João Guilherme Zeni

PARAQUANTUMSAT: UM ALGORITMO *SAT SOLVER* DISTRIBUÍDO

Monografia submetido ao Curso de Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dra. Jerusa Marchi
Coorientador: 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

Dedico este trabalho a aqueles queridos a mim.

AGRADECIMENTOS

Gostaria de agradecer aos meus pais, por desde de minha infância terem me incentivado a estudar e me mostrado a importância do conhecimento, além disso me deram completo apoio durante todo o periodo de faculdade. Também agradeço a meus colegas de classe pelas horas de estudo e pela ajuda em todas as matérias do curso ao longo dos anos de faculdade. Gostaria de agradeçer aos meus colegas de labóratório pelas discuções e pela ajuda em desenvolver o conhecimento ao longo do curso. Gostaria de agradeçer especialmente aos colegas do IATE pela ajuda neste trabalho, tanto na parte de computação quanto no desenvolvimento do texto. Gostaria de agradeçer ao prof. Marcio Castro e a Setic por me disponibilizarem uma máquina para realizar os testes do trabalho. Por fim gostaria de agradeçer aos meus orientadores, profa. Jerusa Marchi e prof. Mario Dantas, pelo apoio no desenvolvimento deste trabalho e os valiosos conselhos para o trabalho.

Quem luta com monstros deve velar por que, ao fazê-lo, não se transforme também em monstro. E se tu olhares, durante muito tempo, para um abismo, o abismo também olha para dentro de ti.

(Friedrich Nietzsche, 1886)

RESUMO

O problema SAT é um problema clássico e bem conhecido, sendo o primeiro provado ser NP-completo . Por sua importância, já que diversos problemas igualmente NP-completo podem ser solucionados via redução ao problema SAT, o estudo de técnicas e algoritmos para a sua solução é sempre um tema atual de pesquisa. Uma forma de buscar acelerar a obtenção da solução é através de paralelismo. Computação paralela é uma área de estudo crítica, visto que hoje a capacidade de processamento dos computadores cresce com a paralelização dos processadores, sendo um dos principais desafios para a área 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

The SAT problem is a classic and well known problem, been the first proved NP-complete. For its importance, since many equally NP-complete problems can be solved through reduction to the SAT problem, the study of techniques and algorithms for its solution is always a current issue of research. One way of speed up the achievement of a solution is through parallelism. Parallel computing is a critical study field, since the current speed of processing of computers grows with the parallelisation of processors, been one of the main challenges of the field the building of efficient parallel algorithms to solve classical computing problems. This work intends to develop a parallel SAT solver algorithm, using as base a sequential SAT solver algorithm.

Keywords: Keyword 1.Parallel Algorithms Keyword 2.SAT solver

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

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)*, onde diversos dispositivos inteligentes 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).

Dentre os algoritmos paralelos desenvolvidos e estudados encontram-se algoritmos para a solução do problema de *satisfazibilidade booleana* (SAT). O problema 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, como por exemplo a estrutura mantida para cor-

relacionar literais e cláusulas, além disso também escolhemos uma linguagem de programação pouco popular em *SAT solvers*, que igualmente mostrou-se vantajosa devido a facilidade e bom desempenho das ferramentas da mesma para a comunicação dos nós distribuídos.

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.

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 grande 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 divido em outros seis capítulos, conforme segue.

No capítulo dois é a fundamentação teórica do trabalha, neste capítulo são apresentados os conceitos necessários para a comprensão do trabalho desenvolvido, este capítulo apresenta uma introdução a lógica proposicional, a definisação problema SAT, os diversos algoritmos *SAT solver* e por fim o algoritmo base utilizado no trabalho.

No capítulo três é apresentada um introdução a computação paralela, em seguida é apresenta a interseção entre o problema SAT e computação paralela e por são apresentadas as diversas técnicas de paralelização do problema SAT.

No capítulo quatro 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.

No capítulo cinco é apresentado o ambiente de testes e os resultados obtido com relação a tempo, escalabilidade e eficiência.

No capítulo seis é apresentada uma conclusão e os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são descritos os fundamentos necessários para a comprensão do trabalho desenvolvido. Inicialmente são apresentados os conceitos de lógica proposicional necessários para a compressão do problema SAT. Utilizando os conceitos apresentados o problema SAT é formalmente definido. A seguir é apresentada a importância do problema SAT. Na seção seguinte são apresentados os diversos SAT *solvers* sequenciais. Por último é apresentado o algoritmo base utilizado neste trabalho.

2.1 O PROBLEMA SAT

O problema SAT consiste em saber se há alguma atribuição de valores verdade que satisfaz uma fórmula lógica proposicional descrita em CNF. Iniciamos este capítulo apresentando um pouco sobre a linguagem lógica proposicional, formas normais canônicas e finalmente sobre 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 têm sido amplamente aplicados em diversas áreas de ciência da computação como gramaticas formais, teoria da computação, algoritmos entre outros.

Uma proposição, que consiste de conjunto de variáveis ou literais, que possuem valores verdade, sendo os valores Verdade ou Falso, e pode ser formada de outras proposições com o uso de conectores lógicos, a um proposição pode ser atribuído um sentido, chamado de asserção. Conectores lógicos compões proposições através da ligação de variáveis e literais, 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 descrita 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.

Fórmulas proposicionais são escritas utilizando operadores lógicos, que são símbolo utilizado 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. Como dito anteriormente para este trabalho notam-se os operadores *AND* representado por $\wedge$, *OR* representado por $\vee$ e *NOT* representado por $\neg$.

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 representa 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.

$$\Phi = C_1 \wedge C_2 \wedge C_3 \ldots \wedge C_k \mid \forall C, C_n = L_1 \vee L_2 \vee L_3 \ldots \vee L_k \tag{2.1}$$

As fórmulas proposicionais em DNF são representadas como uma disjunção de cláusulas, sendo cada cláusula um 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 SAT, portanto uma tarefa NP-Completa. Para realizar esta conversão existem diversas técnicas, neste trabalho será apresentada em detalhes uma delas.

$$\Phi = D_1 \vee D_2 \vee D_3 \ldots \vee D_k \mid \forall D, D_n = L_1 \wedge L_2 \wedge L_3 \ldots \wedge L_k \tag{2.2}$$

Estes conceitos são importantes pois o problema SAT é amplamente

relacionada com uma sub-classe de fórmulas proposicionais.

2.1.2 Definição do Problema

O problema SAT é definido sobre uma linguagem de lógica proposicional $\mathscr{L}(P)$, onde P = $p_1, p_2, \ldots, p_n$ sendo que todo p é um símbolo proposicional sem valor verdade definido. Uma fórmula $\Phi \in \mathscr{L}(P)$ pode ser escrita em CNF, logo $\Phi = \{C_1 \wedge C_2 \ldots \wedge C_k | \forall C, C_n = L_i \vee \neg L_j \ldots \vee L_z\}$, onde $L_i = p \vee \neg p$.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:

$$0 : (\neg p_4 \vee p_2 \vee \neg p_3) \wedge$$
$$1 : (\neg p_3 \vee p_2 \vee \neg p_1) \wedge$$
$$2 : (p_5 \vee \neg p_2 \vee \neg p_3) \wedge$$
$$3 : (\neg p_2 \vee p_5 \vee \neg p_1) \wedge$$
$$4 : (\neg p_3 \vee p_5 \vee p_1) \wedge$$
$$5 : (p_4 \vee p_5 \vee p_2) \wedge$$
$$6 : (\neg p_5 \vee p_3 \vee \neg p_2) \wedge$$
$$7 : (\neg p_1 \vee p_4 \vee p_3) \wedge$$
$$8 : (\neg p_3 \vee \neg p_2 \vee \neg p_1) \wedge$$
$$9 : (\neg p_3 \vee \neg p_4 \vee \neg p_1)$$

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 diz-se que Φ é satisfazível.

Outra característica do problema SAT é o número de símbolos em cada cláusula, o problema usualmente é chamado de kSAT 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 escrita 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* consiste 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 importante aplicação da solução SAT 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 dai 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 exitem muitos outros *solver* baseados nesse método, em (MILTERSEN; RADHAKRISHNAN; WEGENER, 2005) (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 aleatoriamente baixar o tempo médio de execução (SILVA; SAKALLAH, 1996). O funcionamento básico do CDCL é mostrado na Figura 2.

