TESIS DEFENDIDA POR

Fermin Alberto Armenta Cano

Y APROBADA POR EL SIGUIENTE COMITÉ

Dr. José Alberto Fernández Zepeda

Director del Comité

Dr. Andrey Chernykh

Miembro del Comité

Dr. Carlos Alberto Brizuela Rodríguez

Miembro del Comité

Dr. Ricardo Arturo Chávez Pérez

Miembro del Comité

Dr. Hugo Homero Hidalgo Silva

Coordinador del programa de posgrado en Ciencias de la Computación

Dr. David Hilario Covarrubias Rosales

Director de Estudios de Posgrado

30 de Junio de 2011.

CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA

PROGRAMA DE POSGRADO EN CIENCIAS EN CIENCIAS DE LA COMPUTACIÓN

SIMULACIÓN DE UN R-MESH EN UN LR-MESH UTILIZANDO EL ALGORITMO DE TRIFONOV

TESIS

que para cubrir parcialmente los requisitos necesarios para obtener el grado de

MAESTRO EN CIENCIAS

Presenta:

FERMIN ALBERTO ARMENTA CANO

Ensenada, Baja California, México, Junio de 2011

RESUMEN de la tesis de **FERMIN ALBERTO ARMENTA CANO**, presentada como requisito parcial para la obtención del grado de MAESTRO EN CIENCIAS en CIENCIAS DE LA COMPUTACIÓN . Ensenada, Baja California, Junio de 2011.

SIMULACIÓN DE UN R-MESH EN UN LR-MESH UTILIZANDO EL ALGORITMO DE TRIFONOV

Resumen aprobado por:

Dr. José Alberto Fernández Zepeda

Director de Tesis

Este trabajo de tesis se ubica en el área de cómputo paralelo, específicamente dentro de los modelos reconfigurables. Estos modelos se dividen en varias ramas, una de las más estudiada es la arquitectura con ductos reconfigurables, específicamente la rejilla reconfigurable (R-Mesh). Debido a que la propagación de cualquier señal por los ductos de este modelo se supone constante (sin importar su longitud) y debido a su capacidad de utilizar sus ductos como herramientas computacionales, este modelo es más poderoso que muchos de los modelos paralelos convencionales, tales como la PRAM. El diseño de algoritmos para el modelo R-Mesh resulta fácil ya que sus ductos facilitan la comunicación entre los procesadores; sin embargo, su implementación es compleja. Por otro lado, existen versiones restringidas de dicho modelo, como el LR-Mesh, cuya implementación es más factible; sin embargo, el diseño de algoritmos en él es más elaborado. Al contar con una simulación óptima en recursos entre los modelos R-Mesh y LR-Mesh se ganan los beneficios y se eliminan los inconvenientes de ambos.

En el presente trabajo de tesis se proponen dos estrategias para diseñar un algoritmo de simulación de un R-Mesh sobre un LR-Mesh. El algoritmo de la Simulación 1 restringe al mínimo la cantidad de procesadores que se utilizan, sin prestar atención al tiempo de ejecución. Debido a esta restricción, el algoritmo se ejecuta sobre un LR-Mesh de $O(N^2 \times N^2)$ procesadores con un tiempo de ejecución de $O(\log N)$ unidades de tiempo. Por otro lado, el algoritmo de la Simulación 2, restringe el tiempo de ejecución al mínimo posible de acuerdo al algoritmo de Trifonov, sin prestar atención al número de procesadores. Debido a esta restricción, el tiempo de ejecución de dicha simulación es $O(\log \log N)$ unidades de tiempo.

El objetivo de diseñar la Simulación 1 es tener un punto de partida desde el cual se pudiera ir optimizando paulatinamente el tiempo de ejecución, sin sacrificar en gran medida al número de procesadores. Por otro lado, el objetivo de diseñar la Simulación 2 es similar al de la Simulación 1, tener un algoritmo de simulación inicial a partir del cual se pudiera ir optimizando paulatinamente el número de procesadores, sin sacrificar el tiempo de ejecución.

Palabras Clave: R-Mesh, LR-Mesh, Simulación de modelos reconfigurables.

ABSTRACT of the thesis presented by **FERMIN ALBERTO ARMENTA CANO**, as a partial requirement to obtain the MASTER IN SCIENCES degree in COMPUTER SCIENCES. Ensenada, Baja California, Junio 2011.

SIMULATION OF A R-MESH ON A LR-MESH USING TRIFONOV'S ALGORITHM

The present thesis focuses in the area of parallel computing, specifically on reconfigurable models. The reconfigurable models are divided in several areas, the area of reconfigurable bus architectures is the most studied of all of them, specifically the reconfigurable mesh (R-Mesh). The R-Mesh assumes that any signal propagates on its buses in constant time (regardless the length of the bus) and is capable of using the bus as computational tool; this model is more powerful than other convencional parallel models such as the PRAM. It is easy to design algorithms for the R-Mesh, since the characteristics of the bus facilitates the comunication among processors; however, its implementation is complicated. On the other hand, there are some restricted versions of this model, one of them is the LR-Mesh, whose implementation is more feasible; however, the algorithm design is more elaborated for this model. It is important to design an efficient simulation between these two models, so we can have the benefits and eliminate the disadvantages of both models.

In this work, we propose two strategies to design an algorithm that simulates a R-Mesh on a LR-Mesh. The algorithm of the Simulation 1 restricts the number of processors to the minimum, regardless the execution time. Due to this restriction, the algorithm executes on a LR-Mesh of $O(N^2 \times N^2)$ processors, with an execution time of $O(\log N)$ time units. On the other hand, the algorithm of Simulation 2 restricts the execution time to the minimum, according to Trifonov's algorithm, without considering the number of processors. Due to this restriction, the execution time of this simulation is $O(\log \log N)$ time units.

The purpose of Simulation 1 is to have a basic algorithm, from which we could gradually optimize its execution time, without sacrificing the number of processors. On the other hand, the purpose of Simulation 2 is to have a basic algorithm, from which we could gradually optimize its number of processors, without sacrificing the execution time.

Keywords: R-Mesh, LR-Mesh, reconfigurable models simulation.

A mis padres

Fernado J. Armenta Zavala y Ma. Teresa Cano Diaz, por el gran cariño y apoyo que me han brindado incondicionalmente durante toda mi vida. Gracias a ustedes he cumplido muchos de mis sueños.

A Vanessa Miranda

Por soportarme todo este tiempo y por haber aceptado la invitación a ver una película en mi casa y ya nunca marcharte.

A mis hermanos

Juan Miguel y Fernando Javier así como a mis sobrinos Wiki y Fernando III, por que la familia es lo más importante en el mundo.

Agradecimientos

A mi asesor el Dr. José Alberto Fernández Zepeda por toda su ayuda.

A los miembros del comité de tesis:
Dr. Carlos Brizuela Rodríguez, Dr. Andrey Chernykh y Dr. Ricardo Chávez Pérez

A todos mis amigos de la generación y del cubo 103, por todos esos buenos momentos que pasamos juntos.

A todos los investigadores y personal del departamento de computación por su enseñanza académica, en especial a Caro y Lidia por sus sabios consejos que siempre me ponían de buen humor.

Al CONACyT y proyecto de investigación por su apoyo económico.

Contenido

Página

Contenido (continuación)

Contenido (continuación)

Lista de Figuras

Lista de Figuras (continuación)

Capítulo I

Introducción

Existe una gran variedad de modelos computacionales, así como muchas clasificaciones basadas en diversos criterios. Uno de estos criterios es el número de procesadores, el cual permite dividir a los modelos computacionales en dos grandes ramas, los secuenciales (los que tienen un solo procesador) y los paralelos (que tienen dos o más procesadores). El presente trabajo de tesis se enfoca y se relaciona con el grupo de modelos computacionales paralelos; los modelos de cómputo paralelo y distribuido que se han propuesto hoy en día, surgen de la necesidad de procesar una gran cantidad de información en tiempo reducido. La Figura 1 muestra la clasificación de los modelos computacionales antes mencionada; en esta clasificación se puede encontrar el modelo PRAM (Parallel Random Access Machine) (JáJá, 1992), el Torus Polimórfico (Li y Maresca, 1989), los grafos acíclicos dirigidos (Piatt *et al.*, 2000) y los modelos reconfigurables (RN (Ben-Asher *et al.*, 1991), RMBM (Trahan *et al.*, 1996), PARBS (Wang y Chen, 1990)), entre otros. Asimismo, dentro de los modelos reconfigurables, una de sus ramas más estudiada es la arquitectura con ductos reconfigurables, específicamente la rejilla reconfigurable (R-Mesh).

Las arquitecturas con ductos reconfigurables se han desarrollado con base en una gran variedad de técnicas y algoritmos (Bertossi y Mei, 2000; Bokka *et al.*, 1995; Chao *et al.*, 1996). Estas arquitecturas consisten de un conjunto de procesadores, los cuales se encuentran conectados por medio de un conjunto de ductos. Cada procesador internamente tiene la capacidad de conectar los ductos externos y formar ductos más largos

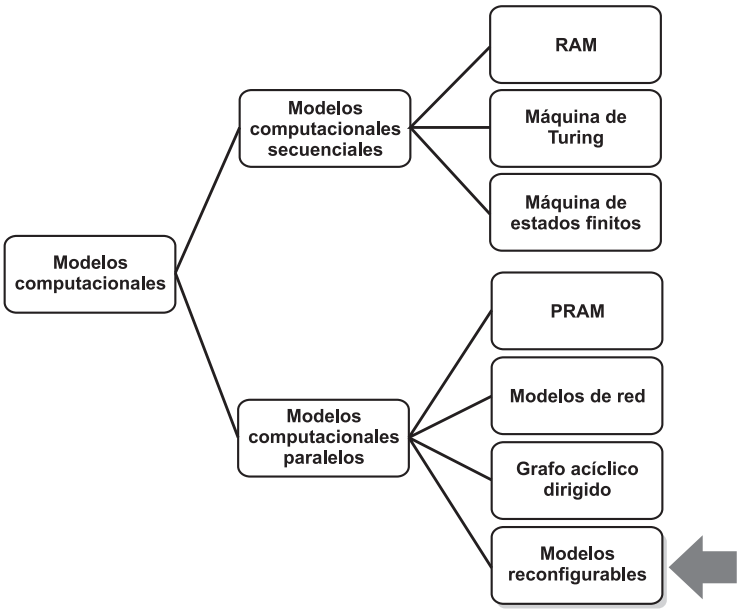

Figura 1. Modelos computacionales.

que permiten agilizar el proceso de comunicación entre los procesadores. En la Figura 2 se muestran la clasificación de los modelos reconfigurables y se señalan los dos modelos en los cuales se enfoca el presente tema de investigación: la rejilla reconfigurable (Reconfigurable Mesh o R-Mesh) y la rejilla reconfigurable lineal (Linear Reconfigurable Mesh o LR-Mesh).

Dentro de las arquitecturas con ductos reconfigurables, el R-Mesh es uno de los modelos más estudiados (Miller *et al.*, 1993; Li y Stout, 1991). Los modelos LR-Mesh y R-Mesh son capaces de resolver los problemas de las clases L y SL en tiempo constante, respectivamente. Antes del 2005 se creía que el LR-Mesh era computacionalmente menos poderoso que el R-Mesh. En el 2005, Reingold (2005) propuso un algoritmo que resuelve el problema USTCON en la máquina de Turing determinística con espacio logarítmico. El problema USTCON (undirected s - t conectivity) se define de la siguiente forma: dado un grafo no dirigido $G = (V, E)$ y dos vértices $s, t \in V$ se desea determinar si existe una trayectoria en G que conecte a "s" con "t") requiriendo $O(\log N)$ unidades de espacio

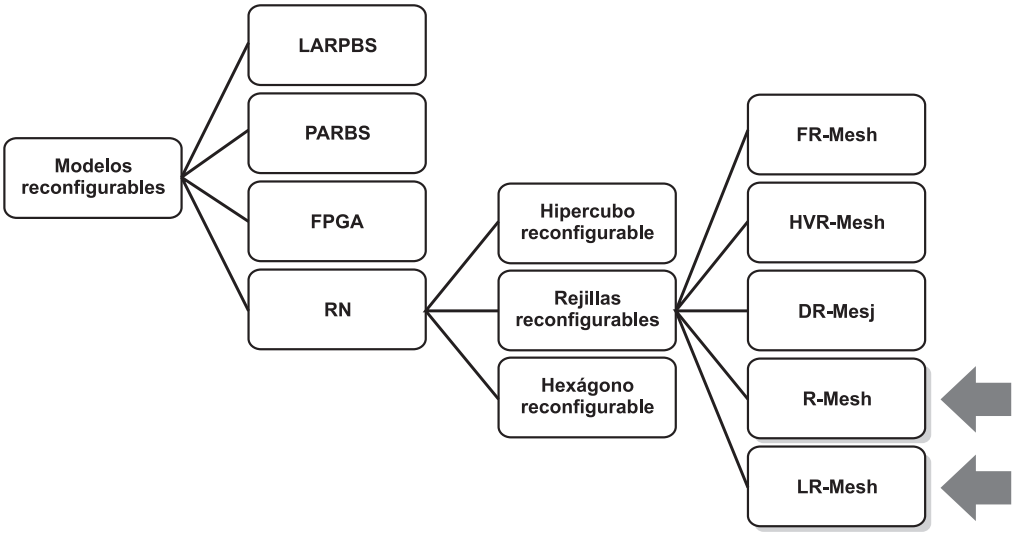

Figura 2. Clasificación de los modelos reconfigurables

(donde N es el tamaño de la entrada) (Alvarez y Greenlaw, 2000).

Este resultado implica que las clases de complejidad L y SL son equivalentes (ver Sección II.3.1). Como el modelo LR-Mesh puede resolver todos los problemas pertenecientes a la clase de complejidad L, por lo tanto, los resultados de Reingold también implican que los modelos LR-Mesh y R-Mesh son computacionalmente igual de poderosos. Se dice que dos modelos X y Y son igual de poderosos, si y sólo si X puede simular un ciclo de máquina arbitrario de Y en tiempo constante y viceversa. Por otro lado, y de forma independiente al trabajo de Reingold, utilizando una técnica diferente, Trifonov (2005) desarrolló un algoritmo en la máquina de Turing determinística que resuelve el problema USTCON en $O(\log N \log\log N)$ espacio.

En este trabajo de tesis, se plantea el diseño de una simulación del modelo R-Mesh en el LR-Mesh, siguiendo las técnicas utilizadas en el algoritmo de Trifonov. Se persigue un balance entre el tiempo de ejecución y el número de procesadores requeridos para dicha simulación. Este enfoque permitiría obtener un tiempo de ejecución de $O(\log\log N)$ con la menor cantidad de procesadores posibles.

A continuación se muestra una breve descripción de los trabajos previos antes mencionados, la motivación para realizar este trabajo, así como la definición formal del problema, la metodología que se utiliza, y finalmente la estructura del documento.

I.1. Trabajos Previos

Fernández-Zepeda *et al.* (2002) diseñaron una simulación de un R-Mesh de $N \times N$ procesadores en un LR-Mesh de $N \times N$ procesadores. La simulación se basa en la técnica de linealización de ductos, cuyo objetivo es transformar cualquier ducto no lineal (aceptado por un R-Mesh) a un ducto lineal equivalente (aceptado por un LR-Mesh). Esta simulación es óptima en el número de procesadores y fue la simulación más rápida hasta el 2005. Su tiempo de ejecución es $O(\log N)$.

La linealización de un ducto es una técnica que se aplica a cualquier ducto no lineal. Esta técnica consiste primero en modelar al ducto como si fuera un grafo, donde los nodos son los puertos por donde pasa el ducto; después se crea iterativamente el árbol de esparcimiento de dicho grafo (un *árbol de esparcimiento* de un grafo G es el subgrafo conectado T de G que contiene todo los vértices de G y ningún ciclo). Una vez encontrado el árbol T, se crea un "pseudo-tour de Euler" de T (un *tour de Euler* es un ciclo dirigido en algún grafo M que recorre en forma cerrada cada arista de M una sola vez). Para crear el tour de Euler, el árbol T se transforma en un grafo T', si cada arista (v_1, v_2) en T se reemplaza por dos aristas dirigidas $\langle v_1, v_2 \rangle$ y $\langle v_2, v_1 \rangle$. Al hacer esta transformación, T' se convierte en un grafo Euleriano y por lo tanto tiene un tour de Euler. El *pseudo-tour de Euler* se genera eliminando una de las arista que conforma el ciclo en T', generalmente la última arista que llega a la raíz. El grafo resultante es la versión lineal del ducto original. El pseudo-tour de Euler permite al LR-Mesh manejar la estructura de cualquier ducto

del R-Mesh como si fuera un ducto lineal. A continuación se describe en forma general los pasos que requiere esta simulación.

En Fernández-Zepeda *et al.* (2002) se define la representación de una configuración arbitraria de un R-Mesh como un grafo G, donde los nodos y las aristas de G representan a los puertos y a los ductos del R-Mesh, respectivamente. Sean v_1 y v_2 dos nodos del grafo G. Existe una arista en G entre los nodos v_1 y v_2 si y sólo si existe una conexión entre el puerto p_1 (simulado por v_1) y p_2 (simulado por v_2) en el R-Mesh.

Para realizar la simulación se ejecutan los siguientes pasos:

1. Se identifica el tipo de ducto como no lineal, lineal cíclico o lineal acíclico.
2. Se eliminan los ciclos de los ductos lineales.
3. Se construye el grafo podado (*raked graph*), esto es, removiendo las cadenas de nodos lineales que estén conectados a nodos no lineales.
4. Se genera un grafo destilado (*distilled graph*) para crear un árbol de esparcimiento.
5. Se genera el árbol de esparcimiento y el pseudo-tour de Euler para cada componente conectado del grafo destilado.
6. Se fusionan los árboles de esparcimiento (esta operación es la de mayor costo ya que requiere $O(\log N)$ unidades de tiempo).

Al finalizar los pasos anteriores, se tiene un grafo lineal acíclico, el cual se puede simular en el LR-Mesh. Asimismo, en Fernández-Zepeda *et al.* (2002) se muestra cómo realizar una elección de líder en tiempo constante en un ducto acíclico, de esta manera las escrituras en el ducto se pueden simular en tiempo constante. El número total de pasos

que requiere la simulación es $O(\log N)$ ya que la fusión de los árboles de esparcimiento requiere $O(\log N)$ unidades de tiempo.

Posteriormente, Córdova-Flores *et al.* (2007) utiliza el algoritmo desarrollado por Reingold (2005) para diseñar una simulación del modelo R-Mesh en el LR-Mesh en tiempo constante, siendo ésta la primera en su clase; sin embargo, esta simulación requiere una gran cantidad de procesadores haciendo que su implementación práctica no sea factible. Córdova aprovecha el algoritmo de Reingold (2005) para resolver el problema de componentes conectados en el grafo que simula a un ducto no lineal. La parte más importante del algoritmo de Reingold (2005) es la transformación de un grafo regular cualquiera en un expansor (un *expansor* es un grafo disperso con alta conectividad). Una vez generado el expansor, determinar la conectividad entre dos vértices cualesquiera es trivial.

Córdova-Flores *et al.* (2007) presentan la simulación en tres fases, las cuales corren en tiempo constante. Estas fases se describen brevemente a continuación.

1. Transforma el grafo G que representa los ductos no lineales de un R-Mesh en un expansor. Para este paso se utiliza el algoritmo de Reingold (2005).

2. Resuelve el problema de componentes conectados en el expansor de G (a través de resolver el problema USTCON para todas las combinaciones de pares de nodos, utilizando fuerza bruta). Esto equivale a calcular la cerradura transitiva (transitive clousure) de G.

3. Maneja ductos no lineales como ductos lineales. Usa la información de la matriz de adyacencias de la transitive clousure para manejar cada ducto no lineal aceptado por un R-Mesh como un ducto lineal aceptado por un LR-Mesh; desarrollando esta transformación la simulación de un R-Mesh en un LR-Mesh es directa.

La diferencia principal entre el trabajo de Córdova-Flores *et al.* (2007) y el de Fernández-Zepeda *et al.* (2002) consiste en que el primero es un algoritmo que corre en $O(1)$ unidades de tiempo, pero utiliza una cantidad extremadamente grande de procesadores, aproximadamente $O(N^{60})$, haciendo imposible su implementación en la práctica. Por otro lado, la simulación de Fernández-Zepeda *et al.* (2002) utiliza una cantidad óptima de procesadores, esto es $O(N^2)$; sin embargo, su tiempo de ejecución es relativamente alto, $O(\log N)$ unidades de tiempo.

I.2. Motivación

El contar con una simulación óptima en recursos entre los modelos R-Mesh y LR-Mesh hace que se ganen los beneficios y se eliminan los inconvenientes de ambos modelos. Por un lado, el diseño de algoritmos para el modelo R-Mesh resulta más fácil ya que sus ductos no lineales facilitan la comunicación entre los procesadores; sin embargo, la implementación de ductos no lineales es compleja. Por otro lado, la implementación de un LR-Mesh es muy viable, de hecho existen arquitecturas basadas en fibras ópticas que se comportan en forma muy similar al LR-Mesh; sin embargo, el diseño de algoritmos en el LR-Mesh es más elaborado.

Así que si se quiere implementar un cierto algoritmo paralelo en una arquitectura reconfigurable se tendría que diseñar primero el algoritmo como si fuera a correr en un R-Mesh (lo cual es fácil). Después se transforma el algoritmo diseñado para el R-Mesh para que corra en un LR-Mesh (lo cual sería eficiente, si la transformación lo fuera). Finalmente, se implementa el algoritmo del LR-Mesh (lo cual es muy factible debido a las características de dicho modelo). Por lo anterior, el diseñar una simulación eficiente entre ambos modelos es de gran importancia. En nuestro trabajo no se pretende realizar

una simulación en tiempo constante (esa ya existe), se busca un balance entre el tiempo de ejecución y la cantidad de procesadores requeridos para la simulación.

Actualmente existen algunas simulaciones de un R-Mesh en un LR-Mesh; sin embargo, todas estas simulaciones involucran un sobre-costo ya sea en tiempo o en el número de procesadores requeridos. La idea de esta investigación es proponer un algoritmo nuevo para la simulación de un R-Mesh en un LR-Mesh, cuyo tiempo de ejecución sea $O(\log \log N)$. Adicionalmente, se busca que la cantidad de procesadores requeridos sea la menor posible.

I.3. Planteamiento del problema

Existen dos simulaciones del modelo R-Mesh en el Modelo LR-Mesh, la primera de Fernández-Zepeda *et al.* (2002) es eficiente en cuanto al número de procesadores que requiere, $O(N^2)$; sin embargo, es relativamente lenta dado que se ejecuta en $O(\log N)$ unidades de tiempo.

En la segunda simulación, Córdova-Flores *et al.* (2007) logra un tiempo de ejecución constante; sin embargo, dicha simulación tiene la desventaja de requerir un número astronómico de procesadores, lo que la hace interesante desde el punto de vista teórico, pero inviable desde el punto de vista práctico.

Por tal razón se busca llegar a un punto medio entre ambos resultados, lo cual es diseñar una nueva simulación que sea más rápida que la de Fernández-Zepeda *et al.* (2002) sin ser constante, y por otro lado que requiera mucho menos procesadores que la de Córdova-Flores *et al.* (2007).

Esta nueva simulación en teoría se puede basar en el algoritmo propuesto por Trifonov (2005) el cual resuelve el problema USTCON en una máquina de Turing de-

terminística con $O(\log N \log\log N)$ espacio. Por lo que, se conjetura que utilizando este algoritmo debe existir un algoritmo que simule un R-Mesh con un LR-Mesh en tiempo $O(\log\log N)$ y cuya cantidad de recursos computacionales se desconoce. En la Figura 3 se muestran las dos simulaciones antes mencionadas junto con la simulación que se pretende realizar.

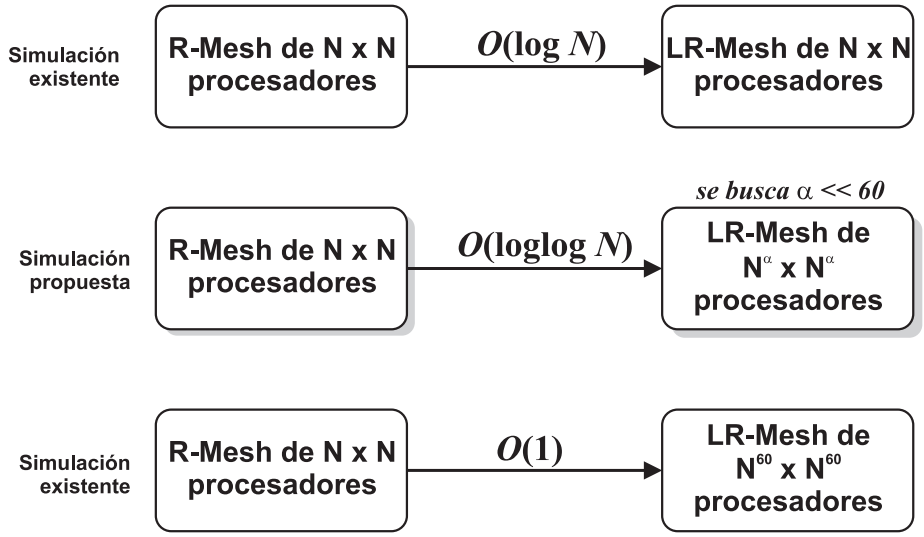

Figura 3. Simulaciones del modelo R-Mesh en el modelo LR-Mesh

I.3.1. Objetivo General

El objetivo general de este trabajo de tesis, consiste en plantear dos estrategias para el diseño de un algoritmo que simule un R-Mesh en un LR-Mesh cuyo tiempo de ejecución sea $O(\log\log N)$.

I.3.2. Objetivos Específicos

1. Calcular la cantidad de recursos computacionales requeridos por las simulaciones con ambas estrategias.

2. Diseñar un algoritmo que resuelva la estrategia de "enganchamiento" (Hooking) utilizada por el algoritmo de Trifonov (2005) en un LR-Mesh en $O(1)$.

3. Diseñar un algoritmo que resuelva la estrategia de "contracción" (Contract) utilizada por el algoritmo de Trifonov (2005) en un LR-Mesh en $O(1)$.

4. Diseñar un mecanismo que permita ejecutar dos o más etapas de enganchamiento y contracción en tiempo constante.

I.4. Metodología

Como etapa previa para llevar a cabo la metodología de la investigación, se estudiaron los diferentes algoritmos existentes para simulación de modelos reconfigurables, enfocándose principalmente en simulaciones de un R-Mesh en un LR-Mesh.

La metodología se divide en las siguientes etapas:

1. Estudio de las arquitecturas reconfigurables.

   En esta etapa se estudiaron los modelos de reconfiguración, principalmente los modelos R-Mesh y LR-Mesh, con el objetivo de adquirir los conocimientos necesarios y desarrollar las habilidades para comprender, analizar y diseñar algoritmos en estos modelos.

2. Estudio de la máquina de Turing y las clases de complejidad.

   En esta etapa se estudió a detalle el modelo de la máquina de Turing con el objetivo de comprender su funcionamiento. Además, se estudiaron las clases de complejidad L y SL. Con base en estos estudios, se analizó la relación entre el espacio que requiere un problema P que se ejecuta en una máquina de Turing

determinística y el tiempo que requiere el problema P para ejecutarse en el modelo reconfigurable LR-Mesh.

3. Estudio del algoritmo que resuelve el problema USTCON en $O(\log N \log\log N)$ unidades de espacio en una máquina de Turing determinística.

   En esta etapa, se estudió a detalle el trabajo de Trifonov (2005) para comprender las etapas de enganchamiento y contracción utilizadas en su algoritmo. También se estudió el análisis que presenta en su trabajo para calcular el espacio requerido $O(\log N \log\log N)$ unidades de espacio para resolver el problema USTCON.

4. Desarrollo de estrategias para el diseño de las simulaciones de un R-Mesh en un LR-Mesh.

   Se proponen dos estrategias para el diseño de estas simulaciones. En la Simulación 1, se parte de restringir el número de procesadores sin tomar en cuenta el tiempo de ejecución para lograr, en un primer paso, un algoritmo que realice la simulación en $O(\log N)$ unidades de tiempo. En el segundo paso se busca reducir gradualmente el tiempo de ejecución hasta alcanzar $O(\log\log N)$ unidades de tiempo.

   En la Simulación 2, se restringe el tiempo sin tomar en cuenta el número de procesadores, para lograr en el primer paso, un algoritmo que corra en $O(\log\log N)$ unidades de tiempo. En el segundo paso se busca optimizar gradualmente el número de procesadores.

