Relatório Somativa 02

Para este trabalho decidimos executar as versões na chococcino.

1. Introdução

1.1. Problema

Esse relatório é referente ao problema descrito aqui Boolean Constraint Propagator, um problema de Satisfatibilidade Booleana que tem como desafio a aplicação de paralelização.

1.2. Ambiente

O ambiente usado foi a máquina chococcino disponibilizada pelo professor para a execução dos testes. A linguagem utilizada nos códigos foi o **C++** e o comando utilizado para compilação dos arquivos foi esse:

- ***g++ -O2 -static -std=c++17 -W -Wall -Wshadow {nome_do_arquivo}.cpp***

1.3. Testes

Realizamos os testes com o input de 30. Criamos um arquivo de teste nomeado teste.cpp, no qual ele rodava todos os inputs e comparava com os outputs e jogava os tempos em uma tabela.

1.4. Códigos

Os códigos desenvolvidos ao decorrer do trabalho se encontram disponíveis no repositório no github.

2. Algoritmo “naive”

2.1. Abordagem

Na primeira versão utilizamos apenas força bruta, a cada flip ou full alteramos o valor das variáveis e passamos por cada cláusula verificando quais cláusulas ficaram falsas e qual a nova ordem das variáveis falsas.

2.2. Complexidade assintótica

Sendo **V** o número de variáveis e **C** o número de cláusulas.

A complexidade do comando **full** pode ser calculada da seguinte forma:
C (Alteração das variáveis) +
C * V (Cálculo dos literais falsos) +
C + V + V*log (V) (Impressão e ordenação dos literais)

C + C*V + C + V + V*log (V)
~= C*V + V * log(V)

A complexidade do comando **flip** pode ser calculada da seguinte forma:
O(1) (Alteração da variável) +
C * V (Cálculo dos literais falsos) +
C + V + V*log (V) (Impressão e ordenação dos literais)

C*V + C + V + V*log (V)
~= C*V + V * log(V)

No total a complexidade é a soma do número de comandos, assumindo o número de comandos como T a complexidade total é

T*(C*V + V * log(V))

2.3. Paralelização

A paralelização dessa versão foi feita separando as cláusulas a serem calculadas para as threads disponíveis. O cálculo foi feito separando blocos a serem calculados e distribuindo esses blocos para as threads disponíveis.

O principal gargalo dessa versão é na gravação das variáveis e cláusulas falsas, como foi utilizado semáforo, mesmo os processos estando paralelos, eles tinham que esperar outros processos gravarem variáveis e cláusulas falsas para realizar gravações em memória.

