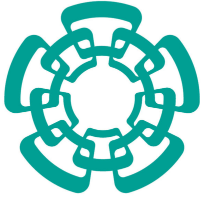

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Cinvestav Unidad Tamaulipas

Algoritmos paralelos para la propagación de etiquetas en redes complejas

Tesis que presenta:

Eder Sayd Camacho Camacho

Para obtener el grado de:

**Maestro en Ciencias
en Ingeniería y Tecnologías
Computacionales**

Dr. Miguel Morales Sandoval, Co-Director
Dr. Arturo Díaz Pérez, Director

Cd. Victoria, Tamaulipas, México | Diciembre, 2019

© Derechos reservados por
Eder Sayd Camacho Camacho
2019

La tesis presentada por Eder Sayd Camacho Camacho fue aprobada por:

____

Dr. Iván López Arévalo

Dr. José Luis González Compeán

Dr. Miguel Morales Sandoval, Co-Director

Dr. Arturo Díaz Pérez, Director

Cd. Victoria, Tamaulipas, México, 17 de Diciembre de 2019

A mi familia, amigos y profesores

Agradecimientos

- Al Dr. Arturo Díaz Peréz, por su atención, consejos y guía durante el desarrollo de esta tesis.
- Al Dr. Miguel Morales Sandoval, por su apoyo constante.
- Al Centro de Investigación y de Estudios Avanzados, Unidad Tamaulipas y toda su comunidad.
- Al Consejo Nacional de Ciencia y Tecnología por financiar esta tesis.
- A mi familia por apoyarme durante todo este trayecto.

Índice General

Índice General i

Índice de Figuras iii

Índice de Tablas v

Índice de Algoritmos vii

Resumen ix

Abstract xi

1. Introducción 1
1.1. Antecedentes . . . 1
1.2. Motivación . . . 3
1.3. Planteamiento del problema . . . 6
1.4. Objetivos . . . 8
1.4.1. Objetivos específicos . . . 8
1.5. Metodología . . . 9
1.6. Organización del documento de tesis . . . 10

2. Fundamentos y estado del arte 13
2.1. Arquitecturas paralelas . . . 13
2.1.1. Arquitecturas Multicore . . . 14
2.1.2. Arquitecturas basadas en GPUs . . . 15
2.2. Modelos de programación para memoria compartida . . . 16
2.2.1. Memoria global compartida en arquitecturas multicore . . . 16
2.2.2. Jerarquía de memoria en GPUs . . . 17
2.3. Estrategias para la construcción de algoritmos paralelos . . . 18
2.3.1. Particionamiento para arquitecturas multicore . . . 18
2.3.2. Patrones de diseño en GPUs . . . 19
2.3.3. Redes Complejas . . . 23
2.3.4. Definición . . . 23
2.3.5. Propiedades . . . 23
2.3.6. Métricas importantes . . . 24
2.4. Núcleo básico para recorrido en amplitud de grafos . . . 25
2.5. Algoritmos para detección de comunidades . . . 26
2.6. Algoritmos para difusión de información . . . 28

3. Algoritmos paralelos para la de detección de comunidades **31**
3.1. Introducción 31
3.2. Algoritmo LP secuencial 32
3.3. Algortimo LP multicore 36
3.4. Algoritmo LP GPU 38

4. Algoritmos paralelos para el proceso de difusión **45**
4.1. Introducción 45
4.2. Algoritmo de difusión secuencial 46
4.3. Algoritmo de difusión multicore 48
4.3.1. Estrategia de particionamiento por comunidad 49
4.3.2. Estrategia de particionamiento por vértice 51
4.4. Algoritmo para la difusión de etiquetas GPU 54

5. Algoritmos paralelos para el proceso de influencia **59**
5.1. Introducción 59
5.2. Algoritmo de influencia secuencial 60
5.3. Algoritmo de influencia multicore 64
5.4. Algoritmo de influencia GPU 66

6. Experimentación y resultados **71**
6.1. Datos de configuración 72
6.1.1. Algoritmos para la detección de comunidades 72
6.1.2. Algoritmos para la propagación de etiquetas por difusión 73
6.1.3. Algoritmos para la propagación de etiquetas por influencia 74
6.2. Métricas de evaluación 74
6.3. Algoritmos paralelos para la detección de comunidades 77
6.4. Algoritmos paralelos para la propagación de etiquetas por difusión 80
6.5. Algoritmos paralelos para la propagación de etiquetas por influencia 83
6.6. Caso de estudio 85
6.6.1. Representación de políticas de control de acceso 87
6.6.2. Reducción de funciones booleanas 88
6.6.3. Conversión de representación 92
6.6.4. Resultados 97

7. Conclusiones y trabajo futuro **101**
7.1. Resumen 101
7.2. Algoritmos para la detección de comunidades 103
7.3. Algoritmos de difusión de etiquetas 103
7.4. Algoritmos de fusión de políticas 105
7.5. Contribuciones 106
7.6. Limitaciones 107
7.7. Trabajo futuro 107

Índice de Figuras

1.1. Resultados esperados para cada una de las etapas definidas en la metodología. . . . 10

2.1. Diagrama de los modelos de memoria en arquitecturas paralelas MIMD. 16
2.2. Organización de la memoria en arquitecturas GPU. 17
2.3. Distintos métodos de particionamiento. 19
2.4. Funcionamiento de la suma por reducción sobre un conjunto de valores numéricos. . 20
2.5. Funcionamiento de la suma de prefijos sobre un conjunto de valores numéricos. . . . 22
2.6. Recorrido en amplitud. 25

3.1. Comunidades detectadas por medio de Label propagation a través de las iteraciones. 34
3.2. Métodos de actualización de etiqueta síncrono y asíncrono. 36
3.3. Distribución de trabajo por vértice y accesos a memoria. 37
3.4. Etapas de procesamiento para el algoritmo LP GPU. 39

4.1. Casos especiales de comunidades donde los vértices difusores se encuentran resaltados de los no difusores. 46
4.2. Entradas y salida del algoritmo de difusión. 47
4.3. Caso que presenta una condición de carrera para dos threads. 52
4.4. Funcionamiento del proceso de difusión sobre un conjunto de vértices. 55

5.1. Proceso de influencia 61
5.2. Proceso para el cálculo de la influencia. 63

6.1. Tiempo de ejecución para el algoritmo LP multicore. 78
6.2. Tiempo de ejecución para el algoritmo LP GPU. 78
6.3. Throughput para el algoritmo LP multicore. 79
6.4. Throughput para el algoritmo LP GPU. 79
6.5. Aceleración algoritmo LP Multicore 80
6.6. Aceleración algoritmo LP GPU 80
6.7. Tiempo de ejecución para el algoritmo de Difusión multicore 81
6.8. Tiempo de ejecución para el algoritmo de difusión GPU. 81
6.9. Throughput para el algoritmo de Difusión multicore. 82
6.10. Throughput para el algoritmo de difusión GPU. 82
6.11. Aceleración algoritmo de difusión multicore. 83
6.12. Aceleración algoritmo de difusión GPU. 83
6.13. Tiempo de ejecución para el algoritmo de influencia multicore 84
6.14. Tiempo de ejecución para el algoritmo de influencia GPU. 84
6.15. Throughput para el algoritmo de influencia multicore. 84
6.16. Throughput del algoritmo de influencia GPU. 84
6.17. Aceleración algoritmo de influencia multicore. 85

6.18. Aceleración algoritmo de influencia GPU. 85
6.19. Ejemplo de variable de una función booleana. 87
6.20. Reducción de la capa tres perteneciente a los vértices terminales. 89
6.21. Reducción y simplificación de los vértices de la capa dos. 90
6.22. Simplificación de los vértices de la capa uno. 91
6.23. Asignación de identificador al vértice raíz y finalización del recorrido. 92
6.24. Extracción de función textual desde la representación de grafo acíclico. 94
6.25. Resultados para el proceso de reducción de funciones en tres etpas: Representación inicial, Reducción, Reversión. 98

Índice de Tablas

2.1. Aspectos generales de trabajos en el estado arte relacionados con esta propuesta de tesis. 30

6.1. Propiedades de los conjuntos de datos utilizados 72

Índice de Algoritmos

1. Kernel suma de prefijos . . . 22
2. Algoritmo BFS . . . 26
3. Label propagation secuencial . . . 35
4. Algoritmo LP multicore . . . 38
5. Kernel cálculo de frecuencia . . . 42
6. Kernel cálculo de comunidad más frecuente . . . 43
7. Algoritmo de difusión secuencial . . . 49
8. Algoritmo de difusión multicore: Distribución por comunidad . . . 51
9. Algoritmo de difusión multicore: Orquestación de threads . . . 52
10. Algoritmo de difusión multicore: Distribución por vértice . . . 53
11. Algoritmo de difusión multicore: Orquestación de threads . . . 54
12. Algoritmo de difusión GPU . . . 58
13. Algoritmo de influencia secuencial . . . 64
14. Algoritmo de influencia multicore . . . 65
15. Algoritmo de influencia multicore: Orquestación de threads . . . 66
16. Kernel de sumatoria: influencia . . . 67
17. Kernel para obtención de fracciones: influencia . . . 68
18. Kernel de actualizado: influencia . . . 69
19. Algoritmo de reversión de funciones booleanas . . . 94
20. Algoritmo de reducción . . . 96

Resumen

Algoritmos paralelos para la propagación de etiquetas en redes complejas

por

Eder Sayd Camacho Camacho
Cinvestav Unidad Tamaulipas
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2019
Dr. Arturo Díaz Pérez, Director
Dr. Miguel Morales Sandoval, Co-Director

Debido a las propiedades estructurales que se presentan en las redes complejas como su estructura en comunidades, distribución de grado en ley de potencia, el fenómeno de mundo pequeño y componentes gigantes, el desarrollo de algoritmos paralelos eficientes para su análisis y procesamiento representa un desafío importante. Dada la baja densidad que tienen los grafos que representan a redes complejas y la gran cantidad de vértices, no es conveniente utilizar representaciones estructuradas para su procesamiento. Sin embargo, las representaciones con listas de adyacencia no permiten aprovechar adecuadamente el procesamiento paralelo debido a la dificultad para tener un balance de carga adecuado, Por otro lado, la diversidad en los grados de los vértices de las redes complejas hace que el procesamiento por listas de adyacencia tenga un pobre desempeño en el uso de la memoria caché. En este trabajo se aborda el problema de propagación de etiquetas utilizando como medio de propagación una red compleja. Este problema representa el núcleo básico de procesamiento para el análisis de redes complejas. El etiquetado de los vértices de la red puede ser de utilidad para realizar segmentación, así como determinar la influencia de los vértices con respecto a otros. Este trabajo se enfoca particularmente en algoritmos de detección de comunidades y difusión de información paralelos, utilizando arquitecturas multicore y GPU. La evaluación de la eficiencia de los algoritmos se realiza con respecto al tiempo de ejecución. Utilizando la aceleración como métrica de comparación entre las diferentes arquitecturas. Obteniendo aceleraciones para redes irregulares de entre 4x y 7x aproximadamente en algoritmos multicore a 10 núcleos y entre 20x y 50x en algoritmos GPU con

448 núcleos CUDA. En este trabajo se muestra también un caso de estudio para darle significado a las etiquetas que se usan en el análisis de redes complejas, viéndolas como políticas de control de acceso CP-ABE en una red de documentos. Se analiza el tratamiento que se podría darle a las etiquetas en el proceso de difusión de etiquetas.

Abstract

Parallel algorithms for label propagation in complex networks

by

Eder Sayd Camacho Camacho
Information Technology Laboratory, CINVESTAV-Tamaulipas
Center for Research and Advanced Studies of the National Polytechnic Institute, 2019
Dr. Arturo Díaz Pérez, Advisor
Dr. Miguel Morales Sandoval, Co-advisor

Due to the structural properties shown in complex networks such as community structure, power law degree distribution, the small world phenomenon and giant components, the development of efficient parallel algorithms for their analysis and processing represents an important challenge. Given the low density of the graphs that complex networks has and the large number of vertices, it is not convenient to use structured representations for processing. However, representations with adjacency lists do not allow adequate use of parallel processing due to the difficulty of having an adequate load balance. On the other hand, the diversity in the complex networks vertices degree causes adjacency list processing to have poor performance on cache memory. This thesis addresses the problem of labels propagation using complex networks. This problem represents the basic processing core for complex network analysis. The vertices labeling of the network can be useful for segmentation, as well as determining the vertices influence with each others. This thesis focuses particularly on community detection and information spreading algorithms, using multicore and GPU architectures. The algorithm efficiency evaluation is carried out with the execution time. Using acceleration as a comparison metric between different architectures. Obtaining accelerations for irregular networks between 4x and 7x in multicore algorithms at 10 cores and between 20x and 50x in GPU algorithms with 448 CUDA cores. This thesis also shows a case study to give meaning to the labels used in the analysis of complex networks, seeing them as CP-ABE access control policies in a network of documents. The treatment that could be given to labels in the process of diffusion of labels is analyzed.

1

Introducción

1.1 Antecedentes

Actualmente las arquitecturas paralelas y masivamente paralelas han tomado un rol importante en el área de las ciencias de la computación, debido a que éstas son esenciales cuando se requieren algoritmos con un alto desempeño [16]. Entre los paradigmas de cómputo paralelo que se utilizan principalmente para acelerar algoritmos o aplicaciones se encuentran, el enfoque multicore, que hace uso de los múltiples procesadores con los que cuentan los equipos de cómputo, y que pueden variar desde unos cuantos hasta varios cientos. Por otro lado se encuentra el enfoque que usa a los aceleradores basados en arreglos de procesadores GPU, los cuales pueden llegar a contar con miles de pequeños procesadores trabajando simultáneamente. Gracias al poder de cómputo que se puede obtener utilizando estos paradigmas se ha podido dar soluciones a problemas complejos o computacionalmente demandantes. Uno de estos problemas es el estudio y análisis de redes complejas, el cual ha llamado la atención de muchos investigadores en los últimos años, esto por las propiedades

estructurales que se encuentran ocultas en las redes, como la distribución de grado en ley de potencias, la propiedad del mundo pequeño y la baja densidad de aristas que provocan que las redes complejas presenten estructuras topológicas diferentes a las encontradas en redes regulares o aleatorias [1, 2, 12].

Las redes complejas pueden definirse como conjuntos de vértices o elementos conectados que interactúan de cierta forma. Estas pueden formarse de diferentes maneras y presentar características únicas dependiendo de como se definan las conexiones entre los vértices. Por ejemplo, en una red social se puede definir a las personas como los vértices de la red y su amistad como la conexión o arista que las une estableciendo una cierta estructura. Pero si la relación entre personas no es la amistad sino las enfermedades que han padecido a lo largo de su vida, la red puede tener una estructura y propiedades totalmente diferentes. Durante el tiempo que se han estudiado las redes complejas se ha observado que éstas pueden formarse a partir de cualquier fenómeno en el que interactúen diferentes entidades. Actualmente con el constante incremento del uso de plataformas sociales, es posible obtener representaciones precisas de las interacciones generadas dando como resultado redes con propiedades especiales que se pueden definir como complejas. Algunas de las redes generadas a partir de este tipo de plataformas pueden alcanzar tamaños en el orden de las decenas de millones de vértices y los miles de millones de aristas, como es el caso para las redes de Friendster con 68,349,466 vértices y 2,586,147,869 aristas y la red de seguidores de Twiter con 41,652,230 vértices y 1,468,365,182 aristas [31].

Considerando que las redes complejas pueden tener varios órdenes de magnitud y que en el mejor caso realizar una operación mínima sobre la red requiere al menos del recorrido de todos sus vértices y aristas, el uso de técnicas de paralelismo para distribuir los recorridos es una idea interesante. Sin embargo, debido a que una de las propiedades de las redes complejas es su baja densidad de aristas, la representación de éstas por medios convencionales como las matrices de adyacencia se encuentran fuera de discusión, debido al uso innecesario de memoria sin mencionar la carga de trabajo adicional. Debido a esto, el uso de listas de adyacencia representa una mejor opción en términos de memoria, a pesar de ello trae problemas de irregularidad debido a la diversidad en las longitudes de las listas,

dificultando el desarrollo de un buen balanceo de carga de trabajo y disminuyendo el aprovechamiento de las memorias cache. Debido a estos problemas se tiene la necesidad de desarrollar algoritmos paralelos que sean capaces de desempeñar tareas de procesamiento en vértices y aristas para acelerar los recorridos y que representen un costo temporal menor al de un algoritmo puramente secuencial, sin dejar de lado el costo espacial para asegurar que los algoritmos ofrezcan la mayor escalabilidad posible.

En este trabajo aborda el problema de la propagación de etiquetas en una red compleja. En el cual se deberá propagar un bloque de información a través de la estructura de la red, donde el bloque de información mínima a propagar será representado por una etiqueta. La propagación de etiquetas en una red se puede entender como un marcado o etiquetado de vértices, el cual es de utilidad para diferentes propósitos, en este caso por ejemplo, la extracción de información acerca de la estructura topológica de la red y su organización, así como el determinar la influencia que tienen los vértices que la conforman.

1.2 Motivación

Con los nuevos avances tecnológicos en el área de la computación y la tendencia observada en la cantidad de procesadores que aparecen en los nuevos equipos de cómputo cada año, la computación paralela o de alto rendimiento se ha convertido en una herramienta importante para la solución de problemas altamente demandantes en cómputo y con una cantidad masiva de datos [29].

Los aceleradores basados en arreglos de procesadores tipo GPU toman un lugar importante en el campo del cómputo de alto rendimiento, debido al descenso en costo observado en los últimos años y su alto poder de cómputo [23]. No solo las arquitecturas basadas en aceleradores han incrementado su capacidad, también las arquitecturas multicore se han visto beneficiadas con mayores cantidades de registros y memoria disponible, así como un incremento en la cantidad de procesadores. En la actualidad, las características básicas de las computadoras de alto rendimiento están disponibles en

la mayoría de los equipos de escritorio modernos y de los servidores no necesariamente especializados. Sin embargo, el diseño de algoritmos paralelos no es una tarea trivial ya que existe una gran variedad de restricciones asociadas directamente a las arquitecturas paralelas que deben tomarse en cuenta, así como las que tienen las propias aplicaciones.

Uno de los grandes desafíos que prevalecen en el cómputo de alto rendimiento es el desarrollo de algoritmos eficientes para problemas con patrones irregulares de acceso a la memoria y que limitan el beneficio del uso de memorias caché rápidas. Tal es el caso de los algoritmos para el análisis de redes complejas, las cuales aparecen en la actualidad en diversas aplicaciones como análisis de redes sociales, análisis de conectividad de redes de transporte, interacciones entre colaboradores de la comunidad académica mundial, por citar solo tres casos. La mayoría de los estudios realizados sobre estos sistemas abarcan un amplio rango de disciplinas en las que se encuentran la biología, economía, física, medicina, neurología y sociología. El interés que se encuentra en las redes complejas surge del descubrimiento de las propiedades ocultas dentro de su propia estructura y en las relaciones que se forman entre sus vértices. Del mismo modo, se ha observado que las redes complejas siguen algunos patrones con diferentes niveles de organización. La densidad de las redes complejas que se generan en fenómenos reales es muy baja, mucho menor a $\mathcal{O}(n^2)$. Por otra parte, las redes tienen una distribución de grados que obedece a lo que se conoce como una ley de potencia, con una fracción pequeña con vértices de muy alto grado y una parte mayoritaria de vértices con grados muy bajos. Finalmente, una propiedad importante de las redes complejas es que tienen el fenómeno del mundo pequeño, en donde la distancia máxima entre cualquier par de vértices es sumamente pequeña comparada con la cantidad de vértices que tiene la red. Lo anterior, provoca que las estructuras de redes complejas sean sumamente irregulares haciendo que el aprovechamiento de la memoria caché sea vea limitado, cuando las redes se están analizando. En [12] se realiza un análisis de las características presentes de las redes complejas basados en el uso de metodologías capaces de exponer las características topológicas más importantes presentes en la red. En dicho artículo se incluyen algunas consideraciones sobre redes complejas así como los modelos principales y los diferentes tipos de métricas existentes.

En [1, 15, 30] se encuentran algunos de los primeros trabajos relacionados con el uso y descripción de las redes complejas y representan un buen punto de partida para el estudio de los sistemas complejos, aunque estos pueden ser definidos de manera simple por algunas de las características principales que los definen:

1. Se componen de múltiples partes que interactúan entre sí.
2. Cada componente esta encargado de llevar a cabo una función específica.
3. El sistema es afectado de manera no lineal por sus componentes.
4. Pueden presentarse comportamientos inteligentes a partir de la interacción entre los elementos más simples de la red(vértices y aristas). Esto se conoce como comportamientos emergentes.

Con el estudio de las redes complejas y de sus propiedades han aparecido una variedad de técnicas para extraer características que puedan llegar a ser de utilidad para diferentes propósitos. Una de estas características son las estructuras de comunidades, en las cuales se forman grupos de vértices altamente conectados entre sí. De modo que toda la red puede dividirse en subredes a partir de las comunidades formadas por la conexiones entre sus vértices. Las comunidades en las redes pueden denotar información valiosa que no se encuentra en las propiedades visibles de la red, si no en su estructura topológica, y que no será revelada hasta que se analicen propiamente las conexiones entre los vértices.

Existen diversos enfoques para realizar la tarea de la búsqueda de comunidades en una red, algunos ejemplos son los algoritmos: Leading Eigen Vector, Edge Betweenness, Label Propagation, Multi-Level, entre algunos otros [11]. Entre estos enfoques uno de los más utilizados es Label Propagation debido a su modo de operación que se basa en el diseminado de etiquetas de manera iterativa a través de la red, de modo que al inicio todos los vértices de la red tienen una etiqueta única la cual cambiará con las iteraciones a la etiqueta con mayor frecuencia en la vecindad de vértices. Debido a su simplicidad existen diversos trabajos que utilizan variantes de este algoritmo

para la detección de comunidades. Una de las variantes más importantes, remueve las decisiones estocásticas (probabilísticas o aleatorias) y las remplaza por decisiones deterministas. Esta propiedad toma importancia cuando se desarrollan algoritmos paralelos sobre aceleradores basados en arreglos de procesadores GPU.

1.3 Planteamiento del problema

Un punto clave en el estudio de redes complejas es el uso de algoritmos especializados para analizarlas y extraer la información relevante que yace en la propia estructura de la red. Sin embargo, se ha observado que las técnicas para el análisis de redes pueden tener un alto costo computacional. Por ejemplo, el análisis de una red compleja usualmente requiere el recorrido de toda la red requiriendo del orden de $\mathcal{O}(n+e)$ pasos, donde n es el número de vértices y e el número de aristas. Sin embargo, para algunas aplicaciones es necesario hacer el recorrido a partir de cada uno de los vértices, por lo que se requieren $\mathcal{O}(n^2 + ne)$ pasos. Ya que usualmente $n < e << n^2$, entonces se requieren algoritmos con complejidad entre $\mathcal{O}(n^2)$ y $\mathcal{O}(n^3)$. Cuando n es de varios órdenes de magnitud, los algoritmos para análisis de redes complejas se vuelven altamente demandantes computacionalmente.

Uno de los principales enfoques que se sigue para reducir los tiempos de análisis, es el pasar de estrategias puramente secuenciales a estrategias multicore o basadas en arreglos de procesadores GPU. Estos enfoques distribuyen la carga computacional requerida para realizar las tareas. Una de las aplicaciones de interés en redes complejas es la búsqueda de islas, subredes o comunidades dentro la red. Al ser ésta una tarea altamente demandante en procesamiento han surgido algunas soluciones multicore que atacan el problema [25, 26]. También se han propuesto algunas soluciones en arquitecturas basadas en arreglos de procesadores GPU [34]. En ambas arquitecturas se trata de aprovechar los beneficios que proporciona el utilizar cada una de ellas, generando soluciones simples y efectivas. En las arquitecturas multicore bajo el modelo de memoria compartida y en el caso de las arquitecturas basadas en arreglos GPU además se debe usar eficientemente la jerarquía de memoria

presente.

En este trabajo se aborda el problema de la propagación de etiquetas a través de estrategias paralelas multicore y GPU en redes complejas. A partir de una red compleja no etiquetada se trata de obtener un grafo etiquetado de acuerdo a la estructura topológica de la red. El problema del marcado de una red para determinar la organización y topología de una red se puede mapear directamente al problema de detección de comunidades, el cual busca determinar los grupos de vértices densamente conectados entre sí, a partir de información limitada solo a los enlaces de la red.

Debido a las propiedades de las redes complejas y las ventajas y desventajas que presentan las arquitecturas paralelas, se plantea la siguiente pregunta de investigación. **¿Es posible definir algoritmos paralelos eficientes en tiempo y espacio sobre arquitecturas multicore y de aceleradores basados en arreglos de procesadores GPU para el etiquetado de redes complejas?**. Tomando en cuenta que la estructura en comunidades de una red puede aportar información organizacional y jerárquica de la red, también es posible obtener modelos de difusión de información a través de las estructuras conformadas por los vértices con mayor grado de conectividad y el alcance de ésta. Por lo que se plante la siguiente pregunta de investigación. **¿Es posible la difusión paralela de etiquetas en redes complejas a partir de un conjunto de vértices influyentes?**. En caso de poder obtener una correcta difusión de etiquetas sobre la red compleja utilizando la estructura etiquetada, puede resultar de interés conocer como afecta la propagación inicial de etiquetas a los vértices y como influye la distribución de éstas sobre los vértices frontera entre etiquetas, de modo que surge la pregunta **¿Es posible determinar la influencia de las etiquetas sobre los vértices a partir de redes complejas etiquetadas?**

Con la ayuda de algoritmos paralelos multicore y GPU para el análisis de redes complejas es posible aprovechar la estructura en comunidades de una red compleja para realizar el marcado de la misma y con base en éste atacar el problema de difusión e influencia de las etiquetas respecto a los vértices de la red.

En resumen, se plantea el desarrollo de esta propuesta de tesis de maestría bajo la siguiente hipótesis: **Es posible desarrollar algoritmos paralelos multicore y basados en arreglos de procesadores GPU eficientes en tiempo y espacio, que exploten la estructura subyacente de las redes complejas para el etiquetado de los vértices** y hacer una difusión eficiente de información.

Las etiquetas pueden representar cualquier elemento de información o relación entre los vértices. Pueden ser palabras clave en una red colaborativa, temas de interés en una red social o enfermedades infecciosas en una comunidad, por citar algunos ejemplos. En este trabajo se aborda también el caso de estudio en donde las etiquetas pueden representar políticas de control de acceso en una red de documentos o de usuarios. Así, además de estudiar los algoritmos de propagación, se analiza el caso de cuando un vértice está relacionado con más de una política de control de acceso y como se pueden combinar las políticas relacionadas para crear nueva políticas o etiquetas.

1.4 Objetivos

El objetivo general de este trabajo es diseñar algoritmos paralelos para arquitecturas multicore y GPUs que sean capaces de acelerar el proceso de propagación de etiquetas sobre una red compleja.

1.4.1 Objetivos específicos

Con la finalidad de alcanzar el objetivo general descrito se definen los siguientes objetivos específicos.

- Diseñar e implementar algoritmos paralelos multicore y GPU basados en propagación de etiquetas para la detección de comunidades en redes complejas
- Diseñar e implementar algoritmos paralelos multicore y GPU para la difusión de etiquetas y el ajuste de influencia de las mismas a través de la estructura de una red compleja partiendo de

vértices difusores

- Proponer una solución adecuada al uso de etiquetas como políticas de control de acceso en el análisis de la difusión e influencia en una red compleja de documentos.

1.5 Metodología

A continuación se describen los pasos generales de la metodología planteada para alcanzar los objetivos de este trabajo:

1. La primera parte abordada será el diseño e implementación de un algoritmo paralelo para la detección de comunidades sobre una red compleja. En esta parte se deberá profundizar en la literatura para analizar las diferentes propuestas de algoritmos paralelos que atacan el problema de la detección de comunidades utilizando estrategias de marcado o etiquetado, así como los resultados que se obtienen con estos enfoques. Posteriormente se realizará el diseño, implementación y evaluación para arquitecturas multicore y otra para GPUs de las estrategias seleccionadas.

2. Como segunda parte de este trabajo se plantea diseñar e implementar un algoritmo paralelo para la difusión de etiquetas tomando como base la red compleja etiquetada por la detección de comunidades. Para la difusión de información sobre redes complejas, se utilizarán estrategias basadas en vértices influyentes. Una vez revisado el estado del arte, se procederá al diseño e implementación de un algoritmo multicore para propagación de etiquetas y se desarrollará también una versión para GPU del algoritmo de propagación.

