Universidad Tecnológica de la Mixteca

PARALELIZACIÓN DE MÉTODOS DE APRENDIZAJE SEMI-SUPERVISADO BASADOS EN GRAFOS

Tesis

para obtener el título de
Maestro en Tecnologías de Cómputo Aplicado

Presenta:
Ing. Moisés Emmanuel Ramírez Guzmán

Director de tesis:
Dr. Raúl Cruz Barbosa

Huajuapan de León, Oaxaca, México Junio de 2018

Agradecimientos

Agradezco infinitamente a la mujer, amiga y compañera más importante de mi vida, mi esposa Celia, y a mis dos hijos, Iker e Ian. Ustedes son mi estímulo y apoyo constante en cada paso que doy.

De igual manera, doy gracias a mis padres, hermanas y hermanos que han estado apoyándome en todo momento.

Agradezco a la Universidad Tecnológica de la Mixteca, por brindarme la oportunidad y el espacio para realizar mis estudios superiores y de posgrado.

Este trabajo no habría sido posible sin el constante apoyo del Dr. Raúl Cruz Barbosa, quien ha dispuesto de su tiempo, consejos, paciencia y conocimientos para ser así un pilar fundamental en mi formación académica y durante todo el tiempo requerido para el buen desarrollo del presente trabajo.

Finalmente, agradezco a mis sinodales, Dra. Lluvia Carolina Morales Reynaga, M.C. Felipe Santiago Espinosa y al Dr. José Anibal Arias Aguilar por el tiempo invertido en la revisión de este trabajo.

Este trabajo de tesis fue apoyado parcialmente por el Consejo Nacional de Ciencia y Tecnología (CONACYT) en el marco del proyecto Cátedra-CONACYT número 1170.

Índice general

Índice de Figuras

Índice de Tablas

Publicación derivada

La publicación generada a partir del presente trabajo de tesis es:

- Ramírez-Guzmán Moisés, Cruz-Barbosa Raúl, **"Parallelization of semi-supervised learning algorithms"**, 8^{th} International Supercomputing Conference in Mexico, Marzo de 2017.

En la publicación se presenta la importancia del aprendizaje semi-supervisado basado en grafos y su aplicación en el área de clasificación. Se muestra la implementación paralela de los algoritmos de Propagación de etiquetas y de Propagación de etiquetas usando el criterio del costo cuadrático. Así mismo, se realiza un análisis de los resultados mostrados a través de gráficas de rendimiento en clasificación y aceleración obtenidas del presente trabajo.

Resumen

El aprendizaje computacional es un área de estudio de algoritmos para obtener predictores basados en información obtenida de experiencias pasadas. Consiste en métodos que permiten que un sistema aprenda a descubrir patrones, tendencias y relaciones entre los datos, que pueden ser usados para la solución de problemas en áreas de ingeniería.

El proceso de aprendizaje puede dividirse en tres tipos: aprendizaje supervisado, no-supervisado, y semi-supervisado. Los algoritmos de aprendizaje semi-supervisado tienen como característica principal que requieren una cantidad reducida de datos con información de clase para su entrenamiento, pero son capaces de aprovechar además la información geométrica de una gran cantidad de elementos que no tienen etiqueta. Estos algoritmos pueden alcanzar rendimientos similares o superiores a los supervisados.

Dentro de las técnicas de Aprendizaje Semi-Supervisado, sobresalen los métodos basados en grafos que representan cada muestra como un nodo y sus relaciones como arcos. Entre sus principales ventajas destacan que hay métodos matemáticos muy sólidos para interpretar y obtener sus propiedades; así mismo, a menudo tienen un objetivo global convexo, presentando altas garantías de convergencia. Esto último resulta atractivo para aplicarlo en problemas con cantidades grandes de datos que pueden ser modelados con grafos de manera natural como son: clasificación de páginas Web, sistemas de seguridad, reconocimiento de voz, entre otros. Para el procesamiento de grafos grandes se han desarrollado métodos muy sólidos que permiten soluciones escalables usando Cómputo de Alto Rendimiento.

El Cómputo de Alto Rendimiento es una herramienta que permite tratar problemas a una velocidad mucho mayor cuando existe una forma de distribuir y paralelizar el procesamiento de los cálculos. Por otro lado, existen problemas que requieren una gran cantidad de datos, y su almacenamiento en equipos comunes no es posible. Otra ventaja destacable de esta tecnología consiste en que es posible distribuir los archivos para su procesamiento en sistemas de archivos distribuidos, o simplemente a través de varios equipos interconectados por una red local, es decir, existen las condiciones para acceder a ellos en el momento que se requieran de manera independiente.

Los algoritmos paralelizados en el presente proyecto de tesis son: el Algoritmo de Propagación de Etiquetas, que tiene por idea principal que puntos cercanos deben tener etiquetas similares, de esta forma, los nodos etiquetados propagan sus etiquetas a los nodos vecinos que no la tienen. La segunda implementación consiste en el Algoritmo de Propagación de Etiquetas usando el Criterio del Costo Cuadrático, éste trata de aprovechar la información generada por la geometría de las relaciones entre los nodos usando la

regla de los k-vecinos más cercanos y una medida de similaridad basada en la distancia Euclidiana.

En el presente trabajo, el uso del cómputo paralelo y distribuido permitió mejorar los tiempos de respuesta de los algoritmos al distribuir el procesamiento en diferentes unidades de procesamiento y/o otros equipos interconectados a través de una red. Las pruebas realizadas fueron hechas sobre conjuntos de datos medianos y grandes, el rendimiento de clasificación obtenida en ambos algoritmos es similar a los de otros autores usando otros algoritmos de aprendizaje supervisado y semi-supervisado. En este trabajo además se reportan las mejoras en tiempo de ejecución de entre 3x a 7x veces para la implementación paralela con respecto a la implementación secuencial y se reporta el procesamiento de un conjunto con hasta 80,000 elementos usando una implementación distribuida.

Capítulo 1

Introducción

El Aprendizaje Computacional (AC o ML, del inglés, *Machine Learning*) es una rama de la Inteligencia Artificial que pretende aprender una función o proceso, el cual se da a partir de casos conocidos que cuentan con una solución de un problema en particular, aunque no se conozca dicha solución para todos los casos posibles [Alpaydin, 2010, Burch, 2001]. Estas técnicas se utilizan principalmente para resolver problemas donde no existe una regla de correspondencia o fórmula específica para su solución.

Generalmente, los algoritmos de aprendizaje computacional actuales trabajan bien con conjuntos de datos pequeños o medianos y cuando se realiza un análisis y extracción de información de estos conjuntos se usan algoritmos secuenciales. Sin embargo, cuando se requiere procesar grandes volúmenes de información se observan limitantes de tiempo, procesamiento y memoria. Es aquí donde surge la importancia del uso de cómputo paralelo y distribuido para reducir dichas limitantes.

Los métodos de aprendizaje tradicionales usan únicamente ejemplos etiquetados para realizar el proceso de entrenamiento. La obtención de etiquetas es un proceso generalmente difícil, caro o que requiere cantidades significativas de tiempo, así como la participación de expertos humanos. El Aprendizaje Semi-Supervisado (ASS o SSL, del inglés *Semi-supervised Learning*) es un área particular del ML que estudia los fundamentos para construir clasificadores usando información inherente en los elementos que no tienen etiqueta que en muchas aplicaciones reales son muy abundantes, y una cantidad generalmente reducida de elementos con etiqueta [Zhu, 2008].

En los algoritmos de SSL destacan los métodos basados en grafos (MBG o GBM, del inglés *Graph-Based Methods*) debido a que han tenido mejor rendimiento que otros algoritmos de SSL en evaluaciones comparativas [Chapelle et al., 2006]. Generalmente estos métodos tienen objetivos convexos, proporcionando garantías de convergencia, que los hacen atractivos para problemas que pueden escalar en tamaño. Además, la optimización del objetivo puede llevarse a cabo usando paso de mensajes, permitiendo que en cada iteración se obtenga una mejora [Bekkerman et al., 2011]. Los GBM modelan cada instancia de información como un nodo y las relaciones de similaridad como los pesos de los arcos que las interconectan, haciéndolos una representación directa de datos para muchos problemas reales [Mayer-Shönberger and Cukier, 2013]. Por lo tanto, los GBM representan un nicho importante para el estudio y aplicación de técnicas basadas en grafos usando

tecnologías de Cómputo de Alto Rendimiento (HPC, del inglés *High Performance Computing*). Finalmente, cabe agregar que la mayor cantidad de GBM tienen una complejidad en órdenes de $O(n^3)$ o superiores [Zhu, 2008, Kajdanowicz et al., 2014].

Debido al incremento de importancia del manejo de volúmenes grandes de información en las últimas décadas [Zikopoulos et al., 2011] [Jeffries, 2014] [Ginsberb et al., 2009], el presente proyecto está orientado al estudio y la implementación paralela y distribuida de dos algoritmos de SSL basados en grafos. Las pruebas realizadas son hechas sobre conjuntos de datos medianos y grandes para obtener mediciones en rendimiento de clasificación y mejora de velocidad de ejecución. Ambos algoritmos realizan operaciones sobre estructuras de datos que modelan la información almacenada en el grafo, para estos casos se detectan secuencias de operaciones (tareas) que pueden ejecutarse de manera independiente (paralelismo de datos). Las tareas paralelas son asignadas a varios procesadores disponibles para su ejecución sobre un solo equipo (o nodo). En el caso de la implementación distribuida se hace uso de una red de interconexión hacia otros equipos disponibles (paralelismo de tareas) para enviar y recibir datos que permiten sincronizar diversas partes del algoritmo a través de paso de mensajes. Para los dos algoritmos implementados se tratan problemas como la representación de los grafos (y su acceso paralelo y distribuido), y la planificación y distribución de carga de ejecución paralela y/o distribuida.

La metodología usada para el presente trabajo, consiste en realizar una investigación detallada acerca de los MBG de SSL. Enseguida, se identifican las operaciones que pueden ser realizadas de manera secuencial, paralela y/o distribuida. Después de tener la definición de las tareas y un análisis de sus posibles particiones independientes para su ejecución paralela y/o distribuida se procede a las implementaciones correspondientes. En la parte final, se realizan pruebas para calcular mediciones en tiempo y rendimiento de clasificación sobre conjuntos de datos con características sugeridas en la literatura.

1.1. Planteamiento del problema

La paralelización de métodos basados en grafos de Aprendizaje Semi-Supervisados tiene aplicaciones en problemas de clasificación que inciden en áreas como: visión artificial, diagnóstico médico, reconocimiento de patrones, procesamiento de lenguaje natural, entre otros [Mohri et al., 2012]. En estos problemas deben procesarse conjuntos grandes de datos con pocos elementos de los cuales se conoce su etiqueta de clase, en comparación con los elementos que no la tienen. Aun cuando existen algoritmos que han mostrado evidencia de mejoría en rendimiento, se han reportado soluciones principalmente a problemas con cantidades pequeñas o medianas de datos, mismos que han sido procesados de manera secuencial [Chapelle et al., 2006].

Los avances en la tecnología de construcción de equipos de cómputo han permitido aumentos considerables en la velocidad y capacidad de procesamiento pero no lo suficiente para cubrir las necesidades de procesamiento de aplicaciones reales. Para esto último, se han construido sistemas de cómputo de alto rendimiento con el objetivo de tratar problemas en los cuales se requiere procesar grandes cantidades de datos en un tiempo razonable. Sin embargo, cuando se implementan soluciones usando estos sistemas surgen

de manera natural, a nivel algorítmico y de implementación problemas como: eliminación y/o reducción de dependencias, distribución de la carga de procesamiento (*load balancing*), capacidad de almacenamiento y planificación de ejecución (*scheduling*) [Quinn, 2003].

Por lo anterior, en este trabajo de tesis se analizan dos algoritmos de SSL basados en grafos y se realizan implementaciones secuenciales, paralelas y distribuidas. Este proceso consiste en una implementación secuencial que tiene por objetivo el análisis de las partes que presentan mayor cantidad de carga de procesamiento numérico paralelizable; así mismo, se analizan las tareas a una granularidad mayor, siendo estas las que permitieron detectar las partes que pueden ser ejecutadas de manera distribuida. Al ser ambos problemas modelables con matrices, existe una cantidad importante de independencia entre las secuencias de cálculos, por lo que se procede a hacer la implementación paralela y distribuida de ambos algoritmos. Finalmente, se realizan pruebas para hacer mediciones de rendimiento en clasificación y aumento de velocidad de ejecución.

1.2. Justificación

En aplicaciones reales como el reconocimiento de voz, diagnóstico médico, entre otras, la obtención de conjuntos de datos donde se conoce la información o etiqueta de clase se caracteriza por ser muy costosa en tiempo y recursos. Por otro lado, la obtención de conjuntos de datos no etiquetados es en general mucho menos costosa. Los métodos de aprendizaje semi-supervisado tienen como principal rasgo el uso de pocos datos etiquetados y un gran número sin etiquetar. Es por esta concordancia con las aplicaciones reales que es una de las áreas importantes de estudio e investigación en los últimos años en el área de aprendizaje computacional, donde el objetivo principal es mejorar el rendimiento de métodos supervisados y no supervisados [Chapelle et al., 2006].

Es importante mencionar que la mayor parte de los avances en los métodos de aprendizaje semi-supervisado han sido usando algoritmos que se ejecutan en arquitecturas de cómputo secuenciales. Por lo tanto, la paralelización de estos algoritmos permite un aumento del rendimiento, obteniendo una reducción en el tiempo de ejecución y como consecuencia poder procesar conjuntos de datos que no podrían tratarse por limitaciones de tiempo y almacenamiento. Los dos algoritmos implementados en este trabajo fueron probados sobre conjuntos de datos medianos y grandes. El rendimiento de clasificación sobre los conjuntos de datos seleccionados es comparado con otros algoritmos de aprendizaje supervisado.

La implementación considera que durante la ejecución de los algoritmos tienen como requerimientos cantidades grandes de memoria y procesamiento que no están disponibles en equipos comunes. Para esto, en la Universidad Tecnológica de la Mixteca (UTM) se cuenta con infraestructura básica que es suficiente para la implementación de algoritmos en entornos paralelos y distribuidos, la cual se podría ampliar a mediano plazo mediante la participación en proyectos de investigación financiados.

Por otro lado, en la UTM no se ha trabajado con algoritmos paralelos de aprendizaje semi-supervisado, por lo que el presente proyecto de tesis permitirá sentar las bases para integrar una línea de investigación en la institución.

1.3. Hipótesis

Con la implementación y adaptación de métodos de aprendizaje semi-supervisado basados en grafos para su ejecución en arquitecturas de cómputo paralelas y distribuidas, es posible obtener mejoras de rendimiento en soluciones a problemas que requieren procesamiento de grandes volúmenes de datos, y al mismo tiempo reducir los tiempos de ejecución.

1.4. Objetivos

Objetivo general

Implementar y adaptar dos algoritmos de aprendizaje semi-supervisado basados en grafos que se puedan ejecutar en entornos paralelos y distribuidos para mejorar sus rendimientos correspondientes.

Objetivos particulares

- Elaborar una investigación del estado del arte acerca de algoritmos de aprendizaje semi-supervisado basados en grafos.
- Hacer una evaluación de herramientas para el desarrollo de aplicaciones paralelas y distribuidas de métodos de aprendizaje computacional.
- Realizar un estudio sobre diferentes métodos para la representación de grafos en problemas de aprendizaje semi-supervisado.
- Seleccionar e implementar dos métodos de aprendizaje semi-supervisado basados en grafos en versiones secuenciales, paralelas y distribuidas usando diferentes representaciones de grafos.
- Realizar comparaciones de rendimiento en las implementaciones descritas en el punto anterior.

1.5. Metas

A continuación se listan las metas buscadas en este trabajo de tesis:

- Reporte acerca de los algoritmos de aprendizaje semi-supervisado basados en grafos.
- Reporte de la comparación de herramientas para desarrollo de aplicaciones de aprendizaje computacional que se ejecutan sobre entornos paralelos y distribuidos; así como de las especificaciones y representación de grafos más adecuadas para los dos métodos a implementar.

- Implementación de los algoritmos secuenciales, paralelos y distribuidos usando las especificaciones descritas en el punto anterior.

- Reporte comparativo del desempeño de las implementaciones realizadas sobre los conjuntos de datos seleccionados.

1.6. Trabajo relacionado

El aprendizaje computacional es un área con una amplia cantidad de técnicas para análisis y predicción de información. En la era moderna, los volúmenes de datos en aplicaciones reales son tan grandes que ofrecen un nicho natural para el desarrollo de aplicaciones usando técnicas de desarrollo y arquitecturas de cómputo paralelas [Upadhyaya, 2013]. En [Thearling, 1993] se trata el incremento de rendimiento usando cómputo paralelo de algunos algoritmos populares de ML como son: razonamiento basado en casos, algoritmos genéticos y redes neuronales. [Zaki, 1999] hace un estudio acerca de los métodos usados en minería de datos basada en reglas (*Association Rule Mining*) en entornos paralelos y distribuidos analizando temas como el balanceo de carga y tipos de paralelismo usado, destaca problemas como la alta dimensionalidad, localización de los datos y balanceo de carga, entre otros.

El aprendizaje semi-supervisado se basa en las suposiciones de suavidad, variedad y agrupamiento aplicado a problemas donde existe una cantidad muy reducida de datos etiquetados en comparación con los que no tienen etiqueta de clase. Por ejemplo, en el caso de navegación autónoma de robots con capacidades de almacenamiento limitadas, los modelos de visión computacional requieren clasificadores eficientes a partir de los pocos modelos disponibles como referencia [Zhu et al., 2009].

Una cantidad importante de problemas técnicos y científicos tienen una naturaleza de red en áreas como: relaciones de citas en publicaciones, modelos de propagación de enfermedades, análisis de comportamiento de usuarios en la web, redes de transporte, redes sociales, análisis de mercado, entre otros. Los grafos son una representación muy eficiente y flexible para modelar estos fenómenos ya que permiten abstraer de manera natural relaciones entre objetos discretos. Existen muchos problemas que han sido resueltos con algoritmos basados en grafos, sin embargo, a medida que los grafos crecen en escala y complejidad, se vuelven insuficientes las capacidades de almacenamiento en memoria y procesamiento de métodos comunes [Lumsdaine et al., 2007, Kajdanowicz et al., 2014].

Para el tratamiento de problemas con conjuntos grandes de datos han surgido herramientas como: *Hadoop* que usa el enfoque *Map-Reduce* [Zikopoulos et al., 2011], *The IBM Parallel Machine Learning Toolbox*[1] que usa ideas del enfoque *Map-Reduce* y las combina con MPI para su implementación distribuida [Bekkerman et al., 2011], y Pregel cuyos objetivos son soportar grandes cantidades de información, alta tolerancia a fallos, y ser herramientas que faciliten el desarrollo de aplicaciones paralelas y distribuidas [Malewicz et al., 2010].

[1] http://www.research.ibm.com/haifa/projects/verification/ml_toolbox/

Por necesidades asociadas a la cantidad de datos existentes, en muchos casos los datos no están almacenados en un solo sitio, por lo que las herramientas han desarrollado sus propios sistemas de administración de archivos como HDFS (del inglés, *Hadoop Distributed File System* [Karanth, 2014]), o contienen protocolos que permiten la comunicación entre plataformas heterogéneas como también lo hace Hadoop y lo contempla de manera automática MPI.

Algunas implementaciones a destacar usando Hadoop son: PLANET que implementa de manera distribuida árboles de decisión [Panda et al., 2009], y que implementa algoritmos de aprendizaje computacional (como el clasificador Naïve Bayes, análisis discriminante gaussiano, *k-means*, regresión logística, redes neuronales, análisis de componente principal, máquinas de soporte vectorial, entre otros) usando memoria compartida [Chu et al., 2006].

Para un tipo particular de problemas con datos que pueden ser modelados con vectores y matrices, y además, la mayor parte de operaciones a realizar es sobre estos vectores, se han hecho desarrollos usando las unidades de procesamiento gráficas (GPU, por sus siglas en inglés). En [Bekkerman et al., 2011] se muestran detalles y comparativas de uso para el algoritmo k-means. Para el procesamiento de grandes volúmenes de información en aplicaciones reales, [di Blas and Kaldeywey, 2009] destaca el uso de unidades especializadas de procesamiento gráfico en aplicaciones como biología molecular, diagnóstico médico o biología, así como su importancia para el procesamiento de datos para empresas como eBay, Netflix o Google.

En el área de aprendizaje semi-supervisado para problemas basados en grafos grandes uno de los casos de estudio es el *Switchboard Transcription Project* [2] aplicado al problema de reconocimiento de voz. Para su implementación se usa un grafo con 118 millones de nodos de grado 10 [Subramanya and Bilmes, 2009, Bekkerman et al., 2011].

1.7. Metodología

Para el problema tratado en el presente trabajo se usa la metodología mostrada en la figura 1.1. Como primer paso se realiza una investigación del estado del arte de los algoritmos de Aprendizaje Semi-Supervisado y de herramientas para desarrollo de software para cómputo de alto rendimiento.

En el caso de SSL, se estudian a detalle los Métodos basados en Grafos, que sugieren mayores ventajas sobre otros algoritmos de SSL, principalmente para su aplicación en conjuntos de datos grandes [Bekkerman et al., 2011]. Después de analizar los algoritmos de SSL, los Algoritmos de Propagación de Etiquetas y de Propagación de Etiquetas usando el Criterio de Costo Cuadrático son seleccionados para su implementación. Para ambos algoritmos se revisan detalles como: opciones para representar los vértices y aristas de los grafos, métricas para definir los pesos de los arcos entre los nodos y las operaciones que se requieren para cada una de las implementaciones.

[2] S3TP: The Semi-Supervised Switchboard Transcription Project, http://melodi.ee.washington.edu/s3tp/

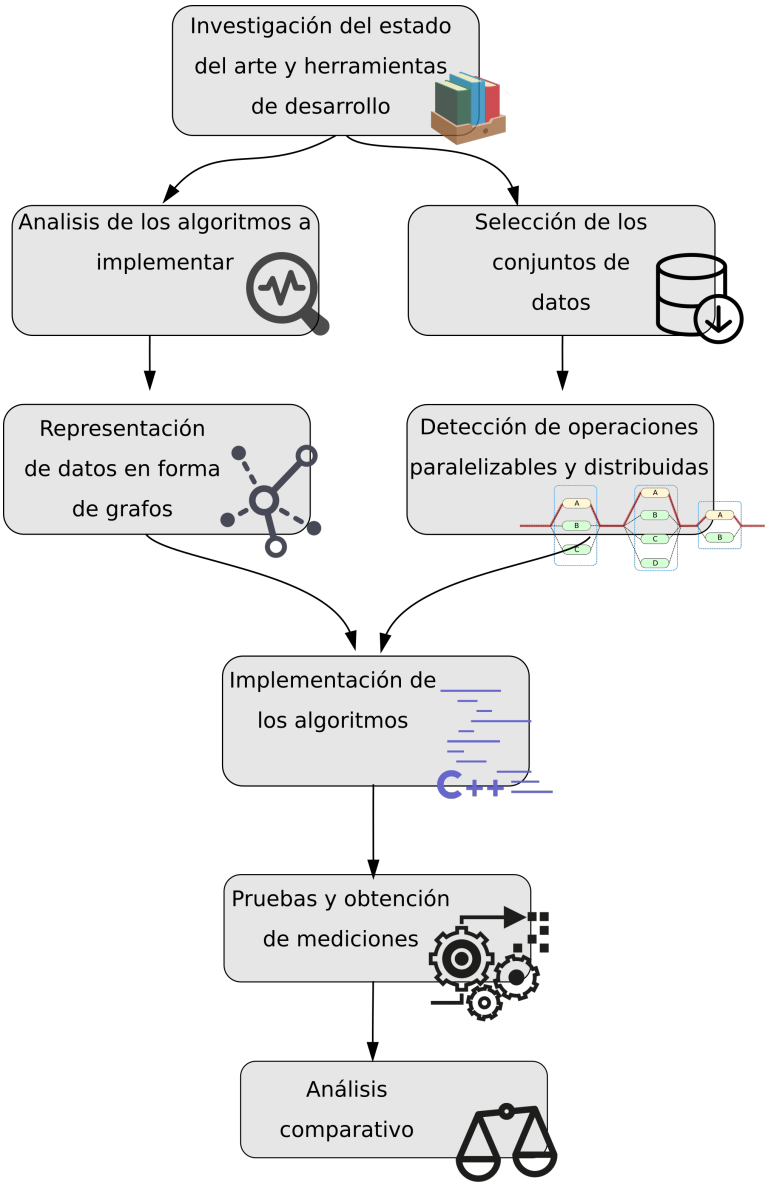


Figura 1.1: Metodología usada en el presente trabajo.

Ambos algoritmos presentan una cantidad importante de operaciones paralelizables sobre vectores y matrices que además pueden ser distribuidas. Al detectar las operaciones que deben ser ejecutadas, se elige el sistema operativo Ubuntu, y como lenguajes de desarrollo C y C++ con las bibliotecas OpenMP y MPI. A partir de lo anterior, se realizan las implementaciones paralelas y distribuidas de una biblioteca de operaciones sobre vectores y matrices, para su posterior uso en las implementaciones de ambos algoritmos.

Tomando como referencia la literatura [Chapelle et al., 2006] sobre algoritmos de SSL, se identifican y seleccionan conjuntos de datos con características que se sugiere probar. Los resultados de clasificación sobre los conjuntos de datos seleccionados son comparados contra la respuesta obtenida usando otros algoritmos documentados. El rendimiento en

tiempo es calculado para validar las implementaciones paralelas y distribuidas, esto se realiza al hacer variaciones en la cantidad de unidades de procesamiento para obtener las medidas de incremento de velocidad y de reducción de tiempo de ejecución considerando la Ley de Amdahl [Patterson and Hennessy, 2006].

El desarrollo de los módulos para los algoritmos es realizado tomando en cuenta los puntos anteriores, haciendo pruebas y comparativas con cada uno de los conjuntos de datos seleccionados.

La redacción de este trabajo se realizó durante el periodo de desarrollo de la tesis.

Capítulo 2

Marco Teórico

El aprendizaje computacional ha tenido éxito en problemas de clasificación donde no se cuenta con un algoritmo específico para tareas como son: reconocimiento de voz para la transcripción de conversaciones [Evermann et al., 2005], clasificación de documentos [Hua et al., 2012, Bsoul et al., 2013], detección de correo no deseado [Tretyakov, 2004], o detección de fraudes para aseguradoras [Kirlidog and Asukb, 2012], por citar algunos.

El aprendizaje computacional consiste en detectar patrones, que al ser usados permitan reconocer procesos o realizar predicciones. El conjunto de algoritmos de aprendizaje computacional puede ser utilizado en los campos de aplicación como son: *minería de datos, visión por computadora, reconocimiento de voz y robótica*, entre otros [Alpaydin, 2010, Zhu et al., 2009].

Este capítulo contiene una breve introducción a los métodos de aprendizaje computacional. En la sección 2.1 se revisa su clasificación basada en el método de entrenamiento y en la sección 2.2 se estudian las principales técnicas de SSL. Luego, en la sección 2.3 se describen los métodos basados en grafos, incluyendo de los métodos más comunes para su representación y construcción. Además, se analizan los dos algoritmos que se implementan de manera secuencial, paralela y distribuida. En la sección 2.4 se revisa la necesidad de usar herramientas de cómputo de alto rendimiento para procesar grandes volúmenes de información. Aquí, se introduce el modelo de programación paralela usando OpenMP y de programación distribuida usando la interfaz de paso de mensajes.

2.1. Categorización de los métodos de aprendizaje

Uno de los objetivos del aprendizaje computacional es obtener la mejor actuación de un sistema en diferentes entornos. Debido a esto, el proceso de aprendizaje puede ser dividido en tres tipos:

- **Aprendizaje supervisado**

  Durante este proceso los algoritmos son entrenados con diferentes ejemplos en los que la solución es conocida [Harrington, 2012]. El objetivo es obtener una función general que permita mapear un conjunto de entradas a salidas conocidas. Esto es, encontrar la imagen o salida adecuada de puntos o entradas que no han sido consideradas en el conjunto de entrenamiento.

  Generalmente, las salidas pueden ser valores nominales (como *cierto* o $falso$, o valores como: reptil, pez, mamífero, planta, etc.). Al proceso de encontrar correspondencias entre entradas de un dominio y un codominio con valores nominales se le conoce como *clasificación*. En caso de que las salidas sean valores continuos, a dicho proceso se le conoce como *regresión*.

- **Aprendizaje no supervisado**

  Para estos métodos, las salidas no son conocidas por lo que el objetivo de estos algoritmos es descubrir regularidades en la estructura y relación entre los datos de entrada. En estos procesos no existe una señal que retroalimente o evalúe una solución potencial.

  Un caso de este modelo de aprendizaje es el análisis de agrupamientos (*clusters*).

- **Aprendizaje semi-supervisado**

  El aprendizaje semi-supervisado es una combinación de los dos tipos anteriores. Estos algoritmos consideran entradas con una cantidad pequeña de salidas conocidas (etiquetas) y una cantidad grande de salidas no conocidas [Chapelle et al., 2006]. La importancia del SSL es que puede realizar tareas de aprendizaje supervisado o no supervisado a partir de pocos datos etiquetados.

2.2. Aprendizaje semi-supervisado

Los algoritmos de aprendizaje semi-supervisado se fundamentan en suposiciones acerca de los datos a procesar. Así dichas suposiciones son utilizadas por distintos enfoques.

Desde un punto de vista matemático, podría decirse que si se conoce $P(\mathcal{X})$ acerca de los datos no etiquetados, esta información podría ser útil para hacer inferencias acerca de $p(y|x)$, donde y es la etiqueta de clase, de no ser así el SSL no podría aplicarse. Al usar SSL se hacen algunas suposiciones acerca de la distribución de los elementos de $\mathcal{X}$, que permiten reducir las desviaciones de respuestas del algoritmo [Chapelle et al., 2006, Bekkerman et al., 2011]:

- **Suposición de Suavidad**: Si dos puntos x_1 y x_2 en una región de alta densidad son cercanos, entonces lo serán también las respuestas y_1, y_2.

- **Suposición de Agrupamiento**: Si hay puntos que pertenecen a un mismo grupo (clúster), entonces es probable que pertenezcan a la misma clase.
- **Suposición de Variedad**: Ésta afirma que los datos con alta dimensionalidad subyacen en una variedad de baja dimensión.

Estas suposiciones se cumplen para muchas aplicaciones del mundo real y es la razón por la que el SSL funciona actualmente.

Algunos enfoques del aprendizaje semi-supervisado

Dentro del SSL destacan algunas técnicas como son:

- **Auto-aprendizaje**: también conocido como *auto-entrenamiento*, *auto-etiquetamiento* o aprendizaje *dirigido por decisiones* (*bootstrapping*) [Zhu, 2008], consiste en:
  - Al inicio, se realiza el proceso de entrenamiento sólo con los datos etiquetados para generar etiquetas para un subconjunto de datos no etiquetados que se considera como un subconjunto altamente confiable. Enseguida, se agrega el nuevo subconjunto al conjunto etiquetado para realizar el entrenamiento. En las siguientes iteraciones, se elige un subconjunto de los datos no etiquetados y se vuelve a entrenar con los datos etiquetados. En este proceso el clasificador usa sus propias predicciones para entrenarse a sí mismo.
- **Co-entrenamiento** [Blum and Mitchell, 1998]: para este método se asume que:
  - Las características pueden ser divididas en dos conjuntos.
  - Cada subconjunto de características es suficiente para entrenar un buen clasificador.
  - Los dos subconjuntos son condicionalmente independientes dada la etiqueta.

  En este enfoque se entrenan dos clasificadores de manera separada con los datos etiquetados para los subconjuntos respectivos, entonces cada clasificador genera una etiqueta para los datos no etiquetados y enseña al otro clasificador con estos datos. Cada clasificador es re-entrenado con los ejemplos de entrenamiento adicionales dados por el otro clasificador. Para este proceso cada clasificador debe confiar en las etiquetas generadas por el otro.
- **Algoritmos basados en grafos** [Bekkerman et al., 2011, Zhu, 2008]: Estos algoritmos asumen que los datos subyacen en una variedad de baja dimensionalidad, la cual puede ser representada por un grafo. Aquí cada dato está representado por un vértice, los pesos de las aristas representan una medida de la similitud entre los vértices que se conectan. Estos métodos son no-paramétricos, discriminativos y transductivos por naturaleza.

Algunas aplicaciones del aprendizaje semi-supervisado

En muchos casos la adquisición de casos con salidas/soluciones conocidas requieren la dedicación de una persona experta o la elaboración de un experimento físico que implican costos y/o tiempos asociados grandes. Por otro lado, la obtención de entradas con salidas no conocidas suelen tener costos asociados muy bajos. Algunos ejemplos de aplicaciones donde se requiere un número pequeño de datos con etiquetas y un gran número sin ellas son:

- **Reconocimiento de voz**: es relativamente simple adquirir una gran cantidad de audios, pero requiere gran cantidad de tiempo que una persona transcriba lo que escucha. En [Subramanya and Bilmes, 2009] se muestra el uso de SSL en el *Switchboard Transcription Project* que busca la transcripción de un conjunto de 2,400 conversaciones entre 543 personas.

- **Detección de correo no deseado**: se puede obtener una gran cantidad de correos en algún servidor, pero se requiere de personas que los clasifiquen. [Tretyakov, 2004] muestra algoritmos de aprendizaje computacional usados para este problema.

- **Secuencias de proteínas**: pueden ser adquiridas a velocidad industrial (por secuenciación genómica, búsqueda computacional de genes o traducción automática), pero resolver su estructura 3D para determinar las funciones de una sola proteína requiere expertos en el área durante periodos largos de tiempo.

