Prova por Resolução Utilizando Estratégia de Proposições Enquadradas em Paralelo Utilizando OpenMp

Diogo Cezar Teixeira Batista
Universidade Federal do Paraná (UFPR)
Departamento de Informática
Caixa Postal 19.081 - CEP 81.531-980 - Curitiba - PR - Brasil
diogoc@c3sl.ufpr.br

Resumo

Within computational issues, logic requires automatic mechanisms that can prove theorems mapped to real world situations. This work has used the method of proof by refutational resolution combined with some techniques that reduce the search space in a structure that is representing a theorem. These problems are characterized by a combinatorial explosion, which motivated the use of parallel processing techniques, with the OpenMP API a technique was implemented to distribute the work of searching structure aiming to prove a theorem.

1. Introdução

A lógica, em um contexto geral, pretende representar um raciocínio válido. Essa validade depende de uma interpretação que é inserida em um determinado contexto. Por meio de axiomas (símbolos) e regras de inferência, consegue-se mapear algumas situações do mundo real para um contexto lógico, podendo assim, delegar explorações para um sistema automático [3].

Para representar situações do mundo real, gera-se prováveis teoremas compostos por premissas (informações que se sabe sobre o fato) e por conclusões (informações que se deseja inferir).

Por exemplo, pode-se descrever a seguinte situação:

1. Sócrates estaria disposto a visitar Platão, se Platão estivesse disposto a visitá-lo ($P \rightarrow S$);

2. Platão não estaria disposto a visitar Sócrates, se Sócrates estivesse disposto a visitá-lo ($S \rightarrow \neg P$);

3. Platão estaria disposto a visitar Sócrates, se Sócrates não estivesse disposto a visitá-lo ($\neg S \rightarrow P$);

1. pergunta-se: Sócrates está disposto a visitar Platão ou não?

Então o teorema que representa tal situação é dado pela representação lógica no formato clausal da conclusão 1:

$$(P \rightarrow S), (S \rightarrow \neg P), (\neg S \rightarrow P) \vdash S \quad (1)$$

Uma das vertentes da lógica, é a tentativa de se provar se um determinado teorema é ou não válido. Essa prova pode ser obtida de diferentes formas, uma delas é a busca axiomática, que visa encontrar a prova para determinado teorema por meio da manipulação de símbolos por regras de inferência; outra forma é a busca semântica, que a partir de transformações na fórmula original, mantém a equivalência preservando as propriedades do teorema original, reduzindo a fórmula e aplicando a prova em um teorema mais simples [3].

Pelas regras de adequação e completude [3], se um teorema for provado pelas regras da semântica diz-se que esse também pode ser provado pelas regras axiomáticas, assim garante-se que um teorema é valido. Nesse contexto o trabalho desenvolvido visa estabelecer a prova pelas regras semânticas.

As Equação 1 representa a fórmula no contexto da lógica proposicional, que consegue representar simples ações do mundo real. Representações mais complexas estão contidas na lógica de primeira ou segunda ordem, que estão fora do escopo desse trabalho.

Existem algumas técnicas para se provar se um teorema é ou não válido, dentre elas estão: sistema dedutivo de Hilbert, resolução, tableau e sequentes [3]. Para o presente trabalho, propõe-se a utilização do método de prova por resolução, utilizando as técnicas de resolução linear e proposições enquadradas, que serão detalhadas na Seção 2.

A grande dificuldade de realizar a prova por resolução de grandes teoremas (com centenas ou milhares de cláusulas) está na explosão combinatória gerada pelo problema. Provar teoremas pode levar uma quantidade de tempo

inviável, ou até depender de recursos de hardware indisponíveis, o que acaba dificultando a geração de uma resposta em um tempo satisfatório.

Por mais rápidos que os processadores possam ser, estamos atingindo o limite físico da velocidade com que os processadores podem trabalhar. Problemas complexos, como a prova por resolução, demandam uma grande quantidade de processamento e dependem de uma solução eficiente. A programação paralela entra nesse contexto, com a idéia de manter vários processadores trabalhando paralelamente com partes menores de um problema maior [5].

O presente trabalho busca explorar uma solução paralela utilizando a divisão de trabalho por *threads* (componentes que agrupam um determinado trabalho a ser executado) executadas paralelamente em múltiplos microprocessadores. Para isso, foi utilizada uma *Application Programming Interface* (API) chamada OpenMP, que se enquadra para aplicações de memória compartilhada e oferece recursos para facilitar a implementação de técnicas para a programação paralela. A interface não é uma linguagem de programação, mas sim, uma extensão da linguagem C/C++ ou FORTRAN. OpenMP possui diretivas que indicam como o trabalho será dividido entre *threads* e a ordem de acesso aos dados compartilhados [8], [2].