3. En tercer lugar se diseñará y desarrollará un algoritmo para el refinamiento de la asignación inicial de etiquetas propuesto por el algoritmo de difusión. Para este caso se realizarán dos implementaciones, una sobre arquitecturas multicore y otra para GPU.

4. Por último se deberá analizar el problema para la aplicación de políticas de control de acceso como etiquetas en los procesos de difusión e influencia en redes de documentos. Se analizará como resolver la fusión y simplificación de políticas de control de acceso en el proceso de etiquetado.

La Figura 1.1 muestra de manera general el proceso de desarrollo de este trabajo, donde se presentan los resultados esperados de cada etapa.

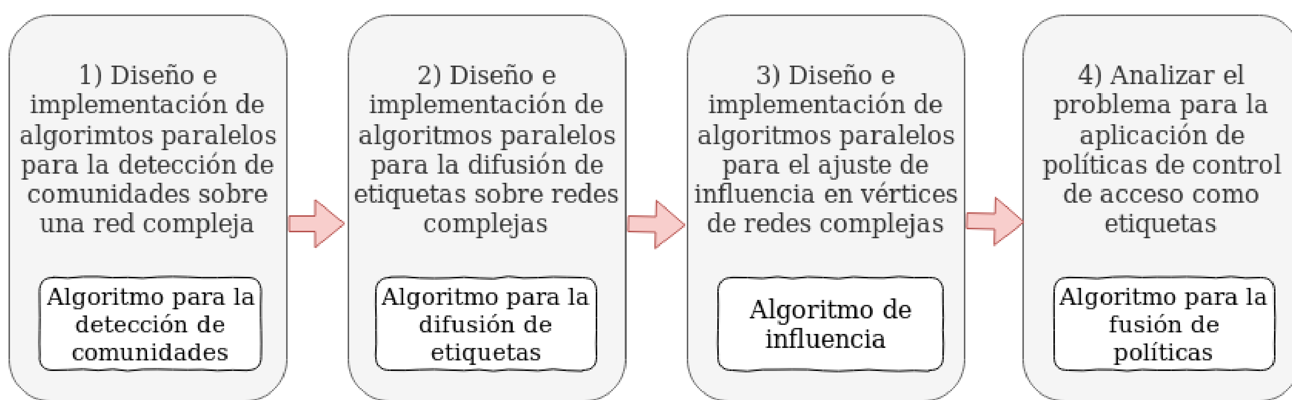

Figura 1.1: Resultados esperados para cada una de las etapas definidas en la metodología.

1.6 Organización del documento de tesis

A continuación se presenta una breve descripción del contenido de los siguientes capítulos.

- En el Capítulo 2 se presentan los fundamentos y el marco teórico utilizado en este trabajo relacionados con arquitecturas paralelas, modelos de programación paralela y redes complejas.
- En el Capítulo 3 se describen los algoritmos secuencial, multicore y GPU para la detección de comunidades en una red compleja.
- En el Capítulo 4 se describen los algoritmos secuencial, multicore y GPU para la difusión de etiquetas en una red compleja.

- En el Capítulo 5 se describen los algoritmos secuencial, multicore y GPU para la influencia de etiquetas en una red compleja.
- En el Capítulo 6 se presentan los resultados principales para los algoritmos de detección de comunidades, difusión, influencia y el caso de estudio de difusión de políticas como etiquetas.
- En el Capítulo 7 se describen las conclusiones principales resultado de este trabajo. Así como las posibles aplicaciones en trabajo futuro.

2

Fundamentos y estado del arte

En este Capítulo se presentan los fundamentos teóricos de los temas abarcados en este trabajo como lo son: Las arquitecturas paralelas y sus modelos de memoria. La construcción de algoritmos paralelos. Redes complejas y sus propiedades. Algoritmos para la detección de comunidades y para la difusión de información.

2.1 Arquitecturas paralelas

El paralelismo se define como la capacidad para realizar trabajo en la misma unidad de tiempo. En el contexto de la computación se puede entender como la división de una tarea en pequeñas tareas que integrándolas después de un procesamiento individual se obtiene un resultado completo.

Un algoritmo puede definirse como una serie de pasos ordenados con el único objetivo de resolver un problema. En algunas ocasiones el algoritmo requerirá de información adicional para llevar a cabo su tarea, la cual será manipulada de alguna manera para obtener un resultado o de otro modo realizando modificaciones sobre la misma. Por otra parte un algoritmo paralelo es aquel en el cual

los pasos a realizar pueden dividirse y ser evaluados en el mismo instante de tiempo para después unirse nuevamente y obtener un resultado correcto equivalente al algoritmo secuencial.

A diferencia de los algoritmos secuenciales que desde el inicio son pensados para ejecutarse sobre una unidad de procesamiento, los algoritmos paralelos deben diseñarse para ejecutarse sobre N unidades de procesamiento permitiendo la escalabilidad del algoritmo en arquitecturas de computadoras con diferentes especificaciones [16]. Un algoritmo paralelo usualmente es ejecutado como una instancia de la mismo conjunto de instrucciones con diferentes parámetros y en una unidad de procesamiento diferente. Al conjunto de instrucciones y datos se le conoce como thread (hilo) que al finalizar su ejecución obtendrá un valor resultado parcial que unido con el resto de los resultados parciales de otros threads formarán un resultado correcto y completo.

Uno de los principales requisitos para diseñar y desarrollar algoritmos paralelos eficientes es y ha sido desde los inicios del cómputo paralelo el conocimiento de la plataforma de hardware y los conceptos claves del paralelismo. Esto debido a que el modelo de programación paralela se aleja de la idea de que el hardware debe acoplarse al software y plantea un nuevo enfoque para el diseño de soluciones pensadas para ser paralelas sobre una arquitectura especifica. Dentro de los principales paradigmas de cómputo paralelo se encuentran las arquitecturas **multicore** y las arquitecturas basadas en arreglos de procesadores **GPU**.

2.1.1 Arquitecturas Multicore

Son arquitecturas del tipo MIMD, que constan de múltiples unidades de procesamiento independientes con capacidad de acceder a múltiples flujos de datos, que a diferencia de las arquitecturas SIMD que ejecutan la misma instrucción en un instante de tiempo(lo que las vuelve síncronas), las MIMD pueden ejecutar diferentes instrucciones en el mismo instante por lo que son consideradas asíncronas [17].

El desarrollo de algoritmos paralelos sobre arquitecturas multicore en la mayoría de los casos plantea un problema de división de trabajo, por lo que generalmente a los algoritmos desarrollados

bajo esta arquitectura se les conoce como soluciones de grano grueso.

Una solución paralela de grano grueso es aquella en la que una tarea es repartida entre p procesadores diferentes, el procesamiento que se lleva a cabo por cada uno de los procesadores suele ser un algoritmo secuencial, en ocasiones con ligeras modificaciones. En estos casos la ganancia que se obtiene al repartir el trabajo estará determinada por el procesador con mayor carga de trabajo, por esa razón mientras más precisa sea la división de trabajo mayor será la ganancia.

2.1.2 Arquitecturas basadas en GPUs

Las arquitecturas basadas en arreglos de procesadores GPU, son arquitecturas basadas en un modelo de procesamiento SIMD. En el cual existen múltiples procesadores que ejecutan instrucciones de manera síncrona sobre diferentes flujos de datos. Las aplicaciones paralelas GPU, normalmente deben desarrollarse específicamente para cada problema, lo que implica que paralelizar un algoritmo bajo este modelo es sumamente complejo ya que se requiere un análisis completo del algoritmo y la descomposición de éste en sus operaciones más básicas. Por esta razón, los algoritmos GPU son considerados como soluciones de grano fino.

Para el desarrollo de soluciones GPU generalmente se utiliza la plataforma CUDA. La cual brinda un entorno para la orquestación de los procesadores de las tarjetas. En ésta es posible desplegar una cantidad variada de configuraciones de threads.

El primer nivel de configuración lo determina el llamado **grid** o malla, el cual tiene la capacidad de albergar de manera bidimensional(x, y) una cierta cantidad de **bloques** determinada por la propia arquitectura.

El segundo nivel de configuración lo determinan el **block** o bloque, en el cual es posible desplegar como máximo **1024** threads sobre tres dimensiones x, y, z.

Utilizando las posibles configuraciones de organización de threads es posible abordar problemas afines a estas configuraciones. Del mismo modo aplicar optimizaciones a nivel de memoria ayuda a incrementar el rendimiento de las aplicaciones.

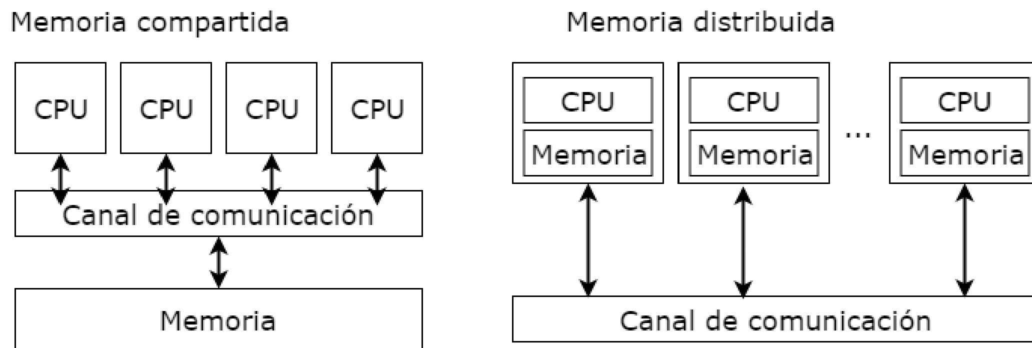

Figura 2.1: Diagrama de los modelos de memoria en arquitecturas paralelas MIMD.

2.2 Modelos de programación para memoria compartida

A pesar de que existen diferentes arquitecturas paralelas y diferentes modelos de memoria para ellas, el problema al que se enfrenta cuando se realizan algoritmos paralelos es la división de las tareas, ya que el rendimiento de la solución estará limitado a la unidad de procesamiento con mayor carga de trabajo.

2.2.1 Memoria global compartida en arquitecturas multicore

Dentro de las arquitecturas MIMD existen dos enfoques que utilizan modelos de memoria diferentes. En primer caso se encuentran los sistemas de memoria compartida en la cual todas las unidades de procesamiento acceden a un bloque común de memoria lo que permite el intercambio de información entre las unidades, aunque esto puede conllevar problemas si más de una unidad intenta modificar la misma región de memoria simultáneamente. Por otra parte están los sistemas que tienen un modelo de memoria distribuida la cual consta de unidades de procesamiento y memoria independientes lo que permite obtener mayor velocidad en los accesos a memoria para las unidades de procesamiento pero limita la comunicación entre las unidades, por lo que la sincronización puede resultar un problema. En la Figura 2.1 se muestra un diagrama de los dos modelos de memoria compartida y distribuida.

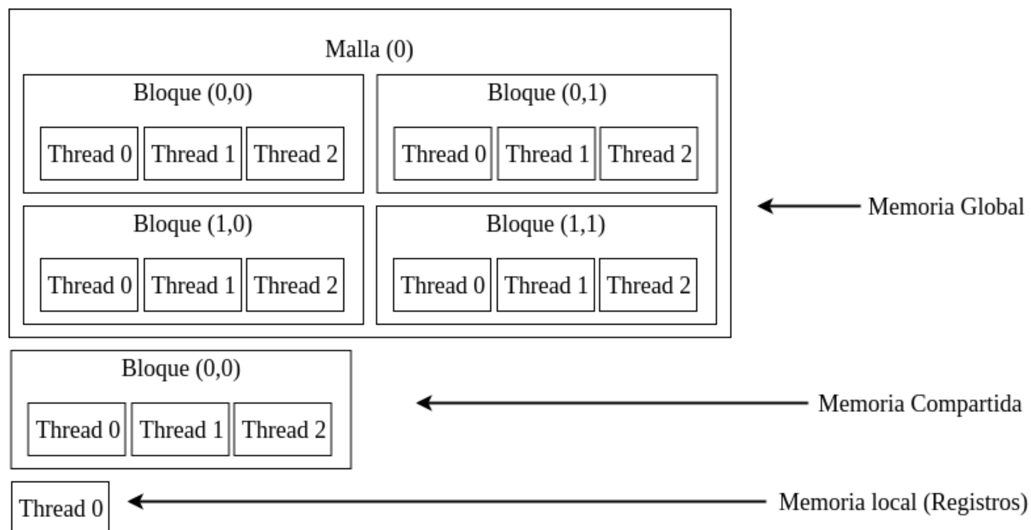

Figura 2.2: Organización de la memoria en arquitecturas GPU.

2.2.2 Jerarquía de memoria en GPUs

En las arquitecturas GPU existen diferentes niveles memoria, principalmente se encuentra la memoria global(global memory), la cual puede ser accedida desde cualquier thread desplegado independientemente del bloque en que se encuentre. El principal inconveniente que presenta la memoria global, es el tiempo de acceso, ya que es aproximadamente 100 veces más lenta que la memoria compartida y en lo posible debe evitarse su uso [3].

Por otra parte se encuentre la memoria compartida(shared memory) la cual solo puede ser accedida por los threads que pertenecen al mismo bloque y que al contrario de la memoria global es de acceso más eficiente. Pero a pesar de esta ventaja la memoria compartida suele ser bastante limitada, por lo que los problemas deben pensarse tomando en cuanta esta restricción.

Finalmente se encuentra la memoria local o registros, que son las unidades de memoria disponibles únicamente para cada thread. Estos suelen ser usados cuando el procesamiento es independiente y no se requiere de valores compartidos, obteniendo generalmente como costo máximo una lectura y escritura a memoria global, mientras que el resto del procesamiento se realiza sobre un registro altamente eficiente. La jerarquía de memoria GPU se presenta en la Figura 2.2.

2.3 Estrategias para la construcción de algoritmos paralelos

Dentro de cada plataforma o arquitectura paralela se pueden encontrar algunas técnicas que explotan las características de la arquitectura, convirtiéndose en los bloques de construcción básicos para el desarrollo de algoritmos paralelos sobre éstas.

2.3.1 Particionamiento para arquitecturas multicore

Entre algunos de los métodos utilizados para dividir la carga de trabajo en aplicaciones paralelas multicore se encuentran principalmente los siguientes:

1. Particionamiento Estático. Funciona principalmente para conjuntos de datos regulares, bajo este enfoque se define una cantidad de elementos que deberá procesar cada unidad de procesamiento. Es decir, que para una entrada de datos con m elementos cada thread procesará una fracción contigua de $\frac{m}{p}$ elementos. Siendo p la cantidad de procesadores.

2. Particionamiento Cíclico. En este enfoque se requiere de un índice i para cada procesador, ej. $i \in \{0, 1, 2, ...p-1\}$. Siguiendo con el ejemplo anterior de división de trabajo a cada procesador se le asigna el vértice que coincide con su índice, esto quiere decir que el procesador **0** se le asignará el trabajo del vértice **0**, para el procesador **1** el trabajo del vértice **1** y del mismo modo hasta llegar a $p-1$. Cuando los procesadores terminan de realizar el trabajo del vértice asignado prosiguen por el trabajo del vértice $i+p$ con i igual al índice de cada procesador. De esta manera el índice de los procesadores se actualiza cada vez que terminan con el trabajo de un elemento. Al organizar la división de trabajo de esta forma se espera se minimice la cantidad de elementos con carga de trabajo irregular.

3. Particionamiento Dinámico. Utilizado para abordar problemas con carga irregular de trabajo.

A diferencia de los anteriores métodos de particionamiento anteriores que desde el inicio se sabe exactamente cuales serán los elementos que deberá procesar cada procesador, en el particionamiento **dinámico** existe un índice de elemento compartido, entre los p procesadores. Todos los procesadores acceden con exclusión mutua al índice tomando el elemento relacionado a éste e incrementándolo para que el siguiente procesador tome el siguiente elemento. Esto ocurre para hasta que se ha completado la totalidad de elementos a procesar.

En la Figura 2.3 se resumen los métodos de particionamiento y como se configuran sobre un conjunto de datos de entrada para una solución paralela.

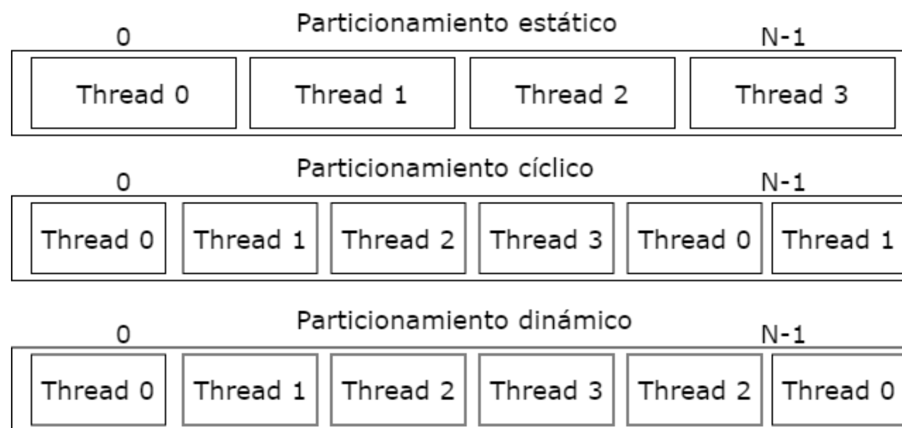

Figura 2.3: Distintos métodos de particionamiento.

2.3.2 Patrones de diseño en GPUs

El diseñar aplicaciones para GPUs suele tener una dificultad elevada, ya que requiere de un amplio conocimiento de la arquitectura y su funcionamiento para aprovechar correctamente la capacidad de cómputo paralelo masivo que ofrece. Para obtener aplicaciones GPU eficientes han surgido una serie de algoritmos conocidos como primitivas paralelas que siguen una estrategia común y se pueden utilizar para distintas aplicaciones. Algunas de las más utilizados son la **reducción** y la **suma de prefijos**.

La reducción es una operación paralela que es normalmente utilizada como un método de suma para un conjunto de datos numéricos, aunque puede ser generalizada para cualquier operación binaria con propiedad asociativa. En primer lugar asumiendo que se tiene una cantidad p de threads y q valores numéricos con $p, q \in \{2^0, 2^1, ..., 2^k\}$ y $p = \frac{q}{2}$ para realizar la reducción paralela cada uno de los p threads se alineara con forme a su índice a los primeros $\frac{q}{2}$ valores, de modo que al procesador i le corresponde el valor i.

El siguiente paso es sumar todos los valores $i < p$ con los $i + p$, al ser p la mitad de q provoca que se sumen la mitad del total de los valores. Terminando con la división por 2 de p una vez se han realizado todas las sumas con el fin de que en la siguiente ronda se sumen los resultados obtenidos en los primero índices y terminado cuando se realice la última suma de los índices 0 y 1. Obteniendo el resultado de la suma de los valores numéricos sobre el índice 0.

El comportamiento de la reducción paralela tiene un costo computacional de $O(log_2(q))$, pero que puede verse afectado por implementación realizada. En la Figura 2.4 se muestra el comportamiento de una reducción paralela sobre un conjunto de valores numéricos para la operación aditiva.

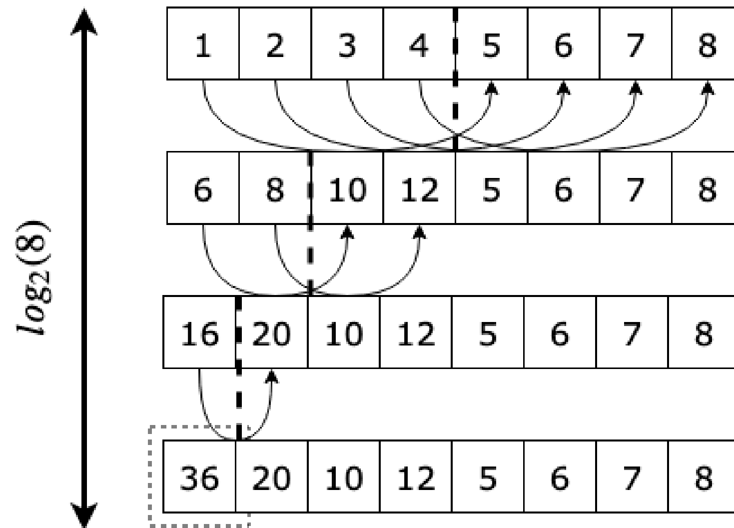

Figura 2.4: Funcionamiento de la suma por reducción sobre un conjunto de valores numéricos.

La suma de prefijos por otra parte, es una operación fundamental que al igual que la reducción

obtiene la suma de un conjunto de valores numéricos. Aunque a diferencia de la reducción ésta obtiene el valor de la suma de todos los valores anteriores al índice i del conjunto. Por ejemplo el resultado de aplicar la suma de prefijos sobre los valores $[1, 2, 3, 4, 5]$ sería $[1, 3, 6, 10, 15]$. Esta operación es importante y bastante utilizada en operaciones de desplazamiento de memoria, agrupamiento y ordenamiento.

El funcionamiento de la suma de prefijos es de alguna forma similar a la reducción, pero en este caso no es descartada la mitad de los threads por iteración. Retomando el ejemplo anterior de p threads y q valores numéricos, esta vez con $q \in \{1, 2, 3, ..., k\}$ y $p = q - 1$, para realizar la suma de prefijos se asignan los primeros $q - 1$ valores a cada uno de los threads i. El siguiente paso es realizar la suma de los valores i con los valores $i + m$, con $m = 1$ en la primera iteración e incrementando m en $m \cdot 2$ con cada una, esto se repetirá mientras $m \leq q$ y los resultados de cada suma se acumularán sobre los índices $i + m$.

Las iteraciones necesarias se encuentran en el orden de $O(log(q))$ al igual que la reducción, pero en este caso cada uno de los índices de los threads tendrá el valor de la suma de todos los valores anteriores, mientras que en la última posición se encontrará el total de la suma.

Una de los principales usos para la suma de prefijos es el agrupamiento de datos. Utilizando un predicado como base se seleccionan los elementos que cumplen con éste para posteriormente identificar el índice que debería tener cada uno si estos fueran contiguos.

Un ejemplo simple sería, dado los valores $[1, 2, 3, 4, 5, 6]$ agrupar aquellos que sean pares. Para ello el primer paso es aplicar el predicado con el cual se obtienen aquellos valores que cumplen y se indican con un **1** y los que no con un **0**, de este modo el resultado sería $[0, 1, 0, 1, 0, 1]$ y aplicando la suma de prefijos se obtienen los índices correspondientes al grupo de valores que cumplen el predicado $[0, 1, 1, 2, 2, 3]$. En la Figura 2.5 se muestra el mecanismo para el cálculo de la suma de prefijos.

En el Algoritmo 2.5 se presenta el kernel de CUDA para la suma de prefijos, en el cual se define una variable de desplazamiento log la que índica en que posición de índice se guardaran los resultados de las sumas parciales. Inicialmente el valor de log será **1** y se incrementará en $log \cdot 2$ con cada

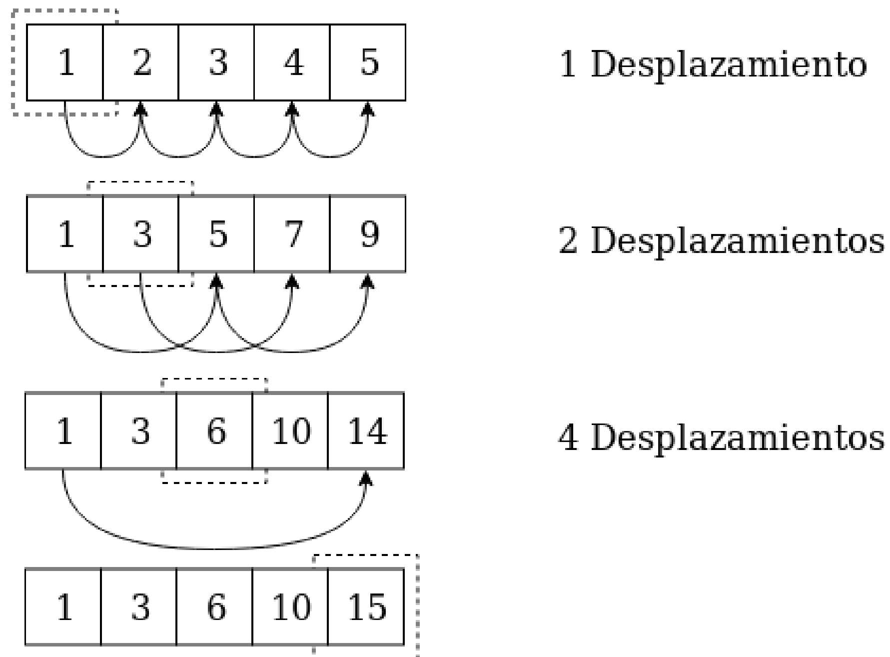

Figura 2.5: Funcionamiento de la suma de prefijos sobre un conjunto de valores numéricos.

iteración. La cantidad de iteraciones estará determinado por $log_2(q)$, con q como la cantidad total de datos. Debido a que las sumas parciales en cada iteración pueden interferir con los resultados de otros threads se utiliza la estructura auxiliar R' en la cual se escriben los resultados, para después de todas las escrituras copiarse al original R.

Algoritmo 1 Kernel suma de prefijos

1: **kernel** $prefix(R, R', log, q)$
2: $index \leftarrow blockIdx.x * blockDim.x + blockIdx.x$
3: **while** $index < q$ **do**
4: $R'[index + log] \leftarrow R[index] + R[index + log]$
5: $index \leftarrow index + gridDim.x * blockDim.x$
6:
7: **for** $i \leftarrow 1$ **to** q **do**
8: $prefix(R, R', i)$
9: $R \leftarrow R'$
10: $i \leftarrow i * 2$

2.3.3 Redes Complejas

2.3.4 Definición

Una red compleja se define como un grafo $R = (\mathcal{N}, \mathcal{E})$ donde $\mathcal{N} = \mathcal{N}(R)$ es el conjunto de elementos denominados nodos o vértices y otro conjunto, $\mathcal{E} = \mathcal{E}(R) \subset \mathcal{N} \times \mathcal{N}$ de elementos denominados aristas [12]. Las redes complejas cuentan con una variedad de propiedades entre algunas de ellos se encuentran las redes con propiedad de mundo pequeño, la caracterización de modelos de libre escala y la detección de comunidades, existiendo diversos métodos para el estudio de estas propiedades [4, 19, 36].

2.3.5 Propiedades

Las redes complejas pueden aparecer a partir de cualquier evento en el cual exista una interacción de algún tipo entre sus elementos, debido a esta propiedad las redes complejas pueden surgir prácticamente de cualquier sitio. A través de las propiedades estructurales de las redes complejas es posible extraer información abundante. Entre las propiedades importantes que se han estudiado de las redes complejas se encuentran: la distribución de grados de acuerdo a una ley de potencias, las redes libres de escala que tienen estructuras que se repiten a diferentes niveles o jerarquías, el fenómeno de "mundo pequeño"que indica que a pesar de la gran cantidad de vértices que puede tener una red la separación máxima que hay entre cualquier par de vértices es relativamente baja, y la organización en comunidades. Las redes complejas tienen una gran cantidad de propiedades que pueden ser explotadas. Muchas de ellas son utilizadas para la clasificación de las redes complejas, por ejemplo en [32] se muestra un marco para la construcción de reglas de clasificación basadas en las estructuras de comunidades. Estas reglas permiten establecer una métrica de similaridad con la cual es posible identificar las redes que comparten características en común a partir de una colección de datos obtenidos de votaciones, recursos financieros y redes sociales, por citar algunos ejemplos. La

estructura de comunidades observada frecuentemente en las redes complejas puede tener una amplia variedad de aplicaciones en diferentes dominios como la informática, sociología, química, biología, entre algunos otros. La detección de comunidades sigue planteando un reto importante para las diferentes técnicas existentes [11]. Por esta razón, han surgido una serie de trabajos que buscan lograr mejorar los resultados observados con los métodos clásicos de detección de comunidades, aportando nuevos algoritmos basados en heurísticas, propiedades estructurales y modelos probabilísticos [2, 38]. La mayoría de los métodos para evaluarlos se basan en redes de estructuras conocidas, por lo que no reflejan resultados válidos en entornos reales. La determinación del mejor enfoque para la detección de comunidades se vuelve aún más difícil cuando se lleva hacia las redes complejas ya que estas redes presentan características difíciles de modelar y una limitada cantidad de generadores capaces de sintetizar este tipo de redes dificultando la evaluación de las técnicas de detección de comunidades [22]. Las estructuras de comunidades se vuelvan aún más interesantes cuando se obtienen a partir de elementos que ya son ricos en información, como las inmensas colecciones de datos existentes en la red (ficheros, documentos, vídeo, audio, texto plano, etc).

