Universidade de Brasília - UnB
Faculdade do Gama - FGA
Adrianne Alves da Silva e Matheus Roberto da Silva

Relatório do desenvolvimento do verificador SAT

Boolean Constraint Propagator

O problema clássico denominado SAT tem sido de muito interesse em diversas áreas da ciência da computação. Ele consiste em decidir uma valoração para um conjunto de variáveis booleanas que tornam uma fórmula verdadeira. Os algoritmos desenvolvidos para solucionar esse problema se baseiam em diversos princípios, como resolução, procura, procura local e outros, sendo eles completos, quando se encontra a solução ou prova que a solução não existe ou estocásticos, que não provam que uma solução não existe.

De 2004 para trás a pesquisa tem se voltado para algoritmos baseados no DPLL Davis–Putnam–Logemann–Lovell, um algoritmo do tipo completo que une backtracking com propagação unitária. Entretanto, os algoritmos considerados mais importantes são Sato, Grasp, Chaff e Berkmin. O básico para solucionar SAT foi proposto por Davis que consistia em descobrir se cláusula era verdadeira com o assinalamento parcial, era recursivo e construía árvore de procura através da técnica dividir para conquistar, mas seu pior caso era exponencial.

Cada uma das heurísticas existentes buscam procurar o resultado de uma fórmula e são melhorias da primeira abordagem, considerando aspectos de estruturas de dados (baseadas em contadores e lazy). Nas baseadas em contadores, as cláusulas são listas de literais e para cada variável é mantida uma lista de todas as cláusulas que têm ocorrência dela. A lazy, por sua vez, mantém, para cada variável, uma lista de um número reduzido de cláusulas onde esta variável aparece.

As estruturas baseadas em contadores apresentam o inconveniente de que, a cada vez que um valor é atribuído à uma variável, devemos analisar todas as cláusulas em que aparece esta variável. Uma melhor alternativa consiste em um valor assinalado a uma variável e então são identificadas as cláusulas que se tornam v e as que passam a não ser mais satisfeitas (SATO). Então, conforme a dissertação analisada, essa estrutura é melhorada no Chaff com o uso de duas sentinelas com

backtracking sem custos. Existem várias e diferentes abordagens que se preocupam em reduzir o tempo de processamento utilizando estruturas de dados mais eficientes, para melhorar a propagação, abordagens para melhorar a performance, reduzir o uso de memória, etc.

Referência:

FUX, Jacques. Análise de Algoritmos SAT para Resolução de Problemas Multivalorados. Tese (Mestrado em ciência da computação) - Universidade Federal de Minas Gerais. Belo Horizonte. 2004.

Somativa 2 - PSPD

O desafio deste trabalho consistiu em iniciar a implementação de um algoritmo para atacar esse problema, mas compreendendo apenas a primeira etapa, a elaboração do verificador. Para isso a implementação foi feita do zero, sem vícios de outras informações ou algoritmos existentes, visto que o foco consistia na análise do algoritmo elaborado e detecção dos gargalos. Apesar dos inúmeros testes realizados foram explicitadas 3 versões do algoritmo, sendo elas a primeira versão sequencial, a segunda versão sequencial e o algoritmo paralelizado. A única restrição consiste em obter um algoritmo sequencial melhor que a versão fornecida pelo professor, ademais a principal atividade consiste na análise do algoritmo para os casos fornecidos que variam em quantidade de cláusulas, quantidade de variáveis e de entradas.

Alunos que desenvolveram esse trabalho:

- Adrianne Silva (160047595)
- Matheus Roberto (130126721)

Link repositório original: https://github.com/drianne/boolean_sat_problem/

Execução dos algoritmos

- Para executar a última versão sequencial,

1. Compilar g++ versao_sequencial.cpp -O2 -static -std=c++14 -o sequencial

2. Executar ./sequencial < entrada.txt, sendo a entrada qualquer arquivo de texto com o formato de entrada especificado.

- Para executar a versão paralela, fazer:

1. Compilar g++ versao_paralelismo.cpp -O2 -static -std=c++14 -o paralelo
2. Executar ./paralelo < entrada.txt, sendo a entrada qualquer arquivo de texto com o formato de entrada especificado.