- **Clasificación de documentos y páginas Web**: existen millones de páginas en el Internet para ser procesadas, pero clasificarlas requiere de personas que las lean. En [Subramanya and Bilmes, 2008] se propone y analiza un algoritmo de SSL para la clasificación de texto, llegando a eficiencias de alrededor de 80 % de exactitud de clasificación.

2.3. Aprendizaje semi-supervisado basado en grafos

Existe una cantidad importante de problemas que pueden modelarse mediante la representación de grafos como son: redes de comunicación, redes sociales, ligas o referencias dentro de una página Web, entre otros. Los mtodos de aprendizaje semi-supervisado basados en grafos (GBSSL, por sus siglas en inglés) usan la estructura del grafo que representa tanto los ejemplos etiquetados como los no etiquetados para hacer el proceso de aprendizaje. Para aplicar estas técnicas, se debe asumir que los datos están embebidos en una variedad de baja dimensionalidad. Cada muestra está representada por un vértice y los pesos en las aristas representan una medida de similaridad entre pares de nodos. Algunas razones importantes para usar GBSSL son [Bekkerman et al., 2011, Subramanya and Talukdar, 2014]:

- **Tiempo**: tienen alto rendimiento al tener operaciones que permiten escalabilidad.
- **Objetivo convexo**: la mayoría de los GBSSL tienen objetivos convexos por lo que proveen altas garantías de convergencia, haciéndolos atractivos al momento de resolver problemas con grandes cantidades de datos.
- **Paso de mensajes**: la optimización del objetivo puede ser hecha por paso de mensajes. En cada iteración el algoritmo realiza un conjunto de actualizaciones en cada nodo, estos valores pueden ser calculados a partir de sus vecinos.
- **Garantía de convergencia**: los algoritmos con paso de mensajes convergen linealmente al óptimo global del objetivo convexo.
- **Mejora de eficiencia**: se pueden derivar heurísticas que permitan el escalamiento de los algoritmos correspondientes en equipos paralelos y distribuidos con muy buena eficiencia.

Los GBSSL pueden ser clasificados en dos categorías: a) Los que usan la estructura del grafo para propagar sus etiquetas a ejemplos no etiquetados, y b) los que optimizan una función de costo basado en restricciones derivadas del grafo.

2.3.1. Representación de información usando grafos

Un grafo $G = \{V, E\}$ es una forma natural de representar relaciones de vecindad a través de arcos (E) que tienen información entre instancias representadas por nodos o vértices (V) del conjunto de datos original. Esta representación garantiza la conservación de la estructura intrínseca del problema a modelar.

Las relaciones representadas por las aristas pueden ser medidas de similaridad como la similitud coseno o el kernel de la función de base radial (*radial basis function kernel*), o medidas de disimilaridad como son la distancia euclidiana o la distancia geodésica [Theodoridis and Koutroumbas, 2006, Bautista, 2011].

Los métodos para la construcción de grafos son un paso importante para la aplicación de los algoritmos basados en grafos. El grafo determina el flujo de la información de un nodo a otro, por lo que la decisión de una métrica incorrecta de vecindad puede producir rendimientos muy pobres.

Las reglas más comunes para construir grafos son [Lee and Verleysen, 2007]:

- **Regla *k***

  Para cada nodo, se agregan conexiones entre un nodo y sus k-vecinos más cercanos, como se muestra en la figura 2.1. Algunas características importantes de usar este método son:

- No hay garantía de que los grafos resultantes sean simétricos. Es decir si **B** está entre los k-vecinos más cercanos de un nodo **A**, no existe garantía de que **A** esté entre los k-vecinos más cercanos de **B**. Con el fin de garantizar la simetría en el grafo, para cada enlace agregado entre el nodo y su vecino cercano también se agrega otro enlace en dirección opuesta, esto es entre el vecino y el nodo analizado.
- Los grafos pueden resultar irregulares: algunos nodos pueden tener grados más altos que otros.

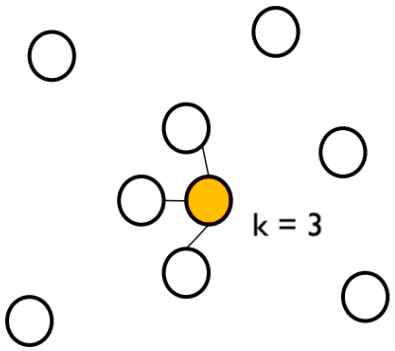


Figura 2.1: Aplicación de la regla k con conexión hacia los k-vecinos más cercanos.

- **Regla ϵ**

  Aquí se establece un radio ϵ, y se establecen conexiones desde el nodo hacia todos los que estén dentro de dicho radio como se muestra en la figura 2.2. Algunas características importantes al usar este método son:

  - El método no es escalable.
  - Es sensible al valor del radio ϵ.
  - Puede generar grafos fragmentados, y por lo tanto generar componentes no conectadas.

Modelos de representación de grafos

Hay dos formas principales de representar un grafo: listas de adyacencia y matrices de adyacencia. El modelo de representación depende directamente del contexto de la aplicación. Generalmente, se opta por representar el grafo mediante una lista de adyacencia, si éste es disperso (casos en que $|E|$ es mucho menor que $|V|^2$) ya que provee una manera compacta de representarlo. En el caso de grafos densos, generalmente se opta por usar una matriz de adyacencia [Bautista, 2011].

La representación de un grafo usando listas de adyacencia consiste en un vector $Adj[]$ de $|V|$ listas, una por cada nodo en V. Para cada $u \in V$, la lista de adyacencia $Adj[u]$

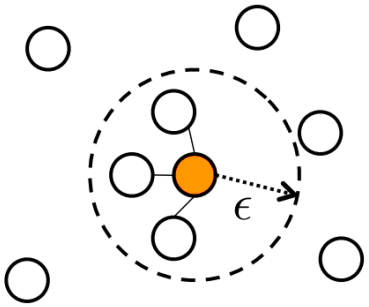


Figura 2.2: Aplicación de la regla ϵ con conexión hacia los vecinos dentro de una vecindad de radio ϵ.

contiene todos los vértices $v \in V$, hacia los cuales existe un arco $(u, v) \in E$ (ver Figura 2.3).

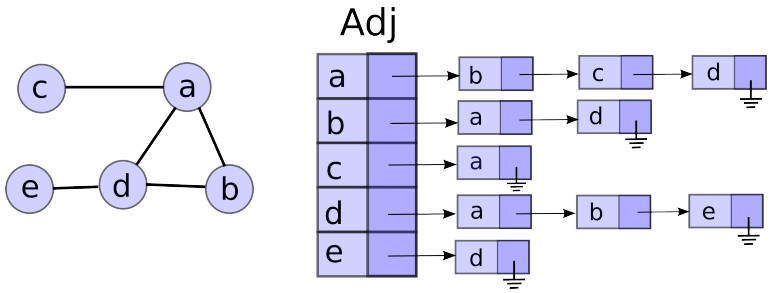


Figura 2.3: Ejemplo de una lista de adyacencia para representar un grafo.

Para la representación de un grafo usando una matriz de adyacencia, se asume que los vértices están enumerados $1, 2, \ldots, |V|$, de tal forma que la representación consiste en una matriz $A = (a_{i,j})$ tal que

$$a_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{en cualquier otro caso.} \end{cases} \tag{2.1}$$

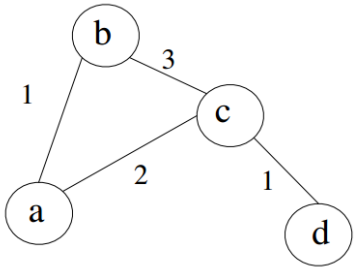
Dependiendo del problema, el valor 1 que corresponde a la existencia o no de una arista, puede ser reemplazado por un valor de similaridad o disimilaridad indicando una relación entre los elementos del conjunto.

La matriz laplaciana de un grafo es otra forma de representación que involucra matrices de adyacencia. Ésta se define como la diferencia entre la matriz de grado de los nodos $\mathbf{D}$ y la matriz de adyacencia $\mathbf{W}$:

$$\mathcal{L} = \mathbf{D} - \mathbf{W} \tag{2.2}$$

donde la matriz de grado de los nodos se define como:

- $\mathbf{D}_{ii} = \sum_j \mathbf{W}_{ij}$
- $\mathbf{D}_{ij(i \neq j)} = 0$



$$\mathcal{L} = \begin{bmatrix} 3 & -1 & -2 & 0 \\ -1 & 4 & -3 & 0 \\ -2 & -3 & 6 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Figura 2.4: Ejemplo de un grafo usando la matriz laplaciana.

La figura 2.4 muestra un grafo y su matriz laplaciana. La matriz laplaciana normalizada de un grafo es otra representación usada comúnmente y se define como:

$$L(G) = D^{-\frac{1}{2}}\ L\ D^{-\frac{1}{2}} \begin{cases} 1 & \text{si } \ i = j \ \text{ y } \ grado(v_j) \neq 0 \\ -\frac{1}{[grado(v_i) \times grado(v_j)]^{1/2}} & \text{si } i \neq j \ \text{ y } v_i \text{ es adjacente a } v_j \\ 0 & \text{en otro caso} \end{cases} \tag{2.3}$$

Necesidades de procesamiento paralelo y distribuido para grafos de tamaños grandes

En las aplicaciones modernas de tecnología de información se generan volúmenes de información cada vez más grandes. Muchas de estas aplicaciones procesan por su naturaleza y necesidad datos que pueden ser representados por grafos, como son los registros de usuarios en servicios de comunicación, correos electrónicos y redes sociales.

En el momento en que las cantidades de datos son más grandes, se hace necesario el uso del paradigma de programación paralela y distribuida para llevar a cabo los procesos de cómputo requeridos. Esto es debido a que en la mayoría de los casos los datos no pueden ser almacenados en la memoria local de las computadoras comunes. Las técnicas más usadas para desarrollar aplicaciones bajo este paradigma son las bibliotecas de Paso de Mensajes (MPI, por sus siglas en inglés) y el esquema *Map-Reduce* (Hadoop, ver [Zikopoulos et al., 2011]).

Los grafos tienen propiedades particulares que hacen que la implementación de algoritmos distribuidos sean un reto complicado aún cuando existen herramientas de procesamiento paralelo y distribuido [Kajdanowicz et al., 2014]:

- *El procesamiento está controlado por datos relacionales.* La mayoría de los algoritmos son ejecutados de acuerdo a la estructura del grafo, donde el procesamiento de cada nodo es estrictamente dependiente de los resultados calculados en sus antecesores. Esto significa que el algoritmo depende mayormente de la estructura más que de algún procesamiento secuencial.

- *Las estructuras no son balanceadas.* Normalmente los datos en los grafos no son balanceados y son irregulares, esto dificulta el particionamiento y balanceo de su procesamiento, y por lo tanto su escalabilidad.

- *Alta carga de acceso a datos.* Los algoritmos típicamente se dedican a explorar los grafos más que a realizar largas secuencias de cálculos en ellos. Por ejemplo, en los problemas en que se requiere encontrar las rutas más cortas, en donde la cantidad de operaciones aritméticas es pequeña comparada con las operaciones de acceso a los datos requeridas. Para este caso, son muy importantes los tiempos de acceso a la memoria en los diferentes niveles de la arquitectura del equipo de cómputo.

2.3.2. Algoritmo de propagación de etiquetas

El algoritmo de propagación de etiquetas (LP, del inglés *label propagation*) fue propuesto por Xiaojin Zhu y Zoubin Ghahramani en 2002. Usa la suposición de suavidad, la cual sugiere que si dos nodos están conectados por una arista con un valor de similaridad grande, entonces ambos nodos deberían ser muy cercanos, por lo que deben compartir la misma etiqueta. De esta forma la etiqueta de un nodo se propaga hacia sus vecinos no etiquetados de acuerdo a alguna medida de similaridad entre los pesos [Zhu and Ghahramani, 2002].

Descripción del algoritmo

Sean $(x_1, y_1)\cdots\ (x_l, y_l)$, ejemplos etiquetados, donde $\mathbf{Y}_L = \{y_1, \cdots y_l\}$ son etiquetas de clase. Se puede asumir que el número de clases $\mathbf{C}$ es conocido, y todas las clases están presentes en los ejemplos etiquetados. Se considera a $(x_{l+1}, y_{l+1})\cdots\ (x_{l+u}, y_{l+u})$ como ejemplos que no tienen etiqueta, donde $\mathbf{Y}_U = \{y_{l+1}, \cdots y_{l+u}\}$ no tienen etiqueta, usualmente $l \ll u$. Sea $\mathbf{X} = \{x_1, \cdots x_{l+u}\}$ donde $x_i \in \mathcal{R}^D$. El problema de estimar $\mathbf{Y}_U$ a partir de $\mathbf{Y}_L$ es un tipo de aprendizaje transductivo.

Intuitivamente, se puede asumir que los puntos cercanos tienen etiquetas similares. Para modelar esto, se puede crear un grafo completamente conectado donde los nodos son ejemplos etiquetados y no etiquetados. Las aristas entre los nodos i, j tienen un peso $w_{i,j}$ que puede ser modelado por una distancia euclidiana (u otras métricas que pueden ser más apropiadas de acuerdo al contexto del problema), y dichos valores de pesos pueden

ser calculados usando:

$$w_{ij} = exp\left(-\frac{\sum_{d=1}^{D}(x_i^d - x_j^d)^2}{\sigma^2}\right) \tag{2.4}$$

Todos los nodos tienen etiquetas que pueden ser modelados por distribuciones de probabilidad. Las etiquetas se propagan a través de las aristas, los pesos más altos tienen probabilidad más alta de ser propagados. La matriz de transición $\mathbf{T}$ (de dimensión $(l+u)\times(l+u)$), donde $\mathbf{T}_{ij}$ modela la probabilidad de ir de un nodo j a un nodo i se calcula con la siguiente expresión:

$$\mathbf{T}_{ij} = P(j \to i) = \frac{w_{ij}}{\sum_{k=1}^{l+u} w_{kj}} \tag{2.5}$$

Así mismo se define la matriz de etiquetas $\mathbf{Y}$ con dimensión $(l+u) \times \mathbf{C}$ donde la i-ésima fila representa la distribución de probabilidad del nodo x_i. La inicialización de filas en $\mathbf{Y}$ para ejemplos no etiquetados no tiene mayor relevancia.

El algoritmo de propagación de etiquetas se define de la siguiente manera [Zhu and Ghahramani, 2002]:

1. Propagación de las etiquetas: $\mathbf{Y} \leftarrow \mathbf{T} \cdot \mathbf{Y}$
2. Normalizar por filas a $\mathbf{Y}$.
3. Hacer persistentes los datos etiquetados en el vector $\mathbf{Y}$. Repetir desde el paso 1 hasta que $\mathbf{Y}$ converja.

En el paso 1, todos los nodos propagan sus etiquetas hacia sus vecinos. El paso 2 busca que los valores de $\mathbf{Y}$ permanezcan en un rango de 0 a 1 y el paso 3 busca la persistencia de las fuentes de etiquetas a partir de los nodos etiquetados, aplicando la influencia inicial de $\mathbf{Y}_L$ en $\mathbf{Y}$.

Obtención del valor σ

Para la obtención del valor σ de la ecuación 2.4 se hace una modificación al algoritmo de Kruskal. El algoritmo original calcula el árbol de expansión mínimo sobre un grafo conectado con aristas que tienen pesos. Éste encuentra el subconjunto de aristas que forman un árbol que conecta a todos los vértices de tal forma que la suma de los pesos de las aristas es mínima [Cormen et al., 2001].

La modificación realizada al algoritmo de Kruskal consiste en que se mantiene una lista de vértices con la misma etiqueta, y en el momento en que se detecta que se intentan

conectar dos componentes que contienen ejemplos con etiquetas diferentes, se detiene. El peso de la arista que conecta a las componentes con etiquetas diferentes tiene el valor d_0, de tal forma que la heurística a usar es tomar $\sigma = d_0/3$. Ver subsección A.1 para más detalles.

2.3.3. Algoritmo de Propagación de etiquetas usando el criterio del costo cuadrático

El criterio del costo cuadrático se deriva a partir de la minimización de una función de costo sobre el grafo G. Este criterio se obtiene de la búsqueda de las etiquetas del grafo que son consistentes con las etiquetas conocidas del grafo y la geometría de los datos inducidos a la estructura del grafo a través de los pesos y arcos en la matriz $\mathbf{W}$.

Descripción del algoritmo

A partir de las etiquetas conocidas ($\hat{Y_l}$) y las no conocidas ($\hat{Y_u}$) al iniciar el algoritmo, se genera el conjunto de etiquetas $\hat{Y}$ obtenidas como respuesta del algoritmo:

$$\hat{Y} = (\hat{Y_l}, \hat{Y_u}) \tag{2.6}$$

La consistencia con el etiquetamiento al iniciar el algoritmo puede ser medido con la expresión:

$$\sum_{i=1}^{l} (\hat{y_i} - y_i)^2 = ||\hat{Y_l} - Y_l||^2. \tag{2.7}$$

donde, $\hat{y_i}$ son las etiquetas obtenidas por el algoritmo, e y_i son las etiquetas conocidas. Tomando en cuenta la consistencia con la geometría de los datos, a partir de las suposiciones de suavidad y variedad, se propone un término de penalización de la siguiente forma:

$$\begin{aligned} \frac{1}{2} \sum_{i,j=1}^{n} W_{ij} (\hat{y_i} - y_i)^2 &= \frac{1}{2} \left(2 \sum_{i=1}^{n} {\hat{y_i}}^2 \sum_{j=1}^{n} W_{ij} - 2 \sum_{i=1}^{n} W_{ij} \hat{y_i} \hat{y_j} \right) \\ &= \hat{Y}^T \cdot (\mathbf{D} - \mathbf{W}) \cdot \hat{Y} \\ &= \hat{Y}^T \cdot \mathbf{L} \cdot \hat{Y}. \end{aligned} \tag{2.8}$$

Donde $\mathbf{L}$ es el grafo laplaciano no normalizado (ver sección 2.3.1). La ecuación 2.8 tiene

como propiedad principal penalizar cambios rápidos en $\hat{Y}$ entre puntos que son cercanos dados por la matriz de similaridad $\mathbf{W}$.

Para considerar soluciones degeneradas, se toma un criterio intermedio entre las ecuaciones 2.7 y 2.8 [Belkin et al., 2004] sugiere una ecuación de costo de la siguiente forma:

$$C(\hat{Y}) = ||\hat{Y}_l - Y_l||^2 + \mu \hat{Y}^T \mathbf{L}\hat{Y} + \mu\epsilon||\hat{Y}||^2. \tag{2.9}$$

Donde el parámetro μ se define en la ecuación 2.10:

$$\mu = \frac{\alpha}{1-\alpha} \in (0, \infty) \tag{2.10}$$

El valor de ϵ es positivo mayor a cero. Con el objetivo de minimizar el criterio del costo cuadrático, se obtiene la primer derivada de la ecuación 2.9. Aquí, se toma en cuenta que el término $||\hat{Y}_l - Y_l||^2$ puede ser reescrito como $||(\boldsymbol{S})\hat{Y} - (\boldsymbol{S})Y||^2$, donde la matriz $\boldsymbol{S}$ de $n \times n$ es una matriz con 1's en la diagonal, pero únicamente en donde las filas corresponden con los valores que tienen etiqueta.

$$\begin{aligned} \frac{1}{2}\frac{\partial C(\hat{Y})}{\partial \hat{Y}} &= \boldsymbol{S}(\hat{Y} - Y) + \mu\mathbf{L}\hat{Y} + \mu\epsilon\hat{Y} \\ &= (\mathbf{S} + \mu\mathbf{L} + \mu\epsilon\boldsymbol{I})\hat{Y} - \boldsymbol{S}Y \end{aligned} \tag{2.11}$$

La segunda derivada es:

$$\frac{1}{2}\frac{\partial^2 C(\hat{Y})}{\partial \hat{Y}\partial \hat{Y}^T} = \boldsymbol{S} + \mu L + \mu\epsilon\boldsymbol{I} \tag{2.12}$$

la cual es una matriz definida positiva cuando $\epsilon > 0$. El costo se minimiza cuando la primera derivada es 0, por lo que se obtiene:

$$\hat{Y} = (\boldsymbol{S} + \mu L + \mu\epsilon\boldsymbol{I})^{-1}\boldsymbol{S}Y \tag{2.13}$$

Esto tiene como resultado que las etiquetas nuevas pueden obtenerse utilizando operaciones sobre matrices. También es importante destacar que la matriz invertida no depende de las etiquetas originales, sino del grafo definido por el laplaciano L.

2.4. Cómputo de Alto Rendimiento

Uno de los mayores auges de desarrollo del procesamiento de datos usando equipos de cómputo fue durante la Segunda Guerra Mundial. A partir de ese momento se empezaron a utilizar computadoras con diferentes tecnologías debido a las necesidades de procesamiento que fueron surgiendo. Después de la guerra, la industria y el gobierno en convenio con diferentes instituciones académicas alrededor de todo el mundo siguieron haciendo uso y promoviendo el desarrollo de estas tecnologías, debido principalmente al incremento de sus necesidades de procesamiento para la solución de problemas complejos [Pacheco, 2011, Parhami, 2005].

Algunos avances importantes, necesidades y limitaciones de este tipo de cómputo son [Patterson and Hennessy, 2006]:

- De 1986 a 2002, el rendimiento de los procesadores se incrementó en promedio un 50 % al año debido principalmente a mejoras en la tecnología de construcción de los dispositivos físicos y a las mejoras arquitectónicas en los equipos [Parhami, 2005] (uso de diferentes niveles de memoria, segmentación, múltiple emisión de instrucciones y extensiones ISA[1], como las más importantes).

- El constante incremento de rendimiento de los procesadores hacía que los desarrolladores de software únicamente tuviesen que preocuparse por esperar una versión de procesadores con mayor velocidad. En los últimos años ha habido un auge importante hacia el cómputo paralelo, las tendencias van más hacia procesadores múltiples que hacia mejorar las arquitecturas de los procesadores monolíticos.

- La mayor parte de los programas han sido escritos para ejecutarse de manera secuencial, esto abre una cantidad importante de oportunidades en investigación y desarrollo de aplicaciones paralelas y distribuidas.

- De un programa paralelo, en general no se puede esperar un rendimiento proporcional al número de unidades de procesamiento (Ley de Amdahl [Pacheco, 2011, Parhami, 2005]).

Necesidad de Cómputo de Alto Rendimiento

Las necesidades de cómputo de alto rendimiento (*High Performance Computing*, HPC, por sus siglas en inglés) se hacen notorias debido al incremento de operaciones que deben ser llevadas a cabo para resolver problemas modernos. Además de ocupar mayores requerimientos de almacenamiento de datos tanto de manera permanente como durante el tiempo en que se procesan para generar información.

[1] Conjunto de instrucciones de la Arquitectura, ISA por sus siglas en inglés

Dichas necesidades de cómputo se ejemplifican mejor a través de aplicaciones en distintas áreas de conocimiento como son[2]:

- **Modelado de eventos climáticos** para entender mejor y poder predecir eventos, en las simulaciones se estudia la interacción entre la atmósfera, los océanos, la tierra, corrientes de aire, el hielo de los polos, entre otros.
- **Modelado de procesos biológicos, físicos y químicos**:
  - En el *Oak Ridge National Laboratory* se simula a 20 petaflops (20×10^{15} operaciones de punto flotante por segundo) el proceso de combustión de moléculas de combustibles complejos en condiciones muy turbulentas [Chen, 2011].
  - El proyecto *Folding@Home* usa un *grid* de más de 160 mil computadoras para simular el proceso de plegamiento proteico, útil para el estudio de configuraciones de moléculas que tienen que ver con enfermedades como el *Parkinson* o *Huntington*.
- **Búsqueda de nuevas medicinas**: el proyecto de investigación sobre el estudio del genoma trata de identificar secuencias y mutaciones críticas en el desarrollo de cáncer en los humanos [Uberbacher, 1997].
- **Análisis de datos**: para el procesamiento de información astronómica, el Gran Colisionador de Hadrones, imágenes médicas, buscadores Web, entre muchas otras. Una referencia que cabe destacar en este apartado es el proyecto *SETI@Home* desarrollado por el *Space Sciences Laboratory*, de la Universidad de California en Berkeley (U.S.) que usa el tiempo de ejecución libre en equipos de cómputo de voluntarios para procesar de manera distribuida señales de radio buscando evidencia de inteligencia extraterrestre.
- **Investigación sobre defensa y armamento**: simulación de armas nucleares, cálculo de trayectorias para el lanzamiento de misiles, entre otros.
- **Visualización de datos**: en áreas como ciencias de los materiales, análisis de reservas de agua, simulación de choques de autos y nanotecnología se requiere visualizar y procesar gran cantidad de información para determinar resultados de experimentos [Toedte and Wooten, 1997].

2.4.1. La taxonomía de Flynn

Michael J. Flynn propuso en 1972 una taxonomía para clasificar los diferentes sistemas de computadoras [Patterson and Hennessy, 2008, Parhami, 2005]. La taxonomía se basa en el diferente número de instrucciones concurrentes y los flujos de datos que deberán ser tratados durante su ejecución como se muestra en la figura 2.5, cuyos elementos son:

[2]TOP500 Supercomputer Sites `http://www.top500.org` (último acceso 15/jun/2018)

- Una instrucción, un flujo de datos (**SISD**): equipos con 1 procesador que ejecutan un único flujo de instrucciones.
- Múltiples instrucciones, un flujo de datos (**MISD**): no se ha documentado la construcción de sistemas de este tipo.
- Una instrucción, múltiples flujos de datos (**SIMD**): en ellas se tiene una sola instrucción que reciben todos los procesadores al mismo tiempo y la ejecutan sobre un vector de datos permitiendo hacer paralelismo.
- Múltiples instrucciones, múltiples flujos de datos (**MIMD**): varios procesadores autónomos que pueden ejecutar simultáneamente diferentes instrucciones sobre diferentes flujos de datos.

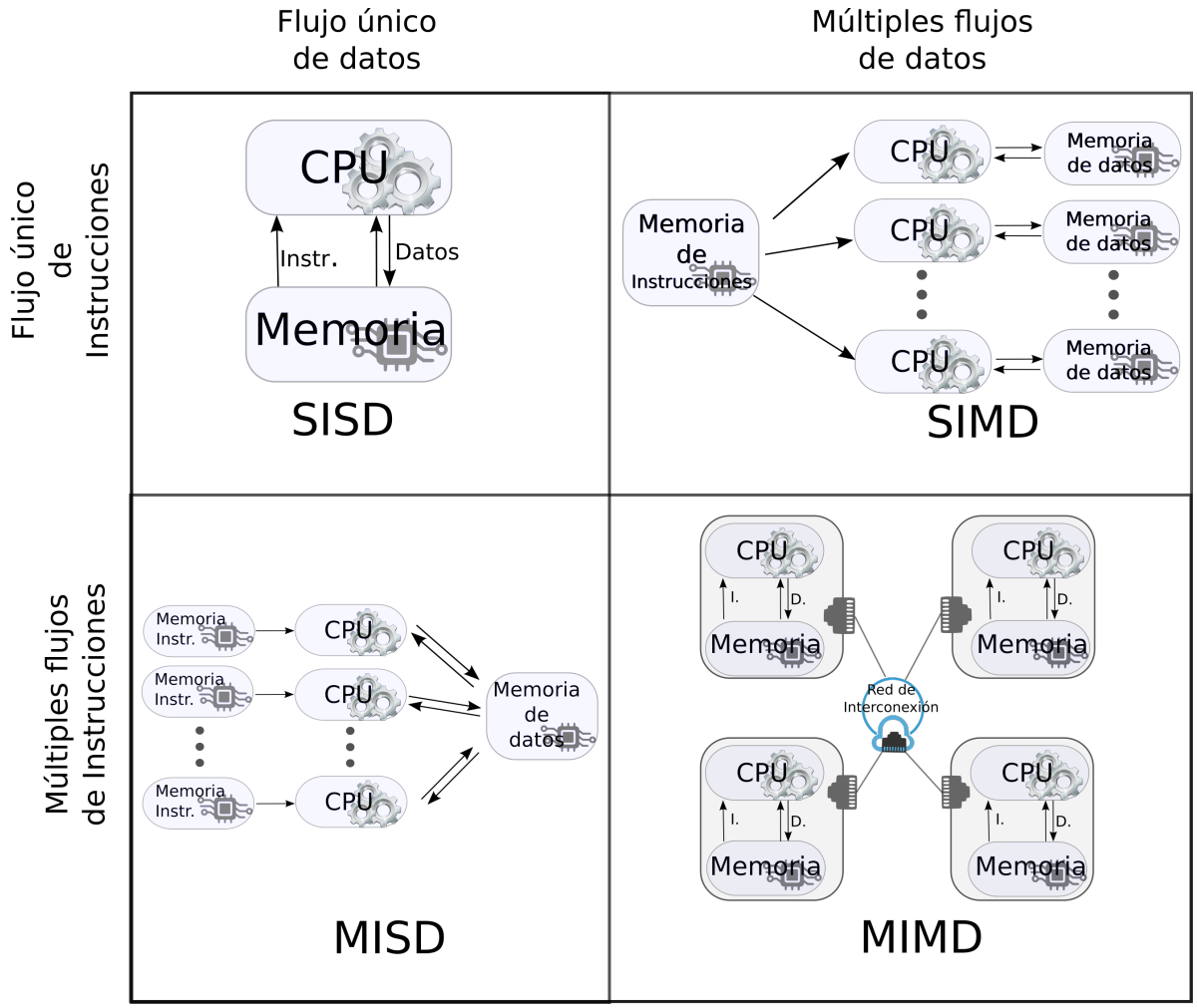


Figura 2.5: Taxonomía de Flynn [Patterson and Hennessy, 2006].

El clúster usado para la ejecución de los algoritmos se encuentra en la clasificación de Múltiples instrucciones y Múltiples flujos de datos (MIMD). Ya que cada una de las

unidades de procesamiento se puede comunicar con el resto de unidades usando una red de interconexión de tal forma que puede estar ejecutando diferentes instrucciones en un instante de tiempo determinado.

En la clasificación MIMD existen dos subclasificaciones: el modelo de memoria compartida (*shared memory*) y de memoria distribuida (*distributed memory*). En el clúster cada equipo de cómputo (nodo) usa el modelo de memoria compartida ya que las unidades de procesamiento (cores) usan la memoria RAM para compartir datos. La memoria distribuida se implementa a través de una red de interconexión que comunica a los diferentes nodos.

2.4.2. Conceptos importantes de programación paralela y distribuida

La programación paralela es un modelo para escribir programas que se ejecuten de manera concurrente sobre equipos con varios procesadores. La ejecución distribuida ocurre cuando un programa se ejecuta en varios equipos de cómputo que se comunican a través de una red de interconexión [Patterson and Hennessy, 2006].

Usualmente, se desea que un programa paralelo pueda ser general y ejecutarse sobre una variedad grande de equipos de cómputo de manera eficiente. La implementación de programas paralelos puede llevarse a cabo usando bibliotecas que pueden invocarse desde lenguajes de programación secuenciales, extensiones a lenguajes o modelos nuevos de ejecución.

El modelo de programación paralela usando memoria compartida puede implementarse usando la biblioteca OpenMP. Esta biblioteca contiene directivas que permiten la ejecución de regiones de código en paralelo.

Debido a que el presente trabajo utiliza programación paralela, a continuación se describen algunos términos importantes [Culler et al., 1997]:

- **Tarea**: es la unidad de trabajo al momento de ejecutarse un programa paralelo. Cada tarea se ejecuta secuencialmente pero existe concurrencia con respecto a otras tareas. Las tareas pueden clasificarse según su granularidad en tareas sencillas y tareas complejas.
- **Proceso**: Son unidades abstractas que realizan las tareas asignadas a los procesadores. Los procesos se comunican y sincronizan para realizar las tareas.
- **Procesador**: es una unidad de procesamiento que ejecuta un proceso.

Para realizar el proceso de paralelización como se muestra en la figura 2.6 se deben llevar a cabo 4 tareas principales [Culler et al., 1997]:

- **Descomposición del problema** en tareas pequeñas. El ideal es que las tareas sean lo más pequeñas posibles representando cantidades similares de trabajo, y evitando redundancia de proceso de cómputo y/o almacenamiento. La identificación de estas tareas debe poderse escalar con el tamaño del problema. Los límites de esta descomposición están acotados por la Ley de Amdahl.
- **Mecanismo de asignación** por el cual las tareas se deben distribuir entre los procesos. Se deben considerar aspectos como equilibrio de la carga y si es necesario algún algoritmo de planificación buscando evitar en la medida de lo posible las comunicaciones y los puntos de detención (*deadlocks*) [Coulouris et al., 2012].
- **Orquestación** para estructurar la comunicación entre los procesos y llevar a cabo la sincronización a través de algunos patrones de comunicación: local/global, estático/dinámico, síncrono/asíncrono o estructurado/no estructurado.
- **Mapeo** es la asignación de procesos o hilos de ejecución a las unidades de procesamiento buscando minimizar el tiempo de ejecución.