2.3.6 Métricas importantes

El análisis de las redes complejas es una área de estudio importante para distintas disciplinas, principalmente en las ciencias naturales, sociales y de la información, pero no solo limitada a ellas, por ello existe una variedad de técnicas para el análisis de redes complejas y métricas que brindan soporte. Entre algunas de las medidas utilizadas en redes se encuentran el grado: que indica la cantidad de enlaces que tiene un vértice, el coeficiente de club de ricos: que determina la fracción de enlaces existentes entre los vértices con grado k de una red, medidas de centralidad como la intermediación(betweenness): que determina la importancia de un vértice a partir de la cantidad de caminos en los que participa, el coeficiente de clustering: que permite caracterizar la presencia de ciclos de orden tres, la longitud del camino promedio: que se obtiene de la media de las distancias entre los vértices de la red, entre otros [12].

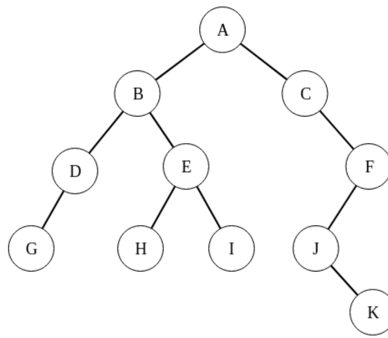

Figura 2.6: Recorrido en amplitud.

2.4 Núcleo básico para recorrido en amplitud de grafos

El recorrido de un grafo puede interpretarse como el proceso mediante el cual se visitan todos los vértices del mismo, a través de sus aristas partiendo desde un vértice inicial.

Existen diferentes algoritmos para realizar búsquedas y recorridos sobre grafos, sin embargo, uno de los más conocidos es el recorrido en amplitud BFS(Breadth First Search) [6]. El cual inicia su recorrido a partir de un vértice inicial y explora cada uno de sus vecinos. Una vez a revisados todos, se marca el vértice como visitado y se selecciona alguno de los vértices visitados como vértice inicial. El procedimiento se repite hasta que todos los vértices se hayan visitado. En la Figura 2.6 se muestra el orden de visita de los vértices usando el algoritmo de recorrido en amplitud.

En el Algoritmo 2 se presenta el mecanismo básico para realizar un recorrido en amplitud. En el cual se inicializa un vector m con el estado de no recorrido para todos los vértices. El algoritmo recibe como entrada una red G y un vértice origen src, de los cuales partirá el procedimiento. Después de la inicialización se pone el vértice src en la cola $queue$. A partir del primer vértice puesto en la cola, se visitarán los vecinos de ese vértice y serán marcados y agregados a ésta. El procedimiento se aplicará mientras la cola contenga elementos.

Algoritmo 2 Algoritmo BFS

```
1: procedure bfs(G, src)
2:     for v ∈ G do
3:         m[v] ← 0
4:     m[src] ← 1
5:     queue.push(src)
6:     while queue.empty() do
7:         v ← queue.pop()
8:         for u ∈ G[v] do                ▷ Para todos los vecinos de v
9:             if m[u] == 0 then
10:                m[u] ← 1
11:                queue.push(u)
```

2.5 Algoritmos para detección de comunidades

Los algoritmos de detección de comunidades han sido ampliamente estudiados. Existen métodos iterativos en los que se asignan comunidades a los vértices y se va iterando considerando vecindades locales de cada vértice y ajustando a la comunidad de la vecindad de acuerdo con algún criterio. En uno de los casos más simples se basa en un procedimiento iterativo donde inicialmente todos los vértices tienen una comunidad asignada y a través de las iteraciones, éstas se van combinando siguiendo ciertas reglas definidas por el propio procedimiento. Al finalizar una cantidad n de iteraciones se recupera la estructura formada por las comunidades restantes las cuales se definen como la estructura de comunidades de la red. Ya que la comunidad se puede ver como una etiqueta para cada vértice en [40] se presentan los conceptos básicos en los métodos de propagación de etiquetas, además de algunas aplicaciones de algoritmos de propagación en medios digitales y las diferentes métricas utilizadas para evaluar el rendimiento de este tipo de algoritmos.

En el caso del estudio de las redes complejas y de las propiedades que presentan se ha buscado explotar los algoritmos existentes para el análisis y medición de este tipo de estructuras, llevando a la realización de diferentes tipos de técnicas paralelas en diferentes arquitecturas con el objetivo de acelerar las limitadas soluciones secuenciales [18].

En [27] se plantea una solución para la detección de comunidades en redes generadas a partir de datos de redes sociales utilizando diferentes herramientas de cómputo distribuido diseñadas sobre el modelo MapReduce. La solución abordada busca acelerar el conocido algoritmo Grivan-Newman que basa su funcionamiento en el agrupamientos jerárquico para la detección de comunidades, explotando el concepto de edge betwenness para particionar la red en diferentes subredes. El algoritmo de Grivan-Newman presenta dificultades para la detección de comunidades en redes de gran tamaño ya que requiere el cálculo de los caminos más cortos de la red para su funcionamiento. No obstante los resultados observados muestran un incremento en la aceleración de entre 4x a 6x. Sin embargo, las soluciones basadas en frameworks para el cómputo paralelo suelen ser poco flexibles para el desarrollo de algoritmos paralelos, debido principalmente a que los problemas deben adaptarse a los patrones de paralelización definidos en estos entornos de trabajo.

Por otra parte, en [28] se plantea una solución paralela para abordar el problema de la detección de comunidades en redes de gran tamaño utilizando una enfoque basado en arquitectura multicore. En este enfoque se toma como base la idea del procesamiento jerárquico observado en el método de Louvian, en el cual inicialmente todos los vértices de la red son considerados como una comunidad que mediante un proceso iterativo de dos fases se actualizan constantemente las comunidades en los vértices de modo que, al final de las iteraciones se obtiene una red particionada cuyas divisiones son consideradas las propias comunidades de la red.

Los trabajos que utilizan las arquitecturas basadas en aceleradores aunque son poco comunes. Por ejemplo en [35] se presenta el diseño de un algoritmo paralelo para la detección de comunidades basado en la propagación de etiquetas. Este enfoque basado en el algoritmo para detección de comunidades tiene la ventaja de trabajar de forma local con base en las relaciones de cada vértice, lo que le brinda la posibilidad de escalar de redes pequeñas a grandes dimensiones. La solución propuesta consta de dos algoritmos uno para plataformas multicore y otro para plataformas GPU observando en los resultados una aceleración de al menos 8x en la arquitectura basada en GPU sobre la multicore.

Con base en los trabajos encontrados relacionados con la detección de comunidades con algoritmos paralelos puede observarse la tendencia a elegir algún tipo de paradigma de cómputo paralelo (Distribuido, multicore y aceleradores basados en arreglos de procesadores GPU), sin embargo, las computadoras paralelas de la actualidad suelen permitir el uso de estos paradigmas de manera conjunta lo que posibilita la interacción entre ellos y la creación de aplicaciones híbridas que sean capaces de abordar problemas aprovechando las ventajas que tiene cada paradigma de cómputo paralelo.

2.6 Algoritmos para difusión de información

Las redes complejas pueden ser consideradas como modelos de interacción entre elementos del mismo tipo. A la propagación de información que ocurre a través de estas redes se le conoce como procesos de difusión. Los fenómenos de difusión que ocurren en las redes no son algo nuevo, se han estudiado ampliamente en la literatura, siendo abordados principalmente por métodos analíticos y numéricos [8]. En los procesos de difusión se estudian los comportamientos de las redes cuando sus vértices interactúan de cierto modo o ante agentes desconocidos. Entre algunos ejemplos de procesos de difusión estudiados se incluyen la propagación de información entre usuarios de redes sociales, la propagación de enfermedades, de virus informáticos e incluso de los protocolos de internet utilizados para establecer rutas.

Se puede decir que los procesos de difusión de información se refieren a la manera en la cual un bloque de información que posee alguno de los vértices de la red es distribuido hacia el resto. Existen diferentes enfoques para la difusión de información sobre redes. Una de las más simples es la difusión secuencial, en la cual, un vértice solamente puede propagar la información a alguno de sus vértices adyacentes a la vez, de modo que la propagación se realiza para toda la vecindad del vértice inicial, una vez completada, se elige alguno de estos vecinos para comenzar de nuevo con el procedimiento y repetirlo hasta que se haya propagado por toda la red. En los procesos de difusión de información uno

de los objetivos principales es lograr que la propagación de información sea lo más rápida posible. En [14] se presentan diferentes métodos para el estudio de epidemias en redes complejas, identificando las características en común que se encuentran en las técnicas de simulación numérica utilizadas y caracterizando los procesos de propagación observados. En el estudio de la difusión de información en redes son utilizadas frecuentemente las redes individuales, que son modeladas a partir de solo una interacción entre los elementos que conformarán la red. Por otra parte existen enfoques que utilizan a las redes multicapa, las cuales se modelan como conjuntos de elementos con diferentes tipos de conexiones, esto quiere decir que la red se forma a partir de diferentes interacciones entre los elementos que pertenecen a ella a diferencia de las redes individuales. En [33] se presentan algunos de los principales modelos y resultados de procesos de difusión sobre redes multicapa además de algunas de sus aplicaciones tales como el estudio de la dinámica de cascadas, maximización de influencia de información en el contexto de marketing, la selección de subconjuntos de vértices en los cuales colocar sensores para detectar la propagación de virus, entre algunas otras.

La Tabla 2.1 resume los trabajos relevantes en cada una de las secciones mostradas anteriormente, así como una breve descripción de sus principales características.

Resumen

En este Capítulo se presentaron los fundamentos teóricos relacionados con las principales arquitecturas paralelas. La descripción de los modelos de memoria y sus principales ventajas y desventajas. De igual modo se presentaron algunos de los trabajos relacionados y los enfoques que se aplican. En el siguiente Capítulo se presentarán los algoritmos para la detección de comunidades.

Autor(es)	Título del articulo	Año	Comentarios
Onnela Jukka-Pekka, Fenn Daniel J, Reid Stephen, Porter Mason A, Mucha Peter J, Fricker Mark D y Jones Nick S.	Taxonomies of networks from community structure.	2012	Se enfoca en la construcción de reglas de clasificación para estructuras de comunidades en redes.
			Define métricas de similaridad para la comparación de propiedades estructurales.
			Utiliza redes generadas a partir de datos reales.
Hafez Ahmed Ibrahem, Hassanien Aboul Ella y Fahmy Aly A.	Testing Community Detection Algorithms: A Closer Look at Datasets	2014	Denota la ausencia de conjuntos de datos útiles para la evaluación de métodos de detección de comunidades.
			Presenta una serie de pruebas a métodos para la detección de comunidades.
			Indica las características que deben presentarse en un conjunto de datos para realizar procesos de pruebas.
S. R. Chintalapudi y M. H. M. K. Prasad.	A survey on community detection algorithms in large scale real world networks.	2015	Presenta los principales algoritmos para detección de comunidades
			Aborda los problemas de comunidades separadas y sobrelapadas.
			Utiliza la métrica de modularidad para determinar el funcionamiento de la detección de comunidades.
			Se determina que los principales problemas de la detección de comunidades son la escalabilidad y la calidad de las soluciones.
Garcia Robledo Alberto, Diaz Perez Arturo y Morales Luna Guillermo.	Accelerating All-Sources BFS Metrics on Multi-core Clusters for Large-Scale Complex Network Analysis	2017	Presenta algoritmos paralelos híbridos de grano grueso para la aceleración de métricas en redes.
			Registran aceleraciones hasta 171x
J. Soman y A. Narang.	Fast Community Detection Algorithm with GPUs and Multicore Architectures	2011	Presenta solución para la detección de comunidades sobre una arquitectura GPU.
			Presenta solución para la detección de comunidades sobre arquitectura multicore.
			Utiliza redes de tamaño considerable para realizar pruebas a las soluciones.
Zoidi Olga, Fotiadou Eftychia, Nikolaidis Nikos y Pitas, Ioannis	Graph-Based Label Propagation in Digital Media: A Review	2015	Se discuten los conceptos fundamentales de la propagación de etiquetas en redes.
			Se abordan diferentes métodos de construcción de redes
			Aplicaciones de algoritmos de propagación de etiquetas.
E. Moradi, M. Fazlali y H. T. Malazi.	Fast parallel community detection algorithm based on modularity	2015	Presenta una solución multicore para la detección de comunidades.
			Funciona con base en el algoritmo de Louvian.
			Incluye componente estocástico para selección.
Seunghyeon Moon, Jae-Gil Lee, Minseo Kang, Minsoo Choy y Jin-woo Lee.	Parallel community detection on large graphs with MapReduce and GraphChi	2016	Su enfoque se basa en el algoritmo Grivan-Newman
			Utiliza datos de redes sociales
			Utiliza herramientas para cómputo distribuido basadas en mapreduce.
M. Salehi, R. Sharma, M. Marzolla, M. Magnani, P. Siyari y D. Montesi.	Spreading Processes in Multilayer Networks	2015	Se presentan los modelo básicos de difusión en redes multicapa.
			Principales aplicaciones de procesos de difusión multicapa.
			Revisión de las posibles aplicaciones futuras.
Guilherme Ferraz de Arruda, Francisco A. Rodrigues y Yamir Moreno.	Fundamentals of spreading processes in single and multilayer complex networks	2018	Se revisan los principales métodos teóricos y numéricos para el estudio de procesos de difusión en red complejos.
			Define métodos de clasificación de procesos de difusión
J. Bethencourt, A. Sahai y B. Waters	Ciphertext-Policy Attribute-Based Encryption	2007	Se introduce por primera vez el sistema CP-ABE (Ciphertext-policy attribute based encryption)
			Se muestran medidas de desempeño para el esquema planteado.
Goyal Vipul, Jain Abhishek, Pandey, Omkant y Sahai Amit.	Bounded Ciphertext Policy Attribute Based Encryption	2008	Se introduce al cifrado basado en atributos (ABE).
			Se presenta la construcción de un esquema CP-ABE.
			Define prueba de seguridad basada en problema decisional Diffie-Hellman.

Tabla 2.1: Aspectos generales de trabajos en el estado arte relacionados con esta propuesta de tesis.

3

Algoritmos paralelos para la de detección de comunidades

3.1 Introducción

El problema de la detección de comunidades se refiere a un grupo de vértices que se encuentran densamente conectados entre sí que con el resto de los vértices pertenecientes a la red. El proceso de detección de comunidades implica la selección de grupos de vértices de una red con la más alta cantidad de aristas dentro de cada grupo.

Extraer la estructura de comunidades de una red representa un problema difícil de evaluar y aún se considera como una discusión abierta. Esto debido a que la evaluación debe realizarse sobre redes de estructura conocida, por lo que en otros casos deberá plantearse una métrica que defina la calidad de las comunidades. Entre algunas de las propuestas más utilizadas para evaluar a las comunidades se encuentra la modularidad que proporciona un valor en el rango -1 y 1 con lo que se

obtiene un punto de partida para evaluar a las comunidades determinadas por un procedimiento. Los métodos utilizados para la detección de comunidades suelen dividirse en dos grupos principalmente, los heurísticos que tienen como base la métrica de la modularidad la cual se busca maximizar y los iterativos que por medio de decisiones tomadas con base principalmente en la cantidad de aristas generan grupos de vértices que se espera estén altamente relacionados.

En cualquier caso el proceso para determinar las comunidades de una red tendrá como entrada una red y como salida una marca de comunidad para cada vértice.

3.2 Algoritmo LP secuencial

La detección de comunidades en redes es un tema ampliamente abordado en la literatura, por lo que es posible encontrar variedad de soluciones y propuestas para este problema. Entre algunas de los métodos más citados se encuentran Leading Eigen Vector, Edge Betweenness, Label Propagation, Multi-Level, por mencionar algunos [11].

En este capítulo se desarrollan algoritmos paralelos para la detección de comunidades en redes complejas.

El algoritmo Label propagation (LP) [39, 40], el cual ha mostrado buenos resultados en casos concretos de redes complejas además de ser un procedimiento intuitivo altamente paralelizable y sin dependencias de resultados. El algoritmo LP consta de las siguientes fases de operación:

1. Inicialización. Cada uno de los vértices en la red se define un valor de etiqueta diferente, el cual identifica la comunidad a la que pertenece cada vértice

2. Cálculo de frecuencia. Se busca que cada vértice revise la etiqueta de comunidad de sus vecinos y determine la frecuencia de cada uno de ellos. Debido a que en la etapa de **Inicialización** se define una etiqueta diferente para cada vértice en el primer cálculo de frecuencia aparecerá un caso especial, ya que todos los vértices vecinos tendrán un valor de frecuencia **1** que deberá resolverse en la etapa siguiente

3. Obtención de frecuencia máxima. Con los resultados del cálculo de frecuencia para los vecinos de cada vértice se deberá obtener la etiqueta de comunidad más frecuente en el conjunto de vecinos. En esta fase se presentan al menos dos casos especiales los cuales pueden abordarse de diferentes maneras dependiendo de los requerimientos de aplicación o restricciones. El primer caso especial es la frecuencia 1 que ocurre en la primera ronda de ejecución cuando todas las etiquetas de la red son diferentes, el segundo ocurre cuando existe más de una etiqueta de comunidad con la misma frecuencia.

En la documentación original del algoritmo LP se establece que selección de etiquetas es de forma aleatoria. Para este caso la estrategia que se utiliza es la selección por orden fijo, eligiendo siempre la primera etiqueta de comunidad entre los iguales. Esto para garantizar que los resultados sean deterministas y evitar añadir complejidad al algoritmo con un mecanismo de selección complejo.

Un factor influyente en la elección de la condición de parada es el comportamiento que se observa directamente en el algoritmo LP, el cual muestra en la mayoría de los casos que las comunidades detectadas a través de las iteraciones disminuye hasta alcanzar un estado fijo. Donde la cantidad de iteraciones se encuentre usualmente en el orden de las decenas, por lo que el costo computacional del mecanismo de parada puede llegar a superar el costo de posibles iteraciones redundantes. Esto se ejemplifica en la Figura 3.1.

Actualmente existen diversas implementaciones de algoritmo LP la diferencia entre ellas usualmente se encuentra en la condición de parada y el mecanismo de selección de etiqueta más frecuente, sin embargo, existen otros factores que afectan el desempeño de la solución, como el diseño del algoritmo y las estructuras de datos utilizadas. En el apartado del Algoritmo 3 se presenta la descripción de la estructura de datos con los atributos utilizados en la implementación del LP utilizado y que es válida para las versiones multicore.

El primer atributo A, define una lista de listas mejor conocido como formato LIL donde el elemento $\mathbf{0}$ es el índice del primer vértice, por lo que el primer elemento en A es la lista de vecinos del vértice $\mathbf{0}$, el segundo elemento son los vecinos del vértice $\mathbf{1}$ y así consecutivamente. Utilizar este tipo de

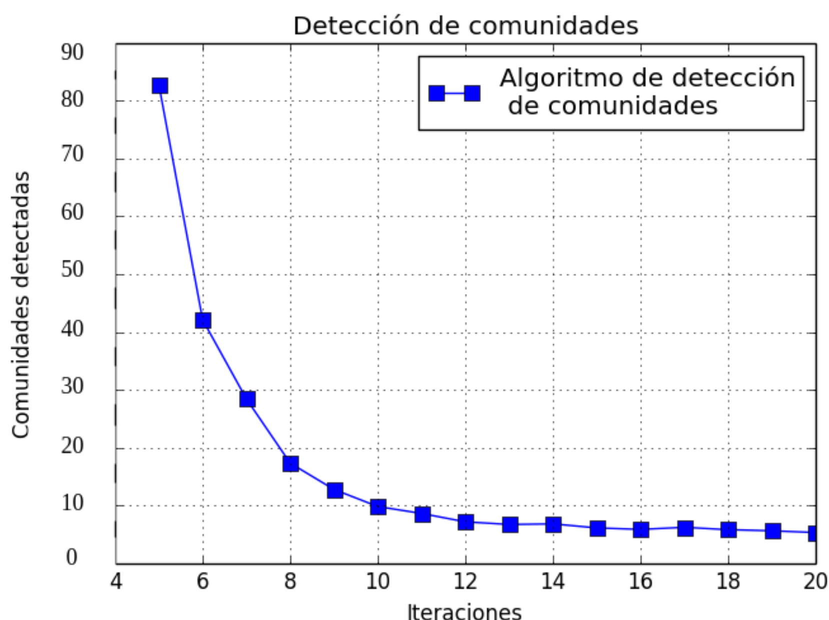

Figura 3.1: Comunidades detectadas por medio de Label propagation a través de las iteraciones.

representación evita el uso excesivo de memoria como pasaría con una representación del tipo matriz de adyacencia, en la cual la mayor parte de la estructura se encuentra sin valores definidos debido a que la densidad de aristas suele ser mucho menor que el cuadrado de la cantidad de vértices.

El segundo atributo W, define los pesos para las aristas de la red. Siguiendo la nomenclatura de A el primer elemento de W es la lista de pesos para los vecinos del vértice **0** ocurriendo del mismo modo para el resto de vértices siguientes.

El tercer atributo C, define la etiqueta de comunidad para cada vértice de la red, aquellos vértices que comparten la misma etiqueta se considera pertenecen a la misma comunidad.

Debido a que en la arquitectura GPU no se permite el uso de memoria dinámica se debe evitar el uso de diccionarios o mapas de hash para la representación de la red, sin embargo, se pueden utilizar estructuras de datos como pilas y listas dentro de las implementaciones secuenciales y multicore que así lo requieran.

En el Algoritmo 3 se presentan los pasos generales para la detección de comunidades por medio del algoritmo LP, el cual recibe como parámetros de entrada un red G y un número de iteraciones

Algoritmo 3 Label propagation secuencial

```
1:  struct Network
2:      Array[ ][ ]A
3:      Array[ ][ ]W
4:      Array[ ]C
5:
6:  function LabelPropagation(G, iterations)
7:      while iterations > 0 do
8:          for node ∈ G do
9:              f ← labelFrecuency(G.A[node])
10:             m ← maxLabel(f)
11:             G.C[node] ← m
12:         iterations ← iterations − 1
13:     return G
```

máximo.

Durante la cantidad de iteraciones definidas se procederá a calcular para cada lista de vecinos la etiqueta de comunidad más frecuente($labelFrecuency$) que obtendrá como resultado una lista con la frecuencia de las etiquetas de comunidad dentro del grupo de vecinos(f). El siguiente paso es seleccionar la etiqueta de comunidad más frecuente y en los casos especiales simplemente elegir la primera etiqueta dentro de la lista de resultados.

El paso final es el reemplazo de la etiqueta de comunidad, el cual puede darse de dos formas diferentes. La primera definida como reemplazo asíncrono, donde la etiqueta de comunidad es reemplazada inmediatamente cuando se obtiene el resultado y se escribe sobre la misma estructura de la cual se leyeron las etiquetas de comunidad. Esto no representan un problema para el resultado final ya que las salidas obtenidas para el mismo conjunto de datos y la misma cantidad de iteraciones deberá ser la misma en todos los casos. Sin embargo, al ser un procedimiento secuencial el orden en que se procesan los vértices de la red influye directamente en el resultado, por lo que si se cambia el orden en el que se procesa los datos de entrada el resultado cambiará aún cuando se conserven los mismos parámetros.

El método de reemplazo síncrono, utiliza dos estructuras de resultado. La primera estructura sirve

para leer y determinar un resultado que será escrito sobre la segunda estructura. Una vez que todas las lecturas y escrituras se hayan realizado para una ronda de ejecución, la estructura de escritura se copia a la de lectura asegurando que el orden en el que se procesen los datos no afecta el resultado de ninguna manera. La Figura 3.2 muestra la diferencia entre el reemplazo síncrono y asíncrono.

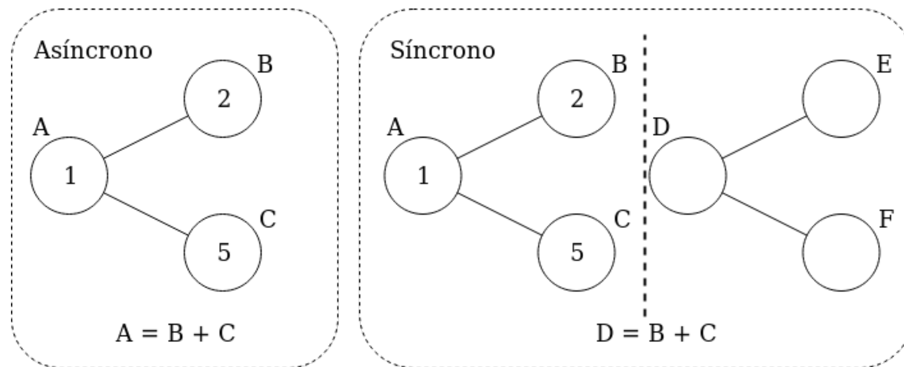

Figura 3.2: Métodos de actualización de etiqueta síncrono y asíncrono.

Tomando en cuenta los diferentes recorridos sobre la red descritos y que el procedimiento se repetirá por una cantidad de iteraciones n, es posible estimar que el costo computacional del algoritmo LP descrito es aproximadamente de $O(n \cdot (2e + v))$.

3.3 Algortimo LP multicore

La solución multicore del algoritmo LP sigue la estrategia de la versión secuencial, sin embargo, contemplando la distribución de trabajo realizada por vértice con un método de particionamiento cíclico. En la Figura 3.3 se muestra la distribución de los threads sobre los vértices de la red, así como las localidades de lectura y escritura denotando un procesamiento libre de condiciones de carrera. Por otra parte en el Algoritmo 4 se describe la estrategia de paralización de LP mediante un mecanismo de distribución cíclico en donde los threads se sincronizan al final de cada iteración.

Dentro del mecanismo de particionamiento descrito en el procedimiento $Propagate(G)$ se observa que se lanzarán p threads correspondientes a un procesador y cada uno de ellos ejecutara el procedimiento $LabelPropagation(G, p, i)$ con el índice de cada thread así como la cantidad de

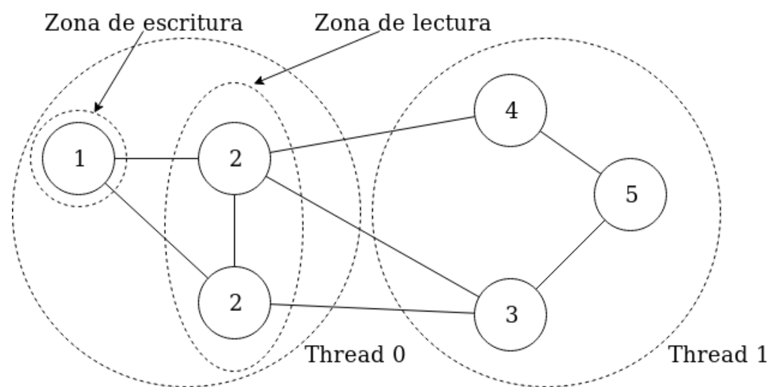

Figura 3.3: Distribución de trabajo por vértice y accesos a memoria.

estos. Cada thread será responsable de procesar los vértices asociados a su índice con respecto a p. Sin embargo, el procesamiento para cada vértice será el mismo de la solución secuencial, pasando en primer lugar por el cálculo de frecuencia de la etiqueta de comunidad. Para después pasar a la determinación de la etiqueta de comunidad más frecuente y por último realizar la selección de ésta.

Idealmente, el tiempo de ejecución de un algoritmo paralelo sería el tiempo de ejecución de un solo procesador dividido entre la cantidad de procesadores. Aunque esta premisa no suele ocurrir en aplicaciones reales el objetivo es aproximarse a esta marca. A pesar de que el algoritmo LP secuencial tiene una complejidad de $O(n \cdot (2e+v))$ con n igual a las iteraciones definidas, e la cantidad de aristas y v la cantidad de vértices. La complejidad del algoritmo LP multicore no se convierte directamente en $O(\frac{n \cdot (2e+v)}{p})$ con p igual a la cantidad de procesadores disponibles, debido a que la partición por vértice no garantiza la regularidad en la carga de trabajo. Por otra parte el trabajo realizado, es decir la cantidad de operaciones básicas realizadas por el algoritmo LP multicore se mantiene igual a la cantidad de operaciones realizadas por la versión secuencial. Mientras que el tiempo que toma terminar la tarea queda determinada por el thread con la mayor carga de trabajo. Esta es la razón de usar un particionamiento cíclico, con la expectativa de que a cada thread le toque una variedad de vértices que en promedio las cargas sean más o menos similares.

