CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
SECCIÓN DE COMPUTACIÓN

Estrategias de particionamiento paralelo para el problema de RNA [1]

TESIS QUE PRESENTA

Lic. Mireya Tovar Vidal

PARA OBTENER EL GRADO DE

Maestra en Ciencias

EN LA ESPECIALIDAD DE

Ingeniería Eléctrica, opción Computación

DIRECTOR DE TESIS:

Dr. Arturo Díaz Pérez

México, D.F. Octubre 2002

[1]Este trabajo fue parcialmente financiado mediante el proyecto CONACYT 31892-A Algoritmos y arquitecturas de computadoras con dispositivos reconfigurables

Resumen

Las secuencias de RNA se utilizan entre otras cosas para caracterizar microorganismos en biomedicina molecular. Una secuencia de RNA consiste de una lista de bases o nucleótidos, a la cual se le conoce como la estructura primaria; cuando las bases se enlazan unas con otras se forma lo que se conoce como estructura secundaria. El problema de RNA consiste en encontrar, de todas las estructuras secundarias posibles, aquella que tiene una estructura secundaria estable. Esto conduce a un problema de optimización el cual en el caso general es considerado como un problema exponencial. Sin embargo, existen dos algoritmos debidos a Sankoff, con complejidades en tiempo $O(n^3)$ y $O(n^4)$, que resuelven casos especiales del problema general. Estos algoritmos se basan principalmente en el problema de múltiples ciclos y para su solución utilizan la estrategia de programación dinámica. A pesar de la complejidad polinomial de los algoritmos, el tiempo para encontrar estructuras secundarias es considerable para resolver secuencias largas. En este trabajo se realiza un análisis de dependencias de datos y se proponen estrategias de particionamiento paralelo para disminuir el tiempo computacional para secuencias de longitud considerable. Estos algoritmos paralelos se implementaron en dos tipos de arquitecturas paralelas: multiprocesadores (memoria compartida) y multicomputadoras (memoria distribuida). Se diseñaron dos esquemas diferentes de comunicación para memoria distribuida con la finalidad de obtener mejores resultados en tiempo de ejecución (comunicación maestro-esclavo y comunicación local). Se presentan los resultados experimentales obtenidos por estos algoritmos y se indican las ventajas y desventajas de cada arquitectura.

Abstract

RNA sequences are used, among other applications, in Molecular Biomedicine to classify microorganisms. A RNA sequence is a list of basis or nucleotides which is named primary structure; when different basis are joined together by chemical processes a RNA secondary structure is built. The RNA problem consists of finding, among all possible secondary structures, that having a stable structure. This leads to an optimization problem which has an exponential complexity in the general case. However, two algorithms due to Sankoff, with $O(n^3)$ y $O(n^4)$ complexities, solve special cases of the problem. Those algorithms are based in the multiple loop problem and they apply a dynamic programming strategy. In spite of their polynomial complexity, time taken to solve long sequences is considerably long. In this thesis, a dependence analysis over the sequential RNA algorithms is performed to propose parallel partitioning strategies; the goal is to reduce the processing time of long sequences. Parallel algorithms have been designed and implemented in two major parallel computer architectures: multiprocessors (shared memory) and multicomputers (distributed memory). For distributed memory, two communication strategies were incorporated in order to reduce execution times: master-slave and local communication. Experimental results of different configurations are presented and major advantages and disadvantages of both parallel architectures are discussed.

Agradecimientos

Agradezco profundamente a mi Madre por su apoyo y comprensión, gracias a tus consejos he logrado satisfacer mis metas y objetivos. A mi Padre y hermanos por su apoyo incondicional.

Le doy las gracias a mi supervisor, Dr. Arturo Díaz Pérez por su ayuda y orientación durante mi trabajo de tesis y estancia en la Maestría de Computación del departamento de Ingeniería Eléctrica.

También deseo darle las gracias a la familia Díaz Férnandez, en especial a Moisés, por su comprensión y apoyo continuo.

Agradezco el apoyo económico otorgado por CONACyT a través de una beca que me permitió cursar la Maestría en el departamento de Ingeniería Eléctrica, Sección Computación del Centro de Estudios Avanzados del Instituto Politécnico Nacional.

De la misma forma, agradezco la beca terminal otorgada por CINVESTAV-IPN para concluir esta tesis de Maestría y el apoyo financiero otorgado por el proyecto CONACYT 31892-A “Algoritmos y arquitecturas de computadoras con dispositivos reconfigurables” para difundir el trabajo realizado en varios eventos nacionales.

A todas esas personas que me mostraron su amistad y afecto durante mi permanencia en la ciudad de México, gracias.

Índice General

Índice de Figuras

Índice de Tablas

Introducción

El diseño de algoritmos paralelos es un área desafiante que requiere la integración de diversas técnicas para construir programas que exhiban un rendimiento aceptable. Aunque se sabe que los pasos fundamentales para construir un algoritmo paralelo deben de resolver los problemas de particionamiento, mapeo y calendarización para lograr algoritmos balanceados, lamentablemente, no existen técnicas generales que se puedan aplicar a cualquier problema [8]. Por tanto, es necesario desarrollar algoritmos paralelos *ad hoc* para cada problema y cada tipo de computadora paralela.

En el ámbito de programación dinámica se han desarrollado algunos algoritmos paralelos con un éxito relativo. Esto se debe fundamentalmente a la presencia de un gran número de dependencias para construir la tabla de programación dinámica [1, 6].

El problema de RNA se refiere a la forma de organizar una secuencia de moléculas de RNA. Una molécula de ácido ribonucleico (RNA) está constituida por una gran cantidad de subunidades (ribonucleicos) ligadas. Cada ribonucleico contiene una de cuatro posibles bases: adenanina (A), citosina (C), guanina (G) y uracil (U), y esta secuencia de bases distingue un tipo de RNA de otro y se le conoce como *estructura primaria* de la molécula de RNA. Bajo condiciones normales, un ribonucleótido cambia al doblarse y las bases forman uniones con otras formando un patrón más complicado, de esta forma las moléculas conforman ciclos. Así que la conformación y el patrón de unión es llamada la *estructura secundaria* de la molécula de RNA [3].

El problema fundamental consiste en encontrar una estructura secundaria óptima a partir de todo el conjunto de estructuras secundarias que se logren formar a partir de la estructura primaria. Tomando en cuenta que este problema es de complejidad $O(2^n)$, donde n es la longitud de la secuencia, se pueden aplicar diferentes estrategias para reducir la búsqueda. Este problema, manifiesta un conjunto de dependencias al organizar las subsecuencias necesarias para obtener una solución parcial [9]. Por lo cual, la estrategia de programación dinámica puede aplicarse para resolver este problema. Se han desarrollado algunos algoritmos que utilizan diferentes estrategias de optimización tales como algoritmos genéticos [20], tanto paralelos [12, 13] como secuenciales [15–18].

Experimentalmente, es costoso y complicado tratar de obtener la estructura secundaria de una secuencia de RNA; se han usado difracción de rayos-x del RNA cristalino y otras técnicas bioquímicas; sería útil obtener estas estructuras a partir de algún método que indique cómo obtener estructuras secundarias para moléculas muy grandes.

Existen en la literatura algunos programas que tratan de representar la estructura normal de la molécula del RNA de manera gráfica. Por ejemplo:

Los programas de computadora de Studnicka *et al.* (1978) [26] y Zuker y Stiegler (1981) [27] ambos producen una salida en línea de impresión de una representación normal, lo cual no es satisfactorio para moléculas muy grandes cuya estructura es altamente ramificable. Feldmann (no publicado) ha escrito un programa en SAIL llamado NUCSHO, produciendo una salida en línea de impresión, la cual es muy elegante e involucra traslapes [10]. Osterburg y Sommer (1981) [28] describen un programa, el cual coloca los pares cerrados de un múltiple ciclo igualmente espaciados sobre un círculo, pero existen traslapes con este método. Lapalme *et al.* (1982) producen una salida más agradable [29]. Shapiro *et al.* (1982) usan algunas características de terminales de vídeo para permitir al usuario indicar las regiones que deberían ser rotadas o redibujadas en un tamaño más grande [30].

Se han desarrollado algoritmos eficientes, tanto secuenciales como paralelos, que encuentran un subconjunto de estructuras secundarias que cumplen ciertas condiciones. En [12] se describen implementaciones paralelas de un algoritmo de programación dinámica para predecir la estructura secundaria del ácido ribonucleico (RNA) basado en la minimización de energía en computadoras de alta ejecución. Se implementaron dos métodos sobre una CRAY Y-MP que consisten en la forma en que se llena la tabla de programación dinámica. El primero se realiza por medio de filas y el segundo en forma de diagonal. Sus resultados demuestran que el método con menos conflictos de filas se ejecuta mejor en un ambiente de un solo procesador. Sin embargo, el otro método utiliza los procesadores de manera más eficiente en el ambiente de multiprocesador de la CRAY Y-MP. Se diseñó un algoritmo paralelo aplicado únicamente a una arquitectura SIMD de memoria distribuida. En un sistema de memoria distribuida, la ejecución es afectada por el costo de comunicación entre procesadores. El algoritmo reduce significativamente el costo de comunicación de datos. Consecuentemente, sus resultados muestran que la ejecución en el sistema MasPar MP-2 es mucho mejor que el de un solo procesador CRAY Y-MP en problemas de tamaño grande. El algoritmo ha sido aplicado a una arquitectura SMP MIMD (ocho procesadores CRAY Y-MP) y puede adaptarse a arquitecturas de memoria distribuida MIMD.

En el ámbito de los algoritmos paralelos que utilizan programación dinámica para resolver este problema, no se han explorado diferentes formas de particionar la matriz que logre una distribución eficiente entre procesadores.

Normalmente las secuencias de RNA son grandes en longitud y los algoritmos conocidos para resolver parcialmente este problema tienen complejidad $O(n^3)$ y $O(n^4)$ [3] . Estos algoritmos son suficientes para secuencias cortas, sin embargo, para secuencias largas el tiempo de ejecución es demasiado lento para utilizarse de manera efectiva. Ambos algoritmos son recursivos, es decir, hacen uso de funciones de recurrencia para encontrar la mínima energía de la molécula de RNA, aplicando apropiadamente la estrategia de programación dinámica. Esta estrategia está dirigida a los problemas de optimización que dependen de soluciones óptimas anteriores (dependencias) para encontrar una nueva solución óptima y necesita de un algoritmo recursivo que encuentre estas soluciones una y otra vez, ahorrándose el recálculo al guardar las soluciones en una tabla. De acuerdo a las ecuaciones de recurrencia, que estos algoritmos usan para encontrar la mínima energía, se origina una triangular superior (Figura 1); para evitar las dependencias entre las soluciones, el almacenamiento se realiza por diagonales iniciando desde la diagonal 1 hasta la diagonal n. Como sólo los elementos de la diagonal son independientes

entre sí, se puede aplicar eficientemente el paralelismo, reduciendo así el tiempo de ejecución de estos algoritmos de manera razonable.

El objetivo principal de este trabajo es diseñar y construir algoritmos paralelos para resolver el problema de RNA, que tengan un comportamiento aceptable al aplicarse a moléculas grandes.

Se han explorado diferentes formas de dividir las diagonales de la triangular superior de la tabla de la Figura 1 entre los procesadores y se han obtenido resultados experimentales aceptables que reducen el tiempo de ejecución para moléculas de gran tamaño [4]. A esta división de diagonales se le ha dado el nombre de "estrategias de particionamiento" y se ha aplicado a los algoritmos descritos en [3], se diseñaron tres (por diagonales, por bloques homogéneos y por bloques no homogéneos) y cada uno presenta diferentes características.

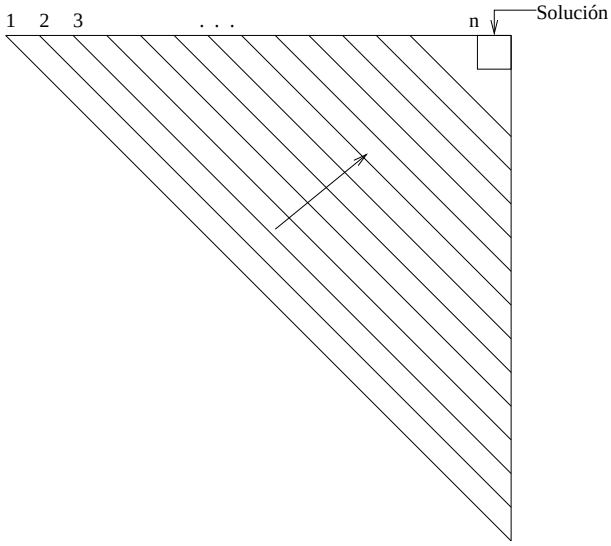


Figura 1: Tabla de Programación Dinámica para el problema del RNA.

Al analizar las operaciones que requiere cada entrada de la tabla de programación dinámica, se observó que sólo los elementos que están en la diagonal son independientes entre sí, lo que facilita la distribución entre procesadores, como ya se mencionó anteriormente. El particionamiento más común (diagonales) que distribuye la diagonal entre los procesadores, provoca un número elevado de comunicaciones y sincronización, lo que originó que se procesaran bloques de diagonales de igual tamaño (bloques homogéneos) reduciendo así considerablemente el tiempo de ejecución de los algoritmos secuenciales de complejidad $O(n^3)$ y $O(n^4)$. Finalmente tomando en cuenta que el tiempo de cálculo de cada entrada de la tabla de programación dinámica era desigual, se ideó otro tipo de particionamiento, por bloques no homogéneos, que consiste en determinar en tiempo de ejecución la longitud del siguiente bloque a procesar, pero requiere más operaciones de sincronización.

El resto del documento está organizado de la siguiente manera: en el capítulo 1 se hace una revisión de los conceptos necesarios para introducir el problema del RNA, se describen los dos algoritmo secuenciales ($O(n^3), O(n^4)$) y se muestran los trabajos relacionados con el problema de RNA . En el capítulo 2 se definen conceptos de paralelismo, tales como los sistemas de mul-

tiprocesadores, multicomputadoras, sincronización, mapeo, calendarización, etc. Se muestran también, los diferentes tipos de particionamiento aplicados a la tabla de programación dinámica para los algoritmos descritos en [3]. En el capítulo 3 se muestran los resultados experimentales obtenidos para memoria compartida y memoria distribuida, para estos dos algoritmos. Finalmente se incluyen las conclusiones finales de este trabajo de tesis y la bibliografía.

Capítulo 1

El problema del RNA

En este capítulo se describe la teoría fundamental acerca del problema de doblamiento del ácido ribonucleico. Posteriormente se presenta una solución parcial a este problema de manera analítica y experimental.

1.1 Conceptos básicos del problema del RNA

Una molécula de ácido ribonucleico (RNA) está constituida por una gran cantidad de subunidades (ribonucleicos) ligadas. Cada ribonucleico contiene una de cuatro posibles bases: adenanina (A), citosina (C), guanina (G) y uracil (U), y esta secuencia de bases distingue un tipo de RNA de otro y se le conoce como *estructura primaria* de la molécula de RNA.

Bajo condiciones normales, un ribonucleótido cambia al doblarse y las bases formarán uniones con otros formando un patrón más complicado; de esta forma las moléculas conforman ciclos. Así que la conformación y el patrón de unión es llamada la *estructura secundaria* de la molécula de RNA.

Las interacciones base a base que forman la estructura secundaria de RNA son de dos tipos, normalmente, hidrógeno enlazado entre una A y una U, e hidrógeno enlazado entre una C y una G. Una molécula es una secuencia o cadena $a = a_1...a_n$, donde cada a_i es una A, G, C o U. La cadena o secuencia a es llamada la *estructura primaria*. Un *par* (enlace) $a_i \cdot a_j$ debe cumplir que $i < j$. La *estructura secundaria* S para a es un conjunto de pares, el cual debe satisfacer las siguientes restricciones:

1. Watson-Crick: Si S contiene $a_i \cdot a_j$, entonces a_i y a_j son A y U, o U y A, o C y G, o G y C.

2. Sin traslape de pares. Si S contiene al par $a_i \cdot a_k$, entonces no puede contener $a_i \cdot a_j$ (con $j \neq k$) o $a_j \cdot a_k$ (con $j \neq i$).

3. Sin nudos. Si $h < i < j < k$, entonces S no puede contener a ambos pares $a_h \cdot a_j$ y $a_i \cdot a_k$.

4. Sin vueltas cerradas. Si S contiene el par $a_i \cdot a_j$, entonces $|j - i| \geqq 4$

Estas limitaciones restringen todas las posibles formas de las estructuras secundarias que podemos obtener a partir de la estructura primaria, vea la Figura 1.1. Las limitaciones 2 y 3 pueden ocasionalmente llegar a violarse, pero los resultados "tríos", "nudos", etc., se consideran características de estructuras de un nivel más alto (estructuras terciarias). La limitación 1 también puede llegar a violarse por la presencia de pares que no son Watson-Crick, como $G \cdot U$, pero no se pierde generalidad al ignorar a este par.

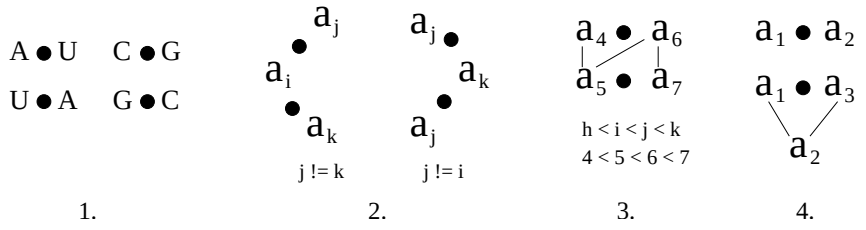


Figura 1.1: Restricciones.

Para denotar al par $a_i \cdot a_j$ se utilizará como notación $i \cdot j$ en definiciones posteriores. Si $i \cdot j$ es un par y $i < r < j$, podemos decir que el par $i \cdot j$ *rodea* a r. Para analizar la estructura secundaria iniciaremos con la notación y terminología que se describe a continuación. La *cadena* desde i a j se escribe como $[i, j]$ y significa la secuencia $i, i+1, ..., j-1, j$. Una cadena es *propia* si, para cada elemento apareado en la cadena, su pareja está también en la cadena. La cadena $[i, j]$ se dice *cerrada* si i está apareado con j.

Suponga que $[i, j]$ es una cadena propia. Un par $p \cdot q$ o un elemento r no apareado en la cadena se dice que son accesibles en $[i, j]$ si no están rodeados por algún otro par en $[i, j]$ excepto posiblemente por $i \cdot j$. Si i y j son apareados, podemos decir que $p \cdot q$ o r es accesible desde $i \cdot j$.

Algún par $i \cdot j$ define un *ciclo* s desde i a j: Si s consiste del par cerrado $i \cdot j$ junto con otros pares $p_1 \cdot q_1$, $p_2 \cdot q_2$, ... accesibles desde $i \cdot j$ y algunos elementos no apareados accesibles desde $i \cdot j$. Si s contiene k pares (incluyendo pares cerrados), se dice que es un $k - ciclo$ o que tiene orden k. Cada par está contenido en una o dos vueltas, pero no en más. Específicamente, cada par está contenido en la vuelta que lo define, y en muchos casos un par es también accesible en otra vuelta. Dado que un ciclo puede tener muchos pares accesibles pero únicamente un par cerrado, la accesibilidad no es simétrica, es decir, muchos pares pueden ser accesibles desde un par $i \cdot j$, pero un par $i \cdot j$ es accesible desde a lo más otro par.

1.1.1 Tipos de ciclos

Las estructuras secundarias del RNA pueden tener muchas formas diferentes de acuerdo a todos los posibles ciclos que se pueden formar. Para cada estructura secundaria S podemos tener diferentes tipos de ciclos (Figura 1.2) y estos son:

1. Si S contiene el par $i \cdot j$ pero ninguno de sus elementos rodeados $i+1, ..., j-1$ son apareados, entonces el ciclo formado es una vuelta muy cerrada (hairpin loop).

2. Si S contiene $i \cdot j, (i+1) \cdot (j-1), ..., (i+h) \cdot (j-h)$ cada uno de estos pares excepto el último, es decir, encimado sobre el siguiente par, entonces, dos pares consecutivos pueden ser referenciados como pares encimados o como un ciclo de pares encimados (stacked pairs).

3. Si $i+1 < p < q < j-1$ y S contiene $i \cdot j$ y $p \cdot q$, pero los elementos entre i y p y entre q y j no están apareados, entonces las dos regiones no apareadas constituyen una vuelta interior (interior loop).

4. Si S contiene $i \cdot j$ y $(i+1) \cdot q$, y existen algunos elementos no apareados entre q y j, estos elementos no apareados forman un bulto (bulge). Simétricamente, un bulto también ocurre si S contiene $i \cdot j$, $p \cdot (j-1)$ y algunos elementos no apareados entre i y p

5. Si S contiene $i \cdot j$ e $i \cdot j$ rodea a dos o más pares $p \cdot q$, $r \cdot s$, ... a los cuales no los rodea otro par, entonces se forma una vuelta múltiple (multiple loop).

6. Si r es no apareado y no hay pares en S rodeando a r entonces r es una región externa sola o abandonada (single-stranded regions).

Estos tipos de ciclos se muestran en las Figuras 1.2, 1.3 y 1.4.

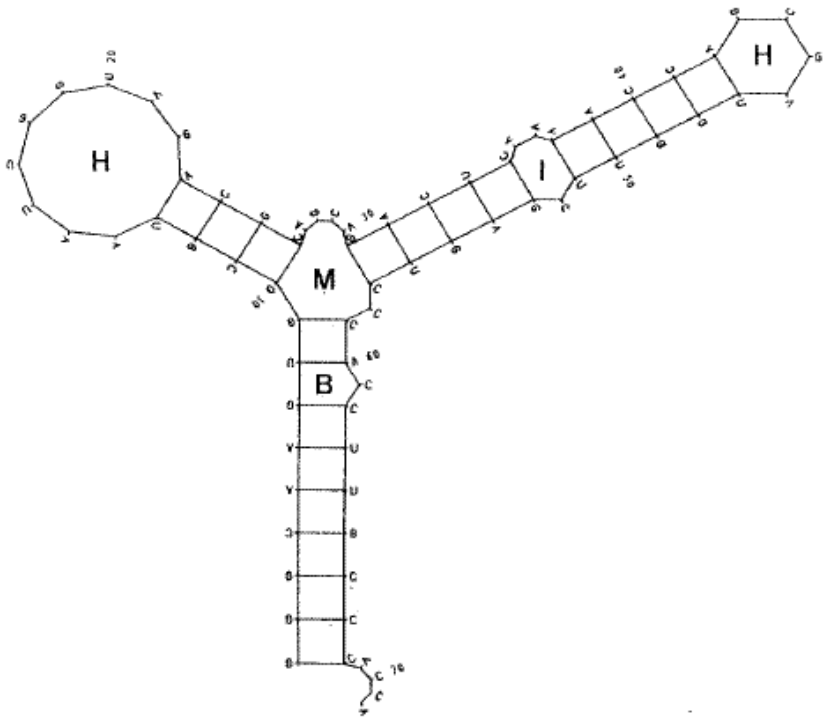


Figura 1.2: Ejemplo de estructura secundaria y tipos de ciclos (H-hairpin loop, I-interior loop, B-bulge, M-multiple loop).

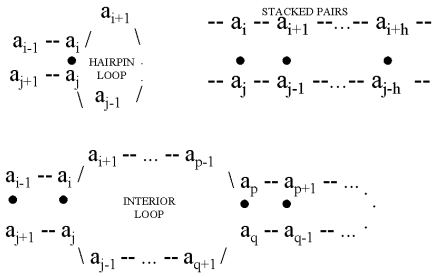


Figura 1.3: Tipos de Ciclos.

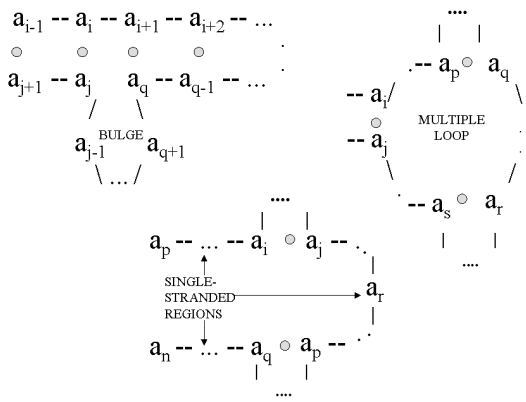


Figura 1.4: Tipos de Ciclos.

Con esto podemos notar que un ciclo que contiene bases no apareadas es una vuelta, y que el único tipo de ciclo que no tiene vueltas es un ciclo de pares encimados.

Sea $[i, j]$ una cadena, y sean p y q tales que $i + 1 \leq p$, $p < q$ y $q < j - 1$. Un ciclo de orden k puede clasificarse como sigue:

i. Si $k = 1$, el 1-ciclo es una vuelta muy cerrada (hairpin).

ii. Si $k = 2$ y $(i + 1) \cdot (j - 1)$ es un par accesible, entonces el 2-ciclo es un ciclo de par encimado (stacked pair).

iii. Si $k = 2$ y $p \cdot q$ es un par accesible, entonces el 2-ciclo es una vuelta interior (interior loop).

iv. Si $k = 2$ y $(i+1) \cdot q$ o $p \cdot (j-1)$ es un par accesible, entonces el 2-ciclo es un bulto (bulge).

v. Si $k \geqq 3$, entonces el k-ciclo es una vuelta múltiple.

1.1.2 Tipos de representación

Existen diferentes formas de representar una estructura secundaria, esto puede resultar más útil que sólo mostrar un conjunto de pares.

Sólo se indican tres tipos de representación:

1. *Pictórica o normal.* La cadena de RNA es representada por líneas curveadas conectando puntos equidistantes. En la Figura 1.2 se muestra un ejemplo de este tipo de representación.

2. *Círculo.* Este tipo de representación fue introducido por Nussinov *et al.* (1978) [31]. Las bases de la molécula de RNA son colocadas equidistantes una de la otra a lo largo de la circunferencia de un círculo. Los enlaces covalentes de las bases son representados por arcos del círculo. En la Figura 1.5 las bases representan los vértices y los arcos son los enlaces covalentes y las caras son el conjunto de todos los k-ciclos.

3. *Arbol.* Cada par de la estructura secundaria se representa por un 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. Pero se pierde información al usar esta información como bases no apareadas que están en los ciclos o regiones externas y la orientación de la molécula, es decir, en donde empieza y en donde termina, vea la Figura 1.6.

Los tres tipos de representación permiten una útil clasificación de estructuras de acuerdo a su complejidad.

1.2 Descripción del problema del RNA

Para analizar la estructura secundaria S, necesitaremos considerar estructuras sobre cadenas que son menores que la secuencia total $[1, n]$. Una estructura secundaria sobre $[i, j]$ la indicaremos como S_{ij}, así que S podría ser designado como S_{1n}

Dada la estructura primaria de un RNA, podemos imaginar un gran número de diferentes estructuras secundarias basadas en parejas de A's con U's y parejas de C's con G's de diferentes formas. De acuerdo a las leyes de la termodinámica, sin embargo, sólo una estructura secundaria será estable, normalmente, la estructura secundaria que optimice la energía libre. Un ejemplo de este comportamiento se muestra en la Figura 1.7 donde a partir de una estructura primaria obtenemos tres tipos diferentes de estructuras secundarias con diferente valor de energía.

Así el problema a resolver es dada una estructura primaria de RNA, encontrar, de todas las estructuras secundarias aquella que optimice la energía asociada a ella.

Experimentalmente, es costoso y complicado tratar de obtener la estructura secundaria de una secuencia de RNA; se ha usado difracción de rayos-x del RNA cristalino y otras técnicas bioquímicas. Por tanto, es altamente necesario el disponer de un método para deducir la estructura secundaria estable directamente desde la estructura primaria. Tal método requiere de dos partes:

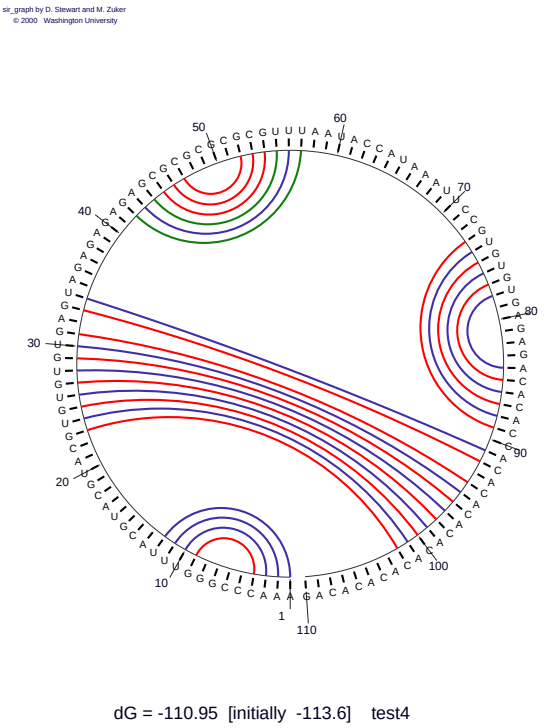


Figura 1.5: Representación de círculo.

1. Un método para especificar la energía libre de alguna estructura secundaria propuesta.

2. Un algoritmo razonablemente rápido que encuentre la estructura secundaria óptima.

Este trabajo de tesis está fundamentado en los algoritmos descritos en [3] que utilizan la estrategia de programación dinámica con diferentes propiedades. Estos algoritmos están basados completamente en un análisis riguroso del problema de "múltiples ciclos" en estructuras secundarias. Estos algoritmos son de complejidad en tiempo $O(n^3)$ y $O(n^4)$, donde n es la longitud de la secuencia de RNA (conjunto de bases) los cuales permiten identificar estructuras secundarias de orden 1 y 2 y de orden 1, 2 y 3 respectivamente.

Los algoritmos se basan en la teoría de la hipótesis de Tinoco-Unlebeck, que indica que la energía puede ser expresada como una suma de términos donde cada término está asociado con uno de los ciclos s_r.

$$E(S) = e(s_1) + e(s_2) + ... + e(s_t)$$

1.3 Algoritmo $O(n^3)$

El algoritmo $O(n^3)$, llena una matriz triangular superior aplicando una ecuación de recurrencia que se mostrará posteriormente. Este llenado se realiza por medio de la técnica de

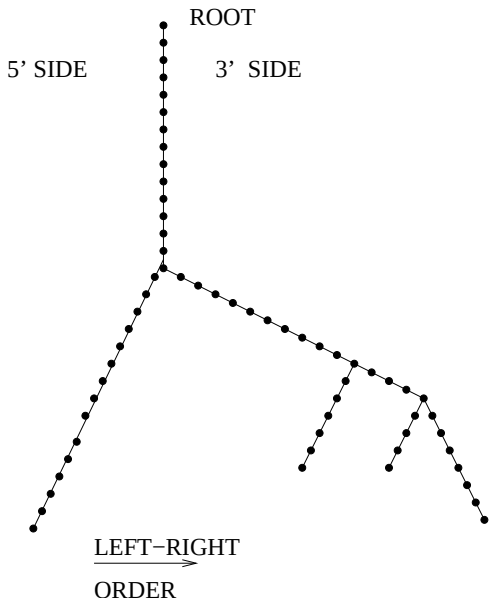


Figura 1.6: Representación de árbol del virus del mosaico de la Coliflor [10].

programación dinámica en la cual se descompone el problema (cadena de bases) en un número más pequeño de subproblemas (subcadenas) y así sucesivamente hasta lograr obtener soluciones triviales. Cada entrada en la tabla de programación dinámica corresponde a un subproblema (valor de la energía de la subcadena).

Para determinar el valor de cada elemento de la tabla es necesario determinar si los extremos del conjunto de bases que en ese momento se está calculando es cerrada o no, es decir, si estos elementos son:

$$\begin{array}{ccc} \mathrm{U}\cdot\mathrm{A} & \mathrm{o} & \mathrm{C}\cdot\mathrm{G} \\ \mathrm{A}\cdot\mathrm{U} & & \mathrm{G}\cdot\mathrm{C} \end{array}$$

En tales casos se forma un ciclo en la estructura de la molécula de RNA, ya mencionado anteriormente. Para analizar una estructura secundaria S, se necesita considerar estructuras en cadenas que son menores que la secuencia total $[1, n]$. Una estructura secundaria sobre $[i, j]$ se indica por S_{ij}, así que S puede indicarse como S_{1n}. Una estructura secundaria S se dice que *induce* a una estructura en alguna cadena propia $[i, j]$: si la estructura inducida está definida al contener todos los pares en S cuyos elementos pertenecen a $[i, j]$.

