Quebra-Cabeças de Loyd: um estudo de paralelização

Júlia A. S. Oliveira, João P. Trevisan, Jayme S. R. Furuyama, Hélio Guardia

Departamento de Computação– Universidade Federal de São Carlos (UFSCar)

Caixa Postal 676 – 13.565-905– São Carlos – SP – Brasil

{julia.sousa, joaotrevisan, jaymefuruyama}@estudante.ufscar.br

Resumo: Este artigo apresenta um estudo sobre a implementação de solução paralela para o *Loyd's Tile Puzzle* (LTP), um desafio de complexidade NP-hard. Para tanto, propõe-se uma abordagem de paralelização usando múltiplas *threads* em ambiente com memória compartilhada. Também é abordada a viabilidade de uso de GPUs para otimizar o processo, discute-se o potencial das GPUs como alternativas promissoras para a paralelização e otimização do problema, explorando extensivamente o espaço de caminhos possíveis.

1. Introdução

Segundo Trapa (2004), o quebra-cabeças de ladrilhos de Loyd (*Loyd's Tile Puzzle* - LTP) foi introduzido por cerca de 1870. O objetivo do jogo é reorganizar as peças em um tabuleiro, fazendo uma sequência de movimentos que deslizam uma peça de cada vez para o espaço vazio, revelando assim outro espaço vazio na posição da peça movida. Uma maneira de resolver esse quebra-cabeça é usar um algoritmo de ramificação e poda, que é derivado da Busca em Profundidade (DEP) atuando regressivamente para retornar a solução ótima [Rodrigues, 2009]. No problema em questão, a função de utilidade escolhe o ramo que tem um valor de distância menor entre dois pontos no plano cartesiano usando a métrica de Manhattan [Approbato, 2019] e considerando uma altura menor na árvore de decisão do algoritmo.

Utilizando como referência o código disponibilizado pela Maratona de Programação Paralela – SBAC-PAD na versão sequencial [Marathon of Parallel Programming, 2008], apresentamos uma proposta de paralelização deste problema usando múltiplas *threads* em um ambiente *multicore* com memória compartilhada, e uma discussão sobre o uso de aceleradores do tipo GPU para solução do problema.

2. Aspectos gerais do problema

Segundo Brüngger (1997), encontrar a solução ótima para o Quebra-Cabeças de Loyd (LTP) é considerado um problema de combinação NP-hard de otimização e cuja solução apresenta dependência e informações limitantes. No estudo do autor, explora-se o uso de árvores de busca largas com mais de 10^{13} nós [Brüngger, 1999].

Tratando-se de um tabuleiro de dimensões n por n com todas as peças diferentes entre si, fica evidente que o espaço de busca é da ordem de n!, o que gera um espaço de busca muito vasto. Borovska (2006) apresenta uma estratégia com uso de *branch-and-bound* com uso de MPI e implementação baseada no paradigma de mestre-trabalhador e consegue um *speedup* de 1.7 com uso de 5 processadores,

apontando uma eficiência de 35% para o caso específico de uma matriz 7x7.

3. Materiais e métodos

No algoritmo para solucionar o quebra-cabeças, cada caminho possível é armazenado em uma pilha, sendo que o custo para completar o quebra-cabeça é baseado na soma das distâncias de Manhattan de cada peça até sua posição final. O processo de definição de um novo estado de teste envolve a execução de um movimento possível no quebra-cabeça atual, o que implica na troca de posição entre o espaço vazio e uma peça adjacente. Após cada movimento, o custo total é recalculado para refletir a nova configuração, considerando o número de movimentos realizados. Uma solução é encontrada quando seu custo é 0, ou seja, todas as peças já estão em suas posições esperadas.

Para a paralelização do código num ambiente computacional com múltiplos processadores e memória compartilhada, pode-se usar o modelo de tarefas (*tasks*) do OpenMP, que é adequado para caso de operações que podem ser executadas de maneira assíncrona, sem a necessidade de uma ordem fixa. O processamento irá verificar o topo da fila (estado atual do quebra-cabeça) e esse caminho é então removido da fila. Quando a solução é encontrada, indica-se o total de movimentos que foram necessários.

Para cada sequência de movimento na pilha de caminhos, uma nova *task* é criada com a diretiva *#pragma omp task* - assim o processamento de cada caminho se dará de forma assíncrona e paralela. Por questões do grau de paralelismo desejado e também do consumo de memória RAM, é relevante limitar em alguma profundidade a criação de tarefas aninhadas.

4. Discussões

Durante as tratativas e execução do código, obteve-se diversas observações quanto à solução proposta que demandam atenção e melhoria para continuidade do estudo. Por se tratar de um contexto com muitos caminhos a serem explorados, tem-se problemas com o gerenciamento de memória e riscos de estouro da pilha, pela grande quantidade de tarefas que podem ser criadas - uma vez que cada *task* é executada em seu próprio contexto, que inclui ter sua própria fila de chamadas. Nesse sentido, no Algoritmo 1, a ideia é fazer uma pré-alocação de um espaço de memória para cada caminho gerado para mitigar o risco de estouro da pilha, visto que para visitar todos os caminhos com um dado número de passos o número de caminhos é multiplicado por 4 em cada passo. Assim, implementa-se uma estrutura de dados compartilhada para armazenamento dos caminhos.