Algoritmo 4 Algoritmo LP multicore

```
1:
2:  procedure LabelPropagation(G, p, i)
3:      while iterations do
4:          for node_i ∈ G do                              ▷ Para todos los vértices en G
5:              f ← labelFrecuency(G.A[node_i])
6:              m ← maxLabel(f)
7:              G.C[node_i] ← m
8:              i ← i + p
9:          syncthreads()
10:         iterations ← iterations − 1
11:
12:
13: procedure Propagate(G)
14:     for i ∈ [0, 1, 2, ..p − 1] do
15:         startThread(LabelPropagation, [G, p, i])
16:     joinThreads()
                                                           ▷ Actualizar sobre copia
17:                                                        ▷ de G.C[node_i](6) para
18:                                                        ▷ procesamiento síncrono
```

3.4 Algoritmo LP GPU

En el algoritmo LP existen dos operaciones principales que conforman el mecanismo de detección de comunidades y son: el cálculo de frecuencia para un vértice y la extracción y selección de la etiqueta de comunidad con mayor frecuencia.

Tomando como base estas dos operaciones principales, más el uso de la suma de prefijos para clasificar los vértices de la red, el enfoque de solución se plantea como un mecanismo de tres pasos y que se describe en la Figura 3.4 en donde el primer paso será la clasificación, la cual se encargará de agrupar a los vértices con una cantidad de vecinos similar, con lo que se busca regularizar la carga de trabajo de los threads.

La clasificación de vértices por el grado es la primera parte en el procesamiento de la solución GPU, donde el grado de un vértice u denotado como $g(u)$, índica la cantidad de relaciones que

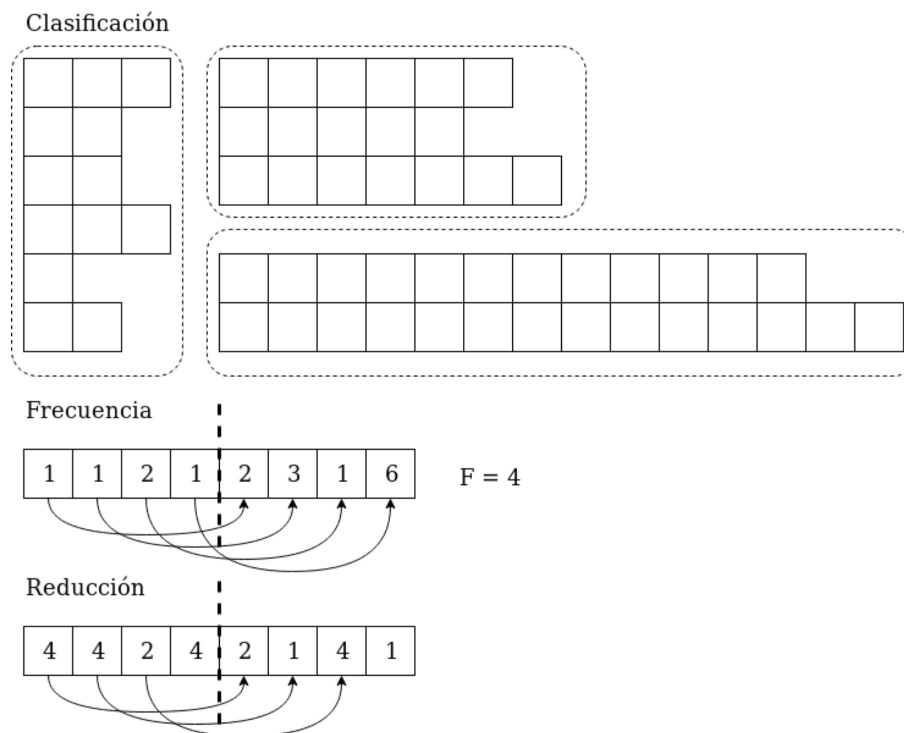

Figura 3.4: Etapas de procesamiento para el algoritmo LP GPU.

tiene con otros vértices. Para aplicar la operación de suma de prefijos y clasificar a los vértices es necesario determinar que vértices cumplen con un predicado. El funcionamiento y algoritmo de la suma de prefijos se describe en el el Capítulo 2. Para esta caso se utilizan al menos tres predicados para conseguir grupos de vértices de tres clases. En primer lugar los vértices con un grado menor que **64**, este grupo de vértices representan a la fracción mayoritaria en las redes complejas y se obtiene tal cantidad debido a las propiedades de la arquitectura GPU, donde los threads se despliegan internamente como grupos de **32** threads que se ejecutan de manera síncrona, es decir todos realizan la misma operación en la misma unidad de tiempo.

Tener vértices con grado a lo más **64** asegura que al menos en la primera operación de una reducción con **32** threads por vértice se utilice la totalidad de los threads y se evita el uso de primitivas de comunicación o sincronización entre los threads ya que la ejecución con bloques de **32** es naturalmente síncrona.

La segunda clase de vértices son aquellos con un grado tal que $64 < g(u) \leq 10,000$, aunque el

limite fijado por **10,000** se encuentra determinado por la cantidad de memoria disponible por bloque en la arquitectura y que usualmente no es superior a los **48**kb resultando en un valor aproximado asumiendo que solo se utilizan valores enteros que requieren al menos **4** bytes.

La tercer y última categoría de vértices son aquellos que no cumplen las características anteriores, es decir los vértices con un grado superior a **10,000**.

El segundo paso representa al cálculo de frecuencia de las etiquetas de comunidad para cada vértice de la red. En esta etapa se calcula la frecuencia para cada uno de los grupos clasificados en la etapa anterior. En la tercera y última etapa se encuentra el cálculo de la etiqueta de comunidad más frecuente y que hace uso del resultado de la etapa de frecuencia. Esta etapa es fácilmente lograda por medio de una reducción ya que el operador $<$, o el $>$ representan operaciones binarias asociativas.

A diferencia de las arquitecturas multicore en las cuales es posible utilizar un gran variedad de estructuras de datos fijas o dinámicas, las GPUs se encuentran limitadas en ese aspecto, ya que solo es posible utilizar bloques de memoria fijos a modo de arreglos unidimensionales y algunas variaciones. Las estructuras utilizadas para el algoritmo LP GPU se presentan a continuación.

- $Array\ A[\ ]$: Representa una lista de adyacencia comprimida por filas similar al formato CSR, en esta estructura se encuentran los índices de los vecinos de cada vértice. Ejemplo: $A = [2, 1, 4, 3, 6, 12, 90, 21, 11, ...]$ denotando que A contiene todos los enlaces que existen en la red y que se encuentran delimitados por L. El tamaño de esta estructura es determinado por la cantidad de aristas e en la red.

- $Array\ L[\ ]$: Delimita la cantidad de vecinos que tiene un vértice. Ejemplo: $L = [4, 7, 9, ...]$ denotando que los primeros **4** elementos de A son los vecinos del vértice **0**, los siguientes **3** elementos pertenecen al vértice **1**, los siguientes **2** al vértice **2** y así sucesivamente. Agregando el índice **0** a esta estructura se obtiene el inicio y final de cada segmento de vecinos de todos los vértices $L = [0] + [4, 7, 9, ...]$. El tamaño de esta estructura es determinado por la cantidad de vértices v más **1**.

- $Array\ C[\ ]$: Determina la etiqueta de comunidad para cada uno de los vértices de la red, su tamaño también es determinado por éstos. Ejemplo: $C = [1, 2, 3, ...]$ donde el primer valor representa la etiqueta de comunidad para el vértice **0**, el segundo para el vértice **1** y del mismo modo sucesivamente.

- $Array\ R[\ ]$: Representa un vector resultado en el que se escriben los resultados parciales o totales de una fase de operación. Su tamaño es determinado por la cantidad de aristas e al igual que A. R' denota un vector de resultados diferente o auxiliar.

La segunda etapa de procesamiento requiere el cálculo de frecuencia para cada vértice. Esta etapa se describe por medio del Pseudocódigo 5 en el cual se muestra el mecanismo utilizado por el kernel de CUDA para abordar el problema. En primer lugar se observa que el índice determinado para cada vértice($index$) se encuentra en función del bloque, por lo que todos los threads del bloque procesarán a un vértice especifico. Como se sabe que los vértices se encuentran clasificados es posible definir una cantidad de threads por bloque que aproveche las ventajas de la sincronía y la memoria compartida.

En un primer paso se determina para cada bloque el inicio y final de los vecinos del vértice que debe procesar($start, end$). Conociendo el segmento de vecinos dentro de A se procede a posicionar un thread para cada uno de los elementos del segmento que representan a los índices de los vecinos. Para cada vecino se realizará un recorrido sobre todo el segmento contando la cantidad de apariciones de la etiqueta de comunidad con respecto a cada uno de ellos. Para mejorar el desempeño del procesamiento se copian los datos desde la memoria global hacia la memoria compartida. Al final del recorrido del segmento por todos los threads, la frecuencia de aparición para cada una de las comunidades de ellos se escribirá sobre el vector resultado R que será utilizado por el siguiente kernel.

La última etapa de procesamiento involucra los resultados de la etapa del cálculo de frecuencia descrita anteriormente cuyos resultados se asume se encuentran en R. Esta etapa busca determinar la etiqueta de comunidad más frecuente para cada uno de los vértices. Para este caso el procedimiento se realiza por medio de una operación de reducción revisada anteriormente, utilizando el operador

Algoritmo 5 Kernel cálculo de frecuencia

```
1:  kernel frequency(C, A, L, R, v)
2:      blockid ← blockIdx.x
3:      com     ← [ ]
4:      while blockid < v do                          ▷ Un bloque por lista de adyacencia
5:          start  ← L[blockid]
6:          end    ← L[blockid + 1]
7:          myc    ← C[blockid]
8:          com    ← copyToLocalMemory(C, start, end)
9:          index ← start + threadIdx.x
10:         total  ← 0
11:         for c ∈ com do
12:             total ← total + (myc = c)
13:         R[index] ← total
14:         blockid   ← blockid + gridDim.x
```

para seleccionar el máximo valor.

El mecanismo de extracción de las etiquetas de máxima frecuencia se describe con el Pseudocódigo 6, en cual se inicia determinando los límites de la vecindad de cada vértice con $start, end$ a partir del vector de índices L. Una vez ubicado el segmento se copia directamente a memoria compartida sobre f y se aplica la reducción por cada segmento, obteniendo como resultado la etiqueta de comunidad más frecuente sobre la posición inicial de cada bloque determinada por el índice del bloque $blockid$ y que finalmente será actualizada por el thread **0** de cada bloque.

Cada uno de los kernels descritos funcionan bajo la premisa de redes de gran tamaño por lo que cada bloque de threads deberá realizar trabajo de forma cíclica hasta terminar de procesar los v vértices de la red.

Resumen

En este capítulo se ha descrito la metodología para la detección de comunidades, así como las soluciones propuestas para cada arquitectura paralela multicore y GPU. Del mismo modo se presentaron las estructuras de datos utilizadas y las consideraciones necesarias para la aplicación de

Algoritmo 6 Kernel cálculo de comunidad más frecuente

1: **kernel** $reduction(R, L, P, v)$
2: $blockid \leftarrow blockIdx.x$
3: $bdim \leftarrow blockDim.x$
4: $f \leftarrow [\,]$
5: **while** $blockid < v$ **do** ▷ Un bloque por lista de adyacencia
6: $start \leftarrow L[blockid]$
7: $end \leftarrow L[blockid + 1]$
8: $f \leftarrow copyToLocalMemory(R, start, end)$
9: **while** $bdim > 0$ **do**
10: **if** $threadIdx.x < dim$ **then**
11: **if** $f[threadIdx.x + dim] > f[threadIdx.x]$ **then**
12: $f[threadIdx.x] \leftarrow f[threadIdx.x + dim]$
13: $dim \leftarrow dim/2$
14: **if** $threadIdx.x = 0$ **then**
15: $C[blockid] \leftarrow C[f[0]]$
16: $blockid \leftarrow blockid + gridDim.x$
▷ *Actualizar sobre copia*
17: ▷ *de* $C[blockid]$*(15) para*
18: ▷ *procesamiento síncrono*

los algoritmos.

4

Algoritmos paralelos para el proceso de difusión

4.1 Introducción

En este Capítulo se describe el algoritmo de difusión de información a través de una red compleja. La información para los algoritmos descritos en este capítulo se trata de una etiqueta que tiene como origen un vértice difusor. A partir de éste se propaga la etiqueta a sus vecinos no etiquetados.

La difusión se centra en los vértices difusores, los cuales determinarán el costo de procesamiento de una red. En el proceso de difusión que se aborda en este trabajo se requiere que la red previamente esté particionada en comunidades, es decir que la red deberá estar marcada con etiquetas de comunidad donde los vértices con etiquetas compartidas se asume pertenecen a la misma comunidad.

También se requiere que se hayan identificado los vértices difusores a partir de los cuales se iniciará la difusión de las etiquetas. Los difusores pueden ser seleccionados con base en la el grado del vértice, la centralidad, u otras métricas que determinen la importancia de éste dentro de la comunidad o la red [8, 13].

4.2 Algoritmo de difusión secuencial

El proceso de difusión que se presenta aplica una métrica de distancia para realizar la propagación de la política hacia el resto de los vértices no difusores pertenecientes a una comunidad. La idea general es que para cada vértice no difusor, se calcula una medida de distancia hacia los vértices difusores de la misma comunidad. Aquel vértice con la distancia más corta entre los difusores será seleccionado para propagar su política hacia los no difusores.

En la Figura 4.1 se presentan algunos de los casos que se espera aparezcan frecuentemente en el procesamiento de redes complejas, iniciando con las comunidades menores con una cantidad de vértices limitada y que en el peor de los casos tienen solo un vértice y un difusor, mientras que el otro caso serían las comunidades altamente pobladas con una alta cantidad de difusores.

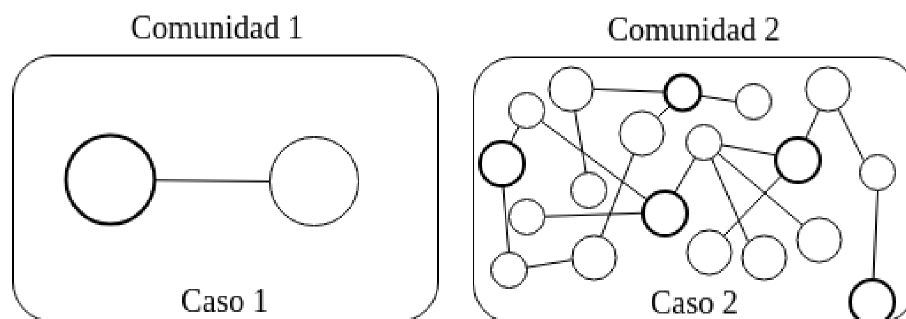

Figura 4.1: Casos especiales de comunidades donde los vértices difusores se encuentran resaltados de los no difusores.

Un caso especial que simplifica el problema es cuando una comunidad contiene únicamente a un difusor. En este caso, todos los vértices de la misma comunidad adquieren la etiqueta de su difusor.

Para realizar el proceso de difusión sobre una red se hace uso de una métrica de distancia, lo que implica que la red de entrada deberá ser contener pesos en sus aristas que denoten la importancia de los caminos, de otro modo puede utilizarse el caso unitario, donde el valor de todos los enlaces en la red tiene un valor de uno.

La Figura 4.2 muestra un diagrama de la entrada y la salida del algoritmo de difusión, siendo la

entrada una red marcada con al menos un vértice difusor y como salida la misma red con el resto de los vértices no difusores con una etiqueta propagada.

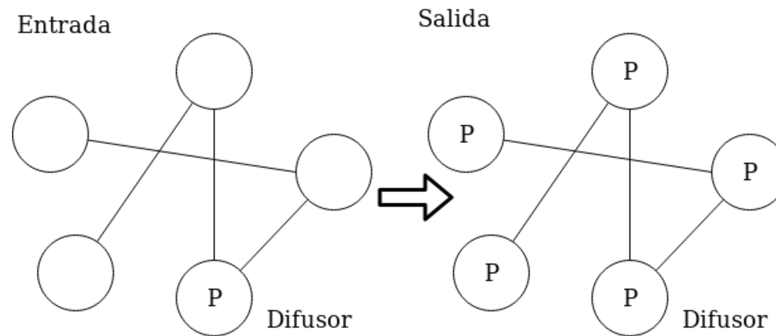

Figura 4.2: Entradas y salida del algoritmo de difusión.

Para realizar el proceso de difusión se requiere que la red presenta tres propiedades fundamentales, la primera es el particionamiento por comunidad que puede ser representado por una etiqueta de comunidad para cada vértice, la segunda es la definición de los vértices difusores, es decir que cada comunidad presente en la red deberá contener al menos uno vértice difusor con su respectiva etiqueta que será el bloque de información que se propagará por las comunidades. La última propiedad necesaria es la definición de pesos de las aristas, que deberá denotar la importancia de los enlaces entre los vértices de la red y que puede ser un factor que determinante en la propagación de las etiquetas en un ambiente intra-comunidad donde un menor costo entre vértices indicaría una mayor probabilidad de propagación.

En el Algoritmo 7 se presenta las estructura de datos utilizada para el algoritmo de difusión. La cual es una extensión de la estructura presentada en el Capítulo 3, agregando los atributos: P que define la etiqueta para cada vértice de la red y S que define a los vértices difusores de la red.

El proceso de difusión que se presenta sigue una estrategia similar a la que se plantea en el algoritmo Dijkstra [5], que es una de las técnicas más conocidas para el cálculo de las rutas más cortas en una red. El primer paso será identificar a los vértices difusores por medio de el vector S el cual índica con un valor numérico 1 si se trata de un difusor o un 0 en caso contrario.

Para cada vértice difusor se realizará un recorrido limitado a los vértices pertenecientes a su

comunidad conocida por medio de C. Se usa una estructura $queue$ que sigue el comportamiento de una cola de prioridad invertida, es decir que la operación $pop()$ retorna el elemento con menor valor contenido en la cola. Los elementos que se insertan en la cola son la distancia D, y el vértice src.

El proceso inicia insertando a todos los difusores en la cola e indicando una distancia cero. A partir de cualquiera difusor se comenzará por recorrer sus vecinos actualizando los valores de distancia para cada uno de ellos. Al mismo tiempo que se actualiza la etiqueta de los vértices vecinos del difusor, esto provocará que para el primer vértice difusor en realizar el recorrido se visite la totalidad de los vértices pertenecientes a la comunidad del difusor.

Cada vez que se recorren los vecinos de un difusor, se revisa si la distancia que existe entre el difusor y el vecino es menor que la determinada en la inicialización, que se establece a un valor determinado como infinito para asegurar que existan valores menores. Cuando se actualiza un valor de distancia para un vértice vecino éste se agrega a la cola de prioridad. Siguiendo la estrategia del algoritmo de Dijkstra para proseguir con la ruta más corta de modo que una vez visitados y agregados los vecinos del difusor inicial. El siguiente paso será extraer el vértice con menor costo en distancia y repitiendo el proceso de revisión, actualización de distancia y de la etiqueta. Esto significa que el siguiente difusor perteneciente a la misma comunidad visita solamente los vértices vecinos cuya distancia sea menor que la establecida por el difusor anterior.

Tomando en cuenta que solo se requiere de un recorrido sobre la red, y asumiendo que se utiliza una pila de prioridad basada en árboles balanceados se estima que el costo computacional del algoritmo de difusión descrito es aproximadamente de $O(e \cdot log_2(v))$.

4.3 Algoritmo de difusión multicore

La parte más importante del proceso de difusión sobre arquitectura multicore está determinado por el particionamiento de los vértices a los diferentes threads disponibles para realizar el procesamiento.

Algoritmo 7 Algoritmo de difusión secuencial

```
1:  struct Network
2:      Array[ ][ ]A
3:      Array[ ][ ]W
4:      Array[ ]C
5:      Array[ ]P
6:      Array[ ]S
7:
8:  function spreading(G, D, src)
9:      queue.push([src, 0])
10:     while queue.empty() do
11:         v, d ← queue.pop()
12:         for ngh ∈ G.A[v] do                    ▷ Para todos los vecinos de v en G.A
13:             nghd ← G.W[v][ngh] + d
14:             if nghd < D[ngh] then
15:                 queue.push([ngh, nghd])
16:                 G.P[ngh] ← G.P[src]
17:
18: for node ∈ G.S do
19:     spreading(G, D, node)
```

4.3.1 Estrategia de particionamiento por comunidad

Para este caso la distribución se puede realizar en dos niveles, ya se por comunidad, en la que se asignará una comunidad para cada thread y dependiendo del modelo de particonamiento se distribuirá la carga del resto de las comunidades por procesar. Por otra parte también se puede realizar una distribución de trabajo por vértice sin tomar en cuenta las comunidades.

El primer enfoque puede derivar en un balanceo de carga no homogéneo. Conociendo que las redes que de entrada tendrán propiedades de redes complejas y se conoce que éstas tienen estructuras irregulares, es altamente probable que las comunidades no se encuentren balanceadas en el número de elementos por comunidad. Esto provocará, al ser una distribución por comunidad que la carga de trabajo irregular afecte el desempeño de la solución.

Por otro lado, al realizar la distribución por comunidad, no es necesario realizar un recorrido para cada uno de los vértices pertenecientes a la misma, sino que es posible que un solo recorrido actualice

distancias y etiquetas en una sola ronda efectuada por comunidad.

Debido a que la determinación de costos entre vértices y actualización de etiquetas pertenecientes a la misma comunidad se realiza por medio de un thread, se evitan los posibles conflictos y condiciones de carrera que pudieran presentarse en lecturas y escrituras sobre las estructuras de datos si se procesarán por más de un solo thread.

El esquema de la distribución de los vértices por comunidad indica que cada comunidad debe ser procesada únicamente por un thread eliminando conflictos y condiciones de carrera, pero con la limitante de un posible descenso en el rendimiento cuando las comunidades presenten ordenes irregulares en la cantidad de elementos.

En el Algoritmo 8 se presenta el algoritmo de difusión multicore, que ejecuta cada uno de los threads, donde la principal diferencia con respecto a la versión secuencial se centra en la condicional agregada en la revisión de distancias en la cual se busca encontrar vértices difusores conforme se procesan los vecinos. De tal manera que de encontrarse un nuevo vértice difusor éste será agregado a la cola de prioridad $queue$ con un valor de distancia inicializado en infinito denotado por INF. Con un valor de infinito se asegura que los vértices difusores no saldrán de la cola hasta el final, de manera que cuando sean extraídos de la cola el resto de vértices no difusores ya habrán sido procesados y actualizadas sus etiquetas. Esto permite que los vértices difusores restantes en la cola realicen una cantidad mínima de operación de actualización y distancia.

Por otra parte en el Algoritmo 9 se describe el mecanismo de distribución de trabajo, siguiendo una estrategia de distribución cíclica por comunidades. En esta estrategia se busca como primer paso agrupar los vértices por comunidad, conociendo que cada vértice tiene una etiqueta de comunidad denotada con C es posible realizar un agrupamiento por medio del recorrido de todos los vértices de la red.

El siguiente paso es el de la distribución del trabajo, que se representa por la llamada al procedimiento $spreading(...)$ invocada por $startSpreading(...)$ que recibe como parámetros la red de entrada G, el grupo de vértices pertenecientes a una comunidad GP, la cantidad de threads

Algoritmo 8 Algoritmo de difusión multicore: Distribución por comunidad

```
1: procedure spreading_(G, D, src)
2:     queue.push([src, 0])
3:     INF ← maxIntV
4:     while queue.empty() do
5:         v, d ← queue.pop()
6:         if G.S[v] then
7:             D[v] ← 0
8:             src ← v
9:         for ngh ∈ G.A[v] do                ▷ Para todos los vecinos de v en G.A
10:            nghd ← G.W[v][ngh] + d
11:            if nghd < D[ngh] then
12:                if G.SS[ngh] then
13:                    D[ngh] ← 0
14:                    queue.push([ngh, INF])
15:                else
16:                    D[ngh] = nghd
17:                    G.P[ngh] ← G.P[src]
18:                    queue.push([ngh, nghd])
```

para realizar el trabajo P y el índice identificador del thread i. Cada thread con índice i deberá procesar las comunidades $i, i+P, i+2P, i+3P$ con P igual al total de threads desplegados para el procesamiento, distribuyendo así cíclicamente las comunidades entre todos los threads.

4.3.2 Estrategia de particionamiento por vértice

A diferencia de la distribución de trabajo por comunidad, en la distribución por vértice es posible obtener una carga de trabajo mejor balanceada, ya que el factor limitante en la carga irregular solo se encuentra dado por el grado de los vértices y no por la cantidad de elementos en la comunidad. Siguiendo una estrategia de distribución por vértice no es necesario realizar un agrupamiento de vértices inicial, sin embargo, se pueden presentar inconsistencias en los resultados provocados por condiciones de carrera que se producen en lecturas y escrituras.

En el caso en que dos o más threads se encuentran procesando vértices pertenecientes a la misma comunidad, se crea un conflicto en la actualización de distancias. Ya que si diferentes threads

Algoritmo 9 Algoritmo de difusión multicore: Orquestación de threads

```
1: procedure spreading(G, GP, P, i)
2:     D ← [0...G.nodes]
3:     for group_i ∈ GP do                    ▷ Para todos los vértices difusores
4:         spreader ← getSpreader(group_i)
5:         spreading_(G, D, spreader)
6:         i ← i + P
7:
8: procedure startSpreading(G)
9:     GP ← groupByComunity(G)
10:    for i ∈ [0, 1, 2, ..P − 1] do
11:        startThread(spreading, [G, GP, P, i])
12:    joinThreads()
```

proceden a revisar y actualizar el valor de distancia del mismo vértice no se puede garantizar que la distancia establecida sea realmente la más corta y afecte directamente al resultado. En la Figura 4.3 se presenta el caso en el que ocurre un conflicto de acceso a la memoria debido a dos threads tratando de actualizar los valores de un vértice en común.

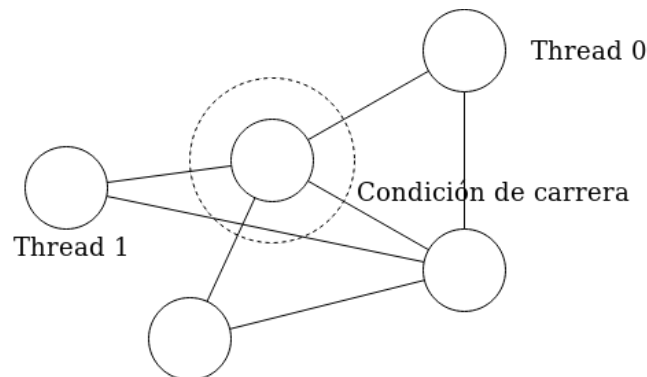

Figura 4.3: Caso que presenta una condición de carrera para dos threads.

Debido a este inconveniente, se requiere de un mecanismo de acceso sincronizado a los vértices de la red. Por lo que se utilizan variables de exclusión mutua, conocidas como mutex, que permiten el acceso exclusivo a un segmento de código que en este caso permitirá realizar la revisión y actualización de distancias y etiquetas sin comprometer la validez de los resultados. Sin embargo, el uso de exclusión mutua agrega un costo adicional al procesamiento, no solo por la secuencialización que produce el

acceso con exclusión, sino por el hecho de que cada thread deberá ejecutar el mecanismo sin importar si es activado o no.

En el Algoritmo 10 se define el algoritmo para el procesamiento de vértices utilizando la estrategia de distribución por vértice. Al ser un enfoque de paralelismo de grano grueso, no existe una diferencia consistente con las estrategias de distribución por comunidad y secuencial. En todo caso la diferencia fundamental se encuentra en el mecanismo de exclusión mutua que se denota por la función $lock()$ que determina el acceso exclusivo para cualquier elemento de las estructuras del vértice ngh, es decir solo el primer thread en bloquear el acceso por medio de la primitiva podrá realizar cambios sobre la etiqueta y la distancia de dicho vértice.

Una vez terminada la revisión y actualización de los valores de un vértice, el thread desbloqueará el acceso y el siguiente thread podrá acceder a los valores ya modificados, evitando por completo las inconsistencias en los resultados.