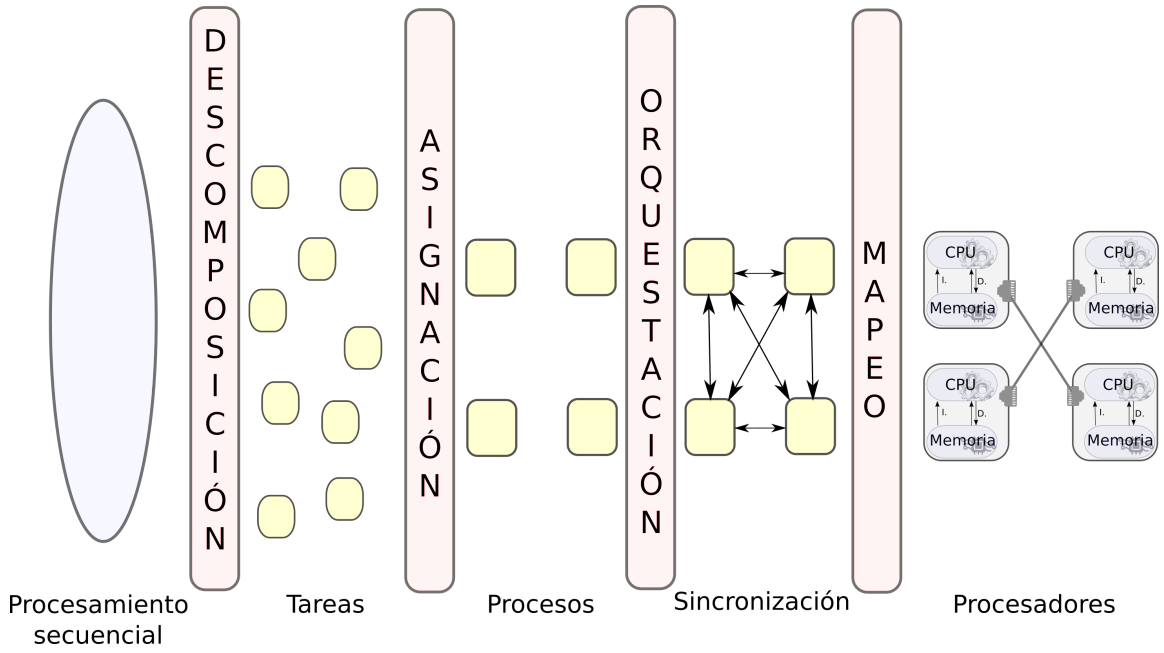


Figura 2.6: Proceso de paralelización [Culler et al., 1997].

Al momento de buscar tareas que se pueden llevar a cabo de manera concurrente es común detectar paralelismo en datos y paralelismo de tareas (o funcional) [Quinn, 2003]:

- El **paralelismo en datos** ocurre cuando existen tareas que aplican la misma operación a diferentes elementos de un conjunto de datos. El ejemplo de la figura 2.7 muestra que la operación de suma puede ser realizada de manera independiente sobre los 100 elementos de los vectores y estas operaciones pueden ejecutarse de manera simultánea por diferentes unidades de procesamiento.

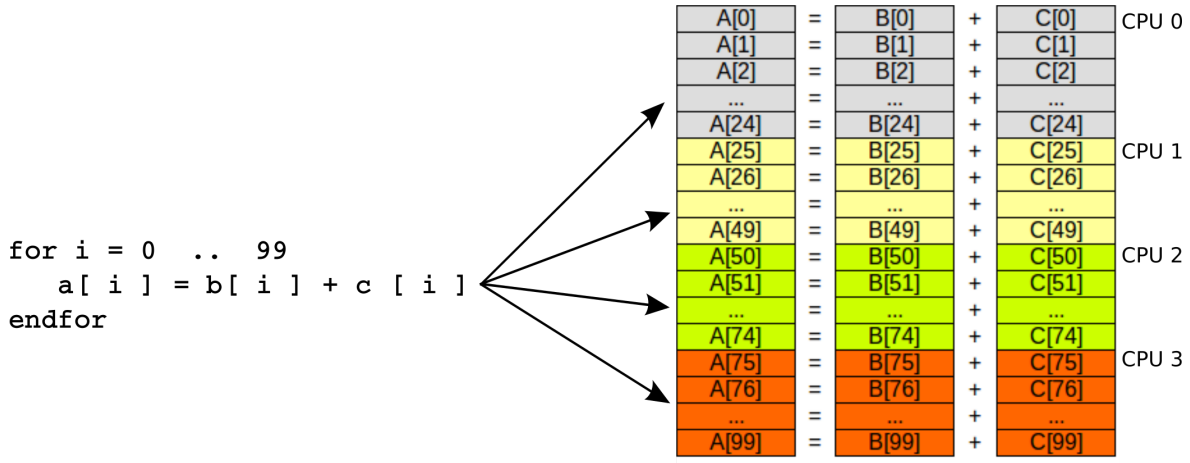


Figura 2.7: Ejemplo del proceso de paralelización a nivel de datos. Se aprecia que el ciclo de asignaciones es dividido en cuatro bloques independientes, donde cada uno es llevado a cabo por una unidad de procesamiento.

- El **paralelismo de tareas** o funcional existe cuando se pueden ejecutar tareas de manera independiente sobre diferentes conjuntos de datos. En el ejemplo de la figura 2.8, las asignaciones hacia **a** y **b**, así como **m** y **q** pueden ejecutarse de manera concurrente, sin embargo, **m** y **q** deben esperar a las asignaciones de **a** y **b** para poderse ejecutar, de la misma manera que h puede ser ejecutado hasta que **m** y **q** obtengan su valor correspondiente.

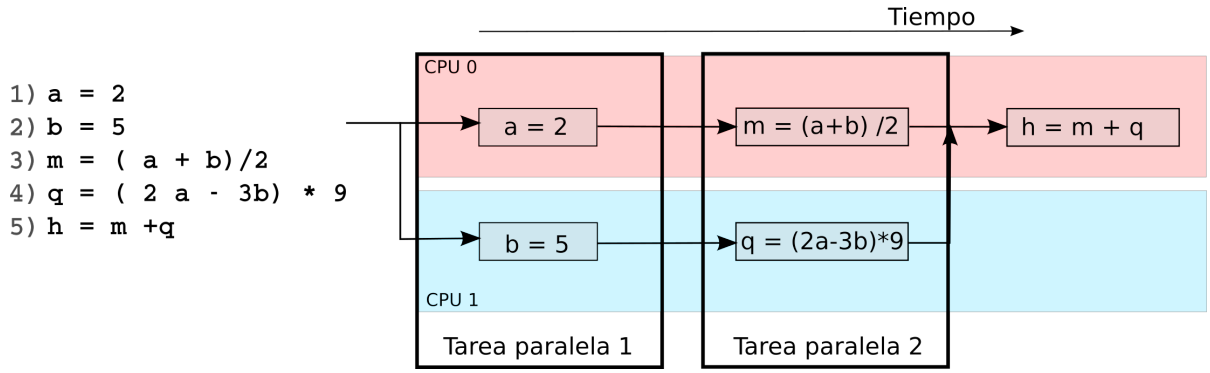


Figura 2.8: Ejemplo del proceso de paralelización a nivel de tareas. Las primeras dos instrucciones pueden ejecutarse en paralelo. Al terminarse, pueden ejecutarse las siguientes dos también en paralelo, pero la quinta instrucción debe esperar hasta que terminen la tercera y cuarta instrucción.

2.4.3. Modelos de programación paralela y distribuida

Modelo de programación usando OpenMP

OpenMP[3] [Pacheco, 2011, Chandra et al., 2001] es una interfaz de programación que permite la ejecución de múltiples procesos usando la arquitectura de memoria compartida a nivel de cada nodo o computadora.

OpenMP permite ejecutar concurrentemente procesos en programas escritos en C, C++ y Fortran, éste usa el modelo **fork-join**, que consiste en tener un nodo maestro que va ejecutando el flujo principal del programa. Para utilizar instrucciones que pueden ser ejecutadas en paralelo se usa una directiva (`#pragma`) que crea hilos de ejecución (*fork*). Al terminar la ejecución de los procesos paralelos se devuelve el control al nodo maestro (*join*). La figura 2.9 ilustra gráficamente este modelo.

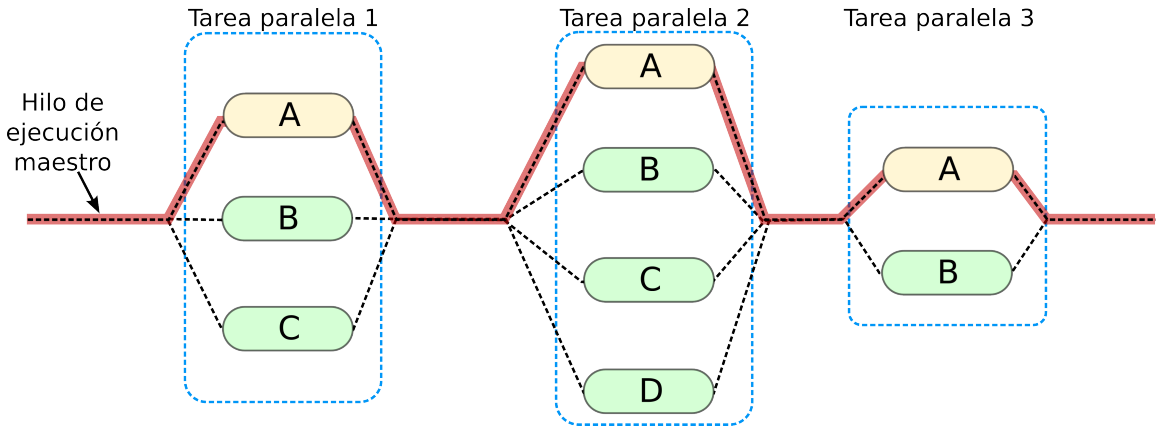


Figura 2.9: Modelo *Fork-Join* [Barney, 2013].

El acceso a memoria con este paradigma de programación es conocido como acceso uniforme a memoria. La figura 2.10 muestra la conexión a través del bus del sistema de **n** procesadores, cada uno con memoria caché a la que puede acceder de manera individual (L1, L2, e incluso L3), y de manera conjunta a la memoria principal del sistema. Un ejemplo de configuración se muestra en la figura 2.11.

Modelo de programación usando la Interfaz de Paso de Mensajes

La *Interfaz de Paso de Mensajes* (MPI, por sus siglas en inglés) [Quinn, 2003] es la primera biblioteca de paso de mensajes que ofreció portabilidad, estandarización y una implementación eficiente para el desarrollo de aplicaciones paralelas y distribuidas. El estándar define la sintaxis y semántica de funciones contenidas en una biblioteca diseñada para ser usada para la creación de programas que permitan usar múltiples procesadores

[3]The OpenMP API specification for parallel programming, http://openmp.org/wp/ (último acceso 15/jun/2018)

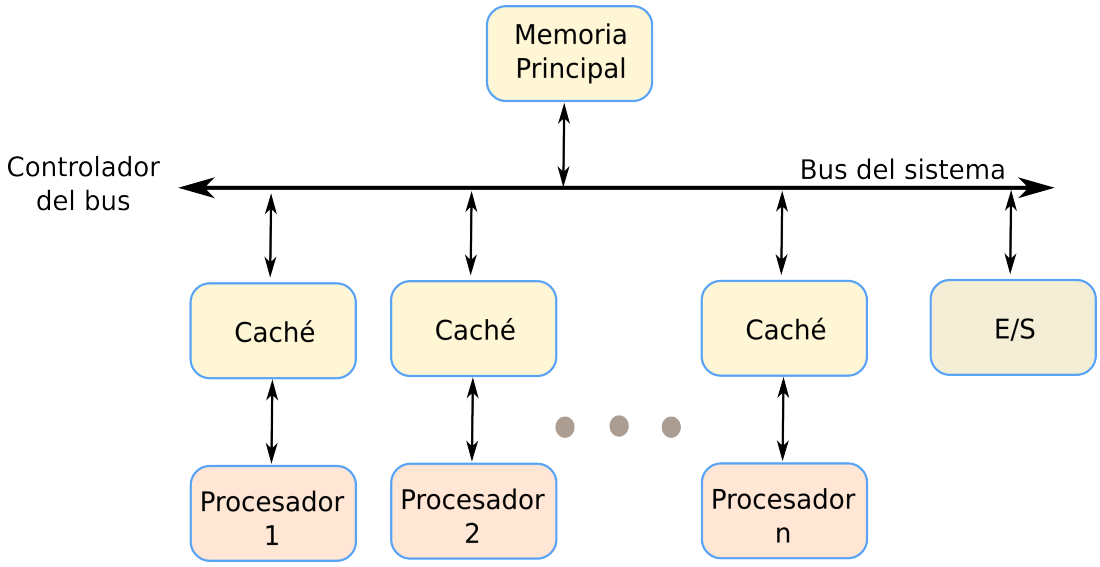


Figura 2.10: Acceso a memoria uniforme [Barney, 2013].

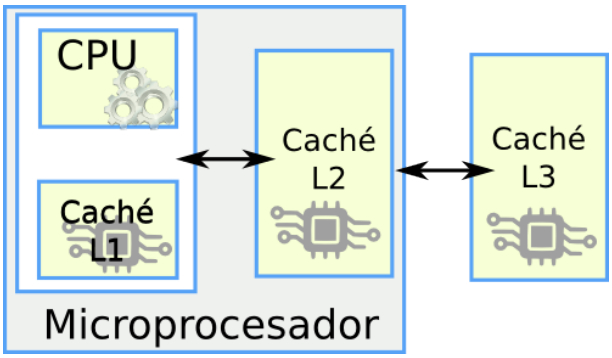


Figura 2.11: Acceso a memoria de cada procesador [Parhami, 2005].

que se comunican ya sea a través de una red de interconexión o usando memoria compartida. Un esquema que combina memoria compartida (OpenMP) y memoria distribuida (MPI) se muestra en la figura 2.12. En esta figura, hay 4 nodos cada uno con 4 unidades de procesamiento y su respectiva memoria caché que acceden a través del bus del sistema y pueden direccionar en la memoria principal (memoria compartida). Los 4 nodos se comunican a través de una red de interconexión.

La estandarización en MPI asegura que las llamadas hechas en tiempo de ejecución por algún equipo se comportan de igual manera en otras máquinas, independientemente de la implementación usada. La portabilidad permite la implementación de programas en diferentes plataformas como Linux, Solaris, UNIX y Windows [Quinn, 2003].

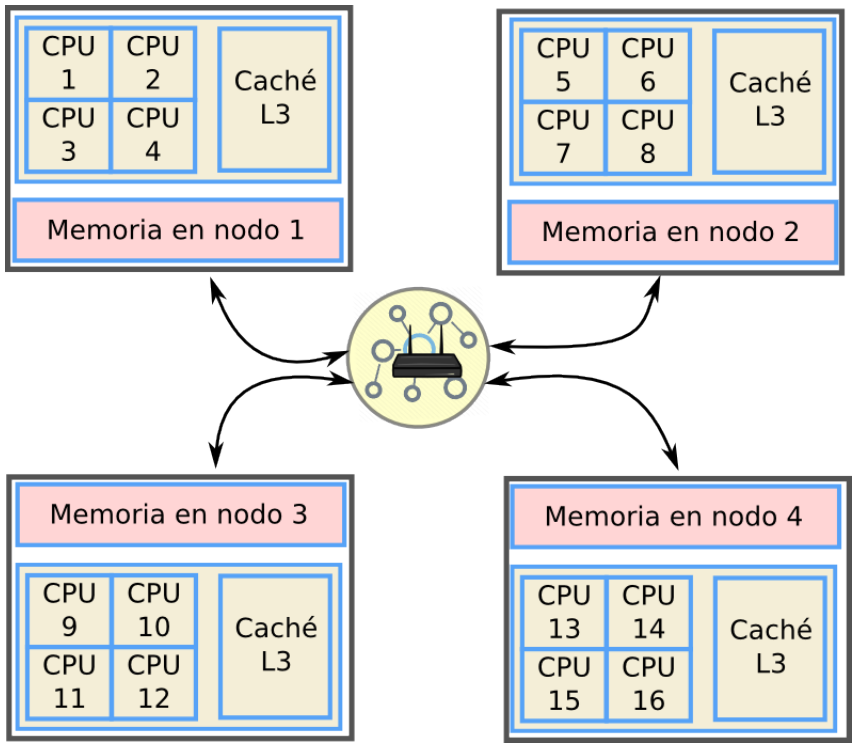


Figura 2.12: Arquitectura que combina memoria compartida y distribuida [Barney, 2013].

Capítulo 3

Desarrollo del proyecto

En este capítulo se describen los requerimientos y características del hardware y software usados para las implementaciones secuencial, paralela y distribuida de los módulos de la biblioteca desarrollada. También se describe un esquema general de los módulos que la componen y los detalles de las funciones más importantes.

3.1. Requerimientos

Requerimientos de hardware y software para la implementación secuencial

La implementación secuencial puede ser utilizada prácticamente en cualquier equipo de cómputo moderno. En la presente investigación se hace uso del modelo SISD de la taxonomía de Flynn (ver sección 2.4.1). Las características mínimas de los equipos con los que se hicieron las pruebas son:

1. Procesador de 1.0 GHz
2. 1 GB de memoria RAM
3. Al menos 1 GB para almacenar los conjuntos de datos y los resultados generados.

La implementación fue probada con el compilador g++ versión 4.8.1 con las distribuciones de Linux de Ubuntu 14.04 y 14.10.

Para la edición de los archivos de las bibliotecas se requiere un procesador de textos básico o algún IDE de desarrollo que generalmente requieren archivos de configuración para automatizar tareas como la compilación, ejecución y depuración.

Requerimientos de hardware y software para la implementación paralela

La implementación paralela cae en el modelo MIMD con memoria compartida (ver sección 2.4.1). Los requerimientos mínimos en hardware son los mismos que para la implementación secuencial excepto por el procesador, que debe ser capaz de ejecutar múltiples hilos de ejecución. Esto es, procesadores con dos o más núcleos de procesamiento.

En el caso del software, se debe instalar una versión de OpenMP, la instalación de OpenMP detecta de manera automática las unidades de procesamiento del sistema y configura las variables de entorno. Sin embargo, de ser necesario puede ser indicado un número de hilos de ejecución directamente.

Requerimientos de hardware y software para la implementación distribuida

La implementación distribuida cae en el modelo MIMD con memoria distribuida, tal como fue descrita en la sección 2.4.1. Para la ejecución de manera distribuida, se requieren al menos dos nodos comunicados por una red de interconexión. La velocidad de la red de interconexión es un factor muy importante para poder aumentar la velocidad de procesamiento, ya que las comunicaciones establecen un cuello de botella durante la ejecución.

Para la implementación distribuida, se debe instalar el paquete de MPI desde el administrador de paquetes de la distribución actual o descargar alguna implementación como MPICH[1]. Para ejecutar aplicaciones distribuidas con MPI, se debe crear un archivo de configuración $mpi\_hostfile$ que contiene las ubicaciones de nodos sobre la red y cantidad de núcleos disponibles para ejecución.

Es posible usar una implementación distribuida sin paralelismo, es decir, en cada nodo puede ejecutarse una tarea de manera secuencial. Así mismo, pueden ejecutarse varias tareas en un solo nodo (generalmente, con el objetivo de hacer pruebas). En este último caso, la biblioteca de MPI simula la transmisión de los mensajes entre los procesos usando memoria compartida. Si se requiere una ejecución paralela y distribuida, en cada nodo debe haber más de una unidad de procesamiento para la ejecución paralela y una red de interconexión para la ejecución distribuida. Sin embargo, las bibliotecas de MPI y OpenMP permiten simular la ejecución en un solo nodo en una única unidad de procesamiento. Esto último generalmente se hace con fines de pruebas.

Ejecución de las aplicaciones en el clúster de cómputo

Para la la implementación distribuida de los algoritmos seleccionados se instalaron dos clústers Beowulf según se muestra en el anexo B. El primer clúster está compuesto por 2 servidores Dell PowerEdge T630, cada uno con dos procesadores Intel© Xeon© CPU

[1] http://www.mpich.org/

E5-2630 v3 @ 2.40GHz con 8 cores, cada core con 2 hilos, para dar un total de capacidad de ejecución de 32 hilos. Cada procesador cuenta con 20MB en caché. Cada servidor cuenta con 16 GB en memoria RAM y 120GB de espacio en disco duro de estado sólido. Los servidores (nodos) están interconectados a través de una red Ethernet, físicamente implementado con un switch gigabit y dos puertos Gigabit Ethernet, como se muestra en la figura 3.1.

Figura 3.1: Elementos del clúster Beowulf usado para las pruebas.

El segundo clúster tiene 3 servidores Dell PowerEdge T310. Cada servidor cuenta con un procesador Intel(R) Xeon(R) i-7 X3440 @ 2.53GHz con 8 hilos de ejecución, 32 GB en memoria RAM y disco duro de 1TB. La red de interconexión usa un Switch HP V1405-8G 10/100/1000Mbps.

Para ejecutar un programa desde el nodo maestro se usa un archivo de configuración que indica los recursos a usar en el clúster. Un requerimiento para cada nodo es tener la misma ruta de acceso a los archivos de entrada y salida (para reportes o almacenar resultados) con respecto al ejecutable de manera local (o usar un sistema de archivos compartido) y la misma clave de acceso (*password*) que puede ser administrada por el comando *ssh-agent*. La distribución del procesamiento entre los nodos se debe realizar por código de manera explícita. Para configurar los recursos de procesamiento (cuántos nodos y sus rutas en la red) se usa el archivo de configuración *mpi_hostfile*, este archivo es enviado por línea de comandos al programa *mpirun* al iniciar la ejecución distribuida. Mayores detalles de este proceso se encuentran en el anexo B.

3.2. Módulos del proyecto

La ejecución de los algoritmos implementados requiere como entrada los conjuntos de datos, que son leídos desde archivos de texto. Estos conjuntos son tomados de bases de datos públicas como se describe en el siguiente capítulo, en la sección 4.1. Una vez en

la memoria, se utiliza una función que recibe el conjunto de datos y separa de manera aleatoria y equilibrada una cantidad definida de elementos de cada clase que serán tomados como ejemplos con y sin etiqueta. La ejecución del algoritmo tiene como objetivo obtener la etiqueta correcta eliminada en el paso anterior. Finalmente, se calcula una matriz de confusión con lo que se obtiene el resultado de rendimiento de clasificación de un algoritmo. Por otro lado, también se calcula el tiempo de ejecución tomando la diferencia entre el tiempo de inicio y final de ejecución. Este tiempo es usado para realizar comparaciones de diferentes configuraciones de ejecución. El proceso anteriormente descrito puede apreciarse en la figura 3.2.

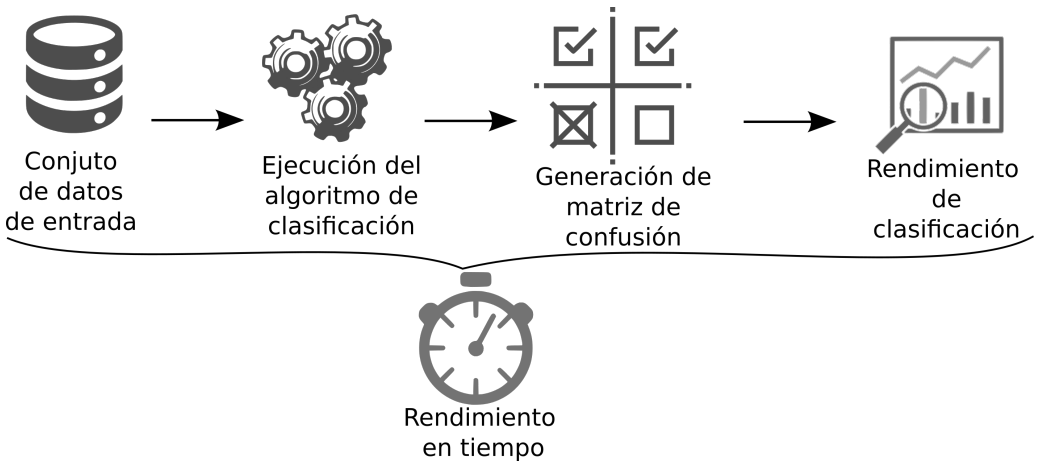


Figura 3.2: Proceso general de ejecución de los algoritmos.

3.2.1. Módulos del Algoritmo de propagación de etiquetas

El algoritmo de propagación de etiquetas requiere el análisis de cada conjunto de datos para extraer el valor del parámetro σ. Para lograr esto, el algoritmo de Kruskal es modificado para mantener una lista de vértices en la estructura *union-find* (biblioteca $Kruskal$) que mantiene identificados los nodos del grafo que tienen etiquetas. El algoritmo mantiene los componentes que van formando el árbol de expansión mínima, y se detiene cuando se intentan conectar dos componentes que contienen ejemplos con etiquetas conocidas diferentes. En ese momento, el valor de σ es calculado como se describió en la sección 2.3.2. La implementación hace uso de los módulos mostrados en la figura 3.3. El archivo `kruskal.cpp` contiene la función `main` que hace las llamadas a las funciones de manera correspondiente.

Al inicio de la ejecución, el conjunto de datos es leído desde un archivo de texto para crear una matriz que representa un grafo completamente conectado donde cada nodo es un ejemplo y calcula la distancia entre ellos. El módulo `generarEtiquetas` permite marcar a una cantidad determinada de ejemplos con etiqueta y al resto la deja sin marca. Posteriormente, se invoca a la función *kruskal* que hace el cálculo del valor de d_0 para obtener así el valor de σ.

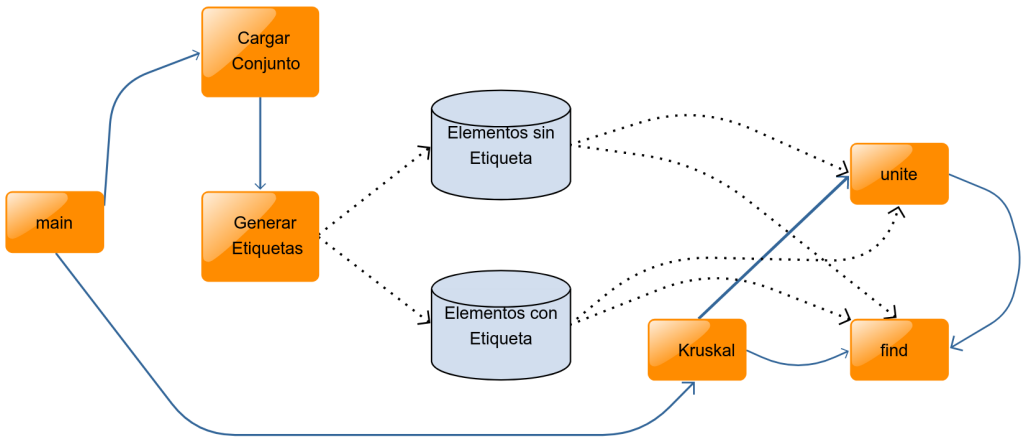


Figura 3.3: Proceso de obtención del valor σ usando el algoritmo de Kruskal modificado.

En la figura 3.3 se muestra la interacción de la función *main* con la función *kruskal*, así como las llamadas desde *kruskal* a las funciones *unite* y *find* que implementan la estructura *union-find*. También muestra la separación de los elementos con etiqueta y los que no la tienen, y el acceso a estos desde los módulos. El cuadro 3.1 muestra una descripción de los módulos más importantes de la implementación.

La figura 3.4a muestra las dependencias con respecto a otros archivos para el algoritmo secuencial de propagación de etiquetas. La figura 3.4b muestra la implementación paralela en el archivo `vectores.h` hace uso de la biblioteca `omp.h`. Por otro lado, la implementación distribuida está programada en la función `main` en el archivo `labelpropagation.cpp`, ésta hace uso de la biblioteca `mpi.h`. Las implementaciones son muy parecidas salvo por el uso de ambas bibliotecas en lugares muy particulares del código. Las dependencias de la

`int`	`find ( ··· )` Operación *find* de la estructura *union-find*. Regresa el valor de la etiqueta asignada al vértice del grafo.
`int`	`unite ( ··· )` Operación unir de la estructura *union-find*. Identifica el punto en que se detendrá el algoritmo al detectar cuando se intenta unir a dos elementos de distintos componentes ya etiquetados.
`double`	`kruskal ( ··· )` Algoritmo de Kruskal modificado, regresa el valor de d_0.
`void`	`generarEtiquetas ( ··· )` Genera un número de elementos etiquetados para el conjunto de datos seleccionado.

Cuadro 3.1: Funciones principales del módulo para calcular el valor d_0 usando la versión modificada del algoritmo de Kruskal.

implementación paralela-distribuida pueden verse en la figura 3.4b, como puede apreciarse al hacer uso de ambas bibliotecas, realiza cómputo en paralelo en cada nodo y distribuye la carga de procesamiento usando el paso de mensajes. En el anexo A.1 se explican los detalles de implementación de las tres versiones.

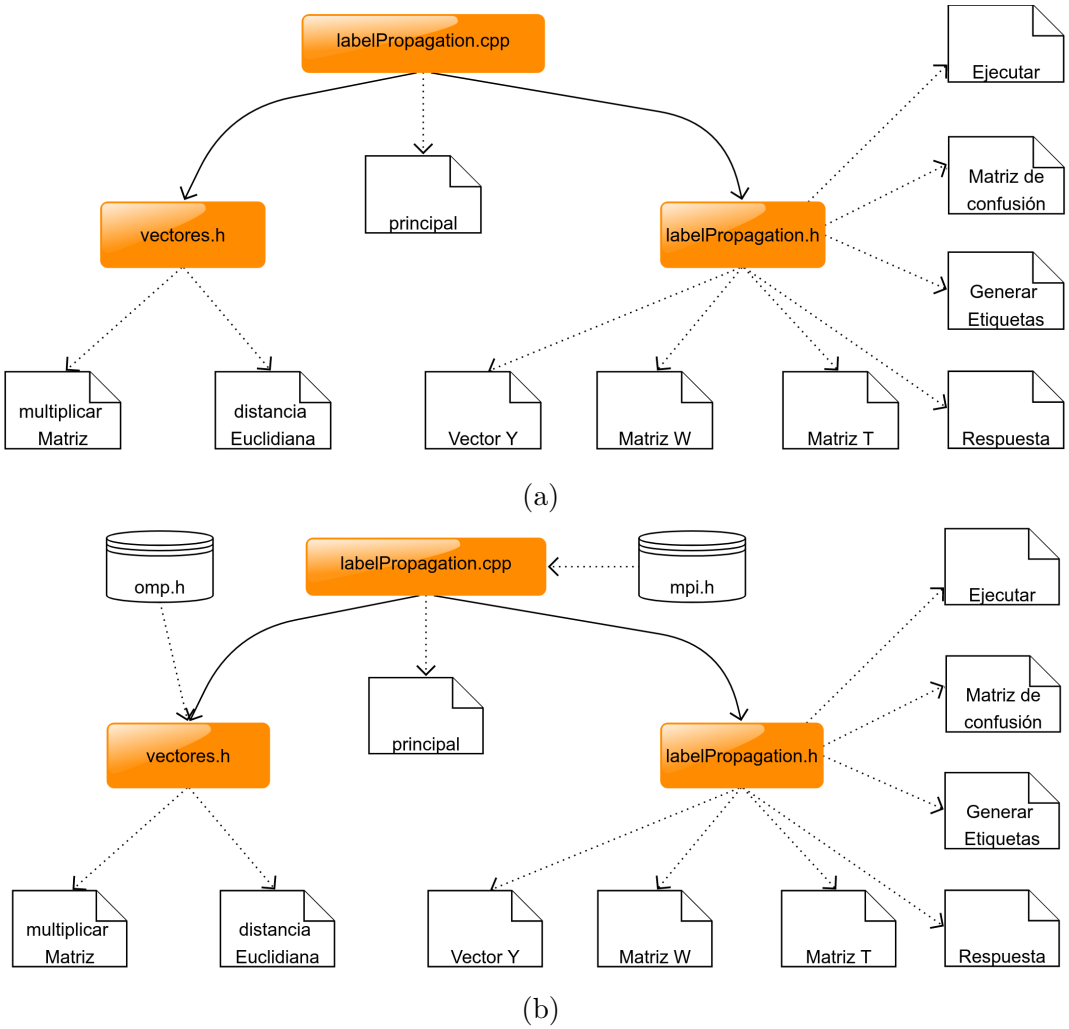


Figura 3.4: Dependencias de los módulos para la implementación (a) secuencial y (b) paralela-distribuida del algoritmo LP.

El cuadro 3.2 muestra las funciones más importantes y la descripción de cada una de ellas. La función `ejecutar` realiza el procesamiento de una iteración del algoritmo. Recibe como parámetros el porcentaje de elementos que deben conservar su etiqueta y el valor de σ calculado previamente.

`void`	`vectorY ( ··· )` Inicialización del vector Y para controlar las respuestas
`void`	`matrizW ( ··· )` Genera la matriz de pesos W, usando como parámetro el valor de σ.
`void`	`matrizT ( ··· )` Matriz que contiene la probabilidad de transición de ir de un nodo i a un nodo j.
`void`	`agregaConfusion ( ··· )` Se agregan valores a una matriz de confusión para poder de calcular el rendimiento del clasificador.
`int`	`respuesta ( ··· )` Se evalua la respuesta del clasificador usando el vector Y que contiene las etiquetas detectadas por el algoritmo y las etiquetas reales del conjunto de prueba.
`void`	`generarEtiquetas ( ··· )` Recibe la cantidad de elementos de cada clase que deben ser etiquetados de manera aleatoria, para el resto de elementos del conjunto de datos se elimina la etiqueta.
`int`	`ejecutar ( ··· )` Ejecución del algoritmo de propagación de etiquetas, recibe como parámetros el porcentaje y el valor de sigma calculados previamente con el algoritmo de Kruskal modificado.

