Um resolvedor SAT paralelo com BSP sobre uma grade

Fernando Corrêa Lima

DISSERTAÇÃO APRESENTADA

AO

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DA

UNIVESIDADE DE SÃO PAULO

PARA

OBTENÇÃO DO TÍTULO DE MESTRE

EM

CIÊNCIAS

Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Marcelo Finger

São Paulo, janeiro de 2007

FERNANDO CORRÊA LIMA

Um resolvedor SAT paralelo com BSP sobre uma grade

Dissertação apresentada ao Instituto de Matemática e Estatística da Universidade de São Paulo, como pré-requisito para a obtenção do título de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação.

Orientador: Prof. Dr. Marcelo Finger.

São Paulo

janeiro de 2007

Agradecimentos

Agradeço a meus pais, que mesmo sem ter recebido educação formal fizeram tudo para que eu não passasse pelas dificuldades que eles viveram. Me fizeram conseguir tudo que tenho em minha vida.

Agradeço imensamente minha namorada, Emilene, pela ajuda e apoio constantes, mesmo tendo um namorado pouco presente. Me faz ser uma pessoa cada vez melhor.

Agradeço meu orientador e amigo, Marcelo Finger, que me aceitou como aluno mesmo sabendo que meu tempo seria dividido com minhas demais responsabilidades profissionais. Foi meu apoio em meus momentos de desânimo e me transmitiu bastante conhecimento.

Agradeço aos amigos da Maps, principalmente meus superiores, que deram toda liberdade de que precisei e compreenderam minhas ausências. Alguns se tornaram excelentes amigos.

Agradeço muito também aos amigos que fiz ao longo de todos estes anos de graduação e mestrado, mas que não citarei nomes para não correr o risco de ser injusto. Eles me ajudaram a aprender tudo que sei hoje.

Agradeço também a todos que se indignaram por eu decidir qual caminho seguir em minha vida ao invés de viver o estilo de vida que julgavam mais ”adequado” para mim. Me fizeram saber exatamente quem sou.

Resumo

O Objetivo deste trabalho é implementar um resolvedor distribuído para o problema de satisfabilidade em lógica proposicional (SAT) que possa ser executado em uma grade de computadores. Será analisada a influência que o número de máquinas utilizadas pela grade para resolver diversas instâncias do SAT exerce sobre o desempenho do resolvedor implementado.

Abstract

The objective of this work is to implement a distributed solver for the satisfability problem in propositional classical logic (SAT) that can be executed in a computing grid. We study the influence of the number of machines used by the grid to solve several SAT instances with respect to solver performance.

Sumário

1 Introdução

1.1 Motivação

Atualmente, existem muitas pesquisas sobre o problema de satisfabilidade em lógica proposicional (SAT). O SAT foi o primeiro problema identificado como NP-Completo [33] e atualmente permanece como um dos mais estudados dessa classe.

O SAT é importante em diversas linhas de pesquisa, como matemática, redes de computadores, robótica, engenharia elétrica e outras, que são obrigadas a lidar com o SAT freqüentemente, e por isso é grande o interesse no desenvolvimento de métodos capazes de resolver de maneira eficiente instâncias complexas do SAT. Além disso, diversas linhas de estudo, como por exemplo teoria dos grafos, também lidam com outros problemas NP-completos, que podem ser reduzidos ao SAT por um algoritmo polinomial.

Devido ao fato do SAT ser NP-completo, atualmente não é possível construir resolvedores com tempo melhor que exponencial no pior caso, porém, resolvedores bem elaborados podem resolver muitas instâncias úteis para diversas áreas do conhecimento.

Como exemplos concretos de uso do SAT podemos citar verificação de hardware [34], inteligência artificial [22], visão computacional [8], diagnóstico, o problema do caixeiro viajante, N-rainhas, problemas em grafos, jogos, entre outros.

Assim, é evidente que avanços no estudo do SAT podem gerar avanços nas pesquisas de outros problemas NP-completos, e consideramos essa a característica mais impressionante desta classe de problemas.

É cada vez mais comum o uso de sistemas distribuídos que disponibilizam grandes quantidades de recursos computacionais, como processador, memória e até arquivos [5, 6, 2, 1]. O uso desse tipo de sistema tem ajudado a resolver problemas computacionais em diversas áreas do conhecimento, pois permite, a um baixo custo financeiro, o uso de aplicações distribuídas que requerem grande poder computacional, como o grupo United Devices [6] que tem projetos que visam por exemplo a cura do câncer e o grupo SETI@home [5] que procura sinais de inteligência extra terrestre.

Devido ao crescimento do número de pesquisas para o desenvolvimento de sistemas distribuídos, é natural pesquisar métodos paralelos para resolver o SAT, mesmo sabendo que paralelizar o SAT é uma tarefa difícil, pois em geral é necessário muita comunicação entre as partes.

Existem diversos modelos de programação paralela [25, 7, 15, 16, 4]. Alguns exigem o gerenciamento manual da comunicação entre os nós enquanto outros fornecem uma interface de comunicação transparente para o usuário.

Para implementar o gerenciamento das partes e a comunicação entre os nós da rede iremos seguir um modelo de computação paralela bastante conhecido, o *Bulk Synchronous Parallel* (BSP), e utilizaremos o Integrade [3] como arcabouço que fornece uma interface BSP [4]. O Integrade foi escolhido por ter sido desenvolvido no mesmo instituto (IME-USP) onde este trabalho foi desenvolvido, o que facilita acesso aos desenvolvedores do Integrade.

O uso do modelo BSP e do Integrade restringirá as possibilidades de paralelização mas facilitará a criação do sistema, pois a comunicação entre os nós é feita de maneira transparente.

Este trabalho consiste em desenvolver um programa de computador capaz de resolver instâncias do SAT (resolvedor SAT) distribuído, baseado no algoritmo DPLL [12] que pode ser executado em qualquer grade de computadores que implemente o modelo BSP [23]. O objetivo principal foi analisar a influência que o número de máquinas participantes da grade exerce

sobre o desempenho de um resolvedor SAT que utiliza as modernas técnicas de otimização do algoritmo DPLL existentes atualmente.

1.2 Objetivos

O objetivo deste trabalho é estudar a viabilidade de resolver o SAT em um ambiente distribuído utilizando um arcabouço que forneça uma interface paralela para uma grade oportunista e que gerencie a comunicação de forma transparente para o usuário. O resolvedor criado para fazermos esta análise será baseado no algoritmo DPLL e usará as principais técnicas de otimização utilizadas atualmente por pesquisadores do SAT.

1.3 Organização do texto

Os capítulos do texto estão organizados da seguinte forma:

1. Explicamos as motivações para o desenvolvimento deste trabalho e fornecemos uma introdução aos principais conceitos utilizados ao longo do texto;
2. Definimos formalmente o SAT e explicamos o algoritmo DPLL, que serviu como base para a criação do resolvedor implementado;
3. Explicamos as principais otimizações utilizadas pelos resolvedores considerados eficientes;
4. Fizemos uma análise sobre dois resolvedores bastante conhecidos, para ilustrar os conceitos apresentados;
5. Explicamos o conceito de programação em grade e discorremos sobre o arcabouço utilizado na implementação;
6. Explicamos como os conceitos dos capítulos anteriores foram utilizados na implementação do resolvedor;
7. Finalmente apresentamos as conclusões finais.

Todos os termos e citações de lógica utilizados neste trabalham referem-se a lógica proposicional clássica (cálculo proposicional). $\wedge$ representa conjunção lógica, $\vee$ a disjunção lógica, $\neg$ a negação lógica e utilizamos T como valor verdade *verdadeiro* e F como *falso*.

2 O SAT

2.1 Definição do Problema

SAT foi o primeiro problema identificado como NP-completo [33] e é atualmente um dos problemas mais estudados desta classe [1].

Podemos enunciar o SAT de maneira informal em poucas palavras: dada uma fórmula lógica proposicional clássica, decidir se existe uma atribuição de valores que a torne verdadeira. Ao longo deste trabalho estudamos apenas resolvedores SAT que tratam fórmulas no chamado formato clausal explicado a seguir. Esta restrição quanto ao formato da fórmula visa facilitar a implementação de resolvedores sem perder generalidade, pois é fácil demonstrar que qualquer fórmula da lógica proposicional clássica tem uma fórmula equivalente no formato clausal.

Para definir formalmente o SAT no formato clausal, também chamado de forma normal conjuntiva, primeiramente definimos sua linguagem:

- Seja $P = \{p_0, p_1, \ldots, p_s\}$ o conjunto de **símbolos proposicionais**, também chamados de símbolos atômicos, átomos ou variáveis.
- Um **literal** é um elemento de P ou sua negação, e neste caso denotamos a negação de p por $\neg p$. Exemplos de literais: p_1, $\neg p_7$.

[1] Uma boa fonte de informações sobre o SAT é o site www.satlib.org.

- Uma **cláusula** $C = \{L_1, L_2, \ldots, L_k\}$ é um conjunto de literais. Exemplos de cláusulas: $\{p_1, p_3, p_{175}, \neg p_{175}\}$,$\{\}$,$\{p_7, \neg p_7\}$.
- Uma **fórmula** $K = \{C_1, C_2, \ldots, C_n\}$ é um conjunto de cláusulas.