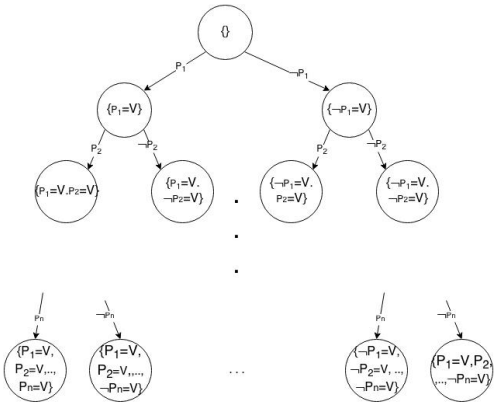

Figura 1 – Arvore de decisão do DPLLi (KATAJAINEN; MADSEN, 2002)

O Algoritmo 1 é o pseudo código de uma versão básica do DPLL. Esta

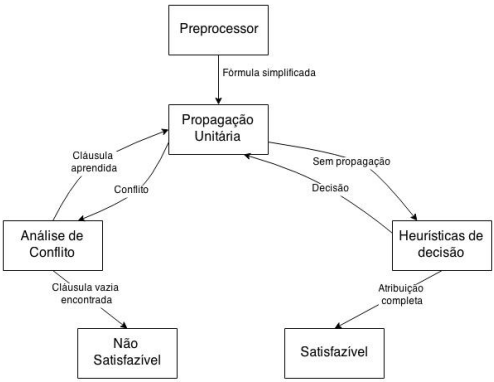

Figura 2 – Diagrama que apresenta os procedimentos e componentes do CDCL (DAVIS; LOGEMANN; LOVELAND, 1962)

versão é bastante simples. O primeiro passo é eliminar as cláusulas unitárias, aquelas que contém apenas um literal, e as cláusulas que contém o literal da cláusula unitária, pois para a fórmula ser satisfazível este literal precisa ser Verdade. O segundo passo é eliminar as cláusulas que contém 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.

Algorithm 1 Pseudo código DPLL

```
1: procedure DPLL(Φ)
2:     for Cláusula unitária l em Φ do
3:         Φ ← atribui(l, Φ)
4:     for Literal puro l em Φ do
5:         Φ ← atribui(l, Φ)
6:     l ← escolhaDeLiteral(Φ)
7:     return DPLL(Φ ∧ l) ∨ DPLL(Φ ∧ ¬l)
```

Uma implementação do algoritmo CDCL é o *solver* MiniSAT(EEN; SÖRENSSON, 2003). Este *solver* implementa várias otimizações que resultam em melhorias significativas em casos médios. Vale destacar a técnica de minimização de cláusulas conflitantes, que marca as cláusulas conflitantes para não visitalas, 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 algoritmo 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(Φ)
2:     for i := 1 a nTentativas do
3:         Φ ← valoresVerdadeAleatórios(Φ)        ▷ atribui valores verdade aleatórios aos literais
4:         for i := 1 a nMudanças do
5:             if Φ é satisfeito then
6:                 return Φ
7:             Φ ← mudancaDeLiteral(Φ)        ▷ 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 está 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 es-

pecíficos. 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, 2005). Na próxima seção é apresentado o algoritmo base para este trabalho, que efetua esta conversão escrevendo a fórmula proposicional na forma DNF utilizado 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* chamado 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 $\overline{\phi^F}$

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(Φ)        ▷ Recebe uma fórmula no formato CNF
2:     for cada φ em Φ do
3:         F(φ) ← Ø                 ▷ Inicializa a lista de quanta
4:     for cada C em Φ do
5:         for cada φ em C do
6:             F(φ) ← 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

$$F_i^G = F_i \bigcap G_\phi, \overline{F}_{i=1}^k = \overline{F}_i \bigcap G_\phi \; eF_{ij} = F_i^G \bigcap F_j^G$$

.

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 $|\overline{F}_i^G - \overline{F}_i j| > |\overline{F}_j^G - \overline{F}_{ij}|$ 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: $\Theta \leftarrow \{\phi^F \mid \phi \in C \; and \; C \in G_\phi\} - X_\phi$ ▷ Encontra o conjunto de possíveis sucessões
4: $Ordene(\Theta, \succ)$ ▷ Ordena Θ 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: $\Phi^+ \leftarrow \Phi \bigcup \{\phi^F\}$
9: **if** $\forall \phi_i^{F_i}, F_i^* \not\subset F$ and $\emptyset \notin R_{\phi+}$ and $\forall C \in R_{\phi+}, C \not\subset X_\phi$ **then**
10: $\Omega \leftarrow \Omega \bigcup \{\Phi^+\}$
11: **return** Ω

Onde R_ϕ representa o conjunto de literais proibidos, sendo que o literal escolhido estiver presente em R_ϕ ele é contraditório, portanto não sendo viável para expansão. F_i^* representa o conjunto de cláusulas exclusivamente cobertas por um dado quantumm.

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

3 COMPUTAÇÃO PARALELA

Este capitulo apresenta a área de computação paralela e suas relações com o problema SAT. A Primeira seção apresenta uma introdução aos fundamentos de computação e seu impacto na implementação de algoritmos. A segunda apresenta as diversas técnicas para se resolver o problema SAT em paralelo.

3.1 FUNDAMENTOS DE COMPUTAÇÃO PARALELA

A computação paralela é a área da computação que estuda os processos que são executados simultaneamente. A computação paralela pode ser divida em diversas subáreas, uma vez que existem diversas formas de se executar processos em paralelo, como por exemplo o paralelismo em nível de instrução (RAU; FISHER,), processadores multi-core, máquinas distribuídas, clusters de computadores (COULOURIS et al., 2011) e clouds elásticas.

Como modelo podemos analizar a área 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).

Também podemos analizar do ponto de vista do tipo 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).

Além disso outro aspecto importante são os 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 programa 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 consiste 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). Também foi utilizado o paradigma de mestre/escravo. 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]

3.2 PARALELISMO NO PROBLEMA SAT

Nesta seção serão apresentadas as diversas técnicas para resolver o problema SAT de forma paralela. Primeiramente é apresenta a técnica portfólio, que consiste em utilizar múltiplos *SAT solver* em paralelo. A seguir é apresentado o particionamento do espaço de busca, que consiste em realizar a busca de soluções em paralelo. Por fim é apresentado a técnica de particionamento do problema, que consiste em dividir a cláusulas do problema e realizar a busca para cada parte do problema em paralelo.

3.2.1 Algoritmos portfólio

Uma maneira simples de resolver SAT em paralelo é executar diversos *SAT solvers* em paralelo, esta técnica é conhecida com 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 completiva (MAR-

[1] http://www.satcompetition.org/2014/

QUES, 2013). Esse funcionamento é representando na Figura 10.

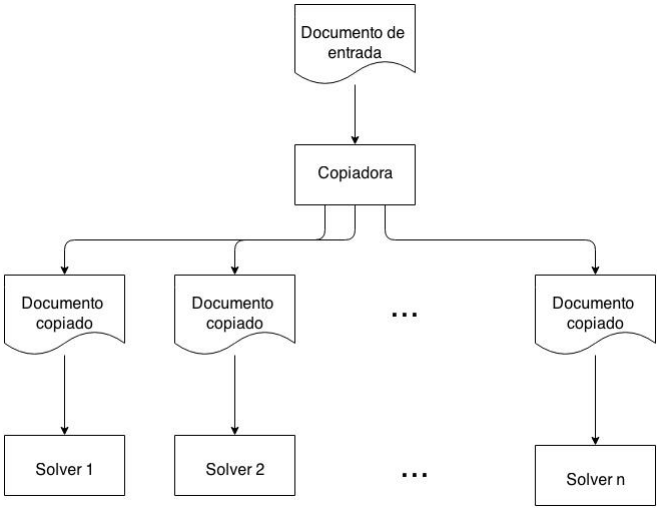

Figura 3 – Funcionamento de um algoritmo portfólio (NELSON, 2013)

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 às 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.

3.2.2 Particionamento do espaço de busca

Esta técnica consiste em dividir o 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 distribuídos . 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 implemen tação 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 mes-

tre 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, 2008) e o Glucose (AUDEMARD; SIMON, 2012) são versões paralelas do MiniSAT (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.

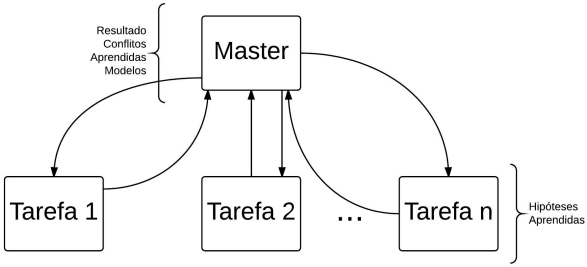

Figura 4 – Arquitetura do PMSat (GIL; FLORES; SILVEIRA, 2008)

Como pode-se ver nas implementações apresentadas esta técnica apresenta vários desafios. O balanceamento 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 distribuídos, 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 algo-

ritmo QuantumSAT.

3.2.3 Particionamento do problema

Esta técnica consiste em dividir a fórmula proposicional 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 presentes exclusivamente em V1 e V2 respectivamente, também é criado um conjunto C3, com as variáveis compartilhadas entre as fórmulas. Cada problema é resolvido independentemente e 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 tendência é um aumento no número de nós dos sistemas.

4 PARAQUANTUMSAT

Nesta seção serão apresentados os diversos aspectos referentes ao desenvolvimento do presente trabalho. Esta seção primeiro apresenta o algorítimo 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.

4.1 IMPLEMENTAÇÃO PARALELA DO ALGORITMO QUANTUMSAT

Como dito anteriormente o algoritmo base 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 espaço para que o problema 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 podem 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. Neste trabalho foi escolhido utilizar a busca em profundidade, a motivação desta descrição é descrita na próxima seção.

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 algorítimo 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:

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

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)
```

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:

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)
```

Na versão implementada neste trabalho o estado é composto por um Ω 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, que é o conjunto de cláusulas que não contém um literal no Ω.

