Lucas Mayr de Athayde
Luiz João C. Motta

INE5410-03208B (20172) - Programação Concorrente

SudokuN

UFSC - Florianópolis

2017

Introdução

Um Sudoku consiste de um quebra-cabeça que consiste de uma matriz de números, sendo ela de tamanho 9x9 e com os possíveis números de 1 a 9. Um Sudoku já vem com dicas preenchidas, sendo que um Sudoku válido possui apenas uma solução. Este quebra-cabeça possui algumas regras, sendo elas:

- Presença de regiões, sendo elas de tamanho 3 x 3 e ao total há 3^2 regiões. Em cada região não é possível ter mais de um número repetido.
- Há 3^4 células, cada número é posto em uma célula, sendo que na linha e na coluna em que essa célula está presente, não é possível ter mais de um número repetido.

Exemplo: em verde uma região, em azul uma linha e em vermelho uma coluna.

Buscando uma generalização do Soduku, temos um SudokuN. Esta mesma generalização é de tamanho N^2xN^2 e possui: N^2 regiões de N x N e N^4 células e podendo conter números de 1 a N^2. Exemplificando: podemos criar um Sudoku4 (N = 4), de tamanho 16x16 possuindo 16 regiões de 4 x 4, com um total de 256 células e elas podendo ser numeradas de 1 a 16.

Problema

O trabalho proposto pela matéria de programação concorrente propõe a implentação de um programa em que possa encontrar todas as soluções possíveis para um SudokuN, sendo necessário a utilização de POSIX threads (Pthreads).

O programa terá com entrada o valor de N, sendo esse não superior a 8, a quantidade de threads em que o programa irá rodar e a matriz inicial do sudoku. Cada célula da matriz de entrada será separada por uma virgula (“,”) e linha separada por uma quebra de linha (“\n”). As células que estão em branco irá estar com o número 0.

Observação: O problema de um SudokuN se trata de um NP-completo, pois é necessário testar todas as possibilidades.

Solução

- ***Mono-thread***

Como no problema proposto é necessário que ocorra um teste de todas as possibilidades possíveis para se ter certeza da quantidade de soluções no quebra-cabeça, a solução mais intuitiva é simplesmente a implementação deste método, que é chamado *brute-force* utilizando *backtracking*.

A implementação deste problema gira em torno de uma recursão, o programa irá testar se um valor é possível numa célula, começando do número 1, realizando um teste de coluna, linha e região. Se o valor testado seja possível, o programa avança testando a próxima célula, caso contrário o programa irá limpar a célula, voltar e testar o próximo valor na célula anterior. Caso não tenha uma célula anterior para ser testada, o quebra-cabeça não possui soluções.

- ***Multi-thread***

A solução escolhida para o *multi-thread* consiste em utilizar o mesmo método escolhido no *mono-thread*, devido ao fácil entendimento e necessitando apenas a paralelização do *mono-thread*. Nesta implementação o programa cria a quantidade desejada de threads e começa tentar a resolver o SudokuN, caso a thread consiga terminar de preencher uma linha do SudokuN, essa solução parcial é colocada numa pilha e então outras threads a partir dessa solução parcial verificam outras possíveis soluções da próxima linha. Caso encontrem, o estado do sudoku é empilhado e a thread continua a resolver a mesma linha por *backtracking* até que nenhuma outra linha seja encontrada, quando uma thread não ache mais nenhuma linha possível ela libera o seu Sudoku e pega outro da pilha, repetindo o processo.

O programa irá finalizar quando não houverem mais sudokus na pilha e todas as threads estiverem aguardando por um estado.

Resultados

- ***Mono-thread***

O algoritmo feito para o *mono-thread* respondeu de forma satisfatória para encontrar a quantidade de soluções de um Sudoku3, já para o Sudoku4 passado de exemplo no enunciado do problema demorou uma grande quantidade de tempo. Sudokus de ordem maior não foram testados.

a) Sudoku N = 3:

```
→ base git:(master) ✗ time ./sudoku < ../inputs/input9_more.in
Solutions: 658./sudoku < ../inputs/input9_more.in  0,20s user 0,00s system 99% cpu 0,197 total
→ base git:(master) ✗ time ./sudoku < ../inputs/input9.in
Solutions: 1./sudoku < ../inputs/input9.in  0,00s user 0,00s system 63% cpu 0,013 total
```

(Sudoku com 658 soluções: 0,197 segundos; Sudoku com 1 solução: 0,013 segundos).