  Esta representação de fórmulas lógicas é conhecida como *forma normal clausal* (CNF) e seu objetivo é facilitar a criação de provadores de teoremas. Demonstrar que qualquer sentença do cálculo proposicional pode ser escrita nesta forma é simples e por isso podemos usá-la sem perda de generalidade.

  Então associamos uma semântica a esta linguagem:
- Uma **valoração** é uma função $V : P \rightarrow \{T, F\}$. Dizemos que uma valoração é parcial se alguns de seus átomos não tem valor definido.

Dados o conjunto de símbolos atômicos P e um literal L, uma cláusula C e uma fórmula K, construídos a partir de P, estendemos a semântica de V da seguinte maneira:

$$V(L) = \begin{cases} V(p_0), & \text{se } L = p_0, \text{ onde } p_0 \in P \\ \neg V(p_0), & \text{se } L = \neg p_0, \text{ onde } p_0 \in P \end{cases}$$

$$V(C) = \begin{cases} T, & \text{se } \exists L_i \in C \text{ tal que } V(L_i) = T \\ F, & \text{caso contrário (inclui o caso } C = \varnothing) \end{cases}$$

$$V(K) = \begin{cases} T, & \text{se } \forall C_i \in K, V(C_i) = T \text{ (inclui o caso } K = \varnothing) \\ F, & \text{caso contrário} \end{cases}$$

Para uma determinada fórmula K, dizemos que uma valoração V não satisfaz K se $V(K) = F$. Chamamos uma fórmula K de **tautologia** se é sempre verdadeira, ou seja, $V(K) = T$ para toda valoração V.

Agora podemos definir o problema SAT: dada uma fórmula, encontrar uma valoração que a torne verdadeira, ou concluir que tal valoração não existe.

$$
\begin{array}{lll}
F_1 = \{ & \{p_1, p_2, \neg p_3\}, & \\
& \{\neg p_2, p_4\}, & F_2 = \{ \quad \{p_0, \neg p_1\}, \\
& \{p_1, \neg p_5\}, & \quad\quad \{p_2, p_3\}, \\
& \{\neg p_1, p_2, p_3, \neg p_4, p_5\}, & \quad\quad \{\neg p_0, p_4, p_5\}, \\
& \{\neg p_3\} & \quad\quad \{\neg p_5, p_6\}, \\
& \{p_2, p_5\} & \quad\quad \{\neg p_4, p_6\}\} \\
& \{p_1\}\} & \\
& (a) & (b)
\end{array}
$$

Figura 2.1: Exemplos de instâncias do SAT

A Figura 2.1 mostra duas instâncias do SAT. A instância (a) tem 5 variáveis e 7 cláusulas, (b) tem 7 variáveis e 5 cláusulas. As soluções para (a) são

$$(x_1, x_2, x_3, x_4, x_5) = \{(T, T, F, T, *); (T, F, F, *, T)\}^2$$

e algumas para (b) são

$$(a, b, p, q, r, s, t) = \{(F, T, F, F, F, F, *); (T, *, F, F, F, F, *)\}$$

Dois outros problemas correlatos ao SAT são:

#SAT: encontrar o número de valorações que tornam uma fórmula verdadeira;

***SAT:** encontrar todas as valorações que tornam uma fórmula verdadeira.

Neste trabalho foi implementado um algoritmo paralelo para resolver o SAT.

2.2 Algoritmo DPLL

A maneira mais simples e intuitiva de resolver o SAT é enumerar todas as valorações possíveis e verificar quais satisfazem a fórmula dada. Este algoritmo irá verificar 2^n valorações, onde n é o número de átomos presentes na fórmula, o que o torna inviável para valores grandes de n.

Em 1962, Davis Logemann e Loveland publicaram o DPLL [12], um algoritmo de busca para resolver o SAT que tenta eliminar caminhos que produzem contradições. O DPLL explora

[2] $*$ significa que o valor verdade da variável é indiferente, tanto faz se é T ou F

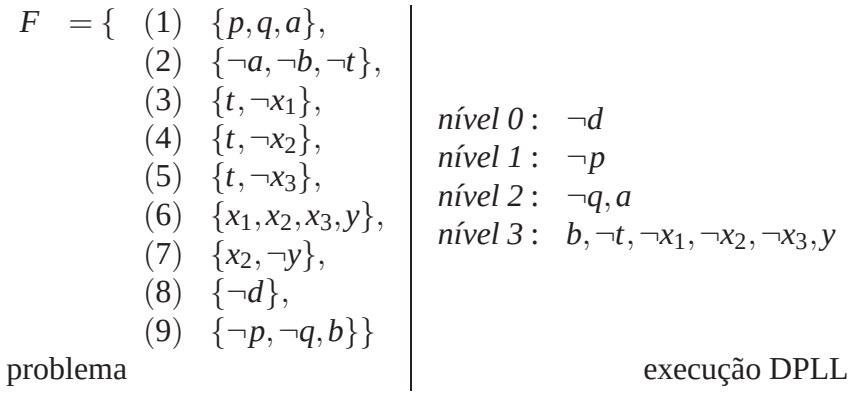


Figura 2.2: Exemplo de execução do DPLL

o espaço de valorações, mas não estabelece a ordem como a busca deve ser feita, e por isso é possível implementar otimizações para melhorar o desempenho de resolvedores SAT. As principais otimizações para o DPLL serão estudadas mais adiante.

O DPLL é exibido em pseudo-código no algoritmo 1.

Algoritmo 1 Dpll(K, V)

Requer: uma fórmula K e uma valoração inicial V
Garante: devolve T caso K seja satisfazível e neste caso V(K)=T, caso contrário devolve F

```
se V(K) = F entao
   devolva F
se V(K) = T entao
   imprima V
   devolva T
se ∃ cláusula C = {L} de tamanho 1 entao
   devolva Dpll(K, V + {V(L) = T})
se ∃ literal L cuja negação não ocorre em K entao
   devolva Dpll(K, V + {V(L) = T})
L ← algum literal indefinido
se Dpll(K ∪ {L}, V) = F entao
   devolva Dpll(K ∪ {¬L}, V)
```

2.2.1 Exemplo

Para ilustrar a execução do DPLL considere o exemplo da figura 2.2 (os números entre parênteses servem apenas para identificar cada cláusula).

Neste exemplo, o DPLL identifica a cláusula (8), que é unitária e atribui F à variável d. A seguir é escolhido o literal p para bifurcar (a escolha neste exemplo é arbitrária, apenas para

ilustrar a execução do DPLL) e é atribuído F, o que satisfaz a cláusula (9) e permite eliminar p da cláusula (1). No próximo nível da execução é feita a bifurcação $V(q) = F$, que torna a cláusula (1) unitária. No nível seguinte da recursão, DPLL implica $V(a) = T$ para satisfazer a cláusula (1). A seguir DPLL faz $V(b) = T$, que torna a cláusula (2) unitária e implica $V(t) = F$, $V(x_1) = F$, $V(x_2) = F$, $V(x_3) = F$, $V(y) = T$.

Neste ponto é obtido um conflito, pois a cláusula (8) é falsa. O DPLL deve retroceder até a última bifurcação e inverter o valor do literal selecionado naquele nível, neste exemplo deve fazer $V(b) = F$ e continuar sua execução.

3 Otimizações no DPLL

3.1 Propagação de Restrições

Um ponto chave do DPLL é a propagação de restrições (Propagação) [34], que consiste em identificar cláusulas que contém apenas um literal indefinido e atribuir seu valor verdade como T, dessa forma estas cláusulas podem ser eliminadas da fórmula, pois estão satisfeitas (tem valor T). Podemos também eliminar as ocorrências da negação do literais que tiveram seu valor definido desta forma, pois são todos falsos. O ato de identificar uma cláusula unitária e atribuir o valor do seu literal livre é chamado de propagação unitária (implicação) e é exibido no algoritmo 2.

Algoritmo 2 PropagaçãoUnitaria(C, K, V)

Requer: uma cláusula C, uma fórmula K e uma valoração V
Garante: caso C seja unitária ela será removida de K
  se C tem apenas um literal L indefinido **entao**
    $V(L) \leftarrow T$
    para todo $C' \in K$ contendo $\neg L$ **faca**
      PropagaçãoUnitaria($C', K - C, V$)

A implicação unitária é feita iterativamente, pois a eliminação de um literal pode gerar outras cláusulas unitárias que também podem ser eliminadas e assim por diante. A propagação é exibida no algoritmo 3.

Algoritmo 3 Propagação(K, V)
Requer: uma fórmula K e uma valoração V
Garante: todas as cláusulas unitárias são removidas de K
  para todo $C \in K$ **faca**
    PropagaçãoUnitaria(C, K, V)

O algoritmo DPLL consiste em explorar o espaço de busca pela atribuição de valores às variáveis. Caso uma atribuição não gere solução, o algoritmo atribui o valor complementar à variável. Se ainda assim não for encontrada uma solução, ela também é desfeita. O processo de desfazer uma atribuição chama-se *backtracking*.

3.2 Heurísticas de Ramificação

O critério de escolha do literal indefinido (heurística de ramificação) determina a maneira como o espaço de busca é explorado e conseqüentemente influencia o tempo de execução do algoritmo.