I.5. Organización del documento

El resto del presente documento se organiza como sigue. En el Capítulo II se presenta el marco teórico, donde se definen los conceptos básicos necesarios para comprender este tema de investigación. En el Capítulo III se explica a detalle los algoritmos de Trifonov (2005) y Chong y Lam (1993) que resuelven el problema USTCON en diferentes modelos computacionales; además, se muestran partes del algoritmo de Trifonov (2005) representados en la máquina de Turing. En el Capítulo IV se presenta la Simulación 1 del modelo R-Mesh en el modelo LR-Mesh, restringiendo la cantidad de procesadores y con un tiempo de ejecución de $O(\log N)$ unidades de tiempo. En el Capítulo V se presenta la Simulación 2 del modelo R-Mesh en el modelo LR-Mesh, restringiendo el tiempo de ejecución a $O(\log \log N)$ unidades de tiempo y sin restringir el número de procesadores. Por último, en el Capítulo VI se presentan las conclusiones finales del este trabajo de investigación y el trabajo a futuro.

Capítulo II

Marco Teórico

II.1. Máquina de Turing

El modelo de la máquina de Turing lo propuso Alan Turing en 1936 (Turing, 1936). A pesar de ser un modelo simple, esta máquina es capaz de efectuar cualquier operación que las computadoras modernas pueden realizar. Turing también demuestra que hay problemas que una máquina de Turing no puede resolver. La Figura 4 muestra el esquema representativo de una máquina de Turing.

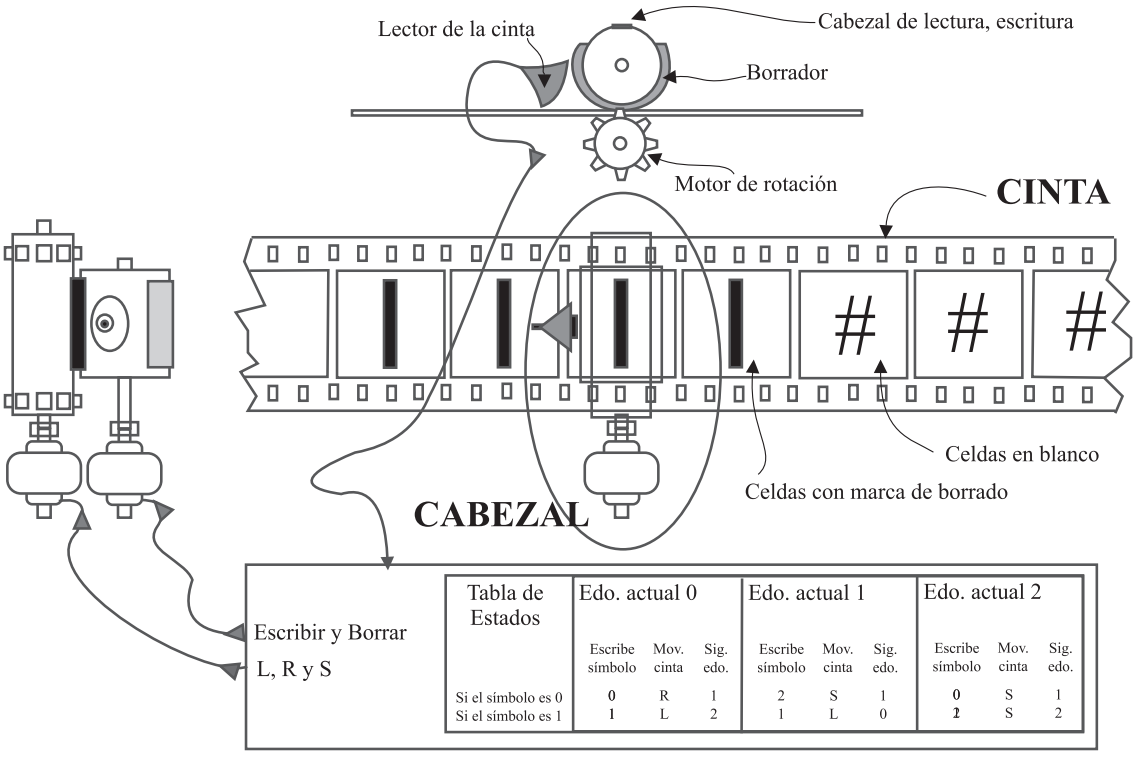

Figura 4. Esquema representativo de una máquina de Turing.

II.1.1. Máquina de Turing determinística

Existen varios tipos de máquinas de Turing; éstas se dividen en dos ramas, deterministas y no deterministas. A continuación se describen brevemente las máquinas de Turing deterministas y posteriormente las no deterministas.

La *Máquina de Turing determinística* (MTD) es el primer modelo teórico de lo que luego sería una computadora. Con el tiempo a este tipo de máquina se le conoció como "máquina de estado finito". La máquina de Turing consiste básicamente de un cabezal de lectura y escritura (el cual puede moverse hacia la izquierda o derecha), un registro de estado (el cual almacena el estado actual y tiene capacidad para un conjunto finito de estados) y una cinta infinita dividida en celdas (cada celda contiene un símbolo que pertenece al alfabeto, donde el alfabeto son todos los símbolos que puede reconocer la máquina de Turing).

La entrada se coloca en la cinta (la entrada es una cadena finita de símbolos que pertenecen al alfabeto) y el resto de la cinta contiene el símbolo especial "vacío" o "blanco", denotado por ♯. El cabezal de la máquina de Turing siempre está sobre una celda de la cinta, al inicio generalmente se coloca en la celda más izquierda con el primer símbolo de la cadena de entrada. En un paso, la máquina de Turing puede leer o escribir en la cinta, cambiar de estado (o quedarse en el mismo) y mover la cabeza a la izquierda o a la derecha. Si la máquina requiere almacenar información, escribe en la cinta; para leer la información que escribió, la máquina puede mover el cabezal hacia atrás.

La máquina de Turing sirve como un modelo preciso para la definición de algoritmos. En Sipser (1996) se definen tres niveles de detalles para describir un algoritmo en una máquina de Turing. El primero es la descripción formal que explica detalladamente y en su totalidad los estados de la máquina de Turing, función de transición y así sucesiva-

mente. Es el nivel de descripción más detallado. El segundo es una descripción de nivel intermedio, llamado descripción de implementación; en esta descripción se usa prosa para describir la forma en que la máquina de Turing mueve el cabezal y la forma en la que almacena los datos en la cinta. En este nivel lo que se hace es no dar detalle de los estados o de la función de transición.

El tercer nivel es la descripción de más alto nivel, donde se usa prosa para describir un algoritmo, ignorando los detalles de la implementación. En este nivel no es necesario mencionar cómo la máquina maneja su cinta o cabezal.

La máquina de Turing es un modelo muy importante en la teoría de la computación, ya que en este modelo es fácil calcular los recursos computacionales que se requieren en un problema dado, por lo cual sirve como referencia para comparar otros modelos de computación y así determinar el poder computacional de algún modelo con respecto a la máquina de Turing. Los dos recursos más importantes son:

Complejidad del tiempo

Generalmente el recurso más importante es el tiempo. La máquina de Turing puede calcular de manera precisa la cantidad de tiempo que se requiere para ejecutar un algoritmo, dado que el número de pasos que requiere la máquina de Turing es equivalente al tiempo que requiere el algoritmo para ejecutarse. Si la máquina de Turing no se detiene entonces el tiempo se considera indefinido.

Complejidad del espacio

Otro recurso computacional importante es la cantidad de memoria utilizada por un algoritmo para su ejecución, es decir, el espacio. La medición de complejidad correspondiente se conoce como complejidad del espacio. La máquina de Turing puede calcular de manera precisa el espacio requerido para ejecutar un algoritmo, dado que la cantidad de celdas que utiliza la máquina de Turing para ejecutar dicho algoritmo es equivalente

al espacio que requiere el algoritmo para ejecutarse.

A continuación se describe formalmente la máquina de Turing tradicional así como algunas de sus variantes, las cuales tienen el mismo poder computacional (reconocen la misma clase de lenguajes).

II.1.2. Máquina de Turing determinística de una cinta

Una máquina de Turing determinística de una cinta se define como una tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ donde:

Q es un— conjunto de estados

Σ es el alfabeto finito de entrada

Γ es el alfabeto de la cinta, donde $\sharp \in \Gamma$ es el carácter blanco y $\Sigma \subseteq \Gamma$

$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ es la función de transición donde L (left) significa un movimiento a la izquierda y R (right) un movimiento a la derecha.

$q_0 \in Q$ es el estado de inicio

$q_a \in Q$ es el estado de aceptación

$q_r \in Q$ es el estado de rechazo, donde $q_a \neq q_r$

II.1.3. Máquina de Turing de k-cintas

La máquina de k-cintas es similar a la de una cinta, cada cinta contiene su propio cabezal para leer y escribir. Normalmente la entrada se coloca en la cinta número uno y las $k - 1$ cintas restantes se encuentran en blanco.

En forma similar a la máquina de Turing de una cinta, se define la máquina de Turing de k-cintas como una tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ donde : $Q, \Sigma, \Gamma, q_0, q_a, q_r$ se definen de la misma forma que en la máquina de Turing de una cinta. La función de

transición se define de la siguiente forma: $\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, S\}^k$. Donde S (stay) significa que el cabezal no se mueve.

II.1.4. Máquina de Turing no determinística

La *Máquina de Turing no determinística* (MTND) es similar a la MTD. La diferencia consiste en que una MTND, en un estado arbitrario y viendo cierto carácter en el cabezal, puede tener cero, una o más posibles transiciones de estado, donde puede elegir cualquiera de ellas.

Formalmente en una MTND se define de la siguiente forma: $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ donde: $Q, \Sigma, \Gamma, q_0, q_a, q_r$ se definen como en la máquina de Turing determinística.

La función de transición se define de la siguiente forma:

$\delta : Q \times \Gamma \longrightarrow P(Q \times \Gamma \times \{L, R\})$, donde P son las posibles transiciones.

El cálculo en una MTND se puede ver como un árbol enraizado, el cual se bifurca en todas las posibles transiciones, ver Figura 5. Si en cualquiera de sus bifurcaciones llega a un estado de aceptación, la máquina acepta la entrada. La profundidad del árbol indica el número de pasos que realizó, siendo equivalente al tiempo de ejecución.

II.1.5. Máquina de Turing simétrica

La *Máquina de Turing simétrica* (MTS) la propusieron Lewis y Papadimitriou (1982). Esta máquina de Turing es parecida a la MTND con poder más limitado; en una MTS, cada movimiento de la máquina se puede revertir, de esta forma la MTS puede leer dos símbolos al mismo tiempo en cada una de sus cintas.

Parecida a la Máquina de Turing de k-cintas, Lewis y Papadimitriou (1982) definen

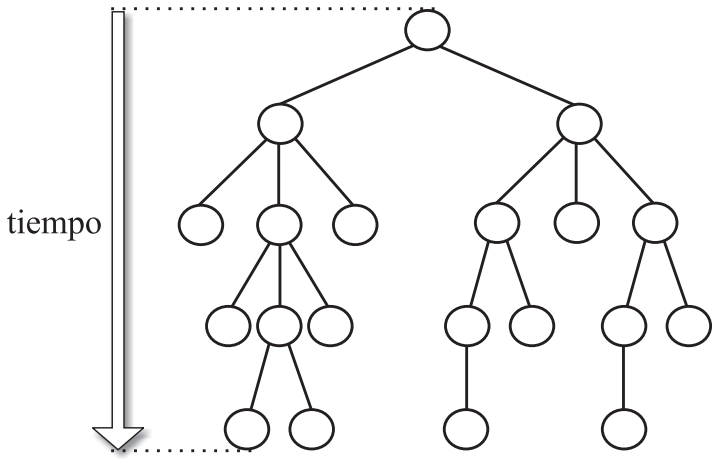

Figura 5. Ejemplo de transiciones en una MTND como árbol enraizado.

la MTS de la siguiente manera: Una máquina de k cintas de longitud infinita con un cabezal en cada cinta, es una 7-tupla $(K, \Sigma, \Sigma_0, k, \Delta, s, F)$ donde:

K es el conjunto finito de estados.

Σ es el alfabeto finito (el alfabeto de la cinta).

$\Sigma_0 \subseteq \Sigma$, es el alfabeto de entrada.

$k > 0$, es el número de cintas.

$s \in K$, es el estado inicial.

$F \subseteq K$, es el conjunto de estados finales.

Δ es el conjunto finito de transiciones.

La diferencia de la MTS con respecto a la de k-cintas son las transiciones. Las transiciones permiten a la MTS observar una celda a la derecha o a la izquierda, mientras se mueven a la derecha o a la izquierda, respectivamente. Específicamente, una transición es de la forma $(p, t_1, ..., t_k, q)$, donde p y q son estados, k es el número de cintas y $t_1, ..., t_k$ son tripletas de la cinta. La tripleta de la cinta se representa como (ab, D, cd), donde $a, b, c, d \in \Sigma$ y además D puede ser $+1$ o -1 o expresado como $(a, 0, b)$, en donde $a, b \in \Sigma$.

Para simplificar la explicación, Lewis y Papadimitriou (1982) consideran el caso

de una cinta. La transición de la forma $(p, (a, 0, b), q)$ significa que cuando la MTS se encuentra en el estado p y lee el símbolo a, la MTS puede reescribir tanto a como b y moverse al estado q, sin mover el cabezal. La transición de la forma $(p, (ab, +1, cd), q)$ se da cuando la MTS se encuentra en el estado p, lee el símbolo a y la celda justo a la derecha de la celda leída contiene un símbolo b, la MTS puede reescribir estas dos celdas con los símbolos c y d, respectivamente y mover el cabezal una celda a la derecha, y cambiar al estado q. En forma similar la transición $(p, (ab, -1, cd), q)$ significa un movimiento potencial hacia la izquierda excepto que ahora se lee el símbolo b y el símbolo a su izquierda es a y se reescriben por d y c, respectivamente.

II.1.6. Notación modular para las máquinas de Turing

La notación modular se utiliza para simplificar la notación convencional de las máquinas de Turing, la cual por lo general es bastante engorrosa. En esta sección se define una notación para la máquina de Turing que permite expresarla de una forma mucho más sencilla. Navarro (2009) define la notación modular en la máquina de Turing como un grafo donde los nodos representan las acciones a ejecutarse y las aristas denotan las condiciones. En cada nodo se puede escribir una secuencia de acciones que se ejecutan al llegar al nodo. Luego de ejecutarlas, se consideran las aristas que salen del nodo. Las aristas consisten de flechas rotuladas con símbolos del alfabeto Σ. Si la flecha que sale del nodo está rotulada con la letra que se tiene bajo el cabezal luego de ejecutar el nodo, entonces se sigue la flecha y se llega a otro nodo. Se permite rotular las flechas con conjuntos de caracteres. Existe un nodo inicial donde la MT comienza a operar y cuando un nodo no tenga otro a donde ir, la MT se detiene.

Las acciones básicas de la notación modular de una MT son:

- Moverse hacia la izquierda ($\triangleleft$). Sin importar el símbolo en el que se encuentre, la MT se mueve hacia la izquierda una casilla y se detiene.

- Moverse hacia la derecha ($\triangleright$). Sin importar el símbolo en el que se encuentre, la MT se mueve hacia la derecha una casilla y se detiene.

- Escribir el símbolo $b \in \Sigma(b)$. Sin importar el símbolo que tenga, la MT escribe b en la cinta y se detiene.

- Moverse por la cinta ($\triangleright_A, \triangleleft_A$). Sin importar el símbolo que encuentre, la MT se mueve a la derecha o la izquierda, respectivamente, hasta encontrar en la cinta al símbolo $A \in \Sigma$.

- Borrar en la cinta (B). Sin importar el símbolo que encuentre, la MT borra la cadena que tiene hacia la izquierda (hasta encontrar el símbolo $\sharp$).

- Para una MT de k-cintas se utiliza el superíndice para indicar la cinta que realiza la operación. Por ejemplo, en la notación ($\triangleright_A^j$), el superíndice j indica que el cabezal en la j-ésima cinta se desplaza a la derecha hasta encontrar una $A \in \Sigma$.

En la Figura 6 se muestra el algoritmo que resuelve el problema de la suma de dos números de x bits utilizando la notación modular. Este algoritmo se describe en la siguiente sección.

II.1.7. Algoritmos representados en una MTD

En esta sección se muestran algunos algoritmos representados en la MTD.

1. Suma de dos números de x bits en la MTD

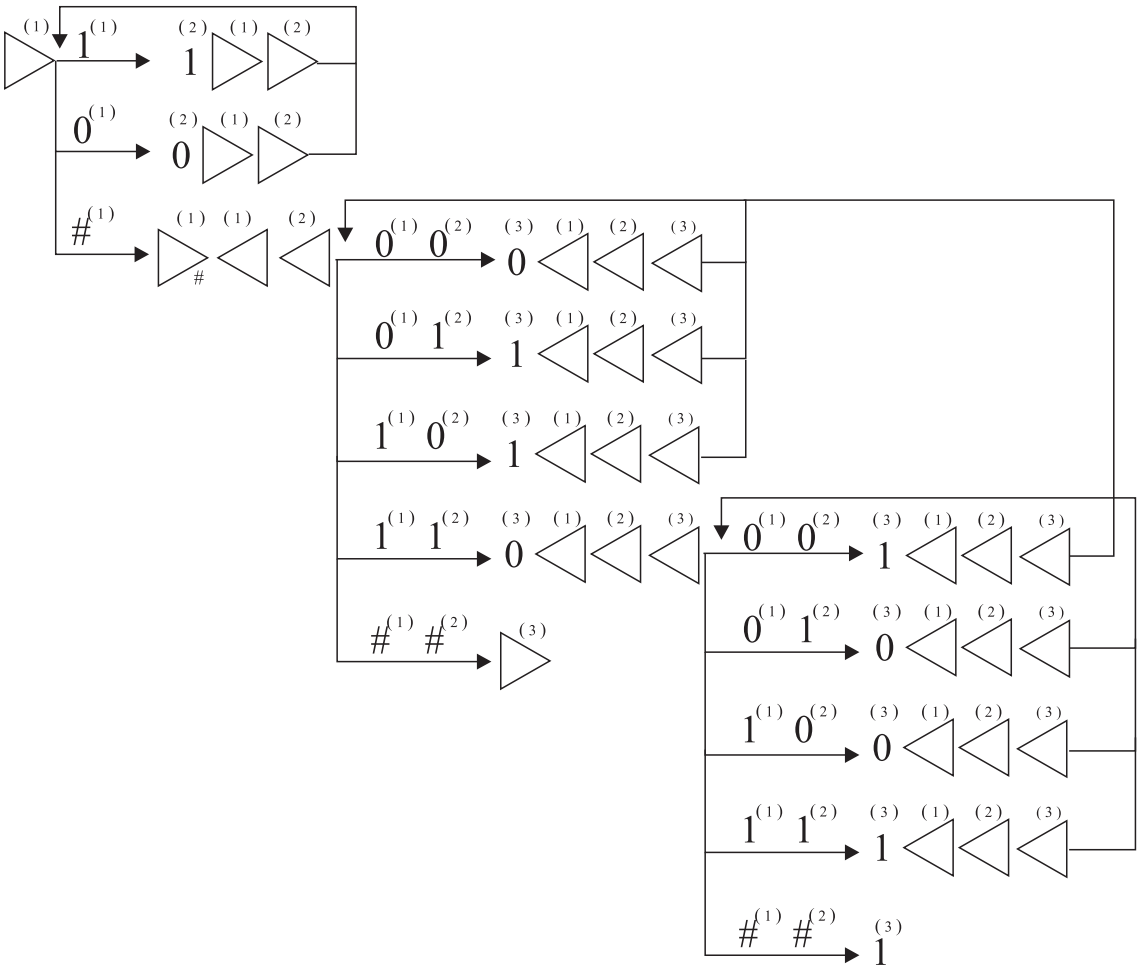

Figura 6. Suma de dos números de x bits utilizando notación modular.

Descripción: Se tiene $\Sigma = \{0, 1, \sharp\}$, los enteros positivos se representan en binario. Cada entero positivo se separa por el símbolo $\sharp$.

Construir una MTD que calcule $f(a, b) = a + b$

Solución en una máquina de Turing determinística.

Entrada: Al inicio se encuentran los dos números representados en binario, separados por el símbolo $\sharp$ en la cinta de entrada.

Salida: El resultado de la suma de los dos números se encuentra en la cinta de salida y la máquina termina en un estado de aceptación.

A continuación se muestra la descripción en alto nivel del algoritmo en la MTD

a) Copia el primer número de la cinta de entrada a la cinta de trabajo y el cabezal de la cinta de entrada queda posicionado en el símbolo ♯ que se encuentra antes del segundo número.

b) El cabezal de la cinta de entrada se recorre hacia la derecha hasta el bit menos significativo del segundo número.

c) El cabezal de la cinta de trabajo se posiciona en el bit menos significativo del primer número.

d) Se realiza la suma binaria ejecutando los siguientes pasos:

Si la MTD se encuentra en el estado sin acarreo, se ejecutan las siguientes reglas.

1) Si los cabezales de las cintas de entrada y trabajo están posicionados en un 0, se escribe un 0 en la cinta de salida. Todas las cintas se recorren una posición a la izquierda.
2) Si el cabezal de la cinta de entrada está posicionado en un 0 y el cabezal de la cinta de trabajo está posicionado en un 1, la cinta de salida escribe un 1. Todas las cintas se recorren una posición a la izquierda.
3) Si el cabezal de la cinta de entrada está posicionado en un 1 y el cabezal de la cinta de trabajo está posicionado en un 0, la cinta de salida escribe un 1. Todas las cintas se recorren una posición a la izquierda.
4) Si los cabezales de las cintas de entrada y trabajo están posicionados en un 1, se escribe un 0 en la cinta de salida y se genera un acarreo, por lo cual la MTD cambia al estado de acarreo. Todas las cintas se recorren una posición a la izquierda.

Si la MTD se encuentra en el estado de acarreo, se ejecutan las siguientes reglas.

1) Si los cabezales de las cintas de entrada y trabajo están posicionados en un 0, se escribe un 1 en la cinta de salida. La MTD regresa al estado sin acarreo. Todas las cintas se recorren una posición a la izquierda.
2) Si el cabezal de la cinta de entrada está posicionado en un 0 y el cabezal de la cinta de trabajo está posicionado en un 1, la cinta de salida escribe un 0. Todas las cintas se recorren una posición a la izquierda.
3) Si el cabezal de la cinta de entrada está posicionado en un 1 y el cabezal de la cinta de trabajo está posicionado en un 0, la cinta de salida escribe un 0. Todas las cintas se recorren una posición a la izquierda.
4) Si los cabezales de las cintas de entrada y trabajo están posicionados en un 1, se escribe un 1 en la cinta de salida. Todas las cintas se recorren una posición a la izquierda.

e) Si la cinta de entrada y trabajo se posicionan en un ♯ y la MTD se encuentra en el estado sin acarreo. La cinta de salida se mueve a la derecha y la MTD cambia al estado de aceptación.

f) Si la cinta de entrada y trabajo se posiciona en un ♯ y la MTD se encuentra en el estado de acarreo. La cinta de salida escribe un 1 y la MTD cambia al estado de aceptación.

El resultado se encuentra en la cinta de salida con el cabezal posicionado en el bit más significativo.

A continuación se muestra la descripción formal del algoritmo en la MTD.

Estado inicial : q_0

Estado de aceptación : q_5

Estados intermedios : q_1, q_2, q_3, q_4.

Alfabeto Σ :$\{0, 1, \sharp\}$

Tabla de transición

Regla 01 $q_0[\sharp, x, x] \longrightarrow q_1[(x, R)(x, S)(x, S)]$

Regla 02 $q_1[0, x, x] \longrightarrow q_1[(x, R)(0, R)(x, S)]$

Regla 03 $q_1[1, x, x] \longrightarrow q_1[(x, R)(1, R)(x, S)]$

Regla 04 $q_1[\sharp, x, x] \longrightarrow q_2[(x, R)(x, S)(x, S)]$

Regla 05 $q_2[0, x, x] \longrightarrow q_2[(x, R)(x, S)(x, S)]$

Regla 06 $q_2[1, x, x] \longrightarrow q_2[(x, R)(x, S)(x, S)]$

Regla 07 $q_2[\sharp, x, x] \longrightarrow q_3[(x, I)(x, I)(x, S)]$

Regla 08 $q_3[0, 0, x] \longrightarrow q_3[(x, I)(x, I)(0, I)]$

Regla 09 $q_3[0, 1, x] \longrightarrow q_3[(x, I)(x, I)(1, I)]$

Regla 10 $q_3[1, 0, x] \longrightarrow q_3[(x, I)(x, I)(1, I)]$

Regla 11 $q_3[1, 1, x] \longrightarrow q_4[(x, I)(x, I)(0, I)]$

Regla 12 $q_4[0, 0, x] \longrightarrow q_3[(x, I)(x, I)(1, I)]$

Regla 13 $q_4[0, 1, x] \longrightarrow q_4[(x, I)(x, I)(0, I)]$

Regla 14 $q_4[1, 0, x] \longrightarrow q_4[(x, I)(x, I)(0, I)]$

Regla 15 $q_4[1, 1, x] \longrightarrow q_4[(x, I)(x, I)(1, I)]$

Regla 16 $q_4[\sharp, \sharp, x] \longrightarrow q_5[(x, S)(x, S)(1, S)]$

Regla 17 $q_3[\sharp, \sharp, x] \longrightarrow q_5[(x, S)(x, S)(x, R)]$

Cada regla es de la siguiente forma $q_i[\alpha, \beta, x] \longrightarrow q_j[(x, C_1)(\alpha, C_2)(\alpha, C_3)]$ donde $\{\alpha, \beta\} \in \Sigma \cup [x]$ y $C_1, C_2, C_3 \in \{L, R, S\}$. Se interpreta de la siguiente manera: si la MTD se encuentra en el estado q_i y el cabezal de la cinta de entrada está posicionado sobre un α y además el cabezal de la cinta de trabajo está posicionado sobre un β, el cabezal de la entrada sin alterar la cinta (x implica que únicamente la cabeza lee la cinta, no la altera) realiza el movimiento indicado por C_1, el cabezal de la cinta de trabajo escribe un α en la cinta y realiza el movimiento indicado por C_2, y el cabezal de la cinta de salida escribe un α en la cinta y realiza el movimiento indicado por C_3. Por último la MTD pasa al estado q_j.

Para sumar los dos números enteros positivos, se copia bit a bit el primer número a la cinta de trabajo; esta operación requiere x pasos ya que es la cantidad de bits que se necesitan para representar el entero positivo. Dado que la suma se ejecuta revisando los dos números bit a bit, el algoritmo requiere $O(x)$ unidades de tiempo. Asimismo, el algoritmo requiere $O(x)$ unidades de espacio, dado que es la cantidad de bits que se copian a la cinta de trabajo para realizar la suma.

2. Grado de un vértice en la MTD

Descripción: Se tiene $\Sigma = \{0, 1, \sharp\}$, los enteros positivos se representan en binario.

Construir una MTD que calcule el grado de un vértice.

Solución en una MTD de k-cintas.

Entrada: En la cinta de entrada número 1 se encuentra la lista de adyacencias del grafo (donde cada vértice puede tener a lo más d vecinos). La lista de adyacencias se representa de la siguiente manera: Entre paréntesis se encuentra el vértice y

entre corchetes sus vecinos, seguido del siguiente vértice con sus vecinos y así sucesivamente. En la cinta de entrada 2, el identificador del vértice v_i, del cual se requiere calcular el grado. Todos los números se representan en binario.

Salida: El grado del vértice v_i en la cinta de salida.

A continuación se muestra la descripción en alto nivel del algoritmo en la MTD

La MTD realiza los siguientes pasos:

a) Se posiciona el cabezal de la cinta de entrada 1 y 2 en el bit más significativo del primer vértice de la lista de adyacencias del grafo y del vértice v_i, respectivamente.

b) Se realiza la comparación bit a bit entre los dos vértices.

c) Si se encuentra un bit diferente, se recorre el cabezal de la cinta de entrada 1 hasta posicionarse en el bit más significativo del siguiente vértice. El cabezal de la cinta de entrada 2 se regresa hasta el bit más significativo del vértice v_i y se repite el paso 2.

d) Si los vértices comparados son iguales, el cabezal de la cinta de entrada 1 se recorre una posición a la derecha.

e) Si el cabezal de la cinta de entrada 1 no encuentra un símbolo "[", el grado de v_i es cero, por lo tanto el cabezal de la cinta de salida escribe un cero. La máquina de Turing pasa a un estado de aceptación.

f) Si el cabezal de la cinta de entrada 1 encuentra un símbolo "[", se recorre a la derecha y cada vez que se posicione en un símbolo "(" quiere decir que existe un vecino, entonces la cinta de trabajo realiza la suma binaria $c = c + 1$ (al inicio $c = 0$).