2. Referêncial Teórico

Uma fórmula proposicional é formada por uma ou mais proposições, que podem estar ligadas por conectivos ($\wedge, \vee, \neg, \rightarrow e \leftrightarrow$) [3]. Cada fórmula, quando envolvida em uma interpretação, distribui valores do tipo verdadeiro ou falso para cada uma de suas proposições. Uma proposição com valor verdadeiro é representada por um símbolo, por exemplo, p e sua negação por $\neg p$. Os conectivos regulam como a interpretação será gerada, e cada fórmula gera sempre um resultado final verdadeiro ou falso para a interpretação atribuída.

2.1. Forma Normal Conjuntiva

Segundo [3], uma fórmula proposicional está em Forma Normal Conjuntiva (FNC) se está na seguinte forma:

$$\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n \tag{2}$$

onde, cada α_i é uma cláusula na seguinte forma:

$$\beta_1 \vee \beta_2 \vee ... \vee \beta_m \tag{3}$$

onde, cada β_i é um literal (uma proposição ou sua negação).

Por exemplo: $p \wedge q \vee r \wedge s$.

2.2. Transformação em FNC

Para toda fórmula proposicional existe uma FNC equivalente [3], [4], que pode ser obtida por meio de regras de equivalência: eliminação das implicações, distribuição da negação e distribuição da disjunção. O trabalho parte da premissa que as fórmulas de entrada já se encontram em FNC.

2.3. Resolução

Esse método parte de uma única regra de inferência [4]:

$$\frac{\alpha \vee \Phi \qquad \neg\alpha \vee \Psi}{\Phi \vee \Psi}\ R(\alpha) \tag{4}$$

Se $\alpha \vee \Phi$ é verdadeiro, e $\neg\alpha \vee \Psi$ também é verdadeiro, então ou Φ é verdadeiro ou Ψ ou ambos, então podemos concluir $\Phi \vee \Psi$. $R(\alpha)$ mostra qual o literal foi resolvido.

Sejam $p \vee q \wedge \neg p \vee r$, $p \vee q \wedge \neg p \vee q$ e $p \vee q \wedge \neg p \vee q$ fórmulas em FNC, poderíamos resolve-la da seguinte forma:

1	$p \vee q$		1	$p \vee q$		1	$p \vee q$	
2	$\neg p \vee r$		2	$\neg p \vee q$		2	$\neg p \vee q$	
3	$q \vee r$		3	q		3	q	

(5)

Ao se obter $q \vee q$ pela resolução de $p \vee q$ e $\neg p \vee q$ podemos simplificar somente para q. Mais detalhes estão explicados na Seção 2.3.3.

Seja $p \vee \neg p$ uma fórmula em FNC, poderíamos resolvelas da seguinte forma:

1	p	
2	$\neg p$	
3	$\emptyset$ ou $\perp$	R($\alpha = p$), 1, 2

(6)

Quando não há mais proposições para serem resolvidas, diz-se que se encontrou uma cláusula vazia, que representa uma contradição (*false*) e é também o objetivo do método.

2.3.1. Refutação por Resolução

Seja um conjunto de cláusulas $\mathbb{C}$, provar α parte-se do princípio:

- Aplica-se a regra de resolução sobre o conjunto $\mathbb{C} \cup \neg\alpha$ buscando encontrar uma contradição;
- Se uma contradição for gerada, pode-se dizer que $\neg\alpha$ não é verdade, logo α é verdade, e prova o teorema;
- Se nenhuma contradição for gerada, então o teorema testado não é um teorema da base de dados original.

Para exemplificar a regra de Refutação por Resolução, vamos utilizar a seguinte fórmula:

$$p \vee q \wedge p \rightarrow r \wedge q \rightarrow r \vdash r \tag{7}$$

Nessa fórmula temos 3 premissas ($\{p \vee q\}$,$\{p \rightarrow r\}$,$\{q \rightarrow r\}$) e uma conclusão a ser provada ($\vdash r$).

O primeiro passo é chegar na forma normal conjuntiva (FNC) da fórmula, aplicando as regras de equivalências nas cláusulas necessárias:

$$p \rightarrow r \equiv \neg p \vee r \tag{8}$$
$$q \rightarrow r \equiv \neg q \vee r \tag{9}$$

Adiciona-se como uma quarta premissa a negação da conclusão a ser provada e inicia-se o método de resolução:

1	$p \vee q$	primeira premissa
2	$\neg p \vee r$	segunda premissa
3	$\neg q \vee r$	terceira premissa
4	$\neg r$	negação da conclusão
5	$\neg p$	R($\alpha = r$), 4, 2
6	q	R($\alpha = \neg p$), 5, 1
7	r	R($\alpha = q$), 6, 3
8	$\perp$	R($\alpha = r$), 7, 4
9	r	refutação

(10)

Dessa maneira consegue-se provar r.

