UNIVERSIDAD DE CUENCA

Facultad de Ingeniería

Escuela de Ingeniería Informática

Implementación de algoritmos de inferencia causal utilizando computación paralela: PC y PC-stable

Trabajo de Titulación Previo a la Obtención del
Título de Ingeniero en Sistemas

AUTOR:

Andrés David Morocho Coronel
CI: 0104911219

DIRECTOR:

Ing. Angel Oswaldo Vázquez Patiño, MSc
CI: 0105725634

CODIRECTOR:

Ing. Esteban Patricio Samaniego Alvarado, PhD
CI: 0102052594

Cuenca - Ecuador

2018

Resumen

Existen sistemas complejos cuya dinámica es difícil de entender y de los cuales sólo se tiene información de ciertas variables. Este problema es un escenario ideal para el uso de técnicas de descubrimiento de conocimiento como el algoritmo de PC y PC-stable. En este trabajo se realizó una implementación eficiente del algoritmo de PC y PC-stable en el lenguaje de programación C, incorporando conocimiento a priori y aplicando técnicas de paralelización para sistemas con memoria compartida (openMP) y distribuida (OpenMPI). El rendimiento de la implementación se comprobó con conjuntos de datos sintéticos y reales, comparándolo con pcalg (R). Además, se reprodujo resultados de aplicar PC-stable en simulaciones de procesos de advección y difusión. Los resultados de las pruebas de rendimiento indican que nuestra implementación con datos sintéticos es hasta 30 veces más rápida que pcalg (serie) y la implementación paralela (openMP) es hasta 3.7 veces más rápida que en serie. Además, nuestra implementación (openMP) con datos reales es hasta 11 veces más rápida que pcalg (paralelo). Aunque los resultados de nuestra implementación con respecto a pcalg varían en 6 de los 1200 conjuntos de datos sintéticos probados, en hasta 22 enlaces (de 910) del DAG resultante, se considera que estas diferencias no tienen impacto en las conclusiones obtenidas de los DAGs resultantes. Al aplicar PC-stable a simulaciones de procesos se encontró que nuestra implementación es más drástica eliminando enlaces en los distintos tiempos. Aunque, los enlaces del DAG resultante tienen el mismo comportamiento que en los resultados que se replican.

Palabras clave: PC, PC-stable, Paralelización, Red Causal, Descubrimiento de Conocimiento

Abstract

There are complex systems whose dynamics are difficult to understand and of which only information of certain variables is available. The amount of information collected from those variables makes the problem an ideal scenario for using knowledge discovery techniques such as PC and PC-stable algorithms. In this work we create an efficient implementation of PC and PC-stable in the C programming language, adding background knowledge and applying parallelization techniques for systems with shared (openMP) and distributed memory (Open MPI). Performance tests of the implementation were performed with synthetic and real data sets and compared with pcalg (R). Also, we reproduced the results of applying PC-stable to simulations of advection and diffusion processes. Results of the performance tests show that our implementation with synthetic data is up to 30 times faster than pcalg (serial) and the parallel implementation (openMP) is up to 3.7 times faster than the serial implementation in C. Furthermore, our implementation (openMP) with real datasets is 11 times faster than parallel pcalg. Although the results of our implementation compared to pcalg differ in 6 of the 1200 synthetic datasets tested, in up to 22 links (of 910) of the resulting DAG. We considered that these differences have no impact on the conclusions obtained from the resultant DAGs. When applying the implementation to simulations of processes, we found that our implementation is more drastic deleting links at different times. Although, the resulting links in the DAG have the same behavior as in the replicated results.

Keywords: PC, PC-stable, Parallelization, Causal Network, Knowledge Discovery.

Contenidos

Lista de Tablas

Lista de Figuras

Lista de Abreviaciones

API	Interfaz de Programación de Aplicaciones (*Application Programming Interface*)
CSV	Valores Separados por Comas (*Comma-Separated Values*)
CPC	PC Conservativo (*Conservative-PC*)
CPDAG	Grafo Acíclico Dirigido Parcialmente Completo (*Completed Partially Directed Acyclic Graphs*)
DAG	Grafo Acíclico Dirigido (*Directed Acyclic Graph*)
FCI	Inferencia Causal Rápida (*Fast Causal Inference*)
GIES	Búsqueda Ambiciosa de Equivalencia Intervencionista (*Greedy Interventional Equivalence Search*)
GSL	Librería Científica de GNU (*GNU Scientific Library*)
hPa	Hectopascal
MB	Manto de Markov (*Markov Blanket*)
MLE	Estimación de Máxima Probabilidad (*Maximum Likelihood Estimation*)
MPI	Interfaz de Paso de Mensajes (*Message Passing Interface*)

Cláusula de Propiedad Intelectual

Andrés David Morocho Coronel, autor del trabajo de titulación "Implementación de algoritmos de inferencia causal utilizando computación paralela: PC y PC-stable", certifico que todas las ideas, opiniones y contenidos expuestos en la presente investigación son de exclusiva responsabilidad de su autor.

Cuenca, 7 de mayo de 2018

Andrés David Morocho Coronel

C.I: 0104911219

Cláusula de licencia y autorización para publicación en el Repositorio Institucional

Andrés David Morocho Coronel en calidad de autor y titular de los derechos morales y patrimoniales del trabajo de titulación "Implementación de algoritmos de inferencia causal utilizando computación paralela: PC y PC-stable", de conformidad con el Art. 114 del CÓDIGO ORGÁNICO DE LA ECONOMÍA SOCIAL DE LOS CONOCIMIENTOS, CREATIVIDAD E INNOVACIÓN reconozco a favor de la Universidad de Cuenca una licencia gratuita, intransferible y no exclusiva para el uso no comercial de la obra, con fines estrictamente académicos.

Asimismo, autorizo a la Universidad de Cuenca para que realice la publicación de este trabajo de titulación en el repositorio institucional, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Cuenca, 7 de mayo de 2018

Andrés David Morocho Coronel

C.I: 0104911219

Agradecimientos

Al departamento de Recursos Hídricos y Ciencias Ambientales (iDRHICA) por permitir el uso de su servidor para realizar pruebas de rendimiento del algoritmo implementado.

Al departamento de Ciencias de la Computación por facilitar el espacio físico utilizado por el autor durante el desarrollo de la tesis.

Al proyecto “Desarrollo de indicadores hidrológicos funcionales para la evaluación del impacto global en ecosistemas Andinos”, financiado por la Universidad de Cuenca y SENESCYT, porque el tema de la tesis surgió dentro del mismo.

Dedicatoria

Este trabajo es la culminación de años de esfuerzo propio y de las personas que me han apoyado a lo largo de mis estudios:

A mi madre Nohemí, que ha estado presente en mis éxitos y fracasos. Eres quien siempre creyó en mí aun cuando yo mismo no creía, has dedicado tu vida a la educación de tus hijos. Este es un trabajo que no lo podría haber logrado sin ti ya que eres la luz que me ha guiado durante todos estos años.

A mi padre Luis, aun estando lejos supiste estar presente en mi vida. Este es el fruto del apoyo que me has brindado durante todos estos años. Sin ti esto no hubiera sido posible y espero que muy pronto nos volvamos a encontrar.

A mis hermanos, Oswaldo y Jessica ya que este es un viaje que hemos realizado juntos. Espero seguir contando con su apoyo.

Capítulo 1: Introducción

1.1. Redes causales

Una red es un conjunto de nodos y enlaces entre ellos. Los nodos pueden representar cualquier tipo de elemento como por ejemplo una persona en una red social o una terminal aérea en una red de transporte aérea. Los enlaces entre los nodos pueden indicar cualquier tipo de relación como por ejemplo una amistad en una red social o una conexión en una red de transporte aérea. Gracias a sus propiedades las redes se utilizan para representar la organización e interacciones de sistemas simples y complejos. Los sistemas complejos se denominan así por la cantidad de elementos que interactúan, el comportamiento que generan y porque representan sistemas reales complejos (e.g. redes telefónicas, de computadoras). El comportamiento de los sistemas complejos es difícil de determinar con solo el conocimiento de los elementos que interactúan. Su estudio se hace mediante la Ciencia de las Redes y ha sido uno de los retos más importantes del siglo XXI ya que juega un rol importante en la ciencia, economía y en la vida de las personas (Barabási & Pósfai, 2016). Esta ciencia surgió a inicios del siglo 21 como consecuencia del interés generado en la creación de mapas de redes (e.g. genéticos, sociales, neuronales), de la facilidad para transmitir información por internet y del surgimiento de las herramientas necesarias para trabajar con estos mapas. El estudio de distintos tipos de redes es importante dentro de diversos campos científicos debido al nuevo conocimiento que se puede extraer de los fenómenos físicos que se pueden representar. Por ejemplo, el estudio de las redes de transporte en la expansión de virus ha permitido generar modelos de expansión de enfermedades (e.g. Ébola).

La causalidad es un tema que se trata frecuentemente en la vida real ya que las personas se interesan por los efectos de las decisiones que toman y la causa de lo que les aqueja. Los filósofos definen dos nociones de causalidad: causalidad específica y causalidad tipo o general. La causalidad específica se enfoca en eventos particulares, en cambio la causalidad tipo, realiza declaraciones generales (e.g. “fumar causa cáncer”), por lo que es en la que se interesan los científicos. La causalidad tipo se enfoca en lo que sucederá en el futuro, esto es importante para los científicos ya que les permite realizar predicciones (Halpern, 2016). El estudio de la causalidad, su modelamiento, razonamiento y simulación es un tema de estudio

que ha estado presente durante muchos años en campos como la inteligencia artificial (Treur, 2016) y en los últimos 10 años, el clima (Deng & Ebert-Uphoff, 2014; Ebert-Uphoff & Deng, 2010, 2012a, 2012b, 2014, 2015a, 2015b; Hammerling, Baker, & Ebert-Uphoff, 2015).

Una red es causal cuando los enlaces entre los nodos son dirigidos y representan relaciones causa-efecto. Matemáticamente esta estructura se define como grafo dirigido. Para el modelamiento causal se realizan abstracciones con el objetivo de evitar la complejidad temporal. Consecuencias de esto es que incluir caminos cíclicos en un grafo causal resulta dificultoso y que no se puede distinguir entre los tiempos de los efectos causales (Treur, 2016). Por este motivo en el modelamiento de redes causales se utiliza Grafos Acíclicos Dirigidos (DAG). A una red causal se la conoce como DAG, también porque que representa la historia de la evolución de un sistema de sustitución[1] (Bayesialab, 2018; Weisstein, 2018a). En una red causal, los enlaces entre los nodos se tratan como relaciones familiares, así, si un nodo A tiene un enlace hacia B, se dice que B es hijo de A. Esto permite otra concepción de red causal que la define como una red bayesiana en la que todo nodo padre es el responsable directo de sus hijos. Debido a que las relaciones entre los nodos están bien definidas, se dice que una red causal modela un entorno como una colección de mecanismos de componentes estables (“Bayesian Networks”, 2018). Esta característica permite observar cómo el cambio de estado de una nodo afecta las nodos de su vecindario (Nielsen & Jensen, 2009).

1.1.1 Redes climáticas

El clima se define en términos de la variabilidad y la media de variables atmosféricas relevantes (e.g. temperatura, precipitación, viento) de sus 5 componentes: atmosfera, hidrosfera, criosfera, la superficie, la biosfera (Goosse, 2015). El estudio del clima es complejo ya que no se puede realizar experimentación directa y puede ser influenciado por muchos factores (e.g. latitud y longitud). Un sistema climático se puede estudiar aplicando conceptos de redes siempre que se asuma que los sistemas dinámicos de interacción pueden formar una red. Al representar un sistema climático como una red se asume que el clima es representado por una

[1] Es un mapa que usa un conjunto de reglas para transformar elementos de una secuencia en una nueva secuencia. Las reglas traducen los elementos de la secuencia original a su transformación (Weisstein, 2018b)

cuadrícula de osciladores de ciclo límite[2] que representa un sistema dinámico que varía en una forma compleja (Anastasios A. Tsonis, Swanson, & Roebber, 2006). Lo que interesa de las redes climáticas es su estructura final y su dinámica; una vez que se ha formado la red, sus propiedades locales y globales pueden ser estudiadas (e.g. número de conexiones) para ganar conocimiento del fenómeno particular estudiado.

En una red climática un nodo representa una serie de tiempo de una variable específica (e.g. temperatura) en un punto geográfico particular. Existen dos enfoques principales para formar los enlaces en una red climática: 1) utilizando la correlación entre las series de tiempo y 2) utilizando descubrimiento causal. A.A. Tsonis & Roebber (2004) introdujeron el concepto de redes climáticas utilizando la correlación entre las series de tiempo para determinar los enlaces en la red (red de correlación). Los resultados que A.A. Tsonis & Roebber (2004) encontraron indican que un sistema climático actúa como una red estable[3] debido a sus conexiones de alto alcance, y tiene propiedades que permiten que la información (fluctuaciones) se transporte eficientemente. Su simplicidad y la moderada capacidad computacional necesaria llevó a que este primer enfoque sea muy utilizado desde entonces en estudios de clima.

Por otro lado, el descubrimiento causal es una forma más directa de encontrar relaciones y construir redes con pocos enlaces. Su objetivo es encontrar la mayor información posible del sistema a partir del conjunto de datos provisto, para lo cual se utilizan técnicas de razonamiento causal (e.g. aprendizaje estructurado) que derivan en redes climáticas que representan mejor las conexiones causales. Ya que la correlación entre dos nodos de una red no implica causalidad, es necesario de otras pruebas (e.g. pruebas de independencia condicional) para determinar sus relaciones. El objetivo de estas pruebas es inferir la máxima información posible acerca de las conexiones causales de los nodos (Ebert-Uphoff & Deng, 2010). Sin embargo, aunque conceptualmente el enfoque sea sencillo y ha dado buenos resultados, se necesita de una gran capacidad computacional para estudiar redes que tengan un gran número de nodos (>5000) y que son comunes dentro de

[2] Algunos sistemas complejos (e.g. clima) pueden ser entendidos como un conjunto grande de elementos simples como los osciladores que tienen diferentes interacciones entre sí (Kuznetsov, Pikovsky, & Rosenblum, 2010).

[3] Una red se considera estable si al eliminar parte de sus enlaces no se fragmenta en partes inconexas (A.A. Tsonis & Roebber, 2004).

climatología. Esto se debe principalmente a la gran cantidad de pruebas de independencia que se deben realizar en el proceso de inferencia de la red. Este problema de alta demanda computacional ha sido la motivación principal de este trabajo.

Los conceptos de aprendizaje estructurado, descubrimiento causal y pruebas de independencia condicional se describen en secciones posteriores en el documento ya que son temas esenciales para el desarrollo de este trabajo.

1.2. Modelos de inferencia causal

Dentro de la lógica existen dos tipos de razonamiento, deductivo e inductivo, que se diferencian debido sus puntos de partida. El razonamiento deductivo parte de premisas generales y obtiene conclusiones específicas, en cambio el razonamiento inductivo parte de un conjunto de experiencias y obtiene una conclusión general (Walliman, 2011). Existen varios tipos de razonamiento inductivo entre las que se encuentran la inferencia categórica, inferencia analógica e inferencia causal (Sofroniou, 2017). Shaughnessy, Zechmeister, & Zechmeister (2014) definen la inferencia causal como la identificación de las causas de un fenómeno siempre que se cumplan 3 condiciones: la covariación de los eventos, una relación de tiempo-orden y la eliminación de causas alternativas plausibles. Con la inferencia causal se llega a la conclusión de qué evento tiene mayor probabilidad de ser la causa de un efecto observado. Por ejemplo, al escuchar el croar de una rana se puede inferir que hay una rana cerca pero también es probable que se una grabación.

La inferencia causal es uno de los objetivos centrales de varias investigaciones empíricas en ciertos campos de la ciencia (e.g. medicina, epidemiología) donde se ven obligados a usar solo información obtenida de observaciones (i.e. conjuntos de datos con muestras de variables) debido a que la experimentación directa no es factible por cuestiones de ética, presupuesto o prácticas (Cousens, Diaz-Ordaz, Silverwood, Keogh, & Vansteelandt, 2018). El proceso de inferir una estructura causal (e.g. red bayesiana) a partir de un conjunto de datos y conocimiento a priori se denomina aprendizaje (Scutari, 2014). Las operaciones que se realizan sobre el conjunto de datos depende del método de aprendizaje que se utilice, que principalmente son: 1) aprendizaje de estructura y 2) aprendizaje de parámetros. El aprendizaje de parámetros parte de un conjunto de datos y de un DAG (conocido o recomendado por el experto del tema) y estima los parámetros de una distribución

global estimando los de distribuciones locales (Scutari, 2014). En cambio, el aprendizaje de estructura se encarga de encontrar un DAG que codifique las independencias condicionales presentes en un conjunto de datos (Scutari, 2014). Además, involucra el aprendizaje de las relaciones causales entre variables lo que permite ordenar eventos que ocurren secuencialmente. Cuenta con dos enfoques de aprendizaje: basado en puntajes y basado en restricciones (Friedman & Goldszmidt, 1998).

El aprendizaje basado en puntajes es una representación de técnicas de optimización heurística donde cada DAG es evaluado con un puntaje (Scutari, 2014). Para el aprendizaje se define un conjunto de DAGs candidatos, un conjunto de operaciones para crear nuevos DAGs (agregar, remover o invertir enlaces) y una función para puntuarlos en base a los datos (De Jongh, 2014). La búsqueda de un DAG con un puntaje óptimo se puede hacer con búsqueda exacta y búsqueda ambiciosa. La búsqueda exacta considera todos los DAGs que se pueden crear a partir del conjunto inicial de nodos, volviéndose más compleja computacionalmente a medida que incrementa el número de nodos (exponencial con respecto al número de nodos). Una opción más viable es la búsqueda ambiciosa, donde se genera un DAG inicial de forma aleatoria y se aplican algoritmos de optimización heurística (e.g. Tabu Search, Hill-Climbing) para encontrar el DAG óptimo. Sin embargo, en problemas reales que involucran una gran cantidad de nodos, la capacidad computacional necesaria sigue siendo alta.

El aprendizaje basado en restricciones realiza pruebas estadísticas para determinar si cada par de variables del conjunto de datos son independientes. Esto con el objetivo de obtener un DAG que explique las independencias encontradas. Las variables del conjunto de datos son los nodos del DAG. A partir de un conjunto de datos se pueden encontrar varios DAG equivalentes entre sí, sin embargo no se puede distinguir entre ellos (Koller & Friedman, 2009). Las pruebas estadísticas se realizan bajo la suposición de que la separación gráfica de los nodos, implica su independencia condicional y viceversa (Scutari, 2014). Todos los algoritmos de aprendizaje basado en restricciones tienen dos fases comunes. En la primera fase se realizan pruebas de independencia condicional para determinar si un par de nodos son independientes, si este es el caso se elimina su enlace. Esta fase no toma en cuenta la dirección de los enlaces y genera una estructura inicial del DAG con enlaces bidireccionales denominada *skeleton*. En la segunda fase se realiza la orientación de todos los enlaces posibles verificando que no se generen

inconsistencias (e.g. ciclos). El resultado de este paso es un grafo acíclico dirigido parcialmente completo (CPDAG) que contiene enlaces dirigidos y bidireccionales y es equivalente a múltiples DAG (Scutari, 2014).

Para este trabajo nos centramos en el aprendizaje de estructura y en particular en el basado en restricciones, cuyo algoritmo más representativo es el algoritmo de PC[4] (Spirtes, Glymour, & Scheines, 1993). El algoritmo de PC cuenta con algunas variaciones y puede ser aplicado a cualquier fuente siempre que se pueda verificar la independencia condicional entre dos variables (Abellán, Gómez-Olmedo, & Moral, 2006). Una de las características del aprendizaje basado en restricciones es que realiza decisiones categóricas desde su primera fase. Esto significa que un error en una de las pruebas estadísticas provoca que la estructura obtenida sea diferente. Este error condiciona toda la ejecución del algoritmo ya que cambia la organización de los nodos. Aun con esta desventaja, el aprendizaje basado en restricciones con el algoritmo de PC tiene una base intuitiva que en las condiciones ideales garantiza la recuperación de un grafo equivalente al modelo verdadero. Además, se considera que realiza una selección inteligente y ordenada de las preguntas para recuperar una estructura causal (Abellán et al., 2006). Finalmente, y muy importante, es un método novedoso para inferir redes climáticas y ha demostrado derivar nuevas hipótesis o conocimiento adicional de hipótesis de fenómenos geofísicos (su potencial aún no ha sido totalmente explotado en otras áreas de la ciencia).

1.2.1 Complejidad computacional

La complejidad computacional estudia el orden de complejidad de un algoritmo de acuerdo al número de pasos que necesita para completarse y la cantidad de recursos (e.g. CPU, memoria RAM) que utiliza. El número de pasos se mide en el peor de los casos, cuando el algoritmo tiene que realizar la mayor cantidad de pasos posibles (Hall, 2013). Un paso es una operación entre dos números: suma, resta, multiplicación, división o comparación. El número de pasos que debe realizar un algoritmo de aprendizaje de estructura depende del número de variables que tenga el conjunto de datos. El número de variables se traduce en el número de nodos que tendrá el grafo o estructura equivalente que genera el algoritmo. El número de respuestas posibles de un algoritmo de aprendizaje de estructura es igual al número

[4] Llamado así por sus creadores Peter y Clark.

de combinaciones posibles de los nodos del grafo. Cada combinación involucra: agregar, eliminar enlaces y definir direcciones de los enlaces. La complejidad computacional de un algoritmo de aprendizaje basado en restricciones es exponencial en relación al número de nodos. Y aunque Scutari, (2014) indique que es polinomial en casos de la vida real, en áreas como el clima, esto no se ha podido verificar. En cambio, un algoritmo de aprendizaje basado en puntajes tiene una complejidad NP-difícil (Le et al., 2016).

Los algoritmos de aprendizaje de estructura disponen de diferentes formas de reducir su complejidad computacional y por consecuencia su tiempo de ejecución. En el aprendizaje basado en puntajes se utilizan técnicas de optimización heurística (e.g. Hill Climbing, Tabu Search) para no considerar todo el espacio de posibles DAGs. Utiliza las respuestas obtenidas en el pasado para considerar las opciones disponibles y reduce el número de puntajes calculados limitando la computación a un grupo pequeño de variables. Otra forma de reducir el tiempo de ejecución es utilizar computación en paralelo (Scutari, 2014). En cambio, el aprendizaje basado en restricciones tiene menos opciones para reducir su tiempo de ejecución. En su primera fase se puede reducir el número de pruebas de independencia condicional al evitar las redundantes y aplicar paralelización en ciertas partes del algoritmo. Sin embargo, aun con estas optimizaciones, el número de pasos a realizar todavía es muy grande. Esto también se debe a la cantidad de variables con las se trabaja en algunos campos de estudio. Por ejemplo, en biología existen conjuntos de datos de expresiones de genes que tienen hasta 5000 variables (considerándose estas como muy grandes). En cambio, en climatología un experimento puede tener más de 200000 variables.

Todo esto pone en relevancia la alta capacidad de computación que se necesita para realizar investigación utilizando modelos de inferencia causal. Y, por otro lado, revela la importancia de estudiar los algoritmos desde el punto de vista computacional para llegar a implementaciones altamente eficientes. Cuestiones como escoger un lenguaje de programación que optimice al máximo la computación y analizar los puntos críticos de los algoritmos para poderlos paralelizar son sólo un par de consideraciones que se deben realizar.

1.3. Necesidad de estudio

Existen sistemas complejos (e.g. clima, genes) y de cuya dinámica sólo se tienen observaciones de ciertas variables. De algunas de estas variables se recolectan series de tiempo para formar conjuntos de datos. La cantidad de información que se recolecta es muy grande por lo que realizar un proceso manual para obtener conocimiento es inviable. Por ejemplo, dentro del campo climatológico desde el año 1980 la cantidad de datos que se recolecta ha tenido un incremento casi exponencial (Kenward, 2011). Además, en tales sistemas no es viable realizar experimentación debido a razones éticas, presupuestarias o de escala. Por estos motivos, encontrar las relaciones causales entre estas variables ha sido de mucha ayuda y es de gran interés en varios campos de la ciencia (e.g. climatología y biología). Con esta gran cantidad de datos recolectados de diversas variables (e.g. atmosféricas, oceánicas y de terreno) se ha hecho necesario la utilización de técnicas automáticas de descubrimiento de nuevo conocimiento. Entre estas técnicas de aprendizaje está el algoritmo de PC. Sin embargo, encontrar una estructura causal (DAG) que represente las independencias presentes en un conjunto de datos es un proceso altamente complejo computacionalmente, ya que el número de posibles DAGs es exponencial con respecto al número de variables del conjunto de datos.

El algoritmo de PC cuenta con varias implementaciones en diferentes lenguajes de programación (e.g. C, Java, MATLAB, R). Las implementaciones cubren los modelos de ejecución en serie y en paralelo e incluyen variaciones que robustecen el algoritmo al eliminar su dependencia del orden en el que están dispuestas las variables en un conjunto de datos.

Las implementaciones en serie del algoritmo entre otras son: TETRAD[5] y pcalg[6] (Kalisch et al., 2014). TETRAD implementa distintas versiones del algoritmo de PC y ofrece una gran variedad de opciones a configurar mediante su interfaz gráfica. Entre las opciones a configurar están: el tipo de prueba estadística, máximo tamaño del conjunto de condicionamiento y conocimiento a priori. Sin embargo, está escrito en el lenguaje de programación Java y utiliza el paradigma de programación orientado a objetos por lo que la implementación no es eficiente con respecto a su

[5] http://www.phil.cmu.edu/tetrad/
[6] https://cran.r-project.org/web/packages/pcalg/index.html

tiempo de ejecución y uso de recursos. Esto debido a que se ha desarrollado con el objetivo de demostrar cómo funcionan los distintos algoritmos que implementa y no de desarrollar una herramienta para procesar grandes cantidades de datos. Pcalg, en cambio, está implementado en el lenguaje de programación R, un lenguaje orientado a la computación estadística. Implementa distintas versiones del algoritmo, además, permite modificar su ejecución al incluir conocimiento a priori, cambiar la prueba estadística y el valor significancia, entre otros parámetros. Su uso para grandes conjuntos de datos (más de 3000 variables) no es factible debido al tiempo necesario para completar su ejecución. En pruebas realizadas con un conjunto de datos sintético de 1500 variables, el tiempo de ejecución excede las 24 horas. Así, el tiempo de ejecución solo incrementará, aún más drásticamente, con conjuntos de datos de más variables. Además, pcalg no cubre el total de posibilidades en cuanto al conocimiento a priori que se puede dar antes de la ejecución del algoritmo, e.g. restringir solamente la dirección de ciertos enlaces.

El algoritmo de PC en su variación estable cuenta con varias implementaciones que permiten su ejecución en paralelo. Yi & Ebert-Uphoff (2014) implementan una versión paralela del algoritmo PC-stable usando el lenguaje de programación C y la librería GSL[7]. Realizan diversas optimizaciones para el acceso a memoria y obtienen un algoritmo 300 veces más rápido que otras implementaciones en R, Java o MATLAB. El tiempo de ejecución del algoritmo en serie con un conjunto de datos de 3000 variables se reduce de 4 días a 20 minutos, según reportan Yi & Ebert-Uphoff (2014). Cuando agregan procesamiento en paralelo los cálculos del algoritmo se aceleran por un factor de 4, esta aceleración del procesamiento les permitió realizar pruebas con conjuntos de datos de más de 36000 variables. Sin embargo, estos resultados no se pudieron verificar ya que no se cuenta con acceso al código fuente o a un ejecutable de algoritmo, además el contacto con el autor no tuvo ninguna respuesta[8]. Le et al. (2016) presenta una implementación paralela de pcalg en R. La paralelización se realizó únicamente sobre el proceso de obtención de la estructura del DAG (*skeleton*). Según sus resultados el tiempo de ejecución

[7] https://www.gnu.org/software/gsl/

[8] Se escribieron dos correos electrónicos a Imme Ebert-Uphoff: el primero, de parte de Angel Vázquez, preguntando acerca de la disponibilidad del código o de una alternativa para poderlo obtener y el segundo, de parte de David Morocho, solicitando mayor información acerca del computador que utilizaban para ejecutar el algoritmo. Ninguno de los correos fue contestado. Un tercer correo fue enviado a Yi Deng pero contestó reenviando las inquietudes a Imme Ebert-Uphoff (tampoco contestó).

mejora considerablemente, con un conjunto de datos de 5361 variables, el tiempo de ejecución de algoritmo en paralelo con 8 núcleos es aproximadamente la octava parte (76 minutos) de la ejecución en serie (608 minutos). Sin embargo, debido a la forma de paralelizar el algoritmo, el uso de memoria RAM se incrementa dramáticamente. Por ejemplo, para un conjunto de datos de 1500 variables, la ejecución del algoritmo con 5 núcleos necesita de al menos 9 GB de memoria RAM. El uso de memoria incrementa con el número de núcleos y el número de variables. Con un conjunto de datos de 2810 variables y 8 núcleos necesita al menos 36 GB de memoria (muy difícil de tener en un computador personal al que un investigador tiene fácil acceso).

Según lo expuesto, existen implementaciones del algoritmo de PC con diversas optimizaciones que reducen considerablemente el tiempo de ejecución del algoritmo. Sin embargo, las limitaciones y disponibilidad de estas implementaciones hacen que su uso no sea factible para grandes conjuntos de datos (más de 3000 variables). Además, debido a que existen diversas variantes del algoritmo de PC (e.g. estable, conservativo), estos desarrollos no siempre implementan la misma versión del algoritmo, lo que provoca que los resultados sean diferentes entre implementaciones. Para dar solución a este problema se plantea la implementación eficiente del algoritmo de PC en su variación estable y conservativa, usando el lenguaje de programación C con librerías libres de cálculo científico (e.g. GSL). La implementación incorpora conocimiento a priori y procesamiento en paralelo. Se usa la librería openMP[9] para implementar el procesamiento en paralelo con memoria compartida y Open MPI[10] para el procesamiento con memoria distribuida. La implementación se basa en pcalg, una implementación que se ha probado durante años y cuenta con un desarrollo activo. La característica fundamental que permitió utilizar pcalg como base, es que su código fuente está disponible y su distribución se realiza bajo la licencia GPL V2.0[11].

1.4. Objetivos

A continuación, se describen los objetivos del proyecto:

[9] http://www.openmp.org/
[10] https://www.open-mpi.org/
[11] GNU *General Public License*, https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html

1.4.1 Objetivo general

Realizar una implementación computacionalmente óptima de los algoritmos PC y PC-stable para que puedan ser ejecutados en paralelo y comprobar su eficiencia respecto a otras implementaciones existentes.

1.4.2 Objetivos específicos

1. Estudiar las redes causales y entender el funcionamiento de los modelos de inferencia causal PC y PC-stable.
2. Implementar el algoritmo de PC y PC-stable en el lenguaje de programación C.
3. Incorporar conocimiento a priori en los algoritmos.
4. Mejorar el rendimiento de la implementación de los algoritmos utilizando computación paralela.
5. Realizar pruebas de rendimiento de la implementación con datos sintéticos y reales.

1.5 Organización del documento

El resto del documento se organiza de la siguiente manera:

- Capítulo 2 presenta las bases teóricas del algoritmo de PC, sus variaciones (PC-stable, CPC) y las oportunidades para paralelizar el algoritmo.
- Capítulo 3 describe la implementación en C del algoritmo de PC, PC-stable y CPC, además, de su paralelización con memoria compartida y distribuida.
- Capítulo 4 describe los datos sintéticos y reales usados en las pruebas de rendimiento y plantea 6 conjuntos de pruebas. Las primeras 5 pruebas para comprobar el rendimiento de la implementación en C (serie y paralelo) con respecto a pcalg en R (serie y paralelo). La última prueba se realiza para comprobar el comportamiento de la implementación con respecto a simulaciones de procesos de advección y difusión.
- Capítulo 5 presenta y argumenta los resultados de los 6 conjuntos de pruebas. Verifica el rendimiento, escalabilidad y comportamiento de la implementación con respecto a resultados de referencia.
- Capítulo 6 presenta las conclusiones del trabajo.

Capítulo 2: Marco Teórico

2.1 Notación, definiciones y suposiciones

Un grafo *G = (V, E)* está compuesto de un conjunto de nodos *V* y enlaces *E*. Dentro de los algoritmos de aprendizaje, los nodos representan variables del conjunto de datos de entrada por lo que el término nodo y variable hacen referencia a lo mismo. Dos nodos *X, Y*, son adyacentes si tienen un enlace que les conecta (X─Y) y los nodos adyacentes a un nodo *X* en el grafo G se denomina *adj(X, G).* Un enlace es dirigido si existe una dirección hacia uno de los nodos. Si el enlace *X* se dirige hacia *Y* (*X* → *Y* o *Y* ← *X)*, se dice que *Y* desciende de *X* o que *X* es el padre de *Y.* Se considera Grafo Acíclico Dirigido (DAG) al que tiene sus enlaces dirigidos y no contiene ciclos dirigidos. Un DAG tiene un orden topológico único si existe un camino dirigido que contenga a todos los nodos ("Acyclic Graph & Directed Acyclic Graph", 2016). Un DAG es completo cuando cada par de nodos es adyacente. El *skeleton* de un DAG es el grafo resultante luego de ignorar las direcciones de los enlaces (la Figura 2 muestra un DAG y su *skeleton*). Una *v-structure* dentro de un DAG es una tripleta de nodos *(X, Y, Z)* donde *X* → *Y* ← *Z* y no existe un enlace entre *X* y *Z*. Dos DAG son equivalentes si tienen el mismo *skeleton* y las mismas *v-structures*. Dentro de un grafo, el manto de Markov o *Markov Blanket* de un nodo *X, MB(X)*, está formado por todos los nodos que contienen información del nodo *X* y no se pueden obtener de otros nodos. Esto abarca a sus padres, hijos y padres de sus hijos (Figura 1). Esta información es la única que se necesita para predecir el comportamiento del nodo (Tan & Liu, 2013).