Los pasos para ejecutar la suma $c = c + 1$ son los siguientes:

1) Si el cabezal de la cinta de trabajo está posicionado en un 1, se escribe un 0 en la cinta de trabajo y se mueve el cabezal a la izquierda.
2) Si el cabezal de la cinta de trabajo está posicionado en un 0, se escribe un 1 en la cinta de trabajo y el cabezal se mueve hasta el bit menos significativo y la MT cambia al estado final.

g) Si el cabezal de la cinta de entrada 1 se posiciona en un símbolo "]" quiere decir que no hay más vecinos, entonces toda la información de la cinta de trabajo se copia (el grado del vértice) a la cinta de salida y la MT cambia a un estado de aceptación.

El resultado se encuentra en la cinta de salida con el cabezal posicionado en el bit más significativo.

A continuación se muestra la descripción formal del algoritmo en la MTD.

Estado inicial : q_0

Estado de aceptación : q_{10}

Estados intermedios : q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9.

Alfabeto Σ :$\{0, 1, \sharp, (,), [,]\}$

Tabla de transición

// Inicio de la comparación

Regla 01 $q_0\{(, \sharp, x, x\} \longrightarrow q_1\{(x, R)(x, R)(x, S)(x, S)\}$

Regla 02 $q_1\{0, 0, x, x\} \longrightarrow q_1\{(x, R)(x, R)(x, S)(x, S)\}$

Regla 03 $q_1\{1, 1, x, x\} \longrightarrow q_1\{(x, R)(x, R)(x, S)(x, S)\}$

// El vértice a comparar no es igual.

Regla 04 $q_1\{0,1,x,x\} \longrightarrow q_2\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 05 $q_1\{1,0,x,x\} \longrightarrow q_2\{(x,R)(x,S)(x,S)(x,S)\}$

// Recorre el cabezal al siguiente vértice.

Regla 06 $q_2\{0,x,x,x\} \longrightarrow q_2\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 07 $q_2\{1,x,x,x\} \longrightarrow q_2\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 08 $q_2\{(,x,x,x\} \longrightarrow q_2\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 09 $q_2\{),x,x,x\} \longrightarrow q_2\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 10 $q_2\{[,x,x,x\} \longrightarrow q_3\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 11 $q_3\{(,x,x,x\} \longrightarrow q_3\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 12 $q_3\{0,x,x,x\} \longrightarrow q_3\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 13 $q_3\{1,x,x,x\} \longrightarrow q_3\{(x,R)(x,S)(x,S)(x,S)\}$

Regla 14 $q_3\{),x,x,x\} \longrightarrow q_3\{(x,R)(x,S)(x,S)(x,S)\}$

//Se posiciona en el siguiente vértice.

Regla 15 $q_3\{],x,x,x\} \longrightarrow q_4\{(x,R)(x,S)(x,S)(x,S)\}$

// Regresa el cabezal de la cinta 2.

Regla 16 $q_4\{x,0,x,x\} \longrightarrow q_4\{(x,S)(x,L)(x,S)(x,S)\}$

Regla 17 $q_4\{x,1,x,x\} \longrightarrow q_4\{(x,S)(x,L)(x,S)(x,S)\}$

// Regresa al inicio de las reglas.

Regla 18 $q_4\{x,\sharp,x,x\} \longrightarrow q_0\{(x,S)(x,S)(x,S)(x,S)\}$

// Encontró el vértice al cual se le va a calcular el grado.

Regla 19 $q_1\{(), \sharp, x, x\} \longrightarrow q_5\{(x, R)(x, R)(x, S)(x, S)\}$

// Si no tiene vecinos la cinta de salida escribe 0 y la MTD termina.

Regla 20 $q_5\{(, x, x, x\} \longrightarrow q_{10}\{(x, R)(x, R)(x, S)(0, S)\}$

Regla 21 $q_5\{[, x, x, x\} \longrightarrow q_6\{(x, R)(x, S)(x, S)(x, S)\}$

// Inicio de la suma $c = c + 1$ para obtener el grado.

Regla 22 $q_6\{(, x, x, x\} \longrightarrow q_7\{(x, R)(x, S)(x, L)(x, S)\}$

Regla 23 $q_6\{0, x, x, x\} \longrightarrow q_6\{(x, R)(x, S)(x, S)(x, S)\}$

Regla 24 $q_6\{1, x, x, x\} \longrightarrow q_6\{(x, R)(x, S)(x, S)(x, S)\}$

Regla 25 $q_6\{(), x, x, x\} \longrightarrow q_6\{(x, R)(x, S)(x, S)(x, S)\}$

Regla 26 $q_7\{x, x, 1, x\} \longrightarrow q_7\{(x, S)(x, S)(0, L)(x, S)\}$

Regla 27 $q_7\{x, x, \sharp, x\} \longrightarrow q_8\{(x, S)(x, S)(1, R)(x, S)\}$

Regla 28 $q_7\{x, x, 0, x\} \longrightarrow q_8\{(x, S)(x, S)(1, R)(x, S)\}$

Regla 29 $q_8\{x, x, 0, x\} \longrightarrow q_8\{(x, S)(x, S)(x, R)(x, S)\}$

Regla 30 $q_8\{x, x, 1, x\} \longrightarrow q_8\{(x, S)(x, S)(x, R)(x, S)\}$

// Termina la suma.

Regla 31 $q_8\{x, x, \sharp, x\} \longrightarrow q_6\{(x, S)(x, S)(x, S)(x, S)\}$

// No hay más vecinos.

Regla 32 $q_6\{], x, x, x\} \longrightarrow q_9\{(x, S)(x, S)(x, L)(x, S)\}$

// Copia la cinta de trabajo a la cinta de salida.

Regla 33 $q_9\{x, x, 0, x\} \longrightarrow q_9\{(x, S)(x, S)(x, L)(0, L)\}$

Regla 34 $q_9\{x, x, 1, x\} \longrightarrow q_9\{(x, S)(x, S)(x, L)(1, L)\}$

Regla 35 $q_9\{x, x, \sharp, x\} \longrightarrow q_{10}\{(x, S)(x, S)(x, S)(S, R)\}$

La regla es de la siguiente forma $q_i\{\alpha, \beta, x, x\} \longrightarrow q_j\{(x, C_1)(\alpha, C_2)(\alpha, C_3)(\alpha, C_4)\}$ donde $\{\alpha, \beta\} \in \Sigma \cup [x]$ y $C_1, C_2, C_3, C_4 \in \{L, R, S\}$. Se interpreta de la siguiente manera: si la MTD se encuentra en el estado q_i y el cabezal de la cinta de entrada 1 está posicionado sobre un α, el cabezal de la cinta de entrada 2 está posicionado sobre un β, y sin importar donde estén posicionados los cabezales de la cinta de trabajo y salida; entonces, el cabezal de la entrada 1 sin alterar la cinta (x implica que únicamente la cabeza lee la cinta no la altera) realiza el movimiento indicado por C_1, el cabezal de la cinta de entrada 2 escribe un α en la cinta y realiza el movimiento indicado por C_2. El cabezal de la cinta de trabajo escribe un α en la cinta y realiza el movimiento indicado por C_3. El cabezal de la cinta de salida escribe un α en la cinta y realiza el movimiento indicado por C_4. Finalmente, la MTD pasa al estado q_j.

Para encontrar el vértice v_i (al que se calcula su grado) dentro de la lista de adyacencias en el grafo original se requieren $O(d^2x)$ unidades de tiempo (donde x es la cantidad de bits necesarios para representar a cada vértice), siendo este procedimiento el más costoso del algoritmo. Por lo tanto, para calcular el grado de un vértice se requieren $O(d^2x)$ unidades de tiempo.

El algoritmo requiere $O(x)$ unidades de espacio, dado que es a lo más ésta la cantidad de bits que se copian a la cinta de trabajo para realizar la suma.

II.2. Cómputo Reconfigurable

II.2.1. Introducción a la rejilla reconfigurable

Las arquitecturas reconfigurables consisten de un conjunto de elementos computacionales, los cuales se encuentran conectados por medio de un ducto capaz de reconfigurarse y que permite agilizar el proceso de comunicación entre los procesadores.

Una arquitectura reconfigurable es la que puede alterar las funcionalidades de sus componentes y la estructura de conexión de los mismos. Cuando la reconfiguración es rápida, con un sobre costo pequeño, se dice que es dinámica. Consecuentemente una arquitectura de configuración dinámica puede cambiar su estructura y funcionalidad en cada ciclo de máquina.

Uno de los modelos más estudiados dentro de las arquitecturas reconfigurables es la llamada rejilla reconfigurable (R-Mesh). Su gran versatilidad permite el diseño de algoritmos de forma sencilla, pero su implementación es difícil en la vida real. En Vaidyanathan y Trahan (2004) se define el arreglo lineal de procesadores. Cada procesador posee dos puertos denominados Este (E) y Oeste (O) que permiten la conexión con sus vecinos. Además, cada procesador es capaz de fusionar sus dos puertos formando un ducto que atraviesa al mismo procesador conectando a sus vecinos. A este arreglo también se le denomina rejilla reconfigurable de una dimensión.

La rejilla de dos dimensiones llamada R-Mesh de $N \times M$, se construye al cambiar el arreglo lineal de procesadores por una rejilla bidimensional con N renglones y M columnas. En la rejilla de dos dimensiones, cada procesador posee cuatro puertos, denominados Norte (N), Sur (S), Este (E) y Oeste (O) que conectan al procesador con sus respectivos vecinos, si es que existen. De igual forma, al modelo de una dimensión, cada procesador es capaz de crear internamente conexiones entre sus puertos, que aunadas a

las conexiones fijas (entre procesadores) se crean diferentes arreglos de ductos entre la estructura de procesadores.

En el modelo R-Mesh cada puerto puede generar 15 configuraciones distintas como se muestran en la Figura 7. La notación utilizada para representar las conexiones es la siguiente: dentro de un juego de llaves se colocan las iniciales de los cuatro puertos separados en grupos por comas. Si un grupo tiene 2 o más puertos indica que ellos están conectados entre sí.

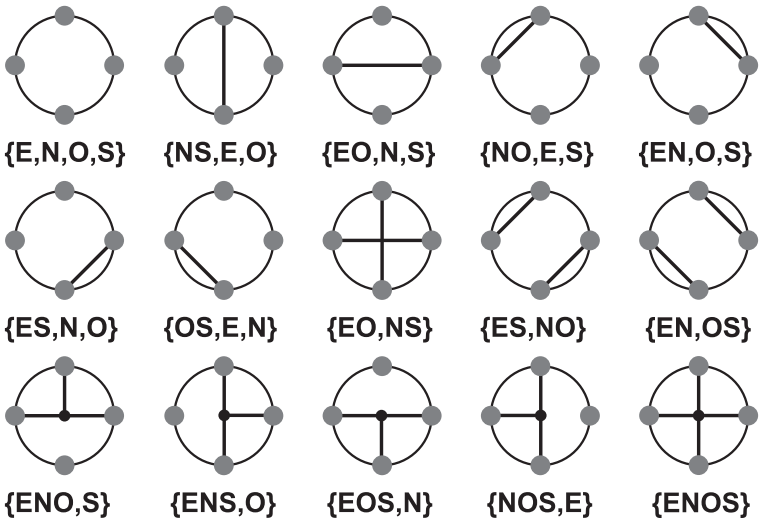

Figura 7. Posibles configuraciones internas para cada procesador de un R-Mesh

Cada procesador en el R-Mesh tiene su propia memoria local. Los procesadores funcionan en forma sincronizada y se comunican entre ellos a través del sistema de ductos descrito anteriormente.

A continuación se describen las 4 fases que comprenden un ciclo de máquina del R-Mesh:

II.2.2. Ciclo de máquina en el R-Mesh

Cada ciclo de máquina de un R-Mesh se divide en cuatro fases:

1. **Configuración:** Cada procesador internamente configura una de las 15 posibles conexiones de sus puertos como se muestran en la Figura 7.

2. **Escritura:** Cada procesador escribe en uno o más de sus puertos.

3. **Lectura:** Cada procesador lee uno o más de sus puertos.

4. **Operación:** Cada procesador ejecuta alguna operación aritmética o lógica.

En un paso arbitrario, un procesador del R-Mesh puede participar en uno o más de los pasos anteriores, o permanecer ocioso.

II.2.3. Modelo LR-Mesh

Existe una versión restringida (en la cantidad de configuraciones internas) del modelo R-Mesh, llamada rejilla reconfigurable lineal (LR-Mesh). Este modelo restringido es más factible de construirse que el modelo R-Mesh, aunque el diseño de algoritmos en él es más complicado. La definición dada en Vaidyanathan y Trahan (2004) del modelo LR-Mesh indica que este modelo sólo permite que cada puerto de cada procesador se pueda conectar a lo más a otro puerto dentro del mismo procesador. Esto implica que un ducto no se puede bifurcar como en el caso del R-Mesh donde cada procesador puede fusionar 3 o 4 de sus puertos. Esta restricción en el modelo reduce la cantidad de conexiones internas a 10, las cuales se muestran en la Figura 8. Los ductos creados en el modelo LR-Mesh se denominan ductos lineales.

A continuación se presentan algunos algoritmos básicos diseñados en el modelo LR-Mesh.

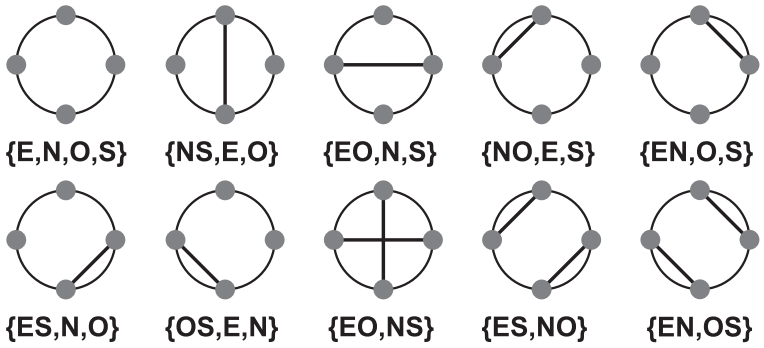

Figura 8. Posibles configuraciones internas para cada procesador de un LR-Mesh

II.2.4. Algoritmos básicos en el LR-Mesh

1. Movimiento de datos

   El movimiento de datos es importante en el diseño de algoritmos en el LR-Mesh. A continuación se presentan como ejemplos, dos de los problemas de movimiento de datos más comunes: broadcast y ruteo de permutación.

   Broadcast: El término *Broadcast* en el LR-Mesh significa enviar la misma información a todos los procesadores que se encuentren conectados por el ducto. La ventaja principal del broadcast en el LR-Mesh es que el retardo en el ducto es constante (la transmisión de la información en el ducto es constante, sin importar el tamaño del LR-Mesh). Para realizar un broadcast entre columnas en el LR-Mesh se ejecuta el siguiente ciclo de máquina: cada procesador realiza la siguiente configuración $\{NS, E, O\}$. El procesador que requiere realizar el broadcast escribe el dato a transmitir en su puerto Norte o Sur. El resto de los procesadores leen su puerto Norte o Sur, finalmente cada procesador almacena el dato leído en su registro correspondiente. De forma similar para realizar el broadcast entre renglones, cada procesador realiza la siguiente configuración $\{EO, N, S\}$. El procesador que requiere realizar el broadcast escribe el dato a transmitir en su puerto Este

u Oeste, todos los procesadores leen su puerto Este u Oeste. Por último, cada procesador almacena el dato leído en su registro correspondiente. Para realizar un broadcast en diagonal en el LR-Mesh el procedimiento es similar a los anteriores.

Ruteo de Permutación: En Vaidyanathan y Trahan (2004) se define el *Ruteo de Permutación* de la siguiente forma: Sea $P = \{p_0, p_1, \ldots, p_{N-1}\}$ un conjunto de procesadores donde cada procesador p_i (para $0 \leq i \leq N-1$) mantiene un dato d_i. y sea $\pi : \{0, 1, \ldots, N-1\} \longrightarrow \{0, 1, \ldots, N-1\}$ una biyección. El *Ruteo de Permutación* con respecto a π requiere que cada procesador p_i mande su dato d_i al procesador $p_{\pi(i)}$.

Para realizar un ejemplo de ruteo de permutación se supone que se tiene una serie de datos almacenados en la primera columna del modelo LR-Mesh y se desea mover los datos al primer renglón del LR-Mesh. Para realizar lo anterior se ejecutan los siguientes pasos:

Permutación de columna a renglón en el LR-Mesh

a) **Configuración:** Cada procesador con índice $i = j$ (los procesadores de la diagonal principal), realiza la siguiente configuración $\{NO, E, S\}$. Cada procesador con $j > i$ (los procesadores arriba de la diagonal principal) realiza la siguiente configuración $\{NS, E, O\}$. Finalmente, cada procesador con $j < i$ (los procesadores abajo de la diagonal principal) realiza la siguiente configuración $\{EO, N, S\}$.

b) **Escritura:** Cada procesador con índice $(i, 0)$ escribe en su puerto O el dato almacenado.

c) **Lectura:** Cada procesador con índice $(0, j)$ lee en su puerto S el dato transmitido.

d) **Operación:** Cada procesador con índice $(0, j)$ almacena el dato leído en su registro correspondiente.

El ruteo de permutación se puede ejecutar en el LR-Mesh en $O(1)$ unidades de tiempo.

2. Operación OR utilizando la partición de ducto

La operación **OR** de x bits se puede ejecutar en el LR-Mesh de X procesadores en $O(1)$ unidades de tiempo, utilizando la técnica de partición de ducto (Bus Splitting) propuesta por Miller *et al.* (1993).

Se supone que cada procesador del LR-Mesh almacena un bit del dato binario de entrada.

a) **Configuración utilizando la operación partición de ducto:** Cada procesador que almacena un bit "1" se configura de la siguiente manera $\{E, N, O, S\}$. Cada procesador que almacena un bit "0" realiza la siguiente configuración $\{EO, N, S\}$. Con esta configuración se lleva a cabo la separación de los ductos en sub-ductos.

b) **Escritura:** Cada procesador que almacena un bit "1" escribe una señal predefinida en su puerto O.

c) **Lectura:** El procesador con índice 0 (el procesador más a la izquierda) lee de su puerto E.

d) **Operación:** Si el procesador con índice 0 recibe la señal predefinida, el resultado de la operación **OR** es 1.

3. Suma binaria

   La **Suma binaria** de x bits se puede ejecutar en un LR-Mesh de $(x + 1 \times x)$ procesadores en $O(1)$ unidades de tiempo.

   Se supone que cada procesador del primer renglón del LR-Mesh $(0, j)$ almacena un bit del dato binario. Para realizar la suma binaria se ejecutan las siguientes fases:

   a) **Cada procesador** $(0, j)$ **ejecuta un broadcast por columna**

   b) **Identificar el índice** i

      1) **Configuración:** Cada procesador que almacena un bit "1" realiza la siguiente configuración $\{EN, OS\}$. Cada procesador que almacena un bit "0" realiza la siguiente configuración $\{EO, N, S\}$.

      2) **Escritura:** El procesador con índice $(0, 0)$ escribe una señal predefinida en su puerto O.

      3) **Lectura:** Cada procesador de la última columna (i, x) lee de su puerto E.

      4) **Operación:** Cada procesador de la última columna (i, x) almacena el valor leído en su variable *Dato*.

   c) **Enviar el índice** i

      1) **Configuración:** Cada procesador mantiene la misma configuración que en la fase anterior.

      2) **Escritura:** Cada procesador de la última columna (i, x) revisa si su variable *Dato* es igual a la señal predefinida; si son iguales, dicho procesador escribe su índice i en el puerto O.

3) **Lectura:** El procesador con índice $(0, 0)$ lee su puerto E o S.

4) **Operación:** El procesador con índice $(0, 0)$ almacena el valor leído en su variable *Total*.

En la Figura 9 se muestra el ejemplo de suma binaria de x bits en el LR-Mesh, para un $x = 4$

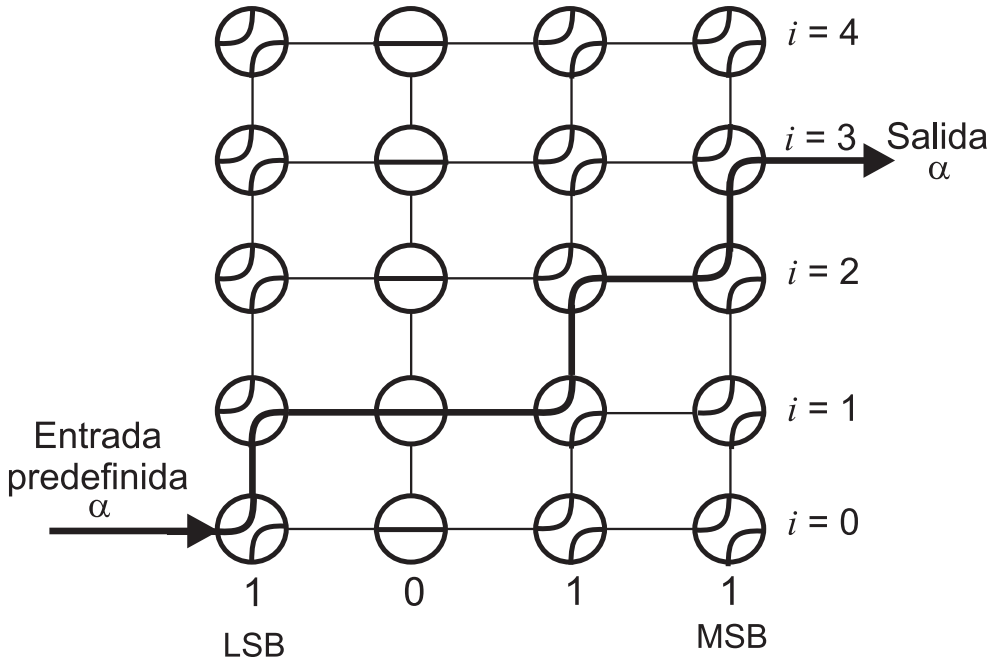

Figura 9. Ejemplo de suma binaria en un LR-Mesh

II.3. La máquina de Turing y el LR-Mesh

En esta sección se muestra brevemente la relación que existe entre la máquina de Turing de espacio logarítmico y el LR-Mesh. Para entender esta relación es necesario definir las clases de complejidad L, NL y SL.

II.3.1. Clases de complejidad

La clase L son todos los problemas de decisión que puede resolver una MTD con espacio logarítmico en función de su entrada.

La clase NL son todos los problemas de decisión que puede resolver una MTND con espacio logarítmico en función de la entrada.

Por último, Lewis y Papadimitriou (1982) definen la clase SL como todos los problemas de decisión que puede resolver una MTS con espacio logarítmico. Esta clase la definieron en función del problema USTCON, al demostrar que dicho problema es completo para la clase SL (este resultado implica que cualquier problema que se pueda reducir a USTCON pertenece a la clase SL). El problema USTCON se define de la siguiente forma: dado un grafo no dirigido $G = (V, E)$ y dos vértices $s, t \in G$, el *problema de conectividad no dirigido* $s - t$ (USTCON) es el proceso de determinar si existe una trayectoria (o camino) en G que conecte a "s" con "t". Antes del trabajo de Lewis y Papadimitriou, USTCON se clasificaba dentro de la clase NL. Finalmente, en el mismo trabajo, ellos demuestran que $L \subseteq SL \subseteq NL$.

II.3.2. Relación entre la MTD y el LR-Mesh

En Ben-Asher y Schuster (1995) se muestra la relación entre el espacio requerido por la MTD y el tiempo de ejecución del LR-Mesh, al demostrar que cualquier problema P que puede resolver el LR-Mesh en tiempo constante con un número polinomial de procesadores pertenece a la clase de complejidad L. Asimismo, ellos demuestran que cualquier problema que pertenece a la clase L lo puede resolver el LR-Mesh en tiempo constante con un número polinomial de procesadores.

Capítulo III

Algoritmo de Trifonov

Trifonov (2005) diseñó un algoritmo en la máquina de Turing para resolver el problema USTCON en $O(\log N \log\log N)$ espacio, el cual se basa en dos trabajos previos. El primero es el de Chong y Lam (1993), donde desarrollan un algoritmo que utiliza las técnicas de enganchamiento (Hooking) y contracción (Contraction) para resolver el problema USTCON. El segundo trabajo es el realizado por Koucký (2002); este trabajo explora árboles de una forma eficiente al aplicar la técnica de caminatas exploratorias; la técnica anterior se combina con la del tour de Euler para permitir un desplazamiento de un vértice a otro, de ida y de regreso, sin necesidad de guardar los vértices intermedios.

Con base en estos trabajos, Trifonov (2005) presenta un algoritmo recursivo para resolver el problema USTCON en un grafo no dirigido $G = (V, E)$, de N vértices, y sin multiaristas. Dicho algoritmo consiste principalmente de dos etapas, las cuales se describen a continuación:

1. **Etapa de enganchamiento**. En esta etapa el algoritmo realiza el enganchamiento; cada vértice revisa a sus vecinos para identificar un vértice (de acuerdo a un criterio dado) a enganchar (puede ser incluso él mismo); la arista que une a dos vértices enganchados se le llama *arista de enganche*. Este procedimiento genera árboles que se denominan *árboles de enganchamiento*.

2. **Etapa de contracción**. Esta etapa consiste en la contracción de los árboles generados por la etapa de enganchamiento; por medio de dos umbrales dados (los cuales cambian según el nivel de recursión del algoritmo), se seleccionan los

árboles a contraer si la suma de los vértices y de las aristas que componen el árbol es menor que dichos umbrales. Con esta operación, la raíz del árbol se denomina como el representante de todos los vértices que componen el árbol. Las multiaristas que unen a un árbol T con otro T' se eliminan y se conserva solo una. También se eliminan todas las aristas internas del árbol T; con esto se reduce el árbol a un solo super-vértice, el cual se convierte en el representante de todos los vértices internos del árbol T; todos estos vértices internos dejan de participar en el algoritmo.

Se ejecuta una de las dos etapas en cada nivel de recursión, donde después de un enganchamiento siempre se ejecuta una contracción; sin embargo, se pueden ejecutar varias contracciones consecutivas. El algoritmo de Trifonov (2005) asegura que en $O(\log N)$ etapas, todos los componentes conectados se reducen a un vértice representativo.

Para determinar si dos nodos s y t pertenecen al mismo componente conectado, el algoritmo revisa si el representante de s es igual al representante de t; si es así, entonces s y t están conectados.

Para entender mejor el algoritmo de Trifonov (2005) se proporciona una descripción breve del algoritmo de Chong y Lam (1993).

III.1. Algoritmo de Chong y Lam

En Chong y Lam (1993) se presenta un algoritmo recursivo que resuelve el problema USTCON. Donde G es un grafo no dirigido con N vértices y M aristas; Chong y Lam resuelven el problema USTCON en $O(\log N \log\log N)$ unidades de tiempo en una PRAM EREW, utilizando $O(N + M)$ procesadores. El algoritmo se basa en la estrategia de enganchamiento y contracción; sin embargo, utiliza un esquema de enganchamiento más sofisticado al que se utilizó en trabajos previos y un control de crecimiento de los árboles generados.

La forma trivial de resolver el problema USTCON utilizando la estrategia de enganchamiento y contracciones la siguiente:

1. Cada vértice se engancha a un vecino de su lista de adyacencias.

2. Cada componente que se forma en el paso 1 se contrae a un vértice representativo.

El problema con esta técnica es que cada componente formado en el paso 1 puede formar componentes desde 2 nodos hasta $O(N)$ nodos; por lo tanto, se requieren $O(\log N)$ unidades de tiempo, en el peor de los casos, para contraer todos los nodos a un solo representante. Si se requiere realizar $O(\log N)$ iteraciones para asegurar la elección del representante, entonces el algoritmo necesita $O(\log^2 N)$ unidades de tiempo para resolver el problema.

Chong y Lam proponen un nuevo esquema de enganchamiento que asegura que el número de elementos en la lista de adyacencias de cualquier componente formado en el paso de enganchamiento es a lo más el cuadrado del número de vértices en el componente. El paso de enganchamiento nunca genera componentes de pocos vértices con una lista de adyacencia larga. De esta forma se asegura que, si genera componentes

con s vértices, la contracción se realiza en $O(\log s)$ unidades de tiempo. Además, el algoritmo aprovecha la recursividad para diferenciar los componentes con respecto a su tasa de crecimiento y así permite que los componentes se enganchen y se contraigan en diferentes instantes de tiempo de acuerdo a su tamaño actual; con esto, se obtiene un tiempo total de $O(\log N \log\log N)$.

