TRABAJO FINAL DE GRADO

Paralelización de algoritmos para verificación simbólica de modelos probabilísticos

Autor:
Pablo Dal Lago
Director:
Dr. Pedro R. D'Argenio

UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE MATEMÁTICA, ASTRONOMÍA Y FÍSICA

Resumen

La verificación de modelos es una técnica útil para analizar el comportamiento de sistemas complejos. En base a una especificación formal, es posible evaluar el cumplimiento de propiedades lógicas de forma automática, asistiendo en la validación de un sistema y permitiendo la detección temprana de fallas. El modelo construido puede incorporar información probabilística, lo cual posibilita un análisis cuantitativo más detallado.

La principal variable que afecta el alcance de esta técnica es el tamaño de los modelos a considerar. La verificación *simbólica* utiliza estructuras de representación compactas para obtener una implementación eficiente. Aún así, el cálculo numérico involucrado es computacionalmente intensivo, demandando una gran cantidad de recursos de procesador y memoria.

Para combatir este problema, en este trabajo presentamos modificaciones a los algoritmos clásicos para ejecución concurrente en sistemas con múltiples procesadores. Las implementaciones paralelas se desarrollaron en base a una herramienta ya existente (PRISM), logrando reducir el tiempo necesario para llevar a cabo la tarea de verificación.

Palabras clave: métodos formales, verificación de modelos probabilísticos, cadenas de Markov, DTMC, MDP, CTMC, verificación simbólica, BDD, MTBDD, lógicas temporales, PCTL, CSL, métodos numéricos, Jacobi, Gauss-Seidel, algoritmos paralelos, programación concurrente.

Clasificación:

- D.2.4 - Software Engineering / Program Verification (Model Checking)
- D.1.3 - Programming Techniques / Concurrent Programming (Parallel Programming)

Índice general

Capítulo 1

Introducción

1.1. Motivación

La revolución informática que se inició a finales del siglo XX ha tenido un impacto tan profundo en el mundo en que vivimos que sus efectos son visibles en casi cualquier actividad del quehacer cotidiano, desde el reloj digital que nos despierta por la mañana o el horno microondas que prepara nuestro desayuno, hasta el teléfono celular que utilizamos como herramienta de comunicación o los sistemas de control y navegación en autos, aviones y otros medios de transporte.

En el centro de esta revolución vive la computadora personal, que hoy en día se encuentra presente en hogares de familia y escritorios de personas con todo tipo de empleos: contadores, ingenieros, diseñadores, etc. Su utilidad descansa en la gran variedad de aplicaciones y programas disponibles, razón por la cual la industria del software se ha convertido en una de las industrias de mayor crecimiento de los últimos tiempos.

Esto ha dado lugar al nacimiento de un área entera de profesionales que trabajan en la creación y desarrollo de programas con distintos fines. Sin embargo, al ser un área relativamente joven y que abarca un rango tan amplio, este proceso de construcción todavía presenta un gran componente artesanal, lo que si bien ha permitido la innovación constante en este campo, también dificulta enormemente la tarea de construcción de software.

Ejemplos de alto perfil como la explosión del cohete Ariane 5 en su vuelo inicial o las muertes por exceso de radiación por fallos en la máquina de radioterapia Therac-25 muestran claramente las consecuencias indeseadas de los fallos en el proceso de construcción. Pero la misma naturaleza del software hace que estos problemas sean difíciles de prevenir. A diferencia de objetos físicos como un avión, donde la falta de un tornillo en el ala no afecta en gran medida su rendimiento, el carácter abstracto del software hace que un error de una sola línea en el código de un programa pueda tener consecuencias catastróficas, como la caída del mismo avión por fallas en el sistema de control de vuelo.

Muchas técnicas han sido desarrolladas para poder combatir estos problemas. Algunas de ellas, como el “unit testing”, se basan en la examinación manual o semi-automática del comportamiento del sistema en situaciones particulares. Pero el gran problema sigue

siendo la incapacidad del ser humano de contemplar el inmenso número de variables cuyas combinaciones afectan el correcto funcionamiento de un programa.

Afortunadamente, el mismo nivel de precisión requerido a la hora de escribir un programa es el que permite realizar una especificación formal del sistema, abriendo las puertas a un nuevo conjunto de técnicas que pueden utilizarse para obtener software más robusto y confiable: los métodos formales de verificación. El subgrupo de métodos que estudiaremos aquí se conocen bajo el nombre de verificación o chequeo de modelos (model checking), en el cual partiendo de un sistema a analizar se construye mediante técnicas de abstracción un modelo que resume sus características esenciales. Esto permite luego verificar el cumplimiento de distintas propiedades, para lo cual es necesario explorar los estados del modelo construido. Otras técnicas de verificación formal están basadas en probadores automáticos de teoremas, que utilizan reglas de inferencia lógica para demostrar la veracidad de ciertas afirmaciones sobre un programa.

La verificación de modelos ha resultado ser una técnica verdaderamente exitosa, al punto que tres de los pioneros en el área (Clarke, Emerson y Sifakis) fueron distinguidos en el año 2007 con el prestigioso premio Turing otorgado por la Association for Computing Machinery (ACM). Este éxito se debe a la potencia de los algoritmos para verificar un enorme número de escenarios de manera automática y a su aplicabilidad tanto a sistemas de hardware como de software.

Herramientas como Spin, NuSMV o Uppaal han sido utilizadas para verificar los algoritmos de control de una barrera contra inundaciones en Holanda, el firmware de un switch telefónico de la empresa Lucent, protocolos de comunicación de audio en tiempo real, un controlador de caja de cambios, sistemas de control de vuelo e incluso ciertos componentes de las sondas espaciales Deep Space 1, Mars Rover, Deep Impact y Cassini.

El problema principal que sigue enfrentando la verificación de modelos es la explosión combinatorial de estados al incrementar el número de variables en un programa. Luego, uno de los objetivos de investigación es encontrar técnicas que permitan analizar modelos de cada vez mayor tamaño. Un avance reciente en este ámbito son las técnicas de verificación simbólica. Éstas utilizan estructuras especiales de representación y almacenamiento que permiten verificar modelos mucho más grandes que lo que es posible con técnicas tradicionales. Una buena introducción al tema se puede hallar en [5].

Es importante señalar sin embargo que ninguna de estas herramientas es la panacea que permite a su usuario producir código sin errores. En particular, todas ellas requieren todavía de un grado de asistencia manual a la hora de escribir las especificaciones del sistema y definir las propiedades a verificar. Sin embargo, la aplicación conjunta de éstas y otras técnicas permite reducir las probabilidades de errores en un programa significativamente, aumentando el grado de confianza en la implementación y acercándonos a un método de construcción de software más preciso y científico.

1.2. Objetivo del trabajo

PRISM[1] es un verificador de modelos simbólico desarrollado por un grupo de investigadores de la Universidad de Birmingham y actualmente mantenido por la Universidad de Oxford en el Reino Unido. El código de PRISM está abierto y publicado bajo la licencia GNU GPL.

La principal característica de PRISM es la posibilidad de describir modelos con comportamiento probabilístico. Por lo tanto, es posible realizar un análisis cuantitativo de propiedades, a diferencia de otras herramientas que sólo calculan si una propiedad se satisface o no en un modelo dado. Con este fin, PRISM provee un lenguaje especializado que permite describir distintos tipos de modelo, y también incorpora un lenguaje de especificación de propiedades que utiliza los operadores de lógicas temporales comunes en el área. Esto permite verificar propiedades como "La probabilidad de que un programa se ejecute por un año sin fallos es superior al 99.99 %" o "La cantidad esperada de mensajes circulando simultáneamente en una red es menor a 10".

La implementación actual de PRISM está diseñada para correr en sistemas con un sólo procesador. Por lo tanto, tanto el procesador como la cantidad de memoria limita el tamaño de modelos que podemos analizar. Históricamente, la evolución de la velocidad de estos componentes ha mostrado un comportamiento tipificado por la conocida "ley de Moore", la cual indica un crecimiento exponencial con una duplicación en el rendimiento cada 18 meses.

Sin embargo, existen límites físicos a los cuales nos estamos acercando cada vez más y que obligan a los ingenieros que fabrican computadoras a buscar alternativas para mantener el ritmo de crecimiento al cual los usuarios se encuentran acostumbrados. Una de las formas más popularizadas de lidiar con este problema es mediante la incorporación de múltiples procesadores que trabajan de forma paralela. Esta no es una solución mágica, ya que requiere adaptar el software actualmente disponible para que haga uso de los recursos adicionales. El diseño de programas concurrentes no es una tarea sencilla, ya que el multiprocesamiento introduce problemas adicionales como "deadlocks", conflictos en el acceso a recursos y condiciones de carrera que hacen que los algoritmos tradicionales dejen de funcionar.

Sin embargo, la proliferación de este tipo de sistemas no parece dejar otra alternativa y los beneficios de una paralelización exitosa son inmediatamente evidentes. La verificación de modelos es un área que puede sin lugar a dudas beneficiarse con la aplicación de estas tecnologías. Luego, en este trabajo nos proponemos modificar la implementación existente de PRISM para adaptarla a este nuevo tipo de arquitecturas. Nuestro objetivo final es construir algoritmos que permitan realizar las misma tareas de verificación que lleva a cabo actualmente PRISM en forma paralela, aprovechando el total de procesadores y memoria disponibles para reducir el tiempo de cómputo y facilitar la verificación de modelos de mayor tamaño.

Nuestro tratamiento se restringe a algoritmos para múltiples procesadores o procesadores con múltiples núcleos (cores). Sin embargo, las técnicas de programación paralela tienen hoy en día un horizonte mucho más amplio, cubriendo desde clusters de computadores conectadas cercanamente entre sí o procesadores con un propósito específico como las unidades de

[1] www.prismmodelchecker.org

procesamiento gráfico (GPUs) más modernas, hasta computadoras trabajando en formas conjunta a través de Internet para resolver un problema en particular (grid computing).

1.3. Lo que sigue

El resto de la tesis se estructura de la siguiente manera:

- En el capítulo 2 se introducen los formalismos utilizados en la verificación de modelos, así como las estructuras para representar estos modelos en la práctica.
- En el capítulo 3 se presentan primero los algoritmos clásicos de verificación y a continuación los algoritmos paralelos desarrollados para este trabajo.
- En el capítulo 4 se evalúan estos nuevos algoritmos con distintos casos de prueba, llevándose a cabo un análisis cuantitativo de los resultados.
- Finalmente, en el capítulo 5 se resumen los logros alcanzados, contrastando con otros trabajos en el área e indicando oportunidades para futura investigación.

Más detalles sobre el verificador PRISM están disponibles en [16] y [19]. Su página web en [20] también contiene una lista de publicaciones relacionadas.

Capítulo 2

Verificación de modelos probabilísticos

La verificación de modelos probabilísticos (Probabilistic Model Checking) es una extensión de la verificación de modelos clásica que nos permite establecer la validez de una propiedad dentro de un modelo dado. En este capítulo exponemos los componentes necesarios para realizar esta tarea (modelos y propiedades) así como los formalismos utilizados para su especificación (sistemas de transición y lógicas temporales respectivamente). Nos concentraremos en los formalismos soportados por la herramienta sobre la que se centra el trabajo.

2.1. Tipos de modelo

Los modelos representan el objeto de estudio en el proceso de verificación. Por lo tanto, el rango de aplicaciones posibles depende del grado de expresividad que ofrezcan estos modelos. Por otro lado, la necesidad de contar con métodos eficientes para la verificación hace necesario restringir el rango de modelos permitidos.

En la práctica, esto ha llevado a una formalización basada en *sistemas de transición de estados* (state transition systems). Un sistema de transición de estados está formado por un conjunto S de estados, asociados entre sí a través de una relación binaria $\rightarrow \subset S \times S$ que describe las transiciones permitidas de un estado a otro. Si $a, b \in S$ son estados del sistema, la notación $a \rightarrow b$ equivale a $(a, b) \in \rightarrow$.

Los sistemas de transición son un concepto estático, ya que describen una entidad inmutable en el tiempo. La otra cara de la moneda es el concepto dinámico de *traza de ejecución*. Éstas describen un posible comportamiento del sistema a lo largo del tiempo y se pueden formalizar como una secuencia infinita de estados $(s_i)_{i \in \mathcal{N}, s_i \in S}$ tales que $\forall_i s_i \rightarrow s_{i+1}$. Esto se entiende como un sistema que se encuentra inicialmente en el estado s_1, y posteriormente progresa a los estados s_2, s_3, etc. Las transiciones permitidas son exactamente las establecidas por la relación binaria $\rightarrow$.

Todo sistema de transición tiene un grafo dirigido subyacente donde los nodos representan los distintos estados y las aristas entre nodos las transiciones posibles. Esto hace posible visualizar gráficamente el sistema a través de un diagrama de nodos y flechas. Las trazas se interpretan como caminos infinitos en este grafo.

La verificación se basa en saber en que casos o con que probabilidad el funcionamiento del sistema satisface ciertas propiedades. Las propiedades se expresan en función de proposiciones básicas que satisfacen o no cada uno de los estados del modelo. Luego, también es necesario contar con una función $L : S \to \mathscr{P}(\text{Prop})$ que a cada estado le asigna un subconjunto del conjunto de todas las proposiciones posibles denotado por Prop. Las proposiciones en Prop se denominan *proposiciones atómicas*.

Hasta ahora no hemos discutido como aparecen las probabilidades en todo esto. Existen distintas alternativas para dotar de probabilidades a un sistema de transición. Cada una de ellas representa una interpretación posible del comportamiento de eventos reales dentro de un sistema que deseamos modelar. Analizamos tres posibilidades a continuación.

2.1.1. Cadenas de Markov de tiempo discreto

La asignación de probabilidades en una *cadena de Markov de tiempo discreto* o *DTMC* (discrete-time Markov chain) se basa en dos principios fundamentales. El primero es que el sistema evoluciona en intervalos discretos de tiempo. Por ejemplo, cada 1 segundo el sistema realiza una transición de un estado a otro, o permanece en el mismo estado. El segundo principio es que la evolución depende únicamente del estado actual, y no de los estados anteriores en una ejecución determinada.

Luego, a cada estado podemos asignarle una distribución de probabilidades discreta sobre el conjunto de transiciones que salen de ese estado. Esta distribución nos indica la probabilidad de tomar cada una de las transiciones en el siguiente paso de ejecución.

Formalmente, la relación $\to$ es ahora una función $P : S \times S \to \mathscr{R}_{\geq 0}$ que para cada estado $s \in S$ satisface $\sum_{s' \in S} P(s, s') = 1$. De esta forma, la probabilidad de transición de un estado a a un estado b está dada por $P(a, b)$. Notar que esto nos obliga a tener al menos una transición que salga de cada estado. Esto no es una restricción importante, ya que podemos modelar un estado terminal t con una única transición de t a t con probabilidad 1.

La asignación de probabilidades también se extiende a las ejecuciones, ya que nos permite interpretar la probabilidad de ejecución de una traza dada. El cálculo para trazas finitas es simple, ya que asumiendo que el estado inicial está fijado de antemano, podemos asignarle a una traza $(s_1, \ldots, s_n)$ una probabilidad dada por $P(s_1, s_2) * P(s_2, s_3) * \cdots * P(s_{n-1}, s_n)$. Esta definición de probabilidad puede ser generalizada a trazas infinitas, pero la definición requiere de herramientas matemáticas más avanzadas por lo que la omitiremos aquí. Para más detalle, consultar [15].

2.1.2. Procesos de decisión de Markov

Los *procesos de decisión de Markov* o *MDP* (Markov decision process) pueden considerarse como una extensión de las cadenas de Markov de tiempo discreto. La novedad reside en la incorporación de un elemento no determinístico al comportamiento del modelo. Esto nos permite modelar sistemas cuyas transiciones no son puramente probabilísticas o hacer cálculos de peor caso cuando algunas probabilidades no son conocidas.

Recordemos que en las cadenas de Markov de tiempo discreto tenemos, para cada estado s, una distribución de probabilidades sobre todos los estados x dada por $x \mapsto P(s,x)$. Esta distribución es la que cuantifica la probabilidad de transición en cada paso en base al estado actual. Pero no hay ningun motivo particular que nos obligue a tener una única distribución posible. Luego, los procesos de decisión de Markov generalizan esta propiedad asignando a cada estado un conjunto no vacío de distribuciones de probabilidad.

Formalmente, se define una función $D : S \to \{P : S \to \mathscr{R}_{\geq 0} \mid \sum_{x \in S} P(x) = 1\}$, que asigna a cada estado x un conjunto $D(x)$ de distribuciones de probabilidad. Pero ahora debemos decidir como interpretar las probabilidades de ejecución dentro de este modelo. Para eso nos valemos del concepto de *planificador* (scheduler). El planificador decide en cada paso cual de las distribuciones se utilizarán en base al estado actual y la traza parcial. Notar que si bien las distribuciones posibles dependen sólo del estado actual, el planificador puede valerse de toda la historia de ejecución incluyendo los estados anteriores para decidir que distribución escoger.

De esta forma, el comportamiento del sistema para un planificador dado es totalmente probabilístico. El no determinismo proviene del desconocimiento del planificador específico que actúa en cada ejecución. De hecho, esta información puede ser desconocida por o estar inaccesible incluso al mismo creador del modelo. Una consecuencia de esto es que las verificaciones se realizan sobre el total de planificadores posibles o algún subconjunto de ellos. De hecho, lo más común es restringir la verificación a planificadores con algún grado de *justicia* o *equidad* (fairness). Para más detalles sobre la definición de equidad utilizada en Prism, ver [7].

2.1.3. Cadenas de Markov de tiempo continuo

El tercer tipo de modelo que vamos a considerar son las *cadenas de Markov de tiempo continuo* o *CTMC* (continous-time Markov chain). La diferencia de estas con la cadenas de tiempo discreto se halla justamente en la forma de modelar el progreso del sistema a lo largo del tiempo. En vez de avanzar de a pasos discretos, en las cadenas de tiempo continuo las probabilidades de transición obedecen a una función de distribución de probabilidad continua.

Más específicamente, asumiendo que una ejecución se encuentra en un estado x, podemos modelar cada una de las transiciones posibles desde x como eventos independientes cuyas probabilidades de ocurrencia están dadas por distribuciones exponenciales con distintos parámetros. El primero de estos eventos en ocurrir activa la transición correspondiente y el proceso se repite desde el nuevo estado.

