Revista Eletrônica de Iniciação Científica em Computação, 2026, 24:1
ISSN: 3085-8461 • doi: 10.5753/reic.2026.7147

Artigo de Pesquisa/Research Paper

Paralelização do Algoritmo de Dijkstra para Grafos não Direcionados de Grande Escala

Parallel Dijkstra's Algorithm for Large-Scale Non-Directed Graphs

Leanderson Gustavo Silva e Silva [Instituto Federal do Maranhão | *leanderson.silva@acad.ifma.edu.br*]
Omar Andres Carmona Cortes [Instituto Federal do Marahão |*omar@ifma.edu.br*]

Bacharelado em Sistemas de Informação – Departamento de Computação (DComp), Instituto Federal do Marahão (IFMA), Av. Getúlio Vargas, 04, Monte Castelo, São Luís, MA, 65030-005, Brasil.

Resumo. A escolha de uma implementação eficiente do algoritmo de Dijkstra para encontrar o caminho mais curto em grafos é crucial, especialmente em aplicações que lidam com grandes volumes de dados, como redes de transporte, sistemas de navegação e redes de telecomunicações. Este estudo investiga e compara diferentes abordagens paralelas do algoritmo de Dijkstra, incluindo implementações com OpenMP, MPI e CUDA, em relação à versão sequencial tradicional. As implementações paralelas foram avaliadas quanto ao tempo de execução, speedup e eficiência, utilizando grafos de diferentes tamanhos. Todas as versões paralelas superaram a implementação sequencial em desempenho, com variações de eficiência conforme a abordagem e o tamanho do grafo, alcançando speedup de 6,31 com 8 threads em OpenMP, 6,64 com 4 processos em MPI e 37,8 em GPU para grafos com 16.384 vértices.

Abstract. The efficient implementation of Dijkstra's algorithm for finding the shortest path in graphs is crucial, especially in applications that handle large volumes of data, such as transportation networks, navigation systems, and telecommunication networks. This study investigates and compares different parallel approaches to Dijkstra's algorithm, including implementations with OpenMP, MPI, and CUDA, against the traditional sequential version. The parallel implementations were evaluated in terms of execution time, speedup, and efficiency using graphs of different sizes. All parallel versions outperformed the sequential implementation, with efficiency varying by approach and graph size, achieving speedups of 6.31 with eight threads in OpenMP, 6.64 with four processes in MPI, and 37.8 on the GPU for graphs with 16,384 vertices.

Palavras-chave: Computação Paralela, Caminho Mínimo, Dijkstra, OpenMP, MPI, CUDA

Keywords: Parallel Computing, Shortest Path, Dijkstra, OpenMP, MPI, CUDA

Recebido/Received: 09 December 2025 • **Aceito/Accepted:** 06 March 2026 • **Publicado/Published:** 27 March 2026

1 Introdução

No contexto da era digital, a análise eficiente de grandes volumes de dados tornou-se uma demanda central em diversas áreas do conhecimento. Em muitos desses cenários, sistemas complexos são modelados por meio de grafos, estruturas que capturam relações entre entidades e que aparecem em domínios como redes de computadores, transporte público, redes sociais e logística.

Dentre os algoritmos aplicados a grafos ponderados, destaca-se o algoritmo de Dijkstra, amplamente utilizado para o cálculo de caminhos mínimos [Cormen *et al*., 2002]. Apesar de sua relevância teórica e prática, a aplicação desse algoritmo em grafos de grande escala apresenta elevado custo computacional, uma vez que envolve a exploração sistemática dos vértices em busca do menor caminho [Noto and Sato, 2000]. Esse desafio torna-se ainda mais crítico em aplicações que exigem respostas em tempo quase real, como sistemas de transporte e redes de comunicação.

Nesse contexto, a computação paralela emerge como uma abordagem promissora para mitigar o custo de processamento associado a algoritmos de caminhos mínimos em grandes grafos. Ao explorar múltiplos recursos de processamento simultaneamente, é possível reduzir o tempo de execução e tornar viáveis aplicações que operam sobre milhões de vértices e arestas . Assim, técnicas baseadas em OpenMP, MPI e CUDA têm sido empregadas para distribuir a carga de trabalho entre múltiplos núcleos de CPU e unidades de processamento gráfico (GPU), visando acelerar a execução de versões paralelas do algoritmo de Dijkstra.

Este estudo tem como foco a paralelização do algoritmo de Dijkstra em grafos de grande escala por meio de técnicas de computação paralela, avaliando implementações em ambientes de multiprocessadores (CPU) com OpenMP, MPI e em GPUs com C-CUDA. O objetivo é analisar o desempenho na resolução de caminhos mínimos em grafos com até 16 mil vértices e 134 milhões de arestas, comparando os resultados obtidos com a versão sequencial quanto a tempo de execução, speedup e eficiência.

