

Algoritmo A* Paralelo para a Solução Ótima dos Problemas do Cubo Mágico e Sliding Puzzle

Leonardo Machado¹, Daniel Sundfeld¹

¹Faculdade de Ciências e Tecnologias em Engenharias (FCTE)
Universidade de Brasília (UnB), DF - Brasil

`machado.goncalves@aluno.unb.br, daniel.sundfeld@unb.br`

Resumo. Algoritmos paralelos estão se tornando cada vez mais importantes para a computação, considerando o caminho seguido pela evolução do hardware. Por essa razão, o estudo de paralelismo torna-se necessário para que os softwares acompanhem essa evolução. O Algoritmo A* é um dos mais famosos algoritmos de busca e pode ser utilizado para solucionar uma quantidade enorme de problemas. No entanto, obter o paralelismo nesse algoritmo pode ser um desafio. Uma das estratégias de maior sucesso na literatura são algoritmos que usam funções de hash para distribuir os nós entre as threads. O presente trabalho, adapta essa estratégia para resolver o problema do Sliding Puzzle e Cubo Mágico. Os resultados experimentais foram obtidos em uma estação de 24 núcleos e 768 GB de RAM e foi possível obter um speedup médio de até $9,90\times$ usando 24 threads.

1. Introdução

O paradigma de melhoria de desempenho dos processadores sofreu uma mudança a partir da década de 2000 [Tanenbaum and Bos 2015]. Devido a limitações físicas que dificultam o aumento contínuo de clock em núcleos individuais, a indústria passou a focar no aumento do número de núcleos de processamento. Contudo, para usufruir dessas melhorias de hardware, o desenvolvimento de software precisa adotar arquiteturas paralelas [Sundfeld et al. 2025, Kobayashi et al. 2011].

Este trabalho apresenta uma proposta de adaptação de algoritmos de busca em grafos para o ambiente paralelo, buscando a solução ótima para dois problemas da classe NP-completo: o Sliding Puzzle e o Cubo Mágico. Para ambos os domínios, encontrar a melhor solução (aquela com o menor número de movimentos) exige algoritmos com melhor direcionamento para encontrar a solução, uma vez que a árvore de possibilidades cresce exponencialmente.

A proposta baseia-se na implementação de versões paralelas do algoritmo A*, avaliando o comportamento da busca paralela, as sobrecargas (overheads) de comunicação e o speedup atingido. Este artigo está estruturado da seguinte forma: a fundamentação teórica é apresentada na Seção 2, a proposta do trabalho e os resultados experimentais na Seção 3 e, finalmente, a conclusão e trabalhos futuros são apresentados na Seção 4.

2. Fundamentação Teórica

A modelagem de quebra-cabeças como o Sliding Puzzle e o Cubo Mágico ocorre através de grafos [Khemani et al. 2018, Kouti and Refanidis 2023]. Cada vértice do grafo representa um estado único do quebra-cabeça, e as arestas mapeiam as transições (movimentos válidos) entre estes estados.

O A* [Hart et al. 1968] é um algoritmo de busca informada clássico capaz de encontrar o caminho mínimo em grafos ao utilizar uma função de avaliação que combina o custo real acumulado com uma estimativa heurística admissível. Para o Sliding Puzzle, adotou-se a distância de Manhattan como heurística; já para o Cubo Mágico, utilizou-se um banco de dados de padrões (*Pattern Databases*), que pré-computa os caminhos ótimos para subproblemas das extremidades do cubo.

Implementações de busca em paralelo com o A* esbarram em desafios de sincronização, condições de corrida e replicação de processamento (overhead de busca). Para lidar com isso, trabalhos como o *Hash Distributed A** (HDA*) [Kobayashi et al. 2011] e o *PA-Star2* [Sundfeld et al. 2025] distribuem a carga utilizando funções *hash*. A cada novo vértice gerado pela expansão da fronteira de busca, aplica-se uma função hash que mapeia o vértice a uma thread específica, garantindo distribuição da carga e impedindo que mais de uma thread expanda o mesmo nó.

3. A* paralelo para o Sliding Puzzle e Cubo Mágico

A metodologia deste trabalho adota a linguagem C++ (padrão C++20) aliada ao POSIX Threads (pthreads) para orquestrar o multiprocessamento. Na arquitetura paralela implementada para ambos os jogos, instanciam-se múltiplas threads ativas. Cada thread gerencia localmente as suas próprias estruturas de busca: (a) Uma Fila de Prioridade (OpenList) para decidir o próximo estado a expandir; (b) Tabelas *Hash Map* (ClosedList) para guardar os nós visitados e na fronteira de expansão; e (c) Uma fila síncrona (*queue*) para recebimento de vértices gerados por outras threads.