La diferencia de la etapa de enganchamiento de Trifonov (2005) con respecto a la de Chong y Lam (1993), es que en esta última todos los vértices revisan el grado de sus vecinos enganchándose al vértice con mayor grado; mientras que en el primero, se le da preferencia a aquellos vértices que no participaron en la fase de enganchamiento en niveles de recursión previos, llamados *vértices inactivos* (es decir, el vértice vc que se va a enganchar), primero busca a un vecino inactivo para engancharse, evitando así revisar el grado de todos sus vecinos; si no lo encuentra, realiza el mismo procedimiento que Chong y Lam.

Otra diferencia significativa del algoritmo de Trifonov (2005) en comparación con el de Chong y Lam (1993) es la utilización de las caminatas exploratorias definidas por Koucký (2002); estas caminatas permiten recorrer los árboles de enganchamiento para realizar diferentes tareas tales como: Comparar dos vértices bit a bit en $\Theta(\log\log N)$ unidades de espacio; revisar si el árbol T se puede contraer en un nivel de recursión dado.

Sin la utilización de las caminatas exploratorias, el algoritmo de Trifonov (2005) requeriría $O(\log N)$ unidades de espacio para realizar las comparaciones entre dos vértices; lo anterior obedece a que se necesitaría almacenar en una variable local $O(\log N)$ bits del vértice a comparar. Por otro lado, con la utilización de las caminatas exploratorias, el algoritmo de Trifonov requiere $O(\log\log N)$ unidades de espacio para realizar las comparaciones; esto es posible ya que únicamente se almacena el índice de cada bit

en una variable local de $O(\log \log N)$ unidades de espacio. Este procedimiento reduce el cuello de botella de $O(\log N)$ a $O(\log \log N)$ unidades de espacio.

Otro punto importante que realiza el algoritmo de Trifonov es que establece un límite superior $v(l)$ en el valor que pueden tomar las variables locales en el nivel de recursión l. El espacio que puede tomar cada variable local depende del nivel de recursión y varía entre $O(\log N)$ y $O(\log \log N)$. Por ejemplo, en la ejecución de todas las funciones en el nivel l, el valor de sus variables locales es a lo más $v(l)$. Así la cantidad de espacio que requiere el algoritmo es $O(\log v(l) + \log \log N)$ unidades de espacio, donde el término $\log \log N$ aparece por el contador usado en las comparaciones entre los vértices.

Para contraer cada componente conectado a un solo vértice representativo, el algoritmo de Trifonov requiere $O(\log N)$ niveles de recursión, y en cada nivel de recursión se requieren entre $O(\log N)$ y $O(\log \log N)$ unidades de espacio. Entonces la cantidad de espacio total requerido por el algoritmo es de $O(\log N \log \log N)$. De esta manera, determinar si dos vértices s y t están conectados se resuelve revisando si ellos tienen el mismo representante.

III.2. Pseudocódigo algoritmo de Trifonov

A continuación se presenta el pseudocódigo propuesto por Trifonov para resolver el problema USTCON. Debido a lo engorroso del diseño de algoritmos en la MTD, Tifonov diseña un pseudocódigo muy similar a la notación utilizada en el lenguaje Pascal, el cual posteriormente traduce a la MTD. En la siguiente sección se presentan algunos ejemplos de los algoritmos de Trifonov representados en la MTD.

III.2.1. Programa principal

Entrada: Un grafo no dirigido G y su lista de adyacencias. Los vértices s y t.

Salida: Cada componente conectado del grafo G se reduce a un vértice representante $R(v)$, donde todos los nodos que pertenecen al mismo componente conectado tienen el mismo representante. El programa regresa τ (True) si y sólo si el representante de s es igual al representante de t; y regresa ϕ (False) de otra forma.

Las dos rutinas principales presentadas por Trifonov son las siguientes:

global function: Connected

$level := 5 \cdot 2^{(loglogN)} - 3$;

$currvertex := arg1Connected;\ MoveToRep$;

$v := currvertex$;

$currvertex := arg2Connected;\ MoveToRep$;

return $v := currvertex$;

procedure MoveToRep

if level > 0 **then**

$Prev(MoveToRep)$;

if $Contraction$ **and** $TreeSize \neq null$ **then** $TreeForward(Root)$;

La función **Connected** manda llamar a la función **MoveToRep** dos veces, una para s y otra para t. Compara ambos representantes y regresa τ si y sólo si el representante

de s es igual al representante de t.

La función **MoveToRep** se manda llamar a sí misma en forma recursiva hasta llegar al nivel más bajo de recursión. Si la función **Contraction** es verdadera para el nivel de recursión actual, y además si el tamaño del árbol al que pertenece el vértice actual es lo suficientemente pequeño (con base en un límite establecido para el nivel de recursión actual), la función **MoveToRep** mueve el vértice actual a su representante.

III.3. Algoritmos utilizados por Trifonov representados en una MTD

En esta sección se presentan algunos de los algoritmos que utiliza Trifonov para resolver el problema USTCON en la MTD.

III.3.1. Problema: Determinar si $v_1 < v_2$ en la MTD

Descripción: Se tiene $\Sigma = \{0, 1, \tau, \phi, \sharp\}$, los enteros positivos se representan en binario. Se diseña un algoritmo en la MTD que revise si el vértice $v_1 < v_2$.

Solución en una MTD

Entrada: En la cinta de entrada se encuentra el identificador del vértice v_1 seguido del símbolo $\sharp$ y el identificador del vértice v_2.

Salida: Se escribe τ en la cinta de salida, si $v_1 < v_2$; se escribe ϕ, de cualquier otra forma.

A continuación se describe en alto nivel al algoritmo en la MTD

La MTD realiza los siguientes pasos:

1. Se copia el identificador del vértice v_1 a la cinta de trabajo.

2. Se posiciona el cabezal de la cinta de entrada en el bit más significativo del vértice v_2 y el cabezal de la cinta de trabajo en el bit más significativo del vértice v_1.

3. Se realiza la comparación bit a bit entre los dos vértices de la siguiente forma:

   a) Si el bit de ambos cabezales es igual, se recorre el cabezal una posición a la derecha en ambas cintas.

   b) Si el bit de ambos cabezales es diferente y el cabezal de la cinta de entrada se posiciona en un 1, el cabezal de salida escribe τ y la MTD pasa a un estado de aceptación. Si el cabezal de la cinta de trabajo se posiciona en un 1, el cabezal de salida escribe ϕ y la MTD pasa a un estado de rechazo.

   c) Si todos los bits de los vértices v_1 y v_2 son iguales, el cabezal de salida escribe ϕ y la MTD pasa a un estado de rechazo.

III.3.2. Problema: Determinar si $v_1 = v_2$ en la MTD

Descripción: Se tiene $\Sigma = \{0, 1, \tau, \phi, \sharp\}$, existen dos cintas de entrada, los enteros positivos se representan en binario. Se diseña un algoritmo para una MTD que revise si el vértice $v_1 = v_2$.

Solución en una MTD

Entrada: En la cinta de entrada 1 y 2 se encuentran los identificadores de los vértices v_1 y v_2, respectivamente.

Salida: Se escribe τ en la cinta de salida, si $v_1 = v_2$; se escribe ϕ, de cualquier otra forma.

A continuación se describe en alto nivel al algoritmo en la MTD

La MTD realiza los siguientes pasos:

1. Se posicionan los cabezales de las cintas de entrada 1 y 2 en el bit más significativo de los vértices v_1 y v_2, respectivamente.

2. Se realiza la comparación bit a bit entre los dos vértices de la siguiente forma:

   a) Si el bit de ambos cabezales es igual, se recorre el cabezal una posición a la derecha en ambas cintas.

   b) Si el bit de ambos cabezales es diferente, el cabezal de salida escribe ϕ y la MTD pasa a un estado de rechazo.

   c) Si todos los bits de los vértices v_1 y v_2 son iguales, el cabezal de salida escribe τ y la MTD pasa a un estado de aceptación.

III.3.3. Problema: Determinar si $v_1 <_d v_2$ en la MTD

La notación $v_1 <_d v_2$ significa que el grado del vértice v_1 es menor que el grado de v_2, o si son iguales en grado, la etiqueta de v_1 es menor que la de v_2.

Descripción: El alfabeto $\Sigma = \{0, 1, \tau, \phi, \sharp\}$. Los enteros positivos se representan en binario. Se diseña un algoritmo para una MTD que revise si el vértice $v_1 <_d v_2$.

Solución en una MTD

Entrada: En la cinta de entrada 1 y 2 se encuentran los grados de los vértices v_1 y v_2, respectivamente, representados en binario.

Salida: Se escribe τ en la cinta de salida, si $v_1 <_d v_2$; se escribe ϕ, de cualquier otra forma.

A continuación se describe en alto nivel al algoritmo para la MTD

La MTD realiza los siguientes pasos:

1. Se posicionan los cabezales de las cintas de entrada 1 y 2 en el bit más significativo del grado de los vértices v_1 y v_2, respectivamente.

2. Se realiza la comparación bit a bit entre los grados de los vértices de la siguiente forma:

   a) Si el bit de ambos cabezales es igual, se recorre el cabezal una posición a la derecha en ambas cintas.

   b) Si el bit de ambos cabezales es diferente y el cabezal de la cinta de entrada 1 se posiciona en un 1, el cabezal de salida escribe ϕ y la MTD pasa a un estado de rechazo. Si el cabezal de la cinta de entrada 2 se posiciona en un 1, el cabezal de salida escribe τ y la MTD pasa a un estado de aceptación.

   c) Si todos los bits del grado de los vértices v_1 y v_2 son iguales, la MTD pasa a un estado donde verifica si la etiqueta de v_1 es menor que la de v_2, utilizando un proceso similar al anterior.

III.3.4. Problema: Revisa si $v_c \in$ D, A o I en la MTD

Descripción: El alfabeto $\Sigma = \{0, 1, \tau, \phi, n, D, A, I, \sharp\}$. Los enteros positivos se representan en binario. Se diseña un algoritmo para una MTD que revise si el vértice $v_c \in D$, A o I (donde D, A e I significan los conjuntos de vértices terminados, activos e inactivos, respectivamente y representan también el estatus de los vértices durante la ejecución del algoritmo).

Solución en una MTD

Entrada: En la cinta de entrada número 1 se encuentra almacenado el valor numérico del nivel de recursión. En la cinta de entrada número 2 se encuentra almacenado el

grado del vértice v_c. La cinta de entrada número 3 tiene almacenada τ si la operación en el nivel actual es de contracción; tiene almacenada ϕ, si es de enganchamiento. En la cinta de entrada número 4 se encuentra almacenado el valor de $2(size(T) - 1)$ (donde $size(T)$ es la cantidad de vértices que pertenecen al árbol T) o n de cualquier otra forma. Todos los números se representan en binario.

Salida: Se escribe D en la cinta de salida si v_c es elemento del conjunto de vértices terminado; se escribe A, si v_c es elemento del conjunto de vértices activo; y se escribe I, si v_c es elemento del conjunto de vértices inactivos.

A continuación se muestra la descripción en alto nivel de los tres algoritmos en la MTD.

La MTD realiza los siguientes pasos para calcular si $v_c \in D$:

1. Si los cabezales de las cintas de entrada 1 y 2 leen el símbolo "0", entonces se escribe D en la cinta de salida y la MTD pasa a un estado de aceptación.

2. Si el cabezal de la cinta de entrada 1 lee un símbolo distinto de "0", si el cabezal de la cinta de entrada 3 lee el símbolo ϕ, y si v_c en el nivel anterior fue D, entonces se escribe D en la cinta de salida y la MTD pasa a un estado de aceptación.

3. Si el cabezal de la cinta de entrada 1 lee un símbolo distinto de "0", si el cabezal de la cinta de entrada 3 lee el símbolo τ, y si v_c en el nivel anterior fue D o no es raíz, entonces se escribe D en la cinta de salida y la MTD pasa a un estado de aceptación.

La MTD realiza los siguientes pasos para calcular si $v_c \in A$:

1. Si el cabezal de la cinta de entrada 1 lee el símbolo "0" y si el cabezal de la cinta de entrada 2 lee un símbolo distinto de "0", entonces se escribe A en la cinta de

salida y la MTD pasa a un estado de aceptación.

2. Si el cabezal de la cinta de entrada 1 lee un símbolo distinto de "0", si el cabezal de la cinta de entrada 3 lee el símbolo ϕ, y si v_c no perteneció al conjunto de vértices activos, entonces se escribe A en la cinta de salida y la MTD pasa a un estado de aceptación.

3. Si el cabezal de la cinta de entrada 1 lee un símbolo distinto de "0", si el cabezal de la cinta de entrada 3 lee el símbolo τ, si v_c no pertenece al conjunto de vértices terminados, y si el cabezal de la cinta 4 lee un símbolo distinto de n, entonces se escribe A en la cinta de salida y la MTD pasa a un estado de aceptación.

La MTD realiza los siguientes pasos para calcular si $v_c \in I$:

1. Si el cabezal de la cinta de entrada 1 lee un símbolo distinto de "0", si el cabezal de la cinta de entrada 3 lee el símbolo ϕ, y si v_c en el nivel anterior no perteneció al conjunto de vértices inactivos, entonces se escribe I en la cinta de salida y la MTD pasa a un estado de aceptación.

2. Si el cabezal de la cinta de entrada 1 lee un símbolo distinto de "0", si el cabezal de la cinta de entrada 3 lee el símbolo τ, si v_c no pertenece al conjunto de vértices terminados, y si el cabezal de la cinta 4 lee el símbolo n, entonces se escribe I en la cinta de salida y la MTD pasa a un estado de aceptación.

III.4. Simulación de una MTD en un LR-Mesh

En esta sección se presenta paso a paso la simulación de una MTD de espacio logarítmico en un LR-Mesh. A continuación se explica los pasos para realizar la simulación en el LR-Mesh utilizando los descriptores de la MTD.

En Ben-Asher y Schuster (1995) se define un *descriptor* de una MTD como la tupla $d = \langle q, [W], i, w \rangle$, donde $q \in Q$ es el estado actual, W es el contenido (total) de la cinta de trabajo, los parámetros i y w son los apuntadores de la posición de la cinta de entrada y de trabajo, respectivamente. Sea M una MTD con una entrada de tamaño N y con una longitud de la cinta de trabajo de $f(N)$, el número de descriptores válidos diferentes para M está limitado (para una constante $c \geq 1$) por: $\mid Q \mid \cdot N \cdot f(N) \cdot \mid \Sigma \mid^{f(n)} = O(N \cdot c^{f(N)})$, donde Σ es el alfabeto de la MTD.

Entonces, cada uno de estos descriptores se puede ver como un procesador del LR-Mesh y existe una conexión entre los procesadores i y j en el LR-Mesh si y sólo si existe una transición entre dos descriptores i y j en la MTD.

En la siguiente sección se presentan dos algoritmos resueltos en la MTD. Primero se generan las reglas que utiliza la MTD, después se obtienen los descriptores, y por último se genera el diagrama de descriptores, el cual se puede tomar como una representación de un LR-Mesh.

III.4.1. Problema: Determinar si existe la misma cantidad de ceros consecutivos que de unos en la MTD

Descripción: El alfabeto $\Sigma = \{0, 1, \sharp\}$. Los enteros positivos se representan en binario. Se diseña un algoritmo para una MTD que compare si en una cadena de bits existe la misma cantidad de ceros consecutivos que de unos.

Entrada: En la cinta de entrada está almacenada la secuencia de x bits, en la cinta de trabajo están almacenados x ceros, y el cabezal está posicionado en la celda más derecha de la cinta de trabajo.

Salida: Si el contenido de la cinta de trabajo es cero, entonces la cadena de bits

tiene el mismo número de ceros y unos consecutivos y la MTD pasa a un estado de aceptación; en caso contrario, la cadena de bits no tiene el mismo número de ceros y unos consecutivos y la MTD pasa a un estado de rechazo.

Estado inicial: q_0; estados finales: q_r y q_a (donde q_r es el estado de rechazo y q_a es el estado de aceptación); estados intermedios: q_1, q_2, q_3, q_4, q_5; alfabeto: $\Sigma = (0, 1, \sharp)$.

Se obtienen las reglas de transición para el algoritmo:

$q_0(0, x) \longrightarrow q_1((x, S)(x, S))$ // Primer cero de la cadena.

$q_0(1, x)) \longrightarrow q_r((x, S)(x, S))$ // Hay un uno al inicio de la cadena.

$q_1(\sharp, x) \longrightarrow q_r((x, S)(x, S))$ // Hay únicamente ceros en la cadena de entrada.

$q_1(0, 1)) \longrightarrow q_1((x, S)(0, I))$ // Inicio de aumento contador.

$q_1(0, 0) \longrightarrow q_3((x, S)(1, D))$ // Termina incremento contador.

$q_3(x, 0) \longrightarrow q_3((x, S)(x, D))$ // Se posiciona en el bit menos significativo el cabezal de la cinta de trabajo y el cabezal de la cinta de entrada.

$q_3(x, 1) \longrightarrow q_3((x, S)(x, D))$ // Avanza una posición a la derecha.

$q_3(x, \sharp) \longrightarrow q_1((x, D)(x, I))$

$q_1(1, x) \longrightarrow q_2((x, S)(x, S))$ // Inicia la recepción de unos.

$q_2(1, 0) \longrightarrow q_2((x, S)(1, I))$ // Inicia decremento de contador.

$q_2(1, \sharp) \longrightarrow q_r((x, S)(x, S))$ // Contador negativo (hay más unos que ceros).

$q_2(1, 1) \longrightarrow q_4((x, S)(0, D))$ // Termina decremento de contador.

$q_4(x, 0) \longrightarrow q_4((x, S)(x, D))$ // Se posiciona en el bit menos significativo el cabezal de la cinta de trabajo y el cabezal de la cinta de entrada.

$q_4(x, 1) \longrightarrow q_4((x, S)(x, D))$ // Avanza una posición a la derecha.

$q_4(x, \sharp) \longrightarrow q_2((x, D)(x, I))$

$q_2(0, x) \longrightarrow q_r((x, S)(x, S))$ // Existe un cero después de un uno.

$q_2(\sharp, x) \longrightarrow q_5((x, S)(x, S))$ // Termina de leer la secuencia de bits.

$q_5(x,1) \longrightarrow q_r((x,S)(x,S))$ // Contador diferente de cero (hay más ceros que unos).

$q_5(x,0) \longrightarrow q_5((x,S)(x,I))$ // Continua revisando si el contador es igual a cero.

$q_5(x,\sharp) \longrightarrow q_a((x,S)(x,S))$ // La cantidad de ceros es igual a la cantidad de unos.

Se obtienen los descriptores que se utilizan para determinar si en una cadena de bits existe la misma cantidad de ceros consecutivos que de unos. Debido a que la cantidad de descriptores se incrementa dependiendo de x, entonces por simplicidad el ejemplo se acota a 4 bits.

$^{0} < q_0, [0], 1, 1 >^{0}$ // Inicio de la comparación.

$^{1} < q_1, [0], 1, 1 >^{0}$ // Existe un cero en la entrada.

$^{2} < q_r, [0], 1, 1 >^{0}$ // Existe un uno en la entrada.

$^{3} < q_3, [1], 1, 0 >^{1}$ // Incrementar contador.

$^{4} < q_1, [1], 2, 1 >^{3}$ // Cabezal de trabajo se posiciona en la segunda posición.

$^{5} < q_1, [0], 2, 2 >^{4}$ // Existe un cero en la segunda posición.

$^{6} < q_3, [1], 2, 1 >^{5}$ // Se mueve el cabezal de trabajo.

$^{7} < q_3, [1], 2, 0 >^{6}$ // Se mueve el cabezal de trabajo.

$^{8} < q_1, [1], 3, 1 >^{7}$ // Se mueve el cabezal de trabajo.

$^{9} < q_2, [1], 2, 1 >^{4}$ // Cabezal de entrada se posiciona en un uno.

$^{10} < q_4, [0], 2, 0 >^{9}$ // Se decrementa en uno al contador.

$^{11} < q_2, [0], 3, 1 >^{10}$ // Cabezales se posicionan en la tercera posición.

$^{12} < q_3, [11], 3, 0 >^{8}$ // Se mueve el cabezal de entrada a la siguiente posición.

$^{13} < q_1, [11], 4, 1 >^{12}$ // Se mueve el cabezal de entrada a la siguiente posición.

$^{14} < q_2, [1], 3, 1 >^{8}$ // Cabezal de entrada se posiciona en un uno.

$^{15} < q_2, [11], 3, 2 >^{14}$ // Se mueve cabezal de entrada.

$^{16} < q_4, [1], 3, 1 >^{15}$ // Se decrementa en uno al contador.

$^{17} < q_4, [1], 3, 0 >^{16}$ //

$^{18} < q_2, [1], 4, 1 >^{17}$ // Se mueve el cabezal de entrada a la siguiente posición.

$^{19} < q_r, [0], 3, 1 >^{10}$ // Existe un cero después de un uno.

$^{20} < q_2, [1], 3, 2 >^{11}$ //

$^{21} < q_2, [11], 3, 3 >^{20}$ //

$^{22} < q_r, [11], 3, 3 >^{21}$ //

$^{23} < q_1, [10], 4, 2 >^{13}$ // Existe un cuarto cero en la posición cuatro.

$^{24} < q_1, [0], 4, 3 >^{23}$ //

$^{25} < q_r, [0], 4, 3 >^{24}$ // Contador mayor a $\log x$.

$^{26} < q_2, [11], 4, 1 >^{13}$ // Existe un uno en la cuarta posición.

$^{27} < q_4, [10], 4, 0 >^{26}$ //

$^{28} < q_2, [10], 5, 1 >^{27}$ //

$^{29} < q_5, [10], 5, 1 >^{28}$ //

$^{30} < q_5, [10], 5, 2 >^{29}$ //

$^{31} < q_r, [10], 5, 2 >^{30}$ // El contador es diferente de cero.

$^{32} < q_r, [1], 4, 1 >^{18}$ // Existe cero después de un uno.

$^{33} < q_4, [0], 4, 0 >^{18}$ // Existe un uno en la cuarta posición.

$^{34} < q_2, [0], 5, 1 >^{33}$ //

$^{35} < q_5, [0], 5, 1 >^{34}$ //

$^{36} < q_5, [0], 5, 2 >^{35}$ //

$^{37} < q_5, [0], 5, 3 >^{36}$ //

$^{38} < q_a A, [0], 5, 3 >^{37}$ // El contador es igual a cero.

En la Figura 10 se muestra el diagrama generado con los descriptores, el cual se puede simular en el LR-Mesh.

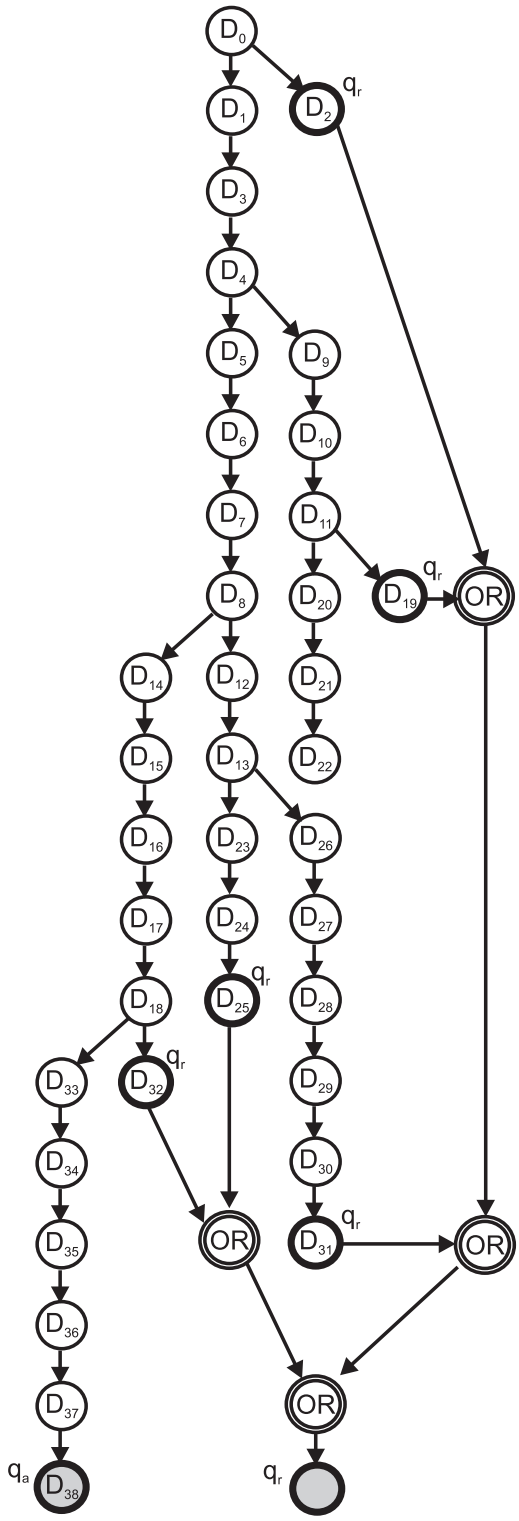

Figura 10. Diagrama de estados para el problema de determinar si existe la misma cantidad de ceros consecutivos que de unos en una MTD.

III.4.2. Problema: Determinar si $v_1 < v_2$ en la MTD

Descripción: Este problema se describe en la Sección III.3.1. Se diseña un algoritmo para una MTD que determine si $v_1 < v_2$.

Se obtienen las reglas de transición para el algoritmo:

$q_0(\sharp, x) \longrightarrow q_1((x, D)(x, S))$ // Se recorre al primer bit de v_1.

$q_1(0, x)) \longrightarrow q_1((\sharp, D)(0, D))$ // Copia v_1 de la cinta de entrada a la cinta de trabajo.

$q_1(1, x) \longrightarrow q_1((\sharp, D)(1, D))$ //

$q_1(\sharp, x)) \longrightarrow q_2((x, D)(x, I))$ // Termina de copiar v_1 y regresa el cabezal al MSB de v_1.

$q_2(x, 0) \longrightarrow q_2((x, S)(x, I))$ //

$q_2(x, 1) \longrightarrow q_2((x, S)(x, I))$ //

$q_2(x, \sharp) \longrightarrow q_3((x, S)(x, D))$ // Avanza una posición a la derecha y se posiciona en el MSB de v_1.

$q_3(0, 0) \longrightarrow q_3((x, D)(x, D))$ // El bit de v_1 y v_2 contiene el mismo valor.

$q_3(1, 1) \longrightarrow q_3((x, D)(x, D))$ // Inicia la recepción de unos.

$q_3(1, 0) \longrightarrow q_a((x, S)(x, S))$ // El bit de v_1 es cero y el bit de v_2 es uno por lo tanto $v_1 < v_2$.

$q_3(0, 1) \longrightarrow q_r((x, S)(x, S))$ // El bit de v_1 es uno y el bit de v_2 es cero por lo tanto $v_1 > v_2$.

$q_3(\sharp, \sharp) \longrightarrow q_r((x, S)(x, S))$ // Todos los bits son iguales por lo tanto $v_1 = v_2$.

Se obtienen los descriptores que se utilizan para determinar si $v_1 < v_2$. Debido a que la cantidad de descriptores se incrementa dependiendo de x, entonces por simplicidad, el ejemplo se acota para valores de v_1 y v_2 de 2 bits.

$^{0} < q_0, [0], 0, 0 >^{0}$ // Inicio.

$^{1} < q_1, [0], 1, 1 >^{0}$ //

$^{2} < q_1, [1], 2, 2 >^{1}$ // Existe un uno en la entrada.

$^{3} < q_1, [0], 2, 2 >^{1}$ // Existe un cero en la entrada.

$^{4} < q_1, [00], 3, 3 >^{2}$ //

$^{5} < q_1, [01], 3, 3 >^{2}$ //

$^{6} < q_1, [10], 3, 3 >^{3}$ //

$^{7} < q_1, [11], 3, 3 >^{3}$ //

$^{8} < q_2, [00], 4, 2 >^{4}$ // Regresa el cabezal de la cinta de trabajo a el MSB de v_1.

$^{9} < q_2, [00], 4, 1 >^{8}$ //

$^{10} < q_2, [00], 4, 0 >^{9}$ //

$^{11} < q_3, [00], 4, 1 >^{10}$ // Inicia a comparar v_1 y v_2.

$^{12} < q_2, [01], 4, 2 >^{5}$ // Regresa el cabezal de la cinta de trabajo a el MSB de v_1.

$^{13} < q_2, [01], 4, 1 >^{12}$ //

$^{14} < q_2, [01], 4, 0 >^{13}$ //

$^{15} < q_3, [01], 4, 1 >^{14}$ // Inicia a comparar v_1 y v_2.