A escolha do algoritmo de Dijkstra justifica-se por suas propriedades de completude e otimalidade, uma vez que, assumindo pesos não negativos, ele garante a obtenção do caminho de menor custo entre dois vértices sem recorrer a funções heurísticas, caracterizando-se como uma estratégia de busca não informada Cormen *et al*. [2002]. Tais propriedades são particularmente relevantes em domínios como o transporte e as redes de computadores, nos quais a minimização de distâncias e atrasos desempenha um papel crucial. Dessa forma, ao investigar a paralelização do algoritmo de Dijkstra, este trabalho busca contribuir para o entendimento do equilíbrio

entre desempenho e escalabilidade em arquiteturas paralelas aplicadas a problemas de caminhos mínimos em grafos de grande porte com origem única.

Nesse contexto, este trabalho está dividido da seguinte forma: a Seção 2 apresenta os trabalhos correlatos enfatizando a importância deste estudo; a Seção 3 apresenta os detalhes de implementação em OpenMP, MPI e C-CUDA, juntamente com a metodologia e as métricas de desempenho; a Seção 4 mostra a análise de desempenho e discute os resultados; finalmente, a Seção 5 apresenta as conclusões do trabalho, suas limitações e os trabalhos futuros.

2 Trabalhos Correlatos

A habilidade de conectar dois pontos no espaço sempre foi de interesse tanto da indústria quanto da área militar, sendo que o método mais tradicional de fazer isso é via grafos [Solka *et al*., 1995]. O algoritmo de Dijkstra explora o grafo para encontrar o melhor caminho entre dois pontos, e Hassoun and Sanghvi [1990] está entre os pioneiros na investigação do uso de computação paralela para acelerar esse algoritmo na década de 1990.

Solka *et al*. [1995] implementaram uma versão paralela de Dijkstra em um processador matricial, uma arquitetura SIMD (*Single Instruction Multiple Data* – Instrução Única e Múltiplos Dados) em que vários processadores simples executam simultaneamente a mesma instrução sobre dados distintos organizados em matriz ou grade. No final dos anos 1990 e início dos anos 2000, com o surgimento e a popularização dos processadores multinúcleo, programadores passaram a adaptar suas aplicações para explorar múltiplos núcleos [Sardar and Faizabadi, 2018], em um cenário em que paradigmas de passagem de mensagem em extensões paralelas de linguagens seriais, como PVM (*Parallel Virtual Machine*) e MPI (*Message Passing Interface*), já disputavam espaço na programação paralela.

Nesse contexto, a paralelização de algoritmos de caminho mínimo tem sido amplamente investigada na literatura, em especial no caso do algoritmo de Dijkstra aplicado a grafos. Awari [2017] propõe um estudo inicial de paralelização do algoritmo de Dijkstra com origem única, utilizando OpenMP e MPI, e discutem o potencial de cada paradigma em arquiteturas multicore e ambientes distribuídos. Esse trabalho evidencia que, mesmo em configurações relativamente simples, a distribuição do cálculo dos caminhos mínimos entre threads ou processos pode proporcionar ganhos relevantes de desempenho em relação à versão sequencial.

Em arquiteturas de memória compartilhada, Calderon *et al*. [2024] apresentam um algoritmo aprimorado em OpenMP para o cálculo de caminhos mínimos entre todos os pares, explorando de forma sistemática a paralelização em plataformas x86. Embora o foco principal seja o problema de todos os pares, as estratégias de balanceamento de carga, afinidade de threads e redução de contenções em seções críticas são diretamente relevantes para a otimização de variantes de Dijkstra em ambientes multicore com origem única.

Mais recentemente, Song [2025] desenvolveu uma implementação de alto desempenho do algoritmo de Dijkstra combinando MPI e CUDA, voltada a grafos de grande escala. Nesse trabalho, MPI é empregado para distribuir o grafo entre processos em ambientes distribuídos, enquanto a GPU, por meio de CUDA, é utilizada para acelerar operações intensivas de processamento, recorrendo a primitivas como `MPI_Allreduce` e `atomicMin` para coordenar a atualização de distâncias. Em conjunto, esses estudos com OpenMP, MPI e GPU não apenas confirmam os benefícios da paralelização para grafos de grande escala, mas também revelam desafios persistentes, como overheads de sincronização em MPI, contenções em OpenMP e divergência de threads em CUDA, o que motiva novas comparações em cenários reais de computação de alto desempenho, como os realizados neste trabalho.

3 Materiais e Métodos

3.1 Algoritmos de Caminho Mais Curto

Algoritmos de caminho mais curto são uma classe de algoritmos fundamentais na Teoria dos Grafos e na Ciência da Computação. Esses algoritmos têm como objetivo encontrar o caminho mais curto entre dois pontos em um grafo ponderado, em que o termo "curto" é definido como um valor associado a cada aresta, como distância, custo, tempo ou qualquer outra medida relevante ao problema em questão.

Pode-se, por exemplo, modelar um mapa rodoviário como um grafo: os vértices representam intersecções, as arestas representam segmentos de estrada entre intersecções e os pesos das arestas representam as distâncias rodoviárias. A meta, nesse caso, é encontrar um caminho mais curto desde uma dada interseção no Rio de Janeiro até uma dada interseção em São Paulo, por exemplo. Os pesos de arestas podem ser interpretados como medidas, em vez de distâncias. Eles são usados com frequência para representar tempo, custo, penalidades, perdas ou qualquer outra quantidade que se acumule linearmente ao longo de um caminho que alguém deseja minimizar Cormen *et al*. [2002]

Nesse caso, o objetivo é encontrar o caminho mais curto entre um vértice de origem s e um vértice de destino v. O peso do caminho mais curto entre s e v é definido como $\delta(s, v)$, que é igual ao peso mínimo entre todos os caminhos possíveis entre esses dois vértices. Se não houver um caminho entre eles, então $\delta(s, v)$ é definido como infinito.