2.4. Tabelas de tempo

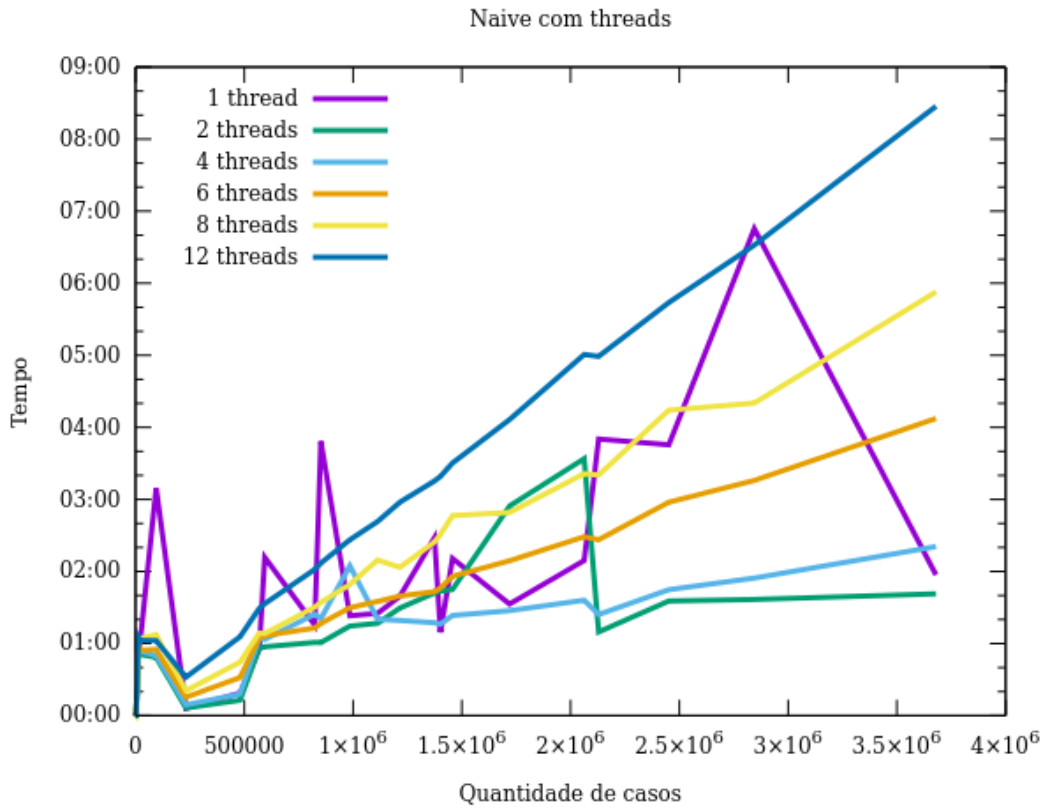
2.4.1. Versão sem threads

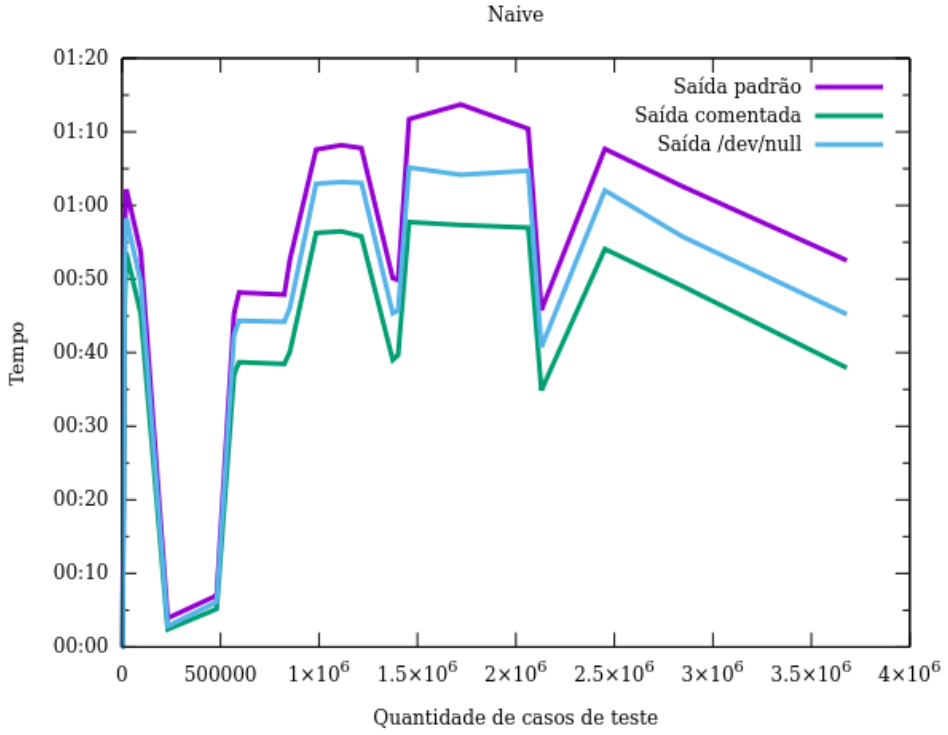
Os demais gráficos, estão no repositório no github em formato csv.

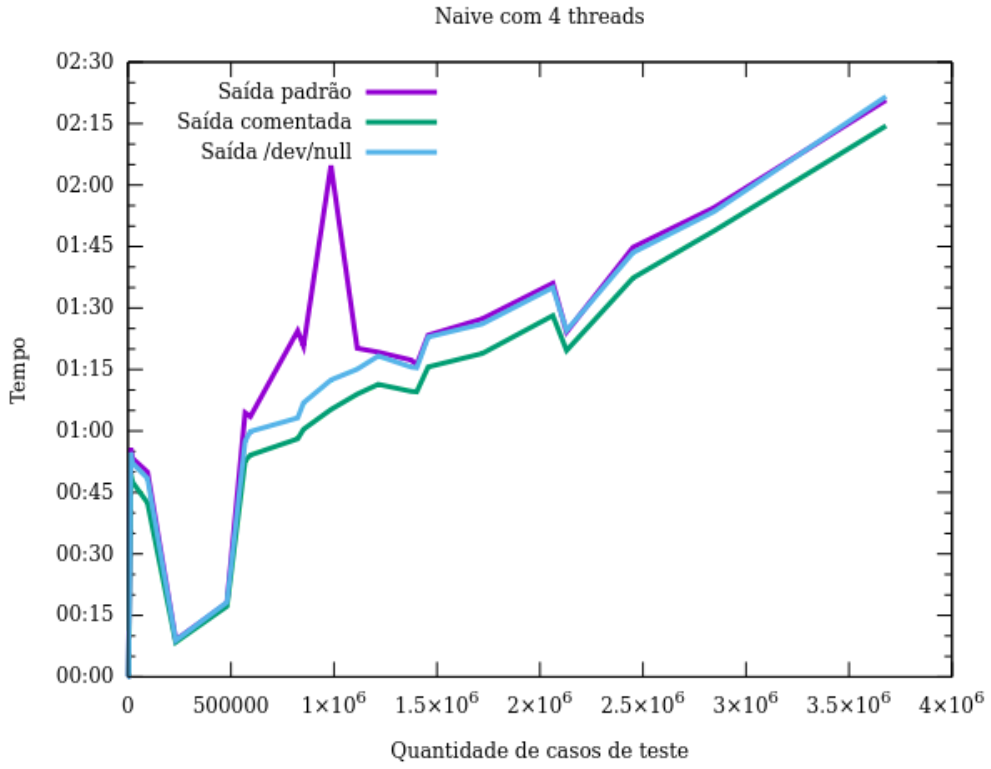
Nome arquivo	Casos de Teste	Variáveis	Cláusulas	Tempo Real	Tempo User	Tempo Sys
uf125-01	2062434	125	538	70.42	67.66	1.97
uf200-01	1215206	200	860	67.8	65.96	1.39
uf20-077	116	20	91	0	0	0
flat175-44	594166	525	1951	48.17	46.64	0.9
flat125-77	823697	375	1403	47.88	46.38	01.04
flat175-34	568693	525	1951	45.32	44.36	0.88
uf20-0115	84	20	91	0	0	0
simples	3	2	3	0	0	0
prova	2539	50	268	0.04	0.04	0
flat75-33	1374335	225	840	50.12	47.73	1.45
uf20-0819	77	20	91	0	0	0
uf20-01	10	20	91	0	0	0
flat30-97	3679939	90	300	52.53	47.89	3.89
pombos-10	2844584	110	561	62.6	59.14	03.05
uf20-0216	42	20	91	0	0	0
uf50-01	480481	50	218	6.99	6.58	0.41
uf20-012	146	20	91	0	0	0
uf75-01	2129068	75	325	45.8	43.01	2.21
uf20-095	24	20	91	0	0	0
uf225-01	1112791	225	960	68.19	66.09	1.36
uf175-01	1457154	175	753	71.73	68.66	1.56
uf20-0123	121	20	91	0	0	0

uf20-018	1290	20	91	0	0	0
flat75-90	1401674	225	840	49.86	48.02	1.59
bmc-ibm-3	13467	14930	72106	56.26	55.35	0.53
uf100-01	2451602	100	430	67.69	64.43	2.94
uf20-0846	7	20	91	0	0	0
bmc-ibm-1	21731	9685	55870	62.2	61.13	0.47
uf250-01	985382	250	1065	67.6	65.83	1.27
bmc-ibm-2	94940	2810	11683	53.71	53.1	0.54
uf20-0462	1621	20	91	0.01	0.01	0
flat125-55	852061	375	1403	52.63	50.75	1.22
flat30-36	231772	90	300	3.91	3.57	0.3
uf150-01	1719560	150	645	73.73	71.17	2.18