Algoritmo 10 Algoritmo de difusión multicore: Distribución por vértice

```
1:  procedure spreading_(G, D, src)
2:      queue.push([src, 0])
3:      while queue.empty() do
4:          v, d ← queue.pop()
5:          for ngh ∈ G.A[v] do                    ▷ Para todos los vecinos de v en G.A
6:              nghd ← G.W[v][ngh] + d
7:              Lock(ngh)
8:              if nghd < D[ngh] then
9:                  queue.push([ngh, nghd])
10:                 G.P[ngh] ← G.P[src]
11:             UnLock(ngh)
```

Por otra parte, en el Algoritmo 11 se presenta el algoritmo para el mecanismo de distribución de trabajo para los threads desplegados, el cual sigue una estrategia cíclica. Para este caso se inicia por medio de $startSpreading(...)$, procedimiento que se encarga de desplegar los P threads solicitados para el procesamiento, donde cada uno de estos realizará una invocación al procedimiento $spreading(...)$ recibiendo com o parámetros: la red de entrada G, la cantidad de threads P para el

procesamiento y el índice identificador de cada thread i, para después proceder con la distribución de trabajo.

Una vez todos los threads hayan terminado de procesar la totalidad de los vértices difusores de la red convergen en un punto determinado debido a la primitiva de sincronización.

Algoritmo 11 Algoritmo de difusión multicore: Orquestación de threads

```
1: procedure spreading(G, P, i)
2:     D ← [0...G.nodes]
3:     for node_i ∈ G.spreaders do                ▷ Para todos los vertices difusores
4:         spreading_(G, D, node)
5:         i ← i + P
6:
7: procedure startSpreading(G)
8:     for i ∈ [0, 1, 2, ..P − 1] do
9:         startThread(spreading, [G, P, i])
10:    joinThreads()
```

4.4 Algoritmo para la difusión de etiquetas GPU

Teniendo en cuenta que se requiere de una estrategia que permita paralelización de un recorrido en anchura que asemeje al encontrado en la versión secuencial y asegure el mismo comportamiento, la solución que se plantea para el algoritmo de difusión GPU sigue la siguiente estrategia de operación:

En primer lugar ocurre una clasificación de los vértices de la red por grado utilizando las estrategias mostradas en los Capítulos 23, de manera que los vértices con una cantidad de vecinos similar se dispongan de manera contigua en memoria. La clasificación de los vértices favorece en gran medida el rendimiento de la solución paralela. El siguiente paso será la aplicación de un recorrido en anchura iniciado por los vértices difusores. Para ello se identifican los vértices difusores de la red independientemente de la comunidad a la que pertenezcan, donde cada thread desplegado procesará la vecindad del difusor que se le haya asignado. Es decir que para este caso la distribución de threads en bloques no juega un papel tan importante debido a la estructura de procesamiento de esta solución.

En cualquier caso se desplegará una cantidad de bloques con una cantidad de threads por bloque menor que 1024.

En la Figura 4.4 se muestra la idea general del proceso de difusión GPU llevado a cabo por rondas, donde en la primera ronda los vértices difusores se encuentran marcados y el procesamiento inicia solo para los vértices no marcados que tengan de vecino a uno marcado, es decir en la primera ronda de ejecución solo los vecinos de los difusores actualizarán su valor de distancia y etiqueta. Cada que un vértice no marcado se procesa, al finalizar el procesamiento se establece como un vértice marcado de manera que con el paso de las rondas se cubrirá la totalidad de los vértices de la red. La cantidad de rondas necesarias para procesar todos los vértices de la red está determinado principalmente por el diámetro de la red, que se define como la mayor distancia que existe entre los caminos más cortos para un par de vértices de la red.

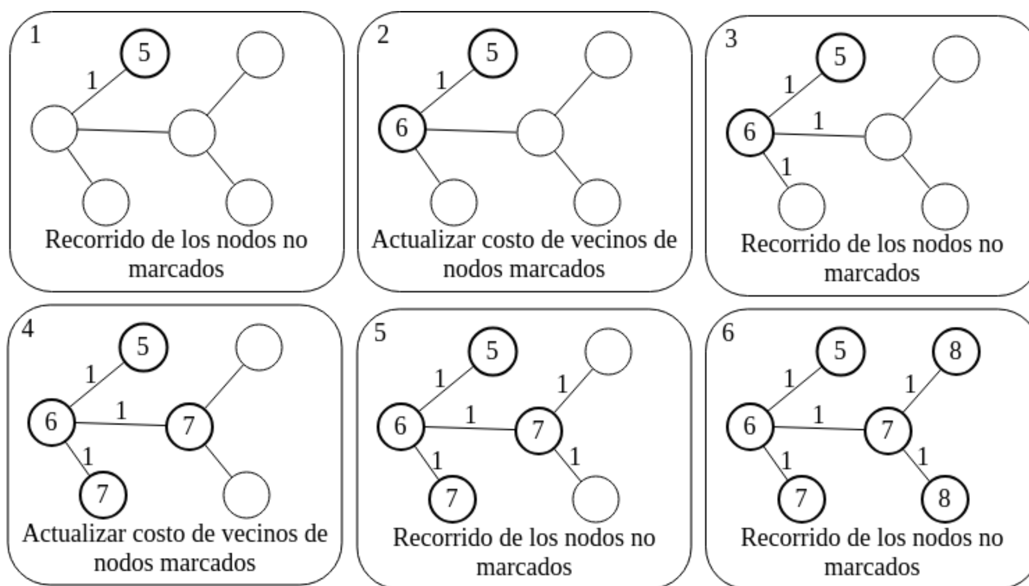

Figura 4.4: Funcionamiento del proceso de difusión sobre un conjunto de vértices.

A diferencia de las soluciones secuencial y multicore del algoritmo de difusión, para la solución GPU se requiere cumplir con ciertas condiciones ligadas a la arquitectura que favorezcan el uso de memoria contigua. A continuación se presentan las estructuras de datos utilizadas en la solución GPU para el algoritmo de difusión, denotando que todas las estructuras son arreglos unidimensionales

simples que deberán cumplir con su equivalente secuencial y multicore. Junto a éstas se incluyen las estructuras A, C, L, R, definidas en el Capítulo 3

- $Array\ W[\ ]$: Representa la lista de pesos asociados a cada vértice con respecto a A, es decir si $A = [1, 0]$ y $L = [0, 1, 2]$, lo que indicaría que existen dos vértices 0 y 1 donde ambos se encuentran conectados, por lo tanto deberá existir un costo de ir desde 0 a 1 y viceversa por lo que en valores unitarios W tendría la forma $W = [1, 1]$.
- $Array\ M[\ ]$: El tamaño de este vector se encuentra en función de la cantidad de vértices de la red y sirve para determinar con un **1** o **0** si un vértice ha sido visitado.
- $Array\ P[\ ]$: Determina la etiqueta de un vértice, su tamaño es determinado al igual que C por la cantidad de vértices en la red.

En la clasificación de vértices por el grado se utiliza la operación de suma de prefijos. En la que se utilizan al menos tres predicados para conseguir grupos de vértices de tres clases. Los vértices con grado menor que **64**, **10,000** y mayores a **10,000**. Una vez se ha realizado la clasificación de los vértices de la red por medio del kernel de suma de prefijos presentado el Capítulo 3. El siguiente paso para el procesamiento, es aplicar el proceso de difusión GPU a cada uno de los grupos de vértices, tomando en cuenta las ventajas que se obtienen al conocer el orden de los vértices agrupados como la sincronización implícita para bloques de 32 threads o menos.

En el Algoritmo 12 se presenta el algoritmo definido para el proceso de difusión GPU. En este kernel de CUDA se muestra el procedimiento que se sigue para la determinación de costos de distancia y propagación de etiquetas desde los vértices difusores hacia los no difusores pertenecientes a una comunidad común. En el proceso de difusión como entrada se tiene una red con pesos en sus aristas, etiquetas de comunidad para cada vértice y una etiqueta para cada uno de los vértices difusores. Por otra parte como resultado se deberá obtener la misma red donde todos los vértices pertenecientes a ella tienen una etiqueta propagada desde alguno de los difusores, así como el costo de ir desde cualquier vértice hacia el difusor más cercano.

El primer paso a realizar en el kernel para la difusión es la determinación de los índices, en este caso se utiliza una distribución de un thread para un vértice por lo que la orquestación de los threads utiliza la distribución que ocurre por defecto donde los bloques y threads son desplegados sobre el eje x en $blockDim$,$blockIdx$, $threadIdx$.

Para el siguiente paso se revisa si el vértice de cada thread no se encuentra marcado utilizando M para ello, donde los vértices se marcan con 1 o 0 para indicar si han sido evaluados. En la primera ejecución del kernel solo los vértices difusores se encuentran marcados pero los vértices que serán procesados serán los no marcados, de este modo para cada uno de estos vértices se determina el inicio y final de sus vecinos en A por medio de str y end para después recorrer A en busca de vértices marcados. Cuando un vértice marcado es encontrado se calcula el costo de distancia en D desde dicho vértice hasta el que se encuentra en procesamiento, del mismo modo ocurre la asignación de la etiqueta. Siguiendo este modo de operación los primeros vértices en ser evaluados serán los vecinos de los difusores que en la primera ronda serán marcados por lo que en la siguiente, los vecinos de los vecinos serán evaluados y del mismo modo hasta completar la totalidad de la red. La cantidad de rondas necesarias para realizar un recorrido completo de los vértices esta determinado por el diámetro de la red, aunque esto no es necesariamente cierto, debido a que la propagación de las etiquetas se encuentra limitada a las comunidades por lo que en realidad la cantidad de rondas deberá ser igual al mayor diámetro encontrado entre las comunidades.

Resumen

En este Capítulo se describieron las estrategias propuestas para resolver el problema de difusión de etiquetas sobre una red compleja. Así como los algoritmos diseñados para las arquitecturas multicore y GPU. En el siguiente Capítulo se abordan los algoritmos para la determinación de influencia de etiquetas.

Algoritmo 12 Algoritmo de difusión GPU

```
1:
2: kernel spreading(A, D, L, P, M, W, v)
3:     index ← blockIdx.x ∗ blockDim.x + threadIdx.x
4:     nodeindex ← index
5:     while nodeindex < v do
6:         if !M[nodeindex] then                          ▷ Si es un vertice no marcado
7:             str ← L[nodeindex]
8:             end ← L[nodeindex + 1]
9:             mds = D[nodeindex]
10:            cst = mds
11:            for i ← str; i < end; i + + do
12:                if M[A[i]] then                        ▷ Si es un vertice marcado
13:                    dst ← D[A[i]] + W[i]
14:                    if dst < mds then
15:                        cst ← dst
16:                        M[nodeindex] = 1
17:                        P[nodeindex] ← P[A[i]
18:        D[nodeindex] ← cst
19:        nodeindex ← nodeindex + gridDim.x ∗ blockDim.x
```

5

Algoritmos paralelos para el proceso de influencia

5.1 Introducción

Como una segunda etapa del proceso de difusión de etiquetas se presenta el proceso de influencia, en el cual se busca un refinamiento de la asignación de etiquetas establecidas por el proceso de difusión. Para el proceso de influencia se espera como entrada el resultado de la difusión, es decir, la red con una etiqueta para cada vértice. Como resultado del proceso de influencia se espera una asignación más precisa de las etiquetas asignadas por el proceso de difusión.

Al igual que en el proceso de difusión que requiere que la red de entrada cumpla con ciertas propiedades como la partición de los vértices por comunidades y la definición de los vértices difusores. Para el proceso de influencia se requiere que las aristas de la red contengan información del camino, en este caso basta con un valor de costo para cada enlace. Se requiere que todos los vértices de la red tengan el costo del camino hacia el vértice difusor del cual proviene la etiqueta asignada. Tal valor del costo al difusor es calculado por el proceso de difusión y almacenado para su posterior uso

en el proceso de influencia.

El proceso de influencia utiliza una función de afinidad que determina el valor de influencia de las etiquetas en el vecindario de un vértice de la red. Es decir que para cada uno de los vértices de la red se determinan las etiquetas presentes en el primer nivel de vecindad y se obtiene un valor ponderable por medio de la afinidad hacia cada grupo de vértices que comparten la misma etiqueta. La etiqueta del grupo con un valor mayor de afinidad reemplazará a la del vértice en evaluación.

5.2 Algoritmo de influencia secuencial

El algoritmo de influencia realiza una nueva asignación de etiquetas con base en las etiquetas anteriores y la fracción de ellas que exista en las vecindades de los vértices.

Para cada uno de los vértices se cuenta la cantidad de etiquetas diferentes que existen en su vecindad y se determina la fracción que representa cada uno de los grupos de vértices que comparten la misma etiqueta. Dicha fracción representa un valor importante para determinar que etiqueta debería asignarse al vértice en procesamiento, ya que si la fracción mayoritaria de los vecinos de un vértice comparten la misma etiqueta significa que ese grupo de vértices son cercanos en costo al mismo difusor. Asumiendo que el grupo con mayor cantidad de vértices son cercanos al mismo difusor es probable que el vértice en cuestión también se encuentre cerca, ya que en los mejores casos el costo de ir desde el vértice en evaluación hacia el difusor del grupo mayoritario es a través de alguno de los vecinos en ese grupo.

Debido a que el proceso de influencia es una segunda etapa de lo que se puede considerar como el proceso de propagación de etiquetas, es posible utilizar el costo que tiene el camino de cada vértice hacia su difusor más cercano.

El proceso de influencia es un proceso iterativo que refina la asignación de las etiquetas en cada iteración. A pesar que de que tiene un mayor costo computacional ya que se requiere la repetición de un proceso sobre un conjunto de datos, al aprovechar los valores precalculados en el proceso de

difusión, las operaciones realizadas terminan siendo poco costosas computacionalmente.

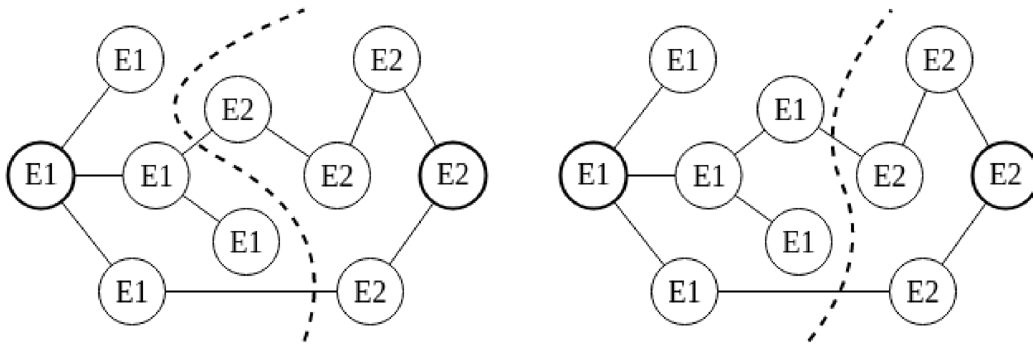

Figura 5.1: Proceso de influencia

Por otra parte, se requiere un criterio de parada para el algoritmo iterativo del proceso de influencia. Deteniendo el algoritmo cuando los cambios entre iteración hayan cesado. Es necesario considerar la situación en donde los cambios de una iteración son revertidos en la siguiente, por lo que no hay una convergencia a cero. Una manera de solventar este problema limitar el número de iteraciones. En la Figura 5.1 se muestran las entradas y salidas del algoritmo de influencia, presentándose la red de entrada marcada con una etiqueta para cada vértice y como resultado se obtiene la misma red con una mejor estructuración de las etiquetas asignadas.

La base fundamental del algoritmo de influencia es la función de afinidad, la cual determina un valor medible entre un vértice y las etiquetas que se encuentran entre sus vecinos, el cual permite definir cual de las etiquetas cercanas al vértice es la de mayor afinidad.

El valor de la afinidad de un vértice con respecto a los vecinos del mismo, se calcula por medio de la Ecuación 5.1.

$$A(v, g_p) = \frac{|g_p|}{|neis_v|} \left(\sum_{w \in neis_p} hrp_{v,w} + hrp_{w,spr_p} \right) \tag{5.1}$$

donde:

$neis_v$ son los vecinos de v.

g_p el grupo de vecinos de v que **comparten** la misma etiqueta p.

P es el conjunto de etiquetas diferentes presentes en los vecinos.

$hrp_{v,w}$ el **costo** de ir desde el vértice v al w.

spr_p el difusor origen de la etiqueta p.

Con el propósito de reducir los costos para el cálculo de la función de afinidad, las estructuras de datos de este algoritmo consideran cálculos realizados durante el proceso de difusión.

En el Algoritmo 13 se presenta la estructura de datos para el algoritmo de influencia. La estructura mostrada es una extensión de la presentada en el Capítulo 4, a la cual se agrega un atributo D, que define una lista con el valor del costo de ir desde cualquiera de los vértices de la red al vértice difusor más cercano. Este valor se calcula en el algoritmo de difusión y se reutiliza en el algoritmo de influencia.

El primer paso del algoritmo de influencia es separar los vértices de la red en difusores y no difusores. El siguiente paso requiere de un recorrido por todos los vértices no difusores, donde para cada uno de los vértices se realizará un agrupamiento de los vértices vecinos. Esto implica que se terminará realizando un recorrido completo de la red, donde para cada vértice se revisa la cantidad de vecinos que tiene, y las etiquetas que se encuentran entre estos.

A diferencia del algoritmo de difusión en el que solo se contemplan los vértices pertenecientes a la comunidad de vértice en evaluación, en el algoritmo de influencia no se tiene esta consideración. Es decir que todos los vértices no difusores se evalúan por igual.

En el recorrido realizado a través de los vecinos de cada vértice se obtienen dos de los elementos principales para el cálculo de la afinidad, que son: La fracción de vértices para cada etiqueta y la suma de los costos hacia los difusores junto con el valor de enlace entre los vecinos y el vértice en evaluación.

En la Figura 5.2 se presenta la idea general de algoritmo de influencia, donde la parte central es el agrupamiento de los vértices por etiqueta y posteriormente el cálculo del valor de afinidad hacia cada grupo.

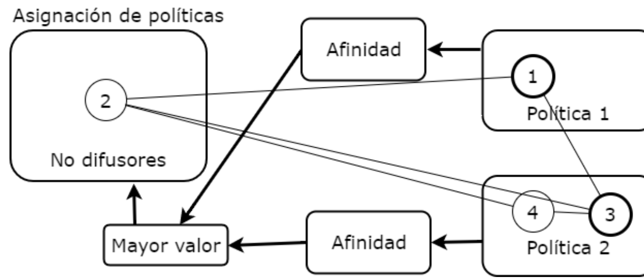

Figura 5.2: Proceso para el cálculo de la influencia.

Una vez se ha calculado el valor de afinidad para cada uno de los grupos de vértices se procede a realizar la selección de la etiqueta con el mayor grado de afinidad, la cual reemplazará a la etiqueta asignada por el algoritmo de difusión. Con lo cual se procede con el siguiente vértice no difusor del total de los vértices de la red bajo el mismo esquema de operación.

Al ser un algoritmo iterativo, el algoritmo de influencia requiere seguir operando a los vértices no difusores e intercambiando las etiquetas de estos hasta que ya no haya cambios de etiqueta o se alcance un número límite de iteraciones. Aunque existen los casos con resultados oscilatorios, es decir aquellos en los cuales las etiquetas conmutan entre los mismos valores de manera indefinida. Para estos casos es posible determinar la parada del algoritmo por medio de ventanas de cambios o de otro modo por el valor de modularidad de los grupos de etiquetas formados entre las comunidades, donde la modularidad indica la calidad de las conexiones en un grupo y se calcula con la Ecuación 5.2.

$$Q = \sum_{C \in P} \frac{E_C}{m} - \left(\frac{k(C)}{2m}\right)^2 \tag{5.2}$$

donde:

P es el conjunto de particiones (etiquetas diferentes en todo el grafo).

E es el número de aristas intra-comunidad para la comunidad C.

m es el número total de aristas.

$k(C)$ el grado de la comunidad C esto es, la sumatoria de los grados de todos los vértices en C.

En el Algoritmo 13 se presenta el algoritmo de influencia secuencial que sigue el comportamiento descrito. Como entradas del algoritmo se encuentra G como la estructura de datos general de la red junto a sus atributos e $iterations$ que indica la cantidad de iteraciones máxima para el algoritmo.

Algoritmo 13 Algoritmo de influencia secuencial

1: **struct** $Network$
2: $Array[\,][\,]A$
3: $Array[\,][\,]W$
4: $Array[\,]C$
5: $Array[\,]P$
6: $Array[\,]S$
7: $Array[\,]D$
8:
9: **function** $Influence(G, iterations)$
10: $distances \leftarrow getDistances(G)$
11: **while** $iterations$ **do**
12: **for** $n \in G.nodes$ **do** ▷ Para todos los vértices en G
13: **if** $G.spreader[n] = false$ **then**
14: $groups[][] \leftarrow groupByPolicy(G.neighbors[n])$
15: $affinity[] \leftarrow getAffinity(groups, n, G.D)$
16: $G.policy[n] \leftarrow maxAffinityPolicy(affinity)$
17: $iterations \leftarrow iterations - 1$
18: **return** G

5.3 Algoritmo de influencia multicore

Como parte fundamental del algoritmo de influencia multicore se encuentra la repartición de los bloques de trabajo. El algoritmo de influencia seguirá un enfoque de distribución por vértice en la cual se puede obtener una distribución menos irregular, únicamente limitada por la cantidad de arcos por vértice. Es decir que la distribución de trabajo será completamente dependiente de la estructura de la red y no del tamaño de las comunidades como sugiere una distribución por comunidad.

Siguiendo el procedimiento del algoritmo de influencia secuencial para obtener el valor de afinidad

de un vértice, no se requiere modificar los valores de los vecinos. Lo cual es un indicador de que no existe una condición de carrera que puede perjudicar el procesamiento en paralelo de los vértices.

Ya que en el algoritmo de influencia no se encuentran condiciones de carrera obvias, obtener la versión paralela es un proceso simple en la cual se utiliza el algoritmo de influencia secuencial y se aplica sobre un conjunto de vértices.

El particionamiento usado para el algoritmo de influencia es cíclico, de modo que distribuye los elementos a procesar uno a uno tomando como base el índice del thread. De esta manera que se espera que la distribución de trabajo total se regularice, a diferencia de lo que ocurre con el particionamiento estático. También es posible escalar el tamaño de la entrada de datos de manera indefinida, sin presentar problemas de sincronización como ocurriría con un particionamiento dinámico.

En el Algoritmo 14 se presenta el algoritmo de influencia multicore que tiene las siguientes entradas: En primer lugar G definido como el grafo o red de entrada, el segundo parámetro es la cantidad de threads T disponibles para el paralelismo. En tercer lugar se encuentra i que en este caso define al índice único de cada thread y con el cual se realiza la distribución de los vértices. Por último se encuentra $iterations$ que define la cantidad de iteraciones que se aplicará el algoritmo de influencia.

Algoritmo 14 Algoritmo de influencia multicore

1: **procedure** $influence(G, T, i, iterations)$
2:     **while** $iterations$ **do**
3:         $i \leftarrow index$
4:         **for** $node_i \in G.nodes)$ **do** ▷ Para todos los vértices no difusores
5:             **if** $G.spreader[i] = false$ **then**
6:                 $groups[][] \leftarrow groupByPolicy(G.neighbors[node_i])$
7:                 $affinity[] \leftarrow getAffinity(groups, node_i)$
8:                 $G.policy[i] \leftarrow maxAffinityPolicy(affinity)$
9:             $i \leftarrow i + T$
10:         $syncthreads()$
11:         $iterations \leftarrow iterations - 1$
▷ *Actualizar sobre copia*
12: ▷ *de $G.policies[i]$(8) para*
13: ▷ *procesamiento síncrono*

Por otra parte, es necesario agregar una primitiva de sincronización al final de cada iteración para evitar condiciones de carrera entre los threads. En el Algoritmo 15 se presenta el algoritmo de invocación, el cual determinará la cantidad de threads y los iniciará con sus respectivos argumentos. En este algoritmo los parámetros son G como la red de entrada definida sobre la estructura de datos presentada en el Algoritmo 13 y T como la cantidad de procesadores disponibles para el paralelismo.

Algoritmo 15 Algoritmo de influencia multicore: Orquestación de threads

```
1: procedure startInfluence(G, T)
2:     for i ∈ [0, 1, 2, ..T − 1] do
3:         startThread(influence, [G, T, i])
4:     joinThreads()
```

5.4 Algoritmo de influencia GPU

Para un aprovechamiento adecuado de la jerarquía de memoria en los GPUs, se deben utilizar estructuras de datos en memoria contigua alineadas con los threads que son invocados. Para el algoritmo de influencia GPU, se cuenta con las estructuras de datos A, W, C, L, M, P, R y R' presentadas en los Capítulos 3 y 4.

Al igual que para los algoritmos GPU presentados en Capítulos anteriores, en el algoritmo de influencia GPU también se requiere de la clasificación de los vértices de la red por su grado.

Una vez concluido el proceso de difusión que determina la asignación inicial de etiquetas, el proceso de influencia reasignará las etiquetas de manera iterativa hasta alcanzar un estado en el que los grupos de etiquetas internos a las comunidades alcance el valor máximo de afinidad con respecto a los vecinos de cada vértice.

La afinidad como se muestra en la Ecuación 5.1 se compone principalmente de dos términos, la sumatoria del costo de ir a los difusores más cercanos y el costo de los enlaces hacia cada vecino. Por otra parte, la fracción de etiquetas que aparecen dentro de los vecinos de cada vértice no difusor. Para ello se definen tres Kernels principales: El primero se encarga de calcular el valor de la sumatoria, el

Algoritmo 16 Kernel de sumatoria: influencia

1: **kernel** $sum(A, D, I, R, S, W, v)$
2: $index \leftarrow blockIdx.x * blockDim.x + threadIdx.x$ ▷ Un thread por vertice
3: $nodeindex \leftarrow index$
4: $blocks \leftarrow gridDim.x$
5: **while** $nodeindex < v$ **do**
6: **if** $!S[nodeindex]$ **then**
7: $str \leftarrow I[nodeindex]$
8: $end \leftarrow I[nodeindex + 1]$
9: $res \leftarrow 0$
10: **for** $nghr \leftarrow str; nghr < end; nghr + +$ **do**
11: $res \leftarrow res + W[nghr] + D[A[nghr]]$
12: $R[nodeindex] \leftarrow res$
13: $nodeindex \leftarrow nodindex + gridDim.x * blockDim.x$

segundo realizará el conteo de las etiquetas y obtendrá la fracción de cada una de ellas con respecto a los vértices no difusores. Por último, el tercer Kernel se encarga de reunir los resultados parciales de los Kernels anteriores.

En el Algoritmo 16 se presenta el funcionamiento del Kernel para la sumatoria inicial. En el cual se realiza una distribución de las listas de adyacencia para cada threads de cada bloque. Es decir que se asignará un thread para cada vértice no difusor. Cada thread realiza un recorrido lineal sobre la lista de adyacencia de su vértice asignado, sumando los valores de W y D. Los resultados de la sumatoria se almacenarán en R y serán usado por el Kernel de integración de resultados.

Como segunda parte del procesamiento en el algoritmo de influencia se encuentra el cálculo de la fracción de etiquetas en las vecindades de los vértices no difusores. Para ello se despliega un kernel especial presentado en 17. El cual, al igual el kernel anterior, realiza un recorrido lineal sobre los vecinos de los vértices no difusores, realizando un conteo individual por thread de las etiquetas presentes en el vecindario del vértice y determinando el valor de afinidad para cada fracción. Finalmente, una vez que se hayan obtenido los resultados de los kernels de sumatoria y del cálculo de fracciones se utiliza el tercer kernel presentado en el Algoritmo 18, el cual se encargará de actualizar los valores de afinidad obtenidos en el kernel de obtención de fracciones en los casos en lo que la

Algoritmo 17 Kernel para obtención de fracciones: influencia

```
1: kernel fraction(A, I, P, R, S, W, v)
2:     index ← blockIdx.x * blockDim.x + threadIdx.x          ▷ Un thread por vértice
3:     nodeindex = blockIdx.x
4:     blocks ← gridDim.x
5:     while nodeindex < v do
6:         if !S[nodeindex] then
7:             str ← I[nodeindex]
8:             end ← I[nodeindex + 1]
9:             res ← 0, wgh ← 0
10:            idx ← str + threadIdx.x
11:            lgh ← end − str
12:            for i ← str; i < end; i + + do  ▷ Para todos los elementos de la lista de adyacencia
13:                if P[A[i]] = P[A[idx]] then
14:                    res ← res + 1
15:                    wgh ← wgh + W[i]
16:            syncthreads()
17:            if idx < end then
18:                R[idx] ← res * lgh * wgh
19:            nodeindex ← nodeindex + blocks
```

afinidad deba actualizarse. Este kernel presenta una distribución de hilos en relación de un thread por vértice, por lo que cada bloque despachará múltiples vértices de manera lineal, esperando que con la suficiente carga de trabajo se logre aprovechar de mejor manera el paralelismo masivo disponible en el GPU.

