

João Guilherme Zeni

**PARAQUANTUMSAT: UM ALGORITMO SAT SOLVER
DISTRIBUÍDO**

Dissertação submetido ao Programa de ...
para a obtenção do Grau de Bacharel em
Ciências da Computação.

Orientador

Universidade Federal de Santa Catarina:
Prof. Dra. Jerusa Marchi

Coorientador

Universidade Federal de Santa Catarina:
Prof. Dr. Mario Dantas

Florianópolis

2013

Ficha de identificação da obra elaborada pelo autor através do
Programa de Geração Automática da Biblioteca Universitária da UFSC.

A ficha de identificação é elaborada pelo próprio autor

Maiores informações em:

<http://portalbu.ufsc.br/ficha>

Este trabalho é dedicado aos meus colegas de classe e aos meus queridos pais.

AGRADECIMENTOS

Inserir os agradecimentos aos colaboradores à execução do trabalho.

Texto da Epígrafe. Citação relativa ao tema do trabalho. É opcional. A epígrafe pode também aparecer na abertura de cada seção ou capítulo.
(Autor da epígrafe, ano)

RESUMO

Computação paralela é uma área de estudo crítica, visto que hoje a capacidade de processamento dos computadores cresce com a paralelização dos processadores. Um dos principais desafios em computação paralela é a construção de algoritmos paralelos eficientes para resolver problemas clássicos de computação. Este trabalho se propõe a desenvolver um algoritmo *SAT solver* paralelo, tendo como base um algoritmo SAT solver sequencial.

Palavras-chave: Palavras Chave 1.Algoritmos Paralelos 2.*SAT Solver*

ABSTRACT

Resumo traduzido para outros idiomas, neste caso, inglês. Segue o formato do resumo feito na língua vernácula. As palavras-chave traduzidas, versão em língua estrangeira, são colocadas abaixo do texto precedidas pela expressão “Keywords”, separadas por ponto.

Keywords: Keyword 1. Keyword 2. Keyword 3.

LISTA DE FIGURAS

Figura 1	Arvore de decisão do DPLL	31
Figura 2	Diagrama que apresenta os procedimentos e componentes do CDCL.....	32
Figura 3	Funcionamento de um algoritmo portfólio(NELSON, 2013) .	38
Figura 4	Arquitetura do PMSat	40
Figura 5	Dinâmica dos nós.....	47
Figura 6	Estados iniciais.....	48
Figura 7	Comparação com o glucose.....	51
Figura 8	Teste de escalabilidade	52
Figura 9	Teste de escalabilidade	54

LISTA DE TABELAS

Tabela 1	Tabela lógica.....	27
Tabela 2	Taxonomia de Flynn.....	37
Tabela 3	Resultados para Glucose e ParaQuantumSAT.....	52
Tabela 4	Tabele de resultados de escalabilidade.....	53

LISTA DE ABREVIATURAS E SIGLAS

DPLL	Davis-Putnam-Logemann-Loveland, uma classe de algoritmos	
	<i>SAT solver</i>	55
CDCL	Conflict-Driven Clause Learning, uma classe de algoritmos	
	<i>SAT solver</i>	55
CPU	55

LISTA DE SÍMBOLOS

ϕ	Phi, representa a fórmula proposicional	55
\prod	Produtório	55

SUMÁRIO

1	INTRODUÇÃO	23
1.1	OBJETIVOS	24
1.1.1	Objetivos Específicos	24
1.2	JUSTIFICATIVA	24
1.3	APRESENTAÇÃO DO TRABALHO	25
2	FUNDAMENTAÇÃO TEÓRICA	27
2.1	O PROBLEMA SAT	27
2.1.1	Lógica Proposicional	27
2.1.2	Funcionamento do Problema SAT	28
2.1.3	Importância do Problema SAT	30
2.1.4	Algoritmos SAT Solvers	30
2.1.5	Algoritmo Base	34
2.2	COMPUTAÇÃO PARALELA	36
2.2.1	Paralelismo no problema SAT	38
2.2.2	Algoritmos portfólio	38
2.2.3	Particionamento do espaço de busca	39
2.2.4	Particionamento do problema	41
3	DESENVOLVIMENTO	43
3.1	TECNOLOGIAS UTILIZADAS	43
3.2	ALGORITMO DE BUSCA	43
3.2.1	Busca Paralela	45
3.3	EXEMPLO	46
4	RESULTADOS EXPERIMENTAIS E ANÁLISES	49
4.1	CENÁRIO	49
4.2	RESULTADOS E ANÁLISE DE TEMPO	50
4.3	RESULTADOS E ANÁLISE DE ESCALABILIDADE	51
4.4	RESULTADOS E ANÁLISE DE USO DOS NÓS	53
5	TRABALHOS FUTUROS E CONCLUSÃO	55
5.1	TRABALHOS FUTUROS	55
5.2	CONCLUSÃO	55
	REFERÊNCIAS	57
	APÊNDICE A – Exemplificando um Apêndice	63
	ANEXO A – Exemplificando um Anexo	67

1 INTRODUÇÃO

A computação paralela surgiu como um campo de estudo utilizado apenas em computação científica, dada a grande demanda por poder computacional desta área(COULOURIS et al., 2011). Hoje porém computação paralela está sendo utilizada em novos cenários, o que tem motivado o crescente estudo de algoritmos paralelos. Um desses cenários é o de *internet of things(IoT)*, pois em IoT funciona com diversos dispositivos inteligente que se comunicam(BUSEMANN et al., 2012). Outro cenário é o de processadores multi-core que se tornaram populares desde que foi atingida a barreira de potência em processadores *single-core*, que fez com que não seja mais possível aumentar a potência de um processador individualmente (PATTERSON; HENNESSY, 2012).

O problema da satisfazibilidade booleana(SAT)(BIERE et al., 2009) consiste em determinar, para uma dada fórmula booleana, se existe uma atribuição de valores para as variáveis para a qual a fórmula é satisfeita. SAT é um problema NP-Completo, logo sua resolução por si só é um dos maiores problemas da computação. Porém o SAT tem aplicações diretas em outras áreas da computação como verificação formal e inteligência artificial, tornando-o um problema interessante para o uso e estudo de computação paralela.

Computação paralela(KUMAR, 2002) começou a despertar o interesse dos pesquisadores de algoritmos *SAT solvers* quando foi alcançada a barreira de potência em processadores *single-core*, pois estes a consideravam uma boa técnica para resolver o problema de forma rápida. Além disso com a massiva utilização de processadores *multi-core* é necessário que os *SAT solvers* funcionem bem nestes ambientes.

A principal abordagem ao problema tem sido aplicar técnicas de paralelismo, como dividir-e-conquistar, em algoritmos de *SAT solving* consolidados, como DPLL(SELMAN; KAUTZ; COHEN, 1995). Porém essa abordagem apresenta alguns problemas, como a necessidade de adicionar mecanismos de controle de carga aos algoritmos. Apesar de apresentar problemas esta abordagem é bastante utilizada em computação paralela por apresentar diversas vantagens, como aproveitar as otimizações já realizadas nos algoritmos sequencias, este algoritmos já serem bastante confiáveis e a popularidade já estabelecida nestes algoritmos.

Neste trabalho, também utilizamos um algoritmo *SAT solver* sequencial como base para o desenvolvimento do algoritmo paralelo, porém utilizamos um novo algoritmo *SAT solver* sequencial que apresenta algumas vantagens para sua paralelização, este algoritmo nunca foi paralelizado anteriormente, além disso também escolhemos uma linguagem de programação

pouco popular em *SAT solvers*, que apresenta algumas vantagens para sua implementação. As vantagens e desvantagens são apresentadas neste trabalho.

1.1 OBJETIVOS

O presente trabalho tem por objetivo geral o desenvolvimento de um algoritmo *SAT solver* paralelo, tendo como base um algoritmo sequencial.

1.1.1 Objetivos Específicos

Os objetivos específicos deste trabalho estão listados abaixo:

1. Compreender as técnicas de computação paralela e quais são mais adequadas para se utilizar no algoritmo *SAT solver*.
2. Compreender a abordagem utilizada no algoritmo *SAT solver* QuantumSAT.
3. Desenvolver a versão paralela do algoritmo *SAT solver*.
4. Testar o algoritmo desenvolvido, usando teorias SAT aleatórias.
5. Comparar os resultados do algoritmo desenvolvido com outros já existentes.
6. Submeter o algoritmo a uma competição, visando sua avaliação definitiva.
7. Redação do TCC e dos artigos técnicos.

1.2 JUSTIFICATIVA

O SAT é um problema que apresenta diversas aplicações na indústria atual, além disso ele também possui grande importância teórica na ciência da computação.

A computação paralela sempre possuiu grande importância quando se trata de aplicações científicas ou de grandes volumes de dados/processamento, porém a partir do momento que atingiu-se a barreira de potência em processadores *single-core* a área tem ganhado importância em todas as áreas da computação, dado que processadores multi-core hoje são os mais usados.

Logo a pesquisa de algoritmos *SAT solvers* paralelos se justifica na intersecção entre as justificativas para a pesquisa nessas duas áreas, dado que hoje todas as áreas que utilizando *SAT solver* tem como ferramenta de trabalho processadores multi-core e necessitam de algoritmos que funcionem de forma eficiente nesses processadores.

1.3 APRESENTAÇÃO DO TRABALHO

Este trabalho é dividido em outros quatro capítulos, conforme segue.

No capítulo dois é a fundamentação teórica do trabalho, neste capítulo são apresentados os conceitos necessários para a compreensão do trabalho desenvolvido, este capítulo apresenta inicialmente o problema SAT, seguindo apresenta a área de computação paralela e por fim a intersecção entre o problema SAT e computação paralela.