Un algoritmo de aprendizaje de estructura (e.g. PC) no encuentra el DAG exacto que representa las independencias condicionales presentes en un conjunto de datos. Por lo que su resultado es un DAG que contiene enlaces bidireccionales. Esto se debe a que las independencias condicionales de un conjunto de datos se pueden representar por varios DAG, donde cada DAG es una posible alternativa que describe las mismas relaciones de independencia condicional (Emmert-Streib & Dehmer, 2008). Estos DAG son equivalentes si tienen el mismo *skeleton* y *v-structures*, aunque pueden diferir en la dirección de algunos de sus enlaces. El grupo de DAGs equivalentes se representa unívocamente por un Grafo Acíclico Parcialmente Dirigido Completo CPDAG y es esta la denominación del resultado de

un algoritmo de aprendizaje de estructura. Un CPDAG contiene el mismo *skeleton* que los DAG equivalentes, enlaces dirigidos obligados y reversibles. Un enlace dirigido se denomina obligado si se encuentra en todos los DAG equivalentes, caso contrario se denomina reversible. Un enlace dentro de un CPDAG es dirigido si forma parte de una *v-structure*, como consecuencia de otras *v-structures* o para evitar la creación de ciclos (Acid & de Campos, 2011). La Figura 2 muestra que lo que diferencia a un DAG y un CPDAG es la inclusión de enlaces no dirigidos *(Y—W)* en el CPDAG. Dado un grafo causal (e.g. DAG), uno de sus nodos se denomina *collider* si es el efecto común de dos nodos. En la Figura 2 el nodo *X* es un *collider* ya que es el efecto común de *Z* y *Y* (*Z*→*X*←*Y*).

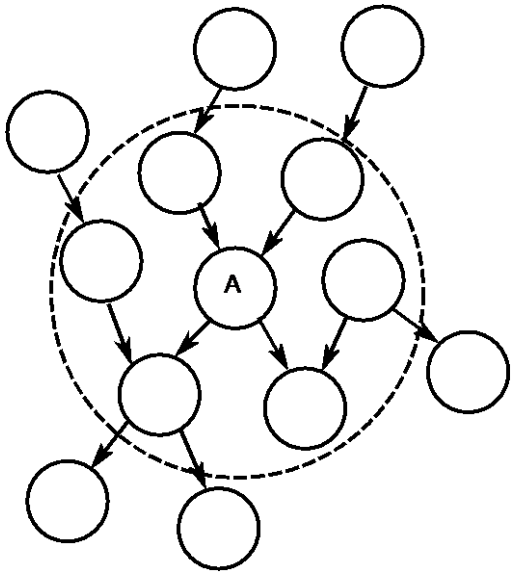


Figura 1: Manto de Markov (Markov Blanket) de un nodo (A) en un grafo.

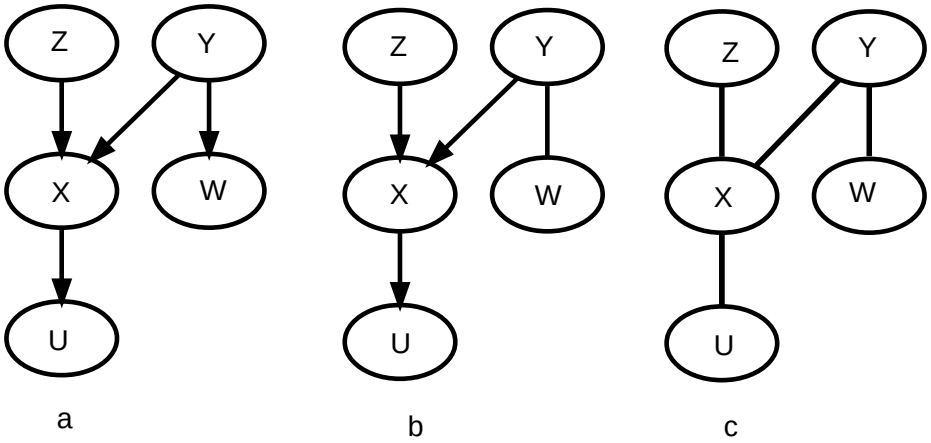


Figura 2: Diferencia entre a) DAG, b) CPDAG y c) *skeleton*.

Una estructura causal (DAG) está compuesta de nodos que están conectados entre sí (existe un camino entre los nodos) y nodos que están separados (no existe un camino entre los nodos). Dado los conjuntos *X*, *Y* y *Z,* formados por los nodos de un DAG *G* se utiliza el término *d-separation* como criterio para decidir si los conjuntos *X* y *Y* son independientes dado el conjunto Z. Esto se basa en la idea de asociar la dependencia de nodos con la existencia de un camino entre los nodos (*d-connected*) y por consecuencia la independencia con la falta de conexión (*d-separated*). Debido a que un DAG contiene enlaces dirigidos, la "*d*" significa "direccional". En la forma más básica, dos nodos están separados si no tienen un enlace que les conecte. En cambio, dos conjuntos de nodos *X* y *Y* están separados si cada nodo de *X* está separado de los nodos de *Y* (Pearl, 2009).

Para determinar si los nodos *X* y *Y* están *d-separated* o *d-connected* se considera tres reglas (Pearl, 2009):

1. **Separación incondicional:** los nodos *X* y *Y* están *d-connected* si existe un camino libre entre los nodos, un camino que se puede recorrer siguiendo la dirección de los enlaces. Caso contrario los nodos están *d-separated*.
2. **Bloqueo por condicionamiento:** al medir las variables Z de un sistema, la dinámica condicional de las demás variables cambia. Para expresar esta dinámica en un DAG se utiliza la dependencia (*d-connected*) de las variables condicionando en las variables *Z*. Por lo que los nodos *X* y *Y* están *d-connected* condicionados sobre el conjunto de nodos *Z*, siempre que haya un camino libre de *colliders* entre *X* y *Y* que no recorra algún nodo de *Z*. Si no existe este camino se dice que *X* y *Y* están *d-separated* condicionados por *Z*.
3. **Condicionamiento en colliders**: si un *collider* es miembro de un conjunto de condicionamiento *Z* o tiene un nodo descendiente en *Z* se dice que no bloquea el camino que traza el *collider*.

Si se encuentra que los nodos *X* y *Y* son independientes (*d-separated*) se utiliza la notación $X \perp Y$, en cambio, sí son independientes dado un conjunto Z se utiliza $X \perp Y \mid Z$.

Un DAG *G* puede ser interpretado de forma casual y probabilística. En la interpretación causal, *G* representa una estructura causal donde sus enlaces dirigidos son causas directas entre los nodos. En la interpretación probabilística *G* representa una distribución de probabilidad *P* que cumple la propiedad de Markov y

hace referencia a una red bayesiana. La propiedad de Markov indica que cada variable es independiente de los nodos que no son sus descendientes condicionando en sus padres (Ramsey, Zhang, & Spirtes, 2012). Dado *G* y *P* al aplicar el algoritmo de PC (explicado en la siguiente sección) se asume la condición causal de Markov, la condición de fidelidad y la suficiencia causal (Le et al., 2016).

- **Condición Causal de Markov**: *G* y *P* satisfacen esta condición si y solo si, dado el conjunto de todos los padres, un nodo de *G* es probabilísticamente independiente de todos los nodos que no son sus descendientes. Esta condición enlaza la interpretación causal *G* con su interpretación probabilística, (Ramsey et al., 2012).
- **Condición de Fidelidad:** *G* y *P* satisfacen esta condición si y solo si, ninguna independencia condicional se cumple a menos que exista la Condición Causal de Markov. Esta condición se representa de forma gráfica por *d-separation*.
- **Suficiencia Causal:** por cada par de variables que tienen sus valores observados en un conjunto de datos dado, todas sus causas comunes también tienen observaciones en el conjunto de datos.

2.2. Algoritmo de PC

El algoritmo de PC es un algoritmo de aprendizaje de estructura basado en restricciones. Utiliza pruebas de independencia condicional para determinar si se elimina los enlaces entre cada par de nodos. El algoritmo se basa en que, si no existe un enlace entre dos nodos *X* y *Y,* entonces debe existir un conjunto de nodos *Z,* formado por los nodos adyacentes a *X* o *Y*, que bloquea todos los caminos entre *X* y *Y.* Esto implica que *X* y *Y* son independientes condicionados en el conjunto *Z*. El algoritmo de PC original tiene cuatro pasos (Spirtes et al., 1993):

1. Calcular la matriz de correlación y crear un grafo completo.
2. Identificar el *skeleton* usando pruebas de independencia condicional.
3. Orientar las tripletas como *v-structures.*
4. Orientar los enlaces restantes.

El primer paso del algoritmo de PC tiene como entrada un conjunto de datos con las muestras recolectadas de las variables de un sistema. Este conjunto datos usualmente se conforma como una matriz donde las columnas representan variables y las filas son las muestras de una serie de tiempo. De este conjunto de

datos se obtiene una matriz cuadrada y simétrica de correlación (n × n, n igual al número de variables del conjunto de datos). La obtención de esta matriz es importante ya que permite investigar la relación de las variables o la dependencia que existe entre ellas al momento de identificar el *skeleton*. Otro paso muy importante y necesario es la creación de un grafo completo. Este grafo se utiliza durante el algoritmo para eliminar los enlaces de nodos independientes y orientar la dirección de los enlaces restantes.

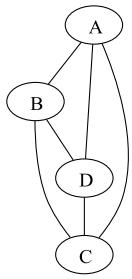
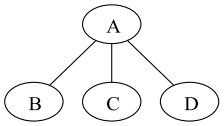
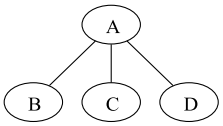
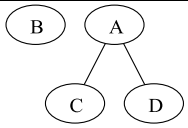
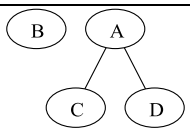
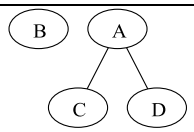
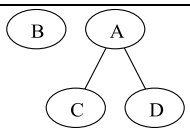
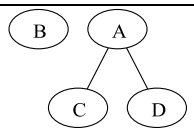
```
Entrada: conjunto de datos, alpha
Salida: skeleton del DAG

nivel = 0
repetir
   por cada enlace adyacente X-Y de G hacer:
      si (|adj(X,G)\{Y }| >= nivel) entonces:
         por cada subconjunto Z ⊆ adj(X,G)\{Y } y |Z| = nivel hacer:
            prueba IC(X,Y |Z)
            si I(X,Y |Z) entonces:
               Eliminar enlace X-Y de G
               Guardar Z en sepset(X,Y)
               salir
            fin
         fin
      fin
   fin
   nivel  = nivel + 1
hasta |adj(X,G)\{Y }| < nivel para cada enlace X-Y en G
```

Algoritmo 1: Segundo paso del algoritmo de PC original, obtención de *skeleton.*

El segundo paso del algoritmo recibe como entrada la matriz de correlación y el grafo completo *G*, el resultado es el *skeleton* del DAG. Durante este paso se actualiza dinámicamente *G* (i.e. elimina enlaces bidireccionales) de forma que el *skeleton* está contenido en *G*. La organización de este paso se observa en el Algoritmo 1. Este paso se compone de varias iteraciones, cada iteración está relacionada con un nivel que indica la cardinalidad del conjunto de condicionamiento para las pruebas de independencia condicional. Dentro de cada iteración (nivel) se recorren todos los pares de nodos adyacentes y se realizan pruebas de independencia condicional. Esto es, dado un par de nodos adyacentes *X*—*Y*, se recupera los nodos *adj(X, G)* y se utiliza combinaciones de estos para formar conjuntos de condicionamiento con una cardinalidad igual al nivel en esa iteración. Cada conjunto se utiliza en las pruebas de independencia condicional para determinar si *X*—*Y* son independientes. Si *X*—*Y* no son independientes, en una iteración posterior el proceso se repite para el par de nodos *Y*—*X* (con los nodos

adj(Y, G)). En cambio, si *X*—*Y* son independientes se elimina el enlace bidireccional entre los nodos en *G* y el conjunto de condicionamiento se almacena como conjunto de separación o *sepset.* El conjunto de separación de *X, Y* se denomina *sepset(X, Y).* El nivel de las iteraciones inicia en cero e incrementa su valor en uno en cada iteración hasta que el número de nodos adyacentes de un par de nodos sea menor al nivel actual. La Tabla 1 muestra un ejemplo del funcionamiento del algoritmo de PC. El algoritmo inicia en el nivel 0 con un conjunto de condicionamiento vacío y en cada nivel está compuesto de una serie de pruebas de independencia condicional que actualizan el grafo inicial (en el nivel 0 no hay conjunto de condicionamiento). El resultado de este paso es un *skeleton*, i.e., la estructura de un DAG sin la dirección de los enlaces.

Tabla 1: Ejemplo de funcionamiento del algoritmo de PC (Le et al., 2016).

Nivel	Grafo	# Prueba IC	Prueba	Resultado	Grafo Actualizado
0		1	¿I(A,B)?	No	
		2	¿I(A,C)?	No	
		3	¿I(A,D)?	No	
		4	¿I(B,A)?	No	
		5	¿I(B,C)?	Sí	
		6	¿I(B,D)?	Sí	
		7	¿I(C,A)?	No	
		8	¿I(C,D)?	Sí	
		9	¿I(D,A)?	No	
1		10	¿I(A,B|C)?	Sí	
		11	¿I(A,C|D)?	No	
		12	¿I(A,D|C)?	No	

El tercer paso del algoritmo consiste en orientar las tripletas que contiene el *skeleton*. Dada una tripleta con los nodos *X*—*Y*—*Z* se recupera el conjunto de separación *sepset(X, Z*). Se comprueba si el nodo *Y* está dentro de *sepset(X, Z)*, si este es el caso, la tripleta no se orienta como *v-structure*. Si *Y* no está en el conjunto de separación, la orientación se realiza en los enlaces de los nodos adyacentes

dando como resultado $X \rightarrow Y \leftarrow Z$. Luego de esto, se orienta los enlaces restantes. Spirtes et al. (1993) presentan dos reglas para orientar los patrones del grafo:

1. Si $A \rightarrow B$, B y C son adyacentes (i.e. $A \rightarrow B - C$), A y C no son adyacentes y no existe un enlace dirigido hacia B, entonces el par de nodos $B - C$ se orienta $B \rightarrow C$.
2. Si existe un camino dirigido desde A hacia B y existe un enlace entre A y B entonces se orienta $A \rightarrow B$.

Meek (1995) en cambio, presenta cuatro reglas para orientar los patrones del grafo, estas reglas se utilizan en varias implementaciones del algoritmo de PC (e.g. pcalg, TETRAD). La Figura 3 muestra esos patrones a buscar dentro del grafo y la regla a aplicar. El resultado de aplicar las primeras tres reglas al grafo es un DAG con la máxima orientación (Meek, 1995).

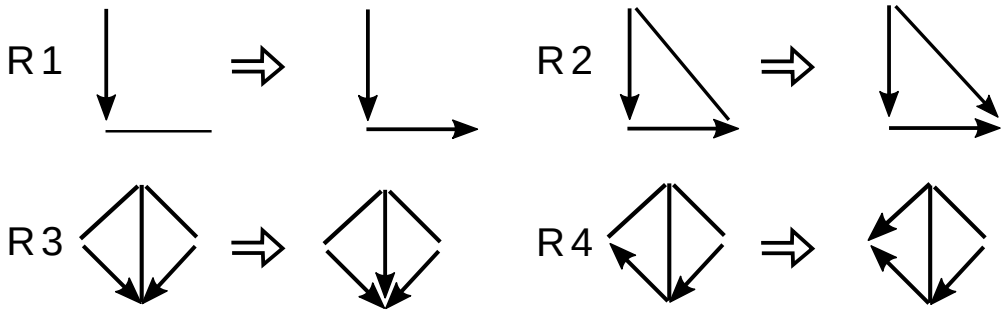


Figura 3: Reglas de Meek para orientar patrones de un grafo (Meek, 1995).

El algoritmo de PC al ser aplicado depende del orden en el que las variables están dispuestas en el conjunto de datos. Esto es un problema presente para todos los conjuntos de datos que se vuelve más pronunciado en conjuntos de datos con gran cantidad de variables ya que provoca que los resultados varíen mucho. Colombo & Maathuis (2014) aplicaron el algoritmo de PC a un conjunto de datos de 5000 variables con 25 variaciones del orden de las variables. Los resultados que obtuvieron indican que: el 30% de los enlaces son estables y aparecen en todas las variaciones, 30% de las variables aparecen en más del 50% de las variaciones y 40% de los enlaces son inestables y aparecen en menos del 50% de las variaciones. Esto indica que al cambiar el orden de las variables en los conjuntos de datos de entrada, se modifica el DAG resultante. Esto debido a que el orden de las variables afecta a la obtención del *skeleton* y de los conjuntos separadores. Afecta al skeleton ya que cambia el orden en el que los enlaces se procesan, si se

determina que *X—Y* es independiente se modifica de forma dinámica la matriz de adyacencia *G* y por consecuencia el vecindario de nodos. Cambiar el vecindario de nodos afecta a los conjuntos separadores que se utilizan para orientar las tripletas de nodos como *v-structures.* El hecho que se varíen los conjuntos separadores también afecta a la aplicación de reglas de orientación a la matriz de adyacencia. Esto debido a que los conjuntos separadores pueden tener información contradictoria y la orientación final depende del orden en el que se procesen las tripletas. Es por estos motivos que el algoritmo de PC no se puede paralelizar y es necesario de variaciones en el algoritmo que garanticen su independencia del orden de las variables. En las siguientes dos secciones se presentan estas modificaciones al algoritmo.

2.2.1. Algoritmo Conservativo de PC

El algoritmo conservativo de PC (CPC) se crea con el objetivo de robustecer el algoritmo con respecto a errores de muestreo (Kalisch et al., 2014). CPC introduce un paso adicional luego de encontrar el *skeleton* en el algoritmo de PC original y antes de orientar las tripletas del *skeleton*. Realiza una serie de pruebas de independencia condicional sobre cada tripleta de nodos candidata a *v-structure* del *skeleton* para determinar si deben ser orientadas. Dada una tripleta de nodos *X—Y—Z,* recupera los conjuntos de nodos vecinos *adj(X, G)* y *adj(Z, G)*. Por cada conjunto de vecinos, *adj*(_, *G*), forma $2^n - 1$ subconjuntos de nodos, donde n es la cardinalidad del conjunto de vecinos *adj*(_, *G*). Los subconjuntos de nodos se forman creando todas las combinaciones de nodos vecinos con cardinalidades 1, 2, …, n. Cada subconjunto a su vez se utiliza como conjunto de condicionamiento para las pruebas de independencia condicional, estas se realizan para determinar si los nodos *X* y *Z* son independientes. Si *X* y *Z* son independientes por un conjunto de condicionamiento *K*, los conjuntos *K* se denominan conjuntos separadores (*sepsets*) y se comprueba si el nodo *Y* pertenece a *K*. Las condiciones que se presentan son las siguientes:

1. Si *Y* está en todos los conjuntos separadores, se acepta que *X—Y—Z* no es una *v-structure* y el nodo *Y* se agrega a *sepset(X, Z)* y a *sepset(Z, X)*.
2. Si *Y* no está en ninguno de los conjuntos separadores, la tripleta de nodos es una *v-structure* por lo que se podrá orientar. El nodo *Y* se remueve de *sepset(X, Z)* y *sepset(Z, X)*, en caso de estar presente.

3. Si no se cumplen las dos condiciones anteriores, la tripleta se marca como infiel y no se podrá orientar. Eso quiere decir que *Y* se encuentra solo en algunos de los conjuntos separadores o que no se han encontrado conjuntos separadores.

Esta versión se considera una generalización conservativa del algoritmo de PC (Ramsey et al., 2012) porque es más cautelosa al obtener conclusiones de la orientación causal de las tripletas. Simulaciones realizadas por Ramsey et al. (2012) indican que esta versión del algoritmo de PC realiza un mejor trabajo al evitar crear enlaces falsos. Esta ventaja viene asociada con un aumento del tiempo de ejecución del algoritmo. Esto principalmente debido a las comprobaciones que realiza para cada tripleta. En las comparaciones con el algoritmo de PC que realiza Ramsey et al., (2012) se observa un incremento en el tiempo de ejecución, aun cuando solo utiliza conjuntos de datos de hasta 80 variables. El número de pruebas de independencia condicional que se realiza para cada tripleta $X{-}Y{-}Z$ es igual a 2^n+2^m-2, donde n y m son la cardinalidad de *adj(X, G)* y *adj(Z, G)*, respectivamente. Debido a esto, mientras más tripletas tenga un *skeleton*, más tiempo de ejecución será necesario para determinar si las tripletas se pueden orientar.

2.2.2. Algoritmo de PC-stable

Las pruebas de independencia condicional en el algoritmo de PC original tienden a cometer errores cuando el número de muestras del conjunto de datos es limitado i.e. obtiene resultados diferentes a lo que obtendría con más muestras en el conjunto de datos (Le et al., 2016). El algoritmo de PC al determinar que un par de nodos es independiente, procede a eliminar el enlace entre los nodos. El resultado es un algoritmo eficiente en el sentido de que evita realizar pruebas de independencia condicional en nodos que ya ha determinado que son independientes. Al momento que el algoritmo mantiene o elimina un enlace, enseguida influye en la organización de los nodos adyacentes, lo que provoca que el resultado del algoritmo (incluso dentro de cada iteración al encontrar el *skeleton*) dependa del orden de las variables en el conjunto de datos de entrada. Colombo & Maathuis (2014) proponen PC-stable, una variación del algoritmo de PC para obtener un mismo resultado sin que importe el orden de las variables en el conjunto de datos. Las modificaciones se centran en el proceso para determinar el *skeleton* y en cambios en lo introducido por CPC.

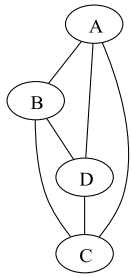
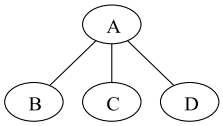
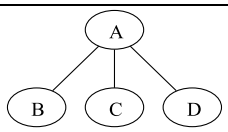
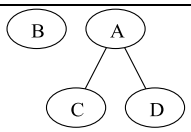
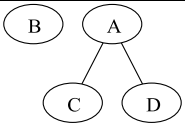
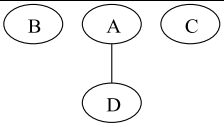
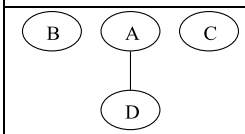
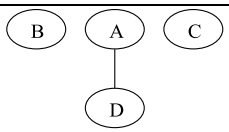
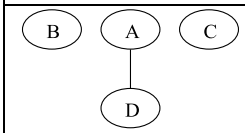
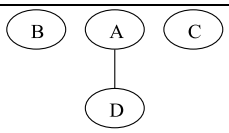
El proceso para determinar el *skeleton* se realiza en el segundo paso del algoritmo de PC y está organizado en iteraciones (niveles). En cada iteración, el grafo se actualiza dinámicamente al encontrar nodos independientes, i.e. los enlaces removidos reducen el número de vecinos de los nodos que estaban conectados por dichos enlaces. La modificación que se realiza en PC-stable se puede observar en el Algoritmo 2 e implica crear una copia del grafo al inicio de cada iteración, esta copia se utiliza para buscar los nodos adyacentes de cada par de nodos sobre los que se realizan pruebas de independencia condicional. Con esto se mantienen intactos los vecindarios de los nodos durante cada iteración. Al finalizar la iteración la copia se elimina y se crea otra con la topología actual para ser usada en la siguiente iteración. Debido a esto las pruebas de independencia condicional se pueden realizar en desorden dentro de la iteración, el orden en el que se realizan las iteraciones de *skeleton* todavía es necesario ya que utilizan la matriz de adyacencia resultante de la iteración anterior. Al determinar que un par de nodos es independiente se elimina su enlace en el grafo original y se almacena el conjunto por el que los dos nodos son independientes (*sepset*). La desventaja con esta modificación es que requiere de más pruebas de independencia condicional para determinar el *skeleton.* Esto debido a que realiza todas las pruebas de independencia posibles en cada iteración ya que el vecindario de nodos es estático.

```
Entrada: conjunto de datos, alpha, grafo completo G
Salida: skeleton del DAG

nivel = 0
repetir
   G' = G
   por cada enlace adyacente X-Y de G hacer:
      si (|adj(X,G')\{Y }| >= nivel) entonces:
         por cada subconjunto Z ⊆ adj(X,G')\{Y } y |Z| = nivel hacer:
            prueba IC(X,Y |Z)
            si I(X,Y |Z) entonces:
               Eliminar enlace X-Y de G
               Guardar Z en sepset(X,Y)
               salir
            fin
         fin
      fin
   fin
   nivel  = nivel + 1
hasta |adj(X,G')\{Y }| < nivel para cada enlace X-Y en G'
```

Algoritmo 2: Segundo paso del algoritmo PC-stable, obtención de *skeleton.*

La Tabla 2 muestra un ejemplo del funcionamiento del algoritmo de PC-stable que tiene los mismos datos de entrada que el ejemplo de ejecución del algoritmo de PC en la Tabla 1. Al comparar los dos ejemplos se observa que en la primera iteración (nivel 0) se generan los mismos resultados. Esto es debido a que en el nivel 0 no se utilizan conjuntos de condicionamiento, i.e. son vacíos. En la segunda iteración (nivel 1), en la prueba de independencia condicional número 10 *I(A, B | C)*, se elimina el enlace entre los nodos *A* y *B.* Esto provoca que en la Tabla 1 el vecindario de nodos del grafo cambie y sea necesario 2 pruebas de independencia condicional más hasta terminar el algoritmo. En cambio, en la Tabla 2 se realizan 3 pruebas de independencia más. Si la prueba de independencia que elimina el enlace entre *A y B* es errónea provoca que no se haga la prueba *I(A, C | B)* que evita la eliminación del enlace entre *A* y *C*. En cambio si se realiza primero la prueba *I(A, C | B)*, se elimina el enlace entre *A* y *C* generando un resultado completamente diferente en la Tabla 1.

Tabla 2: Ejemplo de funcionamiento del algoritmo de PC-Stable (Le et al., 2016).

Nivel	Grafo	# Prueba IC	Prueba	Resultado	Grafo Actualizado
0		1	¿I(A,B)?	No	
		2	¿I(A,C)?	No	
		3	¿I(A,D)?	No	
		4	¿I(B,A)?	No	
		5	¿I(B,C)?	Sí	
		6	¿I(B,D)?	Sí	
		7	¿I(C,A)?	No	
		8	¿I(C,D)?	Sí	
		9	¿I(D,A)?	No	
1		10	¿I(A,B|C)?	Sí	
		11	¿I(A,C|B)?	Sí	
		12	¿I(A,D|B)?	No	
		13	¿I(A,D|C)?	No	

La siguiente modificación se realiza en base al algoritmo conservativo de PC. Colombo & Maathuis (2014) encuentran que CPC puede ser muy conservativo al momento de decidir si una tripleta es infiel, de manera que muy pocas tripletas se marcan como infieles. Lo que proponen es la regla de la mayoría (*major_rule*) que brinda mayor flexibilidad al momento de marcar una tripleta como infiel. Dada una tripleta conformada por los nodos *X*–*Y*–*Z*, la regla de la mayoría entra en funcionamiento solo cuando el nodo *Y* no está en todos los conjuntos de condicionamiento por los que *X* y *Z* son independientes i.e. conjuntos separadores (*sepsets*). La regla inicia determinando el porcentaje de veces que *Y* está dentro de los conjuntos separadores. La tripleta se podrá orientar cuando se encuentre al menos un conjunto separador y *Y* se encuentre en menos del 50% de los conjuntos separadores, en estos casos remueve *Y* de *sepset(X, Z)* y de *sepset(Z, X)*. Caso contrario, la tripleta se marca como infiel, esto quiere decir que el porcentaje es mayor o igual al 50%. Durante el proceso se guarda un registro de las tripletas marcadas como infieles. Este registro se utiliza al momento de orientar las tripletas para consultar si una tripleta es infiel.

2.3. Oportunidades para paralelizar

El algoritmo de PC cuenta con una serie de operaciones (e.g. pruebas de independencia condicional, orientación de enlaces) que se realizan en sus distintos pasos con el objetivo de encontrar una estructura causal (DAG) que represente las independencias que contiene un conjunto de datos. El número de estas operaciones está íntimamente relacionado con el número de variables que contiene el conjunto de datos. A medida que se incrementa el número de variables, las operaciones a realizar crecen exponencialmente. Esto es un problema cuando se trabaja con grandes conjuntos de datos (>3000 variables) ya que el tiempo necesario para obtener una solución se vuelve muy grande (e.g. días). Una solución a este problema o reducción de su impacto es aplicando paralelización al algoritmo. Esto no es posible con el algoritmo de PC original debido a que el orden en el que se realizan cada una de sus operaciones es importante he influyen en el resultado final (una operación siempre depende del resultado de la anterior). Una forma de cambiar el orden en el que se realizan las operaciones es cambiando el orden en el que están dispuestos las variables en el conjunto de datos. Según las pruebas que realiza Colombo & Maathuis (2014), al cambiar el orden de las variables de un conjunto de datos con 5000 variables, comprueba que los DAGs resultantes son

diferentes entre sí, aunque existe un 30% de variables que son constantes en cada resultado. Para mitigar la dependencia del orden de las variables se introdujo PC-stable que incluye cambios en el algoritmo de PC original y en CPC. Estos cambios son los que permiten que la obtención de *skeleton* y la orientación de los enlaces sean independientes del orden y por consiguiente haya una oportunidad de paralelizar ciertos pasos. Aunque existen operaciones como la aplicación de reglas para la orientación de enlaces que aún tienen dependencia del orden en el que se aplican.

De acuerdo a las modificaciones de PC-stable y CPC se determinó que se puede aplicar paralelización al proceso de obtención de *skeleton*, procesamiento de tripletas y a la obtención de la matriz de coeficientes de correlación. La modificación que realiza PC-stable permite la paralelización del proceso para la obtención del *skeleton*, aunque es una paralelización limitada a las operaciones entre cada iteración del proceso. A grandes rasgos lo que se realiza dentro de cada iteración es: (0) crear una copia del grafo *G* denominada *G'*, (1) obtener los enlaces que contiene el grafo *G* (inicialmente completo), (2) dado un enlace *X*─*Y* formar conjuntos de condicionamiento a partir de *adj(X, G')* con cardinalidad igual al nivel de la iteración y (3) por cada conjunto de condicionamiento realizar una prueba de independencia condicional para determinar si *X, Y* son independientes. Las operaciones que son independientes entre sí y que pueden ser paralelizadas son las de (2). Estas operaciones son independientes entre sí ya que se cuenta con *G'* al que se consulta el vecindario de nodos y no cambia en cada iteración. La paralelización consiste en repartir el conjunto de enlaces entre N trabajadores, cada trabajador procesa un enlace a la vez y realizan los pasos (2) y (3). De esta manera se logra procesar N enlaces a la vez reduciendo el tiempo necesario para encontrar el *skeleton.*

El siguiente proceso que se puede paralelizar es la orientación de los enlaces, i.e. determinación de las *v-structures* del grafo *G.* La paralelización se basa en el proceso que introduce CPC para determinar si las tripletas del grafo deben ser orientadas como *v-structures*, por lo que al final, la paralelización se realiza sobre los procesos de CPC. Ramsey et al. (2012) presenta CPC como un proceso para agregar robustez al algoritmo ante errores de muestreo y en cambio, Colombo & Maathuis (2014) lo usa y modifica para que la orientación de los enlaces sea independiente. Eso hace que nuestra implementación sea conservativa al definir la dirección de un enlace (hay más enlaces cuyas direcciones no son definidas) y

robusta en el sentido de que es menos probable que las direcciones de los enlaces encontrados en el grafo final hayan aparecido por efectos del azar. El proceso que realiza CPC a grandes rasgos es: (1) obtener las tripletas que contiene el grafo *G*, (2) dado una tripleta *X—Y—Z* formar subconjuntos de nodos a partir de *adj(X, G) y adj(Z, G)*, (3) usando los subconjuntos realizar pruebas de independencia condicional para determinar los conjuntos separadores de *X, Z* y (4) a partir de los resultados determinar si la tripleta se debe orientar. La paralelización de CPC se puede realizar sobre (2) y (3), esto debido a que consultan el vecindario de nodos de *X* y *Z* en un grafo *G* estático i.e. cada operación es independiente. La paralelización a elegir dependerá de la cantidad de tripletas que contenga el grafo y el número de pruebas de independencia condicional a realizar por cada tripleta. En los dos casos la paralelización consiste en repartir tareas (tripletas o pruebas de independencia condicional) entre N trabajadores de forma que cada trabajador procese una tarea a la vez por lo que se realizan N tareas en paralelo.