Outra característica da implementação deste trabalho é o cálculo dos sucessores, que é feito utilizando uma função de qualidade, a mesma do algoritmo sequencial apresentado na seção 2.1.5. 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.

A paralelização implementada funciona dividindo o espaço de busca, técnica anteriormente apresentada. Neste caso a divisão funciona gerando uma lista de estados a partir da seleção inicial, expandindo apenas alguns filhos dos estados iniciais selecionados. A partir dai estes estados são colocados numa *state pool* e distribuídos aos nós que realizam a expansão até o final da árvore, assim que um nó realizando a expansão termina sua busca ele envia a solução para o *solution pool* e lhe é atribuído um novo estado para a expandir. Está dinâmica é demonstrada na Figura 5.

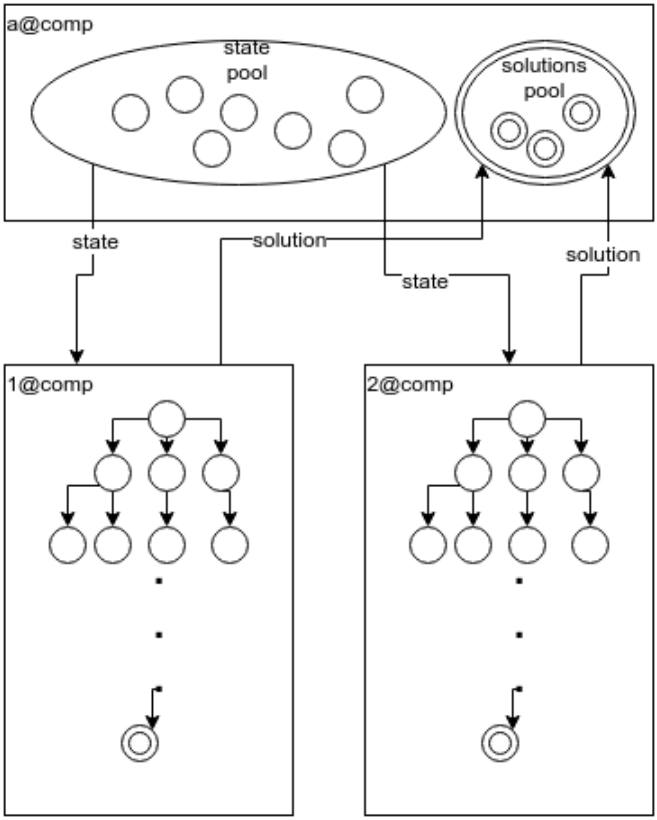

Figura 5 – Dinâmica dos nós

4.2 BUSCA PARALELA

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 decidiu-se utilizar um método empírico, pois as duas implementações são facilmente realizadas e alguns testes rápidos mostraram evidencias suficientes para que uma boa decisão fosse tomada. Realizaram-se 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 balanceamento de carga entre os nós pior, esse cenário pode ser agravado se os nós apresentarem diferenças de performance.

4.3 MODELO E AMBIENTE DE PROGRAMAÇÃO

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 distribuídos, principalmente sistemas que lidam com um grande um número de chamadas e necessitam grande confiabilidade, porém também apresenta alguns usos tem cenários de computação científica (BINGHAM et al., 2010). Está linguagem apresenta como principal vantagem um grande número de ferramenta para computação distribuída, sendo estas ferramentas bastante estáveis e eficientes. Outra vantagem é o 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 (CRONQVIST, 2004). 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.

4.4 IMPLEMENTAÇÃO EM ERLANG

Para o desenvolvimento deste trabalho foi selecionada a linguagem Erlang, a escolha desta linguagem implicou em um série de descisões de projeto e características da implementação. Estes fatores serão apresentados e discutidos nesta seção.

A primeira característica são os *behaviours*, que são comportamentos de diversos tipos de aplicação em Erlang, eles definem as chamadas que um dado processo deve aceitar para ser compativel com aquele comportamento, diversas ferramentas do Erlang utilizam esse comportamentos. A implementação pode ser divida em duas partes, uma que coordena o sistema chamada pqsat e uma que que executa a busca nos nós distribuidos chamada search_tool. Ambas as partes implementam os *behaviours server* e *supervisor*, além disso o pqsat também implementa o *behaviour application*. O *behaviour server* implementa as chamadas para uma relação cliente-servidor, sendo utilizada para distribuir as tarefas em diversos nós. O *behaviour supervisor* implementa as chamadas utilizadas pelo Erlang para monitorar os processos no sistema e como o Erlang deve proceder em caso de morte de um processo.

A segunda característica foi o modelo de programação da linguagem, no caso de Erlang o modelo funcional. Este modelo de programação apresenta diversas vantagens, como o fato de funções puras não utilizarem váriaveis. Porém também apresentou diversos desafios, como a implementação da busca. O algoritmos original possui implementações em três linguagens, C++, Java e LISP. Como LISP também é uma linguagens funcional essa implementação foi utilizada como base foi utilizada como base, porém também foi utilizada a implementação em C++, pois esta estava mais clara.

O pqsat implementa a função *solve*, esta função é responsável por ler os arquivos de entrada, criar um quanta inicial, fazer o spin inicial, resolver o conjunto de inicio, distribuir a busca e receber as respostas. As funções utilizadas para isto são a *calculate_spin*, *make_start_set*, *quantum_maker* e

slave:start.

A search_tool implementa a função *search*, que é a implementação das busca descrita anteriormente. Para isso são utilizadas a funções *get_successors*, que cria a lista de estados sucessores baseado na função de qualidade, a função *make_states*, que cria os estados a serem expandidas localmente e a função *is_final*, que verifica se um dado estado é final.

4.5 EXEMPLO

Neste exemplo utilizaremos dois nós e a pré-execução em paralelo com apenas um nível na arvore. Este exemplo funciona para demostrar 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:
$P_1^{\{3,4,8,9\}}, \neg P_1^{\{1,2,5,6,7,10\}}, P_2^{\{1,3,4,5,6\}}, \neg P_2^{\{2,7,8,9,10\}}, P_3^{\{2,3,4,5,9,10\}}, \neg P_3^{\{1,6,7,8\}}, P_4^{\{1,2,4,5,7,8\}}, \neg P_4^{\{3,6,9,10\}}$

Esse funcionamento é descritp na Figura 5 do algoritmo, na situação citada anteriormente. O nó a@comp é o nó que realiza controle do sistema, ele é o pqsat,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, sendo ambos nós serach_tool, 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 omega: [-1] e o expandirá, assim retornando a solução omega:[-1,4,2,3], omega ..., o nó 2@comp pegará o estado com o omega: [4] e retornará a solução omega: [4,2,3], omega ..., como o nó 2@comp precisa visitar menos estados durante a sua expansão este acabará antes e então pegará o estado com omega: [-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 .

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*.

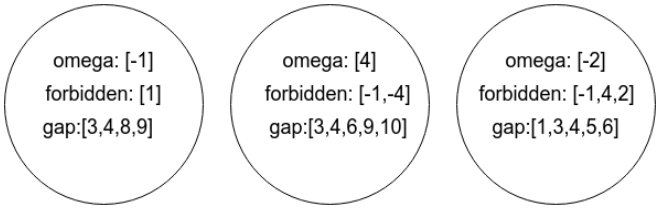

Figura 6 – Estados iniciais

4.6 CONSIDERAÇÕES SOBRE O CAPÍTULO

Este capítulo apresentou o desenvolvimento e implementação do ParaQuantumSAT, alguns pontos devem ser notados a partir daí. Primeiramente a técnica de paralelismo utilizada é bastante conhecida, o particionamento do espaço de busca, para *SAT solvers*, porém a estrutura quantum se mostrou bastante eficiente para realizar esta distribuição. Outro ponto está no fato da linguagem utilizada, neste trabalho o foco está na distribuição da tarefa, não no desempenho absoluto do *SAT solver*. Outro fator bastante interessante é que a estrutura quantum permitiu desenvolver um algoritmo com uma baixa taxa de comunicação e uma boa escalabilidade.

5 AMBIENTE DE TESTES E RESULTADOS EXPERIMENTAIS

Este capítulo apresenta o ambiente onde foram realizados os testes e os resultados obtidos pelos mesmo. Inicialmente são apresentados os ambientes utilizados para se realizar os testes e as motivações para a escolha dos mesmos. A seguir é apresentado o resultado dos tempos de execução do algoritmo implementado e sua comparação com o *SAT solver* Glucose. A seguir é apresentado o teste de escalabilidade do algoritmo com um grande volume de nós. Por fim é apresentado o uso de CPU dos nós do sistema.

5.1 AMBIENTES DE TESTE

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 adéqua 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 de 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 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 possui um core i7-4790 com 16Gb de mémoria ram, a segunda possui um Xeon E31240 com 16Gb de mémoria ram, ambos utilizam o sistema operacional Linux(distribuição Ubuntu). Estas máquinas forma utilizadas para testar uma comparação entre o *SAT solver* implementado e o Glucose, pois este apresenta boa facilidade de uso e um bom desempenho, 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 acessadas 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 foram utilizados 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, sendo cada instancia um problema diferente porém de mesma classe. Também foram utilizados alguns testes da SAT competition para um maior volume de variáveis e cláusulas, porém como foram utilizadas menos instancias o seu valor estatístico fica comprometido.