* O número de threads precisa ser ajustado via código

Sobre os testes

- Os testes do algoritmo sequencial na primeira versão foram feitos em máquina própria com as seguintes configurações:

CPU(s): 4
Thread(s) per núcleo 2
Model name: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
Virtualização: VT-x
cache de L1d: 32K
cache de L1i: 32K
cache de L2: 256K
cache de L3: 4096K

- Os testes do algoritmo final paralelizado e da segunda versão do sequencial foram feitos na choccocino;

As tabelas e gráficos gerados encontram-se todos no link abaixo:

https://docs.google.com/spreadsheets/d/1eEP1vTU4rsEymUXDW3mxZrAzqT_uUPhf1zf1vE6-7VA/edit?usp=sharing

Análise algorítmo sequencial V1

A primeira versão do algoritmo está presente na tag v1.0, bastando dar um git checkout v1.0.

1. Estratégia de implementação

O algoritmo foi implementado em C++. Utilizou-se como estrutura de dados hashes de chave e valor para armazenar as cláusulas (vector<map<int, int>>), um vetor de inteiros para armazenar as mudanças a serem aplicadas e uma estrutura de array de chaves e valores para armazenar os literais errados totais.

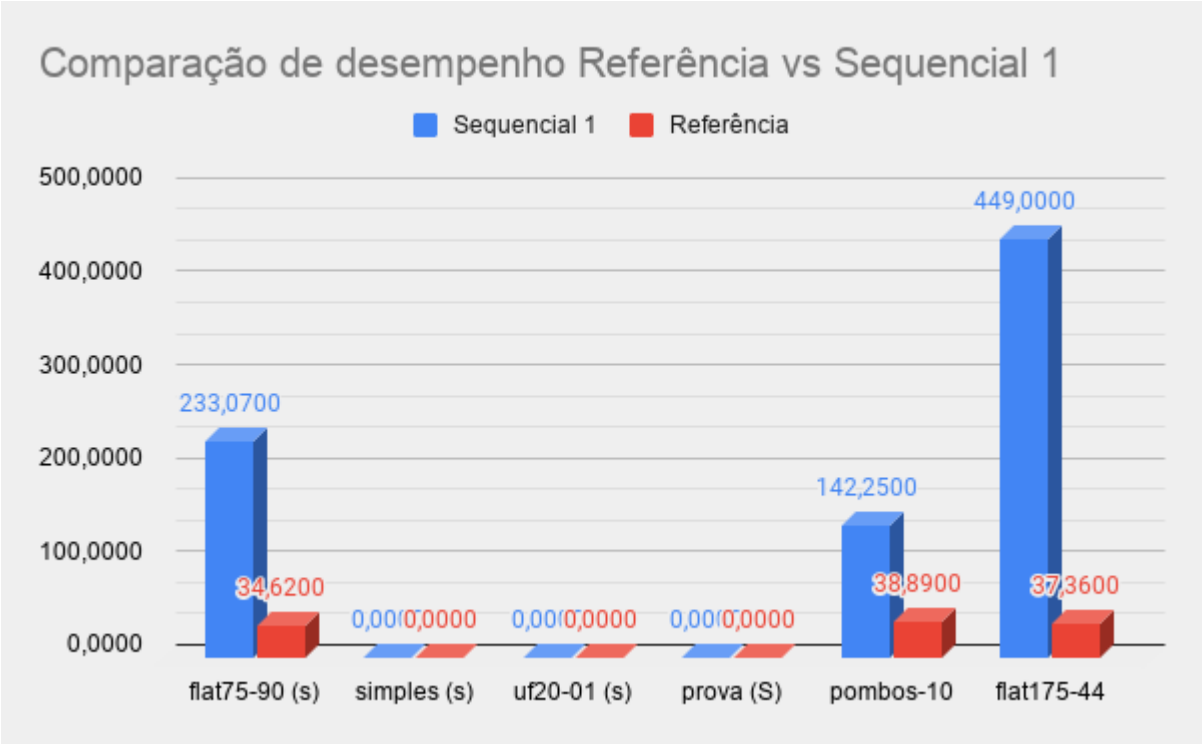
O parser de entrada consiste na leitura inicial do número de variáveis máximo das cláusulas e o número de cláusulas. Então essas cláusulas são lidas e armazenadas de acordo com o seu valor (positivo ou negativo). Então são lidos os comandos, literal por literal, quando full, São lidas as mudanças e essas são armazenadas e então o comando é tratado. Em caso de flip lê-se um valor, armazena a mudança solicitada e o comando é tratado.

A propagação de valores acontece da seguinte forma:

A função verifica_full usa as cláusulas na posição -1 para guardar a contagem de positivos e a para cada variável possível é testada a mudança salva para aquele literal. Se positivo, é contabilizado variáveis positivas, senão é guardado o literal errado da cláusula. Ao final do tratamento do comando, é testado se a cláusula é falsa e somada aos literais errados finais.