A escolha do algoritmo de Dijkstra para este trabalho se deve à sua eficiência e simplicidade na resolução do problema específico de caminho mais curto de uma única origem em grafos densos. Enquanto o algoritmo de Bellman-Ford é mais geral e pode lidar com grafos com ciclos e pesos negativos, sua complexidade assintótica o torna menos adequado para grafos densos Cormen *et al*. [2002].

O Algoritmo 1 apresenta o pseudocódigo do algoritmo de Dijkstra, iniciando com *Init-Origem-Unica*(G, s) no qual as distâncias dos vértices pertencentes a G, em relação à origem s são inicializadas, com todos os vértices recebendo uma distância de infinito, exceto a origem, que recebe o valor zero. O conjunto S, inicialmente vazio, armazena os vértices s cujas distâncias mais curtas já foram determinadas. Uma fila de prioridade Q contém todos os vértices de G em um vetor V que é iterada enquanto não estiver vazia Cormen *et al*. [2002].

Em cada iteração, remove-se o vértice u de Q com a menor distância conhecida. Para cada vértice v adjacente a u, realiza-se a operação de relaxamento $Relax(u, v, w)$, que

Algorithm 1 Dijkstra (G, w, s)

```
1:  Init-Origem-Unica(G, s)
2:  S ← ∅
3:  Q ← V[G]
4:  while Q ≠ ∅ do
5:      u ← Extract-Min(Q)
6:      S ← S ∪ {u}
7:      for each v ∈ Adj[u] do
8:          Relax(u, v, w)
9:      end for
10: end while
```

atualiza a distância mais curta conhecida em direção a v, se necessário, com base na distância atual de u e no peso da aresta entre eles.

3.2 Implementação

Esta implementação adota a versão ingênua do algoritmo de Dijkstra, com busca linear pelo vértice de menor distância não visitado ($O(n^2)$) no pior caso, em detrimento de variantes otimizadas com heap binário ($O((n+m)log\ n)$) ou heap de Fibonacci. Essa escolha visa simplicidade pedagógica e foco na comparação de paralelizações, permitindo isolar os efeitos de OpenMP, MPI e CUDA sem complicações de estruturas avançadas de dados. Como consequência, os tempos sequenciais crescem rapidamente para grafos densos, o que realça os ganhos paralelos observados nos experimentos.

Adicionalmente, o algoritmo de Dijkstra foi implementado com listas de adjacências [Backes, 2016]. A escolha pela estrutura dinâmica se deve ao fato de que, embora o desempenho de acesso à lista seja $O(n)$ e o de matriz de adjacências seja $O(1)$, o uso da lista de adjacências permite um melhor controle do tamanho dos grafos e da alocação de memória, especialmente em cenários com alta densidade de conexões.

O processo de encontrar o menor caminho segue o algoritmo tradicional: selecionar o vértice de menor distância ainda não visitado, marcá-lo como visitado, atualizar as distâncias dos vértices adjacentes e repetir o processo até que todos os vértices sejam visitados.

3.2.1 OpenMP

A ideia de usar OpenMP [OpenMP Architecture Review Board, 1997] é aproveitar a implementação sequencial e, experimentalmente, ir paralelizando o código utilizando as diretivas disponíveis, funcionando particularmente bem em arquiteturas com memória compartilhada.

Uma diretiva é formada pelo seu nome e suas clausulas. De acordo com Silva *et al*. [2022], suas principais diretivas são:

- $\#pragma\ omp\ parallel\ [clusula]\{\}$ - define o trecho de código que será executado por um grupo de threads simultaneamente, Suas principais diretivas são a quantidade de threads ($num_threads$), private que define as variáveis que são privadas em cada thread para evitar condições de corrida e $reduction$ que faz a junção de resultados de acordo com a operação, usualmente soma e multiplicação, dentre outras.
- $\#pragma\ omp\ single\ [clusula]\{\}$ - que identifica um bloco que deve ser executado por um único thread, sendo o primeiro que alcançar o bloco o thread que o executará.
- $\#pragma\ omp\ parallel\ [clusula]$ - identifica blocos que devem ser executados em paralelo, de forma independente, podendo ser tarefas completamente distintas.
- $\#pragma\ omp\ parallel\ for\ [clusula]$ - é a diretiva mais comum para efetuar tarefas paralelas dentro de um laço. É particularmente interessante para realizar tarefas de granularidade fina em vetores sem dependência de dados. Sua principal cláusula é a $schedule$, que define como as tarefas farão o balanceamento de carga entre as threads paralelas.