5.2 RESULTADOS E ANÁLISE DE TEMPO

Os testes 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++, por ser uma linguagem de mais baixo nível apresenta um desempenho melhor do que Erlang, porém Erlang apresenta ferramentas 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, onde os dados estão apresentados em segundos.

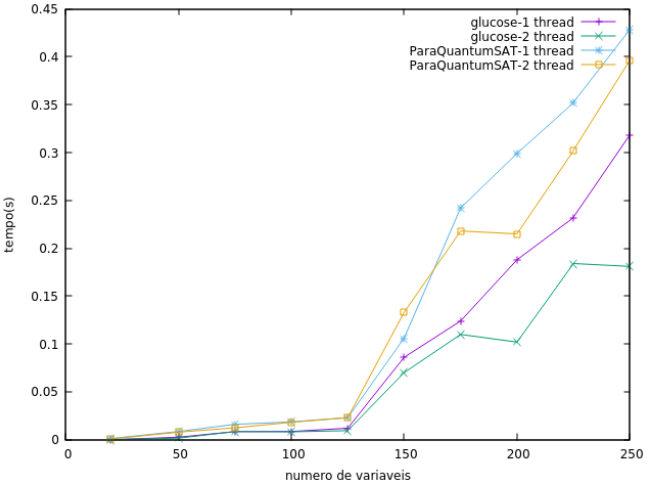

Figura 7 – Comparação com o ParaQuantumSAT e Glucose 1 e 2 threads(multi-core)

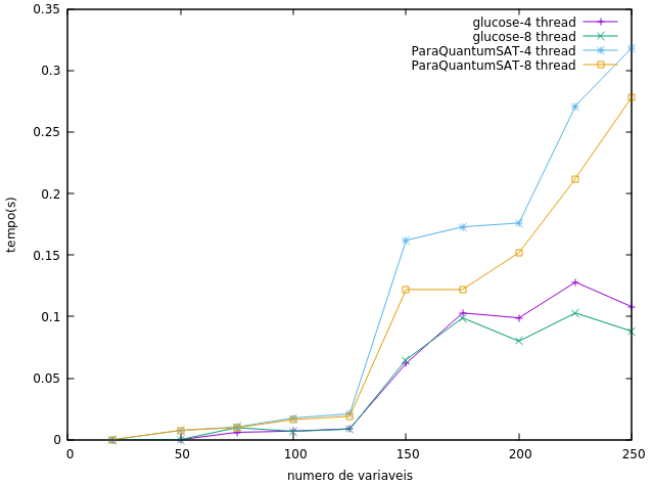

Figura 8 – Comparação com o ParaQuantumSAT e Glucose 4 e 8 threads(multi-core)

Estes resultados são obidos rodando 100 instancias de cada classe do problema, sendo classe o números de variáveis, indo de 20 a 250, e tirando a média dos tempos de execução. Também foi calculado o desvio padrão para os testes de 150 a 250 variáveis, pois para os valores o desvio padrão ficou bastante pequeno. Os desvios padrão paro o Glucose são 0.009 para uf150, 0.033 para uf175, 0.016 para uf200, 0.026 para uf225 e 0.057 para uf250. Os desvios padrão paro o ParaQuantumSAT são 0.012 para uf150, 0.021 para uf175, 0.037 para uf200, 0.042 para uf225 e 0.089 para uf250. Ambos os desvios padrão foram calculados com 1 thread, os desvios padrão para mais threads ficaram proporcionais.

5.3 RESULTADOS E ANÁLISE DE ESCALABILIDADE

Sendo está característica bastante importante em sistemas distribuídos este resultado é bastante importante. Esta implementação apresentou um resultado satisfatório 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(multi-core)

	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

onde os dados estão apresentados em segundos.

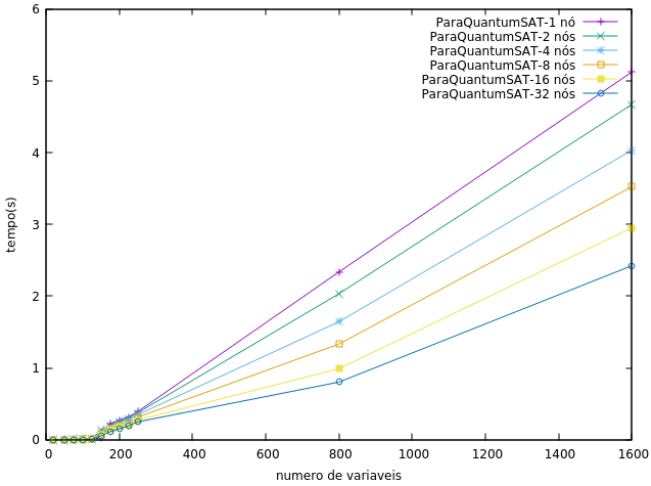

Figura 9 – Teste de escalabilidade(cloud)

Tabela 4 – Tabele de resultados de escalabilidade(cloud)

	ParaQuantumSAT					
Problema	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 em se tratando de um problema NP-completo. Tambem 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.

Como para esse teste é avaliado o tempo para encontrar a primeira solução a adição de nós não apresenta o ganho máximo possível. Sendo assim é bastante difícil de apresentar grandes ganhos de escalabilidade na paralelização do problema SAT.

5.4 RESULTADOS E ANÁLISE DE EFICIÊNCIA

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 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.

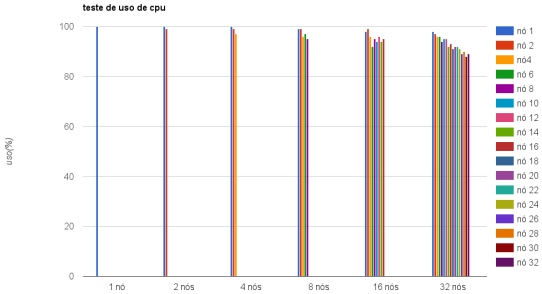

Figura 10 – Teste de escalabilidade

Este resultado é bom de quesito de utilização dos nós, mas devido a solução do problema por se encontrar em qualquer lugar da árvore um maior uso de CPU não necessariamente apresenta um ganho proporcional de desempenho. Isso deve que mesmo processando informação caso a informação processada não seja uma solução esse processamento não é utilizada para nada, além de saber que esta informação é inútil.

5.5 CONSIDERAÇÕES SOBRE O CAPÍTULO

Apesar do baixo desempenho absoluto do *SAT solver* pode-se ver que a escalabilidade da implementação apresentou um bom desempenho. Outro fator a ser observado é que mesmo com 32 nós o uso de CPU dos nós não apresenta uma grande queda nos mesmos. Outro fator observado é que o algoritmo apresenta um desempenho melhor no ambiente de cloud que um CPU multi-core, isso se deve a forma como o algoritmo foi implementado, sendo o ambiente de CPU multi-core gastando um tempo maior na comunicação e na distribuição de carga através do escalonador.

6 TRABALHOS FUTUROS E CONCLUSÃO

6.1 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.

Neste trabalho foi possível compreender as diversas técnicas de computação, o problema SAT e as relações entre ambas as áreas, sendo estudas as diversas técnicas para paralelização dos algoritmos para solução. Com o estudo destas técnicas foi possível compreender qual delas funciona da melhor para paralelizar o algoritmo base utilizado neste trabalho, com base neste estudo pode-se também compreender a qualidade da estrutura quantum para a paralelização do problema SAT. A partir deste estudo foi implementada uma versão paralela do algoritmo, utilizando a linguagem Erlang de forma a obter uma boa performance de escalabilidade e comunicação do algoritmo. Para se estudar a qualidade da implementação a implementação foi testada utilizando problemas SAT aleatórios disponíveis na SATLib e problemas SAT utilizados na *SAT competition*. Para se ter uma referência de comparação foi utilizado o *SAT solver* Glucose, que apresenta uma boa performance e facilidade de uso.

6.2 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 arvore de estados.

Outra melhoria possível está no estudo de melhores heurísticas para realizar a busca de sucessores na arvore, 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.

Também é possível realizar alguns estudos de melhoria da implementação. Podem-se estudar as diversas estruturas de dados utilizadas, de forma a encontrar estruturas mais eficientes. Outro fator possível é adicionar alguns elementos dinâmicos ao algoritmo, sendo um elemento importante quanto nós são gerados na *state pool*, de forma a melhor utilizar os nós do sistema.

REFERÊNCIAS

AUDEMARD, G.; SIMON, L. Glucose 2.1: Aggressive - but Reactive - Clause Database Management, Dynamic Restarts. In: **International Workshop of Pragmatics of SAT (Affiliated to SAT)**. Trento, Italy: [s.n.], 2012. Disponível em: <https://hal.inria.fr/hal-00845494>.

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.

BINGHAM, B. et al. Industrial strength distributed explicit state model checking. In: **Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology**. Washington, DC, USA: IEEE Computer Society, 2010. (PDMC-HIBI '10), p. 28–36. ISBN 978-0-7695-4265-2. Disponível em: <http://dx.doi.org/10.1109/PDMC-HiBi.2010.13>.

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.