No flip para cada cláusula é tratado o valor de verdadeiros para ela, subtraindo ou incrementando o número de verdadeiros no vetor de hashes literais_errados_total.

2. Desempenho comparado à solução de referência



O algoritmo apresenta performance praticamente idêntica em casos de entrada muito pequenas, como das fórmulas simples, uf20-01 e prova. Conforme o tamanho das entradas aumenta, a diferença passa a ser perceptível. No algoritmo pombos a execução é praticamente 4 vezes maior e com entradas ainda maiores, como por exemplo flat75-90 essa diferença aumenta, sendo a execução do algoritmo de referência 12 vezes menor.

3. Gargalo do algoritmo

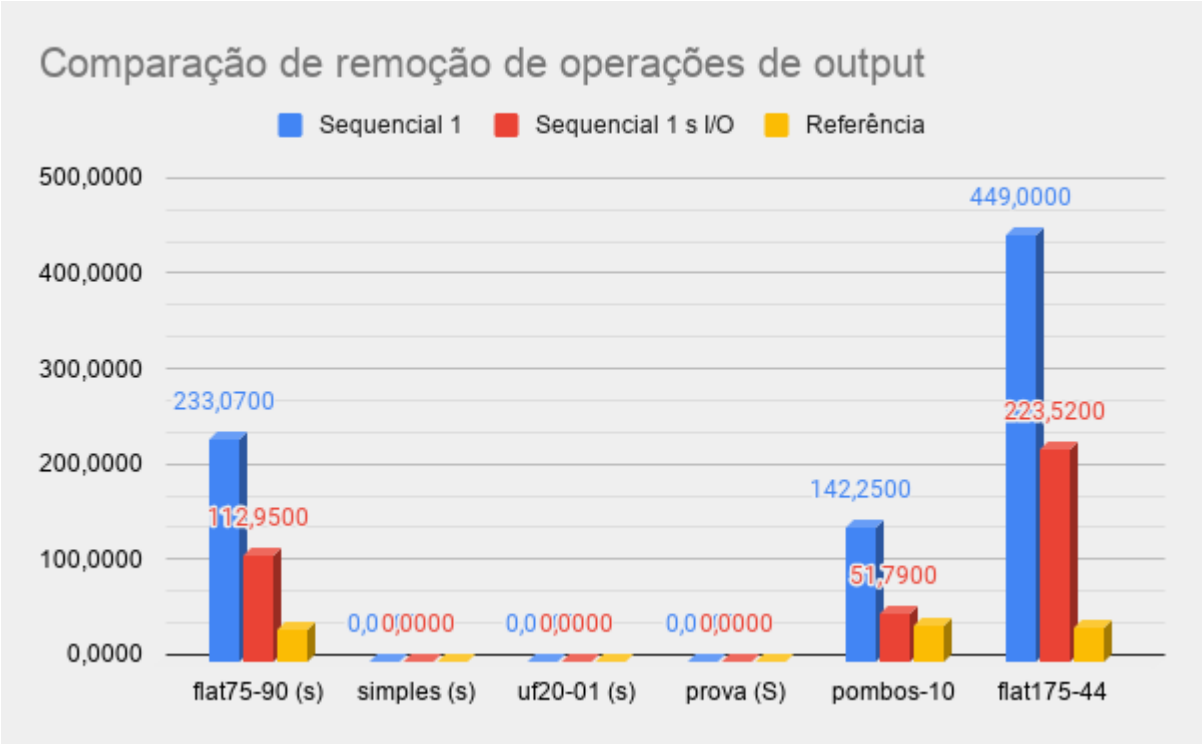
Analisando com o strace pode-se perceber que dentre as chamadas de sistema o tempo mais gasto é com a escrita, esta inclui a manipulação/escrita em memória. Além disso, como visto na imagem abaixo, esse tempo é seguido pelas chamadas “brk” que dizem respeito ao gerenciamento de memória, visto que define o fim do seguimento de dados alocado para o processo referente ao programa rodado. Depois desses dois, outro que representa significativamente o tempo é a leitura dos dados.