2.3.2. Estratégias para Resolução

Ao se aplicar uma estratégia não restringida do método de resolução, a cada passo, encontra-se várias possibilidades de resolução. Resolvendo todos os passos possíveis garante-se que se houver uma cláusula vazia ela será encontrada, entretanto a explosão combinatória torna a implementação inviável. Algumas técnicas de resolução propõem recursos para diminuir o espaço de busca e aumentar a eficiência de um algoritmo de resolução [9], [10].

2.3.3. Simplificação

Durante a resolução das cláusulas, no instante da junção para formar uma cláusula resultante, é comum encontrar a ocorrência de literais repetidos, como por exemplo: $\{p, \neg q, \neg w\}, \{p, \neg q, w\}$.

Nesse caso pode-se conservar apenas uma ocorrência da proposição que poderia se repetir.

2.3.4. Remoção de Tautologias

Uma tautologia é a ocorrência de um literal e sua negação na mesma cláusula, esse tipo de informação de nada adianta na resolução.

Sendo uma cláusula $\{p, \neg p, q\}$, supondo que q já foi resolvido, então para encontrar a cláusula vazia é necessário que haja no conjunto de cláusulas $\neg p$ e p, se elas já estão no conjunto de cláusulas então não é necessário utilizar essa cláusula para efetuar nenhum cálculo, logo essa cláusula pode ser removida.

2.3.5. Remoção de Cláusulas com Literal Puro

Um literal é dito puro, se não existe no conjunto de cláusulas nenhuma ocorrência de seu complementar. Se não há seu complementar, essa cláusula nunca ficará vazia e não ajudará na busca.

2.3.6. Resolução Linear

A estratégia de resolução linear propõe uma ordem para a resolução das cláusulas. Partindo de uma cláusula centro c_0 se obtém na lista de cláusulas, 0, 1 ou mais resolventes, que são origem a uma nova lista r_0. Um membro gerado dessa lista, se torna a nova cláusula centro, e repete-se o procedimento até que a cláusula vazia seja encontrada.

Por exemplo, seja um conjunto de cláusulas:

1	$s \vee r$	
2	$\neg s \vee w$	
3	$\neg r \vee \neg w$	
4	$\neg p \vee t$	
5	$\neg t \vee q$	
6	p	
7	$\neg q$	
8	$r \vee w$	R($\alpha = s$), 1, 2
9	$w \vee \neg w$	R($\alpha = w$), 8, 3
10	*tautologia*	

(11)

No conjunto de cláusulas 11 a cláusula 9 é resolvida utilizando a cláusula 8, uma possível cláusula 10 seria resolvida utilizando a cláusula 9. No a resolução não não foi continuada por encontrar uma tautologia, e como visto na Seção 2.3.4, quando uma tautologia é encontrada, continuar a prova dessa cláusula não chegará na cláusula vazia, então esse método só se torna completo se alguma técnica de *backtrack* for utilizada.

Nesse caso é necessário recuar e verificar se existem outras possibilidades. No exemplo, ao se iniciar com as cláusulas $(1, 2)$ chegou-se a uma situação sem saída, igualmente se a resolução iniciar com as cláusulas $(1, 3)$ ou

$(2, 3)$. Ao resolver a cláusula 4 conseguimos uma situação que leva a cláusula vazia.

2.3.7. Estratégia dos Literais Enquadrados

A estratégia de resolução linear com literais enquadrados (ou *framed*) propõe que os literais resolvidos não sejam removidos das cláusulas resultantes, mas marcados como enquadrados. A estratégia define duas regras que permitem encontrar a cláusula vazia, são elas:

- resolução: se o último literal da cláusula for complementar de um literal p, ou seja $(\neg p)$ enquadrado, então pode-se remover esse literal da lista;
- eliminação: quando se obtém uma cláusula com literal enquadrado não seguido de nenhum literal "não enquadrado" esse também pode ser removido da lista de literais.

