Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Resolución en paralelo para lógicas híbridas

Diego Martin Rubinstein
drubin@dc.uba.ar

Director
Dr. Daniel Gorín

Tesis para acceder al grado de
Licenciado en Ciencias de la Computación

29 de abril de 2011

Resumen

Desde el surgimiento mismo de las primeras computadoras electrónicas, se ha asistido a un incremento sostenido, año a año, en la velocidad de cómputo de los procesadores. Esta tendencia histórica se ha detenido en los últimos años y no se esperan avances significativos en el corto plazo. La carrera actual es por agregar más unidades de procesamiento en un mismo chip y el desafío está ahora en cómo aprovechar esta forma de procesamiento paralelo para acelerar cómputos.

En este trabajo investigamos el problema de cómo paralelizar un algoritmo para el problema de satisfacibilidad en lógicas híbridas de forma de aprovechar la ejecución en arquitecturas de multiprocesador. El algoritmo está basado en un cálculo similar al de resolución para lógica clásica; se trata, por lo tanto, de un algoritmo de saturación con estrategias de eliminación de redundancia. Proponemos una estrategia concreta de paralelización que implementamos como una extensión al prototipo HyLoRes, realizamos pruebas de performance y reportamos resultados preliminares.

Índice general

Capítulo 1

Lógicas modales e híbridas

1.1. Lógicas modales

Originalmente la lógica modal fue concebida como la lógica que trata con enunciados afectados por las modalidades 'posiblemente' y 'necesariamente'. Esta noción se remonta a los tiempos de Aristóteles, sin embargo a través de los años ese concepto fue evolucionando y esas dos modalidades pasaron a formar parte de una amplia variedad de modalidades. Ahora bien que es una modalidad? Podemos definir modalidad como cualquier palabra o frase que puede ser aplicada a un enunciado S para crear un nuevo enunciado que hace una aseveración con respecto al *modo de verdad de S*. De esta manera una modalidad puede indicar cuándo, dónde o cómo S es verdadero o bajo qué circunstancias S es verdadero.

Uno de los responsables de esta evolución es C. I. Lewis cuando publica su trabajo *Survey of Symbolic Logic* [Lewis, 1918] ubicando a las lógicas modales como disciplina matemática. En sus trabajos, Lewis extiende la lógica proposicional agregando el *operador modal* $\Diamond$, que se interpreta como "es posible que" y con él define $\mapsto$, la *implicación estricta*: $\varphi \mapsto \psi \equiv \neg\Diamond(\varphi \wedge \neg\psi)$; esta última fórmula se debe leer como "no es posible (en ningún contexto imaginable) que simultáneamente φ sea verdadero y $\neg\psi$ no lo sea". Usando este lenguaje podemos decir "si llueve, Juan no va a trabajar; pero como Juan sí va a trabajar, entonces no llueve" utilizando $\mapsto$, $((\mathsf{llueve} \rightarrow \neg\mathsf{trabaja}_{\mathsf{Juan}}) \wedge \mathsf{trabaja}_{\mathsf{Juan}}) \mapsto \neg\mathsf{llueve}$, y estaremos afirmando que éste es un hecho que no depende de circunstancias particulares.

$\Diamond$ es un operador modal ya que asigna un *modo* a un valor de verdad. Si $\Diamond$ toma el modo de *posibilidad*, lo usamos para diferenciar lo que *es verdadero* (o falso) de lo que *podría ser verdadero* (o falso). Tomemos dos proposiciones: nublado y llueve, y veamos ejemplos de lo que podemos decir al asociar este modo con dicho operador:

Fórmula	Significado
nublado	Está nublado en este momento
llueve	Está lloviendo en este momento
$\Diamond$nublado	Podría estar nublado en este momento
$\neg\Diamond$llueve	No podría llover en este momento
$\neg$nublado $\wedge$ llueve	En este momento llueve aunque no está nublado
$\neg\Diamond(\neg$nublado $\wedge$ llueve)	No podría suceder que ahora esté lloviendo aunque no esté nublado

En este ejemplo vemos dos casos de la forma "no es posible que suceda X". Intuitivamente, esto es lo mismo que decir "es necesario que no suceda X". La *necesidad* es una modalidad que se suele notar con el operador $\Box$. Como vimos, existe una relación muy fuerte entre *posibilidad* y *necesidad*: $\neg\Diamond\neg\varphi \equiv \Box\varphi$. Cuando dos operadores modales cumplen con esta propiedad, se dice que uno es el *dual* del otro. Tradicionalmente, se asume que si $\Diamond$ es una modalidad cualquiera, $\Box$ es su dual (y viceversa).

A partir de los trabajos de Lewis, surgió el interés de buscar nuevas extensiones *modales* de la lógica proposicional. Se investigaron, de esta forma, las axiomatizaciones de conceptos tales como *obligación*, *creencia*, *conocimiento*, etc. Lo que, en general, estos primeros lógicos modales hicieron fue capturar de una manera puramente sintáctica conceptos que hasta ese momento pertenecían al dominio de la intuición.

Un trabajo que merece un comentario aparte es el que realizó Arthur Prior [Prior, 1957; Prior, 1967] a principios de la década de 1950 con una familia especial de lógicas modales: las llamadas *lógicas temporales* (*temporal logics* o *tense logics*). Prior intentaba representar en una lógica la forma en la cual tratamos las relaciones temporales en los lenguajes naturales (de ahí su nombre original de *tense logics*); para ello introduce la ubicación temporal como modo. En la lógica temporal básica, Prior utiliza la modalidad F para referirse a algún instante indeterminado en el futuro, y la modalidad P para hacerlo sobre algún instante del pasado. Los operadores duales de F y P son G y H respectivamente, y permiten predicar sobre todos los instantes del futuro o del pasado. En este lenguaje podemos, por ejemplo, escribir la frase "siempre que llovió, paró" como $H(\mathsf{llueve} \rightarrow F(\neg \mathsf{llueve}))$. La fórmula $\mathsf{llueve} \rightarrow F(\neg \mathsf{llueve})$ se interpreta como "si llueve (en este momento), entonces en algún momento del futuro no lloverá"; al precederla con el operador H estamos pidiendo que la fórmula sea verdadera en todo instante del pasado.

Prior encontró una importante limitación en la lógica temporal básica: en ella no se pueden expresar ideas tales como "ayer fui al cine" o "si en cinco minutos no llega, me voy". Como veremos más adelante, detrás de esta idea se encuentra una importante limitación de las lógicas modales estándar. En sus últimos años, Prior investigó distintas maneras de enriquecer las lógicas temporales para resolver estos problemas. Los resultados a los que llegó, redescubiertos recientemente, anticipan de alguna manera ideas con las que se está empezando a trabajar hoy en día, casi cuarenta años después.

La lógica temporal es también un ejemplo de lógica multi-modal. Si bien F y P están íntimamente relacionados (e.g. $\varphi \rightarrow \neg F \neg P \varphi$ es verdadero, para todo φ), no es posible escribir uno en función del otro. Ambos deben incluirse como operadores primitivos de la lógica temporal; a partir de ellos es posible derivar G y H. De aquí en más utilizaremos un lenguaje uniforme para notar a todas las lógicas (multi-)modales.

Definición 1.1. Dados PROP un conjunto numerable de símbolos de proposición, y REL un conjunto numerable de símbolos de relación, el conjunto $\mathcal{M}$ de fórmulas bien formadas de la lógica (multi) modal básica definidas sobre PROP y REL se define inductivamente como:

$$\mathcal{M} ::= p \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \langle r \rangle \varphi$$

donde $p \in \mathsf{PROP}$, $r \in \mathsf{REL}$ y $\varphi, \varphi' \in \mathcal{M}$. El operador $[r]$ se define como $[r]\varphi \equiv \neg\langle r \rangle \neg\varphi$. El resto de los operadores lógicos habituales se definen de la forma usual. En los casos en que REL sea un conjunto unitario, usaremos $\Diamond$ y $\Box$ como operadores modales.

Como mencionamos previamente, por muchos años el enfoque que se le dio al estudio de las lógicas modales fue puramente sintáctico, es decir dado un lenguaje se trataba de ver si podía expresarse éste con una lógica modal. Este enfoque parecía estar frenando el desarrollo del área y el interés por las lógicas modales se estaba extinguiendo. Hasta que a principios de la década de 1960 diversos trabajos, entre los que sobresalen los de Samuel Kripke [Kripke, 1959; Kripke, 1963a; Kripke, 1963b], le dieron un enfoque totalmente novedoso al descubrir que las lógicas modales pueden ser interpretadas en términos de estructuras matemáticas precisas. Por lo tanto a partir de ese momento se introdujo una perspectiva semántica que permite utilizar las lógicas modales como herramientas para hablar sobre *estructuras* o *modelos*

Estas estructuras tomadas como modelos de una lógica modal se conocen hoy en día como *modelos de Kripke*. Desde un punto de vista computacional, un modelo de Kripke es un multigrafo dirigido con nodos etiquetados por conjuntos de símbolos proposicionales. Intuitivamente, cada nodo del grafo representa un *contexto* donde es posible evaluar una fórmula modal, las relaciones de accesibilidad entre puntos están asociadas a las distintas modalidades y la etiqueta de cada punto es una *valuación* que permite determinar el valor de verdad de las proposiciones en dicho contexto.

Definición 1.2 (Modelo de Kripke)**.** Un modelo de Kripke es una estructura $M = \langle W, \{R_i\}, V\rangle$ donde

$$\begin{aligned} W &\quad \text{es un conjunto no vacío} \\ R_i \subseteq W \times W &\quad \text{es una relación binaria para cada } R_i \in \mathsf{REL} \\ V(p_i) \subseteq W &\quad \text{para cada } p_i \in \mathsf{PROP} \end{aligned}$$

Las lógicas modales hoy en día se utilizan en diversos campos y aplicaciones. Son utilizadas por ejemplo en las llamadas *Bases de Conocimiento* que a su vez se utilizan en diversos campos como *Teoría de Juegos* e *Inteligencia Artificial*. Una base de conocimiento contienen una serie de datos usualmente en forma de reglas, a partir de los cuales es posible *razonar* nuevos datos o evaluar la validez de un razonamiento. Otra aplicación surgió en las últimas décadas con el aumento en la complejidad del diseño de los semiconductores, que transformó en vital la *verificación formal* de estos. Para este proceso es necesario definir un lenguaje formal de especificación, la lógica temporal es uno de los lenguajes más utilizados para ello.

1.2. Lógicas híbridas

Las lógicas modales nos permiten modelar estructuras, sin embargo tienen la falencia de no poder referenciar elementos específicos de un modelo. Esto hace por ejemplo que en una lógica temporal no se pueda referir a instantes particulares como "ayer" o "en cinco minutos". Tratando de suplir esta falta de expresividad de las lógicas modales es que surgen las lógicas híbridas.

Las lógicas híbridas en su forma más simple se obtienen al extender las lógicas modales agregando símbolos que dan nombre a estados particulares en los modelos. A estos nuevos símbolos se los llama *nominales* y se los combina con las variables proposicionales libremente en las fórmulas. Es importante aclarar que un nominal es verdadero en único punto del modelo, esto hace por ejemplo que $\Diamond(i \wedge p) \wedge \Diamond(i \wedge q) \rightarrow \Diamond(p \wedge q)$ sea una tautología en un lenguaje híbrido si i es un nominal, mientras que podría ser falso si i fuera una variable proposicional común.

Además de los nominales se puede agregar aún más expresividad introduciendo el operador de satisfacción @. Este operador se combina con los nominales y permite saltar a un punto del modelo. Por ejemplo la fórmula $@_i\varphi$ nos posiciona en el punto de evaluación del estado identificado por el nominal i y evalúa φ ahí. Otro operador de algunas lógicas híbridas es el operador $\downarrow$. Este operador permite referirse a un punto del modelo sin necesidad de darle un nombre de antemano. En otras palabras nos permite decir "estoy posicionado en cierto punto (al cual llegue utilizando los operadores mencionado anteriormente) y a ese punto lo llamo x".

Definición 1.3. Dados PROP un conjunto numerable de símbolos de proposición, NOM un conjunto numerable de nominales y REL un conjunto numerable de símbolos de relación, decimos que $\mathsf{ATOM} = \mathsf{PROP} \cup \mathsf{NOM}$ y definimos inductivamente el conjunto $\mathcal{H}$ de fórmulas bien formadas de la lógica híbrida mínima definida sobre ATOM y REL como:

$$\mathcal{H} ::= a \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \langle r\rangle\varphi$$

donde $a \in \mathsf{ATOM}$, $r \in \mathsf{REL}$ y $\varphi, \varphi' \in \mathcal{H}$. El resto de los operadores lógicos se definen de la manera usual.

La semántica de la lógica híbrida mínima está dada por un tipo particular de modelo de Kripke en el cual la valuación garantiza que cada nominal vale en uno y sólo uno de los puntos del modelo. Llamamos *modelos híbridos* a estas estructuras.

Definición 1.4 (Modelo híbrido)**.** Un modelo híbrido es una estructura $M = \langle W, \{R_i\}, V\rangle$ donde

$$\begin{aligned} W &\quad \text{es un conjunto no vacío} \\ R_i \subseteq W \times W &\quad \text{es una relación binaria para cada } R_i \in \mathsf{REL} \\ V(p_i) \subseteq W &\quad \text{para cada } p_i \in \mathsf{PROP} \\ V(n_i) = \{w_i\} \subseteq W &\quad \text{para cada } n_i \in \mathsf{NOM} \end{aligned}$$

1.2.1. Satisfacibilidad y validez

Un problema interesante a resolver en una lógica es el de *satisfacibilidad*. Este consiste en dado una fórmula φ encontrar un modelo M que satisfaga a φ. Este problema es dual al problema de validez de una fórmula; ver si todo modelo M satisface a φ, ya que φ es válida si y sólo si $\neg\varphi$ no es satisfacible.

El problema de satisfacibilidad en la lógica modal básica se ubica dentro de los problemas PSPACE completos. Es decir que puede ser resuelto por un programa utilizando una cantidad polinomial (en función del tamaño de la entrada) de memoria. Se ha demostrado que la complejidad también es PSPACE completo para la lógica híbrida básica. Sin embargo la complejidad en una lógica híbrida más expresiva como la lógica $H(@,\downarrow)$[1] se vuelve indecidible.

Existen varios métodos para resolver el problema de satisfacibilidad (o validez) de una fórmula. Un requerimiento fundamental de estos métodos es que sean *consistentes*, es decir que siempre que den una respuesta esta sea correcta. También es deseable que además sean *completos*, cumpliéndose esta ultima condición cuando pueden dar respuesta a cualquier instancia del problema. Una manera simple de resolver satisfacibilidad es realizar traducciones a otros formalismos en los cuales ya exista un método completo y consistente. Por ejemplo es posible traducir una lógica modal básica así como también $H(@)$ a una lógica de primer orden con esas propiedades.

El método más utilizado para construir demostradores automáticos en lógicas de primer orden es el método de *resolución*. En el próximo capítulo haremos una descripción detallada de este método ya que sobre este se basa el trabajo hecho en esta tesis.

[1] lógica que se obtiene al agregar los operadores @ y $\downarrow$ a la lógica mínima

Capítulo 2

El método de resolución

2.1. Resolución en lógica clásica

El método de Resolución en lógica proposicional permite construir un algoritmo consistente y completo para determinar si una fórmula es satisfacible. Este método trabaja sobre fórmulas proposicionales que estén en *forma clausal*, es decir una conjunción de cláusulas dónde una *cláusula* es o bien un *literal* o bien una disyunción de *literales*, y un *literal* es un átomo o la negación de un átomo. Por ejemplo si p y q son variables proposicionales, p y $\neg p$ son literales y p, $\neg p$ y $p \vee q$ son cláusulas. Alternativamente una fórmula en forma clausal se puede expresar como un conjunto de cláusulas. Existe un procedimiento polinomial para llevar cualquier fórmula proposicional a forma clausal preservando satisfacibilidad, con lo cual el método es de uso general.

El método consiste en aplicar la siguiente regla (llamada de resolución) sobre el conjunto de cláusulas:

$$\frac{p \vee C_1, \quad \neg p \vee C_2}{C_1 \vee C_2}$$

La regla se interpreta de la siguiente manera: si se encuentran cláusulas como las de la premisa, se debe agregar una cláusula como la que indica la conclusión. Un conjunto D está clausurado por la regla de resolución si cuando $p \vee C_1$ pertenece a D y $\neg p \vee C_2$ pertenece a D, entonces $C_1 \vee C_2$ pertenece a D, donde C_1 y C_2 son cláusulas y por definición $C_1 \vee C_2$ también. Resolución en lógica establece que sea D un conjunto clausurado por saturación, D es satisfacible si y sólo si el conjunto vacío pertenece a D. Luego un algoritmo basado en resolución consiste en obtener la clausura del conjunto inicial de cláusulas y verificar la pertenencia de la cláusula vacía. En la sección 2.3 veremos un algoritmo para obtener esta clausura.

Si uno quisiera de aplicar el método de resolución sobre lógica de primer orden una posibilidad sería instanciar un conjunto de cláusulas de primer orden en todas las posibles cláusulas cerradas, es decir cláusulas con fórmulas sin variables libres. Para ello primero es necesario eliminar los cuantificadores universales, lo que es posible mediante un proceso llamado *skolemnización*. Luego por el teorema de Herbrand, el primer conjunto es satisfacible si y sólo si el segundo conjunto con cláusulas cerradas también lo es. Sin embargo como algoritmo resulta inviable porque el segundo conjunto puede ser infinito. Utilizando *unificación* es posible instanciar a medida que se necesite. Esta última es la idea del método de resolución para primer orden propuesto por Robinson [Robinson, 1965] a mediados de la década de 1960. Para saturar el conjunto de cláusulas se utilizan dos reglas, una regla llamada *resolución* y otra denominada *factorización*.

$$\text{(RES)} \quad \frac{C_1 \cup \{\varphi\} \quad C_2 \cup \{\neg\psi\}}{(C_1 \cup C_2)\sigma}$$

$$\text{(FACT)} \quad \frac{C \cup \{\varphi, \psi\}}{(C \cup \{\varphi\})\sigma}$$

donde σ es el unificador más general de los átomos φ y ψ. Es con este unificador con el cual se logra la instanciación gradual de las variables.