No capítulo três apresenta o desenvolvimento do trabalho, neste capítulo é apresentado o algoritmo paralelo desenvolvido, um exemplo de funcionamento do sistema desenvolvido e como as decisões de projeto foram tomadas.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são descritos os fundamentos necessários para a compreensão do trabalho desenvolvido.

2.1 O PROBLEMA SAT

2.1.1 Lógica Proposicional

É o campo da lógica que lida com o estudo de proposições. Esta área é de vital importância para a lógica e para a ciência da computação, tendo sido estudada por um longo período por diversos lógicos, porém estudo moderno da área começa com o livro *Logic of Relatives*(1883) de Charles Peirce, nos anos 1890 Gottlob Frege começa a estudar e desenvolver o conceito de sentenças, sendo o termo função proposicional cunhado por Bertrand Russell em *Principles of Mathematics*(1903). Estes estudos tem sido amplamente aplicados em diversas áreas de ciência da computação como gramáticas formais, teoria da computação, algoritmos entre outros.

Uma proposição, que consiste de conjunto de variáveis ou símbolos proposicionais, que possuem valores verdade, sendo os valores verdade ou falso, e pode ser formada de outras proposições com o uso de conectores lógicos. Conectores lógicos compõem proposições através da ligação de símbolos proposicionais, para este trabalho notam-se os conectores *AND* representado por \wedge , *OR* representado por \vee e *NOT* representado por \neg . A semântica destes operadores é descritos na tabela 1.Outra estrutura são literais, que são variáveis que podem assumir valores verdade. Fórmulas proposicionais são escritas utilizando conectores lógicos, símbolos proposicionais e literais e são utilizado para se trabalhar com sentenças lógicas. Neste trabalho fórmulas proposicionais serão representados pelo símbolo ϕ .

Entradas		Saídas		
A	B	$A \wedge B$	$A \vee B$	$\neg A$
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

Tabela 1 – Tabela lógica

Fórmulas proposicionais podem ser representadas de diversas formas, porém toda fórmula proposicional pode ser representada nas duas formas canônicas, a forma normal conjuntiva (CNF, sigla em inglês) e a forma normal disjuntiva (DNF, sigla em inglês).

As fórmulas proposicionais em CNF são representadas como uma conjunção de cláusulas, sendo cada cláusula uma disjunção de literais, onde cada literal pode assumir Verdade ou Falso, como é apresentado abaixo na equação 2.1. Toda fórmula proposicional pode ser convertida para forma CNF através de um procedimento trivial.

$$F = C_1 \wedge C_2 \wedge C_3 \dots \wedge C_k \mid \forall C, C_n = L_1 \vee L_2 \vee L_3 \dots \vee L_k \quad (2.1)$$

As fórmulas proposicionais em DNF são representadas como uma disjunção de cláusulas, sendo cada cláusula uma conjunção de literais, como é apresentado abaixo na equação 2.2. A conversão de uma fórmula proposicional para forma DNF é por si só uma solução para o problema, portanto uma tarefa NP-Completa. Para realizar esta conversão existem diversas técnicas, neste trabalho será apresentada em detalhes uma delas.

$$F = D_1 \vee D_2 \vee D_3 \dots \vee D_k \mid \forall D, D_n = L_1 \wedge L_2 \wedge L_3 \dots \wedge L_k \quad (2.2)$$

Estes conceitos são importantes pois o problema SAT é amplamente relacionada com uma sub-classe de fórmulas proposicionais.

2.1.2 Funcionamento do Problema SAT

Proposição é uma sentença a qual pode ser atribuído um sentido, chamado de asserção. Uma proposição pode ser simples ou compostas. Neste trabalho utilizaremos proposições lógicas, cujas asserções são Verdadeiro ou Falso. Proposições lógicas simples correspondem a variáveis. Proposições complexas são proposições simples conectadas por operadores lógicos. Quando existem variáveis sem valor atribuído uma proposição chama-se de fórmula proposicional, representada neste trabalho por Φ .

Fórmulas proposicionais são escritas utilizando operadores lógicos, que são símbolos utilizados para se trabalhar com sentenças lógicas. Estes operadores possuem caráter binário, sendo usados para conectar duas sentenças lógicas, e caráter unário, para atribuir uma propriedade a uma sentença lógica. Para este trabalho notam-se os operadores *AND* representado por \wedge , *OR* representado por \vee e *NOT* representado por \neg . As interpretações destes operadores

são descritos na tabela 1.

O problema SAT é definido sobre uma linguagem de lógica proposicional $\mathcal{L}(P)$, onde $P = p_1, p_2, \dots, p_n$ sendo que todo p é um símbolo proposicional sem valor verdade definido. Uma fórmula $\Phi \in \mathcal{L}(P)$ pode ser escrita em CNF, logo $\Phi = \{C_1 \wedge C_2 \dots \wedge C_k | \forall C, C_n = p_i \vee \neg p_j \dots \vee p_z\}$. Com isto pode-se definir o problema SAT como sendo responder se existe um conjunto de atribuição de valores verdade a P , o qual faz com que Φ seja verdade, caso exista esse conjunto Φ é dito satisfazível. O exemplo abaixo apresenta um problema SAT.

Seja Φ :

$$\begin{aligned} 0 : & (\neg p_4 \vee p_2 \vee \neg p_3) \wedge \\ 1 : & (\neg p_3 \vee p_2 \vee \neg p_1) \wedge \\ 2 : & (p_5 \vee \neg p_2 \vee \neg p_3) \wedge \\ 3 : & (\neg p_2 \vee p_5 \vee \neg p_1) \wedge \\ 4 : & (\neg p_3 \vee p_5 \vee p_1) \wedge \\ 5 : & (p_4 \vee p_5 \vee p_2) \wedge \\ 6 : & (\neg p_5 \vee p_3 \vee \neg p_2) \wedge \\ 7 : & (\neg p_1 \vee p_4 \vee p_3) \wedge \\ 8 : & (\neg p_3 \vee \neg p_2 \vee \neg p_1) \wedge \\ 9 : & (\neg p_3 \vee \neg p_4 \vee \neg p_1) \end{aligned}$$

Assim podemos ver que se $\neg p_2$ for Verdade então as cláusulas 2, 3, 6 e 8 serão Verdade, se $\neg p_3$ for Verdade então as cláusulas 0, 1, 4 e 9 serão Verdade e se p_4 for Verdade então as cláusulas 5 e 7 serão Verdade, como foi possível atribuir valores que fizeram com que todas as cláusulas de Φ sejam Verdade disse que Φ é satisfazível.

Outra característica do problema SAT é número de símbolos em cada cláusula, o problema usualmente é chamado de k SAT onde k é número de símbolos em cada cláusula, no caso anterior o problema é um 3SAT. Esta característica é importante pois toda fórmula proposicional pode ser escrito como um 3SAT. Outro fato importante é o de que os problemas que podem ser escritos em 2SAT possuem complexidade $O(n)$, porém não é garantido que todas as fórmulas proposicionais possam ser escritas na forma 2SAT (BIERE et al., 2009).

O padrão de entrada da grande maioria dos *SAT solvers* é o Dimacs, que representa fórmulas proposicionais na forma CNF.

2.1.3 Importância do Problema SAT

A resolução de problemas SAT possui aplicação direta em diversas áreas, nesta seção serão apresentadas algumas delas e como SAT é utilizado nestas aplicações.

O projeto de sistemas digitais é amplamente auxiliado pelas mais diversas ferramentas, muitas delas se utilizam de *SAT solvers* (LAVAGNO; MARTIN; SCHEFFER, 2006). Uma delas é verificação de equivalência, que consiste em verificar se para dois circuitos digitais distintos com o mesmo conjunto de entradas o conjunto de saídas será igual, para se resolver este problema as funções de entradas e saídas dos circuitos são escritos como uma fórmula proposicional e verifica-se a satisfazibilidade com um *SAT solver*.

Uma aplicação bem sucedida da solução de SAT é o *automate planning* em inteligência artificial (KAUTZ; SELMAN, 1992). O problema *planning* consistem em, dado um conjunto de estados e um conjunto de transições entre estados, descobrir se é possível sair de um estado inicial e atingir um estado objetivo em um número finito de transições. Uma solução para este problema se chama *satplan* (KAUTZ; SELMAN, 1992), que consiste em converter o problema do *planning* para uma fórmula proposicional e verificar se a mesma é satisfazível, isto é feito de forma iterativa começando do planejamento com 1 passo e indo até o limite de passos dado pelo problema.

Outra aplicação da solução SAT importante são os *model checkers* (BIERE et al., 2009), que consistem em dado um modelo automaticamente verificar se este modelo atende a dadas especificações. *Model checkers* possuem ampla aplicação, principalmente em teste de sistemas, que vão de hardware a protocolos de comunicação. Este problema apresenta uma solução similar a do *planning*, pois sua solução consiste em converter o modelo em uma fórmula proposicional e a partir daí utilizar um *SAT solver* (CLARKE et al., 2001). Esta técnica é a mais utilizada nos casos de verificadores restritos e quando as especificações são dadas através de fórmulas em lógica temporal.

2.1.4 Algoritmos SAT Solvers

Os *SAT solvers* podem ser divididos em classes, sendo que as duas principais classes de algoritmos *SAT solvers* sequenciais são os baseados em conflito de cláusulas (CDCL, sigla em inglês), a versão mais moderna do DPLL (Davis-Putnam-Logemann-Loveland) (DAVIS; LOGEMANN; LOVELAND, 1962), e os baseados em busca local estocástica (SELMAN; KAUTZ; COHEN, 1995). O algoritmo *SAT solver* desenvolvido neste trabalho é baseado na conversão CNF-DNF. Não existem muitos outros *solver* baseados nesse

método, em (MILTERSEN; RADHAKRISHNAN; WEGENER, 2005a) (KATAJAINEN; MADSEN, 2002) este método é discutido.