Nos últimos anos muitas heurísticas foram propostas [10, 11, 20, 24]. A heurística mais simples é selecionar aleatoriamente um literal indefinido (heurística *RAND*). As demais heurísticas utilizam informação obtida durante a exploração do espaço de busca, como por exemplo, o número de variáveis em cada cláusula indefinida e o tamanho das cláusulas indefinidas. A seguir são apresentadas algumas heurísticas.

3.2.1 Heurística MOM's

Esta é uma das heurísticas mais conhecidas [28]. A idéia é selecionar variáveis que aparecem freqüentemente em cláusulas "pequenas". Intuitivamente, ao atribuirmos valor para tais literais, é provável que algumas das cláusulas envolvidas se tornem unitárias.

Seja $f^*(L)$ o número de ocorrências do literal L nas cláusulas indefinidas de tamanho mínimo, é aceitável considerar que uma boa escolha de variável livre é a que maximiza a função

$$(f^*(x) + f^*(\neg x)) * 2^k + f^*(x) * f^*(\neg x) \tag{3.1}$$

e atribui o seu valor verdade como T se ela aparece nas cláusulas de tamanho mínimo mais vezes como literal positivo, ou F caso contrário.

Intuitivamente, assumindo k suficientemente grande, é dado preferência para variáveis com muitas ocorrências de x ou $\neg x$ simultaneamente, e também às variáveis com muitas ocorrências de x ou de $\neg x$ de maneira isolada. Ou seja, a heurística favorece as variáveis que aparecem muitas vezes positivamente, negativamente ou das duas formas.

3.2.2 Heurística Jeroslow-Wang

Esta heurística foi proposta por Jeroslow and Wang [20] e consiste em computar, para cada literal a função

$$J(L) = \sum_{w_i \in W} 2^{-|w_i|} \tag{3.2}$$

onde W é o conjunto de cláusulas nos quais L aparece. O literal que maximiza J é selecionado para bifurcar. Uma variação consiste em selecionar a variável que maximiza $J(x) + J(\neg x)$ e atribuir T a mesma se $J(x) > J(\neg x)$, e F caso contrário.

3.2.3 Heurísticas DLCS e DLIS

Seja $t_p(x)$ o número de cláusulas nas quais a variável x aparece positivamente e $t_n(x)$ o número de cláusulas nas quais x aparece negativamente. *Dynamic Larges Combined Sum* (DLCS) [28] consiste em selecionar a variável que maximiza $t_p(x) + t_n(x)$ e atribuir o valor T se $t_p(x) > t_n(x)$, ou F caso contrário. *Dynamic Largest Individual Sum* (DLIS) seleciona o literal que maximiza $max(t_p(x), t_n(x))$ [28].

Uma possível variação de DLIS é escolher aleatoriamente o valor verdade atribuído ao literal selecionado, chamamos tal heurística de Random DLIS (RDLIS).

3.3 Aprendizado de Cláusulas

Durante a exploração do espaço de busca, o DPLL encontra conflitos. Os resolvedores mais eficientes utilizam algum método de análise de conflitos para determinar sua causa e aprender novas cláusulas [32].

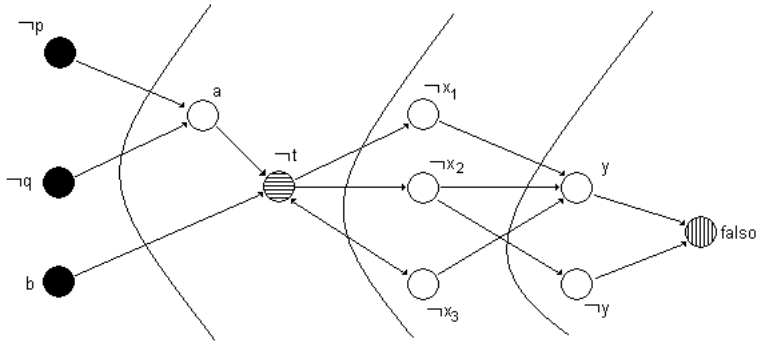


Figura 3.1: Grafo de implicações

Independente do método de análise, a cláusula aprendida é redundante, pois pode ser inferida a partir da fórmula original, e seu objetivo é diminuir o espaço de busca que precisa ser explorado.

Para implementar o aprendizado de cláusulas (aprendizado), é preciso manter informações sobre as causas das implicações (grafo de implicações) ao longo da propagação. Na prática, não é necessário manter o grafo explicitamente, basta anotarmos a cláusula responsável por cada implicação, ou seja, a cláusula que se tornou unitária [29, 27].

Diferentes técnicas de análise de conflito podem gerar diferentes cláusulas que podem ser aprendidas. Por exemplo, o grafo de implicações para a execução mostrada na figura 2.2 é exibido na figura 3.1.

Neste exemplo, a cláusula mais intuitiva de aprender é $\{p, q, \neg b\}$, que é obtida a partir das decisões tomadas pelo DPLL, ou seja, garante que as decisões que levaram ao conflito não sejam feitas simultaneamente.

Podemos também aprender a cláusula $\{t\}$, pois ela é suficiente para gerar o conflito.

Uma terceira possibilidade seria aprender $\{x_1, x_2, x_3\}$, que são as atribuições mais ”recentes” que levam ao conflito.

Estes métodos de aprendizado são conhecidos como esquema de decisão, *unique implication* (UIP) e primeiro corte, respectivamente. E todos eles produzem cláusulas que geram o mesmo conflito.

3.4 Backjumping

Ao obter um conflito, o DPPL retorna um nível na busca, inverte o valor verdade atribuído ao literal selecionado e continua a busca. Caso tanto o valor T quanto F já tenham sido utilizados, o DPLL retorna mais um nível e assim por diante. Este processo não é muito diferente da exploração simples do espaço de busca, no qual testamos 2^n valorações, onde n é o número de variáveis do problema.

A análise do conflito, além de permitir o aprendizado de uma nova cláusula, pode indicar se o último literal bifurcado (literal bifurcado mais recentemente) está ou não na causa do conflito. Se não estiver será inútil retornarmos somente um nível na recursão, pois não importa o valor verdade deste literal e se continuarmos a busca a partir do seu nível iremos obter um conflito. Neste caso podemos retroceder mais de um nível na recursão. O número de níveis que podemos retroceder depende da técnica de análise de conflito utilizada [29, 27].

O exemplo da figura 2.2, que tem o grafo de implicações ilustrado na figura 3.1, ilustra bem o conceito de backjumping.

Para decidir qual será o nível do backtracking basta encontrar a última bifurcação cujo literal bifurcado aparece na cláusula aprendida e o valor atribuído. Caso isso já tenha sido feito, DPLL retorna para a bifurcação anterior e assim sucessivamente.

Suponha que o método utilizado seja o ingênuo esquema de decisão e então neste caso o DPLL irá voltar sempre até a ultima bifurcação. No exemplo apresentado voltará ao nível 3, atribuirá $V(b) = T$ e continuará a execução.

Já utilizando o método UIP, DPLL retorna até o nível 0, já que t não é uma variável de decisão, atribui $V(t) = T$, pois a cláusula aprendida é unitária, e continua a execução. Note que neste caso no nível 0 serão satisfeitas as cláusulas 3, 4 e 5, além de eliminar $\neg t$ da cláusula 2. Portanto este método foi mais eficiente que o esquema de decisão.

4 Alguns Resolvedores SAT Existentes

Existem vários resolvedores SAT famosos e eficiêntes como GRASP [27], SATO [35], SATZ [30], Chaff [29] e GridSat [9]. A seguir utilizaremos Chaff e GridSat para ilustrar os conceitos explicados anteriormente e os detalhes de implementação nos quais baseamos nosso resolvedor.

4.1 Chaff-Modelo Centralizado

Atualmente, Chaff [29] é um dos resolvedores SAT seqüenciais baseados no DPLL mais eficientes. Sua eficiência se deve principalmente a um algoritmo de propagação otimizado e uma heurística de escolha de literais indefinidos que prioriza satisfazer as cláusulas aprendidas nos níveis mais recente do DPLL. Além de aprendizado de novas cláusulas, Chaff também utiliza desaprendizado de cláusulas, ou seja, periodicamente, as cláusulas aprendidas são retiradas da fórmula. As estruturas de dados do Chaff também foram cuidadosamente projetadas visando ganhos de desempenho. As principais características do Chaff são explicadas com maiores detalhes a seguir.

4.1.1 Propagação

Na seção 3.1 definimos o conceito de identificar uma cláusula unitária. A forma mais simples de identifica-la é, ao atribuirmos F à um literal, verificarmos todas as cláusulas nas quais o literal aparece para então identificar as unitárias. Para isso podemos manter um contador na cláusula a fim de sabermos quantos literais indefinidos ela tem e atualizarmos o contador conforme o DPLL atribui valor aos literais. Contudo, para uma cláusula de tamanho S, iremos visitar a cláusula quando 1, 2, 3, ..., S-1 literais se tornarem falsos, sendo que só é realmente necessário verificar se a cláusula se tornou unitária quando o contador passar de S-2 para S-1.

Como uma aproximação para isso, Chaff monitora 2 literais não falsos para cada cláusula e assim uma cláusula só precisa ser verificada quanto a sua unicidade se algum dos seus literais monitorados se tornar falsos.