Si uno aplicara las reglas de resolución libremente sobre todas las fórmulas de todas cláusulas, el conjunto de cláusulas generadas crecería de manera inmanejable desde el punto de vista computacional. Afortunadamente esto no es necesario, se pueden aplicar las reglas usando un orden entre fórmulas y funciones de selección. Una *función de selección* es una función que asigna a cada cláusula C un conjunto vacío o un conjunto unitario con un literal negativo de C. Dada una función de selección S y una relación de orden entre literales $\succ$, el cálculo de resolución para lógica de primer orden con orden y selección se define agregando a la regla de factorización la regla:

$$\text{(RES–OS)} \quad \frac{C_1 \cup \{\varphi\} \quad C_2 \cup \{\neg\psi\}}{(C_1 \cup C_2)\sigma}$$

tal que

1. σ es el unificador más general de los átomos φ y ψ
2. $S(C_1 \cup \{\varphi\}) = \{\}$ y $\varphi\sigma \succ \varphi'$ para todo $\varphi' \in C_1\sigma$
3. $S(C_2 \cup \{\neg\psi\}) = \{\neg\psi\}$ o bien $S(C_2 \cup \{\neg\psi\}) = \{\}$ y $\neg\psi\sigma \succ \psi'$ para todo $\psi' \in C_2\sigma$.

Se puede demostrar que el cálculo de resolución con orden y selección (que incluye la regla de factorización) es refutacionalmente completo para la lógica de primer orden si cumple ciertas condiciones [Bachmair and Ganzinger, 2001].

2.2. Resolución en lógica híbrida

En [Areces and Gorín., 2008] se propone un cálculo basado en resolución aplicado a la lógica híbrida utilizando orden y selección.

Al igual que el cálculo de resolución para lógica de primer orden, el cálculo de resolución híbrido trabaja con conjuntos de *cláusulas*. Aquí también, una cláusula representa una disyunción, pero a diferencia de lo que sucede en los cálculos de resolución mencionados anteriormente, no hay ninguna restricción en cuanto a la forma clausal en que se deben encontrar las fórmulas; son las propias reglas del cálculo las que se encargan de armar cláusulas con fórmulas cada vez más simples. En la Figura 2.1 podemos ver la reglas que se aplican para saturar el conjunto inicial. Podemos agrupar estas reglas de acuerdo a la función que cumplen. Por ejemplo las reglas ($\wedge$), ($\vee$) y (@) se encargan de simplificar las fórmulas. La regla ($\langle R\rangle$) también simplifica en cierta forma una fórmula; podemos pensarla como una suerte de skolemnización mediante la cual se le asigna explícitamente un nombre a un contexto. En la figura mencionada anteriormente podemos ver que este nuevo contexto se lo identifica con la siguiente notación: $\epsilon\langle l, r, \varphi\rangle$, es decir se introduce un nuevo tipo de nominal que es función de otro nominal l, una relación r y una fórmula φ.

La regla (RES) es el equivalente de la regla de resolución proposicional, mientras que la regla ($[R]$) se encarga de propagar información entre distintos contextos. Implícitamente, esta regla está codificando una unificación no trivial y un paso de resolución. Finalmente, las reglas (REF), (SYM), ($\text{SYM}^{\neg}$) , se corresponden con el *paquete* estándar de reglas para manejar resolución en lógica de primer orden con igualdad [Bachmair and Ganzinger, 1998]. Las reglas ($\text{PAR}^{@_{\text{no-}\Diamond}}$), ($\text{PAR}^{@\Diamond}_{\epsilon}$) y ($\text{PAR}^{@\Diamond}_{\text{no-}\epsilon}$) son especializaciones de la regla de paramodulación que maneja también igualdades.

Las condiciones de orden y selección hacen que sólo sea necesario que una única fórmula por cláusula participe de una inferencia. A esta fórmula se la llama *fórmula distinguida*. En [Areces and Gorín., 2008] se demuestra por un lado que éste cálculo es refutacionalmente completo, es decir que sea φ la fórmula de entrada, si φ es insatisfacible entonces la saturación de φ contiene a la cláusula vacía. Pero más importante aún, se prueba que si $\varphi \in H(@)$ el cálculo es un método de decisión para satisfacibilidad. Es decir garantiza la terminación del cálculo.

RES	$\dfrac{C \cup \{@_l \neg p\} \quad D \cup \{@_l p\}}{C \cup D}$		REF	$\dfrac{C \cup \{@_l \neg l\}}{C}$	
SYM	$\dfrac{C \cup \{@_m l\}}{C \cup \{@_l m\}}$	†	$\text{SYM}^{\neg}$	$\dfrac{C \cup \{@_m \neg l\}}{C \cup \{@_l \neg m\}}$	†
$\text{PAR}^{@_{\text{no}}\Diamond}$	$\dfrac{C \cup \{@_l \varphi\} \quad D \cup \{@_l m\}}{C \cup D \cup \{@_m \varphi\}}$	‡			
$\text{PAR}_{\epsilon}^{@\Diamond}$	$\dfrac{C \cup \{@_l \langle r \rangle \epsilon \langle l, r, \varphi \rangle\} \quad D \cup \{@_l m\}}{\begin{array}{l} C \cup D \cup \{\quad @_m \langle r \rangle \epsilon \langle m, r, \varphi \rangle \quad\} \\ C \cup D \cup \{\quad @_{\epsilon \langle l,r,\varphi \rangle} \epsilon \langle m, r, \varphi \rangle \quad\} \end{array}}$	†	$\text{PAR}_{\text{no-}\epsilon}^{@\Diamond}$	$\dfrac{C \cup \{@_l \langle r \rangle n\} \quad D \cup \{@_l m\}}{C \cup D \cup @_m \langle r \rangle n}$	⋆
$\wedge$	$\dfrac{C \cup \{@_l (\varphi_1 \wedge \varphi_2)\}}{C \cup \{@_l \varphi_1\} \quad C \cup \{@_l \varphi_2\}}$		$\vee$	$\dfrac{C \cup \{@_l (\varphi_1 \vee \varphi_2)\}}{C \cup \{@_l \varphi_1, @_l \varphi_2\}}$	
$[r]$	$\dfrac{C \cup \{@_l [r] \varphi\} \quad D \cup \{@_l \langle r \rangle m\}}{C \cup D \cup \{@_m \varphi\}}$		$\langle r \rangle_{\epsilon}$	$\dfrac{C \cup \{@_l \langle r \rangle \varphi\}}{\begin{array}{l} C \cup \{\quad @_l \langle r \rangle \epsilon \langle l, r, \varphi \rangle \quad\} \\ C \cup \{\quad @_{\epsilon \langle l,r,\varphi \rangle} \varphi \quad\} \end{array}}$	*
$[r^{-1}]$	$\dfrac{C \cup \{@_l [r] \varphi\} \quad D \cup \{@_m \langle s \rangle l\}}{C \cup D \cup \{@_m \varphi\}}$	⋆⋆	@	$\dfrac{C \cup \{@_l @_m \varphi\}}{C \cup \{@_m \varphi\}}$	
$\downarrow$	$\dfrac{C \cup \{@_l \downarrow m.\varphi\}}{C \cup \{@_l \varphi(m/l)\}}$				

Condiciones

† $l \succ m$

‡ $l \succ m$, $\varphi \succ m$, $@_l \varphi \in \mathsf{SIMP}$ y $\varphi \neq \langle r \rangle n$, donde SIMP es el conjunto de fórmulas que no se pueden simplificar más aplicando reglas unarias.

⋆ $l \succ m$ y $n \neq \epsilon \langle l, r, \varphi \rangle$

∗ $\varphi \notin \mathsf{LAB}$, donde LAB es la unión del conjunto de nominales iniciales con el conjunto nominales generados por la aplicación de las reglas $\text{PAR}_{\epsilon}^{@\Diamond}$ y $\langle r \rangle_{\epsilon}$

⋆⋆ $r^{-1} = s$.

Condiciones Globales

- en $C \cup \{\psi\}$, ψ es tal que $S(C \cup \{\psi\}) = \emptyset$ y $\{\psi\} \succ C$, o bien $S(C \cup \{\psi\}) = \{\psi\}$
- en $D \cup \{\psi\}$, ψ es tal que $S(D \cup \{\psi\}) = \emptyset$ y $\{\psi\} \succ D$

Figura 2.1: Cálculo de Resolución $\mathbf{R}_{L\epsilon}^{S\succ}[\mathcal{H}^{NNF}(@, \downarrow, \Diamond^{-1})]$, para S una función de selección y $\succ$ un orden.

2.3. Implementación de un cálculo de saturación

El algoritmo más utilizado para saturar un conjunto de cláusulas es el "algoritmo de la cláusula dada"[Voronkov, 2001]. Este algoritmo construye de manera ordenada un conjunto saturado de

cláusulas a partir de un conjunto de cláusulas iniciales, garantizando que ninguna cláusula se demorará infinitamente en ser procesada. Las cláusulas se distribuyen en tres conjuntos de cláusulas distintos: *new*, *clauses* e *in-use*. En *new* se guardan todas las cláusulas que se generan como resultado de la aplicación de una regla; también es el conjunto donde se incluye la fórmula original al comenzar el cálculo. Este conjunto se puede considerar *de paso* ya que las cláusulas sólo permanecen en éste al solo efecto de ser simplificadas (e.g. eliminación de tautologías y contradicciones simples) y luego son almacenadas en *clauses* donde esperan por la aplicación de alguna regla. En *in-use* se guardan las cláusulas a las que ya se les ha intentado aplicar alguna regla al menos una vez.

El algoritmo consta de un ciclo principal donde cada iteración consiste en lo siguiente: primero se pasan a *clauses* todas las cláusulas que estén en *new*; se elige una *cláusula dada* de *clauses* y se intentan aplicar todas las reglas de saturación entre la cláusula dada y todas las cláusulas que estén en *in-use*; las cláusulas nuevas que se generen como resultado de la aplicación de las reglas se guardan en *new*; por último se agrega la *cláusula dada* a *in-use*. Este procedimiento se repite hasta que no queden más cláusulas en *clauses* y por lo tanto el conjunto *in-use* es el conjunto saturado.

Es conveniente que el conjunto *in-use* utilice alguna estructura de datos que soporte indexación, que dada una cláusula permita obtener de manera eficiente todas las cláusulas de *in-use* con las cuales se pueda aplicar alguna regla. Por ejemplo si se está saturando un conjunto con las reglas de la Figura 2.1 y se elige como cláusula dada una cuya fórmula distinguida sea $@_t \langle r \rangle s$, para poder aplicar la regla ($[R]$) habría que buscar en *in-use* todas las cláusulas que posean como fórmula distinguida una de la forma $@_t [r]\varphi$. Por lo tanto es necesario que *in-use* tenga varios índices que dado un nominal devuelva una lista de clausulas cuya fórmula distinguida sea apropiada para aplicar cada una de las reglas binarias.

Es posible utilizar el algoritmo de la cláusula dada para decidir si una fórmula φ es satisfacible. Basta con transformar la fórmula de entrada φ al conjunto $\{\{\varphi\}\}$, luego saturarlo y verificar si el conjunto saturado resultante contiene la cláusula vacía. Esto funciona por un lado si el conjunto de reglas forman parte de un cálculo refutacionalmente completo, que implica que el conjunto saturado contiene a la cláusula vacía si y sólo si φ es insatisfacible. Pero además es necesario que el conjunto saturado sea finito ya que de otra manera en caso de que φ sea satisfacible no terminaría nunca de saturar.

Obviamente si la fórmula inicial es insatisfacible no es necesario saturar totalmente el conjunto, se puede parar en el momento que la cláusula vacía es derivada. Tampoco es necesario agregar todas las cláusulas generadas a *clauses* o las procesadas a *in-use* ya que puede ser que la cláusula a agregar sea redundante. Una cláusula es redundante o está *subsumida* cuando no puede ser que la cláusula sea falsa si todas las demás cláusulas (aquellas que la subsumen) son verdaderas. Un ejemplo simple de subsunción es la inclusión estricta, por cuanto es claro que C subsume a D toda vez que $C \subset D$. La eliminación de clausulas redundantes es sumamente importante y necesario para evitar que el conjunto de cláusulas crezca de tal manera que se vuelva inmanejable desde el punto de vista computacional debido a su tamaño. Existen dos tipos de subsunción que se pueden aplicar al cálculo de resolución. El primero consiste es ver si una cláusula de las que ya se encuentran en *clauses* o en *in-use* subsume a una cláusula que se acaba de generar por la aplicación de una regla. Y de ser así no se agrega esta última a *clauses*. A este tipo de subsunción se la llama *hacia delante* o *forward*. El otro tipo de subsunción se llama *hacia atrás* o *backward* y consiste en verificar si una cláusula recién generada subsume a alguna cláusula que se encuentre en *clauses* o *in-use*. En caso de que así sea, se eliminan del conjunto correspondiente las clausulas subsumidas.

Una heurística habitual en resolución para tratar de derivar la cláusula vacía de manera más rápida, es intentar elegir como cláusula dada aquella fórmula que tenga más chance de poder derivar la cláusula vacía. Una manera de implementar esta heurística es ordenar las cláusulas en *clauses* utilizando una noción de complejidad de cláusula que involucra parámetros como tamaño de la cláusula, máxima profundidad modal, etc. La intuición detrás de esta implementación es que si se va a derivar la cláusula vacía, esta va a ser derivada a partir de una cláusula más simple. Entonces se utiliza una especie de algoritmo goloso que trata de procesar primero las cláusulas más simples. De todas maneras cada n iteraciones se debe elegir la cláusula más vieja del conjunto

para poder garantizar que todas las cláusulas sean eventualmente procesadas.

En el Algoritmo 1 se muestra el pseudocódigo de un algoritmo de la cláusula dada aplicado a resolución. El procedimiento aplicarSubsunciones realiza la eliminación de cláusulas redundantes mencionada anteriormente y modifica los conjuntos que recibe como argumento en consecuencia.

Algoritmo 1 Algoritmo de la cláusula dada utilizado para resolución

```
input: init: conjunto de cláusulas
var: new, clauses, in-use: conjunto de cláusulas
var: given: cláusula
clauses := {}; in-use = {}; new:= init
if {} ∈ new
   return ''insatisfacible''
end if
clauses := new
while clauses ≠ {}
      given := select(clauses)
      clauses := clauses \{given}
      new := applyRules(given,in-use)
      if {} ∈ new
         return ''insatisfacible''
      end if
aplicarSubsunciones(in-use, {given}, clauses, new)
end while
return ''satisfacible''
```

Capítulo 3

HyLoRes, una implementación de resolución

HyLoRes es una implementación en Haskell del método de resolución para la lógica híbrida $H(@, \downarrow, \diamond^{-1})$. Esta implementación está basada en el cálculo propuesto en [Areces and Gorín., 2008]. En este capítulo explicaremos de manera detallada como funciona HyLoRes. Esto servirá a modo de introducción del próximo capítulo donde contaremos como modificamos esta implementación para aprovechar un ambiente multiprocesador, objetivo principal de esta tesis.

La estructura general de HyLoRes es la típica de cualquier algoritmo basado en resolución: recibe como entrada a una fórmula φ que es transformada a un conjunto de cláusulas, el cual es luego saturado utilizando el algoritmo de la cláusula dada y especializaciones de las reglas presentadas en la Figura 2.1. Como explicamos en el capítulo previo, el conjunto new es un conjunto de paso, en éste se agregan las cláusulas generadas al aplicar las reglas así como también las cláusulas iniciales. Antes de agregar una cláusula a new, esta es normalizada. En este proceso de normalización se aplican algunas simplificaciones de bajo costo así como también se calcula la fórmula distinguida. Clauses se implementa utilizando una estructura que posee colas de cláusulas ordenadas de acuerdo a la complejidad de estas, luego, al momento de elegir la cláusula dada, se elige la de menor complejidad. Para determinar esta complejidad se tienen en cuenta distintos atributos de la cláusula: cantidad de fórmulas, profundidad de la fórmula distinguida, etc. Los atributos a tener en cuenta son configurables por parámetro y determinan lo que se denomina *función de selección*. También *clauses* mantiene la información del tiempo de permanencia de cada cláusula ya que de esta manera se puede garantizar que todas las cláusulas son procesadas, eligiendo cada cierta cantidad de iteraciones la cláusula más vieja como cláusula dada.

Como mencionamos en el capítulo anterior es importante eliminar cláusulas redundantes (subsumidas) para evitar que el conjunto saturado crezca en cantidad de cláusulas.

3.1. Eliminación de redundancia

En HyLoRes se realiza primero un chequeo de subsunción *hacia adelante* y *hacia atrás* de manera *local*. Decimos que es local ya que sólo son necesarias las cláusulas que participan en la aplicación de la regla y por lo tanto su costo computacional es bajo. Luego se verifica subsunción de manera *global*, donde interactúan una cláusula nueva y el conjunto de todas las cláusulas ya existentes. El costo del chequeo global es mucho mayor y se utilizan estructuras especializadas para tratar de hacerlo más eficiente como veremos más adelante.

3.1.1. Subsunción local

Un primer chequeo de subsunción local se realiza luego de aplicar una regla del cálculo. Primero se analiza subsunción hacia adelante entre las premisas y el o los consecuentes y si estos

fueran subsumidos entonces no son agregados a *new* y se continua con la próxima iteración del ciclo principal. En caso contrario se realiza un chequeo se subsunción hacia atrás entre el o los consecuentes y las premisas de la regla. En caso de que alguna premisa fuera subsumida es eliminada del conjunto *in-use*. La eliminación de cláusulas redundantes para estos casos está determinada por la implementación propia de la aplicación de cada regla. Ésta es la razón por la cual HyLoRes posee reglas especializadas de paramodulación de acuerdo a la forma de la fórmula distinguida de la premisa principal, ya que de esta manera se pueden determinar las subsunciones para cada caso específico como podemos ver en la Figura 3.1. Vale aclarar que las reglas REF, SYM, SYM$^{\neg}$, @ no fueron incluidas ya que HyLoRes las aplica antes de pasar una fórmula a *new* y no producen subsunciones.