A classe CDCL, e sua versão original o DPLL, são baseados na técnica de *backtracking* (DAVIS; LOGEMANN; LOVELAND, 1962). Estes algoritmos consistem em atribuir um valor verdade a uma literal, simplificar a fórmula e, recursivamente, verificar se a fórmula simplificada é satisfazível. No caso do DPLL a ideia básica é recursivamente montar uma árvore de decisão, como a apresentada na figura 1, além disto são utilizadas diversas heurísticas para melhor eficiência do algoritmo, destacam-se aqui as heurísticas para a escolha do literal a se atribuir o valor verdade (OUYANG, 1996). No caso do CDCL, é montada uma estrutura adicional que o permite ignorar alguns passos, para em teorias geradas aleatório baixar o tempo médio de execução (SILVA; SAKALLAH, 1996). O funcionamento básico do CDCL é mostrado na figura 2.

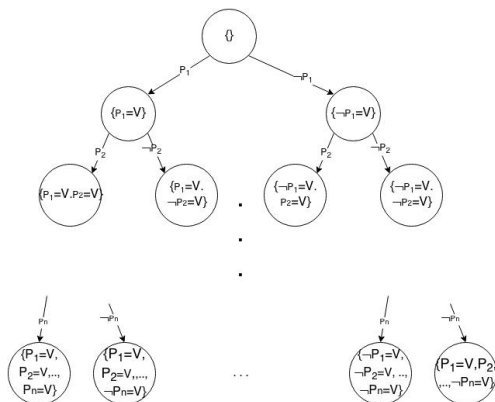


Figura 1 – Árvore de decisão do DPLL

O algoritmo 1 é o pseudo código de uma versão básica do DPLL. Esta versão é bastante simples. O primeiro passo é eliminar as cláusulas unitárias, aquelas que contêm apenas um literal, e as cláusulas que contêm o literal da cláusula unitária, pois para a fórmula ser satisfazível este literal precisa ser Verdade. O segundo passo é eliminar as cláusulas que contêm literais puros, aqueles que apresentam apenas uma polaridade em toda a fórmula, pois sempre é possível se atribuir Verdade a estes símbolos. O terceiro é selecionar aleatoriamente um literal. O quarto é executar este algoritmo recursivamente, considerando Verdade para este literal e para sua negação.

Uma implementação do algoritmo CDCL é o *solver* MiniSAT (EEN; SÅRENSSON, 2003). Este *solver* implementa várias otimizações que re-

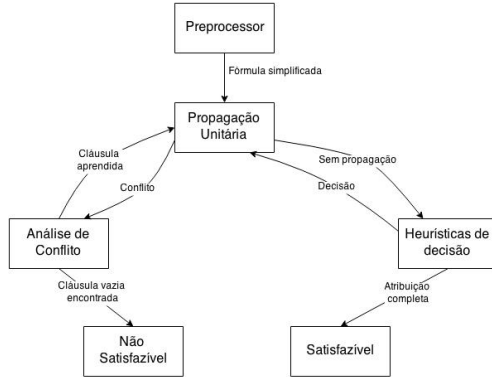


Figura 2 – Diagrama que apresenta os procedimentos e componentes do CDCL

Algorithm 1 Pseudo código DPLL

```

1: procedure DPLL( $\Phi$ )
2:   for Cláusula unitária  $l$  em  $\Phi$  do
3:      $\Phi \leftarrow atribui(l, \Phi)$ 
4:   for Literal puro  $l$  em  $\Phi$  do
5:      $\Phi \leftarrow atribui(l, \Phi)$ 
6:    $l \leftarrow escolhaDeLiteral(\Phi)$ 
7:   return  $DPLL(\Phi \wedge l) \vee DPLL(\Phi \wedge \neg l)$ 

```

sultam em melhorias significativas em casos médios. Vale destacar a técnica de minimização de cláusulas conflitantes, pois em exemplos indústrias é bastante comum que, mais de 30% dos literais em uma cláusula conflitante, sejam redundantes(EEN; SÖRENSSON, 2005).

Já os algoritmos baseados em busca local estocástica consistem em atribuir valores verdade a literais de forma iterativa e ir, constantemente, melhorando a solução até que a fórmula seja satisfazível(SELMAN; KAUTZ; COHEN, 1995). Estes são algoritmos gulosos, pois utilizam os literais que eliminam o maior número de cláusulas. Possuem diversas formas de otimização, por exemplo: aumentar a quantidade de literais a atribuir valor verdade em cada passo, implementar métodos para melhorar a qualidade da seleção de literais de forma iterativa, entre outras. A principal falha deste método é sua incompletude, dado que ele é incapaz de afirmar se uma dada fórmula proposicional é insatisfazível(HOOS; STTZLE, 2004).

O algoritmo 2 é um pseudo código de uma versão básica de um algo-

ritmo de busca local estocástica para SAT. O primeiro passo é executar um laço para o número de tentativas definido. O segundo é atribuir a todos os símbolos um valor verdade aleatório. O terceiro é executar um laço para um número de mudanças de símbolos definido. Dentro deste laço é verificado se Φ está satisfeito, caso esteja é retornado Φ . Caso não esteja é feita uma mudança aleatória no valor verdade de um símbolo. Caso não seja encontrada uma solução o algoritmo informa que não foi possível encontrar uma solução.

Algorithm 2 Pseudocódigo de um algoritmo de busca local

```

1: procedure BUSCALocal( $\Phi$ )
2:   for  $i := 1$  to  $nTentativas$  do
3:      $\Phi \leftarrow \text{valoresVerdadeAleatórios}(\Phi)$        $\triangleright$  atribui valores verdade
        aleatórios aos literais
4:     for  $i := 1$  to  $nMudanças$  do
5:       if  $\Phi$  é satisfeito then
6:         return  $\Phi$ 
7:        $\Phi \leftarrow \text{mudancaDeLiteral}(\Phi)$        $\triangleright$  muda o valor verdade do
        literal que reduzir o maior número de cláusulas
8:   return Não foi encontrada atribuição satisfazível

```

Uma implementação desta técnica é o *solver* WalkSAT (SELMAN; KAUTZ; COHEN, 1995), este *solver* utiliza uma otimização que consiste no modo de escolha do literal a alterar o valor verdade, na maioria dos algoritmos a escolha é feita de modo a atingir o maior número de cláusulas. No WalkSAT esta escolha é feita tendo como base quais são as cláusulas que estão insatisfeitas. Este *solver* possui um bom resultado para fórmulas proposicionais que foram geradas a partir de um problema de *planning*, sendo esta sua principal aplicação (KAUTZ; SELMAN, 1996).

Existem outras diversas classes de *SAT solvers* que possuem usos específicos. Neste trabalho é utilizado como base um algoritmo SAT de uma classe diferente, que é baseada na conversão CNF-DNF. Uma vez que a solução de SAT para fórmulas proposicionais em DNF é trivial, a conversão é, por si só, uma resolução. Esta técnica é bastante antiga sendo sua forma clássica a utilização da lei de De-Morgan para efetuar a conversão (SKIENA, 2008), porém esta solução é extremamente ineficiente (MILTERSEN; RADHAKRISHNAN; WEGENER, 2005b). Na próxima seção é apresentado o algoritmo base para este trabalho, que efetua esta conversão escrevendo a fórmula proposicional na forma DNF utilizando uma nova estrutura chamada quantum, que assegura maior eficácia ao processo de conversão.

2.1.5 Algoritmo Base

Será utilizado como base o algoritmo *SAT solver* QuantumSAT(BITTENCOURT; MARCHI; PADILHA, 2003). Este é um algoritmo *SAT solver* completo, cuja técnica para a solução baseia-se em converter uma fórmula proposicional escrita em CNF para DNF.

A ideia para se calcular a teoria em DNF consiste em encontrar os símbolos a se atribuir Verdade para que cada cláusula possua pelo menos um símbolo com valor atribuído Verdade. Os símbolos encontrados formam uma cláusula DNF e são chamados de cláusulas duais mínimas.

Cada cláusula dual mínima representa um conjunto de atribuições que satisfaz Φ . Cada símbolo da cláusula dual mínima representa pelo menos uma cláusula de Φ .

Para se calcular as cláusulas duais mínimas de Φ o QuantumSAT utiliza uma estrutura chamada quantum. O quantum, consiste de um par (ϕ, F) , onde ϕ é um símbolo da fórmula proposicional e F é um conjunto das coordenadas das cláusulas que contém ϕ . A notação utilizada para o quantum é ϕ^F . O algoritmo utiliza tanto o quantum do literal quanto o da negação do mesmo. Ao coletivo de quantum, chama-se quanta.

O quantum associado a negação de um literal chama-se *mirror* do quantum, sendo representado como o complemento do quantum, portanto o quantum do literal $\neg\phi$ é o *mirror* do quantum do literal ϕ , denotado por $\bar{\phi}^F$.

Seja Φ o mesmo do exemplo anterior, então a lista de quanta de Φ é:

$$P_1^{\{4\}}, \neg P_1^{\{1,2,7\}}, P_2^{\{0,1,5\}}, \neg P_2^{\{2,3,6,8\}}, P_3^{\{6,7\}}, \neg P_3^{\{0,1,2,4,8,9\}}, \\ P_4^{\{5,7\}}, \neg P_4^{\{0,9\}}, P_5^{\{2,3,4,5\}}, \neg P_5^{\{6\}}$$

Para tanto o primeiro passo do algoritmo é construir o conjunto de todos os quanta, o segundo é definir a ordem de quanta iniciais da busca de sucessores, o terceiro é realizar a expansão dos sucessores dos quanta. Nos próximos parágrafos são explicados os passos, além de algumas otimizações propostas para o algoritmo.

Para se construir o conjunto de quanta é feita uma busca, percorrendo todas as cláusulas e verificando quais literais as compõem. Como pode ser visto no algoritmo 3.