Se define $F(i, j)$ como el valor óptimo de energía (E) en $[i, j]$, es decir:

$$F(i,j) = \min_{\substack{S_{ij} \\ i\cdot j}} E(S_{ij}) \tag{1.1}$$

donde S_{ij} se extiende sobre todas las posibles estructuras secundarias en $[i, j]$. Se dice que (i, j) es cerrada si es posible formar un ciclo con $i \cdot j$ como par cerrado, lo que significa que especialmente a_i y a_j cumplen la limitación de Watson-Crick y que $j - i \geq 4$. Si (i, j) es cerrada, sea $C(i, j)$ el valor óptimo E (energía) dado que i esta apareado con j, es decir,

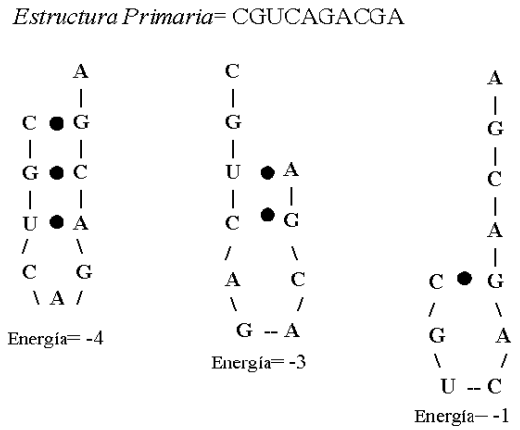


Figura 1.7: Ejemplos de Estructuras Secundarias.

$$C(i,j) = \min_{\substack{S_{ij} \\ i \cdot j}} E(S_{ij}) \tag{1.2}$$

donde S_{ij} se extiende sobre todas las posibles estructuras secundarias en $[i,j]$ que aparean i con j. Note que "C" (*cerrada*) está asociada con una cadena cerrada, mientras que F está asociado con alguna cadena propia.

A partir de la ecuación 1.1, $F(i,j)$ se puede expresar como:

$$F(i,j) = \min \begin{cases} 0 & j - i < 4 \\ C(i,j) & \text{si } [i,j] \text{ es cerrada} \\ \min\limits_{i \leqslant h < j} [F(i,h) + F(h+1,j)] & \text{otra forma} \end{cases} \tag{1.3}$$

El primer caso se da cuando se viola la restricción de vueltas cerradas, es decir, $F(i,j) = 0$. El segundo caso es cuando i, j forman un par cerrado, entonces se llama a la función de recurrencia $C(i,j)$, (ecuación 1.4). Si las dos primeras partes no se cumplen, es decir $j - i \geqslant 4$ y no es un par cerrado, entonces se realiza el tercer caso el cual consiste en dividir a la cadena de bases en posibles subcadenas cumpliendo la relación $i \leqslant h < j$ y se toma de éstas la que tenga menor costo.

La función $F(i,j)$ obtiene la mínima energía que se pueda lograr en la secuencia $[i,j]$. El objetivo es encontrar $F(1,n)$ y la estructura secundaria S^{opt} que la obtenga, i, j deben cumplir la relación $i < j$.

Si (i,j) es cerrada se origina la ecuación a partir de la ecuación .

$$C(i,j) = \min_{k \geqq 1} \min_{\substack{\text{s es un} \\ \text{k-ciclo} \\ \text{desde i a j}}} \left[e(s) + \sum_{(p_h - q_h)} C(p_h, q_h) \right] \tag{1.4}$$

Esta función de recurrencia se aplica cuando i y j forman un par cerrado. Esta función toma la energía asociada a este par $(e(s))$ más la energía que se puede obtener en la subsecuencia $[i+1, j-1]$, como se muestra en la siguiente ecuación.

$$C(i,j) = e(s) + F(i+1, j-1) \tag{1.5}$$

La función $e(s)$ determina el valor de energía del par cerrado $i \cdot j$ (vea la ecuación 1.6) y favorece los enlaces de A's con U's (o U's con A's). Como el problema es de minimización le asocia un valor de energía de -2, a los enlaces de C's con G's (o G's con C's), $e(s)$ toma un valor de -1, en caso contrario toma un valor de 0,

$$e(s) = \begin{cases} -2 & a_i = A \vee a_i = U \text{ y } a_j = A \vee a_j = U, a_i \neq a_j \\ -1 & a_i = C \vee a_i = G \text{ y } a_j = C \vee a_j = G, a_i \neq a_j \\ 0 & \text{en otro caso} \end{cases} \tag{1.6}$$

Las ecuaciones (1.3) y (1.5) únicamente dan la energía óptima de la secuencia S ($E(S^{opt})$), no la estructura óptima S^{opt}. El algoritmo $O(n^3)$ es simplemente una implementación directa de estas dos ecuaciones. Dichas ecuaciones dan origen a una tabla de programación dinámica con forma triangular superior. Para determinar el valor de energía de cada posición de la tabla, se necesita del i-ésimo renglón y de la j-ésima columna al aplicar la ecuación 1.3 a la secuencia S_{ij}. Esto se puede observar en la Figura 1.8.

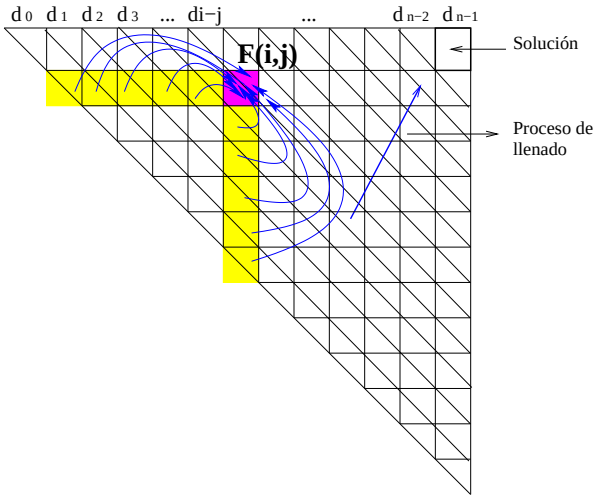


Figura 1.8: Evaluación de $F(i,j)$.

Analizando las ecuaciones se determina que un proceso de llenado adecuado procede por diagonales, a partir de la principal, hasta llegar a la esquina superior derecha de acuerdo a la Figura 1.8. Dado que existen $\dfrac{n(n+1)}{2}$ entradas en la tabla y, en el peor caso, el cálculo de una entrada toma un tiempo $O(n)$, el algoritmo tiene una complejidad en tiempo $O(n^3)$.

Para cadenas de longitud grande el tiempo es elevado, como puede verse en la Figura 1.9, en donde se presenta el tiempo de ejecución de un programa secuencial para diferentes tamaños

de problema. En la Figura 1.9 se muestra el tiempo de ejecución obtenido por diferentes tamaños de problema y la gráfica del polinomio de aproximación generado a partir del tiempo de ejecución y del método de mínimos cuadrados. Las constantes del polinomio de aproximación ($P(x_i) = a_0 + a_i x_i + a_i x_i^2 + a_3 x_i^3$, x_i es la longitud de la secuencia) son:

$a_3 = 2.09960219509183e - 10$
$a_2 = 2.22982643412384e - 07$
$a_1 = -0.000313281501670941$
$a_0 = 0.0898191780296394$

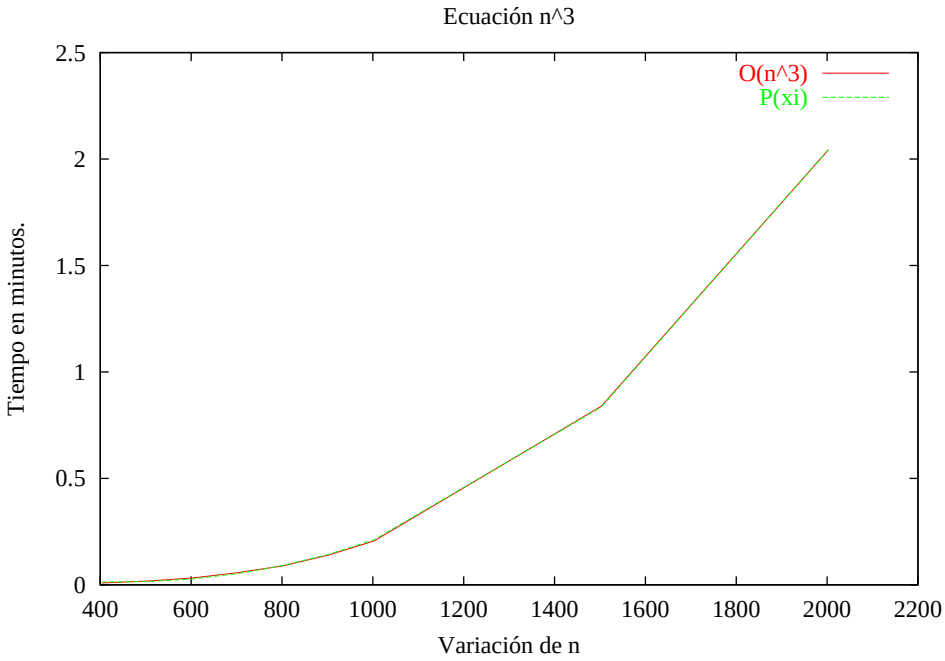


Figura 1.9: Tiempos de ejecución del algoritmo $O(n^3)$ secuencial para el problema del RNA para diferentes tamaños de problema, tiempo en minutos..

Esto nos indica que al evaluar cualquier longitud de secuencia (x_i), se puede tener una estimación del tiempo de ejecución del programa secuencial del algoritmo $O(n^3)$. Por ejemplo, para $x_i = 3000$ bases el tiempo estimado es 6.85 minutos. Este algoritmo tiene la restricción de no considerar todos los tipos de estructuras secundarias que se pueden formar a partir de una estructura primaria.

1.4 Algoritmo $O(n^4)$

Existe otro algoritmo de complejidad en tiempo $O(n^4)$ [3] que también se aplica al problema de múltiples ciclos. Este algoritmo considera otros tipos de estructuras secundarias de orden

$k = 2$ y $k = 3$, pero sólo consideraremos ciclos de orden $k = 1$ y $k = 2$. El valor de $e(s)$ para ciclos s de orden 1 y 2 es:

$$e(s) = \begin{cases} f_1(i,j) & \text{Si } s \text{ es de orden 1} \\ f_2(i,j,p_1,q_1) & \text{Si } s \text{ es de orden 2} \end{cases} \tag{1.7}$$

Normalmente f_1 y f_2 dependen de la identidad de las bases apareadas y del número de bases no apareados, pero pueden ser funciones arbitrarias, y éstas se definen en las ecuaciones 1.8 y 1.9.

$$f_1(i,j) = \begin{cases} c_2 & \text{Si } a_i, a_j \in \{A,U\} \text{ y } a_i \neq a_j. \\ c_1 & \text{Si } a_i, a_j \in \{C,G\} \text{ y } a_i \neq a_j. \\ c_0 & \text{en otro caso} \end{cases} \tag{1.8}$$

$$f_2(i,j,p_1,q_1) = \begin{cases} e(i,j) & \text{si } q_1 - p_1 < 3 \\ -2 + e(i,j) + e(p1,q1) & \text{Si par orden 2} \\ e(i,j) & \text{Si par orden 1} \end{cases} \tag{1.9}$$

La ecuación 1.8 se aplica cuando existen ciclos de orden 1, es decir cuando se forma un par en $i \cdot j$ (par exterior) y en $[i+1, j-1]$ no existen pares cerrados (pares interiores). Dependiendo del tipo de enlace f_1 puede tomar el valor de c_2 si $a_i = A$ o U y $a_j = U$ o A ($a_i \neq a_j$), o el valor de c_1 si $a_i = C$ o G y $a_j = G$ o C ($a_i \neq a_j$) o c_0 en caso contrario. La segunda ecuación (1.9) se aplica cuando existen ciclos de orden 2, es decir cuando existe un par exterior en (i,j) y un par interior en la secuencia $[i+1, j-1]$ $(p_1 \cdot q_1, i < p_1 < q_1 < j)$. La función f_2 toma el valor de energía de $e(i,j)$ (ecuación 1.6) si la longitud de la base interior es menor que 3 ($q_1 - p_1 < 3$) o si no existen pares interiores en $[i,j]$ (ciclo de orden 1), o toma el valor de energía del par interior ($p_1 \cdot q_1$) más el valor de energía de par exterior ($i \cdot j$) más un valor constante (-2) si se formó un ciclo de orden 2.

Sea $G(i,j)$ la energía mínima de todo el conjunto de ciclos de orden 1 y 2, es decir, la función G determina si existe un ciclo de orden 1 en la secuencia S_{ij} y busca ciclos de orden 2 al llamar a la función G_1. Si existe un par base exterior en S_{ij}, (par cerrado en i y j), entonces busca en todo el conjunto de subsecuencias de la secuencia $S_{(i+1)(j-1)}$ un par base cerrado interior ($a_{p_1} \cdot a_{q_1}$) que cumpla la relación $i \leq p_1 < q_1 < j$.

Este algoritmo utiliza la ecuación $G(i,j)$ para calcular cada valor de la tabla de programación dinámica. La siguiente ecuación indica que se revisarán todos las secuencias de S_{ij} de orden 1 $k = 1$ (ciclos de orden 1) y superiores $k > 1$ (ciclos de orden 2).

$$G(i,j) = \min_{k \geq 1} \min_{S_{ij}} E(S_{ij})$$

Para calcular $G(i,j)$ se utiliza la función $G_1(i,j)$ que hace lo mismo que G excepto que la estructura S_{ij} contiene por lo menos un par accesible (note el cambio de $k \geq 1$ a $k \geq 2$) en la siguiente ecuación:

$$G_1(i,j) = \min_{k \geq 2} \min_{S_{ij}} E(S_{ij})$$

donde S_{ij} tiene $k-1$ pares accesibles.

Para $j - i \leq 4$, $G(i,j)$, $G_1(i,j)$ y $C(i,j)$ se pueden expresar mediante las siguientes ecuaciones de recurrencia 1.10, 1.11 y 1.12.

$$G(i,j) = \min \begin{cases} G_1(i,j) \\ \min\limits_{i \leqslant h < j} [G(i,h) + G(h+1,j)] \end{cases} \tag{1.10}$$

En esta función, el primer caso se da cuando se buscan ciclos de orden 1 y 2 al llamar a la función $G_1(i,j)$. El segundo caso consiste en dividir a la cadena de bases en posibles subcadenas cumpliendo la relación $i \leqslant h < j$, se toma de éstas la que tenga menor costo. De los dos casos se toma el valor mínimo.

$$G_1(i,j) = \min \begin{cases} C(i,j) \\ \min\limits_{i \leqslant h < j} [G_1(i,h) + G(h+1,j), G(i,h) + G_1(h+1,j))] \end{cases} \tag{1.11}$$

El primer caso, en la función G_1, se llama a la función $C(i,j)$ que explora los ciclos de orden 1 y de orden 2. El segundo caso realiza traslapes al tomar los valores de las tablas G y G_1. Se divide la cadena de bases en posibles subcadenas que cumplan la relación $i \leqslant h < j$, y se toma la que tenga menor costo. De los dos casos se toma el mínimo valor de energía.

Sea $C(i,j)$ el valor óptimo de la secuencia $[i,j]$ que determina la existencia de ciclos de orden 1 y de orden 2. Si $[i,j]$ es cerrada en el par (i,j), $C(i,j)$ revisa todas las posibles subcadenas que originen ciclos de orden 2. Esta función se define a continuación:

$$C(i,j) = \begin{cases} \min \begin{cases} f_1 \\ \min\limits_{i < p_1 < q_1 < j} [f_2(i,j,p_1,q_1) + C(p_1,q_1)] \\ \min\limits_{i+1 \leqslant h < j-1} [G_1(i+1,h) + G_1(h+1,j-1)] + e(i,j) \end{cases} & \text{si } (i,j) \text{ es cerrada} \\ indefinido & \text{en otro caso} \end{cases} \tag{1.12}$$

En caso de que la secuencia $[i,j]$ sea cerrada se toma el valor mínimo de tres subcasos (primer caso). De lo contrario se realiza el segundo caso que consiste de un valor indefinido que no altera el proceso de minimización. Si $[i,j]$ es cerrada se revisan tres subcasos, el primero calcula el valor de energía la función f_1 (ciclos de orden 1), el segundo subcaso revisa que existan pares interiores al dividir a la cadena de bases en posibles subsecuencias que cumplan la relación $i < p_1 < q_1 < j$, llama a la función f_2 si existió un par interior $p_1 \cdot q_1$ (orden 2) y obtiene la de menor energía. El tercer subcaso divide a la cadena $[i+1, j-1]$ en posibles subsecuencias adicionándoles el valor de energía del par exterior $i \cdot j$ y toma de todas éstas la de menor energía. De los tres subcasos, $C(i,j)$ recibe el mínimo valor.

En la tercera línea de la ecuación 1.12 se hace que este algoritmo use un tiempo proporcional a $O(n^4)$. Las funciones f_1, f_2 logran que se analicen ciclos de orden 1 y 2. Si se desea explorar

estructuras de orden 3 se tendría que adicionar otra función arbitraria f_3 [3], pero la complejidad del algoritmo aumentaría a $O(n^6)$.

Este algoritmo al igual que el $O(n^3)$ utiliza la Hipótesis de Tinoco-Unlebeck. La energía asociada ($e(s)$) a cada par apareado toma el mismo valor como se definió en la ecuación (1.6). El comportamiento de estas ecuaciones (dependencias) se muestra en la Figura 1.10 a) - d). La Figura 1.10a) muestra el comportamiento de la función $G(i,j)$ del segundo caso al buscar todas las posibles subsecuencias que cumplan la relación $i \leq h < j$. La Figura 1.10b) es la aplicación del segundo caso de la función G_1 que traslapa las tablas G y G_1 para encontrar la subsecuencia mínima que cumpla la relación $i \leq h < j$. La Figura 1.10c) muestra el comportamiento del segundo caso de la función C cuando (i,j) es cerrada y se busca otro par interior $(p_1 \cdot q_1)$ que cumpla la relación $i < p_1 < q_1 < j$. La última figura, 1.10d), busca una subcadena óptima en la tabla G_1 que cumpla la relación $i+1 \leq h < j-1$, al buscar en todas las posibles sucadenas. Note que para obtener el valor de (i,j) de las tablas de la Figura 1.10 se necesitan de valores anteriores como en el algoritmo $O(n^3)$ y que sólo los elementos de la diagonal son independientes entre sí.

En la Figura 1.11 se muestra el tiempo de ejecución obtenido por diferentes tamaños de problema y la gráfica del polinomio de aproximación generado a partir del tiempo de ejecución y del método de mínimos cuadrados. Las constantes del polinomio de aproximación ($P(x_i) = a_0 + a_i x_i + a_i x_i^2 + a_3 x_i^3 + a_4 x_i^4$) son:

$a_4 = 2.30664676005088e-11$
$a_3 = 6.96502958140168e-09$
$a_2 = -1.36587909915324e-05$
$a_1 = 0.0072707709085462$
$a_0 = -1.22123110970753$

1.5 Técnicas de optimización

El problema para encontrar la estructura de RNA es un problema de optimización. Aquí se han descrito dos algoritmos que resuelven el problema para casos especiales. Para el problema general se puede usar estrategias de optimización tales como: recocido simulado, algoritmos genéticos,"branch and bound", etc.

A continuación se presenta una breve introducción de dos estrategias de optimización: programación dinámica y algoritmos genéticos los cuales serán de utilidad para entender la sección 1.6.

Programación dinámica

La programación dinámica, como el método "divide y vencerás", soluciona problemas al combinar soluciones a subproblemas. Los algoritmos del tipo divide y vencerás particionan el problema en subproblemas independientes, solucionándolos recursivamente, y luego combinan

sus soluciones para resolver el problema original. Por el contrario, la programación dinámica se aplica cuando los subproblemas no son independientes, es decir, cuando los subproblemas comparten subproblemas. Cuando se aplica un algoritmo de "divide y vencerás" sobre problemas que tienen subproblemas no independientes trabaja más de lo necesario, repetidamente solucionando los subsubproblemas comunes. Un algoritmo de programación dinámica soluciona cada subsubproblema sólo una vez y almacena esta respuesta en una tabla ahorrándose el recálculo cada vez que estos subsubproblemas sean encontrados.

La programación dinámica es en general una estrategia de solución que se aplica principalmente a problemas de optimización. Ha sido aplicada en campos tan diversos como teoría de control, investigación de operaciones, biología y ciencias de la computación. Algoritmos paralelos basados en modelos PRAM han sido propuestos para el problema de la parentización óptima de matrices. Este problema ha sido el favorito de muchos investigadores porque es isomorfo a otros problemas tales como: el problema de triangulación óptima de polígonos convexos y el problema de la construcción óptima de búsqueda en árboles binarios [1].

La estrategia de programación dinámica descompone un problema en un número más pequeño de subproblemas, y éstos a su vez son descompuestos en subproblemas hasta lograr obtener soluciones triviales. Cada entrada en la tabla de programación dinámica corresponde a un subproblema. Entonces, el tamaño de la tabla es el número total de subproblemas incluidos en el mismo problema. Esta estrategia es muy eficiente al involucrar la solución de subproblemas comunes varias veces.

Hay dos ingredientes claves que un problema de optimización debe tomar en cuenta para que la programación dinámica se pueda aplicar y sea efectiva. El primero consiste en que la solución óptima al problema debe contenerse dentro de las soluciones óptimas de los subproblemas. El segundo es que un algoritmo recursivo encuentre el mismo subproblema una y otra vez. La programación dinámica soluciona cada subproblema una vez y almacena estos resultados intermedios en una tabla donde pueden tomarse cuando se necesiten. El desdoblamiento del problema del ácido ribonucleico (RNA) que encuentra una estructura secundaria con mínima energía libre, tiene estas dos características. Por lo que, el orden de cómputo y la colocación de los datos pueden afectar la ejecución del problema del RNA así como otros problemas que usan programación dinámica en arquitecturas paralelas.

Algoritmos genéticos

Un Algoritmo Genético (AG) es un procedimiento de optimización no determinista (Holland, 1975 [19]). Se derivó de los conceptos de la evolución biológica usando operadores basados en la génetica y en el principio de supervivencia del más apto, tales como, cruza, mutación y selección para imitar el proceso evolutivo. Los algoritmos genéticos se han aplicado en campos tan diversos como la ciencia y la ingeniería (Goldberg, 1989 [20]; Holland, 1992 [21]; Miettinen *et al.*, 1999 [22]) y han sido desarrollados para desdoblar secuencias de RNA (Shapiro and Navetta, 1994 [23]).

Los AG fueron introducidos por *John H. Holland* a principios de los 60s y pueden considerarse la técnica evolutiva más popular en la actualidad. El objetivo de Holland fue el estudio

formal de los procesos de adaptación natural y el traslado de estos mecanismos a la computación. Un algoritmo genético se puede definir como [24]:

> un algoritmo matemático altamente paralelo que transforma un conjunto de objetos matemáticos individuales con respecto al tiempo, usando operaciones modeladas de acuerdo al principio Darwiniano de reproducción y supervivencia del más apto, y tras haberse presentado de forma natural una serie de operaciones genéticas de entre las que destaca la recombinación sexual. Cada uno de estos objetos matemáticos suele ser una cadena de caracteres (letras o números) de longitud fija que se ajusta el modelo de las cadenas de cromosomas, y se les asocia con una cierta función matemática que refleja su aptitud.

Un algoritmo genético consta de los siguientes componentes [25]:

1. Una *representación* de las soluciones del problema. La representación tradicional es la cadena binaria para la codificación de las soluciones del problema.
2. Una forma de crear una población inicial de posibles soluciones (normalmente un proceso aleatorio).
3. Una *función de evaluación* que juega el papel de ambiente, clasificando las soluciones en términos de su aptitud.
4. Un mecanismo de *selección* que permita seleccionar a los individuos de acuerdo a su aptitud.
5. Operadores genéticos (cruza, mutación y elitismo) que alteran la composición de los hijos que se producirán para las siguientes generaciones.
6. *Valores* para los diferentes *parámetros* que utiliza el algoritmo genético (tamaño de la población, probabilidad de cruza, probabilidad de mutación, número máximo de generaciones, etc.)

1.6 Algoritmos implementados en la literatura

Podemos encontrar un sinnúmero de programas que tratan de encontrar la estructura secundaria óptima de la molécula de RNA. Iniciaremos con algunos algoritmos diseñados para representar de manera gráfica la estructura secundaria del RNA y posteriormente indicaremos algunos algoritmos que utilizan alguna estrategia de optimización, como programación dinámica, algoritmos genéticos, etc.

1.6.1 Algoritmos visuales

Se han escrito un sinnúmero de programas de computadora para producir la representación de la estructura normal de la molécula de RNA de entre los cuales podemos citar:

Los programas de computadora de Studnicka *et al.* (1978) [26] y Zuker y Stiegler (1981) [27]. Ambos producen una salida en línea de impresión de una representación normal, lo cual no es satisfactorio para moléculas muy grandes cuya estructura es altamente ramificable. Feldmann (no publicado) ha escrito un programa en SAIL llamado NUCSHO, produciendo una salida en línea de impresión, la cual es muy elegante e involucra traslapes [10]. Osterburg y Sommer (1981) describe un programa, el cual coloca los pares cerrados de un múltiple ciclo igualmente espaciados sobre un círculo. Sin embargo, existen traslapes con este método [28]. Lapalme *et al.* (1982) producen una más agradable salida, [29]. Shapiro *et al.* (1982) usan algunas características de terminales de vídeo para permitir al usuario indicar las regiones que deberían ser rotadas o redibujadas en un tamaño más grande [30].

El paquete de Vienna RNA consiste de una librería con código en C y varios programas para predecir y comparar estructuras secundarias de RNA. Los programas prueban tres tipos de algoritmos de programación dinámica para predecir estructuras: el algoritmo de minimización de energía de Zuker y Stiegler (1981) el cual sólo da una estructura óptima, el algoritmo de la función de partición de McCaskill (1990) que calcula la probabilidad de los pares base en el conjunto termodinámico, y el algoritmo subóptimo de doblado de Wuchty et. al. (1999) que genera todas las estructuras subóptimas en un rango de energía de la energía óptima. Para comparar las estructuras, el paquete contiene varias medidas de distancia. Finalmente, prueba un algoritmo de diseño de secuencias con una estructura predefinida [15]. Todos estos programas trabajan de manera secuencial.

RNAdraw es un algoritmo que utiliza la estrategia de programación dinámica basado en el trabajo de M. Zuker y P. Stiegler y usa parámetros de energía de Turner, Freier y Jaeger [16].

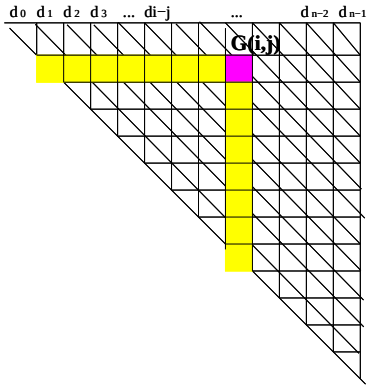
mfold es un paquete de predicción de estructuras secundarias de RNA y DNA que usa reglas termodinámicas de vecinos cercanos, es decir, reglas de energía que son asignadas a ciclos en lugar de pares base. El paquete *mfold* consiste de un grupo de programas escritos en Fortran o C y un grupo de scripts Perl. A algunos de los programas de *mfold* que se han trasladado a C++ y se ejecutan en Windows 98, Windows NT se les llama $RNAstructure$ [17].

Existen otros programas en línea más sofisticados.

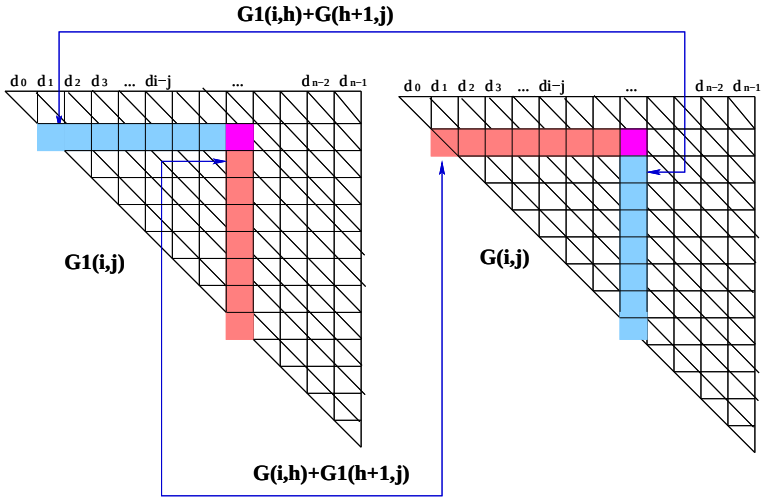
1.6.2 Algoritmos secuenciales

En [11] (Zuker *et al*, 1991) se describe un algoritmo que predice las soluciones óptimas y subóptimas basado principalmente en la minimización de la energía libre. Hace una comparación entre las posibles estructuras secundarias obtenidas a partir de la filogenética y por las dos estructuras óptimas y subóptimas obtenidas a partir del algoritmo propuesto.

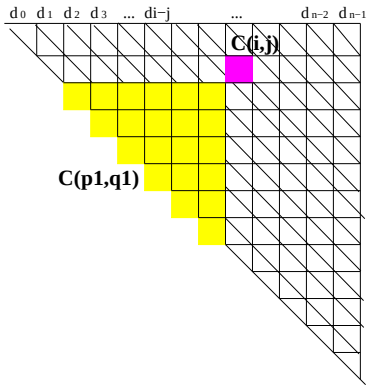
En [18] (Lyngso *et al*, 1999), se describe un método que utiliza la estrategia de programación dinámica, para evaluar ciclos internos utilizando propiedades actuales de la energía. Reducen la complejidad en tiempo de $O(n^4)$ a $O(n^3)$, aun cuando el tamaño de los ciclos internos está acotado por k (de $O(k^2n^2)$ a $O(kn^2)$). Este algoritmo se aplica para determinar la estructura más estable y para calcular la función de partición, utiliza estructuras que no contienen pseudoknots y sólo evalúa ciclos internos y bultos (bulges).



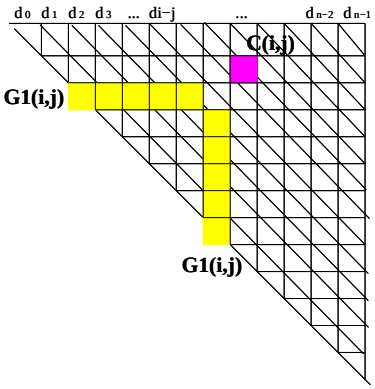
a) Tabla de dependencias para la función G(i,j).



b) Tabla de dependencias para la función G1(i,j).



c) Tabla de dependencias para la función C(i,j).



d) Tabla de dependencias para la función C(i,j), llamando a la función G1(i,j).

Figura 1.10: Tablas de dependencias para el algoritmo $O(n^4)$.

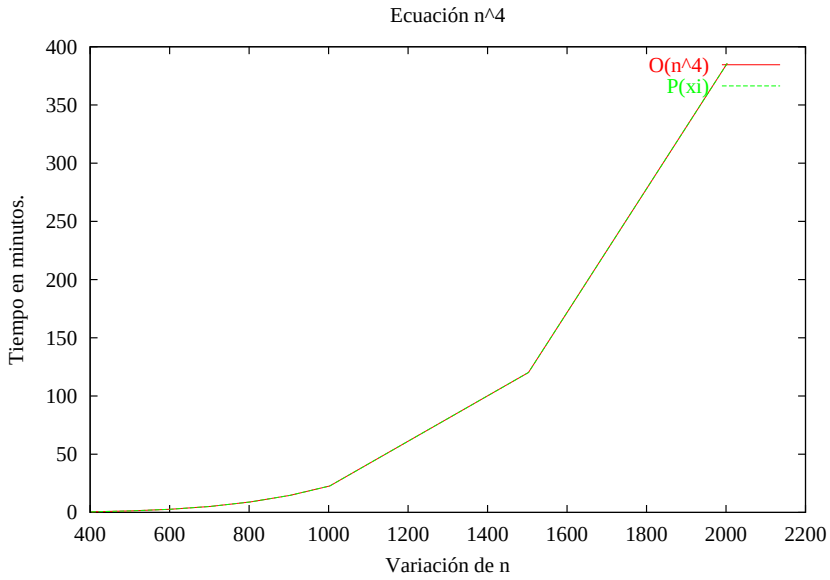


Figura 1.11: Tiempos de ejecución (en minutos) del algoritmo $O(n^4)$ secuencial para el problema del RNA para diferentes tamaños de problema.