Seja um conjunto de cláusulas $\mathbb{C} = \{q \vee p, \neg p \vee r, \neg r \vee \neg p, \neg q\}$, resolve-se aplicando a técnica de resolução linear com literal enquadrado:

1	$q \vee p$		
2	$\neg p \vee r$		
3	$\neg r \vee \neg p$		
4	$\neg q$		
5	$q \vee [p] \vee r$	R($\alpha = p$), 1, 4	(12)
6	$q \vee [p] \vee [r] \vee \neg p$	R($\alpha = r$), 5, 2	
7	$q \vee [p] \vee [r]$	*resolução*	
8	q	*eliminação*	
9	$\perp$	R($\alpha = q$), 8, 4	

2.4. Modelo PRAM

O modelo PRAM (*Parallel Random Access Machine*) representa uma extensão do modelo sequencial RAM (*Random Access Memory*) e é o mais conhecido modelo de computação paralela, representado por um conjunto de processadores que operam de modo síncrono sob controle de um relógio comum [6], [7], [1]. Cada processador é representado por um índice e possui uma memória local própria, podendo comunicar-se com os outros processadores através de uma memória global compartilhada. Com a abordagem PRAM é possível a criação de pseudo-códigos para representar um modelo que descreva o comportamento de um algoritmo a ser executado em paralelo.

3. Implementações

A estratégia adotada para a implementação foi a utilização de estruturas de listas encadeadas para representar as cláusulas. Cada cláusula possui informações que formam uma sêxtupla $C < I, L, F, T, P, N >$:

I é a identificação da cláusula, na estrutura representada por um tipo de dado *int*;

L é o rótulo (*label*) da cláusula, armazena uma *string* que representa a junção de suas proposições, por exemplo $(p \vee q \vee \neg r)$. Na estrutura representada por um tipo de dado *char*;

F é uma dupla $F < A, B >$ que guarda informações sobre quais foram as suas cláusulas pai, A e B são do tipo *int*;

T é a marcação binária de uma tautologia, representada na estrutura por um tipo de dado *int*;

P é uma quádrupla $P < L, N, F, N_e >$:

- L é o rótulo da proposição. Armazena uma string que representa um único literal, por exemplo p. Representado por um tipo de dado *char*;
- N é a marcação binária de negativo da proposição. Na estrutura representado pelo tipo de dado *int*;
- F é a marcação binária de enquadrado (*framed*) da proposição. Na estrutura também representado por *int*;
- N_e é um ponteiro para a próxima proposição;

Cada cláusula armazena um ponteiro para uma estrutura P que guarda suas proposições;

N armazena um ponteiro para a próxima cláusula.

Dentre as estratégias de resolução descritas na Seção 2.3.2 foram implementadas: simplificação, resolução linear, e estratégia dos literais enquadrados. Não foi implementado um mecanismo para remoção de tautologias, mas elas precisam ser identificadas, pois esse é um dos mecanismos que aciona o recurso de *backtrack*.

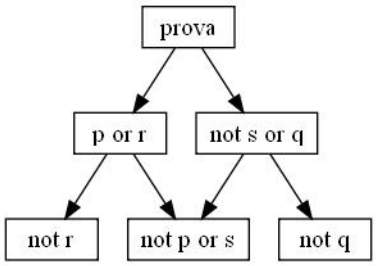
3.1. Idéia Geral

A busca de uma cláusula vazia por resolução linear, gera uma estrutura em árvore. Cada cláusula pode ser combinada com 0, 1 ou n outras cláusulas do conjunto.

Sejam as cláusulas: $p \vee r$, $\neg s \vee q$, $\neg p \vee s$, $\neg q$ e $\neg r$, ao escolher a cláusula 1 $(p \vee r)$ achamos um resolvente em 3 $(\neg p \vee s)$ ou em 5 $(\neg r)$. Mas também poderíamos iniciar a busca pela cláusula 2 $(\neg s \vee q)$ achando resonvente em

3 ($\neg p \vee s$) ou em 4 ($\neg q$), e assim sucessivamente. Dessa forma pode-se encontrar uma ramificação diferente para cada cláusula inicial selecionada.

Figura 1. Estrutura em árvore de cláusulas candidatas



A Figura 1 representa a estrutura em árvore das cláusulas candidatas para resolverem $p \vee r$ e $\neg s \vee q$. A resolução linear propõe uma busca somente por um dos ramos dessa árvore, até que se encontre a cláusula vazia, se a cláusula vazia não for encontrada em todas as sub-ramificações, então deve-se iniciar a busca no próximo ramo da árvore. Em uma implementação sequencial a busca em profundidade ocasiona muito trabalho, caso se tenha que retornar aos primeiros filhos da árvore de busca.

A idéia essencial para as implementações foi a divisão das ramificações por *threds*. Destinou-se p *threads* para resolver a sequência de cláusulas.