A Figura 1 mostra a paralelização do laço mais externo, que instancia x threads, cada qual executará uma parte do laço, buscando o vértice u de menor caminho.

```
#pragma omp parallel for num_threads(x)
for (count = 0; count < NUM_VERTICES - 1; count++) {
  int u = -1;
  for (int i = 0; i < NUM_VERTICES; i++) {
      if (!visitados[i] && (u==-1||distancias[i]<distancias[u]))
      {
        #pragma omp critical
        {
           if (!visitados[i] && (u ==-1||distancias[i]
               <distancias[u])){
                     u = i;
           }
        }
     }
  }
```

Figura 1. Código paralelo em OpenMP para busca do vértice mínimo no laço externo

O laço interno da implementação em OpenMP apresenta uma seção crítica com a diretiva $\#pragma\ omp\ critical$ que garante que apenas um thread tenha acesso ao nó visitado, impedindo condições de corridas na estrutura de dados. Em seguida, as distâncias dos vértices adjacentes ao vértice u são atualizadas de forma sequencial.

3.2.2 MPI

O MPI, na verdade, é um padrão que foi criado para padronizar a programação usando passagem de mensagem. Em outras palavras, seu objetivo é fornecer uma interface padrão para o desenvolvimento de programas usando o paradigma de passagem de mensagens. Adicionalmente, o MPI objetiva criar código portável, flexível e que aproveite as características paralelas do ambiente de execução. A partir disso, várias implementações foram criadas, como por exemplo, MPI LAM, MPICH, OpenMPI e o Microsoft MPI, dentre outros.

A vantagem do MPI é funcionar localmente mantendo seu desempenho [Silva *et al*., 2022], além de poder ser utilizado em ambientes distribuídos, já que, diferentemente de OpenMP, o MPI funciona com processos em vez de threads. A ideia de usar o MPI neste estudo é dividir o trabalho de cálculo dos caminhos mais curtos entre múltiplos processos entre os diferentes núcleos do processador, já que os comparativos são feitos na mesma arquitetura.

A implementação em MPI divide o trabalho de cálculo dos caminhos mais curtos entre os processos disponíveis. Cada processo é responsável por uma parte do grafo e realiza cálculos locais para encontrar o vértice de menor distância. Em seguida, os processos colaboram para identificar o vértice globalmente mais próximo por meio de operações de redução

coletiva ($MPI\_Allreduce$), garantindo que a atualização das distâncias seja consistente em todos os processos.

Como de praxe em MPI, os processos obtêm seus *ranks* e o número total de processos usando as instruções $MPI\_Comm\_rank()$ e $MPI\_Comm\_siz()$, respectivamente. Em seguida, cada processo calcula quantas iterações do laço externo deve executar, como mostra a Figura 2, e a variável $remainder$ ajusta a distribuição para que todos os processos tenham aproximadamente o mesmo número de vértices. O vetor $counts[i]$ define o número de iterações que cada processo i irá realizar. O vetor $displacements[i]$ define o deslocamento inicial das iterações para cada processo. Para processos com índice menor que $remainder$, o deslocamento é aumentado em 1 para cada iteração adicional atribuída.

```
int iterations_per_process = (NUM_VERTICES - 1) / size;
int remainder = (NUM_VERTICES - 1) % size;
int displacements[256];
int counts[256];

for (int i = 0; i < size; i++) {
     counts[i] = iterations_per_process;
     if (i < remainder)
     {
       counts[i]++;
     }
     displacements[i] = i * iterations_per_process +
     (i < remainder ? i : remainder);
}
```

Figura 2. Cálculo das iterações por processo em MPI para balanceamento de carga.

Localmente, cada processo encontra o vértice não visitado com a menor distância, como mostrado na Figura 3. Em seguida, um $MPI_Allredice$ combina os valores de todos os processos determinando a menor distância e o vértice associado a ela de forma global. Essa operação *reduce* é essencial para que todos os processos concordem sobre o próximo vértice a ser visitado.

```
int local_min_dist = INT_MAX;
int local_min_vertex = -1;

for (int i = rank + count * size; i < NUM_VERTICES; i += size *
iterations_per_process){
   if (!visitados[i] && distancias[i] < local_min_dist)
   {
     local_min_dist = distancias[i];
     local_min_vertex = i;
   }
}
```

Figura 3. Busca local do vértice mínimo por processo em MPI.

3.2.3 C-CUDA

As Unidades de Processamento Gráfico (GPUs) oferecem, em geral, uma taxa de processamento de instruções e uma largura de banda de memória significativamente maiores do que as Unidades Centrais de Processamento (CPUs), embora sejam otimizadas para instruções mais simples e fortemente vetorizadas. Sua arquitetura é normalmente classificada como de muitos núcleos (*many-core*). A Figura 4 ilustra as diferenças entre os núcleos de CPU e de GPU, evidenciando a quantidade muito maior de núcleos menores na GPU, configuração mais adequada para tarefas massivamente paralelas com granularidade fina, como operações sobre grandes vetores.