Capítulo 2

Algoritmos paralelos para el problema del RNA

La necesidad que surge para resolver problemas que requieren tiempo elevado de cómputo origina lo que hoy se conoce como computación paralela. Mediante el uso concurrente de varios procesadores se resuelven problemas de manera más rápida que lo que se puede realizar con un solo procesador.

Una computadora paralela es un conjunto de procesadores que son capaces de trabajar cooperativamente para solucionar un problema computacional. Esta definición es muy extensa e incluye supercomputadoras que tienen cientos o miles de procesadores, redes de estaciones de trabajo o estaciones de trabajo con múltiples procesadores.

En este capítulo se hace una revisión rápida de algunos conceptos de paralelismo que servirán como introducción para explicar las técnicas de particionamiento aplicados a los algoritmos paralelos empleados para resolver el problema del RNA.

2.1 Diseño de algoritmos paralelos

Los algoritmos paralelos son extremadamente importantes para solucionar problemas grandes para muchos campos de aplicación. En esta sección se describen las etapas típicas para el diseño de los algoritmos paralelos:

1. Particionamiento. Los cálculos se descomponen en pequeñas tareas. Usualmente es independiente de la arquitectura o del modelo de programación. Un buen particionamiento divide tanto los cálculos asociados con el problema como los datos sobre los cuales opera.

2. Comunicación. Las tareas generadas por una partición están propuestas para ejecutarse concurrentemente pero no pueden, en general, ejecutarse independientemente. Los cálculos en la ejecución de una tarea normalmente requerirán de datos asociados con otras tareas. Los datos deben transferirse entre las tareas y así permitir que los cálculos procedan. Este flujo de información se especifica en esta fase.

3. Aglomeración. Las tareas y las estructuras de comunicación definidas en las dos primeras etapas del diseño son evaluadas con respecto a los requerimientos de ejecución y costos de implementación. Si es necesario, las tareas son combinadas en tareas más grandes para mejorar la ejecución o para reducir los costos de comunicación y sincronización.

4. Mapeo. Cada tarea es asignada a un procesador de tal modo que intente satisfacer las metas de competencia al maximizar la utilización del procesador y minimizar los costos de comunicación.

Estas cuatro etapas se ilustran en la Figura 2.1.

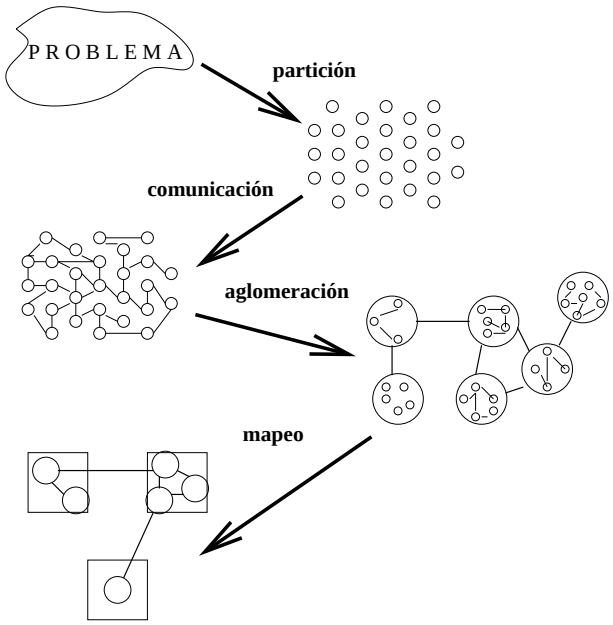


Figura 2.1: PCAM (Particionamiento, Comunicación, Aglomeración y Mapeo): una metodología de diseño para programas paralelos. Iniciando con la especificación de un problema, desarrollando una partición, determinando los requerimientos de comunicación, las tareas aglomeradas, y finalmente el mapeo de tareas a procesadores.

2.2 Arquitecturas paralelas

Para poder resolver eficientemente el problema de mapeo es necesario conocer acerca de la arquitectura y organización de la computadora paralela. Existen dos tipos básicos de arquitecturas paralelas: multiprocesadores y multicomputadoras. A continuación se proporciona una breve descripción de ellos:

2.2.1 Multiprocesadores

Los multiprocesadores son computadoras de múltiples CPU's los cuales comparten memoria; cada procesador es capaz de ejecutar su propio programa. En un multiprocesador de acceso de memoria uniforme (UMA) la memoria compartida es centralizada, en un multiprocesador de acceso a memoria no uniforme (NUMA) la memoria compartida es distribuida. En un sistema de memoria compartida cualquier dirección de memoria es accesible por cualquier procesador.

En los multiprocesadores UMA, la memoria física es uniformemente compartida por todos los procesadores, los cuales tienen el mismo tiempo para acceder a la memoria; por este motivo se le llama memoria de acceso uniforme. Los periféricos también pueden llegar a compartirse. Cuando todos los procesadores tienen el mismo tiempo para acceder a los periféricos, al sistema se le da el nombre de multiprocesador simétrico. En caso de que sólo uno o un subconjunto de procesadores pueden acceder a los periféricos se le conoce como multiprocesador asimétrico.

El patrón más simple de intercomunicación asume que todos los procesadores trabajan a través de un mecanismo de interconexión para comunicarse con una memoria compartida centralizada (vea la Figura 2.2). Existen una gran variedad de formas para implementar este mecanismo, por ejemplo un bus, una red conectada, etc.

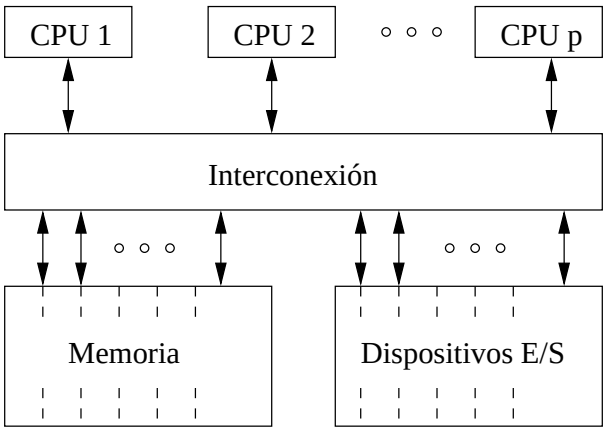


Figura 2.2: Modelo de multiprocesador de acceso a memoria uniforme (UMA). Todos los procesadores acceden a un mecanismo de interconexión para comunicarse con la memoria compartida global y dispositivos de entrada/salida (E/S).

Los multiprocesadores de acceso a memoria no uniforme (NUMA) están caracterizados por un espacio de direcciones compartido. Sin embargo, la memoria se encuentra físicamente distribuida. Cada procesador tiene memoria local, y el espacio de direcciones compartido está formado por la combinación de estas memorias locales. El tiempo necesario para acceder a una localidad de memoria particular en un procesador NUMA depende de si esa localidad es local al procesador. Toda la memoria en el sistema es alcanzable y cada procesador tiene una memoria que puede alcanzar directamente. Sin embargo, cada procesador debe hacer una petición a través de la red para hacer una conexión a otra memoria. La Figura 2.3 muestra una red de árbol aplicada a un sistema de memoria compartida NUMA.

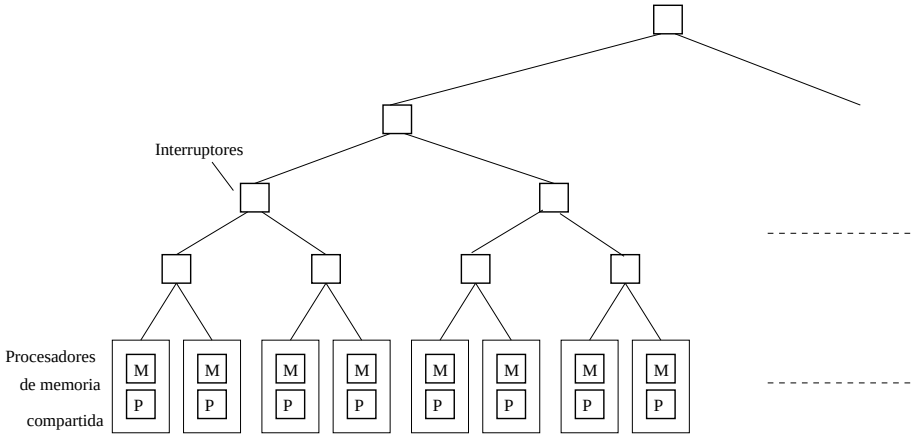


Figura 2.3: Red de árbol NUMA.

La programación para multiprocesadores se hace por medio de bibliotecas de funciones como Pthreads, o por el uso de un lenguaje de programación como SISAL, Haskell, etc [5]. Como el modelo de programación utiliza un solo espacio de direccionamiento, la comunicación es implícita por lo que es necesario tener un mecanismo de sincronización (implícito o explícito).

Los multiprocesadores presentan ciertas desventajas y se describen a continuación:

1. No tan fácilmente se pueden adaptar para extenderse a un gran número de procesadores dado que el bus de interconexión es de capacidad limitada.

2. La competencia por el acceso a la memoria puede reducir significativamente la velocidad del sistema.

3. Las técnicas de sincronización son necesarias para controlar el acceso a variables compartidas y el programador debe incorporar tales operaciones de manera implícita o explícita en las aplicaciones.

2.2.2 Multicomputadoras

Una computadora von Neumann está constituida por una unidad de procesamiento central (CPU) que ejecuta un programa que desempeña una secuencia de operaciones de lectura y escritura en una memoria adjunta. Una multicomputadora consiste de un conjunto de computadoras von Neumann, o nodos ligados por una red de interconexión. Cada nodo de una multicomputadora es una computadora autónoma constituida por un procesador, memoria local y algunas veces con discos adjuntos o periféricos E/S; esta arquitectura se muestra en la Figura 2.4. Cada nodo ejecuta su propio programa, el cual puede acceder a la memoria local y puede enviar y recibir mensajes sobre toda la red. Los mensajes se usan para comunicarse con otras computadoras o, para leer o escribir en memoria remota. Un atributo de este modelo es que accede a la memoria local (mismo nodo) y es menos caro que acceder a la memoria remota (de un nodo diferente), es decir la lectura y escritura a memoria es menos costoso que el enviar y

recibir un mensaje. Por lo tanto, es preferible acceder a datos locales que a datos remotos. Por tanto, en las aplicaciones para multicomputadoras es necesario localizar los datos cerca de los procesadores que los utilizan (localidad de datos).

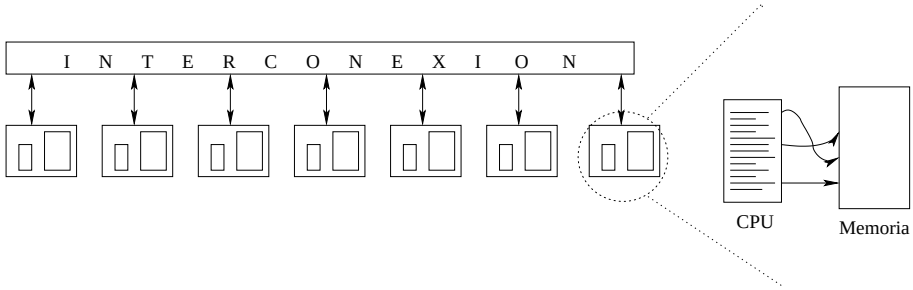


Figura 2.4: La multicomputadora, un modelo de computadora paralela idealizado. Cada nodo consiste de una computadora *von Neumann*: un CPU y memoria. Un nodo puede comunicarse con otro nodo por el envío y recibo de mensajes sobre una red de interconexión.

Las multicomputadoras son un ejemplo de las computadoras MIMD (multiple instruction multiple data) de memoria distribuida. MIMD significa que cada procesador puede ejecutar un flujo de instrucciones en su propia área de datos local; la memoria es distribuida entre los procesadores, en lugar de colocarlos de manera central. La principal diferencia entre una multicomputadora y una computadora de memoria distribuida MIMD es que en la última, el costo de enviar un mensaje entre dos nodos puede no ser independiente de la localización del nodo y del tráfico de la red. Ejemplos de esta clase de computadoras incluyendo la IBM SP, Intel Paragon, Thinking Machines CM5, Cray T3D, Meiko CS-2 y nCUBE [8].

Los algoritmos de multicomputadora no pueden ejecutarse eficientemente en una computadora SIMD (single instruction multiple data); en éstas todos los procesadores ejecutan el mismo flujo de instrucciones sobre una pieza diferente de datos. La MasPar MP es un ejemplo de esta clase de computadoras.

En los sistemas de memoria distribuida, cada procesador tiene acceso a su propia memoria, entonces la programación es más compleja ya que cuando los datos a usar por un procesador están en el espacio de direcciones de otro, es necesario solicitarla y transferirla a través de mensajes. De esta forma, es necesario impulsar la localidad de los datos para minimizar la comunicación entre procesadores y obtener un buen rendimiento. La ventaja que proporcionan estos tipos de sistemas es la escalabilidad, es decir, el sistema puede crecer un número mayor de procesadores que los sistemas de memoria compartida y, por lo tanto, es más adecuado para las computadoras paralelas con un gran número de procesadores.

Sus principales desventajas son que el código y los datos tienen que ser transferidos físicamente a la memoria local de cada nodo antes de su ejecución, y esta acción puede constituir un trabajo adicional significativo. Similarmente, los resultados necesitan transferirse de los nodos al sistema *host*. Los datos son difíciles de compartir. Las arquitecturas de multicomputadoras son generalmente menos flexibles que las arquitecturas de multiprocesadores. Por ejemplo los multiprocesadores podrían emular el paso de mensajes al usar localidades compartidas para al-

macenar mensajes, pero las multicomputadoras son muy ineficientes para emular operaciones de memoria compartida.

Los programas desarrollados para multicomputadoras pueden ejecutarse eficientemente en multiprocesadores, porque la memoria compartida permite una implementación eficiente de paso de mensajes. Ejemplos de esta clase de computadoras son la Silicon Graphs Challenge, Sequent Symmetry, y muchas estaciones de trabajo con multiprocesadores.

En la última fase del diseño de algoritmos paralelos (mapeo) lo que se toma en cuenta es la arquitectura de la computadora paralela, debido a que los algoritmos diseñados para memoria compartida presentan otras características que los diferencia de los algoritmos diseñados para arquitecturas de memoria distribuida. En la siguiente sección se muestra la programación aplicada a cada arquitectura.

2.3 Programación para computadoras paralelas

La programación paralela es diferente para cada arquitectura. La programación con hilos se aplica principalmente a las arquitecturas de multiprocesadores con memoria compartida y la programación con paso de mensajes a las arquitecturas de multicomputadoras. A continuación se hace una revisión de los tipos de programación utilizados en cada arquitectura.

2.3.1 Programación para memoria compartida

En los sistemas de multiprocesadores, cada procesador puede acceder a toda la memoria, es decir, hay un espacio de direccionamiento compartido. Todos los procesadores se encuentran igualmente comunicados con la memoria principal y pueden acceder por medio de un ducto común. En esta configuración se debe asegurar que los procesadores no accedan de manera simultánea a las regiones de memoria de tal manera que se provoque un error. Por ejemplo, si dos o más procesadores desean actualizar una variable compartida se deben establecer procedimientos que permitan acceder a los datos compartidos como exclusión mutua. Esto se logra mediante el uso de candados o semáforos y mecanismos de sincronización explícitos o implícitos como las barreras. Debido a que el modelo de programación en estos sistemas usa un solo espacio de direccionamiento la comunicación entre procesadores es implícita.

La programación en estos sistemas es más sencilla, ya que los datos se pueden colocar en cualquier módulo de la memoria y los procesadores la pueden acceder de manera uniforme. Se pueden utilizar bibliotecas de funciones con un lenguaje secuencial, para programar multiprocesadores, tal es el caso de pthreads, que es una especificación estándar para soporte de *multithreading* a nivel de sistema operativo [5].

En un lenguaje de alto nivel normalmente se programa un hilo empleando procedimientos, donde las llamadas a procedimientos siguen la disciplina tradicional de pila. Con un único hilo, existe en cualquier instante un único punto de ejecución. El programador no necesita aprender nada nuevo para emplear un único hilo. Cuando se tienen múltiples hilos en un programa significa que en cualquier instante el programa tiene múltiples puntos de ejecución, uno en cada uno de sus hilos. El programador puede ver a los hilos como si estuvieran en ejecución

simultánea, como si la computadora tuviese tantos procesadores como hilos en ejecución. El programador decide cuando y donde crear múltiples hilos, ayudándose de un paquete de bibliotecas o un sistema de tiempo de ejecución; pero debe estar consciente de que la computadora no necesariamente ejecuta todos los hilos simultáneamente.

Los hilos se ejecutan en un único espacio de direcciones, lo que significa que el hardware de direccionamiento de la computadora está configurado para permitir que los hilos lean y escriban en las mismas posiciones de memoria. En un lenguaje de alto nivel, esto corresponde normalmente al hecho de que las variables globales (fuera de la pila) son compartidas por todos los hilos del programa. Cada hilo se ejecuta en una pila separada con su propio conjunto de variables locales y el programador es el responsable de emplear los mecanismos de sincronización del paquete de hilos para garantizar que la memoria compartida es accedida de forma correcta.

Un hilo es un concepto sencillo : un simple flujo de control secuencial. Un *thread* (hilo) se define como un proceso ligero que comparte el mismo espacio de memoria y variables globales que el proceso a partir del cual se crea. La creación de hilos es una operación más rápida que la creación de procesos. Un hilo tiene acceso inmediato a variables globales compartidas y su sincronización es más eficiente que la sincronización de procesos, realizándose con una sola variable.

El acceso a datos compartidos tiene que hacerse de acuerdo a las siguientes reglas: primero, la lectura múltiple del valor de una variable no causa conflictos; segundo, la escritura a una variable compartida tiene que hacerse con exclusión mutua. El acceso a secciones críticas se puede controlar mediante una variable compartida. El mecanismo más simple para asegurar exclusión mutua de secciones críticas es el uso de un candado (*lock*). Un *lock* en Pthreads se implementa con variables de exclusión mutua.

Pthreads

Pthreads es la abreviatura de POSIX threads. Es el estándar sobre hilos IEEE POSIX1003.1c, aprobado en Junio de 1995 [32]. También es la abreviatura de la implementación del estándar POSIX diseñada por Chris Provenzano, desarrollada en el MIT (Massachussets Institute of Technology). Es una biblioteca de programación para el lenguaje C, que permite desarrollar aplicaciones empleando las técnicas de programación relacionadas con los hilos siguiendo el estándar POSIX [32].

En Pthreads, el programa principal es un hilo mismo. En general existen 3 mecanismos básicos para programar con hilos: creación de hilos, exclusión mutua y espera de eventos.

1. Creación de hilos:
   La primitiva de creación de hilos POSIX es pthread_create() cuya sintaxis se muestra a continuación:

   int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

   A esta primitiva se le deben pasar los siguientes argumentos según el orden :

- una variable de hilo, donde se nos indicará el identificador de hilo asociado al hilo creado.

- una variable de atributos de hilo que indica las características del hilo creado, aunque se suelen emplear atributos por defecto proporcionados por la variable global phtread_attr_default.

- una función de inicio, que será la rutina inicial en la que el nuevo hilo comenzará su ejecución.

- un argumento para la función de inicio, que nos permitirá pasar información en forma de parámetros a la rutina inicial del hilo.

2. Exclusión mutua (mutual exclusion):
   La forma más simple de que un hilo interactúe con otros es a través de acceso a variables compartidas. Como los hilos se ejecutan en paralelo, el programador debe planificar los accesos de forma explícita para evitar errores cuando más de un hilo trata de acceder a las variables compartidas. La herramienta más simple para hacer esto es mediante las variables de exclusión mutua, las cuales son primitivas que proporcionan acceso a regiones críticas. Se trata de indicar que una región particular de código sólo puede ejecutarse por un determinado hilo al mismo tiempo. Una variable de exclusión mutua debe primero ser creada mediante la primitiva pthread_mutex_init(), aunque inicialmente la variable de exclusión mutua debe contener el valor de inicialización pthread_mutex_initializer.

   La sintaxis de la primitiva de inicialización es :

   int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *attr);

   Una variable de exclusión mutua tiene dos estados: bloqueado y desbloqueado. Inicialmente la variable está desbloqueada. La primitiva pthread_mutex_lock() bloquea la variable, y continua la ejecución. Al finalizar se debe desbloquear la variable mediante la primitiva pthread_mutex_unlock(). Se suele decir que un hilo posee el control de la variable cuando se encuentra en ejecución entre las primitivas de bloqueo (lock) y desbloqueo (unlock). Si otro hilo intenta adquirir el control de la variable cuando ya está bloqueada, el segundo hilo se suspende hasta que la variable es liberada por el primer hilo. Se puede conseguir una exclusión mutua de un conjunto de variables con solo asociarlas con una variable de exclusión mutua.

3. Variables de condición:
   Se puede ver a una variable de exclusión mutua como un tipo simple de mecanismo de

planificación de recursos. El recurso a planificar es la memoria compartida accedida dentro de las primitivas de bloqueo y desbloqueo de una variable de exclusión mutua, y la política de planificación es un único hilo al mismo tiempo. Pero a veces se necesita expresar políticas de planificación más complicadas. Esto requiere la utilización de un mecanismo que permita a un hilo suspenderse hasta que suceda algún determinado evento. Este mecanismo de espera por un evento se consigue mediante una variable de condición.

Una variable de condición está siempre asociada con una variable de exclusión mutua particular, y con los datos protegidos por ella. En general, un monitor consta de una serie de datos, una variable de exclusión mutua, y algunas variables de condición. Una variable de condición particular es siempre usada junto con la misma variable de exclusión mutua y sus datos.

La primitiva pthread_cond_init() permite inicializar una variable de condición para su uso posterior. La primitiva pthread_cond_wait() desbloquea la variable de exclusión mutua y suspende el hilo de forma atómica, en una sola acción. La primitiva pthread_cond_signal() no hace nada a menos que exista un hilo bloqueado en la variable de condición, en cuyo caso despierta a uno de los hilos bloqueados a la espera. La primitiva pthread_cond_broadcast() es similar a la anterior excepto que despierta a todos los hilos suspendidos a la espera de la variable de condición. En la Figura 2.5 se muestra un ejemplo de la aplicación de las variables de condición.

```
pthreads_mutex_t    mutex1;
pthreads_cond_t   cond1;
.
pthread_cond_init( &cond1, NULL);
pthread_mutex_init ( &mutex1, NULL);
```

```
action()
{
   .
   pthread_mutex_lock( &mutex1 );
   while( c <>  0 )
      pthread_cond_wait( cond1, mutex1 );   <----------
   pthread_mutex_unlock( &mutex1 );
   take_action();
   .
}
```

```
counter()
{
   .
   pthread_mutex_lock( &mutex1);
   c--;
   if( c == 0 )
      pthread_cond_signal( cond1 );
   pthread_mutex_unlock( &mutex1 );
   .
 }
```

Figura 2.5: Variables de condición.

Esta biblioteca se utilizó para la implementación de los algoritmos paralelos propuestos para resolver el problema de RNA.

2.3.2 Programación con paso de mensajes

Si la memoria está distribuida entre los procesadores, es decir, cada procesador tiene acceso a su propia memoria, entonces la programación es más compleja ya que cuando los datos a usar por un procesador están en el espacio de direcciones de otro, será necesario solicitarla y transferirla a través de mensajes. De este modo, es necesario impulsar la localidad de los datos para minimizar la comunicación entre procesadores y obtener un buen rendimiento.

Los programas con paso de mensajes, crean múltiples tareas; cada tarea encapsula un dato local e interactúa mediante el envío y recibo de mensajes. En los sistemas de multicomputadoras, el código de cada procesador se carga en la memoria local y algunos de los datos requeridos se almacenan localmente. Los programas se particionan en partes separadas y se ejecutan concurrentemente por procesadores individuales. Cuando los procesadores necesitan acceder a la información de otros procesadores, o enviar información a otros procesadores, se comunican mediante el envío de mensajes.

El paso de mensajes se ha usado como un medio de comunicación y sincronización entre una colección arbitraria de procesadores, incluyendo un solo procesador. Los programas que usan paso de mensajes son totalmente estructurados. Frecuentemente, todos los nodos ejecutan copias idénticas de un programa, con el mismo código y variables privadas (modelo SPMD).

La ventaja que presentan estos sistemas es su escalabilidad, es decir, el sistema puede crecer un número mayor de procesadores que los sistemas de memoria compartida y, por lo tanto, es más adecuado para las computadoras paralelas.

Los procedimientos de envío y recepción de mensajes en los sistemas de paso de mensajes frecuentemente tienen la siguiente forma:

send(parámetros)
recv(parámetros)

donde los parámetros identifican los procesadores fuente y destino, y los datos. Estas rutinas pueden dividirse en dos tipos: síncrono o bloqueante y asíncrono o no bloqueante.

Las rutinas síncrono o bloqueante no permiten que los procesos continúen hasta que la operación ha sido completada. Las rutinas asíncronas o no bloqueantes permiten que los procesos continúen a pesar de que la operación no se haya terminado; es decir, las instrucciones que continúan de la rutina se ejecutan a pesar de que la rutina no haya sido concluida. Las rutinas de envío bloqueante esperaran hasta que el mensaje completo haya sido transmitido y aceptado por el proceso receptor. Una rutina de recepción bloqueante esperará hasta que el mensaje esperado es recibido. Las rutinas bloqueantes ejecutan dos acciones: la transferencia de datos y la sincronización de procesos.

Las rutinas de envío de paso de mensajes no bloqueantes permiten a los procesos continuar inmediatamente después de que el mensaje ha sido construido sin esperar por la aceptación o el recibo. Una rutina de recibo no bloqueante no esperará el mensaje y permitirá a los procesos continuar. Las rutinas no bloqueantes generalmente decrementan el proceso de tiempo de ejecución. El paso de mensajes no bloqueante implica que las rutinas tengan buffers para almacenar

los mensajes, pero estos son de longitud finita y podrían llegar a bloquearse cuando el buffer este lleno.

Una de las herramientas para programación con paso de mensajes más utilizada es MPI (Message Passing Interface) que es la especificación estándar de un conjunto de funciones para paso de mensajes. Esta biblioteca utiliza este tipo de rutinas y es la que se ha usado para la implementación de las estrategias de particionamiento que se describen en la penúltima sección de este capítulo, para la simulación de memoria distribuida.

2.4 Estrategias de particionamiento paralelo para el problema de RNA

Como se ha mencionado anteriormente, una de las primeras actividades en el diseño de un algoritmo paralelo es el particionamiento del trabajo computacional que debe realizar el algoritmo. Específicamente, para el problema de RNA, en los algoritmos $O(n^3)$ y $O(n^4)$ el trabajo computacional está centrado en el proceso de llenado de una tabla de forma triangular superior. En esta sección se discuten las diferentes formas de particionar la tabla de programación dinámica para lograr una mejor distribución del trabajo entre los procesadores, iniciando con el más común (diagonales) y finalizando con los más generales (bloques homogéneos y no homogéneos).

Haciendo uso de las ecuaciones 1.3 y 1.10, se puede examinar que en ambos algoritmos se trata de llenar una tabla de forma triangular superior, $F(i, j)$ y el cálculo de una entrada requiere los datos como se presenta en la Figura 2.6. Para calcular cada valor de la diagonal de las ecuaciones 1.3, 1.10, 1.11 y 1.12 se necesitan los resultados de la diagonal 0 hasta la diagonal $(j - i - 1)$ y para encontrar el valor de energía de una secuencia dada se procede desde la diagonal 0 $(i = j)$ hasta la diagonal $n - 1$ $(j - i = n - 1)$. De aquí, el proceso de llenado de la tabla para ambos algoritmos ($O(n^3)$ y $O(n^4)$) puede realizarse por orden inverso de renglones (de abajo hacia arriba), de izquierda a derecha por columnas, o por diagonales iniciando en 0 hasta $n - 1$, este comportamiento se puede observar en la Figura 2.7. Estas tres formas de llenado se deben a que en la tabla existen dependencias de datos y para eliminarlos es necesario que los valores que se necesiten para calcular una solución ya hayan sido calculados anteriormente. De estas tres formas de llenado, la única que ofrece la posibilidad de explotar paralelismo es el llenado por diagonales.

Para que dos o más cómputos paralelos se ejecuten simultáneamente, los cálculos de un proceso no pueden depender de los resultados de otros procesos. Si se calcula F por columnas, los cálculos de un elemento en una columna necesita de los resultados de los elementos de la misma columna, y esto prohibe que los cálculos en una columna sean concurrentes. Lo mismo sucede con el llenado por renglones. Sin embargo, si los cálculos en una diagonal dependen de las datos de diagonales anteriores pero son independientes de los datos en la diagonal actual, los cálculos pueden proceder simultáneamente para todos los elementos en la diagonal.

Por lo anterior, se eligió el llenado por diagonales en la implementación de los algoritmos para el llenado de la tabla.

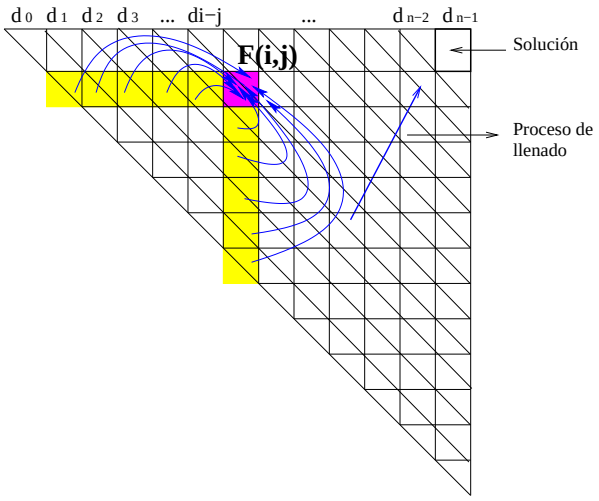


Figura 2.6: Cálculo de $F(i, j)$.

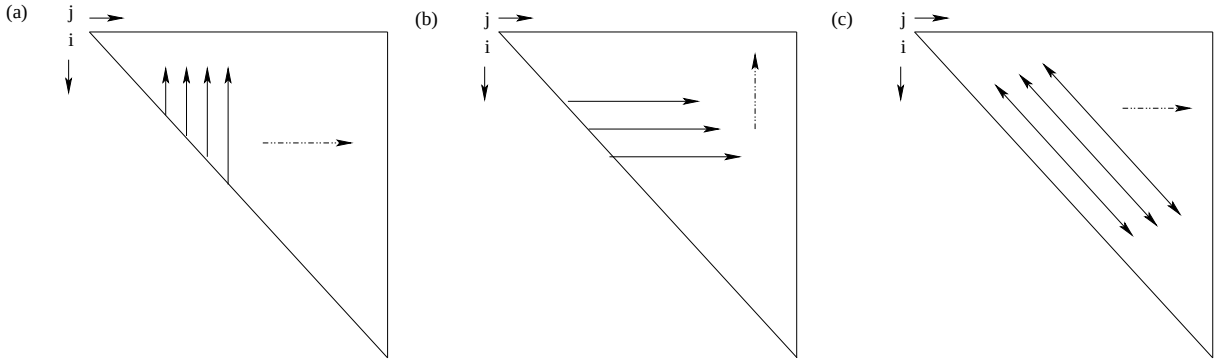


Figura 2.7: Tipos de llenado de la tabla: a)Por columnas de abajo hacia arriba y de izquierda a derecha. (b)Por renglones, de abajo hacia arriba y de izquierda a derecha y c)Por diagonales en ambos sentidos.