Em uma abordagem PRAM, poderíamos obter o seguinte pseudo-código:

Código 1. Abordagem PRAM para resolução linear com adição de *threads*

```
1 while not_empty_clause(c) do
2         proc j, 1 <= j < n, do
3                 c := resolve(c[j],
                      search_denied_clauses(c[j]));
4                 c := mark_resolved(c[j]);
5         end
6         n := count(c);
7 endwhile
```

A solução mostrada no Código 1 propõe a adição de *threads* a medida em que novas cláusulas são adicionadas no conjunto. Para cada nova cláusula se aloca um novo *thread*. As cláusulas estão alocadas na variável c. A Linha 1, garante a execução do código enquanto uma cláusula vazia não for encontrada no banco de cláusulas. Para processadores que vão de 1 a n (com n sendo o número de cláusulas iniciais), se executa as seguintes instruções: *c := resolve(c[j], search_denied_clauses(c[j]))* que adicionam ao banco de cláusulas todas as soluções de $c[j]$ resolvidas com um outro conjunto que retorna os resolventes de $c[j]$ (*search_denied_clauses(c[j])*). O próximo passo é marcar quais foram as cláusulas resolvidas (*c := mark_resolved(c[j])*), que não serão selecionadas em uma próxima iteração. E por fim, se incrementa o número de processadores que irão executar no próximo passo do algoritmo. A criação de *threads* nesse modelo, se dará em proporções de explosão combinatória, entretanto uma única *thread* será responsável por executar a criação dos filhos da cláusula tratada por ela.

Uma segunda abordagem mais refinada e adaptada para uma implementação real propõe:

Código 2. Abordagem PRAM para resolução linear

```
1 proc j, 1 <= j < n, do
2         while not_empty_clause(c) do
3                 c := resolve(c[j],
                      search_denied_clauses(c[j]));
4                 c := mark_resolved(c[j]);
5         endwhile
6 end
```

Na abordagem do Código 2 apresenta-se algo bastante semelhante com o Código 1, entretanto cada *thread* fica responsável por resolver uma ramificação gerada a partir da cláusula de origem que lhe foi definida. Nesse caso não há adição de novos *threads*, o que possibilita uma adaptação do modelo para uma implementação real, pois nela não teríamos disponíveis um número de *threads* igual ao número de cláusulas.

Para isso, criou-se um mecanismo para distribuir as *threads* a medida em que as ramificações forem resolvidas e em paralelo propõe-se a criação de uma lista anexa, que possui um repositório de elementos a serem resolvidos (essa estrutura corresponde a marcação das cláusulas resolvidas) que se chamou de *lista de pendências*. Inicialmente a essa lista inicia com uma cópia das cláusulas iniciais, e a partir de então segue as regras:

1. ao se resolver uma cláusula, ela deve ser removida da lista de pendências;

2. ao ser gerada uma nova cláusula, ela deve ser adicionada na lista de pendências;

Dessa forma se garante que essa lista armazenará todas as cláusulas que ainda precisam ser resolvidas. Ao se sacar o último elemento inserido nessa lista, realiza-se uma busca em profundidade pela árvore. Para iniciar uma busca em largura, basta selecionar o primeiro elemento inserido na lista, entretanto a busca em largura mostra-se ineficiente

para o problema proposto, pois para cada nível gerado, todos os filhos devem ser criados, impossibilitando seu armazenamento em memória.

O método de *backtrack* é garantido pela resolução de todos os elementos da lista de pendências. Quando a próxima cláusula dessa lista a ser resolvida for uma tautologia (Seção 2.3.4) então, ignora-se essa cláusula e parte-se para a resolução da próxima até que a lista esteja vazia ou contenha apenas tautologias.

A lista de pendências indica para toda a árvore quais são as cláusulas a serem resolvidas, entretanto compartilhalha entre as *threads* gera um custo de sincronização que inviabiliza a implementação, por isso, cada *thread* possui uma cópia da lista de cláusulas inicial (lista privada) e uma lista de pendências própria. Dessa forma eliminou-se qualquer tipo de colaboração entre as *threads*, entretanto quase não há custo de sincronização, cada *thread* pode trabalhar de forma independente, e é responsável por resolver apenas a ramificação inicial que lhe foi atribuída, caso essa ramificação não encontre uma cláusula vazia, então uma nova ramificação lhe é designada.