$^{16} < q_2, [10], 4, 2 >^{6}$ // Regresa el cabezal de la cinta de trabajo a el MSB de v_1.

$^{17} < q_2, [10], 4, 1 >^{16}$ //

$^{18} < q_2, [10], 4, 0 >^{17}$ //

$^{19} < q_3, [10], 4, 1 >^{18}$ // Inicia a comparar v_1 y v_2.

$^{20} < q_2, [11], 4, 2 >^{7}$ // Regresa el cabezal de la cinta de trabajo a el MSB de v_1.

$^{21} < q_2, [11], 4, 1 >^{20}$ //

$^{22} < q_2, [11], 4, 0 >^{21}$ //

$^{23} < q_3, [11], 4, 1 >^{22}$ // Inicia a comparar v_1 y v_2.

$^{24} < q_3, [00], 5, 2 >^{11}$ // El MSB de v_1 es igual al MSB v_2.

$^{25} < q_a, [00], 4, 1 >^{11}$ // El MSB de v_1 es cero y de v_2 es uno por lo tanto $v_1 < v_2$.

$^{26} < q_3, [00], 6, 3 >^{24}$ // El LSB de v_1 es igual al MSB v_2.

$^{27} < q_a, [00], 5, 2 >^{24}$ //

$^{28} < q_r, [00], 6, 3 >^{26}$ // $v_1 = v_2$.

$^{29} < q_3, [01], 5, 2 >^{15}$ // El MSB de v_1 es igual al MSB v_2.

$^{30} < q_a, [01], 4, 1 >^{15}$ // $v_1 < v_2$.

$^{31} < q_3, [01], 6, 3 >^{29}$ // El LSB de v_1 es igual al MSB v_2.

$^{32} < q_r, [01], 6, 3 >^{31}$ // $v_1 = v_2$.

$^{33} < q_r, [01], 5, 2 >^{31}$ // $v_1 > v_2$.

$^{34} < q_3, [10], 5, 2 >^{19}$ // El MSB de v_1 es igual al MSB v_2.

$^{35} < q_r, [10], 4, 1 >^{19}$ // $v_1 > v_2$.

$^{36} < q_3, [10], 6, 3 >^{34}$ // El LSB de v_1 es igual al MSB v_2.

$^{37} < q_r, [10], 6, 3 >^{36}$ // $v_1 = v_2$.

$^{38} < q_a, [10], 5, 2 >^{34}$ // $v_1 < v_2$.

$^{39} < q_3, [11], 5, 2 >^{23}$ // El MSB de v_1 es igual al MSB v_2.

$^{40} < q_r, [11], 4, 1 >^{23}$ // $v_1 > v_2$.

$^{41} < q_3, [11], 6, 3 >^{39}$ // El LSB de v_1 es igual al MSB v_2.

$^{42} < q_r, [11], 6, 3 >^{41}$ // $v_1 = v_2$.

$^{43} < q_r, [11], 5, 2 >^{31}$ // $v_1 > v_2$.

En la Figura 11 se muestra el diagrama generado con los descriptores, el cual se puede simular en el LR-Mesh.

III.4.3. Problema: Determinar si $v_1 < v_2$ en la MTD utilizando $O(x)$ compuertas OR.

Se obtienen las reglas de transición para el algoritmo:

$q_0(\sharp, \sharp) \longrightarrow q_1((x, D)(x, D))$ // Se recorre al LSB de v_1 y v_2.

$q_1(0, 0)) \longrightarrow q_1((x, D)(x, D))$ // inicia la resta de v_1–v_2..

$q_1(0, 1) \longrightarrow q_2((x, D)(x, D))$

$q_1(1, 0)) \longrightarrow q_1((x, D)(x, D))$

$q_1(1, 1) \longrightarrow q_1((x, D)(x, D))$

$q_2(0, 0)) \longrightarrow q_2((x, D)(x, D))$ // inicia la resta de v_1–v_2.

$q_2(0, 1) \longrightarrow q_2((x, D)(x, D))$

$q_2(1, 0)) \longrightarrow q_1((x, D)(x, D))$

$q_2(1, 1) \longrightarrow q_2((x, D)(x, D))$

$q_1(\sharp, \sharp) \longrightarrow q_r((x, S)(x, S))$ // $v_1 \geq v_2$.

$q_2(\sharp, \sharp) \longrightarrow q_a((x, S)(x, S))$ // $v_1 < v_2$.

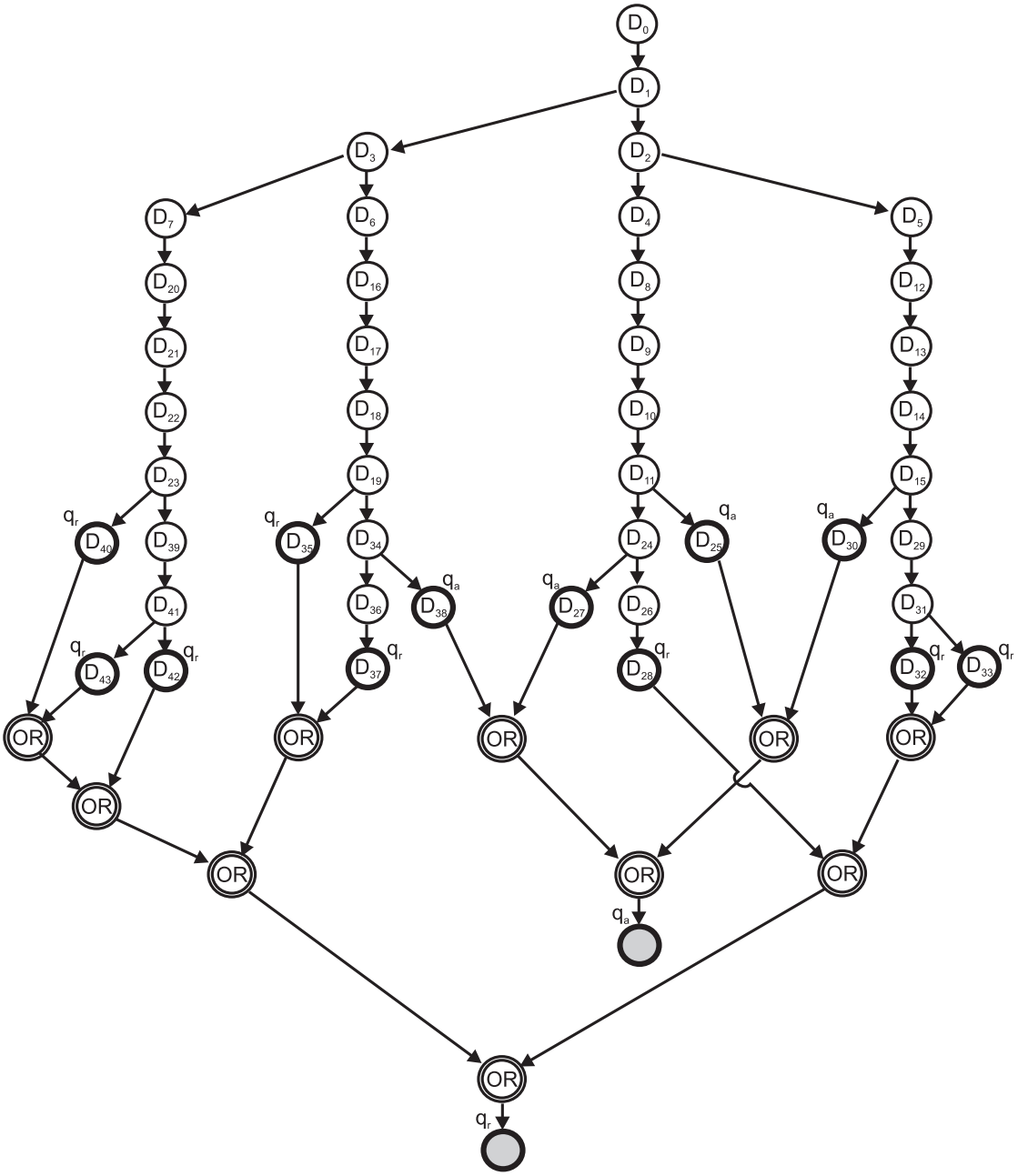

Figura 11. Diagrama de estados para el problema que determina si $v_1 < v_2$ en la MTD.

Se obtienen los descriptores que se utilizan para determinar si $v_1 < v_2$. Debido a que la cantidad de descriptores se incrementa dependiendo de x, entonces por simplicidad, el ejemplo se acota para valores de v_1 y v_2 de 4 bits.

$^{0} < q_0, [], 0, 0 >^{0}$ // Inicio.

$^{1} < q_1, [], 1, 1 >^{0}$ // Se inicia la comparacion de cada bit.

$^{2} < q_1, [], 2, 2 >^{1}$ //

$^{3} < q_2, [], 2, 2 >^{1}$ //

$^{4} < q_1, [], 3, 3 >^{2o3}$ //

$^{5} < q_2, [], 3, 3 >^{2o3}$ //

$^{6} < q_1, [], 4, 4 >^{4o5}$ //

$^{7} < q_2, [], 4, 4 >^{4o5}$ //

$^{8} < q_1, [], 5, 5 >^{6o7}$ //

$^{9} < q_2, [], 5, 5 >^{6o7}$ //

$^{10} < q_r, [], 5, 5 >^{8}$ // $v_1 > v_2$.

$^{11} < q_a, [], 5, 5 >^{9}$ // $v_1 < v_2$.

En la Figura 12 se muestra el diagrama generado con los descriptores, el cual se puede simular en el LR-Mesh.

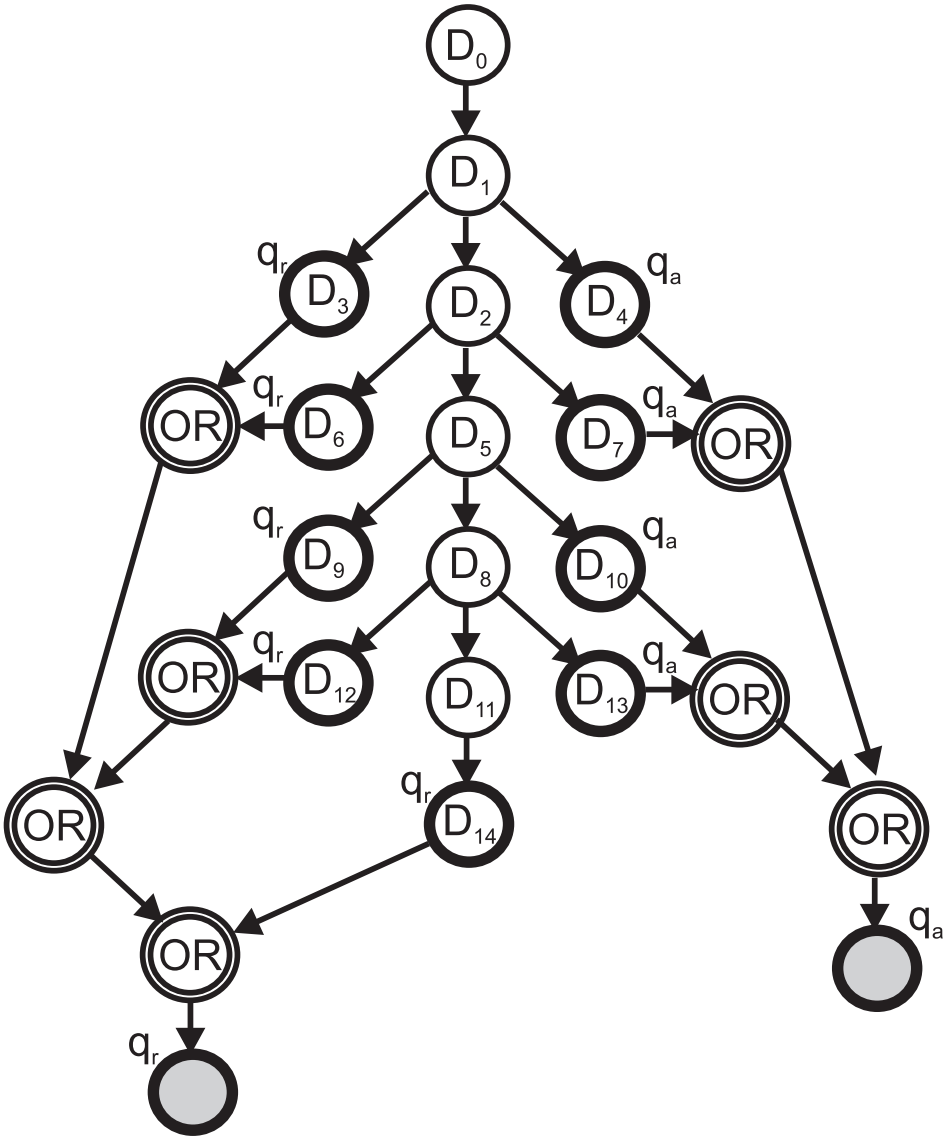

Figura 12. Diagrama de estados para el problema que determina si $v_1 < v_2$ utilizando $O(\log x)$ compuertas OR en la MTD.

Capítulo IV

Simulación eficiente en procesadores de un R-Mesh en un LR-Mesh

En el presente capítulo se muestra una simulación eficiente en cuanto al número de procesadores de un R-Mesh en un LR-Mesh. Primero se muestra el algoritmo que empotra una configuración arbitraria de un R-Mesh en un LR-Mesh. Posteriormente se presentan los algoritmos que se utilizan para resolver el problema USTCON en un LR-Mesh.

IV.1. Mapeo de un R-Mesh a un LR-Mesh

Este algoritmo empotra una configuración arbitraria de un R-Mesh en un LR-Mesh

Descripción: Mapeo de un R-Mesh de $N \times N$ a un LR-Mesh de $M \times M$.

Observaciones: Los procesadores en el LR-Mesh se identifican por sus índices (*renglón*, *columna*), denotado como (i, j).

Pseudo código *Mapeo_de_un_R-Mesh_a_un_LR-Mesh*

Entrada: Una configuración arbitraria de un R-Mesh de $N \times N$.

Salida: La configuración del R-Mesh de $N \times N$, empotrada en el LR-Mesh de $M \times M$

Se supone que cada procesador en el R-Mesh tiene cuatro puertos, N, S, E, O. La configuración interna se puede ver como una partición del conjunto de puertos; cada puerto cuenta con su lista de adyacencias, la cual indica su configuración. Se utilizan las variables α, β, γ y δ para almacenar la lista de adyacencias y el dato que se pretende escribir en el puerto N, O, S y E, respectivamente.

Inicialmente se mapea el procesador (i, j) del R-Mesh donde $0 \leq i, j < N - 1$ al procesador $(4i, 4j)$ del LR-Mesh

El LR-Mesh realiza los siguientes pasos:

1. Se transmite β.

   a) Cada procesador con índice $j \bmod 4 = 0$, realiza en sus puertos la configuración $\{E, N, O, S\}$. El resto de los procesadores realizan la configuración $\{EO, N, S\}$ para toda $0 \leq i, j \leq 4N - 1$.

   b) Cada procesador con índice $j \bmod 4 = 0$ escribe el dato almacenado en β a través de su puerto E.

   c) Cada procesador con índice $j \bmod 4 = 1$ lee su puerto O.

   d) Cada procesador con índice $j \bmod 4 = 1$ almacena la información leída en su registro correspondiente.

2. Se transmite γ.

   a) Cada procesador con índice $j \bmod 4 = 0$, realiza en sus puertos la configuración $\{E, N, O, S\}$. El resto de los procesadores realizan la configuración $\{EO, N, S\}$ para toda $0 \leq i, j \leq 4N$.

   b) Cada procesador con índice $j \bmod 4 = 0$ escribe el dato almacenado en γ a través de su puerto E.

   c) Cada procesador con índice $j \bmod 4 = 2$ lee su puerto O.

   d) Cada procesador con índice $j \bmod 4 = 2$ almacena la información leída en su registro correspondiente.

3. Se transmite δ.

a) Cada procesador con índice $j \bmod 4 = 0$, realiza en sus puertos la configuración $\{E, N, O, S\}$. El resto de los procesadores realizan la configuración $\{EO, N, S\}$ para toda $0 \leq i, j \leq 4N$.

b) Cada procesador con índice $j \bmod 4 = 0$ escribe el dato almacenado en δ por su puerto E.

c) Cada procesador con índice $j \bmod 4 = 3$ lee su puerto O.

d) Cada procesador con índice $j \bmod 4 = 3$ almacena la información leída en su registro correspondiente.

4. Mover los datos de renglón a columna.

   a) Los procesadores con índice $i \bmod 4N = j$, realizan la configuración $\{NO, E, S\}$. Cada procesador con índice $j > i \bmod 4N$ realiza la configuración $\{NS, E, O\}$. Cada procesador con índice $j < i \bmod 4N$ realiza la configuración $\{EO, N, S\}$.

   b) Cada procesador con índice $i \bmod 4N = 0$ escribe su dato a transmitir en su puerto S.

   c) Cada procesador con índice $j = 0$ lee en su puerto E.

   d) Cada procesador con índice $j = 0$ almacena la información leída en su registro correspondiente.

A continuación se presentan los algoritmos que se utilizan para resolver el problema USTCON en un LR-Mesh.

Trifonov (2005) presenta un algoritmo que se compone principalmente de dos fases (enganchamiento y contracción) con el cual resuelve el problema USTCON. Con base en ese trabajo se diseñaron los siguientes algoritmos en el LR-Mesh para cada fase.

Los algoritmos que se diseñaron en el LR-Mesh para la fase de enganchamiento son:

1. Enganchamiento a vecinos inactivos.
2. Obtener el grado de los vecinos de cada vértice en G.
3. Mapeo del grafo G al pseudo-bosque G'.
4. Construcción del tour De Euler.
    a) Preparación del LR-Mesh.
    b) Conexiones de cada pseudo-árbol en G'.
    c) Realizar broadcast en cada pseudo-árbol en G'.

Los algoritmos que se diseñan en el LR-Mesh para la fase de contracción son:

1. Ordenar los vértices en G' por componente.
2. Calcular el número de vértices en G' por componente.
3. Determinar las aristas externas de cada árbol

A continuación se explica a detalle los algoritmos para la fase de enganchamiento y posteriormente se hace para la de contracción.

En la fase de enganchamiento, cada vértice busca un vecino inactivo para engancharse a él; si no existe dicho vecino, entonces el vértice busca al vecino de mayor grado en su vecindario propio (la búsqueda lo incluye a él), y se engancha a dicho vecino (que puede ser incluso él).

IV.2. Enganchamiento a vecinos inactivos

Descripción: Cada vértice con estatus inactivos envía su estatus a sus vecinos.

Pseudo código *Enganchamiento_a_vecinos_inactivos*

1. Cada procesador de la primera columna realiza lo siguiente:

   Cada vértice i en G con estatus inactivo, envía su estatus a todos sus vecinos. Para cada vecino j, el procesador $(i, 0)$ transmite su estatus por medio del procesador (i, j).

2. El procesador (i, j) envía la información al procesador (j, j).

3. El procesador (j, j) envía la información al procesador $(j, 0)$.

4. Si el estatus del procesador $(j, 0)$ es activo, entonces marca como su arista de enganche a la arista que lo conecta con su vecino i (procesador $(i, 0)$); esto es, se engancha al vértice i. Por otro lado, si el estatus del procesador $(j, 0)$ es inactivo, ignora el mensaje.

En el paso 2 del algoritmo anterior puede existir una colisión; la colisión se puede presentar cuando dos o más procesadores envían al mismo tiempo la información al procesador (j, j) (se tienen dos o más vecinos inactivos); la colisión se resuelve utilizando la técnica partición de ducto (ver Sección II.2.4) propuesta por Miller *et al.* (1993). Con esta técnica únicamente un procesador escribe en cada ducto.

Por lo tanto, el algoritmo que ejecuta el enganchamiento a vecinos inactivos en el grafo G en el LR-Mesh de $4N^2 \times 4N^2$, tiene un tiempo de ejecución de $O(1)$ unidades de tiempo.

Como se mencionó anteriormente, cada vértice con al menos un vecino inactivo puede determinar a qué vértice se engancha en G' sin necesidad de revisar el grado

de sus vecinos. Donde G' es un pseudo-bosque que se genera después de la etapa de enganchamiento.

IV.3. Obtener el grado de los vecinos de cada vértice activo en G

Descripción: A continuación se explica de forma general el algoritmo para obtener el grado de todos los vecinos de cada vértice activo en G, posteriormente se explica dicho algoritmo con más detalle.

1. Cada procesador de la primera columna que representa a un vértice activo realiza lo siguiente:

   Cada vértice activo i en G desea saber el grado de su vecino j, así que el procesador $(i, 0)$ transmite el valor j al procesador (i, j).

2. El procesador (i, j) realiza la petición al procesador (j, j).

3. El procesador (j, j) realiza la petición al procesador $(j, 0)$.

4. El procesador $(j, 0)$ contesta la petición del procesador $(i, 0)$ utilizando la misma trayectoria pero en sentido opuesto.

En el paso 2 también puede existir una colisión, la cual se resuelve de la misma manera que en la Sección IV.2. De esta forma únicamente un procesador realiza la petición; los otros tres procesadores al no recibir la respuesta de su petición, lo intentan nuevamente, y el procesador que tuvo éxito realiza su siguiente petición. En el peor de los casos un procesador debe esperar a la cuarta petición para recibir una respuesta y al

tener a lo más cuatro vecinos, el algoritmo asegura en 16 fases que todos los procesadores del LR-Mesh obtienen el grado de todos sus vecinos en $O(1)$ unidades de tiempo.

Por lo tanto, el algoritmo que obtiene el grado de todos sus vecinos para todos los vértices de G, se ejecuta en un LR-Mesh de $4N^2 \times 4N^2$ con $O(1)$ unidades de tiempo.

A continuación se explica en forma detallada los pasos que ejecuta el LR-Mesh para obtener el grado de los vecinos de cada vértice en G.

Observaciones: Los procesadores en el LR-Mesh se identifican por sus índices (i, j). Donde $0 \leq i \leq N-1$ es el procesador que desea realizar la petición y $0 \leq j \leq N-1$ es el índice del procesador que recibe la petición.

Pseudo código: *Determina_grado*

Entrada: La lista de peticiones de cada vértice activo en G en el LR-Mesh de $4N^2 \times 4N^2$. Donde cada petición es la solicitud del grado a uno de sus vecinos.

Salida: Cada vértice activo en G obtiene el grado de todos sus vecinos y lo almacena en sus registros correspondientes en el LR-Mesh de $4N^2 \times 4N^2$.

El LR-Mesh realiza los siguientes pasos hasta que no existan peticiones:

1. Transmitir la petición de $(i, 0)$ a (i, j).

   a) Cada procesador realiza la configuración $\{EO, N, S\}$.

   b) Cada procesador con índice $(i, 0)$ escribe la petición en su puerto E.

   c) Cada procesador con índice (i, j) lee su puerto O.

   d) Cada procesador con índice (i, j) almacena la información leída en su registro correspondiente.

2. Transmitir la petición de (i, j) a (j, j), resolviendo el problema de colisión utilizando la técnica partición de ducto de Miller *et al.* (1993).

a) Cada procesador (i, j) que no realiza petición efectúa la configuración $\{NS, E, O\}$

Cada procesador (i, j) que desea realizar una petición efectúa la configuración $\{E, N, O, S\}$

b) Cada procesador con índice $i < j$ escribe la petición en su puerto S.

Cada procesador con índice $i > j$ escribe la petición en su puerto N.

c) Cada procesador (j, j) lee los puertos N y S.

d) Cada procesador (j, j) recibe un dato por el puerto N y almacena el valor obtenido en su registro correspondiente. Si el procesador (j, j) no recibe un dato por el puerto N, pero sí lo recibe por el puerto S, entonces almacena el valor obtenido en su registro correspondiente.

3. El procesador (j, j) envía el valor obtenido al procesador $(j, 0)$

a) Cada procesador realiza la configuración $\{EO, N, S\}$.

b) El procesador (j, j) escribe la petición en su puerto O.

c) El procesador $(j, 0)$ lee su puerto E.

d) El procesador $(j, 0)$ almacena la petición en su registro correspondiente.

El diseño del algoritmo que envía el grado del vértice j al vértice i es similar al algoritmo descrito anteriormente, se utiliza la misma trayectoria pero en sentido opuesto; sin embargo, para este algoritmo no se presenta el problema de colisión dado que cada procesador responde la petición a un único procesador.

IV.4. Mapeo del grafo G al pseudo-bosque G'

Descripción: Para obtener el vértice al cual se engancha cada vértice del grafo G y generar el pseudo-bosque G', se realizan los siguientes pasos:

Pseudo código: *Enganchamiento_de_cada_vértice_de_G*

Entrada: Cada vértice activo del grafo G, posicionado en la primera columna de un LR-Mesh de $4N^2 \times 4N^2$, conoce su grado y el de sus vecinos en G.

Salida: Para cada vértice activo de G, se determina a qué vértice se engancha en G'. Donde G' en un pseudo-bosque y todos los vértices de G' se encuentran en la primer columna de un LR-Mesh de $4N^2 \times 4N^2$.

El LR-Mesh realiza los siguientes pasos:

1. Cada vértice i de la primera columna compara su grado con el grado de sus vecinos y determina el vértice de mayor grado (puede ser incluso él) y se engancha a dicho vértice marcando la arista que los conecta. Si dos vértices k y j tienen el mismo grado, se elige el vértice con la etiqueta más grande para romper simetría.

2. Si el vértice i no se engancha a un vecino, entonces revisa si todos sus vecinos se engancharon a él. Este procedimiento se puede realizar de forma similar al algoritmo presentado en la Sección IV.3 con algunas modificaciones mínimas.

   Si todos los vecinos del vértice i se engancharon a él, entonces el vértice i se engancha a sí mismo (por su arista cero).

   Si uno o más vecinos del vértice i no se engancharon a él, el vértice i pregunta el grado y la etiqueta del vértice al que se enganchó su vecino o vecinos. Compara el grado del vértice o vértices y se engancha al vecino que se enganchó al vértice de mayor grado marcando la arista que los conecta.

Este algoritmo es similar al algoritmo presentado en la Sección IV.3 con algunas modificaciones mínimas. Por lo tanto el algoritmo se ejecuta en $O(1)$ unidades de tiempo.

IV.5. Construcción del tour de Euler

JáJá (1992) describe la técnica del tour de Euler en el contexto de la PRAM de la siguiente forma: Sea $T = (V, E)$ un árbol arbitrario y sea $T' = V, E'$ el grafo dirigido obtenido de T, cuando cada aristas $(u, v) \in E$ se remplaza por dos aristas $\langle u, v\rangle$ y $\langle v, u\rangle$. Dado que el grado de entrada de cada vértice de T' es igual a su grado de salida, T' es un grafo Euleriano; esto quiere decir que T' posee un circuito dirigido que atraviesa cada arista exactamente una vez. Entonces el circuito de Euler de T' se puede usar para calcular en forma óptima y paralela muchas funciones en T.

IV.5.1. Preparación del LR-Mesh para el tour de Euler

Descripción: Para preparar al LR-Mesh, se necesita un conjunto de 8×8 procesadores del LR-Mesh para simular a un vértice de G, con sus 4 puertos de salida y 4 puertos de entrada.

Observaciones: Los procesadores en el LR-Mesh se identifican por su índices (i, j).

Pseudo código: *Preparación_del_LR-Mesh*

Entrada: Los vértices de G que contienen toda la información del R-Mesh, se encuentran localizados en los $4N^2$ procesadores de la primera columna del LR-Mesh.

Salida: Los $4N^2$ vértices de G que contienen toda la información del R-Mesh, en la primera columna del LR-Mesh de $32N^2 \times 32N^2$ separados en múltiplos de ocho.

El LR-Mesh realiza los siguientes pasos:

1. Mover los primeros $4N^2$ vértice de G, de la primer columna al primer renglón.

   a) Cada procesador con índice $i = j$ realiza la configuración $\{NO, E, S\}$.

   Cada procesador con índice $j > i$ realiza la configuración $\{NS, E, O\}$.

   Cada procesador con índice $j < i$ realiza la configuración $\{EO, N, S\}$.

   b) Cada procesador con índice $j = 0$ escribe el dato almacenado en su puerto E.

   c) Cada procesador con índice $i = 0$ lee en su puerto S.

   d) Cada procesador con índice $i = 0$ almacena la información leída en sus registros correspondientes.

2. Se realiza el broadcast

   a) Cada procesador realiza la configuración $\{NS, E, O\}$.

   b) Cada procesador con índice $i = 0$ escribe en su puerto S.

   c) Cada procesador con índice $(8i, i)$ lee su puerto N, donde $1 \leq i \leq 4N^2$.

   d) Cada procesador con índice $(8i, i)$ almacena la información leída en sus registros correspondientes, donde $1 \leq i \leq 4N^2$.