2.4.1 Particionamiento por diagonales

Existen tres formas comunes de particionar la tabla de programación dinámica (vea la Figura 2.8): la primera es por renglones, de tal forma que a cada procesador le corresponde un conjunto determinado de renglones, la segunda es por columnas, de tal forma que a cada procesador le corresponde un conjunto de columnas. La tercera forma de particionar la tabla es por diagonales, de tal forma que a cada procesador le corresponde una parte de la diagonal. El problema con el particionamiento por renglones o por columnas es que la distribución de trabajo no es equitativa para cada procesador. Claramente, en ambos casos a un procesador le corresponde calcular un número mayor de entradas. Por ejemplo, de acuerdo a la Figura 2.8, en el particionamiento por renglones el procesador p_0 realiza un mayor número de evaluaciones que el procesador p_n y en el particionamiento por columnas es el caso contrario, es decir, el procesador p_n realiza un número mayor de evaluaciones que el procesador p_0. En el particionamiento por diagonales

la distribución del trabajo es más equitativa, de forma que ningún procesador trabaja más de lo necesario. Por lo tanto, el particionamiento por diagonales es el que se ha implementado en esta sección y se describe a continuación.

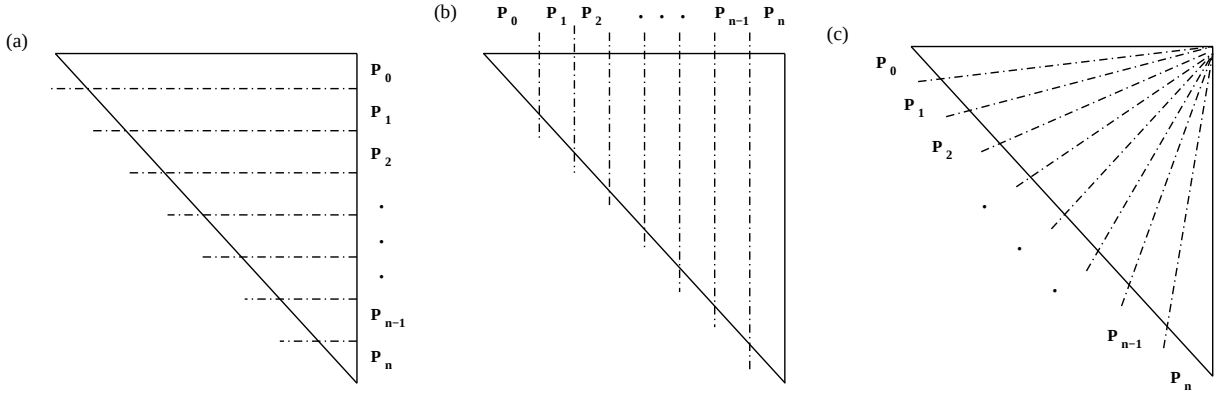


Figura 2.8: Particionamiento de la tabla de programación dinámica para el problema de RNA, (a) Por renglones, (b)Por columnas y (c)Por diagonales.

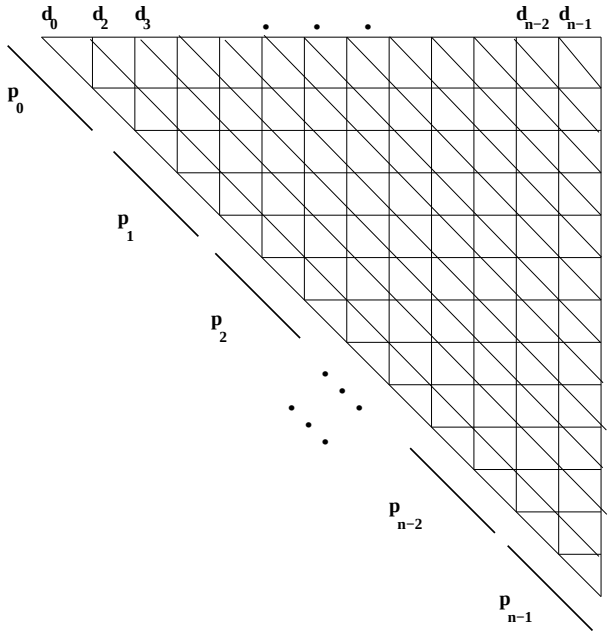


Figura 2.9: Particionamiento por diagonales.

Como se necesita que todos los elementos de la diagonal se resuelvan, los procesos o tareas no pueden avanzar a la siguiente diagonal hasta que la diagonal anterior haya sido completamente calculada. De esta manera, es necesario establecer un proceso de sincronización entre procesadores o una barrera por cada diagonal, como se ilustra en la Figura 2.9. Conforme se va

avanzando los elementos de la diagonal disminuyen, así como las operaciones que cada proceso debe realizar.

A cada tarea $p_0 ... p_{n-1}$ le corresponde una parte de cada diagonal d_k. Es decir, $longitud_{tarea} = (n - d_k)/t$, donde n es la longitud de la secuencia, d_k es la k-ésima diagonal y t es el número de tareas. Con estas variables se calcula la longitud del segmento de la diagonal d_k que le corresponde a cada tarea. El Algoritmo 1 muestra la implementación de esta estrategia de particionamiento, los cálculos se realizan por diagonales iniciando en $d = 3$ (por la restricción de $j - i \leq 3$) hasta $n - 1$. Por cada diagonal se determina su longitud y se divide entre el número de procesadores disponibles, se determinan los límites que le corresponden al procesador actual y se evalúan las funciones $F(i, j)$ y $G(i, j)$. Al término de la evaluación de la región que le correspondió al procesador actual, se realiza un proceso de sincronización, ya sea una barrera para memoria compartida o paso de mensajes si el algoritmo es para memoria distribuida. Para que el procesador continúe a la siguiente diagonal, los procesadores restantes deben terminar antes sus cálculos, y esto se realiza hasta que d tome el valor de n, donde n es la longitud de la secuencia.

De acuerdo al pseudocódigo, por cada diagonal se debe establecer un proceso de sincronización o comunicación. Para secuencias de n elementos se requieren n procesos de sincronización, lo que provoca que para secuencias grandes el tiempo de ejecución sea muy alto.

Algoritmo 1 Particionamiento por diagonales.

```
n ← longitud de la secuencia, d ← diagonal
longDia ← longitud de la diagonal
p ← número de procesador, Np ← número de procesadores
seccionDia ← sección que le corresponde al procesador p
limInicial ← límite inicial de la sección de la diagonal
limFinal ← límite final de la sección de la diagonal

d ← 3
for d < n do
   longDia ⇐ n − d
   seccionDia ⇐ longDia/Np
   Determina los límites del procesador p
   for i = limInicial to limFinal do
      j ⇐ i + d
      Determina el valor de energía de la subsecuencia (i, j)
   end for
   Proceso de sincronización: barrera (memoria compartida) o paso de mensajes
   (memoria distribuida)
end for
```

El tiempo aproximado de está estrategia para memoria compartida esta representado por la siguiente ecuación:

$$T_p \approx \frac{a \cdot n^3}{p} + (n-1)s$$

donde:
- a es una constante.
- p es el número de procesadores.
- n es la longitud de la secuencia.
- s es el tiempo de sincronización y comunicación.

El primer término indica el tiempo del algoritmo secuencial $a \cdot n^3$ dividido entre el número de procesadores disponibles, mientras que el segundo término es el número de operaciones de sincronización y comunicación por el tiempo que éstas requieran. El término $n-1$ se debe a que existen $n-1$ diagonales en la tabla y cada diagonal necesita de un proceso de sincronización (barrera).

A continuación se describen dos alternativas de particionamiento que reducen razonablemente el tiempo de ejecución al disminuir el número de operaciones de sincronización entre procesadores.

2.4.2 Particionamiento por bloques 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 del i-ésimo renglón y de la j-ésima columna, lo cual construye un triángulo equilátero (vea figura 2.6). Esto origina otra forma de particionar la matriz, por bloques de longitud fija (homogéneos), es decir se toma un subconjunto continuo de diagonales (bloque) y se forman triángulos para lograr eliminar las dependencias entre las soluciones. Con esto se disminuye el tiempo de sincronización entre los procesadores. El cálculo de los elementos que constituyen al triángulo se hace de manera secuencial. En el primer bloque sólo se calculan los triángulos superiores y en los bloques restantes se calculan bloques cuadrados eliminando así un proceso de sincronización.

La Figura 2.10 muestra este tipo de particionamiento, denominado por bloques homogéneos, dado que la tabla se divide entre B bloques de longitud fija y cada región de ese bloque se calcula de forma triangular. Los elementos que se encuentran en la primera diagonal, de acuerdo a la figura, forman sólo un triángulo superior, pero en el siguiente bloque se forman dos triángulos uno superior y otro inferior, calculando primero, por las dependencias que existen en la diagonal actual con las diagonales anteriores, el triángulo inferior y posteriormente el triángulo superior, y así sucesivamente hasta procesar los p bloques que dividieron a la matriz para su solución total.

El número de triángulos superiores que se encuentran en la primera diagonal equivale al número de bloques B; este número B disminuye conforme se va avanzando hacia los siguientes $B-1$ bloques. Con este tipo de particionamiento, se trata de disminuir el tiempo de sincronización entre tareas, al darle más trabajo por calcular a cada tarea. A la tarea h le corresponde un número determinado de triángulos del bloque que se está procesando en ese momento. El Algoritmo 2 muestra la implementación de esta estrategia. Inicialmente, se debe determinar la longitud del bloque. Cada procesador determina los límites que le corresponde por número

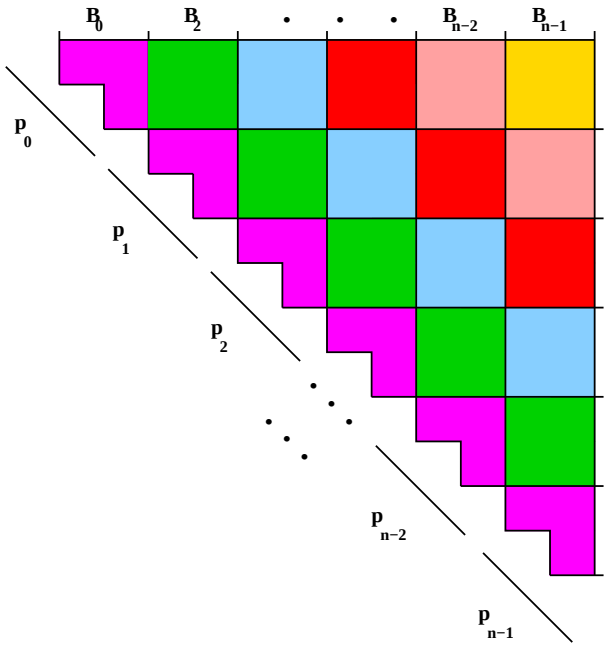


Figura 2.10: Particionamiento por bloques homogéneos.

de bloques (no por diagonales), es decir el bloque inicial y el bloque final, y calcula la región de bloques que le corresponden. Si se encuentra en la primera diagonal (primer bloque) los bloques que se forman son triángulos y es un caso especial del algoritmo. Para los bloques restantes se forman cuadrados. Al término de la evaluación de los bloques o triángulos se realiza un proceso de sincronización como en el algoritmo por diagonales. La diagonal d se incrementa con la longitud del bloque que se procesó en ese momento.

El tiempo aproximado de esta estrategia para memoria compartida está representado por la siguiente ecuación:

$$T_p \approx \frac{a \cdot n^3}{p} + B \cdot s$$

donde:

- a es una constante.
- p es el número de procesadores.
- n es la longitud de la secuencia.
- B es el número de bloques.
- s es el tiempo de sincronización y comunicación.

El primer término indica el tiempo del algoritmo secuencial $a \cdot n^3$ dividido entre el número de procesadores disponibles.
El segundo término es el número de operaciones de sincronización y comunicación por el tiempo que éstas requieran. El término B se debe a que se forman B bloques en la tabla y cada bloque necesita de un proceso de sincronización (barrera).

Algoritmo 2 Particionamiento por bloques homogéneos.

```
d ← diagonal, B ← número de bloques
p ← número de procesador, Np ← número de procesadores
longDia ← longitud de la diagonal
seccionB ← sección de triángulos que le corresponde al procesador p
limInicial, limFinal ← límite inicial y final de la sección de la diagonal
n ← longitud de la secuencia, longB ← longitud del bloque, numB ← número total de triángulos

longB ← (n − 3)/B
for d = 3 to n do
  longDia ⇐ n − d
  numB ⇐ longDia/longB
  seccionB ⇐ longDia/Np
  Determina los límites del procesador p
  if d es un triángulo then
    val ⇐ d
  else { es un cuadrado}
    val ⇐ d − longB + 1
  end if
  Calcula cada valor de la sección
  del número de triángulos
  for inicio = limInicial to limFinal do
    Calcula cada triángulo de manera individual
    dd ⇐ val
    final ⇐ inicio + longB − 1
    while dd < (d + longB) do
      if dd > d then
        j ⇐ dd + inicio
        final ⇐ final − 1
      else
        j ⇐ d + inicio
      end if
      i ⇐ j − dd
      while i <= final do
        Determina el valor de energía de la subsecuencia (i, j)
        Incrementa i, j
      end while
      dd ⇐ dd + 1
    end while
    inicio ⇐ inicio + longB
  end for
  Proceso de sincronización: barrera (memoria compartida) o paso de mensajes
  (memoria distribuida).
  d ⇐ longB
end for
```

2.4.3 Particionamiento por bloques no homogéneos

En el particionamiento por bloques homogéneos se busca que cada bloque sea del mismo tamaño. Sin embargo, conforme el proceso de llenado de la tabla avanza el tiempo de cómputo invertido para calcular cada bloque va variando. En la Figura 2.12 se presenta el tiempo de cómputo que se invierte en calcular cada una de las diagonales de la tabla triangular superior para un problema de tamaño 1000. Claramente se puede observar que el tiempo de cómputo se incrementa de manera cuadrática hasta alcanzar un nivel máximo en la diagonal situada a la mitad de la tabla para luego decrementarse cuadráticamente hasta llegar a la esquina superior derecha.

Esto sugiere que al inicio el cálculo de bloques es rápido para después hacerse más lento a la mitad de la tabla. Esto se debe a que el número de operaciones que se realizan en las primeras diagonales es menor porque se requieren menos resultados anteriores para calcular cada entrada de la diagonal. En las últimas diagonales el número de cálculos por cada entrada en la diagonal es menor porque la longitud de las últimas diagonales son cortas. Las diagonales que se encuentran en el centro de la tabla son de longitud mayor y requieren un mayor número de valores anteriores para obtener un valor de entrada. Con el propósito de tener un algoritmo balanceado en cómputo se puede proponer un particionamiento cuyos bloques no sean del mismo tamaño pero que sí inviertan la misma cantidad de tiempo computacional. Esto da origen a un particionamiento por bloques no homogéneos.

Este particionamiento es una generalización del particionamiento por bloques homogéneos; en tiempo de ejecución se determina la longitud del siguiente bloque a procesar. Para esto se necesita conocer el número de operaciones que se tienen que realizar para procesar cada entrada en la tabla. Se procede a determinar la longitud del bloque (diagonal siguiente) que obtenga el mismo tiempo de cálculo que los restantes $B-1$ bloques. La finalidad de esta estrategia es balancear el tiempo de cómputo y, a pesar de que son bloques de longitud variable, son iguales en el número de operaciones.

Bloques no homogéneos para el algoritmo $O(n^3)$

Para el algoritmo $O(n^3)$ el número de operaciones que se necesitan por cada entrada de la diagonal d al evaluar la función $F(i,j)$ es: $d+d$ valores por cada elemento que esta en la diagonal actual $(n-d+1)$ (ver la Figura 2.11). Lo anterior da como resultado que el tiempo para calcular todos los valores de la diagonal d esté representado por la siguiente ecuación:

$$f(d) = 2(n-d+1)d \tag{2.1}$$

Se busca conocer el valor de la diagonal d, que cumpla que el costo computacional de la diagonal 1 hasta la diagonal d sea igual al costo computacional de las diagonales restantes, desde la $d+1$ hasta la n, dividido entre $B-1$ bloques. La ecuación 2.2 muestra lo que se desea calcular:

$$F(n, 1 \leftrightarrow d) = \frac{F(n, (d+1) \leftrightarrow n)}{B-1} \tag{2.2}$$

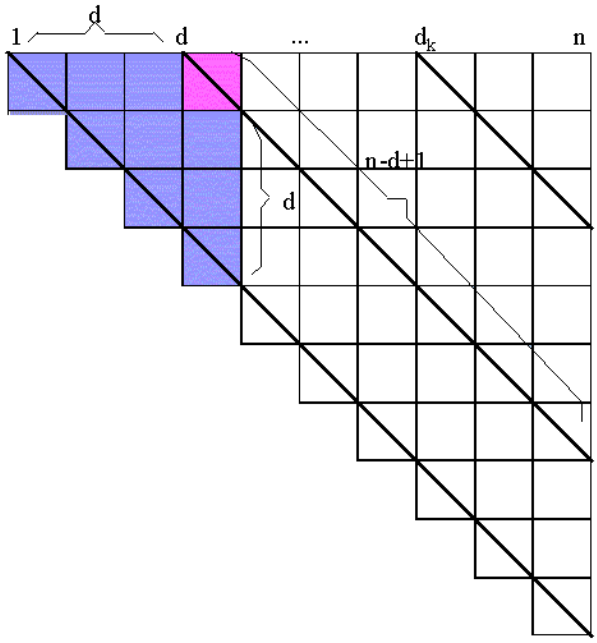


Figura 2.11: Número de operaciones necesarias para calcular cada entrada de la diagonal d.

Donde: $1 \leftrightarrow d$ indica que va de la diagonal 1 a la diagonal d y $(d+1) \leftrightarrow n$ indica que va de la diagonal $d+1$ hasta la diagonal n.

El tiempo total invertido para el llenado de la matriz se puede conocer calculando el área bajo la curva de la función $f(x)$ definida en la ecuación 2.1.

$$\int_1^n f(x)dx = \int_1^n 2x(n-x+1)dx = \frac{n^3}{3} + n^2 - n - \frac{1}{3} \tag{2.3}$$

De acuerdo a las leyes del cálculo, esta función se puede descomponer utilizando la siguiente propiedad:

$$\int_1^n g(x)dx = \int_1^k g(x)dx + \int_k^n g(x)dx \tag{2.4}$$

Así, el cálculo para un bloque estará determinado por la diagonal d_1, tal que,

$$\int_1^{d_1} f(x)dx = \int_1^{d_1} 2x(n-x+1)dx = \frac{-2d_1^3}{3} + (n+1)d_1^2 - n - \frac{1}{3} \tag{2.5}$$

Los bloques restantes invertirán en total el esfuerzo computacional dado por la siguiente relación:

$$\int_{d_1}^n f(x)dx = \int_{d_1}^n 2x(n-x+1)dx = \frac{2d_1^3}{3} - (n+1)d_1^2 - \frac{n^3}{3} + n^2 \tag{2.6}$$

Se pretende encontrar aquel valor de d_1 que cumpla con la siguiente relación:

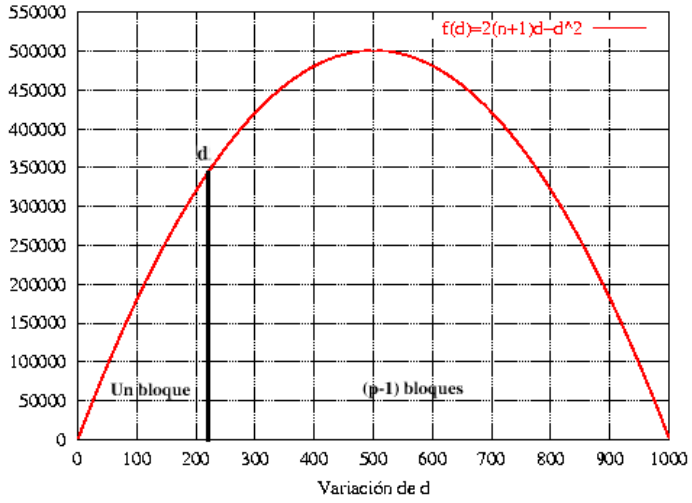


Figura 2.12: Costo computacional por cada entrada en la matriz.

$$\int_{1}^{d_1} f(x)dx = \frac{\int_{d_1}^{n} f(x)dx}{B-1} \tag{2.7}$$

De la ecuación 2.7 y de las expresiones representadas por 2.5, 2.6 y 2.3, se determina la siguiente ecuación de tercer grado, la cual, al resolverla, determina el valor de d_1 buscado.

$$\frac{-2}{3}[(B-1)+1]d_1^3 + (n+1)[(B-1)+1]d_1^2 - (B-1)n - \frac{(B-1)}{3} - \frac{n^3}{3} - n^2 = 0 \tag{2.8}$$

La ecuación 2.8 permite calcular sólo una diagonal d_1 dividiendo el trabajo en dos partes, la primera realizada por un bloque y la segunda por $B-1$ bloques. El mismo razonamiento puede aplicarse para encontrar las $B-2$ diagonales restantes que separen el trabajo realizado por los B bloques. De esta forma se puede aplicar repetidamente la siguiente relación:

$$\int_{d_k}^{d_{k+1}} f(x)dx = \frac{\int_{d_{k+1}}^{n} f(x)dx}{B-(k+1)},\ 0 \leq k < B \tag{2.9}$$

donde $d_0 = 1$.

La ecuación general obtenida para encontrar el valor de d_{k+1} dado d_k es la siguiente:

$$\begin{aligned} &\frac{-2}{3}[(B-(k+1))+1]d_{k+1}^3 + \frac{2}{3}[(B-(k+1))+1]d_k^3 + \\ &(n+1)[(B-(k+1))+1]d_{k+1}^2 - (B-(k+1))(n+1)d_k^2 - \frac{n^3}{3} - n^2 = 0 \end{aligned} \tag{2.10}$$

De esta forma, conociendo el número de bloques B y la diagonal anterior d_k podemos conocer la siguiente diagonal d_{k+1}, para lograr así una mejor distribución del costo computacional

entre todos los B bloques. La tabla puede partirse como se muestra en la Figura 2.13 donde cada bloque es de diferente tamaño (variable). Observando la Figura 2.13, cada bloque forma un número variable de triángulos, a diferencia del particionamiento por bloques homogéneos que era fijo y disminuía conforme se avanzaba el cálculo de las diagonales. En este particionamiento no se forman en el mismo bloque rombos (triángulos inferiores y superiores), sino un triángulo superior y posteriormente un triángulo inferior, lo que provoca que se realice primero el cálculo de todos los triángulos superiores y posteriormente el cálculo de todos los triángulos inferiores que se forman en el bloque que en ese momento se está procesando. Esto requiere de un proceso de sincronización al término del cálculo de todos los triángulos superiores, para pasar al cálculo de los triángulos inferiores y así lograr eliminar los conflictos de dependencias de datos.

En la Figura 2.13 se observa que los bloques que están en el centro de la tabla son más delgados, esto se debe a que en los extremos el número de operaciones por cada posición es menor en comparación con las diagonales que se encuentran en el centro. Este comportamiento también se observa en la Figura 2.12 de la función $f(d)$ que indica que sólo las diagonales que se encuentran en el centro necesitan de más tiempo de procesamiento que las diagonales que se encuentran en los extremos. El tiempo en las primeras diagonales crece, se mantiene en las diagonales del centro y disminuye conforme la diagonal d se acerca a n $(d = n)$, dado que la longitud de la diagonal d es menor y necesita de menos operaciones.

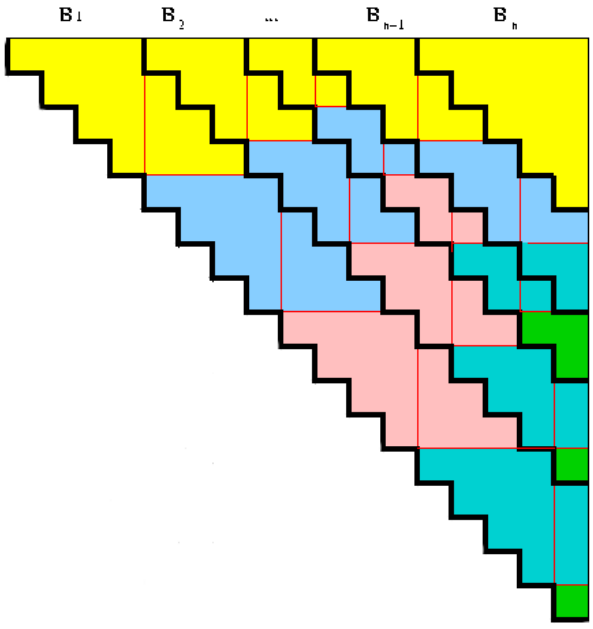


Figura 2.13: Particionamiento por bloques no homogéneos.

En el algoritmo 3 se muestra el pseudocódigo para la implementación de esta estrategia. En este algoritmo, se determina la diagonal que determina la longitud del bloque actual. De acuerdo a la longitud de bloque (determinado por la diagonal actual y la diagonal que limita al bloque) se determina el conjunto de triángulos superiores que le corresponden al procesador

actual (límites), se evalúan las funciones $F(i,j)$ ó $G(i,j)$ y se establece un proceso de sincronización para que todos los procesadores inicien al mismo tiempo los cálculos de los triángulos inferiores, procediendo de la misma forma que en triángulos superiores. La diagonal se actualiza con la longitud del bloque actual. Para determinar la diagonal que limita al bloque actual (siguiente) se evalúa la ecuación 2.10 proporcionándole los parámetros diagonal actual o anterior, la longitud de la secuencia y el número de bloques actual $B2$. Este número disminuye conforme se van procesando los bloques y después se debe aplicar un método de cálculo de raíces para conocer el valor de la diagonal siguiente.

Algoritmo 3 Particionamiento por bloques no homogéneos.

$d \leftarrow$ diagonal, $B \leftarrow$ número de bloques, $B2 \leftarrow$ auxiliar del número de bloque
$p \leftarrow$ número de procesador, $Np \leftarrow$ número de procesadores
$n \leftarrow$ longitud de la secuencia
$longDia \leftarrow$ longitud de la diagonal, $longB \leftarrow$ longitud del bloque
$limInicial$, $limFinal \leftarrow$ límite inicial y final de la sección de la diagonal

$d \leftarrow 3$
while $d < n$ & $B2 >= 0$ **do**
  Determina la diagonal siguiente ($diagonalSig$) a partir de la diagonal anterior de la longitud de la secuencia y del número de bloque actual
  $longB \leftarrow diagonalSig - d$

  PARTE SUPERIOR
  Determina los límites de los triángulos superiores
  Calcula los triángulos superiores a partir de los límites
  Proceso de Sincronización: barrera (memoria compartida) o paso de mensajes (memoria distribuida)

  PARTE INFERIOR
  Determina los límites de los triángulos inferiores
  Calcula los triángulos inferiores a partir de los límites
  Proceso de Sincronización: barrera (memoria compartida) o paso de mensajes (memoria distribuida)

  $d \Leftarrow d + longB$
  $B2 \Leftarrow B2 - 1$
end while

El tiempo aproximado de esta estrategia para memoria compartida está representado por la siguiente ecuación:

$$T_p \approx \frac{a \cdot n^3}{p} + 2 \cdot B \cdot s$$

donde:

a es una constante.
p es el número de procesadores.
n es la longitud de la secuencia.
B es el número de bloques.
s es el tiempo de sincronización y comunicación.

El término $\frac{a \cdot n^3}{p}$ indica el tiempo del algoritmo secuencial $a \cdot n^3$ dividido entre el número de procesadores disponibles.
El segundo término ($2 \cdot B \cdot s$) es el número de operaciones de sincronización y comunicación por el tiempo que éstas requieran. El término $2 \cdot B$ se debe a que se forman B bloques en la tabla y cada bloque necesita de dos procesos de sincronización (barreras): la primera para pasar de triángulos inferiores a triángulos superiores y la segunda es el caso inverso.

Bloques no homogéneos para el algoritmo $O(n^4)$

La derivación del particionamiento no homogéneo para el algoritmo $O(n^4)$ es similar al caso anterior. Sin embargo, para determinar el costo computacional por cada entrada de la diagonal actual para el algoritmo $O(n^4)$ se debe analizar el número de operaciones que realiza cada función G, G_1 y C (ecuación 1.10, 1.11 y 1.12).

El número de operaciones que realiza cada función se muestra a continuación:

Tabla 2.1: Costo computacional de cada función.

Función	Operación	Costo computacional
G	$\min\limits_{i \leqslant h < j}[G(i,h) + G(h+1,j)]$	$2d(n-d+1)$
G_1	$\min\limits_{i \leqslant h < j}[G_1(i,h) + G(h+1,j), G(i,h) + G_1(h+1,j))]$	$2d(n-d+1)$
C	$\min\limits_{i+1 \leqslant h < j-1}[G_1(i+1,h) + G_1(h+1,j-1)] + e(i,j)$	$2(d-2)(n-d+1)$
	$\min\limits_{i < p_1 < q_1 < j}[f_2(i,j,p_1,q_1) + C(p_1,q_1)]$	$\frac{(d-1)d}{2}(n-d+1)$

Las funciones G y G_1 tienen un comportamiento similar a la función F del algoritmo $O(n^3)$ y necesita cada una de $2d(n-d+1)$ operaciones. Al llamar a la función C si se formó un par de orden 2 el número de operaciones que se necesitan es $2(d-2)(n-d+1)$. Al evaluar el tercer caso de la función C el número de operaciones necesarias es $\frac{(d-1)d}{2}(n-d+1)$ y entonces el

costo computacional para cada entrada de la tabla de la función C es la suma de estos términos tal y como se muestra en la ecuación 2.11.

$$(2(d-2)(n-d+1)+\frac{(d-1)d}{2}(n-d+1)) \tag{2.11}$$

A partir de la suma de las dos primeras ecuaciones de la tercera columna de la tabla 2.1 y de la ecuación 2.11 se obtiene la función de costo computacional que calcula la diagonal d para el algoritmo $O(n^4)$ y ésta se expresa en la siguiente ecuación.

$$h(d) = 2d(n-d+1)+2d(n-d+1)+2(d-2)(n-d+1)+\frac{(d-1)d}{2}(n-d+1) \tag{2.12}$$

Entonces:

$$h(d) = \frac{-d^3}{2}+(\frac{n}{2}-5)d^2+(\frac{11n}{2}+\frac{3}{2})d-4n-4 \tag{2.13}$$

La función $h(d)$ se puede visualizar en la Figura 2.14, que indica que las diagonales que toman mayor tiempo computacional están localizadas en $2/3$ de la tabla.

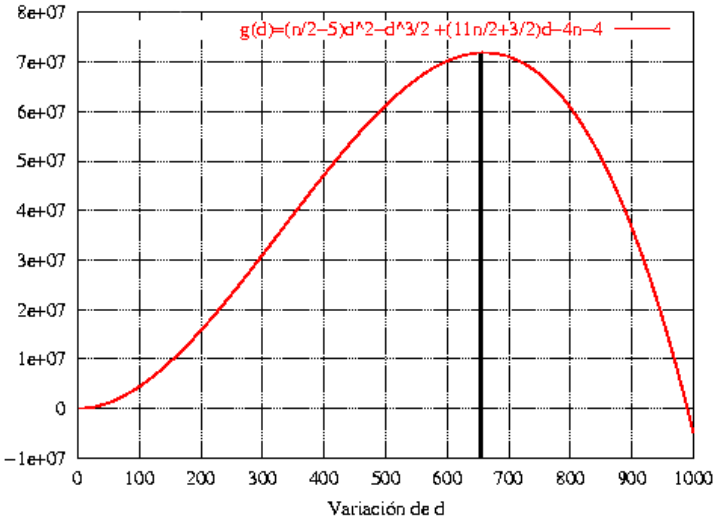


Figura 2.14: Costo computacional por cada entrada en la matriz, para el algoritmo $O(n^4)$.

