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

Diogo Cezar Teixeira Batista
Universidade Federal do Paraná

6 de junho de 2010

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 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.

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

Por exemplo, pode-se descrever a seguinte situação:

premissas
1. Sócrates estaria disposto a visitar Platão, se Platão estivesse disposto a visitá-lo;
2. Platão não estaria disposto a visitar Sócrates, se Sócrates estivesse disposto a visitá-lo;
3. Platão estaria disposto a visitar Sócrates, se Sócrates não estivesse disposto a visitá-lo;

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

Com tais informações, é possível mapear cada uma das sentenças para a lógica, gerando então um teorema. Assumindo que:

- Sócrates é representado por S;
- Platão é representado por P;
- a premissa 1 é representada por $P \rightarrow S$;
- a premissa 2 é representada por $S \rightarrow \neg P$;
- a premissa 3 é representada por $\neg S \rightarrow P$;

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

$$(P \rightarrow S), (S \rightarrow \neg P), (\neg S \rightarrow P) \vdash S \tag{1.1}$$

A conclusão ainda pode ser representada pela seguinte notação:

$$P \rightarrow S \wedge S \rightarrow \neg P \wedge \neg S \rightarrow P \vdash S \tag{1.2}$$

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, 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 Equações 1.1 e 1.2 representam fórmulas 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;
- sequentes;

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 no Capítulo 2.

A grande dificuldade de realizar a prova por resolução de grandes teoremas (com centenas ou milhares de cláusulas) está na explosão combinatória que esse tipo de problema gera. Provar teoremas pode levar um tempo muito grande, ou até depender de recursos de hardware indisponíveis, o que acaba tornando o problema inviável para implementações sequenciais.

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].

Dentre as maneiras de paralelismo estão:

- computadores inter-conectados por uma rede;
- processadores de múltiplos núcleos;
- computadores com p processadores;
- placa gráfica *Graphics Processing Unit* (GPU), aproveitando micro-processadores.

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 micro-processadores. Para isso, foi utilizada uma *Application Programming Interface* (API) chamada OpenMP, que se enquadra para aplicações de memória compartilhada e facilita 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 [7], [2].

O Capítulo 2 detalha os conceitos teóricos da lógica proposicional, da prova por resolução e de algumas estratégias para resolução. O Capítulo 3 descreve as implementações propostas para a resolução linear com literais enquadrados em paralelo utilizando OpenMP. No Capítulo 4 estão os testes realizados, mostra-se os resultados para os experimentos de Colaboração entre Cláusulas e Divisão de Trabalho por Threads e ainda aborda uma discussão sobre esses resultados. Por fim o Capítulo 5 mostra as conclusões do trabalho e os trabalhos futuros.

Capítulo 2

Referencial Teórico

Esse capítulo discute os conceitos teóricos da lógica proposicional, da prova por resolução e de algumas estratégias para resolução.

2.1 Lógica Proposicional

A lógica proposicional é um sistema no qual suas fórmulas representam proposições (valores verdadeiros ou falsos) interligadas por conectivos lógicos. Esse sistema é formado por [4]:

- linguagem: se refere ao modo como se escreve uma sentença;
- semântica: se refere ao estudo do significado;
- axiomática: se refere ao conjunto de premissas iniciais e das proposições que são derivadas desse conjunto.

2.1.1 Linguagem

A linguagem proposicional é formada por símbolos que irão representar nossas sentenças, normalmente representadas por letras minúsculas do alfabeto, possívelmente indexadas [4]:

São exemplos: $p, q, r, p_1, q_1, r_1 \ldots$

2.1.1.1 Conectivos

São os elementos que executam as operações com os símbolos proposicionais [3].

- $\wedge$ - corresponde ao conectivo *e* (conjunção);
- $\vee$ - corresponde ao conectivo *ou* (disjunção);

- $\neg$ - corresponde ao conectivo *não* (negação);
- $\rightarrow$ - corresponde ao conectivo de atribuição ou *implica em* (implicação);
- $\leftrightarrow$ - corresponde ao conectivo *se e somente se* ou *bi-implicação* (dupla implicação);

2.1.2 Fórmulas

Uma fórmula proposicional é formada por uma ou mais proposições, que podem estar ligadas por conectivos [3].

Por exemplo: $p \vee q$.

2.1.3 Semântica

O contexto semântico refere-se ao significado. Podemos atribuir uma interpretação para uma fórmula proposicional, que valora as proposições com verdadeiro ou falso [3].