2.4.2. Gráficos




Naive
Saída padrão
Saída comentada
Saída /dev/null
Tempo
01:20
01:10
01:00
00:50
00:40
00:30
00:20
00:10
00:00
0
500000
1×10^6
1.5×10^6
2×10^6
2.5×10^6
3×10^6
3.5×10^6
4×10^6
Quantidade de casos de teste


Naive com 4 threads
Saída padrão
Saída comentada
Saída /dev/null
Tempo
02:30
02:15
02:00
01:45
01:30
01:15
01:00
00:45
00:30
00:15
00:00
0
500000
1×10^6
1.5×10^6
2×10^6
2.5×10^6
3×10^6
3.5×10^6
4×10^6
Quantidade de casos de teste

3. Algoritmo com memorização

3.1. Abordagem

A segunda abordagem foi utilizar a programação dinâmica para memorizar processos já realizados de maneira inteligente. Nessa abordagem foi memorizado o número de variáveis falsas em cada cláusula, dessa maneira só processamos as cláusulas que ficaram falsas ou que ficaram verdadeiras, evitando processamento a mais de cláusulas.

3.2. Complexidade assintótica

Sendo **V** o número de variáveis e **C** o número de cláusulas. A complexidade do comando **full** pode ser calculada da seguinte forma:
C (Alteração das variáveis) +
C*V (Cálculo dos literais falsos) +
C + V + V*log(V) (Impressão e ordenação dos literais)

C + C*V + C + V + V*log(V)
~= C*V + V*log(V)

A complexidade do comando **flip** pode ser calculada da seguinte forma:
O(1) (Alteração da variável) +
C*(V + log(V)) (Cálculo dos literais falsos) +
C + V + V*log (V) (Impressão e ordenação dos literais)

C * log(V) + C + V + V * log(V)
~= C*(log(V) + V) + V*log(V)

Essa abordagem tem uma melhora significativa em relação a primeira no cálculo do flip, diminuindo a complexidade de C*V para C*log(V) no melhor caso, onde nenhuma cláusula foi alterada.
No total a complexidade é a soma do número de comandos, assumindo o número de comandos como T a complexidade total é

T*(C*V + V*log(V))

A complexidade assintótica não mudou, mas é possível ver uma melhora no tempo pelo fato do comando flip estar mais eficiente.

3.3. Otimização

Nessa versão foi percebido que havia um gargalo na hora de procurar as variáveis no comando flip deixando um passo linear na complexidade assintótica (3.2), visto isso, foi utilizado memorização para saber exatamente em quais cláusulas cada variável estava, deixando esse passo linear no número de cláusulas que a variável do flip se encontrava.

3.3.1. Complexidade assintótica

A complexidade assintótica muda para o comando flip da seguinte maneira, sendo **K** o número cláusulas que a variável está, tal que **0 <= K <= C**:
O(1) (Alteração da variável) +
K*V (Cálculo dos literais falsos) +
C + V + V*log (V) (Impressão e ordenação dos literais)

K*V + C + V + V*log(V)
~= C + K*V + V*log(V)

A complexidade assintótica é a mesma pois K é igual a C no pior caso, mas em geral o tempo foi bem menor.

3.4. Paralelização

A paralelização dessa versão foi feita no número de cláusulas a serem processadas, dividindo as cláusulas entre as threads. Ao final do processo as variáveis são atualizadas com auxílio de semáforos. O ganho de desempenho é percebido em testes com poucos casos de teste e muitas cláusulas. À medida que esses valores se invertem o tempo aumenta bastante, por conta da constante do código paralelizado ser maior, e por conta da complexidade na utilização de semáforos e criação de threads. Em geral essa versão é bem performada em comandos com muitas cláusulas a serem processadas.

3.5. Tabelas com Tempo

3.5.1. Versão original

Nome arquivo	Casos de Teste	Variáveis	Cláusulas	Tempo Real	Tempo User	Tempo Sys
simples	3	2	3	0	0	0
uf20-0846	7	20	91	0	0	0