CRONQVIST, M. Troubleshooting a large erlang system. In: **Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang**. New York, NY, USA: ACM, 2004. (ERLANG '04), p. 11–15. ISBN 1-58113-918-7. Disponível em: <http://doi.acm.org/10.1145/1022471.1022474>.

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. **An Extensible SAT-solver [ver 1.2]**. 2003.

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

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. 71–98, 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>.

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 – Artigo

ParaQuantumSAT: um algoritmo SAT solver distribuído

João G. Zeni[1], Jerusa Marchi[1], Mario Danta[1]

[1]Instituto de Informática – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

zenister@gmail.com, {jerusa.marchi, mario.dantas}@ufsc.br

Abstract. *The SAT problem is a classic and well known problem, been the first proved NP-complete. For its importance, since many equally NP-complete problems can be solved through reduction to the SAT problem, the study of techniques and algorithms for its solution is always a current issue of research. One way of speed up the achievement of a solution is through parallelism. Parallel computing is a critical study field, since the current speed of processing of computers grows with the parallelisation of processors, been one of the main challenges of the field the building of efficient parallel algorithms to solve classical computing problems. This work intends to develop a parallel SAT solver algorithm, using as base a sequential SAT solver algorithm.*

Resumo. *O problema SAT é um problema clássico e bem conhecido, sendo o primeiro provado ser NP-completo . Por sua importância, já que diversos problemas igualmente NP-completo podem ser solucionados via redução ao problema SAT, o estudo de técnicas e algoritmos para a sua solução é sempre um tema atual de pesquisa. Uma forma de buscar acelerar a obtenção da solução é através de paralelismo. Computação paralela é uma área de estudo crítica, visto que hoje a capacidade de processamento dos computadores cresce com a paralelização dos processadores, sendo um dos principais desafios para a área 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.*

.

1. Introdução

Este trabalho apresenta o desenvolvimento de um algoritmo *SAT solver* distribuído desenvolvido a partir de um algoritmo *SAT solver* sequencial. O desenvolvimento de algoritmos *SAT solver* distribuídos é de vital importância pois computação paralela tem se tornado o modelo de crescimento de desempenho computacional na atual era da computação. Sendo o problema

SAT um problema clássico da computação e apresentando diversas aplicações práticas, este problema apresenta um bom estudo de caso para o desenvolvimento de algoritmos distribuídos.

Este trabalho foca em um algoritmo capaz de ser executado em um cluster de computadores ou em uma cloud. Utilizando a técnica de particionamento do espaço de busca para distribuir as tarefas nos diversos nós do sistema. A linguagem escolhida para implementação foi Erlang, sendo escolhida por suas características positivas na distribuição de tarefas. O algoritmo base apresenta algumas inovações, utilizando uma estrutura de dados auxiliar, chamada quantum, para realizar a busca no espaço de dados.

2. Fundamentação teórica

Para a compreensão deste trabalho é necessário o conhecimento de alguns elementos teóricos apresentados nesta seção. Sendo eles a definição do problema SAT, o modelo de distribuição utilizado no trabalho e o algoritmo base utilizado no trabalho.

O problema SAT é definido sobre uma linguagem de lógica proposicional L (P), onde P = p 1 , p 2 , . . . , p n sendo que todo p é um símbolo proposicional sem valor verdade definido. Uma fórmula Φ ∈ L (P) pode ser escrita em CNF, logo Φ = {C 1 ∧C 2 . . . ∧C k | ∀C,C n = L i ∨ ¬L j . . . ∨ L z }, onde Li = p ∨ ¬p. 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.

Outra característica do problema SAT é o número de símbolos em cada cláusula, o problema usualmente é chamado de kSAT 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 escrita 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).

Este trabalho utiliza como base o algoritmo SAT solver chamado Quantum-SAT (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.

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. O algoritmo é apresentado abaixo.

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: $Ordene(\Theta, \succ)$ ▷ Ordena Θ de acordo com o critério ≻
5: **if** $\exists C \in R_\phi, \Theta \cap C = \emptyset$ **then**
6: **return** $\emptyset$
7: **for** ϕ^F *in* Θ **do**
8: $\Phi^+ \leftarrow \Phi \bigcup \{\phi^F\}$
9: **if** $\forall \phi_i^{F_i}, F_i^* \not\subset F$ and $\emptyset \notin R_{\phi+}$ and $\forall C \in R_{\phi+}, C \not\subset X_\phi$ **then**
10: $\Omega \leftarrow \Omega \bigcup \{\Phi^+\}$
11: **return** Ω

Algoritmo 1. Algoritmo de busca de sucessores de um quantum

O particionamento do espaço de busca consiste em dividir o 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 distribuídos. 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.

3. ParaQuantumSAT

Como dito anteriormente o algoritmo base 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 espaço para que

o problema seja provado insatisfazível ou para que sejam encontradas todas as soluções.

O algoritmo implementa uma busca em profundidade que 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.

Na versão implementada neste trabalho o estado é composto por um Ω 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, que é o conjunto de cláusulas que não contém um literal no Ω.

A paralelização implementada funciona dividindo o espaço de busca, técnica anteriormente apresentada. Neste caso a divisão funciona gerando uma lista de estados a partir da seleção inicial, expandindo apenas alguns filhos dos estados iniciais selecionados. A partir daí estes estados são colocados numa state pool e distribuídos aos nós que realizam a expansão até o final da árvore, assim que um nó realizando a expansão termina sua busca ele envia a solução para o solution pool e lhe é atribuído um novo estado para a expandir. Está dinâmica é demonstrada na Figura 1.

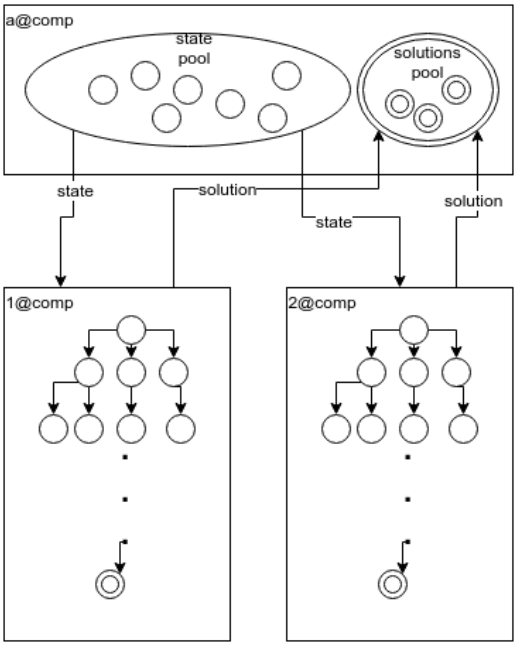

Figura 1. Dinâmica dos nós

O código implementado pode ser dividido em duas parter pqsat e search_tool. O pqsat implementa a função solve, esta função é responsável por ler os arquivos de entrada, criar um quanta inicial, fazer o spin inicial, resolver o conjunto de inicio, distribuir a busca e receber as respostas. A search_tool implementa a função search, que é a implementação a busca descrita anteriormente.

4. Resultados

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 adéqua 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 facilitar, padronizar e ter uma execução mais rápida dos testes foram utilizados 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, sendo

cada instancia um problema diferente porém de mesma classe. Também foram utilizados alguns testes da SAT competition para um maior volume de variáveis e cláusulas, porém como foram utilizadas menos instâncias o seu valor estatístico fica comprometido.

Para os testes de tempo de execução foi realizada uma comparação com o algoritmo SAT *solver* Glucose, pois este apresenta um bom desempenho, já tendo ganho uma SAT competition. Os resultados de tempo são apresentados nos dois gráficos abaixo.

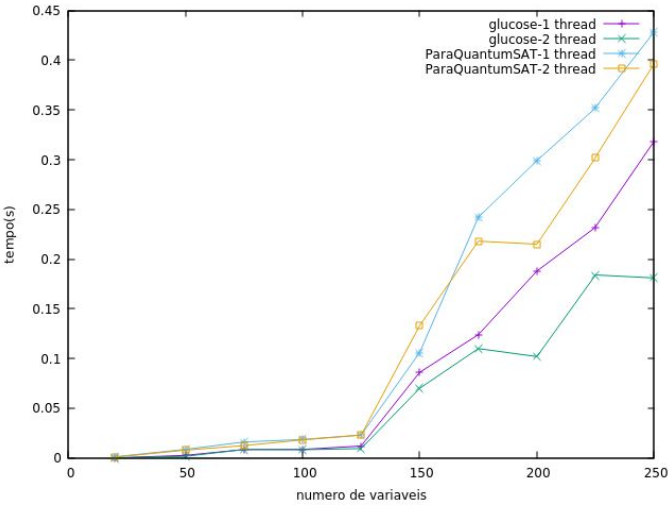

Figura 2.Comparação com o ParaQuantumSAT e Glucose 1 e 2 threads(multi-core)

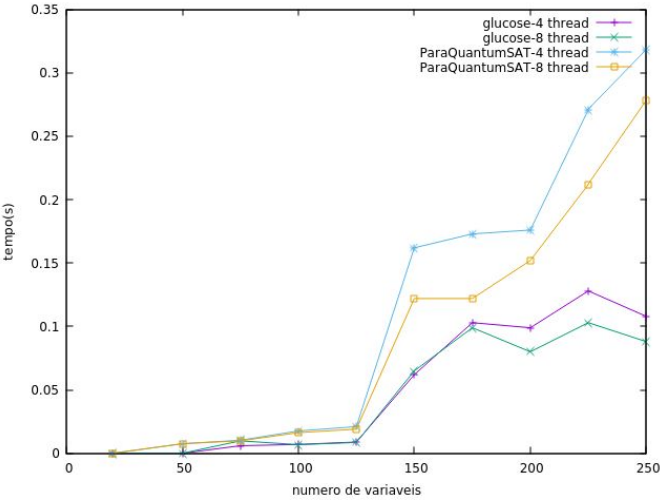

Figura 3.Comparação com o ParaQuantumSAT e Glucose 4 e 8 threads(multi-core)

Também foi realizado um teste de escalabilidade, este teste foi realizado utilizando mais nós e executando os teste em uma cloud. Este teste também utilizou alguns problemas da SAT *competition* que são maiores. Os resultados são apresentados no gráfico abaixo.