As soluções possíveis para $\{p, q\}$ são:

$$I(p) = 0, I(q) = 0$$
$$I(p) = 1, I(q) = 1$$
$$I(p) = 1, I(q) = 0$$
$$I(p) = 0, I(q) = 1$$

2.1.4 Axiomática

Propõe a manipulação da fórmulas por meio de regras axiomáticas e de inferência, a axiomática de Hilbert é um exemplo de manipulação axiomática [3].

2.1.5 Forma Normal Conjuntiva

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

$$\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n \tag{2.1}$$

onde, cada α_i é uma cláusula na seguinte forma:

$$\beta_1 \vee \beta_2 \vee ... \vee \beta_m \tag{2.2}$$

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

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

2.1.5.1 Transformação em FNC

Para toda fórmula proposicional existe uma FNC equivalente [3], [4], que pode ser obtida por meio das seguintes regras de equivalência:

1. Eliminação das implicações
   $\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$
   $\alpha \leftrightarrow \beta \equiv (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$
   $\neg(\alpha \leftrightarrow \beta) \equiv (\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)$

2. Distribuição da negação:
   $\neg\neg\alpha \equiv \alpha$
   $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
   $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$

3. Distribuição da disjunção:
   $\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

O trabalho parte da premissa que as fórmulas de entrada já se encontram em FNC.

2.2 Resolução

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

$$\frac{\alpha \vee \Phi \qquad \neg\alpha \vee \Psi}{\Phi \vee \Psi}\, R(\alpha) \tag{2.3}$$

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.

Seja $p \vee q \wedge \neg p \vee r$ uma fórmula em FNC, poderíamos resolve-las da seguinte forma:

$$\begin{array}{l|ll} 1 & p \vee q & \\ 2 & \neg p \vee r & \\ \hline 3 & q \vee r & \text{R}(\alpha = p),\ 1,\ 2 \end{array} \tag{2.4}$$

Seja $p \vee q \wedge \neg p \vee q$ uma fórmula em FNC, poderíamos resolve-las da seguinte forma:

$$\begin{array}{l|ll} 1 & p \vee q & \\ 2 & \neg p \vee q & \\ \hline 3 & q & \text{R}(\alpha = p),\ 1,\ 2 \end{array} \tag{2.5}$$

É importante lembrar que ao se obter $q \vee q$ podemos simplificar somente para q. Mais detalhes estão explicados na Seção 2.2.2.1.

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

$$\begin{array}{l|ll} 1 & p & \\ 2 & \neg p & \\ \hline 3 & \emptyset \text{ ou } \bot & \mathrm{R}(\alpha = p),\ 1,\ 2 \end{array} \tag{2.6}$$

E nesse ponto encontramos uma cláusula vazia, que representa uma contradição (*false*) e é também o objetivo do método.

2.2.1 Refutação por Resolução

O princípio da resolução é uma regra de inferência que dá origem a uma técnica de demonstração por refutação para sentenças e inferências da lógica proposicional e da lógica de primeira ordem [3].

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, nada se pode dizer sobre o teorema.

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

$$p \vee q \wedge p \rightarrow r \wedge q \rightarrow r \vdash r \tag{2.7}$$

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:

$$p \rightarrow r \equiv \neg p \vee r \tag{2.8}$$

$$q \rightarrow r \equiv \neg q \vee r \tag{2.9}$$

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	$\quad \neg r$	negação da conclusão
5	$\quad \neg p$	R($\alpha = r$), 4, 2
6	$\quad q$	R($\alpha = \neg p$), 5, 1
7	$\quad r$	R($\alpha = q$), 6, 3
8	$\quad \bot$	R($\alpha = r$), 7, 4
9	r	refutação

(2.10)

Dessa maneira consegue-se provar r.

2.2.2 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 [8], [9].

2.2.2.1 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.

1	$p \vee \neg q \vee \neg w$		1	$p \vee \neg q \vee \neg w$		
2	$p \vee \neg q \vee w$		2	$p \vee \neg q \vee w$		
3	$p \vee \neg q$	correto	3	$p \vee \neg q \vee p \vee \neg q$	errado	

(2.11)

2.2.2.2 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.

2.2.2.3 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.

2.2.2.4 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: $\mathbb{C} = \{q \vee p, \neg p \vee r, \neg r \vee \neg p, \neg q\}$

1	$q \vee p$	
2	$\neg p \vee r$	
3	$\neg r \vee \neg p$	
4	$\neg q$	
5	$\quad p$	$\mathrm{R}(\alpha = q)$, 1, 4
6	$\quad r$	$\mathrm{R}(\alpha = p)$, 5, 2
7	$\quad \neg p$	$\mathrm{R}(\alpha = r)$, 6, 3
8	$\quad \bot$	$\mathrm{R}(\alpha = \neg p)$, 7, 5

(2.12)

Como visto na Seção 2.2.2.2, 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.