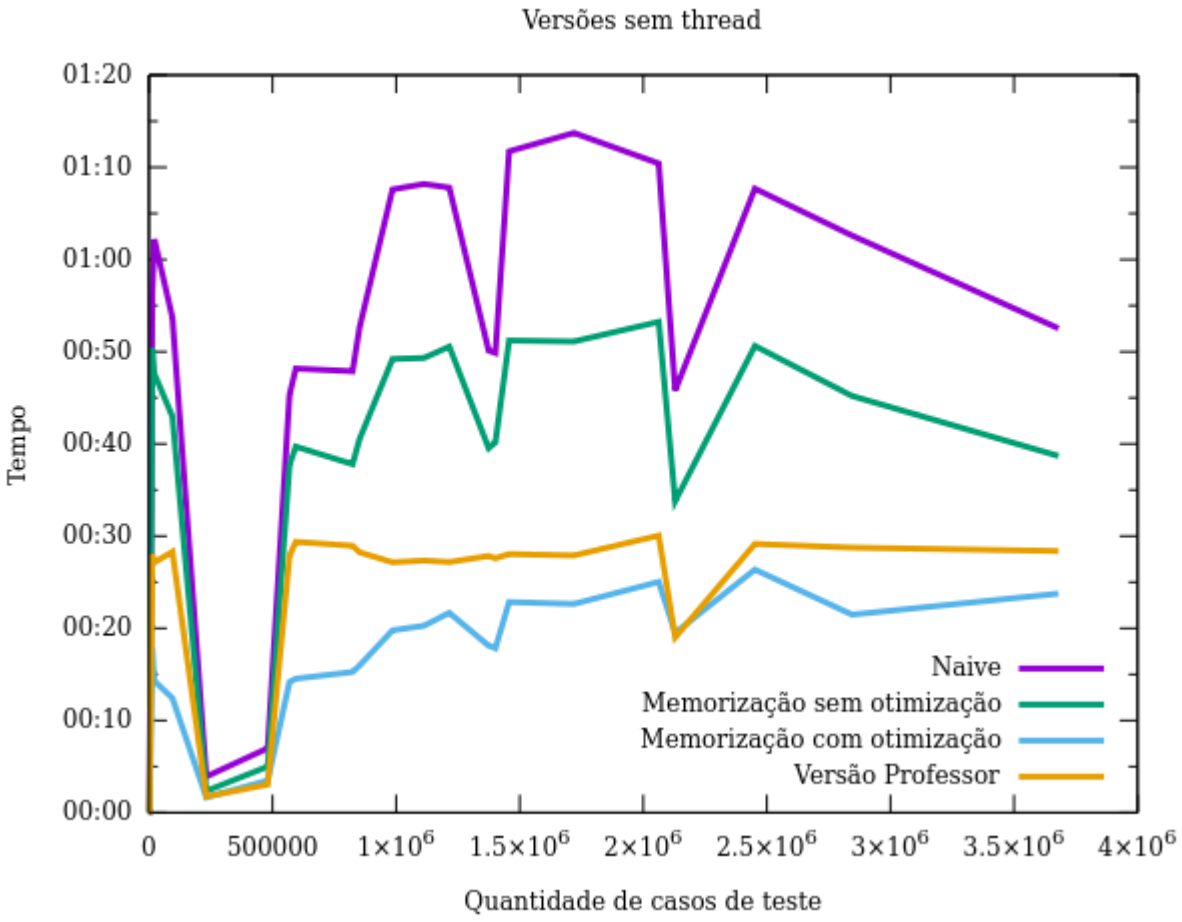
uf20-01	10	20	91	0	0	0
uf20-095	24	20	91	0	0	0
uf20-0216	42	20	91	0	0	0
uf20-0819	77	20	91	0	0	0
uf20-0115	84	20	91	0	0	0
uf20-077	116	20	91	0	0	0
uf20-0123	121	20	91	0	0	0
uf20-012	146	20	91	0	0	0
uf20-018	1290	20	91	0	0	0
uf20-0462	1621	20	91	0	0	0
prova	2539	50	268	0.03	0.03	0
bmc-ibm-3	13467	14930	72106	50.34	49.09	0.61
bmc-ibm-1	21731	9685	55870	47.61	46.8	0.45
bmc-ibm-2	94940	2810	11683	42.96	42.44	0.43
flat30-36	231772	90	300	2.35	2.33	0.01
uf50-01	480481	50	218	5	4.97	0.03
flat175-34	568693	525	1951	37.73	36.76	0.47
flat175-44	594166	525	1951	39.71	38.34	0.99
flat125-77	823697	375	1403	37.8	36.83	0.38
flat125-55	852061	375	1403	40.53	39.23	0.57
uf250-01	985382	250	1065	49.21	48.32	0.46
uf225-01	1112791	225	960	49.32	48.86	0.33
uf200-01	1215206	200	860	50.55	49.66	0.43
flat75-33	1374335	225	840	39.54	38.65	0.51
flat75-90	1401674	225	840	40.16	38.98	0.51
uf175-01	1457154	175	753	51.21	50.4	0.49
uf150-01	1719560	150	645	51.11	50.47	0.36
uf125-01	2062434	125	538	53.24	52.43	0.47
uf75-01	2129068	75	325	33.8	33.13	0.33
uf100-01	2451602	100	430	50.63	49.72	0.54
pombos-10	2844584	110	561	45.2	44.38	0.41
flat30-97	3679939	90	300	38.69	38.05	0.43