Se procede de la misma forma que en el algoritmo $O(n^3)$, para determinar el valor de la diagonal siguiente d_{k+1} a partir de la diagonal actual d_k. La ecuación general para el algoritmo $O(n^4)$ se muestra en la ecuación 2.14.

$$\frac{-[(B-(k+1))+1]d_{k+1}^4}{8}+\frac{[B-(k+1)]d_k^4}{8}+$$
$$[\frac{n}{6}-\frac{5}{3}][(B-(k+1))+1]d_{k+1}^3-[\frac{n}{6}-\frac{5}{3}][B-(k+1)]d_k^3+$$
$$[\frac{11n}{4}+\frac{3}{4}][(B-(k+1))+1]d_{k+1}^2-[\frac{11n}{4}+\frac{3}{4}][B-(k+1)]d_k^2-$$
$$(4n+4)[(B-(k+1))+1]d_{k+1}+(4n+4)[B-(k+1)]d_k-\frac{n^4}{24}-\frac{13n^3}{12}+\frac{13n^2}{4}-4n=\ 0 \quad (2.14)$$

2.5 Esquemas de comunicación

Los dos tipos de comunicación que se emplean en la paralelización de los algoritmos $O(n^3)$ y $O(n^4)$ se describen a continuación.

2.5.1 Maestro-esclavo o comunicación global

Este tipo de comunicación consiste en la implementación de un componente principal llamado *maestro* que se encarga de recolectar la información procesada por cada *esclavo* (componente secundario) y de distribuirla en su totalidad a cada uno de ellos. Permite el procesamiento en paralelo, pues cada esclavo trabaja independientemente de los demás. También, se le conoce como comunicación global, porque la información se concentra en un procesador (maestro). La Figura 2.15 muestra el comportamiento de este esquema de comunicación.

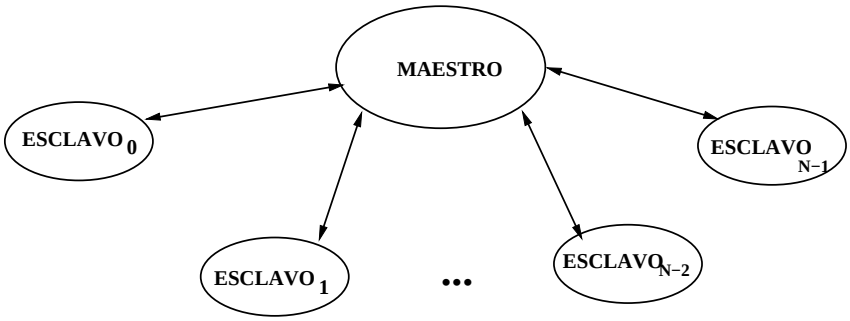


Figura 2.15: Esquema Maestro-Esclavo.

2.5.2 Esquema SPMD o comunicación local

Este esquema utiliza el modelo Single Program Multiple Data (Programas Simples Múltiples Datos). Se escribe únicamente un programa y todos los procesadores ejecutarán el mismo

programa. Múltiples datos (MD) se refiere a que los datos se dividen en pedazos, y se le asigna un pedazo a cada procesador.

A diferencia de la comunicación global, al aplicarse este esquema a las estrategias de particionamiento, no existe un proceso maestro. La comunicación del procesador N es sólo con sus vecinos más cercanos, es decir, con el $N-1$ y el $N+1$ si no es el procesador 0 o el último. Si es el procesador 0 éste sólo se comunica con el procesador 1 y el último procesador sólo se comunica con el anterior.

Una de las características principales para la aplicación eficientemente del paralelismo es que no deben existir dependencias de datos entre los procesadores, de lo contrario se tendría que usar sincronización para que la evaluación del problema sea correcto. Por lo tanto, se requiere tener una buena orquestación del problema a paralelizar.

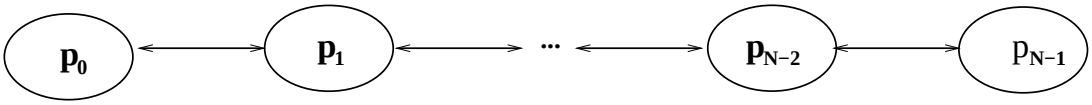


Figura 2.16: Esquema SPMD: Comunicación Local.

2.6 Trabajos relacionados

A continuación se discuten algunos algoritmos para el problema de RNA, que han sido diseñados para arquitecturas paralelas de memoria compartida y memoria distribuida.

En [12] (Jih-H *et al.*, 1998), se describen implementaciones paralelas de un algoritmo de programación dinámica para predecir la estructura secundaria del ácido ribonucleico (RNA) basado en la minimización de energía en computadoras de alta ejecución.

El algoritmo de programación dinámica calcula la mínima energía posible de una secuencia de RNA dada, al calcular sistemáticamente la mínima energía para cada fragmento de la secuencia. El algoritmo más riguroso y exacto necesita examinar todas las posibles estructuras que podrían formarse en la secuencia, lo cual implica que el algoritmo requiera cálculos del orden de 2^n para secuencias de n nucleótidos. El algoritmo que implementaron es de complejidad en tiempo $O(n^3)$. Los resultados fueron obtenidos en una supercomputadora CRAY de memoria compartida con ocho procesadores y también sobre una computadora MasPar de memoria distribuida con 16 384 procesadores. Se analizaron tres formas distintas para calcular cada valor de la tabla: columnas, renglones y diagonales, de las cuales sólo el cálculo por diagonales permite aplicar adecuadamente el paralelismo. Se implementaron dos versiones de este algoritmo vea la Figura 2.17, con dos formas distintas de calcular los elementos de la diagonal.

A pesar de que son equivalentes, la forma en que realizan la paralelización produce resultados diferentes. Nótese que en ambas implementaciones paralelizaron el ciclo *for* de la variable i. Consideraron dos formas de almacenar las tablas de programación dinámica en dos arreglos unidimensionales: en forma de renglones ($(i,j) \Rightarrow (n+1)(i-1)+j$) y en forma de diagonales ($(i,j) \Rightarrow (j-i)n+i$). Al hacer las pruebas en cada computadora, y una combinación de las dos implementaciones y de la forma de almacenamiento, el método dos (segunda implementación

Tipo A

```
do k=1 to n-1
 do i=1 to n
  j=k+i
  do kp=1 to k
   W(i,j)=min{W(i,j),
             W(i,kp+i-1)+W(kp+i,j)}
  enddo
 enddo
enddo
```

Tipo B

```
do k=1 to n-1
 do kp=1 to k
  j=k+i
  do i=1 to n
   W(i,j)=min{W(i,j),
             W(i,kp+i-1)+W(kp+i,j)}
  enddo
 enddo
enddo
```

Figura 2.17: Dos implementaciones del cálculo por diagonales. $W(i,j)$ es la función que calcula la energía óptima de la subsecuencia (i,j).

y almacenamiento por diagonales) resultó mejor que el método uno (primera implementación y almacenamiento por renglones) para el problema de RNA. Para reducir el costo de comunicación entre los procesadores, procesaron bloques de diagonales que están en función de la distancia de comunicación. Con excepción del primer bloque, todos los otros consisten del mismo número de diagonales.

A continuación se muestran los resultados obtenidos en este trabajo.

Ejecución en una computadora vector.

- Computadora: CRAY Y-MP, utiliza una arquitectura de memoria compartida y dispone de 8 procesadores (Tabla 2.2). Observe que los resultados obtenidos en el método 2 utilizando el almacenamiento por diagonales es mejor que el método 1 utilizando cualquier forma de almacenamiento de datos, y esto se debe principalmente a la forma en como se paraleliza el ciclo $for\ i$.

Tabla 2.2: Tiempo de comparación de cuatro combinaciones diferentes de métodos computacionales y estrategias de colocación de datos en el cálculo de la tabla W (tiempo en segundos)

Longitud	Método tipo-1		Método tipo-2	
Secuencia	Reglones	Diagonales	Diagonales	Reglones
512	5.55	8.57	3.83	6.35
1000	37.8	49.1	28.2	36.3
1024	40.0	63.6	30.2	50.3
2000	286.0	397.0	226.0	318.0
2048	302.0	488.0	240.0	400.0
2049	412.0	504.0	243.0	295.0

W es la ecuación de recurrencia que resuelve el problema de RNA.

- Sistema de un solo procesador, los resultados se muestran en la Tabla 2.3.

Tabla 2.3: Tiempo de comparación del método 1, el método 2 y el algoritmo no optimizado en una Cray Y-MP/1 (tiempo en segundos)

Longitud	Método 1	Método 2	No optimizado
433	28	22	316
1065	244	200	1898
1542	652	532	6581
2871	3680	2970	32664

- Sistema de múltiple procesador, los resultados se muestran en la Tabla 2.4.

Tabla 2.4: Tiempo y aceleración del método 1 y el método 2 usando 1, 2, 3 y 8 procesadores de la Cray Y-MP/8 (tiempo en segundos)

Número	Reloj de pared		Aceleración	
Procesadores	Método 1	Método 2	Método 1	Método 2
1	244.0	200.0	1.00	1.00
2	123.7	102.4	1.97	1.95
4	64.4	55.6	3.79	3.60
8	36.0	33.0	6.78	6.06

La longitud de la secuencia es 1065.

Ejecución en un sistema de procesamiento masivamente paralelo.

- MasPar MP-2 SIMD, 16 384 procesadores, los resultados se muestran en la tabla 2.5. Estos resultados indican que el uso de bloques disminuye considerablemente el tiempo de ejecución.

Las principales características del trabajo realizado en este artículo y el trabajo que se presenta en este documento son:

- Los dos trabajos se basan en la estrategia de programación dinámica y en el artículo descrito por Sankoff en 1983 [3]. El algoritmo implementado en ambos, es de orden $O(n^3)$.
- Este artículo y este documento realizan la evaluación del problema en orden por diagonales, pero la forma de almacenar los resultados es diferente. En este trabajo se utiliza una tabla triangular superior y se evalúa el problema en forma diagonal. En el artículo se almacenan los cálculos en arreglos de redireccionamiento en forma de diagonales y en forma de renglones.

Tabla 2.5: Tiempo y distancia de comunicación total para varios tamaños de bloques en un sistema MasPar MP-2 (tiempo en segundos)

Longitud	Tiempo CPU	Costo de Comunicación	Razón
n=1065			
1	437.0	200 758 180	100.0000
5	335.8	41 198 628	20.5215
10	323.5	22 510 804	11.2129
20	318.4	15 250 692	7.5965
27	317.9	14 654 990	7.2998
30	317.5	14 669 588	7.3071
50	318.5	17 510 381	8.7221

La ultima columna es el porcentaje de costo de comunicación comparado con el tamaño de bloque 1.

- Los dos trabajos utilizan el procesamiento de bloques de diagonales para reducir los costos de comunicación pero en el artículo no se menciona la forma en que se distribuyen los cálculos de cada bloque entre los procesadores. En este documento se divide en bloques y a su vez a cada procesador le corresponde una sección del bloque que se está procesando en ese momento.

- En el artículo, para determinar el número de bloques se toma en cuenta la distancia de la comunicación entre los procesadores. En este documento de tesis, el número de bloques es un parámetro determinado por el usuario y con base en el número de operaciones requeridas para evaluar cada entrada de la tabla se determina la longitud del bloque (particionamiento por bloques no homogéneos).

En [13] (Shapiro *et al.,* 2001), se describe la implementación de una algoritmo genético masivamente paralelo. Los operadores básicos (cruza, mutación y selección) que utiliza se describen a continuación: Utiliza un banco de stems (es una serie contigua de pares base, y se describe por una 3-tupla que incluye la posición inicial 5', la posición final 3' y el número de pares base en el stem). Los stems son generados desde la secuencia de RNA (representación) y cada procesador almacena una estructura de RNA (población de individuos). La energía mínima 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, P_1 y P_2, con mejor energía. Esta selección se hace desde un conjunto de 9 estructuras, originarias de 8 procesadores vecinos y la del mismo procesador.

El operador mutación que utiliza es el de recocido (annealing) que depende de la longitud de la secuencia. Para aplicar el operador de cruza, se seleccionan dos hijos del banco de stems C_1 y C_2. En cada procesador el AG realiza la operación de cruza entre (P_1, P_2) y (C_1, C_2) distribuyendo stems de P_1 y P_2 en C_1 y C_2 para formar dos nuevas estructuras. El AG elige la estructura hijo con mejor energía, que será la nueva estructura del procesador para la siguiente generación. Si el promedio ponderado de energía es menor que un valor de incertidumbre,

entonces el AG converge y termina. La estructura que más procesadores posea, se considera la solución que el AG ha generado para una secuencia en particular. Para reducir el costo de comunicación entre los procesadores utilizaron procesadores virtuales, es decir, si se tienen 16 procesadores físicos se pueden tener 2048, 4096, 8192, etc., procesadores virtuales.

Las pruebas de este algoritmo genético fueron hechas en una MasPar MP-2 con 16384 procesadores físicos, en la SGI ORIGIN 2000 con 64 procesadores y en la CRAY T3E con 512 procesadores. Las secuencias de prueban fueron: PSTVd (Potato Spindle Tuber Viroid) con 359 nucleótidos de longitud y el virus del polio 1, 2 y 3 con 742 nucleótidos de longitud.

La convergencia del AG se incrementa con el tamaño de la población, es decir, entre más grande sea la población (más variedad de soluciones) el algoritmo encontrará una solución óptima. Esto provoca que el algoritmo sea más lento, al buscar una solución adecuada entre toda la población, más aun si se incrementa el número de generaciones.

Las principales diferencias de este trabajo de tesis con este artículo son:

- La estrategia de solución es diferente, el que se utiliza ya que en ese artículo se usan algoritmos genéticos y nosotros usamos programación dinámica para este trabajo de tesis.
- La forma de encontrar la solución es distinta, ellos utilizan un banco de stems para encontrar una solución óptima y en este documento se crea la solución óptima al utilizar las ecuaciones de recurrencia.
- La estrategia de algoritmos genéticos puede encontrar un conjunto de soluciones, la estrategia de programación dinámica sólo encuentra una solución.
- Al utilizar los procesadores ellos utilizan procesadores físicos y procesadores virtuales y en este documento sólo se utilizan procesadores físicos.

Capítulo 3

Análisis de resultados experimentales

Una de las mayores dificultades que enfrentan los problema de complejidad exponencial es el tiempo de ejecución. Entre más grande sea el problema, el tiempo aumenta considerablemente. Aun en el caso de los algoritmos descritos en esta tesis que tienen complejidad polinomial, $O(n^3)$ y $O(n^4)$, por el grado del polinomio los tiempos de ejecución tienden a crecer rápidamente al incrementar el tamaño del problema. El propósito de la programación paralela es disminuir el tiempo de ejecución. Este capítulo muestra los resultados experimentales de la aplicación de las estrategias de particionamiento discutidas en el capítulo anterior, a los algoritmos $O(n^3)$ y $O(n^4)$. Se analizan resultados sobre dos tipos de sistemas paralelos (memoria compartida y memoria distribuida) obtenidos a partir de dos tamaños diferentes: 1500 y 3000 bases.

La infraestructura que se utilizó es dos computadoras de multiprocesamiento denominadas, elvis.cs.cinvestav.mx (148.247.2.64) y presley.cs.cinvestav.mx (148.247.2.58) ubicadas en el laboratorio de paralelismo de la sección de computación. Esta computadoras tienen las siguientes características:

- 4 procesadores Intel Xeon de 750 MHz (elvis.cs.cinvestav.mx) y 550 MHz (presley. cs. cinvestav. mx).
- Disco Duro SCSI de 28 GB.
- 1 GB de Memoria RAM.

Se utiliza el sistema operativo Linux Red Hat 7.1 con el kernel 2.4.2-2.

Las pruebas que se realizaron en estas computadoras se muestran en la tabla 3.1.

3.1 Resultados de estructuras secundarias

El problema de RNA al igual que la parentización óptima de matrices utiliza la estrategia de programación dinámica para su solución [6]. Tomando como base el balance de paréntesis que

Tabla 3.1: Reporte de resultados experimentales

Algoritmo	Memoria compartida	Memoria distribuida			Figura
		Maestro-Esclavo	SPMD	Maestro-Esclavo vs. SPMD	
$O(n^3)$	√				3.6
	√				3.7
	√				3.8
	√				3.9
		√			3.10
		√			3.11
		√			3.12
		√			3.13
			√		3.16
			√		3.17
			√		3.18
			√		3.19
				√	3.20
				√	3.21
				√	3.22
$O(n^4)$	√				3.24
	√				3.25
	√				3.26
			√		3.28
			√		3.29
			√		3.30

se proporciona para la construcción de la salida óptima de la parentización de matrices, se puede construir la estructura secundaria óptima que los algoritmos $O(n^3)$ y $O(n^4)$ originan.

El problema de RNA puede adaptarse adecuadamente, si se indica que al abrir y al cerrar un paréntesis se forma una unión (ciclo), independiente de las bases, para construir la estructura secundaria. La estructura secundaria óptima será el resultado del apareo de paréntesis.

A continuación se muestran las estructuras secundarias que se obtuvieron a partir de los algoritmos secuenciales $O(n^3)$ y $O(n^4)$.

La Figura 3.1 muestra un ejemplo de estructura secundaria propuesta por el algoritmo $O(n^3)$ y la Figura 3.2 muestra la estructura secundaria obtenida por el algoritmo $O(n^4)$. Esta estructura está reportada en [12] y se muestra en la Figura 3.3. Su estructura primaria es:

Estructura primaria= GGGCAAGUGGCGUAAUUGGUAGACGCAGCAGACUCAAAACCAG
CGAUGGUUCGAGUCCCACCUUGCCCACCA

La cadena de paréntesis que dio como solución el algoritmo $O(n^3)$ es: ((((((((((((((..)(((..)).)).)
.)((((((..(((....))))).)))).)))).)))..).)).
Los puntos indican que las bases que se encuentran en esa posición no están apareadas. La cadena de paréntesis que da como resultado el algoritmo $O(n^4)$ es: ((((((((((.....((((..)).))))
.(((((((..(((....))))).))))))..)).)))..).)).

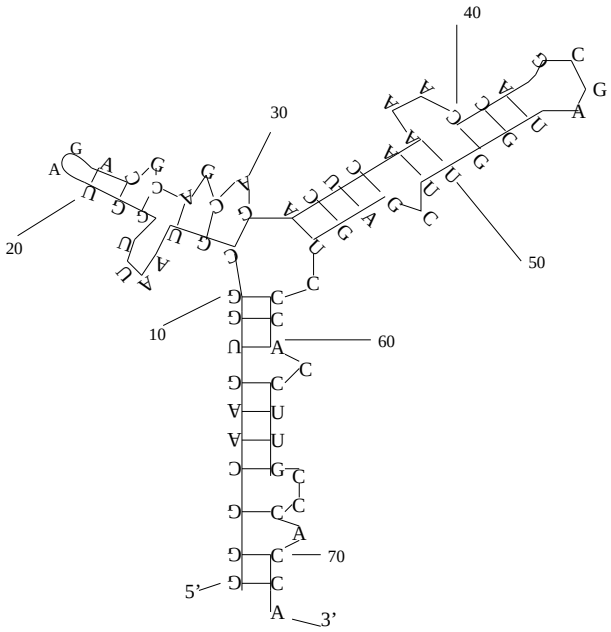


Figura 3.1: Propuesta de una estructura secundaria proporcionada por el algoritmo $O(n^3)$.

Observe que de las tres figuras, la estructura que dió como resultado el algoritmo $O(n^4)$ presenta un mayor número de ciclos y evidentemente la energía asociada es mucho menor que la energía que proporcionan las otras dos estructuras.

La Figura 3.4 muestra un ejemplo de estructura secundaria obtenida por los dos algoritmos secuenciales y la Figura 3.5 muestra la estructura reportada en [33] para el SV11 RNA. La estructura primaria para esta estructura secundaria es:

Estructura primaria= GGGCACCCCCCUUCGGGGGGUCACCUCGCGUAGCUAGCUACG
CGAGGGUUAAAGGGCCUUUCUCCCUCGCGUAGCUAACCACGCGAGGUGACCCCC
CGAAAAGGGGGGUUUCCCA

La cadena de paréntesis que proporcionaron los dos algoritmos secuenciales es: (((.((((((((((
((((((((((((((((.(.((((((((((((((..((((...))))..))))))))))))))).).)))))))))))))))..)))))))))..))).

La estructura secundaria que propone el algoritmo $O(n^3)$ es la misma que origina el algo-

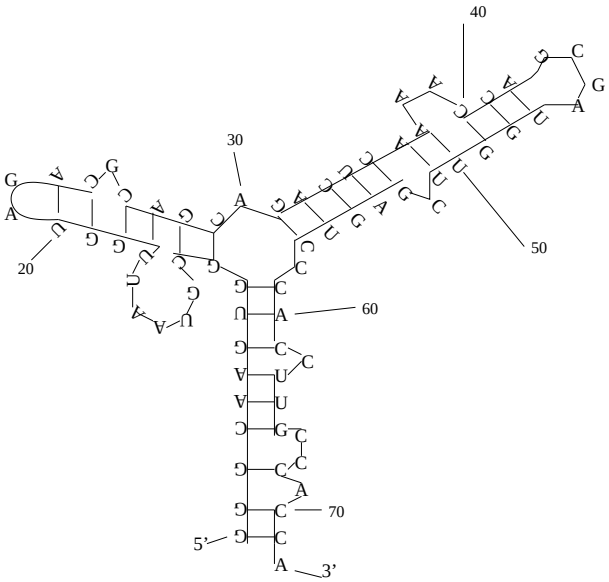


Figura 3.2: Propuesta de una estructura secundaria proporcionada por el algoritmo $O(n^4)$.

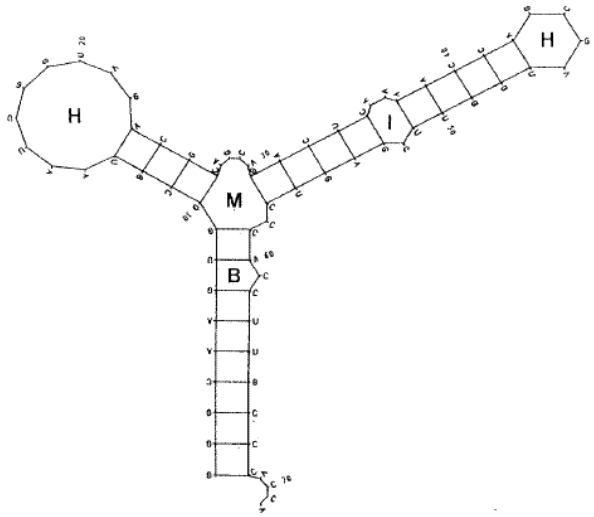


Figura 3.3: Estructura secundaria, [12].

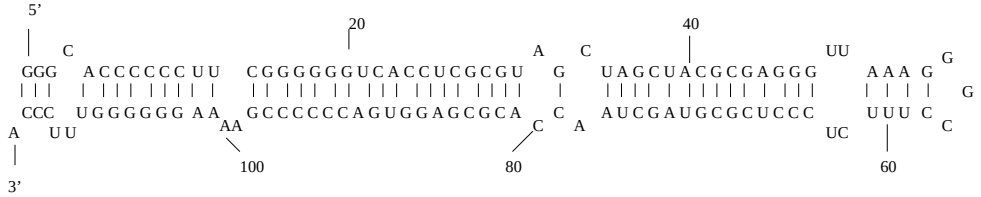


Figura 3.4: Propuesta de una estructura secundaria para el SV11 RNA, resultado de ambos algoritmos.

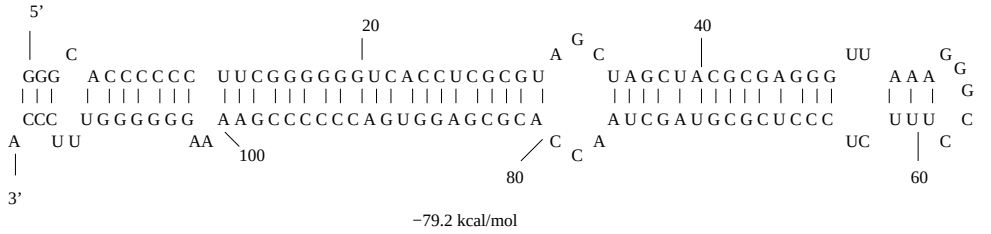


Figura 3.5: Estructura estable de SV11 RNA, [33].

ritmo $O(n^4)$ para el SV11 RNA (Figura 3.4) y al compararla con la estructura propuesta en la Figura 3.5 se observa que la primera tiene un ciclo de más que la segunda, es decir, ambas son muy parecidas pero en una de ellas (en la primera) se forma un ciclo en la posición (33,79) mientras que en la otra no existe unión. Esto nos indica que la energía de la estructura secundaria que se propone por los algoritmos $O(n^3)$ y $O(n^4)$ es más estable (tiene menor energía) que la que se reporta por el algoritmo genético de [33].

3.2 Algoritmo $O(n^3)$

En esta sección se muestran los resultados experimentales obtenidos para memoria compartida y memoria distribuida para el algoritmo $O(n^3)$.

3.2.1 Memoria compartida

En los sistemas de memoria compartida, cualquier dirección de memoria es accesible por cualquier procesador. Comparten el mismo espacio de direccionamiento, el cual es como una dirección única a cada localidad de memoria. Debido a esto, la comunicación es implícita y se requieren mecanismos de sincronización (implícitos o explícitos). En esta sección los algoritmos desarrollados (diagonal, bloques homogéneos y bloques no homogéneos) fueron implementados utilizando las bibliotecas de pthreads para C.

Se tomaron los divisores exactos de n (longitud de la secuencia) para el número de bloques, porque mostraron un mejor comportamiento en tiempo de ejecución que cualquier otro valor asignado de manera aleatoria. El número de hilos es igual al número de procesadores (h) y el tiempo de ejecución está medido en minutos. Para las pruebas se usó una computadora de multiprocesamiento (elvis) con cuatro procesadores físicos, la cual está disponible en el laboratorio de paralelismo. La longitud de la secuencia empleada para realizar las pruebas es de 3000 bases, todas las secuencias son generadas de manera aleatoria y cada resultado de las tablas y figuras es el promedio de 5 pruebas. El tiempo del algoritmo secuencial para 3000 bases es de 7.105554 minutos, tiempo obtenido en la computadora elvis. Las tablas de resultados se muestran en el apéndice que se encuentra al final de este documento. En esta sección sólo se muestran de manera visual los resultados obtenidos.

La Figura 3.6 permite visualizar los resultados obtenidos por cada uno de los algoritmos implementados variando el número de bloques (divisores exactos de n), empleando cuatro hilos para los algoritmos de particionamiento. De los dos tipos de particionamiento, el de bloques homogéneos muestra mejores resultados, minimizando el tiempo obtenido por el algoritmo secuencial. Esta disminución nos indica que el uso de bloques disminuye considerablemente el tiempo de ejecución.

Las Figuras 3.7 y 3.8 muestran los tiempos de ejecución variando el número de bloques (divisores) que proporcionaron los mejores resultados en bloques homogéneos y no homogéneos. Al variar el número de hilos para cada número de bloque se observa que el tiempo de ejecución disminuye considerablemente al compararlo con los resultados obtenidos por el particionamiento por diagonales. En estas figuras se puede observar que el particionamiento por diagonales no ofrece buenos resultados en comparación con los resultados obtenidos por bloques homogéneos y no homogéneos. El número de procesadores físicos de la computadora de prueba es de cuatro, lo cual provoca que al aumentar el número de hilos en las pruebas, el tiempo de ejecución no disminuya debido a que interfieren mecanismos de comunicación y sincronización para simular el aumento de procesadores. Esto indica que los algoritmos por bloques (homogéneos y no homogéneos) son escalables. Es decir, conforme aumenta el número de procesadores físicos el tiempo de ejecución disminuye.

La Figura 3.9 ilustra el rendimiento de los procesadores (aceleración) aplicados a los particionamientos de bloques homogéneos y no homogéneos, comparándolo con la aceleración ideal y la aceleración obtenida por el particionamiento por diagonales. De acuerdo a esta figura, se puede observar que la aceleración de los mejores resultados, tanto para bloques homogéneos y no homogéneos, tienen un mejor rendimiento que la aceleración obtenida por el particionamiento por diagonales, acercándose a la aceleración ideal. El decremento en la aceleración de 4 a 8 hilos se debe a que la computadora de prueba solamente dispone de cuatro procesadores físicos, lo que provoca que el rendimiento se vea afectado cuando intervienen mecanismos de comunicación y sincronización para simular el aumento de hilos.

De acuerdo a las gráficas, se observa que los resultados obtenidos por estos algoritmos de particionamiento utilizando memoria compartida son mejores que los del programa secuencial. El algoritmo de bloques homogéneos presenta un mejor rendimiento que los otros dos algoritmos, al variar el número de bloques y de procesadores. En la variación del número de bloques en la Figura 3.6 se observa que el tiempo de ejecución tiende a disminuir conforme se va aumentado el número de bloques y después de $n/15$ se mantienen los resultados. Esto nos indica que existen valores óptimos de número de bloques y en este caso para memoria compartida los valores óptimos se encuentran en el rango de $n/15$ a $n/25$ donde n es la longitud de la secuencia y se aplica sobre divisores exactos de n. El uso de bloques disminuye considerablemente el tiempo de ejecución y esto se debe principalmente a que el tiempo de sincronización y comunicación es mucho menor que el número de operaciones de sincronización que se realizan en el particionamiento por diagonales. Recordemos que el número de operaciones de sincronización para los particionamientos por bloques es el número de bloques (B) y para el particionamiento por diagonales es la longitud de la secuencia (n).

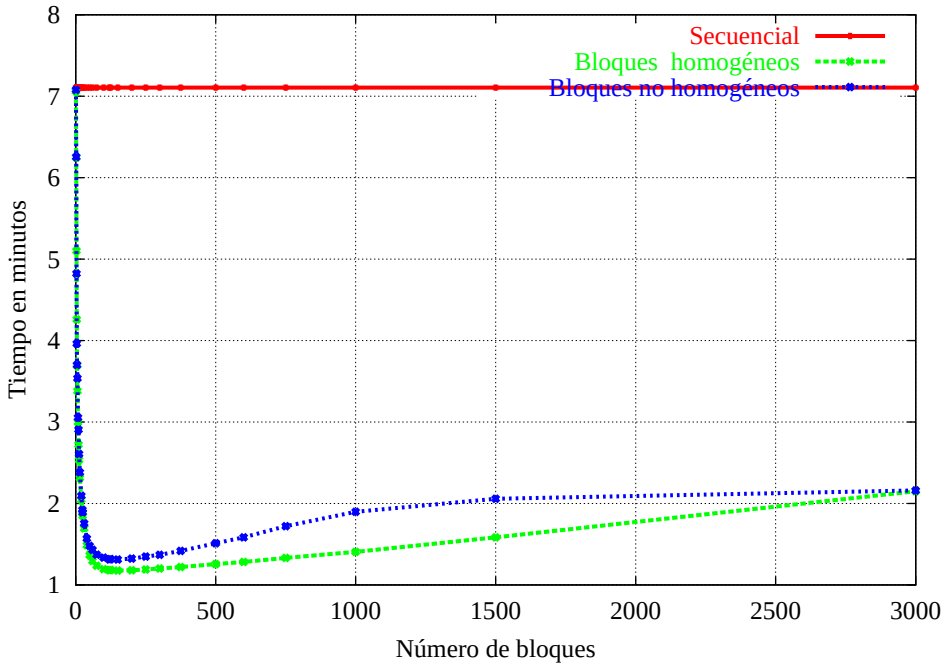


Figura 3.6: Variación del número de bloques para bloques homogéneos y no homogéneos.