Podemo ver a grande diferença de um Sudoku que possui apenas uma solução de outro que possui mais solução já com N = 3.

b) Sudoku N = 4:

```
→ base git:(master) ✗ time ./sudoku < ../inputs/input16_other.in
Solutions: 3./sudoku < ../inputs/input16_other.in  38,93s user 0,03s system 99% cpu 38,967 total
```

(Sudoku com 3 soluções: 38.967 segundos)

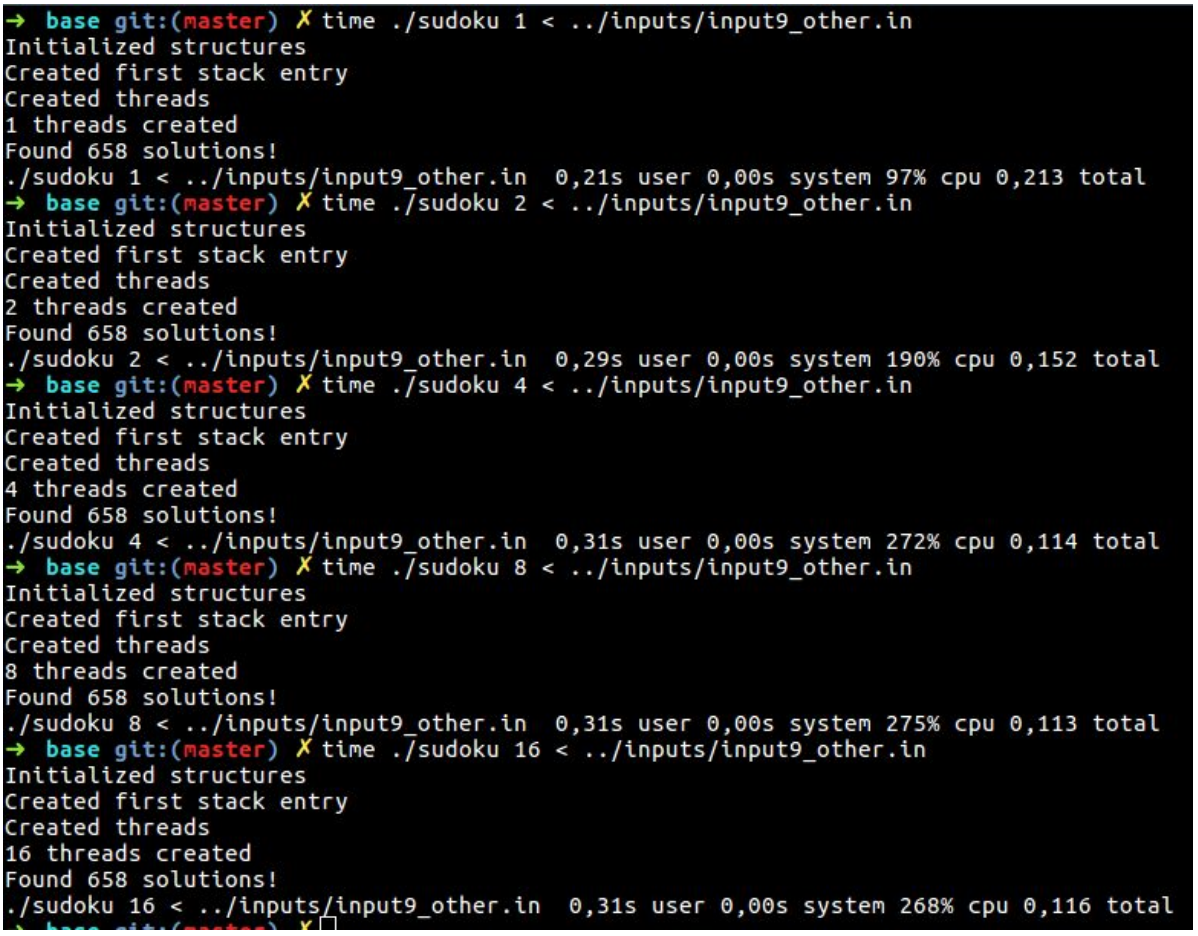
Com o aumento de N, temos um grande aumento no tempo para a conclusão do programa mono-thread.

- ***Multi-thread***

Assim como o *mono-thread* o algoritmo conseguiu resolver sem problemas os Sudokus de ordem N = 3, mas teve problemas em encontrar a quantidade de soluções para Sudokus com grandes possibilidades ao longo de sua execução.

Uma atenção especial ao ‘grandes possibilidades’, pois como se trata de um algoritmo de força bruta, no Sudoku4 dado como exemplo no enunciado da proposta, em uma parte da execução, há uma enorme quantidade de possibilidades, sendo assim, necessário uma grande quantidade de memória para armazenar todas essas possibilidades na pilha, acarretando alguns problemas de falta de memória.

a) Sudoku N = 3



(Sudoku com 658 soluções em 1, 2, 4, 8 e 16 threads)

Podemos observar uma diminuição do tempo de execução com o aumento do número de threads até um número em que um aumento de threads começa a prejudicar o andamento do algoritmo.