Formalmente, tenemos una función $R : S \times S \to \mathscr{R}_{\geq 0}$ tal que para cada estado x, $R(x,y)$ representa el parámetro de la distribución exponencial que caracteriza la probabilidad de ocurrencia del evento asociado a la transición $x \to y$. Como la distribución es exponencial y los eventos son independientes, podemos obtener luego la probabilidad de elegir una transición dada. Ésta corresponde a la probabilidad de que un evento ocurra antes que todos los otros con los cuales compite y es igual a $\frac{R(x,y)}{\sum_{z \in S} R(x,z)}$.

Parecería ser entonces que estamos en presencia de una distribución de probabilidades discreta. Si bien es cierto que las probabilidades de transición de un estado a otro son discretas

(pues el conjunto de estados en sí es discreto), esta última distribución no toma en cuenta los tiempos asociados a cada transición. Son estos tiempos los que nos permitendescribir un sistema que evoluciona con el mismo patrón que otro pero dos veces más despacio, o un sistema con transiciones más rápidas y otras más lentas.

2.2. Lógicas Temporales

Una vez establecidos los distintos tipos de modelo con los cuales trabajar, es necesario definir el lenguaje que utilizaremos para hablar de ellos y construir las propiedades que pretendamos verificar. Un buen punto de partida lo constituyen los operadores estándar de la lógica proposicional. Pero para poder hablar de la evolución del sistema necesitamos también operadores que puedan observar el aspecto temporal, para poder referirnos al estado en un momento preciso o relacionar el comportamiento en dos momentos dados. El resultado es lo que se conoce como *lógicas temporales*.

2.2.1. PCTL

La *lógica de árbol computacional probabilística* o *PCTL* (probabilistic computational tree logic) introducida en [12] es una extensión de la lógica CTL a modelos probabilísticos, y nos permite en particular trabajar con los dos modelos de tiempo discreto: DTMC y MDP. La gramática formal para esta lógica es la siguiente:

$$\phi ::= \textbf{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathscr{P}_{\bowtie p}[\psi]$$
$$\psi ::= \mathscr{X}\phi \mid \phi \mathscr{U}^{\leq k} \phi \mid \phi \mathscr{U} \phi$$

donde a denota una proposición atómica, k un número natural y $\bowtie$ uno de los operadores $<$, $\leq$, $>$ ó $\geq$.

Las fórmulas de esta lógica se dividen en dos tipos: *fórmulas de estado* (ϕ) y *fórmulas de camino* (ψ). Las fórmulas de estado hacen referencia al comportamiento del sistema en un instante de tiempo. De esta forma, podemos chequear la validez de una proposición atómica a para el estado actual, así como utilizar los conectores $\wedge$ y $\neg$ (y otros derivados como **false**, $\vee$ y $\rightarrow$) para construir sentencias más complejas.

Pero el verdadero trabajo es llevado a cabo por el operador $\mathscr{P}$ que nos permite hablar de la evolución del sistema a partir de un punto dado. Este operador tiene como argumentos una fórmula de camino junto con una cota de probabilidad, y permite establecer si la probabilidad de seguir un camino que satisfaga la fórmula dada está dentro de la cota provista.

Debemos explicar entonces como se interpretan las fórmulas de camino. Podemos pensar este tipo de fórmula como una forma de seleccionar un subconjunto de trazas de ejecución dentro del conjunto de trazas posibles. De esta forma, el operador $\mathscr{X}$ (*next*) sólo es válido para trazas en las cuales, después de haber realizado una transición del estado inicial, el estado que sigue satisface una fórmula ϕ dada.

Los operadores $\mathscr{U}$ (*until*) y $\mathscr{U}^{\leq k}$ (*until acotado*) por otro lado toman como parámetros dos fórmulas de estado, ϕ_1 y ϕ_2, y restringen las trazas válidas a aquellas en donde la primera condición ϕ_1 debe cumplirse hasta llegar a un estado que satisfaga la segunda condición ϕ_2. El segundo operador permite ajustar todavía más la restricción estableciendo un límite máximo de k transiciones para llegar a un estado donde la segunda condición sea verdadera.

2.2.2. CSL

Los modelos continuos tienen la característica adicional de contar con un período variable de transición entre un estado y otro. La lógica utilizada en estos casos se denomina *lógica estocástica continua* o *CSL* (continuous stochastic logic), y fue introducida en [2] y ampliada en [6]. Sus operadores son similares a los de PCTL pero sustituyen los valores discretos de tiempo por valores continuos. La gramática para esta lógica es:

$$\phi ::= \textbf{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathscr{P}_{\bowtie p}[\psi] \mid \mathscr{S}_{\bowtie p}[\psi]$$
$$\psi ::= \mathscr{X}\phi \mid \phi\,\mathscr{U}^{\leq t}\,\phi \mid \phi\,\mathscr{U}\,\phi$$

Los operadores compartidos funcionan igual que antes, salvo en el caso del until acotado que en vez de tomar como argumento una cota máxima para el número de transiciones, lleva en cambio un número real que representa el tiempo máximo para que se satisfaga la segunda fórmula de estado.

El único operador nuevo es $\mathscr{S}$, que nos permite especificar lo que se conoce como *propiedades de estado estable* (steady-state properties). Es decir, a diferencia de $\mathscr{P}$ que calcula probabilidades sobre ejecuciones posibles que satisfagan una fórmula de camino, el operador $\mathscr{S}$ calcula la probabilidad de que a largo plazo (cuando el tiempo tiende a infinito) el modelo se estabilice en un estado que satisfaga la propiedad dada. Esto es una herramienta muy útil cuando tenemos un modelo con una fase de inicialización de tiempo variable, pero lo que verdaderamente nos interesa es el comportamiento final del sistema.

2.3. Algoritmos de verificación

Habiendo visto los formalismos para modelos y propiedades, el próximo paso es explicar como se procede para verificar una fórmula lógica en un modelo dado. El resultado más sencillo de la verificación es un valor de verdad que indica si la fórmula se satisface o no. Sin embargo, veremos que para las formulas de camino es necesario modificar el proceso para permitir calcular en vez de un valor de verdad, la probabilidad de que la fórmula se satisfaga. También es posible extender una fórmula con parámetros que representan una probabilidad. En este caso, el resultado de la verificación es un rango de probabilidades que hacen que la fórmula sea válida.

Cualquiera sea la lógica o modelo utilizado, la verificación de una fórmula se hace siempre de forma recursiva. Esto significa que primero se analizan las subfórmulas que la componen, y en función de ellas se establece el valor de verdad de la fórmula final. Esto resulta directo

para los operadores lógicos booleanos como $\wedge$ y $\neg$. El caso base de la recursión lo constituyen las constantes booleanas y las proposiciones atómicas, cuya veracidad para un estado dado se puede obtener ya sea directamente o apelando a la función L del sistema de transición.

Nos concentramos entonces en los operadores propios a las lógicas temporales.

2.3.1. Verificación de DTMC

Para las cadenas de Markov de tiempo discreto, los operadores de la PCTL que nos interesan son $\mathscr{P}_{\triangleright\triangleleft p}$, $\mathscr{X}$ y las dos variantes de $\mathscr{U}$.

Para el operador $\mathscr{P}_{\triangleright\triangleleft p}[\psi]$, tenemos que determinar si la probabilidad de los caminos especificados por ψ satisface la cota dada por $\triangleright\triangleleft p$. Si bien nos alcanza con conocer si el valor de probabilidad está dentro de la cota o la excede, en la práctica esto es igual de difícil que obtener el valor exacto de probabilidad para los caminos que satisfacen ψ. Una vez obtenido este valor, es sencillo realizar una comparación para saber si satisface o no la cota.

Luego, debemos poder calcular la probabilidad de que partiendo de un estado inicial s, la ejecución satisfaga una fórmula de camino dada. Para el operador next en la fórmula $\mathscr{X}\phi$, conociendo el estado inicial y las probabilidades de transición, es posible calcular este valor sumando las probabilidades de transición de s a todos los estados que satisfacen ϕ.

Es útil expresar este tipo de cálculos en notación matricial. Luego, si asignamos a cada estado del modelo un número de 1 a n, y tenemos una matriz P donde el elemento $P_{i,j}$ indica la probabilidad de transición del estado i al estado j, entonces el valor buscado se encuentra en la posición asociada al estado s dentro del vector producido al multiplicar $P \cdot x$, donde x es un vector tal que $x_i = 1$ si el estado i satisface ϕ y 0 en caso contrario. La matriz P se conoce con el nombre de *matriz de transición*.

Para el operador $\mathscr{U}^{\leq k}$, el cálculo es similar pero debemos realizar varias multiplicaciones. La idea es hacer inducción sobre el parámetro k. Tomemos la fórmula $\phi_1 \mathscr{U}^{\leq k} \phi_2$. Si $k = 0$, entonces la fórmula es cierta para un estado s si y sólo si dicho estado satisface ϕ_2. En caso contrario, la probabilidad es 0. Para $k > 0$, hay tres casos:

1. Si el estado s satisface ϕ_2, entonces la probabilidad es 1.
2. Si el estado s satisface ϕ_1, entonces la probabilidad se obtiene al multiplicar la matriz de transición por el vector para el paso $k-1$ (i.e. el vector para la fórmula $\phi_1 \mathscr{U}^{\leq k-1} \phi_2$).
3. En cualquier otro caso, la probabilidad es 0.

Luego, cada paso inductivo se puede resumir en una multiplicación por una matriz de transición modificada, donde los valores en las filas correspondientes a estados que satisfacen ϕ_2 o no satisfacen ϕ_1 ni ϕ_2 se reemplazan por ceros salvo un uno en la diagonal. En total, son necesarias k multiplicaciones por esta matriz.

Finalmente, debemos analizar el operador $\mathscr{U}$. Este es ligeramente más complicado, ya que una ejecución podría requerir una cantidad de pasos arbitrariamente grande hasta llegar a un estado que satisfaga ϕ_2. Sin embargo, hay estados para los que podemos calcular la

probabilidad directamente: aquellos que satisfacen ϕ_2 (para los cuales la probabilidad es 1) y aquellos que no satisfacen ϕ_1 ni ϕ_2 (para los cuales la probabilidad es 0).

De hecho, es necesario ampliar estos dos conjuntos. Por ejemplo, si tenemos un estado que satisface ϕ_1 y no satisface ϕ_2, pero todas las transiciones posibles llevan a estados que satisfacen ϕ_2 entonces el estado en consideración también tendrá probabilidad 1. La definición más amplia es la siguiente:

- Si no existe una traza finita desde el estado inicial a un estado que satisfaga ϕ_2, pasando únicamente por estados intermedios que satisfagan ϕ_1, entonces la probabilidad del estado inicial es 0.
- Si no existe una traza finita desde el estado inicial a un estado con probabilidad 0 (i.e. que cumpla la condición del ítem anterior), sin pasar por ningún estado que satisfaga ϕ_2, entonces la probabilidad del estado inicial es 1.

Una vez que hemos analizado estos casos, debemos calcular las probabilidades para los estados restantes. Para esto, supongamos que ya tenemos el vector solución x con la probabilidad para cada uno de los estados. Luego, este vector debe satisfacer la ecuación $P' \cdot x = x$, donde P' es la matriz de transición modificada de la misma forma que para el operador $\mathscr{U}^{\leq k}$. Pero para poder resolver este sistema y asegurarnos que la solución es única, debemos utilizar los conjuntos de estados con probabilidad conocida que definimos inicialmente, reemplazando el vector x a la derecha de la ecuación por un vector x' con los valores apropiados para esos estados.

2.3.2. Verificación de MDP

Si bien la comprobación de procesos de decisión de Markov es similar a la utilizada para DTMCs, la principal dificultad consiste en la introducción de decisiones no deterministas que nos obligan a considerar más de un adversario al evaluar la probabilidad para una fórmula de camino. Más aún, la cantidad de adversarios a considerar es potencialmente infinita.

El truco para resolver este problema reside en la semántica del operador $\mathscr{P}$, para el cual sólo interesa si la probabilidad de un camino es mayor (o menor) que una cota dada. Luego, sólo es necesario considerar el peor caso de un adversario que produce la probabilidad más baja (o más alta) posible.

Veamos un ejemplo de como incorporar este concepto a la verificación del operador $\mathscr{X}$. Podemos modelar un MDP con una matriz de transiciones de forma similar a la utilizada para DTMCs, pero con la diferencia que una fila puede tener varias alternativas posibles correspondiente a las distintas elecciones no deterministas para cada estado. Luego, al multiplicar esta matriz por el vector de estados que satisfacen ϕ, si una fila tiene más de una alternativa posible debemos multiplicar por cada una de estas alternativas y de todos los resultados obtenidos tomar el máximo o mínimo dependiendo del tipo de cota que estemos deseando verificar.

La idea para aumentar la verificación del operador $\mathscr{U}^{\leq k}$ es similar. Empezamos con un vector correspondiente a $k = 0$ y realizamos inducción para llegar al k deseado. En cada paso

inductivo, la multiplicación se hace de la misma forma que para el operador $\mathscr{X}$, probando todas las alternativas posibles para cada fila y quedándonos con el resultado más grande o más chico en cada paso.

El tercer operador posible ($\mathscr{U}$) es también el más complicado, ya que presenta tres problemas respecto a los algoritmos que veníamos viendo hasta ahora. Hablaremos de como resolver dos de ellos, y mencionaremos el tercero sin dar demasiados detalles.

Recordemos que para DTMCs, el primer paso era identificar dos conjuntos de estados para los que las probabilidades eran exactamente 0 o 1. Luego debíamos resolver un sistema lineal de ecuaciones para obtener la probabilidad para los demás estados.

El primer inconveniente es que al tener varios adversarios posibles, la definición de los dos conjuntos iniciales varía. Por ejemplo, si estamos tratando de calcular la probabilidad máxima de obtener un camino que satisface $\phi_1 \mathscr{U} \phi_2$ partiendo de un estado dado, los estados con probabilidad 0 serán al igual que antes aquellos para los cuales no existe una traza finita desde el estado inicial a un estado que satisfaga ϕ_2, pasando únicamente por estados intermedios que satisfagan ϕ_1. Es decir, estados para los cuales la probabilidad es 0 bajo cualquier adversario.

Sin embargo, para que un estado tenga probabilidad máxima 1 alcanza con que exista algún adversario para el cual esto sea cierto. El conjunto de estados con esta propiedad se puede caracterizar por la siguiente condición:

- Todos los estados satisfacen ϕ_1 o ϕ_2.
- Para cada estado, existe una alternativa no determinística tal que:
  - Existe probabilidad no nula de llegar a un estado que satisface ϕ_2.
  - Todas las transiciones llegan a estados dentro del mismo conjunto.

Luego, la determinación de estos estados requiere un algoritmo de punto fijo que encuentre el conjunto más grande que satisfaga estas condiciones. Para probabilidades mínimas la dificultad es similar aunque las definiciones exactas varían.

Una vez que tenemos definidos estos dos conjuntos, el segundo problema surge al tener que calcular la probabilidad para los demás estados. A diferencia de las DTMC donde teníamos una multiplicación por una matriz y luego podíamos simplemente resolver un sistema de ecuaciones lineales, para los MDP tenemos también un máximo o mínimo. Esto nos obliga a resolver un problema de programación lineal, donde la matriz de transiciones acota las probabilidades para los estados. La función objetivo es la suma de la probabilidad de todos los estados, que debe ser maximizada o minimizada según corresponda.

Finalmente, al tener decisiones no determinísticas y trazas infinitas, entra en juego la noción de "fairness" que introducimos cuando definimos los procesos de decisión de Markov. Luego, si queremos restringir la verificación de una propiedad únicamente a los adversarios que satisfacen la condición de fairness, los algoritmos que estábamos utilizando dejan de funcionar. En los trabajos [7] y [4] se exponen algoritmos que resuelven este problema.

2.3.3. Verificación de CTMC

Para las cadenas de Markov de tiempo continuo, la lógica utilizada es CSL que sólo presenta ligeras variaciones sobre la lógica PCTL. Más aún, como comentamos anteriormente, toda CTMC tiene asociada una cadena discreta cuya función de probabilidad de transición P se puede calcular en términos de la función R como $P(x,y) = \frac{R(x,y)}{\sum_{z \in S} R(x,z)}$. Luego, para operadores como $\mathscr{X}$ y $\mathscr{U}$ que no dependen de la variable adicional de tiempo que introduce las CTMCs, la verificación se lleva a cabo sobre la DTMC asociada de forma idéntica que para las demás cadenas discretas.

Restan sólo dos operadores nuevos a considerar: $\mathscr{U}^{\leq t}$ y el operador de estado estable $\mathscr{S}_{\rhd\lhd p}$. Para analizar estos operadores, primero definimos la *matriz generadora* Q de una CTMC como:

$$Q_{x,y} = \begin{cases} R(x,y) & \text{si } x \neq y \\ -\sum_{z \neq x} R(x,z) & \text{si } x = y \end{cases}$$

Analicemos primero el operador $\mathscr{U}^{\leq t}$. Recordemos que un camino sólo satisface $\phi_1 \mathscr{U}^{\leq t} \phi_2$ si se alcanza un estado que satisface ϕ_2 antes del tiempo t, y todos los estados anteriores satisfacen ϕ_1. Luego, igual que antes tenemos dos conjuntos donde la probabilidad es trivialmente 1 o 0: los estados que satisfacen ϕ_2 y los que no satisfacen ϕ_1 ni ϕ_2 respectivamente.

Una primera idea aquí es modificar la CTMC eliminando las transiciones que salen de dichos estados. Luego, un camino que satisface la fórmula $\phi_1 \mathscr{U}^{\leq t} \phi_2$ en un instante anterior al tiempo t debe pasar por un estado que satisfaga ϕ_2, y como en la nueva cadena este estado no tiene transiciones salientes, seguirá estando en el mismo estado en el instante t.

Por otro lado, los caminos que no satisfacen la fórmula son aquellos que no llegan a un estado que satisfaga ϕ_2 antes del instante t, o los que pasan por un estados con probabilidad 0 y quedan allí trabados al no existir transiciones salientes de dichos estados. Es decir que los caminos válidos para la cadena original son exactamente aquellos que en la nueva cadena se hallan en el instante t en un estado con probabilidad 1.

En otras palabras, verificar el operador $\mathscr{U}^{\leq t}$ en la cadena original es equivalente a verificar el operador $\mathscr{U}^{=t}$ en la nueva cadena. Esto último es un ejemplo de *análisis de comportamiento transitorio* (transient analysis) en cadenas de Markov de tiempo continuo, donde la palabra transitorio hace referencia al análisis del comportamiento después de un período determinado de tiempo.

