

# Análise da Melhoria de Desempenho do Algoritmo A\*

## Utilizando OpenMP

Eduardo da Silva<sup>1</sup>, Rogério Aparecido Gonçalves<sup>1</sup>, João Fabrício Filho<sup>1</sup>

<sup>1</sup>Universidade Tecnológica Federal do Paraná – Campus Campo Mourão  
Via Rosalina M Santos, 1233 – 87301-899 – Campo Mourão – PR – Brasil

eduardosilva.2005@alunos.utfpr.edu.br, {rogerioag, joaof}@utfpr.edu.br

**Abstract.** Applications can be improved by dividing the work so that multiple tasks are processed simultaneously. The goal of this work is to decrease the response time of the A\* search algorithm through parallelization and achieve better performance compared to its serial version. To do this, this work implemented the search in parallel using OpenMP, dividing the search into four threads in the main loop of the algorithm. The result is a time gain from parallelism compared to serial between 8,49% and 9,32%, a percentage below what is expected when considering parallelized codes due to the way the algorithm works, using lists that cannot be updated simultaneously.

**Resumo.** Aplicações podem ser melhoradas quando se divide o trabalho a ser feito de forma com que com que múltiplas tarefas sejam processadas simultaneamente. O objetivo deste trabalho é diminuir o tempo de resposta do algoritmo de busca A\* por meio da paralelização e alcançar um melhor desempenho em relação à sua versão serial. Para tanto, este trabalho implementou a busca de forma paralela utilizando OpenMP, dividindo a busca em quatro threads no principal laço do algoritmo. O resultado é um ganho de tempo do paralelo em relação ao serial entre 8,49% e 9,32%, uma porcentagem abaixo do esperado quando se considera códigos paralelizados por conta do modo que o algoritmo trabalha, utilizando de listas que não podem ser atualizadas simultaneamente.

## 1. Introdução

Com o passar dos anos a programação paralela veio se desenvolvendo cada vez mais, algo que faz sentido visto suas potenciais vantagens em termos de economia de tempo nas aplicações [Kirk and Wen-Mei 2016], porém não são todos os casos nos quais se compensa realizar este tipo de programação, por exemplo códigos que tenham que atualizar uma variável global em certa ordem. Mais de uma sequência de código sendo executada e atualizando simultaneamente o programa poderia vir a comprometer a integridade do código.

Este seria o caso do algoritmo A\* [Patrick 2005], um algoritmo que tem como objetivo a busca por um caminho em uma matriz, utiliza listas para guardar informações cruciais para seu funcionamento e aí está o desafio de se paralelizar este algoritmo de forma satisfatória. Entretanto o modelo de programação paralela OpenMP (Open Multi-Processing) possui ferramentas que possibilitam isto, as chamadas regiões

---

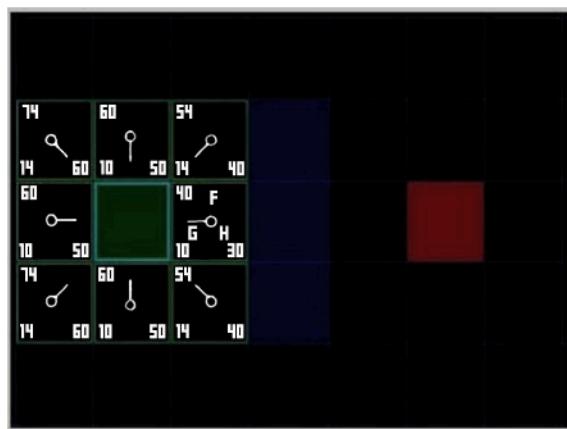
Agradecimentos ao CNPq pelo auxílio financeiro e concessão de bolsa - Projeto 01947 SISPEQ/UTFPR.

críticas, nas quais um bloco de código pode ser executado por uma *thread* por vez [Chapman et al. 2007]. Todavia, a utilização desta ferramenta nos dá resultados não tão eficientes quanto o esperado, pois a região crítica não permite paralelização. Os resultados obtidos mostram uma melhora de 9,32% quando comparado ao algoritmo sequencial em uma busca realizada em uma matriz de dimensões  $600 \times 600$ . Esses resultados são alterados à medida que se ajusta o tamanho da matriz por conta do número de vezes que o laço paralelizado é utilizado.

## 2. Proposta

O algoritmo proposto é desenvolvido em C e segue o padrão do algoritmo A\*[Patrick 2005]. Um algoritmo de busca em matrizes que combina os princípios de busca em largura com uma heurística para encontrar o caminho mais curto entre um nó inicial e um nó objetivo. Ele mantém uma lista de prioridade que guia a expansão dos nós com base no custo acumulado e na estimativa heurística do custo restante até o objetivo.

O algoritmo funciona utilizando alguns recursos, estes são duas listas, uma aberta que contém os próximos passos disponíveis e uma fechada com os já utilizados, além das listas ele utiliza três valores,  $F$ ,  $G$  e  $H$  como na Figura 1 [Patrick 2005] para calcular seus próximos passos, onde  $G$  é o custo do movimento do ponto de início até o quadrado determinado na malha seguindo o caminho criado para chegar ao destino,  $H$  é o custo estimado do movimento daquele quadrado determinado até o destino final, e  $F$  é a soma destes dois valores.



**Figura 1. Valores e direções do algoritmo A\* com origem e destino mostrados em verde e vermelho, respectivamente. Fonte: [Patrick 2005].**

A busca pelo próximo nó, ou vizinho, ocorre em um laço `for` de índice quatro, para que o algoritmo verifique nas quatro direções possíveis, por conta do uso da heurística *Distância de Manhattan*. A fim de alcançar o nó de destino com o menor trajeto possível, analisa as alternativas de caminho que possui e qual a melhor opção para seguir. Assim, a paralelização dessa busca se torna válida pois quatro *threads* podem fazer essa busca simultaneamente. Porém ao fazer isso surge o problema de quatro *threads* atualizarem as listas de próximos passos disponíveis e os já utilizados de forma síncrona.

