC$
5: $ListaMinimos \leftarrow$ lista donde se almacenará la posición de los mínimos encontrados
Require: $j - i \geq 4$
6: **for** $p_1 \Leftarrow i + 1$ **to** $j - 1$ **do**
7: **for** $q_1 \Leftarrow p_1 + 4$ **to** $j - 1$ **do**
8: $MatrizC[p_1][q_1] = FuncionC(p_1, q_1)$
9: **if** $MatrizC[p_1][q_1] + f_2(i, j, p_1, q_1)$ es un mínimo **then**
10: agregar p_1, q_1 a $ListaMinimos$ e indicar que se produjo en este caso
11: **end if**
12: **end for**
13: **end for**
14: **for** $h \Leftarrow i + 1$ **to** $j - 2$ **do**
15: $MatrizG_1[i + 1][h] = FuncionG_1(i + 1, h)$
16: $MatrizG_1[h + 1][j - 1] = FuncionG_1(h + 1, j - 1)$
17: **if** $(MatrizG_1[i + 1][h] + MatrizG_1[h + 1][j - 1] + e(i, j))$ es un mínimo **then**
18: agregar h a $ListaMinimos$ e indicar que se produjo en este caso
19: **end if**
20: **end for**
21: **if** $f_1(i, j)$ es el mínimo encontrado **then**
22: $PointerC[i][j] \leftarrow$caso mínimo
23: **end if**
24: **if** El valor mínimo se encontró en $MatrizC[p_1][q_1]+f_2(i, j, p_1, q_1)$ o $(MatrizG_1[i+1][h]+MatrizG_1[h + 1][j - 1] + e(i, j))$ **then**
25: Guardar todos los elementos de $ListaMinimos$ en $PointerC[i][j]$ e indicar el caso
26: **end if**
27: **return** Valor mínimo encontrado
28: **endprocedure**

Algoritmo 3.6 Reconstrucción de todas las estructuras secundarias estables para el algoritmo $O(n^4)$ utilizando $PointerG$

1: **procedure** $RastreaG(i, j, num)$
2: $i \leftarrow$inicio de la subcadena
3: $j \leftarrow$fin de la subcadena
4: $num \leftarrow$número de cadena a llenar
5: $PointerG \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizG$
Require: $j - i \geq 4$
6: **if** $PointerG[i][j]$ indica redirección a $PointerG_1$ **then**
7: $RastreaG_1(i, j, num)$
8: **else**
9: $RastreaG(i, pointerG[i][j] \rightarrow h, num)$
10: $RastreaG(pointerG[i][j] \rightarrow h + 1, j, num)$
11: **end if**
12: **endprocedure**

a representación pictórica por medio del programa **RNAplot** el cual forma parte de un conjunto de programas de código libre para el tratamiento del problema del RNA denominado Vienna RNA [21].

Las figuras 3.1 y 3.2 presentan el resultado del algoritmo $O(n^3)$ y $O(n^4)$ respectivamente para una longitud de 73 bases. Cabe mencionar que en las dos figuras se utilizó la misma estructura primaria además de utilizar el mismo valor para las bases apareadas, debido a que el algoritmo $O(n^3)$ no permite el análisis de ciclos múltiples. De tal manera, la forma de la estructura secundaria estable sólo dependerá de estos valores. El algoritmo $O(n^4)$ sí permite el análisis de ciclos múltiples por lo que la forma de la estructura secundaria puede ser "alterada" no sólo por el valor de las bases apareadas. Por ejemplo, en la figura 3.2 se da preferencia a la aparición de la subestructura denominada *pares apilados* (capítulo 1.) Es visible la diferencia que presentan las estructuras secundarias estables obtenidas a pesar de que la longitud de la secuencia primaria es corta. Las figuras 3.3 y 3.4 presentan un ejemplo más claro de la diferencia en forma que se puede obtener de una misma estructura primaria

Algoritmo 3.7 Reconstrucción de todas las estructuras secundarias estables para el algoritmo $O(n^4)$ utilizando $PointerG_1$

1: **procedure** $RastreaG_1(i, j, num)$
2: $i, j \leftarrow$inicio y fin de la subcadena
3: $num, offset \leftarrow$número de cadena actual a llenar y cadena disponible a utilizar
4: $PointerG \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizG$
5: $cadena \leftarrow$arreglo de cadenas a llenar
Require: $j - i \geq 4$
6: **if** $PointerG_1[i][j]$ indica redirección a $PointerC$ **then**
7: $RastreaC(i, j, num)$
8: **else**
9: Verificar caso que entrego el mínimo en $PointerG_1[i][j]$
10: **if** Caso $(MatrizG[i][h] + MatrizG_1[h+1][j])$ **then**
11: $RastreaG(i, pointerG[i][j] \rightarrow h, num)$
12: $RastreaG_1(pointerG[i][j] \rightarrow h+1, j, num)$
13: **for** $ActualPoint$ $\Leftarrow$primer elemento en $PointerG[i][j]$ **to** último elemento en $PointerG[i][j]$ **do**
14: Realizar una copia de la reconstrucción actual en $cadena[num + offset]$
15: $RastreaG(i, ActualPoint[i][j] \rightarrow h, num + offset)$
16: $RastreaG_1(ActualPoint[i][j] \rightarrow h+1, j, num + offset)$
17: **end for**
18: **else**
19: $RastreaG_1(i, pointerG[i][j] \rightarrow h, num)$
20: $RastreaG(pointerG[i][j] \rightarrow h+1, j, num)$
21: **for** $ActualPoint$ $\Leftarrow$primer elemento en $PointerG_1[i][j]$ **to** último elemento en $PointerG_1[i][j]$ **do**
22: Realizar una copia de la reconstrucción actual en $cadena[num + offset]$
23: $RastreaG_1(i, ActualPoint[i][j] \rightarrow h, num + offset)$
24: $RastreaG(ActualPoint[i][j] \rightarrow h+1, j, num + offset)$
25: **end for**
26: **end if**
27: **end if**
28: **endprocedure**

Algoritmo 3.8 Reconstrucción de todas las estructuras secundarias estables para el algoritmo $O(n^4)$ utilizando $PointerC$

1: **procedure** $RastreaC(i, j, num)$
2: $i \leftarrow$inicio de la subcadena
3: $j \leftarrow$fin de la subcadena
4: $num \leftarrow$número de cadena a llenar
5: $PointerG \leftarrow$matriz de rastreo para la tabla de programación dinámica $MatrizG$

Require: $j - i \geq 4$

6: Verificar caso que entrego el mínimo en $PointerC[i][j]$
7: **if** Caso $f_1(i, j)$ **then**
8: $estructura[num][i] \Leftarrow' ('$
9: $estructura[num][j] \Leftarrow')'$
10: **else if** Caso $(MatrizG_1[i+1][h] + MatrizG_1[h+1][j-1] + e(i, j))$ **then**
11: **if** (i, j) forma un par **then**
12: $estructura[num][i] \Leftarrow' ('$
13: $estructura[num][j] \Leftarrow')'$
14: **end if**
15: $RastreaG_1(i + 1, pointerC[i][j] \rightarrow h, num)$
16: $RastreaG_1(pointerC[i][j] \rightarrow h + 1, j - 1, num)$
17: **else**
18: **if** (i, j) forma un par **then**
19: $estructura[num][i] \Leftarrow' ('$
20: $estructura[num][j] \Leftarrow')'$
21: **end if**
22: $RastreaC(pointerC[i][j] \rightarrow p_1, pointerC[i][j] \rightarrow q_1, num)$
23: **end if**
24: **endprocedure**

dada. La primera presenta el resultado obtenido por el algoritmo $O(n^3)$ y la segunda el obtenido por el algoritmo $O(n^4)$. Se utilizó la misma política antes mencionada, acentuar la aparición de pares apilados, para el algoritmo $O(n^4)$. Se puede observar que a pesar del incremento en la longitud de la secuencia, el algoritmo $O(n^4)$ es capaz de rastrear la secuencia que cumpla con la política establecida.

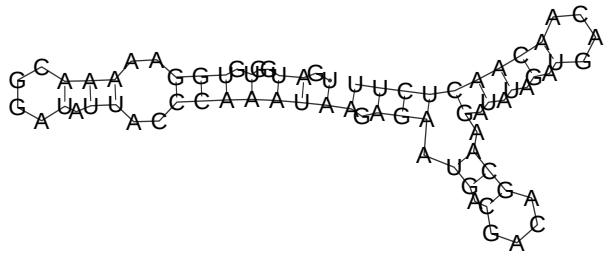

Figura 3.1: Estructura secundaria obtenida con el algoritmo $O(n^3)$ (73 bases)

3.2.1. Obtención de todas las estructuras secundarias estables

Como se explicó previamente, *generalmente* sólo una estructura será estable, es decir, puede aparecer más de una estructura secundaria estable. Por lo anterior, se decidió satisfacer el objetivo de encontrar todas las estructuras secundarias estables que se presentan dada una estructura primaria. Esto se logró modificando el tipo de rastreo de la solución óptima en la programación dinámica como se presentó en la sección anterior.

Las figuras 3.5 y 3.6 presentan el resultado de rastrear todas las estructuras secundarias estables obtenidas y dos estructuras secundarias estables para una misma estructura primaria dada. La salida de nuestro algoritmo es la siguiente:

```
Energia minima: -37
```

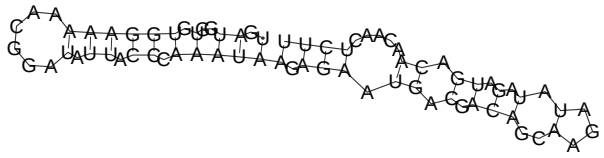

Figura 3.2: Estructura secundaria obtenida con el algoritmo $O(n^4)$ (73 bases)

```
Cadena: (.((..(.(((..(((....).))..)))))))(.(((.((.(....))).(.(.(.(.(....)))))))))
Cadena: (.((..(.(((..(((....).))..)))))))(.(((((.(((....)..).))).(.((....).))))))
```

Por lo que se utilizó el programa **RNAplot** para obtener su representación pictórica. Se puede observar que la "parte izquierda" de las dos estructuras comparten la misma forma, pero la "parte derecha" varía en la posición de las bases apareadas. Este resultado indica, que en el espacio de búsqueda, estas dos estructuras comparten la misma liberación óptima de energía.

El cuadro 3.1 presenta un análisis del número máximo de estructuras secundarias estables (diferentes) encontradas para un cierto número n de bases en la estructura primaria. Se puede observar que en el algoritmo $O(n^4)$ es más dificil encontrar repetición de estructuras secundarias estables debido a que al analizar ciclos de mayor orden agrega condiciones (por ejemplo, longitud de los ciclos internos) que dificultan la repetición de las mísmas.

En este capítulo se presentó una posible manera de reconstruir todas las estructuras secundarias estables utilizando programación dinámica, se presentaron los algoritmos necesarios para este fin así como los resultados que entrega la implementación de los mismos. Se pudo observar que efectivamente puede aparecer más de una solución óptima al problema del RNA. El siguiente capítulo está dedicado al diseño e implementación de la paralelización

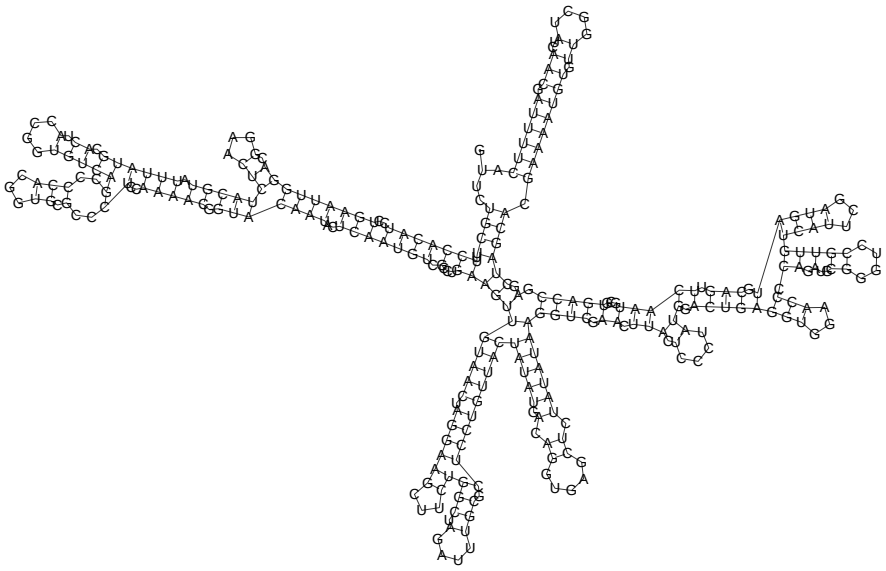

Figura 3.3: Estructura secundaria obtenida con el algoritmo $O(n^3)$ (300 bases)

del algoritmo con complejidad $O(n^4)$, lo cual es el segundo y último objetivo de esta tesis.

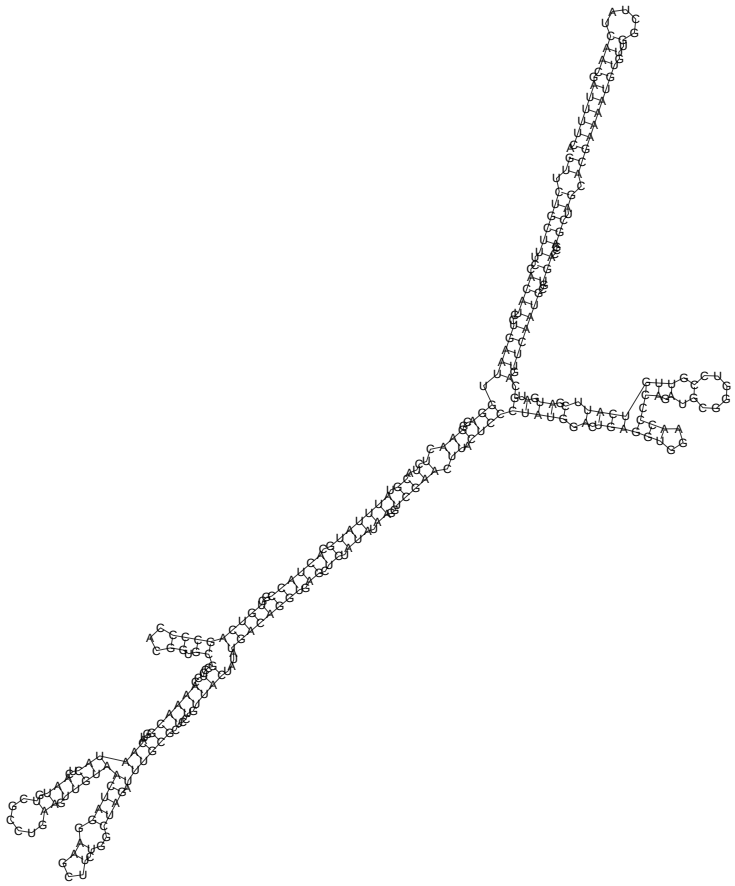

Figura 3.4: Estructura secundaria obtenida con el algoritmo $O(n^4)$ (300 bases)

Tamaño n	$O(n^3)$	$O(n^4)$
50	1	1
100	2	1
500	2	1
1000	2	1
1500	2	2
2000	3	2

Cuadro 3.1: Número máximo de estructuras secundarias estables encontradas para los algoritmos $O(n^3)$ y $O(n^4)$

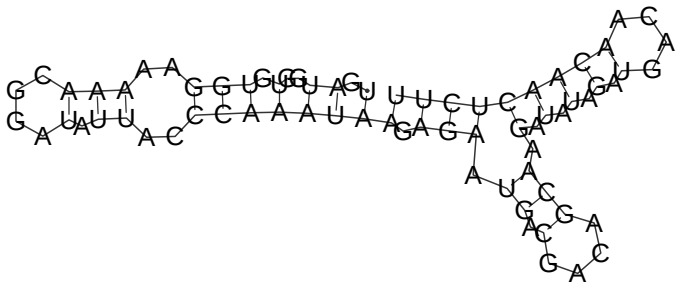

Figura 3.5: Primera estructura secundaria estable obtenida

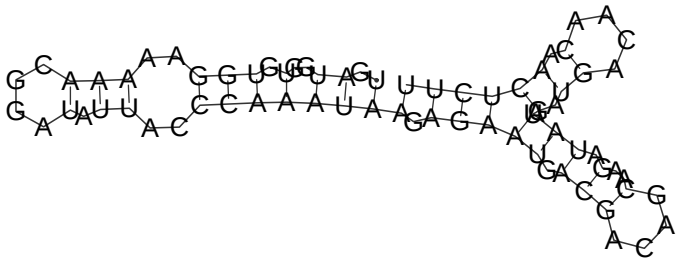

Figura 3.6: Segunda estructura secundaria estable obtenida

Capítulo 4

Diseño del algoritmo paralelo para el problema del RNA

En este capítulo se presenta el diseño e implementación del algoritmo paralelo para resolver el problema del RNA. También se incluye un análisis de las dependencias que presentan las estructuras de datos utilizadas para la solución del problema del RNA. Esto con la finalidad de, posteriormente, realizar un correcto particionamiento de datos para la implementación paralela.