O fluxo de funcionamento dita que uma thread retira o estado mais prioritário e gera suas adjacências (movimentos válidos). Em seguida, a função `boost::hash_combine` é executada em cima das características do nó recém-gerado. Se o resultado da hash indicar a mesma thread geradora, o nó é inserido nas listas locais; do contrário, ele é despachado de forma segura para a fila da thread alvo responsável por ele. A rotina é repetida independentemente por todas as threads até que o estado resolvido seja descoberto, propagando um sinal de parada global.

Os experimentos foram conduzidos em uma máquina com 24 núcleos físicos (2x Intel Xeon E5-2680 2.50GHz) e 768 GiB de RAM. O tempo de execução e a evolução do speedup foram avaliados utilizando configurações de 1, 2, 4, 8, 16 e 24 threads. Foi imposto um teto de limite de memória de 200 GiB para barrar explosões de estado em problemas excessivamente profundos.

Para o Sliding Puzzle, as baterias envolveram malhas 3x3, 4x4 e 5x5. Em cada malha, foram executados 10 casos de teste gerados aleatoriamente. No cenário da malha 4x4, que apresenta um espaço de estado moderado-alto, obteve-se um speedup médio de 8,64 utilizando 24 threads (chegando a 9,81 em determinados casos). Já na malha 5x5, muitos casos esbarraram no limite de memória antes da conclusão devido à explosão combinatória. Todavia, nos cenários concluídos, o speedup médio alcançou 7,24x em 24 threads. O tempo de resolução de um puzzle 5x5 caiu de mais de 47 minutos (2822 segundos) de forma sequencial para apenas cerca de 6 minutos (383 segundos) paralelizado. Os resultados multithread foram comparados com os de uma versão puramente sequencial. Em ambos, a lógica do A*, a heurística, a função de hash, e as estruturas de dados usadas são as mesmas, sendo o overhead de paralelismo a única diferença.

Para o Cubo Mágico, foram gerados testes em distâncias inversas da solução variando de 7 até 13 movimentos, sendo 10 casos de teste aleatórios para cada distância. Os ganhos de speedup escalaram progressivamente de acordo com a carga exigida. Em configurações simples (distância 7), o overhead de paralelização impediu grandes ganhos (máximo médio de 2,50x em 16 threads). No entanto, nas instâncias de alta complexidade em distância 13, o cenário é inverso: 24 threads atingiram um speedup médio de 9,59x. Um dos testes demorou mais de 2 horas (7575 segundos) com uma thread, e foi concluído em pouco mais de 11 minutos (666 segundos). Diferentemente do *Sliding Puzzle*, o Cubo Mágico não teve uma versão puramente sequencial implementada, sendo usada a versão paralela com uma thread para comparação. A motivação dessa abordagem se deve à ordem de grandeza do tempo necessário para solucionar cubos mais complexos, que torna o tempo de overhead uma proporção muito pequena do tempo de solução.

4. Conclusão e Trabalhos Futuros

A estratégia de dividir a carga de estados da fronteira de busca através de hash (PA-Star) provou-se vantajosa para os dois problemas propostos. Conclui-se que o ganho é escalável: problemas com grafos maiores se beneficiam de uma maior alocação de threads, diluindo os overheads inerentes da comunicação. Sendo o principal fator limitante de execução a memória disponível. Sugerimos como trabalhos futuros a integração de heurísticas mais fortes e elaboradas para refinar a poda da árvore e reduzir o uso de memória. Além disso, o uso de novas heurísticas permite realizar testes comparativos com outras versões de A* utilizadas para solucionar os problemas apresentados. Paralelamente, implementações utilizando memória secundária, para lidar com a limitação também devem ser analisadas.

Referências

- Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans Sys Sci and Cybernetics*, SSC-4(2):100–107.
- Khemani, C., Doshi, J., Duseja, J., Shah, K., Udmale, S., and Sambhe, V. (2018). Solving rubik's cube using graph theory. In *Computational Intelligence: Theories, Applications and Future Directions-Volume I: ICCI-2017*, pages 301–317. Springer.
- Kobayashi, Y., Kishimoto, A., and Watanabe, O. (2011). Evaluations of hash distributed a* in optimal sequence alignment. In *IJCAI*, pages 584–590.
- Kouti, A. M. and Refanidis, I. (2023). Cpu and gpu parallelism of the a* algorithm on solving n-puzzle problems. In *Proceedings of the 27th Pan-Hellenic Conference on Progress in Computing and Informatics*, pages 21–25.
- Sundfeld, D., Teodoro, G., and Melo, A. C. (2025). Pa-star2: fast optimal multiple sequence alignment for asymmetric multicore processors. In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 146–153. IEEE.
- Tanenbaum, A. S. and Bos, H. (2015). *Modern operating systems*. Pearson Education, Inc.