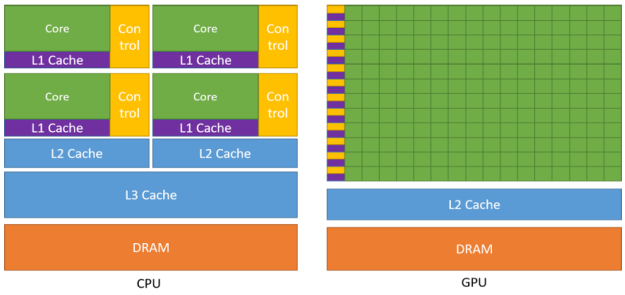


Figura 4. CPU vs GPU [NVIDIA Corporation, 2023]

Vale destacar que esta é uma arquitetura geral de GPU; modelos mais recentes são consideravelmente mais complexos e incluem unidades de processamento especializadas, como as *Tensor Cores*, projetadas para acelerar operações de álgebra linear densa (por exemplo, multiplicações de matrizes) típicas de aplicações de aprendizado de máquina e redes neurais profundas.

A versão CUDA do algoritmo de Dijkstra visa paralelizar a busca pelo menor caminho distribuindo as operações através de múltiplos núcleos de GPU, sendo especialmente útil em grafos com muitos vértices e densamente conectados. Para executar o código em C-CUDA, foi necessário configurar o ambiente de desenvolvimento através da instalação do NVIDIA CUDA Toolkit 12.4, disponível gratuitamente no site da NVIDIA.

Em termos de programação, a execução em GPU tem algumas exigências. A primeira é que toda estrutura de dados criada na CPU deve ser alocada também na GPU e os dados devem ser transferidos. Como essa operação é custosa, deve-se evitar transferências desnecessárias entre GPU e CPU. Adicionalmente, os programas CUDA utilizam kernels, que são subrotinas executadas no dispositivo CUDA. Um kernel é definido com o especificador $\_\_global\_\_$, indicando que pode ser chamado pelo host. Transformar rotinas computacionais em kernels é uma estratégia eficaz para acelerar aplicações GPGPU [Farber, 2011].

Para encontrar a distância mínima utiliza-se um kernel, cuja responsabilidade é encontrar o vértice não visitado com a menor distância local, como mostrado na Figura 5. A variável tid identifica a thread atual. A função atômica ($atomicMin$) garante que a atualização de $minDistIndex$ seja segura em um ambiente paralelo.

Para encontrar o próximo nó a ser visitado, utiliza-se a função apresentada na Figura 6, que copia os dados da CPU para a GPU e define a quantidade de blocos necessários para executar o kernel CUDA, assegurando que todos os elementos de um array de tamanho n sejam processados.

A quantidade de threads pro bloco é definida como uma constante $\#defineBLOCK\_SIZE$ 256, garantindo que a divisão sempre arredonde para cima quando n não for múltiplo de $BLOCK\_SIZE$. Finalmente, a função principal executa o algoritmo usando os kernels CUDA. A função tem estrutura semelhante à função de Dijkstra sequencial, mas possui as peculiaridades necessárias do CUDA.

A função $dijkstra\_CUDA$ é apresentada na Figura

```
__global__ void findMinDistance(int* d, bool* visited, int*
minDistIndex, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;

  if (tid < n && !visited[tid])
  {
     if (d[tid] < atomicMin(minDistIndex, d[tid]))
     {
          *minDistIndex = tid; }
     }
  }
}
```

Figura 5. Kernel CUDA para encontrar o índice da menor distância não visitada.

```
int findNextVertex(int*d_dev,bool*visited_dev,int* minDistIndex_dev,
int n) {

int minDistIndex = INT_MAX;

cudaMemcpy(minDistIndex_dev,&minDistIndex,sizeof(int),
cudaMemcpyHostToDevice);

int numBlocks = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
findMinDistance<<<numBlocks,BLOCK_SIZE>>>(d_dev,visited_dev,
minDistIndex_dev, n);

cudaMemcpy(&minDistIndex,minDistIndex_dev,sizeof(int),
cudaMemcpyDeviceToHost);

return minDistIndex;
}
```

Figura 6. Função para execução de kernel CUDA e encontrar próximo vértice.

7. A função inicia com a configuração de estruturas e variáveis essenciais para o funcionamento do algoritmo. O array $distancias[NUM\_VERTICES]$ armazena as distâncias mínimas de cada vértice em relação ao vértice inicial, sendo que a distância inicial de cada vértice é definida como INT_MAX, o que indica que ainda não foram alcançadas.

Além disso, a variável booleana $visitados[NUM\_VERTICES]$ indica quais vértices já foram visitados, com todos inicialmente marcados como não visitados $(false)$. Um loop *for* percorre todos os vértices do grafo, inicializando as distâncias com INT_MAX para indicar a ausência de caminhos conhecidos e marcando todos como não visitados. Adicionalmente, a função aloca toda a memória necessária na GPU, sendo que no laço principal é executado o trabalho da GPU, de fato, na chamada da função $findNextVertex$, que retornará um valor inteiro atribuído à variável u. Após isso, as distâncias são atualizadas e o processo se repete até o fim de todas as iterações.

```
void dijkstra_CUDA(struct Grafo* grafo, int inicio) {
     int distancias[NUM_VERTICES];
     bool visitados[NUM_VERTICES];

     for (int i = 0; i < NUM_VERTICES; i++) {
          distancias[i] = INT_MAX;
          visitados[i] = false;
     }
[...]
}
```

Figura 7. Função principal Dijkstra CUDA com laço principal na GPU.

3.3 Metodologia

Os experimentos foram conduzidos em um ambiente controlado com diversos tamanhos de grafos e quantidades de elementos de processamento, com as seguintes especificações de hardware e software de acordo com a Tabela 1.