Cuadro 3.2: Funciones principales para el algoritmo de propagación de etiquetas

3.2.2. Módulos del Algoritmo de propagación de etiquetas usando el criterio del costo cuadrático

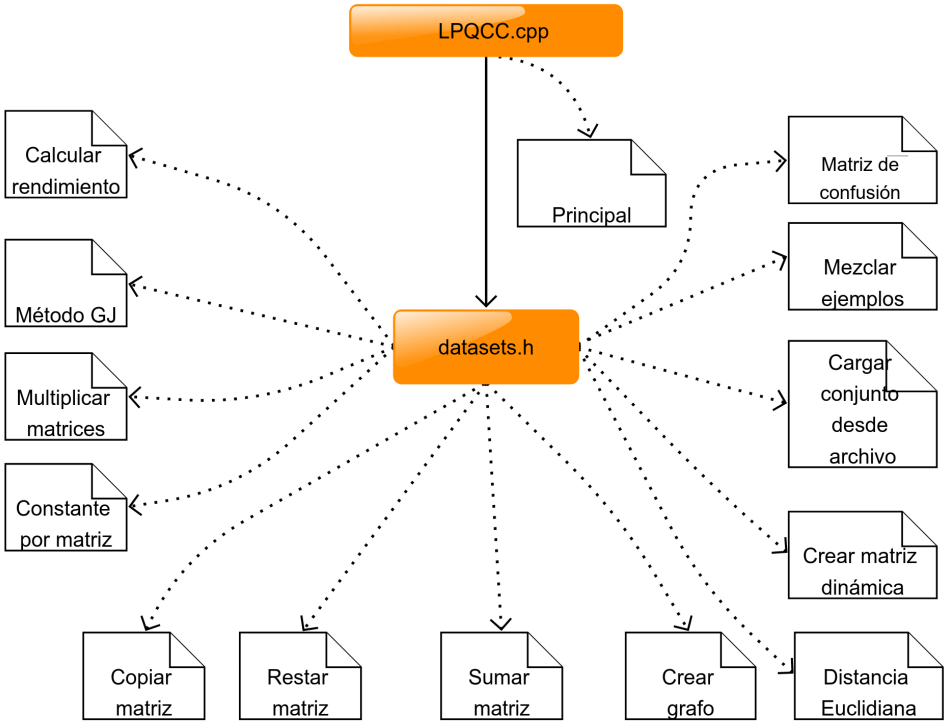
La figura 3.5 muestra las dependencias con respecto a las funciones usadas para las implementaciones secuencial y paralela-distribuida. Los módulos de manejo de vectores están localizados en el archivo *datasets.h* y su correspondiente implementación en *datasets.cpp*. Debido a que la paralelización de operaciones sobre matrices se hizo a nivel unidades de procesamiento en cada nodo (servidor), las operaciones se realizan en *datasets.cpp*. La versión secuencial y paralela tienen la misma estructura, excepto por la llamada a `omp.h` que permite ejecutar algunas partes del código de forma paralela. Por otro lado, la distribución de la carga de procesamiento entre los nodos a través de la red se hizo usando las funciones de paso de mensajes de *mpi.h* desde la función *main* en el archivo `LPQCC.cpp` (ver Figura 3.5b). En el anexo A.2 se describen los detalles de implementación de las tres versiones.

El cuadro 3.3 muestra las funciones que contienen la lógica más importante del algoritmo. La función `loadDataSetFromFile` se encarga de cargar los datos desde un archivo de texto. `shufflingExamples` hace la selección aleatoria de los ejemplos que conservan y los que no su etiqueta. La función `createGraphW` crea la matriz de pesos W usando la función `euclideanDistance` que es usada en las siguientes etapas del algoritmo. Las funciones `addMatrices, substractMatrices, copyMatrix, constantByMatrix, methodGJ` realizan operaciones necesarias para el algoritmo. Si se requiere, con las directivas de compilación se habilita la ejecución paralela o, de otra forma se ejecutan las operaciones de

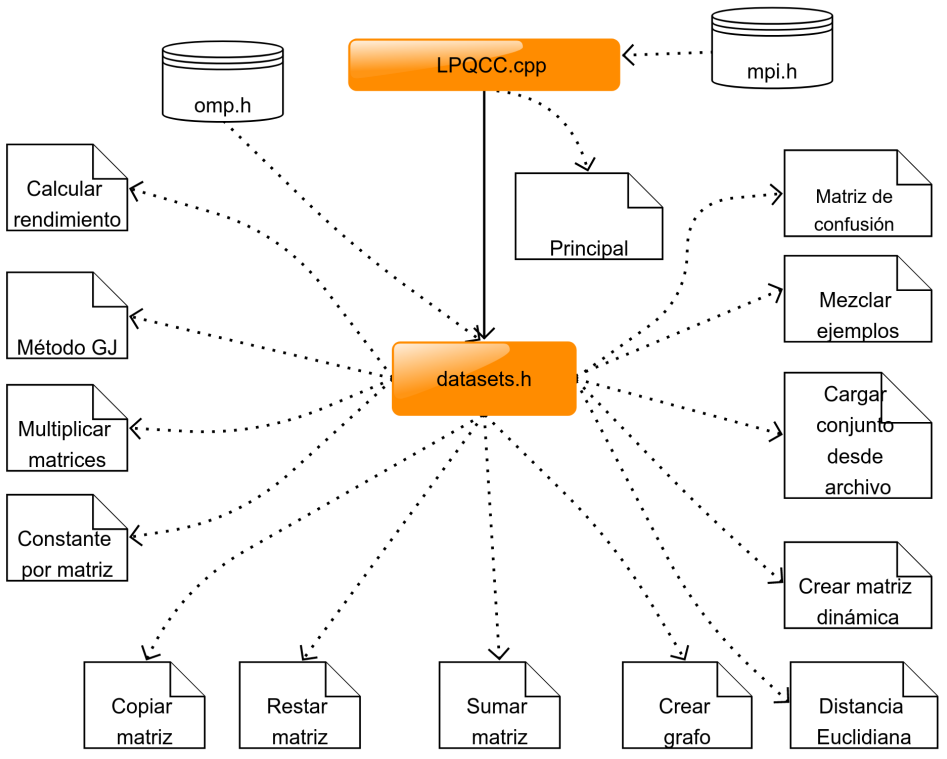
int	`loadDataSetFromFile ( ··· )` Carga el conjunto de datos desde un archivo de texto recibido como parámetro.
void	`shufflingExamples ( ··· )` Recibe la cantidad de elementos de cada clase que deben ser etiquetados de manera aleatoria, al resto le es eliminada la etiqueta.
double **	`createDynamicMatrix ( ··· )` Genera una matriz dinámica para almacenar datos.
double	`euclideanDistance ( ··· )` Calcula la distancia euclidiana entre dos elementos con n características.
void	`createGraphW ( ··· )` Crea la matriz de pesos W a partir de un conjunto de datos recibido como parámetro.
void	`addMatrices ( ··· )` Operación de suma entre dos matrices.
void	`substractMatrices ( ··· )` Operación de resta entre dos matrices.
void	`copyMatrix ( ··· )` Copia los valores de una matriz a otra de las mismas dimensiones.
void	`constantByMatrix ( ··· )` Operación producto entre una constante y una matriz.
void	`multiplyMatrix ( ··· )` Operación de producto entre dos matrices.
void	`methodGJ ( ··· )` Implementa el método de Gauss-Jordan para la obtención de la matriz inversa.
void	`addConf ( ··· )` Se agregan los resultados a la matriz de confusión.
void	`getPerformance ( ··· )` Obtención del rendimiento del clasificador a partir de la matriz de confusión.

Cuadro 3.3: Funciones principales para el algoritmo de propagación de etiquetas usando el criterio del costo cuadrático.

manera secuencial. Finalmente, `addConf` y `getPerformance` son usadas para el cálculo del rendimiento en clasificación del algoritmo. La medición en tiempo es realizada usando la función `time` del sistema operativo y con directivas de la biblioteca *omp.h*.



(a)



(b)

Figura 3.5: Dependencias de los módulos de la implementación (a) secuencial y (b) paralela-distribuida del algoritmo LPQCC.

Capítulo 4

Resultados

En este capítulo se presenta la configuración experimental utilizada para cada uno de los algoritmos implementados y los conjuntos de datos seleccionados para realizar los experimentos. Para cada experimento se muestran las métricas usadas para medir la calidad de los resultados obtenidos con cada algoritmo que sirven para realizar un análisis comparativo en la sección final. Asimismo, se presentan los detalles más importantes a tomar en cuenta con respecto a cómo se distribuyen los datos durante las ejecuciones paralela y distribuida. Finalmente, se presentan los resultados de rendimiento en clasificación y tiempo obtenidos y la forma en que se calculó la calidad de los resultados de la ejecución de los algoritmos de propagación de etiquetas y propagación de etiquetas usando el criterio del costo cuadrático.

4.1. Conjuntos de datos usados

Para los experimentos reportados en este trabajo se han seleccionado conjuntos de datos tomados del repositorio de la UCI [1] (*Iris*) y del sitio web del libro *Semi-Supervised Learning*[2] (*digit1, g241c, COIL* y *SecStr*). Ambos sitios son repositorios de conjuntos de datos ampliamente estudiados y que se han tomado como referencia para una cantidad importante de publicaciones en el área de aprendizaje semi-supervisado [Chapelle et al., 2006].

Iris

El conjunto de datos de flores *Iris* contiene 150 ejemplos de tres especies de la flor Iris (Iris setosa, Iris virginica e Iris versicolor). Cada ejemplo está formado de 4 características: longitud y ancho (en centímetros) tanto de los pétalos como de los sépalos [Fisher, 1936].

[1] UC Irvine Machine Learning Repository https://archive.ics.uci.edu/ml
[2] The Benchmark Data Sets http://olivier.chapelle.cc/ssl-book/benchmarks.html

digit1

Es un conjunto que consiste en puntos de datos de alta dimensionalidad que subyacen en una variedad de baja dimensión. Cada muestra es una imagen del dígito **1** con una resolución 16×16, inicialmente desarrolladas en [Hein and Audibert, 2005]. Las imágenes originales son transformadas de tal forma que se les aplican máscaras y transformaciones de escalamiento y ruido a algunas dimensiones. La figura 4.1 contiene una imagen con el dígito **1** antes y después de aplicarle máscaras y transformaciones. El conjunto de datos contiene 1500 muestras que pertenecen a dos clases, donde cada muestra está formada por 241 dimensiones.

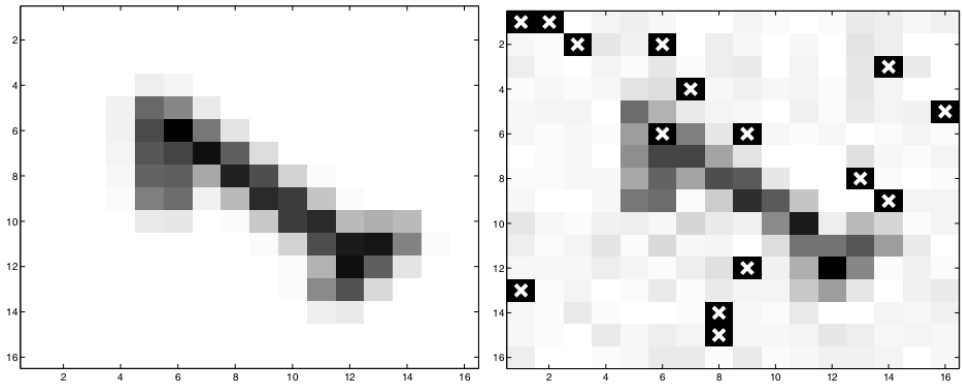


Figura 4.1: Imagen del dígito **1**, en la parte izquierda aparece la imagen original, en la parte derecha la imagen después de escalar, agregar ruido y aplicar máscaras en las posiciones marcadas [Chapelle et al., 2006].

g241c

Es un conjunto de datos de 1500 muestras pertenecientes a dos clases. Las muestras son obtenidas de dos Gaussianas isotrópicas con varianza unitaria con centros ubicados a una distancia de 2.5 con dirección aleatoria. La etiqueta de clase de cada punto representa la Gaussiana de la cual se obtuvo el punto, habiendo 750 puntos de cada una. La figura 4.2 muestra la proyección en dos dimensiones del primer componente principal obtenido del conjunto, la clase etiquetada como +1 usa círculos negros, la clase con etiqueta -1 está marcada con cruces grises

COIL

El conjunto *Columbia Object Image Library* (COIL-100) es un conjunto de imágenes de color de 100 objetos diferentes tomados con una resolución de 128×128, que fueron colocados sobre una base giratoria motorizada con un fondo negro. Las imágenes de los

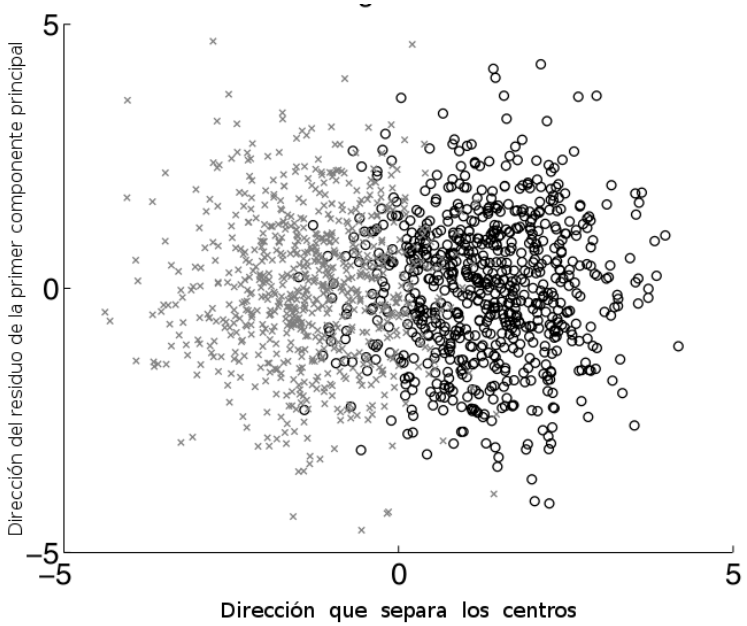


Figura 4.2: Proyección en dos dimensiones del primer componente principal del conjunto de datos g241c [Chapelle et al., 2006].

objetos fueron tomadas con una cámara de color que permanece en una posición fija con respecto a la base que gira con intervalos de 5 grados, dando un total de 72 imágenes.

Cada imagen se procesó tomando bloques de 16×16 píxeles sobre el canal rojo, y usando promedios se genera una reducción a bloques de 8×8. Posteriormente, se toman aleatoriamente imágenes de 24 de los 100 objetos (para dar un total de 24*(360/5) = 1728 imágenes). El conjunto de los 24 objetos seleccionados se particiona en seis clases de cuatro objetos cada una, y se procede a descartar de manera aleatoria 38 imágenes de cada clase para dejar 250 imágenes de cada clase. Finalmente, se aplica un algoritmo para aplicar ruido Gaussiano a los datos con $\sigma = 2.0$ para ocultar la estructura de la imagen. Al final de este proceso se obtiene un total de 1500 muestras con 241 características.

SecStr

Uno de los propósitos de este conjunto de datos es medir la capacidad de algunos métodos para tratar con problemas de gran escala. El objetivo del conjunto es predecir la estructura secundaria de un aminoácido en una proteína basada en una secuencia centrada en una ventana [Chapelle et al., 2006]. En el conjunto existen 28,968 α-helicoidales y 18,888 proteínas β que forman la clase -1 (con 47,856 elementos), el resto de ejemplos con forma de espiral integran la clase +1 (con 35,823 elementos). El conjunto de datos tiene 83,679 ejemplos con 315 dimensiones.

4.2. Medidas de rendimiento

Las medidas de rendimiento permiten evaluar los resultados de clasificación de los algoritmos implementados. En el área de aprendizaje computacional es muy común utilizar la **exactitud** como la cantidad de ejemplos de todas las clases que fueron clasificados correctamente.

La exactitud puede ser extraída a partir de una matriz de confusión. Una matriz de confusión permite visualizar los resultados de clasificación. En dicha matriz, cada columna representa el número de predicciones de cada clase, mientras que cada fila representa a las instancias que pertenecen a la clase real o verdadera.

El beneficio más importante de una matriz de confusión es que facilita ver cuántos elementos de cada clase son identificados de manera correcta por un clasificador, y permite establecer con qué clases no funciona de manera correcta. Esto es útil ya que en muchas aplicaciones reales, las etiquetas de clase tienen ciertas relaciones que se aprecian en las matrices. La figura 4.3 muestra el formato de una matriz de confusión para el caso de clasificación binaria.

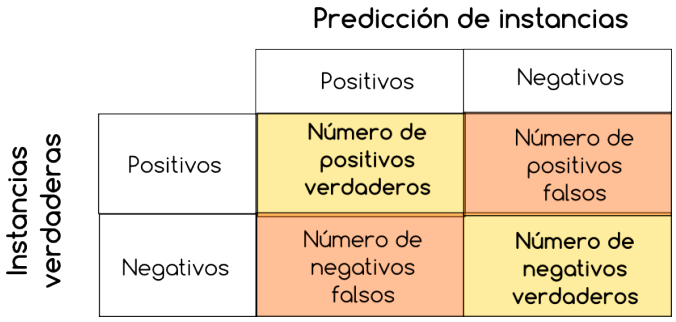


Figura 4.3: Matriz de confusión.

Coeficiente de correlación de Matthews

El coeficiente de correlación de Matthews (MCC) es una herramienta para analizar los resultados de un clasificador. Hace una medición entre la capacidad discriminatoria, la consistencia y el comportamiento coherente con el número de clases y puede aplicarse a conjuntos no balanceados [Cruz-Barbosa et al., 2015]. Los valores que se obtienen están en el intervalo $[-1, 1]$. Un valor de 1 indica predicciones perfectas (correlación total entre las clases observadas y predichas) realizadas por el clasificador, 0 indica que no existe correlación y un valor de -1 que realiza clasificaciones opuestas (correlación negativa) [Jurman et al., 2012].

Para el caso de clasificación binaria, el MCC se calcula usando la siguiente expresión:

$$MCC = \frac{cov(X,Y)}{\sqrt{cov(X,X) \cdot cov(Y,Y)}} \tag{4.1}$$

donde X y Y son los elementos de la clase 1 y 2, respectivamente. $cov(X,Y)$ es la covarianza entre los elementos de la clase 1 y los elementos de la clase 2.

Para el caso de clasificación de N-clases, la ecuación 4.1 se extiende según se muestra en la ecuación 4.2, la cual utiliza los elementos de una matriz de confusión multi-clase para su cálculo. Los detalles de estas ecuaciones pueden encontrarse en [Gorodkin, 2004, Jurman et al., 2012].

$$MCC = \frac{\sum_{k,l,m=1}^{N} C_{kk}C_{ml} - C_{lk}C_{km}}{\sqrt{\sum_{k=1}^{N}\left[\left(\sum_{l=1}^{N} C_{lk}\right)\left(\sum_{f,g=1,\ f\neq k}^{N} C_{gf}\right)\right]}\sqrt{\sum_{k=1}^{N}\left[\left(\sum_{l=1}^{N} C_{kl}\right)\left(\sum_{f,g=1,\ f\neq k}^{N} C_{fg}\right)\right]}} \tag{4.2}$$

4.3. Configuración experimental

El modelo de ejecución de todos los programas en las bibliotecas generadas para cada algoritmo tiene tres etapas, las cuales se muestran en la figura 4.4 y se describen a continuación:

1. Se carga un conjunto de la base de datos extraídos previamente del repositorio reconocido o generado de manera sintética con propiedades que se deseen resaltar.

2. Se envía al clasificador un subconjunto de ejemplos de prueba (datos con etiquetas y mayormente sin etiquetas). Se determina un porcentaje de elementos que conservarán sus etiquetas de manera aleatoria. El resto de ejemplos entran al algoritmo sin etiqueta.

3. Se mide el rendimiento promedio del clasificador a través de un número fijo de repeticiones, donde se le presenta un conjunto de prueba seleccionado por muestreo aleatorio. Los resultados se muestran usando la matriz de confusión.

Para el modelo de memoria compartida, todas las unidades de procesamiento comparten el conjunto de prueba y acceden a los datos de manera independiente. La figura 4.5 muestra la forma de usar más de un procesador para realizar el experimento. Durante los experimentos se pueden usar uno o más procesadores, según se requiera. En la figura 4.5 se muestra específicamente el uso de cuatro unidades de procesamiento en la etapa de

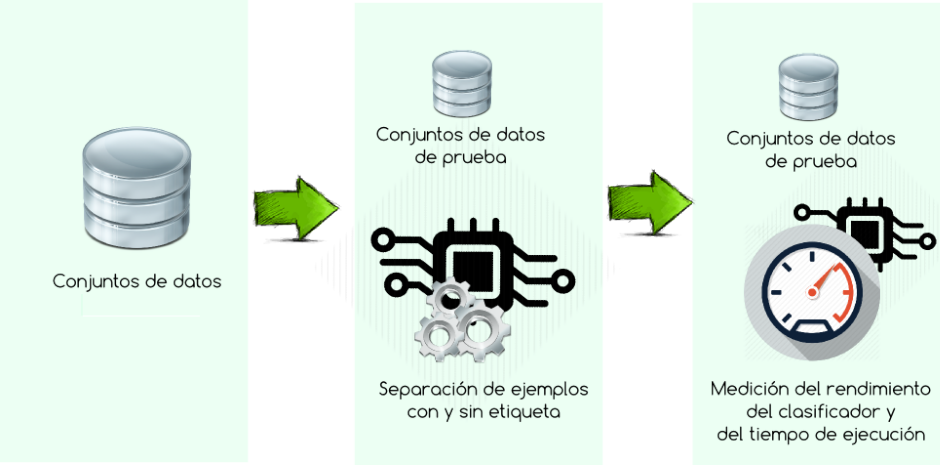


Figura 4.4: Modelo de ejecución de los programas generados por la biblioteca.

pruebas con el objetivo de obtener la medición del rendimiento del clasificador así como el tiempo de ejecución para posteriores comparaciones.

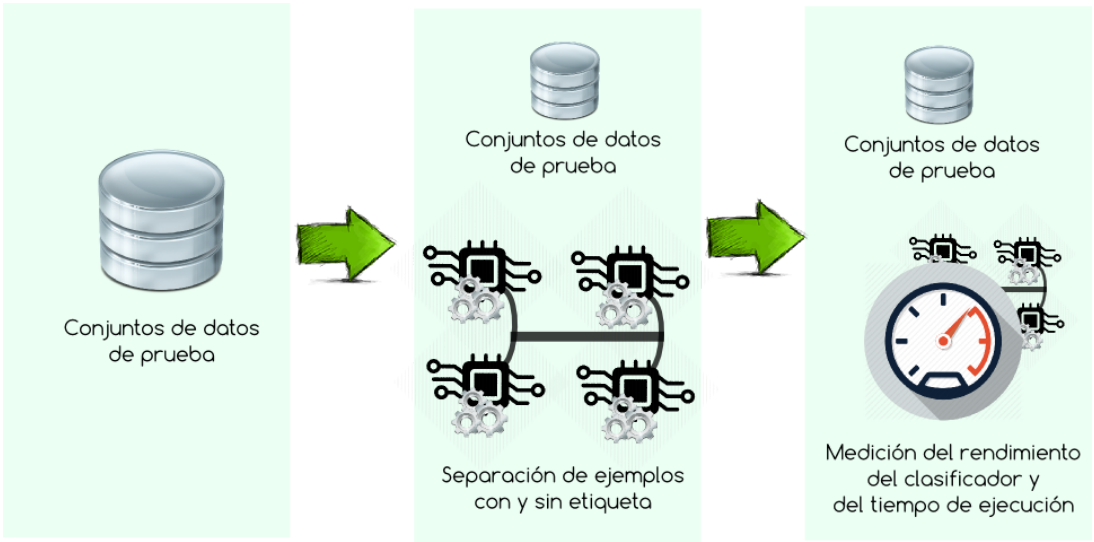


Figura 4.5: Modelo de ejecución de los programas generados por la biblioteca que se ejecutan sobre un entorno paralelo.

Para los modelos de ejecución paralelos-distribuidos, se realiza procesamiento paralelo a nivel de cada nodo. Entre los nodos se establece comunicación para la transferencia de datos y sincronización de las tareas que se hayan definido de manera independiente (ver figura 4.6). Existe una interacción entre los nodos durante la separación para la transferencia de datos y distribución de procesamiento de tareas, así mismo cada nodo tiene una copia de los ejemplos etiquetados. Una de esas copias es usada por el clasificador para hacer la medición del rendimiento.

Durante la etapa de pruebas, dependiendo de las características del conjunto de datos

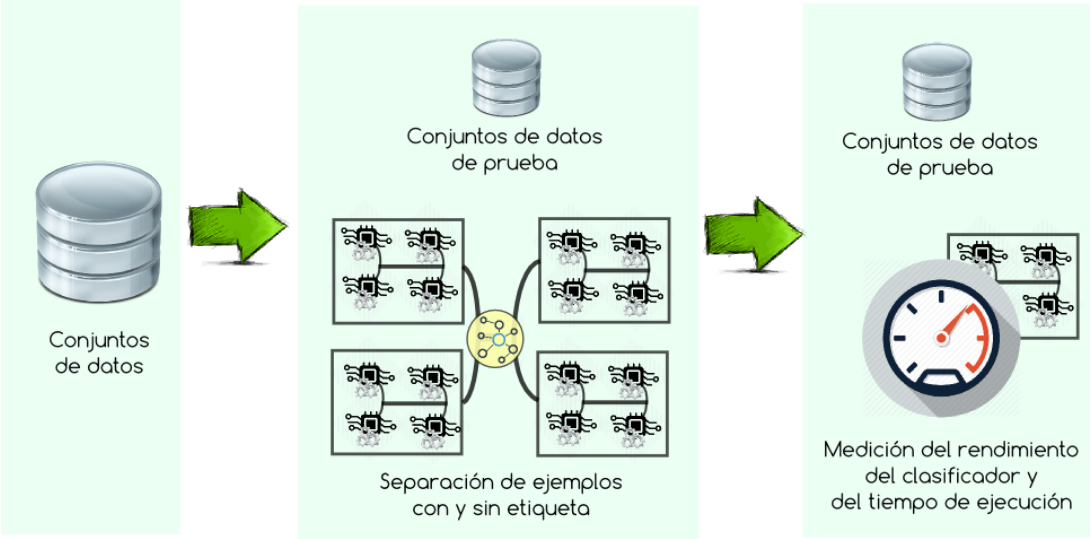


Figura 4.6: Modelo de ejecución de los programas generados por la biblioteca que se ejecutan sobre un entorno paralelo y distribuido.

puede emplearse más de un procesador y/o más de un nodo. La figura 4.6 muestra el procesamiento paralelo en la etapa de experimentación con cuatro nodos, donde cada nodo tiene cuatro unidades de procesamiento.

Implementación paralela y distribuida del algoritmo de propagación de etiquetas

La implementación paralela se ejecuta en una computadora con más de una unidad de ejecución (*core*). Como ya se comentó anteriormente, la ejecución es realizada usando el modelo *fork-join*, donde, el procesamiento en las regiones paralelas es dividido entre los hilos de ejecución disponibles. Por otro lado, puede ser cambiada la cantidad de hilos en tiempo de ejecución usando directivas de OpenMP, esto último independientemente de la cantidad de núcleos físicos disponibles en el equipo en que se está ejecutando. Durante la ejecución, el archivo con el conjunto de datos es cargado en la memoria compartida por todas las unidades de procesamiento, éstas dividen la carga de procesamiento entre el número de hilos disponibles usando variables privadas que mantienen zonas de memoria independientes entre sí, para poder aplicar el paralelismo a nivel de datos. En la figura 4.7 se muestra cómo se realiza el cálculo de la matriz W del algoritmo LP de manera paralela, donde la matriz de datos contiene P filas (cada fila corresponde a una observación), con Q columnas (dimensión de cada observación).

Las operaciones sobre los elementos de la matriz W utilizadas para obtener la matriz T, también pueden ser realizadas de manera independiente usando la misma división de datos entre las unidades de procesamiento, como se muestra en la figura 4.8. Todas las unidades comparten un apuntador a las variables que mantienen el acceso a las matrices

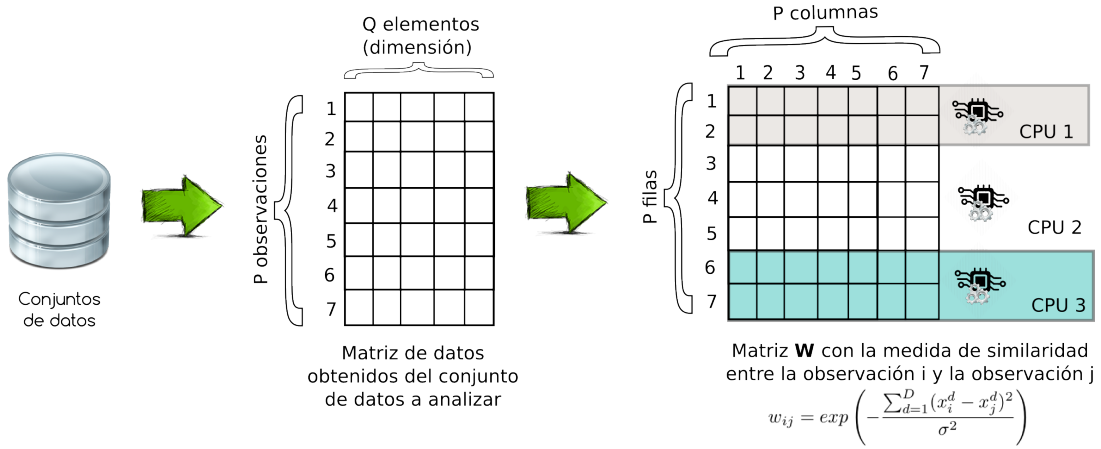


Figura 4.7: Matriz que almacena el conjunto de datos para calcular la matriz W de manera concurrente.

W y T, pero acceden de manera privada a regiones independientes permitiendo así la ejecución concurrente.

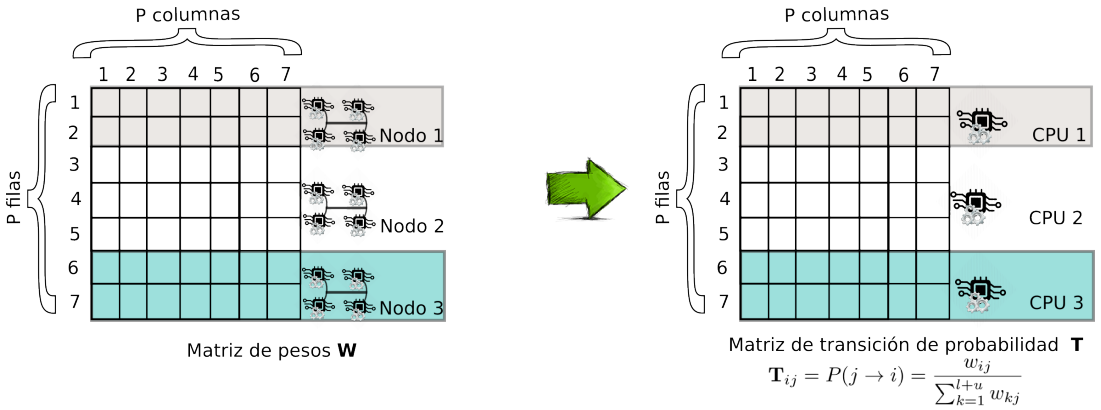


Figura 4.8: Uso de la matriz W para calcular la matriz T.

La parte que requiere mayor cantidad de procesamiento es la multiplicación de matrices. De manera análoga a las figuras 4.7 y 4.8, la figura 4.9 muestra el cálculo del producto $Y' = T \times Y$. La versión implementada considera que la matriz Y' mantiene además escrituras en la región marcada para cada unidad de procesamiento, por otro lado, la matriz Y es de acceso de sólo lectura por lo que no existe problema de conflictos. Internamente, al terminar el producto con una complejidad de $O(p^2 \cdot c)$, donde p es la cantidad de ejemplos y c la cantidad de clases se realiza una copia de los datos de Y a Y' para la siguiente iteración.

La versión distribuida del programa carga un archivo replicado en todos los servidores (nodos) en que se está ejecutando con la información de cada una de las observaciones a través de una matriz. El archivo es cargado completamente por cada nodo, ya que en cada caso se requiere acceso completo a todos los datos para calcular la matriz W usando

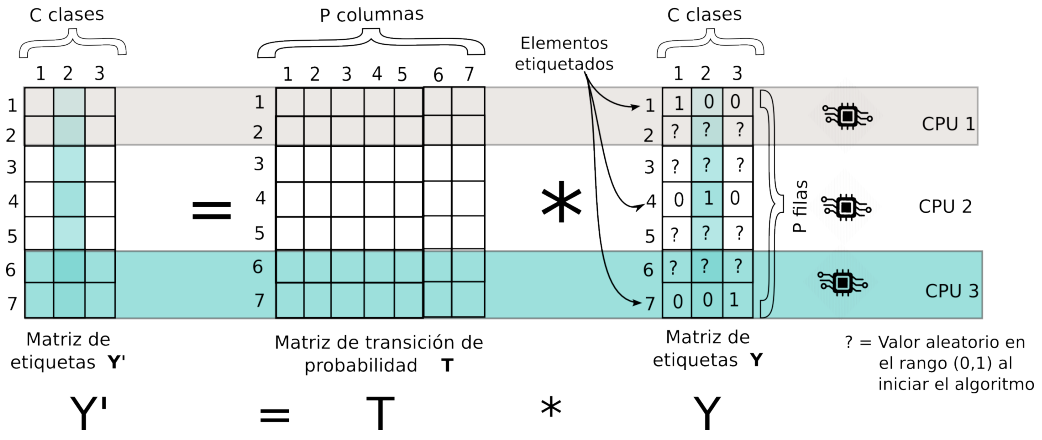


Figura 4.9: Distribución del cálculo paralelo del producto de $Y' = T \times Y$.

la ecuación 2.4. La matriz W es de dimensión $P \times P$ y está distribuida en los nodos del clúster, la figura 4.10 muestra un ejemplo de la distribución de la matriz W en 3 nodos para su procesamiento.