Algorithm 3 Algoritmo de montagem dos quanta

```

1: procedure QUANTUMMAKER( $\Phi$ )    ▷ Recebe uma fórmula no formato
   CNF
2:   for each  $\phi$  in  $\Phi$  do
3:      $F(\phi) \leftarrow \emptyset$                                 ▷ Inicializa a lista de quanta
4:   for each  $C$  in  $\Phi$  do
5:     for each  $\phi$  in  $C$  do
6:        $F(\phi) \leftarrow C$ 
7:   return  $F$ 
  
```

Com a lista quanta de Φ montada encontrar a representação CNF de Φ é uma busca em um espaço de estados, em que cada estado corresponde a um caminho para uma possível cláusula dual mínima.

Para realizar esta busca é utilizado um algoritmo do tipo A*(RUSSELL; NORVIG, 2003), este precisa de três funções básicas, uma para definir os estados iniciais, uma para definir o estado vizinho de um estado e uma para saber qual estado é final.

Para se definir a ordem dos quanta iniciais da busca de sucessores, chamados de estados iniciais, é utilizada uma heurística que tem como critério a frequência de ϕ nas cláusulas de Φ , sendo ordenados dos mais frequentes aos menos frequentes. Realizar essa ordenação é fácil pois consiste simplesmente em se ordenar a lista dos quanta pelo tamanho de F .

Para realizar a busca de vizinhos são utilizadas algumas estruturas adicionais. Uma delas é o *gap*, sendo este representado por G_ϕ , que é a lista de cláusulas que não possuem nenhum literal associado a um conjunto incompleto ϕ , logo $G_\phi = \Phi - \bigcup_{i=1}^k F_i$. Esta estrutura é utilizada pois todos os quanta associados a G_ϕ são possíveis sucessores.

Outra estrutura é a lista de quanta proibidos, representado por X_ϕ , que é a lista de *mirrors* dos quanta associados a ϕ . Esta estrutura é usada pois é necessário garantir que um quantum e seu *mirror* jamais serão atribuídos à mesma cláusula dual mínima.

A partir destas estruturas a busca consiste em reduzir o G_ϕ , garantido as propriedades da CNF. Com isso a busca é feita utilizando os critérios de qualidade \succ são apresentados a seguir.

Seja

$$F_i^G = F_i \cap G_\phi, \bar{F}_{i=1}^k = \bar{F}_i \cap G_\phi \text{ e } F_{ij} = F_i^G \cap F_j^G$$

Então seguem os critérios

- se $|F_i^G - F_{ij}| > |F_j^G - F_{ij}|$ então $\phi_i \succ \phi_j$ senão $\phi_j \succ \phi_i$
- se $|F_i^G - F_{ij}| = |F_j^G - F_{ij}|$ então, se $|\overline{F}_i^G - \overline{F}_{ij}| > |\overline{F}_j^G - \overline{F}_{ij}|$ então $\phi_i \succ \phi_j$ senão $\phi_j \succ \phi_i$

O algoritmo 4 é o algoritmo completo de busca de sucessores.

Algorithm 4 Algoritmo de busca de sucessores de um quantum

```

1: procedure SUCCESSORS( $F(\phi)$ )                                ▷ Recebe o quantum
2:    $\Omega \leftarrow \emptyset$                                        ▷ Inicializa a lista de sucessores
3:    $\Theta \leftarrow \{\phi^F \mid \phi \in C \text{ and } C \in G_\phi\} - X_\phi$   ▷ Encontra o conjunto de
   possíveis sucessões
4:    $\text{Ordene}(\Theta, \succ)$                                        ▷ Ordena  $\Theta$  de acordo com o critério  $\succ$ 
5:   if  $\exists C \in R_\phi, \Theta \cap C = \emptyset$  then
6:     return  $\emptyset$ 
7:   for  $\phi^F$  in  $\Theta$  do
8:      $\Phi^+ \leftarrow \Phi \cup \{\phi^F\}$ 
9:     if  $\forall \phi_i^F, F_i^* \not\subset F$  and  $\emptyset \notin R_{\phi^+}$  and  $\forall C \in R_{\phi^+}, C \not\subset X_\phi$  then
10:       $\Omega \leftarrow \Omega \cup \{\Phi^+\}$ 
11:  return  $\Omega$ 

```

Os estados finais ocorrem quando $G_\phi = \emptyset$ ou quando não existem mais sucessores possíveis.

2.2 COMPUTAÇÃO PARALELA

A computação paralela é a área da computação que estuda os processos que são executados simultaneamente. A computação paralela pode ser dividida em diversas subáreas, uma vez que existem diversas formas de se executar processos em paralelo, indo desde o paralelismo em nível de instrução(RAU; FISHER,) até *clusters* de diversos computadores independentes(COULOURIS et al., 2011).

Do ponto de vista da arquitetura de computação paralela existem diversas classificações. Uma delas é a taxonomia de Flynn(FLYNN, 1972), apresentada na tabela 2, que divide as arquiteturas paralelas em relação aos seus dados e processos, hoje grande parte dos sistemas são MIMD.

	Única Instrução	Múltipla Instrução
Dado Único	SISD	MISD
Múltiplos Dados	SIMD	MIMD

Tabela 2 – Taxonomia de Flynn

Outra forma de se pensar em arquiteturas paralelas é com relação a memória, sendo essa uma divisão clássica, dividindo-se as arquiteturas entre sistemas que utilizam memória compartilhada e sistemas que utilizam memória distribuída(PATTERSON; HENNESSY, 2012).

Existem diferentes tipos de paralelismo: no paralelismo em nível de instruções, um processador executa as instruções do programa através de um *pipeline*(PATTERSON; HENNESSY, 2012). No paralelismo em nível de dados, uma dada tarefa pode ser executada em paralelo dividindo-se os dados a serem processados em um nó diferente(HILLIS; STEELE JR., 1986). E em nível de tarefa, as diferentes partes de uma tarefa são distribuídas entre diferentes nós(QUINN, 2003).

Existem diversos paradigmas de programação paralela(KUMAR, 2002). Um deles é o mestre/escravo, no qual a um nó é atribuída a tarefa de mestre e este envia tarefas a serem realizadas pelos nós escravos. Outra é a programação único múltiplos dados, no qual o espaço de dados é dividido entre múltiplos nós e estes executam a mesma tarefa com a sua parte de dados. Outro paradigma parecido com este é o de *pipeline* de dados, em que o processamento dos dados é dividido em etapas e cada etapa é realizada em um nó diferente. Um paradigma importante é o dividir e conquistar, que consistem em recursivamente dividir a tarefa entre os nós.

Neste trabalho, tem-se como foco a subárea denominada computação distribuída, pois o trabalho trata de um algoritmo que funciona através da troca de mensagens e sem memória compartilhada(COULOURIS et al., 2011). Sendo assim o algoritmo desenvolvido pode funcionar tanto em um *cluster* de computadores quanto em um processador *multi-core*.

O *Track* de computação paralela da *SAT Competition* é feito em um processador de 12 núcleos com 36 GB de memória RAM.¹

¹<http://www.satcompetition.org/2014/>

2.2.1 Paralelismo no problema SAT

2.2.2 Algoritmos portfólio

Uma maneira simples de resolver SAT em paralelo é executar diversos *SAT solvers* em paralelo, esta técnica é conhecida como portfólio. Esta técnica funciona executando diversas versões do *SAT solver* ou diversos *SAT solvers* com diferentes características, sendo que em cada nó deve rodar um *solver* otimizado para um problema diferente assim obtendo uma performance média melhor. Esta técnica pode ser denominada como completa (MARQUES, 2013). Esse funcionamento é representado na figura 9.

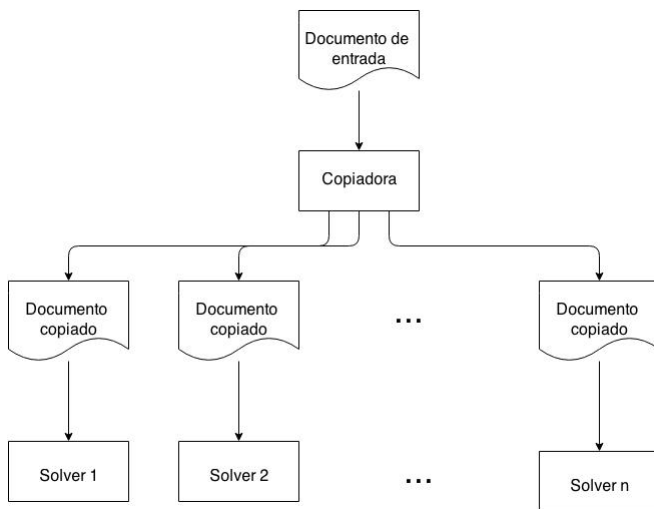


Figura 3 – Funcionamento de um algoritmo portfólio (NELSON, 2013)

Uma das vantagens desta técnica é sua fácil implementação, uma vez que pode ser realizada utilizando apenas um script que inicia os diversos *SAT solvers* nos nós do sistema. Outra vantagem desta técnica é a possibilidade de sua execução em qualquer sistema paralelo, uma vez que cada instância do problema a ser resolvido é completamente independente.

Para *solvers* paralelos portfólio não existem problemas de controle ou em sua arquitetura, porém eles não apresentam um grande ganho de desempenho. Existe um ganho médio quando, no portfólio existe um *solver* ou heurística que apresenta uma performance muito boa para o problema

a ser resolvido, porém isso pode não acontecer ou o ganho pode ser pequeno. Outra questão é que boa parte do poder computacional utilizado é simplesmente desperdiçado, assim apresentando uma má utilização dos recursos disponíveis. Por fim esta técnica não apresenta nenhum ganho para problemas que são insatisfazíveis, e mesmo neste cenário apresenta um custo muito maior.