A análise dos resultados foi feita utilizando as implementações: sequencial, OpenMP e MPI (com 2, 4 e 8 threads/-processos) e CUDA. Cada versão foi testada com diferentes quantidades de nós, variando de 256 a 16.384 vértices. Foram realizados 30 testes de execução para cada configuração, e os resultados foram analisados estatisticamente. Para cada bateria de 30 testes, foram calculados: Média aritmética, Mediana, Variância, Desvio padrão.

Além disso, também foram calculados os *speedups* para as versões paralelas, que é uma medida que indica o quão mais rápido a versão paralela do algoritmo é em comparação com a versão sequencial (BERLIN, 2024), sendo seu calculo efetuado pela Equação 1, na qual S_p é o tempo de execução do código sequencial e T_p é o tempo de processamento paralelo.

$$Sp = \frac{T_s}{T_p} \tag{1}$$

A eficiência é definida como a fração entre o speedup (Sp) e o número de elementos paralelos de processamento (np), conforme a Equação 2.

$$Ef = \frac{Sp}{np} \tag{2}$$

4 Resultados e Discussão

Os tempos médios de execução na versão sequencial por quantidade de vértices são apresentados na Tabela 8b, associados ao seu respectivo gráfico de tempo, indicando que a versão sequencial cresce rapidamente com o número de vértices, atingindo crescimento rápido a partir de 8192 vértice, O que é comum para um problema $O(n^2)$ como o caminho mínimo de implementação simples, ou seja, com implementação ingênua de Dijkstra com busca linear pelo vértice mínimo.

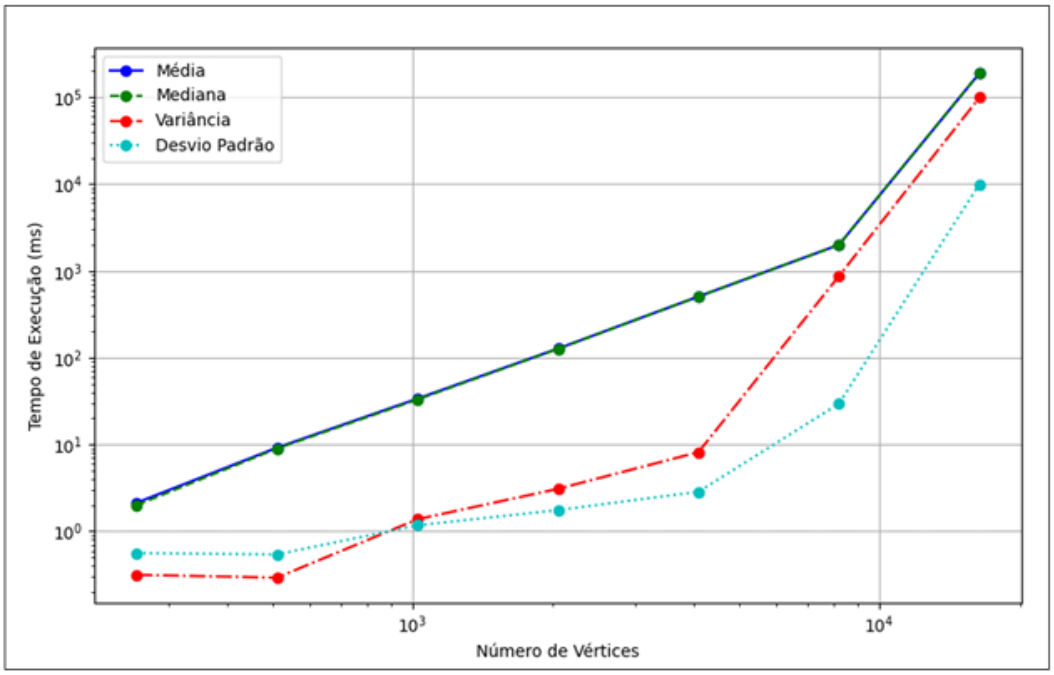
As Tabelas 2 e 3 mostram o tempo de execução por threads e processos, respectivamente, para OpenMP e MPI. Enquanto OpenMP e MPI reduzem substancialmente o tempo absoluto para praticamente todos os tamanhos de instância. Em grafos pequenos (256–512 vértices), o ganho é menor e, às vezes, o *overhead* de criação de threads/processos e de comunicação faz com que as versões paralelas fiquem relativamente menos vantajosas, o que é um comportamento esperado em algoritmos de caminho mínimo.