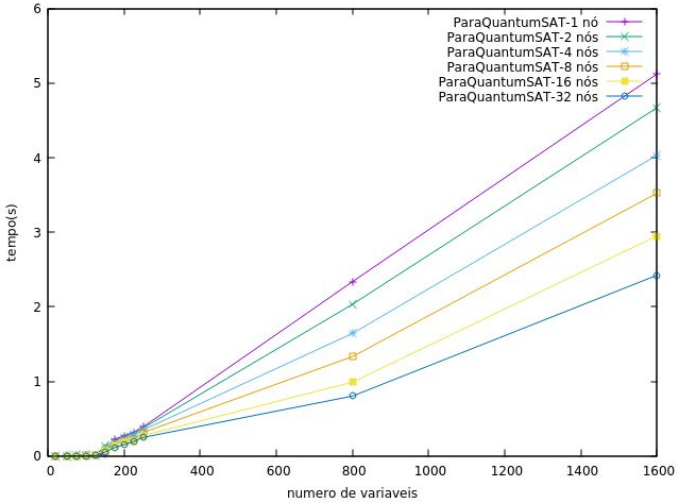

Figura 4.Teste de escalabilidade(cloud)

5. Conclusão e trabalhos futuros

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.

Neste trabalho foi possível compreender as diversas técnicas de computação, o problema SAT e as relações entre ambas as áreas, sendo estuda as diversas técnicas para paralelização dos algoritmos para solução. Com o estudo destas técnicas foi possível compreender qual delas funciona da melhor para paralelizar o algoritmo base utilizado neste trabalho, com base neste estudo pode-se também compreender a qualidade da estrutura quantum para a paralelização do problema SAT. A partir deste estudo foi implementada uma versão paralela do algoritmo, utilizando a linguagem Erlang de forma a obter uma boa performance de escalabilidade e comunicação do algoritmo. Para se estudar a qualidade da implementação a implementação foi testada utilizando

problemas SAT aleatórios disponíveis na SATLib e problemas SAT utilizados na SAT competition. Para se ter uma referência de comparação foi utilizado o SAT solver Glucose, que apresenta uma boa performance e facilidade de uso

References

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.