3. Se copian los datos de renglón a columna

   a) Cada procesador con índice $(8i, i)$ realiza la configuración $\{E, N, O, S\}$, donde $1 \leq i \leq 4N^2$.

   El resto de los procesadores realizan la configuración $\{EO, N, S\}$.

   b) Cada procesador con índice $(8i, i)$ escribe en su puerto O, donde $1 \leq i \leq 4N^2$.

c) Cada procesador con índice $j = 0$ lee su puerto E.

d) Cada procesador con índice $j = 0$ almacena la información leída en su registro correspondiente.

De esta forma, cada procesador con índice i múltiplo de ocho contiene la información de un vértice de G.

IV.5.2. Conexiones de cada pseudo-árbol en G'

Descripción: En el grafo G' existe una arista entre los vértice i y j si y sólo si pertenecen al mismo árbol de enganchamiento. Cada vértice del grafo G' requiere conectarse con a lo más cuatro vecinos, por lo tanto se requieren ocho procesadores del LR-Mesh por cada vértice de G' para simular las cuatro aristas de entrada y las cuatro aristas de salida.

Pseudo código: *Conexiones_de_cada_pseudo-árbol_en_G'*

Entrada: Cada vértice de G en la primera columna del LR-Mesh de $32N^2 \times 32N^2$ separados en múltiplos de ocho.

Salida: El LR-Mesh de $32N^2 \times 32N^2$ con todas las conexiones internas necesarias para conectar a cada vértice de G' con sus vecinos.

1. Por convención, cada vértice i etiqueta sus aristas en sentido contrario al de las manecillas del reloj. Cada vértice i de G' realiza la configuración interna de sus procesadores en el LR-Mesh de forma que si un dato entra por la arista j del vértice i en G', entonces el dato sale por la arista $(j + 1)$ mod (grado del vértice i en G'). Para generar las conexiones internas se requieren cuatro columnas de procesadores al lado izquierdo adicionales a la matriz de $32N^2 \times 32N^2$.

2. Se realizan las conexiones internas entre cada vecino en G' en el LR-Mesh de $32N^2 \times 32N^2$

Para ejecutar los dos pasos descritos anteriormente se emplea un algoritmo similar al algoritmo presentado en la Sección IV.4 con algunas modificaciones mínimas.

IV.5.3. Realizar broadcast en cada pseudo-árbol en G'

Descripción: Cada vértice raíz en G' envía su índice a todos los vértices que pertenecen a su árbol.

Pseudo código: *Broadcast_en_cada_pseudo-árbol_en_G'*

Entrada: Las conexiones entre cada vecino en el árbol G' en el LR-Mesh de $32N^2 \times 32N^2$.

Salida: Cada vértice de G' obtiene el índice de la raíz del árbol al que pertenece en el LR-Mesh de $32N^2 \times 32N^2$.

El LR-Mesh realiza los siguientes pasos:

1. Cada vértice en G' que se engancha a sí mismo, manda una señal con su etiqueta por su primer vecino en el pseudo-árbol.

2. Cada vértice en G' con índice múltiplo de ocho, lee sus 4 puertos y almacena la etiqueta de la raíz del árbol al que pertenece en el registro correspondiente.

A continuación se presenta la fase de contracción.

IV.6. Ordenar los vértice en G' por componente

Descripción: Para saber cuántos vértices pertenecen a cada árbol es necesario ordenar los vértice de G' en forma creciente con respecto a la etiqueta de la raíz del árbol

al que pertenecen.

Pseudo código: *Ordenar_los_vértices_en_G′*

Entrada: Cada vértice de G' conoce el índice de la raíz del árbol al que pertenece. Todos los vértice de G' se localizan en la primera columna del LR-Mesh de $32N^2 \times 32N^2$, cada vértice separado en múltiplos de ocho.

Salida: Los vértice de G' localizados en la primera columna del LR-Mesh y ordenados en forma creciente con respecto a la etiqueta de la raíz del árbol al que pertenecen en el LR-Mesh de $4N^2 \times 4N^2$.

1. Agrupar todos los vértice de G'.

   Este algoritmo es similar al algoritmo que se presenta en la Sección IV.5.1, en el cual se expande el LR-Mesh de $4N^2 \times 4N^2$ a $32N^2 \times 32N^2$. La diferencia es que en este algoritmo se realiza el proceso contrario, cada vértice con renglón múltiplo de ocho de la primer columna, se agrupa en las primeras $4N^2$ posiciones en el LR-Mesh, de esta forma el LR-Mesh se contrae de $32N^2 \times 32N^2$ a $4N^2 \times 4N^2$.

2. Utilizando el algoritmo Jang y Prasanna (1995), el cual corre en tiempo constante en un LR-Mesh, se ordenan los vértices de G' en forma creciente con respecto a la etiqueta de la raíz del árbol al que pertenecen. Cada vértice almacena su índice original antes de ser ordenado.

IV.7. Calcular el número de vértices en G' por componente

Descripción: El algoritmo realiza el cálculo del tamaño de cada árbol en el bosque. El algoritmo utiliza la técnica partición de ducto de Miller *et al.* (1993). Para realizar

el cálculo de la cantidad de vértices por componente. Todos los vértices se encuentran agrupados por componente.

Pseudo código: *Calcular_el_número_de_vértices_en_G'*

Entrada: Los vértice de G' localizados en la primera columna del LR-Mesh y ordenados en forma creciente con respecto a la etiqueta de la raíz del árbol al que pertenecen en el LR-Mesh de $4N^2 \times 4N^2$.

Salida: Cada vértice de G' conoce la cantidad de vértices del árbol al que pertenece. Cada vértice de G' se localiza en la posición original antes de ordenarse, en el LR-Mesh de $4N^2 \times 4N^2$.

1. Cada procesador de la primer columna en el LR-Mesh, pregunta por sus puertos N y S si sus vecinos pertenecen al mismo componente.

2. Se utiliza la técnica partición ducto para calcular el número de vértice en G' por componente de la siguiente forma:

   a) Si ambos vecinos del procesador i pertenecen al mismo componente, entonces el procesador i realiza la configuración $\{NS, E, O\}$.

   Si ambos vecinos del procesador i no pertenecen al mismo componente, entonces realiza la configuración $\{E, N, O, S\}$. El procesador almacena un uno en su registro correspondiente (el vértice es aislado).

   Si uno de los dos vecinos del procesador i no pertenece al mismo componente, entonces el procesador i realiza la configuración $\{E, N, O, S\}$.

   b) Cada procesador u cuyo vecino conectado por su puerto S no pertenece a su mismo componente (el cual se llama "último vértice del componente"), escribe su índice por su puerto N.

c) Cada procesador p cuyo vecino conectado por su puerto N no pertenece a su mismo componente (el cual se llama "primer vértice del componente"), lee su puerto S.

d) Cada primer vértice p del componente almacena el dato leído en su registro correspondiente.

Cada primer vértice p del componente resta su índice al valor almacenado en su registro, la diferencia de estos dos valores es la cantidad de vértices en el componente. El resultado lo almacena en su registro $SizeT$ y realiza un broadcast a todos miembros de su componente enviando el valor de $SizeT$ por su puerto S. Cada procesador lee su puerto N y almacena el valor de $SizeT$ en su registro correspondiente.

Cada procesador revisa el valor de $SizeT$, si el valor es mayor a la cota establecida para el nivel de recursión actual, actualiza su status a inactivo. Si el valor de $SizeT$ es menor que la cota establecida para el nivel de recursión actual, actualiza su estatus a activo.

Cada vértice regresa a su posición original, utilizando el valor de su índice original que almacena antes de ordenarse. Este procedimiento se puede realizar utilizando un algoritmo similar al que se presenta en la Sección IV.4 por lo tanto su tiempo de ejecución es $O(1)$ unidades de tiempo.

IV.8. Determinar las aristas externas de cada árbol

Descripción: El algoritmo realiza el cálculo de la cantidad de aristas externas en cada árbol de vértices activos.

Pseudo código: *Determinar_las_aristas_externas*

Entrada: Cada vértice de G' conoce el índice de la raíz del árbol al que pertenece. Todos los vértices de G' se localizan en la primera columna del LR-Mesh de $4N^2 \times 4N^2$.

Salida: Cada vértice activo de G' conoce sus aristas externas (si es que las hay). Todos los vértice de G' se localizan en la primera columna del LR-Mesh de $4N^2 \times 4N^2$.

El LR-Mesh realiza los siguientes pasos:

1. Cada procesador i que representa a un vértice activo pregunta a cada uno de sus vecinos a qué componente pertenece, si el vecino j pertenece a un componente diferente, entonces el procesador i almacena el valor del componente de j (el cual se llama "posible arista externa") en su registro correspondiente.

2. Ordena todos los vértices activos con respecto a su componente utilizando el algoritmo Jang y Prasanna (1995), el cual corre en tiempo constante en un LR-Mesh. Cada vértice almacena su índice original.

   Como cada vértice activo en G' tiene a lo más tres aristas externas, entonces por cada vértice activo en G' se requieren tres procesadores en el LR-Mesh para posteriormente poder ordenar utilizando el algoritmo de Jang y Prasanna (1995).

3. Se expande el LR-Mesh a tres procesadores por cada vértice activo en G'. Este algoritmo es similar al que se muestra en la Sección IV.5.1, la diferencia es que en lugar de ocho procesadores ahora se requieren tres procesadores. Entonces se expande el LR-Mesh de $4N^2 \times 4N^2$ a $12N^2 \times 12N^2$.

4. Cada procesador i múltiplo de tres, conserva una arista externa (si es que la hay) y transmite las otras dos aristas externas (si es que las hay) a los dos procesadores $i+1$ e $i+2$ (uno a la vez).

5. Cada componente i ordena las aristas externas de acuerdo a su índice, utilizando el algoritmo de Jang y Prasanna (1995).

6. Utilizando la técnica partición de ducto, cada componente elimina las aristas externas repetidas para no generar multi-aristas entre dos componentes.

7. Se agrupan todos los vértices de G'.

   Este algoritmo es similar al algoritmo que se presenta en la Sección IV.5.1, en el cual se expande el LR-Mesh de $4N^2 \times 4N^2$ a $32N^2 \times 32N^2$. La diferencia es que en este algoritmo se realiza el proceso contrario, cada procesador con renglón múltiplo de tres de la primer columna, se agrupa en las primeras $4N^2$ posiciones en el LR-Mesh, de esta forma el LR-Mesh se contrae de $12N^2 \times 12N^2$ a $4N^2 \times 4N^2$.

8. Cada vértice regresa a su posición original utilizando el valor de su índice original que almacena antes de ser ordenado. Este procedimiento se puede realizar utilizando un algoritmo similar al que se presenta en la Sección IV.4.

Al ejecutar los algoritmos de enganchamiento y contracción en el LR-Mesh en $O(1)$ unidades de tiempo, cada vértice del grafo G obtiene su representante inmediato (la raíz de su árbol), entonces para que cada vértice obtenga su representante global (la raíz del componente maximal al que pertenece), se requieren $O(\log N)$ iteraciones, por lo tanto el problema USTCON se resuelve en un LR-Mesh de $O(N^2 \times N^2)$ procesadores con un tiempo de ejecución de $O(\log N)$ unidades de tiempo.

IV.9. Análisis de la simulación

En este capítulo se propone la Simulación 1, la cual restringe el número de procesadores que se utilizan para simular un R-Mesh en un LR-Mesh, por tal motivo, se sacrifica

el tiempo de ejecución de dicha simulación. Los algoritmos diseñados en este capítulo permiten ejecutar un enganchamiento y una contracción del algoritmo de Trifonov en tiempo constante; sin embargo, existen $O(\log N)$ enganchamientos y contracciones, resultando en un tiempo de ejecución de $O(\log N)$ unidades de tiempo en un LR-Mesh.

El tamaño del LR-Mesh se calcula a continuación. Las mallas de los algoritmos que se presentan en las Secciones IV.2, IV.3, IV.4, y IV.7 requieren $4N^2 \times 4N^2$ procesadores cada una; la malla del algoritmo que se presenta en la Sección IV.8 requiere $12N^2 \times 12N^2$ procesadores; por último, las mallas de los algoritmos que se presentan en las Secciones IV.5.1, IV.5.2, IV.5.3 y IV.6, requieren $32N^2 \times 32N^2$ procesadores cada una. Como cada uno de los algoritmos en estas mallas se ejecutan en instantes de tiempo distinto, ellas se pueden reutilizar. Por lo que en total, el tamaño del LR-Mesh requerido es igual al tamaño de la malla más grande, esto es $O(N^2 \times N^2)$ procesadores (ver Figura 13).

La Simulación 1 tiene la desventaja de no aprovechar el tamaño de los árboles a contraer (realiza el mismo procedimiento para árboles de 4 vértices que para árboles de $O(N)$ vértices) como lo hace el algoritmo de Trifonov. Un siguiente paso para optimizar dicha simulación, es buscar un mecanismo que aproveche el tamaño de los árboles relativamente pequeños para combinar en tiempo constante, un número no constante de fases de enganchamiento, procurando en lo posible mantener el número procesadores (se hace notar que para combinar varias fases en tiempo constante ya no es posible reutilizar los procesadores como se mencionó anteriormente, por lo que es necesario incrementar el número de procesadores). El diseño de este mecanismo excede los objetivos de este trabajo de tesis y se plantea como trabajo futuro.

En la Figura 13 se muestra la Simulación 1. Note que esta simulación es similar a la propuesta por Fernández-Zepeda *et al.* (2002), pero con un sobrecosto en el número de procesadores. Ambas simulaciones utilizan técnicas diferentes; la Simulación 1 modela

la configuración del R-Mesh como un grafo de $O(N^2)$ vértices, requiriendo rejillas de al menos $O(N^2 \times N^2)$ procesadores para ejecutar las operaciones de ruteo.

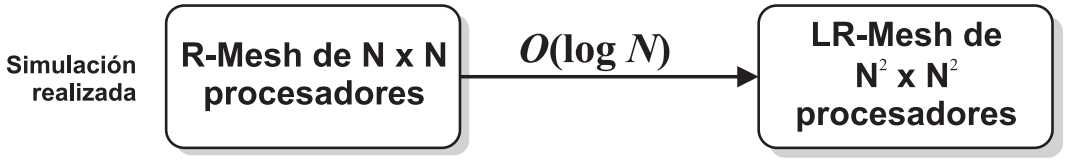

Figura 13. Simulación del modelo R-Mesh en el modelo LR-Mesh

En el siguiente capítulo se presenta la Simulación 2 de un R-Mesh en un LR-Mesh. En esta nueva simulación se utiliza un enfoque opuesto al que se muestra en el presente capítulo, al restringir el tiempo de ejecución a $O(\log \log N)$ unidades de tiempo, sacrificando el número de procesadores. En forma similar, se plantea posteriormente diseñar un mecanismo que gradualmente optimice el número de procesadores.

Capítulo V

Simulación eficiente en tiempo de un R-Mesh en un LR-Mesh

En esta sección se presentan los algoritmos diseñados como base para realizar una simulación eficiente en tiempo de un R-Mesh en un LR-Mesh, a la cual nos referiremos como Simulación 2.

Con base en el algoritmo de Trifonov (2005), en este capítulo se proponen dos tipos de algoritmos de enganchamiento y contracción en el grafo. El primer algoritmo de enganchamiento y contracción se ejecuta en el grafo original y el segundo se ejecuta sobre los super-vértices formados después de ejecutar el primer algoritmo.

El primer algoritmo de enganchamiento y contracción se compone de las siguientes mallas que se ejecutan en un LR-Mesh:

1. Comparación de dos números de x bits.
2. Malla comparadora para obtener el mayor de d números.
3. Malla comparadora para obtener td (el grado en el árbol) del vértice cv.
4. Malla para ejecutar el tour de Euler partiendo del vértice cv.
5. Malla para mover el vértice cv a la raíz de su árbol.

V.1. Comparación de dos números de x bits

Descripción: Se realiza la comparación de dos números de x bits, v_1 y v_2, en el LR-Mesh de $3 \times x$; se supone que $v_1 \neq v_2$. La comparación se realiza bit a bit, del más significativo al menos significativo. En la Figura 14 se muestran los tres tipos de configuraciones que cada procesador puede realizar de acuerdo a los dos bits de entrada (un bit de v_1 y otro de v_2). Si los valores de los bits de v_1 y v_2 son uno y cero, respectivamente, significa que $v_1 > v_2$, por lo tanto la señal se desvía al renglón de procesadores superior inmediato. Por el contrario, si los valores son cero y uno, respectivamente, significa que $v_1 < v_2$, por lo tanto la señal se desvía al renglón de procesadores inferior inmediato. Por último, si los valores de ambos bits son iguales, la señal continúa sin cambios hasta el siguiente par de bits a comparar.

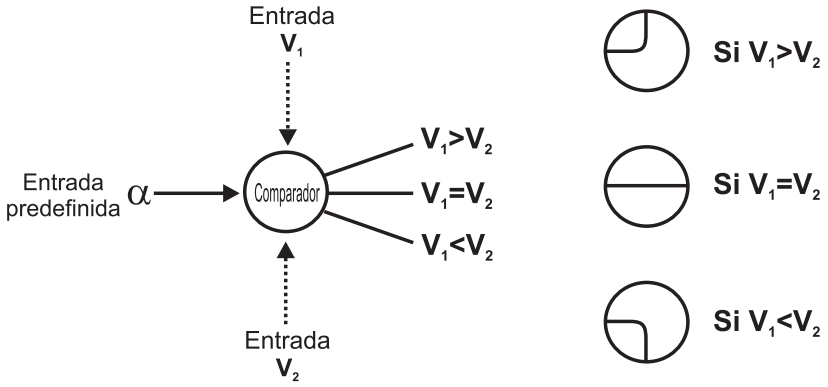

Figura 14. Posibles configuraciones del procesador comparador en un LR-Mesh.

Observaciones: El algoritmo se ejecuta en un LR-Mesh de $3 \times x$ procesadores si se supone el uso de procesadores tipo OR. Los procesadores en el LR-Mesh se identifican por sus índices (*renglón*, *columna*), denotado como (i, j), respectivamente; la numeración inicia en la esquina superior izquierda.

Pseudo código *Compara_dos_números*

Entrada: Los números v_1 y v_2 se encuentran representados en binario y ubicados

en el segundo renglón de la malla, un bit por procesador, y el más significativo en la columna más izquierda. La Figura 15 muestra el esquema para realizar la comparación de dos números.

Salida: Si $v_1 > v_2$, la señal sale por puerto E del procesador $(1, x)$. De forma similar, si $v_1 < v_2$, la señal sale por puerto E del procesador $(3, x)$.

Como se puede observar en la Figura 15, se consideran procesadores tipo OR. Estos procesadores facilitan el entendimiento del funcionamiento de la malla, pero como su implementación no es trivial en un LR-Mesh, éstos se reemplazan usando repetición de mallas como se explica más adelante.

La Figura 15 muestra el diagrama generado para calcular el mayor de dos números cuando $x = 4$ bits.

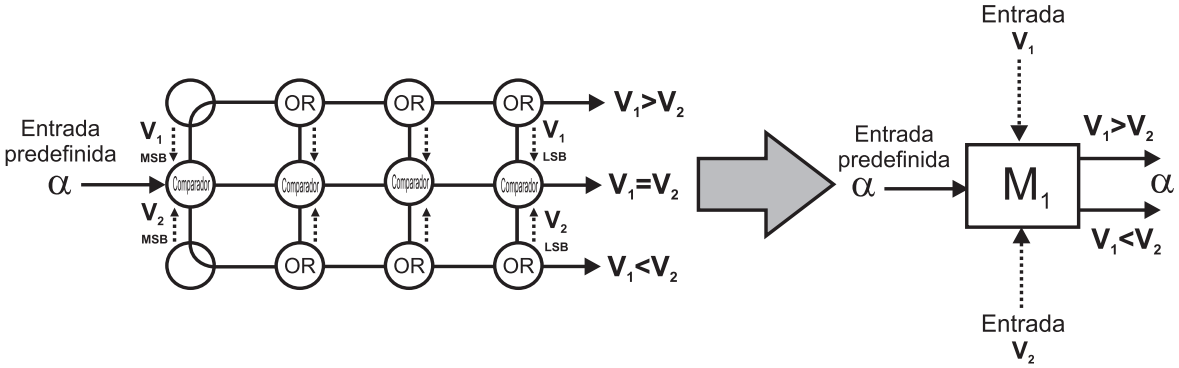

Figura 15. La malla M_1 compara dos números de x bits. Para este ejemplo, $x = 4$.

Eliminando los procesadores tipo OR: Como los ductos de un LR-Mesh no se pueden bifurcar, ni dos ductos distintos pueden converger, se realiza una modificación a la malla de la Figura 15, como se muestra a continuación. La idea central de esta modificación se expresa en la Figura 16. En esta figura, al eliminarse el procesador tipo OR, se tiene que incluir una copia de la malla M para cada una de las líneas de entrada que tenía la compuerta OR. Esta modificación genera una nueva malla con un funcionamiento equivalente a la anterior, pero con un sobrecosto en el número de

procesadores. La malla resultante que se genera al quitar los procesadores tipo OR de la malla de la Figura 15 se muestra en la Figura 17. Note que el tamaño de esta nueva malla es de $(2x + 1) \times x$ procesadores. En los algoritmos subsecuentes, por simplicidad, se seguirán usando procesadores tipo OR en las figuras.

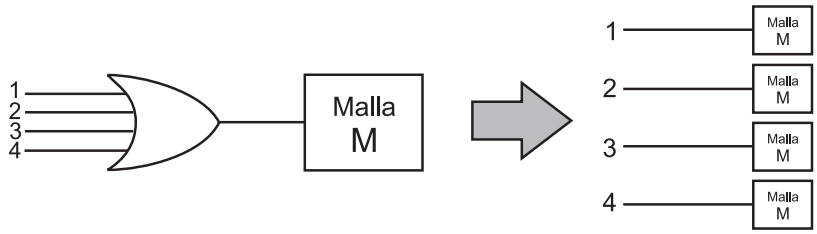

Figura 16. Esquema de modificación de procesadores tipo OR.

La Figura 17 Muestra la malla resultante al quitar los procesadores tipo OR

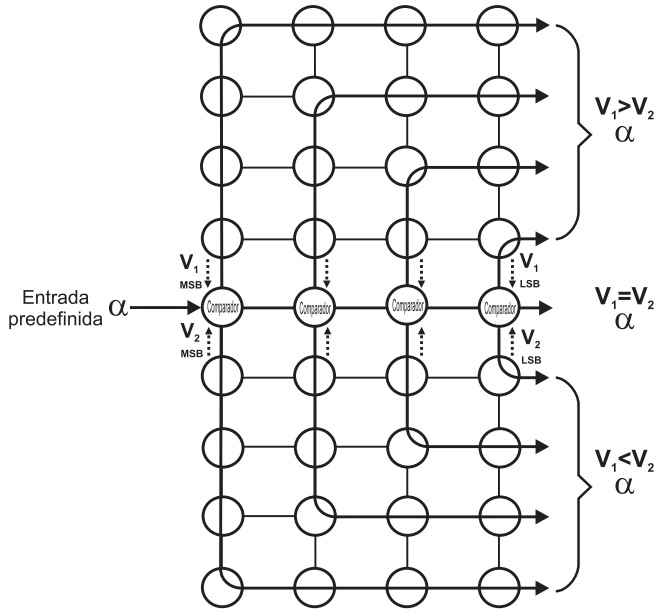

Figura 17. Malla resultante al quitar los procesadores tipo OR.

Cantidad de procesadores requeridos: Utilizando mallas con procesadores tipo OR se requieren $3 \times x$ procesadores. Utilizando procesadores convencionales se requiere $(2x + 1) \times x$ procesadores.

La Figura 18 ilustra la comparación de dos números de tres bits ($v_1 = 101_2$ y $v_2 = 100_2$). Note que la señal α sale por el procesador (3,3), indicando que $v_1 > v_2$.

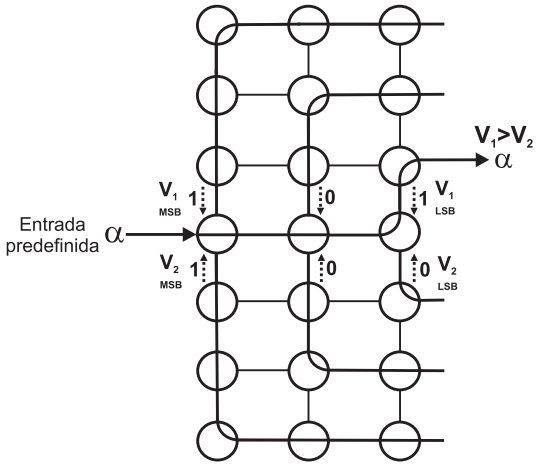

Figura 18. Malla M_1. Ejemplo de una comparación dos números de tres bits en un LR-Mesh.

V.2. Malla comparadora para obtener el mayor de d números

Descripción: La Figura 19 muestra la malla M_2 que genera el mayor de d números de x bits. Esta malla realiza la comparación de d números de x bits en un LR-Mesh (donde d es una constante) al conectar en cascada un conjunto de mallas M_1 (ver la Sección V.1). Cada malla M_1 realiza una comparación entre dos números como se muestra en la Figura 17.

Pseudo código: *Mayor_de_d_números.*

Entrada: Los d números se encuentran posicionados en un LR-Mesh de la siguiente forma: cada cuadro que se muestra en la Figura 19 representa una malla M_1 (como la que se muestra en la Figura 15a). En la malla superior izquierda de la Figura 19 se comparan los vértices v_1 y v_2. En la siguiente columna a la derecha, se compara al mayor

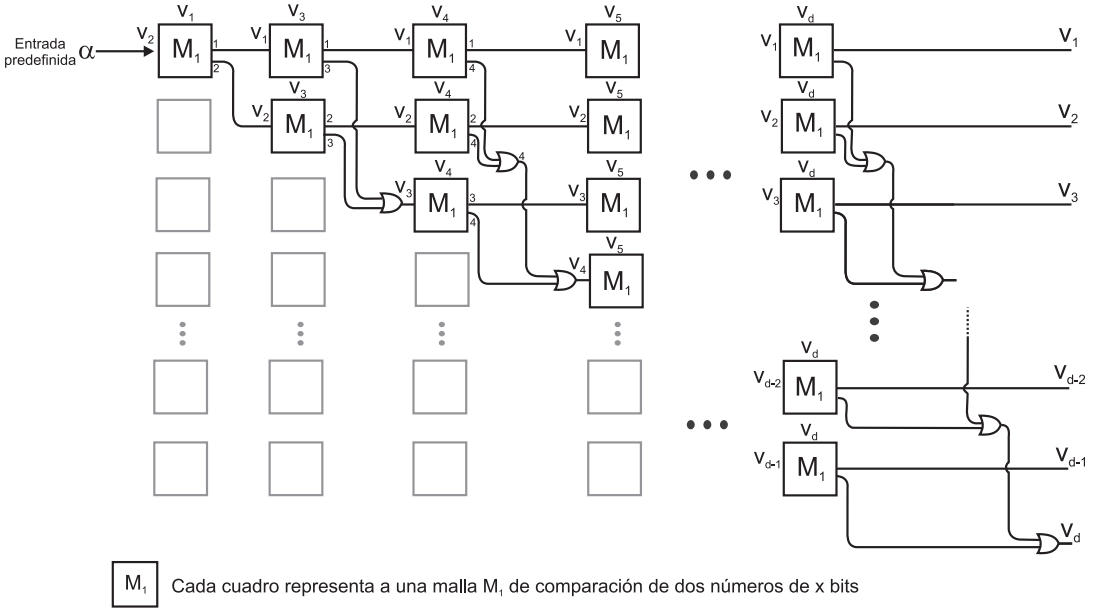

Figura 19. Malla M_2. Esta malla calcula el mayor de d números de x bits.

de v_1 y v_2 con v_3, y así sucesivamente, comparando un nuevo vértice contra el mayor de los anteriores. Finalmente, en la columna más derecha se proporciona una señal que indica cuál de los d números fue mayor.

Salida: Emerge una señal predefinida α por una de las d posibles salidas de la malla. Esta señal representa al mayor de los d números.

Cantidad de procesadores requeridos: Utilizando mallas con procesadores tipo OR se tiene lo siguiente. El ancho de la malla M_2 está formada de d mallas M_1, cada una con un ancho de x procesadores, lo que da un total de dx procesadores. Por otro lado, se requieren d^2 procesadores tipo OR para implementar las compuertas OR entre cada columna de mallas M_1. Así, el ancho de la malla M_2 es de $O(dx + d^2)$ procesadores. El alto de la malla M_2 está formado por d mallas M_1; lo anterior da un total de $3d$ procesadores. Por lo tanto, la malla M_2 requiere $O(d \times (dx + d^2))$ procesadores.