Quando uma thread encontra uma cláusula vazia, necessita comunicar as demais que o trabalho pode ser encerrado, nesse momento, há a necessidade de se criar um ponto de sincronismo. A estratégia adotada foi a utilização de um semáforo com a diretivas do OpenMP `omp_func_lock`, que possibilitam a gerência de sincronismo. Em determinados casos, em testes variados é possível que a a cláusula vazia seja encontrada em lugares diferentes no espaço de busca, pois um *thread* pode ser executados mais rapidamente, ou ter prioridade de execução, para isso foram analisados o conjunto de cláusulas finais para o experimento.

4. Resultados

4.1. Descrição dos Testes

Para a realização dos testes utilizou-se fórmulas proposicionais não satisfatíveis. A busca por uma prova parte de um sistema refutacional e uma formula não satisfatível pode ser provada por esse sistema. As fórmulas testadas foram retiradas do *DIMACS Sat Benchmarks* e estão detalhadas na Tabela 1.

Tabela 1. Fórmulas testadas

Nome	Tipo	Literais	Cláusulas
aim-50-1_6-no-1	aim	50	80
aim-50-1_6-no-3	aim	50	80
aim-50-1_6-no-4	aim	50	80
dubois25	dub	75	200
dubois50	dub	150	400
dubois100	dub	300	800
hole6	hole	42	133
hole7	hole	56	204

Na Tabela 1 estão informações referentes ao nome da cláusula, seu tipo, a quantidade de literais e a quantidade de cláusulas.

Os testes foram realizados na máquina *Alt* que se encontra no Departamento de Informática da UFPR. Sua configuração é composta por 8 núcleos de processamento Intel Xeon E5345 de 2.33GHz e 7GB de memória RAM.

Nos testes, cada fórmula foi testada dez vezes, tendo algumas que levaram alguns minutos para alcançar o resultado, o que impossibilitou experimentos com maior número de repetições. Para essa população foi gerada a média do tempo de execução e o desvio padrão. Os testes foram realizados com 1, 2, 4, 8, 16, 32 e 64 *threads* e a unidade de medida de tempo adotada foi segundos.

4.2. Resultados Obtidos

As tabelas 2 e 3 dispõem os resultados obtidos e contém para cada fórmula, informações correspondentes ao número de *threads* executados. Em cada célula encontram-se: um ou mais números de cláusulas geradas até a obtenção da cláusula vazia e as *threads* em que foram encontradas. A segunda linha indica o tempo gasto para a execução, e a terceira linha o desvio padrão.

5. Discussão dos Resultados

Os gráficos plotados nas Figuras 2 e 3 indicam a quantidade de cláusulas geradas até encontrar a cláusula vazia em relação ao número de *threads*. A relação de tempo de execução por número de *threads* respectivamente.