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.78    0.232002           4     63472           write
  0.12    0.000273           4        72           brk
  0.10    0.000229           3        73           read
  0.00    0.000000           0         2           fstat
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         1           readlink
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.232504                 63624         1 total
```

Usando o /usr/bin/time -v verificou-se um alto número de page faults do tipo menor o que prejudica o tempo de execução do algoritmo. No caso do minor há a possibilidade de a página ter sido carregada na memória no momento da falha mas não ter sido marcada na unidade de gerenciamento de memória como carregada. Pode acontecer por compartilhamento da memória com outros programas em que estes tenham levado a página à memória anteriormente. Como essas falhas não envolvem latência de disco, elas são mais rápidas e menos caras do que as principais falhas de página. Além disso, na execução do flat75-90 houve um uso de 53% da CPU. Efetivamente, foram gastos 10604 kbytes e houveram 2566 trocas de uso de CPU cedidas por processos e 1155 compartilhamentos de CPU.

```
Command being timed: "./a.out"
User time (seconds): 13.99
System time (seconds): 0.61
Percent of CPU this job got: 53%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:27.29
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 10604
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 2315
Voluntary context switches: 2566
Involuntary context switches: 1155
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

Com a remoção da impressão no stdout os tempos vivenciados se diferenciaram de forma expressiva



Para melhorar a performance desse algoritmo seria preciso otimizar a propagação de valores por meio das estruturas de dados utilizadas, o que nos levou a elaborar o segundo algoritmo.

Análise algoritmo sequencial V2

A segunda abordagem teve como foco mudar a maneira como os dados se propagam, com ajuste das estruturas de dados e do tratamento deles, a fim de reduzir o custo das manipulações de memória.

Para acessar a segunda versão, como não foi gerada a tag, fazemos:
git checkout 1348ce6f17b5cb4f24b845029afab34f827aa5f6

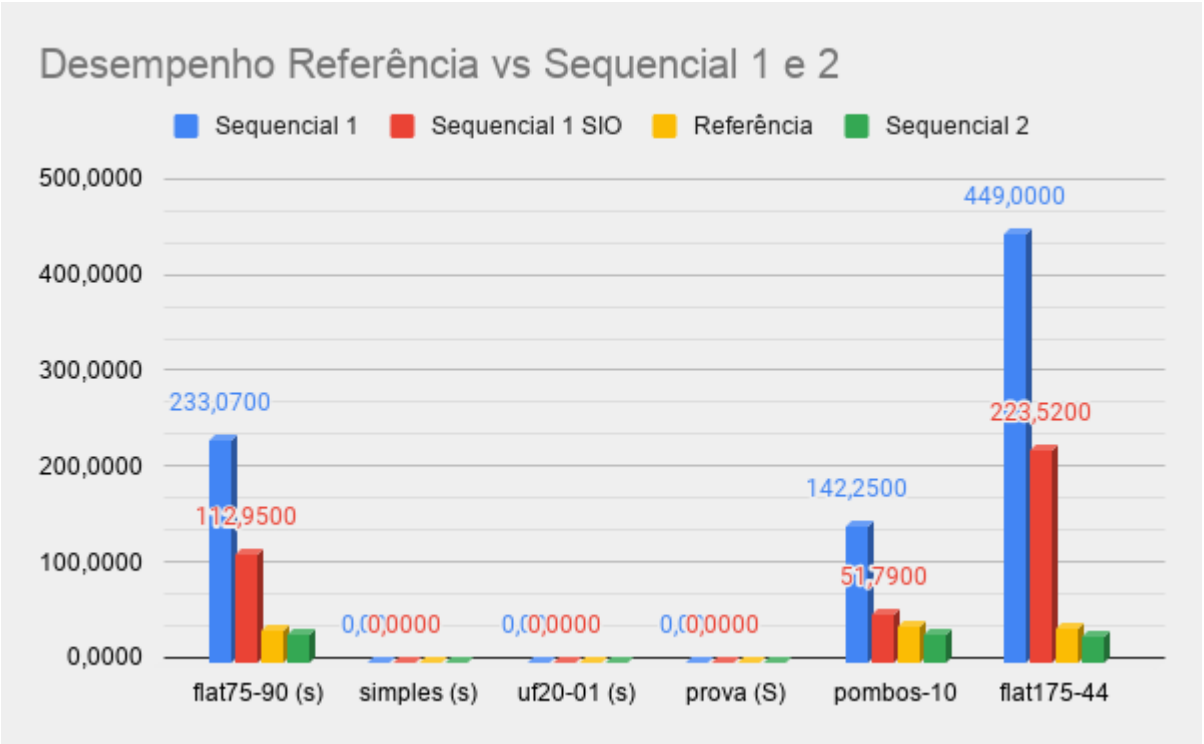
1. Estratégia de implementação

Como mencionado para essa nova implementação sequencial focou-se em otimizar a propagação dos comandos e das estruturas de dados utilizadas. Dessa forma, as cláusulas deixaram de ser map e se tornaram um vetor e foi criada uma nova estrutura do tipo map para salvar todas as cláusulas de um literal. Essa mudança por si só representou uma melhora, mas a mudança crucial consistiu na otimização do tratamento dos comandos flip, deixando de varrer todas as cláusulas no

processo, percorrendo o loop só nas cláusulas em que o literal a ser tratado aparece. Além disso, foi criado um map de vetor para guardar as variáveis das cláusulas, para facilitar a pesquisa pelo literal procurado e obter as cláusulas pertencentes.