La implementación distribuida considera también paralelismo a nivel de cada nodo. Cada nodo ejecuta usando las unidades de procesamiento disponibles los cálculos necesarios con la información local. En el caso de la matriz W, se carga la región que deberá procesar en la memoria de cada servidor, la diferencia radica en que los cálculos sobre la matriz son realizados de manera concurrente en subconjuntos independientes que cumplen con la propiedad de paralelismo de datos por diferentes unidades de procesamiento. El tamaño de cada subconjunto depende del número de unidades de procesamiento disponibles en cada nodo con el objetivo de mantener un balanceo.

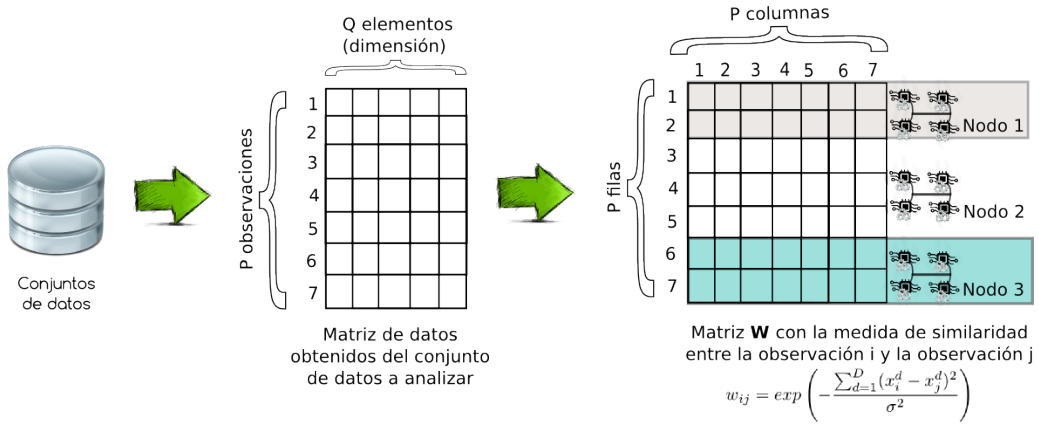


Figura 4.10: Matriz que almacena el conjunto de datos para calcular la matriz W de manera distribuida y paralela.

Cada servidor contiene un subconjunto de filas de W definido por código que utiliza para calcular el mismo subconjunto de filas en la matriz de transición T usando paralelismo (ver ecuación 2.5 y figura 4.11).

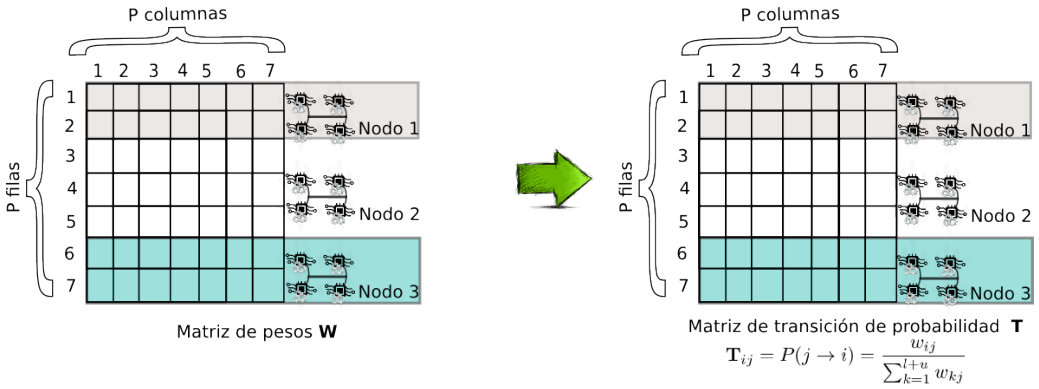


Figura 4.11: Matriz W usada para calcular la matriz T de manera distribuida y paralela.

Todos los servidores tienen una copia de la matriz de etiquetas Y que se usa para calcular el producto $Y' = T \times Y$ ejecutado de manera distribuida en cada iteración, al realizar este producto, se obtienen los mismos elementos en Y que corresponden a las filas en T, como se muestra en la figura 4.12. Al terminar, cada multiplicación se envían las partes de Y' de cada nodo esclavo al nodo maestro, generando así la actualización de Y'. Posteriormente, éste las une y reenvía el vector Y actualizado para la siguiente iteración.

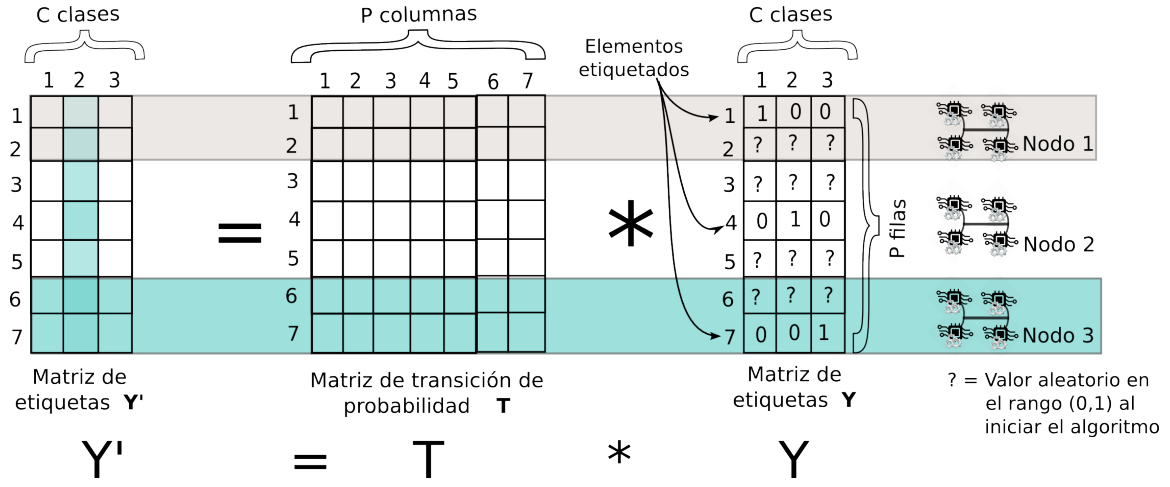


Figura 4.12: Matrices de transición T y de etiquetas Y para calcular el producto $Y' = T \times Y$ de manera distribuida en el algoritmo de propagación de etiquetas.

En la sección A.1 se muestra con mayor detalle los módulos correspondientes para las implementaciones del algoritmo LP.

Implementación paralela y distribuida del algoritmo de propagación de etiquetas usando el criterio del costo cuadrático

Al igual que la implementación paralela del algoritmo de Propagación de etiquetas, el nodo mantiene los datos de las matrices requeridas durante el algoritmo en memoria

y los núcleos de procesamiento acceden a regiones independientes de datos durante su procesamiento en el caso de operaciones de lectura/escritura. La carga del conjunto de datos desde un archivo se realiza de igual forma que para el algoritmo de propagación de etiquetas que se aprecia en la figura 4.7.

La figura 4.13 muestra el cálculo de la matriz W para este algoritmo, que requiere el cálculo de los k-vecinos más cercanos, por lo que esta matriz es de dimensión inicial $P \times k$. El cálculo de los k-vecinos se realiza de manera independiente por cada fila que tiene acceso de sólo lectura al conjunto de datos para calcular la función de similaridad $sim(x_i, x_j)$, de tal forma que en la matriz sólo quedan almacenados los k más similares a la muestra i. Cada unidad de procesamiento tiene asignada un rango de elementos que deberá procesar de manera independiente.

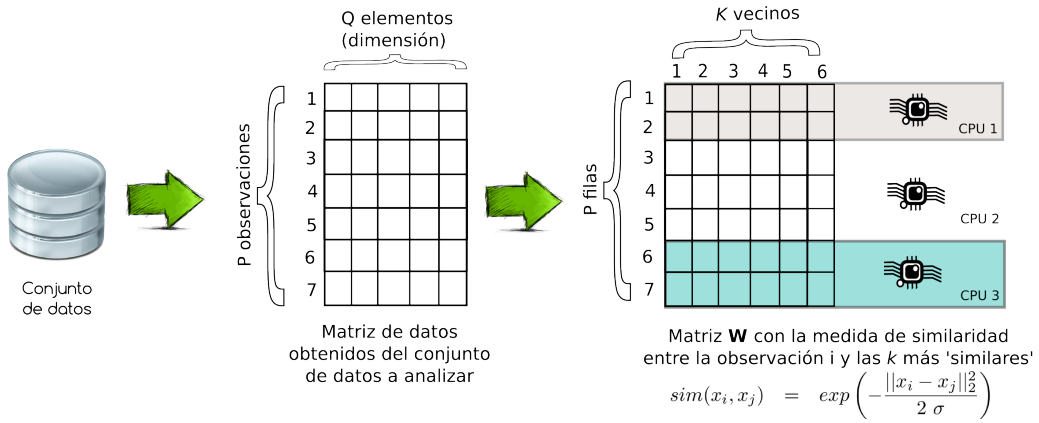


Figura 4.13: Matriz que almacena el conjunto de datos para calcular la matriz W.

Al terminar este proceso, la matriz W cumple con la propiedad de los k-vecinos más cercanos, pero el algoritmo requiere que sea simétrica. Para cumplir la propiedad de simetría, se debe ejecutar para cada elemento $W_{i,j}$ de la matriz la siguiente asignación $W_{i,j} = W_{j,i}$.

Después del proceso anterior, se obtienen las matrices que se muestran en la ecuación 2.13, donde la matriz L es la matriz W, después del proceso anterior. La matriz S mantiene 1's en la diagonal en las filas donde hay elementos con etiqueta y 0's en todos los demás elementos. La matriz Y contiene los ejemplos etiquetados y no etiquetados usando la representación 1 de M. El cálculo de $MI \leftarrow (\boldsymbol{S} + \mu L + \mu\epsilon\boldsymbol{I})$ se realiza concurrentemente asignando rangos independientes de filas a las unidades de procesamiento. Así mismo los detalles de la multiplicación de $MI \times SY$ se pueden apreciar en la figura A.12. Como puede apreciarse, las filas de MI son asignadas a las diferentes unidades de procesamiento, mientras que la matriz SY es leída por todas las unidades para poder generar el resultado en otro vector. Siendo el mismo proceso para la implementación distribuida.

Al igual que en algoritmo de Propagación de Etiquetas, la implementación distribuida requiere que todos los nodos mantengan una réplica del archivo que contiene el conjunto de datos a procesar; un archivo con el conjunto de datos a procesar contiene P filas,

correspondientes a cada ejemplo, Q observaciones, y P etiquetas correspondientes a cada uno de los ejemplos. A partir de este archivo, se carga la información de las observaciones en la matriz de datos (ver figura 4.14). La matriz de datos es usada para calcular la matriz **W**, de manera análoga al algoritmo de propagación de etiquetas; sin embargo, al requerir únicamente k-vecinos más *similares* a la observación i se mantienen únicamente dichos k elementos más similares a cada ejemplo como se muestra en la figura 4.14.

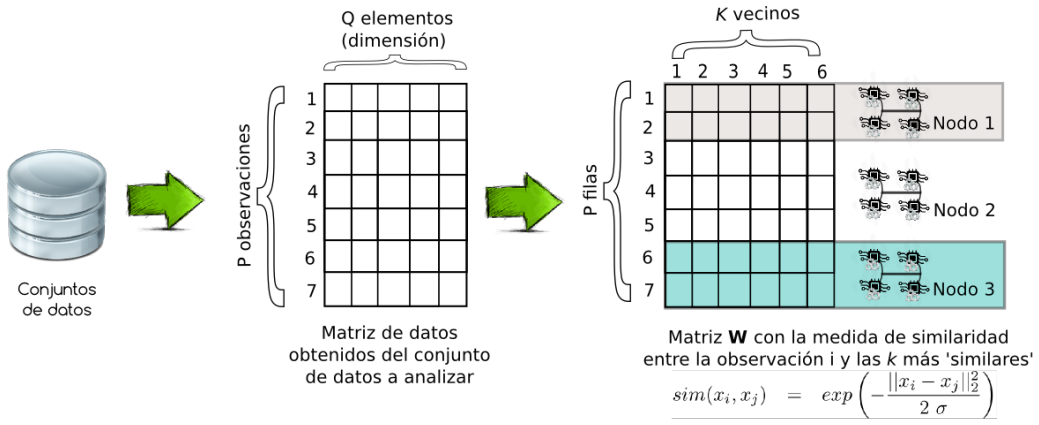


Figura 4.14: Los datos son leídos desde un archivo de texto para generar la matriz **W**.

El cálculo de la matriz **W** se realiza de manera distribuida, asignando porciones de **W** a las diferentes unidades de procesamiento disponibles. El cálculo de todas las distancias posibles desde el ejemplo i hacia cualquiera de las otras observaciones se realiza de manera paralela. La implementación de **W** se realizó usando una lista de adyacencia como se muestra en la figura 2.3. La implementación usa un vector de mapas (*map*) de la biblioteca de plantillas estándar (*STL*, por sus siglas en inglés). Los mapas están implementados usando árboles binarios balanceados cuyas operaciones de inserción y búsqueda tienen complejidad $O(logN)$.

Para conservar la propiedad de simetría en **W**, al igual que en la versión paralela, se recorre nuevamente la matriz y se asignan $W[i][j] = W[j][i]$. En este caso, cada nodo envía todas las actualizaciones al resto de nodos, es decir, si hay N nodos, cada nodo envía N-1 actualizaciones que son recibidas por cada nodo utilizando comunicaciones MPI.

En la sección A.2 se muestra con mayor detalle los módulos correspondientes para las implementaciones del algoritmo LPQCC.

4.4. Resultados del algoritmo de propagación de etiquetas

Como paso previo para la ejecución del algoritmo se hace uso de la heurística del algoritmo de Kruskal modificado (ver anexo A.1) para la obtención del valor σ para cada

uno de los conjuntos de datos a procesar (ver sección 4.1). En el cuadro 4.1 se listan los valores obtenidos de σ para los conjuntos seleccionados.

Conjunto de datos	$\sigma \leftarrow d_0/3$
Iris	0.14142/ 3
COIL	4.2223/ 3
g241c	19.587/ 3
digit1	15.724/ 3
SecStr	4.472136/3

Cuadro 4.1: Resultados del uso de la heurística del algoritmo de Kruskal modificado para encontar el valor σ de los conjuntos de datos seleccionados.

El valor obtenido se usa en la ecuación 2.4 para calcular la matriz de pesos W. El siguiente paso es generar la matriz T (ver ecuación 2.5), que es usada durante el algoritmo para aproximar el valor de Y. Al final de esta sección se muestran los resultados de clasificación y tiempo de ejecución.

Perfil de ejecución del algoritmo de propagación de etiquetas

A partir de la implementación secuencial, se obtuvo el perfil de ejecución (*profiling*) del algoritmo LP, éste permite identificar las partes del algoritmo que ocupan mayor cantidad de tiempo de procesamiento. El perfil se puede obtener usando la aplicación *gprof*[3] que permite identificar las partes del código (módulos o funciones) que requieren mayor cantidad de llamadas y tiempo de ejecución en un programa [Graham et al., 1982]. Como ejemplo, la figura 4.15 muestra el perfil de ejecución para el conjunto de datos *digit1*.

% time	cumulative seconds	self seconds	calls	name
35.87	23.30	23.30	256	mulMat
29.89	42.71	19.41	5621250	distanciaEuclidiana
19.42	55.32	12.61	114	matrizW
14.36	64.65	9.33	114	matrizT

Figura 4.15: Perfil de ejecución del algoritmo de propagación de etiquetas para el conjunto de datos *digit1*.

El perfil de ejecución de la figura 4.15 es similar para los otros conjuntos de datos durante su ejecución. Esto es, destaca que tienen en común que la mayor cantidad de procesamiento es para el cálculo del producto matricial, la distancia euclidiana y la obtención de las matrices W y T del algoritmo. Estas son las partes principales que prometen un aumento de rendimiento en caso de que sea posible hacer mejoras considerables en los algoritmos.

[3]Es un programa para análisis del perfil de ejecución de aplicaciones compiladas en C.

Resultados de rendimiento de clasificación

Dado el análisis anterior, se opta por paralelizar los módulos mencionados anteriormente (ver figura 4.15) a excepción del cálculo de la distancia, debido a que ésta usaría mayor cantidad de tiempo en la sincronización de los hilos de ejecución paralela que en la realización de los cálculos, además de ser dependiente directamente del cálculo de la matriz W.

Una vez que se tiene la implementación del algoritmo de propagación de etiquetas de manera secuencial y paralela se procede a mostrar los resultados de rendimiento.

Las medidas usadas para valorar el rendimiento del modelo semi-supervisado son la *exactitud* de clasificación promedio sobre el número de ejecuciones y el coeficiente MCC. En un principio, todas las muestras de los conjuntos de datos seleccionados cuentan con sus etiquetas de clase correspondiente. Con el fin de evaluar el modelo de manera semi-supervisada, las etiquetas son removidas aleatoriamente en cada ejecución de los experimentos. Para esto, se hizo variar la disponibilidad efectiva de etiquetas de clase desde una configuración muy extrema (1 % de elementos etiquetados) a una relativamente relajada (10 %).

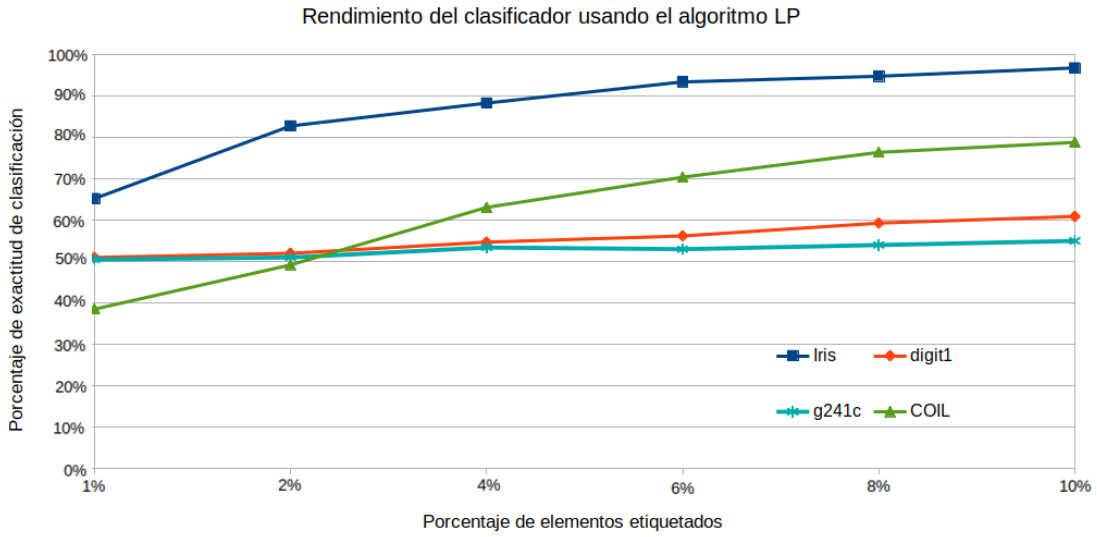


Figura 4.16: Resultados de exactitud promedio usando el algoritmo LP.

Los resultados de clasificación promedio (utilizando 100 ejecuciones de las implementaciones secuencial o paralela) para los conjuntos seleccionados son mostrados gráficamente en la figura 4.16. En esta figura se aprecia buen comportamiento de rendimiento en los conjuntos *Iris* y *COIL*, aún para la configuración extrema, el cual aumenta conforme se incrementa el porcentaje de datos etiquetados disponibles. Sin embargo, el rendimiento se incrementa de manera poco significativa en los otros dos conjuntos de datos. En el caso del conjunto *g241c*, fue diseñado para cumplir la propiedad de agrupamiento, pero no

de variedad (ver sección 2.2. Por otro lado, el conjunto *digit*1, cumple con la propiedad de variedad, pero muestra un agrupamiento débil. Los resultados para los dos últimos conjuntos mencionados indican que el algoritmo LP no es capaz de recuperar las etiquetas correspondientes para conjuntos generados con las suposiciones de agrupamiento y variedad.

Cabe notar que el rendimiento es aún lejano con respecto a los resultados obtenidos para los conjuntos *digit1* y *g241c* por *1-NN*, *SVM*, *MVU + 1-NN*, *Laplacian RLS* o *Cluster Kernel* en [Chapelle et al., 2006].

Los resultados para el conjunto de datos *SecStr* son obtenidos usando la implementación distribuida del algoritmo (ver sección 4.5). Debido a que el algoritmo se basa en un grafo completamente conectado, se usa una matriz implementada usando una lista de vectores de adyacencia. La matriz que tiene la información del grafo se distribuye en los nodos para poder ser procesada, ya que un equipo de cómputo común no tiene la capacidad de procesar este grafo.

Resultados de mejora de tiempo de ejecución paralela

El rendimiento en tiempo de ejecución de la implementación paralela del algoritmo LP genera una mejora importante sólo cuando se analizan conjuntos medianos y grandes. Esta es la razón por la cual quedan excluidos los resultados correspondientes al conjunto de datos *Iris*.

El cuadro 4.2 resume el resultado en tiempo de ejecución del algoritmo LP en sus versiones secuencial y paralela para los conjuntos de datos seleccionados sobre un Servidor Dell PowerEdge T310 (con procesador Xeon X3440 a 2.53GHz). Se observa que la mejora en tiempo está en el orden de la mitad del tiempo de ejecución aproximadamente en los tres casos. En la columna de "algoritmo secuencial" se usa un procesador en el equipo durante la ejecución y el tiempo para "algoritmo paralelo" se usan 8 unidades de procesamiento generando así una mejora de de rendimiento de hasta aproximadamente 2 veces (2x).

Conjunto de datos	**Algoritmo LP Secuencial**	**Algoritmo LP Paralelo**
digit1	136 seg.	58 seg.
g241c	214 seg.	123 seg.
COIL	337 seg.	175 seg.

Cuadro 4.2: Resultados de rendimiento en tiempo de ejecución para las implementaciones secuencial y paralela del algoritmo LP usando los conjuntos de datos seleccionados.

De manera gráfica, en la figura 4.17 se muestra la mejora de rendimiento del algoritmo paralelo, usando desde 1 hasta 8 procesadores. Todas las pruebas fueron hechas sobre un servidor DELL con 8 unidades de procesamiento. Las gráficas muestran un aumento de

rendimiento al pasar de 1 a 2 unidades de procesamiento y hasta 4; sin embargo, se aprecia una disminución en la mejora al llegar a 8 unidades de procesamiento.

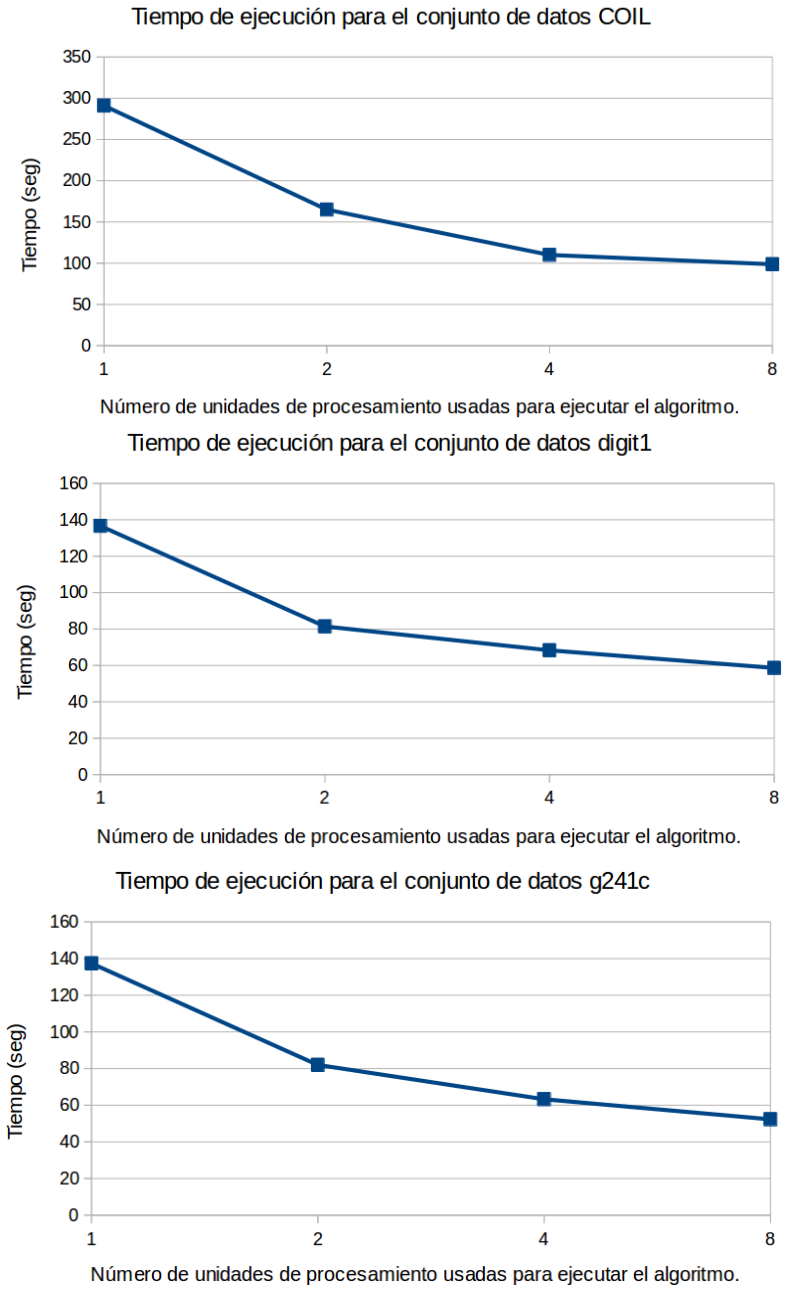


Figura 4.17: Aumento de rendimiento en tiempo del algoritmo LP al usar diferentes cantidades de unidades de procesamiento en los conjuntos de datos $COIL$, $digit1$ y $g241c$.

El conjunto de datos *SecStr* no fue probado en un solo equipo de los disponibles, ya que la memoria de un sólo equipo es insuficiente para almacenar toda la información en las pruebas realizadas. Esto es debido a que se debe mantener un grafo completamente conectado teniendo así necesidad de mantener en el orden de n^2 de almacenamiento,

por lo que éste debe estar distribuido para poder ser almacenado en la memoria de los equipos. En el caso de la prueba se hizo con 20,000 nodos, se mantiene una matriz de $20,000 \times 20,000$ distribuida en 3 equipos Dell PowerEdge T310. Durante las ejecuciones a pesar de usar memoria dinámica para el almacenamiento de los datos después de rebasar 2 ó más Gigabytes el procesamiento se empieza a hacer más lento, por esto es necesario distribuir el procesamiento en varios equipos. El conjunto de datos *SecStr* fue ejecutado en el clúster formado por equipos Dell PowerEdge T310 y en cada nodo se usaron 8 unidades de procesamiento.

La figura 4.18 muestra la mejora promedio en tiempo al usar un procesador en cada nodo (para dar un total de 3), y hasta 8 unidades en cada nodo para la ejecución del algoritmo (para un total de 24). Al igual que los resultados de la figura 4.17 existe una mejora significativa al pasar de 1 a 2 procesadores en cada nodo, pero se aprecia una ligera reducción de la mejora al usar una mayor cantidad de procesadores en cada nodo. En resumen, al procesar un conjunto de datos grande (20,000 patrones) de manera distribuida, se observa una aceleración aproximada de 3.5x.

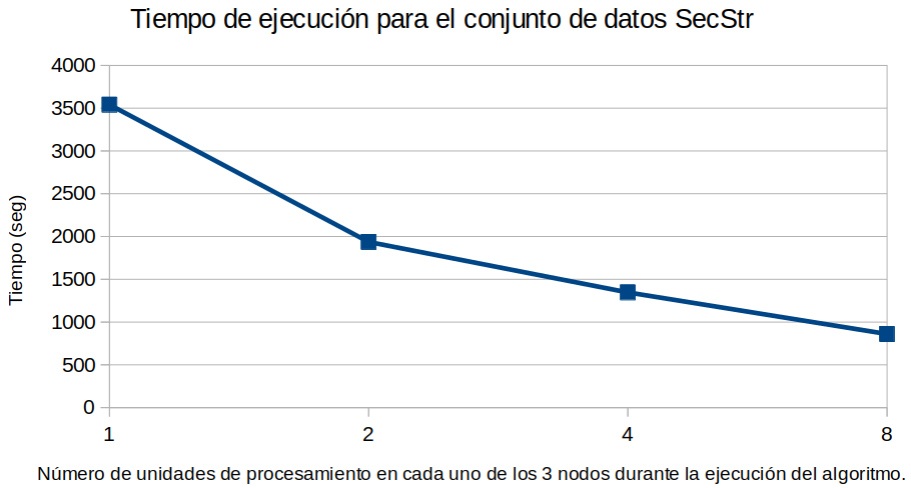


Figura 4.18: Aumento de rendimiento en tiempo del algoritmo LP al usar diferentes cantidades de unidades de procesamiento para el conjunto de datos *Secstr*.

4.5. Resultados del método de propagación de etiquetas usando el criterio del costo cuadrático

El primer paso para el algoritmo es el cálculo de tres parámetros: ϵ, α y k. Se tomaron como punto de partida los valores para α desde 0.1 hasta 0.5 con intervalos de 0.1, de manera similar los valores de ϵ fueron 0.01, 0.2, 0.4 hasta 0.32, y los valores de k de 10 a 50 con pasos de 10. Los valores finales de los parámetros dependen del conjunto de datos y del porcentaje de etiquetas de clase disponible para el clasificador.

Rendimiento del clasificador para el conjunto de datos *digit1, g241c* y *COIL*

Se obtuvieron todas las combinaciones para cada valor de ϵ, α y k. Para cada combinación se obtuvo el promedio de 5 ejecuciones para cada uno de los conjuntos de datos *digit1, g241c* y *COIL*. El valor promedio que generó el mejor rendimiento para cada uno de los conjuntos se muestran en el cuadro 4.3.

	Número de elementos conocidos por clase					
Parámetro	**1.0 %**	**2.0 %**	**4.0 %**	**6.0 %**	**8.0 %**	**10.0 %**
α	0.2	0.5	0.3	0.4	0.1	0.1
ϵ	0.02	0.01	0.02	0.08	0.02	0.32
k	10	10	20	20	20	10

(a) *digit1*

	Número de elementos conocidos por clase					
Parámetro	**1.0 %**	**2.0 %**	**4.0 %**	**6.0 %**	**8.0 %**	**10.0 %**
α	0.3	0.2	0.1	0.1	0.4	0.3
ϵ	0.32	0.32	0.32	0.32	0.16	0.16
k	40	20	50	50	50	50

(b) *g241c*

	Número de elementos conocidos por clase					
Parámetro	**1.0 %**	**2.0 %**	**4.0 %**	**6.0 %**	**8.0 %**	**10.0 %**
α	0.3	0.2	0.4	0.2	0.2	0.1
ϵ	0.04	0.16	0.08	0.32	0.32	0.32
k	10	10	10	10	10	10

(c) *COIL*

Cuadro 4.3: Búsqueda de los valores con mayor rendimiento de clasificación de los parámetros ϵ, α y k para el conjunto de datos (a)*digit1*, (b)*g241c* y (c)*COIL*.

El siguiente paso realizado fue tomar como constantes los valores de α y ϵ, para hacer una búsqueda de k en la vecindad del valor obtenido. En este caso la exploración fue en los rangos de $k \pm 5$. Los resultados de la búsqueda fina para cada conjunto se muestran en el cuadro 4.4. Cada valor mostrado en la tabla es el promedio de 100 ejecuciones. Para cada vecindad de K, se remarcan los porcentajes de mayor rendimiento obtenido, mostrando en la mayor parte de los casos una mejora con respecto al valor de k original.

La figura 4.19 muestra los resultados de rendimiento de clasificación para los conjuntos de datos seleccionados. Al contrastar con los resultados de la figura 4.16, se observa un mejor rendimiento general mayor para el algoritmo LPQCC. Destaca el rendimiento para el conjunto *digit1*, donde se observa un rendimiento superior al 95 % usando únicamente un 1 % de elementos con etiqueta, a diferencia de un resultado cercano al 50 % con la misma cantidad de elementos con etiqueta; al considerar el 10 % de elementos etiquetados, el uso del algoritmo LP no muestra una mejora considerable, mientras que el algoritmo LPQCC se acerca al 100 %.