Capítulo 3: Implementación

El lenguaje de programación que se utilizó para la implementación fue C. Se incorporaron los cambios que introducen PC-Stable y CPC, además, se incluyó conocimiento a priori en base a la implementación de TETRAD. A continuación, se describen los detalles de la implementación.

3.1 Elección del lenguaje de programación

Un lenguaje de programación define la semántica y sintaxis necesaria para expresar ideas y comunicarse efectivamente con un computador (Jaffar, 2016). Los lenguajes de programación abstraen el funcionamiento de las computadoras para poder expresar instrucciones al computador en forma legible para los humanos. Las instrucciones se transforman en funciones y procedimientos. Existen diferentes tipos de lenguajes de programación que se clasifican dependiendo de su nivel. El nivel indica qué tan cerca está un lenguaje de programación del lenguaje máquina. Un lenguaje de alto nivel (e.g MATLAB, Java, R) cuenta con diversas abstracciones (e.g. asignación y liberación de recursos) que facilitan el desarrollo de programas. Las abstracciones encapsulan procesos complejos de la computadora para que el programador se dedique solo a pensar en las ideas a expresar en código. Aun dentro del mismo nivel, cada lenguaje provee de más o menos abstracciones. Las facilidades que brinda un lenguaje de programación en su mayoría se hacen a costa del rendimiento final del programa. Los programas implementados en un lenguaje de alto nivel requieren de más recursos (e.g. memoria, CPU) y no se comunican directamente con el hardware del computador (Prakash, 2017). Debido a que no todos los lenguajes en un mismo nivel proveen las mismas abstracciones, la cantidad de recursos adicionales para cada lenguaje varía.

Para la implementación del algoritmo de PC se eligió el lenguaje de programación C. C es un lenguaje de programación simple y eficiente que no dispone de ciertas características (e.g. colector de basura, tipado dinámico, manejo de excepciones) que facilitan el trabajo al programador, pero agregan capas de abstracción que aumentan el uso de recursos. El objetivo de usar C es crear una implementación con código portable y mantenible minimizando el uso de recursos. Es un lenguaje portable ya que la mayoría de arquitecturas cuentan con un compilador C y en el peor de los casos requiere pocas modificaciones para su funcionamiento. Debido a

su sencillez, el código escrito en C requiere poco esfuerzo para entender su funcionamiento lo que facilita el mantenimiento del código.

C se considera un lenguaje de medio nivel ya que contiene características de lenguajes de alto y de bajo nivel (Rongala, 2015). C es un lenguaje lo suficientemente eficiente que se utiliza para el desarrollo de compiladores e intérpretes de otros lenguajes de programación. El uso de C permite un manejo eficiente de los números y debido a su longevidad se ha probado y mejorado durante años, cuenta con compiladores capaces de generar código altamente eficiente y de una gran colección de librerías (e.g GSL) que facilitan el desarrollo. La principal desventaja de C es que demanda más atención que otros lenguajes de programación de alto nivel, por ejemplo, requiere que el programador entienda conceptos de asignación y liberación de memoria. Debido a esto es muy fácil cometer errores al programar haciendo que un mal manejo de la memoria puede causar problemas de seguridad. Sin embargo, las ventajas que brinda C en cuanto a su eficiencia opacan sus desventajas. Existen otros lenguajes que tienen un manejo más eficiente de las estructuras de datos (e.g. Fortran), pero no se han elegido debido a que su curva de aprendizaje es más lenta.

3.2 Implementación del algoritmo de PC

En qué se basa la implementación

En esta sección se describen los detalles de la implementación del algoritmo de PC original. La implementación se realizó en base a pcalg. Pcalg es un paquete escrito en el lenguaje de programación R e incluye varios algoritmos para el aprendizaje de estructuras causales (e.g. PC, FCI[12], GIES[13]). Para esta implementación solo se tomó en cuenta la implementación de PC. En las siguientes secciones se implementa PC-stable y CPC basados en la implementación que aquí se describe. Se utilizó pcalg como base de la implementación debido a que cuenta con varios años de desarrollo, sus primeras versiones (e.g. 0.1-3) datan del año 2006 y la última versión disponible al momento de implementar nuestra versión es la 2.5-0, liberada en julio de 2017. Durante sus años de desarrollo se ha utilizado en diversos proyectos, desde análisis de redes en el campo de la psicopatología (Borsboom &

[12] *Fast Causal Inference*
[13] *Greedy Interventional Equivalence Search*

Cramer, 2013) hasta la obtención de estadísticas de alta dimensión con vistas a aplicaciones en biología (Bühlmann, Kalisch, & Meier, 2014).

Librerías usadas para la implementación

Para la implementación de PC se usó desarrollos de terceros para varias de las funciones necesarias. Se utilizó la librería GSL y la implementación de Bogaerts (2016) para obtener la pseudoinversa de una matriz. Se utilizaron debido a que realizar un desarrollo propio de las funciones que proveen requeriría más tiempo de lo planeado, además, cuentan con un recorrido en el que se han probado y es más probable que estén libre de errores. Sin embargo, la desventaja es que, si se encuentra un error y no se entiende la implementación, solucionar el error puede ser un proceso complejo. GSL se utiliza para trabajar con las matrices y vectores que utiliza internamente el algoritmo de PC. El motivo de su uso es que: implementa eficientemente varias de las funciones necesarias y provee una API[14] para trabajar desde C. Algunas de las operaciones que se realizaron con GSL son: transposición de matrices, lectura de archivos con el conjunto de datos de entrada para el algoritmo, copia de datos entre matrices, reserva de memoria para matrices y descomposición de valores singulares. El algoritmo de PC utiliza la inversa generalizada o pseudoinversa del tipo Moore-Penrose[15] para las pruebas de independencia condicional. Se utilizó la implementación de Bogaerts (2016) para obtener la pseudoinversa debido a que su código fuente está disponible y libre, está escrito en C y utiliza la librería GSL.

Cómo se desarrolló la implementación

El primer paso en la implementación es la obtención de la matriz de correlación de las variables del conjunto de datos. Para lograr este objetivo se lee el archivo que contiene el conjunto de datos que se va a utilizar en el algoritmo. El conjunto de datos se encuentra en un archivo con formato CSV[16] y contiene muestras de las variables del sistema que representa. El archivo contiene m filas donde cada fila se conforma de n campos o muestras, una por cada variable del sistema. El archivo

[14] *Application Programming Interface*

[15] Moore-Penrose es descrito en detalle en (Greville, 1958).

[16] *Comma-separated values*, es un formato de archivo que almacena información en texto plano, cada línea contiene campos separados por comas. Cada fila tiene la misma secuencia de campos.

CSV se lee usando la librería GSL y se copian los datos a una matriz (de dimensiones *m* x *n*) del tipo *double*. La matriz se utiliza como entrada para el método *correlationMatrix()* para obtener la matriz de correlación con los coeficientes de correlación entre variables. Para almacenar los coeficientes de correlación se utiliza una matriz de *n* x *n* (*n* igual al número de variables) del tipo *double*. El método *correlationMatrix()* recibe como entrada la matriz con el conjunto de datos, el número de muestras, el número de variables y devuelve la matriz de correlación. Dentro de *correlationMatrix()* se recorre cada variable del conjunto de datos y se utiliza el método *gsl_stats_correlation()* para calcular el coeficiente de correlación de Pearson[17].

El siguiente paso del algoritmo consiste en crear una matriz de adyacencia *G* de tamaño *n* x *n* (*n* igual al número de variables del conjunto de datos) que representa el grafo completo con el que trabaja el algoritmo. Mediante *G* se representan las relaciones entre los nodos del grafo. Se utiliza los valores "1" y "0" para representar la existencia o no de un enlace dirigido entre los nodos. Un enlace dirigido $X \rightarrow Y$ se representa con el valor "1" en la posición [*X*, *Y*] y "0" en [*Y, X*] de la matriz de adyacencia. En cambio, un enlace bidireccional $X - Y$ se representa con el valor "1" en las direcciones [*X, Y*] y [*Y, X*] de la matriz de adyacencia. La diagonal principal de la matriz contiene ceros ya que no existen enlaces entre los mismos nodos.

Skeleton

El segundo paso del algoritmo es la obtención del *skeleton* para lo cual se implementó un método con el mismo nombre. *Skeleton()* recibe como entrada la matriz de correlación *G* y el nivel de significancia[18] *alpha*. Los resultados del algoritmo son: la matriz de adyacencia *G* que representa *skeleton* y una matriz de *n* x *n* conformada de listas enlazadas, cada lista enlazada es un *sepset*. Cada *sepset* contiene el conjunto de condicionamiento por el que *X* y *Y* son independientes, *sepset(X, Y)*. La ubicación del sepset en la matriz indica a qué par de nodos pertenece. El método *skeleton()* se organiza internamente en iteraciones, cada iteración se asocia con un nivel. El nivel inicia en 0, incrementa en 1 en cada iteración y se utiliza como cardinalidad del conjunto de condicionamiento para las pruebas de independencia condicional. Al inicio de cada iteración se recupera el

[17] La correlación de Pearson mide el grado y dirección de la relación lineal entre dos variables (Gravetter & Wallnau, 2010).

[18] Es la probabilidad de rechazar una hipótesis nula cuando es verdadera.

número de enlaces que existe en *G*. Por cada enlace *X — Y*, se recupera los nodos adyacentes *adj(X, G)* y se genera un conjunto de combinaciones C con cardinalidad igual al nivel de la iteración. Por cada combinación C_i se realiza una prueba de independencia condicional $I(X, Y \mid C_i)$ que devuelve un valor numérico *pval*. Si *pval* es mayor que *alpha* los nodos *X* y *Y* son independientes, se agrega "0" en las posiciones *[X, Y]*, *[Y, X]* de la matriz de adyacencia *G* y se agrega el C_i al *sepset(X, Y).* El proceso termina cuando la cardinalidad de *adj(X, G)* es menor al nivel de la iteración. La implementación se realizó de forma que el método reciba y devuelva referencias (punteros) a variables y no una copia de la variable, de esta forma se evita el uso indebido de memoria y se mantiene un control de las variables que se crean.

Pruebas de independencia condicional

Dentro del paquete pcalg se utiliza la función *gaussCItest()*[19] para realizar pruebas de independencia condicional. En nuestra implementación su funcionamiento se implementó en el método *condIndFisherZ()*. Este método recibe como entrada las variables del enlace *X — Y*, un límite del nivel de significancia, la matriz de correlación y el número de muestras del conjunto de datos. Al igual que en métodos anteriores, se reciben referencias a las variables (punteros) y no una copia de ellas. El resultado de *condIndFisherZ()* es un valor numérico *pval*, este se compara con *alpha* para determinar si los nodos del enlace son independientes. *CondIndFisherZ()* inicia encontrando la correlación parcial[20] de los nodos del enlace *X — Y* a partir del conjunto de condicionamiento y la matriz de correlación. La correlación parcial se obtiene utilizando la inversa generalizada o pseudoinversa de una matriz formada por los coeficientes de correlación de los nodos *X*, *Y* y del conjunto de condicionamiento. A partir de la correlación parcial se obtiene la transformación Z de Fisher[21], que se utiliza para encontrar la función de distribución acumulativa, esto se realiza con la función *gsl_cdf_gaussian_P()*. El resultado *pval* es el doble de valor obtenido en la función de distribución acumulativa.

[19] https://www.rdocumentation.org/packages/pcalg/versions/1.0-0/topics/gaussCItest

[20] La correlación parcial indica la fuerza de la relación entre las dos variables (*X — Y*) mientras se controla el efecto de las variables del conjunto de condicionamiento (Iversen & Gergen, 2012).

[21] La transformación Z de Fisher es una forma de convertir la distribución de muestreo r de Pearson (coeficiente de correlación) a una normalmente distribuida (Chalmer, 1986).

Orientación de tripletas

El siguiente paso del algoritmo de PC es la orientación de las tripletas del skeleton, el método usado internamente se denomina *udag2pdagRelaxed(),* nombre heredado de pcalg. *Udag2pdagRelaxed()* es un método genérico para todas las variaciones del algoritmo de PC que se implementaron. Sin embargo, en esta sección se describen los detalles concernientes al algoritmo de PC original. *Udag2pdagRelaxed()* recibe como entrada la matriz de adyacencia *G* y la matriz con los *sepsets* de cada nodo. El resultado es la matriz *G* modificada con enlaces dirigidos. A partir de *G*, se recupera todos los enlaces del grafo. La parte principal del método consiste en iterar sobre la lista de enlaces recuperada. A partir del enlace con los nodos X─Y, se cuenta y recuperan todos los nodos adyacentes *adj(Y, G)*, este conjunto se denomina Z. En este punto se tiene todas las tripletas de nodos que se pueden formar con el enlace X─Y. El siguiente paso consiste en comprobar cuáles de las tripletas pueden ser *v-structure.* Para determinar si X─Y─Z_i puede ser una *v-structure*, se comprueba que no exista el enlace X ─ Z_i y que el nodo Y no se encuentre en los conjuntos de nodos *sepset*(X, Z_i) y *sepset*(Z_i, X). Si se comprueba que X─Y─Z_i es una *v-structure* se orienta de la forma $X \rightarrow Y \leftarrow Z_i$.

Aplicación de las reglas de Meek

Luego de orientar las *v-structures* se orientan los enlaces restantes para lo cual se aplica las primeras 3 reglas de Meek (1995) en orden. Con las 3 reglas es suficiente para obtener el grafo con la máxima orientación de los enlaces (Meek, 1995), la Figura 3 muestra las reglas de Meek. Las primeras 3 reglas de Meek se implementaron en los métodos *rule1(), rule2() y rule3().* Las reglas se aplican en un bucle *do-while* a la matriz de adyacencia *G*. La condición para que el bucle continúe es que al aplicar las 3 reglas haya cambios en la matriz de adyacencia *G*. Cada método recibe como referencia la matriz de adyacencia *G* y devuelve una matriz con la regla aplicada.

- *Rule1()* busca tripletas de nodos que cumplan el patrón $X \rightarrow Y$─Z para orientarlas a la forma $X \rightarrow Y \leftarrow Z$. Para encontrar las tripletas que cumplan el patrón, *rule1()* itera sobre los enlaces dirigidos $X \rightarrow Y$ que existan en *G*. Dado el enlace $X_i \rightarrow Y_i$ comprueba si en los nodos adyacentes *adj*(Y_i, *G)* existe un nodo Z_i que sea independiente de X_i y que tenga un enlace bidireccional con Y_i.

- *Rule2()* busca tripletas de nodos que cumplan el patrón $X \rightarrow Y \rightarrow Z$, $X - Z$ para orientarlas a la forma $X \rightarrow Y \rightarrow Z$, $X \rightarrow Y$. La búsqueda se basa en iterar sobre el conjunto de enlaces bidireccionales $X - Z$ que contiene la matriz de adyacencia. Dado el enlace $X_i - Z_i$ comprueba si existe un nodo Y_i que tenga enlaces dirigido de la forma $X_i \rightarrow Y_i \rightarrow Z_i$.
- *Rule3()* involucra la búsqueda de conjuntos de nodos de cardinalidad 4 que cumplan el patrón $W_i - X_i$, $W_i - Y_i$, $W_i - Z_i$, $X \rightarrow Y_i$ y $Z_i \rightarrow Y_i$. El método se compone de un bucle que recorre los enlaces $W - Y$ que contiene la matriz de adyacencia *G.* Dado el enlace $W_i - Y_i$ se busca en $adj(W_i, G)$ y $adj(Y_i, G)$ un conjunto de nodos X de forma que cada nodo forme los enlaces: $W_i - X_i$ y $X_i \rightarrow Y_i$. *Rule3* continua siempre que el conjunto X tenga dos o más nodos, si este es el caso se crean permutaciones del conjunto X, cada permutación tiene dos nodos. Dada la permutación (X_i, X_{i+1}), se comprueba si son independientes, en tal caso X_{i+1} representa el nodo Z_i en el grupo de nodos que se busca.

Generación de archivo de respuestas

Para realizar comparaciones de rendimiento y resultados se requiere mantener un registro de la ejecución del algoritmo. Por tal motivo, de cada ejecución del algoritmo se registra el tiempo de ejecución medido en ciclos de CPU y usando el reloj de la computadora o *wall time*. Además, durante la ejecución del algoritmo se crean archivos intermedios con los resultados que cada paso del algoritmo genera. Estos archivos se utilizan para comparar la ejecución con otras implementaciones (e.g. pcalg). Si los resultados de ejecución son diferentes, con estos archivos se puede revisar el proceso paso a paso y verificar en qué punto varían los algoritmos comparados. A más de realizar comparaciones con otros algoritmos, tener registro de todos los resultados, ayuda a tener mayor información para poder llegar a más conclusiones de acuerdo al problema específico que el investigador esté estudiando. Al finalizar el algoritmo se cuenta con los siguientes archivos en texto plano:

1. *Skeleton.csv:* archivo CSV que contiene una matriz de adyacencia *G* que representa al *skeleton* que se encuentra en el segundo paso del algoritmo.
2. *Sepset_after_skeleton.csv:* es un archivo organizado en filas, cada fila contiene los nodos de un *sepset*. Los *sepset* se obtuvieron durante la obtención del *skeleton*.

3. *Resultado.csv:* un archivo CSV que contiene la matriz de adyacencia del grafo resultante de la ejecución del algoritmo.
4. *Execution.txt:* contiene el tiempo de ejecución del algoritmo.

Uno de los mayores retos que se tuvo durante la implementación fue lograr que la implementación en C realice los mismos pasos que pcalg de manera que produzca los mismos resultados y se los pueda comparar. El reto surge debido a que el lenguaje R cuenta con varias funciones (e.g. *which(), setdiff(), cbind(), rbind()*) que simplifican el código de pcalg. De estas funciones fue necesario revisar su documentación y código fuente para determinar cómo funcionan exactamente, antes de implementar en C un método que realice la misma función. Uno de los problemas que se presentó, es que en una parte del algoritmo se necesita recuperar e iterar sobre los enlaces de la matriz de adyacencia. Pcalg lo realiza con una función que busca los enlaces en cada columna de la matriz de adyacencia. Al replicar esta función en C, la búsqueda de los enlaces inicialmente se realizó por cada fila, por lo que el resultado final era diferente. En la mayoría de los casos una función de R que recupera un conjunto de nodos de la matriz de adyacencia, se implementó en C con varias estructuras *for*. Esto debido a que en C se necesita saber la cantidad de información que se va a guardar para reservar el espacio de memoria adecuado. Por lo que se requiere un *for* que cuente los datos a recuperar, reservar el espacio de memoria y un *for* que recupere los datos. Además, al trabajar con punteros se necesita mantener variables que indiquen el tamaño de las estructuras a iterar.

Problemas que se encontraron durante la implementación

Debido a que la implementación en C del algoritmo de PC se basó en pcalg, se realizaron varias pruebas para comprobar que la implementación en C produzca los mismos resultados que pcalg. Para comparar los resultados de los dos algoritmos se usaron 1200 conjuntos de datos sintéticos generados de acuerdo al procedimiento explicado en la sección 4.1. De los 1200 conjuntos de datos, 1000 se componen de 80 variables y 200 de 300 variables. Estas pruebas se realizaron durante el desarrollo de la implementación para comprobar si existen errores y a acuerdo a los archivos generados en cada ejecución encontrar el punto en el que las dos implementaciones difieren, esto fue importante para encontrar de forma ágil los errores. De esta manera se encontraron y solucionaron gran parte de los errores presentes durante el desarrollo. Sin embargo, con 6 conjuntos de datos de 300 variables, los resultados que se obtuvieron con la implementación en C varia con

respecto a los de pcalg. Esto representa un 3% de los conjuntos de datos de 300 variables y un 0.5% de total de conjuntos de datos probados. Los resultados difieren en el número y orientación de algunos enlaces de los DAG resultantes de las dos implementaciones. En el peor de los casos, los conjuntos de datos variaban en 22 enlaces i.e. un 2.4% de los enlaces resultantes (910) y un 0.024% del número de enlaces posibles (90000). Al investigar el error se comprobó que era un problema con las pruebas de independencia condicional y en concreto con la pseudoinversa. La pseudoinversa que se obtiene en la implementación en C varía con respecto a la que obtiene pcalg. Esto se debe a que para una matriz existen varias pseudoinversas, aunque se comprobó que en la implementación en C se obtiene la pseudoinversa de Moore-Penrose al igual que en pcalg. Sin embargo, la diferencia en los resultados tiene que ver con la tolerancia con la que se calcula la pseudoinversa; aunque en los dos casos se utiliza *1e-15*, cada implementación lo interpreta de diferente manera (e.g. en sentido absoluto, relativa al máximo valor singular). Al analizar este problema se consideró que no era de gran importancia ya que el algoritmo de PC no busca encontrar el DAG exacto que represente las independencias condicionales presentes en el conjunto de datos. Además, el porcentaje de enlaces diferentes entre los DAG es pequeño en relación al número de enlaces resultantes y posibles, y esto no cambiaría las conclusiones que un investigador tenga en base a los resultados[22].

3.2.1 Algoritmo Conservativo de PC (CPC)

La implementación de CPC se realizó luego de la implementación del algoritmo de PC original. Los cambios que realiza CPC se implementaron como opciones adicionales que se pueden elegir al ejecutar el algoritmo (*conservative* = *TRUE*). El resultado es un algoritmo que puede ejecutar el algoritmo de PC original y CPC. Al igual que la implementación de PC original, CPC se implementó en base a la implementación de CPC realizada en *pcalg*. CPC introduce el "procesamiento de tripletas" que determina si cada tripleta del skeleton se debe orientar. El método interno que contiene este proceso se denomina *pc_cons_intern()*. *Pc_cons_intern()* se ejecuta a continuación de *skeleton()* y recibe como entrada la matriz de adyacencia *G* con el *skeleton,* la matriz con los *sepset* y la matriz de coeficientes de correlación.

[22] Esto es lo más probable, aunque, por no tener experiencia en todo el gran espectro de posibles aplicaciones, puede haber excepciones.

Cómo se implementó el algoritmo de PC conservativo

Internamente *pc_cons_intern()* comprueba si existen enlaces en el *skeleton* provisto, caso contrario el método termina. El método recorre cada enlace que contiene el *skeleton* y lo guarda en una matriz de dos columnas de tipo *size_t*. Se usó *size_t* ya que es el tipo de dato que utiliza C para representar el tamaño de objetos, además, garantiza que podrá almacenar el índice de cualquier matriz. El primer paso del método consiste en recuperar todas las tripletas de nodos $X - Y - Z$ candidatas a *v-structure* que contiene el *skeleton*, para lo que cada tripleta debe cumplir que $X - Y$, $Y - Z$ y los nodos X y Z sean independientes. Para lograr esto, recupera e itera sobre todos los enlaces $X - Y$ que contiene el *skeleton*. A continuación, dado el enlace $X_i - Y_i$, el algoritmo recupera el conjunto de nodos Z formado por los nodos $adj(Y_i, G)$. Con cada nodo de Z, se forma la tripleta de nodos $X_i - Y_i - Z_i$ siempre que los nodos X_i y Z_i sean independientes, es decir existe un "0" en las posiciones $[X_i, Z_i]$ y $[Z_i, X_i]$ de la matriz de adyacencia G. Una vez recuperado el conjunto de tripletas $X - Y - Z$, se determina si se pueden orientar. Esto se lo realiza con el método *checkTriple()*.

CheckTriple() recibe como entrada la tripleta de nodos $X_i - Y_i - Z_i$, los conjuntos de nodos adyacentes $adj(X_i, G)$ y $adj(Z_i, G)$, los conjuntos de nodos $sepset(X_i, Z_i)$ y $sepset(Z_i, X_i)$, la matriz de correlación y el valor de significancia *alpha*. El resultado es un valor numérico *res* que indica si $X_i - Y_i - Z_i$ se podrá orientar como *v-structure* (1), si no se podrá orientar (2) o si es infiel por lo que tampoco se podrá orientar (3). Para determinar si la tripleta de entrada se puede orientar, se realiza pruebas de independencia condicional $I(X_i, Z_i \mid K_i)$. Los conjuntos de condicionamiento K se forman a partir de cada conjunto de nodos $adj(X_i, G)$ y $adj(Z_i, G)$. Dado uno de los conjuntos de nodos adyacentes, se crean todas las combinaciones de nodos con cardinalidad 1,2,3…n, donde n es el número de nodos adyacentes. Si los nodos X_i, Z_i son independientes dado el conjunto de condicionamiento K_i (conjunto separador) se comprueba si el nodo Y_i de la tripleta se encuentra dentro del conjunto separador y se mantiene un registro de todas las veces que así sea. Si no se encuentran conjunto separadores *res* toma el valor de "1", este es un cambio que realiza pcalg ya que según la sección 2.2.1 la tripleta debería ser infiel. En el caso de que Y_i haya estado en todos los conjuntos separadores, *res* toma el valor de "2" y se agrega Y_i a los conjuntos $sepset(X_i, Z_i)$ y $sepset(Z_i, X_i)$. Si no está en los conjuntos separadores *res* toma el valor de "1", y se remueve Y_i de los conjuntos $sepset(X_i, Z_i)$ y $sepset(Z_i, X_i)$ en caso de estar presente. Sin embargo, si ha estado solo en algunos de los

conjuntos de condicionamiento *res* toma el valor de “3”. Las variables de entrada se pasan como referencias por lo que los cambios que se realicen dentro del método se realizan sobre la variable global y no sobre una copia.

Por cada tripleta de nodos que se define como infiel (*res* = “3”) se mantiene un registro que se consulta antes de orientar una tripleta. El registro de tripletas infieles es una lista de enteros (*int*), cada entero representa una tripleta. Debido a esto para almacenar una tripleta en el registro, primero se obtiene un valor que la representa. Se utiliza el método *triple2num()* que se asemeja a una función hash: recibe la tripleta y devuelve un valor numérico único que representa a la tripleta. El método *triple2numb()* es el mismo que implementa pcalg, esto permite comparar resultados. Al recorrer todas las tripletas del *skeleton* termina el método *pc_cons_intern().* Lo siguiente que implementa CPC es una modificación al proceso de orientación de tripletas como *v-structures*, método *udag2pdagRelaxed()*. Dentro de este método, una vez se ha determinado que $X{-}Y{-}Z_i$ se debe orientar como *v-structure*, se comprueba si la tripleta se marcó como infiel anteriormente en el método *pc_cons_intern()*. Para lo cual se utiliza el método *triple2numb()* para obtener el número que representa a $X{-}Y{-}Z_i$ y se comprueba si el registro de tripletas infieles contiene ese número. La tripleta no orienta solo en caso de que $X{-}Y{-}Z_i$ se encuentre registrada como infiel.

Al implementar CPC se generan dos archivos más:

- *sepset_after_skel.csv:* contiene la matriz de sepsets actualizada que se obtiene luego del método *pc_cons_intern()*. Contiene la misma estructura que *sepset_after_skel.csv.*
- *ambiguous_after_pcons.csv*: contiene el registro de tripletas infieles definidas en *pc_cons_intern()*.

3.2.2 Algoritmo de PC-stable

En esta sección se detalla la implementación del algoritmo de PC-Stable como continuación de lo ya implementado en PC y CPC. Los cambios que se introducen son opcionales de forma que del mismo algoritmo se pueda obtener el resultado del algoritmo de PC original, CPC o PC-stable. Los cambios se basan en la implementación que realiza pcalg del algoritmo de PC-Stable. Como ya se explicó, la versión estable del algoritmo de PC incluye modificaciones al proceso de

obtención del *skeleton* (segundo paso del algoritmo de PC) y al procesamiento de tripletas introducido por CPC.

Cómo se implementó PC-stable

El método *skeleton()* está organizado en interacciones. Cada iteración tiene un nivel y representa la cardinalidad del conjunto de condicionamiento para las pruebas de independencia condicional. La modificación que se introduce con PC-stable consiste en crear una copia de la matriz de adyacencia *G* al inicio de cada iteración. Esta copia se mantiene intacta durante la iteración y se utiliza para realizar consultas del vecindario de nodos. Durante la iteración se procesa cada enlace que contiene *G*. Dado el enlace *X*─*Y*, se buscan y recuperan los nodos *adj(X, G)* en la copia de *G*. Si se determina que *X* y *Y* son independientes, el registro de su independencia se guarda en la matriz de adyacencia *G* (“0” en las posiciones [*X, Y*] y [*Y, X*] de la matriz de adyacencia).

En el procesamiento de tripletas, en el método *pc_cons_intern()*, se modifica su comportamiento para agregar más flexibilidad al momento de definir una tripleta como infiel. Se agrega como variable de entrada *maj_rule* que permite indicar si se aplican las modificaciones que PC-stable realiza sobre este método. El procedimiento que realiza *pc_cons_intern()* en líneas generales es: (1) determinar y recuperar las tripletas que contiene la matriz de adyacencia *G*, (2) dada la tripleta *X*─*Y*─*Z* recuperar sus conjuntos separadores y (3) de acuerdo a las veces que *Y* se encuentra dentro de los conjuntos separadores decidir si se debe orientar la tripleta. Los pasos (2) y (3) se realizan en el método *checkTriple()* que es donde se aplican las modificaciones. Una vez se han encontrado los conjuntos separadores de la tripleta *X*─*Y*─*Z*, se calcula el porcentaje de veces que *Y* se encuentra en los conjuntos separadores de *X* y *Z.* La regla de la mayoría entra en funcionamiento solo cuando el porcentaje es diferente de 0% y 100%. Caso contrario se aplican las reglas que define CPC. Si el porcentaje es menor al 50%, la tripleta se puede orientar como v-structure y se remueve el nodo *Y* de *sepset(X, Z)* y de *sepset(Z, X)* para garantizar que así sea. En cambio si el porcentaje es mayor al 50%, la tripleta no es una *v-structure* por lo que no se podrá orientar, se agrega el nodo *Y* a *sepset(X, Z)* y a *sepset(Z, X)* ya que para orientar una tripleta se requiere que *Y* esté ausente de estos conjuntos de nodos. Si el porcentaje es igual al 50%, entonces la tripleta se denomina infiel por lo que no se podrá orientar.

3.2.3 Conocimiento a priori

El conocimiento a priori es información que se tiene del comportamiento del sistema que se está estudiando y que no se necesita inferir de ninguna manera. Generalmente es proveída por un experto en el área de estudio del sistema o por experiencias pasadas. El conocimiento indica la existencia de una relación causal o la ausencia de esta. El conocimiento a priori es importante para reducir la complejidad del sistema a modelar evitando soluciones que no tienen viabilidad y por consiguiente reduciendo la complejidad computacional. En ciertos casos el conocimiento a priori es lo que permite obtener resultados válidos ya que orienta al proceso de búsqueda de una solución. El conocimiento a priori se denota como $F = (R, P)$ donde R representa un conjunto de los enlaces dirigidos requeridos dentro del grafo y P representa los enlaces dirigidos que están prohibidos. El conocimiento a priori F es consistente con el grafo G si todos los enlaces dirigidos indicados en R están dentro del grafo G y los enlaces en P no existen en el grafo G (Meek, 1995). El conocimiento a priori se incorpora al algoritmo de PC debido a que su inclusión modifica el comportamiento del algoritmo.

Consideraciones técnicas para la implementación del conocimiento a priori

Existen varias implementaciones del algoritmo de PC que permiten agregar conocimiento a priori. Pcalg implementa esta opción; el método principal del algoritmo *pc()* acepta como entrada dos matrices de adyacencia: *fixedGaps* y *fixedEdges*. En *fixedGaps* se especifican los enlaces bidireccionales que el resultado final no debe tener. En cambio, en *fixedEdges* se especifican los enlaces bidireccionales que se deben mantener en DAG resultante. Sin embargo, pcalg solo aplica este conocimiento a priori durante la obtención del *skeleton* (sólo para fijar o eliminar enlaces de antemano) y, debido a que *fixedGaps* y *fixedEdges* deben ser matrices simétricas (los enlaces fijos y prohibidos deben ser bidireccionales), no se puede prohibir una dirección. Esto quiere decir que sólo se puede agregar conocimiento de la existencia o no de ciertos enlaces, pero, de darse un enlace, no se puede restringir su dirección. Por otro lado, TETRAD (versión 6.2.0) cuenta con una implementación del conocimiento a priori que permite un control más granular de los enlaces dirigidos que se requieren o prohíben. Además, permite crear grupos temporales de dos tipos: entre los que se requiere que se creen enlaces dirigidos (Grupo1 → Grupo2) y entre los que está prohibido una de las direcciones del enlace (Grupo1 → Grupo2). Debido a todas las características que implementa, se decidió

utilizar como referencia TETRAD para implementar el conocimiento a priori en la implementación en C. El uso de TETRAD es posible ya que su código está disponible públicamente[23] bajo la licencia GNU GPL V2.