Para llevar a cabo un correcto análisis en la dependencia de datos se estudió a detalle el comportamiento de las ecuaciones que caracterizan el algoritmo con complejidad $O(n^4)$ y se estudiaron las requisiciones en datos que presentan cada una de éstas y la manera en que se podían agrupar estas dependencias. En la orquestación se decidió utilizar un lenguaje que permita, bajo arquitecturas de memoria distribuida, una eficiente utilización de los recursos. Así mismo, se desarrollaron las ecuaciones que modelan el comportamiento para esta arquitectura. Debido a la utilización de una arquitectura homogénea, donde todas las computadoras a utilizar tienen las mismas capacidades, el mapeo se realiza de manera directa y el número de procesos se divide equitativamente entre el número de procesadores.

Por último, se propone un algoritmo de particionamiento eficiente para el algoritmo con complejidad $O(n^4)$. Se presentan los algoritmos que logran este objetivo, así como una descripción de los mismos.

4.1. Trabajos previos

Existen diversas implementaciones para encontrar la estructura secundaria óptima, formada a partir de una secuencia de nucleótidos llamada estructura primaria, de manera paralela.

En 1990, Michael Ess [22] presenta una implementación paralela sobre una computadora **Cray-2** mostrando las ventajas de una búsqueda paralela masiva de la solución obteniendo aceleraciones superlineales. Esto ocurre debido a que el programa mantiene valores de estructuras que ya fueron revisadas y se convierten en sub-árboles que no volverán a ser revisados, logrando de esta manera una poda en la búsqueda de la solución óptima. En una computadora con sólo un procesador, estos valores son encontrados de una manera determinística, pero en una máquina multi-procesador estos "valores de poda" pueden ser encontrados antes por un procesador y ser utilizados por los demás procesadores eliminando la búsqueda en sub-árboles previamente calculados.

Se pone a disposición pública el paquete Vienna RNA [21] el cual consiste de una biblioteca en código C (con funciones para cómputo paralelo) y algunos programas para predecir y comparar estructuras secundarias de RNA (en implementaciones para cómputo secuencial), entre otros algoritmos de programación dinámica. Este paquete prueba el algoritmo propuesto por Zuker y Stiegler [23], el cual **obtiene sólo una estructura óptima**. La salida del programa RNAfold, el cual es el programa que se usa para el problema de la estructura secundaria óptima, consiste de una representación de paréntesis de la estructura secundaria estable, así como de un PostScript en el que se presenta la representación gráfica de la estructura. La representación de paréntesis asigna a cada base de la estructura primaria, ya sea un punto o un paréntesis (abierto o cerrado). Los puntos indican que las bases que se encuentran en esa posición no están apareadas. Los paréntesis indican que las bases a las cuales les sea asignado un paréntesis abierto se aparearán con las bases que tengan el paréntesis cerrado correspondiente.

Estas dos últimas implementaciones **no optimizan las comunicaciones entre los procesadores**, aunque en el paquete Vienna RNA su aritmética para el balance de carga es satisfactorio.

En 2001 Shapiro [24] publica un algoritmo genético masivamente paralelo, en el cual la energía mínima (lograda en una generación) se usa como criterio para mejorar la población de estructuras de todos los procesadores. Los procesadores se ordenan en una configuración rectangular y cada procesador selecciona dos estructuras de RNA con mejor energía. Esta selección se hace desde un conjunto de nueve estructuras, originarias de ocho procesadores vecinos y del mismo procesador. El operador de mutación utilizado es de recocido, el cual depende de la longitud de la secuencia. El operador de cruza selecciona dos padres de un banco de estructuras, se generan dos hijos y se selecciona el mejor, es decir, el que presente una menor energía liberada. Si el promedio ponderado de energía es menor a un valor umbral previamente seleccionado el algoritmo logra su convergencia y termina. Se seleccionará como solución la estructura que aparezca en mayor número de procesadores.

Las pruebas de este algoritmo se llevaron a cabo en una computadora de cálculo masivo MasPar MP-2 con 16384 procesadores físicos, en la SGI ORIGIN 2000 con 64 procesadores y en la CRAY T3E con 512 procesadores. Cabe mencionar que la secuencia más larga probada en este algoritmo es de 742 nucleótidos de longitud.

En el 2002 Mireya Tovar Vidal [3] presenta la tesis "Estrategias de particionamiento paralelo para el problema del RNA", en la cual se estudian a detalle tres estrategias de particionamiento para el algoritmo de complejidad $O(n^3)$ optimizando el tiempo de comunicación entre los procesadores. Las estrategias de particionamiento empleadas fueron:

- Por diagonales: En esta técnica a cada procesador le corresponde una parte de las diagonales en que se divide la tabla de programación dinámica.

- Por bloques de longitud fija (homogéneos): Para evitar tantos pasos de sincronización se puede agrupar el procesamiento por bloques. Al calcular la posición (i, j) de la tabla de programación dinámica, se necesita el i-ésimo y la j-ésima columna, lo cual construye un triángulo equilátero. Esto origina esta otra forma de particionar la matriz.

- Particionamiento por bloques no homogéneos: En el particionamiento por bloques homogéneos se busca que cada bloque sea del mismo tamaño. Sin embargo, como el proceso de llenado de la tabla avanza, el tiempo de cómputo invertido para calcular cada bloque varía, es decir: "Al inicio el cálculo de bloques es rápido para después

hacerse más lento a la mitad del proceso y vuelve a acelerarse al final del proceso." Por lo que se propuso realizar el cálculo mediante bloques de longitud variable.

Sin embargo, no se estudia a detalle el algoritmo de complejidad $O(n^4)$, y al igual que la mayoría de los paquetes en la actualidad obtiene sólo una estructura óptima.

En 2002 [25] se publica el artículo "Optimal tiling for the RNA base pairing problem", el cual propone un nuevo esquema de particionamiento para el problema del RNA (presentando éste una subclase particular de recurrencia con espacio de iteración triangular y dependencias no-uniformes). El esquema que se propone, pretende reducir el tiempo de comunicación dejando (en la mayor cantidad posible) las dependencias de datos requeridas por un procesador en el mismo procesador. A diferencia del trabajo de Mireya Tovar esta propuesta presenta un tipo de particionamiento en bloques rectangulares (siendo bloques homogéneos un caso particular de éste) que pretende mediante aritmética modular alojar futuras dependencias en el procesador que las requerirá.

Cabe mencionar que este trabajo sustenta sus resultados sólo de una manera teórica. Sin embargo, su propuesta merece ser tomada en cuenta, pues la verificación práctica de este esquema puede arrojar resultados que muestren beneficios reales al algoritmo original. El algoritmo utilizado en esta propuesta es el de complejidad $O(n^3)$.

Existen otros trabajos [2], [26], [27] especializados en formaciones específicas llamadas seudo-nudos, problema el cual representa un área de investigación importante debido a que estos seudo-nudos han mostrado ser funcionalmente importantes entre los diferentes tipos de RNA. Los seudo-nudos juegan un papel regulatorio, catalítico o estructural. Los métodos biofísicos actuales para identificar la presencia de estos seudo-nudos son extremadamente costosos tanto en tiempo como en dinero, por lo que las aproximaciones bioinformáticas para predecir de una manera certera la aparición de estas estructuras son muy importantes.

4.2. Análisis de dependencia de datos

Para poder realizar un correcto particionamiento de un programa es necesario estudiar la dependencia que presentan sus datos. El algoritmo que se utilizó para resolver el problema de RNA utiliza como herramienta la programación dinámica por lo que es necesario estudiar

las dependencias presentes en las tablas de programación dinámica utilizadas.

Por facilidad de lectura a continuación se repiten las ecuaciones 1.4, 1.5 y 1.6 presentadas en el capítulo 1.

$$G(i,j) = min \begin{cases} G_1(i,j) \\ \min\limits_{i \leq h < j} [G(i,h) + G(h+1,j)] \end{cases} \tag{1.4}$$

$$G_1(i,j) = min \begin{cases} C(i,j) \\ \min\limits_{i \leq h < j} [G_1(i,h) + G(h+1,j), G(i,h) + G_1(h+1,j)] \end{cases} \tag{1.5}$$

$$C(i,j) = \begin{cases} min \begin{cases} f_1(i,j) \\ \min\limits_{i < p_1 < q_1 < j} [f_2(i,j,p_1,q_1) + C(p_1,q_1)] \\ \min\limits_{i+1 \leq h < j-1} [G_1(i+1,h) + G_1(h+1,j-1)] + e(i,j) \end{cases} \\ \qquad\qquad \text{si } (i,j) \text{ forma un par,} \\ \qquad\qquad \text{de otra forma,} \\ indefinido \end{cases} \tag{1.6}$$

Se puede observar en el algoritmo de Sankoff [1] que es necesaria la utilización de tres tablas de programación dinámica. Las funciones $G(i,j)$, $G_1(i,j)$ y $C(i,j)$ (ecuaciones 1.4, 1.5 y 1.6) requieren de una tabla dónde almacenar los datos que se obtienen para cada par i,j evaluado. La figura 4.1 muestra las dependencias que presentan los datos en la tabla de programación dinámica para la función $G(i,j)$. Cada una de las tablas a utilizar presenta una forma triangular superior. Para que la paralelización sea eficiente se debe buscar que los valores requeridos para calcular un elemento hayan sido previamente calculados.

Cada par i,j a analizar en la función $G(i,j)$ nos genera dependencias hacia las tablas de programación dinámica para las funciones $G_1(i,j)$ y $C(i,j)$, en las figuras 4.2 y 4.3 podemos observar gráficamente estas dependencias.

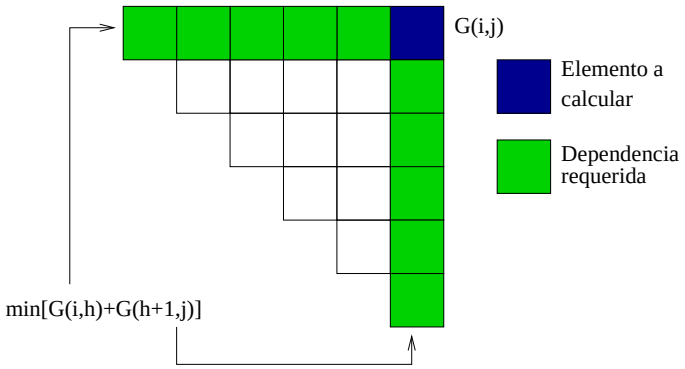

Figura 4.1: Dependencias en la tabla de programación dinámica para la función $G(i, j)$

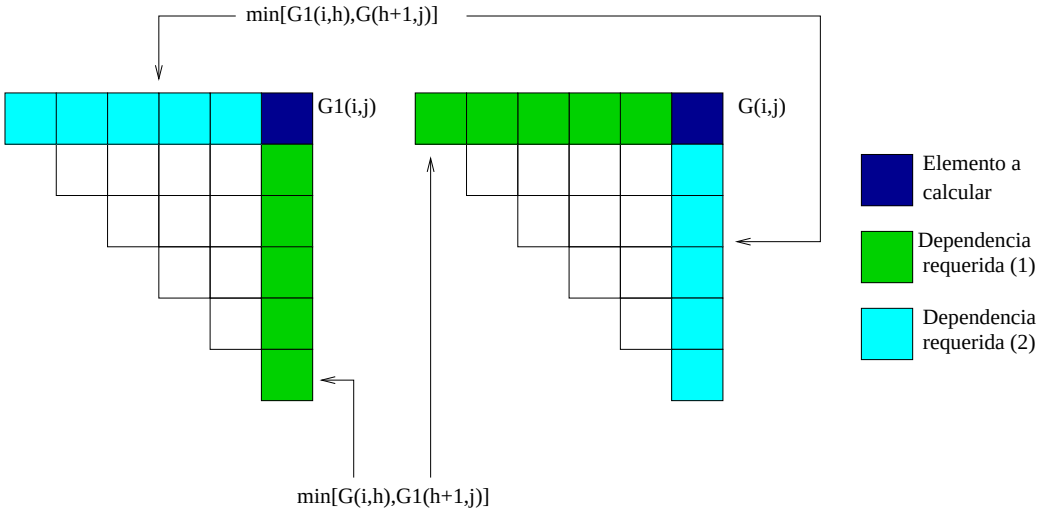

Figura 4.2: Dependencias en la tabla de programación dinámica para la función $G_1(i, j)$

4.2.1. Dependencias entre diagonales

Se puede ver que si se realiza un particionamiento por diagonales, desde la principal hasta la ubicada en la esquina superior derecha, se tendrán ya calculados los valores que se requieren en alguna diagonal. Por otra parte cada uno de los elementos de las diagonales en las tres tablas es independiente entre sí y cada diagonal d_i depende solamente de las diagonales $d_0, d_1, ..., d_{i-1}$. El proceso total utilizará n diagonales siendo n el tamaño de la estructura primaria, es decir, la longitud de la secuencia de bases a analizar. La figura 4.4 muestra las dependencias entre los elementos en una diagonal.

