Implementação paralela do Sokoban utilizando Open MP

Leonardo Nascimento

Centro de Matemática, Computação e Cognição (CMCC)

Universidade Federal do ABC (UFABC) – Santo André – SP – Brasil

nascimento.leonardo@aluno.ufabc.edu.br

1 Otimizações

Tentei utilizar as duas implementações em C disponíveis no Rosetta Code em outro projeto, chamado ufabc-bcc/ohsok. Entretanto, não consegui paralelizar nenhuma delas por conta dos problemas de falha de segmentação, e mesmo com a ajuda da ferramenta valgrind não soube resolver.

No projeto entregue, chamado ufabc-bcc/psokoban, tentei criar uma função de hash para armazenar um tabuleiro como um número inteiro no histórico de movimentos que seria uma árvore binária. Também tentei fazer a compressão da string do tabuleiro e armazenar o histórico de movimentos em uma Trie. Falhei nas duas implementações.

Por fim, utilizei as macros da biblioteca uthash para armazenar o tabuleiro como o hash de uma string em uma tabela de histórico de movimentos.

2 Implementação sequencial

A implementação utilizada como base foi a implementação em C# presente no Rosetta Code, onde o código é mais compacto e mais fácil de entender.

3 Implementação paralela

Usando como base a implementação sequencial e OpenMP, distribuímos para cada thread, por round-robin, quatro caminhos para serem explorados.

A exploração é feita de forma independente até que a solução seja encontrada, onde todos os threads são avisados.

O histórico de movimentos é compartilhado entre todas as threads, e quando uma alteração for feita no histórico, esta será dentro de uma seção crítica.

Vale ressaltar que o algoritmo não é determinístico, pois a o caminho escolhido será o da thread que encontrar uma solução primeiro.

4 Resultados

No meu computador portátil com processador Intel Core i7-8550U (Quad Core, 8M Cache, 1.8GHz, 15W) e 16GB de memória RAM (Dual In-Line Memory Module, 8GB, 2400Mhz, DDR4) foi possível resolver apenas os níveis -1 e 00, além do nível A, que é uma versão simplificada do nível 01. Vale salientar que as análises foram feitas com o dispositivo conectado à tomada.

Tabela 1. Solução e tempo para os níveis iniciais com 8 threads.

Nível	Solução	Tempo (s)	Threads
-1	R	0,004	8
00	ulULLulDDurrrdd lULrruLLrrUruLL LulD	0,126	8
01	N/A	N/A	N/A
A	ullluuullllddlldRR uuurruurrDullddD DDRRRuuuLLrrd ddRRRRRRRRllll lllllllluuUUdlldddR RRRRRRRRRRdr Ullllllddllll UUlluu urrdDuullddddrRR RRRRRRRRuRRl Dlllllllluuuluuull D DDDDuullddddrRR RRRRRRRRRldR R	5,468	8

5 Speedup e eficiência

Para os níveis -1, 00 e A obtemos um speedup máximo de 3,744 e eficiência máxima de 0,862. Entretanto, isolados, esses valores não são relevantes, uma vez que eu apenas posso apenas estar dando sorte da minha solução ser escolhida.

Vale notar que, para o nível A, o speedup aumenta quando vai de 8 threads para 16 threads sendo que diminui quando foi de 4 threads para 8 threads, o que é um forte indício de que por acaso houve uma divisão onde a solução pôde ser encontrada mais rápido.

Para o nível 00, como o speedup é crescente até 8 threads, que é a quantidade de threads do processador, e depois há uma queda, é possível acreditar que a melhora de no tempo não é apenas sorte.

Tabela 2. Tempo de execução, speedup e eficiência para o nível -1.

Nível -1					
Threads	1	2	4	8	16
Tempo (s)	0,004	0,004	0,004	0,006	0,007
Speedup	1,000	1,000	1,000	0,667	0,571
Eficiência	1,000	0,500	0,250	0,083	0,036

Tabela 3. Tempo de execução, speedup e eficiência para o nível 00.

Nível 00					
Threads	1	2	4	8	16
Tempo (s)	0,307	0,239	0,089	0,082	0,086
Speedup	1,000	1,285	3,449	3,744	3,570
Eficiência	1,000	0,642	0,862	0,468	0,223

Tabela 4. Tempo de execução, speedup e eficiência para o nível A.

Nível A					
Threads	1	2	4	8	16
Tempo (s)	12,385	8,158	4,982	5,615	5,480
Speedup	1,000	1,518	2,486	2,206	2,260
Eficiência	1,000	0,759	0,621	0,276	0,141

6 Escalabilidade

Considerando o nível A como proporcional ao nível 00, podemos dizer que o algoritmo não é escalável.

7 Consumo de memória

O consumo de memória para resolver os níveis -1 e 00 é muito baixo. Para o nível A o consumo chega a consumir 1GB e para o nível 01 a aplicação é suspensa quando chega próximo dos 14GB consumidos.

A estrutura para manter o histórico de movimentos é pouco eficiente e é a responsável pelo consumo elevado.

8 Balanceamento de carga

Na branch elapsed-time, onde são feitas as medidas de tempo de execução de cada thread, é possível notar que pelo menos metade das threads terminam muito próximas, e em nenhuma execução o processamento ficou apenas para uma thread, com exceção do nível -1.

9 Considerações finais

Dividir o processamento de uma árvore para diferentes threads não é uma tarefa fácil.

Referências

Notas de aula de Programação Paralela. Disponível em <http://professor.ufabc.edu.br/~e.francesquini/2019.q1.pp/> Acesso em: 21 abr. 2019.

Sokoban. Disponível em <https://rosettacode.org/wiki/Sokoban> Acesso em 8 abr. 2019.

A hash table for C structures. Disponível em <http://troydhanson.github.io/uthash/> Acesso em: 5 mai. 2019.