Uma implementação popular da técnica portfólio para o problema SAT é o ManySAT(HAMADI; SAIS, 2009). Este *SAT solver* utiliza diferentes configurações aplicadas ao *solver* MiniSAT e ao preprocessador SatElite, com isso ele obtém um bom ganho de desempenho pois os algoritmos baseados em DPLL modernos são bastante sensíveis as diferentes configurações de suas otimizações. O ManySAT foi amplamente testado em usos industriais de *solver* e ficou em primeiro lugar do *track* de paralelo da SAT-Race de 2008. Este *solver* foi desenvolvido pela Microsoft e pelo CRIL-CNRS.

2.2.3 Particionamento do espaço de busca

Esta técnica consistem em dividir os espaço de busca do problema a ser explorado em espaços de busca menores a serem processados nos vários nós do sistema. A divisão da busca pode ser feita de diversa maneiras, sendo aplicada tanto em sistemas multi-core quanto sistemas conectados através de uma rede. Esta técnica também é bastante popular, sendo bastante utilizada na paralelização de algoritmos baseado tanto em DPLL quanto em CDCL. As diferentes formas de implementação desta técnica são apresentadas a seguir. Este trabalho também utiliza esta técnica, na seção de desenvolvimento serão apresentados detalhes de como esta técnica foi utilizada no algoritmo QuantumSAT.

O DPLL clássico pode ser paralelizado através da execução em paralelo de sua busca recursiva. Para se fazer isto todos os nós recebem Φ e então cada nó recebe uma fórmula simplificada diferente. Além das otimizações presentes na versão sequencial do DPLL podem ser feitas otimizações para a melhor execução em paralelo deste algoritmo, destaca-se a escolha da forma como serão distribuídas as fórmulas simplificadas.

Existem diversas implementações do DPLL em paralelo. A primeira delas foi o PSatz(JURKOWIAK; LI; UTARD, 2005), esta foi uma implementação simples da técnica que não utilizava nenhuma otimização além das apresentadas no DPLL clássico, descrito no algoritmo 1. Esta implementação foi feita originalmente utilizando processadores *single core* conectados em rede e posteriormente foi portada para processadores *multi core* com memória compartilhada(HÖLLDOBLER et al., 2011). Na comparação entre estas duas

versões foi visto que a versão para rede tem uma performance melhor que a para memória compartilhada, isto se dá pois na versão com memória compartilhada os processos entram em uma condição de corrida para obter acesso aos recursos o que causa um *overhead* maior que a comunicação por rede.

O CDCL é um algoritmo mais complexo de se paralelizar, pois com a adição da estrutura de cláusulas aprendidas a busca é feita de maneira pouco ordenada, o que dificulta o processo de divisão das tarefas. Boa parte das versões paralelas do CDCL são implementadas utilizando múltiplos processadores conectados em rede, dado que o acesso não uniforme aos dados causa um *overhead* muito grande quando se utiliza memória compartilhada.

A maioria dos algoritmos CDCL paralelos utiliza a técnica de mestre e escravo, onde o mestre possui as cláusulas aprendidas e passa as tarefas para os escravos. Para se otimizar este algoritmo utilizam-se algumas técnicas, como criação dinâmica de escravos e a utilização de *lookahead* para seleção de ramificação das cláusulas. Existem alguns algoritmos que dividem esta tarefa em nós iguais, normalmente estes compartilham todas as cláusulas aprendidas com todos os nós, isto causa um grande *overhead* de comunicação na rede.

O PMSat(GIL; FLORES; SILVEIRA,) é uma versão paralela do Mini-SAT(SORENSSON; EEN, 2002). Este algoritmo usa um método bastante comum em *SAT solvers* paralelos, que é o particionamento do espaço de busca, assim dividindo as cláusulas em diversos nós, neste *solver* o usuário pode configurar a heurística para divisão do espaço de busca. Além disto ele também utiliza heurísticas para cooperação entre nós através do compartilhamento das cláusulas aprendidas e dos conflitos. Este *solver* foi desenvolvido em C++ e MPI e sua arquitetura é apresentada na figura 4.

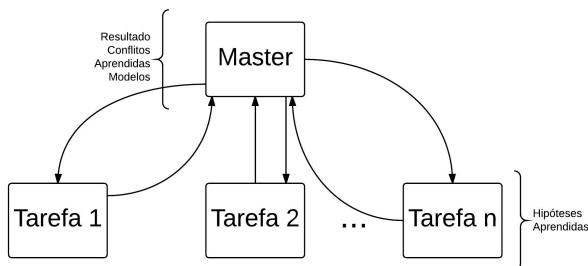


Figura 4 – Arquitetura do PMSat

Como pode-se ver nas implementações apresentadas esta técnica apresenta vários desafios. O balanço de carga é um problema complexo em toda

a área de computação paralela, neste contexto a solução esta na divisão do espaço de busca em um grande número de espaços, e dinamicamente se distribuir as tarefas a medida que os escravos ficam inativos. Outro problema existe em sistemas conectados através de uma rede, onde pode existir uma corrida pelos recursos da rede, este problema pode ser mitigado utilizando redes com uma maior banda. Em sistemas de memória compartilhada pode-se estourar a memória quando são criados muitos subespaços de busca, isto é mitigado adicionando mais memória ao sistema.

Este trabalho também utiliza esta técnica, na seção de desenvolvimento serão apresentados detalhes de como esta técnica foi utilizada no algoritmo QuantumSAT.

2.2.4 Particionamento do problema

Esta técnica consiste em dividir a fórmula proporcional a ser resolvida em fórmulas menores, realizar a busca da solução destas fórmulas em paralelo e combinar as soluções. Apesar da busca pela solução poder ser feita em paralelo existe a necessidade de comunicação entre os nós, pois existem variáveis compartilhadas entre as fórmulas.

O *sat solver* JACK-SAT (SINGER; MONNET, 2008) implementa esta técnica, sua implementação original divide a fórmula proporcional V em duas fórmulas $V1$ e $V2$, criando assim dois problemas $P1=(V1,C1)$ e $P2=(V2,C2)$ sendo $C1$ e $C2$ as variáveis que estão presente exclusivamente em $V1$ e $V2$ respectivamente, também é criado um conjunto $C3$, com as variáveis compartilhadas em as fórmulas. Cada problema é resolvido independentemente e é o após isso o algoritmo tenta combinar as soluções, procurando as soluções não conflitantes com as variáveis em $C3$. Existem diversas melhorias nesta implementação que não serão discutidas, pois a ideia central da implementação é a mesma. Outra melhoria possível seria a tentativa de se evitar conflitos com $C3$ durante a busca pelas soluções de $P1$ e $P2$.

Esta técnica não é muito popular devido aos seus problemas. Sendo um deles que a busca da solução dos subproblemas também apresenta complexidade exponencial. Outro é que a adição da etapa de junção das soluções cresce de forma exponencial com o número de soluções encontradas para os subproblemas. Além disto esta técnica apresenta uma escalabilidade muito baixa, o que não é desejado pois a tendencia é um aumento no número de nós dos sistemas.

3 DESENVOLVIMENTO

Nesta seção serão apresentados os diversos aspectos referentes ao desenvolvimento do presente trabalho. Esta seção primeiro apresenta o algoritmo desenvolvido. Após são descritas as tecnologias utilizadas durante o desenvolvimento do trabalho, seu funcionamento e a motivação de sua escolha. Além disso também será apresentado o ambiente e metodologia de testes.

3.1 TECNOLOGIAS UTILIZADAS

Nesta implementação escolheu-se como linguagem de programação o Erlang, esta decisão resulta em um grande impacto nos resultados do trabalho.

Erlang apresenta como característica maior a sua grande utilização em sistemas distribuídas, principalmente sistemas que lidam com um grande número de chamadas e necessita grande confiabilidade, porém também apresenta alguns usos tem cenários de computação científica (). Esta linguagem apresenta como principal vantagem um grande número de ferramenta para computação distribuída, sendo estas ferramentas bastante estáveis e eficientes. Outra vantagem é fato de Erlang ser uma linguagem funcional, este paradigma é bastante eficiente para computação distribuída pois faz com que o código apresente poucas condições de corrida e não utilize variáveis compartilhadas(). Erlang também permite facilmente se integrar com outras linguagens de programação mais eficientes em outras tarefas.

Também foi utilizada a tecnologia de *cloud computing*, neste trabalho foi utilizada a *cloud* da Amazon. Esta tecnologia permite a criação de virtualmente infinitas máquinas virtuais, sendo possível assim fazer testes sobre a escalabilidade do sistema. Esta tecnologia também permite a criação de máquinas virtuais com as características desejadas para a pesquisa, neste caso diversas máquinas com apenas um core.

3.2 ALGORITMO DE BUSCA

Como dito anteriormente este algoritmo realiza uma busca em um espaço de estados. Esta busca é também a parte mais longa do algoritmo, uma vez que é necessário que a mesma visite todos os estados possíveis do problema para que o mesmo seja provado insatisfazível ou para que sejam

encontradas todas as soluções.

Existem dois modos de realizar uma busca em um espaço de estados, em largura(BFS, sigla em inglês) ou em profundidade(DFS, sigla em inglês). A busca em largura funciona partindo de uma raiz e visitando todos os seus vértices. A busca em profundidade funciona partindo de uma raiz e explorando cada ramo do espaço até que seja encontrado um nó folha, quando este é encontrado é realizado o *backtracking*.

Ambas as buscas pode ser exaustivas, assim visitando todos os nós possíveis, ou parar quando a primeira solução é encontrada. Sendo assim a implementação pode resolver tanto o problema SAT quanto suas extensões, como o UNIQUE-SAT, onde o problema tem apenas uma solução. Para maioria dos casos é melhor parar na primeira solução encontrada, pois assim o algoritmo é mais rápido e raramente o tamanho da solução é importante.