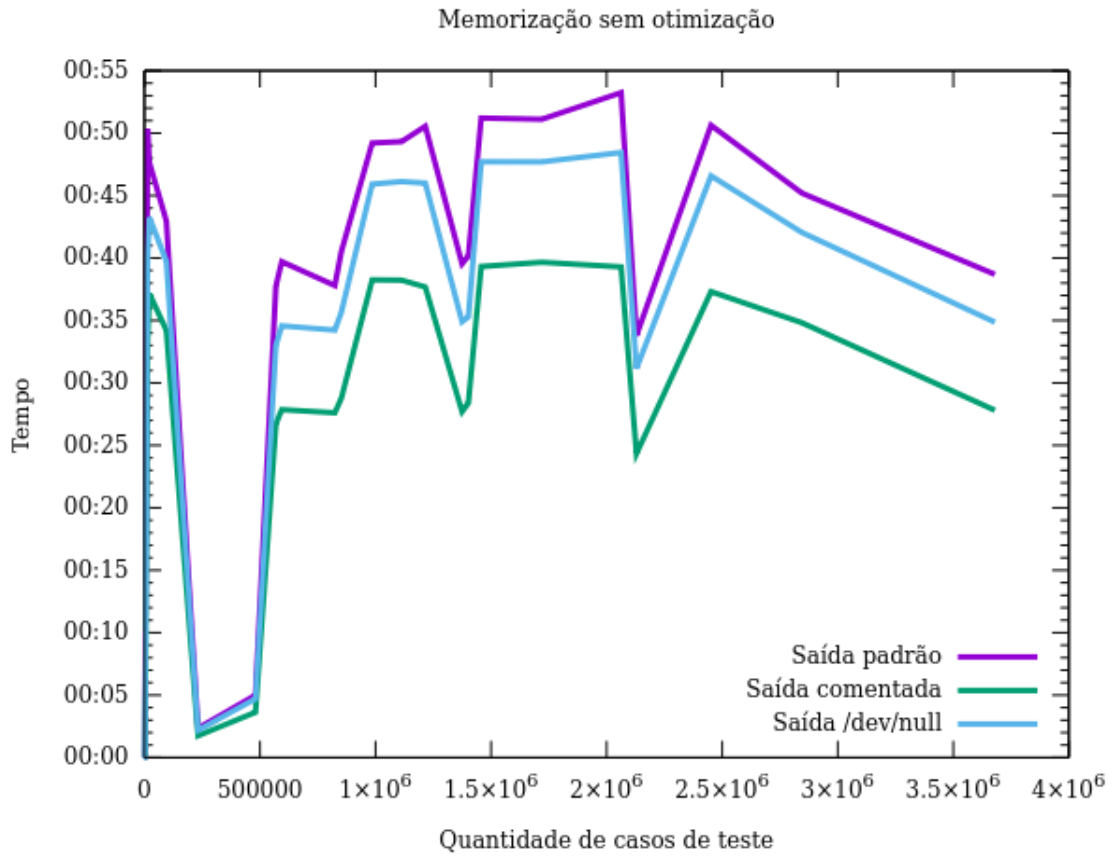
3.5.2. Versão otimizada

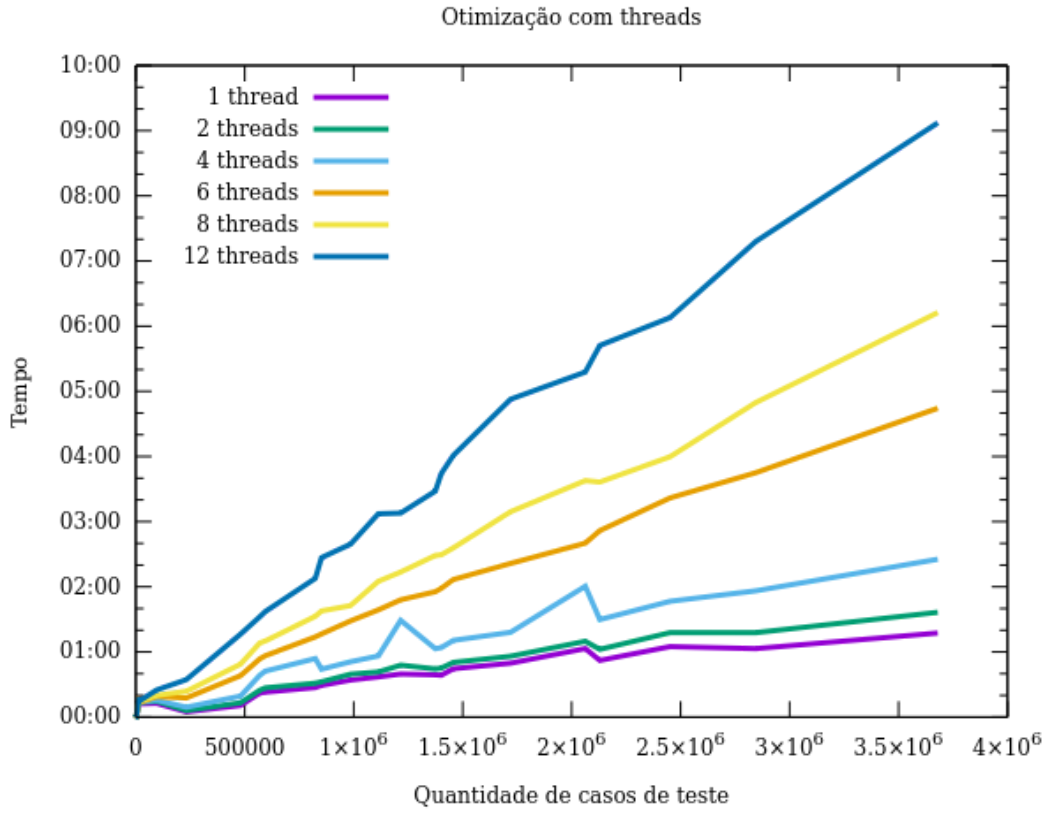
Nome	Casos de	Variáveis	Cláusulas	Tempo Real	Tempo User	Tempo Sys

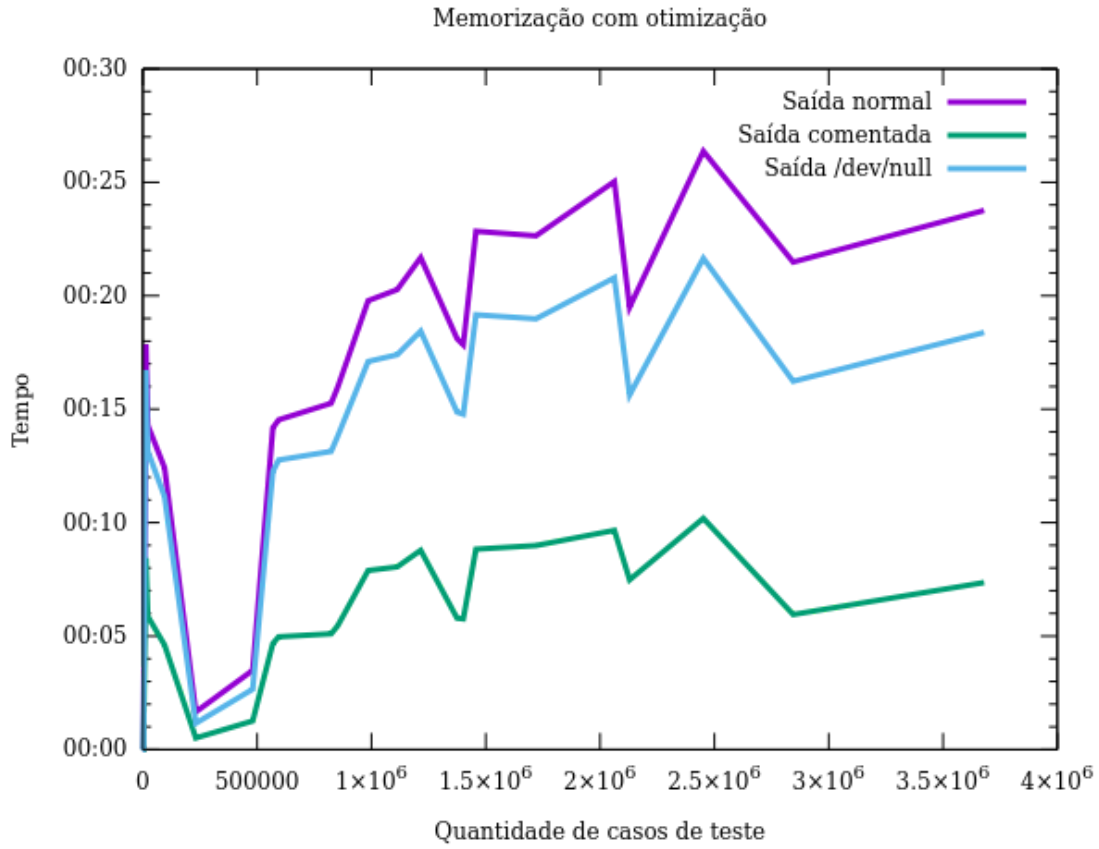
arquivo	Teste					
simples	3	2	3	0	0	0
uf20-0846	7	20	91	0	0	0
uf20-01	10	20	91	0	0	0
uf20-095	24	20	91	0	0	0
uf20-0216	42	20	91	0	0	0
uf20-0819	77	20	91	0	0	0
uf20-0115	84	20	91	0	0	0
uf20-077	116	20	91	0	0	0
uf20-0123	121	20	91	0	0	0
uf20-012	146	20	91	0	0	0
uf20-018	1290	20	91	0	0	0
uf20-0462	1621	20	91	0	0	0
prova	2539	50	268	0.02	0.01	0
bmc-ibm-3	13467	14930	72106	17.84	17.3	0.54
bmc-ibm-1	21731	9685	55870	14.29	13.77	0.51
bmc-ibm-2	94940	2810	11683	12.42	11.95	0.46
flat30-36	231772	90	300	1.66	1.11	0.44
uf50-01	480481	50	218	3.49	2.66	0.83
flat175-34	568693	525	1951	14.18	12.76	1.27
flat175-44	594166	525	1951	14.53	13.25	1.27
flat125-77	823697	375	1403	15.26	13.54	1.71
flat125-55	852061	375	1403	15.96	14.29	1.66
uf250-01	985382	250	1065	19.78	17.68	1.95
uf225-01	1112791	225	960	20.27	18.15	02.01
uf200-01	1215206	200	860	21.68	19.22	2.2
flat75-33	1374335	225	840	18.11	15.6	2.27
flat75-90	1401674	225	840	17.84	15.22	2.55
uf175-01	1457154	175	753	22.84	19.76	2.73
uf150-01	1719560	150	645	22.64	19.63	3
uf125-01	2062434	125	538	25.03	21.45	3.58
uf75-01	2129068	75	325	19.5	16.07	3.42
uf100-01	2451602	100	430	26.37	22.28	04.09
pombos-10	2844584	110	561	21.48	16.53	4.95
flat30-97	3679939	90	300	23.76	17.9	5.85