De esta manera se puede diseñar un algoritmo que explote la paralelización del algoritmo de complejidad $O(n^4)$ por medio de la paralelización del cálculo de sus diagonales. Esto debe

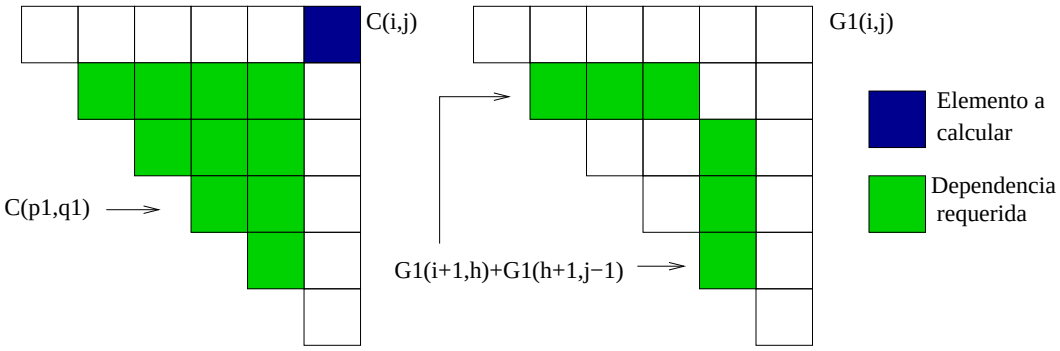

Figura 4.3: Dependencias en la tabla de programación dinámica para la función $C(i, j)$

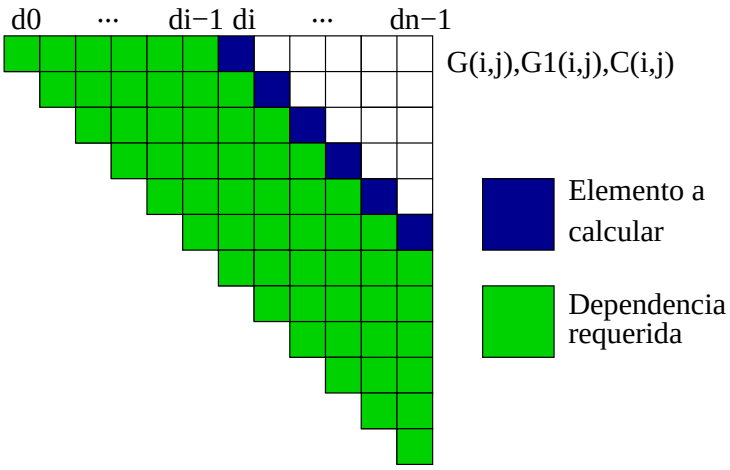

Figura 4.4: Dependencias en las tablas de programación dinámica al realizar el cálculo de una diagonal

realizarse en cada una de las tres tablas de programación dinámica a utilizar en el proceso. Pero como se puede observar, el particionamiento a realizarse aún es de "grano fino", es decir, los elementos a utilizar son atómicos. Es necesario estudiar el comportamiento de las dependencias para lograr la agrupación de un número de elementos mayor y que respete la característica que se busca en una correcta paralelización: *que los valores requeridos para calcular un elemento hayan sido previamente calculados o que se calculen en el mismo proceso en que se encuentra el elemento a calcular.*

4.2.2. Dependencias entre bloques

Se puede agrupar una mayor cantidad de datos, los cuales sólo tengan dependencias entre ellos mismos y elementos previamente calculados. A esta agrupación de datos se le denomina *bloque*. La figura 4.5 muestra las dependencias presentadas en el cálculo de un bloque.

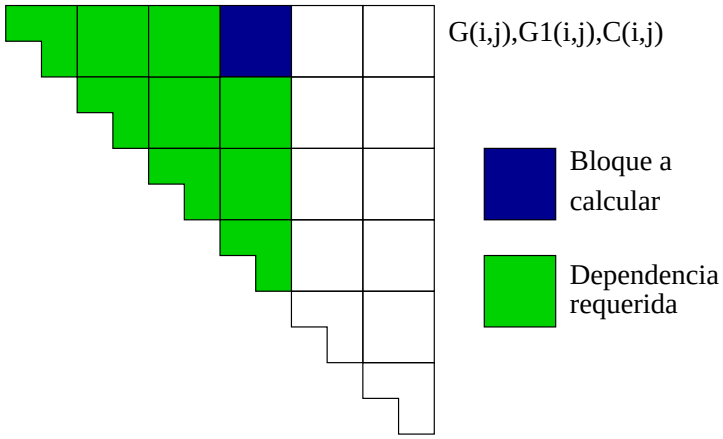

Figura 4.5: Dependencias en las tablas de programación dinámica al realizar el cálculo de una diagonal de bloques

En esta nueva agrupación para el cálculo de los datos podemos observar que cada uno de los bloques i de las diagonales en las tres tablas es independiente entre sí, es decir, sólo dependerá de los elementos que serán calculados en el mismo bloque y que para el cálculo de la diagonal de bloques B_i sólo requerimos de las $B_0, B_1, ..., B_{i-1}$ diagonales anteriores. De esta manera se puede asegurar que con la división por bloques, al igual que la división por diagonales de elementos se logra la característica buscada para la correcta paralelización. La figura 4.6 muestra las dependencias entre las diagonales de bloques.

4.3. Estrategias de particionamiento implementadas

El análisis de las dependencias en los datos de las tablas de programación dinámica nos permite tomar una mejor decisión acerca de las estrategias de particionamiento posibles en la paralelización del algoritmo. De manera natural debido a las dependencias observadas se utilizaron particionamientos por diagonales y particionamientos por bloques.

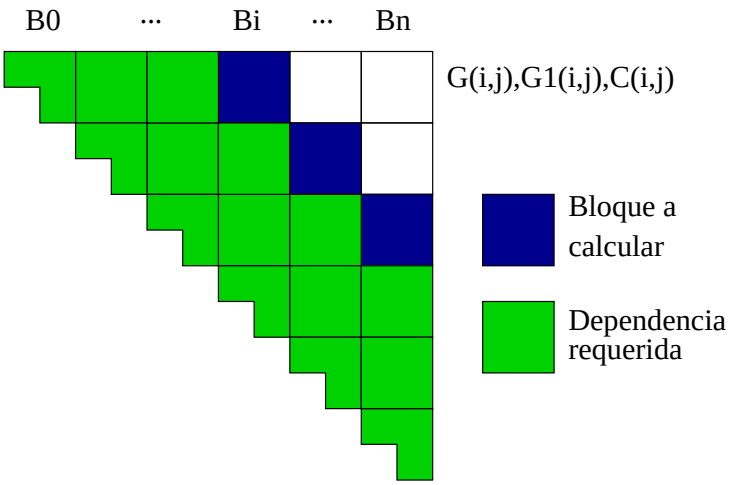

Figura 4.6: Dependencias en las tablas de programación dinámica al realizar el cálculo de una diagonal de bloques

4.3.1. Particionamiento por diagonales

Debido a que los elementos de las diagonales en las tres tablas de programación dinámica son independientes entre sí, y cada diagonal d_i depende solamente de las d_{i-1} diagonales anteriores, se realizó una paralelización que explote esa característica. La idea es distribuir el procesamiento de los elementos que caen sobre la misma diagonal entre varios procesadores. La figura 4.7 muestra un ejemplo de la partición a realizar con tres procesadores ($P0$, $P1$ y $P3$). El algoritmo 4.1 presenta la manera en que se implementó este particionamiento. La línea 7 nos muestra que el número de iteraciones a realizar, las cuales incluyen el cálculo de los valores (línea 11) y el proceso de sincronización (línea 13), crece de manera proporcional a $N-4$, lo cual representa un número de procesos de sincronización grande. Cabe mencionar que la línea 13 del algoritmo 4.1 puede ser reemplazada por un proceso de sincronización por medio de barreras en caso que se utilice una implementación con hilos.

Se puede observar que a cada uno de los procesadores utilizados le corresponde una parte proporcional de la matriz triangular superior. Esto nos entrega un buen balanceo de carga entre los procesadores pero nos requiere $n-4$ procesos de sincronización, siendo $n-4$ el número de diagonales a calcular. Esto es, al terminar de calcular su porción de la diagonal, cada proceso deberá esperar los resultados de los demás procesos. Un modelo matemático que aproxima el tiempo de ejecución para este algoritmo es el siguiente:

Algoritmo 4.1 Particionamiento por diagonales para las tablas $G(i,j)$,$G_1(i,j)$ y $C(i,j)$

1: **procedure** *Calcula diagonales*
2: *inicio* ←inicio de mi porción de la diagonal
3: fin ←fin de mi porción de la diagonal
4: $NumDiag$ ←diagonal a calcular
5: N ←longitud de la secuencia
6: p ←mi identificador de proceso
7: **for** $NumDiag \Leftarrow 4$ **to** N **do**
8: Obtener valor para *inicio* y fin según p
9: **for** $i \Leftarrow inicio$ **to** fin **do**
10: $j \Leftarrow NumDiag + i$
11: Calcula valor para $G(i,j)$, $G_1(i,j)$ y $C(i,j)$
12: **end for**
13: Proceso de sincronización (envío de mensajes entre los procesadores)
14: **end for**
15: **endprocedure**

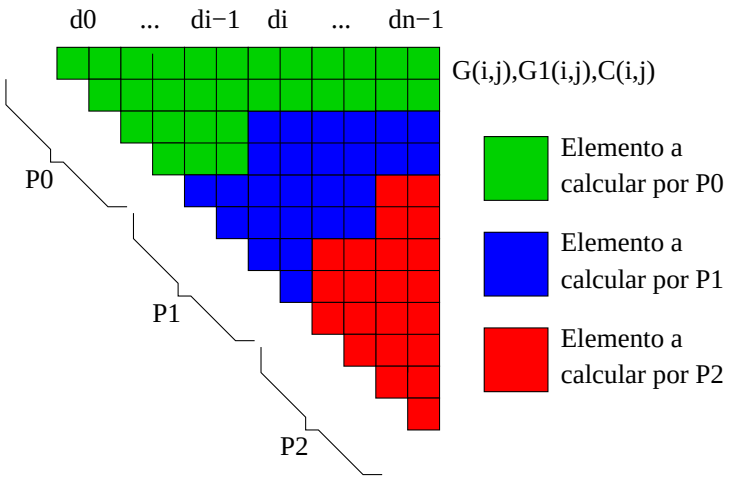

Figura 4.7: Particionamiento por diagonales para las tablas $G(i,j)$,$G_1(i,j)$ y $C(i,j)$

$$T_p \approx \frac{T_c \cdot n^4}{p} + (n-4) \cdot T_s \tag{4.1}$$

donde:

T_c es una constante que indica el tiempo utilizado en el cálculo

p es el número de procesadores a utilizar

n es el número de bases en la estructura primaria (tamaño del problema)

T_s es el tiempo requerido para la comunicación y sincronización

Este modelo es válido para una arquitectura de memoria compartida y muestra que el tiempo requerido para el cálculo de los valores es dividido entre el número de procesos a utilizar. A éste se le agrega el número de pasos de comunicación y sincronización multiplicado por el tiempo requerido para este fin. En este esquema de particionamiento, el número de pasos crece a razón de $n-4$. *El tiempo de sincronización generalmente es varios órdenes de magnitud mayor que el tiempo de cómputo.* Debido a lo anterior se buscó un particionamiento que permita una utilización más eficiente de los recursos. Este particionamiento es el denominado *particionamiento por bloques.*

Cada diagonal requiere un tiempo de cómputo diferente, el cual está determinado por el número de llamadas a 1.4, 1.5 y 1.6. En [3] se presenta un estudio del tiempo que requiere cada una de las diferentes diagonales en el proceso. La ecuación 4.2 presenta la función $h(d)$, la cual muestra el tiempo que se requiere para el cálculo de cada una de estas diagonales.

$$h(d) = \frac{-d^3}{2} + \left(\frac{n}{2} - 5\right) d^2 + \left(\frac{11n}{2} + \frac{3}{2}\right) d - 4n - 4 \tag{4.2}$$

donde:

d es el número de diagonal a calcular

n es el tamaño del problema

La figura 4.8 muestra el comportamiento en tiempo del cálculo de cada una de las diagonales del proceso. En esta gráfica se puede ver que el mayor tiempo lo requieren las diagonales localizadas a $\frac{2}{3}$ de la tabla.

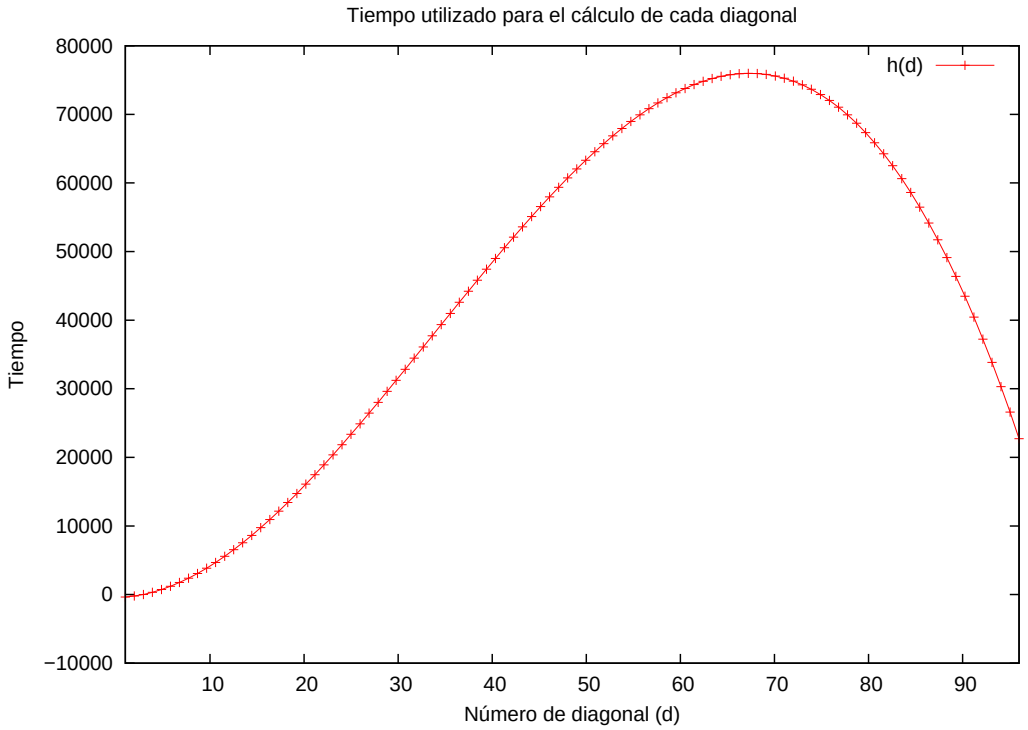

Figura 4.8: Tiempo requerido por cada diagonal (n=100)

4.3.2. Particionamiento por bloques

Lo que se busca en un particionamiento eficiente no es sólo tener un buen balance de carga entre los procesadores sino requerir del menor número de procesos de sincronización. Para la resolución del problema del RNA mediante programación dinámica se puede agrupar más de un elemento en lo que se denomina *bloque*. De esta manera podemos reducir el número

de sincronizaciones de n a $\frac{n}{b}$, donde b es el tamaño del bloque. Es decir, no se requerirá un proceso de sincronización por cada diagonal de elementos a calcular sino por cada diagonal de bloques. La figura 4.9 muestra un ejemplo de la partición por bloques a realizar con tres procesadores ($P0$, $P1$ y $P3$).

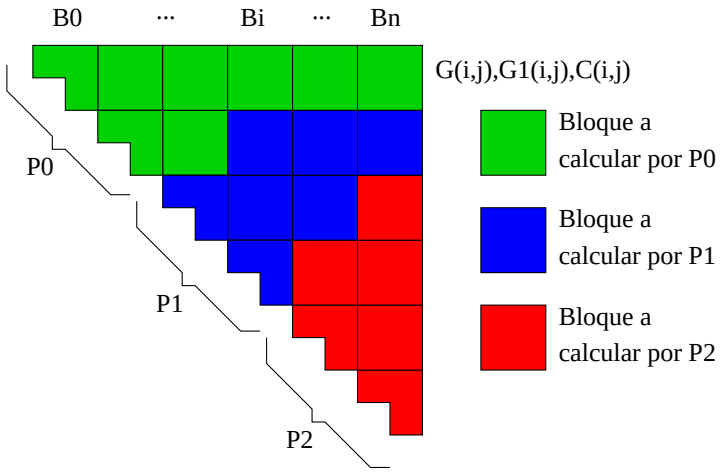

Figura 4.9: Particionamiento por bloques para las tablas $G(i,j)$,$G_1(i,j)$ y $C(i,j)$

Modificación al particionamiento por bloques

La figura 4.9 presenta un ejemplo del llamado "Particionamiento por bloques homogéneos", en el cual los bloques que se presentan son cuadrados e idénticos entre sí. Este particionamiento presenta el problema de división inexacta, es decir, *la suma de la longitud de los bloques no cubre todo el espacio requerido*. Esto se presentará siempre que la longitud de la estructura primaria no sea un múltiplo de la longitud de los bloques a utilizar.