Figura 2. Gráfico das menores cláusulas geradas

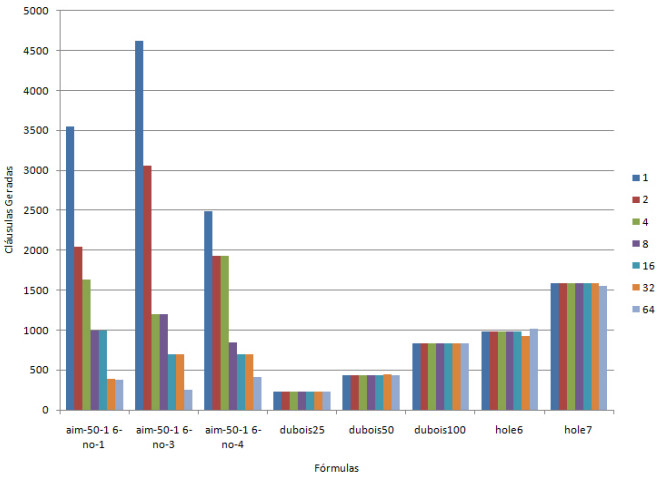


Tabela 2. Resultados para Divisão de Trabalho com 1, 2, 4 e 8 *threads*

	threads			
nome	1	2	4	8
aim-50-1_6-no-1	3548 200,121s 3,322	2047 - 2 30,578s 1,325	1631 - 4 55,773s 4,906	990 - 8 61s 7,446
aim-50-1_6-no-3	4627 171,875s 3,021	3056 - 2 37,757s 3,33	1200 - 3 23,234s 3,90	1200 - 3 77,82s 11,316
aim-50-1_6-no-4	2487 312,121s 3,321	1927 - 2 22,468s 0,441	1927 - 2 53,328s 1,445	845 - 7 30,687s 6,529
dubois25	234 0,001s 0,001	234 - 2 0,062s 0,019	234, 238, 243, 252 - 1, 2, 3, 4 0,156s 0,06	234, 243, 252 - 1,3,4 0,39s 0,118
dubois50	434 0,845s 0,0124	434 - 2 0,125s 0,019	452, 438, 434 - 3, 4, 1 0,375s 0,062	434, 438 - 1,4 0,945s 0,241
dubois100	834 0,125s 0,002	834 - 1 0,226s 0,032	843, 852, 834 - 1, 2, 3 0,804s 0,158	834, 838, 843, 865 - 1, 2, 3, 5 2,054s 0,255
hole6	979 0,875s 0,045	980 - 2 1,601s 0,098	980, 982, 979 - 1, 3, 4 3,992s 0,208	980, 984, 979 - 1,6,4 9,203s 0,460
hole7	1582 2,375s 0,012	1583, 1582 - 1, 2 3,289s 0,166	1584, 1582, 1583 - 1, 2, 3 7,773s 0,242	1583, 1587, 1584 - 2, 3, 6 18,64s 0,395

Tabela 3. Resultados para Divisão de Trabalho com 16, 32 e 64 *threads*

	threads		
nome	16	32	64
aim-50-1_6-no-1	990 - 8 127,429s 17,8	389 - 17 35,187s 0,4	378 - 62 46,851s 3,675
aim-50-1_6-no-3	700 - 10 57,914s 9,024	700, 716 - 10, 26 103,476s 28,51	247 - 58 24,296s 0,549
aim-50-1_6-no-4	692 - 14 50,921s 0,568	692 - 14 104,679s 16,46	413 - 64 52,539 8,060
dubois25	264, 234, 265, 238 - 1,12, 13, 16 0,851s 0,193	264, 238, 234, 265 - 1, 4, 9, 20 1,828s 0,341	234, 252, 265, 238 - 1, 3, 4, 37 3,351s 0,978
dubois50	452, 434, 464, 452 - 3, 1, 12, 3 1,757s 0,347	465, 452, 465, 443 - 1, 3, 9, 3,773s 0,772	438, 434, 464, 465 - 4, 1, 12, 21 8,429s 1,017
dubois100	834, 864, 865 - 9, 12, 4 4,312s 0,63	834, 864, 865 - 1, 28, 32 9,375s 1,018	864, 865, 834 - 1, 17, 48 16,648s 1,575
hole6	1019, 979, 973 - 1, 5, 7 24,539s 3,107	1019, 931 - 7, 3 51,539s 4,799	1019 - 7 105,148s 16,298
hole7	1588, 1586, 1583 - 2, 5, 7 43,71s 15,364	1588, 1583, 1587 - 2, 6, 7 129,656s 27,591	1558, 1586 - 7, 5 273,929s 59,808

A Figura 3 mostra um gráfico, no qual em alguns casos (*aim-50-1_6-no-1*, *aim-50-1_6-no-3* e *aim-50-1_6-no-4*) é notória a redução de tempo a medida em que se adiciona *threads*, entretanto em um cenário geral isso não foi observado pois, cada *thread* cuida de determinada ramificação de cláusulas, e para cada uma dessas ramificações são gerados números diferentes de filhos. A geração de todos os filhos garante que o método será completo, entretanto o custo de geração em um ambiente paralelo de memória compartilhada, gera concorrência na escrita, o que aumenta o tempo de execução em alguns casos.

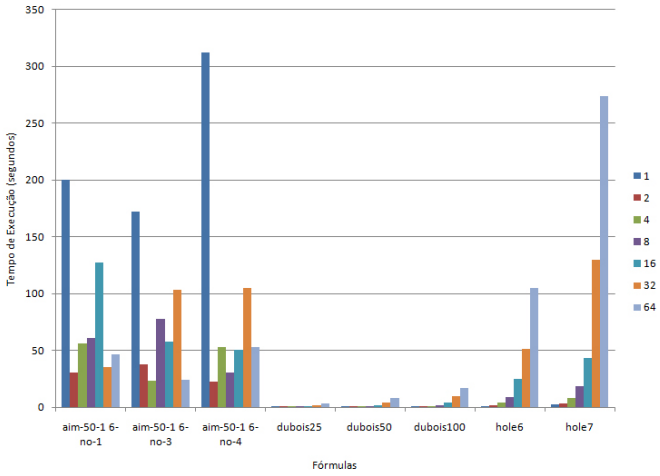
A Tabela 4 ilustra a quantidade de filhos gerados a partir da soma de todos os filhos por todos os ramos para as fórmulas *aim-50-1_6-no-3*, *hole6* e *dubois25* bem como a média de filhos encontrados pelas *threads*. Os dados foram computados para 1, 2, 4, 8, 16, 32 e 64 *threads*.