Para a implementação deste trabalho foi utilizada a linguagem erlang, uma linguagem funcional, assim foi implementada uma versão recursiva do algoritmo de busca em profundidade. A versão clássica deste algoritmo funciona recebendo um grafo e um vértice a ser visitado. Marca-se o vértice como visitado e recursivamente chama-se a função para todos os vértices adjacentes ainda não marcados como visitados. Esta versão pode ser vista no pseudo-código a seguir:

Algorithm 5 Pseudo-código de um algoritmo funcional de busca em profundidade(DFS)

```

1: procedure DFS(G,v)
2:   marque v como visitado
3:   for todo vértice w em G.verticesAdjacentes(v) do
4:     if w não é marcado como visitado then
5:       DFS(G,w)

```

Porém no caso deste trabalho o grafo de estados não está pronto no momento em que o algoritmo é chamado, ele é construído no momento em que o estado é visitado em seus sucessores são calculados. Sendo assim foi necessário modificar o algoritmo clássico para o cenário do trabalho. Para isso foi modificada a entrada da função, nesta implementação a função recebe uma pilha de estados a visitar. A partir daí o algoritmo desempilha o primeiro estado, calcula os seus sucessores, empilha em ordem os estados sucessores e recursivamente chama a função até que a pilha esteja vazia. Esta implementação básica é apresentada no pseudo código a seguir:

Na versão implementada neste trabalho o estado é composto por um ϕ incompleto, que é a lista de quantum associados ao estado, um forbidden, que é a lista dos quanta que não podem mais ser associados ao estado, e o gap,

Algorithm 6 Pseudo-código de um algoritmo funcional DFS para grafo incompleto

```

1: procedure BUSCA(S)
2:   if S é vazio then
3:     return
4:   else
5:     Estado = desempilhe(S)
6:     Succ = encontreSucessores(Estado)
7:     empilhe(Succ, S)
8:     Busca(S)

```

que o conjunto de clausulas que não contém um literal no ϕ .

Outra característica da implementação deste trabalho é o cálculo dos sucessores, que é feito utilizando uma função de qualidade. Esta função seleciona o quantum que cobre o maior número de cláusulas, como critério de desempate é selecionado o quantum cujo *mirror* cobre o maior número de cláusulas. O estado cujo gap é vazio deve ser colocado como uma solução e não deve ser adicionado nos sucessores. Também deve-se considerar como final o estado que não reduz o tamanho gap, este não deve ser adicionado aos sucessores nem as soluções.

3.2.1 Busca Paralela

Uma busca em grafos pode ser realizada de duas formas, em profundidade e em largura. A busca em profundidade explora os nós do grafo seguindo através dos filhos dos nós. A busca em largura explora os nós do grafo seguindo através dos vizinhos dos nós.

A implementação sequencial deste algoritmo utilizou uma busca em largura, esta busca foi escolhida pois a utilização desta implementação necessitava da menor solução para o problema SAT, sendo assim necessitava fazer uma busca completa, com essa restrição o modo de busca era pouco importante.

Para o caso desta implementação desejamos apenas encontrar uma solução qualquer, o mais rápido possível, neste caso a forma da busca pode ter algum impacto. Para verificar isto decidi utilizar um método empírico, pois as duas implementações são facilmente realizadas e alguns testes rápidos mostrariam evidencias suficientes para que uma boa decisão fosse tomada. Realizei os testes utilizando algumas instâncias do problema SAT aleatórias pequenas.

A busca em largura apresentou algumas vantagens no balanceamento de carga, uma vez que foi possível cada nó do sistema realizar uma porção menor do trabalho, porém essa característica apresenta uma desvantagem ela necessita de muito mais comunicação entre os nós. Outra desvantagem está no fato de a forma com que a busca é realizada faz com que sejam explorados mais estados antes da primeira solução ser encontrada, na média das buscas.

A busca em profundidade apresentou as vantagens de possuir menos comunicação entre os nós e de encontrar a primeira solução explorando menos estados. Porém apresenta um balanço de carga entre os nós pior, esse cenário pode ser agravado se os nós apresentarem diferenças de performance.

3.3 EXEMPLO

Neste exemplo utilizaremos dois nós e a pré-execução em paralelo com apenas um nível na árvore. Este exemplo funciona para demonstrar a lógica de funcionamento paralelo do algoritmo, porém a implementação na prática funciona com parâmetros diferentes. O exemplo utiliza o seguinte ϕ

$$0 : (p_2 \vee \neg p_3 \vee p_4 \vee \neg p_1) \wedge$$

$$1 : (\neg p_2 \vee \neg p_1 \vee p_4 \vee p_3) \wedge$$

$$2 : (p_2 \vee p_3 \vee \neg p_4 \vee p_1) \wedge$$

$$3 : (p_1 \vee p_4 \vee p_3 \vee p_2) \wedge$$

$$4 : (p_2 \vee p_4 \vee \neg p_1 \vee p_3) \wedge$$

$$5 : (p_2 \vee \neg p_1 \vee \neg p_3 \vee \neg p_4) \wedge$$

$$6 : (\neg p_1 \vee \neg p_3 \vee p_4 \vee \neg p_2) \wedge$$

$$7 : (p_4 \vee \neg p_3 \vee p_1 \vee \neg p_2) \wedge$$

$$8 : (p_1 \vee \neg p_2 \vee \neg p_4 \vee p_3) \wedge$$

$$9 : (p_3 \vee \neg p_1 \vee \neg p_2 \vee \neg p_4)$$

Sendo assim o quanta gerado é o seguinte:

$$P_1^{\{3,4,8,9\}}, \neg P_1^{\{1,2,5,6,7,10\}}, P_2^{\{1,3,4,5,6\}}, \neg P_2^{\{2,7,8,9,10\}}, P_3^{\{2,3,4,5,9,10\}}, \neg P_3^{\{1,6,7,8\}}, \\ P_4^{\{1,2,4,5,7,8\}}, \neg P_4^{\{3,6,9,10\}}$$

A imagem a seguir descreve o funcionamento do algoritmo na situação citada anteriormente. A nó a@comp é o nó que realiza controle do sistema, ele responsável por manter os estados que ainda necessitam ser expandidos, as soluções dos que já foram expandidos e a ordem correta da comunicação,

fisicamente este nó pode mudar de localização em caso de falha, esta tarefa cabe aos supervisors. Os nós 1@comp e 2@comp são os trabalhadores eles recebem um estado a ser expandido, realiza sua expansão e retorna o resultado para o nó a@comp.

Neste caso o nó 1@comp pegará o estado com o phi: [-1] e o expandirá, assim retornando a solução phi: [-1,4,2,3], phi ..., o nó 2@comp pegará o estado com o phi: [4] e retornará a solução phi: [4,2,3], phi ..., como o nó 2@comp precisa visitar menos estados durante a sua expansão este acabará antes e então pegará o estado com phi: [-2] e assim por diante. Lembrando que este exemplo serve para exemplificar o funcionamento do algoritmo por isso está buscando todas as soluções, a execução real para assim que a primeira solução é encontrada e caso o nó não encontre solução ele simplesmente retorna .

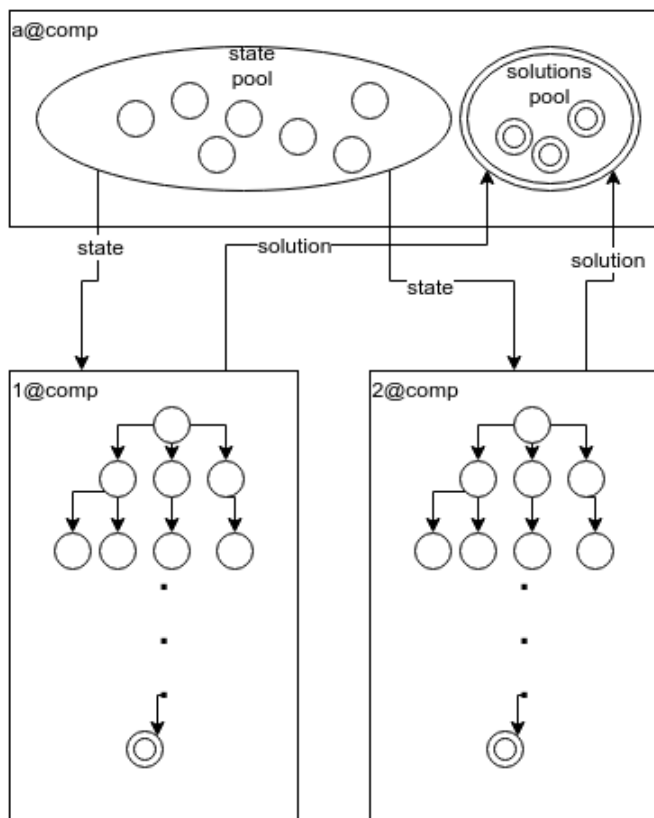


Figura 5 – Dinâmica dos nós

Para demonstrar o funcionamento paralelo do algoritmo também é importante conhecer o funcionamento do *state pool*, a figura a seguir apresenta os três primeiros estados da *state pool*.

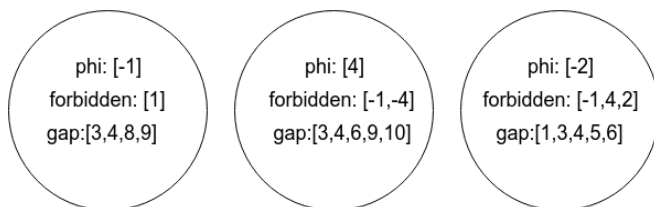


Figura 6 – Estados iniciais

4 RESULTADOS EXPERIMENTAIS E ANÁLISES

4.1 CENÁRIO

Em termos de hardware foram utilizados dois modelos de hardware para os testes, um cenário distribuído de máquinas virtuais rodando em uma *cloud* o outros sendo uma máquina multi-core, para este trabalho o primeiro cenário é mais importante pois a implementação realizada se adequa mais a este cenário, a máquina multi-core foi utilizada para poder realizar comparação com outros *SAT solvers*, melhorias possíveis para o cenário multi-core são discutidas na conclusão.