O parser de entrada permaneceu o mesmo, porém a propagação de valores foi modificada. As mudanças, armazenadas num vetor booleano são modificadas para cada valor das cláusulas no comando full, somando em um vetor de inteiros a quantidade de verdadeiro para aquela cláusula. Caso não haja nenhuma, é inserido em cláusulas falsas e literais erradas. No flip verifica a mudança para aquele literal e trata em caso de verdadeiro apagando das cláusulas falsas (um set int) ou adicionando em caso de verdadeiro.

2. Desempenho comparado à solução de referência



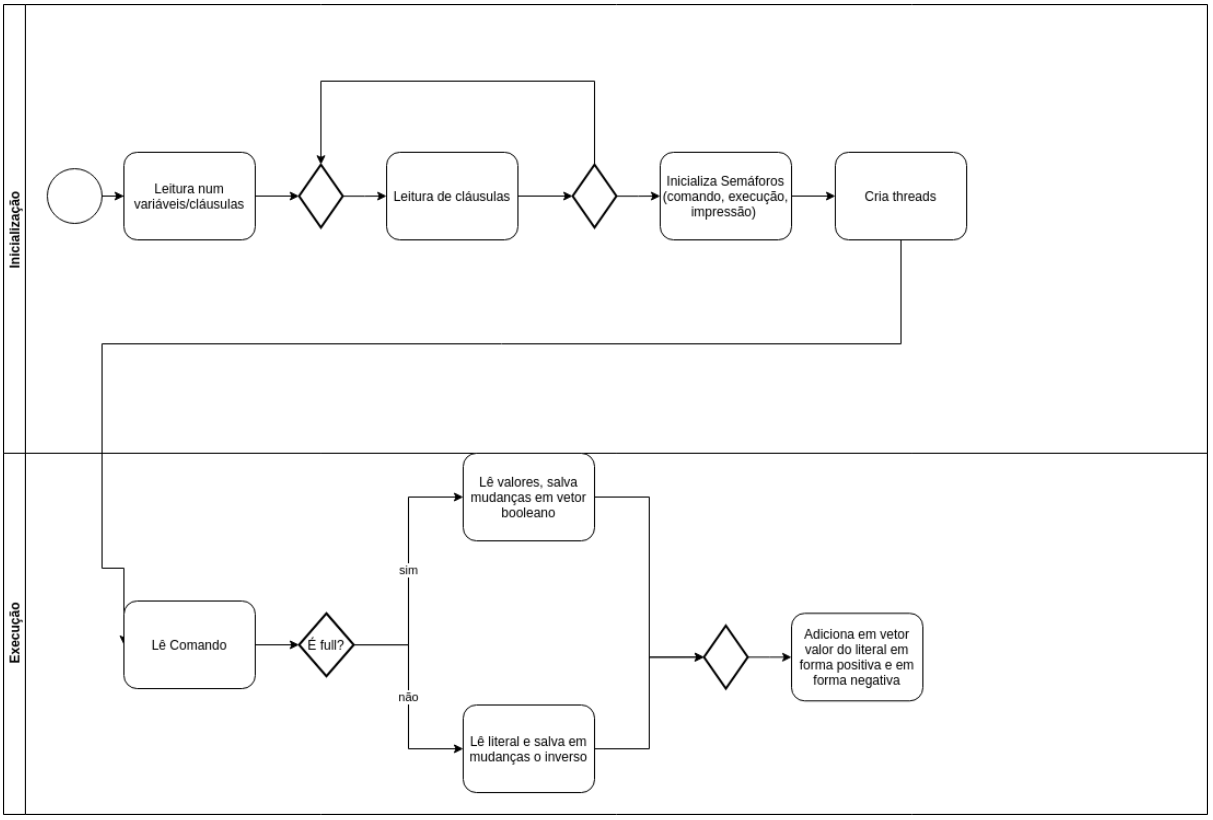
Percebe-se que a versão sequencial final com a mudança realizada apresentou desempenho melhor que a versão de referência para os arquivos de entrada testados, superando inclusive o uso do recurso de saída, que apresentava-se bastante oneroso na versão anterior.