Para llevar a cabo este análisis, se hace uso de una técnica llamada *uniformización*. En base a la matriz Q se calcula una matriz uniformizada P dada por $P = I + Q/q$, donde q es cualquier entero positivo mayor o igual en valor absoluto a todos los elementos de la diagonal de Q. Luego, en base a P podemos obtener la matriz con las probabilidades de transición para una CTMC y un intervalo de tiempo t de la siguiente manera:

$$P_t = \sum_{i=0}^{\infty} \frac{e^{-qt} \cdot (qt)^i}{i!} \cdot P^i$$

En la práctica, la sumatoria se detiene una vez que se ha alcanzado la precisión necesaria. Finalmente, la probabilidades para cada estado se obtienen sumando las probabilidades

de transición de dicho estado hacia todos los estados que satisfacen ϕ_2 dadas por P_t. En símbolos, $P_t \cdot x$ donde $x_i = 1$ si el i-ésimo estado satisface ϕ_2 y 0 en caso contrario.

Consideremos ahora el operador de estado estable $\mathcal{S}_{\triangleright\triangleleft p}$. En este caso, queremos caracterizar el comportamiento del sistema a largo plazo. Es decir, nos interesa calcular la probabilidad de que el sistema evolucione hacia un estado determinado cuando la variable tiempo tiende a infinito. Se puede demostrar que para las CTMC que permite especificar PRISM, estas probabilidades no dependen del estado inicial, lo que simplifica la tarea.

Sea y el vector de probabilidades estacionarias. Es decir, y_i representa la probabilidad de hallarse en el estado i-ésimo a largo plazo. No es difícil ver que este vector debe satisfacer las dos condiciones $y \cdot Q = 0$ y $\sum_i y_i = 1$. Al mismo tiempo, esto sugiere una forma de calcular los valores de y:

1. Resolvemos el sistema de ecuaciones $y_0 \cdot Q = 0$.
2. Normalizamos la solución obtenida para que la suma de sus elementos sea 1, obteniendo $y = \frac{y_0}{|y_0|}$.

2.4. Diagramas de decisión binarios

Hasta ahora hemos visto como representar modelos y realizar su verificación utilizando matrices que almacenan las probabilidades de transición. Sin embargo, PRISM es un verificador *simbólico*. Esto significa que los modelos se representan no con matrices explícitas, sino con estructuras que permiten aprovechar la regularidad de los modelos que aparecen en la práctica para obtener una representación mucho más compacta que lo que es posible utilizando sólo matrices.

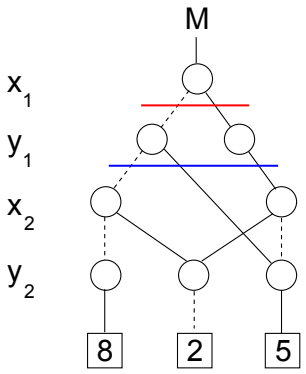
Estas estructuras se conocen por el nombre de *diagramas de decisión binarios* o *BDDs* (binary decision diagrams), y fueron introducidas en [18] y [1] y popularizadas por Bryant en [8]. Estos diagramas permiten codificar funciones booleanas. Es decir, funciones cuyos parámetros sólo admiten los valores 0 y 1, y que producen un 0 o 1 como resultado.

Uno podría representar estas funciones muy fácilmente utilizando arboles binarios completos, donde las hojas tienen un valor 0 o 1 y cada nivel del árbol corresponde a un parámetro diferente de la función. Luego, para evaluar la función para una asignación determinada de valores, comenzamos por la raíz y miramos el valor del primer argumento. Si este es 0, tomamos la rama izquierda. Si es 1, la derecha. Luego de descender tantos niveles como argumentos tenga la función, llegamos a una hoja del árbol que contiene el resultado.

Esta representación, si bien es aceptable y permite evaluaciones muy eficientes, no ahorra demasiado espacio. Por ejemplo, si la función evalua a 0 para cualquier combinación de argumentos, igual se necesitan todas las ramas del árbol por más que el resultado sea siempre el mismo. Es aquí donde aparecen los BDDs, que implementan dos heurísticas que permiten reducir el tamaño del árbol:

1. Para cada nivel del árbol, todos los nodos en ese nivel con exactamente la misma estructura en sus subárboles se unifican en un solo nodo.

Figura 2.1: Codificación de matrices con BDDs



Filas $(x_1 x_2)$	Columnas $(y_1 y_2)$ 0	1	2	3
0	0	8	0	5
1	2	0	0	5
2	0	0	0	5
3	0	0	2	0

2. Si las ramas izquierda y derecha de un nodo son iguales, el nodo se elimina (y los nodos que apuntaban a él pasan a apuntar a sus hijos).

Por ejemplo, para aplicar la primera heurística podemos partir del árbol original y procesar cada nivel de abajo hacia arriba. En cada nivel, unificamos los nodos que sean iguales. En el último nivel, dos hojas son iguales si tienen el mismo valor, por lo que hay a lo sumo dos nodos terminales posibles: uno para el 0 y otro para el 1. En los otros niveles, dos nodos serán iguales si sus hijos izquierdos y derechos coinciden (aprovechando el hecho de que ya hemos procesado los niveles inferiores unificando duplicados).

El resultado de este proceso y de aplicar luego la segunda heurística es un grafo dirigido acíclico, que se conoce como BDD. Sin embargo, no existe ninguna razón por la cual los únicos valores permitidos en las hojas del árbol deban ser 0 y 1. Si permitimos valores arbitrarios, entonces obtenemos los denominados *diagramas de decisión binarios con terminales múltiples* o *MTBDD* [10, 3]. Estos pueden emplearse para representar funciones que adoptan valores enteros o reales.

Finalmente, podemos utilizar MTBDDs para representar funciones con parámetros enteros dentro de un rango finito. La idea es considerar los dígitos en la representación binaria de cada argumento de la función como distintas variables booleanas. De esta forma, podemos codificar funciones sobre el conjunto de estados de un modelo o incluso matrices de transición, mediante funciones con dos parámetros enteros (la fila y la columna dentro de la matriz) y múltiples terminales con valores reales para las probabilidades.

Una observación importante aquí es que el tamaño de la estructura resultante dependerá fuertemente de la codificación elegida para los estados (i.e. la asignación de números a cada estado), por lo que es importante evaluar cuál es la codificación que permite lograr el mayor grado de compresión. En particular, PRISM codifica las matrices comenzando por los bits más significativos e intercalando bits correspondientes a los parámetros de fila y columna. Esto equivale a subdividir la matriz a la mitad horizontal o verticalmente cada vez que se desciende un nivel en el BDD.

La figura 2.1 muestra un ejemplo de tal codificación. El primer nivel del BDD M corresponde al primer bit del parámetro fila. El valor de 0 para este bit está asociado al hijo en la rama izquierda, que representa una submatriz A correspondiente a la mitad superior de la

matriz original. Análogamente, el segundo nivel corresponde al primer bit del parámetro columna y sus dos valores posibles corresponden a las dos mitades en que se subdivide A verticalmente. Para más detalles sobre esta codificación, consultar [19].

Capítulo 3

Paralelización

El objetivo de este capítulo es exponer las modificaciones necesarias para adaptar los algoritmos de verificación que utiliza PRISM a un sistema con múltiples procesadores. Para esto describimos primero las implementaciones originales utilizadas por la herramienta y a continuación presentamos las implementaciones paralelas desarrolladas para este trabajo. Destacamos también algunos detalles en la implementación que son cruciales para lograr una paralelización eficaz.

3.1. Motores de PRISM

Como vimos en el capítulo anterior, la verificación de modelos probabilísticos se reduce a operaciones con la matriz de transición y vectores que representan las probabilidades para los distintos estados del modelo. También introdujimos una estructura que sirve para codificar las probabilidades de transición de forma más compacta aprovechando la regularidad de los modelos.

PRISM presenta entonces tres motores distintos de verificación basados en estas ideas:

1. Un motor explícito
2. Un motor simbólico
3. Un motor híbrido

El primero de estos es el método clásico de verificación que utiliza tanto matrices como vectores *ralos* (sparse) para representar las probabilidades de transición. Esta representación permite efectuar los cómputos necesarios para la verificación de manera directa, a la vez que ahorra memoria almacenando únicamente los valores para las posiciones de la matriz donde la probabilidad es no nula. Como la matriz de transición tiene normalmente una gran cantidad de ceros (dado que el conjunto de transiciones para un estado es en general relativamente pequeño), esto permite reducir en gran medida la cantidad de memoria utilizada.

Si bien las matrices ralas evitan almacenar valores nulos, no toman ventaja de la regularidad de los modelos que ocurren en la práctica. Esta regularidad proviene tanto del lenguaje utilizado para la descripción de los modelos como de la naturaleza de los modelos mismos. Luego, la principal característica de los verificadores simbólicos es el uso de los BDDs para codificar tanto las matrices de transición de los modelos como los vectores de probabilidad. Estos sacrifican velocidad en los algoritmos para lograr una reducción de varios ordenes de magnitud en la cantidad de memoria utilizada, lo que permite analizar modelos más grandes de lo que es posible utilizando técnicas explícitas.

Si bien los métodos simbólicos han sido muy exitosos en el tratamiento de modelos no probabilísticos, una desventaja de su aplicación a modelos probabilísticos es la aparición de otros valores aparte de 0 y 1. Aunque este inconveniente puede solucionarse técnicamente mediante la utilización de MTBDDs, un examen del proceso de verificación muestra que el rendimiento de estos en comparación con los métodos explícitos deja algo que desear. El obstáculo más grande lo constituye la pérdida de regularidad en los vectores utilizados para almacenar las probabilidades de cada estado a lo largo de la verificación. Esto hace que los requerimientos de memoria para los MTBDDs sean iguales o incluso mayores que lo que se puede obtener utilizando un vector ralo directamente.

La principal contribución de PRISM como herramienta es la incorporación de un motor *híbrido* que combina las ventajas de los dos métodos, implementando algoritmos que permiten realizar la verificación utilizando MTBDDs para la representación de matrices y vectores ralos para las probabilidades de los estados. De esta forma, el rendimiento obtenido es muy similar al del motor explícito pero con un consumo de memoria notablemente inferior.

La innovación no está sólo en el hecho de mezclar ambas estructuras, sino en la utilización de algoritmos que no modifican en gran medida la matriz de transiciones a lo largo de la verificación, preservando la regularidad explotada por los MTBDDs para obtener una representación más eficiente. En particular, la solución de sistemas lineales se lleva cabo a través de métodos iterativos como el método de Jacobi en vez de los métodos directos tradicionales como la eliminación Gausianna o las descomposiciones LU o QR. Lo mismo ocurre con los problemas de programación lineal que aparecen al tratar MDPs, donde no se utilizan los métodos clásicos como el algoritmo símplex.

3.2. Verificación en PRISM

Vamos a describir ahora los algoritmos que se utilizan para la verificación. Para esto nos concentraremos en el motor híbrido, que es la principal novedad que ofrece PRISM. Podemos dividir a los operadores lógicos en dos categorías de acuerdo a la operación fundamental requerida para su evaluación:

1. Operadores que requieren multiplicaciones por matrices
   - $\mathscr{X}$ en DTMC, MDP y CTMC
   - $\mathscr{U}^{\leq k}$ en DTMC y MDP
   - $\mathscr{U}^{\leq t}$ en CTMC

- $\mathscr{U}$ en MDP

2. Operadores que requieren solución de un sistema lineal de ecuaciones
   - $\mathscr{U}$ en DTMC
   - $\mathscr{S}$ en CTMC

Las matrices utilizadas se obtienen generalmente a partir de la matriz de transiciones, realizando ligeras modificaciones para cada operador. Notar además que para MDPs, la solución puede requerir también el calculo de máximos o mínimos, pero la operación fundamental sigue siendo la multiplicación por matrices.

Para los operadores en la segunda categoría, se utilizan métodos iterativos que modifican el vector de soluciones para acercarlo cada vez más a una solución válida del sistema. Estos algoritmos tienen la ventaja de no modificar la matriz, por lo que podemos usar BDDs para representarla sin sufrir problemas. Las dos algoritmos que implementa PRISM son el método de Jacobi y el método de Gauss-Seidel.

El resto de esta sección describe estos métodos y su implementación. En particular, encontramos que el método de Jacobi puede reescribirse como una multiplicación por una matriz. Luego, esto nos permite obviar la descripción para los operadores de la primer categoría, puesto que su implementación es análoga a la utilizada para Jacobi.

3.2.1. Matrices ralas incrustadas

Antes de describir los métodos de resolución, debemos presentar una optimización más que afecta profundamente a las implementaciones usadas. Recordemos que los BDDs son similares a grafos dirigidos obtenidos a partir de árboles binarios eliminando niveles y consolidando nodos.

La principal ventaja de los BDDs es el grado de compresión que logran al almacenar la estructura de un modelo. Esto tiene como contrapartida una perdida de eficiencia. A diferencia de las matrices ralas donde los elementos de la matriz pueden ser extraídos en tiempo constante recorriendo una estructura en forma secuencial, en los BDDs estos valores están almacenados en los nodos terminales. Por lo tanto, para acceder a un elemento el tiempo es proporcional a la cantidad de niveles del BDD.

En la práctica, la utilización de un BDD puro no siempre es necesario. Si contamos con suficiente memoria, es posible lograr un compromiso realizando una conversión parcial de matriz a BDD. Es decir, en vez de un BDD puro donde los nodos terminales representan elementos dentro de la matriz, es posible utilizar un BDD mixto en el que los niveles más altos son similares a los de un BDD común pero al llegar a un nivel dado, todos los nodos se reemplazan por matrices ralas que contienen la información de un fragmento del BDD original. De esta forma, una vez que llegamos a un nodo que contiene una matriz, podemos extraer los elementos de forma directa sin necesidad de seguir recorriendo el BDD para cada valor.

Llamaremos *BDDs con matrices incrustadas* a estos BDDs mixtos con matrices como terminales. La cantidad de memoria utilizada se puede ajustar eligiendo un nivel más alto o

más bajo para incorporar las matrices. Mientras más alto sea el nivel, más eficiente será la estructura pero requerirá a su vez mayor cantidad de memoria.

Finalmente, para algunos algoritmos es necesario deshacernos del todo de los BDDs. En estos casos, los niveles superiores del BDD mixto se transforman en otra matriz llamada *matriz de bloques*. Esta matriz es una subdivisión de la matriz original que contiene en cada posición un bloque correspondiente a la matriz incrustada. Es decir que tenemos dos niveles de matrices: la matriz de bloques y las matrices incrustadas.

Uno podría cuestionarse si esto no es lo mismo que volver al modelo explícito basado exclusivamente en el uso de matrices ralas. La diferencia reside en que al utilizar el BDD para obtener la matriz de bloques, el resultado es una estructura explícita que explota la regularidad del modelo original como lo hace el BDD del cual fue derivada. Luego, puede considerarse a esto como una aproximación simbólica donde los BDDs están explícitos. La misma idea ha sido aplicada en trabajos como [21].

3.2.2. Jacobi

Supongamos que tenemos un sistema $Ax = b$ donde A es una matriz de coeficientes y b el vector de resultados. Si extraemos de este sistema la ecuación para la primera fila obtenemos $a_{1,1} * x_1 + \ldots + a_{1,n} * x_n = b_1$, donde $a_{i,j}$ es el elemento en la fila i y columna j de la matriz A y x_i (b_i) es el i-ésimo elemento del vector x (b respectivamente). Ahora, podemos despejar el valor de x_1 en esta ecuación obteniendo:

$$x_1 = \frac{b_1 - (a_{1,2} * x_2 + \ldots + a_{1,n} * x_n)}{a_{1,1}}$$

Esta última ecuación puede escribirse para cada fila de la matriz y sólo se satisfará cuando x sea una solución del sistema. Sin embargo, si tenemos un vector x que no es una solución, podemos utilizar la ecuación para calcular un nuevo valor de x_1. Si bien esto no nos asegura que el nuevo valor satisfaga las demás ecuaciones, la idea es que realizando este proceso múltiples veces para todas las filas podemos ir acercándonos a la solución deseada. Éste es el razonamiento detrás del *método de Jacobi*.

Un pseudocódigo que implementa este algoritmo utilizando una representación explícita para matrices se puede ver en el listado 1.

Un detalle a notar es la condición de detención utilizada. El algoritmo se detiene cuando la diferencia entre los vectores de dos iteraciones consecutivas es suficientemente pequeña. En este caso utilizamos la máxima diferencia entre elementos de los dos vectores, como se ve en la línea 15 (en otras palabras, la norma infinito de $x - x'$). También es común utilizar la diferencia relativa, reemplazando la linea 15 por:

max-error $\leftarrow$ máx(max-error, $|x_i - x'_i|/|x_i|$)

En la práctica, existe la posibilidad de que el método no converja por lo que también debemos limitar el número de iteraciones. Si bien existen otro métodos que garantizan convergencia (como el método de las potencias), no son muy utilizados ya que por lo general el método de Jacobi converge y su rendimiento es superior.

Listado 1 Jacobi(A,b)

Entrada: A - una matriz $n \times n$, b - un vector $n \times 1$
Salida: x - un vector $n \times 1$ tal que $Ax = b$

```
1:  for i = 1 to n do
2:      x_i ← 1
3:  end for
4:  max-error ← ∞
5:  while max-error > ε do
6:      max-error ← 0
7:      for i ← 1 to n do
8:          x'_i ← b_i
9:          for j ← 1 to n do
10:             if i ≠ j then
11:                 x'_i ← x'_i − a_{i,j} * x_j
12:             end if
13:         end for
14:         x'_i ← x'_i / a_{i,i}
15:         max-error ← máx(max-error, |x_i − x'_i|)
16:     end for
17:     x ← x'
18: end while
19: return x
```

El problema de la implementación anterior es que si la matriz A viene representada por un BDD, resulta extremadamente costoso extraer los elementos en el orden requerido (fila por fila). Sin embargo, es posible reestructurar los cálculos para que el algoritmo funcione. Notemos para empezar que podemos describir este proceso utilizando una notación multiplicativa de la siguiente manera: primero definimos una matriz D cuyos únicos elementos no nulos son los de la diagonal de A:

$$D_{i,j} = \begin{cases} A_{i,j} & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

Luego, a partir de un vector inicial x podemos calcular una mejor aproximación a la solución haciendo:

$$x' = (b - (A - D) * x) * D^{-1}$$

Esta ecuación equivale a una iteración del método de Jacobi. El paso que demanda la mayor cantidad de tiempo aquí es la multiplicación por $A - D$, ya que debemos multiplicar cada elemento de esta matriz por un elemento de x y sumar los resultados en cada fila. Pero como la suma es conmutativa, el orden en que recorremos la matriz no importa realmente. Esto nos sugiere una forma de implementar el algoritmo para BDDs: recorremos todos los caminos del árbol binario implícito en el BDD, y cada vez que llegamos a un terminal tenemos el valor de un elemento de la matriz que multiplicamos por la posición correspondiente del

vector x. En el motor híbrido el vector x se representa de manera explicita, por lo que ofrece acceso aleatorio a cualquiera de sus elementos.