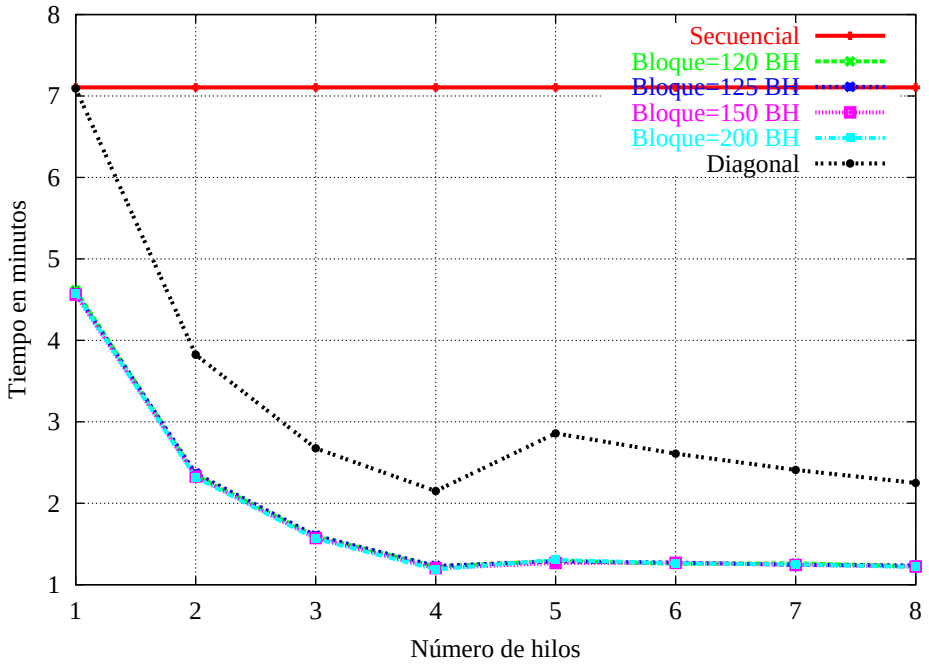


Figura 3.7: Variación del número de hilos para bloques homogéneos.

3.2.2 Memoria distribuida

En los sistemas de memoria distribuida, cada procesador tiene acceso a su propia memoria, así cuando los datos a usar por un procesador están en el espacio de direcciones de otro, es

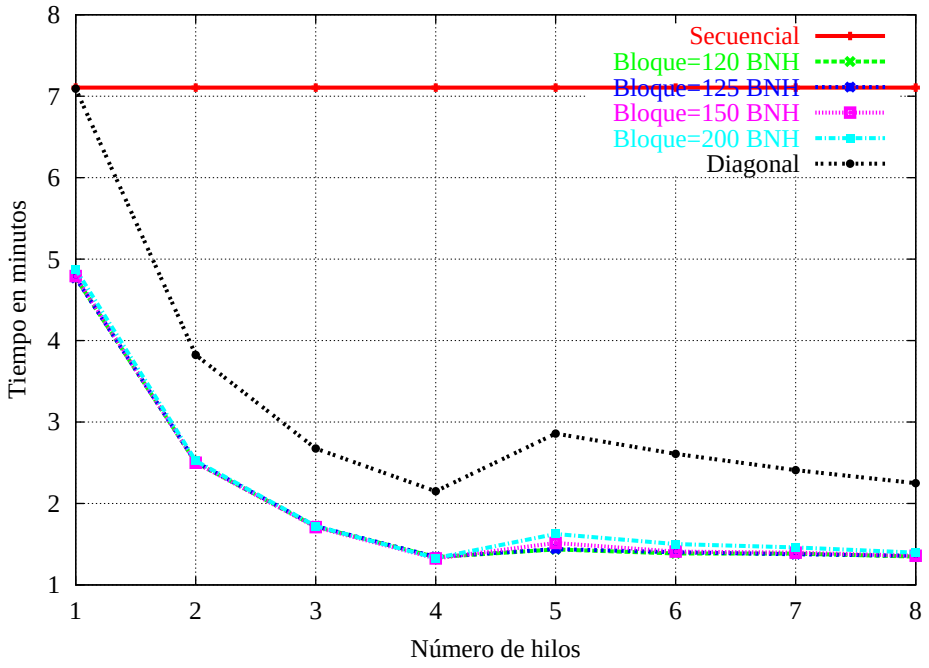


Figura 3.8: Variación del número de hilos para bloques no homogéneos

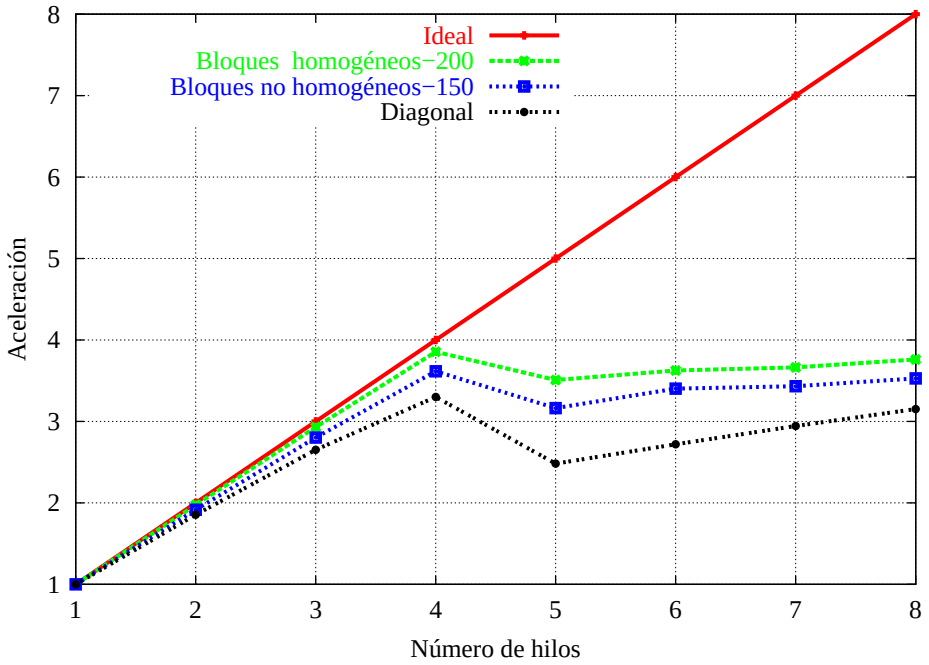


Figura 3.9: Aceleración.

necesario solicitarla y transferirla a través de mensajes. De esta forma, es necesario impulsar la localidad de los datos para minimizar la comunicación entre procesadores y obtener un buen

rendimiento.

Los programas implementados utilizan las bibliotecas de MPI en el lenguaje C. Se usaron dos enfoques diferentes de comunicación: comunicación global (modelo maestro-esclavo) y comunicación local (modelo SPMD). Esta sección muestra los resultados obtenidos al implementar las tres estrategias de particionamiento en ambos esquemas para el algoritmo $O(n^3)$.

Esquema Maestro-Esclavo o Comunicación Global

Este esquema consiste en la implementación de un componente principal llamado *maestro* que se encarga de recolectar la información procesada por cada *esclavo* (componente secundario) y de distribuirla en su totalidad a cada uno de ellos. Es un esquema muy común pero presenta la desventaja de que necesita de por lo menos 2 procesadores (un maestro y un esclavo) para solucionar el problema.

Al igual que en memoria compartida, se tomaron los divisores exactos para el número de bloques, porque mostraron un mejor comportamiento en tiempo de ejecución que cualquier otro valor aleatorio. Cada resultado de las tablas que se encuentran en el apéndice de resultados y de las figuras son un promedio de 5 pruebas realizadas para secuencias de $n = 3000$ bases generadas de manera aleatoria. El tiempo de ejecución está medido en minutos. El resultado del algoritmo secuencial es de 10.050317 minutos y se obtuvo en la computadora presley.cs.cinvestav.mx.

La Figura 3.10 muestra los resultados obtenidos por cada uno de los algoritmos implementados variando el número de bloques (divisores exactos de n); se utilizaron cuatro procesadores para obtener estos resultados. En esta figura se logra apreciar el decremento del tiempo del algoritmo secuencial al utilizar uno de los algoritmos de particionamiento paralelo. De los dos particionamientos por bloques, el de bloques homogéneos presenta un mejor rendimiento. De acuerdo a la figura es evidente un rango de números de bloques óptimos. En las Figuras 3.11 y 3.12 se muestran los divisores que proporcionaron los mejores resultados en bloques homogéneos (BH) y no homogéneos (BNH) para secuencias de 3000 bases, variando el número de procesadores para cada número de bloque. En estas figuras se observa que el particionamiento por bloques homogéneos presenta mejores resultados que los particionamientos por bloques no homogéneos y diagonales. El decremento de dos a tres procesadores en el particionamiento por diagonales es mayor que en los particionamientos por bloques, porque para obtener el tiempo de ejecución con dos procesadores se realizaron más operaciones de sincronización y el cálculo del problema sólo lo realizó el procesador esclavo; a diferencia del particionamiento por bloques homogéneos, el número de bloques provoca que se efectúen menos operaciones de sincronización lo que facilita que se obtengan resultados más inmediatos. El algoritmo por bloques homogéneos es mejor que el algoritmo por bloques no homogéneos porque en el primero se realizan menos operaciones de sincronización mientras que en el segundo se realizan el doble. El número de operaciones de sincronización para bloques homogéneos es el número de bloques (B) y en bloques no homogéneos se realizan dos operaciones de sincronización por cada bloque ($2B$). A esto se debe que el algoritmo por bloques homogéneos presente un mejor

rendimiento.

El rendimiento de los procesadores (aceleración) se muestra en la Figura 3.13 para bloques homogéneos y no homogéneos comparándolos con la aceleración ideal y el particionamiento por diagonales. La evaluación de la aceleración para los particionamientos por bloques, se realizó de la siguiente manera: se tomó el tiempo de ejecución de dos procesadores del particionamiento por diagonales y se dividió entre los resultados obtenidos por los particionamientos por bloques al variar el número de procesadores (de dos a ocho). A esto se debe que la aceleración para el particionamiento por diagonales con dos procesadores sea de 1 y para los particionamientos por bloques sea mayor a 1.

De acuerdo a las figuras, los resultados obtenidos por estos algoritmos de particionamiento utilizando el esquema maestro-esclavo para memoria distribuida, proporcionan mejores resultados que el programa secuencial. A su vez, el algoritmo de bloques homogéneos presenta un mejor rendimiento que el algoritmo de bloques no homogéneos. Al variar el número de procesadores los mejores resultados se observan en el algoritmo de bloques homogéneos al compararlo con el algoritmo de la diagonal y el de bloques no homogéneos, y a su vez el algoritmo de bloques no homogéneos resulta ser mejor que el algoritmo de la diagonal pero no supera al algoritmo de bloques homogéneos.

Cuando se varía el número de bloques en la Figura 3.10 se observa que el tiempo de ejecución tiende a disminuir conforme se va aumentado el número de bloques y después de $n/10$ aumenta ligeramente. Esto se debe principalmente a que existe mayor comunicación y sincronización entre procesadores, porque los bloques se vuelven más finos, incrementando el número de mensajes. Esto nos indica que existen valores óptimos de número de bloques y en este caso para memoria distribuida en el esquema maestro-esclavo se encuentran en el rango de $n/20$ a $n/8$, donde n es la longitud de la secuencia.

La Tabla 3.2 muestra el tiempo de ejecución para secuencias más grandes. De acuerdo a estos resultados el tiempo obtenido por el algoritmo secuencial disminuye 3 veces al utilizar el algoritmo por diagonales. En este caso se utilizaron 4 procesadores y el número de bloques fue $(n-3)/10$ (n =longitud de la secuencia). Al comparar los resultados, el algoritmo por bloques homogéneos tiene un mejor rendimiento que los otros algoritmos para tamaños elevados o secuencias más grandes.

Tabla 3.2: Resultados para varios tamaños de problema.

Tamaño n	Secuencial	Diagonal	Bloques Homogéneos	Bloques No Homogéneos
403	0.012672	0.010037	0.006283	0.006645
3003	10.050317	3.374686	2.651768	2.811020
4003	25.283995	8.249938	6.611055	7.194827
5003	52.579041	16.455309	14.423708	15.264579
6003	100.184985	35.093009	30.456730	31.552643
7003	184.354199	65.436536	53.604272	65.311414

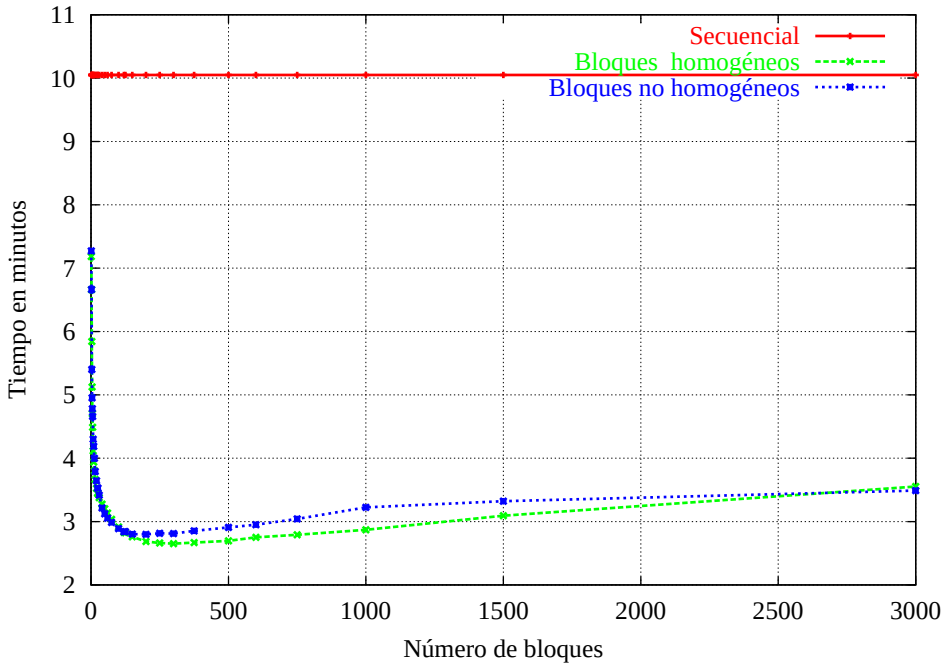


Figura 3.10: Variación del número de bloques para: bloques homogéneos y no homogéneos.

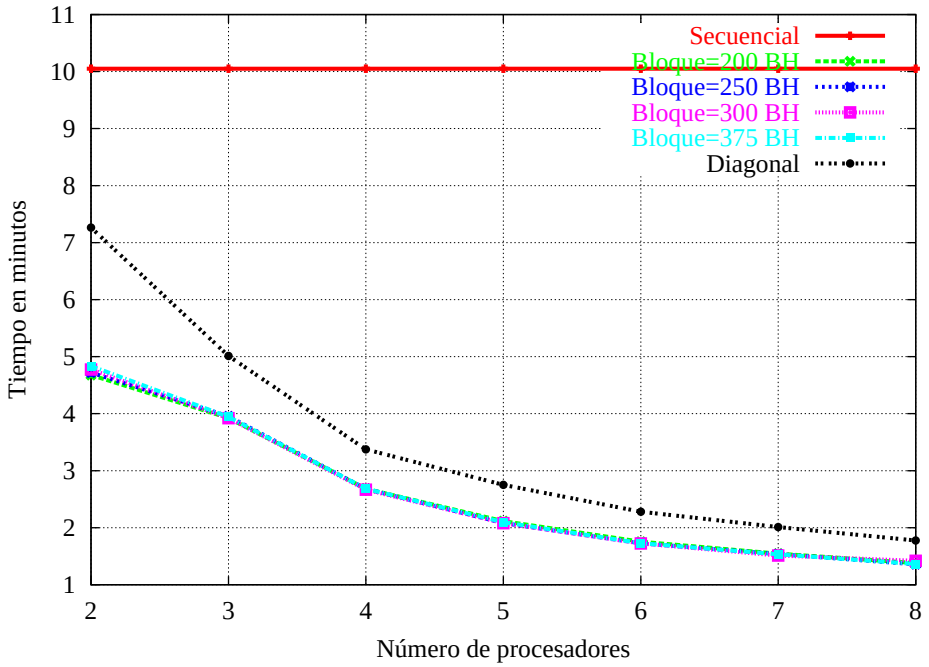


Figura 3.11: Variación del número de procesadores para bloques homogéneos.

Esquema SPMD o Comunicación Local

En este esquema no existe un procesador maestro que administre la información obtenida por cada esclavo, sino todo lo contrario, la comunicación del esclavo p es sólo con sus vecinos

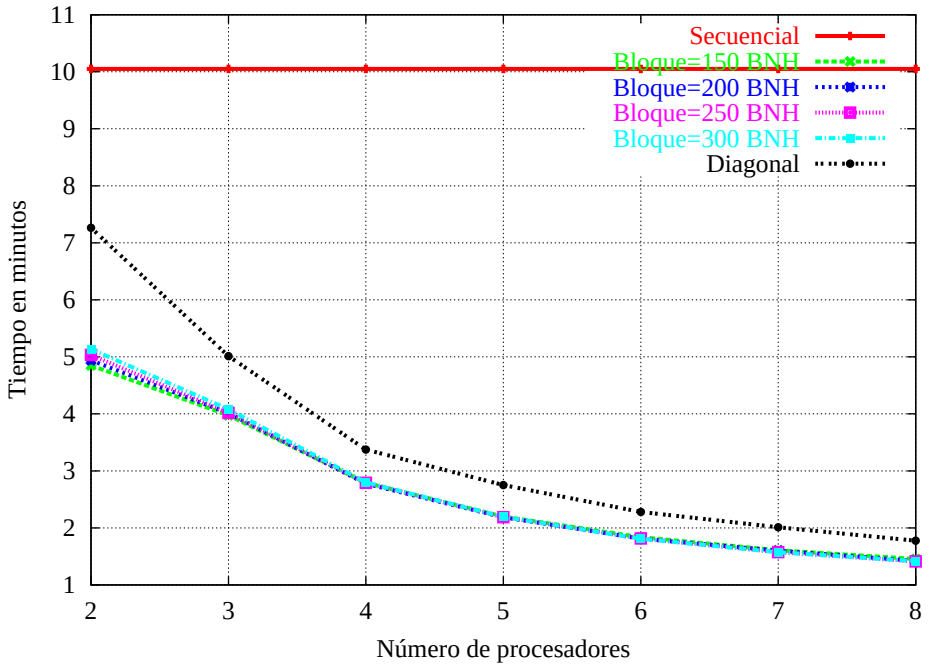


Figura 3.12: Variación del número de procesadores para bloques no homogéneos.

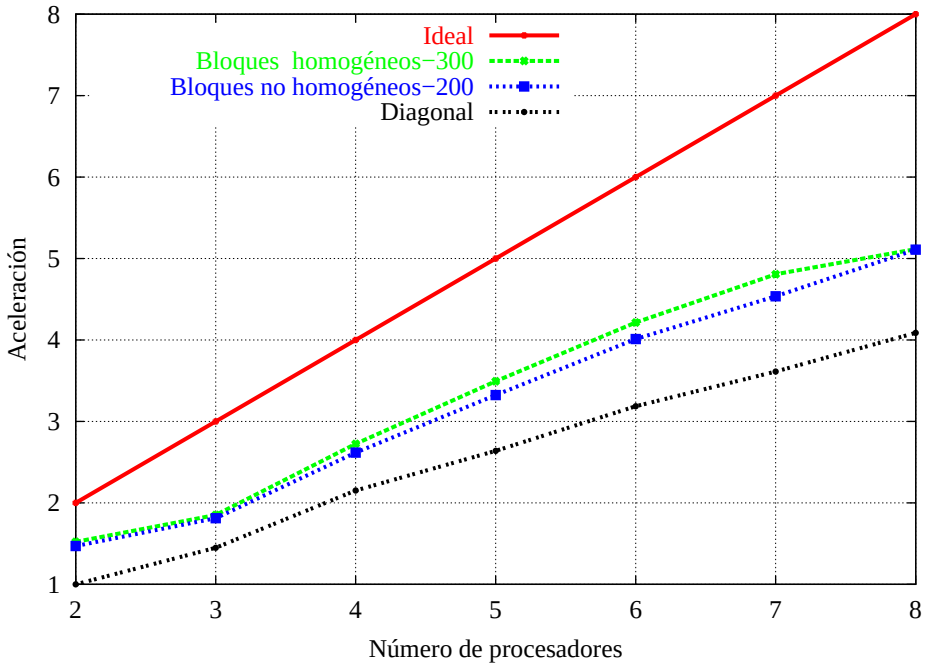


Figura 3.13: Aceleración.

con el $p-1$ y con el $p+1$. Sin embargo, si es el esclavo 0 éste sólo se comunica con el 1 o si el esclavo es el último procesador, éste sólo se comunica con el esclavo anterior.

La Figura 3.14 muestra la distribución de los cálculos entre los procesadores para este tipo de comunicación. El propósito es proporcionarle a cada procesador una sección de la diagonal, tratando de que todos tengan algo que calcular.

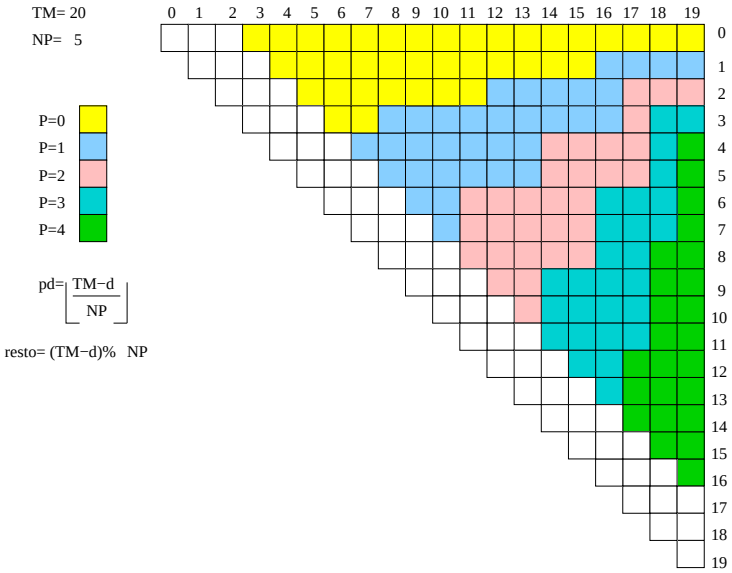


Figura 3.14: Comportamiento de la distribución de la tabla entre el número de procesadores, para el particionamiento por diagonales.

Para el particionamiento por diagonales y por bloques homogéneos este esquema ofrece un buen comportamiento, pero para el particionamiento por bloques no homogéneos se presentan dependencias entre procesadores. La Figura 3.15 muestra un ejemplo muy sencillo en el que al aplicar el particionamiento por bloques no homogéneos no da resultados correctos. La longitud de la secuencia para este problema es de 8 ($n = 8$), el número de bloques es de $B = 3$ y el número de procesadores es $nP = 5$, al distribuir el trabajo entre los procesadores en el primer bloque, a cada procesador le corresponde una celda de la diagonal actual porque la longitud del bloque es de 1. Al avanzar al segundo bloque la longitud de éste es de 2, por lo que sólo se forman 2 triángulos en la tabla y al distribuirla entre los procesadores sólo trabajan el procesador 0 y el procesador 1 (vea los colores del segundo bloque). Esto origina que el procesador 1 necesite de información anterior que en ese momento no tiene, provocando resultados erróneos.

A continuación se muestran los resultados experimentales de este esquema, para los tres tipos de particionamiento, pero es necesario indicar que debido a la orquestación del algoritmo de bloques no homogéneos no se puede probar para todas las combinaciones de bloques y número de procesadores, por lo que sólo para algunos casos los resultados son completamente correctos.

La longitud de la secuencia es de 3000 bases, generada de manera aleatoria. El tiempo del algoritmo secuencial es de 10.050317 minutos, obtenido en la computadora presley. Los resultados mostrados en las figuras es el resultado del promedio de 5 pruebas.

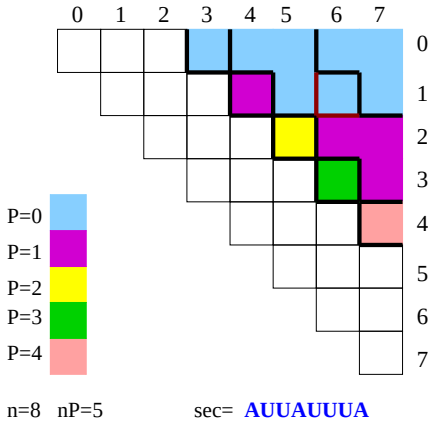


Figura 3.15: Comportamiento del particionamiento por bloques no homogéneos para la comunicación local.

La Figura 3.16 muestra los resultados obtenidos por cada uno de los algoritmos implementados variando el número de bloques (divisores exactos). Los algoritmos de particionamiento utilizan cuatro procesadores. En esta figura se puede observar que el algoritmo por bloques homogéneos ofrece mejores resultados que el algoritmo por bloques no homogéneos y que el algoritmo secuencial. De acuerdo a la figura existe un rango de números óptimos. En las siguientes figuras se muestra el comportamiento de estos números de bloque al variar el número de procesadores.

Las Figuras 3.17 y 3.18 muestran los divisores que proporcionaron los mejores resultados en bloques homogéneos y no homogéneos, variando el número de procesadores para cada número de bloque. Se puede observar que los algoritmos por bloques presentan un mejor rendimiento al incrementar el número de procesadores que el algoritmo por diagonales.

La aceleración se muestra en la Figura 3.19 para los tres particionamientos comparándolos con la aceleración ideal. En esta figura se observa que la aceleración de los particionamientos por bloques homogéneos y no homogéneos es mejor que la aceleración de la diagonal. Estos resultados nos indican que los algoritmos por bloque (principalmente el de bloques homogéneos) son más escalables, es decir, el tiempo de ejecución disminuye conforme el número de procesadores aumenta.

Una característica importante de este esquema es que utiliza todos los procesadores, a diferencia del esquema maestro esclavo que dedica un procesador para el proceso maestro. En la Figura 3.16 el particionamiento por bloques no homogéneos supera al inicio al algoritmo secuencial, porque necesita de dos pasos de sincronización (uno para triángulos superiores y el otro para triángulos inferiores), desperdiciando tiempo de ejecución, pero al ir aumentando el número de bloques, se disminuye el tiempo.

De acuerdo a los resultados obtenidos se observa que al igual que en la comunicación global existe un rango de número de bloques óptimo y en este caso para la comunicación local este rango o intervalo es de $n/12$ a $n/4$.

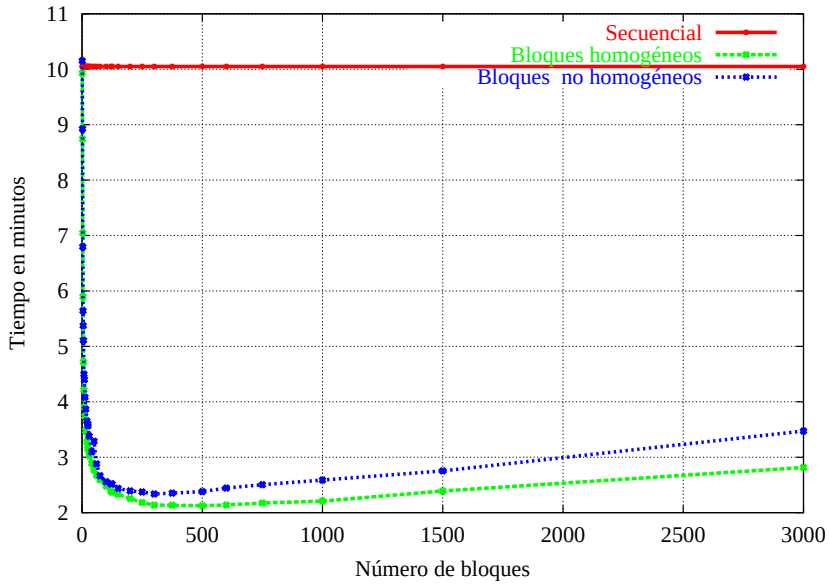


Figura 3.16: Variación del número de bloques para bloques homogéneos y no homogéneos.

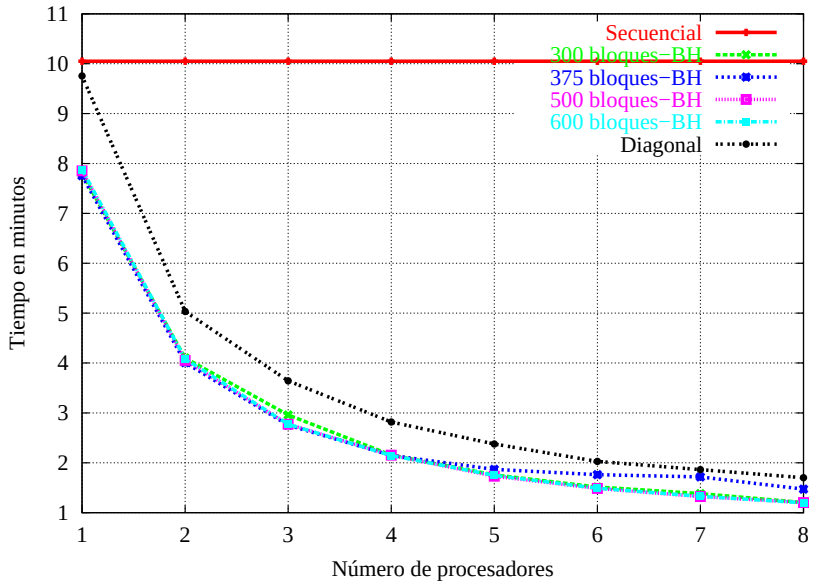


Figura 3.17: Variación del número de procesadores para bloques homogéneos.

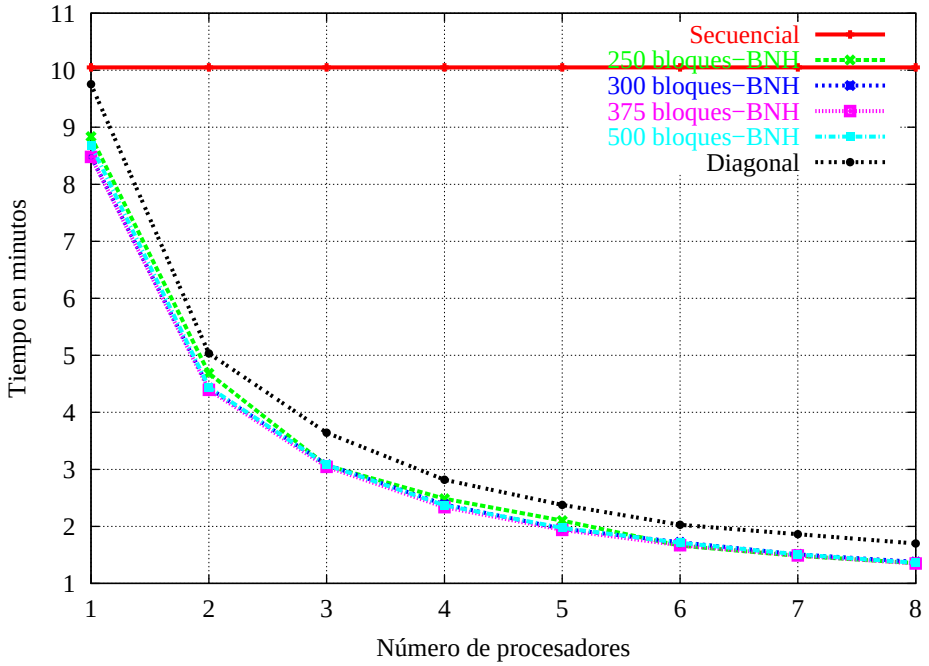


Figura 3.18: Variación del número de procesadores para bloques no homogéneos.