Para solucionar este problema se propuso una modificación al particionamiento en la cual se realiza una distribución del espacio en bloques rectangulares. De esta manera no existen residuos en las diagonales de bloques en el proceso. Además, este particionamiento asegura que la diferencia máxima entre las dimensiones de rectángulos contiguos sea a lo más uno. Esto último con la finalidad de mantener el buen balance de carga entre los procesadores. La figura 4.10 muestra un ejemplo del particionamiento por bloques propuesto.

El algoritmo 4.2 presenta la manera de calcular las coordenadas de cada uno de los bloques rectangulares a utilizar. En la línea 7 de este algoritmo se obtiene el valor real de la

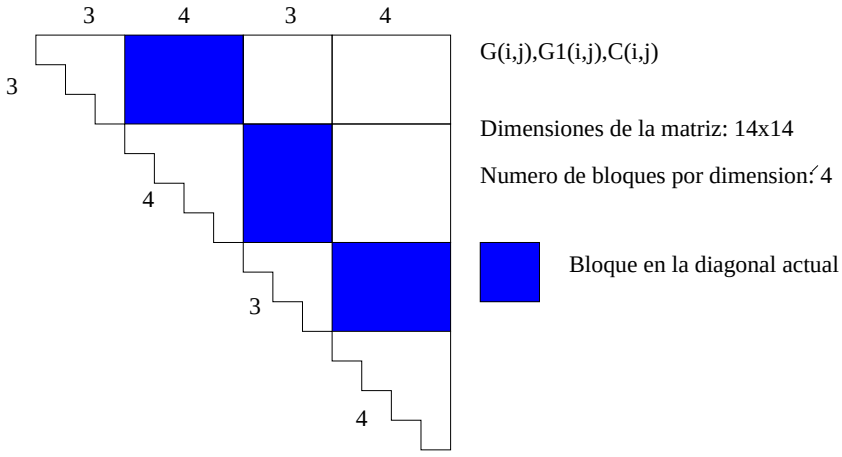

Figura 4.10: Modificación propuesta al particionamiento por bloques

longitud que debiese tener cada una de las secciones. Las líneas 9 y 10 calculan el inicio y fin (con valores enteros) de cada uno de los bloques a utilizar, de tal manera que el residuo en el espacio de búsqueda se distribuya de manera uniforme a lo largo de la diagonal de bloques.

Algoritmo 4.2 Cálculo de la posición horizontal y vertical para cada bloque

1: **procedure** *Calcula dimensiones de bloques*
2: $LongDiagonal \leftarrow$longitud de la diagonal mayor (TamañoTotal - 4)
3: $TotalBloques \leftarrow$número de bloques a utilizar
4: $secc \leftarrow$longitud teórica de cada bloque
5: $posx \leftarrow$ Arreglo de la posición horizontal para cada bloque
6: $posy \leftarrow$ Arreglo de la posición vertical para cada bloque
7: $secc \Leftarrow \frac{LongDiagonal}{TotalBloques}$
8: **for** $i \Leftarrow 0$ **to** $TotalBloques$ **do**
9: $\quad posy[i] = \lfloor i \cdot secc \rfloor$
10: $\quad posx[i] = \lfloor (i+1) \cdot secc \rfloor - 1$
11: **end for**
12: **endprocedure**

Al ejecutar este algoritmo podremos conocer cuales serán las coordenadas de cualquier bloque en cualquier diagonal. Estos valores son los siguientes:

- Coordenada superior izquierda: $(posy[\#Bloque], posy[\#Diagonal])$

- Coordenada superior derecha: $(posy[\#Bloque], posx[\#Diagonal])$
- Coordenada inferior izquierda: $(posx[\#Bloque], posy[\#Diagonal])$
- Coordenada inferior derecha: $(posx[\#Bloque], posx[\#Diagonal])$

Cabe mencionar que las coordenadas se entregan de la misma forma en la que se organiza una matriz en lenguaje C, es decir, (y, x).

El algoritmo 4.3 presenta la implementación del particionamiento por bloques. La línea 10 nos muestra que el número de iteraciones a realizar, las cuales incluyen el cálculo de los valores de cada bloque (línea 14) y el proceso de sincronización (línea 16), crece de manera proporcional al número de bloques a utilizar B, lo cual representa un número de procesos de sincronización menor al utilizado en el algoritmo 4.1. Al igual que en el particionamiento por diagonales la línea 16 del algoritmo 4.3 puede ser reemplazada por un proceso de sincronización por medio de barreras en caso de que se utilice una implementación con hilos.

El modelo de rendimiento para este algoritmo es el siguiente:

$$T_p \approx \frac{T_c \cdot n^4}{p} + B \cdot T_s \tag{4.3}$$

donde:
T_c es una constante que indica el tiempo utilizado en el cálculo
p es el número de procesadores a utilizar
n es el número de bases en la estructura primaria
B es el número de bloques a utilizar
T_s es el tiempo requerido para la comunicación y sincronización

Este modelo es válido para una arquitectura de memoria compartida y muestra que el tiempo de requerido para el cálculo de los valores es dividido entre el número de procesos a utilizar. A éste se le agrega el número de pasos de comunicación y sincronización multiplicado por el tiempo requerido para este fin. En este esquema de particionamiento el número de pasos crece a razón del número de bloques B, lo que disminuye el número de procesos de sincronización siempre que la longitud del bloques sea mayor a 1.

Algoritmo 4.3 Particionamiento por bloques para las tablas $G(i,j)$,$G_1(i,j)$ y $C(i,j)$

1: **procedure** *Calcula bloquess*
2: *inicio* ←inicio de mi porción de la diagonal de bloques
3: fin ←fin de mi porción de la diagonal de bloques
4: $NumDiag$ ←diagonal de bloques a calcular
5: N ←longitud de la secuencia
6: p ←mi identificador de proceso
7: $TotalBloques$ ←número de bloques a utilizar
8: $TotalDiagonales$ ←número de diagonales de bloques a calcular
9: $TotalDiagonales \Leftarrow TotalBloques$
10: **for** $NumDiag \Leftarrow 0$ **to** $TotalDiagonales$ **do**
11: Obtener valor para *inicio* y fin según p
12: **for** *bloq* $\Leftarrow$ *inicio* **to** fin **do**
13: Calcular las coordenadas del bloque *bloq*
14: Calcula valor para cada $G(i,j)$, $G_1(i,j)$ y $C(i,j)$ del bloque
15: **end for**
16: Proceso de sincronización (envío de mensajes entre los procesadores)
17: **end for**
18: **endprocedure**

4.4. Estrategia de orquestación

Como se explicó en el capítulo 2 existen diferentes esquemas de comunicación en un programa paralelo. Los posibles esquemas de comunicación son los siguientes:

- Maestro-Esclavo
- Local

En el esquema de comunicación maestro-esclavo un proceso se dedica a coordinar a todos los demás procesos creados. Este proceso maestro concentra toda la información y la reparte entre los demás procesos llamados "esclavos".

Por el contrario, en el esquema de comunicación local no existen diferentes programas a utilizarse (un "maestro" y varios "esclavos"). En este esquema se ejecutan copias de un mismo programa los cuales pueden realizar comunicación entre ellos sin necesidad de un intermediario.

El esquema de comunicación local es más complejo en su programación ya que la sincronización entre los procesos se debe llevar a cabo entre los procesos mismos pues no hay un proceso que organice la sincronización de todos los procesos. Sin embargo, al no existir una concentración de la información en un proceso, el tiempo a utilizar para la resolución del problema es menor, siempre y cuando el esquema sea correctamente programado.

Con base en lo anterior, las herramientas de software utilizadas en este trabajo son las siguientes:

- El lenguaje de programación C y la biblioteca de funciones MPI

Se seleccionó la biblioteca de funciones MPI para su uso ya que permite la migración a diferentes arquitecturas sin necesidad de modificar el código fuente original.

El esquema de comunicación utilizado es conocido como SPMD (Programa Único Múltiples Datos) o comunicación local, esto por ofrecer una utilización más eficiente de los recursos. En este esquema cada uno de los procesadores tienen la misma jerarquía, es decir, no se requiere de un proceso maestro que controle la comunicación entre ellos. La figura 4.11 presenta el patrón de comunicación en los algoritmos, en la que se puede observar que cada uno de

los procesos tiene comunicación con todos los demás procesos, mediante la cual se enviarán los datos requeridos para continuar con su cálculo.

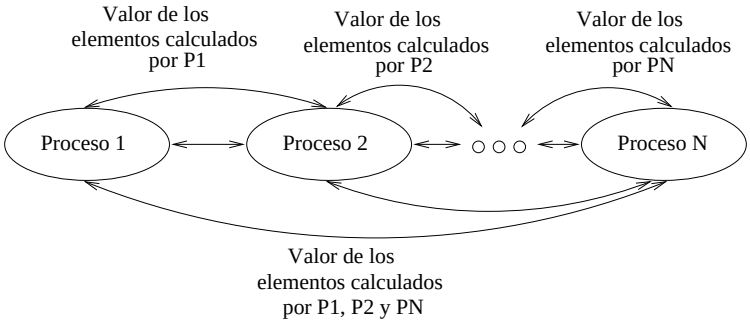

Figura 4.11: Modelo de comunicación implementado

Al utilizar una arquitectura de memoria distribuida se debe realizar un desarrollo más profundo de las ecuaciones 4.1 y 4.3, las cuales modelan el tiempo de ejecución para memoria compartida. La ecuación 4.6 presenta un modelo matemático que aproxima el tiempo de ejecución para memoria distribuida con el particionamiento por diagonales. Cabe mencionar que el modelo de comunicación ($T_{comm} = \lambda + \beta n_c$) es lineal, donde λ es la latencia, β es el tiempo requerido para la comunicación y n_c es la longitud del mensaje. Por lo anterior, este es un modelo simplificado que nos ayuda a aproximar el tiempo total de ejecución.

$$T_p \approx \frac{T_c \cdot n^4}{p} + \sum_{i=4}^{n-1} n_p \left(\lambda + \frac{n-i}{p}\beta \right) \tag{4.4}$$

$$T_p \approx \frac{T_c \cdot n^4}{p} + n_p \left((n-4)\lambda + \sum_{i=4}^{n-1} \left(\frac{n-i}{p}\beta \right) \right) \tag{4.5}$$

$$T_p \approx \frac{T_c \cdot n^4}{p} + n_p \left((n-4)\lambda + \frac{(n-5)(n-4)\beta}{2p} \right) \tag{4.6}$$

donde:

T_c es una constante que indica el tiempo utilizado en el cálculo

p es el número de procesadores a utilizar

n es el número de bases en la estructura primaria (tamaño del problema)

n_p es el número de mensajes

λ es la latencia

β es el tiempo requerido para la comunicación

Para la implementación del algoritmo se utiliza la primitiva "broadcast" de la biblioteca MPI la cual hace un uso eficiente de los canales de comunicación. Esto nos permite que n_p crezca a razón de $p\log(p)$.

La ecuación 4.8 presenta un modelo matemático que aproxima el tiempo de ejecución para memoria distribuida con el particionamiento por bloques.

$$T_p \approx \frac{T_c \cdot n^4}{p} + \sum_{\alpha=1}^{B-1} (n_{p'}(\lambda + n_{c'}\beta)) \tag{4.7}$$

$$T_p \approx \frac{T_c \cdot n^4}{p} + n_{p'} \left((B-1)\lambda + \beta \sum_{\alpha=1}^{B-1} n_{c'} \right) \tag{4.8}$$

donde:

T_c es una constante que indica el tiempo utilizado en el cálculo

p es el número de procesadores a utilizar

n es el número de bases en la estructura primaria (tamaño del problema)

B es el número de bloques a utilizar

$n_{p'}$ es el número de mensajes de longitud $n_{c'}$ a utilizar

λ es la latencia

β es el tiempo requerido para la comunicación

Al igual que en el particionamiento por diagonales $n_{p'}$ crece a razón de $p\log(p)$. El valor para $n_{c'}$ es calculado como $(\frac{n-\alpha\cdot\delta}{p})^2$, el cual es el tamaño de la sección de la diagonal a enviar, siendo α el número de diagonal de bloques actual y δ la longitud de una arista del bloque.

Cabe destacar que los algoritmos están diseñados para arquitecturas homogéneas, donde los procesadores a utilizarse para el procesamiento tienen la misma capacidad. Las arquitecturas homogéneas son las más utilizadas actualmente por los sistemas de supercómputo a nivel mundial [16]. Por tanto, el **mapeo** se hace de manera directa y el número de procesos de divide equitativamente entre el número de procesadores a utilizar.

El siguiente capítulo presenta los resultados prácticos de los algoritmos presentados utilizando las herramientas computacionales descritas en este capítulo.

Capítulo 5

Resultados experimentales

En este capítulo se presentan los resultados obtenidos al realizar la programación de los algoritmos presentados en el capítulo 4.

Como plataforma de pruebas se utilizaron cuatro computadoras de multiprocesamiento con las siguientes características:

- Dos procesadores AMD Opteron a 2 GHz.
- Disco Duro SCSI de 60 GB.
- 2 GB de memoria RAM.
- Interfaz de red GigaBit Ethernet.

Todas las computadoras utilizan Linux como sistema operativo. Las pruebas se realizaron utilizando diferentes tamaños de problema, así como diferente número de procesadores, para de esta manera observar el tiempo de ejecución para cada uno de estos casos y observar el comportamiento al modificar estos dos factores (tamaño de problema y número de procesadores.)

5.1. Paralelización del algoritmo

El problema del RNA es un problema combinatorio en el cual el espacio de búsqueda crece de manera exponencial. Debido a esto, y haciendo uso de la programación dinámica, se

realizó la paralelización del algoritmo con las herramientas computacionales seleccionadas.

El primer punto a presentar en este capítulo son los resultados obtenidos con la paralelización del algoritmo con complejidad $O(n^4)$.

Se realizaron pruebas de cada uno de los tres enfoques programados del algoritmo:

- Versión secuencial
- Versión paralela con particionamiento por renglones
- Versión paralela con el particionamiento por bloques propuesto

Se realizaron evaluaciones de estructuras primarias con longitudes $n = 200, 400, ..., 3000$ para cada una de estas tres versiones mencionadas. Para cada valor n se realizaron cinco evaluaciones, de los cuales se obtuvo un promedio, el cual es presentado como resultado de cada valor n. Además se realizó cada una de estas pruebas utilizando un número de procesadores $p = 2, 4, 8$. Cabe mencionar que no fue necesario realizar un número mayor de evaluaciones dado que la plataforma de pruebas estuvo dedicada de manera exclusiva.

5.1.1. Resultados de la versión secuencial

Los primeros resultados a presentar son los obtenidos a partir de la versión secuencial, los cuales serán útiles para comparar los resultados de las versiones paralelas y de esta manera realizar un estimado de las aceleraciones obtenidas.

La versión secuencial se programó implementando en código C las ecuaciones 1.4, 1.5 y 1.6. Por lo que al hacer la llamada a la función $G(i, j)$, visitando cada una de las casillas de la diagonal principal y continuando con la diagonal superior inmediata, para de esta manera llegar al cálculo de $G(0, n-1)$ para obtener el valor óptimo de energía liberado, el programa calcula de manera recursiva el valor de $G_1(1, j)$, $G(i, h)$ y $G(h+1, j)$, la llamada a $G_1(i, j)$ genera de igual manera llamadas a $C(i, j)$, $G(i, h)$, $G(h+1, j)$, $G_1(i, h)$ y $G_1(h+1, j)$ por lo que al finalizar el proceso de llamadas recursivas la posición en la matriz para $G(0, n-1)$ contendrá el valor óptimo de la estructura secundaria.