3.1.2. Subsunción global

Luego de analizar subsunción local, se realiza un chequeo de subsunción global antes de que las cláusulas de *new* sean agregadas a *clauses*. Primero se chequea subsunción hacia adelante, determinando si alguna cláusula de *clauses* o *in-use* subsume a alguna cláusula nueva. Si esto ocurre la cláusula nueva es descartada. Si no lo es, entonces se analiza subsunción hacia atrás, buscando cláusulas que se encuentren en *in-use* y *clauses* que sean subsumidas por alguna de las cláusulas nuevas. Si existiese alguna, entonces es removida del conjunto correspondiente.

HyLoRes verifica subsunción global utilizando la inclusión estricta de conjuntos. Para ello posee dos estructuras que representan el conjunto de cláusulas existentes (las cláusulas de *in-use* más las cláusulas de *clauses*) de dos maneras distintas.

Una primera es una estructura similar a un trie que es utilizada para verificar subsunción global hacia adelante. En cada nodo del trie se guarda una representación numérica de una fórmula y un puntero *next* a la próxima fórmula de la cláusula, quedando esta última representada por una rama. Además cada nodo tiene un puntero *left* que permite navegar hacia fórmulas cuya representación numérica es menor y otro puntero *right* para navegar hacia fórmulas mayores

La función de inserción la podemos ver en el Algoritmo 2. Esta función recibe a la cláusula representada como una lista de las fórmulas que la componen. La inserción en el trie solo se realiza si la cláusula no es subsumida por alguna de las cláusulas que ya estaban en éste.

Algoritmo 2 Algoritmo de inserción de una cláusula en el trie

```
function insertar
input: cl: Lista de fórmulas
input: trie: SubsumptionTrie

if trie == Nulo
   for i = 1 to length(cl)
       trie.next := agregarNodo(Nodo(cl[i], Nulo, Nulo))
       trie = trie.next
   end for
else
    case compare (trie.val):
         =  :  if trie.next != Nulo
                  trie.next := insertar(tail(cl), trie.next)
               end if
         <  :   trie.left := insertar (cl, trie.left)

         >  :   trie.right := insertar(cl, trie.right)

end if
return trie
```

Regla	Fórmulas	Subsunciones
$\text{PAR}^{=}$	$\dfrac{Cl_1 \cup \{@_i\, k\} \quad Cl_2 \cup \{@_i\, j\}}{Cl_1 \cup Cl_2 \cup \{@_j\, k\}}$	M si $Cl_2 \subset Cl_1$ S si $@_j\, k \in Cl_2$ y $Cl_1 \subset Cl_2$
$\text{PAR}^{\neq}$	$\dfrac{Cl_1 \cup \{@_i \neg k\} \quad Cl_2 \cup \{@_i\, j\}}{Cl_1 \cup Cl_2 \cup \{@_j \neg k\}}$	M si $Cl_2 \subset Cl_1$ S si $j = k$ y $Cl_1 \subset Cl_2$
PAR^{p}	$\dfrac{Cl_1 \cup \{@_i\, p\} \quad Cl_2 \cup \{@_i\, j\}}{Cl_1 \cup Cl_2 \cup \{@_j\, p\}}$	M si $Cl_2 \subset Cl_1$
$\text{PAR}^{\neq p}$	$\dfrac{Cl_1 \cup \{@_i \neg p\} \quad Cl_2 \cup \{@_i\, j\}}{Cl_1 \cup Cl_2 \cup \{@_j \neg p\}}$	M si $Cl_2 \subset Cl_1$
$\text{PAR}^{[r]}$	$\dfrac{Cl_1 \cup \{@_i\, [r]\varphi\} \quad Cl_2 \cup \{@_i\, j\}}{Cl_1 \cup Cl_2 \cup \{@_j\, [r]\varphi(i/j)\}}$	M si $Cl_2 \subset Cl_1$
$\text{PAR}_1^{@\Diamond}$	$\dfrac{Cl_1 \cup \{@_s\, t\} \quad Cl_2 \cup \{@_s\, \langle r\rangle n\}}{\begin{array}{c} C_1 : Cl_1 \cup Cl_2 \cup \{@_t\, \langle r\rangle k\} \\ C_2 : Cl_1 \cup Cl_2 \cup \{@_n\, k\} \end{array}}$	M si $Cl_2 \subset Cl_1$ S si $@_n\, k \in Cl_1$ y $Cl_1 \subset Cl_2$
PAR-UNIT	$\dfrac{Cl_1 \cup \{@_k\, \varphi\} \quad \{@_i\, j\}}{Cl_1(i/j) \cup \{@_k\, \varphi(i/j)\}}$	M Siempre
$[r]$	$\dfrac{Cl_1 \cup \{@_i\, [r]\varphi\} \quad Cl_2 \cup \{@_i\, \langle r\rangle j\}}{Cl_1 \cup Cl_2 \cup \{@_j\, \varphi\}}$	M si $@_j\, \varphi \in Cl_2$ y $Cl_2 \subset Cl_1$ S si $@_j\, \varphi \in Cl_1$ y $Cl_1 \subset Cl_2$
$[r^{-1}]$	$\dfrac{Cl_1 \cup \{@_j\, [r]\varphi\} \quad Cl_2 \cup \{@_i\, \langle s\rangle j\}}{Cl_1 \cup Cl_2 \cup \{@_i\, \varphi\}}$	M si $@_i\, \varphi \in Cl_2$ y $Cl_2 \subset Cl_1$ S si $@_i\, \varphi \in Cl_1$ y $Cl_1 \subset Cl_2$
RES^{p}	$\dfrac{Cl_1 \cup \{@_i \neg p\} \quad Cl_2 \cup \{@_i\, p\}}{Cl_1 \cup Cl_2}$	M si $Cl_2 \subset Cl_1$ S si $Cl_1 \subset Cl_2$
$\text{RES}^{[r]}$	$\dfrac{Cl_1 \cup \{@_j\, [r]\varphi\} \quad Cl_2 \cup \{@_i\, \langle r\rangle \neg\varphi\}}{Cl_1 \cup Cl_2}$	M si $Cl_2 \subset Cl_1$ S si $Cl_1 \subset Cl_2$
$\downarrow$	$\dfrac{C \cup \{@_l \downarrow m.\varphi\}}{C \cup \{@_l \varphi(m/l)\}}$	Ninguna
$\wedge$	$\dfrac{C \cup \{@_l(\varphi_1 \wedge \varphi_2)\}}{C \cup \{@_l\varphi_1\} C \cup \{@_l\varphi_2\}}$	Ninguna
$\vee$	$\dfrac{C \cup \{@_l(\varphi_1 \vee \varphi_2)\}}{C \cup \{@_l\varphi_1, @_l\varphi_2\}}$	Ninguna

M : Premisa principal **S** : Premisa secundaria

Figura 3.1: Reglas de HyLoRes

En la Figura 3.2 podemos ver gráficamente la estructura del trie luego de agregar distintas cláusulas representadas como una lista ordenada de enteros.

Determinar si una cláusula del trie subsume a una cláusula nueva equivale a verificar si alguna cláusula del trie es un subconjunto de la cláusula nueva. Esto se lleva a cabo recorriendo el trie

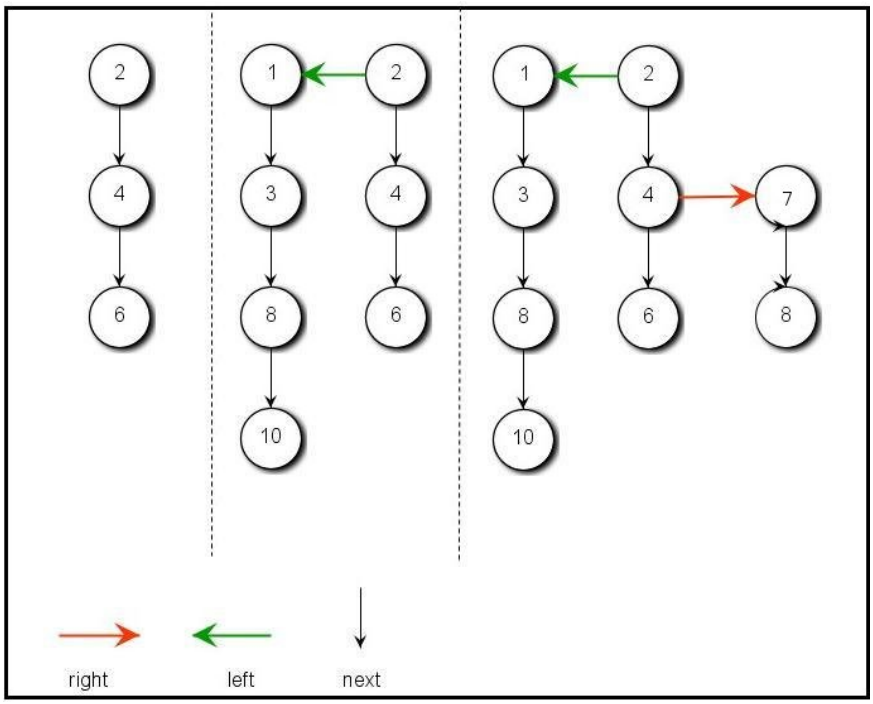

Figura 3.2: Ejemplo de un trie luego de agregar [2,4,6], [1,3,8,10] y [2,4,7,8]

y comparando los valores de los nodos. El pseudocódigo de la función que verifica subsunción se muestra en el Algoritmo 3.

La otra forma de representación que se utiliza para el chequeo global de subsunción hacia atrás es un mapa de cláusulas por fórmula. Para cada fórmula hay una lista de las cláusulas que la tienen como una de sus fórmulas integrantes. Luego cuando se quiere determinar si una cláusula nueva subsume a alguna ya existente, se busca en ese mapa todas las cláusulas que incluyan entre sus fórmulas a todas las fórmulas de la cláusula nueva, y de encontrar algunas, son subsumidas, ya que la cláusula nueva es un subconjunto de estas.

3.2. Optimizaciones

El conjunto *in-use* indexa las cláusulas de acuerdo a la forma de la fórmula distinguida de éstas. Permitiendo de esta manera que a partir de la cláusula dada y una regla, la búsqueda en in-use de las cláusulas candidatas para intentar aplicar esa regla sea eficiente. En la Figura 3.3 podemos ver los distintos índices y para qué reglas son utilizados cada uno de ellos.

HyLoRes utiliza una nueva regla de paramodulación, la regla PAR-UNIT. Esta regla es distinta a las otras reglas en cuanto que es aplicada a todas las cláusulas y no sólo a las cláusulas de in-use. El objetivo de esta regla es evitar que el número de nominales crezca y como consecuencia lo haga también el número de cláusulas generadas. La regla se aplica entre una cláusula cualquiera y una cláusula atómica cuya única fórmula es de la forma $@_i j$ e i es mayor que j de acuerdo al orden establecido, y reemplaza i por j en todas las fórmulas de la primer cláusula que contengan a i.

Algoritmo 3 Algoritmo de chequeo se subsunción de una cláusula por el trie

```
function subsume
input: cl: cláusula
input: trie: SubsumptionTrie

list: formulas := convertir(cl)
if trie == Nulo
   return False
end if

if formulas.length == 0
    return False
end if

case compare (trie.val)}
          =  : return trie.next == Nulo or subsume(tail(formulas), trie.next) or
                      subsume(tail(formulas), trie.right)


          <  :  return subsume(buscarNodo(head(formulas), trie.left) or
                      subsume(tail(formulas), trie)

          >  :  return subsume(trie.right, formulas)
```

Índice	Clave	Fórmula	Reglas
UnitEq	i	$@_i\, j$	PAR-UNIT
NonUnitNeq	i	$@_i\, \varphi$	PAR^{p}, $\text{PAR}^{\neq p}$, $\text{PAR}^{[r]}$, $\text{PAR}^{=}$, $\text{PAR}^{\neq}$, $\text{PAR}_1^{@\Diamond}$
AtNegNom	i	$@_i\, \neg j$	$\text{PAR}^{\neq}$
AtProp	i	$@_i\, p$	PAR^{p}, RES^{p}
AtNegProp	i	$@_i\, \neg p$	$\text{PAR}^{\neq p}$, RES^{p}
AtBoxF	i	$@_i\, [r]\psi$	$\text{PAR}^{[r]}$, $\text{RES}^{[r]}$, $[r^{-1}]$
AtDiamNom	i	$@_i\, \langle r\rangle j$	$[r]$, $\text{PAR}^{[r]}$
AtDiamNomInv	j	$@_i\, \langle r\rangle j$	$[r^{-1}]$

- $i, j \in \mathsf{NOM}$
- $p \in \mathsf{PROP}$
- $\varphi \notin \mathsf{ATOM}$

Figura 3.3: Índices de in-use

Capítulo 4

Resolución en paralelo

En el capítulo anterior describimos la implementación del demostrador automático para lógicas híbridas HyLoRes. En este capítulo vamos a contar cómo modificamos HyLoRes agregándole concurrencia para aprovechar ambientes multiprocesador y qué problemas nuevos surgieron al hacerlo. En algunos casos cuestiones que eran triviales de resolver en un algoritmo serial, dejaron de serlo en paralelo.

Primero presentaremos un algoritmo simple suponiendo ciertas restricciones y dejando de lado temas como subsunción. Una vez presentado este algoritmo desarrollaremos estos temas y trataremos las restricciones dejadas de lado, exponiendo cómo resolvimos los distintos problemas que se nos presentaron para poder obtener finalmente una implementación completa de HyLoRes en paralelo.

4.1. Algoritmo simple

4.1.1. Distribución de cláusulas

Si observamos la Figura 3.1, todas las reglas binarias a excepción de $[r^{-1}]$, $\text{RES}^{[r]}$ y PAR-UNIT, tienen como premisa cláusulas cuyas fórmulas distinguidas deben coincidir en el nominal usado como etiqueta. Dejemos de lado por un rato estas tres reglas y concentrémonos en el resto de las reglas a las que llamaremos simples. El algoritmo de la cláusula dada elige una cláusula y trata de aplicar alguna regla del cálculo. Para cada una de las reglas binarias busca entre todas las cláusulas del conjunto *in-use* cláusulas que sirvan como premisa de la regla. Si miramos las reglas simples de resolución de lógicas híbridas, sólo es necesario evaluar las cláusulas cuya fórmula distinguida compartan el mismo nominal. Esta particularidad de las reglas nos permite introducir una forma de paralelismo que es la partición de los datos. Esto es particionar el conjunto de datos en subconjuntos disjuntos, para luego procesar cada partición de forma concurrente en distintas unidades de procesamiento. En nuestro caso particionamos los conjuntos *in-use* y *clauses* de acuerdo a los nominales y ejecutamos el algoritmo de la cláusula dada en paralelo para cada una de estas particiones.

A cada unidad de procesamiento que ejecuta el algoritmo de la cláusula dada sobre una partición de nominales la llamaremos *Worker*. Cada *Worker* posee un conjunto *new*, *in-use* y *clauses*. Cuando el *Worker* aplica una regla sobre una cláusula de *clauses* genera nuevas cláusulas. Si los nominales etiqueta de éstas pertenecieran siempre a la misma partición del *Worker* podrían ser agregadas a *new* directamente. Sin embargo esto no siempre va a ocurrir como podemos ver en el siguiente ejemplo:

Ejemplo 1. Sea el *Worker* A que trabaja sobre la partición de nominales i, k y el *Worker* B sobre j, donde $i \succ j \succ k$. Si A aplica la regla $\text{PAR}^{=}$ sobre el siguiente par de cláusulas, $@_i\, k$ y $@_i\, j$, se va a generar la cláusula $@_j\, k$ que debiera ser incluida en el conjunto *clauses* de B y no de A.

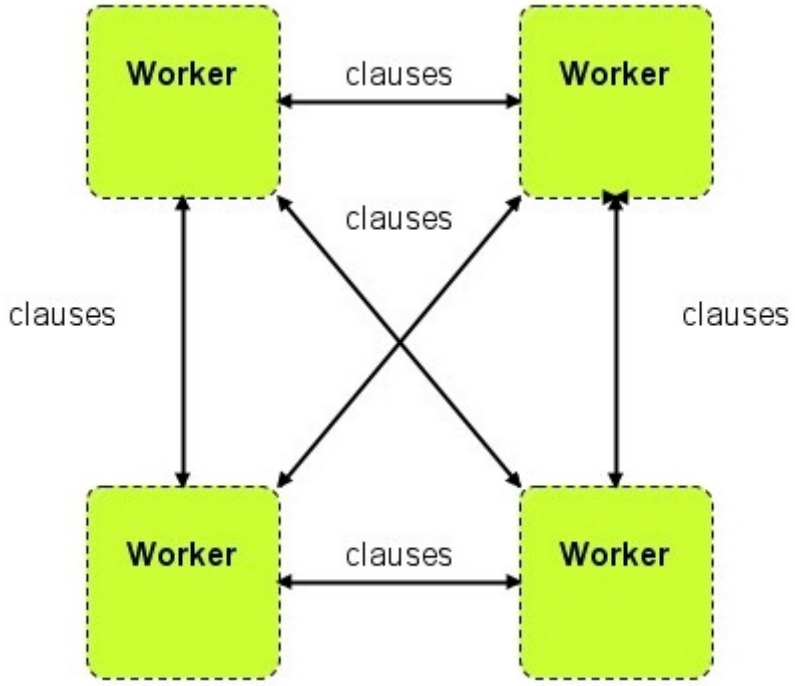

Figura 4.1: Distribución de clausulas sin Dispatcher

Para lograr que las cláusulas se distribuyan al *Worker* correspondiente analizamos las siguientes alternativas:

- Conectar todos los Workers entre sí y que cada Worker envíe las clausulas nuevas al Worker correspondiente según el nominal.(Figura 4.1)
- Introducir un nuevo proceso[1] *Dispatcher* que sea el encargado de recibir las nuevas cláusulas y distribuirlas al Worker correspondiente.(Figura 4.2)

La primera alternativa tiene varios problemas a resolver. El primero es determinar cómo sabe cada Worker a qué Worker le corresponde cada cláusula. Una opción sería utilizar una función que asigne un nominal a un Worker utilizando sólo el nominal. Por ejemplo calculando un *hash* del nominal y normalizarlo en la cantidad de Workers. Sin embargo esto no nos permitiría hacer un buen balanceo de carga de los Workers. Para esto necesitaríamos una función más dinámica que mantenga las información de asignación consistente entre todos los Workers. Otro problema a resolver es cómo determinar que no hay más cláusulas para procesar. De alguna manera cada Worker debería saber el estado de los otros Workers y determinar si en algún momento de la ejecución todos los Workers se quedaron sin cláusulas para procesar y no tienen cláusulas nuevas que distribuir. Esto en un ambiente distribuido resulta complejo de resolver ya que debería haber intercambio de mensajes continuos y alguien que coordine estos mensajes.