É interessante observar que, à medida que o número de threads/processos aumenta, a diferença de desempenho entre OpenMP e MPI diminui até que OpenMP passa a superar MPI em alguns cenários. Esse comportamento sugere que, para grafos de grande escala nesta configuração de hardware, o problema de caminho mínimo se beneficia mais do paralelismo em memória compartilhada (com granularidade fina nas threads) do que do paralelismo distribuído com maior sobrecarga de comunicação

O speedup é apresentado nas Tabelas 4 e 5, respectivamente, mostrando que OpenMP obtém ganhos moderados: em geral, entre 2× e 4× para 2 e 4 threads, chegando a cerca

Tabela 1. Especificações de hardware e software

Categoria	Detalhes
Hardware	Processador: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz (4 Cores, 8 Logical Cores) Memory RAM: 16GB GPU: NVIDIA GeForce GTX 1650 GPU Memory: 4GB dedicated, 8GB shared
Software	Operating System: Windows 11 Pro 64 bits (10.0, Build 22631) Compiler: Use of compilers compatible with OpenMP, MPI, and CUDA Libraries: OpenMP, MPI, CUDA



(a)

Nº de Vértices	Média (ms)	Mediana (ms)	Variância (ms^2)	Desvio Padrão (ms)
256	2,133	2	0,316	0,562
512	9,200	9	0,293	0,542
1024	33,766	33	1,378	1,174
2048	127,333	126,5	3,088	1,757
4096	506,605	507	8,173	2,858
8192	1998,03	1992	871,635	29,52
16384	188862,4	190323	99209,81	9960,375

(b)

Figura 8. Resultado da execução sequencial de acordo com a variação de quantidade de vértices

Tabela 2. Tempo de execução em OpenMP para diferentes números de threads e vértices

Nº de vértices	2 threads		4 threads		8 threads	
	Média (ms)	Desvio (ms)	Média (ms)	Desvio (ms)	Média (ms)	Desvio (ms)
256	0,878	0,363	0,534	0,385	0,700	0,562
512	4,072	0,389	3,019	4,020	1,852	0,698
1024	17,275	0,647	9,139	0,521	6,315	1,292
2048	66,172	1,297	34,813	1,206	28,097	6,842
4096	261,483	4,239	138,003	4,680	83,728	5,266
8192	1036,788	27,312	548,297	45,652	316,580	16,638
16384	96348,200	18142,455	92830,952	24492,325	120769,270	25706,398

Tabela 3. Tempo de execução em MPI para diferentes números de processos e vértices

Nº de vértices	2 processos		4 processos		8 processos	
	Média (ms)	Desvio (ms)	Média (ms)	Desvio (ms)	Média (ms)	Desvio (ms)
256	0,630	0,335	0,321	0,173	0,391	0,212
512	4,701	0,523	1,452	0,361	2,377	0,164
1024	15,107	0,688	5,380	0,416	8,726	0,372
2048	60,257	1,324	20,916	1,543	33,142	0,724
4096	240,578	3,464	118,028	77,540	129,304	1,256
8192	951,075	42,783	N/A	N/A	526,310	17,867
16384	N/A	N/A	N/A	N/A	N/A	N/A

Tabela 4. Speedup e eficiência do OpenMP

Nº de vértices	2 threads		4 threads		8 threads	
	Speedup	Eficiência	Speedup	Eficiência	Speedup	Eficiência
256	2.43	1.21	3.99	1.00	3.05	0.38
512	2.26	1.13	3.05	0.76	4.97	0.62
1024	1.95	0.98	3.69	0.92	5.35	0.67
2048	1.92	0.96	3.66	0.91	4.53	0.57
4096	1.94	0.97	3.67	0.92	6.05	0.76
8192	1.93	0.96	3.64	0.91	6.31	0.79
16384	1.96	0.98	2.03	0.51	1.56	0.20

Tabela 5. Speedup e eficiência do MPI

Nº de vértices	2 processos		4 processos		8 processos	
	Speedup	Eficiência	Speedup	Eficiência	Speedup	Eficiência
256	3.39	1.69	6.64	1.66	5.46	0.68
512	1.96	0.98	6.34	1.58	3.87	0.48
1024	2.24	1.12	6.28	1.57	3.87	0.48
2048	2.11	1.06	6.09	1.52	3.84	0.48
4096	2.11	1.05	4.29	1.07	3.92	0.49
8192	2.10	1.05	–	–	3.80	0.47

de 6× com 8 threads nos grafos médios e grandes. Em MPI, o speedup atinge valores da ordem de 6× a 7× para 4 processos em alguns tamanhos de grafo, mas não cresce de forma monotônica com o número de processos, o que sugere que o aumento do custo de comunicação e sincronização passa a limitar os ganhos de desempenho. Além disso, as tabelas também evidenciam problemas de alocação de memória, que ocorreram na computo dos tempos, impedindo o cálculo do speedup para determinadas combinações de tamanho de grafo e número de processos.

Adicionalmente, a eficiência evidencia o quão bem cada núcleo é utilizado; valores próximos de 1 indicam uso quase ideal dos recursos. Nos seus resultados, OpenMP apresenta eficiências altas para 2 e 4 threads (próximas de 1 para diversos tamanhos), mas cai para valores em torno de 0,6–0,8 com 8 threads, indicando que adicionar mais threads traz ganho, porém com overhead crescente. Para MPI, há casos de "eficiência maior que 1" com poucos processos, o que sugere superlinearidade associada a efeitos de cache ou a variação experimental, mas, para um maior número de processos e grafos maiores, a eficiência tende a ficar bem abaixo de 1, refletindo o custo de comunicação.