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

ANEXO A – Código fonte

```
%%%-------------------------------------------------------
%% @doc pqsat public API
%% @end
%%%-------------------------------------------------------

-module(pqsat_app).

-behaviour(application).

%% Application callbacks
-export([start/2, stop/1]).

%% Export client API
-export([solve/3]).

%%====================================================
%% API
%%====================================================

start(_StartType, _StartArgs) ->
    pqsat_sup:start_link().

%%--------------------------------------------------------
stop(_State) ->
    ok.

%%====================================================
%% Internal functions
%%====================================================

solve(Node, DimacsFile, NodesVal) ->
  pqsat_server:solve(Node, DimacsFile, NodesVal).
```

```
%%%-------------------------------------------------------
%% @doc pqsat top level supervisor.
%% @end
%%%-------------------------------------------------------

-module(pqsat_sup).
```

```
-behaviour(supervisor).

%% API
-export([start_link/0]).

%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).

%%====================================================
%% API functions
%%====================================================

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE,
                          []).

%%====================================================
%% Supervisor callbacks
%%====================================================

%% Child :: {Id, StartFunc, Restart, Shutdown, Type
%% , Modules}
init([]) ->
    ChildSpecs =
        [
          {pqsat_server,
          {pqsat_server, start_link, []},
          permanent,
          1000,
          worker,
          [pqsar_server]}
        ],
    {ok, { {one_for_all, 0, 1}, ChildSpecs} }.

%%====================================================
%% Internal functions
%%====================================================
```

```
-module(pqsat_server).
```

```
-behaviour(gen_server).

%%--------------------------------------------------------------------
%% gen_server callbacks
%%--------------------------------------------------------------------
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, terminate/2, code_change/3]).

%%--------------------------------------------------------------------
%% External exports
%%--------------------------------------------------------------------
-export([start_link/0,stop/0,solve/3]).

-include("state.hrl").

-define(SERVER, ?MODULE).

start_link() ->
{ok, _Pid} = gen_server:start_link({local, ?SERVER}
                                   , ?MODULE, [], []).

stop() ->
gen_server:cast(?SERVER, stop).

solve(Node, DimacsFile, NodesVal) ->
gen_server:call({?SERVER, Node},{solve, DimacsFile,
    NodesVal}).

%%====================================================
%% Server functions
%%====================================================
%%
%% --------------------------------------------------------
%% Function: init/1
%% Description: Initiates the server
%% Returns: {ok, State}          |
%%          {ok, State, Timeout} |
%%          ignore               |
%%          {stop, Reason}
%% --------------------------------------------------------
init([]) ->
```

```
error_logger:info_msg("pqsat_server:init/1
  starting~n", []),
  {ok, []}.

%% --------------------------------------------------------
%% Function: handle_call/3
%% Description: Handling call messages
%% Returns: {reply, Reply, State}          |
%%          {reply, Reply, State, Timeout} |
%%          {noreply, State}               |
%%          {noreply, State, Timeout}      |
%%          {stop, Reason, Reply, State}   |
%%          (terminate/2 is called)
%%          {stop, Reason, State}
%%          (terminate/2 is called)
%% --------------------------------------------------------
handle_call({solve, DimacsFile, NodesVal}, _From,
State) ->
{Quanta, DimacsTheory} =
file_processor:process_file(DimacsFile),
io:fwrite("-----------------------------------------\n"),
io:write(Quanta),
io:fwrite("\n"),
io:write(DimacsTheory),
io:fwrite("\n"),
io:fwrite("-----------------------------------------\n"),
io:fwrite("Clauses \n"),
{Clause, Cover} = spin_calculator:calculate_spin(
              Quanta,DimacsTheory, [], 0),
io:fwrite("-----------------------------------------\n"),
io:write(Clause),
io:fwrite("\n"),
io:write(Cover),
StateSet = make_start_set(Clause, [], Quanta, lists:seq(
                  1,length(DimacsTheory)), []),
io:fwrite("#################################~n"),
Node = self(),
Args = "-pa _build/default/lib/*/ebin
-config etc/app.config
-config etc/simple_bridge.config -eval
application:start(pqsat)'",
```

```
{ok, Host} = inet:gethostname(),
Nodes = lists:foldl(
          fun(Val, CurrentNodes) ->
        {ok, CurrentNode} = slave:start(
                Host,
                integer_to_list(Val), Args),
        CurrentNodes ++ [CurrentNode]
          end, [], lists:seq(1, NodesVal)),
NewStateSet = lists:foldl(
          fun(RemoteNode,
              [CurrentState|CurrentStateSet]) ->
              rpc:cast(RemoteNode,
              search_tool, search,
            [[CurrentState], Quanta, [],
             DimacsTheory, Node]),
              CurrentStateSet
          end, StateSet, Nodes),
io:fwrite("CStateSet ~w~n", [NewStateSet]),
receive_loop(),
{reply, {ok, true}, State};

handle_call({solved, Solution}, From, State) ->
io:fwrite("Solution: ~w| From: ~w~n",[Solution, From]),
{noreply, State}.

make_start_set([], StateSet, _, _, _) ->
StateSet;
make_start_set([Quantum|ClauseTail], StateSet, Quanta,
DimacsTheory, CurrentForbidden) ->
State = #state{
    phi=[Quantum],
  forbidden=CurrentForbidden ++ [Quantum, -Quantum],
gap=DimacsTheory -- orddict:fetch(Quantum, Quanta)
          },
NewForbidden = CurrentForbidden ++ [Quantum],
NewStateSet = StateSet ++ [State],
make_start_set(ClauseTail, NewStateSet,
         Quanta, DimacsTheory, NewForbidden).

receive_loop() ->
receive
```

```
  {[], From, Node} -> io:fwrite(
                    "No Solution | From: ~w@~w~n",
                        [From, Node]), receive_loop();
  {Solution, From, Node} -> io:fwrite(
                     "Solution: ~w | From: ~w@~w~n",
                     [Solution, From, Node])
end.

%% --------------------------------------------------------
%% Function: handle_cast/2
%% Description: Handling cast messages
%% Returns: {noreply, State}          |
%%          {noreply, State, Timeout} |
%%          {stop, Reason, State}
%%          (terminate/2 is called)
%% --------------------------------------------------------
handle_cast(stop, State) ->
    {stop, normal, State}.

handle_info(Msg, State) ->
io:format("Unknown msg: ~p~n", [Msg]),
          {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
      {ok, State}.

terminate(_Reason, _State) ->
      ok.
```

```
-module(search_tool).
-export([search/5, search/4, make_states/4]).
-include("state.hrl").

%% Stack Abstractions
-define(IS_EMPTY(S), S =:= []).
-define(REMOVE_FROM_STACK(S), {hd(S), tl(S)}).
%% Parents Abstractions
-define(EMPTY_PARENTS(), dict:new()).
-define(ADD_TO_PARENTS(Node, Cost, Prev, R),
dict:store(Node, {Cost, Prev}, R)).
```

```
search(StateStack, Quanta, Solutions,
       DimacsTheory, Node) ->
  Solution = search(StateStack, Quanta,
                    Solutions, DimacsTheory),
  Node ! {Solution, self(), node()}.

-spec search([], orddict:orddict(), [], []) -> [].
search(StateStack, Quanta,
       Solutions, DimacsTheory) ->
  case ?IS_EMPTY(StateStack) of
true -> Solutions;
false ->
{State, StackTail} = ?REMOVE_FROM_STACK(StateStack),
io:fwrite("Succ ~w-~w~n", [
    [State#state.phi,State#state.gap], [node()]]),
Successors = get_successors(
               State#state.gap,
               orddict:new(),
               State, Quanta,
               DimacsTheory),
{NewSolutions, NewStack} =
  lists:foldr(
fun(Successor,{PassedSolutions, PassedStack}) ->
case is_final(Successor, Quanta) of
{_, true} ->
  IntermediateSolution = [Successor|PassedSolutions],
  {IntermediateSolution, PassedStack};
{true, _} ->
  {PassedSolutions, PassedStack};
{false, _} ->
  IntermediateStack = [Successor|PassedStack],
  {PassedSolutions, IntermediateStack}
end
    end,
    {Solutions, StackTail}, Successors),
  search(NewStack, Quanta, NewSolutions, DimacsTheory)
end.

get_successors([], Successors, State, _, _) ->
  make_states(merge_sort:sort(
  orddict:to_list(Successors)), [], State, []);
```

```
get_successors([GapClause | Gap],
  Successors, State, Quanta, DimacsTheory) ->
  Clause = lists:nth(GapClause, DimacsTheory),
  NewSuccessors = lists:foldl(
fun(Quantum, IntermediateSuccessors) ->
case lists:member(Quantum, State#state.forbidden) of
true -> IntermediateSuccessors;
false -> Cover = sets:intersection(
  sets:from_list(State#state.gap),
sets:from_list(orddict:fetch(Quantum,Quanta))),
case sets:size(Cover) > 0 of
  true ->
    orddict:store(Quantum,
  sets:to_list(Cover), IntermediateSuccessors);
  false -> IntermediateSuccessors
end
    end
end,
      Successors, Clause),
  get_successors(Gap,
NewSuccessors, State, Quanta, DimacsTheory).

make_states([], StatesSuccessors, _, _) ->
  StatesSuccessors;
make_states([QuantumSuccessor|QuantumSuccessors],
StatesSuccessors, CurrentState, CurrentForbbiden) ->
  {QuantumVal, QuantumCover} = QuantumSuccessor,
  State = #state{
 phi=CurrentState#state.phi ++ [QuantumVal],
 forbidden=CurrentState#state.forbidden ++
[QuantumVal,-QuantumVal] ++ CurrentForbbiden,
 gap=CurrentState#state.gap -- QuantumCover
},
  NewStatesSuccessors = StatesSuccessors ++ [State],
  NewCurrentForbbiden = CurrentForbbiden ++ [QuantumVal],
  make_states(QuantumSuccessors, NewStatesSuccessors,
CurrentState, NewCurrentForbbiden).

is_final(State, Quanta) ->
  GapSize = 0 == length(State#state.gap),
  RemainingQuanta = orddict:fetch_keys(Quanta)
```

```
  -- State#state.forbidden,
  RemainingGap = lists:foldl(
fun(Quantum, IntermediateGap) -> IntermediateGap
      -- orddict:fetch(Quantum, Quanta) end
, State#state.gap, RemainingQuanta),
  RemainingGapSize = 0 < length(RemainingGap),
  {RemainingGapSize, GapSize}.
```

```
-module(search_tool_server).

%%--------------------------------------------------
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2,
handle_info/2, terminate/2, code_change/3]).

%%--------------------------------------
%% External exports
%%--------------------
-export([start_link/0,stop/0,
search/4]).

-include("state.hrl").

%% Server Abstraction
-define(SERVER, ?MODULE).
%% Stack Abstractions
-define(IS_EMPTY(S), S =:= []).
-define(REMOVE_FROM_STACK(S), {hd(S),
tl(S)}).
%% Parents Abstractions
-define(EMPTY_PARENTS(), dict:new()).
-define(ADD_TO_PARENTS(Node, Cost, Prev, R),
dict:store(Node, {Cost, Prev}, R)).

start_link() ->
  {ok, Pid} = gen_server:start_link({local, ?SERVER},
?MODULE, [], []).

stop() ->
  gen_server:cast(?SERVER, stop).
```