El cuadro 5.1 presenta los tiempos de ejecución secuencial para las longitudes mencionadas. La figura 5.1 presenta la gráfica de tiempo contra longitud obtenida. En ésta se puede

observar gráficamente que el crecimiento en tiempo requerido se incrementa en gran medida conforme el número de bases aumenta.

Tamaño n	Tiempo secuencial (minutos)
200	0.02
400	0.10
600	0.55
800	1.77
1000	4.32
1200	9.5
1400	16.57
1600	29.2
1800	46.53
2000	70.25
2200	100.13
2400	141.37
2600	195.85
2800	261.8
3000	356.58

Cuadro 5.1: Tiempos de ejecución para la versión secuencial

Para realizar un estimado en tiempo utilizando un número de bases mucho mayor se recurrió a la realización de un modelo matemático el cual se obtuvo ajustando la curva de crecimiento en tiempo respecto a n (figura 5.1) a un polinomio del tipo:

$$f(x) = ax^4 + bx^3 + cx^2 + dx + e$$

de esta manera se puede predecir el comportamiento de la curva en tiempos no obtenidos prácticamente.

La figura 5.2 presenta la gráfica del modelo matemático obtenido del comportamiento del algoritmo secuencial utilizando los datos del cuadro 5.1.

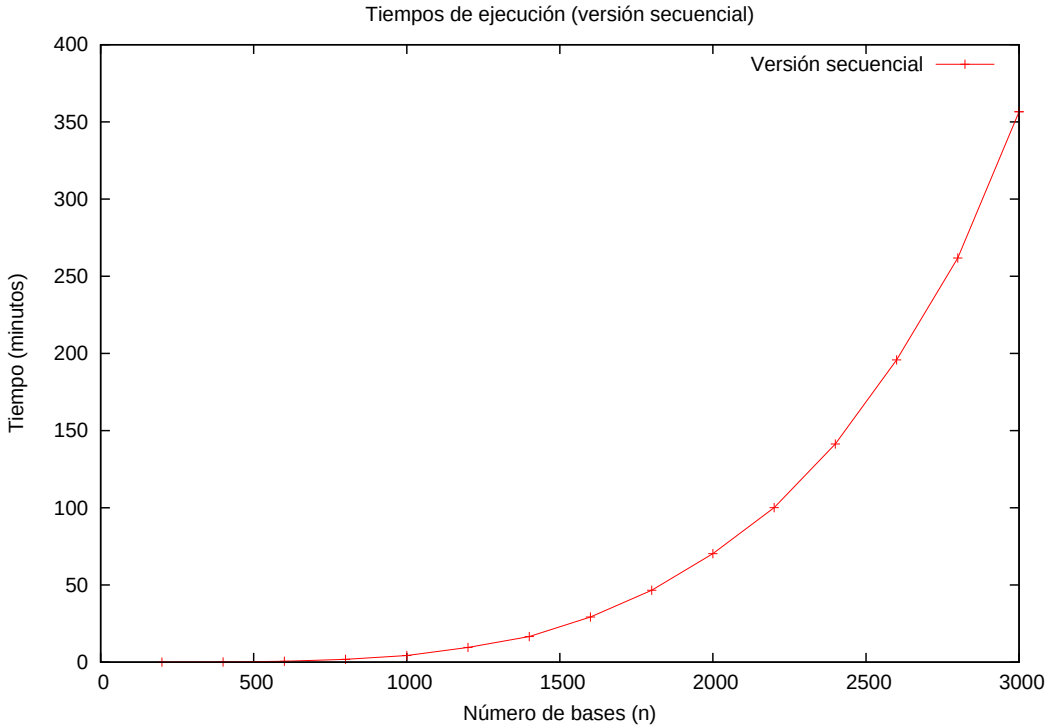

Figura 5.1: Tiempos de ejecución para la versión secuencial

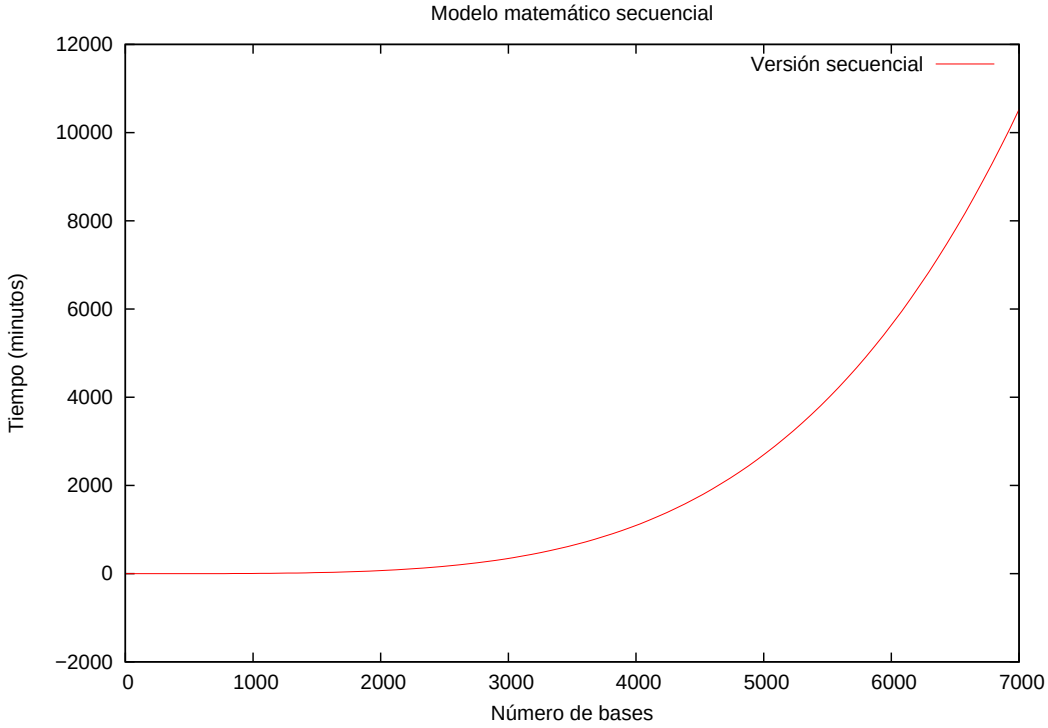

Figura 5.2: Modelo matemático del tiempo de ejecución secuencial

5.1.2. Resultados del particionamiento por diagonales

El particionamiento por diagonales se caracteriza por su buen balance de carga entre los procesadores. Su desventaja es el requerimiento de un total de $n-4$ procesos de sincronización siendo n la longitud de la estructura primaria.

La figura 5.3 presenta el tiempo de ejecución variando el número de procesadores a utilizar. Para diferentes longitudes n, en ésta se puede observar claramente la disminución de tiempo requerido por el algoritmo conforme crece el número de procesadores a utilizar. Otra manera de mostrar este comportamiento lo presenta la figura 5.4 en la cual se puede observar cómo para un número fijo de bases n, el tiempo a utilizar se reduce conforme se utiliza un mayor número de procesadores. Por último se presenta la figura 5.5, en la cual se muestran las aceleraciones obtenidas al utilizar el particionamiento por diagonales.

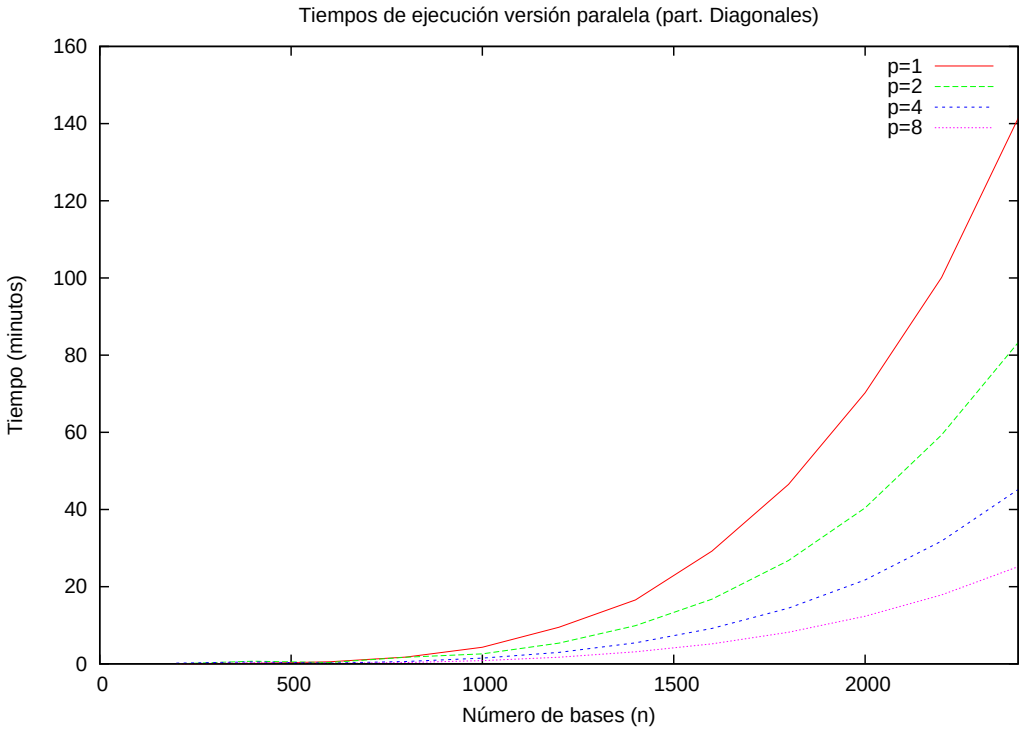

Figura 5.3: Tiempos de ejecución para la versión paralela (Diagonales) para diferentes longitudes n variando el número de procesadores

El cuadro 5.2 presenta los tiempos requeridos para calcular una cantidad de bases igual a las utilizadas en la versión secuencial, además de presentar la aceleración obtenida al realizar el cálculo de cada longitud n. Se puede observar que la aceleración obtenida tiende a crecer

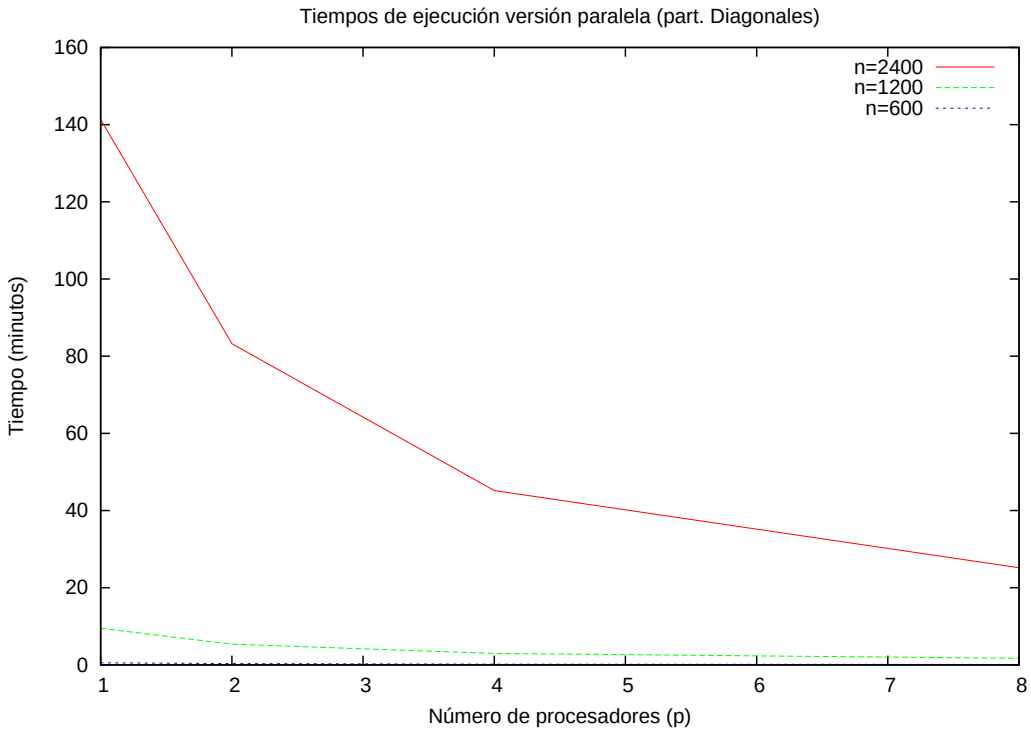

Figura 5.4: Tiempos de ejecución para la versión paralela (Diagonales) para diferente número de procesadores p variando la cantidad de bases a utilizar

junto con el número de bases de la estructura primaria, pero ésta llega a su máximo en

$$\text{aceleración} \approx 6$$

lo que representa que el tiempo utilizado para comunicaciones es de aproximadamente $\frac{2}{8}$, esto debido a la utilización de 8 procesadores.

La figura 5.6 presenta una gráfica del crecimiento en tiempo con respecto al número de bases a utilizar. En ésta se puede observar que las gráficas presentadas en las figuras 5.1 y 5.6 presentan un crecimiento muy similar, sólo afectado por un factor de escalamiento casi constante al cual denominamos anteriormente *aceleración*.

La figura 5.7 presenta la gráfica del modelo matemático (ecuación 5.1) para predecir el crecimiento en tiempo de ejecución utilizando un particionamiento por diagonales en la versión paralela. Este modelo se obtuvo desarrollando la ecuación 4.8, de tal manera que agrupando constantes y simplificando el desarrollo total de la ecuación se obtuvo un modelo que refleja el crecimiento aproximado en tiempo, además de permitir variar el número de procesadores p a utilizar. En esta figura se puede observar claramente que el hecho de

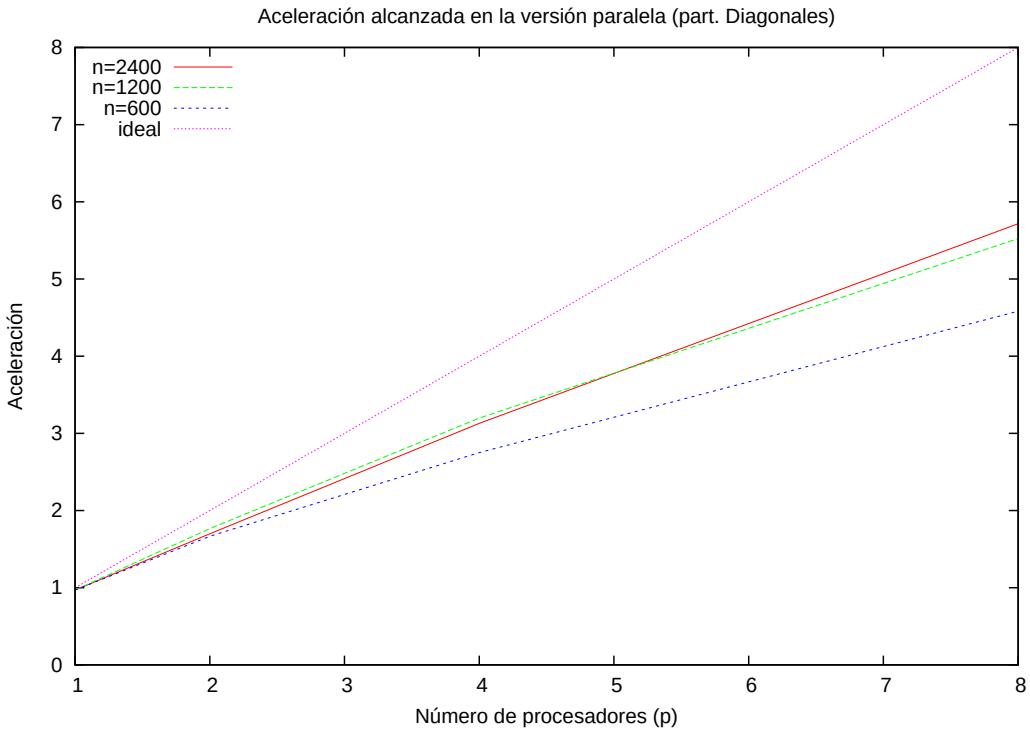

Figura 5.5: Aceleraciones obtenidas mediante el particionamiento por diagonales

aumentar el número de procesadores al doble no implica llevar al doble la eficiencia del programa, esto debido a la sobrecarga de comunicaciones que genera un número mayor de procesadores sobre esta misma arquitectura.