Resumen

En este Capítulo se han presentado diferentes versiones de el algoritmo de influencia para las arquitecturas multicore y GPU. En el siguiente Capítulo se presentan los resultados más importantes de los algoritmos de detección de comunidades, difusión e influencia.

Algoritmo 18 Kernel de actualizado: influencia

```
1:  kernel set(A, I, P, R, S, W, v)
2:      index     ← blockIdx.x * blockDim.x + threadIdx.x        ▷ Un thread por vértice
3:      nodeindex ← index
4:      while nodeindex < v do
5:          if !S[nodeindex] then
6:              str  ← I[nodeindex]
7:              end  ← I[nodeindex + 1]
8:              res  ← 0
9:              max ← 0
10:             mid  ← nodeindex
11:             for nghr ← str; nghr < end; nghr + + do
12:                 if R[nghr] > max then
13:                     max ← R[nghr]
14:                     mid  ← nghr
15:             P[nodeindex] ← P[A[mid]]
                                                  ▷ Actualizar en copia de P para
16:                                                         ▷ procesamiento síncrono
17:         nodeindex ← nodeindex + gridDim.x * blockDim.x
```

6

Experimentación y resultados

En este capítulo se presentan los resultados importantes observados con la implementación de los algoritmos para la detección de comunidades, difusión e influencia, que han sido analizados en secciones anteriores. Además se presenta un caso de estudio de difusión de políticas de seguridad como etiquetas. Los grafos utilizados para las pruebas se encuentran en el repositorio [31] y cuentan con algunas características expresadas sobre la tabla 6.1, entre las que se encuentran la cantidad de vértices y aristas de la red, la cantidad de comunidades encontradas por el algoritmo de comunidades descrito en este trabajo, así como el diámetro de la red. En caso de que la red no sea conexa, es decir que existan vértices o grupos de vértices no conexos, se asume como diámetro de la red a aquel que se encuentre en la componente gigante de la red, siendo el componente gigante de la red una de las propiedades descritas en las redes complejas.

Otra propiedad descrita en la Tabla 6.1 es la cantidad de difusores en la red, siendo los vértices difusores aquellos que contendrán las políticas de seguridad iniciales y que serán propagadas posteriormente por los algoritmos de difusión e influencia. El proceso para determinar la cantidad de

vértices difusores, así como la selección de ellos no se encuentra en el dominio de este trabajo, sin embargo, en algunos casos de las redes utilizadas se han determinado los vértices difusores bajo el siguiente criterio.

En primer lugar, determinar la estructura de comunidades de la red por medio del algoritmo de detección de comunidades descrito en este trabajo. Para el resto de las redes a las que no se haya aplicado el proceso de selección mencionado, se asume que ya han pasado por un proceso de selección similar para la determinación de los vértices difusores.

Nombre	Vértices	Aristas	Tipo	Comunidades	Difusores	Diámetro
Chicago	1,467	**1,298**	No dirigido	283	440	9
Ttk4.0	36,954	**604,258**	No dirigido	147,816	11,086	36
Amazon	334,863	**925,872**	No dirigido	31,205	100,458	47
Ttk4.1	52,212	**1,203,818**	No dirigido	51,813	15,663	43
Skitter	1,696,415	**11,095,298**	No dirigido	67,878	508,924	31

Tabla 6.1: Propiedades de los conjuntos de datos utilizados

6.1 Datos de configuración

Los datos de entrada o conjuntos de parámetros indican los estados iniciales y valores especiales que se requieren para el funcionamiento de un mecanismo, proceso o algoritmo. Estos determinan el tamaño del problema y en algunas ocasiones pueden afectar directamente el comportamiento de una secuencia de instrucciones.

6.1.1 Algoritmos para la detección de comunidades

En este caso, solo se considerarán aquellos que afectan directamente el tamaño del problema y son comunes entre las versiones de los algoritmos. Los parámetros requeridos para el la detección de comunidades se describen a continuación:

1. **Red o Grafo:** En primera instancia se encuentran los datos fundamentales para la detección de comunidades, siendo éste la red de entrada del tipo no dirigida que define la cantidad de vértices y aristas. Para este caso no se requiere que la red cuente con propiedades especiales basta con las relaciones entre los vértices para el correcto funcionamiento de los algoritmos.

2. **Iteraciones:** Como segundo dato de entrada se encuentra un valor numérico que indica la cantidad de iteraciones que se requiere para la culminación del algoritmo.

Debido a que se requiere la comparativa entre las soluciones sobre las principales arquitecturas paralelas, la cantidad de iteraciones para los experimentos se fijará en un valor determinado por la observación y la convergencia de los resultados obtenidos para los distintos conjuntos de datos. Por lo cual para el caso de los algoritmos de detección de comunidades el valor de iteraciones es fijo a **50** iteraciones para ambas soluciones, multicore y GPU. Siendo 50 iteraciones un valor promedio de las iteraciones requeridas para alcanzar un punto de estabilidad en la detección de comunidades para los conjuntos de datos utilizados.

6.1.2 Algoritmos para la propagación de etiquetas por difusión

Para el caso de los algoritmos de difusión de etiquetas los datos de entrada se limitan solo a la red o grafo ponderado. Sin embargo, a diferencia del algoritmo de detección de comunidades los algoritmos de difusión requieren algunas propiedades que deberán estar definidas inicialmente en la red. Tales características incluyen a los vértices **difusores** que describen a vértices especiales que serán las fuentes de difusión, conteniendo una etiqueta que será propagada a través de la comunidad. Otra propiedad requerida es la etiqueta de comunidad, la cual determinará la pertenencia de un vértice a cierta comunidad. Debido a que el proceso de definición de vértices difusores de la red no se contempla dentro del marco de este trabajo se trabajará asumiendo ciertas cantidades de vértices difusores sobre las redes utilizadas respetando algunas restricciones detalladas a continuación.

- Para cada comunidad definida sobre una red con un grado mayor a tres, se seleccionara de

manera aleatoria el 33 % de éstos vértices como difusores. Entendiendo como grado de la comunidad a la cantidad de vértices que la conforman.

- Se asigna de manera aleatoria un vértice difusor para las comunidades de grado 2.
- Las comunidades aisladas de grado uno no se contemplan en el procesamiento.

6.1.3 Algoritmos para la propagación de etiquetas por influencia

En el caso de los algoritmos de influencia los datos de entrada no solo se definen como parámetros del algoritmo o valores configurables sino que se utilizan valores precalculados en anteriores etapas, por lo que en los algoritmos de influencia existen accesos a tablas de datos precalculados. Los datos reutilizados se obtienen del algoritmo de difusión en el cual se conservan los valores de distancias desde cada vértice al difusor más cercano y que se utilizan para el cálculo de la afinidad.

El único requisito en las propiedades de la red, es la existencia de un atributo que describa a la etiqueta la cual ha sido asignada por el algoritmo de difusión.

Dentro del algoritmo de influencia solo es posible afectar los resultados por medio de las iteraciones requeridas por el algoritmo hasta que alcanza un punto de estabilidad, sin embargo, determinar el punto de parada puede llegar a presentar un costo relativamente alto con respecto al del propio algoritmo, por lo que en las comparativas que se registran en secciones posteriores se utiliza un valor fijo limitado a **103** iteraciones, siendo un valor promedio de la cantidad de iteraciones requeridas por el algoritmo de difusión para alcanzar un punto de convergencia sobre los conjuntos de datos utilizados.

6.2 Métricas de evaluación

Para determinar el valor de las soluciones descritas es necesario utilizar métricas de evaluación y definir que se busca evaluar. En el caso de los algoritmos paralelos la ganancia que se busca

obtener es en tiempo de ejecución, es decir reducir el tiempo que toma realizar una tarea siguiendo una estrategia secuencial utilizando una estrategia paralela. Entre las métricas a utilizar para la evaluación se encuentran: El tiempo de **ejecución**, como una de las medidas de comparación más comunes y que describe al tiempo que toma realizar una tarea computacional, y el **throughput** que define la cantidad de trabajo realizada por unidad de tiempo, en este caso la medida de troughput utilizada serán los MEPS(Millones de aristas atravesadas por segundo) que dará una estimación del trabajo realizado por cada algoritmo.

Por otra parte la métrica utilizada para medir el desempeño de los algoritmos paralelos es la aceleración que determina que tan rápido es un algoritmo paralelo con respecto a su versión secuencial y que puede obtenerse de a través de la Ecuación 6.1. Donde la aceleración para un determinado algoritmo E, es igual al cociente entre el tiempo de ejecución secuencial $sequential_t$ y el tiempo de ejecución paralelo $parallel_t(p)$ con p procesadores.

$$A(E) = sequential_t / parallel_t(p) \tag{6.1}$$

El throughput por otra parte se calcula como la cantidad de operaciones realizadas por un algoritmo E, ya sea secuencial o paralelo dividido por el tiempo de ejecución y nuevamente dividido por **1,000,000**. Al contrario de lo que pasa con el tiempo de ejecución para medir la aceleración respecto al throughput se utiliza la expresión descrita en la Ecuación 6.2. Donde la ganancia obtenida en throughput para un algoritmo E es igual al throughput obtenido para p procesadores $throughput(p)$ dividido por el throughput obtenido por la versión secuencial $throughput_{sequential}$.

$$T(E) = throughpt(p) / throughput_{sequential} \tag{6.2}$$

Para realizar una comparativa razonable entre soluciones implementadas sobre diferentes arquitecturas de cómputo, se utilizará la medición de las métricas comparativas con base en el número total de **operaciones requeridas** por el problema. Esto quiere decir que la variable

para la comparación que determinará los resultados será el tiempo de ejecución. Se plantea esta estrategia de comparación debido a que los algoritmos paralelos utilizan técnicas de procesamiento que no son siempre eficientes en trabajo, lo que representa una cantidad superior de operaciones realizadas para obtener el mismo resultado. Siguiendo esa estrategia se establece que la cantidad de operaciones **requeridas** para determinar las comunidades de una red G estará dada por la función $f(e, v, n) = (2e + v) \cdot n$, donde e es la cantidad de aristas, v es la cantidad de vértices ambos pertenecientes a G, mientras que n determina la cantidad de iteraciones que realizará la solución.

6.3 Algoritmos paralelos para la detección de comunidades

Dentro del marco de las aplicaciones paralelas una de las principales suposiciones es que con el uso incremental de la capacidad de cómputo el desempeño de la aplicación mejore. Sin embargo, ésta suposición no siempre es acertada ya que pueden existir ciertos factores externos o internos a la aplicación que determinan parte de su comportamiento. De manera similar ocurre en este caso ya que se conoce que el orden de procesamiento de los vértices de la red influye directamente en la distribución de trabajo, que a su vez afecta el tiempo de ejecución de los algoritmos.

En la Figura 6.1 se observa el comportamiento del algoritmo LP implementado sobre una arquitectura multicore para los distintos conjuntos de datos mostrados en la Tabla 6.1, en la cual se observa un patrón de comportamiento regular, donde a medida que se incrementa la cantidad de procesadores el tiempo de ejecución disminuye.

Esto es mayormente visible para el conjunto de datos Skitter, donde se puede observar el máximo del tiempo de ejecución en los **46.24** segundos utilizando 1 núcleo de procesamiento y culminando en **10.4** segundos utilizando 10 núcleos. De cualquier forma se puede observar que de manera general el comportamiento de los conjuntos de datos es similar en todos los casos, presentando siempre una pendiente negativa hasta alcanzar la máxima capacidad de cómputo.

Por otra parte en la Figura 6.2 se presenta el tiempo de ejecución obtenido para el algoritmo LP GPU, en el cual se observa el tiempo máximo sobre Skitter en los **20.1** segundos y **5.2** en el mínimo, estableciendo una diferencia de **26.3** segundos entre ambos algoritmos para la mínima unidad de procesamiento establecida, mientras que para la máxima capacidad de paralelismo se observa una diferencia de **5.2** segundos a favor del algoritmo GPU.

A diferencia del tiempo de ejecución el throughput muestra la relación existente entre la cantidad de operaciones requeridas para procesar cada entrada y el tiempo que toma procesarla con p unidades

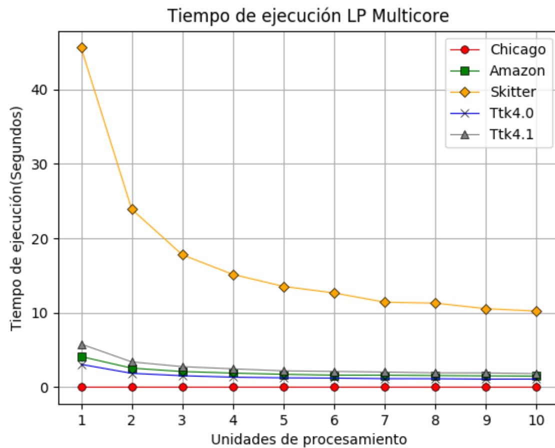

Figura 6.1: Tiempo de ejecución para el algoritmo LP multicore.

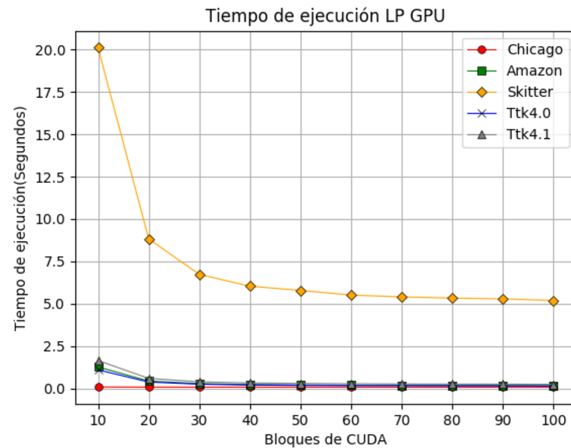

Figura 6.2: Tiempo de ejecución para el algoritmo LP GPU.

de procesamiento. Por lo tanto a medida que se incrementa la capacidad de paralelismo se espera que el tiempo de ejecución disminuya y el throughput se incremente. En la Figura 6.3 se muestra el throughput del algoritmo LP multicore, alcanzando los valores más altos cerca de los **117** MEPS sobre los conjuntos de datos de mayor tamaño. Mientras que para el caso del algoritmo GPU que se presenta en la Figura 6.4 se observa que el punto máximo se encuentra en los **529** MEPS.

En ambas Figuras 6.3,6.4 se observa que el comportamiento para cada conjunto de datos difiere de manera significativa, lo cual es un indicativo que la distribución de trabajo varía con respecto al conjunto de datos. Por otra parte se puede observar que en el caso del algoritmo multicore, el crecimiento del throughput se relaciona directamente al tamaño del conjunto de datos, por lo que se podría asumir que al aumentar el tamaño de la red de entrada se produce una mejora en la distribución de trabajo.

La mejora en el rendimiento que se obtiene al implementar algoritmos paralelos se describe como la aceleración, la cual determina la ganancia que se obtiene de un algoritmo paralelo con respecto al secuencial que resuelve la misma tarea. En este caso la aceleración se calcula con respecto al tiempo, por lo que presentará un indicativo de cuantas veces es más rápida una solución paralela (multicore,

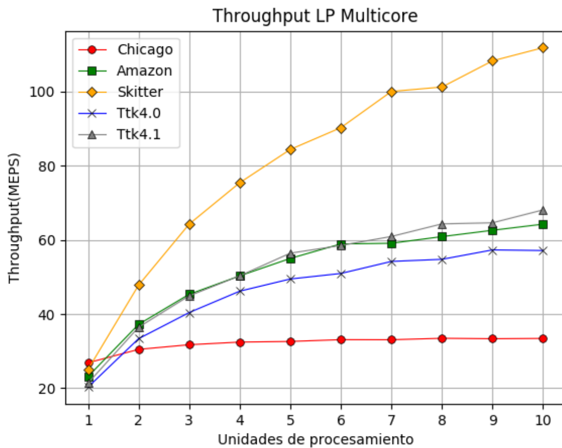

Figura 6.3: Throughput para el algoritmo LP multicore.

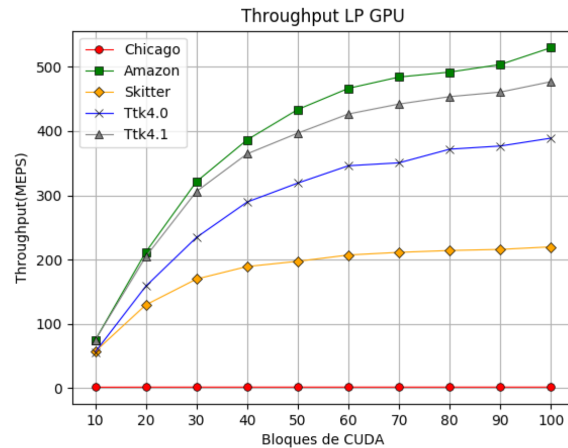

Figura 6.4: Throughput para el algoritmo LP GPU.

GPU) con respecto a una secuencial.

En la Figura 6.5 se presentan las aceleraciones para todos los conjuntos de datos utilizando el algoritmo LP multicore con respecto al algoritmo secuencial, obteniendo una aceleración máxima de **4.5**x con **10** núcleos. Siendo un aprovechamiento limitado de la capacidad de paralelismo disponible, esto debido a la distribución irregular de la carga de trabajo. De igual manera en la Figura 6.6 se presenta la aceleración para el algoritmo GPU. En éste caso se observa que la aceleración es mayor que la observada en el algoritmo multicore, encontrando el punto máximo en los **23**x. A pesar de esto para algunos conjuntos de datos se puede notar que en cierto punto la aceleración converge hacia un valor mientras se incrementa el paralelismo. Esto es causado por dos razones principales: Se ha alcanzado la capacidad máxima de paralelismo que se puede ofrecer y se inicia el encolado de trabajo, o los costos de organización, configuración, manejo de memoria, y despliegue del algoritmo GPU superan el costo del procesamiento de la red de entrada.

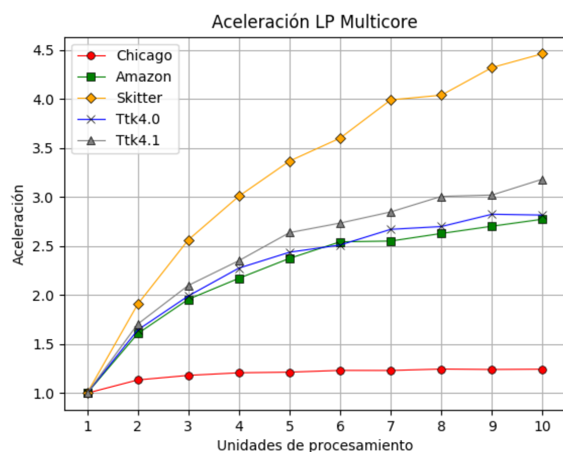

Figura 6.5: Aceleración algoritmo LP Multicore

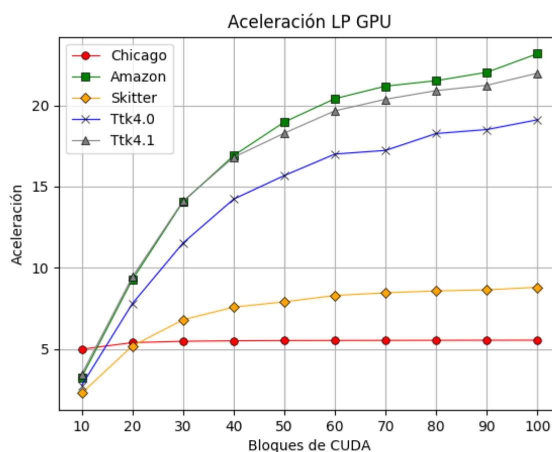

Figura 6.6: Aceleración algoritmo LP GPU

6.4 Algoritmos paralelos para la propagación de etiquetas por difusión

Dentro de los algoritmos paralelos presentados para abordar la propagación etiquetas el algoritmo de difusión se encuentra como paso posterior al algoritmo LP. Al igual que en el algoritmo LP se presenta el tiempo de ejecución, el throughput y la aceleración como las métricas utilizadas.

En la Figura 6.7 se observa el comportamiento en el tiempo de ejecución del algoritmo de difusión multicore. Al igual que en el caso anterior se puede observar una tendencia a la baja conforme se incrementa el paralelismo. Sin embargo, para éste caso se observa para la mayoría de los conjuntos de datos el comportamiento es similar, lo cual indica que la distribución de trabajo es más estable que en el caso del algoritmo LP. Esto ocurre debido a que en el algoritmo de difusión multicore la distribución del trabajo no se determina solo por los vértices como sucede con el algoritmo LP, sino con respecto a los vértices difusores que se encuentran distribuidos en las comunidades de la red, sin limitarse a las vecindades de cada vértice aunque permaneciendo dentro del perímetro de la comunidad. Lo que propicia una distribución más granular que se ve reflejada en los tiempos de procesamiento. Por otro lado en la Figura 6.8 se presentan los tiempos para el algoritmo de difusión GPU, donde se observa

una diferencia sustancial entre las curvas de los conjuntos de datos, a diferencia de lo que ocurre con el algoritmo multicore.

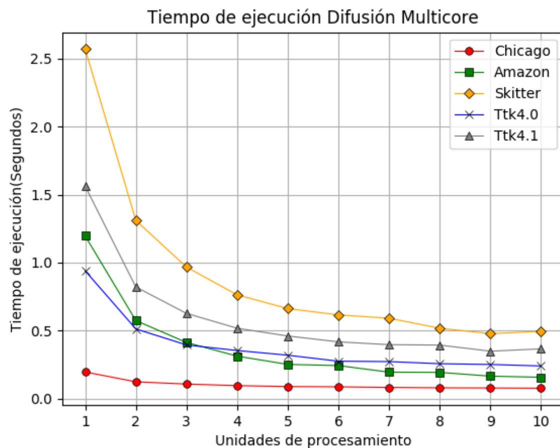

Figura 6.7: Tiempo de ejecución para el algoritmo de Difusión multicore

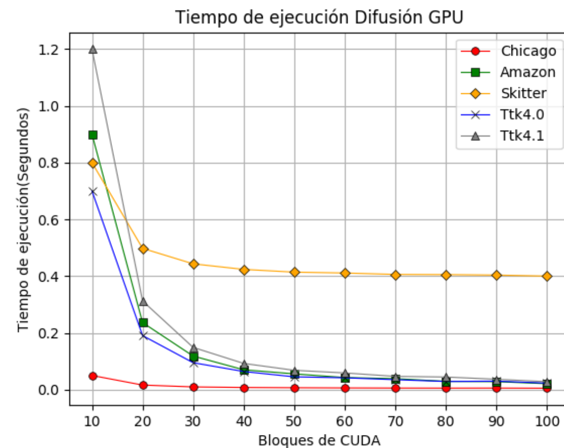

Figura 6.8: Tiempo de ejecución para el algoritmo de difusión GPU.

Observando el comportamiento del throughput en la Figura 6.9 se pueden percibir algunas características particulares. Una de ellas en relación al comportamiento del algoritmo con el conjunto de datos Skitter, específicamente en el último incremento en la cantidad de núcleos se observa que hay un decrecimiento del throughput. Esto debido a que el algoritmo de difusión presenta un componente de sincronización que se explota en mayor medida con el incremento de los vértices difusores presentes en las comunidades, por lo que supondría un costo mayor de procesamiento para las redes con comunidades de mayor tamaño. Por otra parte se puede observar que el comportamiento de la mayoría de los conjuntos es similar, por lo que se puede suponer que estos conjuntos presentan estructuras internas similares.

En la Figura 6.10 se presenta el throughput resultado del algoritmo de difusión GPU, donde se observa una alta variación en throughput del conjunto Skitter con respecto al resto de los conjuntos, lo que puede indicar que la distribución de trabajo para este conjunto se encuentra limitada por algunos vértices con grados altos y que corresponde con el tamaño de la red, es decir que como máximo en un vértice de la red Skitter pueden existir **1,696,415** vértices vecinos, lo que sugiere una

carga muy alta incluso cuando la fracción de vértices con grados similares sea baja.

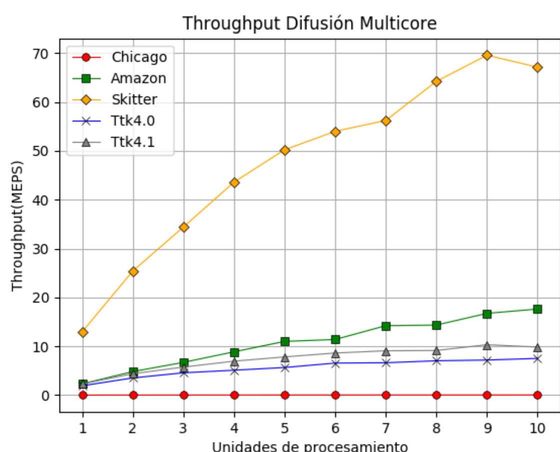

Figura 6.9: Throughput para el algoritmo de Difusión multicore.

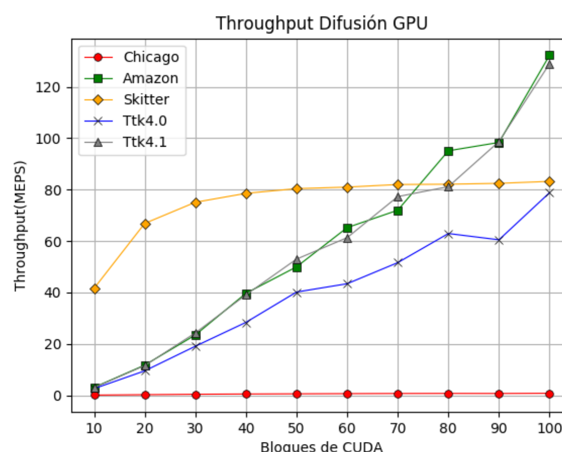

Figura 6.10: Throughput para el algoritmo de difusión GPU.

En la Figura 6.11 se presentan las aceleraciones para el algoritmo de difusión multicore con respecto al algoritmo secuencial, similar a lo que ocurre con el algoritmo LP, en el algoritmo de difusión multicore se observa una perdida con respecto al paralelismo disponible, aprovechando aproximadamente el 70 % éste. La aceleración que se observa para el algoritmo de difusión GPU, cuyos resultados se muestran en la Figura 6.12, es mayor que la obtenida por el algoritmo multicore. A pesar de que es un resultados esperado, debido a la diferencia en la capacidad de generar paralelismo para las dos arquitecturas, sin embargo, en este caso los algoritmos GPU son altamente sensibles a la distribución que presentan las redes. Algo similar ocurre con algunos conjuntos en los cuales se observa que la aceleración converge a un cierto valor sin importar el incremento del paralelismo. Lo que indicaría una carga de trabajo desbalanceada.

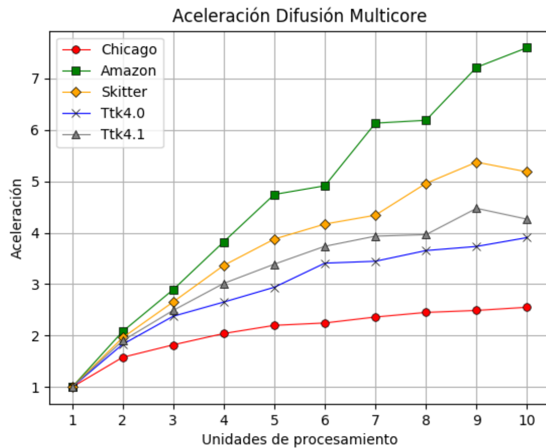

Figura 6.11: Aceleración algoritmo de difusión multicore.

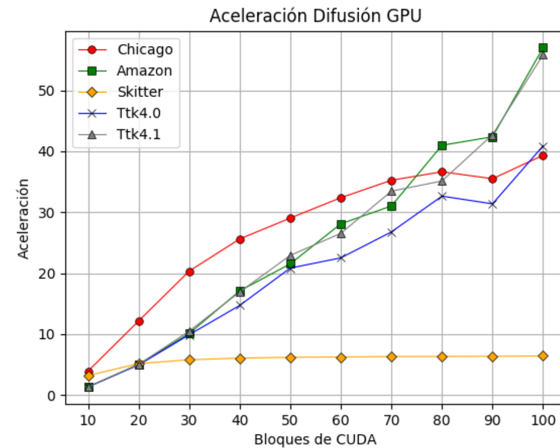

Figura 6.12: Aceleración algoritmo de difusión GPU.