La segunda alternativa resuelve estos problemas de una manera más simple. Para el problema de la distribución de cláusulas la forma de resolverlo es la siguiente: las cláusulas nuevas son enviadas por los *Workers* al nuevo proceso *Dispatcher* y éste, utilizando una función de *dispatching* que veremos más adelante con más detalle, las distribuye a los correspondientes *Workers* de acuerdo al nominal de la etiqueta de la fórmula distinguida. Al ser sólo el *Dispatcher* el encargado de esta

[1] Proceso no se refiere al proceso del sistema operativo sino a una unidad de procesamiento del programa que ejecuta de manera independiente del resto.

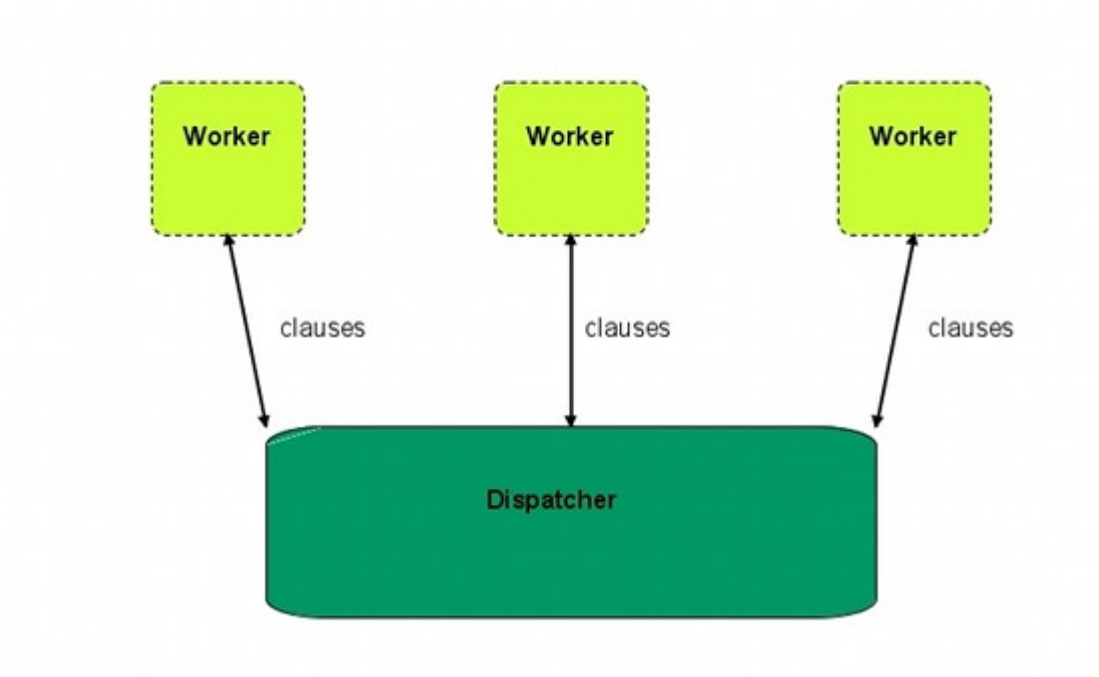

Figura 4.2: Distribución de clausulas con Dispatcher

función no es necesario compartir la información de asignación con ningún otro proceso haciéndolo más simple que la primer alternativa. La manera de determinar que los Workers no tienen más cláusulas a procesar también es más sencilla: dado que el Dispatcher es el encargado de distribuir todas las cláusulas, puede mantener la información de cuántas cláusulas se enviaron a cada Worker y esperar un mensaje del Worker cada vez que procesa una de las cláusulas que recibió. De esta manera puede determinar si un Worker está procesando cláusulas o no. Además dado que el conjunto de cláusulas nuevas están en el Dispatcher también puede determinar si los Workers van a recibir nuevas cláusulas. Con estos dos datos puede decidir cuándo no hay mas cláusulas para procesar.

Analizando las dos alternativas y por la manera de solucionar estos problemas fundamentales para el algoritmo de la cláusula dada decidimos utilizar la segunda alternativa.

En la Figura 4.3 podemos ver una comparación simple de los componentes del algoritmo serial y los de la versión paralela. En esta última aparece un nuevo proceso *Main*. Este proceso es necesario por un lado para lanzar a los otros procesos. También es el encargado de devolver el resultado y de coordinar al resto de los procesos. Es por esto que está comunicado con el *Dispatcher* y con cada uno de los Workers.

4.1.2. Cómo determinar el resultado

Una vez definido cómo distribuimos y procesamos las cláusulas, tenemos que ver cómo determinar el resultado. En el algoritmo serial la forma de determinar si la fórmula inicial es insatisfacible o no, es muy simple. Cuando se deriva la cláusula vacía el algoritmo sale de su ciclo principal y devuelve insatisfacible. Por otro lado cuando no quedan más cláusulas en *clauses* y no se derivó la clausula vacía el algoritmo devuelve satisfacible como resultado.

En el caso paralelo al haber una ejecución distribuida del algoritmo de la cláusula dada no es tan sencillo. Por un lado es más complejo determinar cuándo parar, pero además es necesario coordinar la parada de cada una de las ejecuciones paralelas.

Veamos primero cómo resolvimos el caso de insatisfacibilidad. Si la fórmula inicial es insa-

tisfacible alguno de los *Workers* derivará la cláusula vacía. Por lo tanto utilizamos el siguiente mecanismo para terminar el programa en este caso: cuando un *Worker* deriva la cláusula vacía, le envía un mensaje al proceso *Main* y éste se encarga de avisar al resto de los *Workers* que paren de procesar ya que el resultado es insatisfacible y devuelve el resultado.

En el caso de que la fórmula sea satisfacible se debe determinar de alguna manera cuándo los *Workers* ya procesaron todas las cláusulas y no hay más cláusulas nuevas por procesar. Como mencionamos anteriormente, el *Dispatcher* es el proceso que se utiliza para determinar esto. Posee un contador con la cantidad de cláusulas totales que envió a los *Workers*, cada vez que envía n cláusulas nuevas a un *Worker*, incrementa en n este contador. A su vez, cuando los *Workers* le envían al *Dispatcher* las cláusulas nuevas a distribuir, también le envían la cantidad de cláusulas que procesaron, y el *Dispatcher* decrementa el contador en esa misma cantidad. En caso de que todas las cláusulas procesadas por el *Worker* sean subsumidas y no haya cláusulas nuevas sólo le envía un mensaje al *Dispatcher* con el número de cláusulas procesadas. Finalmente si el contador llega a cero y no hay más clausulas por repartir, entonces la fórmula de entrada es satisfacible. Ya que los *Workers* no poseen más cláusulas en *clauses* porque las procesaron todas y no van a recibir más cláusulas nuevas. Entonces el *Dispatcher* le notifica al proceso *Main* que la fórmula es satisfacible y éste último devuelve el resultado.

Como mencionamos en el Capítulo 2, el cálculo en que se basa HyLoRes es refutacionalmente completo, pero no es un método de decisión para las lógica híbrida $H(@, \downarrow, \Diamond^{-1})$. Si una fórmula es satisfacible nunca se derivará la cláusula vacía. Por otro lado por más que la fórmula sea insatisfacible es posible que el tiempo que se tarde en determinar esto sea inaceptable (nada impide que pueda tardar años). Es por ello que introducimos la posibilidad de configurar un tiempo determinado (*timeout*) como parámetro, luego del cual el proceso *Main* devuelve como resultado que no pudo determinar si la fórmula era satisfacible o no.

Hasta acá ya tenemos un algoritmo paralelo de resolución que funciona bajo ciertas asunciones. En los Algoritmos 4, 5 y 6 se pueden ver los pseudocódigos de los procesos *Workers*, el *Dispatcher* y *Main* respectivamente.

.

4.2. Tratamiento de reglas complejas

Una vez presentada la idea del algoritmo retomemos ahora las reglas que dejamos de lado por no seguir el patrón de que se aplican sobre premisas cuyas fórmulas distinguidas poseen la misma etiqueta. Una de éstas es la regla PAR-UNIT. Como vimos en el capítulo anterior esta regla se aplica si aparece una cláusula cuya única fórmula es una igualdad, es decir tiene la forma $@i_j$ donde i es mayor que j. La regla reemplaza i por j en todas las fórmulas de todas las cláusulas. Por lo tanto para utilizar PAR-UNIT en nuestro algoritmo, no alcanza con enviar las igualdades al *Worker* que procesa i, ya que PAR-UNIT no reemplaza i sólo en las etiqueta sino en cualquier aparición de este nominal, con lo cual esto podría ocurrir en cláusulas cuyo nominal etiqueta no sea i, es decir en cualquier *Worker*. Entonces lo que hace el *Dispatcher* es distribuir estas cláusulas a todos los *Workers*, para que traten de aplicar PAR-UNIT en las cláusulas de *in-use* y *clauses*.

Otro tipo de cláusulas para las cuales no es trivial su distribución, son las cláusulas cuya fórmula distinguida es de la forma $@i\langle r\rangle j$. Ya que para aplicar la regla $[r^{-1}]$ de la Figura 3.1 se utiliza como premisa principal cláusulas cuya fórmula distinguida sea de la forma $@_j\,[r]\varphi$. Luego el *Dispatcher* debe enviar las cláusulas cuya fórmula distinguida es $@i\langle r\rangle j$ no sólo al *Worker* que procesa i sino también al que procesa j.

4.3. Subsunción hacia adelante

Una vez definido cómo se procesan las cláusulas, cómo se distribuyen y de qué manera se determina la parada del programa, falta ver cómo implementar el chequeo de subsunción. Recordemos que en el caso serial se utilizaba un trie para chequear subsunción hacia adelante, y

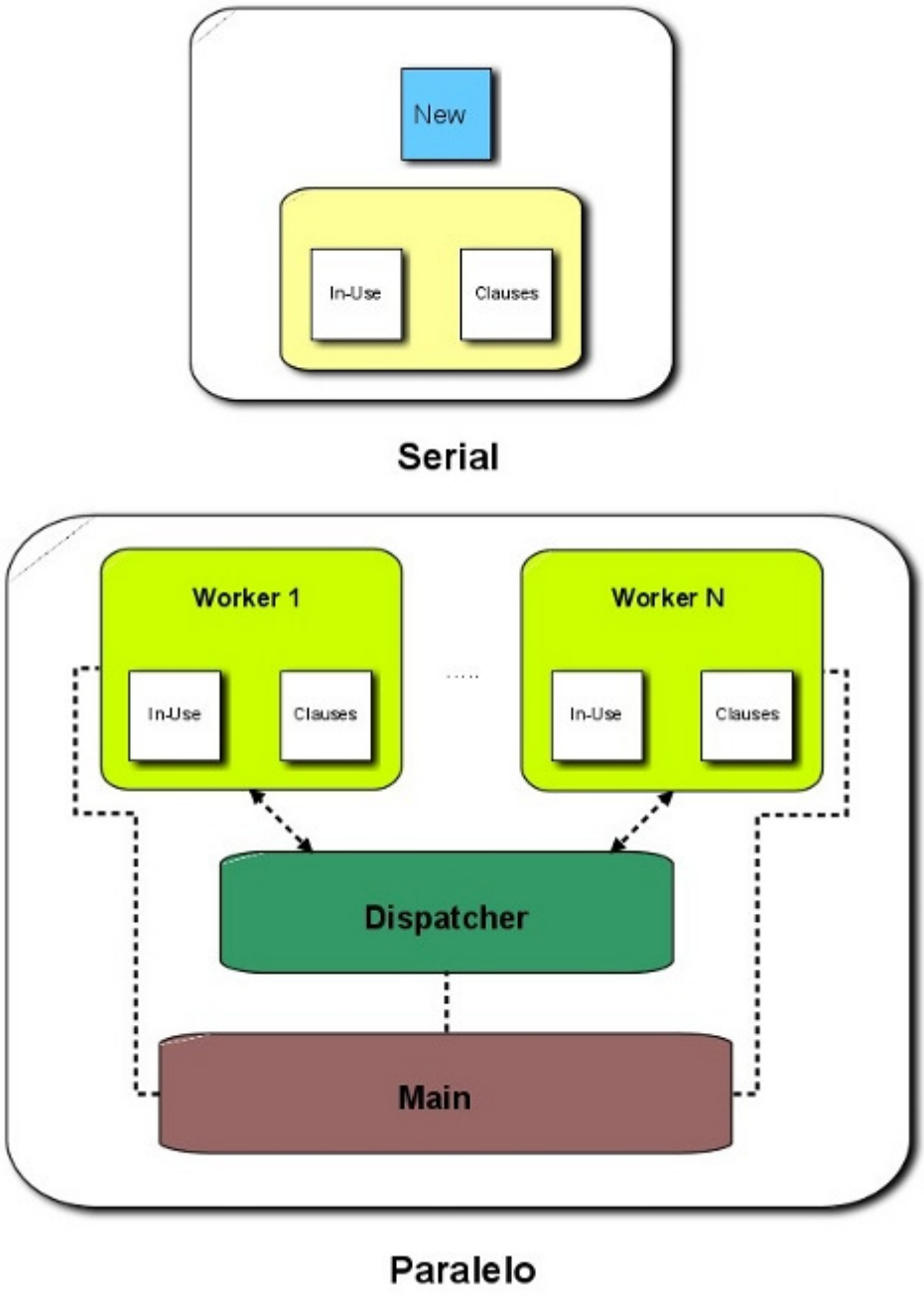

Figura 4.3: Comparación de componentes del algoritmo serial y del paralelo

Algoritmo 4 Algoritmo Worker

```
var: new, clauses, inuse: conjunto de cláusulas
var: given: cláusula
var: processed: int

clauses := {}; inUse := {}
processed := 0
while {true}
        bloquearse := clauses.isEmpty()
        mensaje := leerDeDispatcher(bloquearse)
        if deboTerminar(mensaje)
            enviarInUse
            return
        end if

        newClauses := mensaje
        clausesSize := size(newClauses) ∪ size(clauses)
        new := chequearSubsunción(newClauses)
        clauses := clauses ∪ new

        given := select(clauses)
        clauses := clauses \{given}

       new := applyRules(given,inuse)
       if {} ∈ new
          enviarUnsatToMain
          return
       end if
       inuse := inuse ∪{given}
       enviarADispatcher(new, clausesSize minus size(clauses)
end while
```

Algoritmo 5 Algoritmo Dispatcher

```
input: init: conjunto de cláusulas
var: waiting: int
var: new: conjunto de cláusulas

new := init

while {waiting != 0 and !vacio(new)}
        waiting := waiting + enviarCláusulasAWorkers(new)
        (procesadas, new) := recibirClaúsulasDeWorkers()
        waiting := waiting − procesadas mensaje := leerDeDispatcher(bloquearse)
end while
```

Algoritmo 6 Algoritmo Main

```
input: n: cantidad de workers
input: f: fórmula inicial
input: timeout: tiempo a esperar

lanzarWorkers(n)
lanzarDispatcher({f})

msg := leerMensaje(timeout)

eliminarProcesos()

return msg
```

un mapa con todas las cláusulas para chequear subsunción hacia atrás. En el algoritmo paralelo, la necesidad de tener todas las cláusulas al mismo tiempo atenta contra la idea de partición del conjunto de nominales en distintos procesos. Entonces para solucionar este problema probamos varias alternativas.

La primera implementación de subsunción que realizamos fue agregar un proceso separado del resto que se encargue del chequeo de subsunción. Este proceso recibía las cláusulas nuevas del *Dispatcher*, y le devolvía a éste las que no fueron subsumidas para distribuirlas entre los *Workers*. Este proceso tenía una referencia al *trie* que contiene todas las cláusulas y al que iba agregando éstas a medida que las recibía del *Dispatcher*. El resultado obtenido no fue muy bueno, el nuevo proceso se transformó en un cuello de botella y los *Workers* trabajaban de manera casi serializada.

Como una mejora a esta primera implementación, probamos un mecanismo similar pero asincrónico: el *Dispatcher* enviaba las cláusulas al proceso que chequeaba subsunción y al mismo tiempo también a los *Workers*. Luego si las cláusulas eran subsumidas, el *Dispatcher* notificaba a los *Workers* y éstos eliminaban estas cláusulas de *in-use* y de *clauses*. En esta implementación este nuevo proceso seguía siendo un cuello de botella, era demasiado lento, y la cantidad de cláusulas crecía demasiado ya que los workers procesaban muchas cláusulas que hubieran sido subsumidas y a partir de éstas se generaban a su vez nuevas cláusulas. Con lo cual el número de cláusulas generadas crecía demasiado rápido.

Entonces decidimos eliminar este proceso que chequeaba subsunción y hacer que el chequeo de subsunción lo hicieran los *Workers*. Es decir que funcione similar a la versión no concurrente, donde el chequeo de subsunción se realiza antes de pasar las cláusulas del conjunto *new* a *clauses*. Para poder implementar esto fue necesario como primer paso compartir el *trie* y el mapa de cláusulas por fórmula que presentamos en el capítulo anterior, entre todos los *Workers*. Esto es posible de realizar en un ambiente multiprocesador con memoria compartida, que es el ambiente sobre el cual trabajamos. El nuevo problema que surge es el de acceder a estas estructuras de manera concurrente y modificarlas manteniendo la consistencia de las mismas.