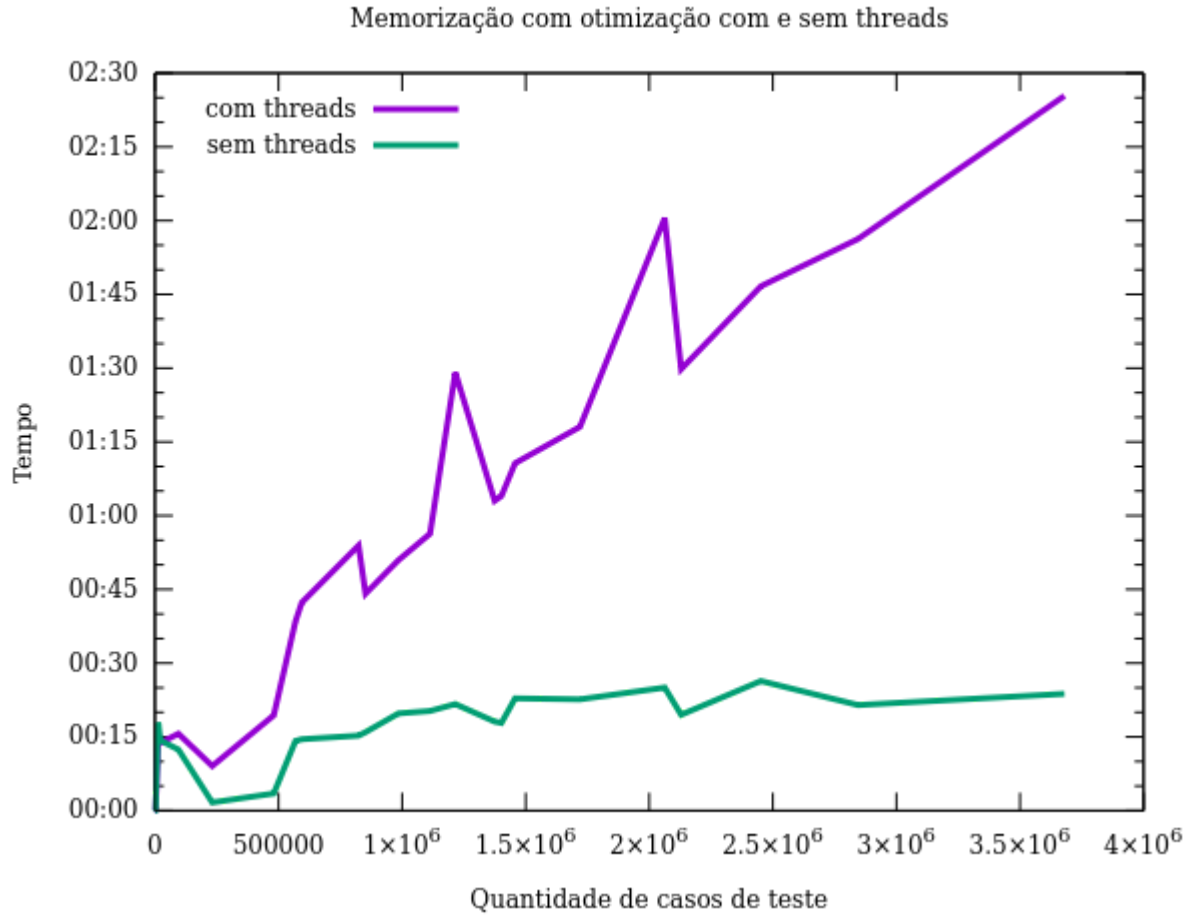
4. Gráficos comparativos

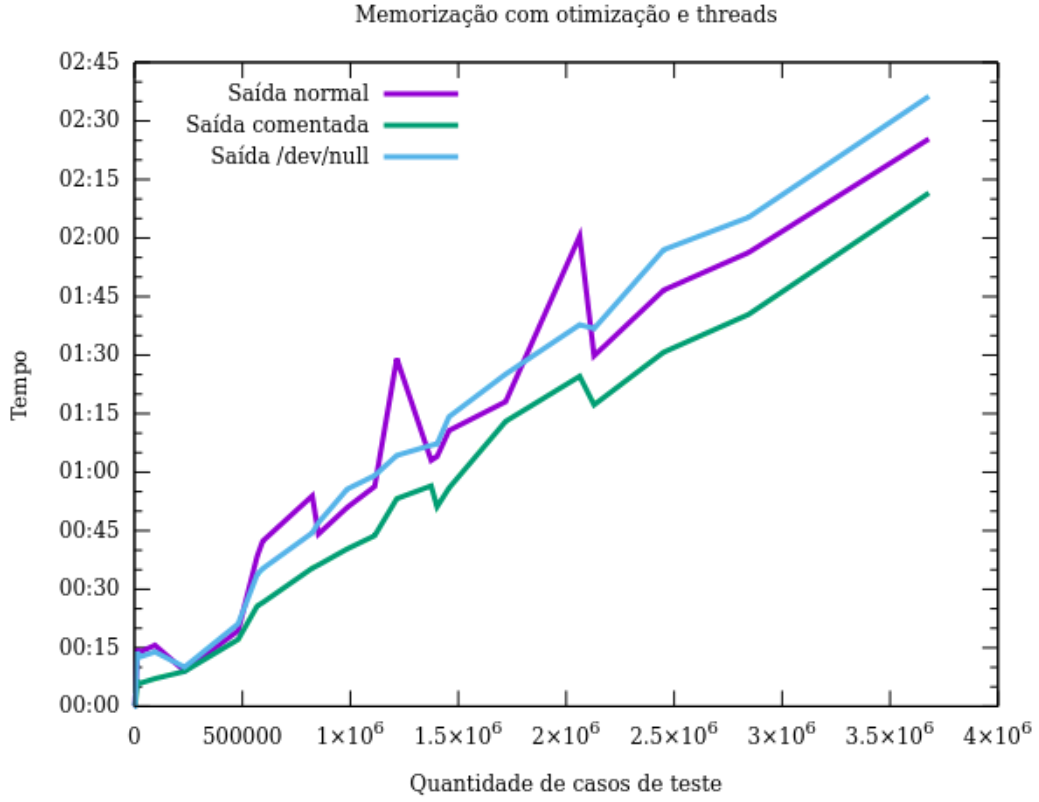



Memorização sem otimização
Tempo
00:55
00:50
00:45
00:40
00:35
00:30
00:25
00:20
00:15
00:10
00:05
00:00
0
500000
1×10^6
1.5×10^6
2×10^6
2.5×10^6
3×10^6
3.5×10^6
4×10^6
Quantidade de casos de teste
Saída padrão
Saída comentada
Saída /dev/null


Otimização com threads
Tempo
10:00
09:00
08:00
07:00
06:00
05:00
04:00
03:00
02:00
01:00
00:00
0
500000
1×10^6
1.5×10^6
2×10^6
2.5×10^6
3×10^6
3.5×10^6
4×10^6
Quantidade de casos de teste
1 thread
2 threads
4 threads
6 threads
8 threads
12 threads