Representación del conocimiento a priori

El conocimiento a priori se implementa en base a PC-stable como una continuación del desarrollo. Los cambios que se introducen son opciones que se puede escoger para la ejecución del algoritmo; para ejecutar el algoritmo sin conocimiento a priori basta con no definir enlaces dirigidos requeridos o prohibidos. Para representar el conocimiento a priori se utilizó una matriz (*bKnowledge*) con las mismas dimensiones que la matriz de adyacencia *G*. La matriz se llena con los valores: "1" cuando se requiere un enlace dirigido y "2" cuando el enlace dirigido está prohibido (un caso de uso se muestra en Anexo A. Debido a que en la matriz se puede requerir o prohibir una sola dirección de un enlace, la matriz no es simétrica. En caso de que se requiera prohibir un enlace en su totalidad se prohíbe las dos direcciones del enlace ($X \rightarrow Y$ y $Y \rightarrow X$). La matriz *bKnowledge* tiene el mismo funcionamiento que una matriz de adyacencia, la posición de un elemento [*X*, *Y*] indica a qué nodos relaciona. Para aplicar conocimiento a priori se implementó cuatro métodos que consultan la matriz *bKnowledge*. Los métodos más simples son *isRequired()* e *isForbidden()* que indica si un enlace dirigido entre dos nodos es requerido o está prohibido, respectivamente. A partir de estos métodos se crean *isColliderAllowed()* e *isArrowpointAllowed()*. El método *isColliderAllowed()* comprueba si una tripleta de nodos se puede orientar como una *v-structure*. Para esto verifica que los enlaces a crear no estén prohibidos o que se requieran en la dirección contraria. En cambio el método *isArrowpointAllowed()* comprueba si el enlace dirigido no está prohibido o se requiere el enlace la dirección contraria. Estos cuatro métodos se usan durante todo el algoritmo donde se aplica el conocimiento a priori.

Inclusión de conocimiento a priori en *skeleton()*

El primer paso del algoritmo donde se agrega conocimiento a priori es *skeleton()*. Al inicio de *skeleton()* se comprueba si en *bKnowledge* existen enlaces bidireccionales prohibidos ("2" en las posiciones [*X, Y*] y [*Y, X*]). Si este fuera el caso, los enlaces se

[23] www.phil.cmu.edu/tetrad

marcan como eliminados en la matriz de adyacencia *G*. De esta manera se evita realizar pruebas de independencia condicional sobre nodos que ya se sabe que son independientes. Dentro de cada iteración de *skeleton()* se recuperan y procesan los enlaces que contiene la matriz de adyacencia *G.* Dado un enlace *X*─*Y*, se comprueba si los enlaces $X \rightarrow Y$ o $Y \rightarrow X$ son requeridos por conocimiento a priori, si uno o los dos enlaces se requieren, no se realizan pruebas de independencia condicional ya que es necesario mantener el enlace. En caso de que no se hayan definido enlaces requeridos, se continua con las pruebas de independencia condicional para determinar si *X*, *Y* son independientes. De cada enlace *X*─*Y*, se recupera el conjunto de nodos *adj(X*, *G)* que se utiliza para formar subconjuntos de condicionamiento. El proceso de recuperación de los nodos adyacentes a *X* se modifica para que considere el conocimiento a priori. Dado el nodo *K*, adyacente a *X* pero diferente a *Y*, se agrega al conjunto *adj(X, G)* si el enlace $X \rightarrow K$ no se ha definido como requerido y el enlace $K \rightarrow X$ no se ha definido como prohibido en el conocimiento a priori.

Orientación y eliminación de enlaces dirigidos con *pcOrientBK()*

El método que se introduce en la implementación de conocimiento a priori es *pcOrientBk()*, se ubica luego de *skeleton()* y se encarga de orientar los enlaces dirigidos requeridos y eliminar los prohibidos. El método recibe como entrada la matriz de adyacencia *G* (contiene el *skeleton*) y la matriz con el conocimiento a priori definido *bKnowledge*. En el método *pcOrientBk()* se toma en cuenta solo aquellos enlaces dirigidos que se han prohibido o se requieren de acuerdo al conocimiento a priori. Los enlaces bidireccionales (el enlace en sí) que se han prohibido ya se aplicaron durante la ejecución de *skeleton().* El método recorre todos los enlaces dirigidos definidos en el conocimiento a priori, tanto requeridos como prohibidos. Dado un enlace dirigido prohibido por conocimiento a priori P_i: $X \rightarrow Y$, si existe el enlace *X*─*Y* en el *skeleton,* entonces se agrega “0” en la posición [*X, Y*] de la matriz de adyacencia *G* (antes [*X*, *Y*] = [*Y*, *X*] = 1). En cambio, si el enlace dirigido *R*i: $X \rightarrow Y$ es requerido y existe un enlace X─Y en el *skeleton*, se agrega “0” en la posición [*Y, X*] de la matriz de adyacencia *G*. En los dos casos se elimina una de las direcciones del enlace bidireccional presente en el *skeleton*, i.e. después de este paso, la matriz de adyacencia *G* deja de ser simétrica. Con los cambios realizados a *skeleton()* (sección anterior) para incluir conocimiento a priori se asegura que todos los enlaces que tengan dirección estén presentes en el *skeleton*.

Conocimiento a priori en la orientación de tripletas

El siguiente paso del algoritmo es la orientación de las tripletas como *v-structures* en el método *udag2pdagRelaxed()*. Debido a que la orientación de las tripletas involucra introducir direcciones, es necesario controlar si tales direcciones están permitidas. El método recupera e itera sobre todas las tripletas candidatas a *v-structure* que contiene la matriz de adyacencia *G* y comprueba si se debe orientar, i.e. la tripleta no es infiel. Una vez se ha determinado que la tripleta $X{-}Y{-}Z$ se debe orientar, se utiliza el método *isColliderAllowed()* para comprobar si se pueden orientar los enlaces $X{-}Y$ y $Y{-}Z$ a $X \rightarrow Y \leftarrow Z$. Al iterar sobre todas las tripletas de *G* termina el método. A continuación, la orientación de los enlaces restantes se realiza con las reglas de Meek (Figura 3). Los métodos que representan las reglas de Meek (*rule1()*, *rule2()* y *rule3()*) buscan determinados patrones de nodos y orientan sus enlaces se acuerdo a la regla. Incluir conocimiento a priori en estas reglas consiste en comprobar si la orientación a realizar de los enlaces está permitida por conocimiento a priori. En cada método al encontrar el patrón de la regla se utiliza el método *isArrowpointAllowed()* para comprobar si la orientación a realizar del enlace está permitido. En caso de que no esté permitido el método de la regla no orienta el enlace y continua su proceso.

3.3 Paralelización de los algoritmos

Existen varios tipos de sistemas que permiten la ejecución de un algoritmo en paralelo, cada uno con un modelo de ejecución. Los tipos de sistemas predominantes son: con memoria compartida y distribuida. Un sistema con memoria compartida cuenta con un solo bloque de memoria que los procesadores pueden acceder (Figura 4). Cada procesador del sistema accede a los mismas direcciones de memoria con la misma velocidad (Chapman, Jost, & Pas, 2008). Los procesadores interactúan y se sincronizan entre sí con variables compartidas. El modelo de paralelización que utiliza se denomina dividir/unir (*fork/join*), al inicio de la ejecución del algoritmo paralelo con memoria compartida existe un solo hilo (MASTER). MASTER se encarga de realizar las operaciones en serie y crear nuevos hilos para las operaciones en paralelo. Cuando no se requieren, los hilos se suspenden o matan (Quinn, 2008). En cambio, un sistema con memoria distribuida es aquel que cuenta con varios procesadores, cada uno con un bloque de memoria privada (Figura 5). El modelo de paralelización se denomina de paso de mensajes (*message-passing*).

Al ejecutar un algoritmo paralelo con memoria distribuida se crean varios procesos. Los procesos se mantienen vivos durante la ejecución del algoritmo. La coordinación de las procesos se realiza mediante el paso de mensajes por un canal de comunicación (Quinn, 2008). Un ejemplo de este sistema es una granja de servidores, cada servidor tiene sus recursos (e.g. procesador, memoria) y se conectan entre sí mediante una conexión de red. Existen sistemas que cuentan con varios procesadores y módulos de memoria que calificarían como sistemas distribuidos pero, permiten la ejecución con memoria compartida, esto gracias a que el sistema operativo crea una abstracción del hardware y muestra a los programas un solo procesador (con varios núcleos) y un solo espacio de memoria.

Dentro de los modelos de ejecución se habla de hilos y procesos, es importante diferenciarlos ya que el impacto que tienen en el sistema es diferente. Un proceso es una abstracción de un programa que se está ejecutando. Tradicionalmente un proceso tiene un solo hilo que ejecuta las tareas secuencialmente. Lo hilos son las unidades de ejecución del proceso, en cambio, los procesos son las unidades de reserva de recursos del sistema (Haldar, 2015). Debido a esto un proceso no ejecuta un programa, solo mantiene un ambiente de ejecución para los hilos, estos pertenecen a un solo proceso y no son visibles para otros procesos. La paralelización de un programa se puede realizar creando varios hilos dentro de un proceso o creando más procesos del mismo algoritmo. Procesos e hilos son entidades independientes de ejecución, cada proceso tiene un ambiente de ejecución y cada hilo tiene su estado de ejecución. Sin embargo, los hilos de un proceso se ejecutan en un espacio compartido de memoria. La ejecución en paralelo con memoria compartida es la que crea varios hilos dentro de un proceso, puede realizar esto debido a que los recursos del sistema están centralizados y accesibles como un solo bloque. La ejecución con memoria distribuida es la que crea varios procesos, ya que los recursos del sistema están distribuidos. El número de procesos e hilos que se creen dependerá de los parámetros de la ejecución en paralelo. La creación de hilos tiene un menor impacto dentro del sistema ya que utiliza el ambiente de ejecución del proceso, el impacto es mayor al crear un proceso ya que crea un nuevo ambiente de ejecución con su hilo principal.

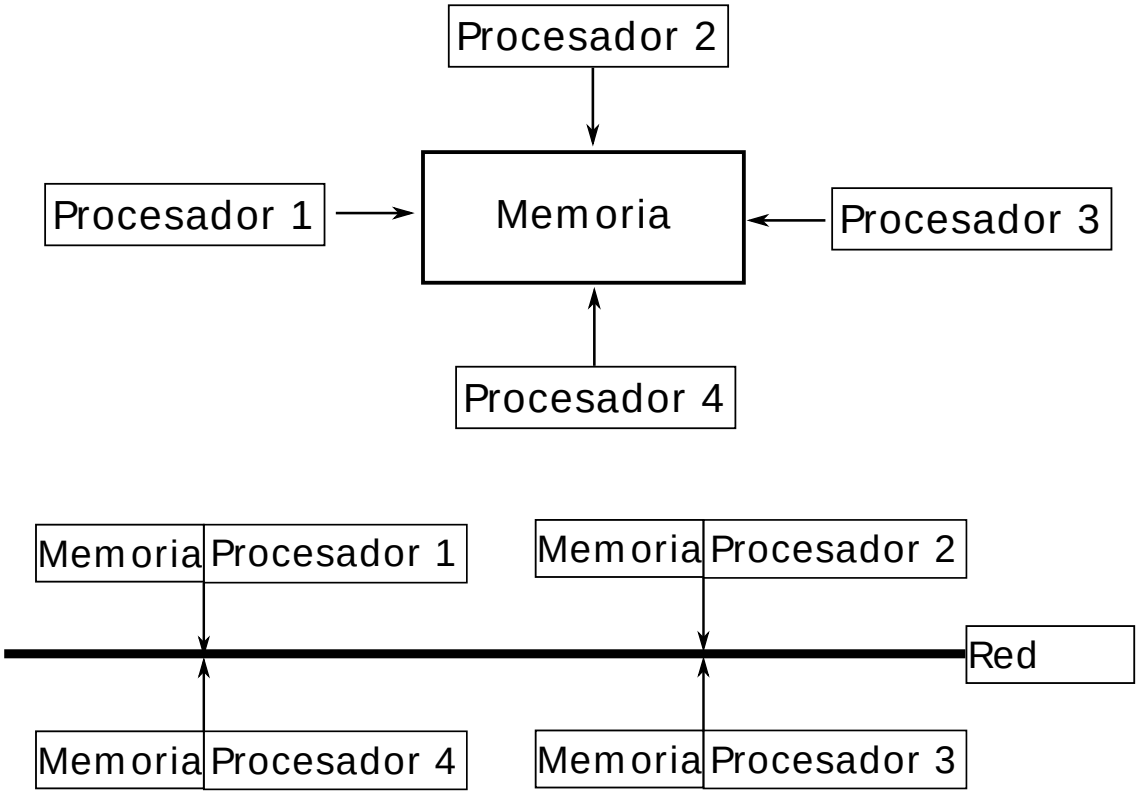


Figura 4: Esquema de la organización de un sistema con memoria compartida.

Figura 5: Esquema de la organización de un sistema con memoria distribuida

El algoritmo de PC original no se puede paralelizar debido a que depende del orden en el que está dispuesto las variables en el conjunto de datos y el que se realizan las pruebas de independencia condicional. Las versiones del algoritmo de PC que se pueden paralelizar son: PC-stable y CPC. La implementación descrita hasta esta sección contiene los cambios que introducen las dos versiones. Por lo que, a partir de este punto en el documento se habla de un solo algoritmo. De igual forma la paralelización se realiza sobre diferentes partes de un mismo algoritmo. Como se describió en la sección 2.3, no todos los pasos del algoritmo se pueden paralelizar, debido a que dependen del orden en el que se realicen sus operaciones internas. De la implementación realizada se pueden paralelizar 3 pasos: (1) la obtención de la matriz de correlación con el método *correlationMatrix()*, (2) obtención del *skeleton* con el método *skeleton()* y (3) procesamiento de las tripletas con *pc_cons_intern().* La paralelización de (1) es posible debido a que la obtención del coeficiente de correlación de una variable no depende del coeficiente de correlación de las demás variables. La paralelización de (2) es posible gracias a los cambios que introduce PC-stable, y la paralelización de (3) es debido a que las pruebas de independencia

condicional que realiza CPC no requieren de un orden específico para obtener los mismos resultados. Las estrategias para la paralelización se describieron en la sección 2.3. Usando la misma teoría para la paralelización del algoritmo de PC, se utilizaron los dos enfoques para paralelizar el algoritmo: con memoria compartida y con memoria distribuida.

3.3.1 Paralelización con memoria compartida

La implementación en paralelo con memoria compartida se realizó usando OpenMP[24]. OpenMP es una API para la programación en paralelo en sistemas con memoria compartida. OpenMP fue creado con el objetivo de proveer una abstracción de forma que la paralelización con hilos sea más fácil. OpenMP está compuesto de un conjunto de directivas para el compilador y una librería de funciones de soporte que trabajan en conjunto con C, C++ y Fortran (Quinn, 2008). Usando las directivas de OpenMP se puede paralelizar un algoritmo secuencial sin realizar grandes cambios en el código. OpenMP permite crear dinámicamente hilos a demanda dentro de la implementación, por lo que permite tener partes secuenciales y paralelas. Se utiliza la directiva *omp parallel* para indicar al compilador que un bloque de código se va a ejecutar en varios hilos. Al inicio de cada bloque paralelizado se especifican las variables compartidas entre los hilos, esto se realiza combinando la directiva *omp parallel* con la cláusula *shared()*: *omp parallel shared()*, por defecto todas las variables son privadas. Las variables compartidas permiten sincronizar los hilos y que estos accedan a los mismos datos. Los bloques de código que se paralelizan usualmente son bucles en el que su número de iteraciones está bien definido (*for*) y no contiene sentencias que provoquen la terminación prematura del bucle (e.g. *break*, *return*). Dentro de *omp parallel shared()* se utiliza la directiva *omp for* para paralelizar un bucle *for*, de esta manera cada iteración del bucle se realiza en un hilo. OpenMP se encarga del ciclo de vida de los hilos. El número de hilos a utilizar se especifica con el método *omp_set_num_threads()*, aunque en la mayoría de los casos no es necesario ya que OpenMP determina automáticamente el número de hilos a utilizar de acuerdo al sistema en el que se ejecuta.

Paralelización del cálculo de la matriz de correlación

[24] http://www.openmp.org/

Lo primero que se paraleliza del algoritmo es la obtención de la matriz de correlación con el método *correlationMatrix()*. El método se compone principalmente de un bucle *for* que recorre las variables del conjunto de datos. En cada iteración de este bucle se calculan los coeficientes de correlación relacionados con la variable de la iteración. La paralelización se realiza sobre este bucle y es posible debido a que para obtener los coeficientes de correlación solo se necesita el conjunto de datos de entrada sin depender de ningún otro resultado. Las iteraciones del bucle se reparten entre varios hilos. A cada hilo se le asigna una variable (iteración del bucle) y obtiene los coeficientes de correlación relacionados con la variable. Las variables que se comparten entre los hilos son: la matriz de correlación (donde se guardan los coeficientes calculados) y la matriz con el conjunto de datos de entrada. Debido a que en cada iteración del bucle se obtienen diferentes coeficientes de correlación, la escritura de los resultados se realiza en diferentes partes de la matriz de correlación, por lo que no se presentan condiciones de carrera. Debido a la estructura de *correlationMatrix()* su paralelización no requirió grandes cambios.

Paralelización de la obtención del *skeleton*

El siguiente paso del algoritmo de PC que se paraleliza es la obtención del *skeleton* con *skeleton().* Para implementar la paralelización se tomó como referencia el trabajo que Le et al. (2016) realiza al paralelizar pcalg. El método *skeleton()* está compuesto principalmente de dos bucles anidados. El bucle exterior (*do-while*) indica la cardinalidad (nivel) de los conjuntos de condicionamiento usados en las pruebas de independencia condicional. En este bucle se recupera el conjunto de enlaces que contiene la matriz de adyacencia *G* y a continuación se ejecuta el bucle interno (*for*) que recorre el conjunto de enlaces. En el bucle interno (for), dado un enlace *X*—*Y*, se forma conjuntos de condicionamiento en base a *adj(X*, *G)* y se realiza pruebas de independencia condicional para determinar si los nodos del enlace son independientes. La paralelización se realizó sobre el bucle interno gracias a los cambios que introduce PC-stable. Una de las mejoras que presenta Le et al. (2016) y que se implementó, consiste en juntar los enlaces que tienen los mismo nodos. Esto evita realizar pruebas de independencia condicional innecesarias ya que elimina la posibilidad de que a dos hilos se les asigne los enlaces *X*—*Y* y *Y*—*X,* respectivamente. Las variables que se comparten entre los hilos son: la matriz de adyacencia *G*, la copia inicial de *G*, la matriz de *sepsets*, el nivel de la iteración y el valor de significancia *alpha*. A cada hilo se le asigna un enlace *X*—*Y*, forma conjuntos de condicionamiento en base a *adj(X, G)* y realiza

pruebas de independencia condicional condicionando en cada conjunto formado. En caso de que los nodos *X, Y* no sean independientes, el proceso se repite para el enlace *Y—X*. Los resultados que obtiene cada hilo se escriben sobre la matriz de adyacencia *G* y la matriz de *sepsets*. Debido a que los hilos trabajan con diferentes enlaces, para escribir los resultados acceden a diferentes partes de las variables por lo que no se crean condiciones de carrera. La paralelización se realiza dentro de cada iteración del bucle externo de *skeleton()* una vez ha recuperado el conjunto de enlaces.

Paralelización del procesamiento de tripletas

El último paso de PC que se paralelizó es el procesamiento de las tripletas candidatas a *v-structures* con el método *pc_cons_intern()*. En resumen, lo que realiza este método es: recuperar las tripletas que contiene la matriz de adyacencia *G* y realizar pruebas de independencia condicional sobre cada tripleta para determinar si se deben orientar. Para realizar las pruebas de independencia condicional de cada tripleta, *pc_cons_intern()* invoca al método *checkTriple()*. Es sobre parte de las tareas que realiza *checkTriple()* que se aplicó la paralelización. El método *checkTriple()* al recibir la tripleta *X—Y—Z* realiza los siguientes pasos: (1) forma los conjuntos de condicionamiento en base a *adj(X*, *G)* y *adj(Z*, *G)*, (2) realiza pruebas de independencia condicional condicionando en cada conjunto formado y (3) con los resultados de las pruebas determina si la tripleta se debe orientar. El bloque de paralelización es el paso (2) de *checkTriple()*, que se compone de un bucle (*for*) que recorre cada conjunto de condicionamiento y realiza una prueba de independencia condicional. La prueba de independencia condicional se realiza para determinar si los nodos *X, Z* son independientes dado un conjunto de condicionamiento (conjunto separador), si este es el caso se comprueba si *Y* se encuentra en el conjunto separador. Las variables que llevan los registros de las pruebas son: *countTrue, countFalse y countTotal*, que indican las veces que el nodo *Y* está en los conjuntos separadores, las veces que no y el total de los conjuntos separadores, respectivamente. Las variables que se comparten entre los hilos son: la matriz de correlación *G*, el valor de significancia *alpha*, *countTrue*, *countFalse*, *countTotal* y los conjuntos de condicionamiento. Al paralelizar el bucle del paso (2) sus iteraciones se reparten entre varios hilos. Al escribir los resultados de las pruebas es posible que se presenten condiciones de carrera. Se utilizó la directiva *omp atomic* para que la escritura de los resultados sea una operación atómica y se realice secuencialmente cuando haya varios intentos de escritura simultáneos. Una

vez ha finalizado la paralelización del paso (2), *checkTriple()* continua con el paso (3) secuencialmente. La paralelización se realiza cada vez que se invoca a *checkTriple()*.

La paralelización del algoritmo se realizó como una opción más para la ejecución del algoritmo. Debido a que solo partes del algoritmo se ejecutan en paralelo es posible agregar condiciones a cumplir para que el código se ejecute en paralelo. Cuando el número de hilos es menor o igual a 1 todo el algoritmo se realiza en serie. Aunque es posible ejecutar un bloque paralelizado con un solo hilo, no es recomendable ya que OpenMP agrega operaciones (e.g. inicio de hilos, sincronización, manejo de directivas) que no están presentes en el código en serie ("Advanced OpenMP Topics", 2012). La conclusión es que un bloque de código paralelizado que se ejecuta con un solo hilo es más lento que el mismo código en serie.

3.3.2 Paralelización con memoria distribuida

La implementación con memoria distribuida se realizó utilizando Open MPI[25], una implementación de código abierto de *Message Passing Interface* (MPI) desarrollada y mantenida por un consorcio de académicos, investigadores y socios de la industria. MPI es una especificación estándar para el paso de mensajes entre procesos que soporta programación en paralelo. Al ejecutar un algoritmo implementado con Open MPI se crean un conjunto de procesos, cada proceso es una entidad independiente con su ambiente de ejecución y espacio de memoria. Los procesos tienen un rango que les identifica, el proceso con rango 0 se denomina MASTER y demás procesos se denominan trabajadores. MASTER es el que coordina los procesos trabajadores. Implementar la paralelización con Open MPI requiere de grandes modificaciones en el código del algoritmo a paralelizar. Las modificaciones al código del algoritmo requieren definir el comportamiento del proceso MASTER y de los procesos trabajadores. El comportamiento de proceso MASTER incluye ejecutar las partes secuenciales del código, preparar y enviar las tareas a los procesos trabajadores, recuperar los resultados parciales de cada proceso y ensamblar los resultados de forma que el algoritmo pueda continuar. En cambio, un proceso trabajador se encarga de recibir datos del MASTER con los que debe trabajar, realizar las operaciones sobre los datos recibidos y enviar los

[25] https://www.open-mpi.org/

resultados obtenidos a MASTER. A diferencia de la implementación con memoria compartida, los procesos se crean al inicio del algoritmo y se mantienen vivos durante la ejecución. No es posible la creación dinámica de procesos. Los procesos trabajan en coordinación con MASTER por lo que se utiliza el paso de mensajes para coordinación y paso de información. Debido a esto la sincronización entre MASTER y los procesos trabajadores es un proceso complejo.

Métodos y funciones usadas durante la implementación

Open MPI cuenta con un conjunto de métodos necesarios para la paralelización. A continuación, se describe los que se usaron para la implementación:

- *MPI_Init():* para iniciar la paralelización con MPI.
- *MPI_Finalice():* para terminar la paralelización con MPI.
- *MPI_Comm_rank()*: recupera el rango asignado al proceso que ejecuta el método.
- *MPI_Comm_size():* recupera el número de procesos con los que se ejecuta el algoritmo.
- *MPI_Bcast():* distribuye datos desde MASTER hacia todos los trabajadores. Es la manera más eficiente de enviar información.
- *MPI_Recv():* usado por MASTER para recuperar información desde los trabajadores, usualmente los resultados.
- *MPI_Send():* usado para enviar información entre trabajadores.

Con los métodos de Open MPI se puede enviar varios tipos de datos (e.g *int*, *char*, *float*) y estructuras (e.g. matrices, vectores). En la implementación en serie del algoritmo de PC se utilizan listas enlazadas, estructura que no soporta Open MPI, lo que requirió crear un vector con los datos de la lista enlazada antes de enviarla. Conversiones similares se realizaron en los casos que se requiere enviar parte de los datos de una matriz.

Preparación antes de paralelizar los métodos

El primer gran cambio que se realiza al algoritmo consiste en identificar qué partes del código corresponden al proceso MASTER y qué partes corresponden a los procesos trabajadores. Si no se diferencia el código todos los procesos realizan las mismas tareas y lo que se logra son varios procesos del mismo algoritmo que

realizan las mismas tareas. Para obtener el rango de un proceso se utiliza *MPI_Comm_rank()* y se utilizan estructuras condicionales para especificar qué proceso debe ejecutar determinado código. Existen casos en los que se requiere que todos los trabajadores realicen las mismas tareas. En nuestra implementación esto se presenta al inicio del algoritmo donde los procesos deben crear las variables generales para todos los métodos. En este modelo de paralelización todos los procesos tienen disponible el mismo código, sin embargo, el código que ejecutan depende del rango de proceso. La secuencia general de eventos que sigue el algoritmo implementado es la siguiente:

1. MASTER inicia sus tareas en serie necesarias para asignar tareas a los trabajadores.
2. Los trabajadores se mantienen en espera de instrucciones de MASTER.
3. MASTER reparte tareas hacia los trabajadores.
4. Los trabajadores reciben las instrucciones, realizan las operaciones necesarias y devuelven los resultados a MASTER.
5. MASTER recibe los resultados y el ciclo se repite hasta que el algoritmo termina.

En nuestra implementación, al paralelizar un conjunto de tareas, todos los procesos (incluido MASTER) realizan tareas del conjunto. Una vez MASTER ha concluido con sus tareas asignadas espera por los resultados de los trabajadores. La paralelización se realizó sobre tres pasos del algoritmo: (1) obtención de la matriz de correlación, (2) obtención de *skeleton* y (3) procesamiento de las tripletas. Los motivos por los que (1), (2) y (3) se pueden paralelizar se explicaron en la sección 2.3.

Paralelización de *correlationMatrix()*

El cálculo de la matriz de correlación se realiza con el método *correlationMatrix()*. Al inicio del algoritmo, todos los procesos leen el archivo con el conjunto de datos de entrada. De esta manera se evita enviar grandes cantidades de información entre trabajadores. El método *correlationMatrix()* se compone principalmente de un bucle que recorre las variables del conjunto de datos. En cada iteración se calcula los coeficientes de correlación relacionados con la variable. La paralelización es posible debido a que para calcular el coeficiente de correlación solo es necesario el conjunto de datos de entrada. La paralelización se realizó de forma que cada

proceso calcule los coeficientes de correlación de un subconjunto del total de las variables. MASTER envía a cada proceso dos números que indican el rango de las variables de las que tiene que calcular los coeficientes de correlación. Una vez se han calculado todos los coeficientes de correlación, MASTER recolecta y une los resultados. A continuación, MASTER con el método *MPI_Bcast()*, envía hacia los trabajadores la matriz de correlación resultante. Es importante que cada proceso tenga esta matriz ya que se utiliza para los siguientes pasos del algoritmo. La paralelización de *correlationMatrix()* es la más simple que se realizó en el modelo con memoria distribuida ya que no necesita de grandes cambios en el código. La ventaja que tiene *correlationMatrix()* para su paralelización es que todos los trabajadores ya cuentan con la información necesaria para trabajar y solo necesitan del rango de variables de las que tienen que calcular el coeficiente de correlación.

Paralelización de *skeleton()*

La obtención de *skeleton* se realiza con el método *skeleton()*. Su paralelización involucra crear el método *skeletonWorker*() y modificar *skeleton()* para repartir las tareas y recuperar los resultados. El método *skeletonWorker()* es el que ejecuta los procesos trabajadores, es una versión simplificada de *skeleton()* que contiene solo las operaciones que cada trabajador debe realizar. A grandes rasgos *skeleton()* se organiza en iteraciones, en cada iteración: recupera los enlaces que existen en la matriz de adyacencia *G,* por cada enlace *X—Y* define sus conjuntos de condicionamiento a partir de *adj(X, G)* y realiza pruebas de independencia condicional para determinar si los nodos del enlace son independientes. Al igual que en la paralelización con memoria compartida, al recuperar los enlaces se juntan aquellos que contienen los mismos nodos, *X—Y* y *Y—X*. La paralelización consiste en que cada trabajador tenga un subconjunto de los enlaces sobre los que realizar pruebas de independencia condicional. Las tareas secuenciales que se realizan antes y después de la paralelización las realiza MASTER. Debido a la cantidad de información que MASTER tiene que enviar para realizar la paralelización, se implementó un control. El control permite especificar un número mínimo de enlaces con el que cada trabador debe ser asignado. En caso de que no sea posible cumplir esta condición, *skeleton()* se realiza en serie. Esto se implementó con el objetivo de que la ejecución en paralelo no sea más lenta que la ejecución en serie. Al inicio de la etapa en paralelo de *skeleton()* cada trabajador recibe: el nivel de la iteración, las modificaciones de la matriz de adyacencia *G* y el subconjunto de los enlaces asignado. En cada iteración los trabajadores deben tener la matriz de adyacencia *G*

actualizada ya que se utiliza para consultar el vecindario de nodos y formar conjuntos de condicionamiento. Por este motivo se envían las modificaciones que MASTER tiene de la matriz. La matriz de adyacencia inicia con un grafo completo, una modificación comprende la falta de un enlace en la matriz (“0”). Las modificaciones se envían en una lista que contiene las direcciones (*X, Y*) de la matriz donde no hay enlaces. Una vez MASTER ha repartido las tareas entre los trabajadores, realiza pruebas de independencia condicional sobre el subconjunto de los enlaces que se le ha asignado. Las pruebas de independencia condicional las realiza de la misma forma que los trabajadores en *skeletonWorker()*, siguiendo los mismos pasos y con una copia de las mismas variables.

Trabajo de *skeletonWorker()*

El método *skeletonWorker()* recibe los datos indicados de MASTER. Procede a actualizar la matriz de adyacencia *G* y su copia, con la lista de cambios recibidos. La copia de *G* se utiliza para mantener el vecindario de nodos intacto durante la iteración (PC-stable). Por cada enlace *X*─*Y* del subconjunto de enlaces asignado: se definen los conjuntos de condicionamiento a partir de *adj(X, G)* y se realizan las pruebas de independencia condicional. De las iteraciones se mantiene la variable *done* que inicia con el valor “0” y toma el valor “1” cuando encuentra una situación en la que el número de nodos de *adj(X, G)* sea menor al nivel de la iteración. Los resultados de las pruebas de independencia condicional se escriben en la matriz de la adyacencia *G* y en la matriz con los *sepsets*. Cuando se han procesado todos los enlaces, los resultados obtenidos deben enviarse a MASTER. Lo que se envía a master comprende: los cambios realizados en la matriz de adyacencia *G,* los cambios en la matriz de sepsets y la variable *done*. La matriz de adyacencia *G* se compara con su copia estática y se crea una matriz de dos columnas con los enlaces que se eliminaron (*X*─*Y*). Un proceso similar se realiza con la matriz de sepsets. Como ya se indicó, esta es una matriz de listas enlazadas con los valores de los conjuntos de condicionamiento. Debido a que Open MPI no permite enviar listas enlazadas, los cambios realizados en esta matriz se trasladan a una matriz de 3 columnas. Las dos primeras indican una posición de la matriz (*X, Y*) y la tercera columna contiene un valor de la lista enlazada ubicada en la posición [*X*, *Y*] de la matriz de sepsets. Los sepsets que se consideran para enviar sus cambios son los que pertenecen al subconjunto de enlaces con los que se trabajó *skeletonWorker()*.