Para el *trie* tratamos primero de utilizar *Software Transactional Memory* (STM), una biblioteca que define variables de tipo transaccionales y luego permite ejecutar operaciones sobre éstas de manera atómica. Es decir que si dos procesos tratan de modificar la misma variable al mismo tiempo, uno lo hará sin problemas y el segundo detectará la colisión y reintentará la operación de manera transparente. Los resultados obtenidos fueron buenos, sin embargo creíamos que se podían mejorar, sobre todo teniendo en cuenta cómo funciona el trie. Observamos que sólo puede haber colisiones cuando se modifica un mismo nodo, y que la probabilidad de que esto ocurra parece baja, con lo cual la utilización de STM agregaba demasiado *overhead*. Entonces probamos con una operación atómica de *compare-and-swap*(CAS). Una operación CAS consta en modificar el contenido de una variable sólo si esta no fue modificada entre que es léida y el nuevo valor es escrito. Si recordamos cómo funciona la inserción de una cláusula al trie (Figura 2), en esta función

se va recorriendo el trie hasta encontrarse con un nodo nulo en el cual insertamos las fórmulas restantes de la cláusula. Luego lo que hicimos es utilizar punteros a nodos en el trie, y cuando insertamos un nuevo nodo lo hacemos utilizando una función atomCAS que devuelve verdadero si pudo modificar el puntero o falso sino. Entonces en el primer caso pasamos a la próxima fórmula de la cláusula y en el segundo volvemos a llamar a la función de inserción con los mismos parámetros. En la Figura 7 podemos ver el pseudocódigo de la función de inserción utilizando CAS.

Algoritmo 7 Algoritmo de inserción de una cláusula en el trie utilizando CAS

```
function insertar
input: cl: Lista de fórmulas
input: trie: SubsumptionTrie

if trie == Nulo
   b := atomCAS trie (head(cl), Nulo, Nulo)
   if b == true
       then insertar (tail(cl), trie.next)
       else insertar (cl, trie)
   end if
else
    case compare (val trie):
         = :  if trie.next != Nulo
                 trie.next := insertar(tail(cl), trie.next)
              end if
         < :   trie.left := insertar (cl, trie.left)

         > :   trie.right := insertar(cl, trie.right)

end if
return trie
```

Los resultados obtenidos con esta nueva implementación fueron mejores a los de la implementación de STM.

Al permitir chequeos de subsunción de manera concurrente, existe la posibilidad que alguna subsunción no se realice como en el caso serial. Es decir puede ser que al momento de que un Worker esté chequeando subsunción, otro esté por insertar una fórmula que subsuma alguna de las cláusulas nuevas del primer Worker. Esto no afecta el resultado final aunque si podría afectar mucho el desempeño. Ya que subsunción es una optimización, por lo tanto si alguna fórmula no es subsumida el cálculo sigue siendo refutacionalmente completo.

4.3.1. Subsunción hacia atrás

Para subsunción hacia atrás tratamos primero de utilizar una variable de tipo MVar. Las variables de tipo MVar son variables mutables sincronizadas. Son utilizadas para comunicar distintos procesos concurrentes. Su acceso es sincronizado, es decir sólo un proceso puede acceder a su valor a la vez. Si un proceso trata de accederla y ya hay otro proceso que la está usando, el primero se queda bloqueado hasta que este último la libere. Si más de un proceso quiere acceder a la variable mientras está siendo accedida por otro proceso, se crea una cola FIFO con los procesos bloqueados para luego ser despertados de uno a la vez una vez que la variable es liberada.

Los resultados que obtuvimos fueron malos: a medida que probábamos con más concurrencia (es decir con más Workers) la performance se degradaba. Analizando los tiempos de cada función del programa nos dimos cuenta de que el chequeo de subsunción hacia atrás ocupaba un tiempo considerable en una ejecución del programa. Lo que estaba ocurriendo era que el chequeo de

subsunción tardaba un tiempo considerable y los Workers se encolaban a la espera del acceso a la variable MVar, con lo cual estaban mucho tiempo bloqueados sin procesar nada. Entonces probamos no utilizar subsunción hacia atrás, pero nos dimos cuenta de que en la versión paralela de HyLoRes es casi indispensable utilizarlo, ya que en caso contrario el número de cláusulas generadas crece demasiado rápido. Entonces lo que decidimos fue hacer una solución intermedia, hacer un chequeo de subsunción hacia atrás pero sólo con las cláusulas que procesa cada Worker. Es decir, cada Worker tiene su propio mapa de cláusulas por fórmulas, y lo va modificando con las cláusulas que va recibiendo del Dispatcher. Y luego al momento de aplicar subsunción hacia atrás lo hace sólo con esas cláusulas. Si bien la cantidad de subsunciones fue menor a las de la versión serial, fue suficiente para mantener la cantidad de cláusulas en un nivel manejable. Igualmente que en el caso de subsunción hacia adelante, si alguna fórmula no es subsumida el cálculo sigue siendo refutacionalmente completo.

4.4. Generación del modelo

En la versión serial de HyLoRes, cuando el resultado es satisfacible, se devuelve también como evidencia un modelo que se construye a partir de las cláusulas de *in-use*. En la versión paralela el conjunto *in-use* está particionado en los distintos *Workers*, por lo tanto se implementó un mecanismo para poder armar el conjunto *in-use* uniendo los conjuntos *in-use* de cada *Worker*. El mecanismo es el siguiente: cuando el *Dispatcher* detecta que el resultado es satisfacible, envía a los *Workers* un mensaje para que le envíen el conjunto *in-use*. Entonces los *Workers* envían el conjunto en otro mensaje y además dejan de procesar ya que el *Dispatcher* determinó que el resultado a devolver es satisfacible. Luego el *Dispatcher* une los distintos conjuntos para obtener el modelo que devuelve junto con el resultado.

4.5. Función de dispatching

El *Dispatcher* es el encargado de distribuir las cláusulas a los *Workers*, para ello utiliza una función de *dispatching* que se indica como parámetro de entrada al programa. Esta función como mencionamos anteriormente, es la encargada de asignar cada nominal a un Worker determinado. Implementamos tres funciones distintas para luego evaluar cuál era la más conveniente.

La primera calcula un *hash* sobre el nominal etiqueta de la fórmula distinguida de la cláusula a distribuir y aplica módulo en la cantidad de *Workers*.

La segunda utiliza el método de *round-robin* para asignar un nuevo nominal a un Worker distinto cada vez. Para distribuir cláusulas con nominales ya asignados se guarda en un diccionario la asignación de cada nominal al *Worker* correspondiente.

La tercera y última implementación trata de hacer un balanceo de carga entre los *Workers*. Por un lado se guarda en un diccionario la cantidad de cláusulas que procesó cada *Worker*, luego cuando hay que asignar un nuevo nominal se le asigna al *Worker* que menos cláusulas procesó hasta el momento. Para los nominales ya asignados utiliza el mismo mecanismo que la segunda función. A ésta tecera implementación la llamaremos *Load*.

4.6. Implementación en GHC

La implementación de Haskell que utilizamos fue GHC (Glasgow Haskell Compiler) versión 6.12. GHC posee una biblioteca de extensiones para concurrencia (Concurrent Haskell). Esta biblioteca posee primitivas que permiten lanzar y matar distintos hilos de ejecución (*threads*). Estos threads lógicos son administrados por el runtime de GHC. En nuestro caso el Dispatcher, Main y cada Worker corresponden a uno de estos threads. Si el programa lo ejecutamos como cualquier otro programa de Haskell, sólo un thread va a estar activo a la vez, entonces no habría ninguna ejecución en paralelo. Para lograr concurrencia es necesario indicarle vía un parámetro al runtime de GHC que utilice n threads del Sistema Operativo. Luego cada thread del Sistema

Operativo intentará ejecutar un thread lógico cuando esté activo. Vale aclarar que no lograremos paralelismo si la computadora donde estamos ejecutando no posee más de un procesador, ya que sólo un thread real estará activo a la vez. Tampoco tiene sentido especificar más threads que cantidad de procesadores existentes ni que la cantidad de threads lógicos. En el primer caso, porque sólo van a poder estar activos simultáneamente n threads, donde n es la cantidad de procesadores. Mientras que en el segundo caso, por más de que existan más procesadores, nunca habrá necesidad de activarlos todos si la cantidad de threads lógicos es menor a la cantidad de estos.

Esta biblioteca también provee variables mutables sincrónicas (MVar) que nos permiten compartir datos en memoria entre distintos procesos sin perder consistencia.

Capítulo 5

Resultados

Existen distintas formas de testear demostradores automáticos. Se pueden utilizar conjuntos de fórmulas generados al azar, fórmulas hechas a medida o problemas que provengan de aplicaciones del mundo real. Utilizar problemas del mundo real suele ser la mejor opción. Sin embargo se requiere que existan repositorios grandes de tests que sean mantenidos y actualizados de manera permanente. Los tests aleatorios suelen utilizarse cuando no se cuenta con tales repositorios. Este es el caso de las lógicas híbridas por lo cual la generación de problemas al azar es el método que utilizamos para realizar pruebas.

Para generar los tests aleatorios y ejecutarlos en HyLoRes utilizamos un *framework* llamado GridTest [Areces *et al.*, 2009]. GridTest utiliza el generador de fórmulas hGen [Areces and Heguiabehere., 2003] para generar los casos a testear de manera aleatoria. HGen genera fórmulas en forma normal disjuntiva, cada fórmula es una conjunción de disyunciones. La forma de especificar la fórmula a generar es a través de los siguientes parámetros:

- cantidad de cláusulas.
- cantidad de nominales.
- cantidad de relaciones.
- probabilidad de aparición de nominales y de relaciones.
- máximo nivel de anidamiento de los @.
- máximo nivel de anidamiento de los $\Diamond$.

GridTest utiliza hGen de la siguiente manera: empieza generando fórmulas con pocas cláusulas y gradualmente va generando fórmulas con mayor cantidad de éstas. Esto es útil ya que las fórmulas con menor cantidad de cláusulas tienden a tener más probabilidad de ser satisfacibles, y a medida que la cantidad de cláusulas crece, esa probabilidad disminuye. De esta manera es posible encontrar un punto donde la probabilidad de satisfacibilidad e insatisfacibilidad es casi la misma. Las fórmulas de este punto tienden ser difíciles de resolver para la mayoría de los demostradores, con lo cual permiten comparar la performance de ellos.

Una vez generadas las fórmulas, GridTest ejecuta uno o más demostradores sobre éstas y obtiene distintas métricas como el tiempo de ejecución o la cantidad de cláusulas generadas. Esta posibilidad de ejecutar más de un demostrador sobre el mismo conjunto de tests nos permitió comparar la versión serial de HyLoRes con la nueva versión paralela. Por último genera un reporte con los resultados obtenidos.

De por sí un test que genera n fórmulas no es relevante desde el punto de vista estadístico. Es por eso que GridTest permite generar lotes para cada tamaño de fórmula y luego utilizar estimadores estadísticos (mediana, promedio, etc.) para obtener resultados más relevantes. Al ejecutar lotes grandes de fórmulas y fórmulas con gran cantidad de cláusulas, el tiempo de ejecución puede ser

considerable. Afortunadamente GridTest está preparado no sólo para ser ejecutado en una sola computadora, sino que puede ser utilizado en un cluster de computadoras. De esta manera en vez de ejecutar un lote de fórmulas de tamaño b en una sola computadora, puede ejecutar lotes de b/n en n computadoras obteniendo una reducción lineal del tiempo requerido de ejecución. En nuestro caso utilizamos un cluster de LORIA (Lorraine Laboratory of IT Research and its Applications). El cluster cuenta con computadoras de hasta 2 procesadores Intel(R) Xeon(R) CPU L5420 @ 2.50GHz de 4 núcleos cada uno y 16 GB de memoria. Para realizar las pruebas el cluster permite elegir nodos homogéneos.

Aun ejecutando en el cluster, el tiempo de ejecución de cada prueba fue de varias horas. Tratamos de ejecutar lotes grandes de fórmulas y con la mayor cantidad de cláusulas por fórmula. Sin embargo estuvimos restringidos por el tiempo de ejecución de cada prueba y por problemas de memoria que surgieron y comentaremos más adelante.

5.1. Paralelismo

Cuando desarrollamos un programa que va a ser ejecutado en una computadora con varios procesadores, queremos maximizar el paralelismo. Es decir, queremos que la mayor cantidad de tiempo los distintos threads estén procesando. En nuestro caso para lograr esto necesitamos que los Workers estén ocupados la mayor cantidad de tiempo, ya que son los procesos que realizan el mayor procesamiento. Una forma de medir la carga de cada Worker es analizar la cantidad de cláusulas que procesó. Idealmente quisiéramos lograr que la carga se distribuya de forma pareja entre los distintos Workers, ya que sería un buen indicador de que hay un alto grado de paralelismo. Para medir el balance de la carga agregamos un gráfico al framework de test que muestra el porcentaje de cláusulas que procesó cada Worker sobre el total de las mismas.

5.1.1. Balance de carga

En las Figuras 5.1, 5.2, 5.3 podemos ver los resultados de carga obtenidos para las distintas funciones de dispatching. En este caso realizamos una prueba con dos Workers y los siguientes parámetros de configuración de las fórmulas: cantidad de lotes: 13, cantidad de proposiciones: 3 (V=3), cantidad de nominales: 5 (N=5), cantidad de relaciones: 10 (R=10), máximo nivel de anidamiento de $\diamond$: 20 (D=20), rango de cantidad de cláusulas por fórmula: entre 100 y 160 (L= [100..160]).

Como era esperado, la función de dispatching con la cual se obtuvo el mejor balance de carga fue con la función Load. Recordemos que esta función asigna un nuevo nominal al Worker que tenga menos nominales asignados. En la Figura 5.4 podemos ver el balance de carga en una ejecución con 5 Workers y la función de dispatching Load.

Otra observación que podemos hacer es que en la mayoría de los lotes la distribución no es muy cercana al balance ideal. Esto no quiere decir que no haya un alto grado de paralelismo, ya que no todas las cláusulas necesariamente tardan el mismo tiempo en ser procesadas. Para reforzar esto utilizamos la herramienta *threadscope* que muestra el estado de los *threads* del sistema operativo durante una ejecución de HyLoRes. En la Figura 5.5 podemos observar cómo en una ejecución con 2 Workers y 3 *threads*, estos están casi todo el tiempo ejecutando, lo que indicaría un muy alto grado de paralelismo al no haber casi nunca threads ociosos. Un thread podría estar ocioso si un Worker estuviera esperando a que el Dispatcher le envíe nuevas cláusulas. En la Figura 5.6 mostramos una ejecución con 3 Workers y 4 threads que tardó 12 segundos. Podemos observar que hay 2 threads que dejan de procesar a los 8 segundos. Seguramente se debe a que 2 Workers se quedaron sin cláusulas para procesar.