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	$\quad r \vee w$	$\mathrm{R}(\alpha = s)$, 1, 2
9	$\quad w \vee \neg w$	$\mathrm{R}(\alpha = w)$, 8, 3
10	$\quad \mathit{tautologia}$	

(2.13)

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.

Se uma tautologia for eleita como a cláusula a ser resolvida, o *backtrack* deve ser acionado, pois não é possível chegar a uma cláusula vazia partindo de uma tautologia.

2.2.2.5 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.

Utilizando o exemplo da resolução 2.12, 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
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

(2.14)

Capítulo 3

Implementações

Nesse capítulo são mostradas as implementações propostas para a resolução linear com literais enquadrados em paralelo utilizando OpenMP. Uma visão geral da estrutura da solução está na Seção 3.1, a primeira abordagem parcial contendo um método de colaboração entre cláusulas está na Seção 3.2 e uma segunda abordagem contendo uma solução completa com a divisão do trabalho por *threads* está na Seção 3.3.

3.1 Estrutura da Solução

A estratégia adotada para a implementação foi a utilização de estruturas de listas encadeadas que representam 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.

Essas estruturas são preenchidas por um arquivo de entrada que aceita dois tipos de formatações, uma delas criada de forma empírica e outra utilizada para testar as fórmulas maiores que segue o padrão *DIMACS Challenge*. Seja uma FNC: $p \vee q \wedge \neg p \wedge \neg q$, os arquivos de entrada aceitos são:

Código 3.1: Exemplo de arquivo de entrada

```
1 p or q
2 not p
3 not q
```

Código 3.2: Exemplo de arquivo de entrada no formato DIMACS Challenge

```
1 p q
2 -p
3 -q
```

A saída do programa assume dois tipos:

- resumido: apresenta uma lista de informações relevantes como: nome da fórmula, binário que mostra se achou ou não a cláusula vazia, a identificação da cláusula em que a cláusula vazia foi encontrada, o número de threads que participaram da busca, o número de *backtracks* efetuados, qual foi a thread que achou a cláusula vazia e o tempo em segundos de execução;
- extendido: mostra o conjunto completo de cláusulas geradas até encontrar a cláusula vazia, bem como o tempo de execução em segundos.

Para aumentar a flexibilidade dos testes, o programa recebe 2 parâmetros obrigatórios que indicam: a fórmula de entrada a ser processada e o número de *threads* que irão processar, respectivamente. Com isso conseguiu-se a criação de *shell scripts* para automatizar a bateria de testes.

Dentre as estratégias de resolução descritas na Seção 2.2.2 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*.

3.1.1 Idéia Geral

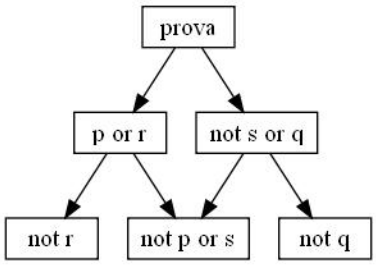
A busca de uma cláusula vazia por resolução linear, gera uma estrutura em árvore. Cada cláusula pode ser combinada com 0, 1 ou n outras cláusulas do conjunto.

Sejam as cláusulas:

$$\begin{array}{c|l} 1 & p \vee r \\ 2 & \neg s \vee q \\ 3 & \neg p \vee s \\ 4 & \neg q \\ 5 & \neg r \end{array} \tag{3.1}$$

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 3.1: Estrutura em árvore de cláusulas candidatas



A Figura 3.1 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 em PRAM [6], [1] poderíamos obter os seguintes pseudo-códigos:

Código 3.3: 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);
5           end
6           n := count(c);
7   endwhile
```

Em um modelo conceitual, a solução mostrada no Código 3.3 propõe a adição de threads a medida em que novas cláusulas são adicionadas no conjunto. Para cada nova cláusula se aloca um novo *thread*. No pseudo-código, 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 adiciona ao banco de cláusulas todas as soluções de $c[j]$ resolvidas com um outro conjunto que retorna os resolventes de $c[j]$. O próximo passo é marcar quais foram as cláusulas resolvidas, 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 também 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 3.4: 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);
5           endwhile
6   end
```

Na abordagem do Código 3.4 apresenta-se algo bastante semelhante com o Código 3.3, entretanto nessa abordagem se define um j *threads* para tratarem de n cláusulas. 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.