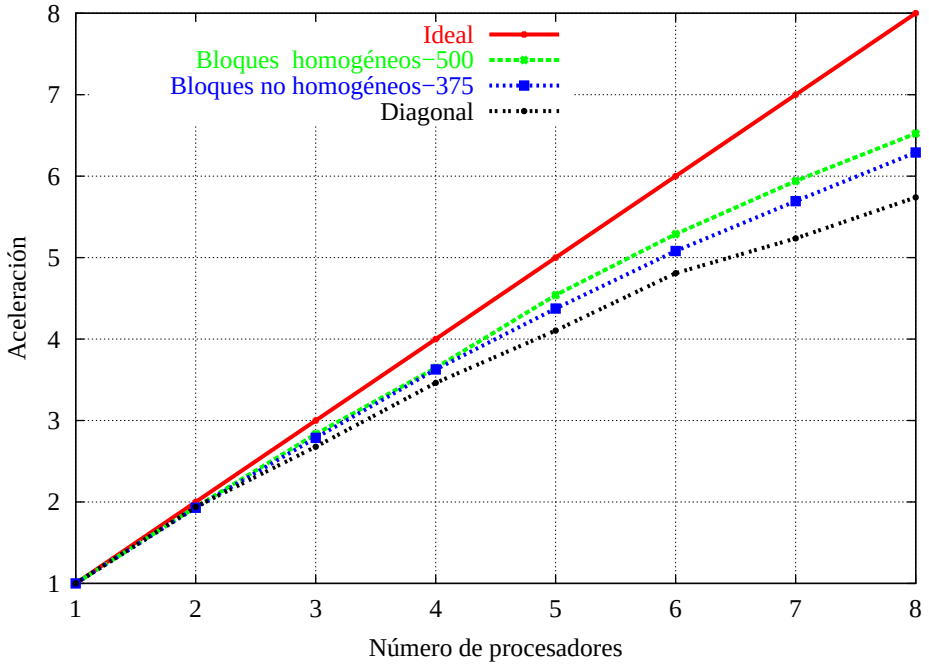


Figura 3.19: Aceleración de la comunicación SPMD.

Maestro-Esclavo vs SPMD

En esta sección se hace una breve comparación de los dos esquemas de comunicación implementados para memoria distribuida aplicados al algoritmo $O(n^3)$. Los resultados obtenidos muestran que el esquema SPMD obtiene mejores resultados que el esquema Maestro Esclavo (MaeEsc).

En la Figura 3.20 se compara los resultados obtenidos por cada esquema usando el particionamiento por bloques homogéneos, porque fue el algoritmo que tuvo un buen comportamiento, para 3000 bases utilizando cuatro procesadores. En esta figura es claro que el esquema SPMD ofrece un mejor desempeño que el esquema maestro-esclavo.

En la Figura 3.21 se compara el mejor número de bloque (divisor exacto de n) de cada esquema al usar el particionamiento por diagonales y bloques homogéneos para 3000 bases, variando el número de procesadores. En esta figura se muestra que el esquema SPMD para el particionamiento por bloques homogéneos tiene un mejor rendimiento que el particionamiento por diagonales SPMD, bloques homogéneos maestro-esclavo y diagonal maestro-esclavo.

La aceleración que se obtuvo se muestra en la Figura 3.22. Esta figura indica que el esquema SPMD es más escalable que el esquema maestro-esclavo; esto es el tiempo de ejecución disminuye al aumentar el número de procesadores.

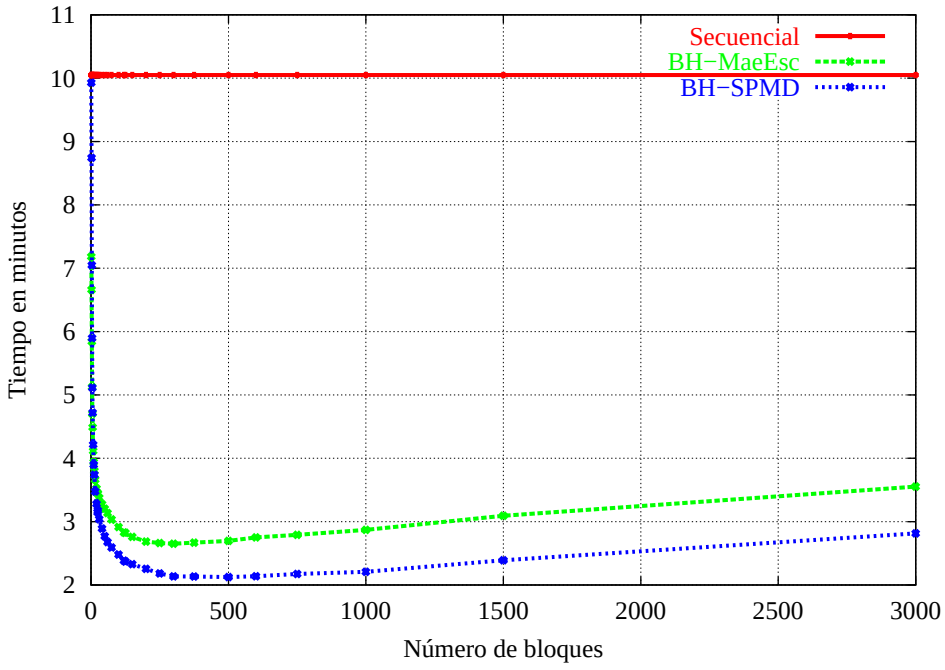


Figura 3.20: Resultados del tiempo de ejecución de bloques homogéneos para Maestro-Esclavo y SPMD.

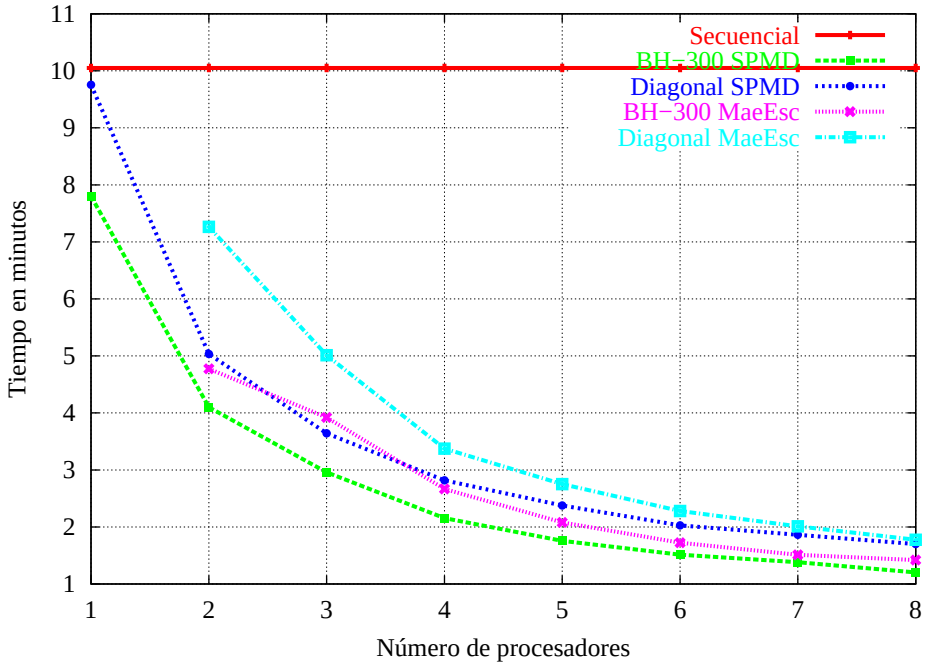


Figura 3.21: Resultados de los mejores divisores de número de bloques para Maestro-Esclavo y SPMD.

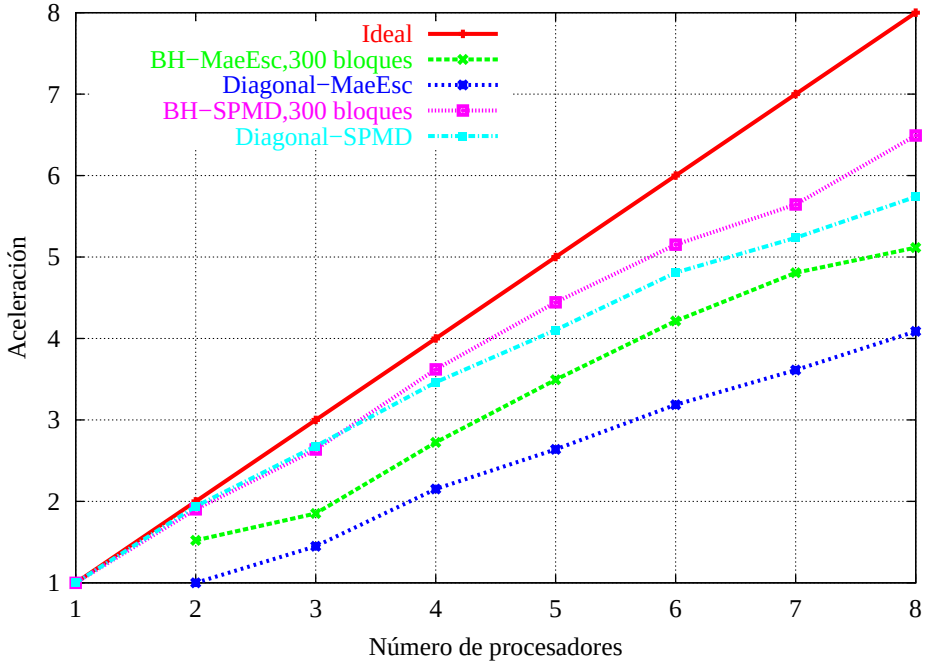


Figura 3.22: Aceleración de Maestro-Esclavo y SPMD.

3.3 Algoritmo $O(n^4)$

Partiendo de los mejores resultados obtenidos en el algoritmo $O(n^3)$ en ambos sistemas, sólo se presentan aquí los resultados para los particionamientos por diagonales y por bloques homogéneos. El esquema de comunicación implementado en memoria distribuida es SPMD. En esta sección se muestran los resultados experimentales obtenidos por el algoritmo $O(n^4)$, sólo para $n = 400$ bases.

3.3.1 Memoria compartida

La Tabla 3.3 muestra los resultados experimentales del programa secuencial, del algoritmo por diagonales y por bloques homogéneos variando la longitud de la secuencia n para memoria compartida. La Figura 3.23 muestra estos resultados; obsérvese que el tiempo de ejecución del algoritmo por diagonales disminuye considerablemente al compararlo con los resultados del programa secuencial y a su vez el programa por bloques homogéneos presenta mejores resultados que el algoritmo por diagonales. De acuerdo a estos resultados el tiempo de ejecución aumenta considerablemente conforme se va aumentando la longitud de la secuencia n. Puede verse que el tiempo de ejecución para 1500 bases es de 35.595785 minutos si se aplica el algoritmo por diagonales con 4 procesadores y para $n = 3000$ bases el tiempo de ejecución estimado por el algoritmo secuencial es de 2371.012582 minutos (39.516876 horas) y el tiempo estimado para el algoritmo por diagonales es de 13.17229212 horas para cuatro procesadores. Debido a que estos tiempos son extremadamente altos sólo se realizaron pruebas para 1500 bases.

La Figura 3.24 muestra los tiempos de ejecución del particionamiento por bloques homogéneos al variar el número de bloques para $n = 1500$ con cuatro procesadores ($h = 4$) comparándolo con el resultado obtenido por el programa secuencial. El número de bloques, son los divisores exactos de n, el número de hilos, h, es equivalente al número de procesadores. El tiempo del algoritmo secuencial es de 120.210612 minutos.

La Figura 3.25 muestra el tiempo de ejecución de los mejores números de bloques, variando el número de procesadores y comparándolos con los resultados del particionamiento por diagonales. Claramente, el particionamiento por bloques homogéneos tienen un mejor comportamiento que el particionamiento por diagonales.

La Figura 3.26 visualiza el rendimiento de los procesadores, mostrando la aceleración ideal, la aceleración del particionamiento por diagonales y por bloques homogéneos. Observe que el particionamiento por bloques homogéneos presenta un mejor rendimiento que el particionamiento por diagonales, acercándose más a la aceleración ideal.

De acuerdo a los resultados obtenidos, al igual que en el algoritmo $O(n^3)$ también existen números óptimos de bloques, y en este algoritmo de memoria compartida se encuentran en el rango de $n/12$ a $n/5$.

3.3.2 Memoria distribuida

El esquema de comunicación implementado para este algoritmo es SPMD o comunicación local, dado que éste presentó los mejores resultados que la comunicación maestro-esclavo en el

Tabla 3.3: Tiempos de ejecución, variando la longitud de la secuencia para memoria compartida (tiempo en minutos)

n	Secuencial	Diagonales	Bloques homogéneos
403	0.553466	0.179790	0.158184
503	1.350443	0.423724	0.379318
603	2.763212	0.845938	0.761234
703	5.186103	1.572753	1.420813
803	9.038905	2.715446	2.437627
903	14.641880	4.380584	3.933761
1003	22.712547	6.731407	6.135244
1503	120.210612	35.595785	32.521459
2003	385.797135	112.952246	103.575838

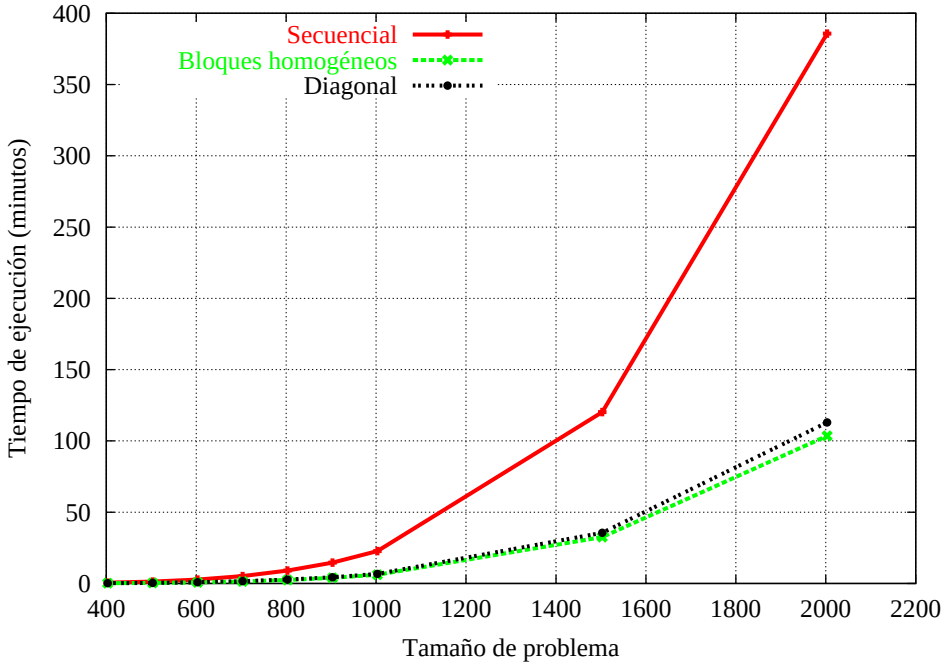


Figura 3.23: Tiempos de ejecución, variando la longitud de la secuencia para el algoritmo $O(n^4)$ (tiempo en minutos).

algoritmo $O(n^3)$. A continuación se describen los resultados experimentales que se obtuvieron para $n = 1500$ bases.

En la Tabla 3.4 se muestran los tiempos de ejecución del algoritmo secuencial, del particionamiento por diagonales y por bloques homogéneos para secuencias grandes; en la Figura 3.27 se muestran estos resultados. El tiempo del algoritmo secuencial aumenta considerablemente conforme se va incrementando la longitud de la secuencia n. Los algoritmos paralelo utilizan 4 procesadores físicos y se toma a $n/10$ como el número de bloques. Se puede ver que el tiempo

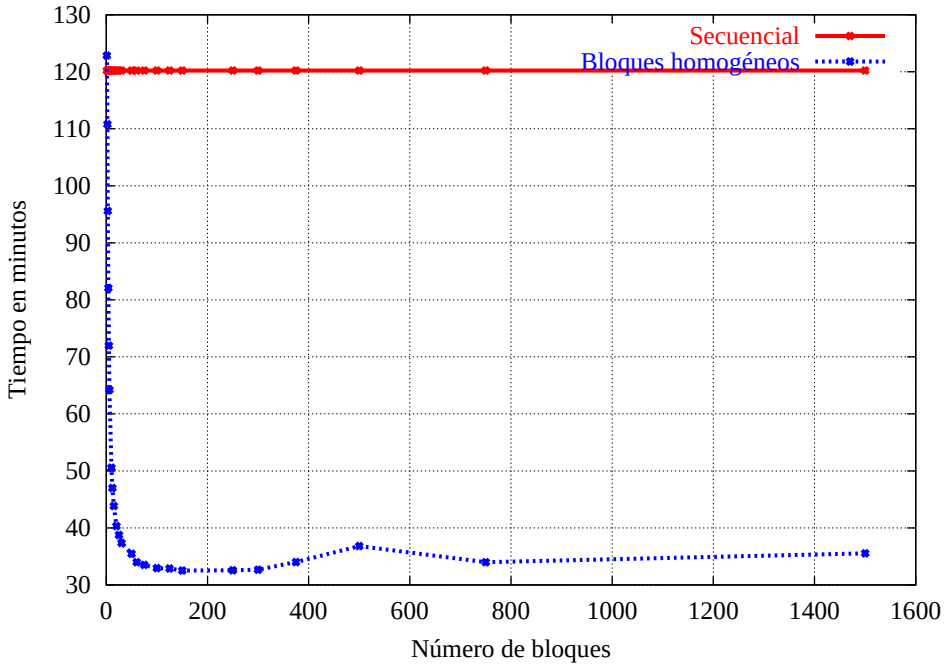


Figura 3.24: Resultados experimentales de bloques homogéneos.

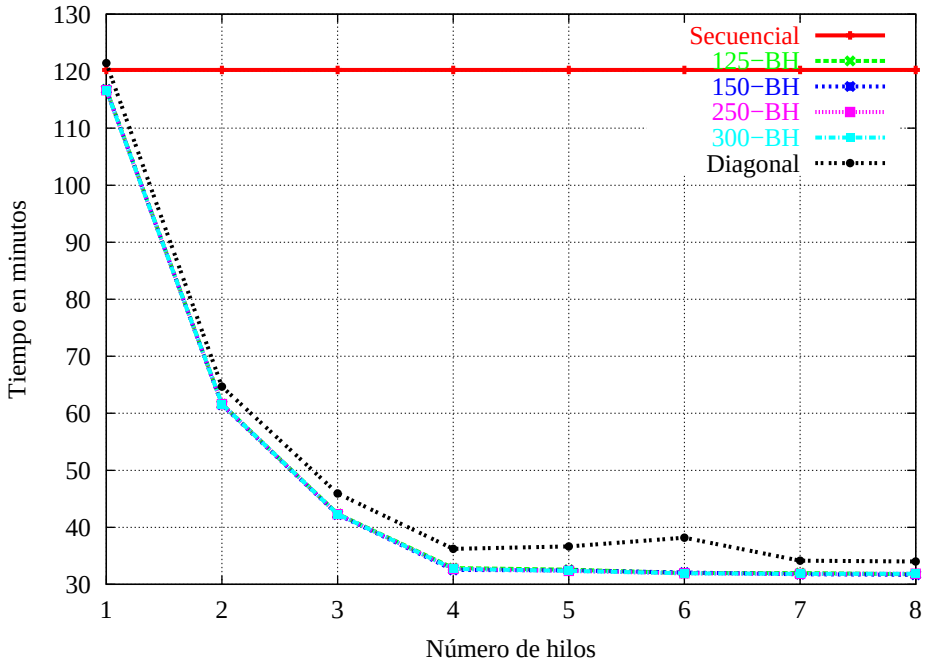


Figura 3.25: Mejores divisores de bloques homogéneos variando el número de hilos.

del algoritmo paralelo disminuye aproximadamente la tercera parte del algoritmo secuencial.

La Figura 3.28 visualiza el tiempo de ejecución que se obtuvo para el particionamiento

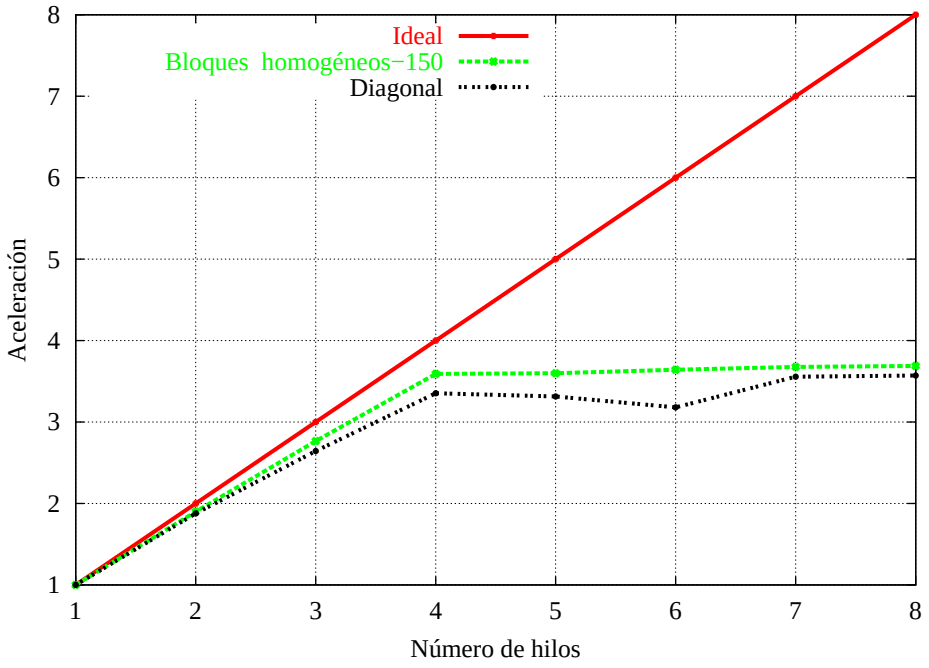


Figura 3.26: Aceleración del particionamiento por bloques homogéneos.

por bloques homogéneos, variando el número de bloques. Estos resultados se obtuvieron con 4 procesadores, $p = 4$, el tiempo secuencial es de 156.091602 minutos.En esta estrategia, también existe un número óptimo de bloques y éste se encuentra en el rango de $n/12$ a $n/2$.

La Figura 3.29 muestra los número de bloques que obtuvieron los mejores resultados del tiempo de ejecución, se comparan con el particionamiento por diagonales y el secuencial, variando el número de procesadores de 1 a 8. Es evidente que la curva del particionamiento por bloques homogéneos está por abajo de la del particionamiento por diagonales.

La Figura 3.30 muestra el rendimiento de los procesadores al aplicar la comunicación SPMD al algoritmo $O(n^4)$. Obsérvese que la estrategia de bloques homogéneos, al igual que en el algoritmo $O(n^3)$, obtiene un buen rendimiento al compararse con la estrategia de diagonales y con el programa secuencial.

Tabla 3.4: Tiempos de ejecución, variando la longitud de la secuencia para memoria distribuida (tiempo en minutos)

n	Secuencial	Diagonales	Bloques homogéneos
403	0.737395	0.234447	0.216951
503	1.795748	0.553319	0.533185
603	3.754962	1.172537	1.062780
703	7.134179	2.207145	2.001547
803	12.181861	3.721623	3.502772
903	19.66852	5.875476	5.593814
1003	30.050391	8.832801	8.556236
1503	156.091602	44.566984	43.669408
2003	488.253939	137.339697	135.714185

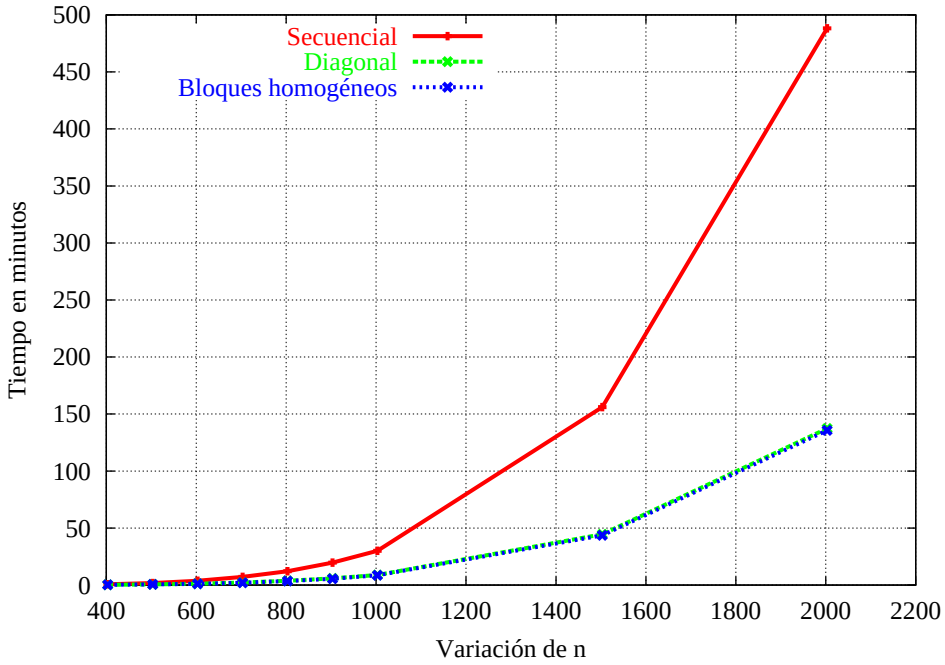


Figura 3.27: Tiempos de ejecución, variando la longitud de la secuencia para memoria distribuida (tiempo en minutos).

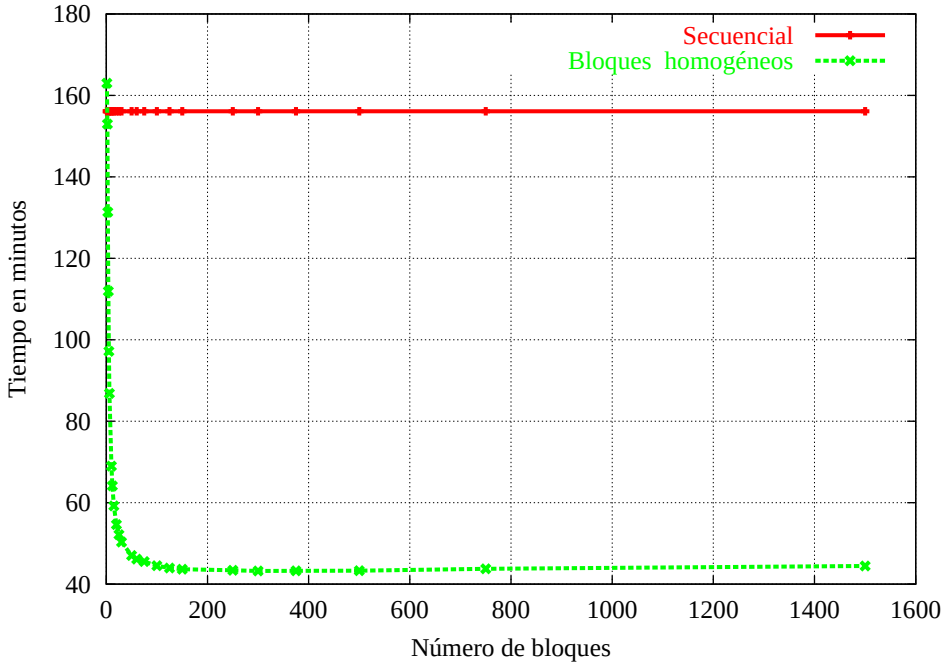


Figura 3.28: Tiempos de ejecución, variando el número de bloques para bloques homogéneos, memoria distribuida (tiempo en minutos)

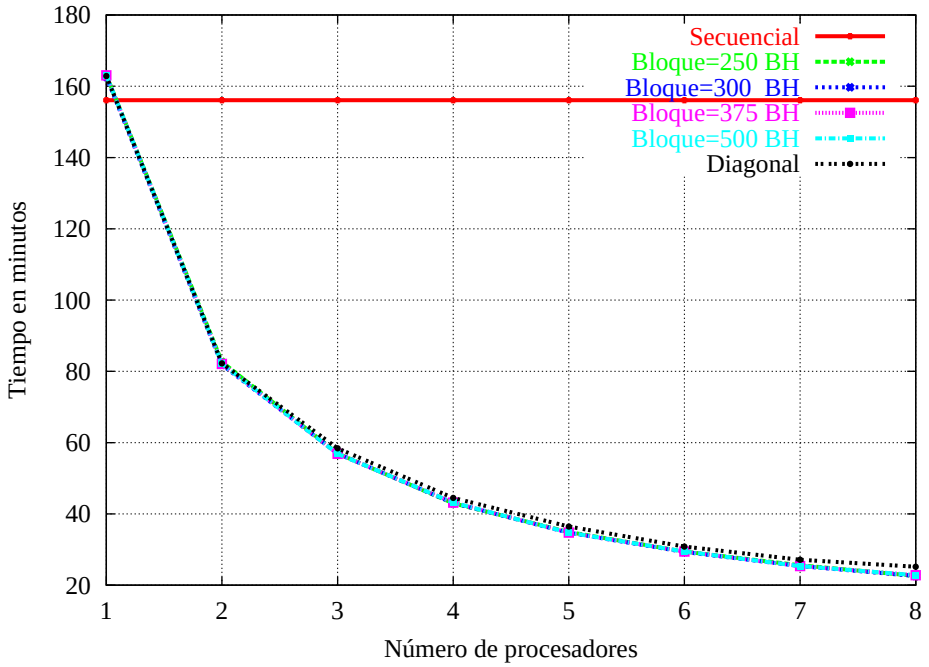


Figura 3.29: Tiempos de ejecución de los mejores número de bloques, variando el número de procesadores para memoria distribuida (tiempo en minutos).

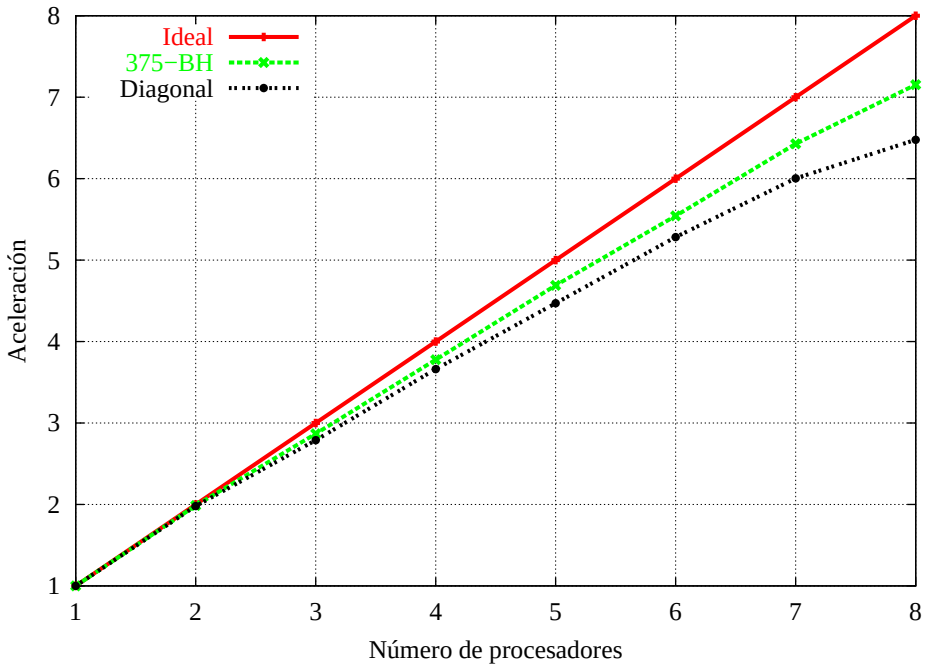


Figura 3.30: Aceleración para memoria distribuida.

Conclusiones

Se partió de dos algoritmos básicos para encontrar la solución al problema de RNA. Ambos algoritmos secuenciales, con complejidades en tiempo $O(n^3)$ y $O(n^4)$, permiten encontrar estructuras secundarias de orden 1 y 2 [3]. Se comprobó experimentalmente que los algoritmos secuenciales funcionan adecuadamente para secuencias de longitud corta. Sin embargo, al aumentar la longitud de las secuencias, los tiempos de ejecución se elevan de tal manera que es poco práctico aplicarlos a secuencias largas. Se presentan en este trabajo tres estrategias de particionamiento para el procesamiento paralelo para resolver el problema de RNA: particionamiento por diagonales, por bloques homogéneos y por bloques no homogéneos. La base para efectuar las estrategias de particionamiento se debe a que los algoritmos utilizan la estrategia de programación dinámica que llena, en ambos casos, una matriz triangular superior.

Mediante un análisis de dependencia de datos se observó que para explotar el paralelismo el proceso de llenado de la matriz triangular puede proceder por diagonales desde la principal hasta la esquina superior derecha, lugar en donde se encuentra la solución al problema. El particionamiento por diagonales trata de explotar este hecho y distribuye equitativamente el número de entradas en una misma diagonal entre el número disponible de procesadores. En este particionamiento no se puede proceder a procesar la diagonal siguiente hasta que no se ha concluido con el procesamiento de la diagonal actual. Por lo tanto, el número de operaciones de sincronización es igual al número de diagonales, aumentando así el tiempo de sincronización y comunicación. Para secuencias pequeñas estas operaciones no tienen un efecto significativo sobre el tiempo de ejecución pero para secuencias grandes estas operaciones aumentan considerablemente.