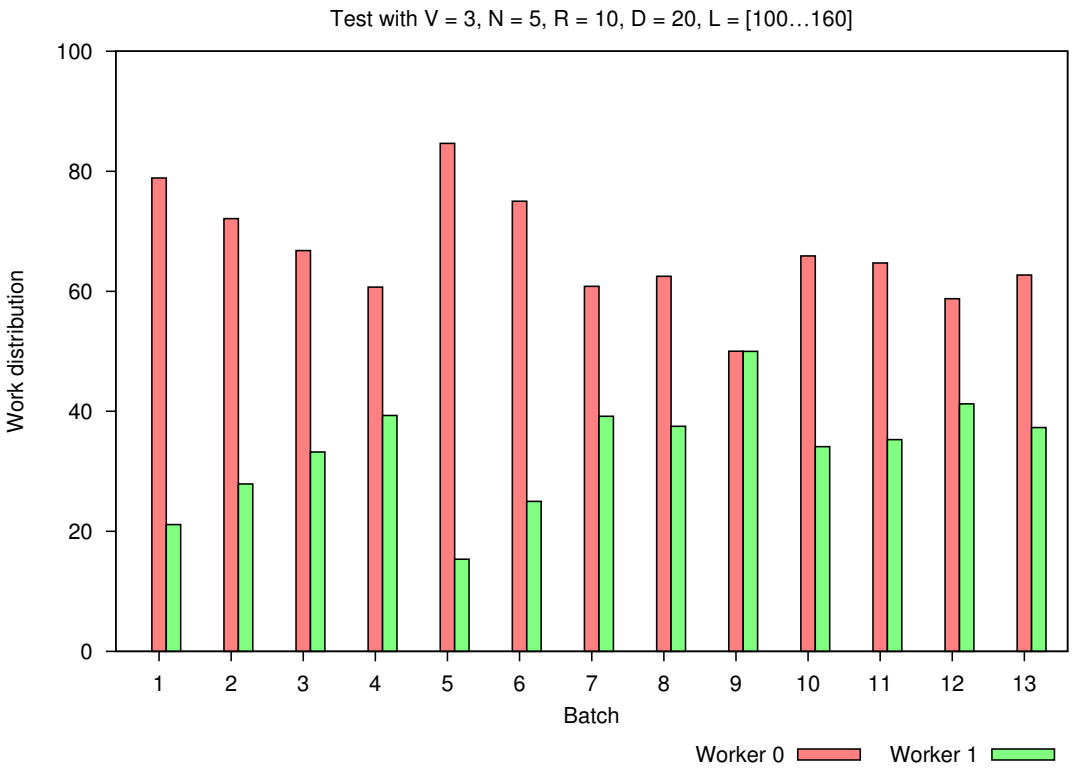

Figura 5.1: Distribución de carga para la función Hash de dispatching

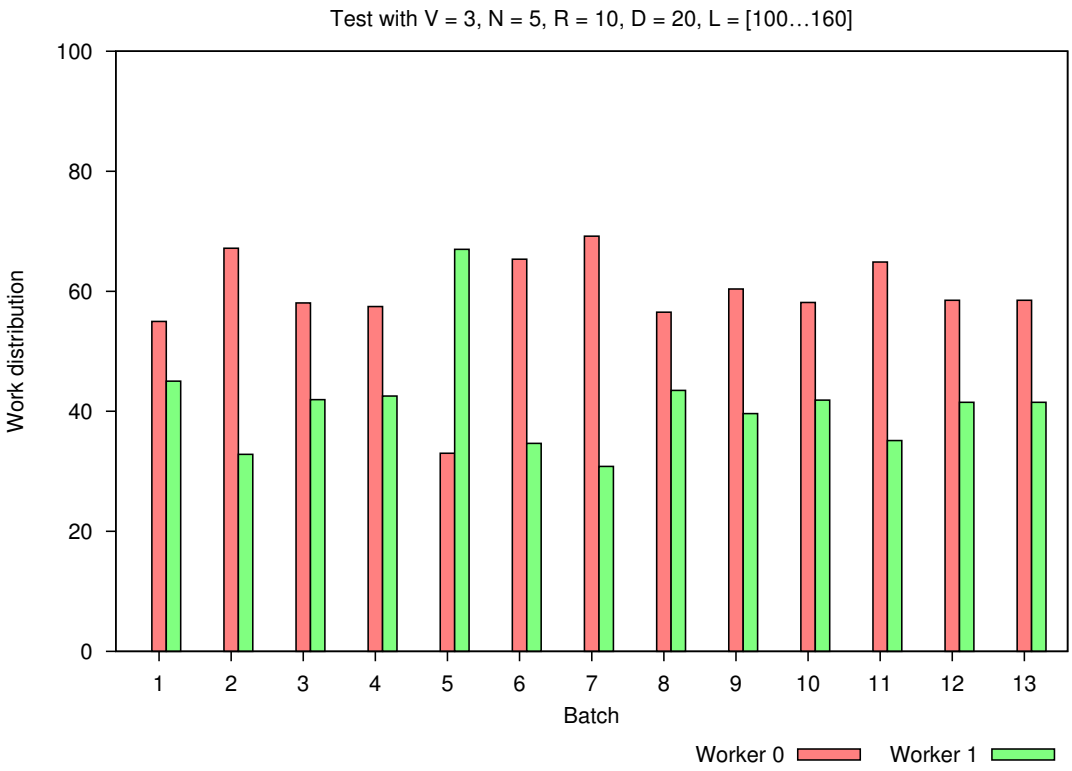

Figura 5.2: Distribución de carga para la función Round-Robin de dispatching

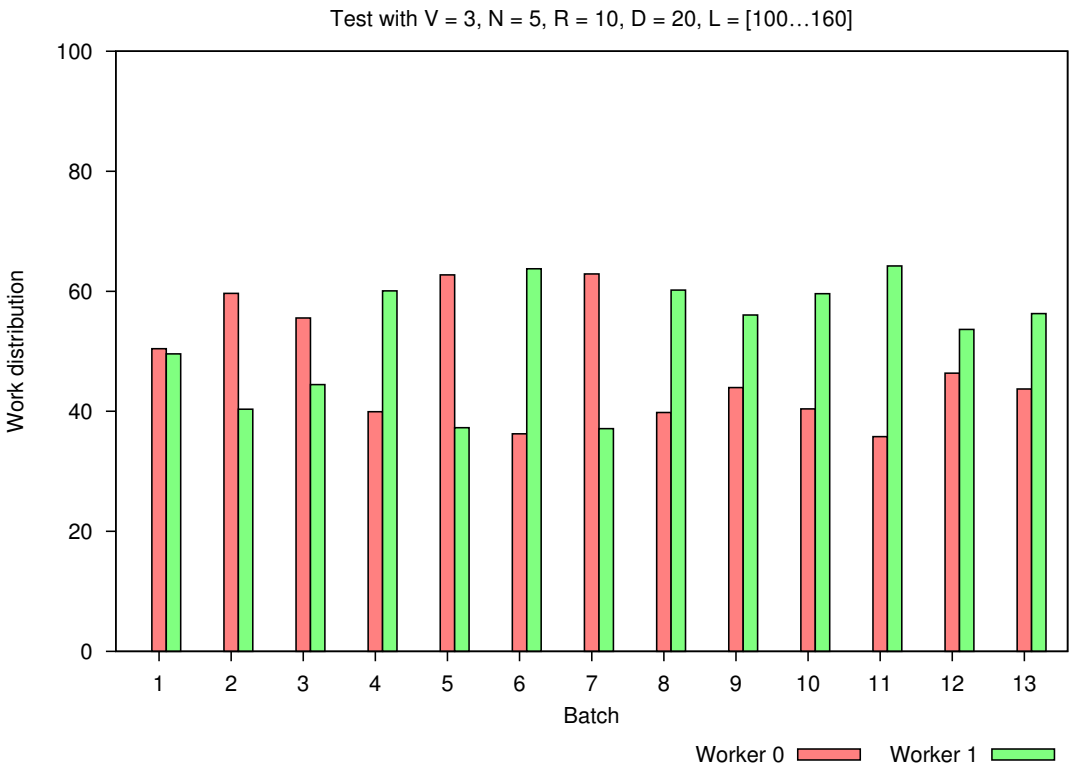

Figura 5.3: Distribución de carga para la función Load de dispatching

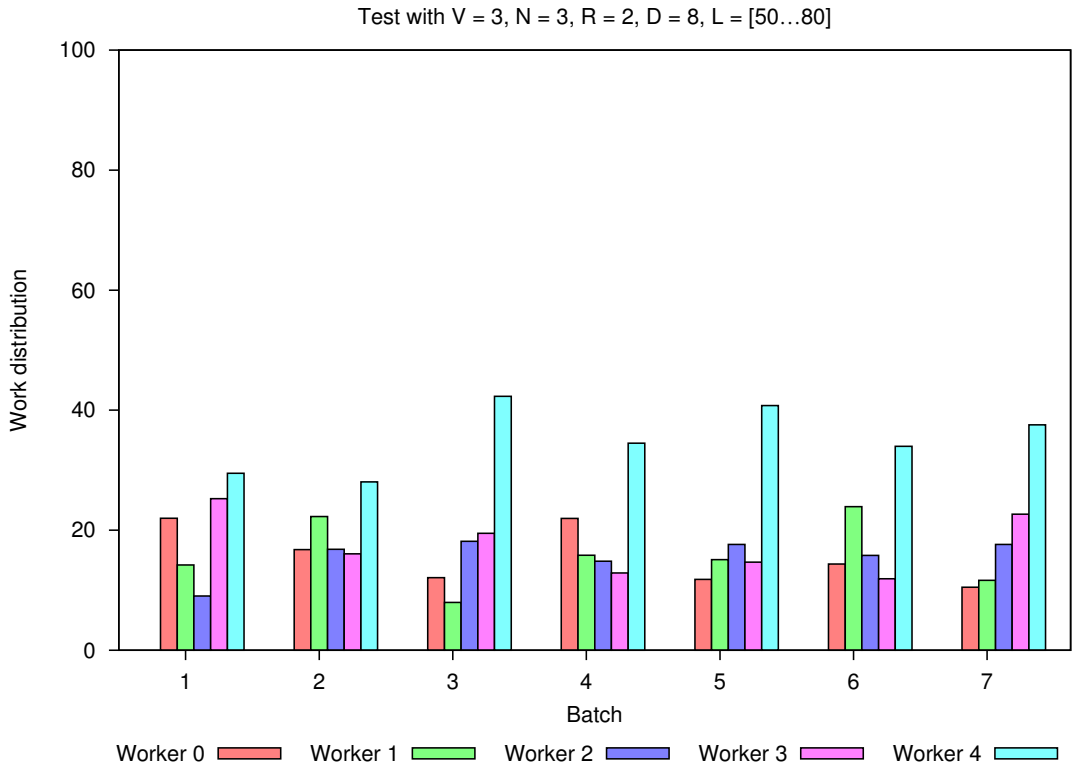

Figura 5.4: Distribución de carga para la función Load con 5 Workers

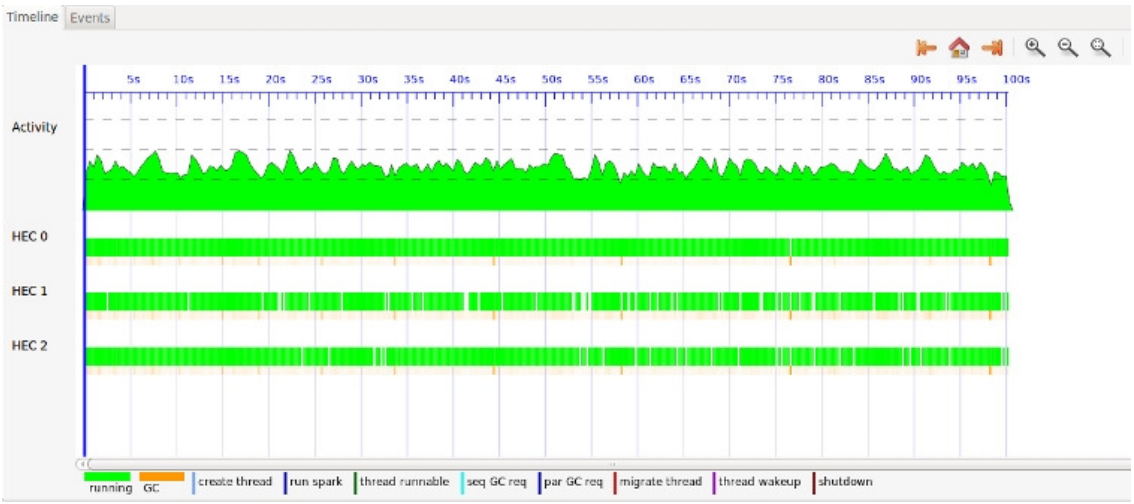

Figura 5.5: Estado de los threads durante la ejecución

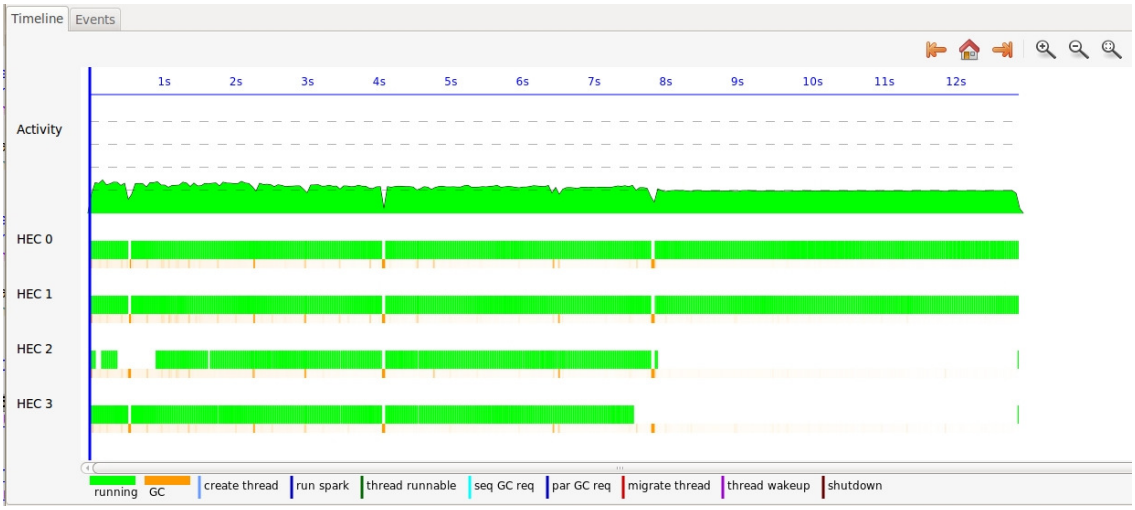

Figura 5.6: Estado de los threads durante una ejecución corta

5.1.2. Comparación del tiempo de ejecución con distintas funciones de dispatching

Para complementar los resultados anteriores realizamos una comparación del tiempo de ejecución entre las distintas funciones de dispatching. Es de esperar que la función que realiza mejor balance de la carga sea la que obtenga un menor tiempo de ejecución. Esto resultó ser así en nuestro caso como podemos ver en la Figura 5.7. El gráfico muestra la comparación de la mediana del tiempo de ejecución de las tres funciones de dispatching a medida que la cantidad de cláusulas de la fórmula a probar aumenta. En el eje X se muestra el tiempo de ejecución en escala logarítmica. También se muestran el tiempo máximo y mínimo de ejecución para cada tamaño de fórmula con una raya en el eje Y.

5.2. Problemas de memoria

Durante la ejecución de las primeras pruebas empezaron a surgir problemas de falta de memoria. Como el framework de test no contemplaba errores, las pruebas abortaban su ejecución y no podíamos obtener ningún resultado. Entonces modificamos el framework para contemplar el caso de falla por falta de memoria se tratara como un timeout. Esto lo decidimos ya que creíamos que aún con estas fallas esporádicas podíamos analizar los resultados.

En las Figuras 5.12, 5.13, 5.14, 5.15 y 5.16 podemos ver el porcentaje de los distintos resultados obtenidos para pruebas con 1, 2, 4, 7 y 8 Workers. Estos gráficos muestran para cada lote de fórmulas del mismo tamaño la fracción de cada resultado. Los resultados posibles son: satisfacible, insatisfacible, timeout, falta de memoria (oom) y error. En este último caerían el resto de las fallas de ejecución no relacionadas con la memoria. Este último tipo de fallas no fue muy frecuente pero nos pareció útil agregarlo, ya que cuando ocurría, el cluster no devolvía el resultado hasta que asumía que todos los nodos habían terminado por timeout. Lo que implicaría esperar muchas horas dependiendo de la cantidad de demostradores y tamaño de las fórmulas.

En los gráficos se puede ver cómo el porcentaje de errores de memoria crece a medida que el tamaño de las fórmulas crece y también es mayor cuando se utilizan más Workers.

En el próximo capítulo profundizaremos más este problema con sus posibles causas y soluciones.

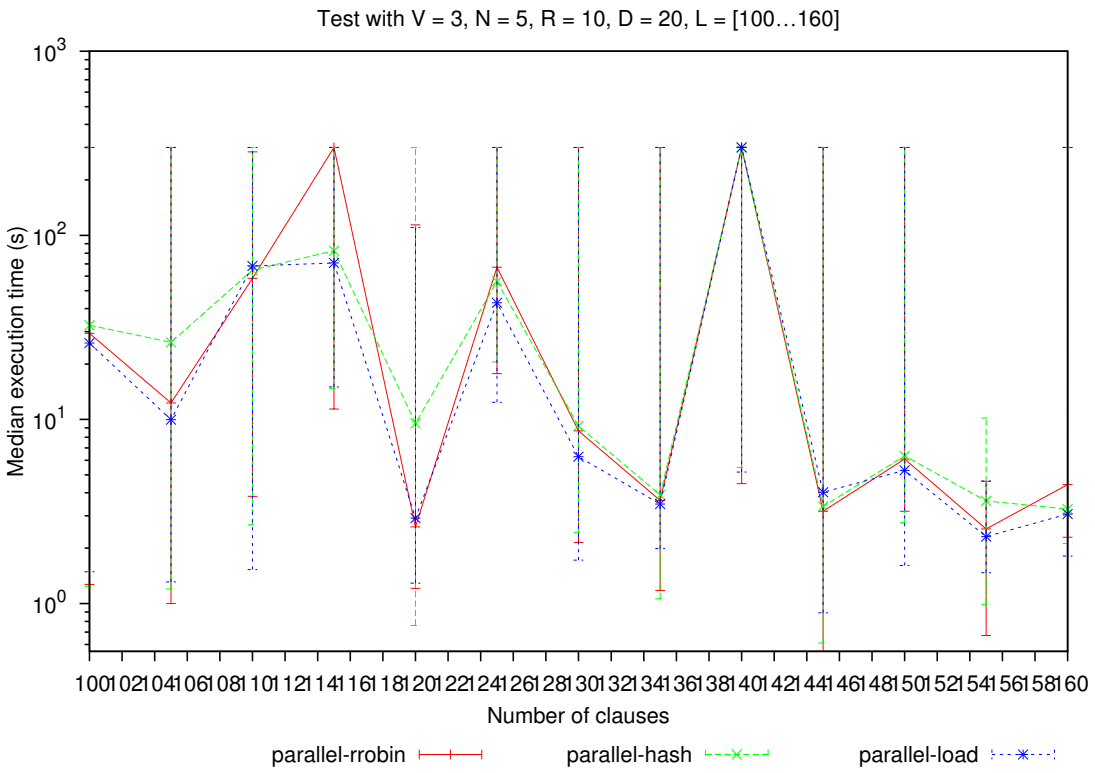

Figura 5.7: Distribución de resultados con distintos algoritmos

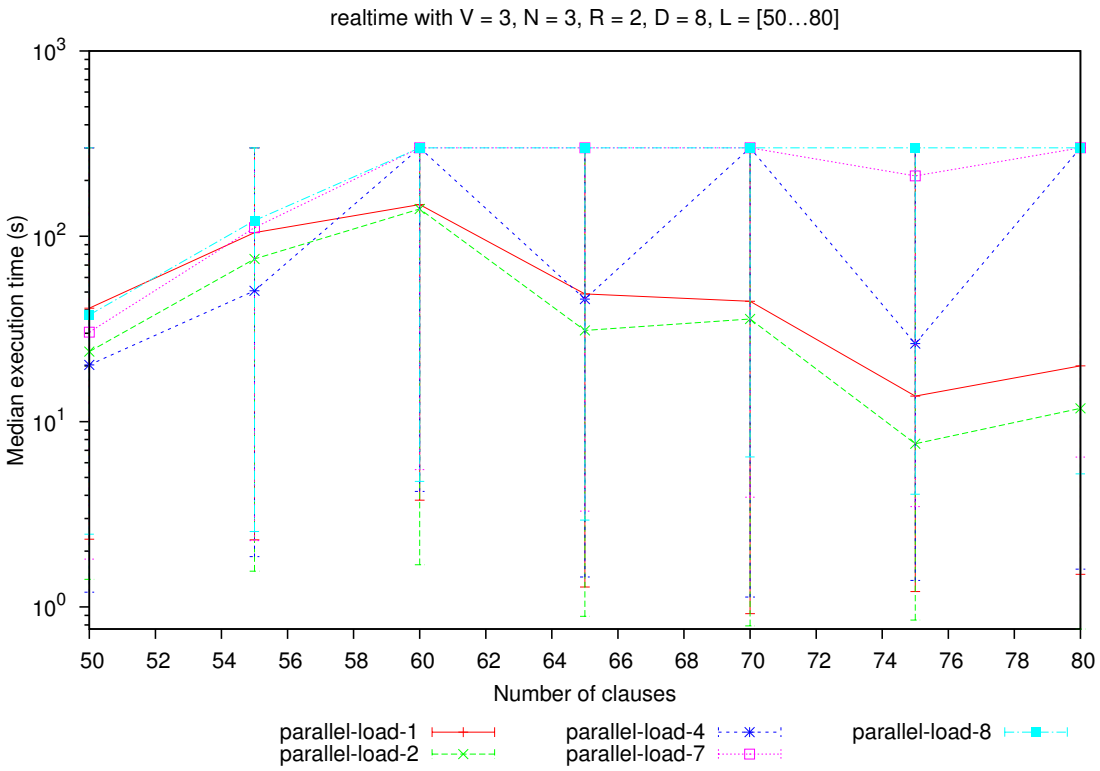

Figura 5.8: Comparación de tiempo de ejecución con distinta cantidad de Workers

5.3. Escalabilidad

Para probar la escalabilidad realizamos pruebas fijando la cantidad de threads del sistema operativo y aumentando la cantidad de Workers. Uno esperaría que a medida que aumenta la cantidad de Workers, el tiempo de ejecución sea menor. Sin embargo el resultado obtenido fue el siguiente: para fórmulas con menos cláusulas, a medida que usamos más Workers mejora hasta llegar a los 4 Workers. Luego a medida que el tamaño crece, el que obtiene mejores resultados es el de 2 Workers. (Figura 5.8) Que la configuración que mejor resultado obtuvo para las fórmulas más grandes sea la de 2 Workers es llamativo y puede estar causado por los problemas de falta de memoria mencionados anteriormente, ya que la cantidad de fallas crece cuando se incrementa la cantidad de Workers y el tamaño de las fórmulas. (Figuras 5.12, 5.13, 5.14, 5.15, 5.16) También es probable que exista un problema de contención de memoria, es decir que los procesadores estén compitiendo por el uso de la memoria compartida. Esto podría explicar también porque al principio el que mejor funciona sea el de 4 Workers y no el de 8. En la próxima sección explicaremos con más detalle este posible problema.