Número de elementos conocidos por clase											
1 %		2 %		4 %		6 %		8 %		10 %	
k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.
5	95.61 %	5	97.28 %	15	97.57 %	15	97.50 %	15	97.70 %	5	97.72 %
6	92.69 %	6	96.61 %	16	97.21 %	16	97.42 %	16	97.35 %	6	97.90 %
7	**96.04 %**	7	**97.36 %**	17	95.93 %	17	96.62 %	17	97.57 %	7	97.99 %
8	89.83 %	8	96.63 %	18	97.10 %	18	97.35 %	18	**97.80 %**	8	**98.22 %**
9	94.12 %	9	96.34 %	19	**97.89 %**	19	**97.60 %**	19	95.84 %	9	98.21 %
10	94.85 %	10	96.46 %	20	97.19 %	20	97.50 %	20	96.19 %	10	97.84 %
11	91.63 %	11	95.62 %	21	97.11 %	21	97.05 %	21	97.78 %	11	97.66 %
12	93.73 %	12	96.08 %	22	96.51 %	22	97.09 %	22	**97.80 %**	12	97.70 %
13	93.16 %	13	96.22 %	23	95.92 %	23	96.48 %	23	96.90 %	13	97.50 %
14	91.62 %	14	94.57 %	24	96.56 %	24	96.71 %	24	96.25 %	14	98.00 %
15	93.71 %	15	94.64 %	25	96.93 %	25	97.15 %	25	96.67 %	15	97.45 %
Mayor rendim.	96.04 %		97.36 %		97.89 %		97.60 %		97.80 %		98.22 %

(a) *digit1*

Número de elementos conocidos por clase											
1 %		2 %		4 %		6 %		8 %		10 %	
k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.
35	**56.44 %**	15	63.59 %	45	**81.06 %**	45	**84.65 %**	45	**86.83 %**	45	86.64 %
36	52.44 %	16	**68.59 %**	46	79.62 %	46	82.92 %	46	85.00 %	46	**88.13 %**
37	51.28 %	17	62.56 %	47	77.96 %	47	80.64 %	47	84.93 %	47	85.14 %
38	52.21 %	18	62.64 %	48	73.32 %	48	81.69 %	48	83.09 %	48	84.47 %
39	52.20 %	19	62.97 %	49	75.01 %	49	79.84 %	49	81.49 %	49	82.03 %
40	53.20 %	20	62.12 %	50	70.78 %	50	76.82 %	50	81.19 %	50	81.94 %
41	50.50 %	21	59.97 %	51	71.29 %	51	75.57 %	51	80.25 %	51	80.16 %
42	51.51 %	22	62.08 %	52	69.57 %	52	76.74 %	52	77.57 %	52	79.78 %
43	46.21 %	23	62.68 %	53	67.65 %	53	76.52 %	53	78.39 %	53	79.27 %
44	49.80 %	24	62.79 %	54	70.92 %	54	73.16 %	54	77.75 %	54	77.53 %
45	49.10 %	25	57.21 %	55	65.68 %	55	76.13 %	55	76.10 %	55	76.40 %
Mayor Rendim.	56.44 %		68.59 %		81.06 %		84.65 %		86.83 %		88.13 %

(b) *g241c*

Número de elementos conocidos por clase											
1 %		2 %		4 %		6 %		8 %		10 %	
k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.
5	**56.44 %**	5	63.59 %	5	**81.06 %**	5	**84.65 %**	5	**86.83 %**	5	86.64 %
6	52.44 %	6	**68.59 %**	6	79.62 %	6	82.92 %	6	85.00 %	6	**88.13 %**
7	51.28 %	7	62.56 %	7	77.96 %	7	80.64 %	7	84.93 %	7	85.14 %
8	52.21 %	8	62.64 %	8	73.32 %	8	81.69 %	8	83.09 %	8	84.47 %
9	52.20 %	9	62.97 %	9	75.01 %	9	79.84 %	9	81.49 %	9	82.03 %
10	53.20 %	10	62.12 %	10	70.78 %	10	76.82 %	10	81.19 %	10	81.94 %
11	50.50 %	11	59.97 %	11	71.29 %	11	75.57 %	11	80.25 %	11	80.16 %
12	51.51 %	12	62.08 %	12	69.57 %	12	76.74 %	12	77.57 %	12	79.78 %
13	46.21 %	13	62.68 %	13	67.65 %	13	76.52 %	13	78.39 %	13	79.27 %
14	49.80 %	14	62.79 %	14	70.92 %	14	73.16 %	14	77.75 %	14	77.53 %
15	49.10 %	15	57.21 %	15	65.68 %	15	76.13 %	15	76.10 %	15	76.40 %
Mayor Rendim.	56.44 %		68.59 %		81.06 %		84.65 %		86.83 %		88.13 %

(c) *COIL*

Cuadro 4.4: Búsqueda en la vecindad de $k \pm 5$ tomando los valores de ϵ y α constantes del cuadro 4.3 para los conjuntos de datos (a)*digit1*, (b)*g241c* y (c)*COIL*

La respuesta del algoritmo LPQCC es superior para el conjunto de datos $g241c$ con porcentajes cercanos a 60 % para 1 % de elementos etiquetados, y llegando a 70 % con el 10 % de elementos etiquetados, en contraste con porcentajes de entre 50 % y 55 %, usando el algoritmo LP, respectivamente. La respuesta para el conjunto $COIL$ también es supe-

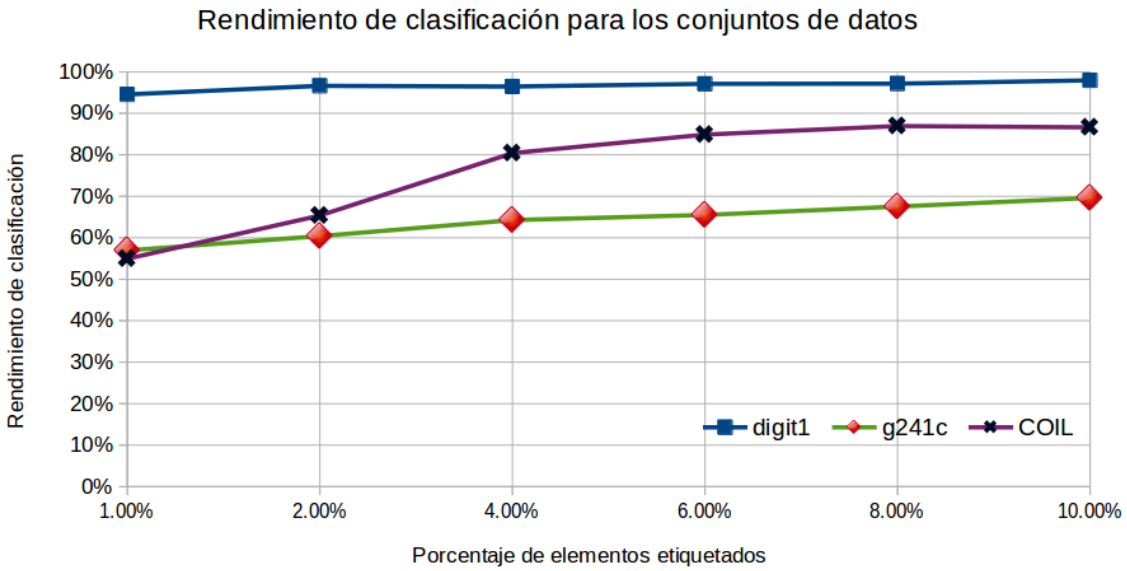


Figura 4.19: Exactitud de clasificación promedio con los mejores parámetros obtenidos para ϵ, α y k para los conjuntos de datos (a)*digit1*, (b)*g241c* y (c)*COIL* usando el algoritmo LPQCC.

rior, pero con una curva de rendimiento más pronunciada con el algoritmo LPQCC, con valores de entre 55 % a 87 %, en contraste con rendimientos de entre 38 % a 78 % usando el algoritmo LP. Considerando las pruebas anteriores, el rendimiento en clasificación es superior en el algoritmo LPQCC.

El valor del coeficiente de correlación de Matthews obtenido para los 3 conjuntos de datos se muestra en la figura 4.20. Se observa una mejor clasificación al incrementar el número de elementos etiquetados. Así mismo, estos son consistentes con el rendimiento del clasificador para los conjuntos *digit*1 y $COIL$. Sin embargo, para el conjunto $g241c$ se observa un rendimiento bajo en esta medida, lo cual indica que el clasificador tiende a sesgarse hacia una clase, esto es posiblemente debido a que el conjunto no cumple con la suposición de variedad.

Rendimiento del clasificador para el conjunto de datos *SecStr*

De la misma forma que se hizo con los conjuntos *digit1, g241c* y *COIL*, se calculó el promedio del rendimiento de 5 ejecuciones para cada una de las combinaciones de α $(0.1, 0.2, 0.3, 0.4, 0.5\ )$, ϵ $(0.1, 0.2, 0.4, \cdots 0.32)$ y k $(10, 20, 30, \cdots 100)$.

Los parámetros que generaron el mejor rendimiento se muestran en el cuadro 4.5. En este caso, se usó la versión distribuida de la implementación del algoritmo LPQCC. Es importante notar que aunque se usaron valores de k hasta 100, no se observaron mejoras al aumentar la vecindad, sino que el mejor valor de k permaneció en valores de hasta 70. Otra observación fue que el valor de k, en este conjunto en particular fue reduciéndose, a diferencia de los los conjuntos anteriores mostrados en el cuadro 4.3. Por lo que se puede

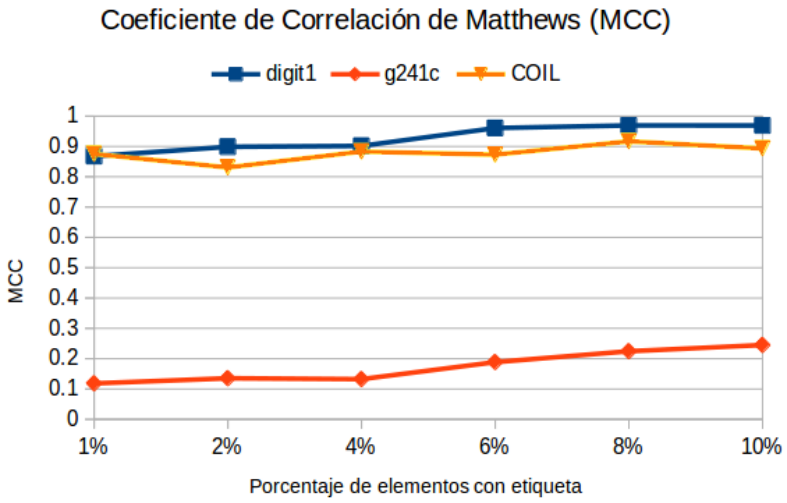


Figura 4.20: Coeficiente de correlación de Matthews para los conjuntos seleccionados utilizando LPQCC.

concluir que no hay un patrón en el valor del parámetro k para el algoritmo.

	Número de elementos conocidos por clase					
Parámetro	**1.0 %**	**2.0 %**	**4.0 %**	**6.0 %**	**8.0 %**	**10.0 %**
α	0.1	0.5	0.5	0.5	0.5	0.1
ϵ	0.32	0.08	0.32	0.08	0.32	0.32
k	70	70	70	50	50	10

Cuadro 4.5: Búsqueda de los valores con mayor rendimiento en clasificación de los parámetros ϵ, α y k para el conjunto de datos *SecStr*.

De igual forma que en los otros conjuntos, tomando como constantes los valores de ϵ, α que generaron la mayor exactitud de clasificación se hizo una búsqueda alrededor de $k \pm 5$, los resultados se muestran en el cuadro 4.6. Cada valor mostrado en el cuadro es el promedio de 10 ejecuciones, el uso de una menor cantidad de ejecuciones es una consecuencia del tiempo requerido, esto como consecuencia del tamaño del conjunto de datos. Los detalles del rendimiento en tiempo se muestran en la siguiente subsección.

La gráfica de la figura 4.21 muestra el porcentaje de exactitud de clasificación promedio obtenido de 10 ejecuciones con los mejores valores obtenidos para los parámetros de ϵ, α indicados en el cuadro 4.5 y el mejor valor de k para los porcentajes de elementos etiquetados, según se muestra en el cuadro 4.6. Los valores obtenidos son cercanos a los obtenidos en [Chapelle et al., 2006], que muestra un rendimiento promedio de 58 % para 100 elementos etiquetados, llegando a 67 % con 1000 elementos etiquetados.

Número de elementos conocidos por clase											
1 %		2 %		4 %		6 %		8 %		10 %	
k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.	k	Rendim.
65	54.19 %	65	54.02 %	65	53.75 %	45	53.84 %	45	55.13 %	5	**61.39** %
66	53.42 %	66	53.58 %	66	56.53 %	46	53.23 %	46	53.87 %	6	60.87 %
67	**54.44** %	67	53.41 %	67	56.35 %	47	**55.30** %	47	56.30 %	7	60.47 %
68	52.94 %	68	54.36 %	68	56.36 %	48	54.23 %	48	54.00 %	8	60.68 %
69	54.20 %	69	53.11 %	69	53.82 %	49	54.02 %	49	55.07 %	9	60.15 %
70	54.12 %	70	54.84 %	70	**57.39** %	50	53.11 %	50	53.41 %	10	59.45 %
71	53.88 %	71	53.79 %	71	55.11 %	51	52.73 %	51	**57.20** %	11	58.50 %
72	53.45 %	72	53.46 %	72	54.63 %	52	54.00 %	52	53.21 %	12	57.84 %
73	53.54 %	73	**55.73** %	73	55.03 %	53	52.10 %	53	54.78 %	13	57.66 %
74	54.05 %	74	53.27 %	74	55.60 %	54	53.45 %	54	56.25 %	14	57.61 %
75	54.40 %	75	54.25 %	75	54.46 %	55	52.41 %	55	56.15 %	15	57.63 %
Mayor rendim.	54.44 %		55.73 %		57.39 %		55.30 %		57.20 %		61.39 %

Cuadro 4.6: Búsqueda en la vecindad de $k \pm 5$ tomando los valores de ϵ y α constantes del cuadro 4.5 para el conjunto de datos *SecStr*.

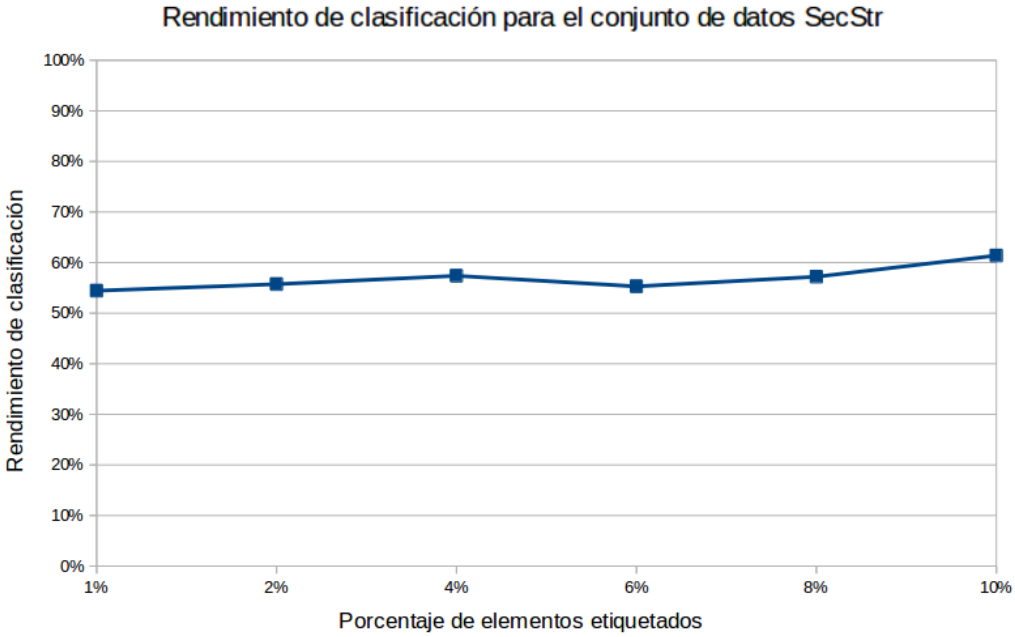


Figura 4.21: Gráfica de rendimiento de exactitud de clasificación para el conjunto de datos *SecStr*.

4.5.1. Rendimiento en tiempo del algoritmo LPQCC

Después de haber implementado la versión secuencial se realizó un análisis usando la herramienta *gprof*, al igual que se hizo en la subsección 4.4. Como puede apreciarse en la figura 4.22, el 93.57 % del tiempo usado para la ejecución del algoritmo es absorbido por la función de cálculo de la matriz inversa. Otra función que ocupa una cantidad importante de tiempo es el cálculo de la distancia Euclidiana. Esta última función no se paralelizó debido a que los cálculos dentro de la función requieren elementos de sincronización que implican que se consuma mayor tiempo. Además, estos cálculos son distribuidos y parale-

lizados en su momento por dependencias del cálculo de la matriz **W**. El perfil de ejecución para los conjuntos *g241c* y *COIL* tienen las mismas características y no se muestran por simplicidad.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
93.57	74.79	74.79	3	24.93	24.93	methodGJ
5.46	79.15	4.36	6750000	0.00	0.00	euclideanDistance
0.46	79.52	0.37	6	0.06	0.06	multiplyMatrix
0.08	79.76	0.06	6	0.01	0.01	addMatrices
0.06	79.81	0.05	19	0.00	0.00	setZerosDynamicMatrix
0.05	79.85	0.04	6	0.01	0.01	copyMatrix
0.04	79.88	0.03	6	0.01	0.01	constantByMatrix

Figura 4.22: Perfil de ejecución del algoritmo LQPCC para el conjunto de datos digit1.

En la figura 4.23 se aprecia el tiempo acumulado de 100 ejecuciones del algoritmo LPQCC usando los parámetros que generan el mejor rendimiento de clasificación para los tres conjuntos antes citados. El tiempo acumulado para el conjunto *digit*1 se muestra en la subfigura 4.23a, se aprecia una curva suave que muestra la reducción de tiempo al incrementar la cantidad de unidades de procesamiento. En el caso de las subfiguras 4.23c y 4.23e, para los conjuntos $g241c$ y $COIL$, respectivamente, es muy reducida la mejora en tiempo al cambiar de 1 a 2 unidades de procesamiento, pero el comportamiento al incrementar las unidades es consistente con la ejecución para el conjunto *digit*1. En los tres casos es importante considerar la desaceleración observada al pasar de 16 a 32 unidades, esto es consistente con la ley de Amdahl. Esta observación es consistente con las curvas de incremento de velocidad que se muestran en las subfiguras 4.23b, 4.23d y 4.23f. En estos casos las curvas de velocidad indican una mejora de entre 6 y 7 veces usando 32 unidades de procesamiento.

Es importante destacar que aunque el conjunto COIL tiene 6 clases, a diferencia de los otros dos conjuntos que tienen dos clases, las curvas de respuesta en tiempo son similares.

Por otro lado, la implementación distribuida del algoritmo LPQCC usando 8,000 ejemplos obtenidos usando prototipos sobre el conjunto de datos *SecStr* tuvo un tiempo promedio de ejecución de 93,461 segundos. El uso de prototipos permite mantener las propiedades del conjunto de datos original, pero reducen el número de instancias a analizar. La ejecución fue realizada usando el clúster formado por 2 equipos Dell PowerEdge con 2 hilos MPI, y en cada equipo se usaron 32 núcleos OpenMP (ver sección 3.1). El objetivo de la implementación distribuida es tener una versión que sea capaz de procesar conjuntos de datos de tamaño grande, mismos que ocupan cantidades grandes de memoria y procesamiento que no están disponibles en equipos de cómputo de uso común.

Es importante notar que en equipos de uso común no es posible hacer este procesamiento, debido a las cantidades de memoria requeridas y la limitante en tiempo. A medida que los conjuntos crecen y debido a la complejidad del algoritmo $O(n^3)$, se requiere mayor cantidad de procesamiento y memoria.

De manera similar que para el conjunto $g241c$, los valores del MCC obtenido para el

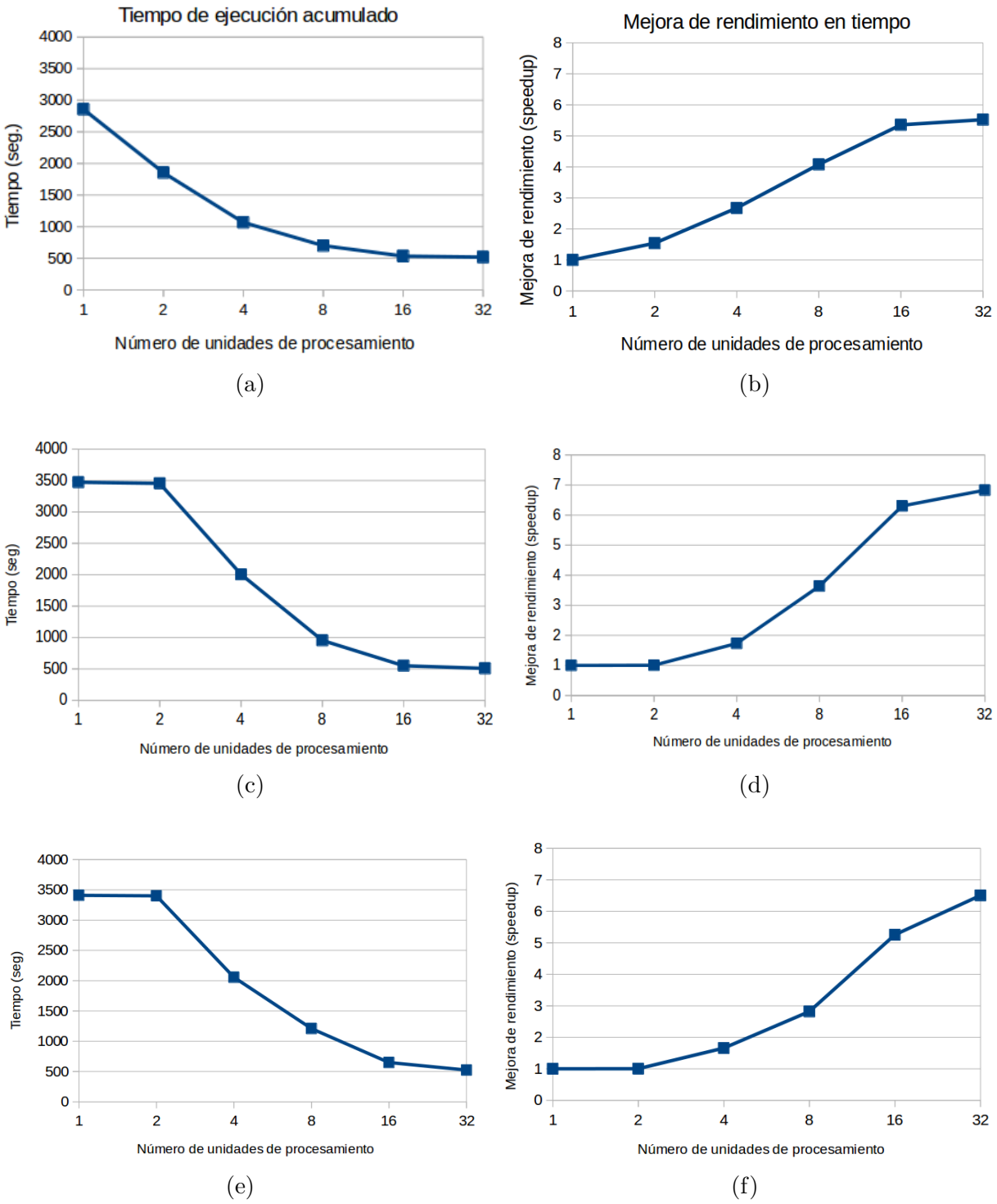


Figura 4.23: Tiempo de ejecución y mejora de rendimiento en tiempo del algoritmo LPQCC usando una cantidad variable de procesadores con los mejores valores de ϵ, α y k para los conjuntos de datos *digit1*, *g241c* y *COIL*, respectivamente.

conjunto *SecStr* varian entre 0.1 y 0.3. Estas indican que LPQCC tiene a sesgarse hacia la clase mayoritaria de este conjunto, además de que es afectado por la cantidad reducida de repeticiones de las pruebas, ocasionadas por la restricción en tiempo para el algoritmo. Sin embargo, los resultados mostrados son comparables con algoritmos de aprendizaje como son: *SVM*, *Cluster Kernel*, *CMN*, *Boosting (Assemble)*, *LapRLS* y *LapSVM* mostrados en [Chapelle et al., 2006].

Capítulo 5

Conclusiones y trabajo a futuro

En este trabajo de tesis se presenta la implementación secuencial, paralela y distribuida de dos algoritmos de aprendizaje semi-supervisado aplicado a cuatro conjuntos de datos construidos considerando que cumplan alguna propiedad de dicho entorno. Los algoritmos programados tienen rendimientos comparables con algoritmos de aprendizaje supervisado, con la consideración de que éstos últimos requieren porcentajes muy grandes de elementos etiquetados para obtener rendimientos similares de clasificación. Las mejoras en tiempo obtenidas para los algoritmos alcanzan entre 6 y 7 veces para los conjuntos medianos. Además se pudo procesar el conjunto *SecStr* utilizando procesamiento paralelo y distribuido, ya que su procesamiento no es posible en equipos comunes.

La construcción y representación del grafo correspondiente a cada conjunto de datos es un punto crucial ya que ésta permite la ejecución de ciertas operaciones sobre los datos necesarias para la ejecución de los algoritmos. Se implementó una representación distribuida para grafos con una cantidad mediana y grande de nodos y una representación usando listas de adyacencia para la ejecución usando los conjuntos de datos medianos o pequeños como lo son el conjunto $digit1$, $g241c$ y $COIL$.

La complejidad de ejecución del algoritmo LP tiene un orden de $O(c \cdot r \cdot n^2)$, donde la mayor parte de los cálculos está relacionada al cálculo de productos de la matriz T con la matriz Y, con la consideración de que la matriz Y es de dimensiones $p \times C$, con C el número de clases existentes en el conjunto de datos, y r es el número de repeticiones que debe hacerse el producto de $T \times Y$. Por otro lado, para el algoritmo LPQCC su complejidad es $O(c \cdot n^3)$, principalmente dependiente del cálculo de la matriz inversa, donde el valor de c está relacionado con operaciones de E/S y el cálculo de la matriz de pesos. Por lo que la implementación paralela de operaciones sobre vectores y matrices de datos, así como su distribución permitió acelerar ambos algoritmos.

El cálculo y análisis de los perfiles de ejecución muestran los módulos que requieren mayor cantidad de procesamiento, y corresponden a los puntos en que se debe buscar mejoras para reducir el tiempo de ejecución. La mejora en tiempo de ejecución es consistente

con la Ley de Amdahl. La implementación distribuida tiene retardos ocasionados por las operaciones de sincronización y comunicación, necesarias para la ejecución del algoritmo. La implementación distribuida permite el procesamiento de conjuntos de datos que no podrían realizarse en equipos de cómputo comunes.

Las primeras pruebas fueron realizadas sobre el conjunto de datos Iris, para el cual ambos algoritmos tuvieron rendimientos superiores al 90 % usando 2 % de datos etiquetados. Las pruebas en el resto de conjuntos mostraron un rendimiento consistentemente superior para el algoritmo LPQCC. Las medidas de similaridad usadas en ambos algoritmos usan como base la distancia euclidiana, con un factor relacionado a propiedades de los conjuntos de datos que deben calcularse antes de la ejecución del algoritmo una sola vez.

Para procesar el conjunto de datos *SecStr* fue necesario usar prototipos, estos tratan de mantener las propiedades del conjunto de datos original, pero reducen el número de instancias a procesar. En este caso se pudo reducir el conjunto de 83,679 a 20,000 y 8,000 ejemplos, respectivamente. Los resultados de clasificación para este conjunto fueron consistentes a los publicados en la literatura.

Como resultado del presente trabajo, fue publicado el artículo *Parallelization of semi-supervised learning algorithms*, en la 8^{th} International Supercomputing Conference in Mexico, que muestra los resultados obtenidos de la paralelización de los algoritmos.

A futuro se considera realizar pruebas usando conjuntos de datos grandes y medianos, así como complementar las pruebas de los algoritmos al aumentar la cantidad de prototipos para el conjunto *SecStr*. Por otro lado, se pretende realizar comparaciones con otros conjuntos de datos reales medianos y grandes que hayan sido analizados usando algoritmos de aprendizaje supervisados. Con respecto a mejorar el tiempo de ejecución de los algoritmos, se considera realizar una implementación usando unidades de procesamiento gráfico además del cómputo paralelo y distribuido, con el objetivo de aprovechar que el cálculo mayormente es realizado por operaciones sobre matrices.

Bibliografía

[Alpaydin, 2010] Alpaydin, E. (2010). *Introduction to Machine Learning (2nd edition)*. The MIT Press.

[Barney, 2013] Barney, B. (2013). OpenMP. https://computing.llnl.gov/tutorials/openMP/. Fecha de consulta: 2018-06-15.

[Bautista, 2011] Bautista, D. (2011). Cómputo de distancias geodésicas para un conjunto de datos grande en problemas de aprendizaje automático. Tesis de licenciatura, Universidad Tecnológica de la Mixteca, México.

[Bekkerman et al., 2011] Bekkerman, R., Bilenko, M., and Langford, J. (2011). *Scaling Up Machine Learning: Parallel and Distributed Approaches (1st edition)*. Cambridge University Press.

[Belkin et al., 2004] Belkin, M., Matveeva, I., and Niyogi, P. (2004). Regularization and semi-supervised learning on large graphs. *International Conference on Computational Learning Theory, COLT 2004*, 3120:624–638.

[Blum and Mitchell, 1998] Blum, A. and Mitchell, T. (1998). Combining labeled and unlabeled data with co-training. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT'98, pages 92–100. ACM.

[Bsoul et al., 2013] Bsoul, Q., Salim, J., and Zakaria, L. Q. (2013). An intelligent document clustering approach to detect crime patterns. *Procedia Technology, 4th International Conference on Electrical Engineering and Informatics, (ICEEI 2013*, 11(6):1181–1187.

[Burch, 2001] Burch, C. (2001). A survey of machine learning. Technical report, Pennsylvania Governor's School for the Sciences, U.S.A.

[Chandra et al., 2001] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel Programming in OpenMP (1st edition)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, U.S.A.

[Chapelle et al., 2006] Chapelle, O., Schlkopf, B., and Zien, A. (2006). *Semi-Supervised Learning (1st edition)*. The MIT Press, U.S.A.

[Chen, 2011] Chen, J. H. (2011). Petascale direct numerical simulation of turbulent combustionfundamental insights towards predictive models. *Proceedings of the Combustion Institute*, 33(1):99–123.

[Chu et al., 2006] Chu, C., Kim, S. K., Lin, Y., Yu, Y., Bradski, G., Ng, A., and Olukotun, K. (2006). Map-reduce for machine learning on multicore. In Schölkopf, B., Platt, J. C., and Hoffman, T., editors, *Advances in Neural Information Processing Systems 19*, number 19 in NIPS'06, pages 281–288. The MIT Press, Canada.

[Cormen et al., 2001] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms (2nd Edition)*. The MIT Press, U.S.A.

[Coulouris et al., 2012] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2012). *Distributed Systems: Concepts and Design (5th edition)*. Addison-Wesley Publishing Company, U.S.A.

[Cruz-Barbosa et al., 2015] Cruz-Barbosa, R., Vellido, A., and Giraldo, J. (2015). The influence of alignment-free sequence representations on the semi-supervised classification of class CG protein-coupled receptors. *Medical & Biological Engineering & Computing*, 53(2):137–149.

[Culler et al., 1997] Culler, D. E., Gupta, A., and Singh, J. P. (1997). *Parallel Computer Architecture: A Hardware/Software Approach (1st edition)*. Morgan Kaufmann Publishers, U.S.A.

[di Blas and Kaldeywey, 2009] di Blas, A. and Kaldeywey, T. (2009). Why graphics processors will transform database processing: Data monster. http://spectrum.ieee.org/computing/software/data-monster/0. Fecha de consulta 2018-06-15.

[Evermann et al., 2005] Evermann, G., Chan, H., Gales, M., Jia, B., Mrva, D., Woodland, P., and Yu, K. (2005). Training LVCSR systems on thousands of hours of data. *International Conference on Acoustics, Speech and Signal Processing, ICASSP 2005*, pages 209–212.

[Fisher, 1936] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Human Genetics*, 7(2):179–188.

[Ginsberb et al., 2009] Ginsberb, J., Mohebbi, M. H., Patel, R. S., Brammer, L., Smolinski, M. S., and Brilliant, L. (2009). Detecting influenza epidemics using search engine query data. *Nature Letters*, 457:1012–1015.

[Gorodkin, 2004] Gorodkin, J. (2004). Comparing two k-category assignments by a k-category correlation coefficient. *Computational Biology and Chemistry*, 28(5-6):367–374.

[Graham et al., 1982] Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A Call Graph Execution Profiler. *ACM Sigplan Notices*, 17(6):120–126.

[Harrington, 2012] Harrington, P. (2012). *Machine Learning in Action (1st edition)*. Manning Publications.

[Hein and Audibert, 2005] Hein, M. and Audibert, J. Y. (2005). Intrinsic dimensionality estimation of submanifolds in R^d. In *Proceedings of the* 22^{nd} *International Conference on Machine Learning*, ICML '05, pages 289–296. ACM.

[Hua et al., 2012] Hua, B. J., Xian, Z. X., Xin, L. Z., and Ping, L. X. (2012). Mixture models for web page classification. *2012 International Conference on Solid State Devices and Materials Science*, 25:499–505.

[Jeffries, 2014] Jeffries, A. (2014). New York City is using big data to predict fires. http://www.theverge.com/2014/1/25/5344334/new-york-city-is-using-big-data-to-predict-fires. Fecha de consulta: 2018-06-15.

[Jurman et al., 2012] Jurman, G., Riccadonna, S., and Furlanello, C. (2012). A comparison of MCC and CEN error measures in multi-class prediction. *Public Library of Science*, 7(8).

[Kajdanowicz et al., 2014] Kajdanowicz, T., Kazienko, P., and Indyk, W. (2014). Parallel processing of large graphs. *Future Generation Computer Systems*, 32:324–337.

[Karanth, 2014] Karanth, S. (2014). *Mastering Hadoop, (1st edition)*. Packt Publishing.