El resultado es el algoritmo descripto en los listados 2 y 3. La función *JacobiBDD* recibe como parámetros un puntero *root* a la raíz del BDD que codifica la matriz $A-D$, un vector d con los elementos de la diagonal de D y el vector b de términos independientes. Esta función se encarga al igual que antes de la inicialización de los vectores x y x', y de iterar hasta que la solución converja.

Por otro lado, para la multiplicación en sí se utiliza una rutina recursiva auxiliar *MultiplyBDD* que multiplica un fragmento del BDD por el vector x y acumula el resultado en x'. Los parámetros i y j almacenan la posición de la submatriz representada por el nodo p dentro de la matriz original. Luego, cuando llegamos a un nodo terminal del BDD, el valor del nodo corresponde a un elemento de la matriz cuya posición está dada por i y j.

Listado 2 JacobiBDD($root,d,b$)

Entrada: *root* - un puntero a la raíz del BDD, d, b - dos vectores $n \times 1$
Salida: x - un vector $n \times 1$ tal que $Ax = b$

```
1:  for i = 1 to n do
2:      x_i ← 1
3:  end for
4:  max-error ← ∞
5:  while max-error > ε do
6:      max-error ← 0
7:      for i ← 1 to n do
8:          x'_i ← b_i
9:      end for
10:     MultiplyBDD(root,x,x')
11:     for i ← 1 to n do
12:         x'_i ← x'_i / d_i
13:         max-error ← máx(max-error, |x_i − x'_i|)
14:     end for
15:     x ← x'
16: end while
17: return x
```

Listado 3 MultiplyBDD(p,i,j,x,x')

Entrada: p - un puntero a un nodo del BDD, i, j - dos números enteros - x, x' - dos vectores $n \times 1$

```
1: if p es un nodo terminal then
2:     x'_i ← x'_i − val(p) * x_j
3: else
4:     MultiplyBDD(p.left, i, j, x, x')
5:     MultiplyBDD(p.right, i+offset_i(p), j+offset_j(p), x, x')
6: end if
```

Hay dos detalles que debemos destacar aquí: en vez de almacenar la posición absoluta para cada nodo, utilizamos *desplazamientos* (offsets) que contienen la posición relativa

respecto al nodo padre. Junto con la codificación de estados mencionada en la sección 2.4, esto nos permite calcular la posición de manera incremental a medida que recorremos el BDD, pero reduce a la mitad la cantidad de información almacenada ya que para el hijo izquierdo el desplazamiento siempre es igual a cero.

El otro detalle es como manejar BDDs con matrices incrustadas. En este caso, si llegamos a un nodo que contiene una matriz incrustada, en vez de seguir descendiendo por el BDD debemos recorrer todos los valores de esta matriz en forma secuencial, procesando cada elemento de la misma forma que lo hacemos con los nodos terminales del BDD.

3.2.3. Gauss-Seidel

El algoritmo de Jacobi que presentamos necesita dos vectores que almacenan el resultado de una iteración y de la siguiente. Como en el modelo híbrido las matrices se representan con BDDs, son los vectores los que terminan ocupando la mayor cantidad de memoria. Luego, si logramos reducir eliminar el vector adicional necesario para cada iteración, podemos reducir casi a la mitad el consumo de memoria.

Una forma de lograrlo es utilizar el *método de Gauss-Seidel*. Este método funciona de forma similar al de Jacobi, recorriendo la matriz fila por fila y actualizando los valores de los x_i. Sin embargo, a diferencia de Jacobi donde los valores calculados se almacenan en un nuevo vector x', en Gauss-Seidel los valores se almacenan en el mismo vector x. Es decir que al calcular el nuevo valor del elemento i-ésimo del vector x, los valores utilizado para los x_j con $j < i$ son los de la iteración actual, en vez de la anterior.

El pseudocódigo presentado en el listado 4 es muy similar al de Jacobi y de hecho es más sencillo porque no necesita un vector adicional.

El ahorro de memoria no es la única ventaja importante de este método. Al utilizar en cada paso del algoritmo los valores más recientemente calculados del vector x, la velocidad de convergencia del método también mejora. Por lo tanto, se requiere una menor cantidad de iteraciones para aproximarse a una solución lo que reduce también el tiempo total de ejecución.

Sin embargo, el algoritmo también tiene algunas desventajas. Si bien el orden en que recorremos las filas de la matriz no es importante (siempre y cuando recorramos cada fila una vez por iteración), si es necesario procesar cada fila por completo antes de movernos a la siguiente. Es decir que a diferencia de Jacobi, donde podíamos ir alternando entre dos o más filas, en este método tenemos que proceder de manera secuencial una fila por vez. Esto se debe a que al almacenar los resultados en el mismo vector x, si estamos procesando dos elementos x_i y x_j simultáneamente y para calcular x_i necesitamos el valor de x_j, entonces la fila i no podrá continuar hasta que j termine. Peor aún, si x_j necesita el valor de x_i (lo que sucede a menudo) entonces tenemos una dependencia circular y ninguna de las dos filas podrá continuar, ya que no podemos acceder ni a los valores originales (que se pierden al iniciar el nuevo cálculo) ni a los valores actualizados (que sólo se obtienen al terminar de procesar la fila).

El no poder reorganizar los cálculos es un problema particularmente grave cuando utilizamos BDDs, ya que como hemos visto estos no permiten fácil acceso a los elementos de una

Listado 4 Gauss-Seidel(A,b)

Entrada: A - una matriz $n \times n$, b - un vector $n \times 1$
Salida: x - un vector $n \times 1$ tal que $Ax = b$

```
1:  for i = 1 to n do
2:      x_i ← 1
3:  end for
4:  max-error ← ∞
5:  while max-error > ε do
6:      max-error ← 0
7:      for i ← 1 to n do
8:          x̃_i ← x_i
9:          x_i ← b_i
10:         for j ← 1 to n do
11:             if i ≠ j then
12:                 x_i ← x_i − a_{i,j} * x_j
13:             end if
14:         end for
15:         max-error ← máx(max-error, |x_i − x̃_i|)
16:     end for
17: end while
18: return x
```

fila de manera secuencial. Existen dos estrategias para enfrentar este problema. La primera se conoce por el nombre de *Gauss-Seidel por bloques* (Blocked Gauss-Seidel) y esta basada en las matrices de bloques presentadas en [17].

Si podemos transformar el BDD en una matriz de bloques, entonces es posible procesar la matriz original un bloque de filas por vez. Luego, para cada fila de la matriz de bloques (que representa un conjunto de filas de la matriz original) tomamos cada bloque dentro de esa fila y lo utilizamos para calcular los nuevos valores de los x_i. De hecho, las únicas dependencias entre dos filas de un mismo bloque están en la diagonal de la matriz de bloques, por lo que sólo estas posiciones deben ser procesadas de manera secuencial y deben estar codificadas utilizando matrices ralas. Los demás bloques de una misma fila pueden ser procesados en orden arbitrario, dejando el bloque diagonal para el final. El listado 5 muestra el pseudo-código para este algoritmo.

Otra estrategia posible, presentada en [19] bajo el nombre de *Pseudo Gauss-Seidel*, se basa en la relajación de las condiciones requeridas por el método de Gauss-Seidel. Recordemos que en el método original, para calcular el valor de x_i se utilizan para $j < i$ los valores de x_j calculados en la iteración actual, y para $j > i$ los de la iteración anterior. Asumiendo de vuelta que tenemos disponible la matriz de bloques correspondiente al BDD y calculamos de a bloques de filas como en Gauss-Seidel por bloques, la relajación consiste en utilizar los valores nuevos de los x_j sólo cuando j pertenece a un bloque anterior de filas que ya ha sido procesado. Para las demás filas, incluyendo las del bloque actual, utilizamos los valores de la iteración anterior. Esto requiere espacio adicional proporcional a la cantidad de filas de un bloque para almacenar los valores original del bloque con el que estamos trabajando, de forma similar al método de Jacobi.

Listado 5 Gauss-Seidel-Bloques(A,b)

Entrada: A - una matriz de $n \times n$ bloques de dimensión $m \times m$, b - un vector $(n * m) \times 1$
Salida: x - un vector de $(n * m) \times 1$ que cumple $Ax = b$

1: **for** $i \leftarrow 1$ **to** n **do**
2: **for** $i' \leftarrow (i-1) * m + 1$ **to** $i * m$ **do**
3: $x_{i'} \leftarrow b_{i'}$
4: **end for**
5: **for** $j \leftarrow 1$ **to** n **do**
6: **if** $j = i$ **then**
7: **continue**
8: **end if**
9: $B = A_{i,j}$ {B es un bloque i.e. una matriz de $m \times m$}
10: **for** $i' \leftarrow 1$ **to** m **do**
11: **for** $j' \leftarrow 1$ **to** m **do**
12: $x_{(i-1)*m+i'} \leftarrow x_{(i-1)*m+i'} - B_{i',j'} * x_{(j-1)*m+j'}$
13: **end for**
14: **end for**
15: **end for**
16: $B \leftarrow A_{i,i}$
17: **for** $i' \leftarrow 1$ **to** m **do**
18: **for** $j' \leftarrow 1$ **to** m **do**
19: **if** $j' = i'$ **then**
20: **continue**
21: **end if**
22: $x_{(i-1)*m+i'} \leftarrow x_{(i-1)*m+i'} - B_{i',j'} * x_{(j-1)*m+j'}$
23: **end for**
24: $x_{(i-1)*m+i'} \leftarrow x_{(i-1)*m+i'} / B_{i',i'}$
25: **end for**
26: **end for**
27: **return** x

La ventaja de este método es que a diferencia de Gauss-Seidel por bloques, los bloques dentro de una misma fila de la matriz de bloques pueden ser procesados de manera no secuencial. Luego, la matriz de bloques no necesita almacenar en cada posición una matriz rala incrustada, sino que puede tener simplemente un puntero a un nodo del BDD. De esta forma, ahorramos espacio requerido para almacenar la matriz rala a costa de utilizar espacio adicional para almacenar una fracción del vector x.

Sin embargo, es necesario advertir que al realizar esta modificación ya no estamos trabajando con el método de Gauss-Seidel original, por lo que la velocidad de convergencia no es la misma para ambos. De hecho, se lo puede pensar como un nivel intermedio entre Jacobi y Gauss-Seidel, dependiendo del tamaño de la matriz de bloques. Si la matriz de bloques es de 1×1 tenemos el método de Jacobi, mientras que si es del mismo tamaño que la matriz original tenemos Gauss-Seidel.

3.3. Algoritmos paralelos

Introducimos ahora de forma gradual las modificaciones efectuadas a los métodos de la sección anterior, para permitir su ejecución en un entorno paralelo con múltiples procesadores. Las preocupaciones principales al desarrollar cualquier programa que se ejecutará de forma paralela están asociadas estrechamente a los dos requisitos de un buen programa: corrección (eficacia) y rendimiento (eficiencia).

En programas concurrentes, el aspecto de corrección se relaciona con el uso apropiado de mecanismos de comunicación y sincronización entre procesos o hilos que garanticen la ejecución en un orden válido de los distintos pasos de un algoritmo, evitando problema clásicos en el área de concurrencia como “deadlocks”, condiciones de carrera u otros errores lógicos que pueden afectar el comportamiento de un programa.

El segundo aspecto de eficiencia juega un papel casi tan importante, ya que afecta directamente la *escalabilidad* de un sistema distribuido: la capacidad de resolver problemas más grandes y más rápidamente, en forma proporcional a la cantidad de recursos adicionales (en nuestro caso, procesadores y memoria) disponibles. En este caso, algunos obstáculos a superar son: bajo nivel de paralelismo por dependencias entre tareas, desbalance de trabajos entre distintas unidades de cómputo, ineficiencia en los métodos de sincronización y mal aprovechamiento de la jerarquía de memoria.

3.3.1. Jacobi para BDDs

Nos interesa ahora analizar como implementar de forma paralela los métodos expuestos. El método de Jacobi es el principal candidato, ya que su estructura libre de dependencias permite explotar la concurrencia de manera simple dividiendo las operaciones necesarias para cada iteración entre varios procesadores.

El problema principal reside entonces en como dividir y asignar las tareas a los distintas unidades de ejecución. Una estrategia posible es utilizar la matriz de bloques, distribuyendo los bloques entre distintos procesadores. La desventaja de esta idea es que si la matriz de

bloques es muy pequeña y los bloques son desparejos entre sí (es decir, algunos tienen mayor cantidad de elementos no nulos que otros), es difícil distribuir de forma que cada procesador realice aproximadamente la misma cantidad de trabajo. Esto se conoce como el problema de *balanceo de cargas* (load balancing).

Por otro lado, si los bloques son pequeños, la cantidad de memoria requerida para la matriz de bloques se eleva, lo que sumado al requerimiento del método de Jacobi de utilizar dos vectores para cada iteración hace que el consumo total de memoria se torne inaceptable.

La observación importante aquí es que si bien queremos dividir el BDD en partes para poder procesarlo, no requerimos una regularidad como la que ofrecen las matrices de bloques. Luego, cualquier división que otorgue a cada procesador un pedazo del BDD de manera relativamente equitativa y minimizando la cantidad de partes permite resolver nuestro problema.

El algoritmo que utilizamos entonces es una división recursiva. Para esto, primero definimos un modelo de costos: si queremos procesar todos los hijos de un nodo del BDD, una buena aproximación al tiempo total de cómputo es la cantidad de llamadas recursivas a realizar (que no es necesariamente lo mismo que la cantidad de nodos distintos a visitar debido a las técnicas de compresión que usan los BDD). Luego, podemos calcular y almacenar este valor para cada nodo con una pasada preliminar por el BDD. Con esta información, el listado 6 presenta el pseudo-código del algoritmo de división para n procesadores.

El método toma como parámetros un nodo del BDD, y una lista de porcentajes para cada procesador. Estos porcentajes indican qué fracción del BDD representado por el nodo dado le corresponde a cada procesador. Si sólo existe un procesador con porcentaje no nulo (e igual a 100 %), la función asigna todo el BDD a dicho procesador. Si el BDD tienen un único nodo, la función se lo asigna a un procesador con porcentaje no nulo. De lo contrario, la función mira los dos hijos del nodo dado y divide los porcentajes en función de los costos para cada hijo. Por ejemplo, si queremos dividir el BDD entre los procesadores 1, 2 y 3, dándole el 33 % a cada uno, y el modelo de costos nos dice que los dos hijos requieren el mismo poder de cómputo, entonces dividimos uno de los hijos entre los procesadores 1 y 2 con proporción 66 %/33 % y el otro entre los procesadores 2 y 3 con proporción 33 %/66 %. Por otro lado, si el primer hijo requiere el doble de cómputo que el segundo, entonces dividimos el primero hijo 50 %/50 % entre los procesadores 1 y 2 y damos el 100 % del segundo al procesador 3. El proceso se inicia llamando al método con el nodo raíz del BDD y un mismo porcentaje para cada procesador igual a $100\,\%/n$.

En la práctica, sólo es necesario realizar este proceso una vez al comenzar las iteraciones. Otro detalle de implementación es que debido a la ordenación de las variables en los BDDs de matrices, resulta más conveniente bajar de a dos niveles por vez en el BDD, con lo cual en vez de dos hijos para cada nodo tenemos cuatro.

Por último, notar que en cada llamada recursiva la cantidad de procesadores con porcentajes no nulos ya sea disminuye o se mantiene igual. En este último caso, alguna de las dos llamadas termina inmediatamente ya que sólo asigna porcentaje a uno de los procesadores. Luego, la cantidad total de llamadas a `assign` es a lo sumo $N * h$, donde h es la cantidad de niveles del BDD. Es decir que en promedio, cada procesador recibe h pedazos del BDD original. Esto implica que la memoria utilizada por cada procesador para almacenar su lista

Listado 6 divide(*root*, *split*)

Entrada: *root* - nodo raiz del BDD que queremos dividir, *split* - arreglo de n porcentajes

```
1: for i ← 1 to n do
2:    if split[i] = 100 or (split[i] > 0 and root es un nodo terminal) then
3:       assign(root, i) {Asignar todo el BDD al procesador i}
4:       return
5:    end if
6: end for
7: threshold ← cost(root.left) / (cost(root.left) + cost(root.right))
8: accumulated ← 0
9: splitleft ← [0,...,0]
10: splitright ← [0,...,0]
11: {Calcular nuevos porcentajes para cada hijo}
12: for i ← 1 to n do
13:    if acummulated < threshold and accumulated + split[i] ≥ threshold then
14:       splitleft[i] ← threshold - accumulated
15:       splitright[i] ← accumulated + split[i] - threshold
16:    else if accumulated < threshold then
17:       splitleft[i] ← split[i]
18:    else if accumulated ≥ threshold then
19:       splitright[i] ← split[i]
20:    end if
21:    accumulated ← accumulated + split[i]
22: end for
23: {Normalizar listas}
24: for i ← 1 to n do
25:    splitleft[i] ← splitleft[i] / threshold
26:    splitright[i] ← splitright[i] / (1 - threshold)
27: end for
28: divide(root.left, splitleft)
29: divide(root.right, splitright)
```

de fragmentos es muy poca en comparación con lo que consume el BDD o la matriz de bloques.

3.3.2. Jacobi con matrices incrustadas

Si bien el método de subdivisión anterior funciona correctamente para BDDs puros, la introducción de matrices incrustadas presenta algunos problemas que afectan al rendimiento. En particular, el modelo de costos se ve trastornado ya que tenemos ahora dos tipos muy distintos de operaciones dentro del mismo algoritmo: las correspondientes al recorrido del BDD y las realizadas al procesar una submatriz. Esto hace que predecir las cargas que incurren distintos fragmentos del BDD sea más complicado, ya que esto depende de detalles como el nivel en que se incorporan las matrices, la estructura que se utiliza para su almacenamiento en memoria e incluso detalles de la arquitectura como velocidad de acceso a memoria y tamaño de la caché.

Una forma de solucionar estos problemas consiste en utilizar una estrategia de asignación de tareas dinámica. Es decir, en vez de otorgarle a cada procesador un conjunto fijo en cada iteración, lo que hacemos es darle tareas a medida que lo necesite. Una forma de implementar esto es utilizar una cola o "pool" global de tareas pendientes. Luego, cada procesador realiza un ciclo en el que retira una tarea de esta colección, la procesa, retira otra tarea, la procesa y así sucesivamente.