Algoritmo 1 Função CALCPATHS de pré cálculo dos caminhos possíveis

```
1: procedure CALCPATHS(pilha, num_passos)
2:     num_caminhos ← 4^num_passos
3:     for i ← 0 to num_caminhos do
4:         j ← i
5:         while j ≠ 0 do
6:             PUSH(pilha, j%4)
7:             j ← j/4
8:         end while
9:     end for
10: end procedure
```

Outro destaque é a proteção de acesso concorrente com uso de *mutexes* ou outras primitivas voltadas à sincronização, como forma de garantir que apenas uma *thread* possa acessar a estrutura de cada vez, de forma a manter a consistência nos dados, como visto no Algoritmo 2. Cada *thread* irá atuar adquirindo um *lock*, retirar um caminho da estrutura de dados compartilhada, liberar o *lock* para permitir novamente a leitura na pilha, fazer o processamento desse caminho e, se necessário, finalizar a execução do programa tendo encontrado um caminho que resolve o problema.

Algoritmo 2 Função de execução de caminho

```
1: procedure LOYD(pilha)
2:     MUTEX_LOCK(())
3:     caminho ← POP(pilha)
4:     MUTEX_UNLOCK(())
5:     PERCORRER(caminho)
6: end procedure
```

Conforme as novas *tasks* são geradas (novos caminhos do quebra-cabeça), elas são colocadas num *pool* de tarefas (implícito em OpenMP), onde as *threads* definidas para a região paralela atual podem pegá-las para a execução, conforme se explicita nas diretivas *#pragma omp single* e *#pragma omp task* do Algoritmo 3.

Algoritmo 3 Resumo da função principal

```
1  function MAIN
2     CALCPATHS (pilha, num_passos)
3     ...
4     #pragma omp parallel
5      #pragma omp single
6     for i <- 0 to num_caminhos do
7        #pragma omp task
8        LOYD (pilha)
9     end for
10 end function
```

5. Próximos Passos

A solução descrita neste trabalho faz uso da API OpenMP e se mostra viável a partir das tratativas destacadas no pseudocódigo, porém se faz necessária uma reestruturação do código inicialmente apresentado pela Maratona de Programação Paralela – SBAC-PAD na versão sequencial [Marathon of Parallel Programming, 2008] para maior aderência à estratégia proposta anteriormente como solução. Subsequentemente, a análise dos resultados se fará possível e trará materialidade quando a corretude de nossas hipóteses.

O recurso de aceleradores de hardware é uma opção atraente. O problema de atacar este desafio com o modelo de programação em GPU pousa no fato de que encontrar uma solução recursiva para este problema implica delimitar qual é a totalidade do seu espaço de busca, conforme outros os estudos trouxeram como observação. Tendo em vista o uso de GPU junto da biblioteca CUDA, considera-se aplicar uma estratégia que use os índices lógicos das *threads* de uma grade de blocos como forma de mapear os caminhos a serem tentados por cada uma delas, mostrando uma alternativa para busca extensiva dentro de todo o espaço amostral de caminhos.

O mapeamento dos movimentos para os valores das movimentações (0, 1, 2 e 3) permitiria fazer uso dos módulos das sucessivas divisões do índice lógico de uma *thread* por 4, armazená-los em um vetor e aplicar cada um dos valores como instrução para se mover o espaço vazio no tabuleiro do LTP, semelhante à abordagem com uso de OpenMP. Dessa maneira teríamos uma solução que visitaria todos os caminhos possíveis e ainda garantiria encontrar o mais curto que se adapta ao paradigma de programação em GPU.

6. Referências Bibliográficas

APPROBATO, Daví Carlos Uehara. (2019) Distância, na matemática e no cotidiano. Tese (Doutorado) - Universidade de São Paulo, São Paulo, 2019.

BOROVSKA, Plamenka. (2006). Parallel Combinatorial Search on Computer Cluster: Sam Loyd's Puzzle. In: International Conference on Computer Systems and Technologies-CompSysTech.

BRÜNGGER, Adrian. (1997). Solving hard combinatorial optimization problems in parallel: two case studies. ETH Zurich.

BRÜNGGER, A.; MARZETTA, A.; FUKUDA, K. et al. (1999). The parallel search bench ZRAM and its applications. Annals of Operations Research, v. 90, p. 45-63. Disponível em: https://doi.org/10.1023/A:1018972901171. Acesso em: 10 mar. 2024.

RODRIGUES, Rodrigo Matheus da Costa. (2009). Uma comparação experimental entre algoritmos para seleção de características. Monografia (BI) - Centro de Ciências Exatas e Tecnológicas, Universidade Estadual do Oeste do Paraná, Cascavel.

TRAPA, Peter. (2004). Permutations and the 15-Puzzle. Math Circle Notes, v. 2, n. 02, University of Utah. Disponível em: http://155.101.98.133/mathcircle/notes/permutations.pdf. Acesso em: 10 mar. 2024.

Marathon of Parallel Programming**.** (2008). Disponível em: <http://lspd.mackenzie.br/marathon/08/problems.html>. Acesso em: 7 abr. 2024