Para o cenário distribuído foi utilizada a *cloud* da Amazon, a plataforma completa é comercialmente conhecida como AWS, neste trabalho foi utilizado serviço EC2 que permite criar servidores virtuais. Esta plataforma apresenta a vantagem ser possível criar diversas máquinas virtuais conectadas para se realizar o teste em um cenário com memória distribuída e com nós se comunicando através de mensagens, sua principal desvantagem consiste no fato de serem máquinas virtuais rodando em ambiente desconhecido o que implica em menos controle do cenário, por exemplo quais outros processos estão rodando na máquina física ou qual rede está conectando as máquinas. O ambiente ideal seria um cluster onde se fossem conhecidas todas as características, porém não conseguimos acesso a um cluster como esse, sendo assim a cloud se mostrou uma boa alternativa. Para estes teste foram criadas máquinas virtuais single core, sendo criados os nós para execução da busca e um nó para integração e distribuição de tarefas.

Para o cenário multi-core foram utilizadas duas máquinas. A primeira é possui um core i7-4790 com 16gb de memória ram, utilizando o sistema operacional Linux(distribuição Ubuntu). O segundo é a máquina da UFSC, não consegui acessar ela hj. Estas máquinas forma utilizadas para testar uma comparação entre o *SAT solver* implementado e o clucose, além disso também foram utilizadas para testes de decisão do modelo de busca a ser utilizado. A principal vantagem de se usar máquinas multi-core está no fato de ser fácil de executar os SAT solvers. Ambas as máquinas rodaram os testes sendo acessas por SSH e rodando a versão mais leve do sistema operacional, assim reduzindo o impacto da execução do sistema nos testes.

Para facilitar, padronizar e ter uma execução mais rápida dos testes utilizei os testes disponíveis na SATLIB(<http://www.cs.ubc.ca/hoos/SATLIB/-benchm.html>), utilizando os testes 3-SAT randômicos uniformes indo de 20 variáveis a 250 variáveis e indo de 91 clausulas a 1065 clausulas, sendo que

utilizei a média de 100 instancias. Também utilizei alguns testes utilizados na SAT competition para um maior volume de variáveis e cláusulas, porém como utilizei menos instancias neste caso o seu valor estatístico comprometido.

4.2 RESULTADOS E ANÁLISE DE TEMPO

Os teste de tempo foram realizados em ambos os cenários, no cenário de computação podemos ter uma melhor ideia do comportamento do algoritmo no seu cenário ideal. Na máquina multi-core podemos realizar uma comparação com um outro *SAT solver*, neste caso o *SAT solver* escolhido foi o Glucose pois este é uma versão paralela de um algoritmo CDCL, apresenta um bom desempenho e é de fácil utilização.

No teste realizado na máquina *multi-core* a implementação realizada neste trabalho apresentou um desempenho significativamente pior que o *SAT solver* escolhido para comparação. Muitos motivos podem ser apresentados para o desempenho inferior da implementação deste trabalho, primeiramente o Glucose é um *SAT solver* bastante consolidado e otimizado, sendo desenvolvido desde 2009. Outro motivo está na linguagem utilizada, neste trabalho utilizamos Erlang enquanto o Glucose é implementado em C++, C++ por ser uma linguagem de mais baixo nível apresenta um desempenho melhor do que Erlang, porém Erlang apresenta ferramenta que facilitam a implementação distribuída e ferramentas de comunicação bastante eficientes. Os resultados comparativos podem ser vistos na tabela e gráfico abaixo.

Quando os testes são executados em uma máquina distribuída podemos ver uma clara melhoria na implementação desenvolvida, isso se deve a implementação realizada ter esse cenário em mente. Apesar disso se comparamos aos tempos do Glucose com mesmo número de nós ele ainda apresenta um resultado melhor, porém essa comparação é bastante imprecisa uma vez que os cenários são completamente diferentes, esse resultado também se deve aos fatores apresentados anteriormente. Apesar disso para uma primeira implementação os resultados são bastante satisfatórios. Os resultados são apresentados na tabela e no gráfico abaixo.

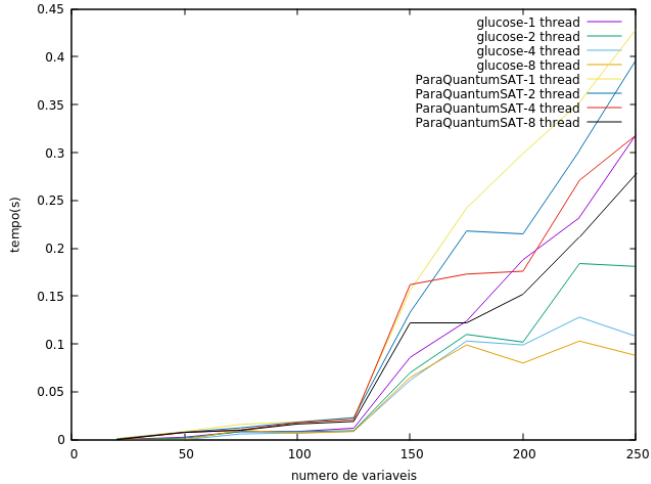


Figura 7 – Comparação com o glucose

4.3 RESULTADOS E ANÁLISE DE ESCALABILIDADE

Sendo esta característica bastante importante em sistemas distribuídos este resultado é bastante importante. Esta implementação apresentou um bom resultado com relação a escalabilidade, sendo testado com um grande volume de nós e escalando bem com a adição de nós. Também é claro que com o aumento do tamanho dos problemas a eficiência da execução em paralelo aumenta, isso se deve a uma diminuição relativa do *overhead* causado pela comunicação e outras características paralelas em relação ao tempo de execução. Os resultados são apresentados na tabela e no gráfico abaixo.

Tabela 3 – Resultados para Glucose e ParaQuantumSAT

Glucose				
Problema	1 thread	2 thread	4 thread	8 thread
uf20	0.0000	0.000	0.000	0.000
uf50	0.0026	0.0016	0.000	0.000
uf75	0.0084	0.0084	0.0060	0.0098
uf100	0.0085	0.0082	0.0070	0.0068
uf125	0.012	0.0094	0.0089	0.0086
uf150	0.086	0.070	0.062	0.065
uf175	0.124	0.110	0.103	0.099
uf200	0.188	0.102	0.099	0.080
uf225	0.232	0.184	0.128	0.103
uf250	0.318	0.181	0.108	0.088
ParaQuantumSAT				
Problema	1 thread	2 thread	4 thread	8 thread
uf20	0.0012	0.0008	0.000	0.000
uf50	0.0086	0.0078	0.0076	0.0076
uf75	0.016	0.0124	0.0103	0.0098
uf100	0.0186	0.0182	0.0176	0.0163
uf125	0.023	0.023	0.021	0.019
uf150	0.106	0.133	0.162	0.122
uf175	0.242	0.218	0.173	0.122
uf200	0.299	0.215	0.176	0.152
uf225	0.352	0.302	0.271	0.212
uf250	0.428	0.396	0.318	0.278

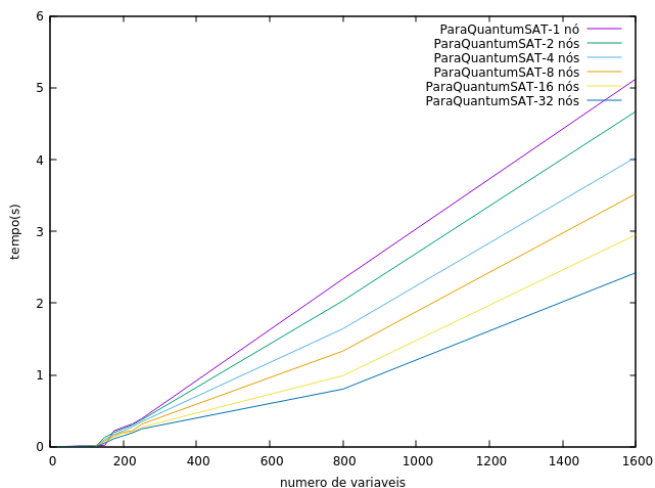


Figura 8 – Teste de escalabilidade

Tabela 4 – Tabele de resultados de escalabilidade

Problema	ParaQuantumSAT					
	1 nós	2 nós	4 nós	8 nós	16 nós	32 nós
uf20	0.000	0.000	0.000	0.000	0.000	0.000
uf50	0.0042	0.0016	0.000	0.000	0.000	0.000
uf75	0.0086	0.0064	0.0044	0.000	0.000	0.000
uf100	0.0122	0.0103	0.0079	0.0066	0.000	0.000
uf125	0.0134	0.0098	0.0089	0.0081	0.0069	0.0054
uf150	0.0196	0.133	0.102	0.082	0.061	0.052
uf175	0.222	0.198	0.171	0.152	0.141	0.112
uf200	0.269	0.252	0.221	0.202	0.186	0.151
uf225	0.317	0.299	0.275	0.222	0.201	0.192
uf250	0.389	0.373	0.348	0.313	0.274	0.248
SATComp-800	2.337	2.033	1.643	1.332	0.989	0.802
SATComp-1600	5.123	4.671	4.032	3.524	2.952	2.424

Analisando os dados apresentados podemos ver que na maioria dos testes com 32 nós a performance é em torno de 50% maior que com apenas um nó, um resultado bastante positivo se tratando de um problema NP. Também podemos ver que com o dobro de nós temos um ganho de desempenho de em média 16%, também bastante positivo dado a classe do problema.