En nuestro caso, la tarea a procesar será la multiplicación de una porción del BDD. Por lo tanto, al igual que antes debemos dividir el BDD en pedazos que formarán parte de la cola de tareas. Como utilizamos asignación dinámica, no nos interesa tanto que los distintos pedazos sean parejos entre sí, ya que si un procesador termina antes que otro, entonces puede ir a la cola y obtener un nuevo pedazo que procesar. Pero esto también significa que necesitamos suficientes tareas para que los procesadores siempre estén ocupados. Una posibilidad es utilizar todos los nodos en un mismo nivel del BDD, repitiendo cada nodo por cada ocurrencia como subbloque en la matriz original. Es decir, una tarea corresponde a un bloque de la matriz de bloques, pero el bloque no necesariamente debe estar representado por una matriz incrustada sino que como en Pseudo Gauss-Seidel también es posible que sea un BDD.

Otro problema importante a considerar es la posibilidad de colisiones entre procesadores que trabajan sobre elementos de la misma fila de la matriz. Si bien este problema también afecta al algoritmo de la sección anterior, el efecto es mucho más notable al tener matrices incrustadas ya que se consume menos tiempo recorriendo el BDD y más tiempo actualizando el vector x'. Notar que el BDD y el vector x no están afectados, ya que los accesos son de sólo lectura.

Existen tres alternativas para manejar esta situación:

1. Utilizar una copia del vector x' por cada procesador
2. Asegurarse que las modificaciones se realicen de forma atómica
3. Evitar conflictos asignando tareas de manera inteligente

La primera de estas alternativas funciona eliminando la escritura compartida pero requiere un paso de consolidación al final de cada iteración, que acumula los vectores de cada procesador en un vector global de resultado. La segunda soluciona el problema asegurándose que si dos procesadores modifican la misma posición de memoria, ninguna de estas modificaciones se pierda. La tercera opción se basa en no asignar a distintos procesadores bloques de memoria que puedan causar conflictos (es decir, dos bloques con filas en común).

Tanto la segunda como la tercera alternativa pueden implementarse utilizando mecanismo de exclusión mutua, aunque para la segunda también pueden utilizarse operaciones atómicas. Sin embargo, la diferencia reside en donde se aplican estos mecanismos. En el segundo caso, debemos utilizar "mutexes" o semáforos para proteger el acceso a posiciones de la matriz. Luego, si una posición está siendo modificada y otro procesador quiere acceder a ella, deberá esperar que la modificación termine. En la práctica estos mecanismos incurren en un costo que no permite garantizar exclusión a un nivel muy alto de granularidad, lo que conlleva una mayor cantidad de conflictos y de tiempo de procesador desperdiciado.

Por otro lado, la tercera alternativa sólo exige exclusión mutua a la hora de asignar una tarea a un procesador, para asegurarse que esta no ocasione conflictos con alguna de las otras que ya están siendo procesadas. Si bien esto reduce la oferta disponible, también asegura que una vez que un procesador recibe una tarea, puede trabajar independientemente de los demás sin incurrir en ningún tipo de conflictos y sin necesidad de protección. Dado que el paso de asignación requiere exclusión mutua de por si para evitar asignar a dos procesadores la misma tarea, la sobrecarga adicional es mínima.

La implementación realizada para este trabajo utiliza una cola circular de tareas con este fin. Cada vez que un procesador solicita un fragmento de matriz para procesar, se extrae el primer elemento de la cola. Si este contiene alguna fila que ya está siendo procesada, se inserta al final de la cola y se extrae el próximo elemento hasta conseguir uno que no ocasione conflictos. Únicamente cuando ninguno de los elementos satisface esta condición (lo que en general sólo sucede cuando quedan pocos elementos en la cola), el procesador deberá detenerse a esperar que se libere alguna fila.

Si las filas se agrupan de a bloques y cada tarea afecta a lo sumo a un bloque, otra implementación posible es utilizar una cola por fila más una cola adicional de filas disponibles. Esto permite asignar una tarea a un procesador en tiempo constante, buscando primero en la cola de filas para encontrar un bloque de filas libre, y extrayendo luego de la cola correspondiente a esas filas una tarea a realizar.

3.3.3. Gauss-Seidel

Vimos que el método de Gauss-Seidel es más eficiente ya que utiliza menos memoria para los vectores y requiere en promedio una menor cantidad de iteraciones. Luego, si nuestra intención es reducir el tiempo requerido para la verificación, este algoritmo es un buen objetivo para nuestros esfuerzos de paralelización.

Sin embargo, en este caso la dificultad es mucho mayor. Recordemos que el método de Jacobi al utilizar dos vectores, no presenta dependencias entre filas dentro de una iteración. Este tipo de problemas se denominan *trivialmente paralelizables* (embarrassingly parallel).

Por el contrario, el método de Gauss-Seidel tal como lo hemos presentado requiere al procesar una fila tener calculados todos los valores para las filas anteriores.

Ya vimos como esta restricción de secuencialidad ocasionaba problemas en la implementación para BDDs, llevándonos a utilizar métodos alternativos como Gauss-Seidel por bloques o Pseudo Gauss-Seidel. Luego, estos métodos representan el punto de partida para la implementación paralela. Las técnicas que describimos a continuación se aplican de igual forma a cualquiera de ellos.

Una de las ventajas es que para poder llevar a cabo Gauss-Seidel necesitamos de entrada tener la matriz de bloques del BDD. Luego, podemos utilizar esta matriz para dividir las tareas a realizar, sin necesidad de un proceso de división previo como en Jacobi. Teniendo en mente las lecciones aprendidas con Jacobi, elegimos realizar una asignación dinámica para evitar el problema de desbalance entre varios procesadores.

También utilizamos las mismas técnicas para evitar conflictos de escritura, teniendo cuidado de no asignar a dos procesadores bloques con filas en común. Más aún, es necesario fortalecer esta condición para evitar dejar filas a medio terminar, lo que puede impedir el procesamiento de otras filas que dependan de éstas. Luego, en vez de asignar un fragmento, cada procesador recibe una fila entera de la matriz de bloques y procesa de forma secuencial todos los bloques contenidos en esta fila. De esta forma, garantizamos que una vez que se empieza, el procesamiento de una fila termine lo más rápido posible. Como tenemos de antemano la restricción de que dos procesadores no pueden trabajar sobre las mismas filas, esta modificación no impacta en gran medida al grado de concurrencia posible.

Todavía no hemos resuelto el problema de dependencias entre las filas. Una primera observación es que si bien el algoritmo original procesa las filas de la matriz de arriba a abajo, este orden es totalmente arbitrario. De hecho, cualquier otro orden es equivalente a una permutación de las filas de la matriz lo que claramente no altera la solución del sistema.

Pero la observación crucial es la misma que permite la representación eficiente de modelos probabilísticos con matrices ralas: la abundante cantidad de ceros en la matriz. Recordemos que el valor de x_i sólo depende del valor de x_j si $A_{i,j} \neq 0$. Luego, las dependencias reales entre filas dependen de la estructura de la matriz. Si tanto $A_{i,j}$ como $A_{j,i}$ son iguales a cero, podemos calcular los valores de x_i y x_j simultáneamente.

La misma observación se puede llevar al nivel de bloques. Para esto, primero construimos un *grafo de dependencias* entre bloques en base a la matriz de bloques. Cada nodo del grafo corresponde a un bloque de filas, y dos nodos i y j son vecinos si y sólo si alguno de los bloques $A_{i,j}$ o $A_{j,i}$ contienen algún elemento no nulo.

Este grafo puede utilizarse en el proceso de asignación para asegurarse que dos procesadores no trabajen sobre filas que dependan una de la otra (i.e. que sean vecinas en el grafo). Una alternativa es realizar un coloreo del grafo, donde filas vecinas reciben colores distintos. Luego, podemos realizar varias pasadas, procesando en cada una las filas de un mismo color. Como estas filas no tienen dependencias, cada pasada es inherentemente paralelizable. Esta aproximación es adoptada con buenos resultados en [21]. Una dificultad de esta idea es que para aumentar el grado de paralelismo en cada pasada debemos reducir la cantidad de colores utilizados. Esto es similar al problema de determinación del número cromático de un grafo, un clásico ejemplo de problema computacionalmente difícil de resolver.

Nuestra implementación utiliza una idea ligeramente diferente que además de ser más general, puede ser implementada en forma dinámica tal como nos propusimos inicialmente, aunque requiriendo un mayor costo por cada asignación. Para esto, cada nodo del grafo de dependencias tiene un color que puede ser blanco o negro. Inicialmente, todos los nodos son blancos. Cada vez que queremos asignar una tarea, primero elegimos un nodo blanco. Luego, cambiamos el color de este nodo y de todos sus vecinos a negro. Finalmente, devolvemos la fila correspondiente al nodo elegido.

Esto impide procesar al mismo tiempo dos nodos vecinos, ya que el primero que sea asignado pintará de negro al otro, que por lo tanto no puede ser elegido. Por otro lado, cuando terminamos de procesar una fila debemos volver a colorear los nodos vecinos de blanco y eliminar el nodo que acabamos de procesar del grafo o marcarlo para que no vuelva a ser visitado en la misma iteración.

Un detalle más a considerar ocurre cuando un nodo es pintado de negro por dos o más vecinos. Esto está señalando dos o más conflictos, por lo que sólo debemos desmarcarlo cuando ambos vecinos lo hayan pintado de blanco. Tenemos entonces que llevar la cuenta del número de veces que el nodo ha sido pintado de negro (cuantas "capas" de pintura negra tiene) y disminuir esta cuenta en uno (quitar una capa) cada vez que un vecino termine de ser procesado, hasta que el nodo vuelva a ser blanco.

3.3.4. Gauss-Seidel con prioridades

Supongamos que utilizando el método de Gauss-Seidel, tenemos en un momento la elección de dos filas a procesar. Digamos, la fila i y la fila j. Más aún, supongamos que $A_{i,j} \neq 0$ pero $A_{j,i} = 0$. Es decir que el valor de x_i depende del de x_j, pero la relación inversa no es cierta, ya que x_j no depende de x_i.

De cualquier manera, la validez de una sola de estas relaciones fuerza la vecindad de estos nodos en el grafo de dependencia, por lo cual como vimos en la sección anterior, ambas filas no pueden ser procesadas simultáneamente. Sin embargo, el grafo no nos dice nada sobre que nodo nos conviene procesar primero.

Si procesamos primero la fila x_i, entonces el valor de x_j será el calculado en la iteración anterior. Por otro lado, si procesamos primero x_j, entonces x_i utilizará el valor de x_j calculado en esa misma iteración. Obrando siempre bajo la suposición de que el vector x se acerca más y más a una solución con cada iteración, parece evidente que la segunda es la mejor opción.

En otras palabras, el método de la sección anterior admite la siguiente optimización: cuando debemos elegir una fila a procesar, entre todas las filas disponibles elegimos una que tenga la menor cantidad de dependencias sin procesar. La esperanza es que esto permita reducir la cantidad total de iteraciones al reacomodar las cuentas para usar siempre información lo más fresca posible.

Para implementar este método, necesitamos una estructura adicional además del grafo de dependencias: una cola de prioridades. Esta cola contiene únicamente nodos blancos. Luego, cuando un nodo se pinta de negro debemos quitarlo de la cola, y cuando se pinta de blanco lo podemos volver a insertar. La prioridad que asignamos a cada nodo es igual a la cantidad de dependencias no procesadas. Cuando sacamos un nodo de la cola, siempre sacamos el de prioridad más baja.

En términos de la matriz de transición, la prioridad inicial para el nodo i es igual a la cantidad de elementos no nulos en la i-ésima fila de la matriz. Cada vez que terminamos de procesar un nodo debemos reducir en uno la prioridad de todos los nodos que dependen de él. Por ejemplo, para el nodo j debemos recorrer la columna j-ésima de la matriz y reducir la prioridad de todos los nodos i para los cuales $A_{i,j} \neq 0$.

El tiempo total requerido para actualizar esta estructura es entonces proporcional a $\text{neighbours}(i) * \log n$, donde $\text{neighbours}(i)$ es la cantidad de vecinos del nodo i en el grafo de dependencia y n es la cantidad total de filas en la matriz.

Capítulo 4

Resultados

En este capítulo evaluamos los algoritmos paralelos presentados en la sección 3.3. Presentamos en primer lugar las métricas y casos de estudio utilizados para la evaluación. A continuación, damos detalles de nuestras implementaciones y del entorno de ejecución sobre el que trabajamos. Finalmente presentamos los resultados de las pruebas realizadas, comparando los distintos algoritmos e interpretando el comportamiento observado.

4.1. Conceptos básicos de rendimiento

La evaluación de programas que se ejecutan en paralelo requiere cuidado, ya que las métricas incorrectas pueden conducirnos a conclusiones erróneas sobre los resultados.

Los algoritmos sobre los que se basa nuestro trabajo son algoritmos iterativos. Es decir que tienen dos fases muy claras de ejecución: una fase preliminar de inicialización o "set-up" y un lazo principal que ejecuta las iteraciones. Omitimos en nuestras consideraciones la fase de finalización que se ejecuta luego de terminadas las iteraciones, ya que esta realiza más que nada tareas de limpieza y liberación de memoria que son secundarias a los algoritmos utilizados y no representan una fracción significativa del tiempo de ejecución.

La evaluación del rendimiento se basa en el tiempo consumido por las distintas fases, además de la cantidad de memoria utilizada. El T_{setup} es el tiempo que toma la fase de inicialización. Por otro lado, el tiempo de iteración lo dividimos en dos partes: el tiempo de la sección de código que corre de manera secuencial (T_{seq}) y el tiempo de la sección paralelizada (T_{par}). El tiempo total de iteraciones T_{iter} es igual a $T_{\text{seq}} + T_{\text{par}}$. El tiempo total de ejecución es la suma de T_{setup} y T_{iter}.

La métrica principal que utilizamos para analizar el rendimiento es la *aceleración* (speedup). Si llamamos T_i al tiempo de ejecución de nuestro programa utilizando i procesadores, entonces la aceleración S_N para N procesadores es igual a $\frac{T_1}{T_N}$. La aceleración ideal es una curva que crece de forma lineal con la cantidad de procesadores.

Una métrica relacionada es la *eficiencia*, dada por la fórmula $E_N = \frac{T_1}{N \cdot T_N}$. Esta puede interpretarse como el porcentaje de trabajo útil realizado por cada procesador en relación a un procesador trabajando de manera secuencial. Las disminuciones en la eficiencia se deben

en general al tiempo adicional de sincronización o comunicación entre hilos requeridos por la implementación paralela.

La importancia de los tiempos T_{setup} y T_{seq} proviene de la llamada "ley de Amhdal", que cuantifica la aceleración máxima que es posible obtener al paralelizar un programa. Según esta ley, si el fragmento del programa que se paraleliza corresponde a una fracción P del tiempo total de ejecución para el algoritmo serial, la máxima aceleración posible utilizando N procesadores es igual a:

$$\frac{1}{(1-P)+\frac{P}{N}}$$

En otras palabras, la escalabilidad de una implementación paralela está determinada no sólo por la eficiencia en la paralelización, sino también por la cantidad de código secuencial.

4.2. Casos de estudio

En esta sección presentamos los casos de estudio que fueron usados para evaluar el comportamiento de nuestro algoritmo. Las implementaciones utilizadas para cada uno de los modelos son las que vienen incluidas en el directorio `prism-examples` de la distribución estándar de PRISM. Más detalles sobre estos y otros modelos se pueden encontrar en *http://www.prismmodelchecker.org/casestudies/*.

4.2.1. Protocolo de retransmisión acotada

El primer caso de estudio que analizamos es el protocolo de retransmisión acotada o *BRP* (Bounded retransmission protocol) presentado en [13] y analizado también en [11].

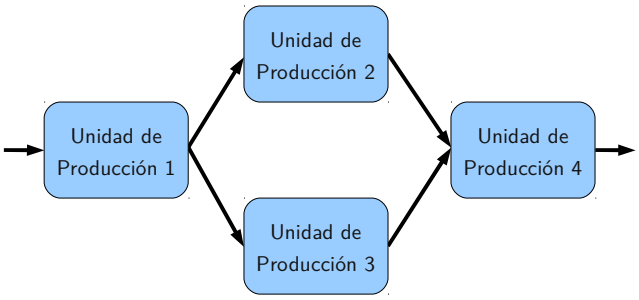
El sistema es una DTMC o cadena de Markov de tiempo discreto que modela dos actores que desean comunicarse entre si y consta de 4 entidades: el emisor, el receptor y dos canales unidireccionales de comunicación, uno en cada sentido. El modelo cuenta con dos parámetros configurables: N, la cantidad de fragmentos a enviar y MAX, el número máximo de intentos de retransmisión.

El mensaje a ser enviado se divide en N fragmentos. El emisor puede transmitir sólo un fragmento a la vez. A su vez, el receptor envía de vuelta mensajes de confirmación por cada fragmento recibido. Ambos canales tiene una probabilidad no nula de perder un mensaje.

El comportamiento del emisor definido por el protocolo es el siguiente: para enviar un mensaje, el emisor transmite sólo el primer fragmento y espera una confirmación de recepción. Cuando el emisor recibe esta respuesta, procede con el siguiente fragmento hasta agotar todo los fragmentos disponibles.

Si el emisor no recibe una confirmación, ya sea porque el fragmento o la confirmación misma se pierden en el camino, el emisor intenta una retransmisión del fragmento. En nuestro modelo, el emisor nunca realiza una retransmisión antes de tiempo (i.e. cuando hay un mensaje o confirmación en camino). En la práctica, la decisión de retransmisión

Figura 4.1: Organización de las unidades de producción en el modelo Kanban



generalmente se basa en un temporizador, por lo que este comportamiento es equivalente a asumir que después de una demora máxima ya no existen chances de recibir un mensaje de confirmación.

Como lo dice el nombre del protocolo, la cantidad de retransmisiones está acotada. Por lo tanto, si después de un número máximo de retransmisiones no se recibe una confirmación, el emisor se da por vencido y activa una condición de error.

El comportamiento del receptor es el inverso: cada vez que este recibe un fragmento, primero comprueba si es el fragmento esperado basándose en su número de secuencia. De ser así, el receptor almacena este fragmento y envía un mensaje de confirmación al emisor. Por otro lado, una comprobación no exitosa indica que el emisor está reenviando un fragmento antiguo al no haber recibido todavía confirmación, por lo que el fragmento se descarta y el receptor reenvía la confirmación de recibo.

4.2.2. Sistema de manufactura Kanban

El segundo caso que empleamos para el análisis está basado en el sistema de manufactura presentado en [9] que emplea la metodología Kanban para controlar la producción. El sistema en este caso se modela con una CTMC, que especifica el comportamiento de 4 unidades de producción. Las unidades se encuentran conectadas en una línea de producción como lo indica la figura 4.1. Los insumos ingresan al sistema por la unidad 1 y el producto final sale del sistema luego de ser procesado por la unidad 4.