6.5 Algoritmos paralelos para la propagación de etiquetas por influencia

A continuación se presentan los principales resultados de los algoritmos de influencia multicore y GPU descritos en secciones anteriores. En la Figura 6.13 se observa el tiempo de ejecución que se obtiene para el algoritmo de influencia multicore, mientras que en la Figura 6.14 el tiempo tomado por el algoritmo GPU. Para ambos casos el comportamiento es regular decreciente, teniendo casos de **18.2** segundos para el algoritmo multicore y **3** segundos para el GPU, utilizando la mínima capacidad de paralelismo.

Por otra parte en la Figura 6.15 se muestra el throughput para el algoritmo de influencia multicore. En éste se observa una la relación directa del throughput con el tamaño de la red, similar a casos anteriores. Sin embargo, en este caso las lineas trazadas sobre la gráfica tienen una menor separación entre sí, lo que indicaría una menor perdida del paralelismo total. En la Figura 6.16 se presenta el throughput del algoritmo de influencia GPU. A diferencia de lo que ocurre con el algoritmo multicore, se puede observar que el throughput registrado para los conjuntos de datos siguen un patrón regular

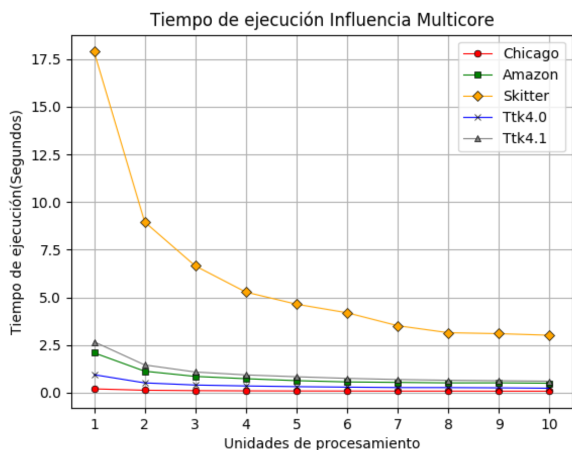

Figura 6.13: Tiempo de ejecución para el algoritmo de influencia multicore

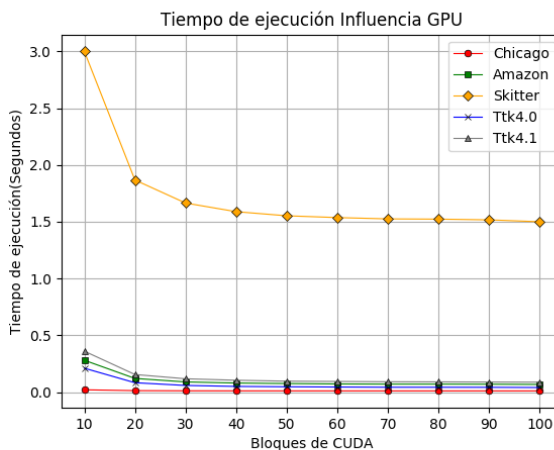

Figura 6.14: Tiempo de ejecución para el algoritmo de influencia GPU.

de comportamiento. Sin embargo, existe una diferencia considerable en la variación de los mínimos y máximos del throughput. Este comportamiento del algoritmo de influencia GPU indica que para ciertos conjuntos se aprovecha de mejor manera la capacidad de paralelismo disponible, lo que implica que los resultados sean altamente dependientes de la estructura interna de las redes.

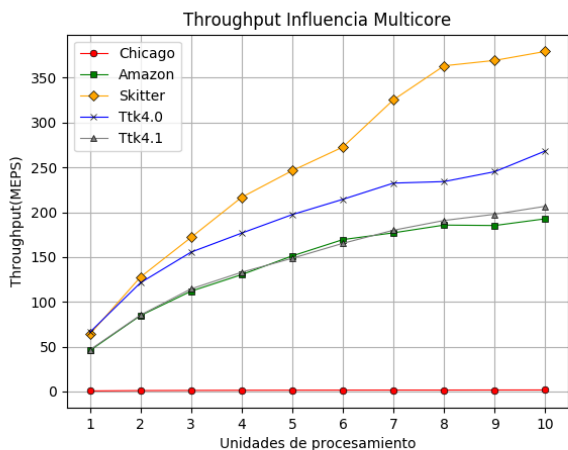

Figura 6.15: Throughput para el algoritmo de influencia multicore.

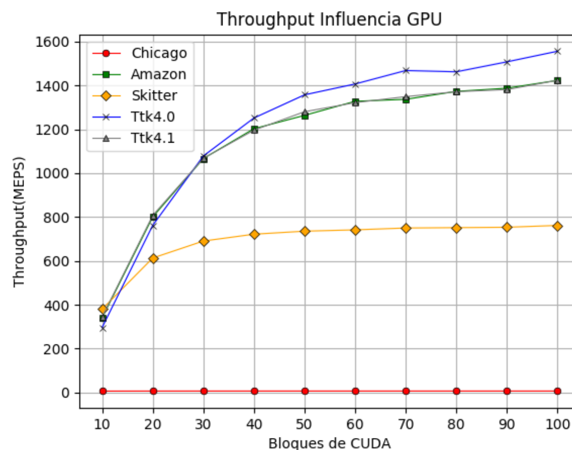

Figura 6.16: Throughput del algoritmo de influencia GPU.

Finalmente, en la Figura 6.17 se presentan las aceleraciones para el algoritmo de influencia multicore con respecto al secuencial. Observando los resultados del algoritmo se pude notar que

existe una variación significativa en las aceleraciones que se presentan para cada conjunto de datos. Del mismo modo ocurre para el algoritmo de influencia GPU que se presenta en la Figura 6.12, sin embargo, para este caso no se encuentra una relación directa entre la aceleración y el tamaño del conjunto de datos. Por lo que se puede asumir que la estructura interna de las redes puede afectar tanto de manera positiva como negativa la distribución de trabajo.

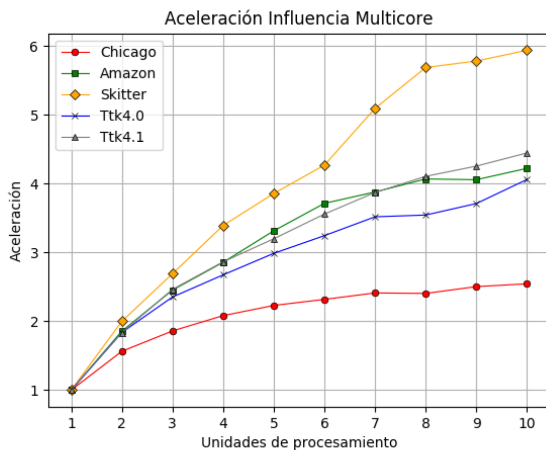

Figura 6.17: Aceleración algoritmo de influencia multicore.

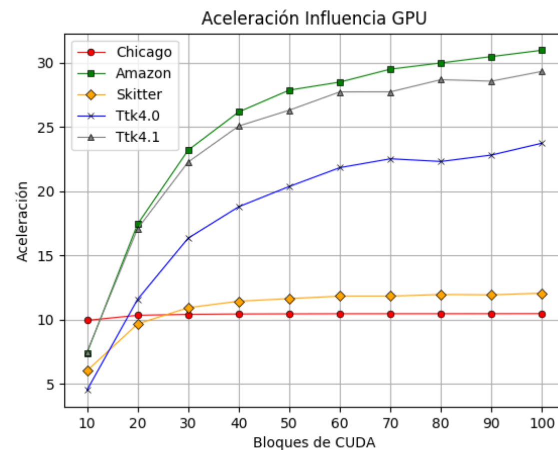

Figura 6.18: Aceleración algoritmo de influencia GPU.

6.6 Caso de estudio

La parte medular de este trabajo se centra en la propagación de etiquetas sobre redes complejas. Estableciendo algunos algoritmos que abordan algunos problemas de redes por medio del marcado de los vértices. En esta sección se presenta un caso de estudio en el que se involucran las soluciones paralelas que se presentan en el trabajo aplicadas a políticas de control de acceso. Es decir que las etiquetas propagadas por los algoritmos serán reemplazadas por políticas de control de acceso definidas sobre el esquema de cifrado basado en atributos [21].

En un sistema CP-ABE(CipherText-Policy Attribute-Based Encryption) las llaves de usuario y los elementos cifrados son marcados con un conjunto de atributos descriptivos y una llave capáz de

descifrar cualquier elemento solo si existe una relación entre los atributos del bloque cifrado y la llave del usuario [7, 20].

Una política de control de acceso, se define como una estructura de acceso tal que:

1. $\{P_1, P_2, ..., P_n\}$ es un conjunto de colecciones de atributos.

2. Una colección $A \subseteq 2^{\{P_1, P_2, ..., P_n\}}$ es monótona si $\forall\ B, C : if\ B \in A \wedge B \subseteq C \Rightarrow C \in A$.

3. Una estructura de acceso es una colección A de conjuntos no vacíos de $\{P_1, P_2, ..., P_n\}$ tal que $A \subseteq 2^{\{P1, P2, ..., Pn\}} \setminus \{\emptyset\}$.

4. Los conjuntos en A son llamados los conjuntos autorizados, mientras que los que no pertenecen a A se les conoce como los conjuntos no autorizados.

Una política de seguridad puede interpretarse como una función booleana que se evalua como **Verdadera** ó **Falsa** dado un conjunto de atributos W. Se dice que W satisface A si A es **Verdadera** en W [24].

A pesar de que el cifrado basado en atributos CP-ABE puede brindar el servicio de confidencialidad, en muchos casos es difícil la aplicación de este esquema debido al problema de definir las políticas de control de acceso, ya que deben crearse individualmente para cada elemento a cifrar. Esto representa una tarea difícil de realizar ya que deberá existir algún recurso humano encargado de ello. Tomando en cuenta lo anterior se plantea la posibilidad de la propagación de políticas de seguridad como etiquetas. Donde el medio de propagación será una red modelada a partir de las relaciones existentes entre los elementos a cifrar.

El objetivo de la propagación de políticas como etiquetas es la automatización del proceso de definición y asignación de políticas. Para ello se requiere de mecanismos que permitan la creación y manipulación de las políticas.

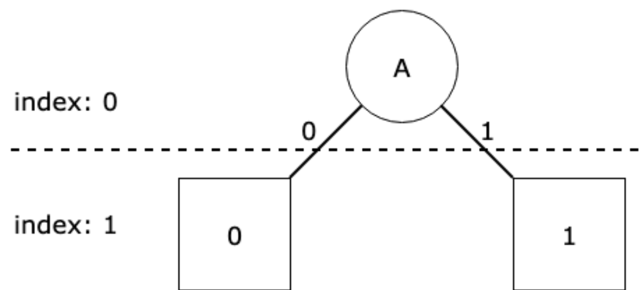

Figura 6.19: Ejemplo de variable de una función booleana.

6.6.1 Representación de políticas de control de acceso

Las políticas de control de acceso se pueden representar por medio de diferentes estructuras como: tablas de verdad, arboles, grafos, entre algunas otras. Sin embargo, para la mayoría de los casos no es posible la manipulación de éstas. Por ello, se propone utilizar algunas estrategias presentadas en [9] para la representación y fusión de funciones booleanas.

La representación inicial de la función será un grafo acíclico que representará los posibles estados de las variables de la función, es decir que comenzando por el vértice raíz y realizando un recorrido en profundidad se alcanzará al resultado de la función representada en alguna de sus hojas.

En la Figura 6.19 se presenta la representación de un parámetro de una función lógica con el modelo presentado. Donde los vértices hojas se denotan por la figura cuadrada mientras que el resto se denotan por la circular. A diferencia de los vértices hojas el resto inicialmente no tienen un valor asignado por defecto ya que estos vértices representan a las variables de la función booleana, por lo que pueden tomar ambos valores. Para ello se asume que al moverse desde el vértice raíz hacia el vértice inferior izquierdo el valor de la variable que representa el vértice raíz es falso. Por otra parte si se mueve desde la raíz hacia el vértice inferior derecho se asume que el valor de la variable raíz es verdadero. En el caso de las hojas, se encargan de determinar el resultado de la expresión booleana siguiendo un camino determinado desde la raíz.

6.6.2 Reducción de funciones booleanas

Debido a que se espera que durante los procesos de difusión se deban generar nuevas políticas, es posible que las funciones de las políticas crezcan en función de la cantidad de políticas definidas para la red. Debido a este problema, es necesario el uso de un mecanismo de reducción que mitigue el crecimiento de las funciones.

En el Algoritmo 20 se presenta un algoritmo para la reducción de una función booleana. El algoritmo utiliza algunas inferencias simples para reducir el tamaño del árbol de posibles valores para cada variable. Sabiendo que se tiene una expresión booleana de cinco variables, el árbol que se genera en el peor de los casos es un árbol binario de profundidad seis con una cantidad de 6^2 vértices. El objetivo es obtener una representación equivalente a la expresión original, reduciendo el espacio necesario para representar la expresión. Permitiendo la unión de expresiones junto a una reducción, se pretende escalar en la cantidad de políticas fusionadas sin comprometer los mecanismos de seguridad expresados por el cifrado CP-ABE [10].

El algoritmo de reducción de funciones opera de la siguiente manera:

En el primer paso se realiza una clasificación de los vértices del árbol que conforma la expresión a partir del valor de sus índices, tomando como índice de cada vértice la profundidad que se encuentra con respecto al vértice raíz. Por ejemplo el vértice raíz en cualquier caso tendría el índice 0 y los vértices subyacentes el índice 1, y del mismo modo para el resto de vértices.

Una vez clasificados los vértices por índice, se recorren los vértices clasificados desde las hojas hacia la raíz, asignando un identificador a los vértices que formarán la nueva representación de la función.

En la Figura 6.20 se muestra el primer paso que se realiza sobre una función booleana de entrada, denotada en la parte izquierda de la figura la función de entrada original y en la derecha el resultado del proceso aplicado. En este caso primer lugar se busca trabajar los vértices terminales, los cuales contienen como valor asignado el resultado de la función siguiendo un cierto camino denotado por

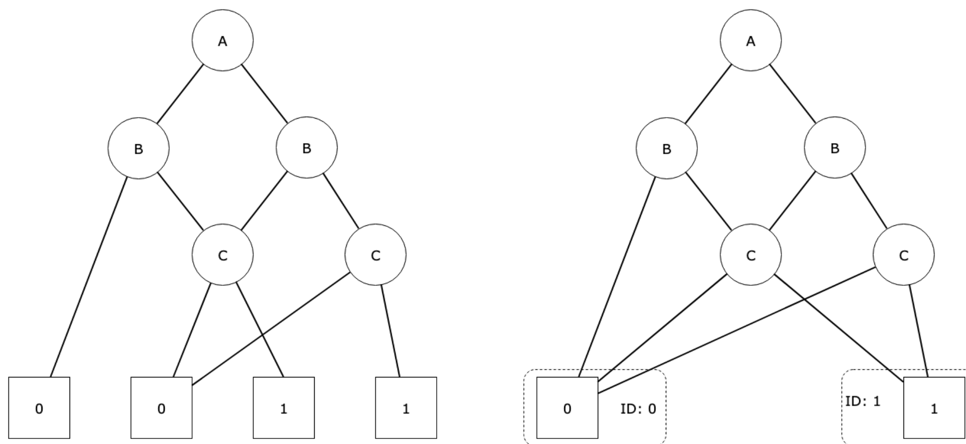

Figura 6.20: Reducción de la capa tres perteneciente a los vértices terminales.

los vértices no terminales. Al ser una representación para funciones booleanas se conoce que el resultado solo puede tomar dos resultados, por lo que solo se requieren de dos vértices terminales como máximo. Por ello en primer lugar se agrupan los vértices de la capa tres, asumiendo que se han agrupado en capas iniciando con la capa cero siendo el vértice raíz y la capa tres los vértices terminales.

Por lo tanto y solo en el caso de la última capa se agruparán los vértices uniendo aquellos con el mismo valor asignado. La agrupación de los vértices se realiza por medio de una cola de prioridad, en donde se pondrán todos los vértices de la capa en procesamiento y se extraerán siguiendo el orden establecido por la cola. Debido a que se conoce que los únicos vértices que tienen un valor asignado son los terminales y que solo existen dos valores posibles para ellos(cero y uno) al realizar un agrupamiento por valor la cantidad de vértices siempre será reducida a solo dos vértices a diferencia de el resto de los vértices cuyo valor relacionado a la cola de prioridad serán los identificadores de los vértices.

En el siguiente paso se procesa la capa de vértices inmediata superior, siendo ésta la capa número dos, donde los vértices que se encuentran son solo no terminales. Para este caso se aplica nuevamente la agrupación de vértices por medio de la cola de prioridad, pero en esta caso la agrupación se realiza con base en los identificadores de los vértices hijos denotados por la propiedad id de un vértice

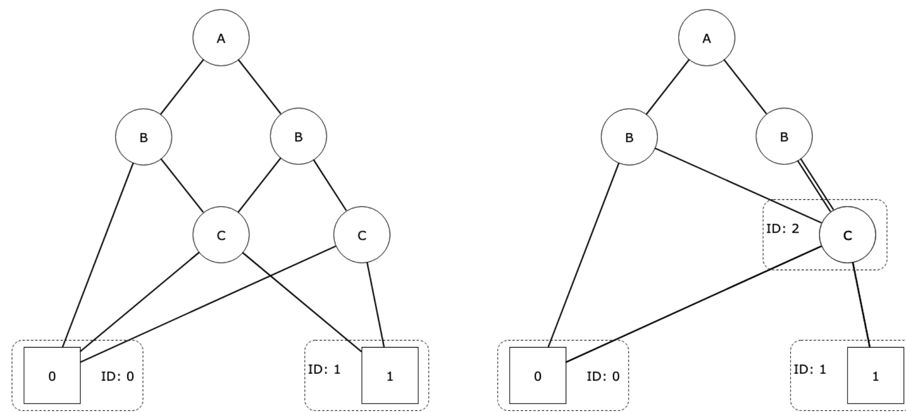

Figura 6.21: Reducción y simplificación de los vértices de la capa dos.

definida en la estructura de datos utilizada para modelar los vértices. De manera que los vértices que compartan los mismos identificadores en sus hijos serán agrupados bajo el mismo identificador. Estos se puede observar en la Figura 6.21 donde los vértices de la capa dos correspondientes a la variable C, tienen ambos hijos a la izquierda y derecha vértices con el mismo identificador, cero y uno respectivamente por lo que uno de ellos es innecesario.

Pasando al procesamiento de la capa número uno se puede observar en la Figura 6.22 que los vértices en ésta capa pertenecientes a la variable B, existe al menos uno que tiene enlaces hacia vértices inferiores con el mismo identificador, lo que indica que es un vértice redundante que no aporta al resultado de la función. Por esta razón el vértice redundante toma el valor identificador de sus hijos con los que es eliminado de la función. Esto deja solamente a la capa número uno con solo un vértice B.

Por último se observa en la Figura 6.23 el agrupamiento de los vértices en la capa cero, siendo el único vértice existente el vértice raíz, desde el cual parten los posibles valores para la variable A. En este caso simplemente se asigna un identificador para dicho vértice y se termina con el procesamiento. Al final de realizar el procesamiento para cada capa de vértices de la función quedarán los vértices originales de la función asignados a un identificador creciente que inicia en los vértices terminales y que servirá para generar el nuevo árbol que representará a la misma función inicial de manera reducida.

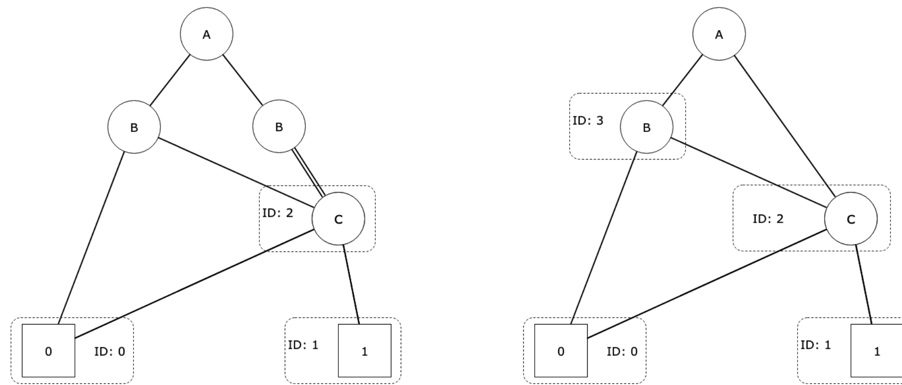

Figura 6.22: Simplificación de los vértices de la capa uno.

Esto se consigue ya sea por medio de una estructura de vértices nueva o reescribiendo los valores de la estructura de datos original. Sea cualquiera el caso a partir de la asignación de los primeros dos identificadores, es decir después de la capa de los vértices terminales, se procede a seleccionar a un vértice representativo del identificador asignado, es decir si existen dos o más vértices con el mismo identificador, se deberá seleccionar alguno y descartar el resto.

A partir de las capas de vértices no terminales y después de realizar la asignación de identificadores, los vértices seleccionados como representantes del identificador se conectarán con los vértices indexados por el identificador establecido. Por ejemplo, asumiendo que existe una nueva estructura con la misma cantidad de vértices que la original y sin enlaces entre vértices, generar la estructura reducida se logra uniendo los vértices no terminales hacia los vértices hijos por medio del valor del identificador, es decir que el vértice asignado al índice 2 tendrá como hijos a los vértices 0 y 1, mientras el vértice con índice 3 tendrá como hijos a los vértices 0 y 2 mientras que el vértice 4 a los vértices 3 y 2.

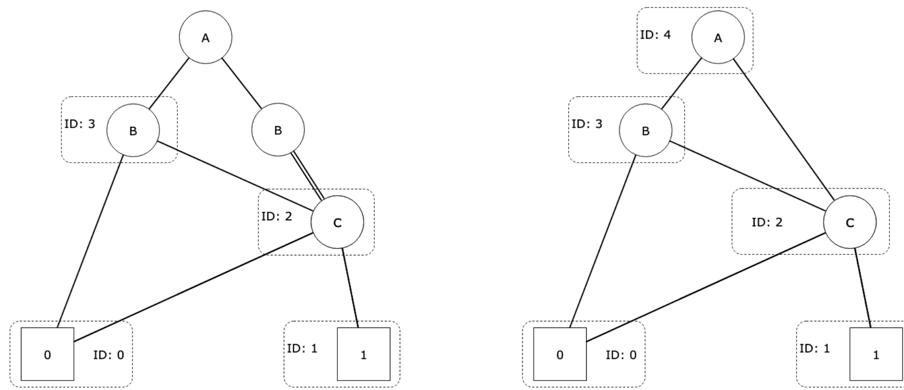

Figura 6.23: Asignación de identificador al vértice raíz y finalización del recorrido.

6.6.3 Conversión de representación

Una política de control de acceso válida para CP-ABE utiliza una representación de compuertas umbral denotando la cantidad de atributos que se requieren para satisfacer una operación [37]. Por ejemplo una conjuntiva lógica entre dos variables A y B se expresa en compuertas umbral como $A, B(2of2)$, es decir que se requieren los dos atributos A y B para satisfacer la conjunción.

Debido a lo anterior se requiere de un proceso de transformación, que llevaría a una función booleana representada como grafo acíclico a la representación en compuertas lógicas o de umbral. Para ello son necesarios los valores de los índices establecidos por el algoritmo de reducción, por lo que es importante almacenarlos hasta que se termine completamente cualquier proceso de fusión de políticas.

Sabiendo que para cualquier función booleana reducida solo existe un vértice para cada nivel del grafo y después del proceso de reducción cada vértice tiene asignado un índice que indica el nivel al que pertenece en el grafo. A partir de los índices de los vértices es posible reconstruir la función booleana original, asumiendo que la existencia de una relación hacia el hijo derecho de un vértice representa una restricción de dependencia o el equivalente a una compuerta conjuntiva, mientras que la relación del hijo izquierdo es equivalente a una compuerta disyuntiva.

Con base en este razonamiento la reconstrucción de una función booleana cualquiera, que haya

sido mapeada a la estructura utilizada en este trabajo sigue la siguiente estrategia:

Asumiendo que los vértices del grafo han sido marcados por el proceso de reducción, se conoce el identificador de cada vértice que indica la profundidad invertida de éste y sin tomar en cuenta los vértices hojas. Se procede desde el vértice raíz hacia las hojas siguiendo una estrategia *top down* en la cual se comparan en primera instancia los hijos del vértice raíz por medio del identificador. A partir del vértice hijo con el identificador de mayor valor, es decir aquel que se encuentra más cercano a la raíz se establecerá la primera operación entre vértices que dependerá de si se trata del hijo izquierdo o el derecho. En caso de tratarse del hijo izquierdo, la operación entre el vértice raíz y el hijo izquierdo será un **OR** lógico o una compuerta umbral uno de dos(1of2), por otra parte si se trata del hijo derecho la operación cambiará a un **AND** lógico o una compuerta umbral dos de dos(2of2).

En la Figura 6.24 se presenta el proceso de transformación de la representación como grafo acíclico a una representación de compuerta umbral. Donde se encuentra inicialmente la representación de una función booleana $A \vee B \wedge C$ con tres variables ya reducida por el proceso de reducción, en la cual tomando como punto de partida el vértice raíz A se comparan los identificadores de ambos hijos y se avanza hacia aquel con mayor valor o más cercano con base en los valores del identificador. En este caso se avanza desde el vértice raíz A hacia el vértice B, con los que se construye la primer operación. Al ser B el hijo izquierdo de A la operación entre A y B será una compuerta umbral **1of2**. Una vez se ha movido hacia el vértice B se procede a realizar la comparación nuevamente y excluyendo a los vértices hojas se avanza hacia el vértice C el cual se encuentra hacia la derecha del vértice B lo que índica una compuerta umbral **2of2** que evaluará el resultado de la compuerta anterior con C reproduciendo la función booleana original.

En el Pseudocódigo 6.24 se presenta el funcionamiento de la estrategia para regresar a la representación inicial de una función booleana particular. El proceso de reversión de la función inicia a partir del vértice raíz, del cual se revisará que pueda ser procesado, es decir que tenga al menos un vértice que no es una hoja. Esto se consigue sumando los identificadores de sus dos hijos, izquierdo

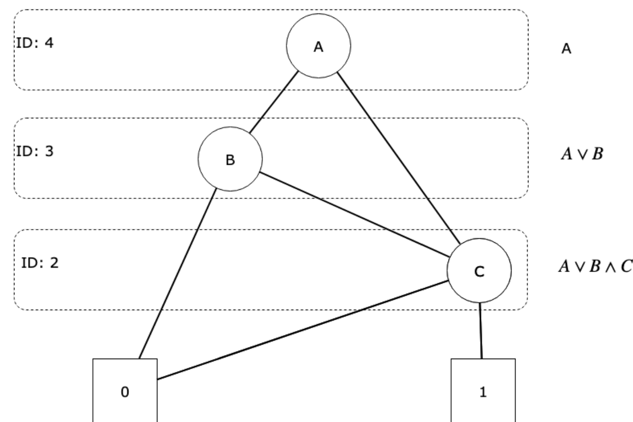

Figura 6.24: Extracción de función textual desde la representación de grafo acíclico.

y derecho, ya que solo los vértices hoja tienen los identificadores uno y dos, por lo que cualquier vértice que no pertenezca a éstos tendrá un identificador mayor a uno.

Algoritmo 19 Algoritmo de reversión de funciones booleanas

1: **Function** $restore(Vertexroot, String\ foo \leftarrow null)$
2: **if** $root.right.id + root.left.id > 1$ **then** ▷ Excluye a los vértices hojas
3: **if** $foo = null$ **then**
4: $foo \leftarrow root.id$
5: **if** $root.right.id > root.left.id$ **then**
6: **return** $restore(root.right, "foo \wedge root.right.id)")$
7: **else**
8: **return** $restore(root.left, "foo \vee root.left.id)")$
9: **else**
10: **return** foo

Con esta primera condición se excluyen los vértices hojas y se procede a construir la función por medio de compuertas umbral. A continuación se revisa la variable auxiliar foo que se utiliza como parámetro por omisión en el caso inicial, es decir que dicha condición solo se cumplirá en la primera llamada al método, lo que asignará el valor del identificador del vértice raíz.