Esta é apresentada como segunda versão por conter maiores mudanças, entretanto, até essa versão foram feitas diversas otimizações. Nesse ponto

estávamos preparados para paralelizar a solução e melhorar os tempos de execução considerando o tempo para entrega do algoritmo e o custo de tempo para testar casos de testes muito grandes, seja em número de entradas ou de cláusulas.

Análise algoritmo paralelizado

Abaixo segue o fluxo da primeira parte do algoritmo.



A partir do momento que o valor do literal é armazenado, as threads entram em ação paralelizando em cláusulas para cada comando e, através dos semáforos de controle, libera a entrega do resultado, onde este é ordenado e impresso finalizando o algoritmo.

1. Estratégia de implementação

Foi feita uma versão inicial de paralelização onde as threads foram inicializadas após cada comando, precisando serem iniciadas e finalizadas, o que gerava um custo enorme, ultrapassando os tempos gastos na versão sequencial. Então foram feitas otimizações, levando a inicialização das threads para o começo. Nesse

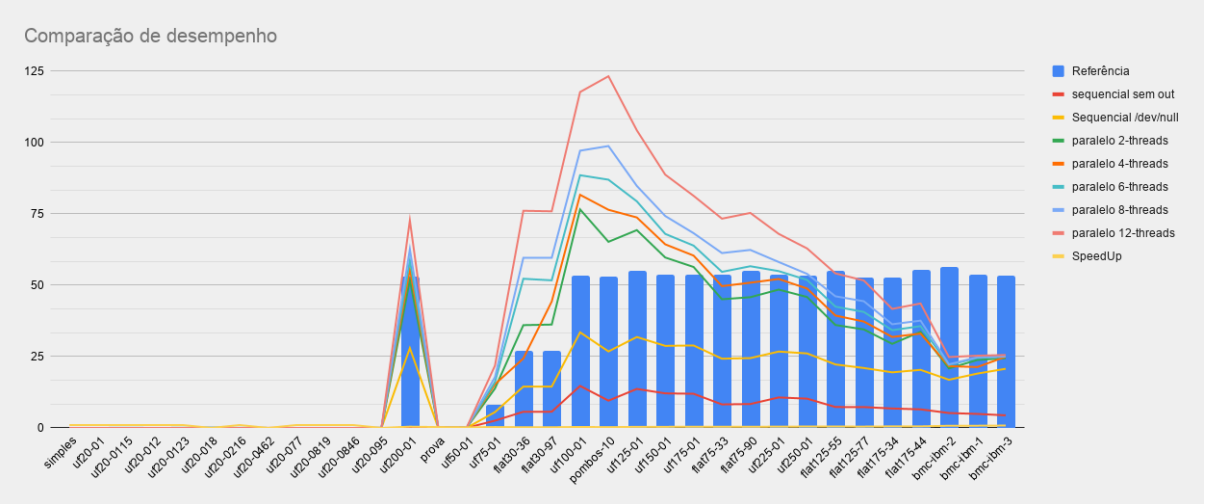
modelo os comandos são passados para elas e elas são controladas por semáforos e mutex para funcionarem adequadamente.

Procurou-se utilizar o mesmo algoritmo sequencial para a paralelização. Dessa forma, as estruturas permaneceram iguais, a propagação de valores também, entretanto, todo o tratamento dos comandos passou a ser feito por threads controlados por meio de mutex e semáforos.

Assim, o flip recebe o literal que deseja alterar e altera nas mudanças o correspondente para aquele flip. Assim, é feito um loop nos vetores de valores positivos e negativos e para cada um verifica-se a mudança tornou um cláusula verdadeira ou falsa incrementando nas estruturas de controle. A manutenção dessa estrutura guardando as cláusulas em que cada literal se encontra foi primordial para melhorar o desempenho do flip, pois possibilitou o tratamento exclusivo delas, deixando de ser necessário percorrer todas as cláusulas. No full são varridos todos os valores da cláusula e verificados se corresponde ao valor das mudanças. Essas comparações servem de insumo para a contagem de literais e cláusulas certas ou erradas.