La técnica Kanban es un método de planificación originalmente ideado por la empresa Toyota para asegurar el correcto funcionamiento de un sistema de producción. Esta técnica utiliza la demanda de los consumidores para determinar el ritmo de producción en cada fragmento del proceso de fabricación. Por lo tanto, el sistema se comporta de forma reactiva, en vez de intentar anticiparse a la demanda realizando pronósticos de consumo.

La descripción clásica hace uso de las denominadas tarjetas Kanban. Estas tarjetas se adjuntan a los lotes procesados por cada unidad de producción. Cada unidad de producción posee un número limitado de tarjetas, por lo que cuando éstas se agotan la producción se

detiene. Por otro lado, cuando una fase posterior hace uso de uno de estos lotes, esta también envía la tarjeta a la unidad que lo produjo, lo que funciona como notificación para reactivar la producción. El número de tarjetas en una unidad dependerá de factores como la velocidad de fabricación que afecten la capacidad de una unidad de suministrar insumos a sus clientes. La intención es que este número sea lo más pequeño posible. Hoy en día, el uso de tarjetas ha sido en muchos casos suplantado por sistemas electrónicos de notificación, aunque el concepto subyacente sigue siendo el mismo.

Resta describir el modelo que utilizamos para las unidades de producción. Este es un modelo parametrizable en el que para cada unidad, si existen tarjetas e insumos disponibles, se crea un lote nuevo de producción al cual se le asigna una tarjeta. A continuación, existen dos posibilidades: el lote se procesa correctamente y queda disponible para posibles consumidores o se detecta un error en la fabricación y el lote debe ser reprocesado, obedeciendo a otra de las máximas fijadas por Kanban según la cual una unidad de producción no debe entregar nunca productos defectuosos a unidades subsiguientes.

Al modelar el sistema con una CTMC, la ocurrencia de los eventos de fabricación (creación de un lote nuevo, procesamiento correcto de un lote, detección de defectos y suministro a un consumidor) están controlados por distribuciones de probabilidad exponencial con distintos parámetros, que varían para cada unidad de producción. Un parámetro t controla la cantidad de tarjetas Kanban o "tokens" asignadas a cada unidad, y al ser incrementado es el que influencia la complejidad del modelo obtenido.

4.2.3. Sistema de sondeo cíclico

El tercer caso estudiado es un sistema que modela un servidor con un esquema de "polling" o sondeo cíclico. Tal sistema consta de dos o más estaciones que reciben y almacenan mensajes, y un servidor centralizado que procesa estos mensajes. El término polling hace referencia a un esquema donde el servidor es quien debe consultar constantemente cada una de las estaciones para verificar si han recibido algún mensaje nuevo, en vez de ser estas últimas quienes notifican al servidor. Este sistema es analizado en detalle en [14] como un ejemplo de aplicación de técnicas de verificación de modelos.

En PRISM, la representación utilizada es una CTMC que modela el servidor y las estaciones periféricas. La cantidad de estaciones en el modelo se denota por N. Cada estación tiene una cola que almacena como máximo un mensaje. Cuando esta cola esta vacía, el tiempo hasta recibir el próximo mensaje está caracterizado por una distribución exponencial de parámetro λ.

El servidor realiza un barrido cíclico de todas las estaciones, sondeando cada una para verificar si ha recibido o no un mensaje nuevo. El tiempo de sondeo está modelado por una distribución exponencial de parámetro γ. Si la respuesta es positiva y la estación informa que tiene un mensaje en espera, el servidor retira y procesa dicho mensaje. El tiempo de procesamiento también obedece a una distribución exponencial con parámetro μ. El servidor sólo puede procesar un mensaje por vez y luego de procesar un mensaje procede directamente a la siguiente estación, por lo que nunca procesa más de dos mensajes seguidos de una misma estación en una misma ronda de polling.

4.3. Implementación

La figura 4.2 muestra la arquitectura básica de PRISM La herramienta está implementada principalmente en el lenguaje Java y C++. Java se utiliza para la interfaz de usuario, el parser de modelos y propiedades, la lógica de verificación de más alto nivel y para la interfaz con otras herramientas. Por otro lado, tanto los algoritmos de verificación como la librería de manejo de BDDs (CUDD), el código para matrices ralas y sus extensiones están implementadas en C++ para lograr el mayor grado de eficiencia. El framework JNI (Java Native Interface) permite acceder a estos métodos desde Java de manera transparente.

Nuestra implementación paralela está basada en el uso de la librería POSIX de hilos (pthreads). Esta decisión se debe a la flexibilidad necesaria para implementar los algoritmos propuestos. Otras librerías como OpenMP simplifican el proceso de paralelización pero están principalmente pensadas para la paralelización de ciclos, por lo que resulta difícil adaptarlas a algoritmos recursivos o tener control fino sobre la planificación de los hilos.

El código modificado se halla en la carpeta **src/hybrid** que corresponde a la implementación del motor híbrido. Los otros motores se encuentran en **src/mtbdd** (simbólico puro) y **src/sparse** (matrices ralas). Todos los algoritmos hacen uso de un mismo framework para hilos implementado en `threads.cc`. Para su utilización, el usuario debe definir cuatro métodos: *units_left*, *get_unit*, *process_unit* y *free_unit*. El primero determina si existen todavía elementos a ser procesados, el segundo devuelve un ítem para su procesamiento, el tercero procesa un ítem dado y el último permite ejecutar código específico a un ítem luego de que este ha sido procesado.

A modo de ejemplo, estos métodos pueden implementarse en base a una cola de tareas, obteniendo lo que se conoce como un "thread pool". Luego, con cada llamada a la rutina *run_threads*, los distintos hilos ejecutan hasta agotar las tareas en la cola. La cantidad de hilos puede definirse a través de la función *set_num_threads*. La librería también asegura la exclusión mutua durante la asignación de tareas, por lo que el cliente debe preocuparse por el acceso a recursos compartidos en la implementación de un algoritmo sólo durante la fase de procesamiento.

Las implementaciones de los algoritmos de solución se hallan en `PH_JOR.cc` (Jacobi), `PH_SOR.cc` (Gauss-Seidel por bloques) y `PH_PSOR.cc` (Pseudo Gauss-Seidel). Estos métodos son utilizados por las rutinas en `PH_ProbUntil.cc` y `PH_StochSteadyState.cc`, que implementan la verificación del operador $\mathscr{U}$ para DTMCs y el cálculo de probabilidades de estado estable para CTMCs respectivamente.

Como fue mencionado anteriormente, la verificación de los demás operadores se implementan en base a la multiplicación de matrices, imitando el modelo utilizado para el método de Jacobi. Estos incluyen los códigos en `PH_ProbBoundedUntil.cc` y `PH_StochBoundedUntil.cc`, que verifican el operador $\mathscr{U}^{\leq k}$ para DTMC y CTMC respectivamente, así como los códigos para MDP en `PH_NondetUntil.cc` y `PH_NondetBoundedUntil.cc` que implementan las dos versiones del operador until.

Finalmente, la medición de tiempos se realiza a través de la función *clock_gettime* de la librería *glibc*, que permite acceso a los relojes provistos por el kernel de Linux. Los relojes utilizados son **CLOCK_MONOTONIC** para medir el tiempo total del proceso y **CLOCK_THREAD_CPUTIME_ID** para medir el tiempo por hilo.

Figura 4.2: Arquitectura de PRISM

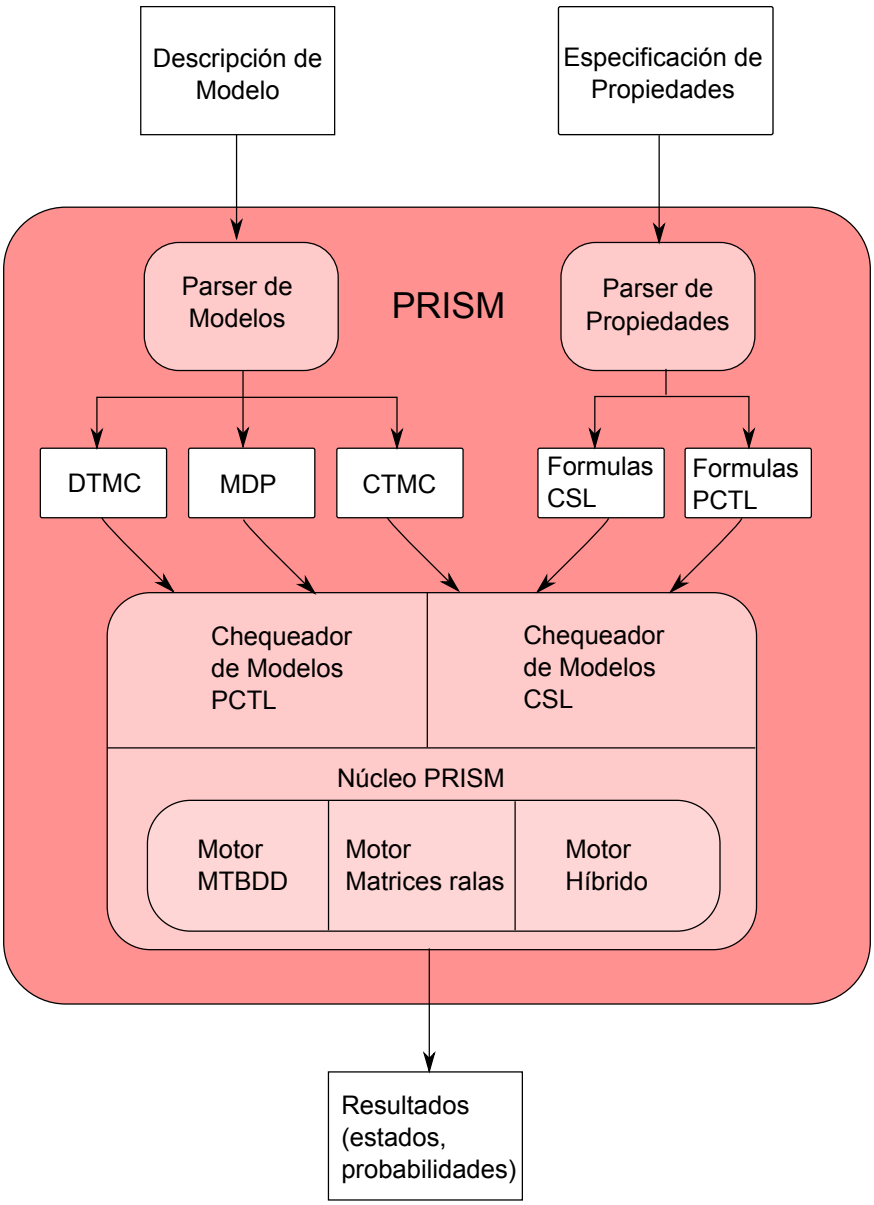


Tabla 4.1: Tiempos para BRP (N = 512, MAX = 30) con Jacobi para BDDs

Hilos	T_{seq}	T_{par}	T_{iter}	T_{setup}	Total	Speedup (paralelo)	Eficiencia (paralelo)	Speedup (total)	Eficiencia (total)
1	4.152	57.467	61.619	0.083	61.702	1.000×	100.0%	1.000×	100.0%
2	4.587	30.066	34.653	0.080	34.733	1.911×	95.6%	1.776×	88.8%
3	4.954	22.283	27.237	0.082	27.319	2.579×	86.0%	2.259×	75.3%
4	5.539	19.062	24.601	0.082	24.683	3.015×	75.4%	2.500×	62.5%
5	5.884	17.273	23.157	0.080	23.237	3.327×	66.5%	2.655×	53.1%
6	5.895	14.510	20.405	0.083	20.488	3.961×	66.0%	3.012×	50.2%
7	6.076	13.668	19.744	0.092	19.836	4.204×	60.1%	3.111×	44.4%
8	6.455	13.574	20.029	0.084	20.113	4.234×	52.9%	3.068×	38.3%
9	6.926	13.080	20.006	0.085	20.091	4.394×	48.8%	3.071×	34.1%
10	7.029	11.291	18.320	0.087	18.407	5.090×	50.9%	3.352×	33.5%
11	7.170	10.563	17.733	0.087	17.820	5.440×	49.5%	3.463×	31.5%
12	6.956	10.391	17.347	0.085	17.432	5.530×	46.1%	3.540×	29.5%
13	7.538	10.685	18.223	0.103	18.326	5.378×	41.4%	3.367×	25.9%
14	9.482	10.938	20.420	0.088	20.508	5.254×	37.5%	3.009×	21.5%
15	7.675	13.322	20.997	0.088	21.085	4.314×	28.8%	2.926×	19.5%
16	6.895	11.159	18.054	0.085	18.139	5.150×	32.2%	3.402×	21.3%

4.4. Experimentos y análisis

Presentamos ahora los resultados de nuestro experimentos para evaluar la performance de los algoritmos propuestos. Los experimentos fueron realizados en *Shiva*, una máquina con 4 procesadores quad-core Opteron 8350 corriendo a 2Ghz con 120Gb de memoria principal. Esta es una arquitectura NUMA (Non-Uniform Memory Access o Acceso a Memoria No Uniforme), por lo que de los 120Gb de memoria total, 30Gb son locales a cada procesador. Una red de interconexiones denominada HyperTransport permite la sincronización y comunicación entre cualquier par de procesadores. El sistema operativo usado fue Ubuntu Linux para servidores, kernel 2.6.32-29 con soporte SMP. Todos los tiempos medidos están expresados en segundos. Si bien reportamos resultados con hasta 16 procesadores, debemos notar que al momento de correr nuestro experimentos el sistema tenía una carga promedio equivalente a entre 1 y 3 procesadores en uso.

4.4.1. Jacobi

La tabla 4.1 muestra los resultados para el método de Jacobi presentado en la sección 3.3.1. El caso de estudio analizado es el protocolo de retransmisión acotada con 512 fragmentos por paquete y un limite de 30 retransmisiones. La propiedad verificada es una modificación de una de las propiedades estudiadas en [11] y corresponde a la probabilidad de que el emisor no reporte un envío exitoso en a lo sumo 1000 pasos. La fórmula PCTL que la expresa es

Tabla 4.2: Tiempos para BRP (N = 2048, MAX = 50) con Jacobi para BDDs

Hilos	T_{seq}	T_{par}	T_{iter}	T_{setup}	Total	Speedup (paralelo)	Eficiencia (paralelo)	Speedup (total)	Eficiencia (total)
1	22.693	38.530	61.223	0.248	61.471	1.000×	100.0%	1.000×	100.0%
2	28.963	26.540	55.503	0.263	55.766	1.452×	72.6%	1.102×	55.1%
3	33.185	23.341	56.526	0.250	56.776	1.651×	55.0%	1.083×	36.1%
4	34.157	25.001	59.158	0.248	59.406	1.541×	38.5%	1.035×	25.9%
5	37.147	25.661	62.808	0.249	63.057	1.502×	30.0%	0.975×	19.5%
6	37.080	21.761	58.841	0.252	59.093	1.771×	29.5%	1.040×	17.3%
7	35.507	20.510	56.017	0.264	56.281	1.879×	26.8%	1.092×	15.6%
8	38.431	24.327	62.758	0.252	63.010	1.584×	19.8%	0.976×	12.2%
9	40.221	23.647	63.868	0.255	64.123	1.629×	18.1%	0.959×	10.7%
10	39.594	21.105	60.699	0.253	60.952	1.826×	18.3%	1.009×	10.1%
11	38.916	21.274	60.190	0.256	60.446	1.811×	16.5%	1.017×	9.2%
12	38.754	21.264	60.018	0.255	60.273	1.812×	15.1%	1.020×	8.5%
13	42.746	23.120	65.866	0.261	66.127	1.667×	12.8%	0.930×	7.2%
14	41.781	23.526	65.307	0.269	65.576	1.638×	11.7%	0.937×	6.7%
15	43.047	26.516	69.563	0.261	69.824	1.453×	9.7%	0.880×	5.9%
16	40.974	27.789	68.763	0.266	69.029	1.387×	8.7%	0.891×	5.6%

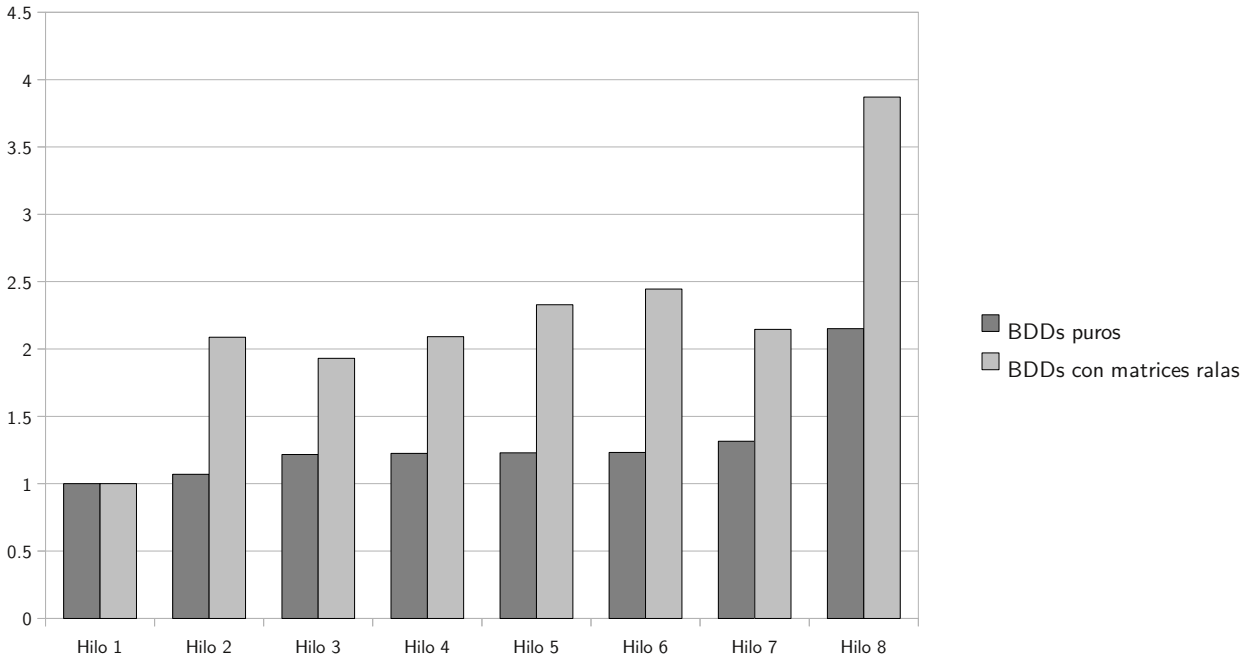
$\mathscr{P}[\textbf{true}\ \mathscr{U}^{\leq 1000}\ s = 5 \wedge T]$. El BDD utilizado para almacenar el modelo es un BDD puro sin matrices ralas, aunque para el vector de iteraciones utilizamos un vector explícito ya que estamos trabajando con el motor híbrido.