No momento em que um literal L se torna falso, Chaff verifica as cláusulas nas quais L é monitorado, e uma das seguintes situações ocorre:

- A cláusula tem pelo menos dois literais não negativos, sendo um deles o outro literal monitorado, ou seja, pelo menos um literal não monitorado é não negativo, e passamos a monitorar este literal ao invés de L, mantendo o invariante de monitorar dois literais não negativos para cada cláusula.
- A cláusula é unitária, será eliminada da fórmula durante a propagação e tratada como uma cláusula satisfeita.

Suponha que iremos resolver uma determinada instância do SAT e que uma de suas cláusulas seja

$$\neg v_1 \vee v_4 \vee \neg v_7 \vee v_{11} \vee v_{15}$$

Escolhemos, de maneira arbitrária neste exemplo, monitorar os literais $\neg v_1$ e v_{15}. Indicamos os literais monitorados com um subscrito:

$$\underline{\neg v_1} \vee v_4 \vee \neg v_7 \vee v_{11} \vee \underline{v_{15}}$$

Suponha que o resolvedor decida $\neg v_1 = F$. Neste caso é necessário verificar se a cláusula se tornou unitária pois um de seus literais monitorados se tornou falso. Como a cláusula não é unitária selecionamos outro literal para monitorar, por exemplo v_4.

$$\neg v_1 \vee \underline{v_4} \vee \neg v_7 \vee v_{11} \vee \underline{v_{15}}$$

Se as próximas decisões forem $v_7 = T$ e $v_{11} = F$, então não é necessário verificar a cláusula quanto a sua unicidade, pois devido aos literais monitorados sabemos que ela não se tornou unitária.

Uma característica interessante desse esquema é que ao fazer *backtracking* podemos manter os literais monitorados fazendo com que a desatribuição de uma variável seja feita em tempo constante. Além disso, atribuir valor a uma variável que recentemente teve seu valor atribuído e desatribuído tende ser mais rápido que a primeira atribuição, pois o conjunto de cláusulas nas quais este literal é monitorado fica menor após esta operação.

No exemplo anterior, se o resolvedor decide desfazer todas as atribuições, v_4 e v_{15} podem ser mantidos como literais monitorados.

4.1.2 Heurística de Ramificação

A heurística de escolha do próximo literal livre, chamada pelos autores de *Variable State Independente Decaying Sum* (VSIDS) [29], consiste em:

1. Cada literal possível possui um contador, inicializado com zero;
2. Ao adicionar uma cláusula à fórmula, o contador associado a cada literal desta cláusula é incrementado;
3. O literal selecionado será sempre o que tiver o maior contador;
4. Empates são resolvidos aleatoriamente;
5. Periodicamente, todos contadores são divididos por uma constante.

Para minimizar o tempo de seleção do literal é mantida uma fila de prioridades.

Esta estratégia de seleção de literais visa resolver as contradições tentando satisfazer os conflitos mais recentes, ou seja, as cláusulas que foram aprendidas há menos tempo, o que é eficiente principalmente em problemas difíceis, que tendem a gerar muitos conflitos [29].

4.1.3 Aprendizado de cláusulas

Durante a propagação, Chaff mantém informações sobre o estado da árvore de busca e ao encontrar uma contradição é capaz de determinar quais atribuições de variáveis geraram o conflito. A cláusula formada pela conjunção dessas atribuições é suficiente para gerar o conflito e pode ser encontrada em outros pontos do espaço de busca.

O esquema de aprendizado utilizado por Chaff é baseado no UIP e consiste em encontrar o nó dominante no grafo de implicações, ou seja, um nó tal que qualquer caminho entre o último literal bifurcado e o conflito passe através dele. Tal nó divide o grafo de implicações em duas partes. Chamaremos de corte a parte que contém os literais atribuídos após a atribuição de valor para o nó dominante.

Como podem existir vários nós dominantes, Chaff seleciona o que está mais próximo do conflito, por isso o método é chamado *First UIP* e o literal associado ao nó dominante encontrado FirstUIP.

A cláusula aprendida é constituída pela conjunção da negação do FirstUIP com as negações dos literais bifurcados que implicam literais que estão dentro do corte [29, 27].

Chaff adiciona a negação desta cláusula à fórmula existente antes dessas atribuições serem feitas. Este processo é o ponto fundamental para diminuir o tamanho do espaço de busca, evitando caminhos que levem ao mesmo conflito e é base para implementação do *backjumping* [27].

O uso de aprendizado gera uma sobrecarga na execução que é compensado pela redução no espaço de busca que precisa ser explorado. Porém, quanto maior for o número de conflitos gerados durante uma execução, maior será o número de cláusulas aprendidas, e este crescimento pode ser exponencial no número de variáveis do problema.

4.1.4 Desaprendizado de cláusulas

Algumas instâncias do SAT podem gerar um número exponencial de conflitos e por isso o aprendizado pode gerar um crescimento exponencial do conjunto de cláusulas. Chaff faz um corte no conjunto de cláusulas, para evitar um esgotamento da memória, através da remoção de cláusulas aprendidas em conflitos. Este processo é chamado desaprendizado de cláusulas (desaprendizado).

Na verdade, para diminuir a sobrecarga causada pela adição e remoção de cláusulas da fórmula, a remoção de uma cláusula consiste simplesmente em marcá-la como removida. A remoção real é feita por uma rotina de compactação da fórmula, executada periodicamente, que as apaga efetivamente do conjunto.

Chaff periodicamente zera o estado das variáveis, reiniciando a exploração do espaço de busca e utilizando a informação aprendida durante a busca anterior. Isto permite explorar ramos da árvore de busca que não foram explorados na busca anterior.

4.1.5 Backjumping

Chaff implementa o *backjumping*, ou backtracking não cronológico.

Com o *backtracking* clássico, se no nível x da árvore de busca é encontrada um conflito, o DPLL retrocede para o nível $x-1$, inverte o valor dado à variável bifurcada em $x-1$ e continua a busca. Caso chegue a outro conflito, retrocede até o nível $x-2$ e assim por diante.

O *backjumping* analisa o conjunto de variáveis que gerou o conflito (Chaff faz isso no mesmo processo utilizado para o aprendizado) e determina para qual nível algoritmo deve retornar, que pode ser anterior a $x-1$. Esta otimização diminui a porção do espaço de busca que precisa ser examinada. É interessante notar que ao fazer backjumping, a cláusula aprendida por Chaff é unitária e pode ser implicada antes do DPLL continuar sua execução.

Uma explicação detalhada sobre *backjumping* pode ser encontrada em [14].

4.2 GridSAT-Modelo Distribuído

Atualmente existem poucos resolvedores SAT distribuídos. GridSAT [9] é uma implementação paralela para resolver SAT que usa Chaff como algoritmo base e é executado em várias máquinas conectadas por uma rede. GridSAT usa backjumping, aprendizado e desaprendizado de cláusulas e compartilhamento de cláusulas aprendidas entre os nós da grade.

GridSAT tenta manter a resolução de um problema o mais seqüencial possível e só aloca novos nós quando os recursos computacionais existentes estão prestes a se esgotar. Isto é feito da seguinte maneira, inicialmente apenas uma máquina é designada para resolver o problema inteiro e quando esta máquina prevê que seus recursos irão se esgotar, novos recursos são alocados. Este processo se repete para cada uma das máquinas recém alocadas. Neste processo cada máquina é livre para alocar novos nós.

4.2.1 Paralelização do SAT no GridSAT

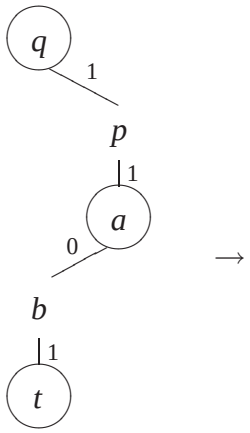
GridSAT adquire novos recursos computacionais de acordo com a necessidade, ou seja, novos nós são alocados sob demanda.

Tomemos a resolução da instância 2.1.b e suponhamos que o nó responsável por este problema tenha gerado a pilha de execução mostrada na figura 4.1. GridSAT divide o problema tal que o primeiro nível do nó original contenha a união dos níveis 0 e 1 do problema original (literal $\neg a$ neste exemplo), o novo nó alocado tem no nível 0 a união do nível zero original com a negação do literal bifurcado no nível 1 original (a). Esta forma de divisão garante que o espaço de busca foi efetivamente dividido entre os dois nós.

No exemplo a variável a assume valores opostos no nível 0 dos nós resultantes do processo de divisão, o que faz o espaço de busca ser dividido.

Após o processo de divisão, cláusulas satisfeitas pelo conjunto inicial de atribuições são eliminadas. Neste exemplo ambos podem eliminar as duas primeiras cláusulas, pois estão satisfeitas.

Um problema com esta metodologia é que a sobrecarga devida à divisão dos problemas, eliminação das cláusulas satisfeitas e coleta dos resultados pode ser maior que o tempo gasto



Execução com problema original

$$K' = \{ \{\neg p, r, s\}, \{\neg s, t\}, \{\neg r, t\}\}$$
$$v'_0 = (q, p, a, b, r, s, t) = (1, 1, 0, 1, *, *, *)$$
sub-problema a

$$K'' = \{ \{\neg p, r, s\}, \{\neg s, t\}, \{\neg r, t\}\}$$
$$v''_0 = (q, p, a, b, r, s, t) = (1, 1, 1, *, *, *, *)$$
sub-problema b

Figura 4.1: Exemplo de divisão de problema

para resolver o problema efetivamente, o que tornaria o resolvedor menos eficiente que um resolvedor seqüencial [21, 30].