[Kirlidog and Asukb, 2012] Kirlidog, M. and Asukb, C. (2012). A fraud detection approach with data mining in health insurance. *Procedia - Social and Behavioral Sciences*, 62:989–994.

[Lee and Verleysen, 2007] Lee, J. A. and Verleysen, M. (2007). *Nonlinear Dimensionality Reduction (1st edition)*. Springer Science & Business Media.

[Lumsdaine et al., 2007] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berr, J. (2007). Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146. ACM.

[Mayer-Shönberger and Cukier, 2013] Mayer-Shönberger, V. and Cukier, K. (2013). *Big Data: A revolution that will transform How we live, work and Think (1st edition)*. Houghton Mifflin Harcourt Publishing Company.

[Mohri et al., 2012] Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2012). *Foundations of Machine Learning (1st edition)*. The MIT Press.

[Pacheco, 2011] Pacheco, P. (2011). *An Introduction to Parallel Programming (1st edition)*. Morgan Kaufmann.

[Panda et al., 2009] Panda, B., Herbach, J. S., Basu, S., and Bayardo, R. J. (2009). PLANET: Massively parallel learning of tree ensembles with MapReduce. *Proceedings of the VLDB Endowment*, 2(2):1426–1437.

[Parhami, 2005] Parhami, B. (2005). *Computer Architecture: From Microprocessors to Supercomputers (1st edition)*. Oxford University Press.

[Patterson and Hennessy, 2006] Patterson, D. and Hennessy, J. (2006). *Computer Architecture: A Quantitative Approach (4th edition)*. Morgan Kaufmann.

[Patterson and Hennessy, 2008] Patterson, D. and Hennessy, J. (2008). *Computer Organization and Design: The Hardware/Software Interface (4th edition)*. Morgan Kaufmann.

[Quinn, 2003] Quinn, M. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 1st edition.

[Subramanya and Bilmes, 2008] Subramanya, A. and Bilmes, J. (2008). Soft-supervised learning for text classification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 1090–1099. Association for Computational Linguistics.

[Subramanya and Bilmes, 2009] Subramanya, A. and Bilmes, J. (2009). The semi-supervised switchboard transcription project. In *Tenth Annual Conference of the International Speech Communication Association, INTERSPEECH*, pages 1915–1918.

[Subramanya and Talukdar, 2014] Subramanya, A. and Talukdar, P. P. (2014). Graph-based semi-supervised learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(4):1–125.

[Thearling, 1993] Thearling, K. (1993). Massively parallel architectures and algorithms for time series analysis. In *Lectures in Complex Systems*. Westview Press.

[Theodoridis and Koutroumbas, 2006] Theodoridis, S. and Koutroumbas, K. (2006). *Pattern Recognition (3rd Edition)*. Academic Press, Inc.

[Toedte and Wooten, 1997] Toedte, R. and Wooten, D. (1997). Scientific visualization at ORNL. http://www.ornl.gov/info/ornlreview/v30n3-4/scien.htm. Fecha de consulta: 2018-06-15.

[Tretyakov, 2004] Tretyakov, K. (2004). Machine learning techniques in spam filtering. In *Data Mining Problem-oriented Seminar, MTAT*, volume 3, pages 60–79.

[Uberbacher, 1997] Uberbacher, E. (1997). Computing the genome. http://www.ornl.gov/info/ornlreview/v30n3-4/genome.htm". Fecha de consulta: 2018-06-15.

[Upadhyaya, 2013] Upadhyaya, S. (2013). Parallel approaches to machine learning-a comprehensive survey. *Journal of Parallel Distribited Computing*, 73(3):284–292.

[von Hagen, 2010] von Hagen, W. (2010). *Ubuntu Linux Bible: Featuring Ubuntu 10.04 LTS (3rd edition)*. Wiley.

[Zaki, 1999] Zaki, M. (1999). Parallel and distributed association mining: A survey. In *IEEE Concurrency*, pages 14–25.

[Zhu, 2008] Zhu, X. (2008). *Introduction to Semi-Supervised Learning*. Morgan and Claypool Publishers.

[Zhu and Ghahramani, 2002] Zhu, X. and Ghahramani, Z. (2002). Learning from labeled and unlabeled data with label propagation. Technical Report 02-107, Carnegie Mellon University, USA.

[Zhu et al., 2009] Zhu, X., Goldberg, A. B., and Khot, T. (2009). Some new directions in graph-based semi-supervised learning. In *Proceedings of the 2009 IEEE International Conference on Multimedia and Expo*, ICME'09, pages 1504–1507, U.S.A. IEEE Press.

[Zikopoulos et al., 2011] Zikopoulos, P., Eaton, C., deRoos, D., Deutsch, T., and Lapis, G. (2011). *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data (1st edition)*. McGraw Hill.

Anexos

Anexo A

Pseudocódigo de los algoritmos implementados

A.1. Algoritmo de Propagación de etiquetas

El algoritmo de Propagación de etiquetas tiene como principal idea que los puntos que son cercanos entre sí tengan etiquetas similares. Esto significa que la etiqueta de un nodo se propaga hacia sus vecinos de acuerdo a una relación de proximidad. Para esto, el algoritmo representa cada muestra como un nodo dentro de un grafo completamente conectado, donde cada arco representa la medida de similaridad entre cada par de nodos [Zhu and Ghahramani, 2002].

Heurística del algoritmo de Kruskal modificado para encontrar el valor de σ

Para la experimentación sobre cada conjunto de datos, se usó la heurística del algoritmo de Kruskal modificado para la obtención de los valores de σ sobre los conjuntos de datos seleccionados. La figura A.1 muestra gráficamente la obtención del valor de d_0 para la heurística.

El algoritmo de Kruskal es un algoritmo voraz que calcula el árbol de expansión mínimo sobre un grafo conectado con aristas que tienen pesos. El algoritmo implementado mantiene la lista de las aristas del árbol que se va formando ordenadas de menor a mayor. Una estructura $union - find$ se usa para mantener la relación de los nodos del grafo en grupos de manera eficiente. La operación $find$ permite encontrar la etiqueta a la que pertenece el nodo y la operación *union* hace que dos grupos compartan la misma etiqueta [Cormen et al., 2001].

La modificación consiste en mantener una lista de nodos de los que se conoce su etiqueta. Al usar la función unir de la estructura *union-find*, ésta detecta cuando se trata

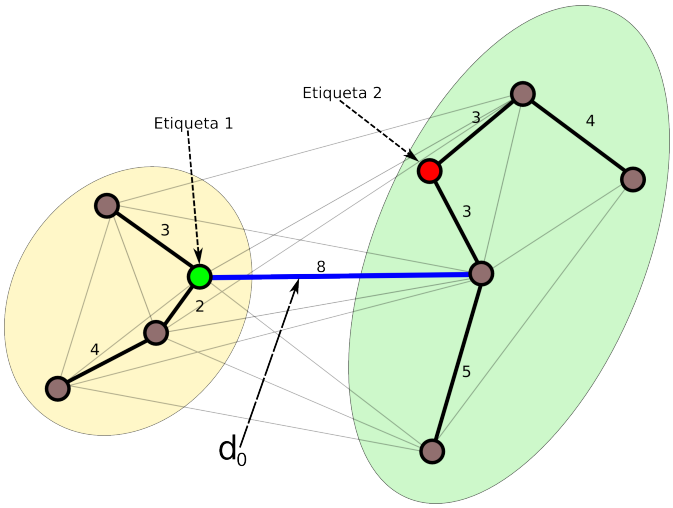


Figura A.1: Representación gráfica de la obtención del valor d_0 para el algoritmo LP.

de unir dos componentes que tienen diferente etiqueta conocida y se detiene el algoritmo regresando el valor del arco d_0. El algoritmo 1 muestra el pseudocódigo con los detalles más importantes.

Algoritmo 1 Obtención del valor de d_0 para el algoritmo LP.

mientras existan arcos a evaluar **hacer**
  Sea $s \leftarrow$ el arco de menor peso
  si s conecta a dos vértices del mismo grupo **entonces**
    Eliminar s del conjunto S debido a que forma un ciclo
  sino si s conecta a dos vértices con etiqueta desconocida **entonces**
    Agregar (union) s a alguno de los dos árboles con etiqueta desconocida
  sino si s conecta a dos vértices con una etiqueta desconocida y una conocida **entonces**
    Agregar (union) s al árbol con etiqueta conocida
  sino si s conecta a dos vértices con etiquetas conocidas diferentes **entonces**
    Regresar la longitud del arco $s \leftarrow d_0$
  fin si
fin mientras

Matriz Y y representación 1 de M

Si se consideran P ejemplos en un problema de C clases, la matriz Y, de $P \times C$ representa las etiquetas a las que corresponde cada ejemplo. Para el ejemplo de la figura A.2, se tiene P=12 ejemplos y C=3 clases. Al inicio del algoritmo LP, se toma como entrada la cantidad de elementos de cada clase que conservarán sus etiquetas, estos ejemplos son seleccionados de manera aleatoria, dejando al resto de ejemplos sin etiqueta. El algoritmo tiene por objetivo calcular la etiqueta para cada ejemplo al que se ha eliminado su etiqueta y la exactitud se calcula sobre estos elementos.

Num Ejemplo	Etiqueta
1	1
2	2 -> ?
3	3 -> ?
4	1 -> ?
5	1 -> ?
6	2
7	3 -> ?
8	3
9	1 -> ?
10	2 -> ?
11	3 -> ?
12	2 -> ?

Matriz Y

	C=1	C=2	C=3
1	1	0	0
2	?	?	?
3	?	?	?
4	?	?	?
5	?	?	?
6	0	1	0
7	?	?	?
8	0	0	1
9	?	?	?
10	?	?	?
11	?	?	?
12	?	?	?

Figura A.2: Representación 1 de M en la matriz de etiquetas Y.

En la parte izquierda de la figura A.2 se muestran en gris a los elementos que han sido seleccionados para conservar sus etiquetas, mismos que serán usados para calcular las etiquetas de los elementos marcados con *?*. Al iniciar el algoritmo los valores en Y para los elementos no etiquetados son inicializados con un valor aleatorio entre 0 y 1, estos valores van cambiando durante las iteraciones hasta mostrar convergencia.

La **representación 1 de M** indica que para el ejemplo i, se toma como la clase a la que pertenece a la columna donde está el mayor valor de probabilidad, por ello al inicio y en el paso 3 del algoritmo LP se coloca 1 en la columna que corresponde a la clase de la etiqueta y 0 en el caso opuesto.

Paralelización y distribución del algoritmo para el cálculo del producto de $T \times Y$

La parte del algoritmo LP que requiere la mayor cantidad de procesamiento es el producto de $T \times Y$. Para paralelizar este proceso, cada unidad de procesamiento tiene una copia completa de Y, debido a que es un vector relativamente pequeño. La matriz W es distribuida en las unidades de procesamiento. La figura A.3 muestra el uso del vector de índices para distribuir el cálculo.

En la versión distribuida, es necesario enviar las regiones correspondientes de la matriz Y' de cada procesador hacia el nodo maestro y éste las reenvía hacia el resto de los nodos (*broadcast*) debido a que es necesario mantener la última versión de Y para realizar el producto en cada iteración. El algoritmo 2 muestra el proceso que se lleva a cabo en cada nodo. El valor de *desplazamiento* en el algoritmo indica el inicio de la región por la que el nodo es responsable de toda la matriz. El valor i es una variable local que permite acceder a la fila almacenada localmente. Por lo tanto el valor $i + desplazamiento$ permite acceder al renglón de la matriz completa.

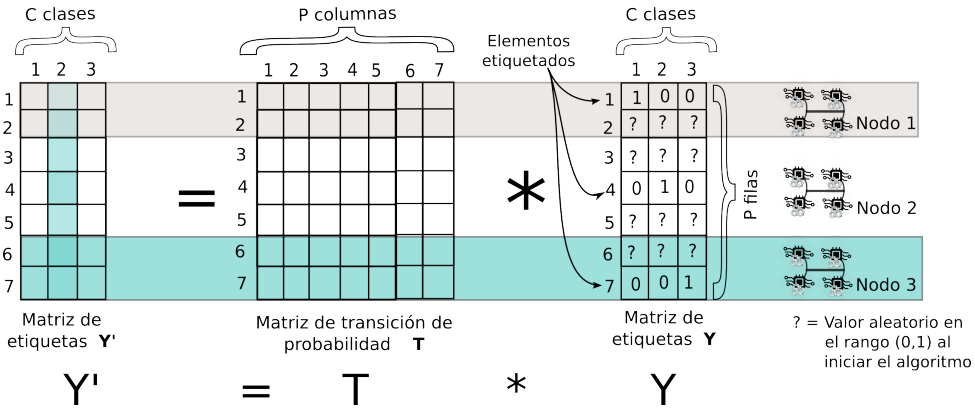


Figura A.3: Paralelización del producto $Y' \leftarrow T \times Y$.

Algoritmo 2 Cálculo del producto $Y' \leftarrow T \times Y$.

```
desplazamiento ← fila inicial a procesar.
numElementos ← número de filas que evaluará.
para i ← 1 hasta numElementos hacer
    para j ← 1 hasta número de clases hacer
        para k ← 1 hasta número de ejemplos del conjunto de datos hacer
            Yp[i+desplazamiento][j] ← T[i+desplazamiento][k] × Y[k][j]
        fin para
    fin para
fin para
```

Conjuntos de datos sintéticos de prueba

Para probar el algoritmo, se generó un conjunto de datos de dos espirales usando las muestras obtenidas de ($x = cos(t)$, $y = sin(t)$, $z = 6 * t$) con $t \in [3\pi/2, 9\pi/2]$ para la primer espiral, y la segunda con un desplazamiento de π en la fase de las funciones trigonométricas, de tal forma que las espirales no se tocan en ningún punto. En la figura A.4 se muestran gráficamente las espirales en 3 dimensiones y el valor d_0 obtenido por la heurística del algoritmo de Kruskal modificado. En la parte izquierda de la figura A.4, la línea indica la conexión entre elementos de espirales distintas que corresponde al valor de d_0 obtenido con la heurística. Por otro lado, en la figura A.5 se muestra el resultado de la ejecución del algoritmo y la clasificación de los elementos de dicho conjunto de datos.

Para la segunda prueba realizada, se usó un conjunto de tres bandas de puntos sobre un plano X, Y. Todos los valores de X fueron obtenidos de una distribución uniforme en el intervalo $[0, 1]$. Los valores en Y también son generados por una distribución uniforme, pero están ubicados de manera aleatoria en los rangos $[0, 0.5]$, $[1, 1.5]$ y $[2, 2.5]$.

Una vez encontrado el parámetro σ para los conjuntos de datos antes mencionados, se procede a usar el algoritmo LP. Los resultados se ilustran en la figura A.6. Esta figura se divide en dos partes, en la parte superior se muestra el conjunto original donde la flecha en

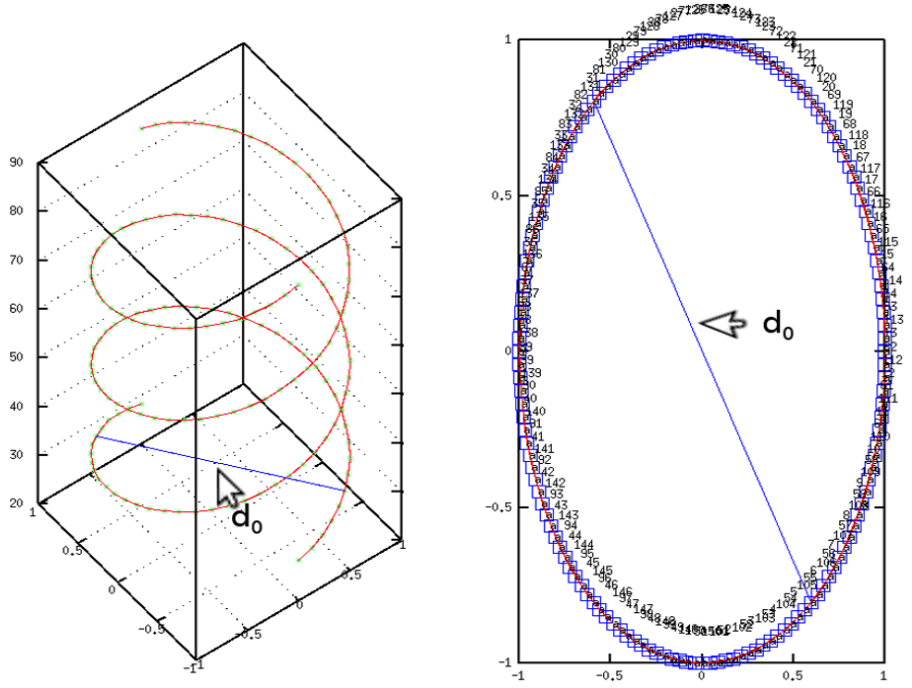


Figura A.4: Uso de la heurística del algoritmo de Kruskal modificado para encontrar $\sigma = d_0/3$ del conjunto de datos de 2 espirales.

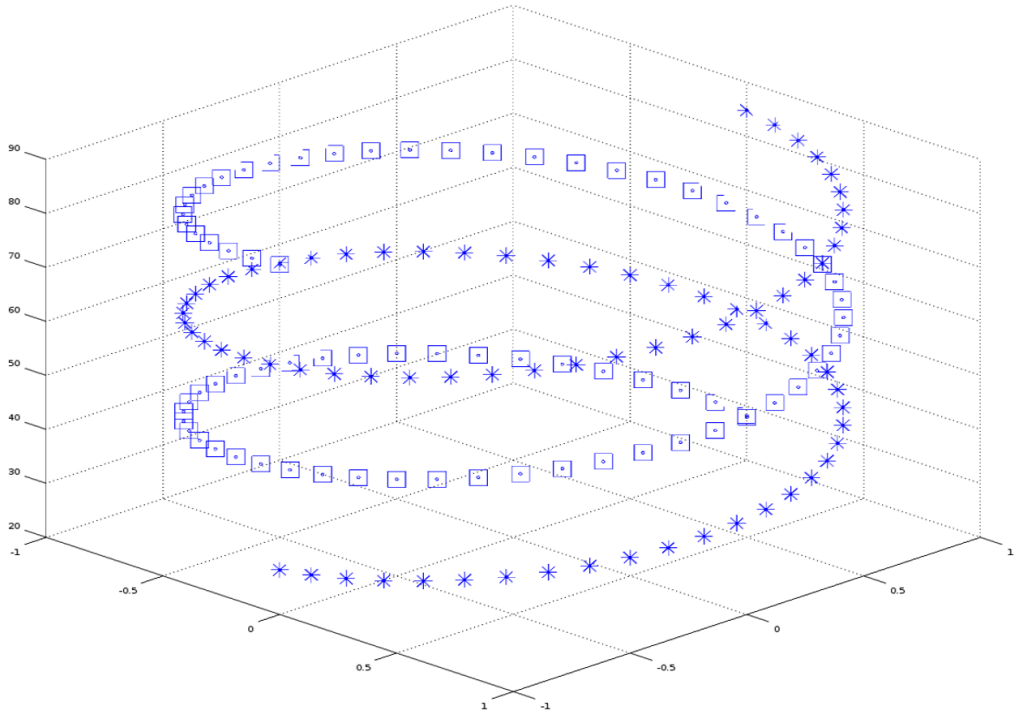


Figura A.5: Ejecución del algoritmo LP para el ejemplo de las espirales.

cada una de las bandas muestra los elementos etiquetados. En la parte inferior, se aprecia el resultado de la ejecución del algoritmo, en donde puede apreciarse una separación (clasificación) completa de los elementos de cada una de las bandas.

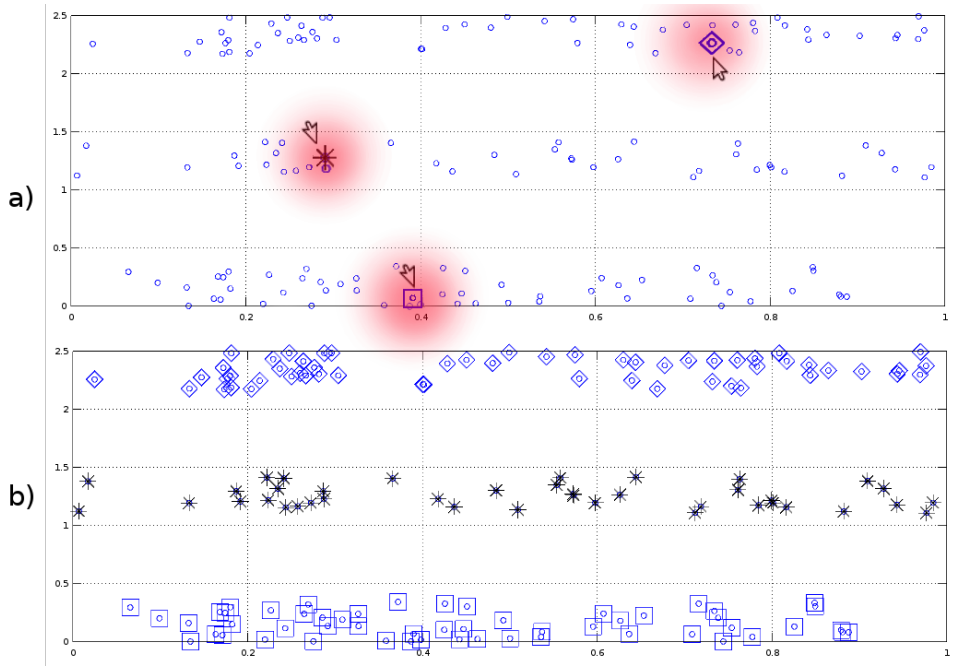


Figura A.6: Ejecución del algoritmo de propagación de etiquetas para el conjunto de las 3 bandas. a) Conjunto inicial con una etiqueta por clase. b) Resultados de clasificación usando el algoritmo LP.

Los resultados anteriores (ver figuras A.5 y A.6) muestran que la heurística para calcular el valor de σ funciona al menos con datos sintéticos, por lo cual es adoptada para los experimentos de este trabajo.

A.2. Algoritmo de Propagación de etiquetas usando el criterio del costro cuadrático

Este es una extensión del algoritmo de propagación de etiquetas, cuya principal característica es tratar de explotar las relaciones entre los datos etiquetados y no etiquetados, y la geometría de sus relaciones mapeados a un grafo creado usando la regla k a través de una medida de similaridad. El valor de k es obtenido usando una búsqueda de parámetros, como se describe en la sección 2.3.3

El algoritmo se basa en la minimización del error definido en la ecuación A.1:

$$\sum_{i=1}^{l}(\hat{y}_i - y_i)^2 = ||\hat{Y}_l - Y_l||^2. \tag{A.1}$$

La matriz de etiquetas $\hat{Y}$ se obtiene usando la ecuación A.2 que contiene los parámetros μ y ϵ que son obtenidos usando una búsqueda en malla.

$$\hat{Y} = (\boldsymbol{S} + \mu L + \mu\epsilon\boldsymbol{I})^{-1}\boldsymbol{S}Y \tag{A.2}$$

En el resto de esta sección se muestran los detalles principales de la implementación del algoritmo.

Selección de los elementos etiquetados

Como entrada al algoritmo, se lee un conjunto de datos con las características de cada ejemplo y su etiqueta (ver figura 4.4). Como parte de la experimentación, se deben seleccionar de manera aleatoria una cantidad de elementos con etiqueta por cada clase, eliminando las etiquetas del resto.

En la figura A.7(a), se muestra un conjunto de datos, para el que se han seleccionado dos elementos de cada clase de manera aleatoria (elementos sombreados). El resto de etiquetas no se toman en cuenta durante el algoritmo, ya que el objetivo de éste es que las pueda calcular. La matriz Y, en la figura A.7(b) muestra la representación 1 de M para los ejemplos que conservan su etiqueta y para los demás el símbolo **?** indica que el valor deberá ser calculado.

Matriz S

Una vez seleccionados los ejemplos, se crea la matriz de etiquetas S, ésta mantiene 1's en la diagonal en los sitios donde hay elementos con etiqueta y 0's en todos los demás

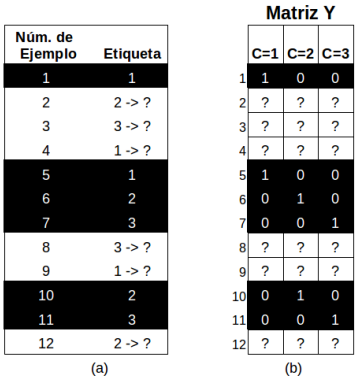


Figura A.7: Selección de los ejemplos de manera aleatoria que permanecen con etiqueta y la representación 1 de M para la matriz Y.

elementos. Para el ejemplo de la figura A.7 la matriz S correspondiente se muestra en la figura A.8. En este caso se muestran 2 elementos etiquetados aleatoriamente por clase.

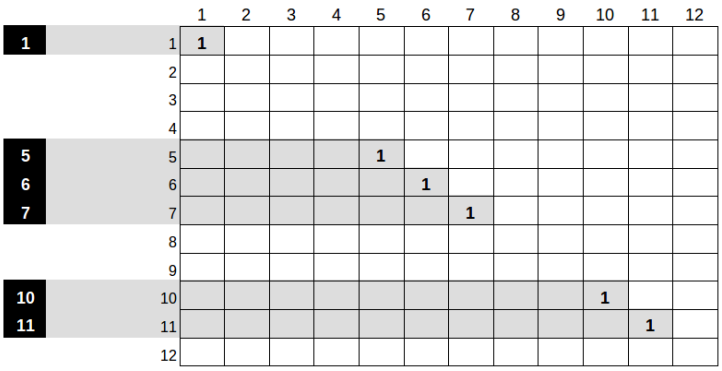


Figura A.8: Matriz S

Obtención de la matriz de similaridad $W(k, \sigma)$ y el laplaciano L

La matriz W representa la similaridad entre los elementos de los ejemplos a procesar, a diferencia del algoritmo de propagación de etiquetas, se almacenan únicamente los k elementos más *similares*, por lo que para cada fila se calculan las medidas de similaridad hacia el resto de ejemplos, pero se almacenan únicamente k, desechando el resto.

La matriz W se genera usando la función kernel de base radial (kbr ó rbf, del inglés *radial basis function*), con amplitud σ (que representa la desviación estándar) para la medida de similaridad. El primer paso es calcular los valores $W_{i,j}$ usando la función rbf para cada elemento i con respecto a la información del resto de elementos j. El siguiente paso es almacenar las k medidas de similaridad mayores en una lista. Finalmente, el algoritmo requiere que la matriz sea simétrica, por lo que se recorren todos los elementos de la matriz W y se realiza la operación de $W_{j,i} \leftarrow W_{i,j}$ para forzar que la matriz sea

simétrica.

En la implementación paralela, las filas se distribuyen en los procesadores usando directivas de OpenMP para el cálculo de los k ejemplos más similares de manera concurrente. El procesamiento para hacer simétrica a $W(k, \sigma)$ se hace de manera secuencial debido a las dependencias de lecturas/escrituras.

En la implementación distribuida se usa el algoritmo 3. Antes de la ejecución se mantiene un vector compartido entre todos los nodos con las asignaciones de filas que procesa cada uno. En la primer parte, cada nodo hace el cálculo de $W(k, \sigma)$ para las filas que le fueron asignadas y hace que $W_{j,i}$ se vuelva simétrico al realizar la operación $W_{j,i} \leftarrow W_{i,j}$. Sin embargo, al hacer que $W_{j,i}$ sea simétrica, algunas filas salen del rango asignado, estas filas son enviadas al final a los nodos correspondientes usando paso de mensajes, y al mismo tiempo se reciben los valores correspondientes generados por otros nodos.

Algoritmo 3 Generación de la matriz $W(k, \sigma)$ para el algoritmo LPQCC.

1: **para** i $\leftarrow$ en el rango de las filas asignadas al nodo **hacer**
2: **para** j $= 0 \leftarrow$ para cada columna de W **hacer** ▷ Cálculo de $W(k, \sigma)$
3: Calcular $w_{ij} \leftarrow exp\left(-\frac{\sum_{d=1}^{D}(x_i^d - x_j^d)^2}{\sigma^2}\right)$
4: **fin para**
5: $W_i \leftarrow$ rbf de los k ejemplos con valor mayor.
6: **fin para**
7: **para** i $\leftarrow$ cada índice de filas de W **hacer** ▷ La matriz W debe ser simétrica
8: **para** $W_{i,j} \leftarrow$ cada elemento en W_i **hacer**
9: $W_{j,i} \leftarrow W_{i,j}$
10: **fin para**
11: **fin para**
12: **Enviar** los valores de de $W(k, \sigma)$ que no fueron asignados al nodo actual hacia el resto de nodos para forzar la simetría.
13: **Recibir** los valores de $W(k, \sigma)$ del resto de nodos que deben ser asignados al nodo actual para forzar la simetría.

El laplaciano L para ser aplicado en la ecuación A.2 se obtiene usando la ecuación 2.2 sobre la matriz $W(k, \sigma)$. Las operaciones en la implementación distribuida se realizan sobre los datos existentes en cada nodo de manera independiente, de igual forma que en las implementaciones secuencial y paralela.

Cálculo de la inversa de una matriz

La parte más importante del algoritmo es el cálculo de la matriz inversa, para esta tarea se usó el algoritmo de Gauss-Jordan. Esta operación ocupa más del 90 % del tiempo de procesamiento del algoritmo según las pruebas realizadas con *gprof* (ver sección 4.5.1).

En la primer parte del algoritmo 4 se toma el valor que está sobre la diagonal de la matriz extendida $[M, I]$, para convertirlo en 1. El resto del algoritmo, para cada fila es

Algoritmo 4 Algoritmo de Gauss-Jordan para el cálculo de la matriz inversa.

```
1: para i ← cada fila de la matriz hacer
2:     Asignar aux ← M_{i,i}.                    ▷ Si aux es 0, terminar el programa.
3:     para j ← índice de cada columna dentro de la fila i hacer
4:         Asignar M_{i,j} ← M_{i,j}/aux.
5:     fin para
6:     para k ← índice de cada fila de M, k ≠ i hacer
7:         Asignar aux ← M_{k,i}
8:         si aux ≠ 0 entonces
9:             para j ← índice de cada columna dentro de la fila k hacer
10:                Asignar M_{k,j} ← M_{k,j}/aux − M_{i,j}.
11:            fin para
12:        fin si
13:    fin para
14: fin para
```

hacer que el resto de valores sobre la columna se transformen en 0 realizando operaciones con la fila pivote y cada una de las filas recorridas usando el índice k.

Algoritmo 5 Algoritmo paralelo de Gauss-Jordan para el cálculo de la matriz inversa.

```
1: para i ← cada fila de la matriz hacer
2:     Asignar aux ← M_{i,i}.                    ▷ Si aux es 0, terminar el programa.
3:     para j ← índice de cada columna dentro de la fila i hacer
4:         Asignar M_{i,j} ← M_{i,j}/aux.
5:     fin para
6:     Distribuir de manera dinámica o estática las filas a procesar por cada unidad de procesamiento.
7:     para k ← índice de cada fila de M asignado por directivas de OpenMP, k ≠ i hacer
8:         Asignar aux ← M_{k,i}
9:         si aux ≠ 0 entonces
10:            para j ← índice de cada columna dentro de la fila k hacer
11:                Asignar M_{k,j} ← M_{k,j}/aux − M_{i,j}.
12:            fin para
13:        fin si
14:    fin para
15: fin para
```

El algoritmo 5 para el cálculo de la inversa de una matriz usa directivas de OpenMP. El ciclo principal que determina la fila pivote i se ejecuta de manera secuencial. Después de transformar en 1 el elemento $M_{i,i}$, al dividir todos los elementos de la fila sobre $aux \leftarrow M_[i,i]$, se debe indicar a las directivas de OpenMP que la fila pivote (i) es privada y el resto de vectores ($k \neq i$) de la matriz son compartidos. Cabe notar que no hay fallos en las escrituras debido a que no presentan dependencias.

Para el algoritmo 6 se toma en consideración que la matriz a procesar ya está distribuida en los nodos en que se realiza el procesamiento, de tal forma que cada nodo controla los

índices de las filas que tiene asignadas. Para cada iteración del ciclo principal el nodo que controla al resto es el que tiene entre sus filas asignadas el pivote. El algoritmo muestra la parte en que el nodo tiene la fila pivote i, y envía esta fila al resto de los nodos. Todos los nodos saben en qué ciclo procesarán la fila pivote.

Algoritmo 6 Envío y actualización del renglón i cuando el hilo actual contiene el pivote.

```
1: para i ← cada fila de la matriz hacer
2:     Asignar aux ← M_{i,i}.                    ▷ Si aux es 0, terminar el programa.
3:     para j ← índice de cada columna dentro de las filas asignadas al nodo i hacer
4:         Asignar M_{i,j} ← M_{i,j}/aux.
5:     fin para
6:     Enviar la fila i al resto de nodos para eliminar los valores en la columna i de sus respectivos
   renglones asignados.
7:     para k ← índice de cada fila de M asignado al nodo, k ≠ i hacer
8:         Asignar aux ← M_{k,i}
9:         si aux ≠ 0 entonces
10:            para j ← índice de cada columna dentro de la fila k hacer
11:                Asignar M_{k,j} ← M_{k,j}/aux − M_{i,j}.
12:            fin para
13:        fin si
14:    fin para
15: fin para
```

Para el caso en que un nodo contiene el pivote, el nodo debe transformar en 1 al elemento en $M_{i,i}$ al dividir toda la fila entre este valor. Enseguida, debe enviar la fila pivote hacia el resto de nodos para que sea usada para transformar en 0 a todos los elementos de la columna i usando la fila pivote en las filas que tienen asignadas los otros nodos.