Utiliza-se um índice para controle das threads para reduzir a diferença do custo entre a execução de cada comando. Tal que as threads ficam livres para acessar até que chegue ao número total de cláusulas, ao contrário da abordagem testada em que as cláusulas eram divididas em um número fixo de threads. Entretanto, para regulá-lo é usado um mutex que aumenta o custo da operação.

2. Desempenho comparado às demais soluções e solução de referência e gargalos

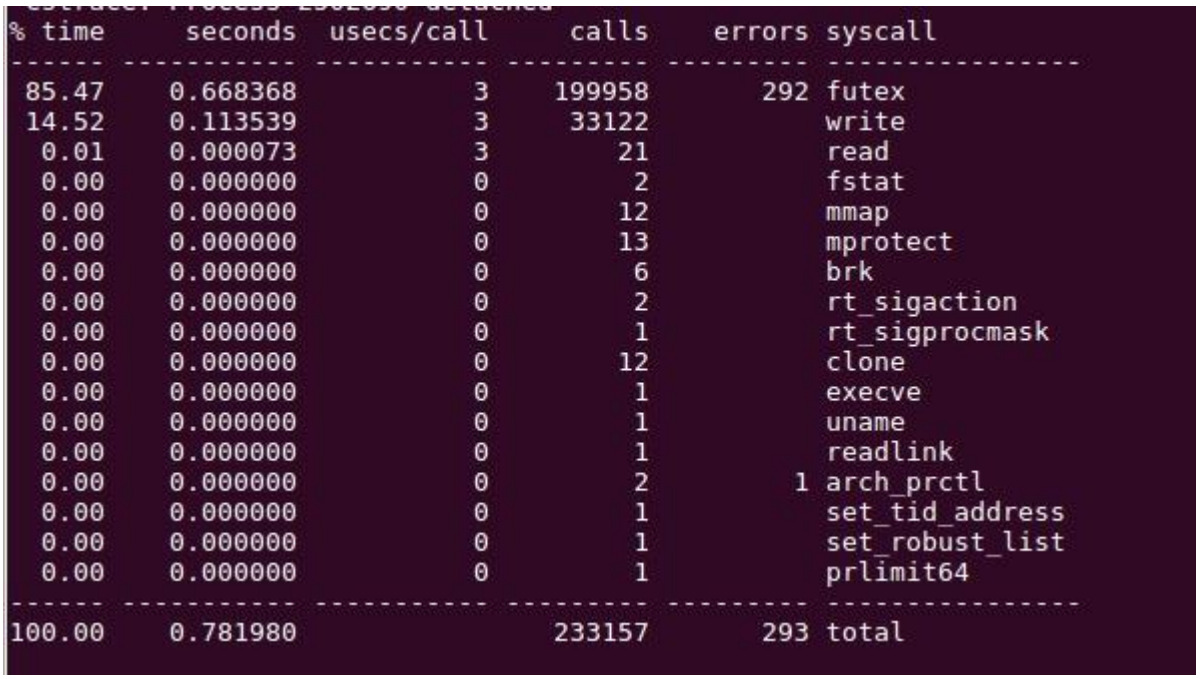


O gráfico gerado demonstra que a versão sequencial sem a impressão das saídas tem o melhor desempenho dentre os demais. Ao contrário do que se esperava, a abordagem utilizada para a paralelização não melhorou com o aumento das threads, pelo contrário, quanto maior o número de threads, maior o tempo de execução. O motivo de não ter melhorado se deve principalmente às chamadas blocantes utilizadas na função flip com dependência uma da outra. Isso impede que as threads executem em paralelo sem a ação do mutex que a cada chamada paralisa a execução. Apesar da utilização do índice para controlar a cláusula e otimizar a execução pelas threads, o ganho pensado apenas ocorre melhor o comando full que pode ter variações no tamanho de cada cláusula.

Assim, para melhorar esse gargalo teria que ser feita uma melhor otimização do flip voltada à paralelização, visto que no caso dele uma thread estará sempre esperando a outra concluir para executar.