Un punto que puede causar confusión es el operador $\mathscr{U}^{\leq k}$ cuya verificación no requiere la aplicación del método de Jacobi. Sin embargo, como mencionamos en la sección 3.2, cuando hablamos de Jacobi hacemos en realidad referencia a todos los algoritmos de verificación implementados en términos de multiplicación de matrices.

Se puede ver como el código paralelo escala razonablemente bien hasta aproximadamente 12 procesadores, con una eficiencia que se mantiene sobre el 50%. Para 13 procesadores en adelante hay un poco de inestabilidad, que puede responder a una combinación de carga del sistema y saturación de recursos de la arquitectura. El rendimiento total también es menor a causa del tiempo consumido por el código secuencial que representa una porción importante del tiempo total. De todos modos, la aceleración final máxima de 3,54× obtenida para 12 procesadores es más que respetable y resulta una buena muestra de las ganancias que se pueden obtener con este tipo de algoritmos.

La tabla 4.2 muestra el comportamiento del mismo método aplicado a BDDs con matrices ralas incrustadas. Este algoritmo es más rápido que el anterior por lo que el tamaño del modelo se incrementa a 2048 fragmentos y 50 retransmisiones. Las perspectivas son bastante desalentadoras. La aceleración del código paralelo parece oscilar entre 1,4× y 1,8×, independiente de la cantidad de procesadores que se utilizan. La eficiencia también decae rápidamente, y el tiempo total es comparable (y en algunos casos peor) al de la versión

Figura 4.3: Tiempo de procesamiento por hilo



secuencial.

Si bien los resultados en este último caso no son positivos, es importante destacar las causas que producen esta degradación en el rendimiento. El primer culpable es el algoritmo de balanceo estático. Como mencionamos en la introducción de la sección 3.3.2, las matrices ralas en BDDs mixtos dificultan la predicción del cómputo requerido al modificar los patrones de acceso a memoria. Luego, el modelo de costos simple que utilizamos ya no funciona correctamente. Esto se puede ver en el gráfico 4.3 que muestra el tiempo relativo de procesamiento por hilo para una ejecución con 8 hilos.

Observando las barras en gris oscuro, se puede ver que la división de tareas es relativamente pareja para el algoritmo aplicado a BDDs puros, con una diferencia máxima de 2,15× entre el hilo más lento y el hilo más rápido. Sin embargo, al introducir matrices ralas esta diferencia se eleva a 3,87×, lo que significa que un hilo está demorando casi 4 veces más que los demás Aún ajustando el modelo de costos, resulta muy difícil predecir como factores de bajo nivel, como la contención en el acceso a memoria, afectarán el rendimiento de cada hilo en particular, lo que es una debilidad inherente a cualquier algoritmo de balanceo estático.

El segundo responsable del bajo rendimiento paralelo es irónicamente la misma optimización que se incorpora para mejorar el funcionamiento del algoritmo: las matrices ralas. Al mejorar la localidad en el acceso a memoria y reducir el tiempo requerido para recorrer el BDD, estas reducen la proporción de tiempo consumido por el cómputo que queremos paralelizar. Como consecuencia, las ganancias totales se ven reducidas tal como lo describe la ley de Amhdal. Esto se compone con otro problema, que es el incremento de los tiempos

Tabla 4.3: Tiempos para Kanban (N = 4) con Jacobi para matrices ralas

Hilos	T_{seq}	T_{par}	T_{iter}	T_{setup}	Total	Speedup (paralelo)	Eficiencia (paralelo)	Speedup (total)	Eficiencia (total)
1	3.671	12.858	16.529	1.153	17.682	1.000×	100.0%	1.000×	100.0%
2	3.762	7.240	11.002	1.149	12.150	1.776×	88.8%	1.455×	72.8%
3	3.894	5.356	9.250	1.144	10.394	2.401×	80.0%	1.701×	56.7%
4	3.835	4.280	8.115	1.142	9.258	3.004×	75.1%	1.910×	47.7%
5	3.877	3.771	7.648	1.150	8.798	3.410×	68.2%	2.010×	40.2%
6	3.959	3.517	7.476	1.154	8.629	3.656×	60.9%	2.049×	34.2%
7	3.920	3.392	7.312	1.153	8.465	3.791×	54.2%	2.089×	29.8%
8	4.188	3.324	7.512	1.146	8.658	3.868×	48.4%	2.042×	25.5%
9	4.462	3.069	7.531	1.148	8.679	4.189×	46.5%	2.037×	22.6%
10	4.583	3.052	7.636	1.152	8.787	4.213×	42.1%	2.012×	20.1%
11	4.686	2.824	7.510	1.147	8.657	4.553×	41.4%	2.042×	18.6%
12	5.092	3.065	8.157	1.152	9.309	4.195×	35.0%	1.899×	15.8%
13	4.426	3.167	7.594	1.153	8.747	4.060×	31.2%	2.021×	15.5%
14	4.532	3.028	7.560	1.161	8.721	4.246×	30.3%	2.027×	14.5%
15	5.094	3.515	8.610	1.149	9.759	3.658×	24.4%	1.812×	12.1%
16	4.911	3.380	8.291	1.153	9.443	3.804×	23.8%	1.872×	11.7%

secuenciales causados por el paso de acumulación al final de cada iteración. A diferencia de los algoritmos analizados en el resto de este capítulo, la división irregular que empleamos aquí dificulta el uso de exclusión mutua para sincronización. Luego, nuestra implementación utiliza un vector por hilo, acumulando los valores de estos vectores al final de cada iteración. Evidentemente, la acumulación demanda más tiempo mientras mayor sea la cantidad de hilos.

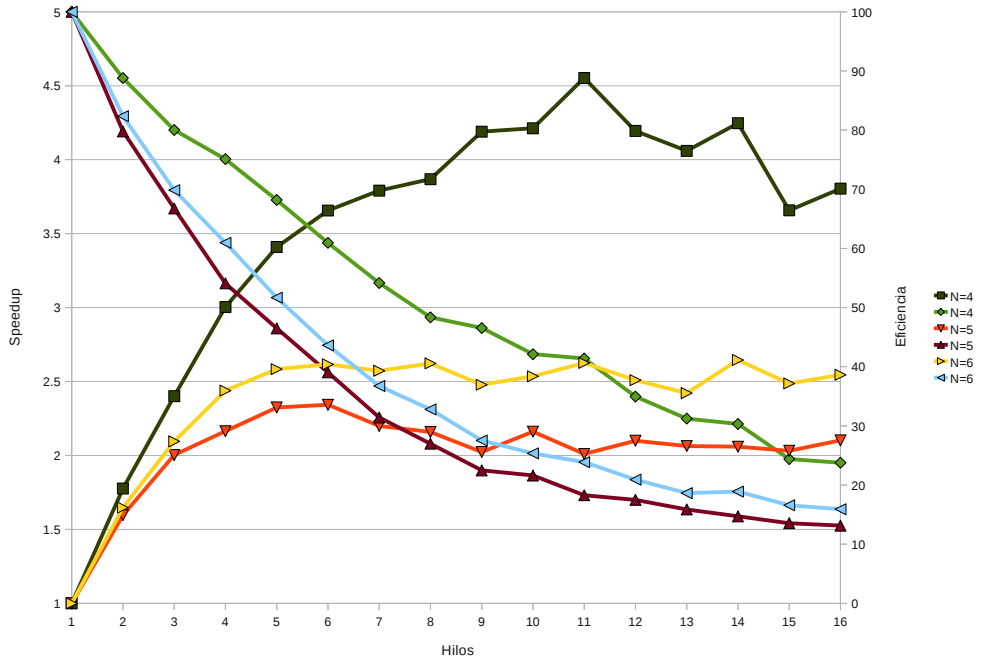
En resumen, el algoritmo desarrollado funciona satisfactoriamente para los BDDs puros que son la piedra base de los métodos de verificación simbólicos. Por otro lado, las optimizaciones empleadas pierden efectividad con la introducción de matrices ralas incrustadas que cambian la dinámica del proceso de verificación. Para enfrentar este problema, debemos recurrir a los algoritmos que describimos en las siguientes secciones.

4.4.2. Jacobi con matrices incrustadas

En la tabla 4.3 vemos los resultados del método de Jacobi modificado para BDDs con matrices ralas. La propiedad verificada cuantifica la probabilidad de encontrarse en un estado en el que la primera unidad de producción tiene al menos un producto terminado o en producción. La fórmula CSL correspondiente es $\mathscr{S}[w1 > 0]$.

La cantidad máxima de hilos para los que evaluamos está limitada por el número de núcleos en nuestro sistema (16). Esta cantidad es suficiente para saturar el total de procesadores, ya que por la naturaleza de los algoritmos usados, los hilos no bloquean por largos

Figura 4.4: Kanban con Jacobi para matrices ralas (código paralelo)



períodos de tiempo. Luego, agregar más hilos sólo aumenta las demoras por sincronización reduciendo el rendimiento total del programa.

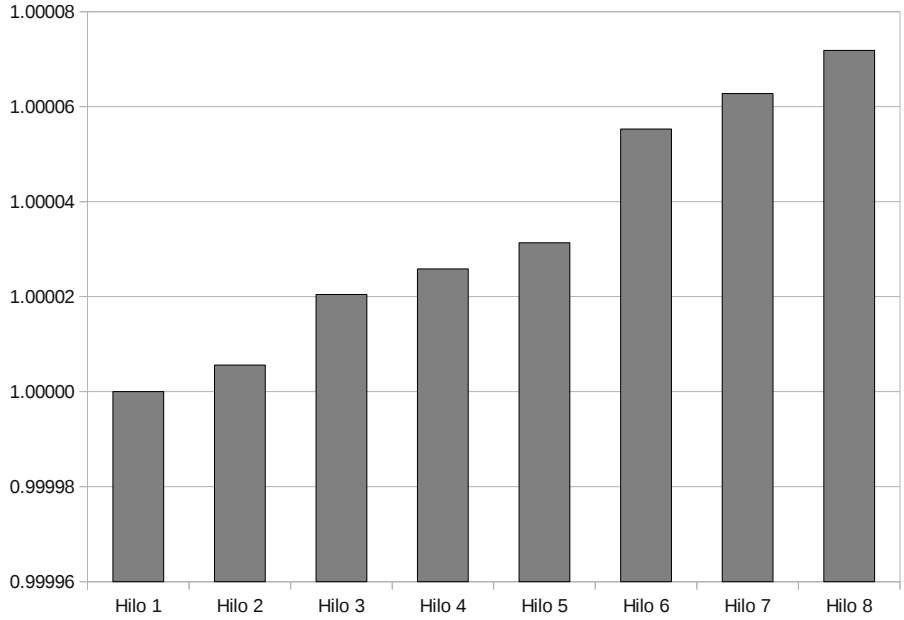
El método de Jacobi posee una inicialización bastante sencilla, por lo que el tiempo de inicialización no tiene un gran impacto sobre el tiempo total de ejecución, como podemos comprobar viendo la tabla. Por otro lado, el código secuencial sigue representando una fracción importante de cada iteración por lo que de acuerdo a la ley de Amhdal, el grado máximo de aceleración se encuentra bastante limitado. Por ejemplo, para un sistema Kanban con $N = 5$ la aceleración máxima para las iteraciones con 16 procesadores no puede superar $\frac{174688042}{29003209+145684833/16} = 4{,}58$. Como se comprueba para el caso $N = 4$, esto reduce una aceleración de 3,8 en el código paralelo a una aceleración total de 1,87.

El gráfico de la figura 4.5 demuestra la efectividad de la asignación dinámica de tareas para distribuir el trabajo entre hilos. Como se puede observar en dicho gráfico, los tiempos por hilos son todos muy parejos, sin las desviaciones que se observaban en el método anterior al introducir las matrices ralas. Luego, podemos decir que este algoritmo soluciona exitosamente el problema del balanceo de cargas.

Sin embargo, los datos de la tabla también muestran que el código paralelo no escala suficientemente bien, con una eficiencia que sólo se mantiene por encima de un 70 % al correr con 4 hilos o menos. Al aumentar la cantidad de hilos, los costos de sincronización comienzan a manifestarse llevando a situaciones donde el uso de más hilos aumenta el tiempo total y termina resultando contraproducente. La figura 4.4 muestra que la situación no mejora al aumentar el tamaño del modelo.

El análisis que realizamos nos lleva a hipotetizar que la principal causa de este problema

Figura 4.5: Tiempo de procesamiento por hilo



son las escrituras a memoria compartida. Recordemos que en este método se utilizan dos vectores, uno de sólo lectura correspondiente a la iteración anterior y un segundo donde calculan los resultados para la iteración actual. Para empezar, en nuestra implementación el segundo de los vectores está almacenado completamente en la memoria de un único procesador. Luego, todos los otros procesadores deben acceder a memoria no local lo que implica una ligera demora adicional y puede también saturar el ancho de banda de la memoria utilizada.

Una solución posible, como mencionamos en la sección 3.3.2, es utilizar memoria local a cada hilo para realizar los cálculos durante cada iteración. Esto produce un vector por cada hilo que luego deben ser acumulados en un vector global antes de la siguiente iteración. Sin embargo, el problema de la escritura compartida sigue estando presente a la hora de consolidar los vectores y es peor a medida que aumenta la cantidad de hilos. Experimentos realizados con esta aproximación ofrecieron resultados comparables con los anteriormente obtenidos.

Una observación al margen que surge de este análisis concierne la prevención de conflictos utilizando exclusión mutua. Al intentar calcular las demoras que esto genera, podemos comparar el código con una versión donde las unidades asignadas a distintos procesadores pueden acceder a las mismas posiciones en memoria. Si bien esto provoca resultados cada vez más incorrectos al aumentar la cantidad de hilos y luego de conflictos, también simplifica la asignación de tareas reduciendo la sección crítica del código. Podemos esperar entonces un programa que, aunque incorrecto, sea más veloz que el original y así medir el impacto de los métodos de sincronización usados.

En la práctica, los resultados de este experimento son altamente contraintuitivos: el programa no sólo produce valores incorrectos sino que también es más lento que el original.

Tabla 4.4: Tiempos para Kanban (N = 4) con Gauss-Seidel

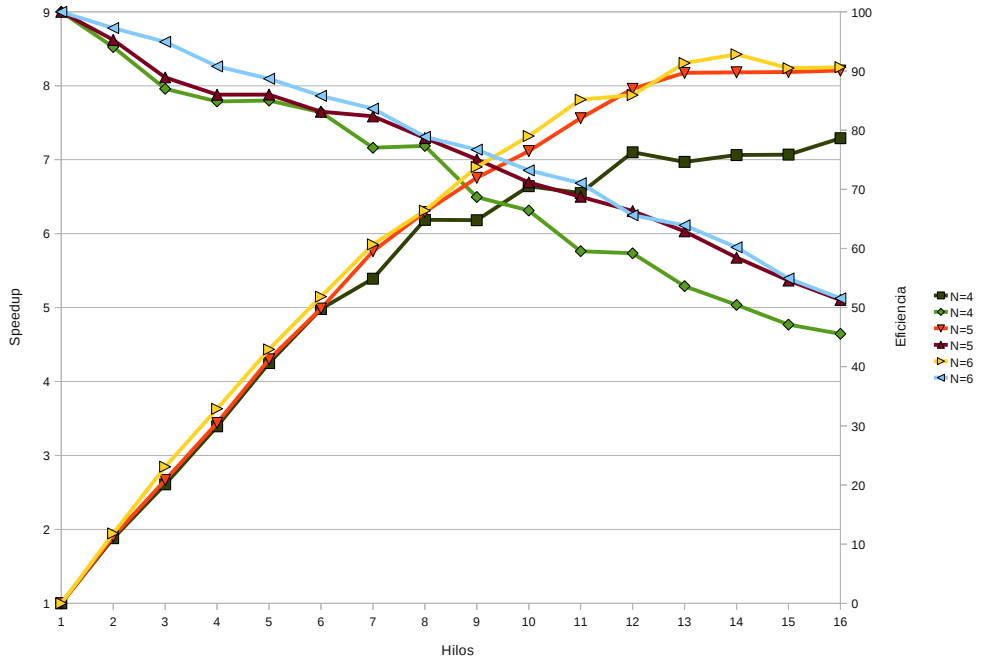
Hilos	T_{seq}	T_{par}	T_{iter}	T_{setup}	Total	Speedup (paralelo)	Eficiencia (paralelo)	Speedup (total)	Eficiencia (total)
1	0.003	13.955	13.958	2.038	15.996	1.000×	100.0%	1.000×	100.0%
2	0.003	7.419	7.422	2.035	9.456	1.881×	94.1%	1.692×	84.6%
3	0.003	5.347	5.350	2.034	7.384	2.610×	87.0%	2.166×	72.2%
4	0.003	4.112	4.114	2.035	6.149	3.394×	84.9%	2.601×	65.0%
5	0.003	3.284	3.286	2.038	5.324	4.250×	85.0%	3.004×	60.1%
6	0.003	2.801	2.804	2.037	4.841	4.982×	83.0%	3.304×	55.1%
7	0.003	2.588	2.591	2.020	4.611	5.391×	77.0%	3.469×	49.6%
8	0.003	2.256	2.259	2.047	4.305	6.186×	77.3%	3.715×	46.4%
9	0.003	2.257	2.260	2.030	4.290	6.182×	68.7%	3.728×	41.4%
10	0.003	2.101	2.104	2.034	4.139	6.641×	66.4%	3.865×	38.6%
11	0.003	2.131	2.134	2.032	4.166	6.549×	59.5%	3.839×	34.9%
12	0.004	1.965	1.969	2.026	3.996	7.101×	59.2%	4.003×	33.4%
13	0.003	2.002	2.005	2.039	4.044	6.971×	53.6%	3.955×	30.4%
14	0.004	1.976	1.980	2.043	4.022	7.063×	50.4%	3.977×	28.4%
15	0.003	1.974	1.978	2.033	4.011	7.068×	47.1%	3.988×	26.6%
16	0.003	1.914	1.918	2.035	3.953	7.290×	45.6%	4.046×	25.3%

Esto no parece razonable, ya que hemos disminuido el código y aumentado el grado de entrelazado (interleaving) posible. La respuesta a este enigma se encuentra en un aspecto crucial de sistemas con múltiples procesadores: la coherencia entre cachés. Cuando dos o más procesadores trabajan sobre una misma porción de memoria, las escrituras de uno de estos puede afectar posiciones de memoria que estaban cacheadas en el segundo. Esto invalida la caché de los demás procesadores, lo que penaliza su rendimiento. Esto es un problema aún cuando el entrelazado de estas instrucciones no afecte el resultado del algoritmo. Luego, la exclusión mutua no sólo asegura resultados correctos sino que al permitir acceso a un procesador por vez tiene también el efecto no anticipado de reducir los problemas de coherencia.