MASTER, juntar los resultados

Cuando MASTER ha procesado el subconjunto de enlaces que se le ha asignado inicia un bucle para recuperar los resultados de cada proceso trabajador. Lo primero que recibe es la lista de cambios en la matriz de adyacencia *G* que tiene cada trabajador. Los cambios se copian en la matriz de adyacencia *G* de MASTER. Lo siguiente que recibe es los cambios realizados en la matriz de *sepsets*, estos también se copian en la matriz de *sepsets* que mantiene MASTER. Finalmente recibe la variable *done* del trabajador. Esto se repite para todos los procesos trabajadores. En el caso de que todos los trabajadores tengan la variable done con el valor "1", *skeleton()* termina. Caso contrario todo el proceso se repite desde que MASTER recupera el conjunto de enlaces de la matriz de adyacencia *G*. Una vez se obtiene el *skeleton* lo siguiente que se paraleliza es el procesamiento de las tripletas que contiene *skeleton.*

Paralelización de *pc_cons_intern()*

El procesamiento de tripletas para determinar si se deben orientar se introduce en CPC y se realiza en el método *pc_cons_intern()*. Lo que realiza *pc_cons_intern()* se resume en: determinar y recuperar las tripletas candidatas a *v-structure* que contiene *skeleton* y por cada tripleta realizar pruebas de independencia condicional para determinar si se deben orientar. Las pruebas de independencia condicional se realizan en el método *checkTriple()* que recibe una tripleta. Es en este método donde se introduce la paralelización. En *checkTriple()*, dada la tripleta *X*─*Y*─*Z*, se forman los subconjuntos de condicionamiento basados en los nodos de *adj(X*, *G)* y de *adj(Z*, *G)*. La paralelización consiste en repartir entre los procesos trabajadores los conjuntos de condicionamiento. Se creó el método *checkTripleWorker()* que ejecuta los procesos trabajadores. Debido a la cantidad de información que MASTER necesita enviar hacia cada proceso trabajador, la paralelización se realiza solo si el número de conjuntos de condicionamiento es mayor a cierto umbral. Esto se implementó con el objetivo de evitar que el proceso en paralelo sea más lento que su versión en serie. Antes de repartir los conjuntos de condicionamiento, en *pc_cons_intern()*, MASTER envía hacia los trabajadores el número de tripletas encontradas. Este valor indica el número de veces que se invoca *checkTriple()* por lo que es el número de veces que *checkTripleWorker()* debe esperar información de MASTER. Si se ha determinado que se paraleliza, MASTER envía hacia cada

proceso trabajador la tripleta y parte de los conjuntos de condicionamiento. Ya repartidas las tareas a los trabajadores, MASTER realiza las pruebas de independencia condicional con los conjuntos de condicionamiento que se le ha asignado.

Dentro de *checkTripleWorker()* se recibe el número de veces que los procesos trabajadores deben esperar datos desde MASTER para lo cual se crea un bucle *for*. En cada iteración se recibe una confirmación de que se va a paralelizar, si es negativa, se salta a la siguiente iteración. En el caso de que sí se paralelice, se recibe la tripleta *X–Y–Z* y los conjuntos de condicionamiento. Con cada conjunto de condicionamiento se realiza una prueba de independencia condicional para determinar si *X* y *Z* son independientes, de esta manera se identifican los conjuntos separadores. La información que MASTER requiere de las pruebas de independencia condicional es el número de veces que el nodo *Y* se encuentra dentro de los conjuntos separadores. Para lo cual, los registros que se mantiene de las pruebas son: *countTrue*, *countFalse* y *countTotal*, que indican el número de veces que *Y* está en los conjuntos separadores, que no lo está y el número de conjuntos separadores, respectivamente. Al finalizar todas las pruebas de independencia condicional el proceso trabajador envía hacia MASTER los registros de las pruebas. El proceso que sigue un trabajador para realizar las pruebas de independencia condicional es el mismo que sigue MASTER para realizar las pruebas con los conjuntos de condicionamiento asignados. Una vez MASTER finaliza las pruebas de independencia condicional, inicia la recuperación de los resultados de cada trabajador. De cada trabajador recibe: *countTrue*, *countFalse* y *countTotal* que se suman a las variables que mantiene MASTER. Al finalizar la paralelización, *checkTriple()* continua secuencialmente para determinar si, de acuerdo a los resultados, la tripleta se debe orientar. Esto se repite para cada tripleta identificada.

Capítulo 4: Pruebas de Rendimiento

El rendimiento del algoritmo implementado fue comprobado ejecutándolo N veces con diferentes conjuntos de datos y tomando su tiempo de ejecución (*wall-time*). Las pruebas se realizaron de manera que las condiciones de ejecución del algoritmo sean las mismas en todos los casos con el objetivo de obtener datos comparables. Debido a que se implementó varios modelos de ejecución del algoritmo (serie y paralelo) se utilizaron las siguientes configuraciones para las pruebas de rendimiento:

- **En serie**: las operaciones de los algoritmos se realizan de forma secuencial.
- **En paralelo openMP**: utiliza memoria compartida y se realizan pruebas con diferente número de hilos: 2, 3, 4, 5, 6, 7, 8, 10, 12, 14 y 16.
- **En paralelo open MPI:** utiliza memoria distribuida y se ejecutan con diferente número de procesos: 2, 3, 4, 5, 6, 7, 8, 10, 12, 14 y 16. La paralelización se realizó usando la siguiente configuración:
    - En el primer control (*skeleton*) se requiere que haya al menos 50 tareas para cada trabajador.
    - En el segundo control (procesamiento de tripletas) se requiere que haya al menos 20 tareas para cada trabajador.

En las configuraciones en paralelo se especifica el número de hilos (memoria compartida) y procesos (memoria distribuida) para ejecutar el algoritmo. Desde este punto del documento se usará la denominación “número de núcleos” para hacer referencia a la cantidad de hilos o procesos con el que se ejecutan los algoritmos. Esto con el objetivo de que las gráficas de los resultados tengan una denominación común para compararlas.

Puntos de control Open MPI

El número de operaciones mínimas para el procesamiento en paralelo con memoria distribuida (open MPI) se eligió de manera empírica. Los números de tareas utilizados (50 y 20) se justifican al verificar la cantidad de operaciones a paralelizar en cada iteración de los puntos de control del algoritmo implementado. En los dos controles (*skeleton* y procesamiento de tripletas) el número mínimo de operaciones por trabajador es fijo durante el algoritmo. En el **primer control** (*skeleton*) para que

un conjunto de operaciones se realice en paralelo se necesita de al menos i*50 operaciones, donde “i” es el número de núcleos usados en la configuración. Tomando como ejemplo un conjunto de datos sintéticos (sección 4.1) de 1250 variables se verificó cuál fue el número de operaciones (enlaces) con los que se encuentra la implementación paralela en el primer control. En la Tabla 3 se muestra el número de operaciones (enlaces) y su frecuencia de aparición en cada iteración del algoritmo. Lo más destacable es que en cada iteración hay más de 4000 operaciones a distribuir, lo que garantiza que en este conjunto de datos todas las operaciones del primer control se realicen en paralelo. Los resultados obtenidos no se pueden generalizar para todos los conjuntos de datos ya que el número de operaciones depende de factores externos al algoritmo, e.g. la forma en la que están relacionadas las variables y el número de variables del conjunto de datos. Un muy buen ejemplo de cómo varía el tiempo de cómputo con un mismo número de variables pero que representan dinámicas diferentes de un sistema se puede encontrar en (Ebert-Uphoff & Deng, 2015b).

Tabla 3: Frecuencia de operaciones a distribuir del primer paso de PC (*skeleton*). Datos tomados de un conjunto de datos sintético de 1250 variables con 5000 muestras

Operaciones a distribuir (Enlaces)	Frecuencia	Frecuencia acumulada
4496	5	5
4497	1	6
4499	1	7
4503	1	8
4534	1	9
4611	1	10
4779	1	11
5218	1	12
6044	1	13
7678	1	14
780625	1	15

Tabla 4: Frecuencia del número de operaciones a distribuir en un conjunto de datos sintético de 1250 variables con 5000 muestras. El número de operaciones se obtuvo del procesamiento de tripletas introducido en CPC.

Operaciones a distribuir (Conjuntos de condicionamiento)	Frecuencia	Frecuencia Acumulada
2	56	56
4	197	253
8	1049	1302
16	2755	4057
32	3792	7849
64	7214	15063
128	9350	24413
256	13613	38026
512	9818	47844
1024	7716	55560
2048	4223	59783
4096	1525	61308
8192	541	61849
16384	104	61953
32768	119	62072
Total de tareas		43176444

La Figura 6 muestra el porcentaje de tareas paralelizadas para los 6 conjunto de datos sintéticos (sección 4.1) usado en las pruebas de rendimiento. Los conjuntos tienen de 250 a 1500 variables, entre cada conjunto hay un salto de 250 variables y cada variable tiene 5000 muestras. Por cada gráfica se muestran dos barras que indican el porcentaje de paralelización que se obtiene al usar 50 y 20 como número mínimo de operaciones a paralelizar en el primer (*skeleton*) y segundo control (procesamiento de tripletas) respectivamente. Por cada conjunto de datos se muestra el porcentaje de paralelización que se obtiene al usar configuraciones con diferente número de núcleos. Lo primero que se observa es que, en 5 de los 6 conjuntos de datos se paraleliza el 100% de las operaciones del primer control. Solo con el conjunto de datos de 250 variables, en su configuración con 6 núcleos, el

porcentaje de paralelización decrece hasta el 95%, llegando hasta el 94% con 8 núcleos. Para que en todas las configuraciones la paralelización sea del 100%, se debería haber usado 34 como valor mínimo para el primer control. Esto no garantiza que el tiempo de ejecución sea menor ya que al realizar más tareas paralelizadas se necesita más coordinación entre los núcleos. Por esta razón 50 fue tomado como el número de tareas mínimo adecuado para que se realice la paralelización.

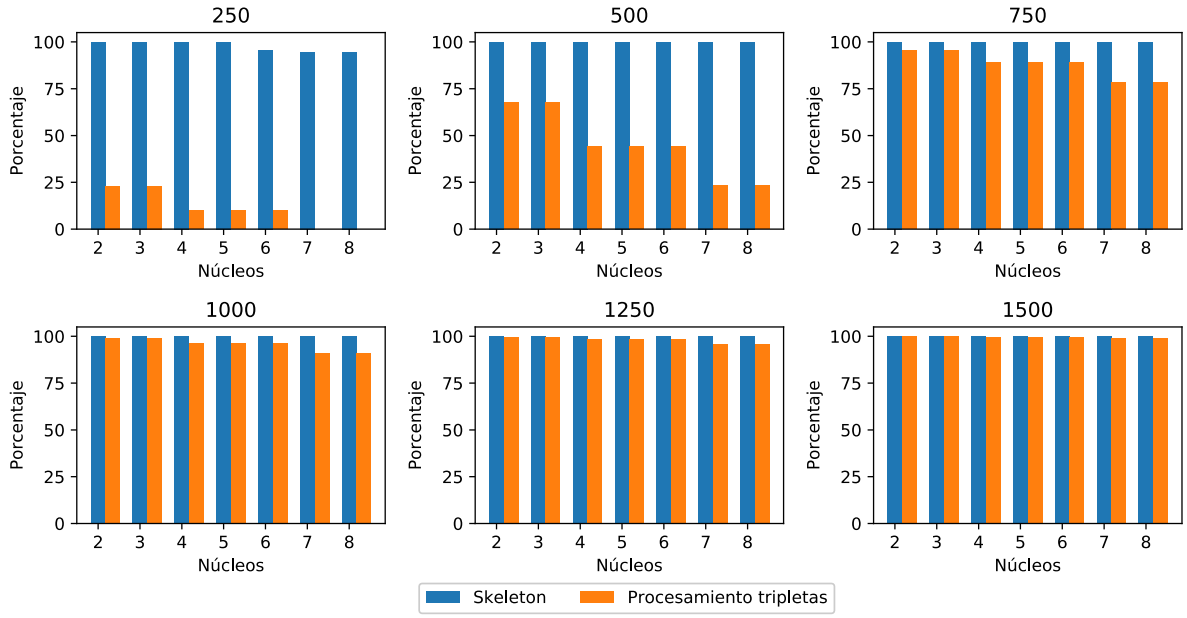


Figura 6: Porcentaje de tareas paralelizadas para cada conjunto de datos sintéticos usando la implementación paralela del algoritmo de PC-stable con memoria distribuida (Open MPI). Se muestra el porcentaje para el primer control (*skeleton*) y para el segundo control (procesamiento de tripletas).

El número mínimo de operaciones para el **segundo control** es más difícil de definir debido a la variabilidad que existe en el número de operaciones a distribuir. Para verificar esto se tomó como ejemplo el conjunto de datos sintético de 1250 variables. De éste se obtuvo el número de operaciones (conjuntos de condicionamiento) que se presenta con cada tripleta de nodos del segundo control del algoritmo. Lo que se observa es que, en contraste con las operaciones del primer control (Tabla 3), los números de operaciones que se presentan son más variados. Cuando el número de operaciones es pequeño (e.g. 2, 4, 8) es mejor realizarlos en serie ya que se evita el costo computacional de la coordinación entre núcleos. Al elegir 20 como mínimo número de tareas, se requiere que haya al menos i*20 operaciones para realizar en paralelo, donde “i” es el número de núcleos con el que se ejecuta el algoritmo. En los casos que no se cumpla esto, las

operaciones se realizan en serie. Con el conjunto de datos de 1250 variables, en el mejor de los casos se paraleliza el 99.5% de las operaciones (2 núcleos) y en el peor se paraleliza el 95.7% de operaciones (8 núcleos). Este porcentaje de paralelización no necesariamente significa que el tiempo de ejecución del algoritmo sea menor ya que existen casos en los que la coordinación de los hilos es más costosa computacionalmente que el procesamiento que realiza cada núcleo.

En la Figura 6 se incluye el porcentaje de paralelización que se obtiene en el segundo control para cada conjunto de datos sintético. Los resultados más dramáticos se observan en el conjunto de datos de 250 variables, donde en el mejor de los casos se paraleliza el 23% de sus operaciones (2 y 3 núcleos). Para obtener el mayor porcentaje de paralelización en el conjunto de datos de 250 variables, se debería usar 1 como valor mínimo en el segundo control, con lo que se obtendría un 97% de paralelización con la configuración de dos núcleos, y un 63% con la configuración de 8 núcleos. Sin embargo, este valor mínimo no es viable ya que cuando el número de operaciones es muy pequeño (e.g. 2, 4, 8, 16, 32) resulta mejor realizarlos en serie. En los siguientes conjuntos de datos (500-1500 variables) se observa que, a medida que se incrementa al número de variables de los conjuntos de datos, el porcentaje de paralelización para el segundo control aumenta. Esto se debe a que, al incrementar el número de variables, se incrementa el número de operaciones. Es importante tomar en cuenta que lo que se quiere lograr es que el tiempo de coordinación entre núcleos sea menor al tiempo de procesamiento. El tiempo de coordinación incrementa en proporción con el número de núcleos que se utiliza en la ejecución. Por lo que tener un porcentaje menor de operaciones en paralelo no necesariamente significa mayor tiempo de ejecución ni viceversa. Es por estos motivos que se eligieron los parámetros mencionados.

Configuración del algoritmo.

Debido a que las implementaciones del algoritmo (e.g. pcalg, implementación propia) se pueden ejecutar con diferentes configuraciones de sus parámetros, se decidió usar una misma configuración de parámetros para las pruebas de rendimiento. Esto con el objetivo de que en cada ejecución del algoritmo se realice la misma cantidad de operaciones de forma que produzcan los mismos resultados y sus tiempos de ejecución promedio sean comparables. Esta configuración no involucra el modelo de ejecución del algoritmo ni el número de núcleos con el cual se ejecuta. En cambio, toma en cuenta la variación del algoritmo (stable,

conservativo), el valor de significancia *alpha*, el tipo de prueba de independencia condicional y la regla de la mayoría. La Tabla 5 muestra la configuración usada. Se eligió la versión estable de PC ya que es la que permite paralelizar el algoritmo. Por otro lado, la versión conservativa robustece el algoritmo ante errores de muestreo. Los demás parámetros se han elegido de manera empírica o son los que están por defecto al ejecutar el algoritmo.

Tabla 5: Configuración del algoritmo de PC utilizada para las pruebas de rendimiento

Algoritmo de PC	• Conservative • Stable
Nivel significativo (*alpha*)	0.01
Regla de la mayoría (*maj_rule*)	FALSE
Resolver conflictos (*solve_conf*)	FALSE
Prueba de independencia (*indepTest*)	gaussCItest
Version.Unf	[1, 0]

Tabla 6: Características del servidor usado para las pruebas de rendimiento con conjuntos de datos reales.

Modelo Procesador	Intel(R) Xeon(R) CPU E5-4620 v2 a 2.60GHz
Sockets	4
Núcleos por socket	8
Hilos por núcleo	2
Memoria RAM	256GB
Memoria SWAP	4GB
Sistema operativo	CentOS 7.4.1708 64 bits

Sistema sobre el que se ejecutan los algoritmos

Para la ejecución de las distintas configuraciones de los algoritmos se utilizaron dos sistemas: un servidor cuyas características se detallan en la Tabla 6 y una computadora de escritorio con las características detalladas en la Tabla 7. Se utilizaron dos sistemas para las ejecuciones debido a que cuentan con dos enfoques distintos. La computadora de escritorio tiene características mínimas con las que

cuenta un PC o laptop común. Utilizar este sistema permite tener una idea del rendimiento en un computador al que cualquier investigador tiene acceso y donde la dificultad de configuración y ejecución es mínima (al contrario de un servidor, por ejemplo). En esta computadora se ejecutaron los algoritmos con conjuntos de datos sintéticos de hasta 1500 variables. En cambio, el acceso a un servidor es más complejo debido a sus características y requerimientos para su funcionamiento. En el servidor se ejecutaron los algoritmos con conjuntos de datos reales de hasta 5361 variables. Debido a que el servidor cuenta con un procesador con más núcleos (64) en comparación con la computadora de escritorio (8), las ejecuciones del algoritmo en el servidor permitieron comprobar la escalabilidad del algoritmo implementado con hasta 16 núcleos. Se eligió la ejecución con hasta 16 núcleos ya que se comprobó que a partir de este número el tiempo de ejecución de cada implementación del algoritmo no variaba significativamente. Los conjuntos de datos utilizados se explican en sección 4.1.

Tabla 7: Características de la computadora usada para las pruebas de rendimiento con conjuntos de datos sintéticos

Modelo Procesador	Intel Core I7-4770 a 3.4GHz
Núcleos físicos	4
Hilos por núcleo	2
Memoria RAM	4GB
Memoria SWAP	2GB
Sistema operativo	Ubuntu 17.10 64 bits.

Número de ejecuciones del algoritmo

Cada configuración del algoritmo con un determinado modelo de ejecución, conjunto de datos y número de núcleos (en los algoritmos paralelos) se ejecutó varias veces para asegurar que los datos obtenidos sean estadísticamente significativos. El número veces de la ejecución no fue el mismo para todas las configuraciones debido al tiempo requerido para completarlas. En el peor de los casos se necesitó más de 24 horas para realizar una sola ejecución. Para mitigar este problema se comprobó que con el número de ejecuciones haya una diferencia estadísticamente significativa entre los resultados obtenidos que se comparan. Se utilizó la prueba de T-student con un valor significativo de 0.01 de manera que se tenga un 99% de

probabilidad de que existe una diferencia significativa entre las ejecuciones comparadas. El valor significativo (0.01) es la probabilidad de que los resultados no tengan una diferencia significativa.

El conjunto de pruebas de rendimiento se automatizó con un *script*, de manera que cada ejecución inicie con una temperatura similar de procesador. Esto se logró haciendo que el *script* de ejecución espere 3 minutos hasta que el sistema de ventilación de la computadora o servidor enfríe su procesador. La temperatura inicial del procesador es la que se presenta cuando la computadora se mantiene sin realizar tareas que no sean las del sistema operativo. Esto es importante para la obtención de datos consistentes ya que, si el procesador estuviera más caliente, sus protecciones harían que se mantenga en una velocidad de reloj menor para evitar sobrecalentamiento. Como consecuencia de esto, el algoritmo tomaría más tiempo de lo necesario. Esto se comprobó al ejecutar la implementación paralela del algoritmo de PC con memoria compartida en una computadora portátil que por su diseño tiene poca capacidad de disipación de calor. Al no dejar que la computadora se enfrié, el tiempo de ejecución del algoritmo incrementaba con cada ejecución.

4.1. Datos sintéticos y reales

Para realizar las pruebas de rendimiento de la implementación se utilizaron datos sintéticos y reales. Dentro de los datos sintéticos se utilizaron datos propios y de simulaciones. A continuación, se detallan cada grupo de datos usado.

4.1.1 Datos sintéticos

Las pruebas de rendimiento se realizaron con conjuntos de datos sintéticos y reales. Los datos sintéticos se generaron usando la librería *tigramite* (Runge, 2014), que se utiliza para el análisis causal de series de tiempo. *Tigramite* permite reconstruir grafos a partir de conjuntos de datos y modelar las dependencias causales para medición y análisis de predicciones. Se utilizó el método *tigramite.data_proesssing.varprocess()* para generar las series de tiempo. Como semilla para *numpy.random()* y como coeficiente del modelo lineal se utilizó: uno dividido para el número de variables (n) del conjunto de datos a generar (1/n). El método *varprocess*() requiere de un diccionario con el formato *{..., j:[((var, lag), coeff), ...], ...}*. Cada elemento (*j*) del diccionario hace referencia a una variable del conjunto de datos y los enlaces que posee hacia las demás variables (*var*)

atrasadas (*lag)* con un coeficiente lineal (*coeff*). El diccionario se formó de manera que por cada variable del conjunto de datos se incluyera 3 enlaces hacia otras variables atrasadas en -1 (*lag*) y con coeficiente lineal 1/n (*coeff*). El primer enlace es hacia la misma variable atrasada en -1, los dos enlaces restantes son hacia variables aleatorias del conjunto de datos. Se usó -1 como *lag* para cada variable ya que según la documentación de *tigramite* (Runge, Sejdinovic, & Flaxman, 2017) debe ser un número menor o igual a cero. Con este procedimiento se generaron 6 conjuntos de datos con diferente número de variables: 250, 500, 750, 1000, 1250 y 1500, para cada variable se obtuvieron 5000 muestras.

4.1.2 Datos reales

Los conjuntos de datos reales que se utilizan para evaluar la eficiencia de los algoritmos son descritos en la Tabla 8 y sus referencias se obtuvieron de Le et al. (2016). Los primeros 3 conjuntos de datos (NCI-60, MCC y BR51) se utilizan para la investigación de relaciones regulatorias entre microRNAs[26]. Los últimos dos conjuntos de datos (S.aureus y S.cerevisiae), en cambio, se utilizan para descubrir relaciones regulatorias entre factores de transcripción[27]. Se decidió usar estos conjuntos de datos debido a que Le et al. (2016) los utiliza para medir el rendimiento de pcalg con respecto a una versión paralela de pcalg que implementa, ambas en R. De esta manera se podrá realizar comparaciones con los resultados de la versión paralela de pcalg.

Tabla 8: Detalles de los conjuntos de datos reales de expresiones genéticas.

Conjunto de datos	Número de muestras	Número de variables
NCI-60	47	1190
MCC	88	1380
BR51	50	1592
S.aureus	160	2810
S.cerevisiae	63	5361

[26] Son un tipo de genes reguladores.
[27] Proteínas involucradas en el proceso de convertir DNA en RNA.

4.1.3 Simulaciones de procesos de advección y difusión.

Los últimos datos utilizados son simulaciones de procesos de advección y difusión que realiza Ebert-Uphoff & Deng (2015b). Advección es un proceso que actúa como un mecanismo de transporte de una sustancia o propiedad por un fluido. Un ejemplo de advección es el transporte de calor por un fluido en el que el movimiento del fluido se describe como un vector, mientras que la temperatura es un valor escalar que cambia con el tiempo. En cambio, difusión es un proceso que provoca el esparcimiento de una señal mientras el punto más alto de la señal se mantiene en el mismo lugar (e.g. agregar agua caliente sobre un recipiente de agua fría). Estos procesos son comunes y en ciertos casos dominantes en varios procesos dinámicos de la naturaleza. Ebert-Uphoff & Deng (2015b) realiza varias simulaciones de estos procesos de los que obtiene varios conjuntos de datos, los que se han elegido para reproducir los resultados de aplicar el algoritmo implementado son: (1) advección pura con un punto de ruido, (2) advección pura con varios puntos de ruido, (3) difusión pura con un punto de ruido y (4) difusión pura con varios puntos de ruido. Cada conjunto de datos cuenta con 100 variables y 6900 muestras, cada variable se ubica en una matriz bidimensional por lo que cuenta con una coordenada (*x, y*) en un matriz de 100x100. La distancia horizontal y vertical entre cada variable es de 10.

Antes de ejecutar el algoritmo con los conjuntos de datos, se tiene que incorporar "tiempo" a la simulación realizada. Esto involucra crear y agregar variables adelantas y retrasadas a cada conjunto de datos a partir de las variables originales. El procedimiento que se sigue para adelantar o retrasar las variables lo expone Kalisch et al. (2017). En nuestro caso para recrear las mismas condiciones de ejecución se agregan variables adelantadas. Por ejemplo, si la variable original X_0 tiene N muestras en un rango [*0, N-1*], la variable X_i adelantada en i tiempos tiene $N-i$ muestras y sus muestras corresponden al rango [*i, N - 1*] de las muestras de la variable X_0. Al final, el número de muestras que se utiliza para todas las variables del conjunto de datos, lo dicta la última variable adelantada, por lo que se eliminan las muestras restantes en las variables anteriores. Para cada variable de los conjuntos de datos se crearon 19 variables adelantadas, de forma que los conjuntos de datos resultantes tienen 2000 variables y 6981 muestras.

Incluyendo las variables originales, cada conjunto de datos resultante cuenta con variables en 20 tiempos. Ya que se crean variables adelantadas, los enlaces dirigidos del DAG resultante deben ser desde el pasado hacia el futuro. Dado las variables X_s y X_t (adelantadas en tiempos s y t) se prohíben los enlaces dirigidos

$X_t \rightarrow X_s$, siempre que *s* sea menor o igual a *t*. Esta restricción se agrega como conocimiento a priori para la ejecución del algoritmo. Dado que el conjunto de datos cuenta con 2000 variables divididas en 20 tiempos (100 por cada tiempo), una vez se han obtenidos los resultados se eliminan los dos primeros tiempos para asegurar una apropiada inicialización de modelo (Ebert-Uphoff & Deng, 2015b). Es decir, las variables en tiempo T=3Δt se convierten en T=0 y la matriz resultante es de 1800x1800 nodos

4.2. Configuración de las pruebas

En las pruebas de rendimiento se compara pcalg, la versión paralela de pcalg (Le et al., 2016) y la implementación de algoritmo realizada en este trabajo. Para cada implementación se utilizó la configuración de sus parámetros descrita en la Tabla 5. Se organizaron varios conjuntos de pruebas donde se compara cada modelo de ejecución de las implementaciones. Para las pruebas se utilizaron conjuntos de datos sintéticos y reales. Además, se reproducen los resultados de aplicar PC-stable a simulaciones de procesos de advección y difusión con varios puntos de ruido. La descripción de cada conjunto de pruebas se describe a detalle a continuación.

4.2.1 Ejecución en serie con datos sintéticos

En este conjunto de pruebas se compara el rendimiento de las implementaciones en serie del algoritmo con conjuntos de datos sintéticos. Los algoritmos que se comparan son: pcalg y la implementación en serie del algoritmo. Las pruebas se realizaron con los 6 conjuntos de datos sintéticos generados (250-1500 variables). Dado un conjunto de datos, el algoritmo se ejecuta varias veces para obtener un tiempo de ejecución promedio, la ejecución se realiza sobre la computadora de escritorio descrita en la Tabla 5. El número de veces que se ejecuta cada algoritmo (con un determinado conjunto de datos) es el necesario para comprobar que existe una diferencia estadísticamente significativa entre los tiempos de ejecución promedio de las implementaciones. Para esta y todas las demás pruebas, para comprobar que haya una diferencia significativa, se utiliza la prueba de T-student con un valor significativo de 0.01 de forma que haya un 99% de probabilidad de que exista la diferencia significativa. Los resultados de esta prueba se presentan en la sección 5.1.

4.2.2 Ejecución en paralelo (memoria compartida) con datos sintéticos

En este conjunto de pruebas se compara la implementación paralela del algoritmo con memoria compartida (openMP) con los resultados de nuestra implementación obtenidos en la ejecución en serie. Al igual que en las ejecuciones en serie, se utilizaron los 6 conjuntos de datos sintéticos generados (250 -1500 variables). Dado un conjunto de datos se realiza una serie de ejecuciones con diferente número de núcleos (2-8). Las ejecuciones se realizaron sobre la computadora de escritorio descrita en la Tabla 5. En las mismas condiciones del algoritmo (conjunto de datos y número de núcleos), el número de ejecuciones que se realiza es el suficiente para comprobar que exista una diferencia significativa al comparar con los resultados en serie de la implementación con el mismo conjunto de datos. Los resultados de esta prueba se presentan en la sección 5.2.

4.2.3 Ejecución en paralelo (memoria distribuida) con datos sintéticos

Para este conjunto de pruebas se utiliza la implementación paralela del algoritmo con memoria distribuida (open MPI). Sus resultados se comparan con los obtenidos en la implementación en serie y paralela con memoria compartida (openMP). Para las ejecuciones del algoritmo se utilizaron los 6 conjuntos de datos sintéticos generados (250 -1500 variables). Por cada conjunto de datos se realizaron varias ejecuciones con diferente número de núcleos (2-8) sobre la computadora de escritorio descrita en la Tabla 5. Dada las ejecuciones del algoritmo con un conjunto de datos y un determinado número de núcleos, se obtiene su tiempo de ejecución promedio. En estas condiciones del algoritmo (conjunto de datos y número de núcleos), el número de veces que se ejecuta es el suficiente para comprobar que exista una diferencia significativa con los resultados de las ejecuciones en serie con el mismo conjunto de datos. Los resultados de esta prueba se presentan en la sección 5.3.

4.2.4 Ejecución en paralelo con datos reales

En este conjunto de pruebas se utiliza la implementación paralela del algoritmo con memoria compartida (openMP), memoria distribuida (open MPI) y la versión paralela de pcalg. Paras las ejecuciones se utilizó los 5 conjuntos de datos reales descritos en la Tabla 8. Por cada conjunto de datos se utilizó diferente número de núcleos (4-8). Se eligió iniciar la ejecución de los algoritmos con 4 núcleos debido a que en las pruebas realizadas con datos sintéticos se comprobó que hasta la configuración con 4 núcleos el comportamiento de los algoritmos era similar. Las ejecuciones de los algoritmos se realizan sobre el servidor descrito en la Tabla 6. Se decidió utilizar el servidor ya que los conjuntos de datos reales tienen hasta 5361 variables y se necesita más memoria RAM de la que la computadora de escritorio posee (Tabla 5). Dado uno de los algoritmos, un conjunto de datos y un determinado número de núcleos, se realiza varias ejecuciones para obtener el tiempo de ejecución promedio del algoritmo. En estas condiciones, el número de veces que se ejecuta el algoritmo son las suficientes para comprobar que existe una diferencia significativa entre las ejecuciones de los distintos algoritmos comparados. En este grupo de pruebas no se utiliza la versión en serie del algoritmo ya que se quería comprobar el comportamiento de las implementaciones paralelas. Los resultados de esta prueba se presentan en la sección 5.4.

4.2.5 Escalabilidad de la implementación paralela

En este conjunto de pruebas se utiliza la implementación paralela del algoritmo con memoria compartida (openMP), con memoria distribuida (open MPI) y la implementación paralela de pcalg. Estas pruebas se realizan para comprobar la escalabilidad de los algoritmos al ejecutarlos con diferente número de núcleos y con los conjuntos de datos reales descritos en Tabla 8. Dado un conjunto de datos, cada algoritmo se ejecuta con 10, 12, 14 y 16 núcleos. Estos resultados se juntan con parte de los resultados obtenidos en el conjunto de pruebas de la sección 4.2.4, para generar gráficas con los tiempos de ejecución promedio obtenidos con: 4, 8, 10, 12, 14 y 16 núcleos. Esto es posible ya que en los dos conjuntos de pruebas se utilizan las mismas implementaciones, los mismos conjuntos de datos y se ejecutan sobre el mismo servidor. Dado una de las implementaciones del algoritmo con un conjunto de datos y un determinado número de núcleos, se realizaron varias ejecuciones para obtener el tiempo promedio. En estas condiciones del algoritmo, el

número de ejecuciones que se realizó es el suficiente para comprobar que exista una diferencia significativa entre los resultados de los diferentes algoritmos. Los resultados de esta prueba se presentan en la sección 5.5.

4.2.6 Experimentos con advección y difusión

En este conjunto de pruebas se reproducen los resultados que obtiene Ebert-Uphoff & Deng (2015b) al aplicar el algoritmo PC-stable a simulaciones de procesos de advección y difusión en una grilla bidimensional. Para recrear los resultados se utilizó la implementación paralela con memoria compartida (openMP), los parámetros usados en la implementación se presentan en la Tabla 5, aunque se cambió el valor de significancia *alpha* a 0.05 debido a que es el valor que se usa en las ejecuciones de Ebert-Uphoff & Deng (2015b). Los parámetros no son los exactos que usa Ebert-Uphoff & Deng, (2015b) debido a que no presenta todos los detalles usados para la ejecución. Lo que indica que utilizan es PC-stable y el valor *alpha* de 0.05. En nuestro caso se utilizó la implementación que combina PC-stable y CPC debido a que los cambios que introduce CPC permiten que el algoritmo sea más robusto con respecto a errores de muestreo. Las ejecuciones del algoritmo se realizaron en el servidor detallado en la Tabla 6. No se obtuvo un tiempo de ejecución promedio ya que lo que nos interesa son los resultados del algoritmo. En total se realizaron 4 ejecuciones, una por cada conjunto de datos, cada ejecución se realizó con 50 núcleos.