5 Programação em Grade

5.1 Motivação

Nos últimos anos foi possível perceber um crescimento no número de computadores pessoais utilizados por pesquisadores e estudantes em universidades e também por funcionários de empresas em suas tarefas diárias. Esses computadores em geral ficam ociosos grande parte do tempo e, quando estão sendo utilizados, geralmente é consumida apenas parte de seus recursos computacionais, como processador e memória.

Esse fenômeno coexiste com a grande demanda por recursos computacionais existentes em diversas áreas de pesquisa como física, química, biologia e economia e em empresas com tarefas que precisam de grande poder computacional, como simulações no mercado financeiro e tratamento de multimídia.

É comum instituições que precisam de grande poder computacional terem recursos ociosos em computadores pessoais que poderiam ser usados de maneira mais efetiva, minimizando custos de aquisição de novos recursos.

A tentativa de utilizar recursos ociosos distribuídos em uma rede, muitas vezes separados por uma longa distância física, é tema de estudo da Programação em Grade. Este trabalho se concentra em uma subcategoria de grades computacionais, as *grades oportunistas*.

As grades oportunistas permitem que aplicações usem máquinas que podem ser não dedicadas à grade. Neste tipo de grade são utilizados recursos de computadores que costumam ficar a maior parte do tempo ociosos e que mesmo quando estão sendo utilizados fora da grade usam apenas uma fração de seus recursos. A grande maioria dos computadores pessoais podem fornecer recursos para uma grade oportunista pois, em geral, os usuários usam aplicações simples, como editores de texto ou navegadores.

O objetivo de uma grade oportunista é possibilitar o uso mais efetivo de recursos computacionais existentes e dessa forma minimizar custos financeiros e melhorar o desempenho de aplicações.

5.2 Modelo BSP

O *Bulk Synchronous Parallelism* (BSP) [23] é um modelo de computação para implementação de aplicações paralelas que facilita a criação de sistemas tolerantes a falhas sem gerar grande sobrecarga ao ambiente. Outra característica do modelo BSP é a facilidade oferecida para criação de aplicações portáveis entre as diversas arquiteturas paralelas existentes.

No modelo BSP as duas fases básicas da computação paralela, comunicação e sincronização, são separadas. Esta separação permite a criação de aplicações compatíveis com diferentes arquiteturas paralelas.

O modelo BSP divide a computação paralela em uma seqüência de *super-passos* e cada um deles é constituído por três fases distintas, que são executadas na seguinte ordem:

1. Cada nó participante realiza computações locais, somente com valores armazenados localmente e invisíveis para os demais nós.
2. Os nós comunicam-se, transferindo dados entre si. Durante a transição os dados transmitidos não necessariamente são visíveis para os receptores.
3. Barreira de sincronização: cada nó espera os demais concluírem a fase de comunicação, após isso os dados transmitidos tornam-se visíveis para os destinatários.

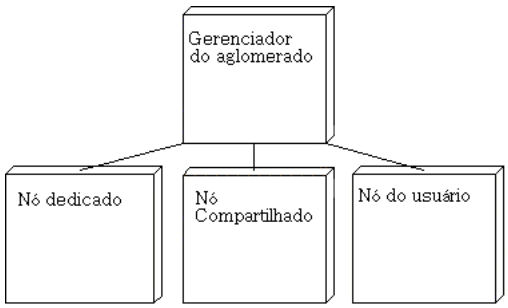


Figura 5.1: Estrutura de um aglomerado

5.3 Integrade

Integrade é um serviço de gerenciamento de recursos distribuídos que permite a execução remota de aplicações em uma rede de computadores [3, 18, 17, 19]. A escolha da máquina que irá efetivamente executar a aplicação (ou parte dela, caso seja uma aplicação distribuída) é transparente ao usuário e baseia-se na quantidade de recursos disponíveis no sistema e nas exigências de recursos impostas pelo usuário no ato da submissão da aplicação.

O Integrade organiza as grades em aglomerados, que são agrupamentos de 1 a 100 computadores. A Figura 5.1 mostra a estrutura de um aglomerado.

O *gerenciador do aglomerado* representa o nó responsável pelo gerenciamento deste aglomerado e a comunicação com gerenciadores de outros aglomerados. Um *nó dedicado* é uma máquina que disponibiliza seus recursos apenas para a grade, ou seja, não é utilizada fora dela. Um *nó compartilhado* é uma máquina que exporta apenas parte de seus recursos computacionais para a grade e que eventualmente pode ser utilizada por aplicações de fora da grade. Um *nó do usuário* é uma máquina utilizada para submeter aplicações para serem executadas na grade.

Os aglomerados são organizados em um hierarquia, o que torna possível o gerenciamento de grades com milhares de máquinas. A Figura 5.2 mostra um exemplo de uma grade com 4 aglomerados.

Após a submissão de uma aplicação o sistema se encarrega de alocar máquinas com recursos computacionais livres suficientes para executá-la e devolver os dados de saída para o usuário. O sistema também é tolerante a falhas, cuida de aspectos de segurança e garante a

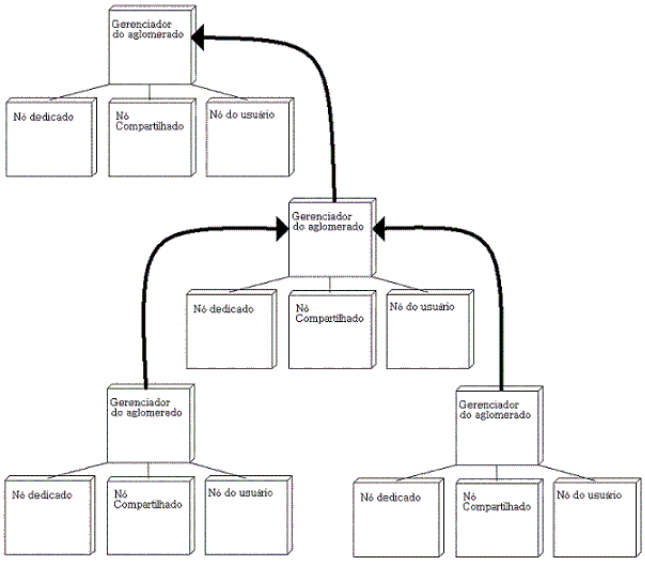


Figura 5.2: Estrutura de uma grade

comunicação dos dados de aplicações paralelas de forma transparente ao usuário, mais detalhes podem ser obtidos em [18, 26, 17, 19, 3].

O Integrade é capaz de executar três tipos de aplicações [17]:

- seqüenciais: aplicações que tem uma única linha de execução;
- paramétricas: aplicações seqüenciais que precisam ser executadas diversas vezes, usando parametros de entrada diferentes
- paralelas: aplicações que tem processamento paralelo para resolver um problema e podem precisar de comunicação entre as diversas linhas de execução.

As aplicações seqüenciais e paramétricas não precisam ter nenhuma característica especial para serem executadas na grade, enquanto que as aplicações paralelas devem utilizar as funções da biblioteca BSP do Integrade. Assim, as aplicações devem ser programadas utilizando o modelo BSP descrito na seção 5.2. A utilização do modelo BSP torna possível a criação de aplicações compatíveis com outras grades que utilizem o mesmo modelo [4].

6 Implementação Usando Integrade e BSP

6.1 Descrição do resolvedor

Ao longo deste trabalho implementamos um resolvedor SAT baseado no algoritmo DPLL. O resolvedor utiliza RDLIS como heurística de ramificação e foi utilizada a técnica de aprendizado de cláusulas baseada no FirstUIP, seguindo a metodologia de monitoramento de dois literais por cláusula para identificar cláusulas unitárias de maneira eficiente.

O resolvedor foi escrito em C++ e utiliza a interface BSP exposta pelo Integrade.

A decisão de escrever um resolvedor ao invés de utilizar algum existente visa compreender melhor o processo de resolução do SAT utilizando o algoritmo DPLL.

Junto com o aprendizado de cláusulas foi implementado também backtracking não cronológico, para diminuir o espaço de busca explorado.

O resolvedor recebe como entrada um arquivo com a descrição do problema no formato clausal e imprime na saída padrão se o problema é satisfazível ou não e o tempo gasto na resolução.

Representamos os literais como objetos que contém um número inteiro que representa o símbolo atômico utilizado e um sinal para indicar se o literal é a negação de um átomo. Cada literal armazena também o seu valor verdade, que pode ser T, F ou indefinido. Para cada literal é mantida uma lista das cláusulas onde ele aparece e outra lista com as cláusulas nas quais ele

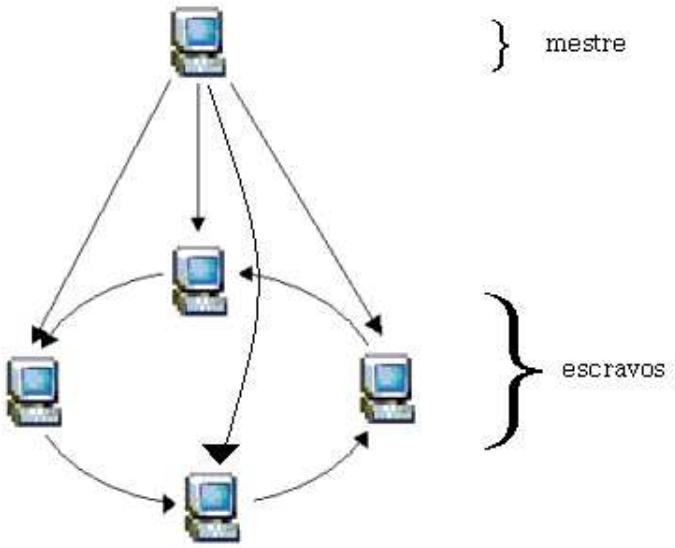


Figura 6.1: Estrutura da aplicação para 5 nós

é monitorado, isso permite identificar facilmente cláusulas unitárias, como descrito na seção 4.1.1.