El algoritmo 7 muestra el caso en que la fila pivote no está en el rango de filas que actualmente está procesando el nodo, en cada iteración i se espera la fila i. Después de recibir la fila, se procesan las filas que tiene asignadas el nodo y se hacen 0 los valores en la columna i. El algoritmo muestra la parte en que el nodo recibe la fila pivote i y actualiza la fila que tiene asignada haciendo 1 el elemento (i, i) y 0 el resto de elementos de la columna i.

Algoritmo 7 Recepción y actualización del renglón i cuando el hilo actual no contiene el pivote.

```
1: para i ← cada fila de la matriz hacer
2:     Recibir la fila i al resto de nodos para eliminar los valores en la columna i de sus respectivos renglones asignados.
3:     para k ← índice de cada fila de M asignado al nodo, k ≠ i hacer
4:         Asignar aux ← M_{k,i}
5:         si aux ≠ 0 entonces
6:             para j ← índice de cada columna dentro de la fila k hacer
7:                 Asignar M_{k,j} ← M_{k,j}/aux − M_{i,j}.
8:             fin para
9:         fin si
10:    fin para
11: fin para
```

En la figura A.9 se muestra gráficamente el proceso del cálculo de la matriz inversa para tres nodos. En primer lugar aparecen las filas que procesa cada nodo y el vector de asignaciones. En la segunda fila se aprecia el procesamiento de la fila pivote para transformar en 1 el elemento en la posición $M_{i,i}$ y dividir sobre este valor el resto de valores que lo acompañan. Así mismo, se muestra que el resto de filas en el nodo deben ser usadas para la eliminación de los elementos en la columna i, y el envío de la fila pivote al resto de los nodos. En los otros nodos se aprecia el proceso de eliminación en las filas que mantienen. En el tercer renglón se muestran otras iteraciones realizadas en el primer nodo.

Multiplicación de matrices

Después de obtener la inversa en la ecuación A.2, el resultado se debe multiplicar por $SY \leftarrow S \times Y$. El producto de $S \times Y$ debe haber sido previamente calculado, este producto es procesado y almacenado por todos los nodos de manera concurrente debido a que la cantidad de procesamiento y almacenamiento es muy pequeño con respecto a lo necesario para la matriz inversa. El orden de esta operación es de $O(p+C \times p)$, donde p es el número de ejemplos en el conjunto de datos procesados.

Sea $MI \leftarrow (\boldsymbol{S}+\mu L+\mu\epsilon\boldsymbol{I})$, el producto de $MI \times SY$ en la versión paralela se distribuyen las filas de MI en las unidades de procesamiento usado directivas de OpenMP y se conserva una copia de SY compartida. Por otro lado, para la implementación distribuida, de manera análoga, se mantiene el vector de asignaciones con información acerca de las filas de la matriz que están almacenadas en cada nodo.

En la figura A.10 la matriz MI está distribuida en 3 nodos, el primero almacena de la fila 1 a la 7, y los otros dos nodos de la fila 8 a la 14, y de la 15 a la 20, respectivamente. Todos los nodos tienen una copia de la matriz SY. En las figuras A.12(a), A.12(b), y A.12(c) se muestran los datos procesados por cada uno de los nodos durante el producto

matricial, la región de la matriz $\hat{Y}$ obtenida como respuesta las etiquetas conocidas (remarcadas en negro) y no conocidas (marcadas como $C = \#$) en cada región de la matriz $\hat{Y}$, estos últimos son los que el algoritmo tiene por objetivo obtener.

La figura A.12 muestra la multiplicación de la matriz MI en tres nodos. En A.12(a) se mantienen las filas 1 a 7, en A.12(b) las filas 8 a 14, y en A.12(c) las filas 15 a 20. La matriz SY es local para cada nodo ya que ocupa poco espacio. El producto obtenido es local para cada nodo y es una región de la matriz $\hat{Y}$. En cada figura se muestran con fondo negro los nodos etiquetados y el resto de nodos son no etiquetados, para éstos últimos es que se desea calcular la etiqueta.

Obtención de las etiquetas

En la figura A.11 se muestra en la parte izquierda una matriz $\hat{Y}$ para los que hay elementos etiquetados (fondo negro) y elementos para los que se desea calcular la etiqueta. La etiqueta para dicha fila corresponde con el índice de la columna en la que está el máximo valor. Para la implementación paralela y secuencial, los valores están localmente en el nodo en que se está ejecutando la aplicación.

Para la implementación distribuida, las etiquetas son calculadas concurrentemente, como se muestra en la parte derecha de la figura A.11. Después de calcular las etiquetas, éstas son transmitidas al nodo principal y éste es el encargado de generar la matriz de confusión para calcular la precisión.

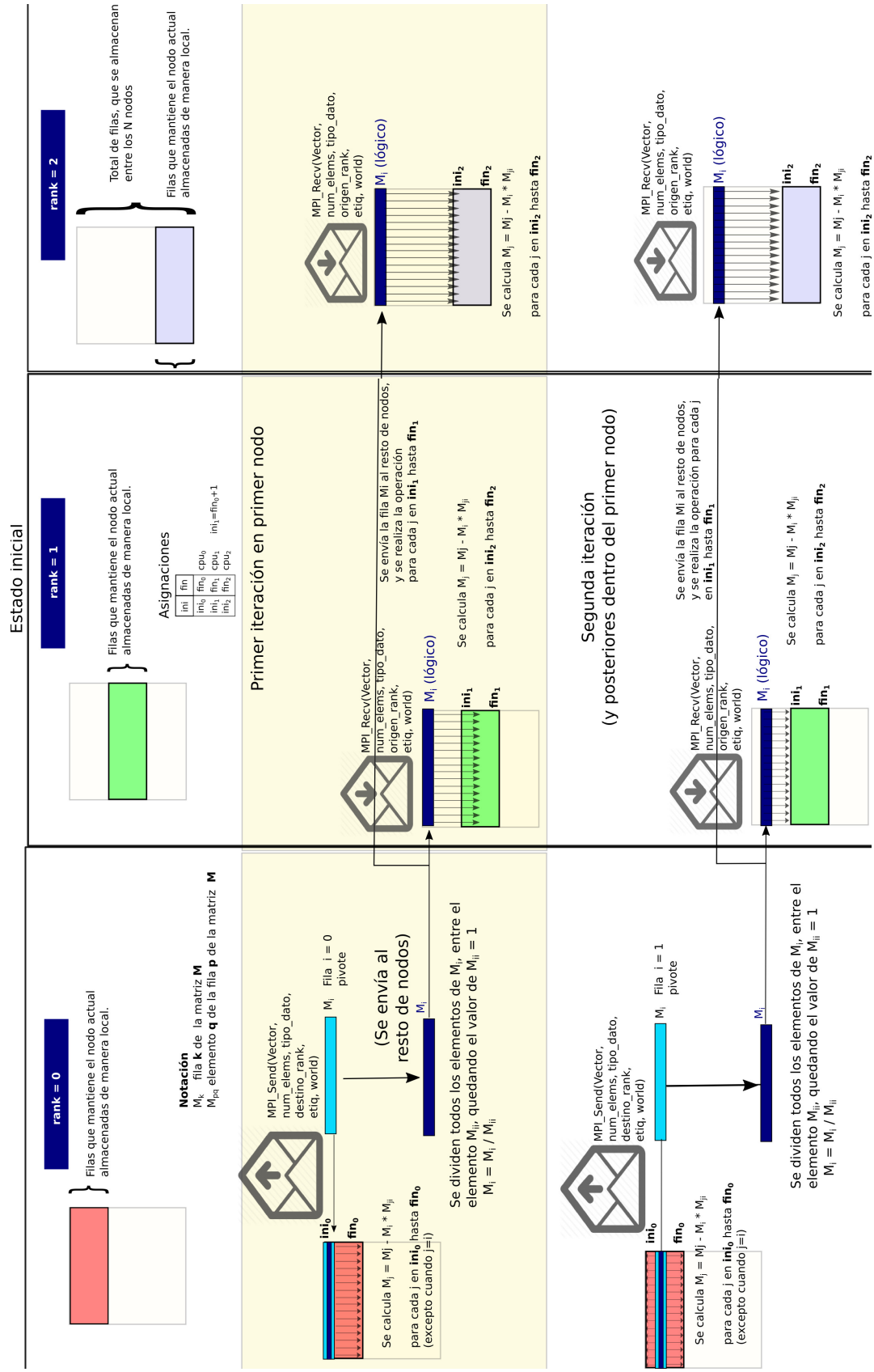


Figura A.9: Detalles más importantes del algoritmo para el cálculo de la matriz inversa.

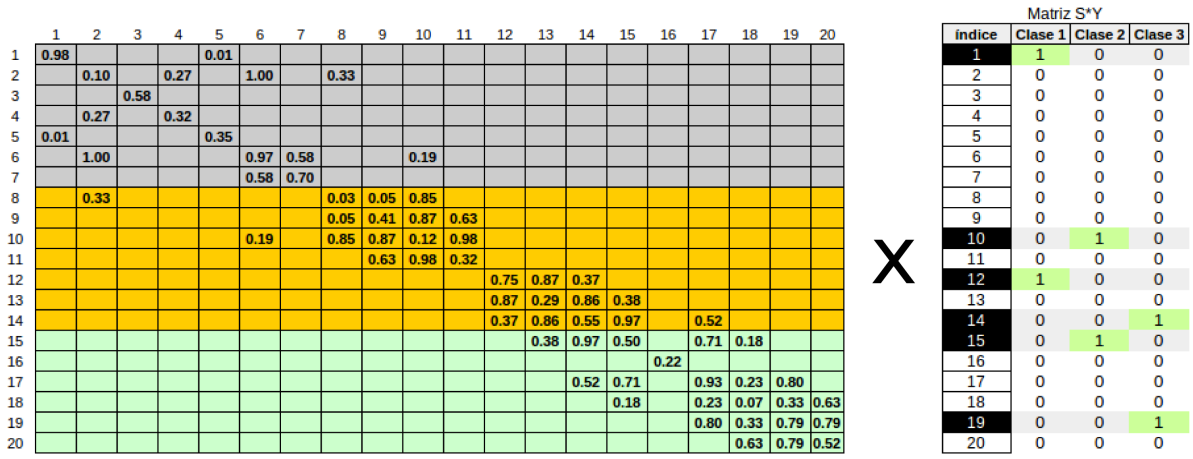


Figura A.10: Multiplicación de la matriz MI y la matriz SY de manera distribuida.

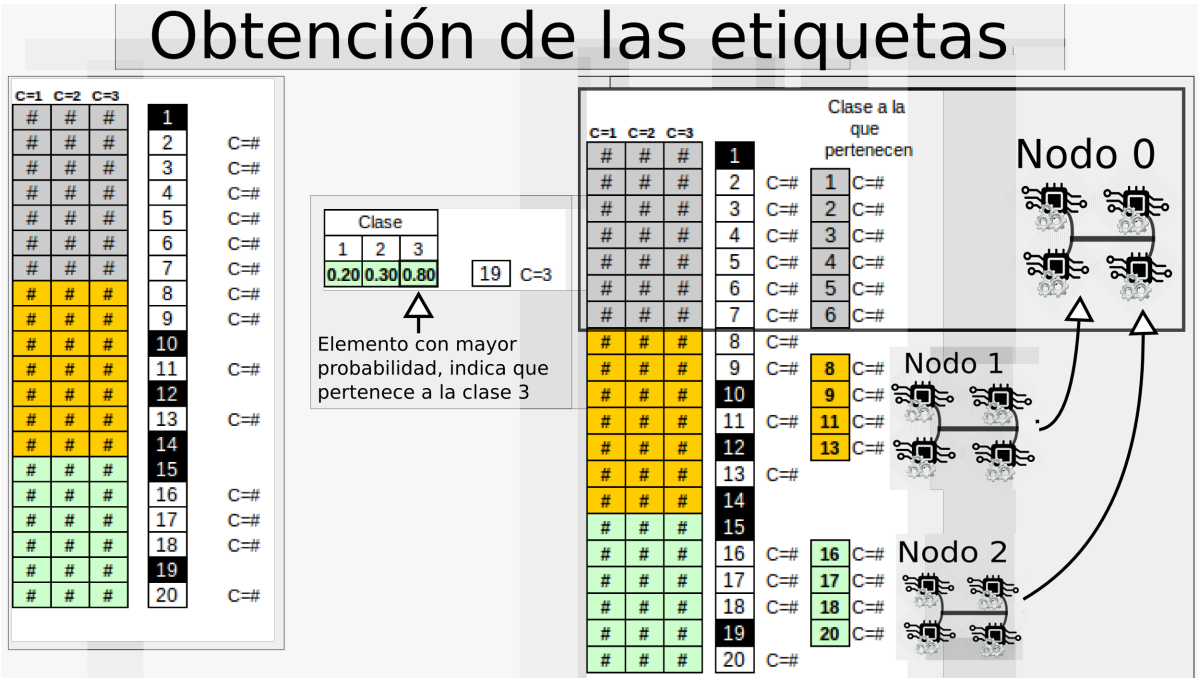


Figura A.11: Obtención de la etiqueta para los ejemplos no etiquetados.

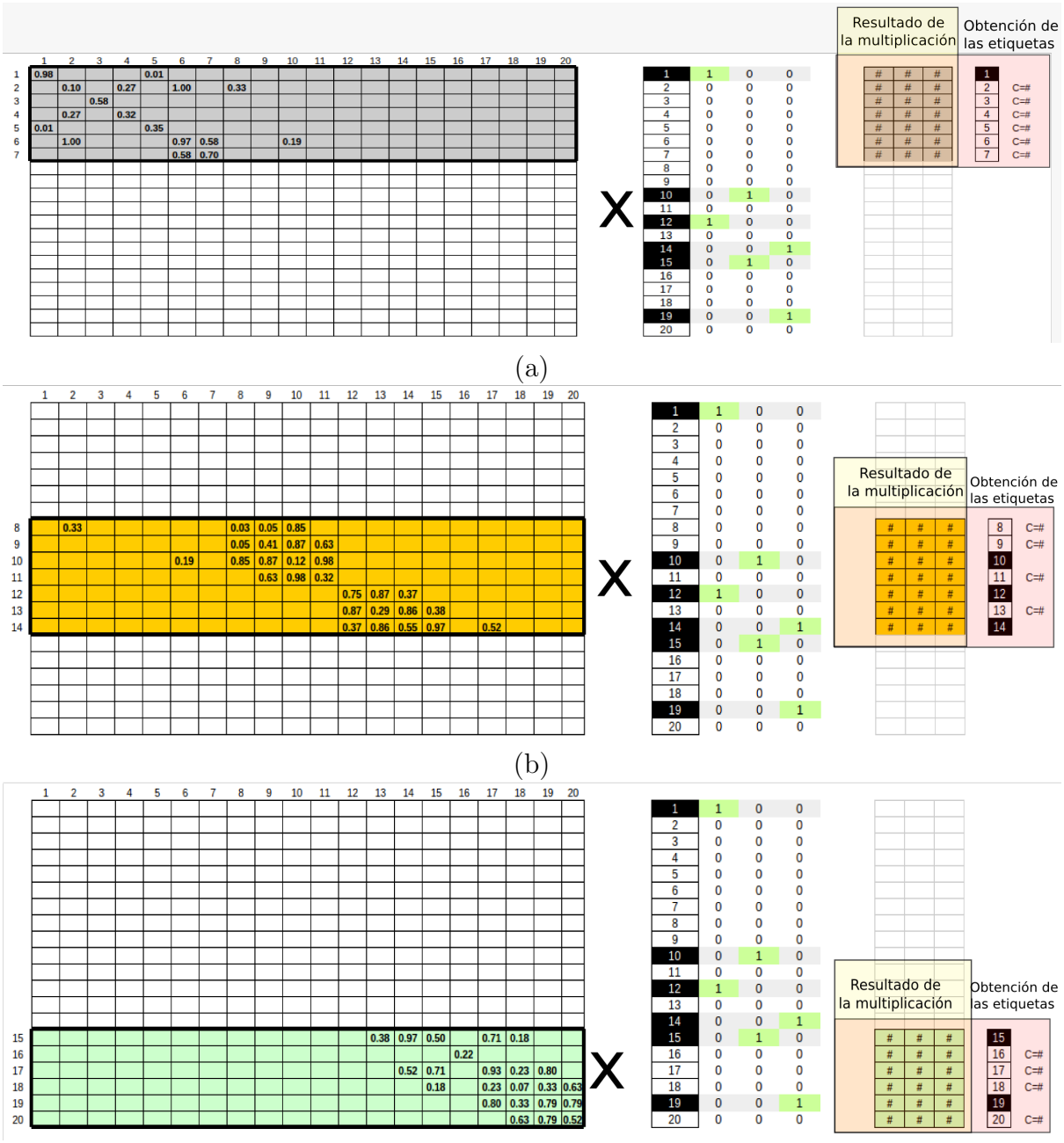


Figura A.12: Multiplicación distribuida de $MI \times SY$ en 3 nodos.

Anexo B

Instalación de un *Cluster Beowulf*

En este anexo se muestra el proceso de instalación de los paquetes necesarios para compilar aplicaciones paralelas y distribuidas usando la biblioteca de paso de mensajes MPI. Esta configuración es conocida como *Cluster Beowulf* y permite acceder desde cualquier nodo en una red definida por la configuración hacia cualquier otro nodo para enviar datos y asignar la ejecución de secciones de las aplicaciones paralelas de manera distribuida.

B.1. Instalacion de la biblioteca de paso de mensajes

Para instalar paquetes se debe ingresar como *superusuario* (*root*). El ingreso al sistema es comúnmente realizado usando un usuario administrador, para ingresar con permisos de *superusuario* a la cuenta de *root* se usa el comando **sudo su**.

El primer paso es la instalación de los paquetes que se requieren, para ello es necesaria la actualización del repositorio. La orden **apt-get** [von Hagen, 2010] con la opción *update* permite actualizar la lista de paquetes que pueden instalarse desde los repositorios que están en Internet:

```
apt-get update
```

La opción *update* tiene como propósito indicar al repositorio local dónde están las fuentes para los programas que se instalarán en los siguientes pasos.

Los siguientes paquetes a instalar son el compilador de C/C++ y las bibliotecas para MPI y OpenMP, la siguiente instrucción usa el comando *apt-get* y la lista de paquetes que se deben instalar para poder usar la biblioteca de paso de mensajes:

```
apt-get install g++ openmpi-bin openmpi-common libopenmpi1
  libopenmpi-dev libgomp1
```

Al momento de instalación de los paquetes, en los comandos se listan los paquetes

principales, la gran mayoría de los paquetes que se muestran a continuación tienen dependencias, es decir al instalar el paquete se instalarán otros paquetes que son necesarios para su funcionamiento, todo esto lo hace de manera automática el comando *apt-get*.

B.2. Cuentas y conexión SSH

Para poder establecer conexiones a través del protocolo SSH (*Secure SHell*, o intérprete de órdenes segura), se debe crear en todos los nodos un usuario con el mismo nombre. Este usuario será el que ejecutará en el resto de servidores el programa que solicitará el nodo maestro. Cada usuario debe tener exactamente la misma ruta para los archivos que se ejecutarán y tendrá una copia de dichos archivos. Si se supone que el usuario es "**hpc**", dicho usuario puede crearse con la siguiente línea:

```
adduser hpc
```

SSH es un protocolo que permite acceder a máquinas remotas a través de una red [von Hagen, 2010]. Éste permite acceso completo a la computadora remota mediante un intérprete de comandos y redirigir el control del sistema gráfico de la máquina remota a la máquina donde se accede.

Es necesario instalar un **servidor SSH** en cada uno de los nodos esclavos para poder permitir el servicio de conexión desde el nodo maestro al momento de ejecutar la aplicación distribuida. La siguiente instrucción permite dicha instalación.

```
apt-get install openssh-server
```

El nodo maestro debe tener un **cliente ssh** para poder solicitar el servicio de ejecución de programas en los nodos esclavos. La siguiente orden instala el cliente *ssh*:

```
apt-get install openssh-client
```

Claves de acceso desde el nodo maestro

Si se desea hacer una conexión desde el nodo maestro hacia cualquiera de los nodos esclavos, se puede hacer uso del comando **ssh** con con la siguiente sintaxis:

```
ssh login@nombreservidor
```

donde *nombreservidor* es el nombre del dominio del servidor o su IP. En este caso se solicitará el password del usuario y podrá establecer la conexión.

Si no se desea solicitar el password al hacer cualquier conexión, se pueden crear claves de conexión seguras que permitirán eliminar dicho paso. Las claves de conexión se deben

guardar en una carpeta llamada **.ssh** en el directorio $HOME de los usuarios en cada uno de los nodos.

Las claves se guardarán en el archivo **authorized_keys**, éste se lee cada vez que se desea acceder a otro nodo al momento de ejecutar el programa paralelo con memoria distribuida desde el nodo maestro. El comando

```
ssh_keygen -t dsa
```

pedirá un password con que se generará los archivos `id_dsa.pub` e **id_dsa** que contendrán el password encriptado para hacer las conexiones.

La figura B.1 muestra la ejecución de la instrucción **ssh-keygen**, note que se solicita el nombre del archivo en que se guardará la clave privada. Al ejecutarse, se crean dos archivos **id_dsa** e **id_dsa.pub** que corresponden a la clave privada y pública, respectivamente.

```
hpc@nodo01$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/hpc/.ssh/id_dsa): id_dsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your  identification has been saved in authorized_keys.
Your public key has been saved in authorized_keys.pub.
The key fingerprint is:
96:f1:85:a5:4f:fd:92:85:2f:54:df:e3:52:bc:ae:09

The key's randomart image is:
+--[ DSA 1024]----+
|             .   .|
|           + ..oo|
|        . o o +++|
|         + + .o=o|
|        S . ..+oo|
|       .      oo |
|           E   . |
|             . o  |
|              o   |
+-----------------+
```

Figura B.1: Creación de las llaves de acceso para el protocolo SSH.

El archivo **id_dsa.pub** deberá copiarse en cada nodo esclavo con el nombre **authorized_keys** en la carpeta **.ssh** dentro de su carpeta (**$HOME/home/hpc**, en el caso de nuestra instalación). Para crear una copia del archivo **id_dsa.pub** con un nuevo nombre puede usar el siguiente comando estando en la carpeta **$HOME/.ssh**:

```
cp id_dsa.pub authorized_keys
```

El comando **scp** (*Secure CoPy*) permite copiar un conjunto de archivos a otras equipos, se requiere el nombre del usuario, la IP y la ruta absoluta en que se desea acceder en el nodo externo. En este caso para copiar el archivo **authotized_keys** al resto de nodos se puede usar la siguiente instrucción:

```
scp authorized_keys hpc@192.168.11.106:/home/hpc/.ssh
```

Archivo de configuración MPI

Al momento de ejecutar un programa distribuido se puede usar la interfaz de paso de mensajes (*MPI*, Message Passing Interface). MPI requiere un archivo con el nombre **.mpi_hostfile** en la carpeta $HOME del nodo maestro, este archivo contiene información sobre cuántas unidades de ejecución existen en cada nodo (maestro y esclavos) del *cluster*. Para poder en su momento conectarse y crear hilos de ejecución en cada uno de los nodos. En este archivo de indica cada una de las IPs de los nodos que tienen unidades de procesamiento y cuántas hay en cada uno. En este caso nuestro archivo de configuración se muestra en la figura B.2:

```
#hostfile for mpi
#master node slots=8
192.168.11.104 slots=8

#slave node slots=8
192.168.11.105 slots=8

#slave node slots=8
192.168.11.106 slots=8
```

Figura B.2: Archivo de configuración `.mpi_hostfile`.

Los comentarios inician con el caracter #. Las IPs que hay configuradas son 192.168.11.10x en cada uno de los 3 nodos. Es posible indicar más o menos unidades de procesamiento, en caso de ser más, MPI asignará más de un proceso a las unidades de ejecución de manera automática. También con este archivo es posible aumentar más nodos al clúster, siempre que cumplan con los requerimientos planteados en esta sección.

La figura B.3 resume el proceso de instalación y configuración del software para un clúster Beowulf.

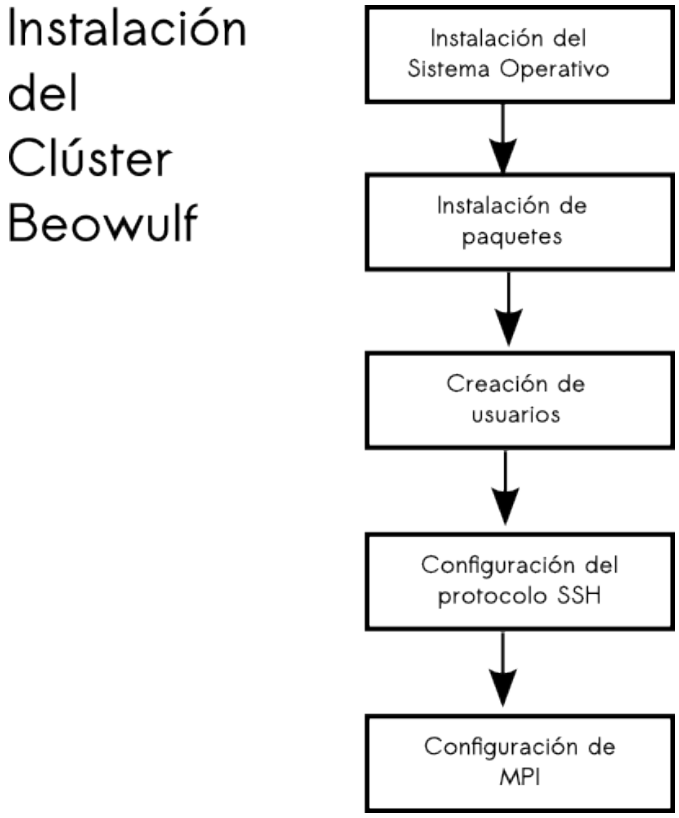


Figura B.3: Configuración del Clúster Beowulf.

Anexo C

Manual de usuario de la biblioteca

El software desarrollado para este trabajo de tesis se implementó en lenguaje C y C++, con el compilador g++ 4.8.4 y las bibliotecas de paso de mensajes (MPI) y de multiprocesamiento (OpenMP), instaladas según se indicó en el anexo B.

En primer lugar, se muestra cómo ejecutar el programa usando el código ya compilado, y enseguida los cambios que se deben realizar para poder compilar las aplicaciones para ser usadas con otros conjuntos de datos.

Ejecución del algoritmo LP

El archivo ejecutable **ejecutableLP** recibe como parámetros los siguientes parámetros:

```
./ejecutableLP nRuns <#ejecs> hm <#ejemplos etiq. x clase> numExec <#prueba ejec.>
```

Donde:

- **nRuns** es el número de veces que se ejecutará el algoritmo, a partir del cual se generará una matriz de confusión promedio.
- **hm** es el número de ejemplos etiquetados por cada clase. Por ejemplo, si el conjunto de datos tiene 2 clases y se indican 2 ejemplos, en total habrían 4 ejemplos con etiqueta y al resto se eliminaría la etiqueta.
- **numExec** el número de ejecución de la prueba. Este parámetro sirve para crear un archivo .tex en la carpeta **out** con el número de dicha ejecución que contendrá en dicho formato los resultados obtenidos en la prueba.

Un ejemplo de salida, para la ejecución del algoritmo para el conjunto COIL es el siguiente:

```
Sigma = 1.408
Porcentaje 1.0\%

Valores de las etiquetas

Matriz de confusion promedio
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
& Clase 0& Clase 1& Clase 2& Clase 3& Clase 4& Clase 5\\
Clase 0 &175.7  & 19.9  & 33.7  & 7.3   & 9.4   & 4.0   \\
Clase 1 &3.8    & 219.1         & 2.3   & 9.2   & 5.1   & 10.4  \\
Clase 2 &33.6   & 3.2   & 193.0         & 7.7   & 3.4   & 9.1   \\
Clase 3 &2.6    & 7.3   & 7.6   & 213.7         & 9.7   & 9.1   \\
Clase 4 &3.6    & 4.5   & 4.3   & 2.1   & 235.6         & 0.0   \\
Clase 5 &6.5    & 12.1  & 14.2  & 11.3  & 3.3   & 202.6         \\
\hline
\end{tabular}
Aciertos 1239.76/1500
Precision 82.6507 \%

real    3m5.112s
user    9m36.813s
sys     0m1.554s
```

Ejecución del algoritmo LPQCC

El archivo ejecutable **ejecutableLPQCC** recibe como parámetros los siguientes argumentos:

```
./ejecutableLPQCC nRuns <#ejecucs> hm <#ejemplos etiq. x clase> knn <#vecinos mas cerc>
  alpha <param alpha>  epsilon <param epsilon> numExec <#prueba ejec>
```

Donde:

- **nRuns**, **hm** y **numExec** tienen la misma función que para el algoritmo LP.
- **knn** el número de vecinos con que se creará el grafo W para el algoritmo.
- **epsilon** el valor de ϵ para el algoritmo.
- **alpha** el valor de α para el algoritmo.

Para las versiones distribuidas, se debe agregar el archivo hostfile como parámetro, y el número de nodos disponibles donde se ejecutará el programa. Se usa el comando **mpirun** de la siguiente manera:

```
mpirun --hostfile archivo -np <num procesos>  ./ejecutableLPQCC  nRuns <#ejecucs>
  hm <#ejemplos etiq. x clase>  knn <#vecinos mas cerc> alpha <param \alpha>
  epsilon <param \epsilon> numExec <#prueba ejec>
```

Donde:

- **–hostfile**, indica que el siguiente valor es un archivo con información de las IPs de los nodos y número de unidades de ejecución en ellos (ver sección B.2).

- **-np**, número de nodos en que se ejecutará el programa, tomando como referencia el archivo hostfile.

Cambios en los archivos

Ambos programas usan los archivos de librerías **datasets.h** y **datasets.cpp**. Si se desea procesar un conjunto de datos diferente, se deben hacer los siguientes cambios en el archivo **datasets.h**:

```
#define C 2
#define NC 315
#define NR 8000
```

Donde:

- **C**, es el número de clases en el conjunto de datos.
- **NC**, es el número de columnas en el conjunto de datos.
- **NR**, es el número de ejemplos.

Los datos que aparecen, en particular son para el conjunto *SecStr*. Para el archivo main.cpp, en ambos programas se pueden modificar las siguientes variables globales:

```
char basePath[] = "./";
char datasetFilename[100] = {"in/secStr8k.in"};
char outFile[100] =  {"out/secStr8k.out"};
char fileLatx[] = {"out/secStr8k.tex"};
char title[]= "Label Propagation ";
char subtitle[] = "Using Quadatic Criterion";
```

- **basePath**, ruta absoluta o relativa donde están los archivos de entrada y donde se desea generar las salidas.
- **datasetFilename**, ruta relativa y archivo en donde está el conjunto de datos.
- **outFile**, ruta relativa y archivo en donde se generarán salidas en formato de texto acerca de la ejecución.
- **fileLatx**, ruta relativa y archivo latex en donde se guardarán resultados de ejecución.
- **title**, título para el documento en latex.
- **subtitle**, subtítulo para el documento en latex.

Como se menciona anteriormente, se recibe un archivo con el conjunto de datos, este archivo debe tener en cada fila **NC** datos que corresponden a valores de las características de los conjuntos de datos para cada ejemplo. Al final del archivo deben haber **NR**

etiquetas (empezando en 0, 1, etc.) para cada uno de los ejemplos. El siguiente fragmento corresponde al conjunto de datos iris, en las primeras líneas del archivo están los datos de cada ejemplo, y en las últimas líneas se muestran las etiquetas para cada ejemplo.

```
5.1 3.5 1.4 0.2
4.9 3.0 1.4 0.2
4.7 3.2 1.3 0.2
...
4.6 3.1 1.5 0.2
5.9 3.0 5.1 1.8

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ... 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 ... 2
```

Compilación y ejecución paralela y distribuida

En ambos proyectos hay un archivo llamado **compilarMPI.sh**, este archivo contiene un ejemplo de ejecución del programa que puede ser usado de referencia.

```
mpirun -np 16 -hostfile ./hostfile ./ejecutable nRuns 4 hm 40 knn 30 alpha 0.1
  epsilon 0.01 numExec 55
```

En este caso particular, el comando **mpirun** se encarga de ejecutar el programa en 16 nodos cuyas direcciones y recursos están en el archivo **./hostfile**; se harán 4 ejecuciones del programa con 40 ejemplos de cada clase usando la regla k con 10 vecinos, y los parámetros $\alpha = 0.1$ y $\epsilon = 0.01$. Los resultados se guardarían en el archivo **out001.tex**. Un fragmento de la salida en consola de la ejecución se muestra a continuación:

```
>> nRuns = 4
>> howManyByClass = 40
>> number of neighbors = 10
>> alpha = 0.100000
>> epsilon = 0.010000
>> numExec = 1
Run 0000 ...
Run 0001 ...
Run 0002 ...
Run 0003 ...

Resumen:

  3492   468
  3234   726
Num. de ejecuciones               4
Num. total de ejemplos            8000
Num. de ejemplos no etiquetados   80
Num. de aciertos promedio  4218.50
Num. de fallos promedio    3701.50
Rendimiento       53.26 %
```

El archivo *out*001.*tex*, después de ser compilado con *pdflatex* genera un archivo PDF, como se muestra en el cuadro C.1. Este cuadro resume la ejecución del algoritmo según los argumentos recibidos en el programa.

	Clase 1	Clase 2	Rend/cl
Clase 1	**3492**	468	88.19 %
Clase 2	3234	**726**	18.34 %

Cuadro C.1: Ejemplo de cuadro generado como resultado de la ejecución del algoritmo LPQCC.