El DAG resultante cuenta con 3 tipos de enlaces: internos, externos y concurrentes. Los enlaces internos son los que conectan un nodo en un tiempo T= iΔt, al mismo nodo en un tiempo futuro T = (i+j)Δt. Estos enlaces representan la memoria local del modelo y se grafican como un punto. Los enlaces externos conectan a un nodo en un tiempo T= iΔt, a cualquiera de los nodos en un tiempo futuro T= (i+j)Δt, además, codifican el flujo de información entre distintas localizaciones y se representan como un enlace dirigido. Finalmente, los enlaces concurrentes son los que conectan nodos en un mismo tiempo por lo que estos enlaces se grafican sin orientación. Los resultados se presentan con gráficas de los enlaces (internos o externos) presentes en un grupo de variables en un tiempo T. Para acceder a los enlaces dirigidos de las variables en un tiempo T, se extrae una matriz cuadrada (100 x 100) de la matriz de adyacencia resultante. Por ejemplo, para obtener los enlaces en T = 0 se toma la cuadrícula formada por el rango de filas y columnas [0:100]. En cambio, si se quiere obtener los enlaces en T = 2Δt se toma la cuadricula formada por el rango de filas

[0:100] y el rango de columnas [200:300]. Debido a que cada variable cuenta con una coordenada dentro de una grilla bidimensional de 100 x 100, graficar un enlace consiste en crear una línea entre dos coordenadas. Los resultados de estas pruebas se presentan en la sección 5.6.

Capítulo 5: Resultados y Discusión

5.1 Ejecución en serie con datos sintéticos

En esta sección se compara el rendimiento del algoritmo implementado en serie con pcalg. En la Figura 7 se muestra los resultados de las ejecuciones, cada punto de la gráfica representa el tiempo de ejecución promedio obtenido de realizar las ejecuciones con cada conjunto de datos. Con la implementación en C se realizaron 10 ejecuciones para cada conjunto de datos, en cambio, con pcalg se realizaron 10 ejecuciones solo hasta el conjunto de datos de 1250 variables. Con el conjunto de datos de 1500 variables, el tiempo de ejecución se volvió muy grande (más de 24 horas), por lo que se realizaron solo 3 ejecuciones que fueron suficientes para encontrar una diferencia significativa.

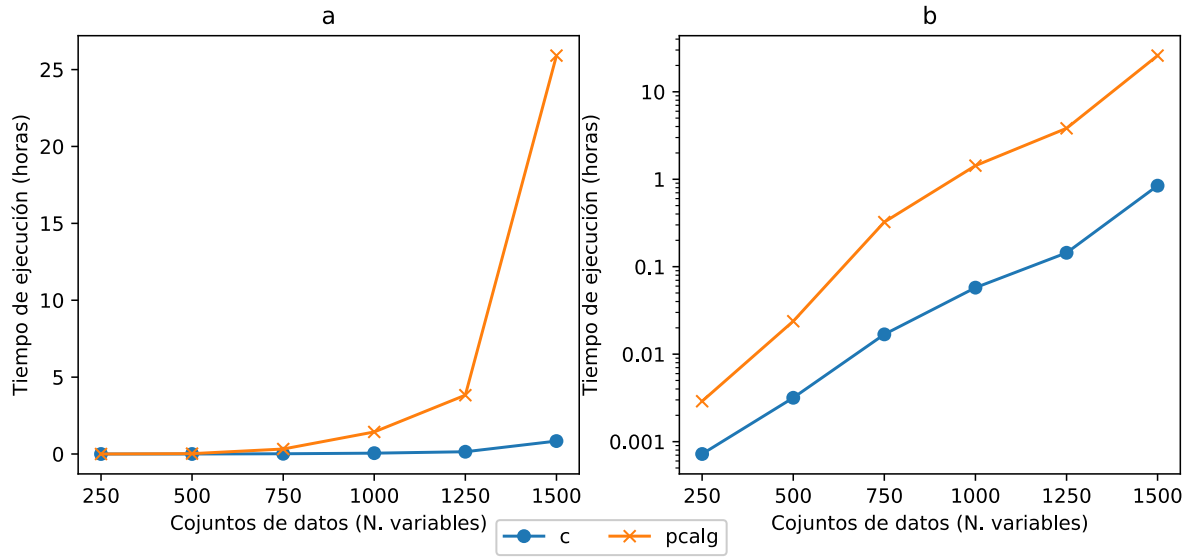


Figura 7: Comparación de los tiempos de ejecución promedio del algoritmo en serie implementado en C y pcalg, con 6 conjuntos de datos sintéticos. a) Ejes en escala lineal y b) Eje del tiempo en escala logarítmica.

Lo primero que se observa en los resultados de la Figura 7, es que a partir del conjunto de datos con 750 variables los tiempos de ejecución promedios se vuelven más distantes entre sí, a la implementación en C le toma 1 minuto completar su ejecución y a pcalg 19.4 minutos. Esto representa que C, con el conjunto de datos

de 750 variables es 19 veces más rápido que pcalg. Con cada salto de 250 variables se observa que la diferencia entre los tiempos de ejecución se hace más grande. Esta diferencia es más notable con el conjunto de datos de 1500 variables donde pcalg tiene un tiempo promedio de ejecución de 25.9 horas y la implementación en C de 50.6 minutos. Si se ejecutara pcalg con conjuntos de datos de más de 1500 variables, de acuerdo a la tendencia de crecimiento de los tiempos de ejecución, le tomaría más de una semana completar su ejecución. Además, se necesitarían realizan varias ejecuciones para comprobar que haya una diferencia significativa entre los tiempos de ejecución que se comparan. Como ya se indicó en la complejidad computacional del algoritmo de PC (sección 1.2.1), el crecimiento de los tiempos de ejecución se debe a que el algoritmo de PC en el peor de los casos tiene un tiempo de ejecución exponencial en relación al número de variables del conjunto de datos (Le et al., 2016). Por lo que, al incrementar el número de variables, el número de pruebas de independencia condicional necesarias y el tiempo de ejecución incrementa exponencialmente. Esto es consistente con la forma en la que crece el tiempo de ejecución promedio de los dos algoritmos.

Si se tomara en cuenta solo los resultados de la Figura 7, se podría concluir erróneamente que en los conjuntos de datos que tienen menos de 750 variables, el tiempo de ejecución promedio no es tan diferente entre los dos algoritmos. Esto se debe a que la gráfica tiene ejes en escala lineal y cuando existe una mezcla de valores pequeños (menos de 1 minuto) y grandes (más de 24 horas), estos últimos opacan a los valores pequeños. Lo que resulta en líneas que se solapan al representar los valores pequeños. Al cambiar la gráfica a una escala logarítmica, Figura 7, se observa que la diferencia entre los tiempos de ejecución promedio está presente desde el primer conjunto de datos (250 variables). Con esta escala también se observa que el tiempo de ejecución promedio de los dos algoritmos crece de forma exponencial con cada conjunto de datos. Esto se debe a que los dos algoritmos utilizan la misma configuración por lo que realizan la misma cantidad de operaciones (i.e. pruebas de independencia condicional) en cada punto que se compara. Debido a la tendencia de crecimiento de los dos algoritmos, la diferencia entre los tiempos de ejecución promedio solo se hace más evidente al incrementar el número de variables.

La Figura 8 tiene como eje horizontal el número de variables usado en cada conjunto de datos y como eje vertical la razón de los tiempos de ejecución promedio de los dos algoritmos (pcalg dividido para la implementación en C). Lo que se

observa es que la razón crece casi linealmente en relación al número de variables del conjunto de datos. Es decir, al aumentar las variables pcalg demora más en comparación a la implementación en C (lo que hace a la implementación en C una mejor opción). Con 250 variables el tiempo de ejecución de pcalg es 4 veces el tiempo de ejecución de la implementación en C, con 500 variables es 7 veces y con 750 es 19 veces. Estos incrementos en la escala lineal de la Figura 7 no se ven tan pronunciados debido a que el tiempo de ejecución promedio de pcalg no supera los 60 minutos. Otra observación es que con el número de variables crece la razón entre los tiempos de ejecución, como consecuencia, los saltos entre los tiempos de ejecución promedio de la Figura 7, se hacen más pronunciados con cada conjunto de datos. Al usar el conjunto de datos de 1250 variables, la implementación en C demora 8 minutos en completar su ejecución, en comparación con pcalg que se demora 26 veces más (3.8 horas). El salto más grande en la gráfica se obtiene al pasar de 1250 a 1500 variables, esto se debe a que pcalg demora más de 30 veces más que la implementación en C, esto se traduce en 25.9 horas para pcalg y 50.6 minutos para la implementación en C. En pruebas que no se han incluido en el trabajo, a la implementación en C le toma alrededor de 6 horas completar su ejecución con un conjunto de datos de 2000 variables. Si se usara la razón obtenida con 1500 variables, se estima que a pcalg le tomaría más de 7 días completar su ejecución.

5.2 Ejecución en paralelo (memoria compartida) con datos sintéticos

En estas pruebas de rendimiento se comparan los tiempos de ejecución promedio de la implementación paralela con memoria compartida (openMP) con los resultados obtenidos en la ejecución en serie del mismo algoritmo (sección 5.1). Las pruebas de rendimiento se realizaron con los 6 conjuntos de datos sintéticos (250-1500 variables) y con diferente número de núcleos (2-8). Por cada conjunto de datos se realizaron 5 ejecuciones y se comprobó que haya un 99% de confianza de que exista una diferencia significativa entre los resultados que se comparan (serie con paralelo). En estas pruebas se utiliza un número fijo de ejecuciones debido a que el tiempo necesario para completarlas es menor en comparación con pcalg.

La Figura 9 muestra los tiempos de ejecución promedio de la implementación en paralelo y en serie. La primera línea (con leyenda serie) se conforma por los tiempos

de ejecución promedio de la implementación en serie, estos son los mismos resultados que se obtuvieron en la sección 5.1. Las siguientes líneas representan los resultados de la implementación en paralelo (openMP) con diferente número de núcleos (2-8). Por cada conjunto de datos se tiene en total 8 tiempos de ejecución promedio a comparar. En la Figura 9, desde el conjunto de datos de 750 variables, se observa que tiene un comportamiento similar a la Figura 7. A partir del conjunto de datos de 750 variables la diferencia entre el tiempo de ejecución promedio en serie y paralelo se vuelve más evidente. Como ya se indicó, esto es un problema de la escala de la gráfica y no de los resultados obtenidos. La Figura 9 muestra el eje del tiempo en escala logarítmica. Se observa de manera clara la diferencia entre los tiempos de ejecución obtenidos para cada conjunto de datos.

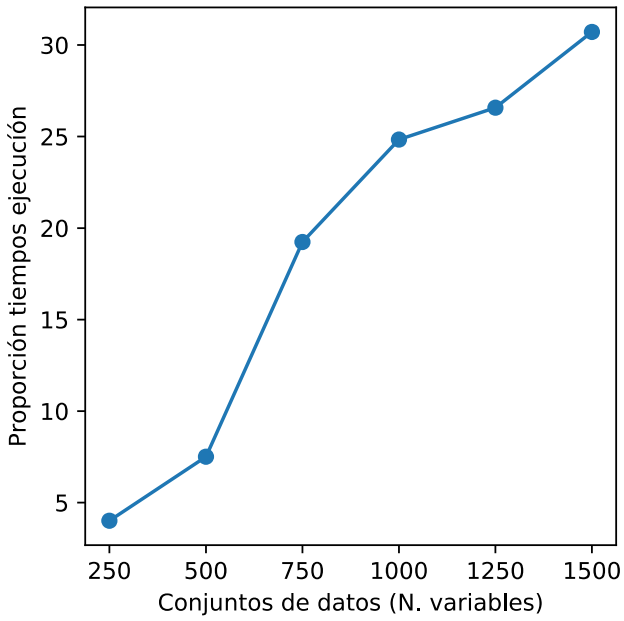
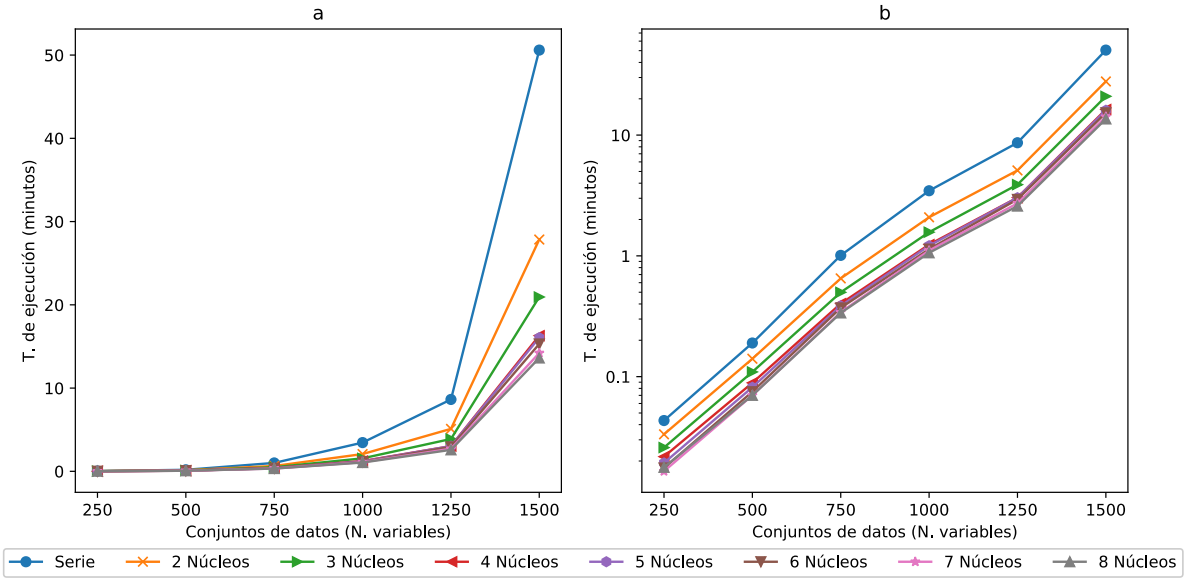


Figura 8: Razón (proporción) de los tiempos de ejecución promedio (pcalg para la implementación en serie en C), se comparan los algoritmos con los mismos conjuntos de datos. Cada punto representa cuántas veces más se demoró la implementación de pcalg con respecto a la implementación en C.

La Figura 9 muestra que la línea que conforma los resultados del algoritmo en serie, está siempre sobre las líneas de los resultados en paralelo. Esto indica que los resultados obtenidos en cualquiera de las ejecuciones en paralelo, tienen un menor tiempo de ejecución promedio en comparación con la implementación en serie. El resultado es lógico si se toma en cuenta que el número de operaciones que realiza el algoritmo de PC en cada ejecución es el mismo y lo que diferencia a sus implementaciones es el número de operaciones que se realizan al mismo tiempo.

En el caso del algoritmo en paralelo existen otras operaciones que se realizan en conjunto con las operaciones del algoritmo. Estas operaciones son las que permiten la coordinación entre los diferentes núcleos, esto no se nota con pocos núcleos (2-4), aunque, se evidencia al observar cómo se distribuyen las líneas de los tiempos de ejecución en la Figura 9 (>4 núcleos). Existe una diferencia marcada entre los tiempos de ejecución en serie, con 2, 3 y 4 núcleos. Esta diferencia se mantiene para todos los conjuntos de datos sin importar el número de variables. Desde el cuarto hasta el octavo núcleo esta diferencia es menor ya que las líneas que conforman los tiempos de ejecución promedio están más juntas y en ciertos casos se solapan. Al incrementar el número de núcleos se realizan más tareas al mismo tiempo, sin embargo, el número de tareas no cambia, en cambio es necesario la



coordinación entre núcleos para que cada núcleo sepa qué tareas realizar.

Figura 9: Comparación de los tiempos de ejecución promedio del algoritmo de PC, tiempos de la implementación paralela con memoria compartida (openMP) con 2-8 núcleos y de la implementación en serie en C. Tiempos de ejecución obtenidos con 6 conjuntos de datos sintéticos (250-1500 variables). a) Ejes en escala lineal y b) Eje del tiempo en escala logarítmica.

La mayor diferencia entre los tiempos de ejecución se presenta entre la implementación serie y paralela con dos núcleos. Esto se cumple para todos los conjuntos de datos que se utilizaron. Con el conjunto de datos de 250 variables se pasa de 2.6 segundos (serie) a 2 segundos (2 núcleos) y con 1500 variables se pasa de 50.6 minutos (serie) a 27.8 minutos (2 núcleos). Al incrementar el número de núcleos se logra reducir el tiempo de ejecución, pero las reducciones se vuelven

cada vez menores debido a la coordinación necesaria entre los núcleos. En las pruebas realizadas se observó que incrementar el número de núcleos puede ser perjudicial para el tiempo de ejecución. Esto se evidencia al pasar de 7 a 8 núcleos en el conjunto de datos de 250 variables, se pasa de 0.98 segundos a 1.06 segundos, es probable que con conjuntos de datos menores esta situación se repita. Sin embargo, con los conjuntos de datos más grandes (más de 500 variables) esto no se presenta. Lo que se observa es que la reducción del tiempo de ejecución es cada vez menor. Con el conjunto de datos de 1500 variables, al pasar de 6 a 7 núcleos se logra una reducción de 65 segundos, al pasar de 7 a 8 núcleos el tiempo de ejecución se reduce en 36 segundos. Siguiendo con la misma tendencia, si se incrementara el número de núcleos se llegará a un punto en el que ya no descienda más el tiempo de ejecución y en cambio incremente. Este es el comportamiento que se observó en el conjunto de datos de 250 variables debido a que el número de operaciones a paralelizar es menor a los otros conjuntos de datos probados.

5.3 Ejecución en paralelo (memoria distribuida) con datos sintéticos

Para este conjunto de pruebas de rendimiento se utilizó la implementación paralela del algoritmo con memoria distribuida (Open MPI) con los 6 conjuntos de datos sintéticos generados (250-1500 variables). Por cada conjunto de datos se realizaron ejecuciones con diferente número de núcleos (2-8). Los resultados de la ejecución se comparan con los obtenidos en la sección 5.1 y con los obtenidos en la sección 5.2. Para obtener el tiempo de ejecución promedio se realizaron 5 ejecuciones con un determinado conjunto de datos y número de núcleos. En total para cada conjunto de datos se realizaron 35 ejecuciones. Este número de ejecuciones fue suficiente para comprobar que existe una diferencia significativa con los resultados de la implementación en serie.

En la Figura 10 se muestra los resultados obtenidos en la ejecuciones. Al igual que en los resultados de las secciones anteriores (sección 5.1 y sección 5.2), a partir del conjunto de datos de 750 variables, la Figura 10 tiene un comportamiento similar. Este comportamiento indica que el tiempo de ejecución promedio de la implementación en serie es mayor al tiempo de ejecución del algoritmo en paralelo con memoria distribuida (2-8 núcleos). En la Figura 10, la línea que conforma los resultados en serie está siempre sobre las líneas de los resultados en paralelo. Los

resultados que más destacan son los de la implementación paralela con 2 núcleos. Esta línea presenta una clara separación entre los resultados en serie con los demás resultados en paralelo (3-8 núcleos), la separación se mantiene para todos los conjuntos de datos. Este comportamiento indica que, al incrementar el número de núcleos se reduce el tiempo de ejecución, aunque estos resultados no tengan una tendencia clara como la observada en la sección 5.2. En la Figura 10 se observa que solo con el conjunto de datos de 1500 variables hay una clara distinción entre los tiempos de ejecución obtenidos. Con el conjunto de datos de 1250 variables la diferencia solo es apreciable hasta el tercer hilo, con 1000 variables los resultados del algoritmo en paralelo forman un solo punto indistinguible. Al cambiar a la escala logarítmica (Figura 9b) la distribución de los resultados no mejora. Se distinguen solo los resultados en serie y en paralelo con dos núcleos.

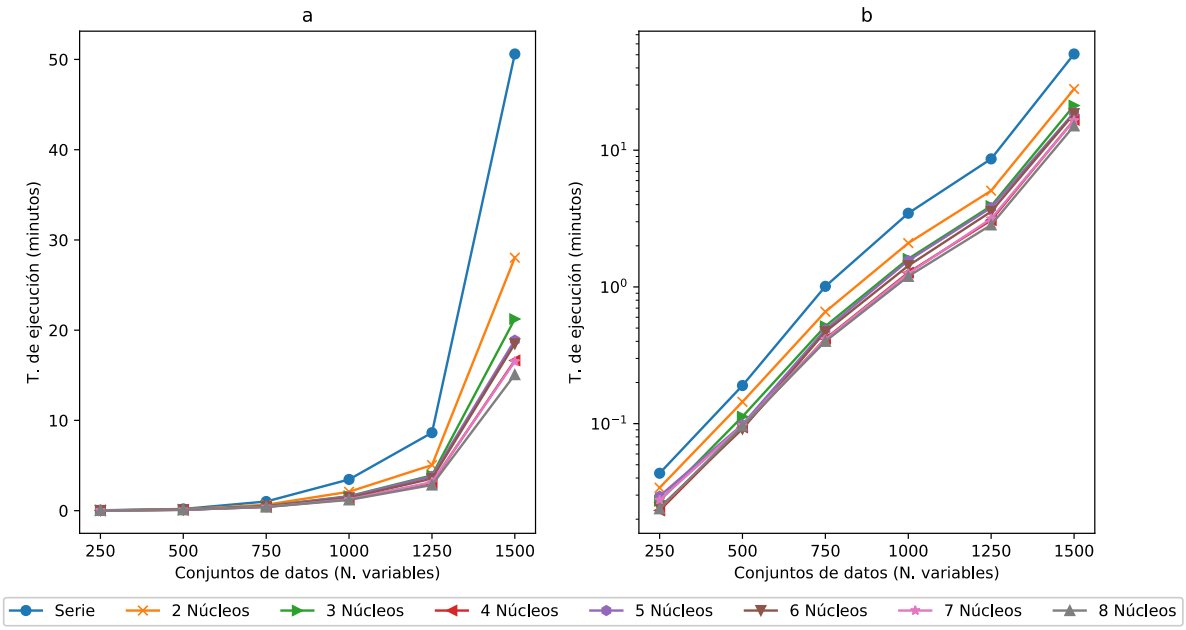


Figura 10: Comparación de los tiempos de ejecución promedio de la implementación paralela con memoria distribuida (Open MPI) con 2-8 núcleos y de la implementación en serie en C. Tiempos obtenidos con 6 conjuntos de datos sintéticos. a) Ejes en escala lineal y b) Eje del tiempo en escala logarítmica.

La Figura 11 muestra sólo los resultados de la implementación paralela con memoria distribuida (Open MPI), como eje vertical se tiene el tiempo de ejecución y como eje horizontal el número de núcleos usado. Cada línea está formada con el

tiempo de ejecución obtenido para un conjunto de datos con diferentes núcleos. Lo primero que se observa es la separación que existe entre cada línea de los resultados. En la parte superior se encuentran los resultados del conjunto de datos de 1500 variables en el que los tiempos de ejecución obtenidos son los más grandes (15-28 minutos), seguidos del conjunto de datos de 1250 variables (2.8 - 5 minutos) y de esta manera hasta el conjunto de datos con 250 variables donde se obtienen los tiempos de ejecución más pequeños (0.9 - 3 segundos). A medida que incrementa el número de variables la separación entre cada línea de resultados se vuelve más grande, además, tiempo de ejecución promedio no siempre se reduce a medida que se incrementa el número de núcleos. Esto es más notable al pasar de 4 a 5 núcleos, con el conjunto de datos de 1500 variables se pasa de 16.6 minutos a 18.8 minutos. Este comportamiento está presente para todos los conjuntos de datos analizados.

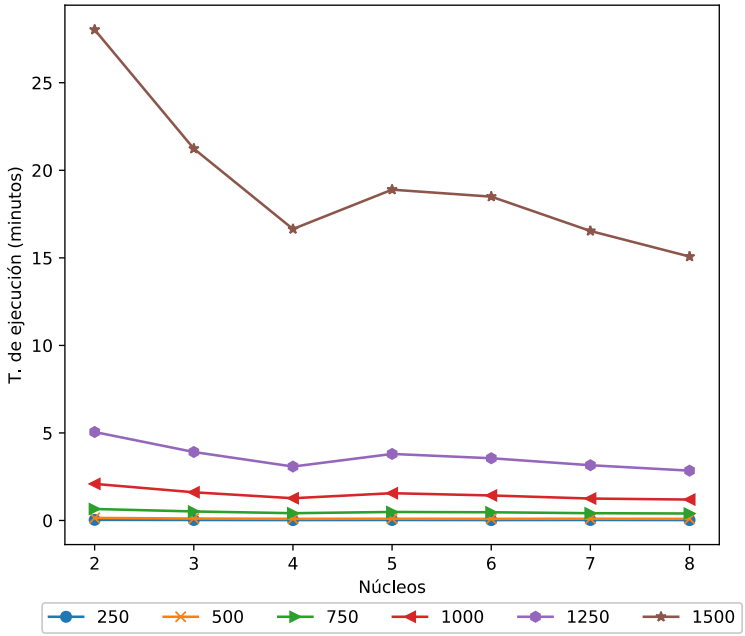
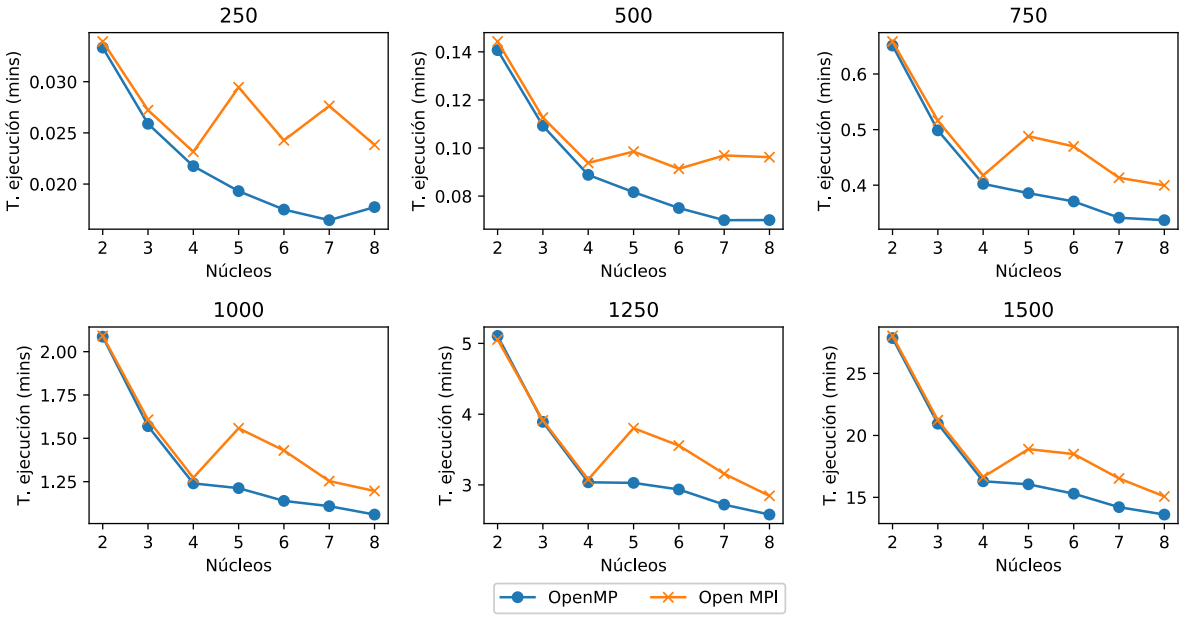


Figura 11: Tiempos de ejecución promedio obtenidos con la implementación paralela con memoria distribuida (Open MPI), con 6 conjuntos de datos sintéticos (250-1500) y de 2-8 núcleos.

En cambio, en la Figura 12 se presentan 6 gráficas donde se comparan los resultados obtenidos con la implementación paralela con memoria compartida (openMP) y con memoria distribuida (Open MPI). La comparación se realiza para cada conjunto de datos sintético utilizado con diferente número núcleos (2-8). En los resultados con 2, 3 y 4 núcleos las implementaciones tienen una tendencia similar:

al incrementar el número de núcleos se reduce el tiempo de ejecución promedio. Con 5 núcleos en la implementación con memoria distribuida, el tiempo de ejecución promedio incrementa con respecto a la configuración de 4 núcleos. Este comportamiento se debe al control que se utiliza para decidir qué tareas se realizan en paralelo cuando se utiliza memoria distribuida. Sin tomar en cuenta la coordinación entre núcleos, el algoritmo con cualquier número de núcleos tiene el mismo número de operaciones a realizar. Hasta el cuarto núcleo el costo computacional de repartir el trabajo entre núcleos se compensa con la paralelización que se realiza de las operaciones. Esto se evidencia al ver el comportamiento que



tiene la gráfica hasta el 4to núcleo.

Figura 12: Comparación de los tiempos de ejecución promedio obtenidos con las implementaciones paralelas (OpenMP y Open MPI) para los conjuntos de datos sintéticos (250-1500 variables) con 2-8 núcleos.

En la Figura 6 se presenta el porcentaje de operaciones que se paralelizan para cada conjunto de datos sintético usado en las pruebas de rendimiento. Con 4, 5 y 6 núcleos el porcentaje de paralelización para “procesamiento de tripletas” es el mismo, lo que se repite para 7 y 8 núcleos. Al incrementar el número de núcleos, el número de operaciones asignadas a cada núcleo se reduce y en cambio incrementa la coordinación entre núcleos. Tomando en cuenta solo los resultados de la implementación paralela con memoria distribuida (Open MPI) se tienen las siguientes observaciones. Al pasar de 4 a 5 núcleos se mantiene el porcentaje de paralelización, sin embargo, el costo computacional de la coordinación entre núcleos

no se compensa con el procesamiento de operaciones que realiza el nuevo núcleo. Esto de acuerdo al incremento del tiempo de ejecución promedio que se observa en la Figura 12. Al pasar de 5 a 6 núcleos, el porcentaje de paralelización se mantiene y en cambio el tiempo de ejecución promedio se reduce. Lo que nos lleva a concluir que el costo computacional de pasar de 5 a 6 núcleos es menor al costo de pasar de 4 a 5 núcleos. Al pasar de 6 a 7 núcleos, el tiempo de ejecución promedio vuelve a incrementar en los dos primeros conjuntos de datos (250 y 500 variables), sin embargo, en los demás conjuntos de datos el tiempo se reduce. De acuerdo a la Figura 6 pasar de 6 a 7 núcleos reduce el porcentaje de paralelización en todos los conjuntos de datos, siendo los más afectados los de 250 y 500 variables. Con 250 variables el "procesamiento de tripletas" pasa del 10% a 0% y con 500 variables pasa del 44% al 23%. Cuando se pasa a los 8 núcleos: el porcentaje de paralelización y el tiempo promedio de ejecución se mantiene para el conjunto de datos de 500 variables, en cambio el tiempo promedio se reduce para los demás conjuntos.

5.4 Ejecución en paralelo con datos reales

Hasta este punto las pruebas de rendimiento se han realizado solo con datos sintéticos por lo que se decidió utilizar los conjuntos de datos reales descritos en la Tabla 8. Esto con el objetivo de comprobar el comportamiento y rendimiento del algoritmo implementado ante conjuntos de datos que representan un sistema complejo real. Para las pruebas se utiliza la implementación paralela con memoria compartida (openMP), con memoria distribuida (Open MPI) y la implementación paralela de pcalg realizada por Le et al. (2016). El algoritmo que Le et al. (2016) realiza está escrito en R y se denomina "PC-Parallel R" en este conjunto de pruebas. Para las ejecuciones se utilizaron desde 4 a 8 núcleos, esto debido a que en las pruebas de rendimiento anteriores se comprobó que hasta el 4to núcleo todos los algoritmos se comportan de forma similar. Dada una implementación con un conjunto de datos y un determinado número de núcleos, se realizaron 5 ejecuciones para obtener el tiempo promedio de ejecución. Esto debido a que con 5 ejecuciones se comprobó que existía una diferencia significativa entre los resultados de las implementaciones. Los algoritmos se ejecutaron sobre el servidor (Tabla 6) debido a que los conjuntos de datos tienen hasta 5361 variables y la computadora de escritorio (Tabla 7) no tiene la suficiente memoria RAM para ejecutarlos. Además, las pruebas de esta sección se realizaron en conjunto con las de la 5.5 Escalabilidad

Los resultados de las pruebas de rendimiento se muestran en la Figura 13. Cada línea se compone de los tiempos de ejecución promedio de cada implementación con un conjunto de datos y distinto número de núcleos. Lo primero que se observa es que en los primeros 4 conjuntos de datos (BR51, MCC, NCI60 y S.aureus) los resultados de las pruebas de ejecución siguen un mismo patrón. El algoritmo "Parallel-PC R" en estos 4 conjuntos de datos tiene un tiempo de ejecución mayor que "openMP" y "Open MPI". Además, las líneas resultantes de estos 4 conjuntos de datos tienen un comportamiento similar, al pasar de 4 a 5 núcleos, en el mejor de los casos el tiempo de ejecución de "Parallel-PC R" desciende en 10 segundos (MCC) o en el peor caso incrementa 14 segundos (S.aureus). A partir del 5^{to} núcleo, en todos los casos, el tiempo de ejecución de "Parallel-PC R" desciende, aunque nunca llega a los niveles de las otras dos implementaciones. En cambio, en estos 4 conjuntos, las implementaciones paralelas "openMP" y "Open MPI" siguen un comportamiento casi idéntico. Los dos algoritmos parten de similares posiciones con 4 núcleos, aunque "Open MPI" parte con un tiempo de ejecución mayor que "OpenMP" y a medida que incrementan los núcleos se acercan a tiempo de ejecución similares. La gran diferencia que se presenta es con el conjunto de datos "S.cerevisiae". Desde la configuración con 4 núcleos los tiempos de ejecución de "Open MPI" y "Parallel-PC R" siguen patrones diferentes. En este caso "Open MPI" es el que tiene un tiempo de ejecución mayor en todas las ocasiones y el que no tiene una tendencia definida. El comportamiento que presenta "Open MPI" es similar al observado en las pruebas de rendimiento con los datos sintéticos, Figura 11. Cabe recalcar que el conjunto de datos "S.cerevisiae" cuenta con 5361 variables, el más grande de los conjuntos de datos reales. Una de las cosas importantes a tomar en cuenta es que en los resultados de los primeros 4 conjuntos de datos, "Open MPI" tiene un comportamiento muy diferente al que presenta en los resultados de la Figura 11. Es solo en el último conjunto de datos que presenta un comportamiento similar.