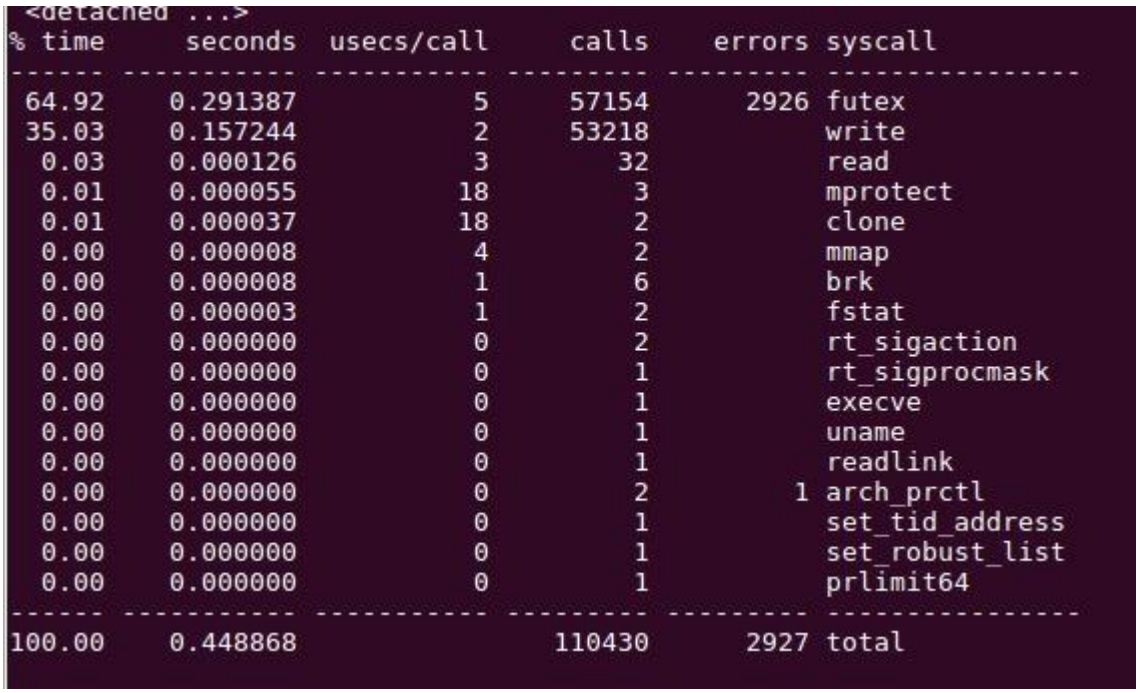
É interessante apontar que a partir 2810 variáveis todas as abordagens tendem para tempos parecidos e a partir de 175 alcançam os tempos do algoritmo de referência. Isso porque a paralelização realizada tende a ter melhor desempenho quando existem números maiores de operações full e menores de operações flip. Além disso, deve-se considerar na equação o número de cláusulas de cada caso a ser tratado que influencia diretamente na propagação dos valores e eficiência da estrutura de dados utilizada, visto que o algoritmo foi paralelizado em termos de cláusulas.

Com o strace, podemos obter, de modo específico, para o exemplo flat175-34 com 525 utilizando 12 threads o seguinte cenário:



% time	seconds	usecs/call	calls	errors	syscall
85.47	0.668368	3	199958	292	futex
14.52	0.113539	3	33122		write
0.01	0.000073	3	21		read
0.00	0.000000	0	2		fstat
0.00	0.000000	0	12		mmap
0.00	0.000000	0	13		mprotect
0.00	0.000000	0	6		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	12		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		uname
0.00	0.000000	0	1		readlink
0.00	0.000000	0	2	1	arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
0.00	0.000000	0	1		prlimit64
100.00	0.781980		233157	293	total

Analisando a imagem percebemos a explosão de chamadas de sistema futex, responsáveis pelo controle dos mutexes no código que inclusive ocasionaram 292 erros. Um dos motivos para esse alto número de chamadas, foi a criação de um índice global para controle das threads, com o intuito de fazer com que todas tivessem um tempo de execução semelhante. Eles fazem com que o código se aproxime de uma versão sequencial. Segue também como segundo culpado da baixa performance do algoritmo as chamadas writes, responsável pela manipulação dos registradores na memória.



% time	seconds	usecs/call	calls	errors	syscall
64.92	0.291387	5	57154	2926	futex
35.03	0.157244	2	53218		write
0.03	0.000126	3	32		read
0.01	0.000055	18	3		mprotect
0.01	0.000037	18	2		clone
0.00	0.000008	4	2		mmap
0.00	0.000008	1	6		brk
0.00	0.000003	1	2		fstat
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		uname
0.00	0.000000	0	1		readlink
0.00	0.000000	0	2	1	arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
0.00	0.000000	0	1		prlimit64
100.00	0.448868		110430	2927	total

O mesmo cenário ocorre com menos threads, evidenciando a necessidade de redução deste número de mutexes no código.

Além disso, o alto custo das leituras se mantém, conforme pode ser visto no gráfico abaixo, ultrapassando todas as variações dos algoritmos (sequencial).