5.4. Comparación del tiempo de ejecución serial vs paralelo

Para analizar HyLoRes serial contra HyLoRes paralelo, realizamos primero una prueba con dos Workers y tres Threads. Elegimos dos Workers ya que vimos anteriormente que es la configuración con la cual se obtuvieron mejores resultados. La cantidad de threads es 3, ya que en el caso ideal que estén todos los procesos ejecutándose de manera simultánea se utiliza un thread para cada Worker y el restante para el Dispatcher. En la Figura 5.9 podemos observar como la mediana del tiempo de ejecución es menor en el caso de la versión paralela. Lo cual indicaría que obtuvimos

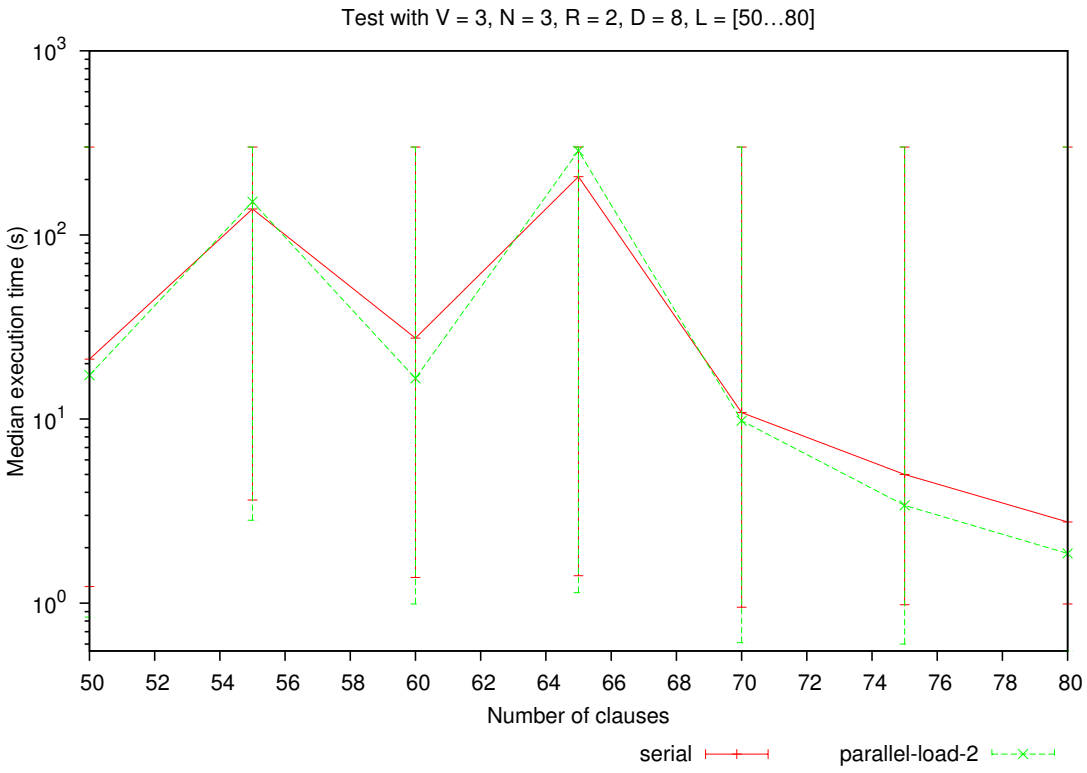

Figura 5.9: Comparación de tiempo de ejecución entre serial y paralelo con 2 Workers

una mejora al paralelizar el programa.

Luego realizamos comparaciones entre el serial y el paralelo con tres y siete Workers. Acá obtuvimos un resultado similar al de las pruebas de escalabilidad. La performance de HyLoRes con tres Workers fue mejor para los lotes iniciales pero luego fue peor a medida que los lotes van creciendo en tamaño de las fórmulas. El resultado para siete Workers fue aún peor, la mediana del tiempo de ejecución fue mayor para todos los lotes. (Figuras 5.10 y 5.11).

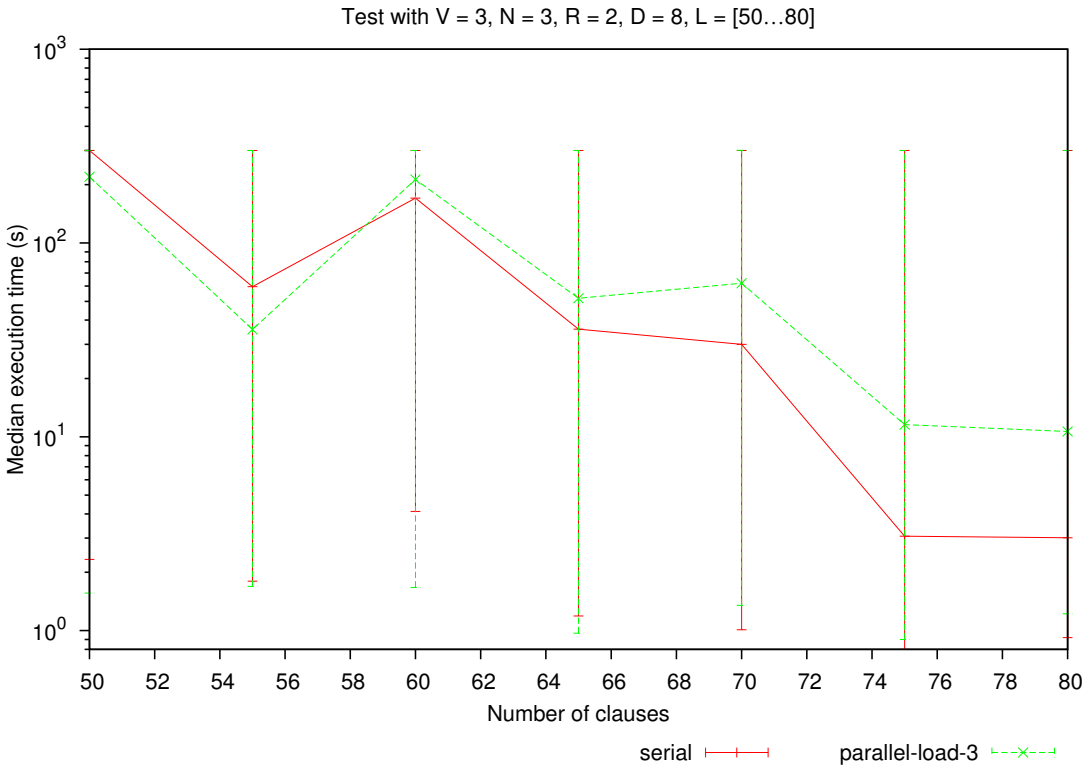

Figura 5.10: Comparación de tiempo de ejecución entre serial y paralelo con 3 Workers

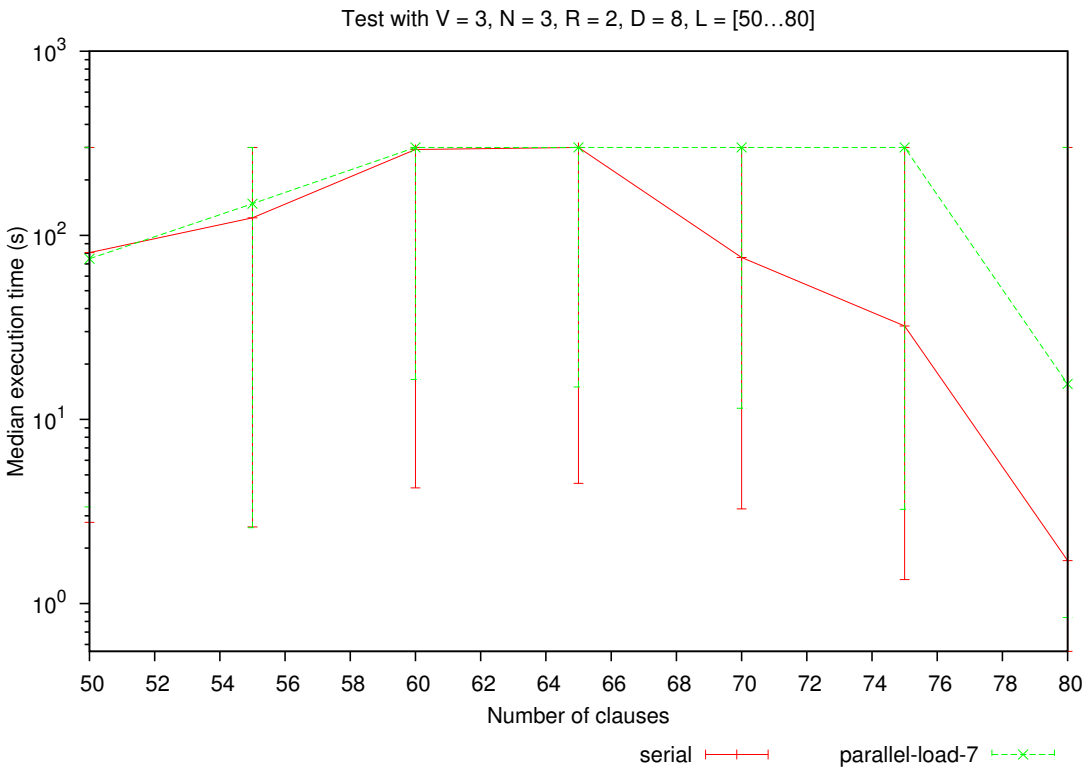

Figura 5.11: Comparación de tiempo de ejecución entre serial y paralelo con 7 Workers