A cláusulas são representadas por um vetor de ponteiros para literais, assim os literais são compartilhados entre as cláusulas e a atribuição de valor a um literal é feita em tempo constante.

Uma formula é uma lista de cláusulas, utilizamos uma lista pois durante a execução do DPLL novas cláusulas são aprendidas e adicionadas à fórmula.

O resolvedor é executado de maneira distribuída utilizando a topologia de mestre-escravos ilustrada na figura 6.1.

As responsabilidades do mestre são:

1. Leitura do problema de entrada;
2. Divisão do problema entre os nós da grade, ficando com uma parte para sí;
3. Fornecer valorações iniciais para os escravos;
4. Impressão do resultado.

As responsabilidades dos escravos são:

1. Receber uma parte do problema, que foi enviada pelo mestre;
2. Receber valorações parciais que são solução para o problema do mestre;
3. Receber valorações no escravo anterior, que são solução para aquele escravo;

4. Refinar as valorações recebidas, gerando valorações que satisfazem sua parte do problema;
5. Enviar as valorações geradas para o escravo seguinte;
6. Enviar para o mestre as valorações que passaram pelo processo de refinamento em todos os nós, pois são soluções para o problema original

É importante perceber que o mestre trabalha na geração de valorações em paralelo com todos os escravos, ou seja, o mestre participa ativamente da resolução do problema.

No início da execução o nó mestre lê o problema a partir do arquivo de entrada, a seguir verifica o número de nós participantes da grade, particiona o problema e distribui as partes entre eles, ficando com uma parte para si também.

Após a distribuição das partes do problema original, que chamamos sub-problemas, para os escravos, o mestre utiliza o DPLL para encontrar todas as valorações que são solução para o seu sub-problema, e cada valoração gerada é enviada para um escravo. Tais valorações podem ser parciais.

Ao receber uma valoração, o escravo usa o DPLL para restringí-la, atribuindo valores a átomos indefinidos, gerando todas as valorações que são solução para seu sub-problema, estas valorações são enviada para o escravo seguinte que irá refiná-las visando satisfazer seu sub-problema e assim sucessivamente, até que alguma valoração tenha passado por todos os escravos, e nesse caso ela satisfaz todos os sub-problemas e portanto é uma solução para o problema original.

O escravo que identifica uma solução global (valoração que passou por todos os nós) envia esta valoração para o mestre, que ao recebê-la envia uma mensagem a todos os escravos para que encerrem sua execução, e a solução é reportada ao usuário e o programa termina.

Caso algum escravo receba uma valoração a partir da qual são geradas apenas conlfitos para seu sub-problema ela é descartada e o escravo espera por uma nova valoração.

Vale observar que cada escravo recebe valorações do escravo anterior e também do mestre, mas o tratamento dado para elas é exatamente o mesmo, refiná-la em busca de uma solução para seu sub-problema e passá-la para o próximo escravo.

$$
\begin{array}{rl}
K = \{ & \{p_1, p_3, \neg p_4\}, \\
& \{p_1, p_2\}, \\
& \{\neg p_1\}, \\
& \{p_3, p_4\}, \\
& \{p_3\} \\
& \{p_4\} \qquad\qquad \}
\end{array}
$$

Figura 6.2: Problema inicial

Caso o problema seja insatisfazível, em algum momento da execução a seguinte situação irá ocorrer:

O nó mestre terá gerado todas as valorações que satisfazem seu sub-problema e transmitido tais valorações para os escravos, ou seja, o mestre não irá gerar mais valorações. Cada escravo já utilizou todas as valorações recebidas, tanto do mestre quanto do escravo anterior, e não irá gerar mais valorações para repassar ao escravo seguinte.

Nesta situação, todos os escravos estão esperando receber alguma valoração para continuar trabalhando. Porém, como nenhum escravo é capaz de gerar uma valoração sem ter recebido uma antes, eles ficarão esperando eternamente. Quando um nó fica parado, esperando valorações para continuar a busca dizemos que ele está *starving*.

Quando um escravo entra em starving, ele avisa o mestre sobre esta condição. Caso todos os nós avisem o mestre, simultaneamente, que estão em starving, ele conclui que o problema é insatisfazível e emite comando para todos os escravos encerrem sua execução. Após isso a aplicação termina e reporta mensagem ao usuário, afirmando que o problema é insatisfazível.

6.1.1 Exemplo de Execução

A seguir ilustramos a execução de nosso resolvedor SAT para o problema da figura 6.2 utilizando quatro nós, sendo um mestre e três escravos.

No início da execução o mestre lê o problema K e divide o problema em quatro partes, isto é exibido na figura 6.3, onde K_i é enviado para o nó i (0 é o nó mestre).

Após a distribuição dos sub-problemas, o mestre utiliza o algoritmo DPLL para encontrar todas as valorações que satisfazem K_0. Note que estas valorações podem ser parciais, ou seja, podem existir átomos indefinidos.

$$
\begin{array}{lll}
K_0 & = \{ & \{p_1, p_3, \neg p_4\}, \\
 & & \{p_1, p_2\} \quad\quad \} \\
K_1 & = \{ & \{\neg p_1\}, \\
 & & \{p_3, p_4\} \quad\quad \} \\
K_2 & = \{ & \{p_3\} \quad\quad \} \\
K_3 & = \{ & \{p_4\} \quad\quad \}
\end{array}
$$

Figura 6.3: Divisão de K entre os nós

As valorações geradas pelo mestre são:

$$(p1, p2, p3, p4) = \{(T, *, *, *), (F, T, T, *), (F, T, F, F)\}$$

que são enviadas, respectivamente, para os escravos 1,2 e 3.

O escravo 1, ao receber $(T, *, *, *)$, percebe que esta valoração torna K_1 falso e descarta esta valoração.

O escravo 2, ao receber $(F, T, T, *)$, percebe que esta valoração torna K_2 verdadeiro e a repassa para o escravo 3.

O escravo 3, ao receber (F, T, F, F), a descarta, pois esta valoração torna K_3 falso.

O escravo 3 recebe a valoração $v = (F, T, T, *)$ enviada pelo escravo 2. Como esta valoração deixa K_3 indefinido, ela é usada como valoração inicial para o DPLL, que irá gerar $v' = (F, T, T, T)$. v' então é repassada para o nó escravo 1.

O escravo 1 recebe v', percebe que $K_1(v') = 1$ e que v' já passou por todos os demais nós da grade. Portanto v' é uma solução global.

O escravo 1 envia v' para o mestre, que avisa todos os escravos que a execução deve ser encerrada, já que uma solução para o problema foi encontrada.

O mestre reporta a solução para o usuário e termina a execução.

É importante perceber que em alguns momentos um determinado nó pode ficar ocioso, esperando uma valoração chegar até ele. Esta característica é usada para identificar instâncias do SAT que são insatisfazíveis. Pois se todos os nós ficarem ociosos simultaneamente, concluímos que o problema não tem solução.

7 Resultados e Conclusões

7.1 Testes

Para testar o desempenho do resolvedor foram gerados instâncias aleatórias do 3-SAT da forma descrita a seguir.

Fixamos o número de símbolos proposicionais utilizados $N = 50$ e variamos o número de cláusulas $C = 10, 20, 30, \ldots, 490, 500$. Para cada par (N, C_i) geramos 100 instâncias do SAT, sempre com 3 literais em cada cláusula (3-SAT) contendo N literais e C_i cláusulas.

Resolvemos cada problema gerado com nosso resolvedor utilizando apenas uma máquina e anotamos o tempo médio gasto para cada valor de C/N. Repetimos este procedimento com os mesmos problemas em uma grade oportunista com 2, 4, 8, 16 e 32 nós. Estes testes foram todos executados utilizando o Integrade com o modelo BSP.

Para facilitar a interpretação dos resultados utilizamos uma spline cúbica [13, 31] sobre os valores médios obtidos para cada valor C/N e dessa maneira geramos as curvas exibidas nos gráficos a seguir.