4.4 RESULTADOS E ANÁLISE DE USO DOS NÓS

Este teste é bastante interessante para se analisar a distribuição de carga do sistema, uma vez que é desejável a maior utilização possível da capacidade computacional do sistema. Isto é desejável pois além de evitar nós em *idle* desperdiçando o seu poder computacional e piorando o desempenho do sistema, estes nós em *idle* também estão desperdiçando energia e consumindo recursos financeiros. O gráfico abaixo mostra o percentual de tempo em que nós passou trabalhando na busca, para verificar isso foi considerado que o nó estava trabalhando na busca quando o CPU do mesmo está com uma taxa de uso de mais de 90%. Para facilitar a visualização são mostrados apenas os nós pares além do nó 1.

5 TRABALHOS FUTUROS E CONCLUSÃO

5.1 TRABALHOS FUTUROS

Este trabalho apresenta o desenvolvimento de uma primeira versão paralela do algoritmo *SAT solver* ParaQuantumSAT, então é passível de várias melhorias.

Como foi mencionado anteriormente o trabalho foi completamente implementado em Erlang, isso implica numa boa eficiência em respeito a comunicação, porém apresenta um baixo desempenho quando se trata de tarefas computacionalmente pesadas. Isso pode ser melhorado utilizando Erlang para a comunicação paralela e utilizar C para a realização da busca na árvore de estados.

Outra melhoria possível está no estudo de melhores heurísticas para realizar a busca de sucessores na árvore, isso em si melhoraria o desempenho do algoritmo, porém quando se trata do cenário paralelo o ganho poderia ser ainda maior, pois neste cenário é possível utilizar múltiplas heurísticas ao mesmo tempo.

5.2 CONCLUSÃO

Neste trabalho foi apresentado um algoritmo *SAT solver* paralelo em sua primeira implementação, este algoritmo não apresentou o melhor desempenho bruto possível, sendo claramente pior que outros SAT solver disponíveis. Porém apresentou algumas características positivas, apresentando uma boa escalabilidade, algo altamente desejado em sistemas distribuídos uma vez que para uma melhoria de desempenho basta adicionar mais nós ao sistema. Outra característica positiva do algoritmo implementado esta no bom uso da capacidade dos nós e no bom balanceamento de carga do sistema. Uma inovação do trabalho também foi o uso da linguagem Erlang, pouco utilizada nestes cenários porém bastante interessante de se pesquisar, pois apresenta um bom desempenho para a computação distribuída e pode ser utilizada em conjunto com outras linguagens mais eficientes na parte de computação pesada.

REFERÊNCIAS

BIERE, A. et al. **Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications**. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009. ISBN 1586039296, 9781586039295.

BITTENCOURT, G.; MARCHI, J.; PADILHA, R. S. A syntactic approach to satisfaction. In: **4th Inter. Workshop on the Implementation of Logic (LPAR03)**. [S.l.: s.n.], 2003. p. 18–32.

BUSEMANN, C. et al. Enabling the usage of sensor networks with service-oriented architectures. In: **Proceedings of the 7th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks**. New York, NY, USA: ACM, 2012. (MidSens '12), p. 1:1–1:6. ISBN 978-1-4503-1610-1. Disponível em: <http://doi.acm.org/10.1145/2405167.2405168>.

CLARKE, E. et al. Bounded model checking using satisfiability solving. **Form. Methods Syst. Des.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 19, n. 1, p. 7–34, jul. 2001. ISSN 0925-9856. Disponível em: <http://dx.doi.org/10.1023/A:1011276507260>.

COULOURIS, G. et al. **Distributed Systems: Concepts and Design**. 5th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011, 9780132143011.

DAVIS, M.; LOGEMANN, G.; LOVELAND, D. A machine program for theorem-proving. **Commun. ACM**, ACM, New York, NY, USA, v. 5, n. 7, p. 394–397, jul. 1962. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/368273.368557>.

EEN, N.; SÖRENSSON, N. **MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition**. 2005.

EEN, N.; SÖRENSSON, N. **An Extensible SAT-solver [ver 1.2]**. 2003.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 21, n. 9, p. 948–960, set. 1972. ISSN 0018-9340. Disponível em: <http://dx.doi.org/10.1109/TC.1972.5009071>.

GIL, L.; FLORES, P.; SILVEIRA, L. M. Pmsat: a parallel version of minisat. **Journal on Satisfiability, Boolean Modeling and Computation**, p. 2008.

HAMADI, Y.; SAIS, L. Manysat: a parallel sat solver. **JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)**, v. 6, 2009.

HILLIS, W. D.; STEELE JR., G. L. Data parallel algorithms. **Commun. ACM**, ACM, New York, NY, USA, v. 29, n. 12, p. 1170–1183, dez. 1986. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/7902.7903>>.

HÖLLDOBLER, S. et al. A short overview on modern parallel sat-solvers. **Proceedings of the International Conference on Advanced Computer Science and Information Systems**, p. 201–206, 2011.

HOOS, H.; STTZLE, T. **Stochastic Local Search: Foundations & Applications**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558608729.

JURKOWIAK, B.; LI, C. M.; UTARD, G. A parallelization scheme based on work stealing for a class of sat solvers. **J. Autom. Reason.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 34, n. 1, p. 73–101, jan. 2005. ISSN 0168-7433. Disponível em: <<http://dx.doi.org/10.1007/s10817-005-1970-7>>.

KATAJAINEN, J.; MADSEN, J. N. Performance tuning an algorithm for compressing relational tables. In: **Proceedings of the 8th Scandinavian Workshop on Algorithm Theory**. London, UK, UK: Springer-Verlag, 2002. (SWAT '02), p. 398–407. ISBN 3-540-43866-1. Disponível em: <<http://dl.acm.org/citation.cfm?id=645901.672631>>.

KAUTZ, H.; SELMAN, B. Planning as satisfiability. In: **Proceedings of the 10th European Conference on Artificial Intelligence**. New York, NY, USA: John Wiley & Sons, Inc., 1992. (ECAI '92), p. 359–363. ISBN 0-471-93608-1. Disponível em: <<http://dl.acm.org/citation.cfm?id=145448.146725>>.

KAUTZ, H.; SELMAN, B. Pushing the envelope: Planning, propositional logic, and stochastic search. In: **Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2**. AAAI Press, 1996. (AAAI'96), p. 1194–1201. ISBN 0-262-51091-X. Disponível em: <<http://dl.acm.org/citation.cfm?id=1864519.1864564>>.

KUMAR, V. **Introduction to Parallel Computing**. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201648652.

LAVAGNO, L.; MARTIN, G.; SCHEFFER, L. **Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set**. Boca Raton, FL, USA: CRC Press, Inc., 2006. ISBN 0849330963.

MARQUES, R. S. **Parallel SAT Solver**. Dissertação (Master's Thesis) — Instituto Superior Técnico, Universidade Técnica de Lisboa, dez. 2013.

MILTERSEN, P. B.; RADHAKRISHNAN, J.; WEGENER, I. On converting cnf to dnf. **Theor. Comput. Sci.**, Elsevier Science Publishers Ltd., Essex, UK, v. 347, n. 1-2, p. 325–335, nov. 2005. ISSN 0304-3975. Disponível em: <<http://dx.doi.org/10.1016/j.tcs.2005.07.029>>.

MILTERSEN, P. B.; RADHAKRISHNAN, J.; WEGENER, I. On converting cnf to dnf. **Theor. Comput. Sci.**, Elsevier Science Publishers Ltd., Essex, UK, v. 347, n. 1-2, p. 325–335, nov. 2005. ISSN 0304-3975. Disponível em: <<http://dx.doi.org/10.1016/j.tcs.2005.07.029>>.

NELSON, M. **A new approach to parallel SAT solvers**. Tese (Doutorado) — Massachusetts Institute of Technology, 2013.

OUYANG, M. **How Good Are Branching Rules in DPLL?** [S.l.], 1996.

PATTERSON, D.; HENNESSY, J. **Computer Organization and Design: The Hardware/software Interface**. Morgan Kaufmann, 2012. (Morgan Kaufmann Series in Computer Graphics). ISBN 9780123747501. Disponível em: <<http://books.google.com.br/books?id=DMxe9AI4-9gC>>.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Education Group, 2003. ISBN 0071232656.

RAU, B. R.; FISHER, J. A. Instruction-level parallelism. In: **Encyclopedia of Computer Science**. Chichester, UK: John Wiley and Sons Ltd. p. 883–887. ISBN 0-470-86412-5. Disponível em: <<http://dl.acm.org/citation.cfm?id=1074100.1074489>>.

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 2. ed. [S.l.]: Pearson Education, 2003. ISBN 0137903952.

SELMAN, B.; KAUTZ, H.; COHEN, B. Local search strategies for satisfiability testing. In: **DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE**. [S.l.: s.n.], 1995. p. 521–532.

SILVA, J. a. P. M.; SAKALLAH, K. A. Grasp—a new search algorithm for satisfiability. In: **Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design**. Washington, DC, USA: IEEE Computer Society, 1996. (ICCAD '96), p. 220–227. ISBN 0-8186-7597-7. Disponível em:
<<http://dl.acm.org/citation.cfm?id=244522.244560>>.

SINGER, D.; MONNET, A. Jack-sat: A new parallel scheme to solve the satisfiability problem (sat) based on join-and-check. In: _____. **Parallel Processing and Applied Mathematics: 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007 Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 249–258. ISBN 978-3-540-68111-3.

SKIENA, S. S. **The Algorithm Design Manual**. 2nd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2008. ISBN 1848000693, 9781848000698.

SORENSSON, N.; EEN, N. **MiniSat v1.13**. [S.l.], 2002.

APÊNDICE A – Exemplificando um Apêndice

Texto do Apêndice aqui.

ANEXO A – Exemplificando um Anexo

Texto do anexo aqui.