A melhor execução registrada foi com 0.113 segundos e com 8 threads.

```
→ base git:(master) ✗ time ./sudoku 1 < ../inputs/input9.in
Initialized structures
Created first stack entry
Created threads
1 threads created
Found 1 solutions!
./sudoku 1 < ../inputs/input9.in  0,01s user 0,00s system 84% cpu 0,014 total
→ base git:(master) ✗ time ./sudoku 2 < ../inputs/input9.in
Initialized structures
Created first stack entry
Created threads
2 threads created
Found 1 solutions!
./sudoku 2 < ../inputs/input9.in  0,01s user 0,00s system 149% cpu 0,008 total
→ base git:(master) ✗ time ./sudoku 4 < ../inputs/input9.in
Initialized structures
Created first stack entry
Created threads
4 threads created
Found 1 solutions!
./sudoku 4 < ../inputs/input9.in  0,02s user 0,00s system 223% cpu 0,007 total
→ base git:(master) ✗ time ./sudoku 8 < ../inputs/input9.in
Initialized structures
Created first stack entry
Created threads
8 threads created
Found 1 solutions!
./sudoku 8 < ../inputs/input9.in  0,02s user 0,00s system 190% cpu 0,008 total
→ base git:(master) ✗ time ./sudoku 16 < ../inputs/input9.in
Initialized structures
Created first stack entry
Created threads
16 threads created
Found 1 solutions!
./sudoku 16 < ../inputs/input9.in  0,02s user 0,00s system 180% cpu 0,009 total
```

(Sudoku 1 solução em 1, 2, 4, 8, 16 threads)

Assim como o Sudoku de 658 soluções houve um ganho de desempenho com o aumento de threads, mas foi alcançado uma melhor eficiência com menos threads.

A melhor execução registrada foi com 0.007 segundos e com 4 threads.

b) Sudoku N = 4

```
→ base git:(master) ✗ time ./sudoku 1 < ../inputs/input16_other.in
Initialized structures
Created first stack entry
Created threads
1 threads created
Found 3 solutions!
./sudoku 1 < ../inputs/input16_other.in  44,31s user 0,03s system 99% cpu 44,494 total
→ base git:(master) ✗ time ./sudoku 2 < ../inputs/input16_other.in
Initialized structures
Created first stack entry
Created threads
2 threads created
Found 3 solutions!
./sudoku 2 < ../inputs/input16_other.in  61,14s user 0,09s system 195% cpu 31,395 total
→ base git:(master) ✗ time ./sudoku 4 < ../inputs/input16_other.in
Initialized structures
Created first stack entry
Created threads
4 threads created
Found 3 solutions!
./sudoku 4 < ../inputs/input16_other.in  60,89s user 0,15s system 244% cpu 24,919 total
→ base git:(master) ✗ time ./sudoku 8 < ../inputs/input16_other.in
Initialized structures
Created first stack entry
Created threads
8 threads created
Found 3 solutions!
./sudoku 8 < ../inputs/input16_other.in  61,36s user 0,04s system 278% cpu 22,065 total
→ base git:(master) ✗ time ./sudoku 16 < ../inputs/input16_other.in
Initialized structures
Created first stack entry
Created threads
16 threads created
Found 3 solutions!
./sudoku 16 < ../inputs/input16_other.in  61,23s user 0,07s system 268% cpu 22,836 total
```

(Sudoku com 3 soluções em 1, 2, 4, 8 e 16 threads)

Assim como no N = 3, tivemos uma melhora no desempenho com mais threads e a partir de um número começa a ter uma queda de desempenho.

A melhor execução registrada foi com 8 threads: 22.065 segundos.

Conclusão

Podemos observar pelos resultados obtidos que o método utilizado para resolução do problema proposto, enquanto atingiu o objetivo de deixar a solução mais rápida para sudokus maiores, chocou-se com outro problema, o de memória insuficiente, e portanto, não foi completamente satisfatório.

A menor diferença de eficiência do *multi-thread* em Sudokus 9x9 é devido a pequena granularidade das operações, já que uma linha de um sudoku pequeno tem poucas combinações quando comparada a de um sudoku de uma ordem maior, nos testes feitos usando-se sudokus 16x16 e utilizando a resolução total de todas as possibilidades de linha com backtracking observamos um aumento na velocidade de resolução entre o algoritmo de backtracking *mono-thread* e nossa implementação *multi-threaded*.

O problema da memória se encontra quando o sudoku dado possui a maioria de seus ‘slots’ em aberto, ocasionando o empilhamento de milhões de possibilidades de sudokus em nossa pilha, este problema fica mais acentuado quando os ‘slots’ vazios se encontram seguidos uns dos outros, onde uma linha vazia é um dos maiores culpados.

Foram pesquisados alguns algoritmos alternativos. Um dos algoritmos pesquisados tratavam o Sudoku como um problema de cobertura exata e outro método resolvia o Sudoku utilizando programação linear. Ambos métodos citados necessitam de um aprofundamento maior em outras disciplinas para se ter um entendimento completo.

Referências

https://en.wikipedia.org/wiki/Sudoku_solving_algorithms - 2017/09/27
https://en.wikipedia.org/wiki/Integer_programming - 2017/09/27
https://en.wikipedia.org/wiki/Exact_cover#Sudoku - 2017/09/27
https://en.wikipedia.org/wiki/Mathematics_of_Sudoku - 2017/09/27