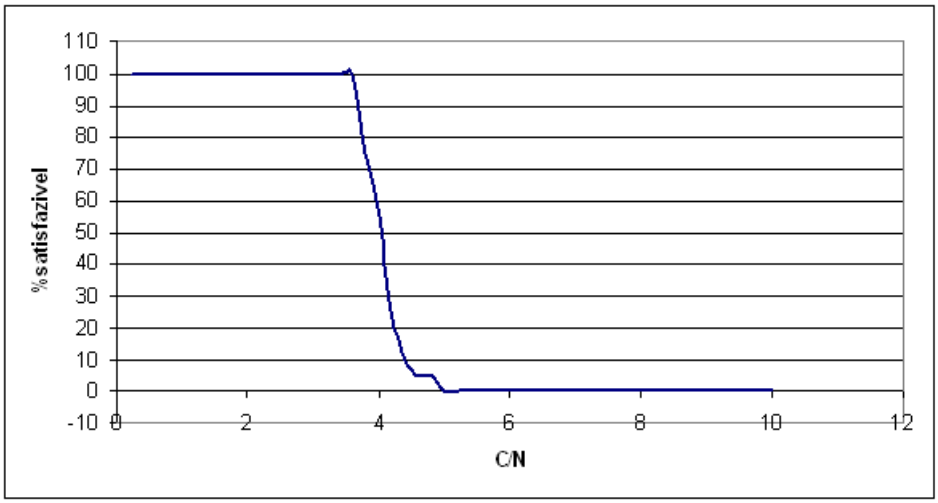


Figura 7.1: Ponto de mudança de fase

7.2 Resultados

Para expor de maneira mais clara os resultados dos teste descritos na seção anterior, construímos alguns gráficos que explicamos a seguir.

Vamos começar analisando a influência que o número de cláusulas e literais exerce sobre o resolvedor.

Problemas que tem poucas cláusulas e muitos literais, ou seja tem C/N próximo de zero, são poucos restritivos e por isso em geral são satisfazíveis. Problemas que tem muitas cláusulas e poucos literais, ou seja, tem C/N suficientemente grande, são muito restritivos e em geral são insatisfazíveis. Para valores intermediários de C/N uma parte dos problemas usados são satisfazíveis e outra insatisfazíveis. Para aumentar a clareza da explicação chamaremos problemas com C/N próximos de zero de *liberais* e problemas com C/N altos de *restritivos*.

A figura 7.1 relaciona o valor de C/N com a percentagem de problemas com C cláusulas e N literais que são satisfazíveis. Como pode ser observado, em torno do ponto $C/N = 4$ exatamente metade dos problemas são satisfazíveis. Este valor é chamado de *ponto de mudança de fase* e é uma característica do SAT. Chamaremos os problemas que tem C/N em torno do ponto de mudança de fase de *críticos*.

A figura 7.2 mostra o tempo médio em função de C/N que o resolvedor gastou para resolver os problemas utilizando apenas uma máquina. É importante notar que os piores tempos

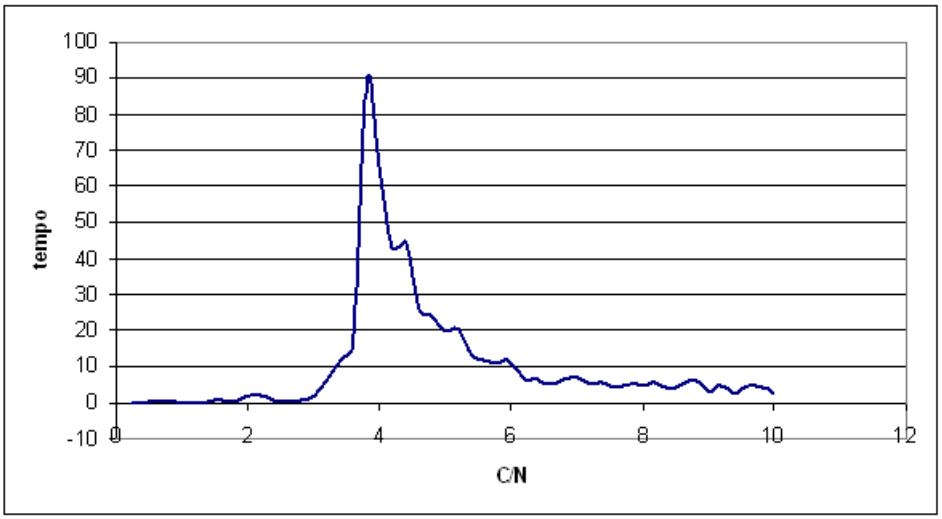


Figura 7.2: Execução para 1 máquina

foram obtidos para problemas críticos, enquanto que para problemas liberais ou restritivos os tempos foram ordens de magnitude menores.

Agora podemos analisar o desempenho do resolvedor utilizando duas máquinas na resolução dos problemas.

A figura 7.3 mostra o tempo em função de C/N utilizando-se duas máquinas na grade. Podemos observar que para problemas liberais e restritivos o tempo é maior que o obtido utilizando apenas uma máquina. Isso acontece pois estamos adicionando sobrecarga devido à comunicação entre os nós, que nestes casos piora o desempenho.

Já para problemas críticos, o desempenho é ligeiramente melhor se comparado ao tempo com apenas uma máquina, neste caso, a sobrecarga com a comunicação entre os nós é compensada pela geração das valorações em paralelo.

As figuras 7.4, 7.5 7.6 e 7.7 mostram os tempo obtidos quando utilizamos 4, 8, 16 e 32 máquinas para resolver os mesmos problemas.

Novamente percebemos, que para problemas liberais ou restritivos, o aumento do número de máquinas piora o desempenho. Esta perda de desempenho é mais acentuada quando comparamos o uso de uma máquina com o uso de duas máquinas. A adição dos demais nós não altera muitos os resultados para estes problemas, já que o aumento de comunicação é pequeno pois cada nó ira gerar pouca informação para transmitir aos demais.

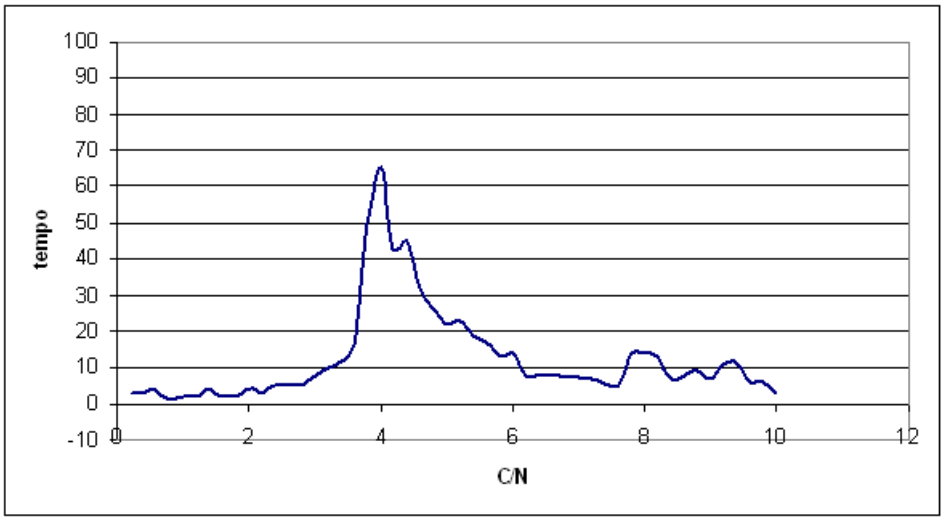


Figura 7.3: Execução para 2 máquinas

Para problemas críticos temos um ganho de desempenho com o aumento do número de nós. Comparando o uso de uma máquina contra 32 por exemplo, o tempo médio caiu de 0.9 segundos para 0.1 segundos, isso ocorre por causa do paralelismo utilizado.

Outra observação interessante é o que ocorre com problemas que tem C/N ligeiramente maior que o ponto de mudança de fase, estes problemas, quando utilizamos apenas uma máquina tiveram um desempenho razoável comparado ao ponto de mudança de fase, mas ao aumentarmos o número de máquinas o crescimento do tempo gasto para resolve-los é alto. $C/N = 5$ por exemplo, passaram de 0.2 segundos com uso de apenas uma máquina para 0.6 segundos com 32 máquinas.

7.3 Conclusões

Após analisar os testes e gráficos gerados obtemos três conclusões principais:

1. Problemas distantes do ponto de mudança de fase, liberais ou restritivos são resolvidos mais facilmente por um resolvedor seqüencial, que não utiliza computação distribuída;
2. Problemas que estão no ponto de mudança de fase geram grandes ganhos de desempenho ao aumentarmos o número de máquinas utilizadas, portanto são indicados para um resolvedor distribuído;