$$T(n,p) = \frac{n}{p}n^4 + b\log(p)n^2 + cp\log(p)n + d \tag{5.1}$$

Tamaño n	Tiempo Part. Diagonales (minutos)	Aceleración
200	0.08	0.25
400	0.3	0.333333
600	0.12	4.58333
800	0.35	4.78378
1000	0.85	5.08235
1200	1.72	5.58824
1400	3.12	5.59393
1600	5.17	5.59387
1800	8.20	5.72325
2000	12.35	5.74877
2200	17.85	5.47158
2400	25.17	5.54392
2600	34.30	5.69663
2800	45.65	5.71865
3000	59.38	6.01113

Cuadro 5.2: Tiempos de ejecución para la versión paralela (Diagonales) y aceleración obtenida utilizando 8 procesadores

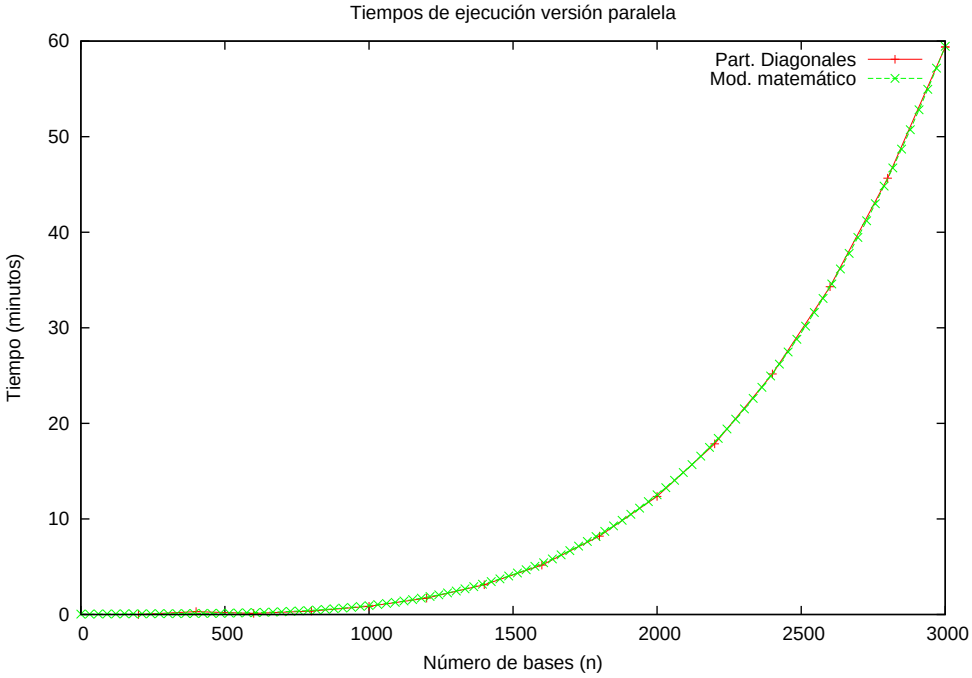

Figura 5.6: Tiempos de ejecución para la versión paralela (Diagonales) vs. su modelo matemático obtenido utilizando 8 procesadores $p = 8$

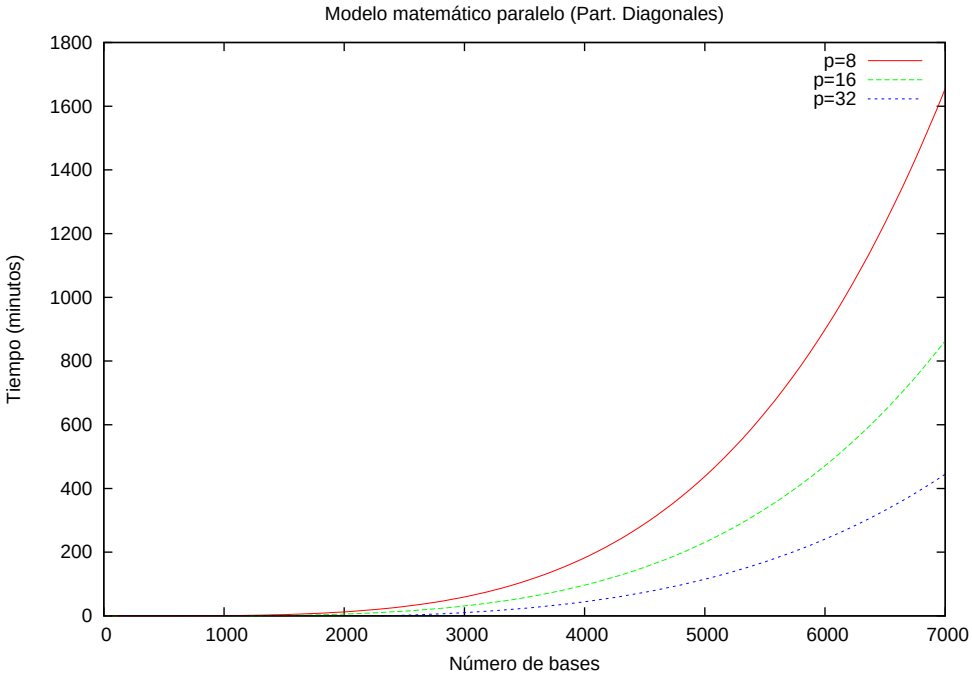

Figura 5.7: Modelo matemático del tiempo de ejecución para la versión paralela (Diagonales) utilizando diferente número de procesadores

5.1.3. Resultados del particionamiento por bloques

Antes de presentar los resultados obtenidos por la segunda versión paralela implementada, utilizando un particionamiento denominado "por bloques", se presentará el análisis de un problema implícito a este tipo de particionamiento.

Número óptimo de bloques a utilizar

El problema de cómo calcular el tamaño óptimo de bloques a utilizar [25] es muy importante para este tipo de particionamiento. Sin un estudio previo de este problema se pueden desperdiciar los beneficios de este particionamiento y requerir más tiempo para encontrar la estructura secundaria óptima que el requerido en el particionamiento por diagonales.

La figura 5.8 muestra un ejemplo del tiempo requerido para que se resuelva el problema del RNA utilizando la misma longitud en la estructura primaria pero variando el número de bloques a utilizar. Esta gráfica permite apreciar el comportamiento que presenta el algoritmo al variar el número de bloques. Lo que se busca es conocer de manera previa el número óptimo de bloques (o un valor muy cercano a éste) para que de esta manera el tiempo requerido sea el mínimo.

Se estimó de manera práctica una expresión para predecir la longitud óptima de las aristas de los bloques, la cual requiere sólo de la longitud del problema y del número de procesadores a utilizar. Sea v el área óptima para cada bloque, en [25] se estima de manera muy general esta área óptima, asumiendo que $p^2 \leq n$ para una utilización eficiente de los recursos se fija una arista del bloque a $\frac{n}{p^2}$ y la otra a un valor desconocido x. La ecuación 5.2 presenta la ecuación para el área óptima.

$$v = x\frac{n}{p^2} \tag{5.2}$$

La ecuación 5.3 presenta un análisis de la longitud de arista óptima, es decir, qué valor debe tener una arista del bloque. Este análisis, realizado de manera teórica, permite aislar distintas constantes dependientes de la arquitectura y características propias de la computadora a utilizar, α es el tiempo utilizado para ejecutar una instancia de $C(i,j)$ (ecuación 1.2), β es un parámetro de sincronización, γ es el inverso del ancho de banda en la red a

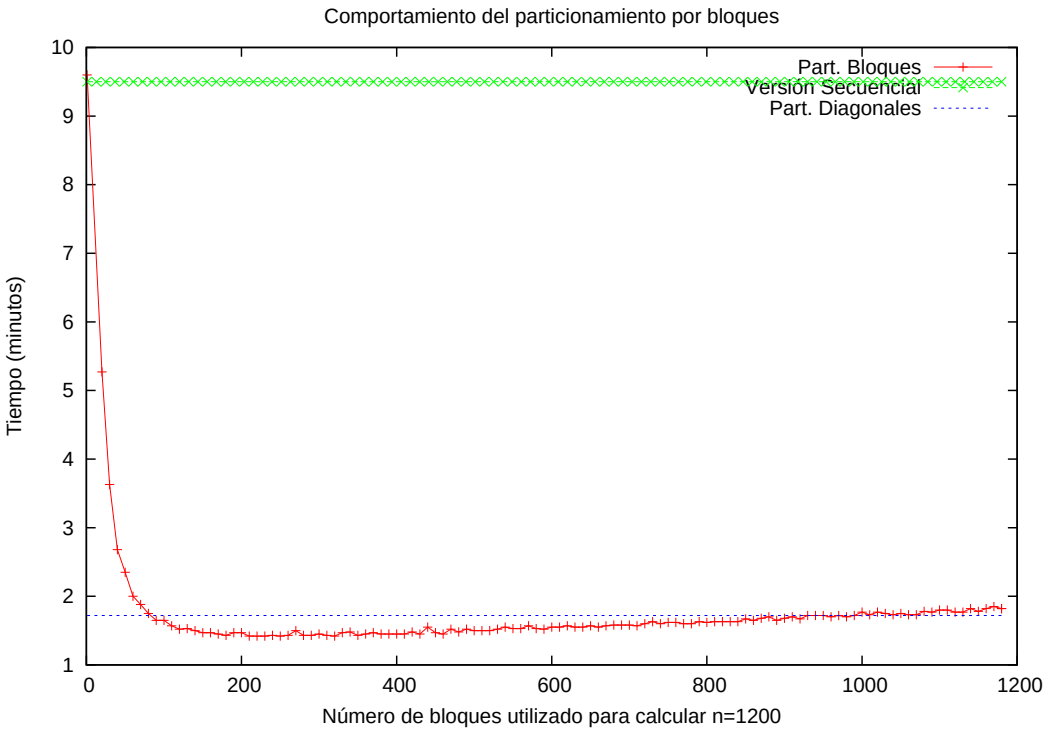

Figura 5.8: Tiempo requerido para una longitud $n = 1200$ variando el número de bloques a utilizar

utilizar y l es un factor por el cual multiplicar el tamaño del problema.

$$\text{Longitud de arista} = \sqrt{\frac{2ln\beta}{p^2 \cdot (\gamma + \alpha)}} \tag{5.3}$$

donde:

n, es la longitud de la estructura primaria

p, es el número de procesadores a utilizar

En este estudio se asumen las diferentes longitudes de los bloques rectangulares como idénticas (bloques cuadrados), debido la mínima diferencia entre las longitudes de las aristas (capítulo 4.) Por esto último y utilizando la ecuación 5.2 se obtuvo la longitud óptima de las aristas del bloque como se muestra en la ecuación 5.4.

$$\text{Longitud de arista} = \sqrt{x\frac{n}{p^2}} \tag{5.4}$$

donde:

x, es una constante dependiente de la arquitectura y características de la computadora

n, es la longitud de la estructura primaria

p, es el número de procesadores a utilizar

De la ecuación 5.4 podemos derivar la expresión para obtener el número óptimo de bloques a utilizar (ecuación 5.5), donde se asume que $c = \frac{1}{\sqrt{x}}$. Cabe destacar que para el cluster utilizado para este trabajo se obtuvo un valor de $c = 0{,}796026381$.

$$\text{número de bloques} = c \cdot \sqrt{n} \cdot p \tag{5.5}$$

La figura 5.9 presenta el crecimiento del número de bloques respecto a n en la arquitectura utilizada.

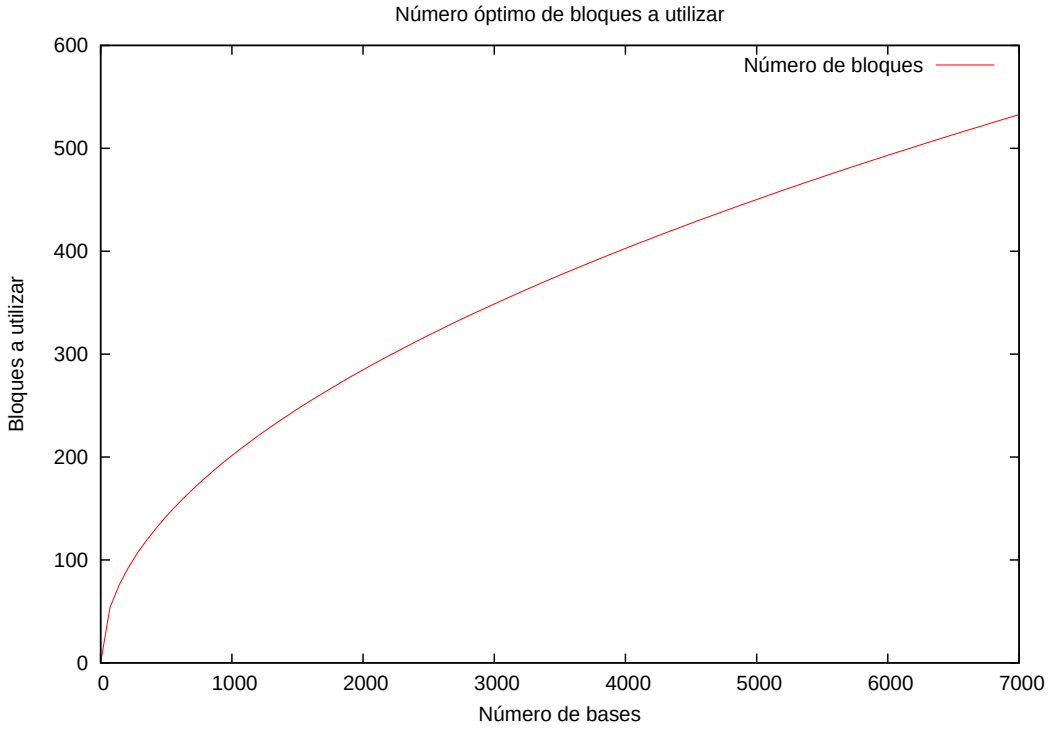

Figura 5.9: Número óptimo de bloques a utilizar (para 8 procesadores)

Con la obtención de este resultado se presentarán los resultados obtenidos del particionamiento por bloques.

Resultados del particionamiento por bloques

Las gráficas 5.10, 5.11 y 5.12 permiten observar de manera sencilla el comportamiento de este particionamiento, antes de analizar éste de manera más detallada. La figura 5.10 presenta

el comportamiento en tiempo variando el número de procesadores a utilizar y utilizando diferentes longitudes n. En ésta se puede observar claramente la disminución de tiempo requerido por el algoritmo conforme crece el número de procesadores a utilizar, el cual es menor al requerido por el particionamiento por diagonales (figura 5.3). Otra manera de mostrar este comportamiento lo presenta la figura 5.11, en ésta se puede observar cómo para un número de bases n el tiempo a utilizar se reduce, a mayor medida que en la figura 5.4, conforme se utiliza un mayor número de procesadores. Por último se presenta la figura 5.12, en la cual se muestran las aceleraciones obtenidas al utilizar el particionamiento por bloques propuesto.