Em uma implementação real não teríamos disponíveis um número de *threads* igual ao número de cláusulas. Para a implementação, criou-se um mecanismo para distribuir as *threads* a medida que as ramificações forem resolvidas e em paralelo propõe-se a criação de uma lista anexa a lista de cláusulas, que possui um repositório de elementos a serem resolvidos (essa estrutura corresponde a marcação das cláusulas resolvidas). Inicialmente a lista de pendências inicia com uma cópia das cláusulas iniciais, e 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.

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 2.2.2.2) 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 compartilha-lha entre as *threads* gera um custo de sincronização que inviabiliza a implementação, por isso gerou-se duas abordagens de implementação: Colaboração entre Cláusulas e Divisão de Trabalho por *Threads* que serão explicadas nas Seções 3.2 e 3.3 respectivamente.

3.2 Colaboração entre as Cláusulas

Nessa abordagem, eliminou-se o recurso de lista de pendências, o que ocasionou uma implementação incompleta. A idéia inicial foi analisar o comportamento de uma possível colaboração entre as cláusulas resolvidas de um *thread* com as cláusulas resolvidas de outro. Cada *thread* fica responsável por sacar um ramo a ser resolvido, e todas as cláusulas resolvidas são compartilhadas entre as *threads*. Uma *thread* não se preocupa caso não ache uma solução, sua participação é a geração das cláusulas. Cada uma delas, também não se preocupa em explorar todas as possibilidades de resolução de uma cláusula, mas assume que a primeira já é suficiente. Esse procedimento torna a abordagem incompleta, entretanto pendurar todas as cláusulas geradas, inviabiliza a colaboração entre as cláusulas.

Supondo que o um *thread* t_1 precise de uma cláusula $\neg r$ para resolver a cláusula atual que é r, linearmente $\neg r$ seria encontrado apenas nas próximas cláusulas ou até mesmo não encontrado nessa ramificação, entretanto a *thread* t_2 resolveu $p \vee r$ e achou r e consequentemente compartilhou na lista global de cláusulas, então a *thread* t_1 encontrará a cláusula vazia em um tempo consideravelmente menor.

Com essa abordagem foi necessário adotar um sistema de sincronização no momento de se inserir uma nova cláusula resolvida no banco de cláusulas, o sistema adotado foi a utilização de um semáforo com a diretivas do OpenMP `omp_func_lock`, que possibilitam a gerência de sincronismo.

Essa característica de colaboração não pode ser aplicada em uma estrutura completa, pois cada resolução gera inúmeras cláusulas a serem sincronizadas com o banco de cláusulas global, o que gera um alto custo de sincronização e alocação de espaço em memória.

3.3 Divisão de Trabalho por *Threads*

Nessa abordagem implementou-se o recurso de lista de pendências, garantindo uma implementação completa. 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 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.

Nessa implementação, apenas um ponto necessita de sincronismo, é no momento de comunicar aos outros *threads* que a cláusula vazia foi encontrada. Como isso encerra o processamento, as demais *threads* também irão encerrar o processamento. A mesma estratégia do OpenMP `omp_func_lock` foi utilizada para garantir o sincronismo.

Capítulo 4

Resultados

Esse capítulo descreve os testes realizados, mostra os resultados para os experimentos de Colaboração entre Cláusulas e Divisão de Trabalho por Threads e aborda uma discussão dos resultados encontrados, que estão respectivamente nas Seções 4.1, 4.2, 4.3 e 4.4

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, uma forma não satisfatível contempla essa condição. As fórmulas testadas foram retiradas do *DIMACS Sat Benchmarks*, as selecionadas estão presentes na Tabela 4.1.

Tabela 4.1: Fórmulas testadas

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

Na Tabela 4.1 estão informações referentes ao nome da cláusula, seu tipo, a quantidade de literais, a quantidade de cláusulas e se é ou não satisfatível.

Formulas do mesmo tipo significam que foram geradas da mesma forma, no qual:

- *aim*: de Eiji Miyano <miyano@csce.kyushu-u.ac.jp>, geradas automaticamente;
- *dub*: de Olivier Dubois <dubois@laforia.ibp.fr>, geradas automaticamente;
- *hole*: de John Hooker <jh38+@andrew.cmu.edu>, problema do buraco de pombo. O problema pergunta se é possível colocar $n + 1$ pombos em n buracos sem ter dois pombos no mesmo buraco.

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.