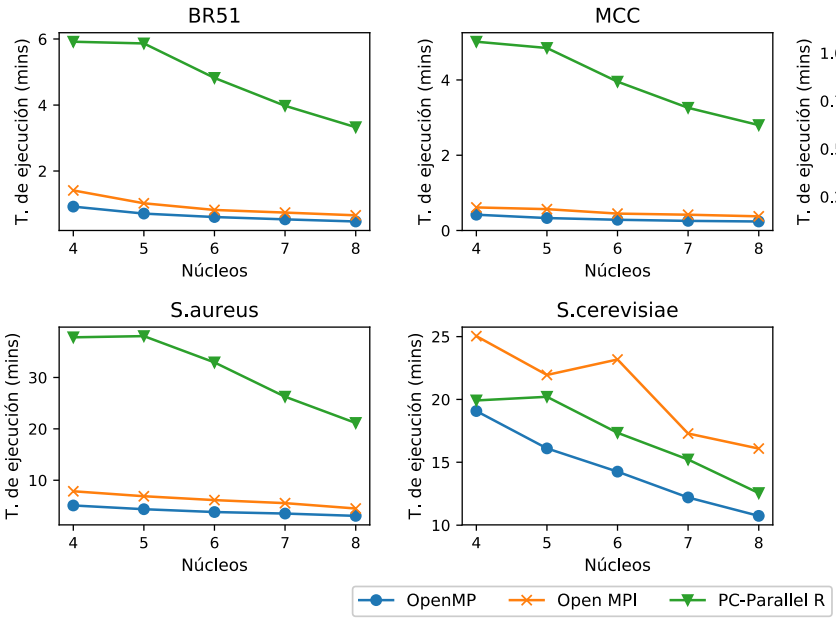


Figura 13: Comparación de los tiempos de ejecución con 5 conjuntos de datos reales, cada punto representa el tiempo de ejecución promedio de 5 ejecuciones. Se compara la implementación paralela con memoria compartida (Open MP), con memoria distribuida (Open MPI) y la implementación paralela de pcalg (PC-Parallel R).

Al comparar los resultados que producen las implementaciones se observó que “openMP” y “Open MPI” producen los mismos resultados para todos los conjuntos de datos, esto es lo esperado ya que se basan en la misma implementación. Sin embargo, al comparar los resultados que produce “openMP” con “PC-Parallel R” se observó que existen diferencias en el resultado que se obtuvo con el conjunto de datos MCC. En el resto de conjuntos de datos reales, las implementaciones producen los mismos resultados. El resultado de MCC varía en la cantidad de enlaces que tiene el DAG resultante. El DAG que produce “openMP” y “Open MPI” contiene 1284 enlaces dirigidos, en cambio, el DAG que produce “Parallel-PC R” contiene 1262 enlaces. La diferencia entre los dos DAGs es de 26 enlaces, un 0.001% de los enlaces posibles y un 2% de los enlaces resultantes. La diferencia entre los enlaces se obtuvo comparando cada enlace entre los dos DAG y no restando los enlaces que tiene cada uno. Como ya se indicó en la implementación del algoritmo, la diferencia que se obtiene entre los dos algoritmos es debido a los valores de la pseudoinversa que se usa para las pruebas de independencia condicional. Ya que los valores que se toman de la pseudoinversa en ciertas ocasiones varían provocando que se eliminen o mantengan ciertos enlaces. Por este motivo el que se obtengan resultados diferentes y que este comprenda un 0.001% de los posibles enlaces no es de gran importancia teniendo en cuenta que

no cambiarían las conclusiones de un estudio. Además, el objetivo del algoritmo es de obtener un DAG que represente las independencias presentes en el conjunto de datos para lo cual existen varios DAG posibles.

Otra diferencia entre las implementaciones es la cantidad de recursos (memoria RAM) que utiliza cada una. Para calcular la memoria RAM se usó la herramienta *htop* de Linux y se registró la memoria usada antes y durante la ejecución de la implementación. Los resultados que se presentan del uso de la memoria RAM se tomaron de una sola ejecución del algoritmo y no es el uso promedio por lo que no son concluyentes, pero sí dan una idea del comportamiento. Esto se realiza con el objetivo de ilustrar acerca el uso de la memoria RAM por las distintas implementaciones y no de realizar una comparación exacta. La implementación paralela con memoria compartida (openMP), es la que se observó que utiliza la menor cantidad de memoria RAM. Con el conjunto de datos BR51 y 4 núcleos utiliza alrededor de 71MB, incluso con el conjunto de datos más grande, S.cerevisiae, y 6 núcleos el uso de la memoria solo sube hasta 1.01GB. En la implementación paralela con memoria distribuida (open MPI), el uso de la memoria RAM incrementa con respecto a la implementación con memoria distribuida. Con el conjunto de datos BR51 y 4 núcleos, la implementación utiliza 225MB; con el conjunto de datos S.cerevisiae y 6 núcleos, utiliza 4.03GB. En cambio, con la implementación “Parallel-PC R” el uso de la memoria RAM crece dramáticamente. Con el conjunto de datos BR51 y 4 núcleos, la implementación utiliza 2.84GB de memoria, al incrementar a 5 núcleos el uso de memoria incrementa hasta los 5.14 GB. Con el conjunto de datos S.cerevisiae y 4 núcleos el uso de memoria incrementa hasta los 44GB. Estos resultados sugieren que la implementación paralela con memoria compartida (openMP) es la que utiliza la menor memoria RAM, en cambio la implementación paralela en R es la que utiliza la mayor memoria RAM de las implementaciones comparadas. Un estudio más exhaustivo del uso de la memoria RAM es necesario para llegar a conclusiones sólidas y se plantea como trabajo futuro.

5.5 Escalabilidad de la implementación paralela

Este conjunto de pruebas es una continuación de las realizadas en la sección 5.4, por lo que se ejecutan las mismas implementaciones en las mismas condiciones. Lo único que cambia es el número de núcleos que se utiliza. Cada implementación con un determinado conjunto de datos real, se ejecutó 5 veces con 10, 12, 14 y 16

núcleos. Estos resultados se juntan con parte de los obtenidos en la sección 5.4 para formar la gráfica de los resultados. Se ejecutaron 5 veces debido a que es el número suficiente de veces con el que se pudo comprobar que exista una diferencia significativa entre los resultados de las implementaciones.

Los resultados de las ejecuciones se presentan en la Figura 14. Cada línea de resultados está formada por los tiempos de ejecución promedio de una implementación con un conjunto de datos y de 4-16 núcleos. Lo primero que se observa es que todas las implementaciones (con 4 núcleos) tienen un tiempo de ejecución diferente, esto es lo esperado ya que a medida que se incrementa el número de variables incrementa el número de pruebas de independencia condicional a realizar y por consecuencia el tiempo ejecución. Además, estos resultados tienen el mismo comportamiento que los presentados en la Figura 12. En los 4 primeros conjuntos de datos (BR51, MCC, NC-60 y S.aureus), “Parallel-PC R” se mantiene con el mayor tiempo de ejecución, a medida que incrementa el número de núcleos el tiempo de ejecución promedio decrece aunque no llega a los tiempos de ejecución de la implementación paralela en C. En cambio, en las implementaciones “OpenMP” y “Open MPI”, a medida que incrementa el número de núcleos el tiempo de ejecución desciende llegando hasta la mitad o en el mejor de los casos un tercio del tiempo inicial con 4 núcleos. La gran diferencia en los resultados sigue siendo los obtenidos con el conjunto de datos S.Cerevisiae donde la implementación “Open MPI” se mantiene con el mayor tiempo de ejecución incluso al incrementar el número de núcleos, aunque logra reducir su tiempo de ejecución inicial a la mitad.

Los resultados más interesantes son los que se obtuvieron de la implementación “OpenMP”. En la Figura 14 se observa que, en todos los conjuntos de datos “OpenMP” tiene un comportamiento constante. Inicia y termina con el menor tiempo de ejecución, además, con cualquiera de los conjuntos de datos al incrementar el número de núcleos tiene el mismo comportamiento: descender el tiempo de ejecución promedio. La implementación “OpenMP” en los resultados, no presenta situaciones en las que, al incrementar el número de núcleos incremente su tiempo de ejecución, como es lo que pasa en ocasiones con “Open MPI” y “PC-Parallel R”. De estos resultados se puede decir que la implementación “OpenMP” tiene un buen escalamiento al incrementar la capacidad de computo (i.e. número de núcleos) y la complejidad del problema (i.e. número de variables del conjunto de datos).

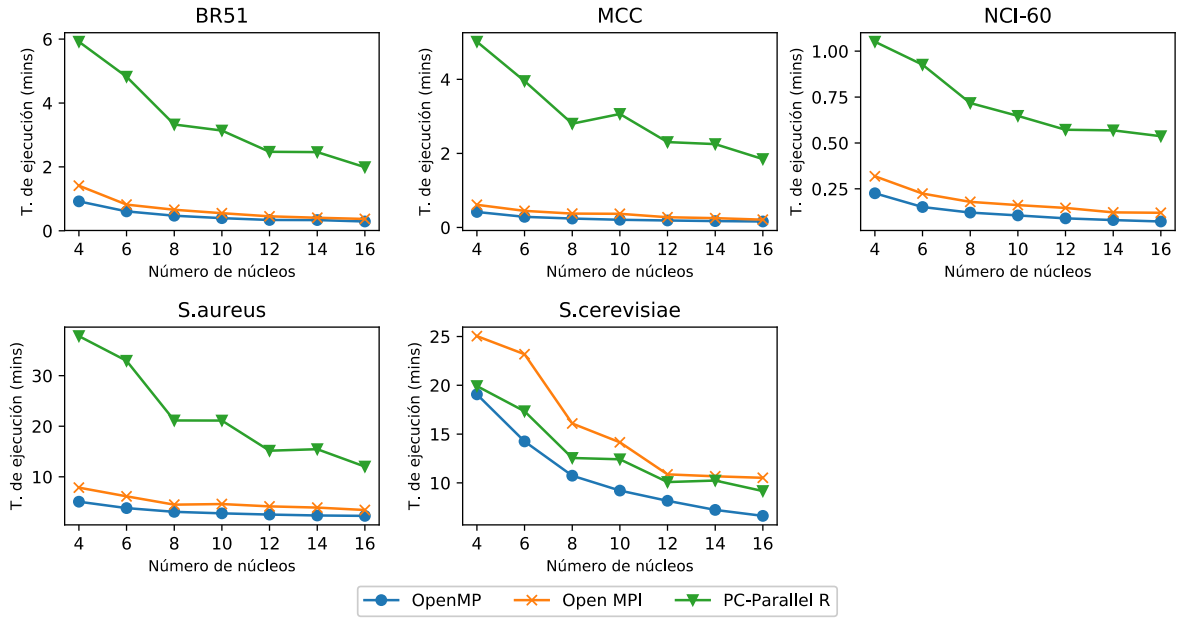


Figura 14: Comparación de los tiempos de ejecución promedio obtenidos con 5 conjuntos de datos reales. Se compara la implementación con memoria compartida (openMP), con memoria distribuida (Open MPI) y la implementación paralela de pcalg (PC-Parallel R).

5.6 Experimentos con advección y difusión

En estas pruebas se reproducen los resultados que Ebert-Uphoff & Deng (2015b) obtiene de aplicar el algoritmo PC-stable sobre simulaciones de procesos de advección y difusión, estos resultados por motivos de comparación se denominan “resultados referencia” en esta sección. La idea aquí no es comprar tiempos de ejecución sino verificar que el resultado de la red causal encontrada sea consistente con otros estudios. De cada proceso obtiene varios conjuntos de datos, los que se eligieron para reproducir los resultados son: (1) advección pura con un punto de ruido, (2) advección pura con varios puntos de ruido, (3) difusión pura con un punto de ruido y (4) difusión pura con varios puntos de ruido. Todos los resultados cuentan con dos filas de gráficas, la fila superior muestra los enlaces internos presentes desde el tiempo T=Δt hasta T= 4Δt, en cambio la fila inferior se presentan los enlaces externos presentes desde el tiempo T=0 hasta T = 3Δt. En los enlaces internos no se toma en cuenta T=0 debido a que en ese tiempo no existe memoria local. En cambio, en los enlaces externos y en específico en T=0 no se toma en cuenta las direcciones de los enlaces ya que se consideran enlaces externos concurrentes, esto debido a que son enlaces entre diferentes nodos en el mismo tiempo.

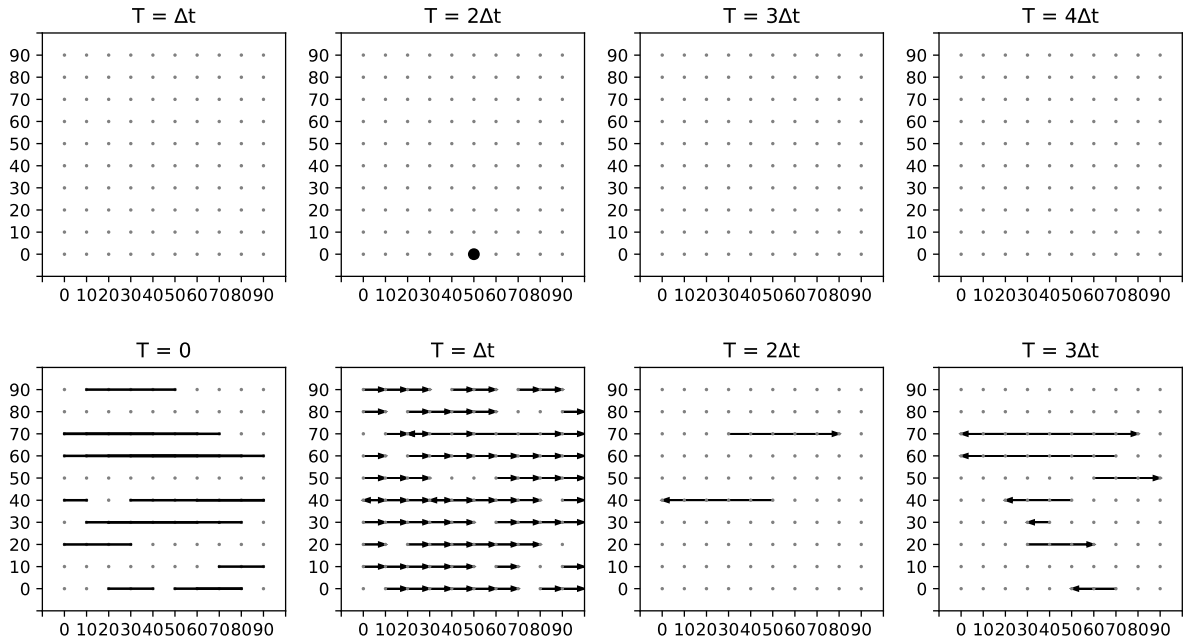


Figura 15: Enlaces internos y externos obtenidos de aplicar la implementación paralela con memoria compartida (openMP) a un conjunto de datos de una simulación del proceso de advección con un punto de ruido. La fila superior contiene los enlaces internos de los nodos, entre 4 tiempos (1-4). En cambio, la fila inferior contiene los enlaces externos entre los nodos, entre 4 tiempos (0-3).

La Figura 15 (el resultado de referencia es la figura 8a de Ebert-Uphoff & Deng (2015b)) muestra los resultados obtenidos del proceso de difusión pura con un punto de ruido. Lo primero que se observa es que, en la fila superior existe un solo enlace interno en T= 2Δt. En cambio, en fila inferior, en los 4 tiempos se presentan enlaces, el tiempo que mayor cantidad de enlaces presenta es T=2Δt, seguido de T=0. Al comparar con los resultados de referencia, se observó que en los dos casos existe un comportamiento similar para los enlaces internos. Aunque en los resultados de referencia haya más enlaces internos (15 en los 4 tiempos), esto no representan una gran diferencia ya que son un porcentaje bajo del total de los enlaces posibles en cada tiempo. Los enlaces externos son los que más se asemejan a los resultados de referencia, tanto en cantidad de enlaces en cada tiempo, como en la dirección de los enlaces dirigidos.

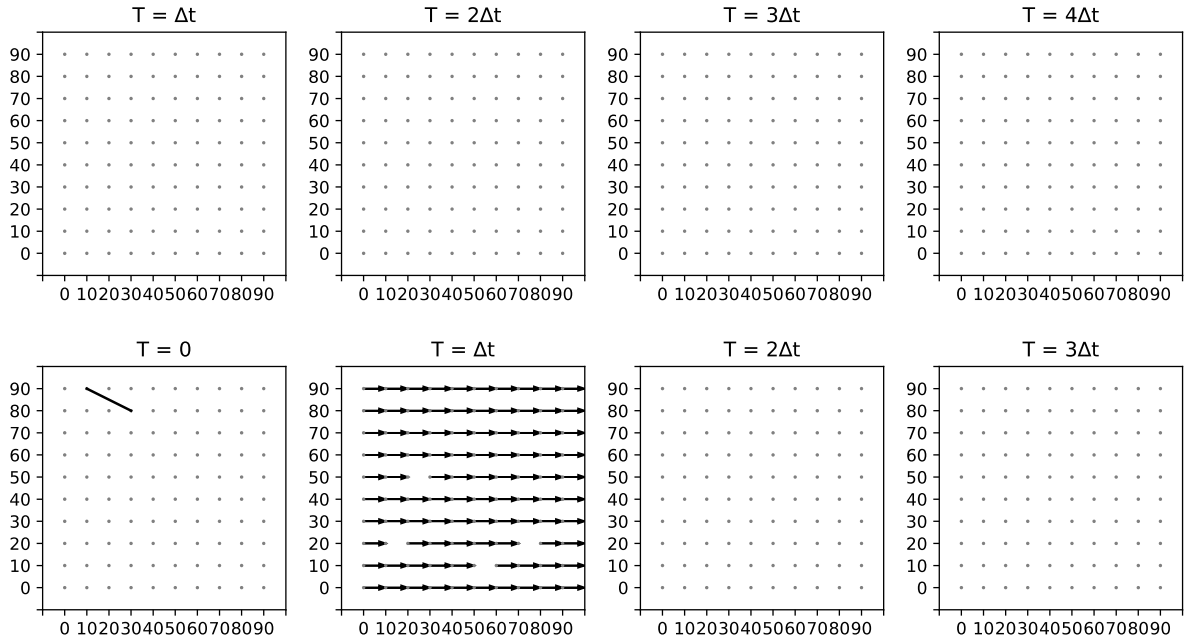


Figura 16: Enlaces internos y externos obtenidos de aplicar la implementación paralela con memoria compartida (openMP) a un conjunto de datos de una simulación del proceso de advección con varios puntos de ruido. La fila superior contiene los enlaces internos de los nodos, entre 4 tiempos (1-4). En cambio, la fila inferior contiene los externos entre los nodos, entre 4 tiempos (0-3).

Los resultados del proceso de advección con varios puntos de ruido se presentan en la Figura 16 (el resultado de referencia es la figura 8b de Ebert-Uphoff & Deng (2015b)). Lo que se observa en la fila superior, es que no existen enlaces internos. En cambio, en la fila inferior existen enlaces externos solo en los dos primeros tiempos. Al comparar con los resultados referencia se observa que tampoco contiene enlaces internos. Es en los enlaces externos donde hay más diferencias, en T=0 en ambos resultados existe un enlace externo concurrente, aunque en nuestros resultados no está ubicado en la misma región que en los resultados de referencia. Los enlaces externos de T= Δt son los más cercanos a los de los resultados referencia ya que incluye casi todos los enlaces y tienen la misma dirección. En los demás tiempos (2Δt, 3Δt) no existen enlaces externos, esto no es tan diferente a los resultados de referencia, ya que se observa que en 2Δt existen 12 enlaces y en 3Δt solo un enlace. Es decir, a medida que pasa el tiempo los enlaces desaparecen, solo que en nuestros resultados esto es más rápido.

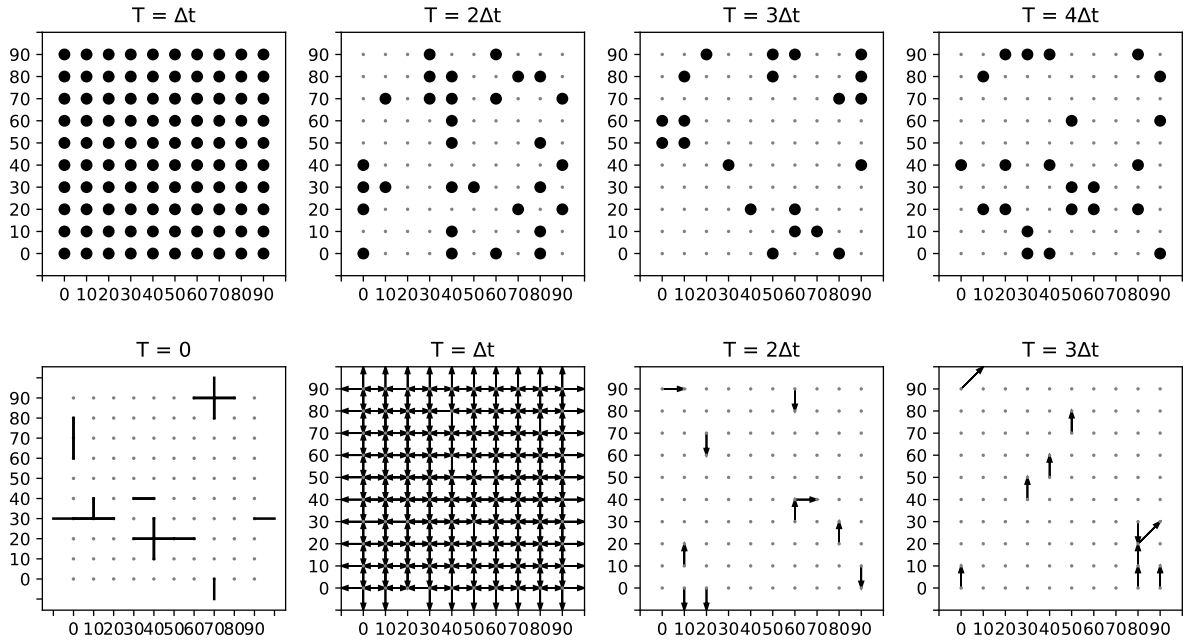


Figura 17: Enlaces internos y externos obtenidos de aplicar la implementación paralela con memoria compartida (openMP) a un conjunto de datos de una simulación del proceso de difusión con un punto de ruido. La fila superior contiene los enlaces internos de los nodos, entre 4 tiempos (1-4). En cambio, la fila inferior contiene los enlaces externos entre los nodos, entre 4 tiempos (0-3).

En la Figura 17 (el resultado de referencia es la figura 6a de Ebert-Uphoff & Deng (2015b)) se presentan los resultados del proceso de difusión pura con un punto de ruido. Lo que se observa es que, en la fila superior e inferior en todos los tiempos existen enlaces internos y externos respectivamente. Al comparar con los resultados de referencia se observa que existen diferencias con respecto a la cantidad de enlaces en algunos tiempos, aunque el comportamiento de los enlaces es el mismo. En nuestro caso, en T= Δt es donde existe igual cantidad de enlaces internos que en los resultados de referencia, en los demás tiempos el número de enlaces internos decrece hasta menos de la mitad de los enlaces en tiempo T=0. Este comportamiento de los enlaces internos a medida que pasa el tiempo es el mismo que se observa en los resultados de referencia. En el caso de los enlaces externos, es en T= Δt donde al igual en que los resultados de referencia se cuentan con la mayor cantidad de enlaces y con igual comportamiento de los enlaces dirigidos, forman grupos de 4 enlaces que apunta hacia un mismo punto. En los demás tiempos el número de enlaces externos decrece, aunque todavía quedan reminiscencias de comportamiento de los enlaces en T= Δt.

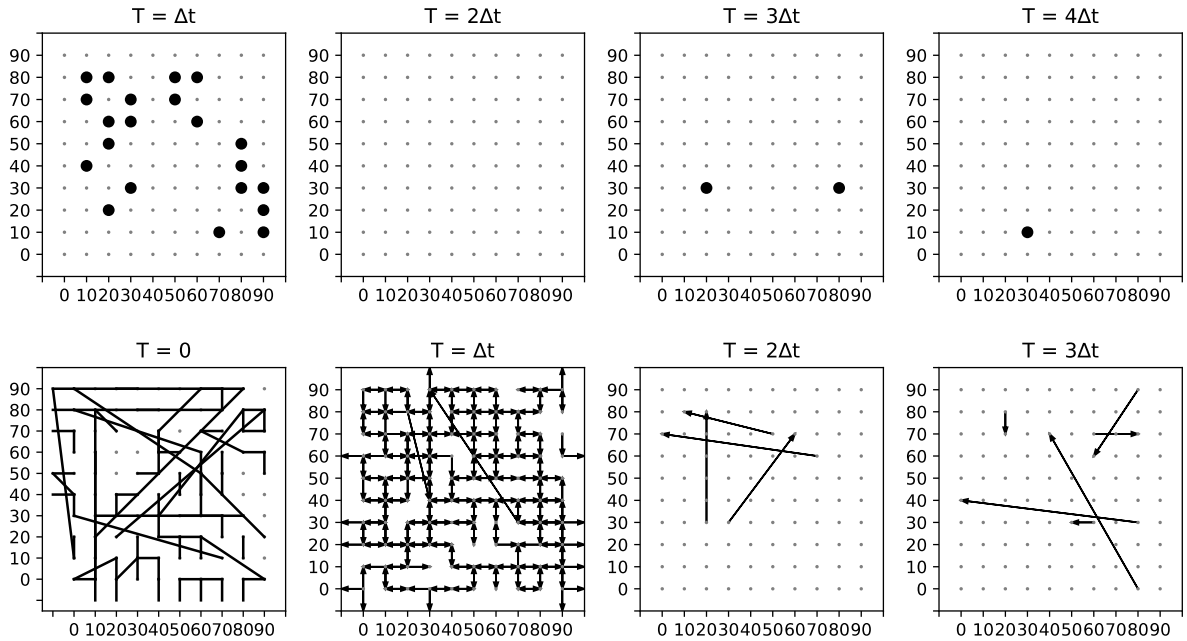


Figura 18: Enlaces internos y externos obtenidos de aplicar la implementación paralela con memoria compartida (openMP) a un conjunto de datos de una simulación del proceso de difusión pura con varios puntos de ruido. En la fila superior se muestran los enlaces internos de los nodos, entre 4 tiempos (1-4). En la fila superior se muestra los enlaces externos entre los nodos, entre 4 tiempos (0-3).

En la Figura 18 (el resultado de referencia es la figura 6b de Ebert-Uphoff & Deng (2015b)) se presentan los resultados del proceso de difusión con varios puntos de ruido. En la fila superior se observa que existen enlaces internos en los tiempos T=Δt, 3Δt y 4Δt, aunque en T= Δt es donde existe la mayor cantidad de enlaces. En cambio, en la fila inferior se observa que todos los tiempos tienen enlaces externos y al igual que en los enlaces internos, es en T= Δt donde se presenta la mayor cantidad de enlaces. Al comparar con los resultados de referencia se observa que en nuestro caso el número de enlaces presente en los diferentes tiempos es menor. En el caso de los enlaces internos, la proporción de enlaces presentes en cada tiempo es similar al de los resultados de referencia. Sin embargo, los enlaces externos son los que tienen un comportamiento más cercano a los resultados de referencia, en T=0 se observa que los enlaces no tienen un comportamiento definido en los dos casos. En el tiempo T = Δt los resultados obtenidos tienen el mismo comportamiento que los resultados de referencia, es decir existen grupos de 4 enlaces que se orientan hacia un mismo punto. En los demás tiempos de los enlaces externos, al igual que en los resultados de referencia, el número de enlaces disminuye y no tienen un comportamiento definido como en T= Δt.

De acuerdo a los resultados obtenidos de reproducir los experimentos se concluye que, en todos los casos, nuestra implementación, en general, se comporta de manera similar a los resultados de referencia. Esto se observa de forma más clara en los enlaces externos presentes en el T= Δt de cada resultado, además, en todos los casos tienen una proporción similar de enlaces. La principal diferencia de los resultados es el número de enlaces en cada tiempo, nuestra implementación es más drástica al eliminar enlaces. Esto es más apreciable en los enlaces internos de los diferentes tiempos. La excepciones son los enlaces internos de la Figura 16 y Figura 17 en el tiempo T= Δt, ya que la cantidad de enlaces internos con respecto a los resultados de referencia es el mismo. La diferencia de la cantidad de enlaces puede deberse a que nuestra configuración de la implementación sea diferente a la usada para obtener los resultados de referencia ya que no se cuenta con todos los detalles usados. Lo que se observa en general es que nuestra implementación funciona correctamente al obtener estructuras con enlaces que tienen mismo comportamiento que los resultados de referencia. No existen aberraciones en los enlaces resultantes, en el peor de los casos se eliminan todos los enlaces en un tiempo, aunque esto está acompañado de un comportamiento similar en los resultados de referencia.

Capítulo 6. Conclusiones

El aprendizaje de estructuras causales a partir de un conjunto de datos es uno de los métodos de aprendizaje automático más importantes en ciertos campos de estudio (e.g. biología, climatología). Esto debido a que no permiten realizar experimentación directa y en cambio cuentan con grandes cantidades de información de variables que representan el comportamiento de un sistema complejo. Uno de los algoritmos de aprendizaje de estructura que realiza una selección inteligente y ordenada de las preguntas para recuperar una estructura causal (DAG) es el algoritmo de PC (Abellán et al., 2006). En este trabajo se implementó el algoritmo de PC original y sus variaciones (PC-stable, CPC) basándose en pcalg y en TETRAD para agregar conocimiento a priori a cada paso del algoritmo. La implementación se realizó en el lenguaje de programación C con la librería GSL para el manejo de matrices y vectores. Debido a que el resultado del algoritmo de PC depende del orden en el que están dispuestas las variables en el conjunto de datos, Colombo & Maathuis (2014) plantearon PC-stable que modifica la obtención de *skeleton* e incluye y modifica lo que plantea CPC. El resultado es un algoritmo independiente del orden que tienen las bases fundamentales para su paralelización, sin embargo, implica realizar más pruebas de independencia condicional lo que conlleva en un aumento del tiempo de ejecución del algoritmo.

Para reducir el tiempo de ejecución del algoritmo se realizó una implementación paralela con dos enfoques: con memoria compartida y distribuida. Se utilizó openMP para paralelizar el algoritmo con memoria compartida y Open MPI para memoria distribuida. En nuestra experiencia al realizar la paralelización con Open MPI se constataron varias desventajas que tiene con respecto a la paralelización con OpenMP que realizamos. Debido a que Open MPI utiliza procesos y no hilos, requiere de más recursos para realizar la paralelización, ya que cada proceso es un ambiente de ejecución independiente. En cambio, en OpenMP los hilos se ejecutan sobre un mismo ambiente de ejecución y aunque mantienen un estado privado, su sincronización es posible con variables compartidas. Open MPI a diferencia de openMP requiere de grandes cambios en el código del algoritmo, de cada método a paralelizar es necesario un método auxiliar que ejecutan los procesos trabajadores. Además, requiere diferenciar las partes que realiza el proceso MASTER de las que realizan los procesos trabajadores. La sincronización entre los procesos es una tarea compleja ya que los procesos no se crean dinámicamente, se crean al inicio y

se mantiene vivos durante la ejecución. Esto es un problema cuando se tienen partes que se ejecutan en paralelo y en serie. Puede darse la situación en la que al paralelizar un conjunto de tareas con Open MPI sea más lento que realizarlo en serie. Esto cuando el tiempo necesario para compartir información entre los trabajadores es muy grande. En nuestra situación usar Open MPI para paralelizar la implementación no fue la mejor elección ya que para paralelizar un conjunto de tareas es necesario compartir mucha información entre los procesos trabajadores. Además, los sistemas donde se ejecutó la implementación son de memoria compartida y al usar OpenMPI, para compartir información entre trabajadores se copia direcciones de memoria a diferencia de OpenMP que se comparte la misma dirección. La solución ideal que plantea (Quinn, 2008) es una implementación que combine OpenMP con Open MPI. De esta manera se crean procesos que a su vez crean hilos. Esto alivianaría la paralelización al requerir el paso de menos mensajes entre procesos. Esta solución se plantea como trabajo futuro.