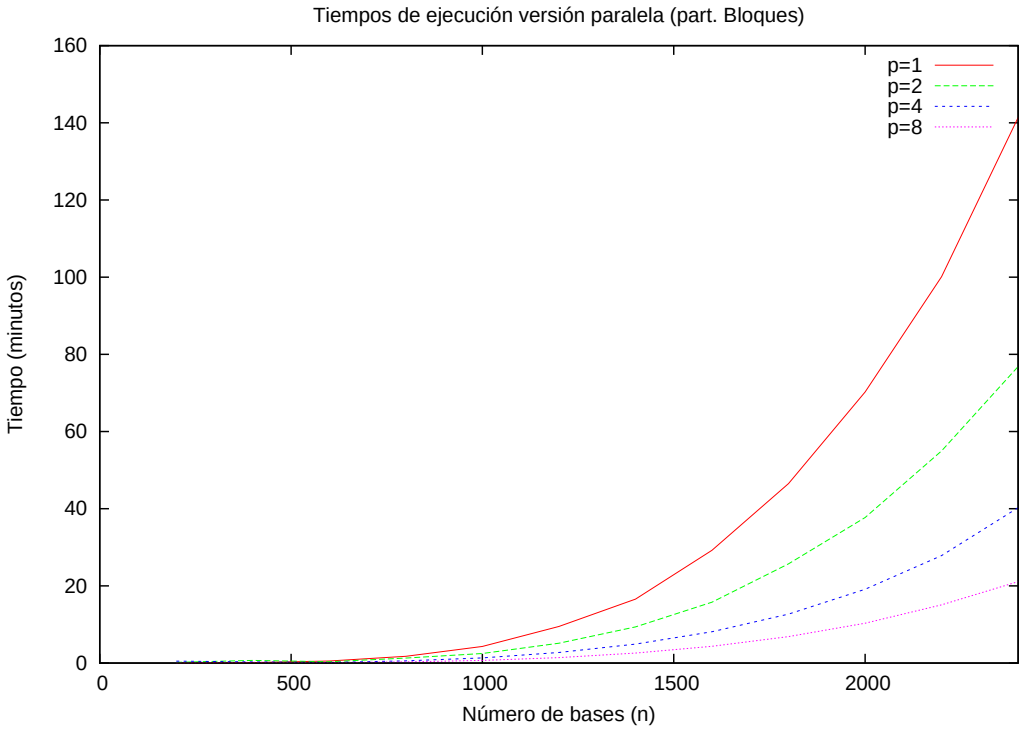

Figura 5.10: Tiempos de ejecución para la versión paralela (Bloques) para diferentes longitudes n variando el número de procesadores

El cuadro 5.3 muestra los tiempos requeridos para la solución del problema del RNA al utilizar una estructura primaria de longitud n, así como lo bloques utilizados para obtener este tiempo y la aceleración alcanzada con respecto al algoritmo secuencial. Cuando el tamaño de problema es muy pequeño usar un algoritmo paralelo no es útil, pues la sobrecarga que genera la programación paralela (latencia, inicialización de las comunicaciones, paso de

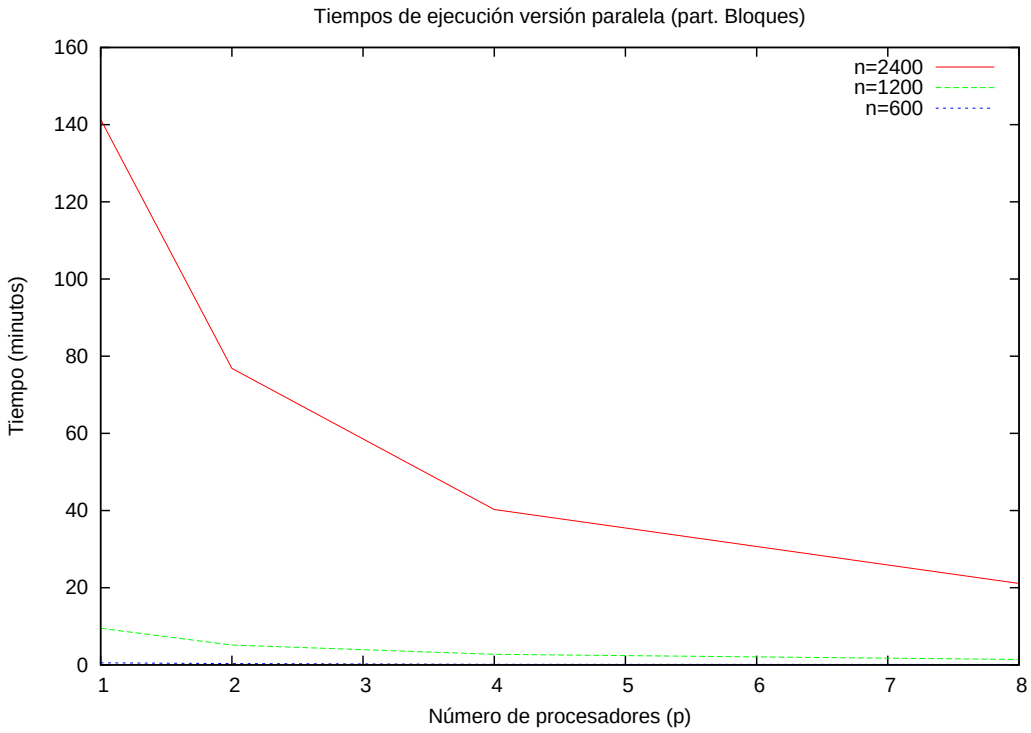

Figura 5.11: Tiempos de ejecución para la versión paralela (Bloques) para diferente número de procesadores p variando la cantidad de bases a utilizar

mensajes) es mayor en tiempo que el requerido por el algoritmo secuencial. Pero se puede observar que conforme se aumenta el tamaño del problema la aceleración crece y comienza a estabilizarse hasta lograr un máximo en

$$\text{aceleración} \approx 7$$

lo que representa que el tiempo utilizado para comunicaciones es de aproximadamente $\frac{1}{8}$, esto debido a la utilización de 8 procesadores.

La figura 5.13 presenta una gráfica del crecimiento en tiempo con respecto al número de bases a utilizar en el particionamiento por bloques. Al igual que con el particionamiento por diagonales en este particionamiento se puede observar que las gráficas presentadas en las figuras 5.1 y 5.13 presentan un crecimiento muy similar, sólo afectado por el factor de escalamiento denominado *aceleración*.

La figura 5.14 presenta la gráfica del modelo matemático para predecir el crecimiento en tiempo utilizando un particionamiento por bloques (5.6) en la versión paralela. Este modelo se obtuvo de la misma manera que el modelo anterior (ahora utilizando la ecuación 4.8) y al

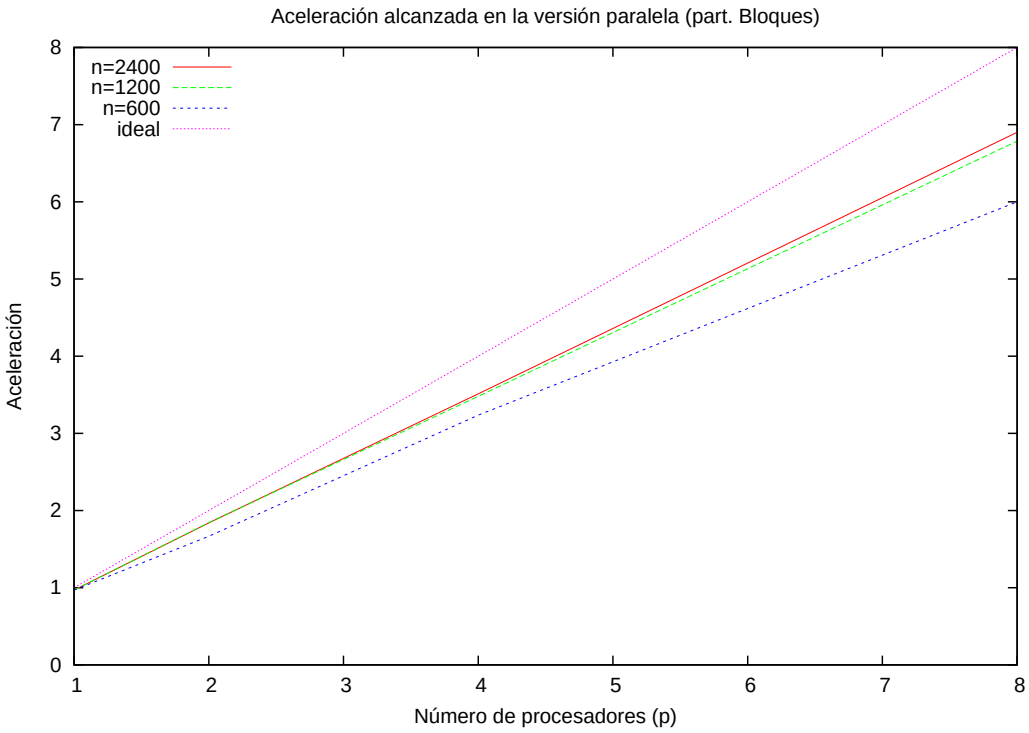

Figura 5.12: Aceleraciones obtenidas mediante el particionamiento por bloques

igual que éste muestra la sobrecarga en comunicaciones que genera agregar más procesadores a esta misma arquitectura.

$$T(n,p) = \frac{n}{p}n^4 + b\log(p)n^2 + cp\log(p) \tag{5.6}$$

Para finalizar se presentan las figuras 5.15 y 5.16, en las cuales se muestran las gráficas obtenidas por las versiones paralelas programadas, es decir, el particionamiento por diagonales contra el particionamiento por bloques propuesto en este trabajo. En la primera se presentan los tiempos de ejecución para diferentes tamaños de problema n. En ésta se observa que el particionamiento por bloques requiere menor tiempo para la solución del problema, lo cual se repite en la figura 5.16 donde las aceleraciones obtenidas por el particionamiento por bloques son mayores en todos los casos presentados de n para los diferentes valores de p.

Con los resultados de este capítulo podemos concluir que basados en pruebas prácticas el particionamiento propuesto en este trabajo, así como el estudio del número de bloques óptimo, presenta una utilización más eficiente de los recursos, lo que se demuestra en una disminución importante del tiempo a utilizar elevando la máxima aceleración obtenida (uti-

Tamaño n	Tiempo Part. Diagonales (minutos)	Bloques utilizados	Aceleración
200	0.05	88	0.4
400	0.2	120	0.5
600	0.10	152	5.5
800	0.30	176	5.9
1000	0.70	200	6.17143
1200	1.40	216	6.78571
1400	2.58	232	6.42248
1600	4.35	248	6.71264
1800	6.85	264	6.7927
2000	10.33	280	6.80058
2200	15.10	296	6.63113
2400	21.10	304	6.7
2600	28.77	320	6.80744
2800	38.28	336	6.83908
3000	50.42	344	7.07219

Cuadro 5.3: Tiempos de ejecución para la versión paralela (Bloques) utilizando 8 procesadores, número de bloques utilizados y aceleración obtenida

lizando 8 procesadores) de 6 (mediante un particionamiento por diagonales) a 7, lo cual representa que el tiempo utilizado por las comunicaciones en el particionamiento por bloques se redujo a la mitad de lo requerido por el particionamiento mediante diagonales. Mediante los modelos matemáticos realizados podemos predecir, de una manera aproximada, que estas relaciones de aceleración se mantendrán para cualquier número de bases n utilizada.

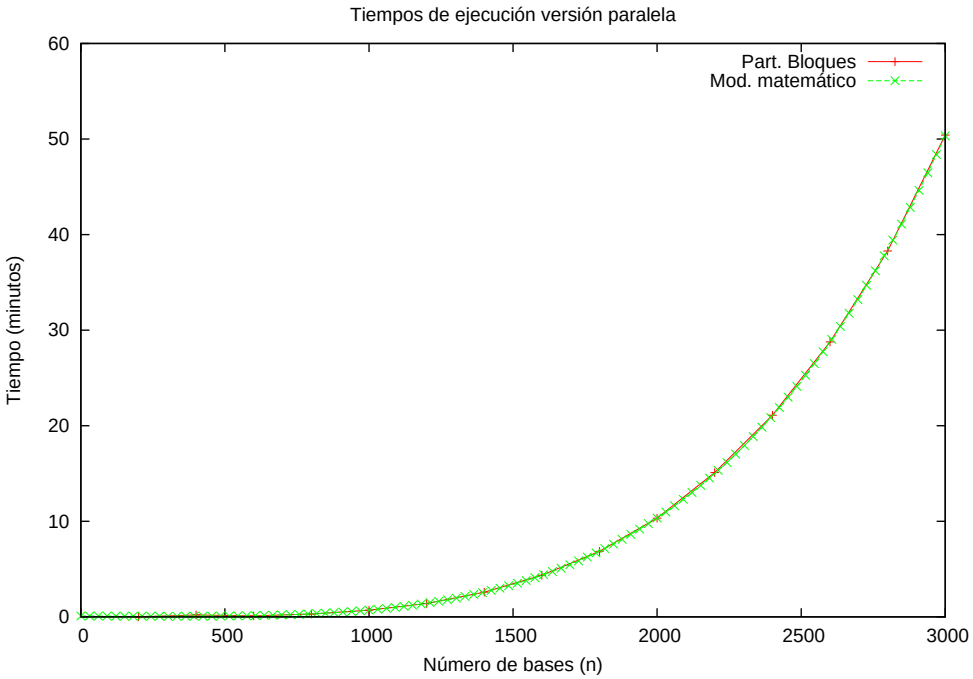

Figura 5.13: Tiempos de ejecución para la versión paralela (Bloques) vs. su modelo matemático obtenido utilizando 8 procesadores ($p = 8$)

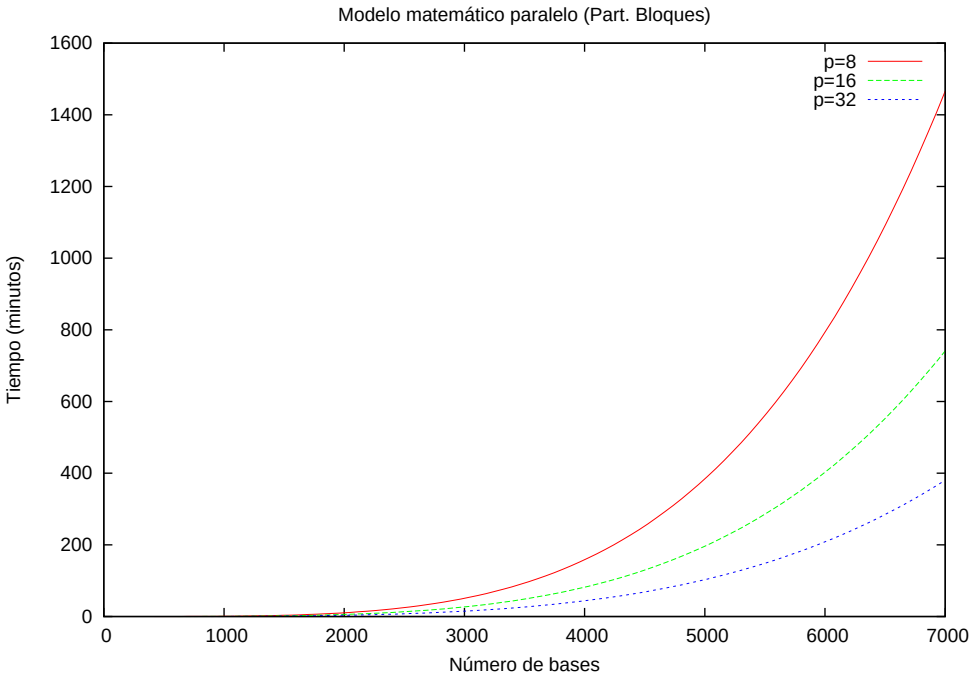

Figura 5.14: Modelo matemático del tiempo de ejecución para la versión paralela (Bloques) utilizando diferente número de procesadores