No teste de Colaboração entre Cláusulas, as fórmulas da Tabela 4.1 foram testadas cem vezes, com isso calculou-se a média de tempo e o desvio padrão para a população gerada. Cada fórmula foi testada para 2, 4, 8 e 16 *threads*. No teste de Divisão de Trabalho por *Threads*, cada fórmula foi testada apenas dez vezes, pois algumas fórmulas demoraram alguns minutos para chegar ao resultado, o que impossibilitou um teste com mais repetições. Para essa população também foi gerado a média do tempo e o desvio padrão. Para essa abordagem utilizou-se testes com 1, 2, 4, 8, 16, 32 e 64 *threads*. Nos testes realizados a unidade de medida de tempo adotada foi segundos.

4.2 Resultados para Colaboração entre Cláusulas

Para garantir que o sistema chegasse a um resultado em um tempo considerável, pelo método ser incompleto, se estabeleceu a quantidade máxima de 3500 cláusulas resolvidas, caso não se ache a solução até esse limite, então nada se pode dizer sobre o teorema, mas caso a cláusula vazia seja encontrada, então o teorema é provado.

A Tabela 4.2 mostra os resultados obtidos para o teste de Colaboração entre Cláusulas:

Tabela 4.2: Resultados para Colaboração entre Cláusulas para 2, 4, 8 e 16 *threads*

	threads			
nome	2	4	8	16
aim-50-1_6-no-1	617-1628 (97%) 0,119s 0,071	630-3306 (43%) 1,978s 1,55	1400-3479 (39%) 2,791s 1,447	1893-3438 (21%) 2,831s 1,446
aim-50-1_6-no-3	934-1149 (86%) 0,221s 0,037	406-3047 (80%) 0,402s 0,634	1286-3485 (34%) 2,195s 1,748	1249-3465 (41%) 2,388s 1,52
aim-50-1_6-no-4	969-2068 (95%) 0,610s 0,257	802-3458 (93%) 1,469s 1,095	620-3492 (56%) 1,699s 1,5	1385-3295 (41%) 2,391s 1,143
dubois25	1136-1384 (89%) 0,672s 0,169	1616-2566 (69%) 3,345s 1,274	1420-3303(19%) 4,548s 2,456	1977-3360 (8%) 6,234s 2,75
dubois50	-	-	-	-
dubois100	-	-	-	-
hole6	156-302 (98%) 0,031s 0,001	216-441 (91%) 0,042s 0,621	236-783 (95%) 0,981s 0,168	591-949 (87%) 1,041s
hole7	244-476 (89%) 0,032s 0,001	239-518 (78%) 0,077s 0,141	417-936 (81%) 0,712s 0,787	722-1354 (21%) 0,987s 1,144

A Tabela 4.2 mostra as fórmulas testadas e para cada uma delas os resultados obtidos para a execução com 2, 4, 8 e 16 *threads* respectivamente. Cada resultado é composto por 3 linhas, a primeira mostra o intervalo de cláusulas necessárias para se obter a cláusula vazia. Entre parênteses estão as porcentagens das fórmulas que conseguiram encontrar a cláusula vazia. A segunda linha mostra a média de execução do

algoritmo e a terceira mostra o desvio padrão da população analisada.

Testes com 1 *thread* foram desconsiderados, pois pela implementação ser incompleta, execução com 1 *threads* pode não chegar a um resultado com o limite imposto.

As fórmulas *dubois50* e *dubois100* não encontraram uma cláusula vazia nos testes realizados.

4.3 Resultados para Divisão de Trabalho por *Threads*

As Tabelas 4.3 e 4.4 mostram os resultados para o teste de Divisão de Trabalho por *Threads*.

Tabela 4.3: 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 0,322	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 0,321	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 4.4: 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

As Tabelas 4.3 e 4.4 contém para cada fórmula, informações correspondentes ao número de *threads* executados. Em cada célula dos resultados encontra-se: o conjunto de 1 ou mais cláusulas e as *threads* em que foram encontradas, a segunda linha mostra o tempo gasto para a execução, e a terceira linha mostra o desvio padrão.