En la tercera condición dentro del algoritmo se revisan a los vértices hijos y se selecciona aquel con mayor valor de su identificador, esto representa al vértice hijo más cercano en relación con los niveles del grafo, usualmente el vértice más cercano se encuentra justo en el nivel inferior del vértice

actual.

Dependiendo de la selección la función se construirá de una u otra forma. En caso de seleccionarse el hijo izquierdo la operación que se encontrará dentro del vértice inicial y su hijo será una compuerta umbral **1of2** equivalente a un $\vee$, la cual queda definida en una cadena que se pasará hacia la siguiente llamada a la función de manera recursiva junto al vértice hijo en cuestión. Del mismo modo sucederá para el caso en que se seleccione el hijo derecho, sin embargo en este caso la compuerta utilizada será una **2of2** equivalente a un $\wedge$.

Con las siguientes llamadas a la función se construirá la totalidad de la función hasta que se alcance el último vértice conectado a las hojas, con el cual se terminan las llamadas recursivas y se retorna sin más la cadena representativa de la función original.

Algoritmo 20 Algoritmo de reducción

```
1:  struct Vertex
2:      left  ← Vertex()
3:      right ← Vertex()
4:      index ← null
5:      value ← null
6:      id    ← null
7:
8:  Function reduce(Vertex[ ]list)
9:      subgraph ← Vertex[size(list)]
10:     vertexmx ← [ ][ ]
11:     for i ← 0 to size(list) do
12:         vertexmx[list[i].index] ← list[i]
13:     for i ← size(vertexmx) − 1 to 0 do
14:         queue ← [ ]                    ▷ Cola de prioridad ordenada por primer elemento
15:         for v ∈ vertexmx[i] do
16:             if v.index = size(vertexmx) then
17:                 queue.push([v.value, null, v])
18:             else if v.left.id = v.right.id then
19:                 v.id ← v.left.id
20:             else
21:                 queue.push(v.left.id, v.right.id, v)
22:         oldkey ← [null, null]
23:         while not queue.empty() do     ▷ Mientras la cola no sea vacia
24:             k1, k2, v ← queue.pop()
25:             if [k1, k2] = oldkey then
26:                 v.id ← nextid
27:             else
28:                 v.id ← nextid
29:                 subgraph[nextid] ← v
30:                 nextid ← nextid + 1
31:                 oldkey ← [k1, k2]
32:                 if v.index! = size(vertexmx) − 1 then
33:                     v.left ← subgraph[v.left.id]
34:                     v.right ← subgraph[v.right.id]
35:     return subgraph[nextid]
```

6.6.4 Resultados

Las uniones entre funciones pueden ser permisivas, es decir que al unir dos funciones diferentes por medio de una compuerta $\vee$ se espera que la función resultante pueda ser satisfecha con una cantidad de atributos menor, mientras que una unión restrictiva creada por una compuerta $\wedge$ requerirá una cantidad de atributos mayor para ser satisfecha.

En la Figura 6.25 se presentan algunos resultados del algoritmo de reducción de funciones, en la cual se muestra el grafo inicial desde la izquierda, para posteriormente pasar por el método de reducción y finalmente volver a la representación de la función en texto plano. En el primer caso se observa la función $A \vee B \wedge C$, que inicialmente cuenta con 5 vértices no terminales que tras pasar por el proceso de reducción se crea una nueva representación con solo tres vértices y que es equivalente a la versión inicial. A partir del grafo reducido se extrae la función de manera textual a partir de los identificadores marcados por el proceso de reducción.

El proceso de reducción tiene como base de funcionamiento algunos casos especiales que permiten inferir cuando existen vértices redundantes. El primer caso se da en los vértices terminales o las hojas, donde se conoce que en una representación mínima solo deben existir a lo más dos vértices diferentes, uno que represente el estado falso o negativo y el otro que represente el estado verdadero o positivo. Con base en los identificadores que se asignan en el recorrido $bottom - up$ del proceso de reducción y el agrupamiento de vértices de la misma profundidad se encuentran los vértices redundantes que serán descartados, donde los identificadores también indicarán la estructura del grafo reducido.

En el peor de los casos el grafo inicial será del tipo de un árbol binario completo donde la profundidad de éste estará determinada por la cantidad de variables en la función más uno. El tercer caso que se muestra en la Figura 6.25 índica un caso similar, en el que inicialmente la representación del grafo tiene una cantidad de vértices muy superior a la representación mínima. En última instancia la tarea final es pasar el grafo representante de la función reducida hacia una representación textual de la función, de tal manera que el resultado pueda ser utilizado sobre el esquema CP-ABE. A pesar

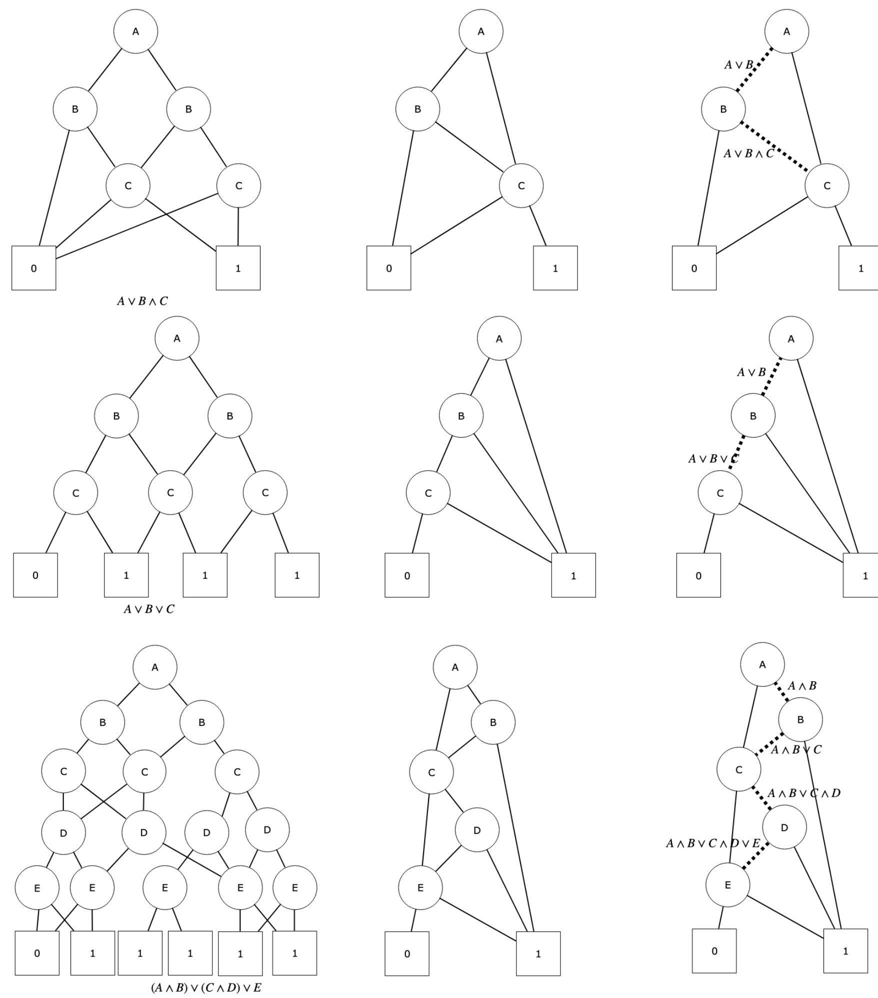

Figura 6.25: Resultados para el proceso de reducción de funciones en tres etpas: Representación inicial, Reducción, Reversión.

que en este enfoque el mapeo entre representaciones se realiza por medio de compuertas lógicas en ciertos casos se requerirá un proceso extra para transformar las compuertas lógicas a compuertas umbral para que la compatibilidad entre tareas sea adecuada.

Resumen

En este Capítulo se presentaron los principales resultados para los algoritmos de detección de comunidades, difusión e influencia. Para ambos casos se evaluaron las soluciones multicore y GPU. Posteriormente se presentó un caso de estudio en el cual las etiquetas son reemplazadas por políticas de control de acceso.

7

Conclusiones y trabajo futuro

7.1 Resumen

En este trabajo se han desarrollado e implementado diferentes algoritmos paralelos utilizando las dos principales arquitecturas paralelas, multicore y GPU.

En el esquema de solución propuesto se describen al menos tres tipos de algoritmos: Algoritmos para detección de comunidades, algoritmos para la difusión de etiquetas que incluye a los algoritmos de difusión e influencia. En ambos casos, se presentaron soluciones secuenciales, multicore y GPU, obteniendo resultados variados entre las soluciones. Para el caso de las soluciones multicore, donde la calidad del particionamiento del trabajo determina la cantidad de paralelismo efectivo que puede obtenerse, se optó por utilizar una estrategia de particionamiento cíclico. Debido a la distribución de grados que se observa en una red compleja, con el particionamiento cíclico se observa un mejor balance de carga entre los threads que con un particionamiento estático.

Aunque un particionamiento dinámico podría considerarse adecuado debido a la diversidad de los

grados de los vértices, por las propias características de las redes complejas una gran cantidad de los vértices tienen un grado muy pequeño. El particionamiento dinámico agregaría un trabajo adicional al manejo de muchos vértices con grado pequeño, los cuales representan la mayoría en las redes complejas.

Dado el caso anterior el problema que en principio se suponía como altamente irregular puede convertirse en uno semi regular afectando el rendimiento de particionamientos basados en mecanismos de acceso sincronizado como el particionamiento dinámico. Por otra parte se espera que las soluciones planteadas en este trabajo sean altamente escalables, por lo que agregar componentes síncronos a las soluciones limitaría el alcance de éstas estableciendo un limite para la capacidad de paralelismo efectivo que se obtendría de las soluciones.

Por otra parte en el caso de las soluciones GPU donde el problema reside en las estructuras de datos altamente irregulares, la cantidad de paralelismo que es posible alcanzar queda determinado por la propia estructura, mientras mayor sea su regularidad se espera que una cantidad de threads por bloque se mantenga activa durante el tiempo de vida del mismo. La irregularidad en las listas de adyacencia de los threads agrupados en un solo bloque provoca que el porcentaje de ocupación de los mismos sea baja. Por ello se sigue una estrategia basada en agrupamiento de los vértices con base en el grado, de manera que las diferentes agrupaciones generadas fueran lo más regulares posibles, para de ese modo aprovechar de mejor manera el modelo SIMD de la arquitectura GPU.

La estrategia asíncrona no tiene un efecto mayor en los resultados finales, sin embargo, su paralelización genera condiciones de carrera lo que resulta en un algoritmo no-determinista. No obstante lo anterior, Las diferencias en el etiquetado de comunidades fueron menores sin un impacto significativo.

La estrategia síncrona permite tener un algoritmo determinista pero a cambio es necesario pagar el costo de la sincronización global al final de cada iteración y el uso de una copia de la estructura de datos para evitar las condiciones de carrera.

7.2 Algoritmos para la detección de comunidades

En la detección de comunidades con algoritmos multicore se observó un incremento en la aceleración con redes densas, sin embargo, esto es contrario al objetivo de evaluar redes complejas de baja densidad, que por sus propiedades dificultan la operación de soluciones multicore. A pesar de esto, se conoce que este tipo de redes contienen al menos una componente gigante, que se puede observar como una subred densa dentro de la red compleja, la cual es la mayor limitante para la paralelización eficiente en arquitecturas multicore.

Para obtener la estructura de comunidades de una red se siguió una estrategia basada en el algoritmo LP, que funciona por medio del conteo de frecuencia de etiquetas entre los vértices. Debido a su funcionamiento es posible expresar el algoritmo en términos de operaciones aritméticas simples y comparativas, lo que se convierte en una ventaja cuando se desarrollan soluciones GPU.

En estos casos es de importancia realizar un análisis del comportamiento, ya que es posible asumir un problema como inherentemente secuencial aún cuando las condiciones de carrera no supongan un problema para la obtención de resultados en el marco completo de funcionalidad del problema.

La diferencia en throughput alcanzado entre las versiones multicore y GPU para el algoritmo de detección de comunidades fue de 405 MEPS. Las aceleraciones alcanzadas en la versión GPU fueron de 27x con respecto a la versión secuencial y de 6x con respecto a la versión multicore.

7.3 Algoritmos de difusión de etiquetas

La difusión de etiquetas se lleva acabo por un algoritmo de difusión de información con base en la distancia, el cual utiliza algunos vértices especiales definidos sobre la estructura de comunidades de la red conocidos como vértices difusores.El proceso de difusión requiere los caminos más cortos de los difusores a todos los demás vértices, Por lo anterior, se utilizó el algoritmo de Dijkstra para el cálculo del camino más corto adaptándolo a la propagación de las etiquetas. La estrategia multicore,

realiza recorridos a partir de cada difusor aplicando una estrategia de grano grueso.

La componente gigante de la red puede representar otro tipo de caso especial, en el cual se aglomera una gran cantidad de vértices que conforman una comunidad.

En redes complejas, aparecen comunidades pequeñas a los cuales típicamente se les asigna un solo difusor por lo que el problema de difusión de etiquetas se simplifica. No obstante que la componente gigante de la red requiere un tratamiento especial, ya que es una componente fuerte, se observó que un solo recorrido de la misma es suficiente para etiquetar todos los vértices en el proceso de difusión. Sin embargo, también justificó el desarrollo del algoritmo de influencia pues el etiquetado varía según el inicio del recorrido.

El algoritmo de influencia permite hacer ajustes al etiquetado si éste fuera desarrollado a partir de otros puntos de origen. Se observó que los cambios de etiquetado en el algoritmo de influencia fueron en solo un 16 % de los vértices etiquetados durante el proceso de difusión.

A pesar de representar una exigencia computacional menor que los algoritmos de detección de comunidades e influencia. El algoritmo de difusión obtuvo los resultados más altos de aceleración. Con una diferencia en el throughput multicore y GPU de 60 MEPS. Sin embargo, las aceleraciones alcanzadas en la versión GPU fueron de 57x con respecto a la versión secuencial y de 8.5x con respecto a la versión multicore.

El algoritmo de influencia es altamente paralelizable ya que cada vértice realiza el análisis de su vecindad. La única consideración que se debe hacer son las condiciones de carrera que se presentan cuando vértices vecinos cambian de etiqueta durante la misma iteración. Un proceso de actualización síncrono, limitó los efectos de los cambios de etiqueta y solo una número pequeño de iteraciones (entre 43 y 112) fueron necesarios para establecer una condición de paro.

La versión multicore del algoritmo de influencia también se benefició de un particionamiento cíclico. En el algoritmo para GPUs, también fue útil la clasificación de los vértices en tres grupos de acuerdo con su grado. La diferencia en throughput alcanzado entre las versiones multicore y GPU fue de 1,170 MEPS lo cual es reflejo del grado de paralelización que se observa en este algoritmo.

Las aceleraciones alcanzadas en la versión GPU fue de 31x con respecto a la versión secuencial y de 6x con respecto a la versión multicore.

De tal modo que la refinación del primer propagado de etiquetas se realiza por medio del segundo algoritmo de propagación, el cual se ha denominado como algoritmo de influencia, ya que determina el grado de influencia que tienen los grupos de vértices con respecto a su etiqueta asignada e inicia un nuevo proceso de reasignación de etiquetas, el cual se extiende hasta alcanzarse un estado de estabilidad en las actualizaciones.

Se requiere de un entendimiento profundo de la arquitectura de los GPUs para conseguir soluciones que presenten buen desempeño con respecto al tiempo y espacio. Soluciones con un rendimiento inferior a las multicore e incluso a las secuenciales es altamente posible.

Por último, las primitivas paralelas utilizadas en los algoritmos GPU, usualmente se encuentran en ordenes logarítmicos con respecto a la totalidad de los elementos con los que se trabaje, sin embargo, cuando el procesamiento se realiza sobre bloques de datos como ocurren en éste caso, el encontrarse con threads sin trabajo asignado es algo recurrente, incluso para soluciones eficientes en trabajo ya que por las mismas propiedades de la arquitectura no es posible una libre selección y configuración del despliegue de los threads. Con base en la anterior se puede afirmar que en aplicaciones que requieran de paralelismo masivo que supere la capacidad total de la GPU y se encuentre en condiciones similares a las anteriormente mencionadas, será más eficiente una solución lineal en la que cada thread se asigne a un elemento a procesar, evitando por completo a los threads sin trabajo asignado.

7.4 Algoritmos de fusión de políticas

Para llevar a cabo la fusión de políticas se requiere de la representación de dos de estas, las cuales pasarán por un proceso de unión y uno de reducción posteriormente. En las siguientes secciones se abordan los procedimientos utilizados para realizar estas tareas, sin embargo, los resultados de fusión

pueden ser dependientes de ciertas restricciones como el orden de los parámetros en las políticas así como la cantidad de políticas que serán unidas.

En la evaluación de la fusión de políticas de control de acceso se estableció una limite en la cantidad de políticas, utilizado una lista de 1,500 políticas de seguridad generadas de manera automática. Para lo cual se obtuvo un orden de crecimiento en el peor caso de $O(v^2)$ operaciones y en el mejor de $O(e + v)$.

La estructura para la representación de las políticas de seguridad presenta una desventaja importante. De no elegir un orden inicial para los parámetros de las funciones, la representación puede crecer hasta 2^v, presentando un problema para el manejo de memoria.

Se ha observado que la unión de políticas por medio de la conjunción lógica se reducen a su mínima expresión de manera más eficiente que las unidas por la disyunción, sin embargo, los casos en que se requiere de una estrategia restrictiva son escasos o de poca utilidad.

7.5 Contribuciones

Las principales contribuciones de este trabajo son las siguientes:

- Un algoritmo de detección de comunidades para arquitecturas multicore, basado en propagación de etiquetas.
- Un algoritmo de detección de comunidades para GPUs basado en propagación de etiquetas.
- Un algoritmo de difusión de etiquetas para arquitecturas multicore
- Un algoritmo de difusión de etiquetas para GPUs.
- Un algoritmo para la determinación de influencia de etiquetas para arquitecturas multicore.
- Un algoritmo de influencia de etiquetas para GPUs.

- Un estudio para evaluar la factibilidad de utilizar algoritmos de propagación de etiquetas que sirvan para propagar políticas de control de acceso sobre colecciones de documentos.

7.6 Limitaciones

Los algoritmos presentados en este trabajo se diseñaron para trabajar sobre las dos principales arquitecturas paralelas, las multicore y GPU. Por lo que su implementación requiere de equipo con tales características. Para ambos casos los algoritmos presentados se encuentran limitados por la memoria ya que no se podrá procesar redes que superen la capacidad de ésta. Sin embargo, esta limitación es particularmente especial en el caso de los algoritmos GPU, debido que presentan una mayor complejidad en la jerarquía de memorias. Lo que provoca que la cantidad de vértices procesados por bloques también se vea limitada por la cantidad de memoria compartida disponible en la arquitectura. La cantidad de threads disponibles para el procesamiento de un bloque de CUDA, se encuentra en función de la versión y tipo de la GPU, por lo que la variación entre plataformas GPU puede presentar cambios sustanciales en el desempeño de los algoritmos propuestos.

7.7 Trabajo futuro

Se debe tomar en cuenta que en los algoritmos paralelos desarrollados en este trabajo, existen diferentes conjuntos de parámetros que pueden modificar el comportamiento. Con base en los resultados obtenidos se puede asumir que el desempeño de los algoritmos, tiempo de ejecución, aceleración y throughput son fuertemente dependientes del orden inicial de los vértices de las redes que conforman el conjunto de datos, por lo que buscar una configuración ideal que se ajuste a cualquier tipo de red de entrada o incluso delimitándolo solo a redes complejas se vuelve una tarea compleja.

Debido a lo anteriormente mencionado una de las tareas a desempeñar como continuación de

este trabajo será la determinación no solo de los parámetros individuales de cada algoritmo planteado en sus diferentes etapas, sino de la mejor configuración de algoritmos(secuencial, multicore y GPU) para cada una de las etapas que resuelva de manera efectiva y eficiente el problema de propagación de etiquetas, prestando especial atención en la eficiencia en tiempo, que es lo que usualmente se busca cuando se utilizan soluciones de cómputo de alto rendimiento.

Una primera aproximación para determinar las configuraciones ideales de los algoritmos utilizados será la categorización de las redes con base en el grado de sus vértices. Si bien aquí se propuso una categorización en tres clases de vértices para tener un buen desempeño en las versiones GPU, existe la posibilidad de proponer una categorización diferente que tenga mejores resultados.

Asumiendo que son redes complejas y que junto a un fuerte conocimiento de las arquitecturas paralelas, sus virtudes y deficiencias es posible enfocar diferentes secciones de la red de entrada hacia los algoritmos que mejor se desempeñen sobre un conjunto determinado. Una posibilidad es construir algoritmos híbridos que utilicen tanto los GPUs como los núcleos disponibles. Los GPUs podrían dedicarse a procesar la gran cantidad de vértices de una red compleja que tienen un grado relativamente pequeño y que representan un alto porcentaje de vértices. Mientras, los CPUs podrían dedicarse a procesar los vértices con grado significativamente alto que existen en las redes complejas.

En el proceso de fusión de políticas se pueden encontrar algunos puntos a mejorar. Principalmente relacionados a la estructura de representación que puede volverse lo suficientemente grande para afectar el rendimiento de el algoritmo de reducción.

El proceso de fusión de políticas de seguridad consta de dos elementos principales. La representación de las funciones booleanas y el algoritmo que reduce las funciones a su mínima expresión. Considerando los resultado de la experimentación, se ha encontrado que existe una limitante en la estructura de representación de funciones. La cual requiere de un mapeo para ir desde el espacio de políticas CP-ABE al de la función booleana. Por ello, se plantea la posibilidad de desarrollar una versión mejorada de el algoritmo de reducción el cual no requiera la transformación entre representaciones.

Del mismo modo se plantea como parte del trabajo futuro la especialización del algoritmo de reducción para el uso de funciones no monótonas. Ya que una parte importante de un sistema CP-ABE es la versatilidad a la hora de diseñar las políticas de control de acceso.

Bibliografía

[1] Albert, R. and Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97.

[2] Amiri, B., Hossain, L., Crawford, J. W., and Wigand, R. T. (2013). Community detection in complex networks: Multi–objective enhanced firefly algorithm. *Knowledge-Based Systems*, 46:1 – 11.

[3] Asaduzzaman, A., Gummadi, D., and Yip, C. (2014). A talented cpu-to-gpu memory mapping technique. pages 1–6.

[4] Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512.

[5] Barbehenn, M. (1998). A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *Computers, IEEE Transactions on*, 47:263.

[6] Belova, M. and Ouyang, M. (2017). Breadth-first search with a multi-core computer. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 579–587.

[7] Bethencourt, J., Sahai, A., and Waters, B. (2007). Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 321–334.

[8] Boccaletti, S., Bianconi, G., Criado, R., del Genio, C., Gómez-Gardeñes, J., Romance, M., Sendiña-Nadal, I., Wang, Z., and Zanin, M. (2014). The structure and dynamics of multilayer networks. *Physics Reports*, 544(1):1 – 122.

[9] Bryant (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

[10] Chaudhari, N., Saini, M., Kumar, A., and Priya, G. (2016). A review on attribute based encryption. In *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 380–385.

[11] Chintalapudi, S. R. and Prasad, M. H. M. K. (2015). A survey on community detection algorithms in large scale real world networks. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1323–1327.

[12] da F. Costa, L., Rodrigues, F., Travieso, G., and Villas Boas, P. (2007a). Characterization of complex networks: A survey of measurements. *Advances in Physics*, 56:167–242.

[13] da F. Costa, L., Rodrigues, F., Travieso, G., and Villas Boas, P. (2007b). Characterization of complex networks: A survey of measurements. *Advances in Physics*, 56:167–242.

[14] de Arruda, G. F., Rodrigues, F. A., and Moreno, Y. (2018). Fundamentals of spreading processes in single and multilayer complex networks. *Physics Reports*, pages 19–24.

[15] Dorogovtsev, S. N. and Mendes, J. F. F. (2003). *Evolution of Networks: From Biological Nets to the Internet and WWW (Physics)*. Oxford University Press, Inc., New York, NY, USA.

[16] Duncan, R. (1990). A survey of parallel computer architectures. *Computer*, 23(2):5–16.

[17] Flynn, M. (2011). *Flynn's Taxonomy*, pages 689–697. Springer US, Boston, MA.

[18] Garcia-Robledo, A., Diaz-Perez, A., and Morales-Luna, G. (2017). Accelerating all-sources bfs metrics on multi-core clusters for large-scale complex network analysis. In Barrios Hernández, C. J., Gitler, I., and Klapp, J., editors, *High Performance Computing*, pages 61–75, Cham. Springer International Publishing.

[19] Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826.

[20] Goyal, V., Jain, A., Pandey, O., and Sahai, A. (2008). Bounded ciphertext policy attribute based encryption. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 579–591, Berlin, Heidelberg. Springer-Verlag.

[21] Goyal, V., Pandey, O., Sahai, A., and Waters, B. (2006). Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 89–98, New York, NY, USA. ACM.

[22] Hafez, A. I., Hassanien, A. E., and Fahmy, A. A. (2014). *Testing Community Detection Algorithms: A Closer Look at Datasets*, pages 85–99. Springer International Publishing, Cham.

[23] J (2012). Evolution and trends in gpu computing. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 289–294.

[24] Jiang, Y., Susilo, W., Mu, Y., and Guo, F. (2016). Ciphertext-policy attribute based encryption supporting access policy update. In Chen, L. and Han, J., editors, *Provable Security*, pages 39–60, Cham. Springer International Publishing.

[25] Jin, S., Lin, W., Yin, H., Yang, S., Li, A., and Deng, B. (2015). Community structure mining in big data social media networks with mapreduce. *Cluster Computing*, 18(3):999–1010.

[26] Moon, S., Lee, J.-G., Kang, M., Choy, M., and Lee, J.-w. (2016a). Parallel community detection on large graphs with mapreduce and graphchi. *Data Knowl. Eng.*, 104(C):17–31.

[27] Moon, S., Lee, J.-G., Kang, M., Choy, M., and woo Lee, J. (2016b). Parallel community detection on large graphs with mapreduce and graphchi. *Data and Knowledge Engineering*, 104:17 – 31.

[28] Moradi, E., Fazlali, M., and Malazi, H. T. (2015). Fast parallel community detection algorithm based on modularity. In *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, pages 1–4.

[29] Navarro, C. A., Hitschfeld-Kahler, N., and Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329.

[30] Newman, M. (2003). The structure and function of complex networks. *SIAM Review*, 45(2):167–256.

[31] of Web Science, I. and at the University of Koblenz–Landau, T. (2013). the Koblenz Network Collection. `http://konect.uni-koblenz.de`. Consultado el 17 de diciembre de 2019.

[32] Onnela, J.-P., Fenn, D. J., Reid, S., Porter, M. A., Mucha, P. J., Fricker, M. D., and Jones, N. S. (2012). Taxonomies of networks from community structure. *Phys Rev E Stat Nonlin Soft Matter Phys*, 86(3 0 2):036104–036104. 23030977[pmid].

[33] Salehi, M., Sharma, R., Marzolla, M., Magnani, M., Siyari, P., and Montesi, D. (2015). Spreading processes in multilayer networks. *IEEE Transactions on Network Science and Engineering*, 2(2):65–83.

[34] Soman, J. and Narang, A. (2011a). Fast community detection algorithm with gpus and multicore architectures. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 568–579.

[35] Soman, J. and Narang, A. (2011b). Fast community detection algorithm with gpus and multicore architectures. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 568–579.

[36] Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393:440 EP –.

[37] Xu, R., Wang, Y., and Lang, B. (2013). A tree-based cp-abe scheme with hidden policy supporting secure data sharing in cloud computing. *Proceedings - 2013 International Conference on Advanced Cloud and Big Data, CBD 2013*, pages 51–57.

[38] Zarandi, F. D. and Rafsanjani, M. K. (2018). Community detection in complex networks using structural similarity. *Physica A: Statistical Mechanics and its Applications*, 503:882 – 891.

[39] Zhang, X.-K., Ren, J., Song, C., Jia, J., and Zhang, Q. (2017). Label propagation algorithm for community detection based on node importance and label influence. *Physics Letters A*, 381(33):2691 – 2698.

[40] Zoidi, O., Fotiadou, E., Nikolaidis, N., and Pitas, I. (2015). Graph-based label propagation in digital media: A review. *ACM Comput. Surv.*, 47(3):48:1–48:35.