O construtor de região crítica do OpenMP da suporte à solução desse problema. Embora limite o paralelismo do algoritmo onde é utilizado, determina áreas de

sincronização no código. O Código 1 mostra a implementação da região crítica para o algoritmo A\*, na qual se marca um início e um fim da área de sincronização.

#### Código 1. Utilização da região crítica.

```

1 if (!openListContains(openList, openCount, neighbor) ||
2 tentativeG <= neighbor->g) {
3 #pragma omp critical
4 {
5     neighbor->parent = current;
6     neighbor->g = tentativeG;
7     neighbor->h = heuristic(neighbor, goal);
8     neighbor->f = neighbor->g + neighbor->h;
9 }
10 if (!openListContains(openList, openCount, neighbor)) {
11 #pragma omp critical
12 {
13     openList[openCount++] = neighbor;
14 }
15 }
16 }
```

### 3. Avaliação Experimental

Para avaliar o desempenho do algoritmo proposto, realizamos uma série de experimentos em um ambiente controlado. Os resultados desses experimentos são detalhados nas subseções seguintes.

#### 3.1. Configuração

Para verificar os ganhos com o algoritmo proposto, avaliamos a implementação na linguagem de programação C em um processador *Intel(R) Core(TM) i7-4790 CPU @ 3,60GHz* com *32GB* de memória RAM. Utilizamos o modelo de programação do OpenMP<sup>1</sup> para a realização dos testes do algoritmo paralelizado.

#### 3.2. Resultados

A Tabela 1 apresenta os tempos que os algoritmos serial e paralelizado levaram para encontrar o melhor caminho nas matrizes de tamanhos variados. A menor é uma matriz de 256 pontos por 256 pontos e a maior 600 pontos por 600 pontos. Na menor matriz houve um percentual de melhoria correspondente a 9,32% enquanto nas matrizes de 512x512 e 600x600 8,49% e 8,79% respectivamente. Esse resultado se dá pelo fato de no algoritmo paralelizado a busca pelos próximos pontos acontecer de forma simultânea. São quatro buscas sendo realizadas paralelamente enquanto no algoritmo serial é necessário que uma busca se encerre para que outra inicie.

Todavia a melhora não é mais expressiva por conta da limitação no momento de se atualizarem as listas com os pontos encontrados pois as regiões críticas possibilitam a paralelização porém tornam o código sequencial na parte em que são utilizadas.

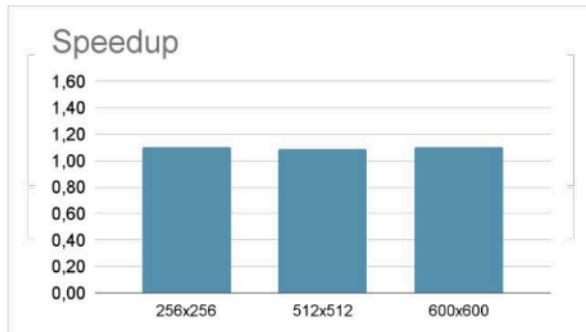
O *speedup* é calculado segundo a Equação 1, na qual  $p$  é o *speedup* teórico,  $T_1$  é o tempo de execução serial e  $T_p$  o tempo de execução paralelo. Utilizando a fórmula foi estimado que para os tamanhos de imagem de 600x600, máximo suportado pelo hardware, 512x512 e 256x256, tamanhos de matrizes que o algoritmo leva tempo de execução o suficiente para comparações, o *speedup* foi de 1,103, 1,093 e 1,103, respectivamente.

---

<sup>1</sup><https://www.openmp.org/>

Tempos de execução em segundos		
Dimensões	Serial	Paralelo
256x256	7,857s	7,122s
515x512	165,074s	150,897s
600x600	405,034s	367,034s

**Tabela 1.** Tempos de execução para diferentes dimensões de processamento, comparando a execução serial e paralela.



**Figura 2.** Speedup do algoritmo A\*.

$$p = \frac{T_1}{T_p} \quad (1)$$

Ao considerar a eficiência paralela obtida utilizando 4 processadores e dividindo o *speedup* entre eles, verifica-se que o desempenho do sistema melhora em média 27,47% na versão paralelizada. Os valores de eficiência foram de 27,55% para 256x256 e 600x600, e 27,32% para 512x512.

#### 4. Considerações Finais

Este trabalho explorou a paralelização do algoritmo A\* e mostrou por meio da experimentação que isto melhora seu desempenho, no caso da menor das matrizes testadas, 256x256 em até 9,32%, no entanto isto é uma baixa eficiência levando conta a paralelização utilizando quatro *threads*. O uso do construtor de regiões críticas possibilita a paralelização do algoritmo porém prejudica sua eficiência. O percentual de melhoria ainda pode ser maior em matrizes mais expressivas, porém por limitações de hardware não foi possível realizar tal teste. Sugiro como trabalhos futuros a paralelização do mesmo algoritmo utilizando GPUs e o teste em matrizes com dimensões maiores.

#### Referências

- Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.
- Kirk, D. B. and Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.
- Patrick, L. (2005). *A\* Pathfinding para iniciantes*. Traduzido por Reynaldo N. Gayoso, [rngayoso@msn.com](mailto:rngayoso@msn.com) em 04 de Abril de 2005. Atualizado em 04 de junho de 2005.