4.4 Discussão dos Resultados

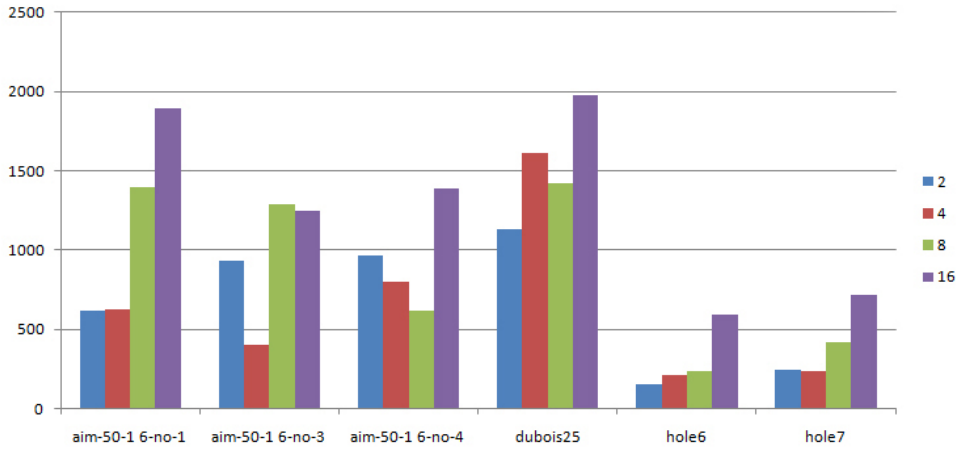
Os resultados obtidos em geral não apresentaram um *speedup* maior, a medida em que o número de *threads* aumenta. Na primeira abordagem (Colaboração entre Cláusulas), isso acontece pela quantidade de sincronização necessária para garantir o compartilhamento entre as cláusulas, bem como a dependência de uma ordem não conhecida para a obtenção da cláusula vazia, e ainda há o tempo desperdiçado com a

resolução de ramificações que não vão gerar a cláusula vazia. Na segunda abordagem (Divisão de Trabalho por *Threads*) como cada ramificação é resolvida por uma *thread* de forma independente, para alguns casos, consegui-se um tempo menor a medida em que threads são inseridas.

4.4.0.1 Discussão dos Resultados para Colaboração entre *Threads*

Para analisar graficamente os dados da Tabela 4.2 apresenta-se dois gráficos nas Figuras 4.1 e 4.2 que resumem as menores cláusulas geradas e o tempo gasto (em segundos) para a abordagem de colaboração entre cláusulas, respectivamente.

Figura 4.1: Gráfico das menores cláusulas geradas para abordagem de colaboração entre cláusulas



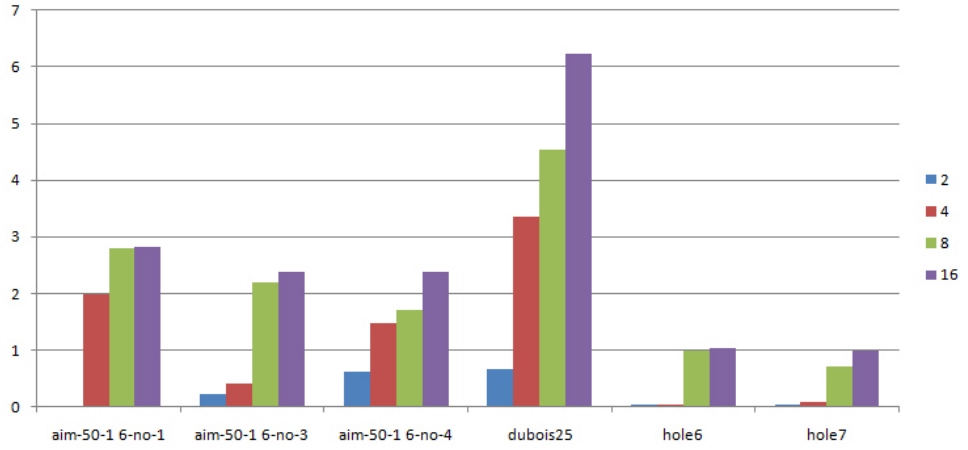
No gráfico da Figura 4.1 é possível observar que as fórmulas *aim-50-1_6-no-3*, *aim-50-1_6-no-4*, *dubois25* e *hole7*, ao serem processadas pelo algoritmo de colaboração entre *threads*, conseguiram encontrar a cláusula vazia com uma menor quantidade de cláusulas geradas a medida em que se adicionou *threads* na execução.

Com esses resultados é possível concluir que o sistema de colaboração de cláusulas, consegue reduzir significativamente o número de cláusulas geradas, entretanto é difícil estabelecer uma padrão de comportamento, pois não se sabe a ordem de execução das *threads*.

Esse método mostrou-se ineficiente se o objetivo for encontrar a cláusula vazia no menor tempo de execução possível, como pode-se observar no gráfico da Figura 4.2, os tempos aumentam a medida em que as *threads* são inseridas, entretanto o método mostra-se eficiente se se deseja obter um menor caminho para se encontrar a cláusula vazia.

As fórmulas *aim-50-1_6-no-1*, e *hole6* não apresentaram resultado significativo no teste.