Utilizando mallas con procesadores convencionales se tiene lo siguiente:

El alto de la malla M_2 se calcula como sigue: El alto de la primer columna es $O(x)$; el

alto de la segunda es $O(x^2)$ ya que se requieren copiar $O(x)$ veces la siguiente etapa de comparadores. Siguiendo este procedimiento, la columna más derecha tiene una altura de $O(x^{d-1})$ procesadores.

El ancho de la malla M_2 está formada por d mallas M_1, cada una con un ancho de x procesadores, lo que da un total de dx procesadores. Por otro lado, para implementar las compuertas OR con sus líneas de ruteo, se requieren al menos $O(x^{d-2})$ procesadores, ya que se requiere una columna de procesadores para rutear un ducto a cada una de las $O(x^{d-2})$ mallas tipo M_1 en la última columna de la malla M_2. El ancho total de la malla M_2 es $O(dx + x^{d-2})$ procesadores.

Por lo tanto, el tamaño de la malla M_2 es de $O((x^{d-1}) \times (dx + x^{d-2}))$ procesadores.

V.3. Malla comparadora para obtener td (el grado en el árbol) del vértice cv

Descripción: La Figura 20 muestra la malla M_3 que calcula el grado td del vértice cv en el árbol. Esta malla realiza dicho cálculo sobre un LR-Mesh de la siguiente forma: se conectan en cascada $d + 1$ etapas de mallas comparadoras tipo M_2 (como la de la Figura 19), donde d es la cantidad de vecinos de cv. Al aplicar una señal predefinida α en la primer malla M_2, ésta sale por el procesador al que el primer vecino de cv, por ejemplo v_1, se engancha en el árbol. Si el vértice al que se engancha v_1 es cv, entonces la señal α pasa al siguiente comparador en el renglón inmediato superior, indicando que el grado de cv es al menos uno. En caso contrario, la señal α fluye hacia el siguiente comparador en el mismo renglón, indicando que hasta el momento el grado de cv en el árbol es cero. Este proceso se repite para los restantes $d-1$ vecinos de cv $(v_2, v_3, \ldots, v_d)$; cada vez que se encuentre un vecino que se engancha a cv, se sube un renglón. De esta forma se contabiliza el número de vecinos de cv que se conectan a él. Finalmente, en la

última etapa de comparadores, el vértice cv determina si él se engancha a sí mismo o a un vecino. En caso de que se enganche a un vecino, cv incrementa su contador a uno (subiendo un nivel), como se ve en la última etapa de la Figura 20.

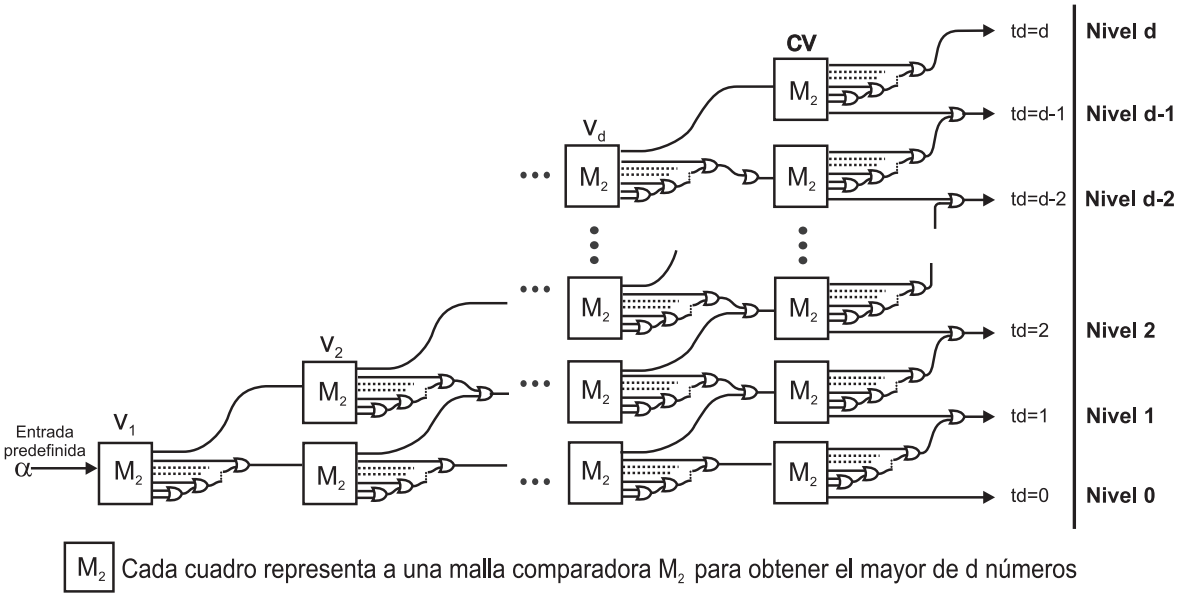

Figura 20. Malla M_3. Esta malla calcula el grado td del vértice cv en su árbol sobre un LR-Mesh.

Pseudo código *Grado_en_el_árbol_td_del_vértice_cv*

Entrada: La lista de adyacencias de cada vértice vecino de cv; la información del vecino v_i se encuentra en la etapa de comparadores i. En la última etapa se almacena la lista de adyacencias de cv.

Salida: La señal predeterminada α fluye y sale únicamente por uno de los niveles dependiendo de la cantidad de vecinos de cv que se engancharon a él.

Cantidad de procesadores requeridos: Utilizando mallas con procesadores tipo OR se tiene lo siguiente. El ancho de la malla M_3 está formada de $O(d)$ mallas tipo M_2, cada una con un ancho de $O(dx + d^2)$ procesadores, lo que da un total de $O(d^2x + d^3)$ procesadores. Por otro lado, la altura de la malla M_3 consiste de $O(d)$ mallas M_2; cada malla M_2 tiene una altura de $O(d)$; por lo que la altura de la malla M_3 es de $O(d^2)$. Por

lo tanto, la malla M_3 se puede implementar con un LR-Mesh de $O((d^2) \times (d^2x + d^3))$.

Utilizando mallas sin procesadores tipo OR se tiene lo siguiente. La primera malla M_2 de la Figura 20 requiere $O((x^{d-1}) \times (dx + x^{d-2}))$ procesadores. La segunda columna consiste de d mallas M_2, la tercera de d^2 mallas M_2, y así sucesivamente hasta la última columna que requiere d^d mallas tipo M_2. Por lo tanto el LR-Mesh requerido es de $O((d^d x^{d-1}) \times (d^2x + dx^{d-2}))$.

V.4. Malla para ejecutar el tour de Euler partiendo del vértice *cv*

Descripción: Para implementar cada paso del tour de Euler, se ejecuta un algoritmo semejante al descrito en la Sección V.3. En el primer paso del tour se necesita encontrar al primer vecino de cv que se engancha a él; entonces la señal predefinida α se transmite al procesador que representa a dicho vecino, por ejemplo v_1. En el siguiente paso del tour, se ejecuta un proceso similar al anterior con los vecinos de v_1 para que la señal α se propague al primer vecino de v_1 que se engancha a él. El algoritmo sigue un orden preestablecido para hacer la búsqueda, utilizando numeraciones locales; si la señal α entra a un vértice por su arista i, la señal debe salir por la arista $[i+1 \mod (\textit{grado del vértice})]$. El proceso anterior se repite hasta que la señal α regrese al vértice cv (el cual se representa en la Figura 23 por el vértice con relleno de color negro); en este caso, el algoritmo realiza un proceso similar al antes descrito, pero enviando la señal α a una malla en un nivel inmediato inferior, indicando que la señal α regresó una vez. Si el grado del vértice cv en el árbol es td, la señal α debe regresar td veces al vértice cv para que el tour de Euler se considere terminado y la señal α fluya por la malla en el nivel más inferior de la Figura 23 (el renglón con $td = 0$). Si la señal α no sale por el nivel $td = 0$, se entiende que el número de vértices en el árbol es mayor al límite permitido

para el nivel de recursión dado. Como no se conoce de antemano el valor de td, ni el tamaño del árbol, la señal α puede llegar al nivel $td = 0$ en la malla M_6 en cualquier etapa de la misma; por esta razón se requiere una serie de procesadores tipo OR en dicho nivel. Por otro lado, si el árbol es más grande que el límite establecido, se requiere una serie de procesadores tipo OR para implementar la generación de una señal única que notifique este hecho. La Figura 21 la forman $d + 1$ mallas tipo M_2 como la que se muestra en la Figura 19. La malla de la Figura 23, denominada M_6, está constituida por $d \times 2(n' - 1)$ mallas tipo M_5. La malla M_5 está constituida por n mallas M_4 como se muestra en la Figura 23. El parámetro n' representa una cota para un nivel de recursión dado.

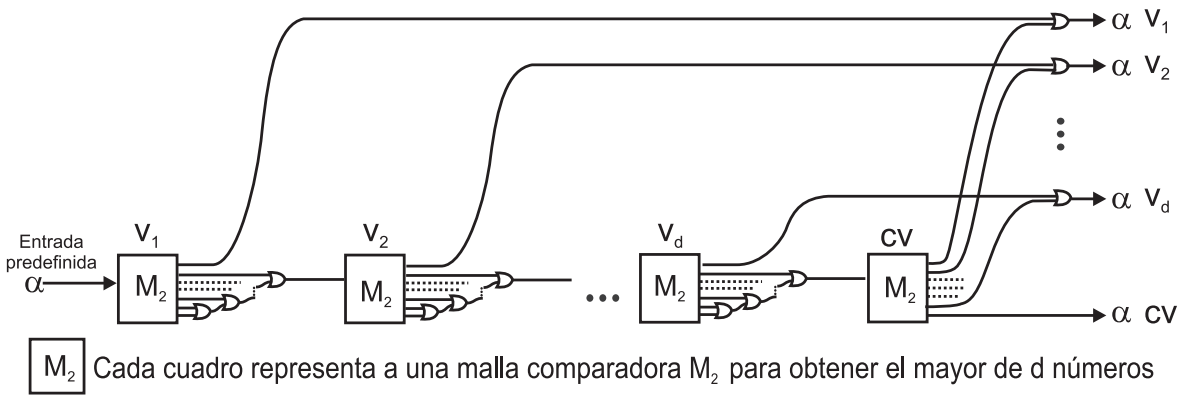

Figura 21. Malla M_4. Esta malla calcula un paso en el tour de Euler sobre el árbol del vértice cv.

Pseudo código *tour_de_Euler_del_vértice_cv*

Entrada: Se requiere la lista de adyacencias de cada vértice para cada una de las mallas M_5, dentro de cada malla M_6.

Salida: Si la cantidad de vértices que componen el árbol al que pertenece cv es menor o igual a la cantidad permitida para el nivel de recursión actual, $2(n' - 1)$, la señal predeterminada α fluye a través de las mallas y la recibe únicamente el procesador localizado más a la derecha de la fila inferior, el cual representa a $td = 0$. Por el contrario,

si la cantidad de vértices es mayor a $2(n'-1)$, la señal predefinida α fluye hacia otro nivel de recursión con un valor $2(n''-1)$, siguiendo un procedimiento similar al anterior.

Cantidad de procesadores requeridos: Utilizando mallas con procesadores tipo OR se tiene lo siguiente. La malla M_4 (ver Figura 21) consiste de $O(d)$ mallas tipo M_2. Así que el tamaño de la malla M_4 es $O(d \times (d^2x + d^3))$ procesadores.

La malla M_5 está formada de N mallas tipo M_4 como se ilustra en la Figura 22. Por lo que la malla M_5 requiere un LR-Mesh de $O(dN \times (d^2x + d^3))$ procesadores.

Cada renglón de la malla M_6 consiste de $2(n'-1)$ mallas tipo M_5 y existen d de estos renglones. Por lo tanto el tamaño de la malla M_6 es de $O(d^2N \times (n'-1)(d^2x + d^3))$ procesadores.

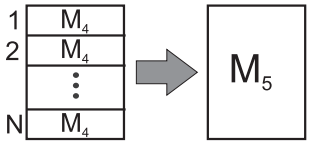

Figura 22. La malla M_5 está formada por N mallas tipo M_4.

Utilizando mallas sin procesadores tipo OR se tiene lo siguiente. La primera malla M_2 de la Figura 21 requiere $O((x^{d-1}) \times (dx + x^{d-2}))$ procesadores. Las columnas subsecuentes son muy similares a las columnas de la malla M_3; de aquí que ambas mallas tengan aproximadamente el mismo número de procesadores. Por lo tanto, la malla tipo M_4 requiere un LR-Mesh de $O((d^d x^{d-1}) \times (d^2x + dx^{d-2}))$ procesadores.

La malla M_5 está formada de N mallas tipo M_4 como se ilustra en la Figura 22. Por lo que la malla M_5 requiere un LR-Mesh de $O((d^d x^{d-1}N) \times (d^2x + dx^{d-2}))$ procesadores.

Cada renglón de la malla M_6 consiste de $2(n'-1)$ mallas tipo M_5, como se ilustra en la Figura 23. La salida de la primera malla M_5 en cualquiera de los renglones se conecta a d^dN mallas M_5 en la segunda columna, éstas a su vez se conectan a $d^{2d}N^2$ mallas M_5 en la tercer columna. Este proceso se repite sucesivamente hasta la última columna, la cual

está formada por $d^{2(n'-1)d}N^{2(n'-1)}$ mallas M_5. Por lo que el tamaño de la malla M_6 es de $O((d^d x^{d-1}N)(d^{2(n'-1)d+1}N^{2(n'-1)}) \times (d^2x + dx^{d-2})(n'-1))$ procesadores; simplificando queda $O((x^{d-1}d^{2d(n'-1)+d+1}N^{2(n'-1)+1}) \times (d^2x + dx^{d-2})(n'-1))$ procesadores.

En la Figura 23 se muestra el diagrama generado para realizar el tour de Euler sobre el LR-Mesh.

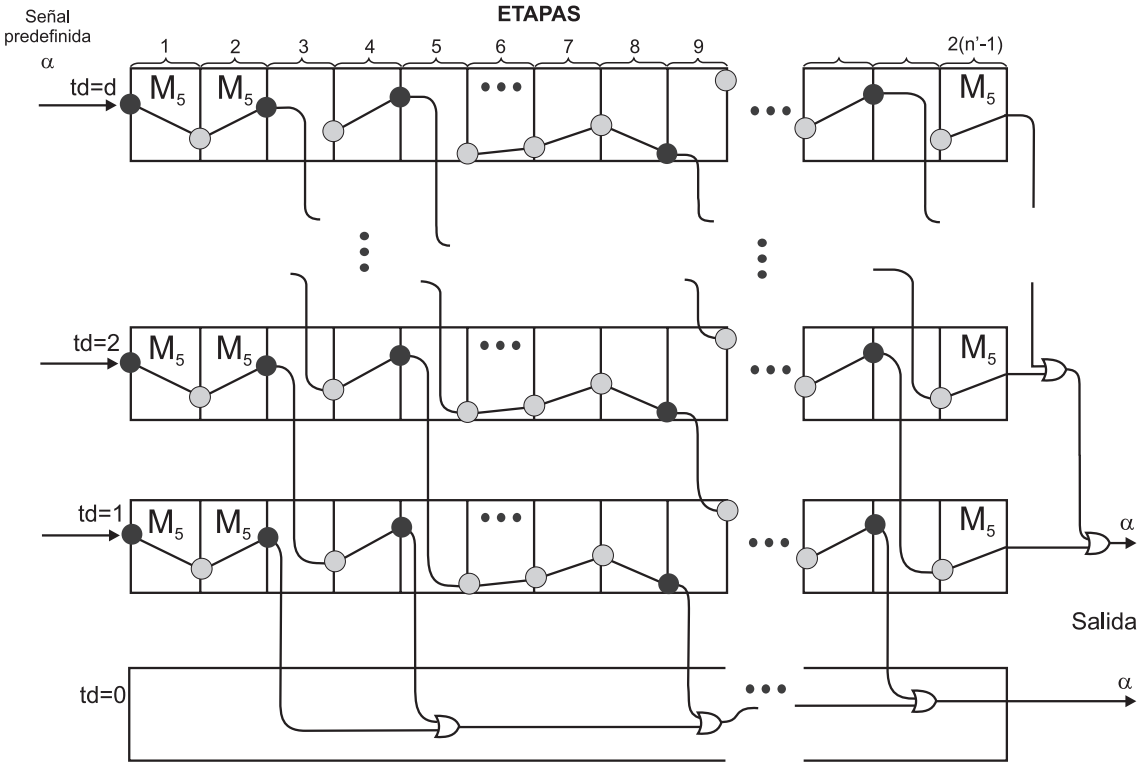

Figura 23. Malla M_6. tour de Euler sobre el LR-Mesh.

V.5. Malla para mover el vértice cv a la raíz de su árbol

Descripción: Se ejecuta el tour de Euler de forma similar a como se describe en la Sección V.4; la única diferencia con este algoritmo es que cuando un vértice se visita por primera vez en el tour, se revisa si dicho vértice se engancha así mismo (verifica si dicho vértice es la raíz del árbol), esto se logra utilizando un algoritmo como el que se muestra

en la Sección V.2. Cuando se encuentra el vértice que representa a la raíz del árbol, la señal predeterminada α fluye hacia dicho vértice, el cual se convierte en la nueva raíz llamada cv'. La Figura 24 ilustra este procedimiento. Se hace notar que cada cuadro de esta figura es una versión muy similar a la malla M_5, que ahora se denomina M_{5bis}. Esta nueva malla incorpora algunos procesadores adicionales para revisar si el vértice que se visita en el tour se engancha así mismo. En la Figura 24 se puede ver, que como no se sabe cuál de todos los vértices en el árbol es la nueva raíz, la señal predefinida α puede salir de la malla en cualquier posición, indicando con eso la nueva raíz en el árbol.

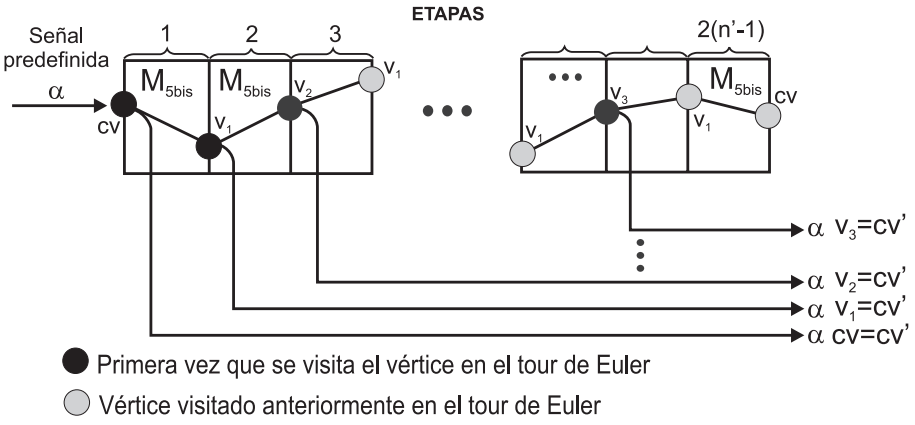

Figura 24. Malla M_7. tour de Euler sobre el LR-Mesh, construido con base en mallas tipo M_{5bis}.

Pseudo código *Mover_el_vértice_cv_a_la_raíz*

Entrada: Se requiere la lista de adyacencias de cada vértice para cada una de las mallas M_{5bis}, dentro de la malla M_7.

Salida: La señal predefinida α fluye únicamente por el vértice que representa la nueva raíz del árbol para el nivel de recursión $2(n'-1)$.

Cantidad de procesadores requeridos: Utilizando mallas con procesadores tipo OR se tiene lo siguiente. La salida de la malla M_6 de la Figura 23 cuando el árbol del vértice cv tiene a lo más $2(n'-1)$ vértices es por el renglón inferior. Esta salida se conecta

a la entrada de la malla M_7 que se ilustra en la Figura 24. La malla M_7 está constituida por $2(n'-1)$ mallas tipo M_{5bis}; la malla M_{5bis} tiene un tamaño equivalente al de la malla M_5, esto es, $O(dN \times (d^2x + d^3))$ procesadores. Por lo tanto, la malla M_7 requiere $O((dN + 2(n'-1)) \times (n'-1)(d^2x + d^3))$ procesadores. Note que en el primer término de la expresión anterior se incluye el sumando $2(n'-1)$, la cual representa las $2(n'-1)$ posibles salidas de la malla M_7.

Como se mencionó anteriormente, se tienen fases de enganchamiento y contracción para vértices en el grafo y para super-vértices. A continuación se describen las fases que corresponden a los segundos.

Un super-vértice cv' es una versión simplificada de un árbol en donde se considera únicamente la raíz del mismo, junto con el conjunto de aristas que conectan al árbol del super-vértice cv' con otros árboles. Todos los vértices dentro del árbol del super-vértice cv' ya no participan en el algoritmo.

Estas fases tienen una estructura similar a las que se describieron en las secciones anteriores y se componen de las siguientes mallas:

1. Malla para calcular el grado d' en el grafo del super-vértice cv'.
2. Malla para calcular el grado td' en el árbol del super-vértice cv'.
3. Malla para ejecutar el tour de Euler en super-vértices partiendo de cv'.
4. Malla para mover al super-vértice cv' a la raíz del árbol de super-vértices.

V.6. Malla para calcular el grado d' en el árbol T del super-vértice cv'

Descripción: El grado d' del árbol T que corresponde al super-vértice cv' se calcula incrementando el contador d' cada vez que se encuentra la primer arista que conecta a

T con un árbol vecino T'. Para calcular este grado en el LR-Mesh, se ejecuta el tour de Euler de forma similar a como se describe en la Sección V.4; este tour se lleva a cabo sobre el árbol T, pero partiendo del super-vértice cv' en lugar de cv (ver Figura 25). En el i-ésimo paso del tour, se revisa si se llega por primera vez a un vértice, por ejemplo v_i (ver etapa 1 de la Figura 25); si esto se cumple, se verifica si el tamaño del árbol al que pertenece cada vecino de v_i es menor al parámetro $2(n' - 1)$ (ver etapa 2 en la misma figura), que corresponde a la cota del nivel de recursión anterior; si es mayor a dicho parámetro, se incrementa el contador d'; en caso contrario, se calcula la raíz de cada uno de los vecinos de v_i utilizando un algoritmo similar al que se presenta en la Sección V.5 (ver etapa 3 en la misma figura). Se compara el índice del super-vértice cv' con el índice del super-vértice de la raíz de cada vecino de v_i para encontrar aristas que conecten al árbol T con otros árboles (ver etapa 4 en la misma figura). Se repite este procedimiento para identificar todas las aristas entre el árbol T con sus árboles vecinos. Como puede haber multiaristas, cada arista nueva que se descubre, se verifica que no se haya descubierto antes (ver etapa 5 en la misma figura). La forma de manejar el contador d' es similar a la forma como se lleva el contador en el algoritmo que calcula el grado td del vértice cv en el árbol (ver Sección V.3). En la Figura 25 se muestra la malla 8, la cual representa el esquema que calcula el grado d' del árbol T que corresponde al super-vértice cv'. Este tipo de mallas son muy complejas y por tal razón, las figuras son a un nivel muy elevado y se omiten muchos detalles.

El valor máximo del grado d' del super-vértice cv' es $n'' - 1$ (si n'' es la cantidad máxima de vértices para un nivel de recursión dado, entonces $n'' - 1$ es la cantidad máxima de vecinos que puede tener el super-vértice cv').

Pseudo código *Grado_del_super-vértice_cv'*

Entrada: Se requiere la lista de adyacencias de cada vértice para cada una de las

diversas mallas.

Salida: La señal predefinida fluye únicamente por una de las $n'' - 1$ posibles salidas, que son la cantidad máxima de vecinos que puede tener el super-vértice cv' para ese nivel de recursión.

Cantidad de procesadores requeridos: Al observar la malla M_8 que se ilustra en la Figura 25 se puede ver que entre cada par contiguo de mallas M_7 (pertenecientes al tour de Euler), potencialmente pueden existir las mallas que se ilustran en la parte inferior de dicha figura. En el peor de los casos, el tamaño de estas mallas es equivalente a obtener $O(d)$ tours de Euler completos; la malla M_8 calcula un equivalente a $2d(n' - 1)$ tours de Euler como el de la malla M_6 que se ilustra en la Figura 23. Por lo que el ancho de la malla M_8 es aproximadamente $4d(n' - 1)^2(d^2x + d^3)$. Por otro lado, para llevar el contador d' en el árbol, se requieren al menos d' renglones como el que se ilustra en la Figura 25; por lo que la altura de la malla M_8 es $O(d^2d'N)$ procesadores. Por lo tanto, para implementar la malla M_8 se requiere al menos un LR-Mesh de $O\left((d^2d'N) \times d(n' - 1)^2(d^2x + d^3)\right.$ procesadores.

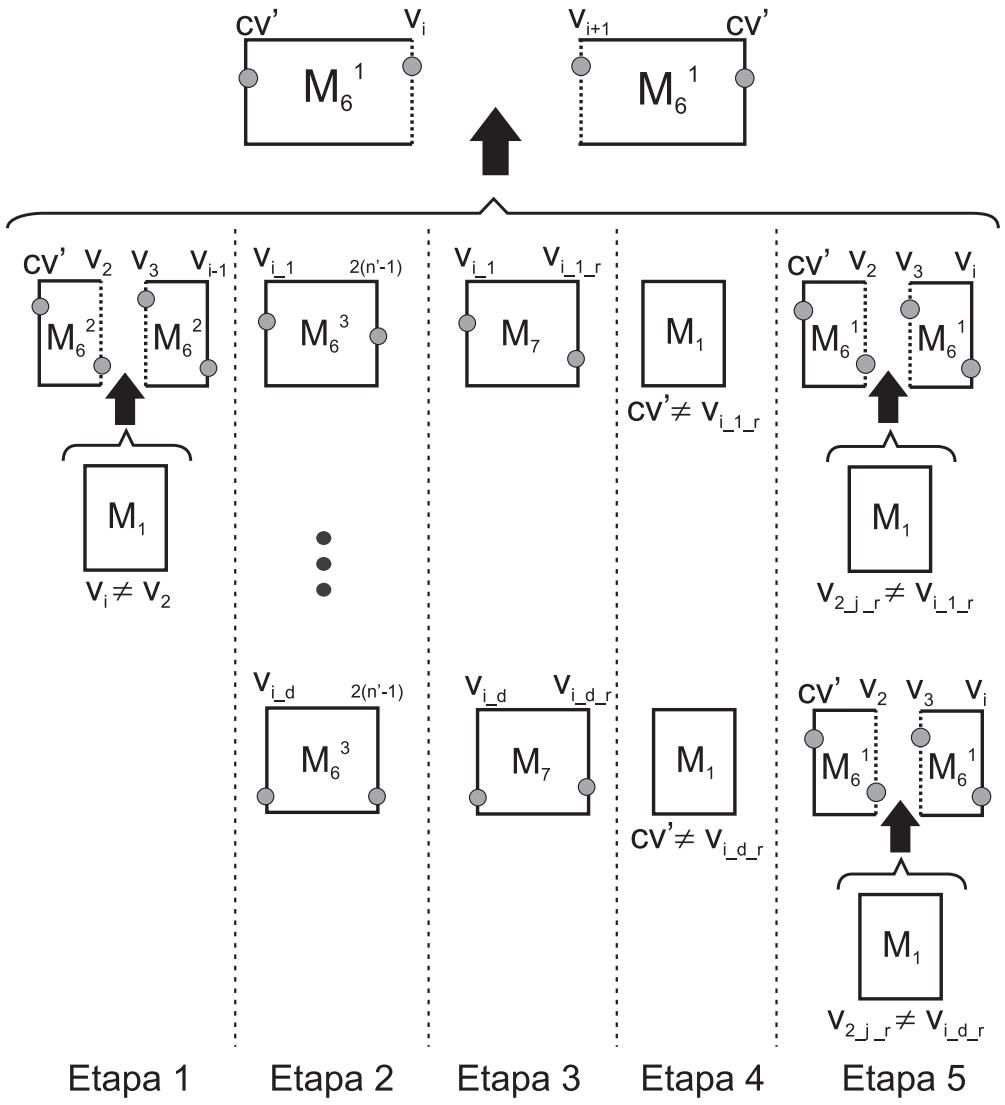

Figura 25. Malla M_8. Esta malla calcula el grado d' en el árbol T del super-vértice cv'.