Nos casos em que a média de cláusulas e a soma dos filhos é menor, o tempo de execução também diminui. Para a fórmula *aim-50-1_6-no-3* a execução com 64 threads foi mais rápida pois, apesar de gerados vários filhos (15241) a

Tabela 4. Soma do número de filhos e média de filhos geradas

		1	2	4	8	16	32	64
aim-50-1_6-no-3	soma	5629	6843	5730	11781	13048	26138	15241
	média	5629	3422	1433	1473	816	817	238
hole6	soma	1073	2132	3844	7249	13897	29395	54186
	média	1073	1066	961	906	869	919	874
dubois25	soma	34	71	101	295	681	1800	2601
	média	34	36	25	37	43	56	41

Figura 3. Gráfico de tempo de execução



média do número de cláusulas geradas ficou em (238). As fórmulas *hole6* e *dubois25* não apresentam uma ganho de *speedup* pois em geral o custo de criação dos filhos cresce linearmente e a redução da média de cláusulas geradas não diminui a ponto de tornar a execução do algoritmo mais rápida.

A forma de construção das fórmulas influencia diretamente no tempo necessário para encontrar uma cláusula vazia, nem sempre o número de literais ou cláusulas é proporcional ao tempo de busca. Por exemplo, as fórmulas do tipo *aim*, que têm cerca de 80 cláusulas, precisaram de um maior esforço para encontrar uma cláusula vazia, enquanto as fórmulas do tipo *dubois*, especificamente a fórmula *dubois100* apresentam 10 vezes mais cláusulas que as do tipo *aim*, e mesmo assim são executadas em um tempo muito menor. Isso se deve a rapidez com que a cláusula vazia é encontrada no espaço de busca. Outro fator que influencia no *speedup* é a quantidade filhos que cada cláusula necessita gerar, pois, quanto mais filhos a serem gerados maior o tempo de execução do algoritmo.

A variação no desvio padrão de alguns testes se dá pela velocidade em que determinados *treads* são executados, caso um *thread* que leve ao resultado mais rapidamente seja executado primeiro ou ganhe prioridade na gravação dos dados em memória, a execução do programa levará menos tempo. Por exemplo, na fórmula *aim-50-1_6-no-4* em média as ocorrências para a execução com 32 threads ficam na casa dos 104 segundos, entretanto em uma das ocorrências o programa consegue terminar em 52,75 segundos.

6. Conclusões

A prova por resolução é um problema que tem como característica a explosão combinatória. Resolvê-lo em um tempo viável depende de quais estratégias são adotadas. Algumas dessas estratégias foram implementadas nesse trabalho, entre elas: resolução linear, literais enquadrados, simplificação e duas abordagens de processamento paralelo.

Apesar do processamento paralelo, a geração de todos os filhos da árvore de prova, mostrou que a adição de *threads* em geral, não aumenta o *speedup* do algoritmo. Pois o custo para criação de todas as combinações de cláusulas, aumenta proporcionalmente a medida em que são adicionados *threads* no sistema. Entretanto ao se aumentar o número de buscas em paralelo pelo espaço de cláusulas, pode-se observar que a cláusula vazia pode ser encontrada por um caminho mais curto.

6.1. Trabalhos Futuros

É possível explorar outro tipo de paralelismo, ao se utilizar *threads* específicos para aplicar estratégias de resolução descritas na Seção 2.3.2, com isso esses *threads* ficam responsáveis por diminuir o espaço de busca e remover cláusulas que não iriam servir para a resolução do problema.

A criação de todas as cláusulas tornou o algoritmo inviável, entretanto pode-se aplicar métodos para que os filhos sejam gerados somente caso um *backtrack* seja necessário, o que tornaria o algoritmo mais eficiente.

Uma outra abordagem também propõe um método relaxado, que abandona ramificações que não foram resolvidas em determinado tempo, essa abordagem aproveita melhor o paralelismo, pois como visto nos resultados, ao testar uma fórmula com vários *threads*, a cláusula vazia é encontrada em um menor tempo.

Referências

[1] C. Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009.

[2] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[3] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, ny, 1972.

[4] M. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann, 1993.

[5] E. Humenay, D. Tarjan, and K. Skadron. Impact of process variations on multicore performance symmetry. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, San Jose, CA, USA, 2007. EDA Consortium.

[6] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[7] B. Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[8] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[9] E. Rich and K. Knight. *Inteligência Artificial*. Makron Books, 2 edition, 1995.

[10] R. S and N. P. *Artificial Intelligence. A Modern Approach*. Prentice Hall, 1995.