Un análisis más profundo demostró que sólo los elementos en el triángulo inferior izquierdo relativos a una entrada $F(i, j)$ en la tabla son necesarios para obtener el valor de dicha entrada. Por lo tanto, para reducir las operaciones de sincronización se proponen los particionamientos basados en bloques.

El particionamiento por bloques homogéneos, divide a la tabla en un determinado número de bloques de diagonales. Cada bloque de diagonales es dividido entre los procesadores disponibles, los cuales aplican localmente un algoritmo secuencial. La operación de sincronización se da solamente cuando un procesador ha concluido con su bloque de diagonales asignado. Por tanto, si la tabla se divide en B bloques, se requieren $B - 1$ operaciones de sincronización y no $n - 1$ como en el particionamiento por diagonales. De acuerdo a las características de este particionamiento sólo los divisores exactos de $n - 3$ proporcionan mejores resultados. La longitud $n - 3$ se debe a que a las primeras 3 diagonales se omiten por la restricción de $j - i \leq 3$ en las

funciones de recurrencia.

Para evitar conflictos de dependencias de datos, inicialmente se forman triángulos equiláteros en la tabla de programación dinámica. Subsecuentemente, los bloques se forman con cuadrados o rombos. La evaluación para cada cuadrado o triángulo se procedió en forma de diagonal.

Un estudio sobre la cantidad de operaciones involucradas para el cálculo de cada una de las diagonales de la tabla, refleja que las diagonales centrales requieren un mayor número de evaluaciones anteriores, por lo que los algoritmos se esfuerzan más en los cálculos de los datos centrales de la tabla que en los datos que se encuentran a las orillas, es decir en las primeras diagonales y en las últimas. El particionamiento por bloques no homogéneos busca que el tiempo de cómputo se distribuya equitativamente entre el número de bloques. Esto provoca que los bloques no siempre sean de la misma longitud, es decir los bloques son variables (no homogéneos), determinándose en tiempo de ejecución la longitud del siguiente bloque a procesar. A pesar de que este particionamiento es una generalización del particionamiento homogéneo la orquestación del programa requiere de mayor número de operaciones de sincronización. Para eliminar conflictos de dependencias de datos se procedió a formar triángulos equiláteros superiores e inferiores, pero como los bloques no siempre son de la misma longitud, los triángulos formados son de diferente tamaño. Por cada bloque, el cálculo de todos los triángulos superiores se debe realizar antes que la evaluación de los triángulos inferiores, y para continuar con la evaluación del siguiente bloque, los cálculos de todos los triángulos inferiores del bloque anterior se deben realizar completamente. Por tanto, es necesario usar dos procesos de sincronización en cada bloque. A pesar de que el número de operaciones de sincronización de este algoritmo es menor al número de diagonales (n) es el doble que el algoritmo por bloques homogéneos, es decir es $2B$.

Las tres técnicas de particionamiento paralelo se aplicaron sobre computadoras de memoria compartida y computadoras de memoria distribuida. Experimentalmente se comprobó que el particionamiento por diagonales con cuatro procesadores reduce el tiempo del algoritmo secuencial en una tercera parte. Sin embargo, al aplicar el particionamiento por bloques se reduce aun más el tiempo del algoritmo secuencial y del particionamiento por diagonales. Se observó también que existe un número óptimo de bloques y de acuerdo al algoritmo y a la arquitectura se encuentra en un rango determinado entre $n/25$ y $n/5$ como número de bloques.

Los algoritmos diseñados para memoria compartida mostraron un buen desempeño y, de las tres estrategias de particionamiento, el algoritmo por bloques homogéneos obtuvo un mejor rendimiento que los otros dos. Para los algoritmos de memoria distribuida, se implementaron dos esquemas de comunicación: maestro-esclavo y SPMD. La comunicación SPMD (local) obtuvo mejores resultados que la comunicación maestro-esclavo (global). El particionamiento por bloques homogéneos, obtuvo también un buen desempeño. De acuerdo a los resultados experimentales, se observó que el número óptimo de bloques para memoria compartida está en el rango de $[n/25, n/15]$ para 3000 bases y para 1500 esta en $[n/12, n/5]$. Para memoria distribuida los óptimos están en el rango de $[n/20, n/8]$ para 3000 bases, y para 1500 en $[n/12, n/2]$ donde n es la longitud de la secuencia.

Es posible desarrollar un modelo de rendimiento que indique, de acuerdo a los diferentes parámetros del algoritmo y de la arquitectura usada, el número óptimo de bloques a usar en los algoritmos. Este estudio no se ha incluido en este trabajo y se deja como un trabajo adicional.

Los algoritmos implantados permiten obtener estructuras secundarias de orden 1 y 2. El algoritmo de complejidad en tiempo $O(n^3)$, sólo permite analizar tales estructuras. El algoritmo de complejidad en tiempo $O(n^4)$ puede ser modificado para obtener estructuras secundarias de orden mayor. Sin embargo, si k es el orden del ciclo a analizar, la complejidad del algoritmo crece a $O(n^{2k})$. La complejidad crece debido a la cantidad de dependencias que en un momento deben analizarse para determinar una entrada de la tabla. La estrategia de solución del problema del RNA es la misma y la forma de la tabla de programación dinámica también se mantiene (triangular superior). Por lo tanto, las estrategias de particionamiento propuestas aquí también se aplicarían al análisis de estructuras secundarias de orden mayor. Es de esperarse, que con estructuras de orden mayor el particionamiento no homogéneo presente mejores resultados que el particionamiento homogéneo. Sin embargo, es necesario realizar trabajo adicional para comprobar tal hipótesis.

En la literatura se pueden encontrar estructuras secundarias de casos de estudios obtenidas por otros medios. Es necesario realizar un trabajo de validación de los algoritmos presentados con los resultados obtenidos por otros autores.

Apéndice A

Tablas de resultados experimentales

A.1 Algoritmo $O(n^3)$

En esta sección se muestran las tablas de resultados obtenidas para memoria compartida y memoria distribuida para el algoritmo $O(n^3)$, las figuras se muestran en el Capítulo 3.

A.1.1 Memoria compartida

Los resultados que se muestran en las siguientes tablas fueron obtenidos en la computadora de multiprocesamiento elvis con 4 procesadores físicos. La longitud de la secuencia empleada para realizar las pruebas es de $n = 3000$ bases. Todas las secuencias son generadas de manera aleatoria. Cada resultado de las tablas es el promedio de 5 pruebas. El tiempo del algoritmo secuencial para 3000 bases es de 7.105554 minutos, tiempo obtenido en la computadora elvis.

La Tabla A.1 muestra los resultados obtenidos por cada uno de los algoritmos implementados variando el número de bloques (divisores exactos de n), para un problema de 3000 bases, generadas de manera aleatoria.

Las Tablas A.2 y A.3 muestran los tiempos de ejecución variando el número de bloques (divisores) que proporcionaron los mejores resultados en bloques homogéneos y no homogéneos para secuencias de 3000 bases, variando el número de hilos para cada número de bloque.

La Tabla A.4 ilustran el rendimiento de los procesadores (aceleración) aplicados a los particionamientos de bloques homogéneos y no homogéneos, comparándolo con la aceleración ideal y la aceleración obtenida por el particionamiento por diagonales.

Tabla A.1: Resultados del tiempo de ejecución: bloques homogéneos y no homogéneos (tiempo en minutos).

No. bloques	BH	BNH	No. bloques	BH	BNH
1	7.059344	7.077331	60	1.293149	1.430572
2	6.247143	6.254931	75	1.235601	1.371505
3	5.101562	4.822503	100	1.193750	1.334412
4	4.258639	3.962723	120	1.181370	1.314279
5	3.692743	3.701277	125	1.182451	1.317478
6	3.380967	3.536408	150	1.177310	1.311164
8	2.996837	3.055022	200	1.180447	1.322946
10	2.723076	2.905282	250	1.190560	1.348633
12	2.534155	2.606948	300	1.200981	1.369735
15	2.313390	2.380466	375	1.220251	1.417760
20	2.046036	2.094809	500	1.255300	1.510763
24	1.881070	1.920102	600	1.282194	1.583894
25	1.849231	1.891595	750	1.331755	1.721684
30	1.695531	1.755303	1000	1.405846	1.897341
40	1.486582	1.569752	1500	1.583718	2.057103
50	1.364801	1.477960	3000	2.150614	2.160078

Tabla A.2: Mejores resultados variando el número de hilos, para bloques homogéneos (tiempo en minutos).

No.de hilos	Bloques homogéneos				Diagonal
	120	125	150	200	
1	4.615985	4.584096	4.560721	4.577725	7.092547
2	2.345158	2.365517	2.328835	2.317558	3.822963
3	1.592197	1.601493	1.571787	1.562185	2.674712
4	1.221977	1.231425	1.203704	1.187331	2.149499
5	1.291300	1.287419	1.267203	1.304604	2.857826
6	1.257608	1.272327	1.269063	1.262645	2.608440
7	1.261891	1.244428	1.243269	1.249769	2.408803
8	1.225700	1.236574	1.223403	1.217288	2.249738

Tabla A.3: Mejores resultados variando el número de hilos, para bloques no homogéneos (tiempo en minutos).

No.de hilos	Bloques no homogéneos				Diagonal
	120	125	150	200	
1	4.771454	4.768904	4.792107	4.876716	7.092547
2	2.503704	2.510131	2.500515	2.523047	3.822963
3	1.716267	1.721680	1.709875	1.723326	2.674712
4	1.342249	1.344404	1.325318	1.330281	2.149499
5	1.436416	1.440215	1.514646	1.625705	2.857826
6	1.385790	1.401658	1.408664	1.501815	2.608440
7	1.384974	1.376288	1.395834	1.460858	2.408803
8	1.347726	1.357337	1.358324	1.396630	2.249738

Tabla A.4: Aceleración

Número de hilos	Ideal	Diagonal	Bloques homogéneos (200 bloques)	Bloques no homogéneos (150 bloques)
1	1.000000	1.000000	1.000000	1.000000
2	2.000000	1.855249	1.975236	1.916448
3	3.000000	2.651705	2.930335	2.802607
4	4.000000	3.299628	3.855475	3.615817
5	5.000000	2.481798	3.508900	3.163846
6	6.000000	2.719076	3.625504	3.401881
7	7.000000	2.944428	3.662857	3.433150
8	8.000000	3.152610	3.760593	3.527956

A.1.2 Memoria distribuida

Esquema Maestro-Esclavo o Comunicación Global

Cada resultado de las tablas son un promedio de 5 pruebas realizadas para secuencias de $n = 3000$ bases generadas de manera aleatoria. El tiempo de ejecución está medido en minutos. El resultado del algoritmo secuencial es de 10.050317 minutos y se obtuvo en la computadora presley.cs.cinvestav.mx.

La Tabla A.5 muestra los resultados obtenidos por cada uno de los algoritmos implementados variando el número de bloques (divisores exactos), usando cuatro procesadores.

Las Tablas A.6 y A.7 muestran los divisores que proporcionaron los mejores resultados en bloques homogéneos (BH) y no homogéneos (BNH), variando el número de procesadores para cada número de bloque.

El rendimiento de los procesadores (aceleración) se muestra en la Tabla A.8 para bloques homogéneos y no homogéneos comparándolos con la aceleración ideal y el particionamiento por diagonales.

Tabla A.5: Resultados del tiempo de ejecución: bloques homogéneos (BH) y bloques no homogéneos (BNH), tiempo en minutos.

No. Bloques	BH	BNH	No. Bloques	BH	BNH
1	7.183859	7.273213	60	3.134868	3.055263
2	6.661540	6.662718	75	3.035656	2.986194
3	5.843049	5.401457	100	2.913673	2.889487
4	5.125811	4.952498	120	2.829053	2.840521
5	4.702116	4.782087	125	2.822476	2.839535
6	4.488957	4.657375	150	2.758943	2.802114
8	4.115027	4.298923	200	2.684543	2.796732
10	3.939182	4.187403	250	2.661267	2.815753
12	3.800337	3.998475	300	2.651768	2.811020
15	3.663178	3.791706	375	2.667686	2.853556
20	3.523953	3.644500	500	2.695862	2.907421
24	3.450956	3.526657	600	2.750530	2.950097
25	3.440335	3.527797	750	2.792066	3.042092
30	3.377687	3.416889	1000	2.869717	3.223790
40	3.275617	3.215635	1500	3.092757	3.322640
50	3.202349	3.122826	3000	3.552217	3.489616

Tabla A.6: Tiempo de ejecución de los mejores divisores, variando el número de procesadores para bloques homogéneos (tiempo en minutos).

No.de Proc.	Bloques homogéneos				Diagonal
	200	250	300	375	
2	4.677504	4.720122	4.774313	4.834519	7.262661
3	3.923106	3.953076	3.921727	3.944304	5.013409
4	2.683370	2.680408	2.666105	2.683122	3.374686
5	2.113975	2.081739	2.078757	2.096237	2.751660
6	1.753192	1.726274	1.722963	1.728395	2.279724
7	1.545460	1.536601	1.511068	1.533696	2.010146
8	1.380210	1.357604	1.419872	1.362827	1.776689

Tabla A.7: Tiempo de ejecución de los mejores divisores, variando el número de procesadores para bloques no homogéneos (tiempo en minutos).

No.de Proc.	Bloques no homogéneos				Diagonal
	150	200	250	300	
2	4.850437	4.939086	5.033176	5.135285	7.262661
3	3.974257	4.009993	4.016685	4.082070	5.013409
4	2.791349	2.775813	2.791227	2.798908	3.374686
5	2.200241	2.185598	2.188782	2.193032	2.751660
6	1.837610	1.810310	1.815147	1.811360	2.279724
7	1.607695	1.600834	1.575239	1.573360	2.010146
8	1.455498	1.421749	1.411827	1.417117	1.776689

Tabla A.8: Aceleración

Número de procesadores	Ideal	Diagonal	Bloques homogéneos (300 bloques)	Bloques no homogéneos (200 bloques)
2	2.000000	1.000000	1.521195	1.470446
3	3.000000	1.448647	1.851904	1.811141
4	4.000000	2.152100	2.724072	2.616409
5	5.000000	2.639374	3.493752	3.322963
6	6.000000	3.185763	4.215216	4.011833
7	7.000000	3.613002	4.806310	4.536798
8	8.000000	4.087750	5.115011	5.108258

Esquema SPMD o Comunicación Local

La longitud de la secuencia que se utilizó para realizar las pruebas es de 3000 bases, generadas de manera aleatoria. Los resultados mostrados en las tablas son el promedio de cinco pruebas. El tiempo del algoritmo secuencial es de 10.050317 minutos obtenido en la computadora presley.

La Tabla A.9 muestra los resultados obtenidos por cada uno de los algoritmos implementados variando el número de bloques (divisores exactos) utilizando cuatro procesadores.

Las Tablas A.10 y A.11 muestran los divisores que proporcionaron los mejores resultados en bloques homogéneos y no homogéneos, variando el número de procesadores para cada número de bloque. El rendimiento de los procesadores (aceleración) para este esquema de comunicación se muestra en la Tabla A.12.

Tabla A.9: Variación del número de bloques para bloques homogéneos (BH) y no homogéneos (BNH), tiempo en minutos.

No. Bloques	BH	BNH	No. Bloques	BH	BNH
1	9.929513	10.150748	60	2.677380	2.880416
2	8.741570	8.925940	75	2.591629	2.666636
3	7.047738	6.797052	100	2.480309	2.558646
4	5.897104	5.643568	120	2.384759	2.519199
5	5.112023	5.372695	125	2.368798	2.516443
6	4.720392	5.108374	150	2.328010	2.437010
8	4.214595	4.496277	200	2.253633	2.393031
10	3.898219	4.400544	250	2.183368	2.373268
12	3.739770	4.082017	300	2.135040	2.338232
15	3.477694	3.867659	375	2.133049	2.352648
20	3.280336	3.651747	500	2.124506	2.379912
24	3.148827	3.602420	600	2.138554	2.442610
25	3.175546	3.567379	750	2.174174	2.505984
30	3.039357	3.380370	1000	2.208467	2.588498
40	2.890754	3.094011	1500	2.391065	2.753931
50	2.769201	3.287895	3000	2.814953	3.472167

Tabla A.10: Tiempo de ejecución de los mejores divisores, variando el número de procesadores para bloques homogéneos (tiempo en minutos).

No.de Proc.	Bloques homogéneos				Diagonal
	300	375	500	600	
1	7.801451	7.757347	7.853059	7.879271	9.756137
2	4.099332	4.007832	4.052012	4.080123	5.033149
3	2.957836	2.755007	2.772161	2.772061	3.643735
4	2.154752	2.147001	2.153519	2.144760	2.818163
5	1.755795	1.867590	1.730105	1.747538	2.377144
6	1.514455	1.762229	1.484913	1.490274	2.028676
7	1.381994	1.715594	1.321999	1.327766	1.863165
8	1.201690	1.472179	1.203827	1.198559	1.699653

Tabla A.11: Tiempo de ejecución de los mejores divisores para bloques no homogéneos, variando el número de procesadores (tiempo en minutos).

No.de Proc.	Bloques no homogéneos				Diagonal
	250	300	375	500	
1	8.836948	8.489602	8.479326	8.686870	9.756137
2	4.683836	4.424248	4.394370	4.441513	5.033149
3	3.067386	3.080516	3.043154	3.078491	3.643735
4	2.485553	2.384622	2.337744	2.368818	2.818163
5	2.101833	1.959745	1.938228	1.973311	2.377144
6	1.662255	1.721749	1.668666	1.712449	2.028676
7	1.483467	1.509410	1.489242	1.508357	1.863165
8	1.355291	1.378126	1.347874	1.360423	1.699653

Tabla A.12: Aceleración de la comunicación SPMD.

Número de procesadores	Ideal	Diagonal	Bloques homogéneos (500 bloques)	Bloques no homogéneos (375 bloques)
1	1.000000	1.000000	1.000000	1.000000
2	2.000000	1.938376	1.938064	1.929589
3	3.000000	2.677510	2.832829	2.786361
4	4.000000	3.461878	3.646617	3.627141
5	5.000000	4.104142	4.539065	4.374783
6	6.000000	4.809115	5.288565	5.081500
7	7.000000	5.236325	5.940291	5.693719
8	8.000000	5.740076	6.523412	6.290889

Maestro-Esclavo vs SPMD

En la Tabla A.13 se comparan los resultados obtenidos por cada esquema usando el particionamiento por bloques homogéneos porque fue el algoritmo que tuvo un buen comportamiento, para 3000 bases.

En la Tabla A.14 se comparan los resultados obtenidos por cada esquema al usar el particionamiento por diagonales y bloques homogéneos para 3000 bases, variando el número de procesadores. La aceleración que se obtuvo para estos resultados se muestra en la Tabla A.15.

Tabla A.13: Comparación entre Maestro-Esclavo y SPDM, variando números de bloques

Número de bloques	Bloques homogéneos	
	MaeEsc	SPMD
1	7.183859	9.929513
2	6.661540	8.741570
3	5.843049	7.047738
4	5.125811	5.897104
5	4.702116	5.112023
6	4.488957	4.720392
8	4.115027	4.214595
10	3.939182	3.898219
12	3.800337	3.739770
15	3.663178	3.477694
20	3.523953	3.280336
24	3.450956	3.148827
25	3.440335	3.175546
30	3.377687	3.039357
40	3.275617	2.890754
50	3.202349	2.769201
60	3.134868	2.677380
75	3.035656	2.591629
100	2.913673	2.480309
120	2.829053	2.384759
125	2.822476	2.368798
150	2.758943	2.328010
200	2.684543	2.253633
250	2.661267	2.183368
300	2.651768	2.135040
375	2.667686	2.133049
500	2.695862	2.124506
600	2.750530	2.138554
750	2.792066	2.174174
1000	2.869717	2.208467
1500	3.092757	2.391065
3000	3.552217	2.814953

Tabla A.14: Comparación entre Maestro-Esclavo y SPDM, variando el números de procesadores

Número de procesadores	Bloques homogéneos		Diagonal	
	MaeEsc 300 bloques	SPMD 300 bloques	MaeEsc	SPMD
1		7.801451		9.756137
2	4.774313	4.099332	7.262661	5.033149
3	3.921727	2.957836	5.013409	3.643735
4	2.666105	2.154752	3.374686	2.818163
5	2.078757	1.755795	2.751660	2.377144
6	1.722963	1.514455	2.279724	2.028676
7	1.511068	1.381994	2.010146	1.863165
8	1.419872	1.201690	1.776689	1.699653

Tabla A.15: Aceleración de Maestro-Esclavo y SPDM

Número de procesadores	Ideal	Bloques homogéneos		Diagonal	
		MaeEsc 300 bloques	SPMD 300 bloques	MaeEsc	SPMD
1	1.000000		1.000000		1.000000
2	2.000000	1.521195	1.903103	1.000000	1.938376
3	3.000000	1.851904	2.637554	1.448647	2.677510
4	4.000000	2.724072	3.620580	2.152100	3.461878
5	5.000000	3.493752	4.443258	2.639374	4.104142
6	6.000000	4.215216	5.151326	3.185763	4.809115
7	7.000000	4.806310	5.645069	3.613002	5.236325
8	8.000000	5.115011	6.492066	4.087750	5.740076

A.2 Algoritmo $O(n^4)$

A.2.1 Memoria compartida

La Tabla A.16 muestra los tiempos de ejecución del particionamiento por bloques homogéneos al variar el número de bloques para $n = 1500$ con cuatro procesadores ($h = 4$). El tiempo de ejecución del programa secuencial es de 120.210612 minutos.

La Tabla A.17 muestra el tiempo de ejecución de los mejores números de bloques, obtenidos a partir de los resultados de la Tabla A.16, variando el número de procesadores comparándolos con los resultados del particionamiento por diagonales.

La Tabla A.18 muestra el rendimiento de los procesadores, mostrando la aceleración ideal y la aceleración del particionamiento por diagonales y por bloques homogéneos.

Tabla A.16: Resultados experimentales para bloques homogéneos (tiempo en minutos)

Numero de bloques	Bloques homogéneos	Numero de bloques	Bloques homogéneos
1	122.845265	50	35.457712
2	110.787653	60	33.973564
3	95.579898	75	33.504279
4	82.078739	100	32.936657
5	71.959183	125	32.878145
6	64.235048	150	32.531433
10	50.546797	250	32.56279
12	47.000872	300	32.670302
15	43.844063	375	34.00119
20	40.320496	500	36.811836
25	38.750362	750	33.983889
30	37.356021	1500	35.530992

Tabla A.17: Mejores divisores de bloques homogéneos (tiempo en minutos)

No.de Hilos	125	150	250	300	Diagonal
1	116.811413	116.785207	116.651172	116.616439	121.410898
2	61.627305	61.489233	61.623761	62.061661	64.673011
3	42.344790	42.237058	42.308856	42.447993	45.938938
4	32.843986	32.521459	32.704768	32.679176	36.211864
5	32.563237	32.453058	32.346387	32.342172	36.649705
6	31.983954	32.063229	31.924243	32.032739	38.189038
7	31.975277	31.766847	31.850483	31.817000	34.140033
8	31.764502	31.668467	31.857492	31.764463	33.995960

Tabla A.18: Aceleración del particionamiento por bloques homogéneos

No.de Hilos	Ideal	Diagonal	Bloques homogéneos 150 bloques
1	1.000000	1.000000	1.000000
2	2.000000	1.899279	1.877304
3	3.000000	2.764994	2.642876
4	4.000000	3.591020	3.352793
5	5.000000	3.598589	3.312739
6	6.000000	3.642341	3.179208
7	7.000000	3.676324	3.556262
8	8.000000	3.687744	3.571333

A.2.2 Memoria distribuida

La Tabla A.19 muestra el tiempo de ejecución que se obtuvo para el particionamiento por bloques homogéneos, variando el número de bloques. Estos resultados se obtuvieron con 4 procesadores, $p = 4$, el tiempo secuencial es de 156.091602 minutos para $n = 1500$ bases.

La Tabla A.20 muestra los número de bloques que obtuvieron los mejores resultados de tiempo de ejecución, se comparan con el particionamiento por diagonales, variando el número de procesadores de 1 a 8.

La Tabla A.21 muestra el rendimiento de los procesadores al aplicar la comunicación SPMD al algoritmo $O(n^4)$.

Tabla A.19: Tiempos de ejecución, variando el número de bloques para memoria distribuida (tiempo en minutos)

No. de bloques	Bloques homogéneos	No. de bloques	Bloques homogéneos
1	162.994450	50	47.036895
2	152.958721	60	46.135294
3	131.358221	75	45.561010
4	111.907253	100	44.525585
5	97.173644	125	43.976337
6	86.833008	150	43.669408
10	68.985576	250	43.390198
12	64.127966	300	43.222037
15	59.200496	375	43.276240
20	54.612013	500	43.307512
25	52.150075	750	43.749054
30	50.321271	1500	44.469412

Tabla A.20: Tiempos de ejecución de los mejores número de bloques, variando el número de procesadores para memoria distribuida (tiempo en minutos)

No. de procesadores	250	300	375	500	Diagonal
1	163.095397	162.601136	163.048796	162.279333	162.878994
2	82.654816	81.845115	82.097593	85.303584	82.229543
3	56.963119	56.944733	56.855949	57.148087	58.396490
4	42.981286	43.032415	43.168975	43.136058	44.477074
5	34.917432	34.711615	34.764648	35.201575	36.439423
6	29.447651	29.300297	29.418745	29.513853	30.834532
7	25.581940	25.366988	25.368909	25.446902	27.133491
8	22.590536	22.561785	22.794900	22.803285	25.150719

Tabla A.21: Aceleración (tiempo en minutos)

No.de Procesadores	Ideal	Diagonal	Bloques homogéneos 375 bloques
1	1.000000	1.000000	1.000000
2	2.000000	1.980784	1.986036
3	3.000000	2.789192	2.867753
4	4.000000	3.662089	3.776990
5	5.000000	4.469857	4.690075
6	6.000000	5.282357	5.542344
7	7.000000	6.002876	6.427111
8	8.000000	6.476117	7.152863

Bibliografía

[1] A. Díaz-Pérez, M. J. Quinn, and G. Morales-Luna. “Designing data-parallel programs for dynamic programming algorithms”. *In Conference on Parallel Processing and Applied Mathematics*, pp. 257–266, September 1997.

[2] A. Díaz-Pérez, M. J. Quinn, and G. Morales-Luna. “The Uniformization of Recurrence Equations”, *Segunda Conferencia de Ingeniería Eléctrica: CIE-96.* México, D. F., 11-13 septiembre de 1996, pp. 80-89.

[3] Sankoff, David and Kruskal, Joseph B., Time warps, string edits, and macromolecules: *The theory and practice of sequence comparison*, Addison-Wesley, Reading, MA, 1983.

[4] Tovar, Mireya. Reporte del proyecto del curso “Programación Paralela”. Agosto, 2001. CINVESTAV-IPN.

[5] A. Díaz-Pérez. Notas del curso “Programación Paralela”. Agosto, 2001. CINVESTAV-IPN.

[6] Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*, McGraw-Hill, 1990.

[7] Michael J. Quinn. *Parallel Computing, Theory and Practice*, McGraw-Hill, 1994.

[8] Foster, Ian T., *Designing and Building Parallel Programs*, Addisson-Wesley.

[9] A. Díaz-Pérez. “A classification of Dynamic Programming Problems”. Febrero, 1996. INAOE.

[10] M. Zuker, D. Sankoff. “RNA secondary structures and their predition”. *Bull. Math Biol.*, **46**:591-621, 1984.

[11] M. Zuker, J.A. Jaeger, D.H. Turner. “A comparison of optimal and suboptimal RNA secondary structures predicted by free energy minimization with structures determined by philogenetic comparison”. *Nucleic Acids Res.*, **19**:2707-2714, 1991.

[12] Chen, Jih-H., Le, S-Y., Shapiro, B.A., Maizel, J. V.: “Optimization of an RNA folding algorithm for parallel architectures”. *Parallel Computing* **24**:1617-1634, 1998.

[13] Bruce A. Shapiro, Jin Chu Wu, David Bengali. "The massively parallel genetic algorithm for RNA folding: MIMD implementation and population variation", *Bioinformatics* **17**:137-148, 2001.

[14] Goldberg, D.E., Kuo C. H. "Genetic algorithms in pipeline optimization". *J. Comput. Civil Eng.*, **Vol.** 1, 2:128-141, 1987.

[15] Ivo L. Hofacker, Walter Fontana, Peter F. Stadler, L. Sebastian Bonhoeffer, Manfred Tacker, and Peter Schuster. "Fast Folding and Comparison of RNA Secondary Structures", *Monatsh.Chem.* **125**: 167-188 (1994).

[16] Ole Matzura and Anders Wennborg, "RNAdraw: an integrated program for RNA secondary structure calculation and analysis under 32-bit Microsoft Windows", *Computer Applications in the Biosciences (CABIOS)*, **Vol.** 12 no. 3:247-249, 1996.

[17] M. Zuker, D.H. Mathews y D.H. Turner "Algorithms and Thermodynamics for RNA Secondary Structure Prediction: A Practical Guide In RNA Biochemistry and Biotechnology", J. Barciszewski y B.F.C. Clark, eds., NATO ASI Series, Kluwer Academic Publishers, 1999.

[18] Rune B. Lyngso, Michael Zuker and Christian N. S. Pedersen. "Fast evaluation of internal loops in RNA secondary structure prediction" *Bioinformatics*, **Vol.** 15 no. 6 440-445, 1999.

[19] Holland, J. H. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[20] Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[21] Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence (Complex A)*. MIT Press, Cambridge, MA, 1992.

[22] Miettinen, K., Neittaanmaki, P., Periaux, J. (eds) *Evolutionary Algorithms in Engineering and Computer Science: Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, Genetic Programming, and Industrial Applications.* Wiley, Chichester, 1999.

[23] Shapiro, B. A., Navetta, J. "A massively parallel genetic algorithm for RNA secondary structure prediction." *J. Supercomput.*, **8**, 195-205, 1994.

[24] Koza John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection,*The MIT Press, Cambridge, Massachusetts, 1992.

[25] Michalewicz Zbigniew, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, third edition, New York, 1996.

[26] Studnicka, G. M., G. M. Rahn, I. W. Cummings and W. A. Salser. “Computer Method for Predicting the Secondary Structure of Single-stranded RNA.” *Nucl. Acids Res.* **5**, 3365-3387, 1978.

[27] Stiegler, P., P. Carbon, M. Zuker, J. P. Ebel and C. Ehresmann. “Structural Organization of the *16S* Ribosomal RNA from *E. coli*. Topography and Secondary Structure.” *Nucl. Acids Res.* **9**, 2153-2172, 1981.

[28] Osterburg, G. and R. Sommer. “Computer Support of DNA Sequence Analysis.” *Comput. Programs Biomed.* 13, 101-109, 1981.

[29] Lapalmee, G., R. J. Cedergren and D. Sankoff. “An Algorithm for the Display of Nucleic Acid Secondary Structure.” *Nucl. Acids Res.* **10**, 8351-8356, 1982.

[30] Shapiro, B. A., L. E. Lipkin and J. Maizel. “An Interactive Technique for the Display of Nucleic Acid Secondary Structure.” *Nucl. Acids Res.* **10**, 7041-7052, 1982.

[31] Nussinov, R., G. Pieczenik, J. R. Griggs and D.J. Kleitman. “Algorithms for Loop Matchings.” *SIAM J. appl. Math.* **35**, 68-82, 1978.

[32] Toribio Vicente Jose M. “Programación utilizando C threads.” Febrero 1997, Universidad de Valladolid (España). http://dsl.upc.es/netscout/doc/tutorial-threads/PROLOGO.htm

[33] Gultyaev Alexander P., van Batenburg F. H. D. and Pleij Cornelis W. A. “The Computer Simulation of RNA Folding Pathways Using a Genetic Algorithm” *J. Mol. Biol.* **250**, 37-51, 1995.