V.7. Malla para calcular el grado en el árbol td' del super-vértice cv'

Descripción: El procedimiento para calcular el grado en el árbol td' del super-vértice cv', es similar al descrito en la Sección V.3; la diferencia es que en lugar de que el vértice cv revise a sus vecinos en el grafo, en este procedimiento el super-vértice cv' revisa a los vecinos del árbol T al que pertenece. Para calcular td', primero se detectan las aristas externas al árbol T utilizando un procedimiento similar al descrito en la Sección V.6. Cada vez que se encuentra una arista externa al árbol T, se revisa si el vecino T' al que se llega por dicha arista se engancha a el super-vértice cv'; en este paso, el algoritmo revisa el grado de cada uno de los vecinos del árbol T', en forma similar al algoritmo que se presenta en la Sección V.6, y obtiene al super-vértice mayor en grado y en etiqueta de todos ellos; si dicho vecino es el árbol T, entonces significa que el super-vértice del árbol T' se engancha a cv'. Por cada vecino del super-vértice cv' que se enganche a él, se incrementa en una unidad el valor del contador td', de forma similar a como de hace en la Sección V.3; por último, si cv' no se engancha a sí mismo (si no es la raíz del nuevo árbol de super-vértices generado) se incrementa el contador td' en una unidad.

Pseudo código *Grado_td'_del_super-vértice_cv'*

Entrada: Se requiere la lista de adyacencias de cada vértice para cada una de las mallas en los distintos tipos de mallas.

Salida: La señal predefinida fluye únicamente por uno de los niveles dependiendo la cantidad de vecinos que se engancharon al super-vértice cv' para ese nivel de recursión, indicando el valor td'.

Cantidad de procesadores requeridos: El cálculo del grado td' del super-vértice cv' en el árbol de super-vértices es similar al cálculo del grado td del vértice cv en su

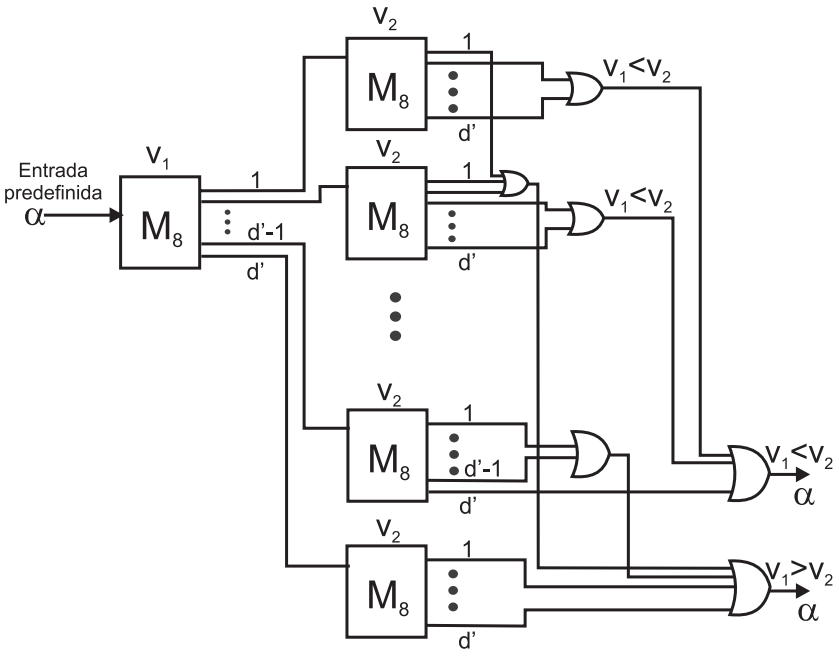

Figura 26. Malla M_1'. Esta malla calcula el mayor de dos super-vértices en grado

árbol. Antes de calcular el grado td, se explica el procedimiento para calcular el mayor de dos números, y posteriormente el mayor de d' números. Se hace notar que en el grafo original, el grado de cada vértice es un dato de entrada; para un super-vértice, dicho grado se tiene que calcular por medio de la malla M_8; en la Figura 26 se ilustra la malla M_1', la cual compara el grado de dos super-vértices (esta figura únicamente hace la comparación del grado de dos super-vértices; en caso de empate se tienen que comparar sus etiquetas, la cual es una malla similar a la malla de la Figura 15); la Figura 26 tiene un ancho equivalente al ancho de dos mallas M_8 y una altura equivalente a la altura de $O(d')$ mallas tipo M_8. Por lo tanto, el tamaño de la malla M_1' es $O\left((d^2(d')^2N) \times d(n'-1)^2(d^2x + d^3)\right)$ procesadores.

Para comparar el mayor de d' vecinos se utiliza la malla M_2' (ver Figura 27), la cual es similar a la malla M_2, pero está formada con mallas M_1' en lugar de M_1. El ancho de la malla M_2' lo constituyen d' columnas de mallas M_1' y su altura por d'

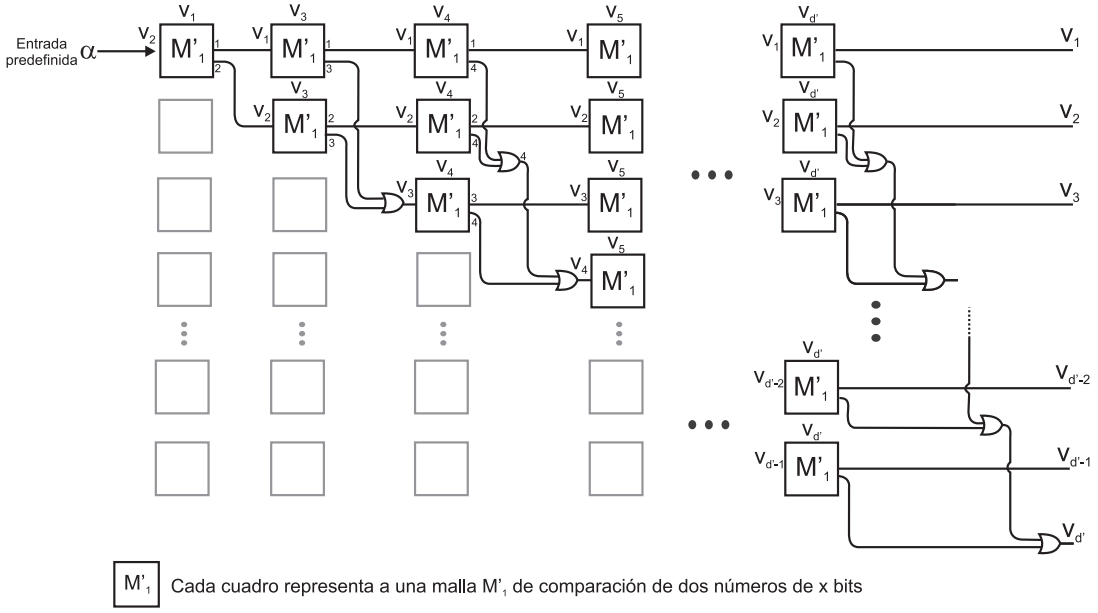

Figura 27. Malla M_2'. Esta malla calcula el mayor de d' vértices en grado y etiqueta.

mallas tipo M_1'. Por lo tanto, el número de procesadores requeridos por la malla M_2' es $O\left((d^2(d')^3N) \times dd'(n'-1)^2(d^2x+d^3)\right)$.

Para calcular el grado td' del super-vértice cv' en el árbol de super-vértices se utiliza la malla M_3' (ver Figura 28); esta malla es similar a la malla M_3, pero está constituida por mallas M_2' en lugar de M_2. El ancho de la malla M_3' está formada de $O(d')$ mallas M_2', y su altura con $O(d')$ mallas M_2'. Por lo tanto, la malla M_3' se puede implementar con un LR-Mesh de $O\left((d^2(d')^4N) \times d(d')^2(n'-1)^2(d^2x+d^3)\right)$ procesadores.

V.8. Malla para ejecutar el tour de Euler en super-vértices partiendo de cv'

Descripción: La ejecución del tour de Euler en super-vértices es similar a la que se presenta en el algoritmo de la Sección V.4. Cada paso en el tour de Euler se realiza a través de una arista del árbol, la cual se encuentra por medio de la malla de la Figura 29, la cual es similar a la que se presenta en la Sección V.7. La diferencia principal entre

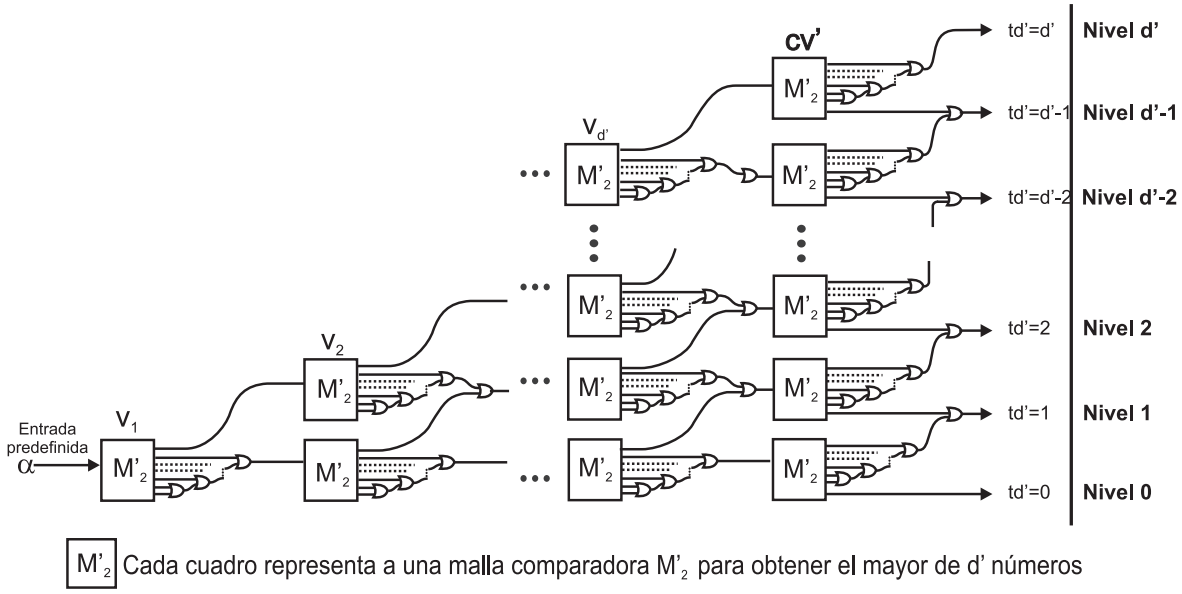

Figura 28. Malla M_3'. Esta malla calcula el grado td' del super-vértice cv'.

el algoritmo de la Sección V.4 y el de la presente sección, es que este último, requiere realizar muchos cálculos en forma recursiva, que no requería el primero. Esta gran cantidad de cálculos repercute en el tamaño y complejidad de las mallas de procesadores cuando se implementa el algoritmo en el LR-Mesh. Para la descripción de esta sección, el lector se puede apoyar en la Sección V.4, considerando únicamente a las figuras 28, 29, 30 y 31, en lugar de las figuras 20, 21, 22 y 23, respectivamente.

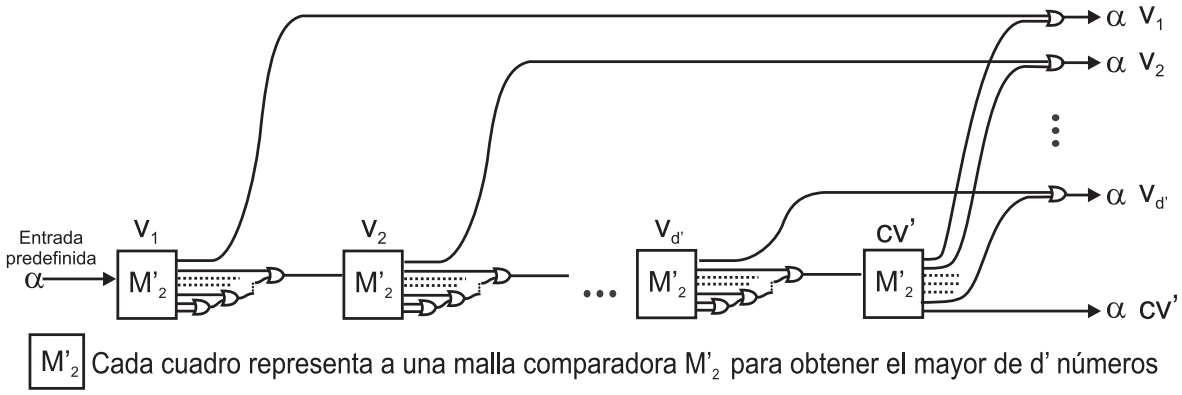

Figura 29. Malla M_4'. Esta malla calcula un paso en el tour de Euler sobre el árbol de super-vértices.

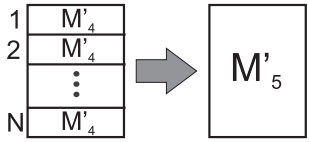

Figura 30. La malla M'_5 está formada por N mallas tipo M'_4

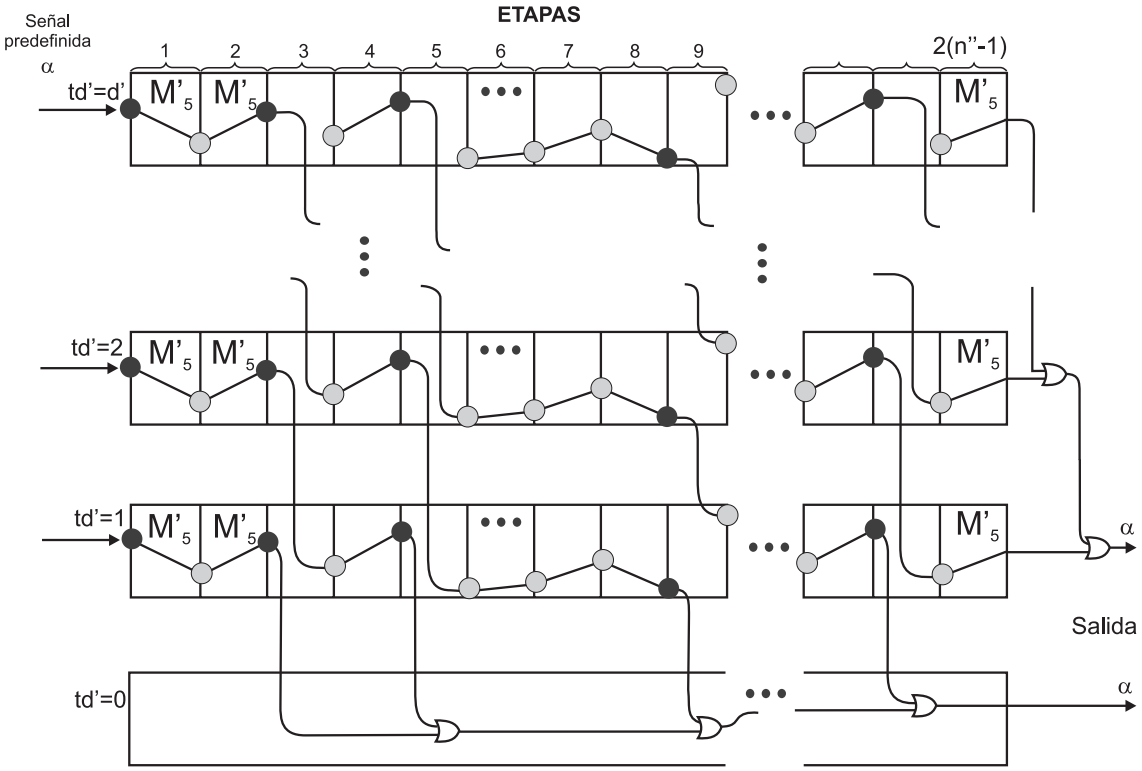

Figura 31. Malla M'_6. tour de Euler sobre el LR-Mesh.

Pseudo código *tour_de_Euler_en_super-vértices*

Entrada: Se requiere la lista de adyacencias de cada vértice para cada una de las diversas mallas.

Salida: Si la cantidad de super-vértices que componen al árbol al que pertenece el super-vértice cv' es menor o igual a la cantidad permitida para el nivel de recursión actual, $2(n'' - 1)$, la señal predeterminada α fluye a través de las mallas y sale por un procesador indicando dicha condición. Por el contrario, si la cantidad de vértices es mayor a $2(n'' - 1)$, la señal predefinida α fluye hacia otro nivel de recursión posterior,

siguiendo un procedimiento similar al que se usa en la Sección V.4.

Cantidad de procesadores requeridos: Utilizando mallas con procesadores tipo OR se tiene lo siguiente. La malla M_4' (ver Figura 29) consiste de $O(d')$ mallas tipo M_2'. Así que el tamaño de la malla M_4' es $O\left((d^2(d')^3N) \times d(d')^2(n'-1)^2(d^2x+d^3)\right)$ procesadores.

La malla M_5' está formada de N mallas tipo M_4' como se ilustra en la Figura 30. Por lo que la malla M_5' requiere un LR-Mesh de $O\left((d^2(d')^3N^2) \times d(d')^2(n'-1)^2(d^2x+d^3)\right)$ procesadores.

Cada renglón de la malla M_6' consiste de $2(n''-1)$ mallas tipo M_5' y existen d' de estos renglones. Por lo tanto el número de procesadores requerido por la malla M_6' es de $O\left((d^2(d')^4N^2) \times d(d')^2(n'-1)^2(n''-1)(d^2x+d^3)\right)$ procesadores.

V.9. Malla para mover al super-vértice cv' a la raíz del árbol de super-vértices

Descripción: Se ejecuta el tour de Euler como se describe en la Sección V.8 para mover cv' a la raíz del árbol de super-vértices. Cada vez que se realiza un paso en el tour de Euler, se revisa si el super-vértice al que se llega, se engancha así mismo (se verifica que sea la raíz del árbol de super-vértices); este procedimiento es similar al que se describe en la Sección V.5, pero considerando la malla M_7' que se ilustra en la Figura 32, en lugar de la malla de la Figura 24. Cuando se encuentra la raíz del árbol de super-vértices, este super-vértice se convierte en el nuevo super-super-vértice cv''.

Pseudo código *Mover_super-vértice_cv_a_la_raíz*

Entrada: Se requiere la lista de adyacencias de cada vértice para cada una de las diversas mallas.

Salida: La señal predefinida α fluye únicamente por una de las n'' posibles salidas,

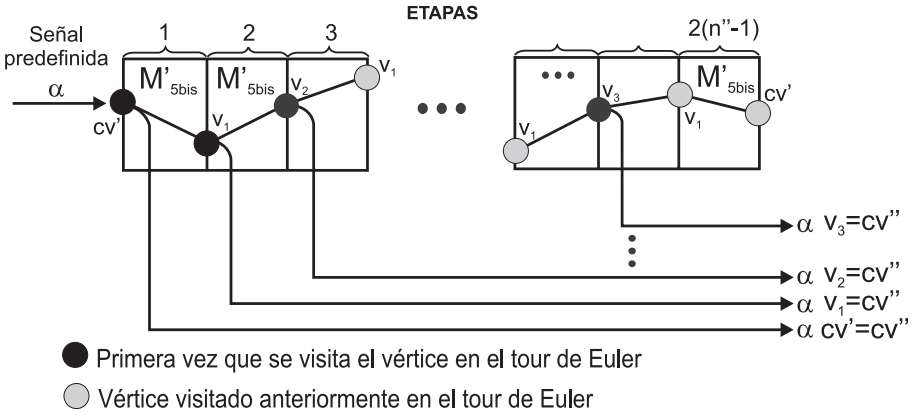

Figura 32. Malla M'_7. tour de Euler sobre el LR-Mesh, construido con base en mallas tipo M'_{5bis}.

que son la cantidad máxima de super-vértices que puede tener el árbol de super-vértices al que pertenece el super-vértice cv' para ese nivel de recursión.

Cantidad de procesadores requeridos: Utilizando mallas con procesadores tipo OR se tiene lo siguiente. La malla M'_7 está constituida por $2(n''-1)$ mallas tipo M'_{5bis}; la malla M'_{5bis} tiene un tamaño equivalente al de la malla M'_5. Por lo tanto, la malla M'_7 requiere $O\left((d^2(d')^3N^2) \times d(d')^2(n'-1)^2(n''-1)(d^2x+d^3)\right)$ procesadores.

V.10. Análisis del algoritmo de la Simulación 2

En el algoritmo de la Simulación 2 del presente capítulo se describen dos tipos distintos de fases de enganchamiento y contracción. El primer par corresponde al grafo original, mientras que el segundo a un grafo de super-vértices. El algoritmo de Trifonov (2005) conecta en cascada un número no constante de este tipo de fases. Note que en la descripción de las fases que se presentan en este capítulo, para toda las mallas siempre ingresa una señal predeterminada α y siempre regresa la misma señal por una única salida. Esta característica de las mallas permite conectarlas en cascada (así como lo hace Trifonov) formando mallas con una cantidad no constante de fases. Como la

propagación de una señal en los ductos de un LR-Mesh se supone constante, entonces la ejecución de los algoritmos en todas estas mallas se lleva a cabo en tiempo constante. Lo anterior implica que el tiempo de ejecución de la Simulación 2 es $O(\log \log N)$ unidades de tiempo. Se conjetura en esta tesis, que se puede diseñar una simulación alternativa que corra en tiempo constante, al conectar en cascada las $O(\log \log N)$ etapas anteriores. Una desventaja de esto, es que el número de procesadores requeridos por el LR-Mesh sería exponencial.

El cálculo del número de procesadores para las mallas anteriores se muestra en forma aproximada. Es muy posible que se puedan simplificar las mallas y disminuir el número de procesadores requeridos. El cálculo de procesadores sin compuertas tipo OR es bastante complejo, por tal razón únicamente se muestra apenas en la Sección V.4. Conjeturamos que el número de procesadores para la Simulación 2 sin usar compuertas tipo OR podría ser exponencial para esta versión del algoritmo; por tal razón, es importante trabajar en la optimización del algoritmo para que dicho número sea polinomial.

Capítulo VI

Conclusiones y trabajo futuro

En el presente trabajo de tesis se proponen dos estrategias para diseñar un algoritmo de simulación de un R-Mesh sobre un LR-Mesh. El algoritmo de la Simulación 1 acota la cantidad de procesadores que se utilizan, sin prestar atención al tiempo de ejecución. Debido a esta restricción, el algoritmo se ejecuta sobre un LR-Mesh de $O(N^2 \times N^2)$ procesadores con un tiempo de ejecución de $O(\log n)$ unidades de tiempo. Por otro lado, el algoritmo de la Simulación 2, acota el tiempo de ejecución de acuerdo al algoritmo de Trifonov (2005), sin prestar atención al número de procesadores. Debido a esta restricción, el tiempo de ejecución de dicha simulación es $O(\log \log n)$ unidades de tiempo.

El objetivo de diseñar la Simulación 1 es tener un punto de partida desde el cual se pudiera ir optimizando paulatinamente el tiempo de ejecución, sin sacrificar en gran medida al número de procesadores. Trifonov saca ventaja del tamaño de los árboles para ahorrar espacio en la máquina de Turing. En esta tesis se conjetura que debe de existir algún mecanismo que permita aprovechar en forma equivalente el tamaño de los árboles para fusionar varias fases de enganchamiento y contracción. Se hace notar que el procedimiento que se empleó para el diseño de esta simulación es completamente independiente del que utilizó el algoritmo de Fernández-Zepeda *et al.* (2002).

El objetivo de diseñar la Simulación 2 es similar al de la Simulación 1, tener un algoritmo de simulación inicial a partir del cual se pudiera ir optimizando paulatinamente el número de procesadores, sin sacrificar el tiempo de ejecución. Se conjetura en este trabajo de tesis que muchas de las mallas que se utilizan se pueden reducir rediseñando

varios de los algoritmos. Incluso se conjetura que es posible incluir fases de depuración de vértices que permitiría una reducción en el número de procesadores en fases posteriores.

Adicionalmente se conjetura que la Simulación 2 posiblemente tenga mayor posibilidad de éxito que la Simulación 1. Esta conjetura se basa en el hecho de que para la Simulación 2 es trivial ver el efecto que tienen árboles relativamente pequeños con respecto a los relativamente grandes. Por lo que se intuye que para la Simulación 2 sí es posible sacar ventaja del tamaño de los árboles.

Como trabajo futuro se plantea diseñar una estrategia que saque ventaja del tamaño de los árboles para optimizar el algoritmo de la Simulación 1 y así poder fusionar un número no constante de fases de enganchamiento y contracción en tiempo constante. También se plantea optimizar varias de las mallas que se emplean en la Simulación 2 con el fin de reducir el número de procesadores.

Bibliografía

Alvarez, C. y Greenlaw, R. (2000). A compendium of problems complete for symmetric logarithmic space. *Computational Complexity*, **9**(2): 123–145 p.

Ben-Asher, Y. y Schuster, A. (1995). The complexity of reconfiguring network models. *Information and Computation*, **121**(1): 41–58 p.

Ben-Asher, Y., Peleg, D., Ramaswami, R. y Schuster, A. (1991). The power of reconfiguration. *Journal of Parallel and Distributed Computing*, **13**(2): 139–153 p.

Bertossi, A. y Mei, A. (2000). A residue number system on reconfigurable mesh with applications to prefix sums and approximate string matching. *Parallel and Distributed Systems, IEEE Transactions on*, **11**(11): 1186–1199 p.

Bokka, V., Gurla, H., Olariu, S. y Schwing, J. L. (1995). Constant-Time convexity problems on reconfigurable meshes. *Journal of Parallel and Distributed Computing*, **27**(1): 86–99 p.

Chao, C., Chen, W. y Chen, G. (1996). Multiple search problem on reconfigurable meshes. *Information Processing Letters*, **58**(2): 65–69 p.

Chong, K. W. y Lam, T. W. (1993). Finding connected components in O(log n loglog n) time on the EREW PRAM. En *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics. Austin, Texas, United States, Mayo 1995. 378–402 p.

Córdova-Flores, C. A., Fernández-Zepeda, J. A. y Bourgeois, A. G. (2007). Constant time simulation of an r-mesh on an lr-mesh. En *IPDPS*, 1–8 p.

Fernández-Zepeda, J. A., Vaidyanathan, R. y Trahan, J. L. (2002). Using bus linearization to scale the reconfigurable mesh. *J. Parallel Distrib. Comput.*, **62**(4): 495–516 p.

JáJá, J. (1992). *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc. United States 576 p.

Jang, J.-W. y Prasanna, V. K. (1995). An optimal sorting algorithm on reconfigurable mesh. *J. Parallel Distrib. Comput.*, **25**(1): 31–41 p.

Koucký, M. (2002). Universal traversal sequences with backtracking. *J. Comput. Syst. Sci.*, **65**(4): 717–726 p.

Lewis, H. R. y Papadimitriou, C. H. (1982). Symmetric space-bounded computation. *Theoretical Computer Science*, **19**(2): 161–187 p.

Li, H. y Maresca, M. (1989). Polymorphic-torus network. *Computers, IEEE Transactions on*, **38**(9): 1345–1351 p.

Li, H. y Stout, Q. (1991). Reconfigurable SIMD massively parallel computers. *Proceedings of the IEEE*, **79**(4): 429–443 p.

Miller, R., Prasanna-Kumar, V. K., Reisis, D. I. y Stout, Q. F. (1993). Parallel computations on reconfigurable meshes. *IEEE Trans. Comput.*, **42**(6): 678–692 p.

Navarro, G. (2009). Teoría de la computación, (lenguajes formales, computabilidad y complejidad), apuntes y ejercicios. Reporte técnico, Departamento de Ciencias de la Computación, Universidad de Chile.

Piatt, J., Cristianini, N. y Shawe-Taylor, J. (2000). Large margin dags for multiclass classification. *Advances in Neural Information Processing Systems*, **12**(3): 547–553 p.

Reingold, O. (2005). Undirected ST-connectivity in log-space. En *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. Baltimore, MD, USA. 376–385 p.

Sipser, M. (1996). *Introduction to the Theory of Computation*. International Thomson Publishing. primera edición. 27–29 p.

Trahan, J. L., Vaidyanathan, R., y Thiruchelvan, R. K. (1996). On the power of segmenting and fusing buses. *Journal of Parallel and Distributed Computing*, **34**(1): 82–94 p.

Trifonov, V. (2005). An o(log n log log n) space algorithm for undirected st-connectivity. En *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. Baltimore, MD, USA. 626–633 p.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, **42**(2): 230–265 p.

Vaidyanathan, R. y Trahan, J. L. (2004). *Dynamic Reconfiguration: Architectures and Algorithms (Series in Computer Science (Kluwer Academic/Plenum Publishers).)*. Plenum Publishing Co.

Wang, B.-F. y Chen, G.-H. (1990). Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. *Parallel and Distributed Systems, IEEE Transactions on*, **1**(4): 500–507 p.