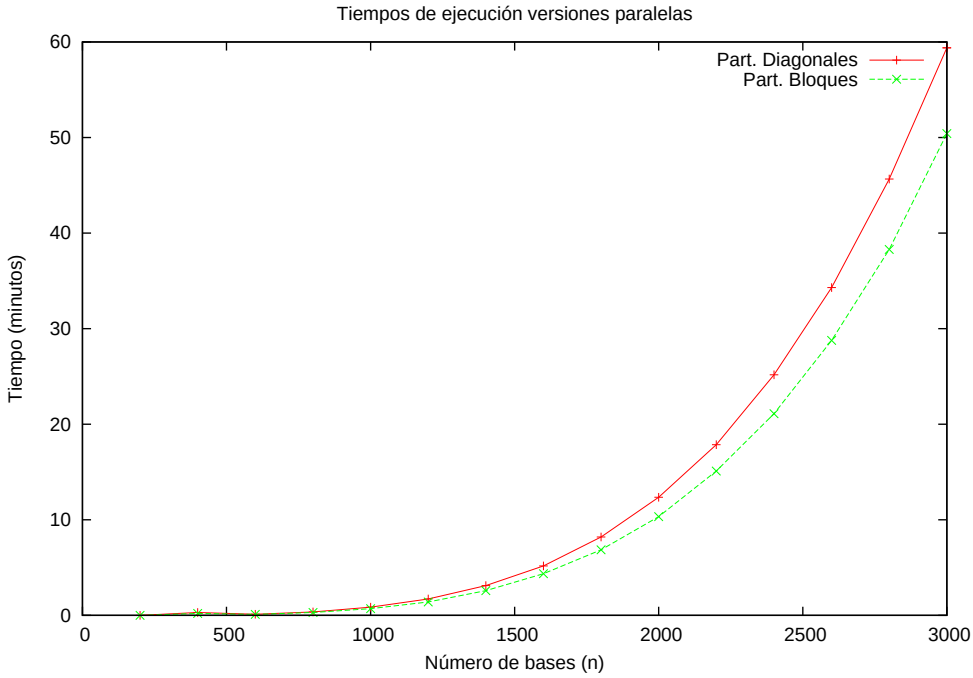

Figura 5.15: Tiempos de ejecución para las versiones paralelas (Bloques vs. Diagonales)

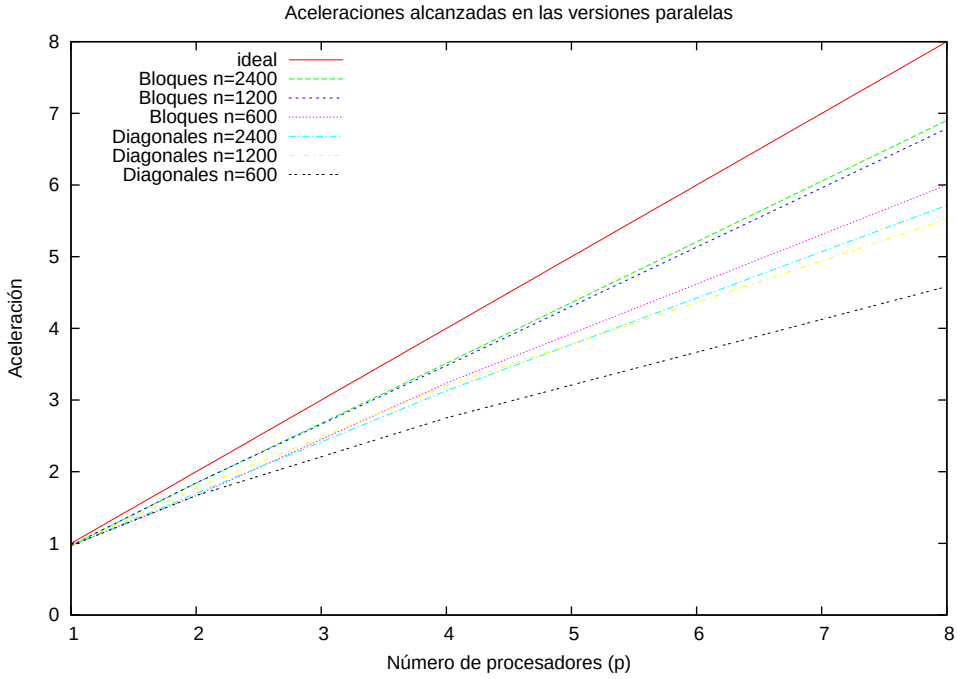

Figura 5.16: Aceleraciones obtenidas para las versiones paralelas (Bloques vs. Diagonales)

Conclusiones

Se propuso un nuevo esquema de paralelización para recurrencias con un espacio de iteración triangular y dependencias no uniformes. Nuestro interés se centra en las recurrencias de la programación dinámica para resolver el "Problema del RNA".

Realizando un análisis del espacio de búsqueda total para este problema, utilizando un método de búsqueda exhaustiva, se obtuvo que el número de soluciones a verificar crece de manera exponencial ($\Omega(2^n)$), por lo que computacionalmente es intratable. A partir de los algoritmos presentados en [1] se determinó que encontrar la estructura secundaria que presenta la liberación óptima de energía utilizando programación dinámica reduce el espacio de búsqueda a $\Theta(n^2)$, pero los tiempos de ejecución requeridos son $O(n^3)$ y $O(n^4)$, por lo que sigue siendo un proceso muy costoso en tiempo. Esto obliga a la utilización de una herramienta que permita reducir los tiempos requeridos por estos algoritmos.

Se observó que aunque el proceso de búsqueda encuentre más de una estructura secundaria estable éste sólo es capaz de rastrear una de las soluciones del espacio de búsqueda. Para dar solución a este último problema presentado se propuso un nuevo tipo de rastreo el cual permite recuperar todas las soluciones óptimas (estructuras secundarias estables) que presente una estructura primaria dada. Además se presentaron los algoritmos que permiten la reconstrucción de todas las soluciones encontradas en el proceso de búsqueda.

La programación paralela permite reducir los tiempos que un proceso requiere haciendo uso de más de un procesador, es decir, dos o más procesadores trabajarán juntos para la solución de un problema común. El problema del RNA utiliza la programación dinámica como medio para buscar una estructura secundaria estable en el espacio de soluciones. Las tablas de programación dinámica a utilizar se pueden representar mediante una matrices triangulares superiores. Es necesario realizar un análisis de las dependencias de datos que

presentan estas tablas de tal manera que se pueda presentar un esquema de particionamiento que sea correcto, es decir, un esquema de paralelización el cual utilice de una manera más eficiente los recursos computacionales.

Una vez analizadas las dependencias entre las tablas de programación dinámica, se presentaron dos tipos de particionamiento los cuales permiten la independencia de cálculo entre los procesos involucrados en la solución del problema. El primer esquema presentado es el denominado "particionamiento por diagonales", el cual presenta un buen balance de carga entre los procesos a utilizar pero requiere de una gran cantidad de pasos de sincronización lo que puede disminuir el rendimiento del programa paralelo. En la implementación de este esquema se obtuvo una aceleración aproximada de 6, utilizando 8 procesadores en una arquitectura homogénea, lo cual implica que para esta arquitectura el tiempo total utilizado en las comunicaciones fue de aproximadamente $\frac{2}{8}$.

El segundo esquema presentado es el denominado "particionamiento por bloques", el cual requiere de un menor número de procesos de sincronización. Se resolvieron algunos problemas que presenta este esquema de manera inherente, el primero es que los bloques a utilizar originalmente son cuadrados, debido a esto el balance de carga puede ser deficiente para los procesos encargados de calcular el residuo de la división de las tablas de programación dinámica entre el número de bloques. Para solucionar este problema se propuso la utilización de bloques rectangulares los cuales cubran todo el espacio de las tablas de programación dinámica. Además, este particionamiento asegura que la diferencia máxima entre las dimensiones de rectángulos contiguos sea a lo más uno. Esto último con la finalidad de mantener el buen balance de carga entre los procesadores (al igual que en el particionamiento por diagonales.)

El segundo problema que presenta el particionamiento por bloques es cómo encontrar el número de bloques óptimo a utilizar, es decir, el tamaño óptimo de bloques con el que se obtiene un menor tiempo total para la solución del problema. La solución de este problema se realizó mediante la formulación de un modelo matemático el cual, dependiente del número de bases (longitud de la estructura primaria) a utilizar, el número de procesadores y una constante que involucre los aspectos propios de la arquitectura, realice el cálculo *a priori* acerca del número de bloques a utilizar para que el particionamiento por bloques presente el

menor tiempo para la solución del problema. Realizando este análisis en nuestra plataforma de pruebas se determinó que para un número de procesadores $p = 8$, la constante dependiente de la arquitectura tiene un valor de $c = 0{,}796026381$, por lo que para un número de bases $n = 1200$ se obtiene que el número óptimo de bloques a utilizar es 216.

Se presentó una implementación de la estrategia propuesta, con la cual se logró alcanzar una aceleración aproximada de 7, utilizando 8 procesadores en una arquitectura homogénea, lo cual implica que para esta arquitectura el tiempo total utilizado en las comunicaciones fue de aproximadamente $\frac{1}{8}$. Con esto se obtuvo que el particionamiento por bloques propuesto requiere la mitad de tiempo de comunicaciones con respecto al particionamiento por diagonales.

Se obtuvieron los modelos matemáticos de los tiempos de ejecución para las dos estrategias presentadas en esta tesis. Estos modelos nos permitieron aproximar los tiempos de ejecución para casos que no fueron realizados de manera práctica, como lo fue un número mayor de bases n y un número de procesadores mayor al disponible en la arquitectura utilizada.

Las aceleraciones obtenidas, con respecto a la versión secuencial del algoritmo, de las versiones paralelas implementadas permiten concluir que el particionamiento por bloques propuesto en este trabajo es un esquema eficiente para la solución del problema del RNA. Además se presentó la manera en que se pueden encontrar todas las estructuras secundarias estables que se presenten en el espacio de búsqueda de nuestro problema.

Algunos aspectos en la implementación realizada en este trabajo son susceptibles a cambios que pueden mejorar su rendimiento. Uno de estos aspectos es el tipo de orquestación realizada, la implementada en esta tesis utiliza un tipo de comunicación "broadcast", es decir, un proceso envía toda su información a los procesos restantes en un solo paso, lo que genera un crecimiento en número comunicaciones de tipo $p\log(p)$. Si se utilizara una arquitectura paralela lo suficientemente eficiente en sus comunicaciones se podría realizar un tipo de comunicación "all-to-all", en la cual todos los procesos envían toda su información en un solo paso, esta directiva de comunicaciones presenta un crecimiento en número de comunicaciones de tipo $\log(p)$, lo que mejoraría el rendimiento aún en arquitecturas de muchos procesadores como las presentadas en [16]. Además se puede realizar un análisis más

exhaustivo acerca del modelado del número óptimo de bloques a utilizar, de tal manera que se logre el rendimiento ideal para el particionamiento propuesto en esta tesis.

A partir de este trabajo se puede realizar una adecuación de los parámetros en el cálculo de la liberación de energía en determinados ciclos y apareamientos de bases, para de esta manera poder validar de una manera práctica los resultados obtenidos al implementar los algoritmos presentados en esta tesis. Cabe destacar que esto no modificaría el trabajo aquí presentado, sólo ajustaría la forma de las estructuras secundarias obtenidas con el fin de complementar este estudio.

Bibliografía

[1] D. Sankoff and J. B. Kruskal, *Time warps, String edits and Macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, 1983.

[2] J.-H. Chen, S.-Y. Le, B. A. Shapiro, and J. Maizel, "Optimization of an RNA folding algorithm for parallel architectures," *Parallel Computing*, vol. 24, pp. 1617–1634, 1998.

[3] M. T. Vidal, "Estrategias de particionamiento paralelo para el problema de RNA," Master's thesis, CINVESTAV-IPN, 2002.

[4] R. Nussinov, G. Pieczenick, J. R. Griggs, and D. J. Kleitman, "Algorithms for loop matching," *SIAM Journal on Applied Mathematics*, vol. 35, pp. 68–82, 1978.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[6] C. A. Coello, "Introducción a la computación evolutiva," 2004.

[7] A. Newell, J. C. Shaw, and H. A. Simon, "The processes of creative thinking," in *Comtemporary approaches to creative thinking*, H. E. Gruber, G. Terrell, and M. Wertheimer, Eds. New York: Atherton Press, 1962, pp. 63–119.

[8] F. Glover and M. Laguna, *Tabu search.* Norwell Massachusetts: Kluwer Academic Publishers, 1998.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983. [Online]. Available: citeseer.ist.psu.edu/kirkpatrick83optimization.html

[10] J. H. Holland, *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[11] V. L. Murthy and G. D. Rose, "RNABase: an annotated database of RNA structures," *Nucleic Acids Research*, vol. 31, 2003.

[12] P. Gendron, S. Lemieuxs, and F. Major, "Quantitative analysis of nucleic acid three-dimensional structures," *Journal Molecular Biology*, vol. 308, pp. 919–936, 2001.

[13] M. J. Quinn, *Parallel Computing. Theory and practice.* McGraw-Hill, 1994.

[14] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kauffman Publishers, 1998.

[15] A. J. van der Steen and J. J. Dongarra, "Overview of recent supercomputers," 2004. [Online]. Available: http://top500.org/ORSC/2004/

[16] "Top 500 supercomputer sites." 2005. [Online]. Available: http://top500.org/

[17] B. W. Kernighan and D. M. Ritchie, *El lenguaje de programación C.* Murray Hill, New Jersey: AT&T Bell Laboratories, 1978.

[18] F. Mueller, "Pthreads library interface," 1993. [Online]. Available: citeseer.ist.psu.edu/article/mueller99pthreads.html

[19] LAM/MPI.Team, "Lam/mpi user's guide version 7.1.1," 2004.

[20] P. S. Pacheco, *Parallel programming with MPI.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[21] I. Hofacker, "Vienna RNA packages," 2000. [Online]. Available: http://www.tbi.univie.ac.at/~ivo/RNA/

[22] M. Ess, "Folding RNA on the cray-2," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, 1990, pp. 103–111.

[23] M. Zuker and P. Stiegler, "Optimal computer folding of large RNA sequences using thermodynamic and auxiliary information," *Nucleic Acids Research*, vol. 9, pp. 133–148, 1981.

[24] B. A. Shapiro, J. C. Wu, and D. Bengali, "The massively parallel genetic algorithm for RNA folding: MIMD implementation and population variation," *Bioinformatics*, vol. 17, pp. 137–148, 2001.

[25] F. Almeida, R. Andonov, D. Gonzalez, L. M. Moreno, V. Poirriez, and C. Rodriguez, "Optimal tiling for the RNA base pairing problem," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM Press, 2002, pp. 173–182.

[26] J. S. Deogun, R. Donis, O. Komina, and F. Ma, "RNA secondary structure prediction with simple pseudoknots," in *CRPIT '29: Proceedings of the second conference on Asia-Pacific bioinformatics*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, pp. 239–246.

[27] R. B. Lyngso and C. S. Pedersen, "Pseudoknots in RNA secondary structures," in *RECOMB '00: Proceedings of the fourth annual international conference on Computational molecular biology*. New York, NY, USA: ACM Press, 2000, pp. 201–209.