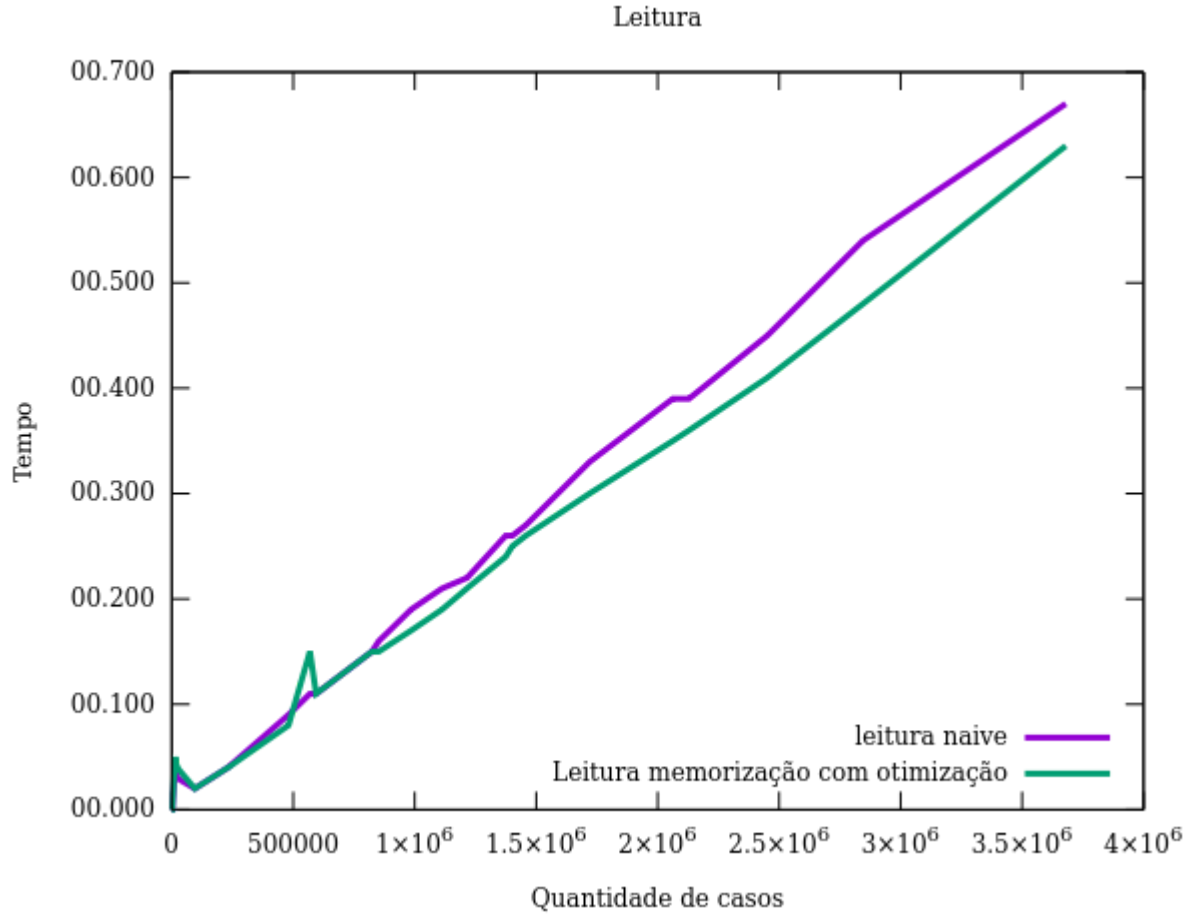

Memorização com otimização
Saída normal
Saída comentada
Saída /dev/null
Tempo
00:30
00:25
00:20
00:15
00:10
00:05
00:00
0
500000
1×10^6
1.5×10^6
2×10^6
2.5×10^6
3×10^6
3.5×10^6
4×10^6
Quantidade de casos de teste

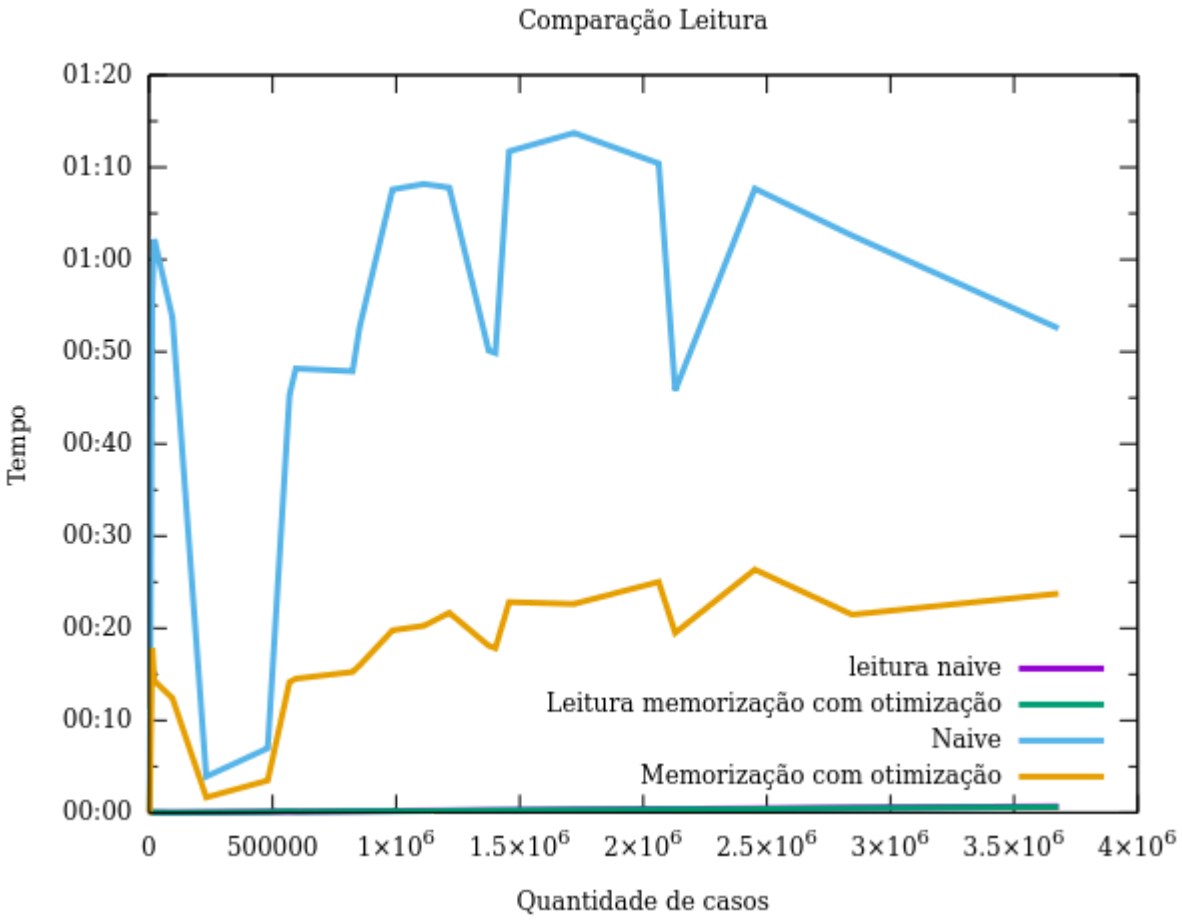

Memorização com otimização com e sem threads
com threads
sem threads
Tempo
02:30
02:15
02:00
01:45
01:30
01:15
01:00
00:45
00:30
00:15
00:00
0
500000
1×10^6
1.5×10^6
2×10^6
2.5×10^6
3×10^6
3.5×10^6
4×10^6
Quantidade de casos de teste


Memorização com otimização e threads
Saída normal
Saída comentada
Saída /dev/null
Tempo
02:45
02:30
02:15
02:00
01:45
01:30
01:15
01:00
00:45
00:30
00:15
00:00
0
500000
1×10^6
1.5×10^6
2×10^6
2.5×10^6
3×10^6
3.5×10^6
4×10^6
Quantidade de casos de teste

5. Tempo de leitura

Foi percebido que os tempos de leitura eram negligenciáveis. Observe os gráfico abaixo. Perceba que as leituras não chegam a 1 segundo.





6. Conclusão

Apesar de parecer 100% superior a versão com memorização utiliza bastante memória, em geral é um gasto aceitável em troca do desempenho ganho em tempo de execução, mas é importante saber das limitações desta versão e entender em que contexto deve ou não ser usada.

Foi notado que a utilização de threads nem sempre trás um ganho significativo de desempenho, existem gargalos específicos em cada implementação que devem ser identificados e processos que são mais trabalhosos são candidatos a paralelização. Foi identificado que o uso de semáforos e criação de threads é um processo demorado no código e pode vir a ser um gargalo em testes com muitos comandos.