Durante el desarrollo de la implementación se realizaron pruebas para determinar si nuestra implementación genera los mismos resultados que pcalg. Se encontró que, en 6 de los 1200 conjuntos de datos sintéticos usados, los DAGS resultantes varían en el peor de los casos en 22 enlaces de 910 resultantes. Esto se debe a la pseudoinversa que obtiene cada implementación y cómo se interpreta el valor de tolerancia al momento de obtener la pseudoinversa. Se concluyó que 22 enlaces representan un porcentaje muy bajo de los enlaces resultantes en el DAG y que esto no cambia la naturaleza del resultado ya que el algoritmo de PC busca una representación de las independencias encontradas en el conjunto de datos y no un DAG exacto.

El rendimiento de nuestra implementación con sus dos modelos de ejecución se compararon con pcalg y la implementación paralela que Le et al. (2016) realiza de pcalg. En las pruebas de rendimiento en serie se comprueba que nuestra implementación llega a ser hasta 30 veces más rápido que pcalg, en cambio, la implementación paralela con memoria compartida (openMP) y distribuida (Open MPI) en el mejor de los casos es 3.7 y 3.35 veces más rápidas que la implementación en serie en C respectivamente, en los dos casos con el conjunto de datos sintético de 1500 variables. Con conjuntos de datos reales se comprobó que generalmente la versión paralela de pcalg “Parallel-PC R” es más lenta que la implementación paralela con memoria compartida (openMP). Se encontró que en el mejor de los casos “openMP” es 11 veces más rápida que “Parallel PC R” con el

conjunto de datos MCC y en el peor de los casos es solo 1.3 veces más rápida con el conjunto de datos S.cerevisiae. De acuerdo a los resultados con los conjuntos de datos reales se concluye que la implementación paralela con memoria compartida (openMP) es la que mejor escala tanto cuando incrementa la capacidad computacional (núcleos) como cuando incrementa la complejidad del problema (número de variables)

Finalmente se comprobó el comportamiento de la implementación paralela con memoria compartida (openMP) ante conjuntos de datos de simulaciones de procesos de advección y difusión que contienen puntos de ruido. Al comparar los resultados obtenidos con los que obtuvo Ebert-Uphoff & Deng (2015b) al usar el algoritmo PC-stable (resultados de referencia) se llegó a varias conclusiones. En nuestros resultados, en la mayoría de los tiempos existen menos enlaces internos y externos, nuestra implementación es más drástica al eliminar enlaces, aun cuando se usó el mismo valor de significancia (0.05). Aunque, existen excepciones ya que en ciertos tiempos (Figura 16, Figura 17) el número de enlaces en los dos resultados es el mismo. De los enlaces resultantes se comprobó que siguen el mismo patrón de aparición que en los resultados de referencia. Además, las direcciones de los enlaces es la misma que en los resultados de referencia, en advección los enlaces se dirigen hacia la derecha y en difusión se crean grupos de enlaces que apuntan hacia un mismo lugar. Es decir que nuestra implementación genera resultados que son consistentes con los obtenidos por Ebert-Uphoff & Deng (2015b).

Realizadas las pruebas de rendimiento, es difícil sacar conclusiones generales en cuanto al tiempo que requiere una implementación para completar su ejecución si solo se conoce el número de variables del conjunto de datos. El tiempo de ejecución depende de otros factores tanto internos como externos al algoritmo e.g. la dinámica del sistema (factor externo), el valor de significancia *alpha* (factor interno). Estos factores afectan a la cantidad de operaciones (e.g. pruebas de independencia condicional) que el algoritmo debe realizar y por consecuencia el tiempo de ejecución. Al concluir este trabajo se ganó experiencia en el desarrollo con el lenguaje de programación C, programación en paralelo con memoria compartida y distribuida y el manejo de grandes cantidades de información. Además, se ganó experiencia en el uso de algoritmos de aprendizaje automático para obtener estructuras causales a partir de grandes cantidades de información de sistemas complejos reales para fines de investigación.

7. Referencias

Abellán, J., Gómez-Olmedo, M., & Moral, S. (2006). Some Variations on the PC Algorithm. En *Probabilistic Graphical Models* (pp. 1–8).

Acid, S., & de Campos, L. M. (2011). Searching for Bayesian Network Structures in the Space of Restricted Acyclic Partially Directed Graphs. *arXiv:1107.0019*. https://doi.org/10.1613/jair.1061

Acyclic Graph & Directed Acyclic Graph: Definition, Examples. (2016). Recuperado el 9 de febrero de 2018, a partir de http://www.statisticshowto.com/directed-acyclic-graph/

Advanced OpenMP Topics. (2012, septiembre). Presentado en NAS Webinar, NASA Advanced Supercomputing Division. Recuperado a partir de https://www.nas.nasa.gov/hecc/assets/pdf/training/Advanced_OpenMP_Topics_2012_09_12.pdf

Barabási, A.-L., & Pósfai, M. (2016). *Network Science* (1era edición). Cambridge, United Kingdom: Cambridge University Press.

Bayesialab. (2018, abril 2). Bayesian Networks. Recuperado el 4 de febrero de 2018, a partir de http://www.bayesia.com/bayesian-networks-reasoning

Bayesian Networks: Reasoning. (2018, abril 2). Recuperado el 4 de febrero de 2018, a partir de http://www.bayesia.com/bayesian-networks-reasoning

Bogaerts, M. (2016, febrero 16). Compute the (Moore-Penrose) pseudo-inverse of a libgsl matrix in plain C. Recuperado el 26 de febrero de 2018, a partir de https://gist.github.com/turingbirds/5e99656e08dbe1324c99

Borsboom, D., & Cramer, A. O. J. (2013). Network Analysis: An Integrative Approach to the Structure of Psychopathology. *Annual Review of Clinical Psychology*, *9*(1), 91–121. https://doi.org/10.1146/annurev-clinpsy-050212-185608

Bühlmann, P., Kalisch, M., & Meier, L. (2014). High-Dimensional Statistics with a View Toward Applications in Biology. *Annual Review of Statistics and Its Application*, *1*(1), 255–278. https://doi.org/10.1146/annurev-statistics-022513-115545

Chalmer. (1986). *Understanding Statistics*. CRC Press.

Chapman, B., Jost, G., & Pas, R. van der. (2008). *Using OpenMP: portable shared memory parallel programming*. Cambridge, Mass: MIT Press.

Colombo, D., & Maathuis, M. H. (2014). Order-independent constraint-based causal structure learning. *Journal of Machine Learning Research*, *15*(1), 3741–3782. Recuperado a partir de http://www.jmlr.org/papers/volume15/colombo14a/source/colombo14a.pdf

Cousens, S., Diaz-Ordaz, K., Silverwood, R., Keogh, R., & Vansteelandt, S. (2018). Causal Inference. Recuperado el 5 de febrero de 2018, a partir de http://csm.lshtm.ac.uk/centre-themes/causal-inference/

De Jongh, M. (2014). *Algorithms for constraint-based learning of Bayesian network structures with large numbers of variables* (Tesis PHD). University of Pittsburgh.

Deng, Y., & Ebert-Uphoff, I. (2014). Weakening of atmospheric information flow in a warming climate in the Community Climate System Model. *Geophysical Research Letters*, *41*(1), 193–200. https://doi.org/10.1002/2013GL058646

Ebert-Uphoff, I., & Deng, Y. (2010). *Causal discovery methods for climate networks*. Georgia Institute of Technology.

Ebert-Uphoff, I., & Deng, Y. (2012a). A new type of climate network based on probabilistic graphical models: Results of boreal winter versus summer. *Geophysical Research Letters*, *39*(19), L19701. https://doi.org/10.1029/2012GL053269

Ebert-Uphoff, I., & Deng, Y. (2012b). Causal Discovery for Climate Research Using Graphical Models. *Journal of Climate*, *25*(17), 5648–5665. https://doi.org/10.1175/JCLI-D-11-00387.1

Ebert-Uphoff, I., & Deng, Y. (2014). Causal Discovery from Spatio-Temporal Data with Applications to Climate Science. En *Proceedings of the 13th International Conference on Machine Learning and Applications* (pp. 606–613). Detroit, USA: IEEE. https://doi.org/10.1109/ICMLA.2014.96

Ebert-Uphoff, I., & Deng, Y. (2015a). Identifying Physical Interactions from Climate Data: Challenges and Opportunities. *Computing in Science & Engineering*, *17*(6), 27–34. https://doi.org/10.1109/MCSE.2015.129

Ebert-Uphoff, I., & Deng, Y. (2015b). *Using Causal Discovery to Track Information Flow in Spatio-Temporal Data - A Testbed and Experimental Results Using Advection-Diffusion Simulations* (Reporte Técnico) (p. 40). Recuperado a partir de https://arxiv.org/pdf/1512.08279v1.pdf

Emmert-Streib, F., & Dehmer, M. (Eds.). (2008). *Analysis of Microarray Data: A Network-Based Approach* (1era edición). Weinheim: Wiley-Blackwell.

Friedman, N., & Goldszmidt, M. (1998). *Learning Bayesian Networks from Data*.

Goosse, H. (2015). *Climate System Dynamics and Modelling* (Reprint edition). New York, NY: Cambridge University Press.

Gravetter, F. J., & Wallnau, L. B. (2010). *Essentials of Statistics for the Behavioral Sciences*. Cengage Learning.

Greville, T. N. E. (1958). The pseudoinverse of a rectangular matrix and its statistical applications. *Amer. Statist. Assoc., Proc. Soc. Statist. Sect*, 116–121.

Haldar, S. (2015). *Operating Systems* (1era Edición). Sibsankar Haldar.

Halpern, J. Y. (2016). *Actual Causality*. MIT Press.

Hammerling, D., Baker, A. H., & Ebert-Uphoff, I. (2015). What can we learn about climate model runs from their causal signatures? En *Proceedings of the Fifth International Workshop on Climate Informatics (CI2015)*. Boulder, Colorado, USA. Recuperado a partir de https://goo.gl/GhVk4Z

Iversen, G. R., & Gergen, M. (2012). *Statistics: The Conceptual Approach*. Springer Science & Business Media.

Jaffar, A. (2016, abril 5). The Importance of Computer Programming. Recuperado el 23 de febrero de 2018, a partir de https://www.linkedin.com/pulse/importance-computer-programming-azirulsyazwan-jaffar

Kalisch, M., Hauser, A., Maechler, M., Colombo, D., Entner, D., Hoyer, P., … Schuerch, M. (2017). pcalg: Methods for Graphical Models and Causal Inference (Versión 2.5-0). Recuperado a partir de https://cran.r-project.org/web/packages/pcalg/index.html

Kalisch, M., Mächler, M., Colombo, D., Hauser, A., Maathuis, M. H., & Bühlmann, P. (2014). More Causal Inference with Graphical Models in R Package pcalg. *Journal of Statistical Software*.

Kenward, A. (2011). Data Storm: What To Do With All This Climate Information? [Blog]. Recuperado el 3 de junio de 2017, a partir de http://www.climatecentral.org/blogs/data-storm-what-to-do-with-all-this-climate-information

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. Cambridge, Mass.: MIT Press.

Kuznetsov, S. P., Pikovsky, A., & Rosenblum, M. (2010). Collective Phase Chaos in the Dynamics of Interacting Oscillator Ensembles. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *20*(4), 043134. https://doi.org/10.1063/1.3527064

Le, T., Hoang, T., Li, J., Liu, L., Liu, H., & Hu, S. (2016). A fast PC algorithm for high dimensional causal discovery with multi-core PCs. *IEEE/ACM transactions on computational biology and bioinformatics*.

Meek, C. (1995). Causal inference and causal explanation with background knowledge. En *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence* (pp. 403–410). Morgan Kaufmann Publishers Inc. Recuperado a partir de http://dl.acm.org/citation.cfm?id=2074204

Nielsen, T. D., & Jensen, F. V. (2009). *Bayesian networks and decision graphs*. Springer Science & Business Media.

Pearl, J. (2009). *Causality: Models, Reasoning and Inference* (2da edición). Cambridge, U.K. ; New York: Cambridge University Press.

Prakash, P. (2017, mayo 17). High level languages - advantages and disadvantages. Recuperado el 23 de febrero de 2018, a partir de https://codeforwin.org/2017/05/high-level-languages-advantages-disadvantages.html

Quinn, M. J. (2008). *Parallel Programming in C with Mpi and Openmp*. Boston, Mass. u.a: McGraw-Hill Education.

Ramsey, J., Zhang, J., & Spirtes, P. L. (2012). Adjacency-faithfulness and conservative causal inference. *arXiv preprint arXiv:1206.6843*.

Rongala, A. (2015, mayo 13). Benefits of C / C++ over Other Programming Languages. Recuperado el 23 de febrero de 2018, a partir de

https://www.invensis.net/blog/it/benefits-of-c-c-plus-plus-over-other-programming-languages/

Runge, J. (2014). Detecting and quantifying causality from time series of complex systems. http://dx.doi.org/10.18452/17017

Runge, J., Sejdinovic, D., & Flaxman, S. (2017). Detecting causal associations in large nonlinear time series datasets. *ArXiv:1702.07007*. Recuperado a partir de https://arxiv.org/pdf/1702.07007.pdf

Scutari, M. (2014). Bayesian network constraint-based structure learning algorithms: Parallel and optimised implementations in the bnlearn r package. *arXiv:1406.7648*.

Shaughnessy, J. J., Zechmeister, J. S., & Zechmeister, E. B. (2014). *Research Methods in Psychology* (10 edition). Dubuque: McGraw-Hill Education.

Sofroniou, A. (2017). *PROCESSES OF THINKING, CREATIVITY AND IDEOLOGIES*. Lulu.com.

Spirtes, P., Glymour, C., & Scheines, R. (1993). *Causation, Prediction, and Search* (Vol. 81). New York, NY: Springer New York. https://doi.org/10.1007/978-1-4612-2748-9

Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). *Unix Network Programming, The Sockets Networking API* (3era edición, Vol. 1). Addison-Wesley Professional.

Tan, Y., & Liu, Z. (2013). Feature selection and prediction with a Markov blanket structure learning algorithm. *BMC Bioinformatics*, *14*(Suppl 17), A3. https://doi.org/10.1186/1471-2105-14-S17-A3

Treur, J. (2016). *Network-Oriented Modeling: Addressing Complexity of Cognitive, Affective and Social Interactions*. Springer.

Tsonis, A.A., & Roebber, P. J. (2004). The architecture of the climate network. *Physica A: Statistical Mechanics and Its Applications*, *333*, 497–504. https://doi.org/10.1016/j.physa.2003.10.045

Tsonis, Anastasios A., Swanson, K. L., & Roebber, P. J. (2006). What Do Networks Have to Do with Climate? *Bulletin of the American Meteorological Society*, *87*(5), 585–596. https://doi.org/10.1175/BAMS-87-5-585

Walliman, N. (2011). *Research methods: the basics*. London: Routledge.

Weisstein, E. W. (2018a, marzo 8). Causal Network [Texto]. Recuperado el 8 de marzo de 2018, a partir de http://mathworld.wolfram.com/CausalNetwork.html

Weisstein, E. W. (2018b, marzo 8). Substitution System [Texto]. Recuperado el 8 de marzo de 2018, a partir de http://mathworld.wolfram.com/SubstitutionSystem.html

Yi, D., & Ebert-Uphoff, I. (2014). High efficiency implementation of PC and PC stable algorithms yields three-dimensional graphs of information flow for the Earth'atmosphere. *Technical report (Colorado State University. Department of Electrical and Computer Engineering); Report Nr. CSU-ECE-2014-1*.

Anexos

Anexo A: Representación del conocimiento a priori

El conocimiento a priori dentro del algoritmo de PC, PC-stable y CPC implementado, se representa con una matriz cuadrada asimétrica. La matriz tiene las mismas dimensiones que la matriz de adyacencia que representa el *skeleton* del DAG y tiene un funcionamiento similar a la matriz de adyacencia. Dentro de la matriz se utiliza el valor “1” para especificar un enlace dirigido requerido y “2” para especificar la dirección de un enlace que está prohibido. En la Tabla 9, el enlace $X \rightarrow Y$ es requerido por conocimiento a priori, en cambio, el enlace $X \rightarrow Z$ está prohibido (no significa que el enlace $Z \rightarrow X$ sea requerido). Cuando se requiere prohibir el enlace en su totalidad (i.e. en ambas direcciones) se utiliza el valor “2” en las dos posiciones del enlace, en la Tabla 9, el enlace entre Z─Y está prohibido.

Tabla 9: Ejemplo de conocimiento a priori.

	X	Y	Z
X		1	2
Y			2
Z		2	

Anexo B: Manual de uso

Este programa es la implementación del algoritmo de PC con los cambios que introduce CPC y PC-stable. La implementación se realizó en base a pcalg[28], TETRAD[29] y una implementación paralela de pcalg (Le et al., 2016). El programa incluye dos modelos de ejecución: serie y paralelo. La ejecución en paralelo es posible sobre sistemas con memoria compartida y distribuida. Las funciones generales del programa que representan cada paso del algoritmo son:

- Cálculo de la matriz de correlación.
- Obtención de *skeleton*.
- Orientación de enlaces mediante conocimiento a priori.
- Procesamiento de tripletas del grafo.
- Orientación de tripletas
- Orientación de enlaces restantes.

El algoritmo se implementó en el marco del desarrollo del trabajo de titulación "Implementación de algoritmos de inferencia causal utilizando computación paralela: PC y PC-stable".

1. Requerimientos

El programa se desarrolló en el lenguaje de programación C y se utilizaron las librerías: GSL[30], openMP[31] y Open MPI[32]. GSL se utiliza para el manejo de matrices y vectores, además para realizar la descomposición de valores singulares, obtención de coeficientes de correlación entre otras funciones. La paralelización del cálculo de la matriz de correlación, obtención de *skeleton* y el procesamiento de tripletas, se realizó con openMP para sistemas con memoria compartida y Open MPI para sistemas con memoria distribuida. El programa se desarrolló para la plataforma Linux aunque es posible que funcione en Windows mediante Cygwin[33], esto último no se ha probado.

[28] https://cran.r-project.org/package=pcalg
[29] http://www.phil.cmu.edu/tetrad/
[30] https://www.gnu.org/software/gsl/
[31] http://www.openmp.org/
[32] https://www.open-mpi.org/
[33] https://www.cygwin.com/

El programa para funcionar requiere de un archivo CSV (separado por espacios) con n muestras de m variables. El archivo CSV se compone de una matriz de n x m, donde n es el número de filas del archivo (igual al número de muestras) y m el número de campos de cada fila. El número de campos es igual al número de variables del conjunto de datos por lo que cada campo representa una muestra de una variable. Dado el código fuente del programa, para su funcionamiento requiere de compilar y enlazar las librerías GSL. Dependiendo de la implementación se compila y enlaza con gcc[34], mpicc[35] y g++[36]. Una vez compilado, la implementación requiere que la variable de ambiente del sistema: "LD_LIBRARY_PATH", contenga la dirección de la carpeta "lib" de la instalación de GSL, caso contrario no ejecutará el programa.

2. Estructura de la implementación

El programa está contenido en 3 archivos, cada archivo representa una implementación con un modelo de ejecución del algoritmo. Esto se realizó de esta manera ya que para realizar la paralelización es necesario realizar cambios estructurales del código. Los archivos son:

- **Main.c**: contiene el código fuente de la implementación en serie del algoritmo.
- **MainOpenMP.c**: contiene el código fuente de la implementación en paralelo con memoria compartida que utiliza openMP. Esta implementación se puede configurar para que ejecute en serie el algoritmo.
- **MainOpenMPI.c**: contiene el código fuente de la implementación en paralelo con memoria distribuida que utiliza Open MPI. Debido a los cambios necesarios para la paralelización con memoria distribuida, solo puede ejecutar la versión paralela.

[34] https://gcc.gnu.org/
[35] Es un envoltorio (*wrapper*) alrededor de un conjunto de compiladores y se utiliza para para compilar y enlazar programas MPI escritos en C.
[36] Es el compilador GNU de C++

3. Configuración y compilación de las implementaciones

Configuración del programa

El código fuente de las 3 implementaciones del algoritmo que componen el programa cuentan con dos métodos (*main* y *pc*) con parámetros a configurar para la ejecución. El método *main()* es el principal de la implementación, se encarga de definir las entradas del algoritmo e invoca al método *pc()*, los parámetros que aquí se deben configurar son:

- **pathResultados**: indica el directorio donde se guardarán los resultados de la ejecución del algoritmo.
- **pathDatasets**: dirección (*path*) del archivo CSV con el conjunto de datos de entrada.
- **p**: número de variables del conjunto de datos.
- **n**: número de muestras de las variables del conjunto de datos.
- **nThreads**: indica el número hilos a usar durante la ejecución de la implementación paralela con memoria compartida.
- **verbose**: variable booleana que indica si se debe mostrar más información de los procesos que se realizan durante la ejecución del algoritmo.

En cambio, *pc()* se encarga de invocar a los métodos que representan cada paso del algoritmo de PC. En este método se define:

- **bknowledge**: matriz cuadrada de n x n, donde se define el conocimiento a priori de las variables del conjunto de datos (Anexo A).
- **alpha:** valor de significancia, usado para en las pruebas de independencia condicional.

Para compilar y ejecutar cualquiera de las implementaciones es necesario que el sistema donde se va a ejecutar tenga instalado el compilador gcc y GSL. Además, es necesario que la variable ambiente del sistema cuente con el directorio de instalación de GSL, para lo cual se utiliza la siguiente sentencia en una terminal:

```
LD_LIBRARY_PATH=/pathToGSL/lib
export LD_LIBRARY_PATH
```

El parámetro “pathToGSL” se reemplaza con el directorio de instalación de GSL que contiene las carpetas: bin, include, lib, y share.

Compilación de la implementación en serie o paralela con memoria compartida

La compilación se realiza para una de las implementaciones (main.c o mainOpenMP.c) con la siguiente sentencia en una terminal de Linux ubicada en el directorio donde están los archivos de la implementación.

```
gcc -O2 -std=c99 -fopenmp -I/pathToGSL/gsl/include -c sourceCode.c -
o  sourceCode.o
```

Los parámetros que se utilizan en la sentencia de compilación son:

- **-02**: Indica al compilador que utilice la optimización de nivel 2, este valor se puede cambiar dependiendo de las necesidades de la compilación (e.g. 03, ofast).
- **std=c99**: indica al compilador que el código fuente está escrito con el estándar de C, c99.
- **-fopenmp**: indica al compilador que en la implementación se utiliza openMP para paralelización. Es necesario para cualquiera de las implementaciones.
- **pathToGSL**: indica el directorio donde se ubica la instalación de GSL.
- **sourceCode.c**: nombre archivo con el código fuente de la implementación a compilar, se reemplaza por main.c (serie) o mainOpenMP.c (paralelo memoria compartida).
- **sourceCode.o**: archivo objeto generado de la compilación.

Una vez compilado el código fuente de la implementación, se genera un archivo con la extensión “o” (sourceCode.o). Este archivo se utiliza con g++ para enlazar las librerías de GSL utilizadas en la implementación. Para este objetivo se utiliza la siguiente sentencia:

```
g++ -L/pathtoGSL/lib -o salida sourceCode.o -s -fopenmp -lgsl -lgslcblas -
lm /pathtoGSL/lib/libgsl.a pathtoGSL/lib/libgslcblas.a
```

El parámetro “salida” indica el nombre del archivo ejecutable a generar, el número de hilos con el que se ejecuta se define en el método *main()* (**nThreads**) al invocar a *pc()*. Para ejecutar la implementación se utiliza el comando:

```
./salida
```

Compilación del programa paralelo con memoria distribuida

Si en cambio se quiere compilar y ejecutar la implementación paralela con memoria distribuida es necesario instalar Open MPI, además de gcc y GSL. Para compilar se utiliza la siguiente sentencia en una terminal de Linux ubicada en el directorio donde se ubica el archivo con el código fuente de la implementación (mainOpenMPI.c).

```
mpicc -std=c99 -fopenmp -I/pathToGSL/include -c mainOpenMPI.c -
o  mainOpenMPI.o
```

Los parámetros usados en la sentencia son los mismos que para compilar la implementación en serie. Una vez compilada la implementación se genera el archivo "mainOpenMPI.o" con el código objeto de la implementación, a continuación, se utiliza la siguiente sentencia para enlazar las librerías GSL:

```
mpicc -L/pathToGSL/lib -o salida mainOpenMPI.o -fopenmp -lgsl -lgslcblas -
lm /pathToGSL/lib/libgslcblas.a /pathToGSL/lib/libgslcblas.so
/pathToGSL/lib/libgsl.so /pathToGSL/lib/libgsl.a
```

Al ejecutar la sentencia se genera el archivo "salida" que es el ejecutable de la implementación, para iniciar la ejecución se utiliza:

```
mpirun salida –np numberOfProcesses
```

La opción "numberOfProcesses" indica el número de procesos para la paralelización con memoria distribuida.

4. Cálculo de la matriz de correlación

Este el primer paso del algoritmo de PC y el método que lo representa es *correlationMatrix()*. Recibe como entrada una matriz con el conjunto de datos leído desde el archivo CSV de entrada, el número de variables, el número de muestras y el número de hilos (paralelización con memoria compartida). El resultado de este método es una matriz cuadrada de n x n simétrica (n igual al número de variables del conjunto de datos) con los coeficientes de correlación de cada variable con respecto a las demás variables. Para obtener el coeficiente de correlación de una variable utiliza parte del conjunto de datos y la función *gsl_stats_correlation()*. Este es uno de los métodos que se ha paralelizado y su paralelización consiste en repartir entre los trabajadores el conjunto de variables. Cada trabajador calcula los coeficientes de correlación de una variable del subconjunto que se le ha asignado.

6. Obtención de *skeleton*

Es el segundo paso del algoritmo de PC, el método que lo representa es *skeleton()*. En este método se obtiene el *skeleton* del DAG, a partir de un grafo completo representado en una matriz de adyacencia *G* con valores “1” y “0” que indican si existe o no un enlace dirigido entre dos nodos. En *skeleton()* se realizan pruebas de independencia condicional entre cada par de nodos para determinar si son independientes, en tal caso se elimina el enlace bidireccional entre los nodos. Recibe como entrada la matriz de correlación, el valor de significancia *alpha,* el conocimiento a priori definido en *pc()* y el número de hilos (implementación paralela con memoria compartida). El proceso se organiza en iteraciones (cada una con nivel) y lo que se realiza en cada iteración es: (1) recuperar los enlaces que contiene la matriz de adyacencia *G*, (2) dado el enlace *X—Y* formar combinaciones de nodos a partir de *adj(X, G)* con cardinalidad igual al nivel de la iteración, (3) con cada conjunto formado realizar una prueba de independencia condicional y (4) si los nodos son independientes dado un conjunto (conjunto separador) guardar en *sepset(X, Y)*. Las iteraciones continúan hasta que el número de nodos de *adj(X, G)* sea menor al nivel de la iteración.

La paralelización de este método consiste en repartir entre varios trabajadores (hilos o procesos) los enlaces recuperados en (2). Al finalizar la obtención de *skeleton* se tiene como resultado la matriz de adyacencia *G* actualizada que contiene el *skeleton* y una matriz de n x n, que almacena cada conjunto separador encontrado. Además, se generan dos archivos CSV que contienen la matriz de adyacencia *G* (*skeleton.csv*) y los conjuntos separadores de cada par de nodos (*sepset_after_skel.csv*). Estos archivos se utilizan para comprobar el comportamiento del algoritmo con respecto a otras implementaciones.

7. Orientación de enlaces mediante conocimiento a priori

Este paso no está presente el algoritmo original de PC, se introduce al incluir conocimiento a priori y lo representa el método *pcOrientbk().*Se realiza luego de obtener el *skeleton* y aplica el conocimiento a priori definido en *bknowledge*. Aquí se definen o eliminan direcciones de enlaces que están presentes en *skeleton*. Dado el enlace requerido R:*X* → *Y*, se elimina el enlace dirigido *Y* → *X* del enlace *X—Y* del *skeleton*. En cambio, si se prohibió el enlace P:X → Y, se elimina el enlace dirigido X → Y del enlace *X—Y* del skeleton. Las direcciones de los enlaces se eliminan agregando “0” en la matriz de adyacencia *G* en las posiciones de los enlaces dirigidos a eliminar e.g. *X* → *Y* corresponde a la posición [*X, Y*] de la matriz *G*. El resultado de este proceso es la matriz de adyacencia *G* actualizada con el conocimiento a priori

8. Procesamiento de tripletas

Este paso tampoco está presente el algoritmo de PC original, lo introduce CPC y lo modifica PC-stable para que tenga mayor flexibilidad al definir tripletas infieles i.e. tripletas que no se deben orientar. Se representa con el método *pc_cons_intern()* y realiza pruebas de independencia condicional con cada tripleta de nodos de la matriz de adyacencia *G* para determinar si se deben orientar como *v-structure.* El resultado que se obtiene al procesar cada tripleta es: si forma una *v-structure,* si no es una *v-structure* y si es una tripleta infiel. En los últimos dos casos la tripleta no se orienta como *v-structure* y en cambio se mantienen los enlaces bidireccionales de la tripleta. A grandes rasgos el proceso que sigue es: (1) recuperar las tripletas de la matriz de adyacencia *G*, (2) dado la tripleta *X—Y—Z* formar conjuntos de nodos con *adj(X, G)* y *adj(Z, G)*, (3) obtener los conjuntos separadores de los nodos *X* y *Z* realizando una prueba de independencia condicional con cada conjunto formado, (4) comprobar si *Y* está dentro de los conjuntos separadores y (5) de acuerdo a los resultados de (4) decidir si se orienta la v-structure.

La paralelización de este proceso se realiza a partir del paso (3). Los conjuntos formados se reparten entre los trabajadores de forma que cada trabajador realiza una prueba de independencia condicional con cada conjunto asignado. El resultado de este proceso es la matriz de conjuntos separadores (sepsets) actualizada y vector que contiene el registro de las tripletas infieles. Al finalizar este proceso se generan dos archivos CSV: uno que contiene la matriz de conjuntos separadores actualizada (*sepset_after_pccons.csv*) y un archivo que contiene el registro de tripletas infieles (*ambiguous_after_pcons.csv*).

9. Orientación de tripletas

Este paso del algoritmo de PC consiste en orientar las tripletas de nodos que contiene la matriz de adyacencia *G* i.e. crear *v-structures* a partir de las tripletas. Este paso lo representa el método *udag2pdagrelaxed()* y recibe como entrada el conocimiento a priori, la matriz de adyacencia *G*, el registro de tripletas infieles y la matriz con los conjuntos separadores (*sepsets*). El resultado de este método es una matriz de adyacencia *G* actualizada, contiene *v-structures*. A grandes rasgos lo que realiza es: (1) recuperar las tripletas que contiene la matriz de adyacencia *G*, (2) dada la tripleta *X—Y—Z* recuperar *sepset(X, Z)* y *sepset(Y, Z)*, (3) comprobar si *Y* está dentro de *sepset(X, Z)* o *sepset(Y, Z)*, (4) comprobar si la tripleta es infiel, (5) comprobar si la tripleta se puede orientar de acuerdo al conocimiento a priori y (6) orientar la tripleta como *v-structure* si (3) y (4) son falsas y (5) es verdadera. Debido a (3), (4) y (5) no todas las tripletas se podrán orientar por lo que se mantienen enlaces bidireccionales en tales casos.

10. Orientación de enlaces restantes

El último paso del algoritmo de PC es la orientación de los enlaces restantes. La orientación se realiza al aplicar las 3 primeras reglas de Meek ya que con estas se logra un DAG con la máxima orientación posible. Los métodos que representan las reglas son *rule1()*, *rule2()* y *rule3().* Las entradas de los métodos son: la matriz de adyacencia *G* y el conocimiento a priori *bknowledge.* Las 3 reglas se aplican en un bucle (*do-while*) mientras se produzcan cambios al aplicar las reglas. Dentro de cada regla se busca un patrón de nodos a los que se orienta sus enlaces de acuerdo a la regla. Una vez se ha encontrado el patrón de la regla, se comprueba si la orientación de los enlaces está permitida por conocimiento a priori. Solo si este es el caso se orientan los enlaces, caso contrario el patrón de nodos queda intacto. El resultado es una matriz de adyacencia *G* actualizada con la máxima orientación posible. Al finalizar el proceso se genera un archivo CSV (*resultado.csv*) que contiene la matriz de adyacencia *G.*

11. Archivos generados

Al finalizar la ejecución se habrán generado los siguientes archivos en texto plano:

- ***skeleton.csv****:* contiene la matriz de adyacencia del *skeleton.*
- ***sepset_after_skel.csv***: traslada una matriz (n x n) que en cada posición tiene un conjunto separador a una lista de n x n filas, donde cada fila contiene los nodos que forman parte de un conjunto separador (*sepset*).
- ***sepset_after_pccons.csv****:* tiene la misma estructura que *sepset_after_skel.csv*, es la actualización realizada al procesar las tripletas.
- ***ambiguous_after_pcons.csv***: lista de tripletas infieles.
- ***resultado.csv****:* matriz de adyacencia con el DAG resultante.
- ***execution.txt****:* contiene el tiempo de ejecución medido en ciclos del CPU y mediante el reloj de la computadora (*wall-time*).