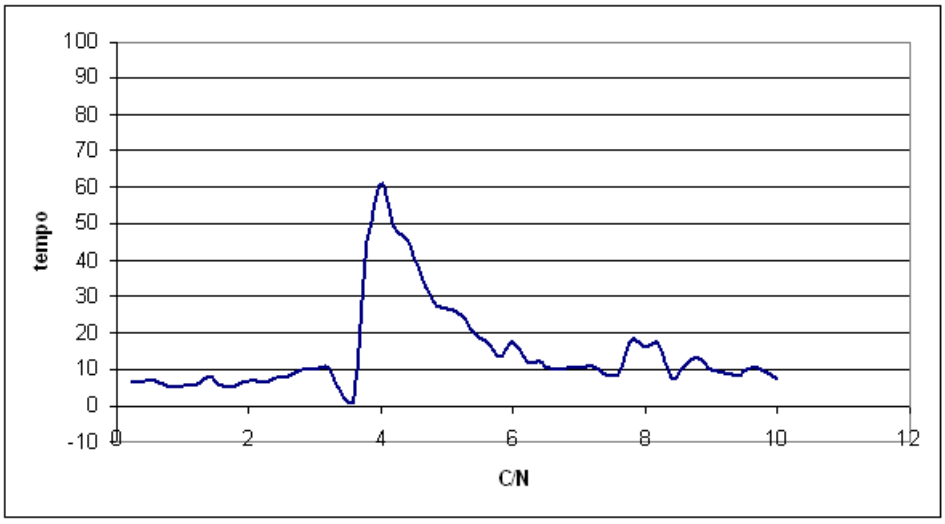


Figura 7.4: Execução para 4 máquinas

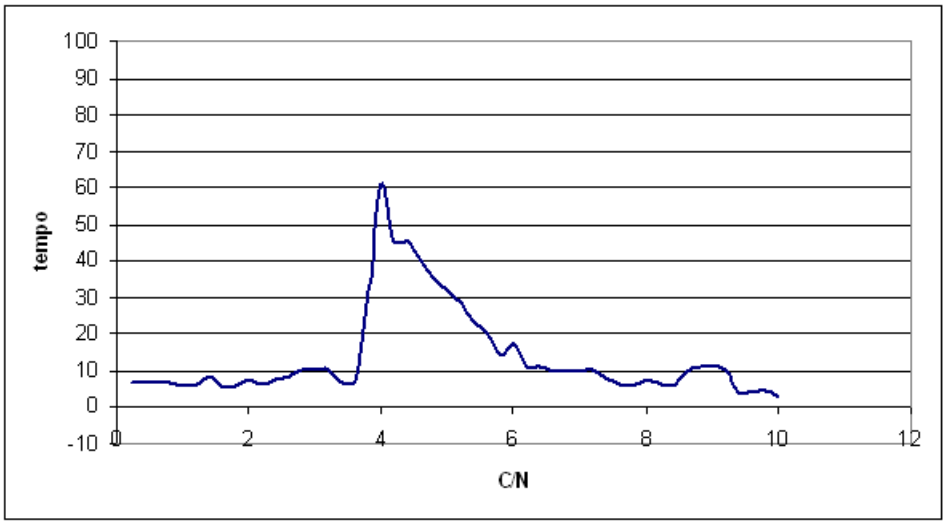


Figura 7.5: Execução para 8 máquinas

3. Problemas que estão ligeiramente acima do ponto de mudança de fase geram uma perda de desempenho grande ao aumentarmos o número de máquinas utilizadas e também são melhor resolvidos por um resolvedor seqüencial.

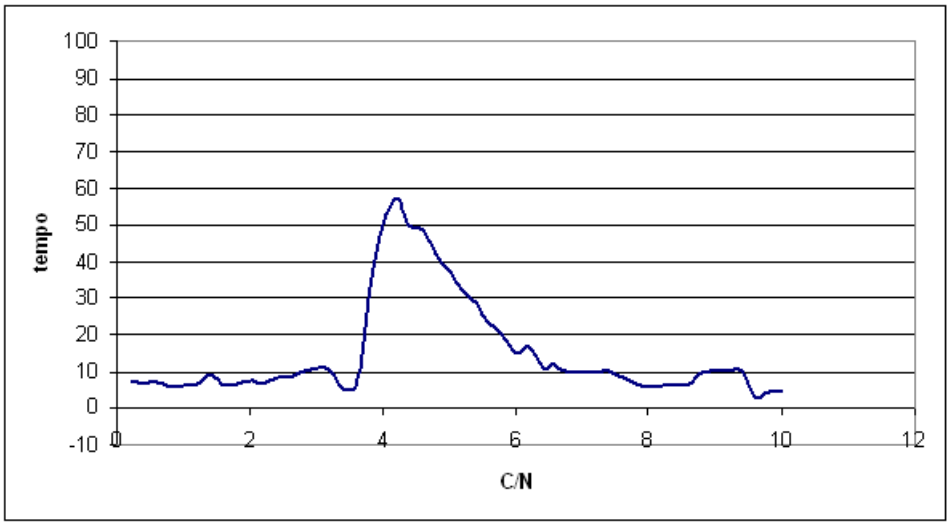


Figura 7.6: Execução para 16 máquinas

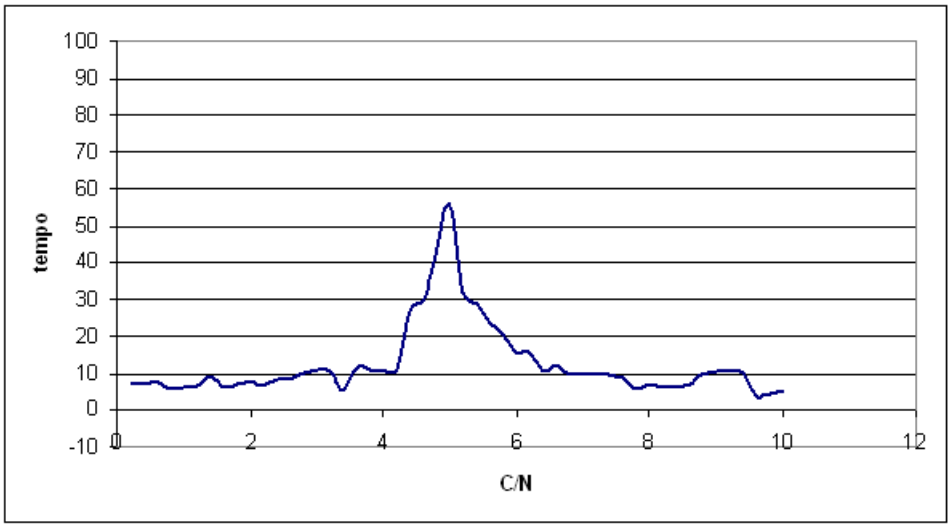


Figura 7.7: Execução para 32 máquinas

Referências Bibliográficas

[1] Bittorrent website. www.bittorrent.com. Acessado em dezembro de 2006.

[2] Emule website. www.emule-project.net. Acessado em dezembro de 2006.

[3] Integrade: Object-oriented middleware for grid computing. http://gsd.ime.usp.br/integrade. Acessado em dezembro de 2006.

[4] The oxford bsp toolset and profiling system. http://www.bsp-worldwide.org/implmnts/oxtool. Acessado em dezembro de 2006.

[5] Search for extraterrestrial intelligence. http://setiathome.ssl.berkeley.edu. Acessado em dezembro de 2006.

[6] United devices. http://www.ud.com/home.htm. Acessado em dezembro de 2006.

[7] C. Amza and A. Cox. Treadmarks: Shared memory computing on networks of workstations, 1996.

[8] Ronald T. Chin and Charles R. Dyer. Model-based recognition in robot vision. In *ACM Computing Surveys*, pages 67–108, 1986.

[9] Wahid Chrabakh and Rich Wolski. Gridsat: A chaff-based distributed sat solver for the grid. November 2003.

[10] J. Crawford and L. Auton. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 22–28, 1993.

[11] J. Gu andP. M. Pardaros D. Du. Satisfiability problem: Theory and applications. *American Mathematical Society*, 1997.

[12] M. Davis, G. Logemann, and D. Loveland. *A machine program for theorem proving*, chapter 5:394-397. Communications of the ACM, 1962.

[13] O. Davydov, G. urnberger, and F. Zeilfelder. Cubic spline interpolation on nested polygon triangulations, 2000.

[14] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems, 2002.

[15] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report UT-CS-93-214, 1993.

[16] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[17] Andrei Goldchleger. Integrade: Um sistema de middleware para computação em grade oportunista. Master's thesis, Department of Computer Science - University of São Paulo, December 2004.

[18] Andrei Goldchleger, Fabio Kon an Alfredo Goldman, Marcelo Finger, , and Germano Capistrano Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice & Experience*, 16:449–459, March 2004.

[19] Andrei Goldchleger, Fabio Kon, Alfredo Goldman Vel Lejbman, Marcelo Finger, and Siang Wun Song. Integrade: Rumo a um sistema de computação em grade para aprovei-

tamento de recursos ociosos em máquinas compartilhadas. Technical report, IME-USP, October 2002.

[20] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1990.

[21] B. Jurkowiak, C. Li, and G. Utard. Parallelizing satz using dynamic workload balancing, 2001.

[22] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planing. In *Workshop on Ligic-Based Artificial Intelligence*, 1999.

[23] Valiant LG. A bringing model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[24] C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, 1997.

[25] L. Lopes and F. Silva. Thread-and process-based implementations of the psystem parallel programming envinment. *Software Practice and Experience*, 27(3):329–351, 1997.

[26] Jeferson Roberto Marques and Fabio Kon. Gerenciamento de recursos distribuídos em sistemas de grande escala. In *20th Brazilian Symposium on Computer Networks (SBRC'2002)*, 2002.

[27] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.

[28] João Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. pages 62–74.

[29] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[30] David D. Palmer. SATZ - an adaptive sentence segmentation system. Master's thesis, 1994.

[31] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd ed pp 107-110*. Cambridge University Press, 1992.

[32] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers. Master's thesis, Simon Fraser University, 2004.

[33] Routo Terada. *Desenvolvimento de algoritmos e complexidade de computação*. Terceira escola de computação, PUC-Rio de Janeiro, julho 1982.

[34] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfability procedure in the formal verification of superscalar and vliw microprocessors. In *Proceedings of the Design Automatin Conference*, pages 226–231, June 2001.

[35] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.