4.4.3. Gauss-Seidel

La tabla 4.4 muestra los resultados para el método de Gauss-Seidel expuesto en la sección 3.3.3. La variante utilizada es Pseudo Gauss-Seidel. Como se puede ver, esta situación es muy diferente a la encontrada con Jacobi. Para empezar, el tiempo de inicialización juega aquí un papel más importante. El incremento en el tiempo de inicialización en comparación con Jacobi se debe principalmente a la construcción de la matriz de bloques y el grafo de dependencias antes de comenzar las iteraciones, lo que requiere recorrer el BDD original. Como vemos para Kanban con $N = 4$, el tiempo de inicialización es la mitad del tiempo total de ejecución, transformando una aceleración de aproximadamente 7,3 en el código paralelo en una aceleración total de 4,04. Afortunadamente, este efecto se reduce a medida

Figura 4.6: Kanban con Gauss-Seidel (código paralelo)



que aumenta el tamaño del modelo.

Por otro lado, la mayor parte del código que se ejecuta durante las iteraciones es paralelizable. Luego, el tiempo del código secuencial tiene un efecto despreciable. El resultado es una aceleración de 7.29x a 8.25x en las iteraciones, o equivalentemente una eficiencia mayor al 50 % que incluso se mantiene sobre el 75 % para menos de 8 procesadores. Estos resultados son comparables a los obtenidos en [21].

Mejor aún, como vemos en la figura 4.6, la escalabilidad del código paralelo aumenta a medida que incrementamos el tamaño del modelo. Las curvas de aceleración crecen en un principio casi linealmente y para los modelos grandes, sólo se estabilizan al alcanzar el límite de procesadores que ofrece nuestra arquitectura. Sumado a la menor influencia de la fase de inicialización, es posible alcanzar una aceleración *total* de entre 7× y 8×, lo que constituye una reducción notable en los tiempos de verificación para modelos de gran tamaño.

Comparando con el método de Jacobi, una lección que se puede extraer es que no siempre un mayor grado de libertad se traduce en mejoras en el rendimiento. Se podría pensar que al planificar bloques de forma independiente se reducen las demoras por exclusión mutua y aumentan el número de tareas disponibles lo que mejora el balanceo de cargas. Sin embargo, el empeoramiento en el uso de la memoria debido a la necesidad de reinicializar y cargar múltiples veces el vector correspondiente a un mismo conjunto de filas tiene un impacto negativo que resulta más notorio en el resultado final.

Tabla 4.5: Kb de memoria utilizada para Kanban (N = 6)

Hilos	Jacobi	Gauss-Seidel
Secuencial	200,268.0	111,337.2
1	200,268.0	143,322.1
2	200,268.0	143,395.6
3	200,268.0	143,469.1
4	200,268.0	143,542.6
5	200,268.0	143,616.1
6	200,268.0	143,689.6
7	200,268.0	143,763.1
8	200,268.0	143,836.6
9	200,268.0	143,910.1
10	200,268.0	143,983.6
11	200,268.0	144,057.1
12	200,268.0	144,130.6
13	200,268.0	144,204.1
14	200,268.0	144,277.6
15	200,268.0	144,351.1
16	200,268.0	144,424.6

4.4.4. Consumo de memoria

La tabla 4.5 muestra la cantidad de memoria consumida por estos últimos dos métodos para un caso particular. Para Jacobi, la cantidad de memoria utilizada no está afectada por la cantidad de hilos, ya que se utilizan siempre los dos mismos vectores para realizar las iteraciones. Además, el incremento de memoria por sobre la implementación secuencial es menor al 1 %, ya que sólo se requiere memoria adicional para almacenar la cola de bloques a procesar.

En Gauss-Seidel, el incremento sobre la implementación secuencial es más notable, ya que se requiere suficiente espacio adicional para almacenar las entradas del grafo de dependencia lo que significa un aumento del 28,7 % en la cantidad de memoria utilizada. También hay un ligero incremento al aumentar la cantidad de hilos, correspondiente a los vectores adicionales que se utilizan para almacenar los resultados temporales de un conjunto de filas. Sin embargo, vemos que con bloques suficientemente pequeños este aumento no es realmente notable. Como vemos en el gráfico de la figura 4.7, el mayor porcentaje sigue estando dedicado al vector de iteración.

4.4.5. Gauss-Seidel con prioridades

La tabla 4.6 ilustra la cantidad de iteraciones necesarias para la convergencia del método de Gauss-Seidel en un sistema de sondeo cíclico. La propiedad verificada es una propiedad de estado estable que representa la probabilidad de que la primera estación se encuentre

Figura 4.7: Gauss-Seidel - Uso de memoria

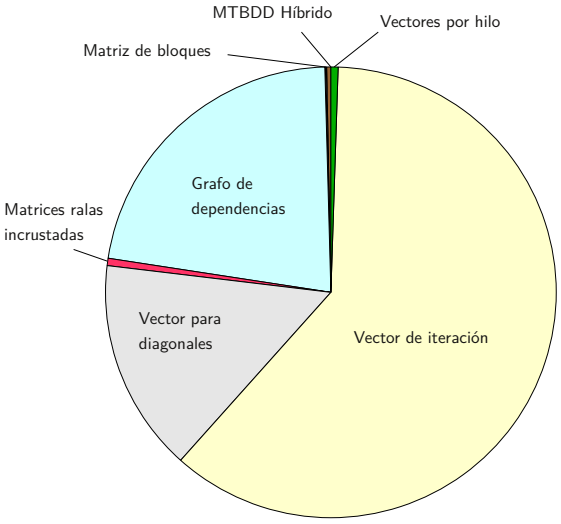


Tabla 4.6: Iteraciones para Polling (Gauss-Seidel con y sin prioridades)

	1 hilo		8 hilos	
N	Con	Sin	Con	Sin
2	46	46	46	46
3	75	88	75	88
4	102	132	119	132
5	128	178	162	163
6	107	225	160	182
7	124	275	204	213
8	141	325	243	237
9	158	377	279	275
10	166	429	307	311
11	182	483	342	327
12	199	537	375	371
13	207	577	383	396
14	223	633	418	427
15	231	694	437	459
16	248	750	466	483

esperando servicio. Su fórmula CSL es $\mathscr{S}[s1 = 1 \wedge \neg(s = 1 \wedge a = 1)]$. La intención es mostrar los resultados de la optimización mencionada en la sección 3.3.4.

Las primeras dos columnas denotadas *Con* y *Sin* muestran resultados altamente positivos. Para un sistema de sondeo con 16 estaciones, el método de Gauss-Seidel con prioridades requiere 1/3 de iteraciones en comparación con el método sin prioridades (248 vs 750). Si bien hay un ligero incremento por la estructura utilizada para el manejo de prioridades, el tiempo por iteración se mantiene prácticamente igual. Luego, esto se traduce en una reducción del 66 % en el tiempo de verificación.

Por otro lado, al ejecutar los mismos algoritmos con 8 hilos en paralelo, la reducción en la cantidad de iteraciones es notablemente menor, rondando un 5 %. Esto se justifica ya que aunque la incorporación de prioridades designa un orden parcial entre los bloques, al ejecutar el algoritmo en paralelo este orden no siempre puede ser obedecido por demoras en el procesamiento de algunos bloques.

Por ejemplo, si el bloque A utiliza los valores de los bloques B y C, la versión secuencial procesará B, C y finalmente A con los valores actualizados de B y C. Por otro lado, si el algoritmo se ejecuta paralelamente con 3 o más hilos, procesará A, B y C simultáneamente, por lo que la ventaja se pierde. Aún si sólo se ejecutan 2 hilos, si uno de los bloques B y C terminan de procesar y no hay otro candidato disponible, para no desperdiciar recursos debemos continuar inmediatamente con el bloque A.

Esto también explica el incremento en el número total de iteraciones para el algoritmo con prioridades al pasar de la versión secuencial a la versión paralela. Sin embargo, la tabla muestra el efecto inverso para el algoritmo sin prioridades: la cantidad total de iteraciones se reduce. Aunque no hemos podido establecer las causas de dicho comportamiento, esto explica también por qué la diferencia entre los algoritmos paralelos es menor.

Capítulo 5

Conclusiones

Hemos presentado en este trabajo tres algoritmos paralelos distintos para resolver el problema de verificación de modelos. Todos están basados en algoritmos existentes, que han sido modificados para atender los problemas característicos causados por la ejecución concurrente. Hemos demostrado que con una implementación correcta es posible obtener algoritmos con un buen grado de escalabilidad que ofrecen mejoras de rendimiento notables en comparación con una versión secuencial. Sin embargo, también vimos como detalles menores, principalmente relacionados a la arquitectura utilizada, toman un papel mucho más relevante a la hora de paralelizar y pueden tener un gran impacto en la utilidad de los algoritmos obtenidos.

El diseño de algoritmos distribuidos de verificación es un área relativamente novedosa y poco explorada, más aún en lo que se refiere a la aplicación de técnicas simbólicas para el tratamiento de modelos probabilísticos. La mayoría de las herramientas disponibles están diseñadas para correr sobre un único procesador y no poseen soporte para concurrencia. Podemos mencionar de todas formas algunos trabajos similares al nuestro dentro de estas líneas. En [17] se presenta una paralelización del método de Gauss-Seidel basada en la variante de Gauss-Seidel por bloques. Si bien las ganancias en rendimiento son considerables, el análisis se restringe al funcionamiento en sistemas con dos procesadores. También se introducen técnicas de precondicionamiento, similares a nuestra variante para Gauss-Seidel con prioridades, que reducen la cantidad de iteraciones necesarias para lograr la convergencia del método.

Por otro lado, en [21] se extiende el alcance a clusters de computadoras, con resultados comparables a los encontrados en nuestro trabajo. El método analizado es nuevamente Gauss-Seidel y se emplean técnicas de *frente de onda* (wavefront) que hacen uso del grafo de dependencias para planificar la ejecución de distintas tareas, aunque en este caso la estrategia de planificación utilizada es estática. Esto último, sumado a preocupaciones propias de la arquitectura utilizada como los mecanismos de pasaje de mensajes o la importancia de reducir la comunicación entre nodos del cluster, modifica el tratamiento de problemas como el balanceo de cargas.

Finalmente, también podemos mencionar a [22], donde se estudian alteraciones a la estructura PRISM para posibilitar procesamiento distribuido a gran escala, también conocido

como “grid computing”. Como es de esperar, las dificultades aquí no se limitan al procesamiento numérico sino que abarcan problemas más generales de administración y seguridad como el control y monitoreo de procesos en estaciones remotas, la autenticación de clientes y el envío de cantidades masivas de datos. El trabajo ofrece una descripción de alto nivel de las tecnologías utilizadas para enfrentar estos desafíos.

Un nueva manera de explotar el paralelismo que ha surgido recientemente es la computación de propósito general utilizando placas de video tradicionalmente empleadas para la aceleración del procesamiento gráfico. La creación y distribución por parte de los fabricantes de herramientas que permiten acceder directamente a los recursos provistos por estas placas han habilitado el acceso a esta nueva tecnología, conocida como GPGPU. La mayor ventaja reside en la gran cantidad de hilos que pueden ser ejecutados de manera simultánea, lo que ha sido explotado exitosamente en la aceleración de algoritmos de calculo numérico.

Resulta interesante evaluar la aplicabilidad de estas tecnologías al problema de verificación simbólica de modelos. El principal obstáculo a tal fin lo constituye la arquitectura altamente no convencional de este tipo de dispositivos, que pueden clasificarse como procesadores SIMD o “única instrucción, múltiples datos” (single instruction, multiple data). Si bien esto permite explotar el paralelismo de datos logrando un alto nivel de “throughput”, no se adapta del todo bien a las irregularidades de estructuras de almacenamiento simbólico como los BDD. Por otro lado, el uso de métodos simbólicos reduce el consumo de memoria, un recurso típicamente escaso en estos sistemas. Luego, una aproximación efectiva al problema requiere establecer un compromiso adecuado entre estos factores y otros como el tiempo de transferencia de datos.

Los resultados obtenidos muestran que una de las consideraciones más importantes para garantizar un buen rendimiento en los algoritmos paralelos es el uso correcto de la memoria caché. Tácticas como el uso de matrices incrustadas, la división en bloques y la planificación apropiada de tareas aumentan la localidad en los accesos a memoria y son esenciales para lograr algoritmos eficientes, a pesar de diluir la pureza de las estructuras simbólicas.

Estas tácticas ofrecen parámetros que deben ser configurados según las características como tamaño de caché o latencia de lectura y/o escritura. En nuestro caso, este ajuste se realizo de forma manual teniendo en cuenta las limitaciones de los algoritmos utilizados y la arquitectura objetivo, lo que no asegura que el rendimiento obtenido sea el mejor posible o que se mantenga estable al migrar a otras arquitecturas.

Dicho esto, una opción que vale la pena explorar es el uso para la verificación de algoritmos “cache-aware” o “cache-oblivious”. En los primeros los parámetros de configuración se hacen explícitos lo que facilita el proceso de optimización, mientras que los segundos están diseñados para aprovechar al máximo la memoria caché sin tener que conocer los valores exactos de estos parámetros. Sospechamos que la paralelización puede beneficiarse del uso de este tipo de algoritmos, aún cuando la versión secuencial de los mismos no exhiba mayores diferencias en rendimiento.

La introducción del paralelismo también pone en evidencia nuevos cuellos de botella en el proceso de verificación, que limitan las ganancias obtenidas tal como lo describe la ley de Amhdal. Uno de estos cuellos de botella es el análisis de alcanzabilidad previo a la verificación. El uso de BDDs fuerza a implementar los algoritmos de alcanzabilidad utilizando métodos de punto fijo, lo que provoca cálculos redundantes y disminuye la eficiencia.

Una vía de solución es extender las técnicas de paralelización presentadas en este trabajo a estos algoritmos, aunque dado que el precálculo no es un proceso cuantitativo sino cualitativo (no utiliza los valores específicos de probabilidad), también es probable que existan otras aproximaciones usando para este paso estrategias alternativas de almacenamiento.

Bibliografía

[1] Sheldon Akers. *Binary Decision Diagrams*, IEEE Transactions on Computers, C-27(6), pp. 509-516, 1978.

[2] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, Robert Brayton. *Verifying Continuous Time Markov Chains*, Proc. 8th International Conference on Computer Aided Verification (CAV '96), volume 1102 of LNCS, pp. 269-276, Springer, 1996.

[3] Iris Bahar, Erica Frohm, Charles Gaona, Gary Hachtel, Enrico Macii, Abelardo Pardo, Fabio Somenzi. *Algebraic Decision Diagrams and their Applications*, Proc. International Conference on Computer-Aided Design (ICCAD '93), pp. 188-191, 1993.

[4] Christel Baier. *On Algorithmic Verification Methods for Probabilistic Systems*, Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[5] Christel Baier, Joost-Pieter Katoen. *Principles of Model Checking*, MIT Press, 2008.

[6] Christel Baier, Joost Pieter Katoen, Holger Hermanns. *Approximate Symbolic Model Checking of Continuous-Time Markov Chains*, Proc. 10th International Conference on Concurrency Theory (CONCUR '99), volume 1664 of LNCS, pp. 146-161, Springer, 1999.

[7] Christel Baier, Marta Kwiatkowska. *Model Checking for a Probabilistic Branching Time Logic with Fairness*, Distributed Computing, 11(3), pp. 125-155, 1998.

[8] Randal Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35(8), pp. 677-691, 1986.

[9] Gianfranco Ciardo, Marco Tilgner. *On the use of Kronecker Operators for the Solution of Generalized Stocastic Petri Nets*, ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.

[10] Edmund Clarke, Masahiro Fujita, Patrick McGeer, Kenneth McMillan, Jerry Yang, Xudong Zhao. *Multi-Terminal Binary Decision Diagrams: an Efficient Data Structure for Matrix Representation*, Proc. International Workshop on Logic Synthesis (IWLS '93), pp. 1-15, 1993.

[11] Pedro D'Argenio, Bertrand Jeannet, Henrik Jensen, Kim Larsen. *Reachability Analysis of Probabilistic Systems by Succesive Refinements*, Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Peformance Modelling and Verification (PAPM/PROBMIV '01), volume 2165 of LNCS, pp. 39-56, Springer, 2001.

[12] Hans Hansson, Bengt Jonsson. *A Logic for Reasoning about Time and Probability*, Formal Aspects of Computing, 6(5), pp. 512-535, 1994.

[13] Leen Helmink, M Sellink, Frits Vaandrager. *Proof-checking a Data Link Protocol*, Proc. International Workshop on Types for Proofs and Programs (TYPES '93), volume 806 of LNCS, pp. 127-165, Springer, 1994.

[14] Oliver Ibe, Kishor Trivedi. *Stochastic Petri Net Models of Polling systems*, IEEE Journal on Selected Areas in Communications, 8(9), pp. 1649-1657, 1990.

[15] John Kemeny, James Laurie Snell, Anthony Knapp. *Denumerable Markov Chains*, D. Van Nostrand Company, 1966.

[16] Marta Kwiatkowska, Gethin Norman, David Parker. *Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach*, International Journal on Software Tools for Technology Transfer (STTT), 6(2), pp. 128-142, 2004.

[17] Marta Kwiatkowska, David Parker, Yi Zhang, Rashid Mehmood. *Dual-processor Parallelisation of Symbolic Probabilistic Model Checking*, Proc. 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04), pp. 123-130. IEEE Computer Society Press, 2004.

[18] C. Y. Lee. *Representation of Switching Circuits by Binary-Decision Programs*, Bell Systems Technical Journal 38, pp. 985-999, 1959.

[19] David Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*, PhD thesis, University of Birmingham, 2002.

[20] PRISM Web Site, *www.prismmodelchecker.org*.

[21] Yi Zhang, David Parker, Marta Kwiatkowska. *A Wavefront Parallelisation of CTMC Solution using MTBDDs*, Proc. International Conference on Dependable Systems and Networks (DSN '05), pp. 732-742, IEEE Computer Society Press, 2005.

[22] Yi Zhang, David Parker, Marta Kwiatkowska. *Grid-enabled Probabilistic Model Checking with PRISM*, Proc. 4th All Hands Meeting Workshop (AHM '05), 2005.