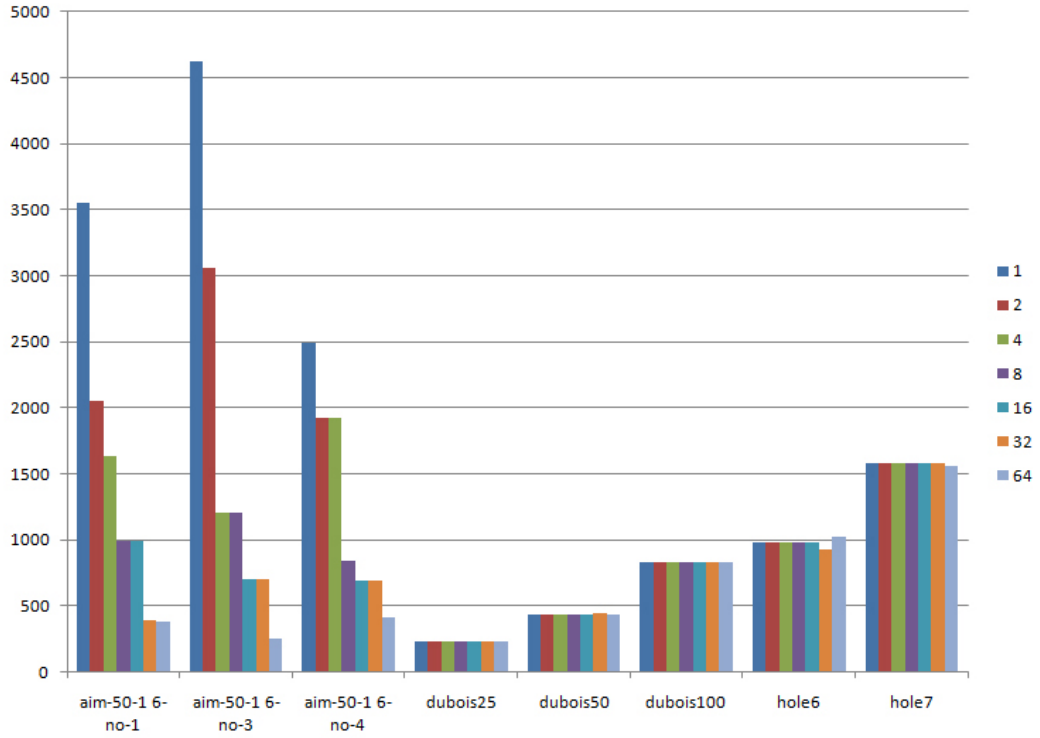
Figura 4.2: Gráfico de tempo de execução para abordagem de colaboração entre cláusulas



4.4.0.2 Discussão dos Resultados para Divisão de Trabalho por *Threads*

Os gráficos contidos nas Figuras 4.3 e 4.4 resumem as menores cláusulas geradas e o tempo de execução, mostrados nas Tabelas 4.3 e 4.4.

Figura 4.3: Gráfico das menores cláusulas geradas para abordagem de divisão por *threads*