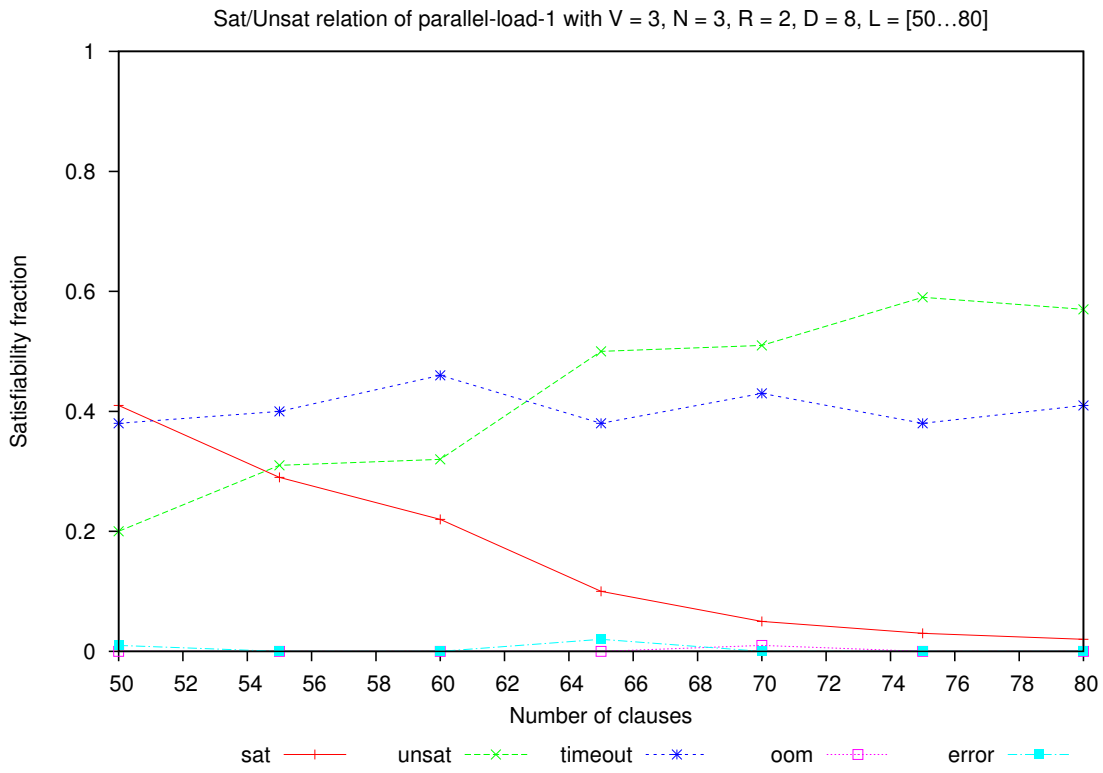

Figura 5.12: Distribución de resultados con 1 Worker

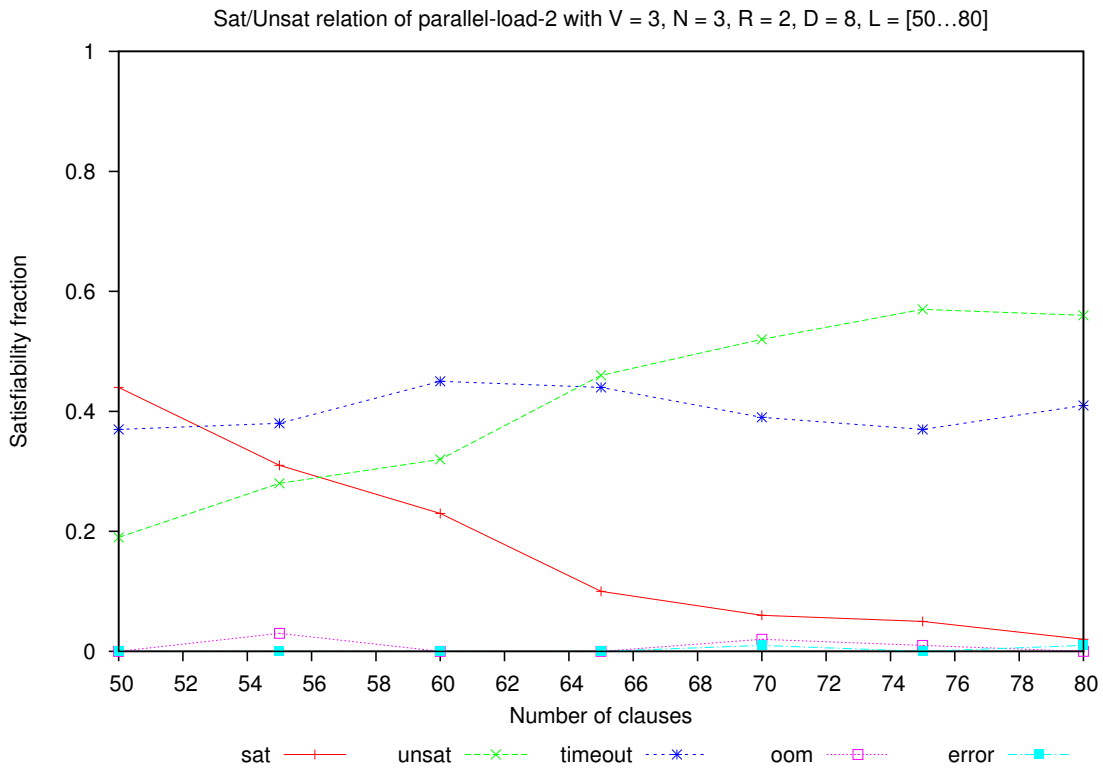

Figura 5.13: Distribución de resultados con 2 Workers

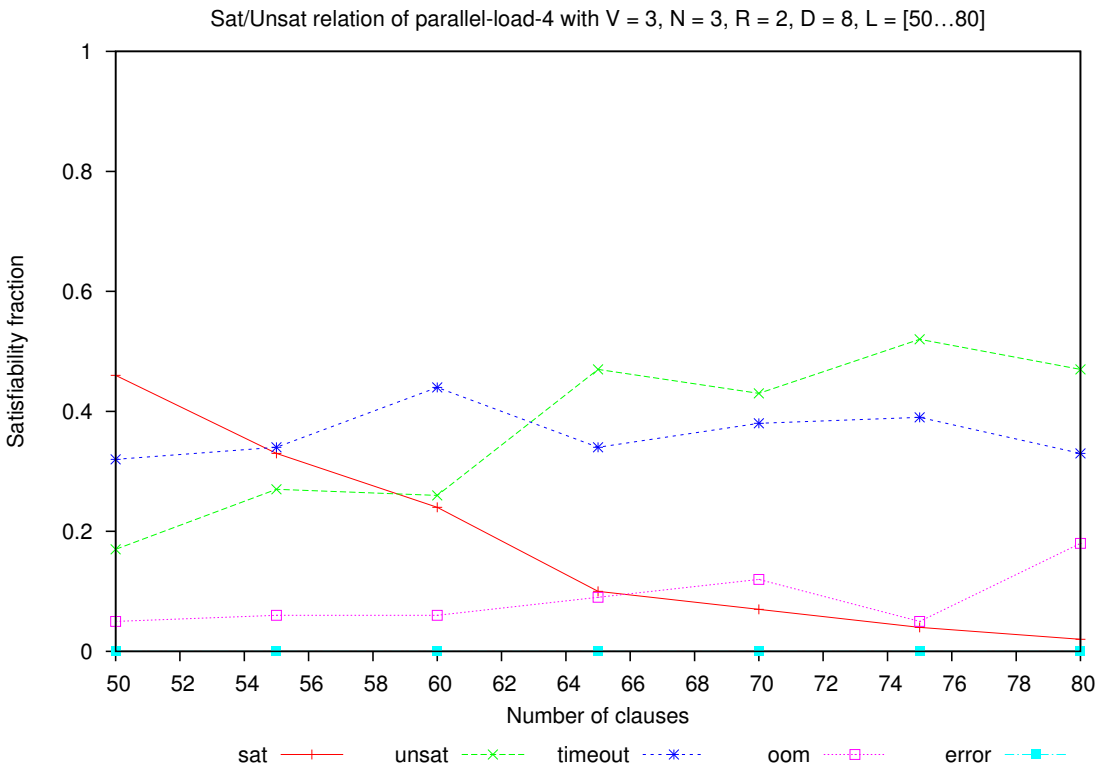

Figura 5.14: Distribución de resultados con 4 Workers

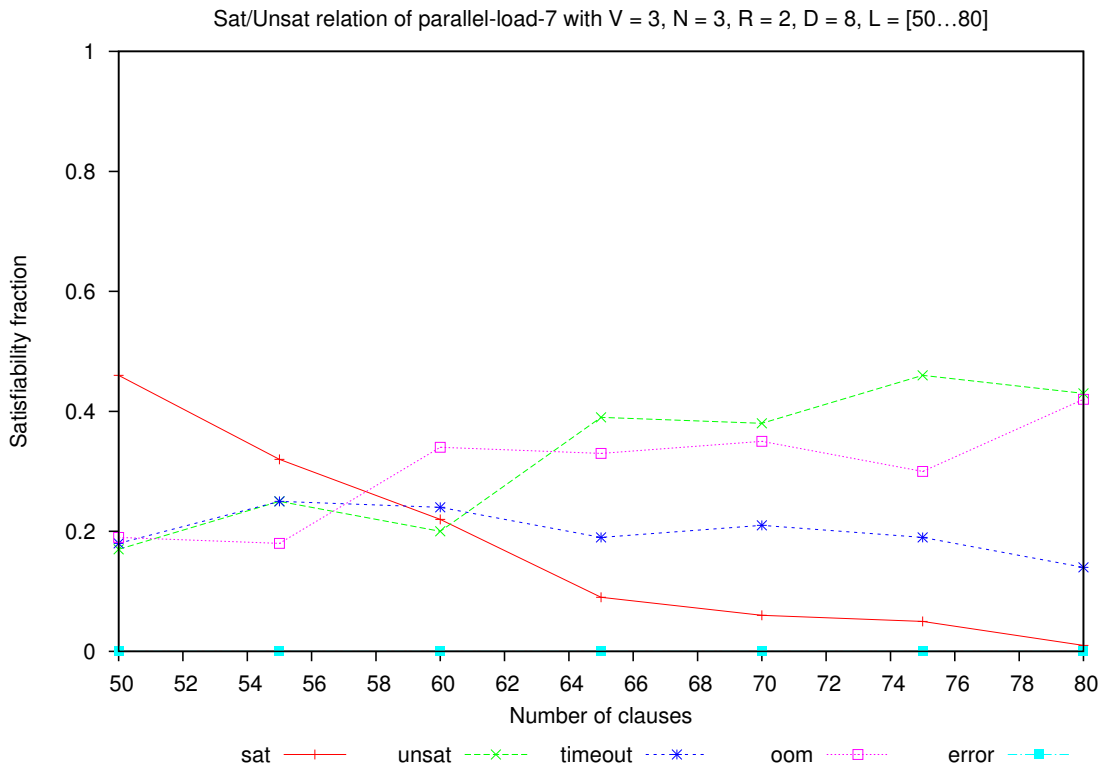

Figura 5.15: Distribución de resultados con 7 Workers

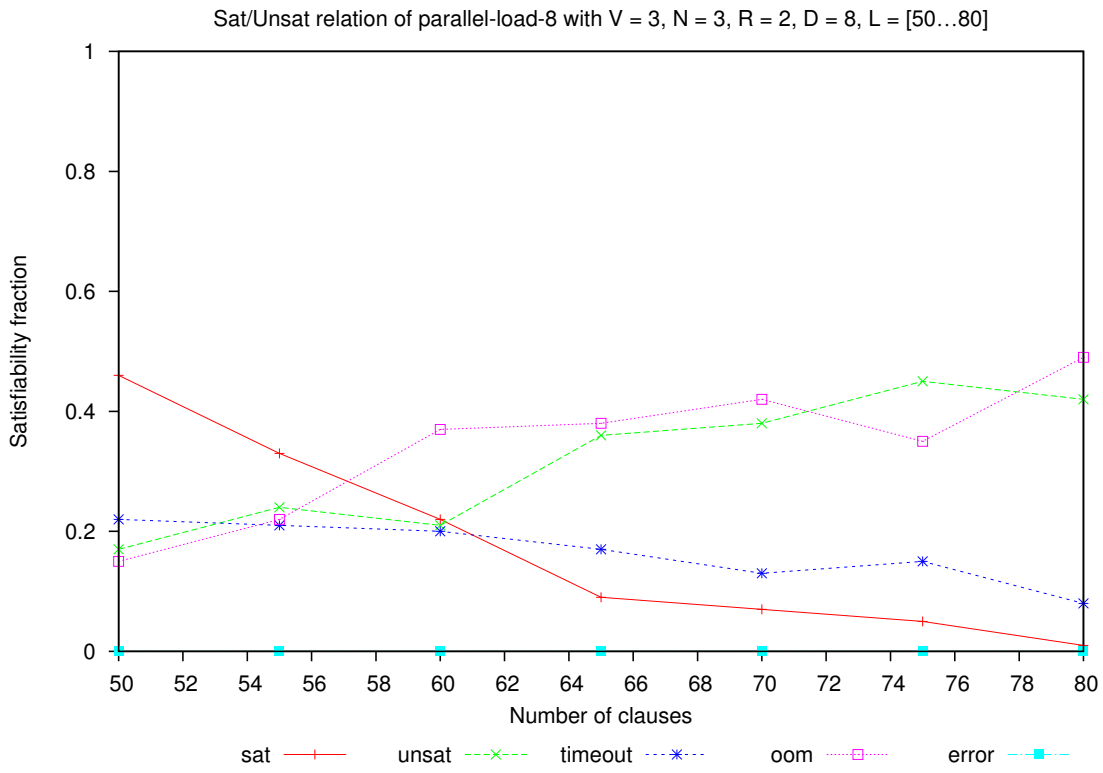

Figura 5.16: Distribución de resultados con 8 Workers

Capítulo 6

Problemas y trabajo futuro

Por último vamos a resumir algunos problemas que encontramos durante la implementación del algoritmo paralelo que pueden llegar a ser disparadores para otros trabajos e investigaciones.

6.1. Cuellos de botella

Cuando se tiene un programa que ejecuta en paralelo, hay que evitar los cuellos de botella. Evitar que algún componente del programa limite el procesamiento del resto de los componentes. Durante el desarrollo de nuestro trabajo encontramos varios de estos. El proceso de subsunción es un ejemplo. Cuando agregamos este proceso vimos que los Workers estaban ociosos mucho tiempo a la espera de nuevas fórmulas para procesar. Se había tornado un cuello de botella en el programa y es por ello que esta alternativa fue desechada.

Si bien en las pruebas que realizamos parecería que los Workers están procesando todo el tiempo y no están limitados por ningún otro componente del programa, es posible que el Dispatcher se transforme en un cuello de botella bajo ciertas circunstancias. Ya avanzados en el desarrollo de la tesis se nos ocurrió una alternativa que contempla la eliminación del Dispatcher. Se podría hacer que los Workers no estén atados a una partición de nominales. Sino que cuando un Worker quiera procesar una cláusula elija sobre qué nominal va a trabajar. De esta manera los Workers serían todos iguales y hasta sería posible aumentar o disminuir la cantidad de estos dinámicamente durante la ejecución del programa. Sin embargo hay ciertas cuestiones que habría que tener en cuenta. Por ejemplo mientras un Worker se encuentre trabajando sobre un nominal, ningún otro debería estar procesando cláusulas con el mismo nominal como etiqueta de la fórmula distinguida. También habría que tener cuidado en el chequeo de subsunción, ya que podría ocurrir una situación similar a la que pasó cuando intentamos subsunción asincrónica en el Dispatcher (4.3). Si bien pensamos que esta opción era factible de implementar, ya habíamos avanzado demasiado en nuestra solución con el Dispatcher y tampoco estábamos convencidos de que fuera a resultar mejor. Sin embargo es una alternativa válida que podría ser explorada en algún trabajo futuro.

6.2. Problemas de memoria

Como vimos en el capítulo anterior, las pruebas realizadas con la versión paralela de HyLoRes terminaron en varias ocasiones con errores por falta de memoria. Esto podría darse por distintos motivos:

- defecto introducido en la nueva versión
- defecto ya existente en la versión serial agravado por la concurrencia
- defecto en el *runtime* de GHC

- real falta de memoria

El primer motivo lo descartamos, ya que hemos realizado algunas corridas con el demostrador serial con número de cláusulas muy grande en el cual también se produjeron errores de falta de memoria. Lo cual nos hace sospechar que es posible que el defecto ya existiese y sólo se hace más evidente cuando hay concurrencia ya que muchas estructuras de datos son replicadas en los distintos procesos. Entonces si hubiera un *leak* de memoria en alguna parte del programa, si esa parte se ejecutara justo en los Workers, la memoria se acabaría más rápido de manera proporcional a la cantidad de Workers. Lamentablemente las herramientas actuales de *profiling* y *debugging* de GHC que nos permitirían dar un diagnóstico más acertado sobre el problema, están muy poco desarrolladas con respecto a la de otros lenguajes como Java o C, con lo cual se dificulta mucho determinar si hay problemas de memoria o de otro tipo. El tercer motivo tampoco puede ser descartado. Hemos visto varias correcciones de defectos en el runtime de GHC relacionados con concurrencia y *garbage collection* en distintas actualizaciones. Por último puede ser que realmente nos estemos quedando sin memoria. El problema que estamos resolviendo tiene complejidad exponencial en espacio. Es decir dado el input, la cantidad de memoria requerida en el peor caso puede ser una función exponencial en función de éste.

El problema de memoria es un ítem que nos quedó sin resolver y podría ameritar un trabajo futuro para tratar de determinar su causa y corregirlo. Creemos que la performance del demostrador paralelo mejoraría notoriamente de ser así.

6.3. Contención de memoria

Existe otra posibilidad por la cual a medida que la cantidad de Workers aumenta la performance se degrada. Puede haber un problema de contención en el acceso a la memoria por parte de los procesadores. En sistemas con múltiples procesadores y memoria compartida, la forma más común de acceso a memoria por parte de cada procesador es utilizando un *bus*. Un bus puede pensarse como un conjunto de cables paralelos que conectan cada CPU con la memoria [Severance and Dowd., 2010]. El bus se comparte entre todos los procesadores, por lo tanto sólo un procesador puede acceder la memoria a la vez a través de éste. Cuando más de un procesador intenta utilizar el bus, sólo uno lo logra y el resto se queda bloqueado hasta que el bus se libere. Esto hace que el bus sea un cuello de botella y se degrade el paralelismo del sistema. Para mejorar el ancho de banda del acceso al bus, se utilizan *caches* locales de memoria en cada procesador. Luego si los datos que un procesador necesita acceder están en el cache local, no es necesario acceder a la memoria y por lo tanto disminuye el tráfico en el bus. Si bien un cache reduce la contención es imposible eliminarla del todo.

Es probable que cuando ejecutamos nuestro programa con 2 Workers, la contención exista, pero la ganancia obtenida por el paralelismo sea mayor a la pérdida de performance que ocasiona ésta. Cuando utilizamos 4 Workers, la contención es mucho mayor dado que hay más procesadores compitiendo por el uso del bus. Luego es posible que la ganancia por ejecutar en paralelo no sea suficiente como para compensar la pérdida de performance ocasionada por la mayor contención.

6.4. Trabajos relacionados

Si bien no existen trabajos previos de resolución en paralelo para lógica híbrida, se pueden encontrar algunos trabajos relacionados con el tema. La mayoría de estos trabajos implementan en paralelo demostradores para lógica proposicional y de primer orden.

En [Kapur and Vandevoorde, 1996] se presenta un método que utiliza interacción con el usuario para probar un teorema. El usuario elige distintos mecanismos de inferencia que luego son ejecutados en paralelo. Si no se obtiene una respuesta en esta primera iteración, el usuario tiene herramientas para ver los resultados parciales e intentar una estrategia diferente.

Otro trabajo que utiliza una estrategia totalmente diferente a la que utilizamos en esta tesis es [Schumann and Letz, 1990]. En este trabajo se presenta una solución que ejecuta un mismo

demostrador con distintos parámetros de manera concurrente. Es decir cada procesador trata de demostrar la misma fórmula con el mismo demostrador, pero utilizando distintos parámetros de configuración del mismo. Luego cuando alguno de los procesadores encuentra la solución le avisa al resto y se termina la ejecución del programa. La ventaja de este *approach* es que puede utilizarse tanto en sistemas con memoria compartida como en un cluster de computadoras, ya que cada proceso no comparte información con el resto. Este mecanismo se podría probar y aplicar con HyLoRes, ya que hay ciertos parámetros como la función de selección que podrían variarse y ejecutar en distintas computadoras a la vez.

Por último podemos mencionar un trabajo donde se paraleliza el algoritmo de la clausulas dada. En [Slaney and Lusk, 1990] se presenta una forma de paralelización del algoritmo de la cláusula dada para saturar un conjunto de clausulas de lógica de primer orden. A diferencia del método que proponemos, este trabajo no particiona los datos ni posee un proceso que distribuye las cláusulas. Utiliza un método similar al que mencionamos anteriormente, donde cada proceso elije un conjunto de cláusulas a procesar. Otra diferencia es que aplica solo subsunción hacia adelante. Esta etapa de subsunción esta dividida en dos partes. Una primera realiza subsunción local en cada proceso. Luego finalmente se hace la subsunción completa, utilizando un mecanismo de lockeo para sincronizar los distintos procesos. En [Lusk *et al.*, 1992] se presenta una implementación de este algoritmo que además introduce subsunción hacia atrás. Utiliza una base de datos que comparten los distintos procesos y es el punto de sincronización entre ellos. En esta implementación no existen procesos especializados, sino que hay un *pool* de procesos y distintas tareas que pueden realizar estos. Una tarea es la generación de nuevas cláusulas aplicando las reglas del cálculo. Otra es el chequeo de subsunción y el agregado de las cláusulas no subsumidas al conjunto de cláusulas en uso. La primer tarea puede ejecutarse de manera concurrente entre más de un proceso, mientras que la de subsunción solo puede ejecutarse por un sólo proceso a la vez.

Bibliografía

[Areces and Gorín., 2008] C. Areces and D. Gorín. Resolution with order and selection for hybrid logics. *Submitted to Journal of Automated Reasoning*, 2008.

[Areces and Heguiabehere., 2003] C. Areces and J. Heguiabehere. hgen: A random CNF formula generator for hybrid languages. In *Proc. of Methods for Modalities 3*, Nancy, France, 2003.

[Areces *et al.*, 2009] C. Areces, D. Gorín, A. Lorenzo, and M. PÃ©rez Rodríguez. Testing provers on a grid – framework description. In *Proceedings of the 22nd International Workshop on Description Logics*, 2009.

[Bachmair and Ganzinger, 1998] L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. In *Automated deduction—a basis for applications, Vol. I*, pages 353–397. Kluwer Acad. Publ., Dordrecht, 1998.

[Bachmair and Ganzinger, 2001] L. Bachmair and H. Ganzinger. Resolution theorem proving. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 19–99. Elsevier, Amsterdam, the Netherlands, 2001.

[Kapur and Vandevoorde, 1996] Deepak Kapur and Mark T. Vandevoorde. Dlp: A paradigm for parallel interactive theorem proving, 1996.

[Kripke, 1959] S. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

[Kripke, 1963a] S. Kripke. Semantic analysis of modal logics I, normal propositional calculi. *Zeitschrift fürr mathemathische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Kripke, 1963b] S. Kripke. Semantical consideration on modal logics. *Acta Philosophica Fennica*, 16:83–94, 1963.

[Lewis, 1918] Clarence Irving Lewis. *A Survey of Symbolic Logic*. Univ. of California Press (Berkeley), Berkeley, 1918. Reprint of Chapters I–IV by Dover Publications, 1960, New York.

[Lusk *et al.*, 1992] Ewing L. Lusk, William McCune, and John K. Slaney. Roo: A parallel theorem prover. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 731–734, London, UK, 1992. Springer-Verlag.

[Prior, 1957] A. Prior. *Time and Modality*. Oxford Unviersity Press, 1957.

[Prior, 1967] A. Prior. *Past, Present and Future*. Oxford University Press, 1967.

[Robinson, 1965] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Schumann and Letz, 1990] J. Schumann and R. Letz. Partheo: A high performance parallel theorem prover. In *In Proceedings of the Tenth International Conference on Automated Deduction*, pages 40–56. Springer-Verlag, 1990.

[Severance and Dowd., 2010] C. Severance and K. Dowd. *Shared-Memory Multiprocessors - Symmetric Multiprocessing Hardware*. Connections Web site, 2010.

[Slaney and Lusk, 1990] John K. Slaney and Ewing L. Lusk. Parallelizing the closure computation in automated deduction. In *Proceedings of the tenth international conference on Automated deduction*, CADE-10, pages 28–39, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Voronkov, 2001] A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning. 1st. International Joint Conference, IJCAR 2001*, number 2083 in LNAI, pages 13–28, Siena, Italy, June 2001.