```
call_search(Node, StateStack, Quanta, DimacsTheory) ->
  gen_server:call({?SERVER, Node},{search, StateStack,
Quanta, DimacsTheory}).

%%====================================
%% Server functions
%%====================================
%%
%% --------------------------------------
%% Function: init/1
%% Description: Initiates the server
%% Returns: {ok, State}          |
%%          {ok, State, Timeout} |
%%          ignore               |
%%          {stop, Reason}
%% --------------------------------------------
init([]) ->
  error_logger:info_msg(
"search_tool:init/1 starting~n", []),
  {ok, []}.

%% --------------------------------------------------
%% Function: handle_call/3
%% Description: Handling call messages
%% Returns: {reply, Reply, State}          |
%%          {reply, Reply, State, Timeout} |
%%          {noreply, State}               |
%%          {noreply, State, Timeout}      |
%%          {stop, Reason, Reply, State}   |
%%(terminate/2 is called)
%%          {stop, Reason, State}
%%(terminate/2 is called)
%% --------------------------------------------------
handle_call({search, StateStack,
Quanta, DimacsTheory}
, From, State) ->
  Solution = search(StateStack, Quanta, [],
DimacsTheory),
  {reply, {ok, Solution}, State}.

%% Function: handle_cast/2
```

```
%% Description: Handling cast messages
%% Returns: {noreply, State}          |
%%          {noreply, State, Timeout} |
%% --------------------------------------------------------
handle_cast(stop, State) ->
    {stop, normal, State}.

handle_info(Msg, State) ->
      io:format("Unknown msg: ~p~n", [Msg]),
          {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
      {ok, State}.

terminate(_Reason, _State) ->
      ok.

%%==========================================
%% Internal functions
%%==========================================
-spec search([], orddict:orddict(), [], []) -> [].
search(StateStack, Quanta, Solutions, DimacsTheory) ->
  case ?IS_EMPTY(StateStack) of
    true -> Solutions;
    false ->
      io:fwrite("------------------------~n",[]),
      {State, StackTail} =
?REMOVE_FROM_STACK(StateStack),
      io:fwrite("Succ ~w~n",
[[State#state.phi,State#state.gap]]),
      Successors = get_successors(
State#state.gap, orddict:new(),
State, Quanta, DimacsTheory),

      {NewSolutions, NewStack} =
        lists:foldr(
          fun(Successor,{PassedSolutions,
PassedStack}) ->
              case is_final(Successor, Quanta)
of %Return->{final, solution}
                  {_, true} ->
```

```
                    IntermediateSolution =
[Successor|PassedSolutions],
 {IntermediateSolution, PassedStack};
{true, _} ->
  {PassedSolutions, PassedStack};
{false, _} ->
IntermediateStack = [Successor|PassedStack],
{PassedSolutions, IntermediateStack}
                end
            end,
 {Solutions, StackTail}, Successors),
search(NewStack, Quanta, NewSolutions, DimacsTheory)
  end.

get_successors([], Successors, State,
_, DimacsTheory) ->
  Succs = merge_sort:sort(
orddict:to_list(Successors)),
  make_states(merge_sort:sort(
orddict:to_list(Successors)),
[], State, []);
get_successors([GapClause | Gap],
Successors, State, Quanta,
DimacsTheory) ->
  Clause = lists:nth(GapClause, DimacsTheory),
  NewSuccessors = lists:foldl(
fun(Quantum, IntermediateSuccessors) ->
case lists:member(Quantum,
State#state.forbidden) of
true -> IntermediateSuccessors;
false -> Cover = sets:intersection(
sets:from_list(State#state.gap),

sets:from_list(orddict:fetch(Quantum,Quanta))),
case sets:size(Cover) > 0 of
 true ->
orddict:store(Quantum,
sets:to_list(Cover), IntermediateSuccessors);
false -> IntermediateSuccessors
end
end
```

```
end,
Successors, Clause),
  get_successors(Gap, NewSuccessors, State,
Quanta, DimacsTheory).

make_states([], StatesSuccessors, _, _) ->
  StatesSuccessors;
make_states([QuantumSuccessor|QuantumSuccessors],
StatesSuccessors,
 CurrentState, CurrentForbbiden) ->
  {QuantumVal, QuantumCover} = QuantumSuccessor,
  State = #state{
phi=CurrentState#state.phi ++ [QuantumVal],
forbidden=CurrentState#state.forbidden ++
[QuantumVal,-QuantumVal] ++ CurrentForbbiden,
gap=CurrentState#state.gap -- QuantumCover
},
  NewStatesSuccessors = StatesSuccessors
++ [State],
  NewCurrentForbbiden = CurrentForbbiden
 ++ [QuantumVal],
  make_states(QuantumSuccessors,
NewStatesSuccessors,
CurrentState, NewCurrentForbbiden).


is_final(State, Quanta) ->
  GapSize = 0 == length(State#state.gap),
  RemainingQuanta = orddict:fetch_keys(Quanta)
-- State#state.forbidden,
  RemainingGap = lists:foldl(
                   fun(Quantum, IntermediateGap) ->
IntermediateGap -- orddict:fetch(Quantum, Quanta) end
                   , State#state.gap, RemainingQuanta),
  RemainingGapSize = 0 < length(RemainingGap),
  {RemainingGapSize, GapSize}.
```

```
-module(search_tool_sup).
```

```
-behaviour(supervisor).

%% API
-export([start_link/0]).

%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).



start_link() ->
    supervisor:start_link({local, ?SERVER},
?MODULE, []).


init([]) ->
    ChildSpecs =
        [
{search_tool_server,
{search_tool_server, start_link, []},
permanent,
1000,
worker,
[search_tool_server]}
        ],
    {ok, { {one_for_all, 0, 1}, ChildSpecs} }.
```

```
-module(spin_calculator).
-export([calculate_spin/4]).

calculate_spin(Quanta, [], CurrentClause,
ClauseCover) ->
  NewCurrentClause =
ordenate_by_cover(CurrentClause, [],Quanta),
  {NewCurrentClause,
ClauseCover};
calculate_spin(
Quanta, [Clause | DimacsTheory], CurrentClause,
ClauseCover) ->
```

```
  Cover = calculate_clause_spin(Quanta,
sets:new(), Clause),
  case length(Cover) > ClauseCover of
    true -> NewCurrentClause = Clause,
NewClauseCover = length(Cover);
    false -> NewCurrentClause = CurrentClause,
NewClauseCover = ClauseCover
  end,

  calculate_spin(Quanta,
DimacsTheory, NewCurrentClause, NewClauseCover).

calculate_clause_spin(Quanta, Cover, [Var | []]) ->
  NewCover = sets:union(Cover,
sets:from_list(orddict:fetch(Var,Quanta))),
  sets:to_list(NewCover);
calculate_clause_spin(Quanta,
Cover, [Var | Clause]) ->
  NewCover = sets:union(Cover,
sets:from_list(orddict:fetch(Var,Quanta))),
  calculate_clause_spin(Quanta, NewCover, Clause).

ordenate_by_cover([], Clause, _) ->
  Clause;
ordenate_by_cover([Quantum | ClauseTail],
[], Quanta) ->
  NewClause = [Quantum],
  ordenate_by_cover(ClauseTail, NewClause,
Quanta);
ordenate_by_cover([Quantum | ClauseTail],
 Clause, Quanta) ->
  [NewQuantum | NewTail] = Clause,
  case length(orddict:fetch(Quantum,Quanta)) >
length(orddict:fetch(NewQuantum,Quanta)) of
    true -> NewClause = [Quantum]
 ++ [NewQuantum] ++ NewTail;
    false -> NewClause = [NewQuantum]
++ [Quantum] ++ NewTail
  end,
  ordenate_by_cover(ClauseTail, NewClause, Quanta).
```

```
-module(quantum_maker).
-export([make_quanta/1]).

process_line_start(Device,
[99 | _]) ->
Line = io:get_line(Device, ""),
process_line_start(Device, Line);
process_line_start(Device,
[112 | T]) ->
Line = io:get_line(Device, ""),
[_ | [NVarsS | _]] = string:tokens(T, " "),
{NVars, _} = string:to_integer(NVarsS),
IDict = orddict:new(),
QuantumI = make_initial_quantum([NVars], IDict),
process_lines(Device, QuantumI, Line, 1).

make_initial_quantum([0], QuantumI) ->
QuantumI;
make_initial_quantum([Val], QuantumI) ->
QuantumIVal = orddict:store(Val,[],QuantumI),
NewQuantumI = orddict:store(-Val,[],QuantumIVal),
make_initial_quantum([Val-1], NewQuantumI).

process_lines(Device, Quantum, eof, _) ->
ok = file:close(Device),
Quantum;
process_lines(Device, Quantum, Line, Clause) ->
VarList = string:tokens(Line, " "),
NewQuantum = process_line(VarList, Quantum, Clause),
NewLine = io:get_line(Device, ""),
process_lines(Device, NewQuantum, NewLine, Clause+1).

process_line([Var | ["0\n"]], Quantum, Clause) ->
    {VarI, _} =
string:to_integer(Var),
    NewQuantum =
orddict:append(VarI, Clause, Quantum),
    NewQuantum;
process_line([Var | T], Quantum, Clause) ->
    {VarI, _} =
string:to_integer(Var),
```

```
    NewQuantum =
orddict:append(VarI, Clause, Quantum),
    process_line(T, NewQuantum, Clause).

make_quanta(FileName) ->
    io:fwrite("O mundo ainda faz sentido\n"),
    {ok, Device} = file:open(FileName, [read]),
    Line = io:get_line(Device, ""),
    process_line_start(Device, Line).
```

```
-module(merge_sort).
-export([sort/1]).

sort([]) -> [];
sort([X | []]) -> [X];

sort(ToSort) ->
  {FirstList, SecondList} =
lists:split(round(length(ToSort) / 2), ToSort),
  merge(sort(FirstList), sort(SecondList)).

merge(X, []) -> X;
merge([], Y) -> Y;
merge([{QuantumX, CoverX} | TailX],
 [{QuantumY, CoverY} | TailY]) ->
  IntersectionCoverXY = sets:intersection(
sets:from_list(CoverX),sets:from_list(CoverY)),
  SizeX = sets:size(sets:subtract(
sets:from_list(CoverX), IntersectionCoverXY)),
  SizeY = sets:size(sets:subtract(
sets:from_list(CoverY), IntersectionCoverXY)),
  case SizeX >= SizeY of
    true -> [{QuantumY, CoverY}] ++
merge([{QuantumX, CoverX} | TailX], TailY);
    false -> [{QuantumX, CoverX}] ++
merge(TailX, [{QuantumY, CoverY} | TailY])
  end.
```

```
-module(gap_calculator).
-export([calculate_gap/1]).
```

```
calculate_gap(Quanta) ->
  ok.
```

```
-module(file_processor).
-export([process_file/1]).
-include("state.hrl").

process_line_start(Device, [99 | _]) ->
    Line = io:get_line(Device, ""),
    process_line_start(Device, Line);
process_line_start(Device, [112 | T]) ->
    Line = io:get_line(Device, ""),
    [_ | [NVarsS | _]] =
string:tokens(T, " "),
    {NVars, _} = string:to_integer(NVarsS),
    IDict = orddict:new(),
    QuantumI = make_initial_quantum(
[NVars], IDict),
    process_lines(Device, QuantumI,
Line, [], 1).

make_initial_quantum([0], QuantumI) ->
    QuantumI;
make_initial_quantum([Val], QuantumI) ->
    QuantumIVal = orddict:store(
Val,[],QuantumI),
    NewQuantumI = orddict:store(
-Val,[],QuantumIVal),
    make_initial_quantum([Val-1], NewQuantumI).

process_lines(Device, Quantum, eof, Clauses, _) ->
    ok = file:close(Device),
    {Quantum, Clauses};
process_lines(Device, Quantum, Line,
Clauses, Clause) ->
    VarList = string:tokens(Line, " "),
    ConvertedClause = lists:map(fun(X) ->
 {Int,_} =
string:to_integer(X), Int end, VarList --
 ["0\n"]),
    NewClauses = Clauses ++
```

```
[ConvertedClause],
    NewQuantum = process_line(VarList,
Quantum,
Clause),
    NewLine = io:get_line(Device, ""),
    process_lines(Device, NewQuantum,
NewLine,
NewClauses, Clause+1).

process_line([Var | ["0\n"]], Quantum, Clause) ->
    {VarI, _} = string:to_integer(Var),
    NewQuantum = orddict:append(VarI, Clause,
Quantum),
    NewQuantum;
process_line([Var | T], Quantum, Clause) ->
    {VarI, _} = string:to_integer(Var),
    NewQuantum = orddict:append(VarI, Clause,
Quantum),
    process_line(T, NewQuantum, Clause).

process_file(FileName) ->
    io:fwrite("O mundo ainda faz sentido\n"),
    {ok, Device} = file:open(FileName, [read]),
    Line = io:get_line(Device, ""),
    process_line_start(Device, Line).
```