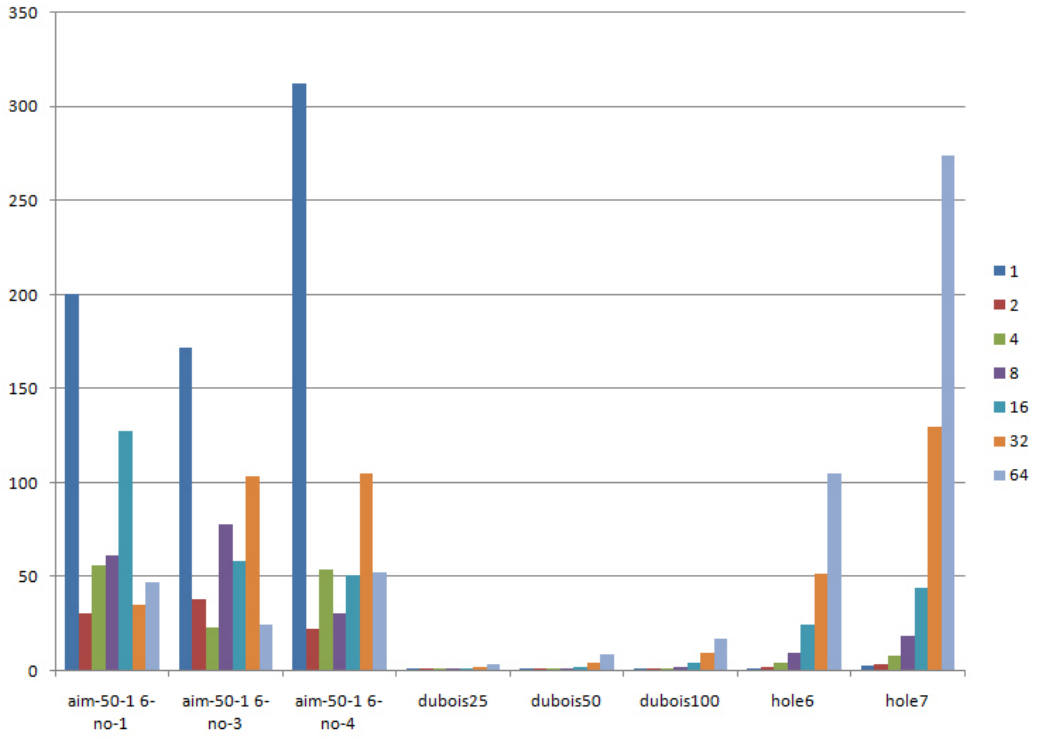


No gráfico da Figura 4.3, as fórmulas *aim-50-1_6-no-1*, *aim-50-1_6-no-3* e *aim-50-1_6-no-4* apresentam

considerável redução de cláusulas geradas a medida em que *threads* são adicionados. As fórmulas *dubois* não tiveram considerável evolução, com isso é possível concluir que as cláusulas vazias nessas fórmulas são encontradas com um mesmo número de cláusulas geradas, independente da ramificação escolhida para trabalhar.

As fórmulas do tipo *hole* apresentaram uma pequena diminuição de cláusulas geradas apenas para testes com 32 ou 64 *threads*.

Figura 4.4: Gráfico de tempo de execução para abordagem de divisão por *threads*



A Figura 4.4 mostra um gráfico, no qual é possível perceber que se consegue uma redução considerável a medida em que se adiciona *threads*, e é possível concluir ainda que, nem sempre um número elevado de threads acha a melhor solução, pois o custo para resolver várias ramificações ao mesmo tempo em um ambiente de memória compartilhada, gera concorrência na escrita em memória.

Esse método ainda se caracteriza em dois tipos de busca, se a intenção for achar o menor caminho, um número maior de threads pode ser definido, entretanto se a intenção for encontrar o a cláusula vazia em um menor tempo não se importando com a quantidade de cláusulas geradas, então poucas *threads* são suficiente.

Para a fórmula *aim-50-1_6-no-1* o melhor desempenho em relação ao tempo foi encontrado quando o processo utilizou 2 *threads*. Mas o caminho mais curto para se achar uma cláusula vazia foi encontrado apenas quando executada com 64 *threads*.

Ao se analisar um panorama geral é possível identificar outra característica interessante, a forma de

construção das fórmulas influenciam diretamente no tempo necessário para encontrar uma cláusula vazia, nem sempre o número de literais ou cláusulas influencia no 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* apresenta 10 vezes mais cláusulas que as do tipo *aim*, e mesmo assim são executadas em um tempo muito menor.

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.

Capítulo 5

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.

Na primeira abordagem buscou-se testar o quanto os *threads* que estão cuidando de ramificações diferenciadas na árvore de prova podem cooperar; na segunda abordagem utilizou-se a estratégia de cada *thread* cuidar de sua ramificação, não interferindo nas buscas efetuadas pelas outras.

Apesar do processamento paralelo, o custo de sincronização na primeira abordagem e da geração de todos os filhos da árvore de prova na segunda, mostrou que a adição de *threads* em geral, não aumenta o *speedup* do algoritmo.

Com a primeira abordagem consegui-se demonstrar que a colaboração entre as cláusulas consegue abreviar a busca pela cláusula vazia, já com a divisão das ramificações por *threads* demonstra-se que a busca por várias ramificações da árvore é vantajosa, entretanto, buscar por ramificações que não tem cláusula vazia, congestionam o acesso a memória com dados que não são aproveitados e fazem o algoritmo demorar mais que o necessário.

5.1 Trabalhos Futuros

Como principal trabalho futuro, deixa-se a tentativa de junção das abordagens, conseguir o compartilhamento entre cláusulas numa abordagem completa pode gerar bons resultados de tempo.

Ainda é possível explorar outro tipo de paralelismo, além da busca em diferentes ramificações, utilizar *threads* específicos para aplicar estratégias de resolução descritas na Seção 2.2.2, com isso esses *threads* ficam responsáveis por diminuir o espaço de busca e remover cláusulas que não iriam servir para a resolução do

problema.

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.

Explorar o segundo modelo em PRAM descrito na Seção 3.1.1 pode ser aplicada a estrutura CUDA que conta com vários multi-processadores. Dessa forma garante-se a busca por todas as cláusulas iniciais da fórmula.

Referências Bibliográficas

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

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

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

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