Do ponto de vista de escalabilidade forte (problema fixo, mais recursos), o código mostra boa escalabilidade até um número moderado de threads/processos, mas sinais claros de saturação quando se aumentam os recursos além disso. Em OpenMP, o speedup aproxima-se de um patamar para 8 threads, coerente com o limite imposto pela fração sequencial do algoritmo e pelas seções que exigem sincronização frequente (relaxações e atualização do vértice mínimo em Dijkstra). Em MPI, o comportamento é ainda mais limitado para grafos menores, pois a comunicação domina o tempo total, e os melhores ganhos aparecem apenas para instâncias maiores, o que indica que sua implementação é mais adequada a cenários de grande escala, mas com escalabilidade sujeita às leis clássicas de Amdahl e aos custos de rede.

Com relação à GPU, a Tabela 6 apresenta os resultados de speedup, mostrando que, para instâncias pequenas, a GPU perde desempenho devido ao overhead de inicialização e transferência de dados. À medida que o tamanho do grafo aumenta, o custo de computação passa a dominar e a GPU passa a escalar melhor, com um speedup ainda pouco expressivo em 4096 vértices, melhora em 8192 vértices e atinge um speedup elevado de 37,80 na maior instância, com 16.384 vértices.

No fim das contas, os resultados confirmam que OpenMP e MPI oferecem ganhos moderados (speedup até 6,3 e 6,6, respectivamente) em grafos médios (até 8k vértices), beneficiando-se do paralelismo fino em memória compartilhada/distribuída, alinhado aos achados de Awari [2017] e Calderon *et al*. [2024]. No entanto, a saturação em 8 threads/-processos reflete overheads de sincronização (seções críticas

Tabela 6. Tempo de execução e speedup em GPU

Nº de vértices	Média (ms)	Desvio	Speedup
256	16,058	4,152	0,13
512	33,619	10,437	0,27
1024	70,208	2,403	0,48
2048	175,258	6,120	0,73
4096	493,614	76,435	1,03
8192	1553,15	110,485	1,29
16384	4993,33	205,767	37,80

em OpenMP, MPI_Allreduce em MPI), limitados pela fração sequencial do algoritmo (busca linear do mínimo). Para MPI, falhas de alocação em grafos grandes (16k vértices) destacam desafios de escalabilidade forte, como discutido por Song [2025]. Já CUDA escala massivamente (speedup 37,8 em 16k vértices), superando CPUs em problemas de grande porte graças ao paralelismo de milhares de threads, apesar do overhead inicial de transferência CPU-GPU, um custo-benefício clássico em GPGPU.

5 Conclusões

A computação paralela é uma abordagem fundamental para a resolução de problemas de grande escala, como a busca de caminhos mínimos em grafos, pois permite dividir o trabalho entre múltiplos núcleos de processamento e reduz significativamente o tempo de execução em cenários de grande escala. Essa capacidade é especialmente relevante em aplicações que manipulam grandes volumes de dados e apresentam elevada complexidade computacional, nas quais a eficiência temporal é um requisito central.

Neste estudo, as implementações paralelas do algoritmo de Dijkstra apresentaram ganhos expressivos em relação à versão sequencial, tomando-se o tempo médio dessa versão como referência para o cálculo do speedup. A implementação com OpenMP, explorando o paralelismo em múltiplos núcleos de CPU, alcançou speedup de cerca de 6,0 vezes para grafos de 4096 vértices e de aproximadamente 6,3 vezes para grafos de 8192 vértices, embora a eficiência passe a se estabilizar em instâncias maiores, como indicado pelo ganho de cerca de 4,5 vezes para grafos de 2048 vértices.

A implementação com MPI, baseada em processos distribuídos, mostrou-se particularmente eficiente para grafos pequenos e médios, atingindo um speedup de aproximadamente 4,9 vezes para 512 vértices e 5,3 vezes para 1024 vértices. Esses resultados indicam que, mesmo na presença de sobrecarga de comunicação, o MPI constitui uma solução robusta para exploração de paralelismo distribuído em sistemas com múltiplos nós, sobretudo quando o tamanho do problema ainda não justifica o uso de arquiteturas massivamente paralelas.

Em GPUs, a implementação em CUDA obteve o melhor desempenho para grafos de grande escala, destacando-se o caso de 16384 vértices, em que se observou um speedup da ordem de 38 vezes em relação à versão sequencial. Embora a sobrecarga de inicialização do dispositivo e de transferência de dados entre CPU e GPU reduza a vantagem em grafos menores, os resultados evidenciam que, a partir de certo porte, o paralelismo massivo da GPU compensa amplamente esses custos adicionais.

De forma geral, a escolha da implementação paralela do algoritmo de Dijkstra deve considerar simultaneamente o tamanho do grafo e os recursos de hardware disponíveis. Abordagens baseadas em CPU, como OpenMP e MPI, mostraram-se adequadas para grafos de pequeno a médio porte, enquanto arquiteturas de GPU, como a utilizada em CUDA, revelaram-se mais eficientes para grafos muito grandes. Assim, combinações que explorem a eficiência das CPUs em problemas menores e a escalabilidade das GPUs em instâncias de grande escala tendem a oferecer soluções mais equilibradas, maximizando o desempenho em diferentes contextos computacionais.

Em termos práticos, para grafos pequenos/médios (até 4k vértices), OpenMP ou MPI em CPU multicore são ideais por simplicidade e eficiência ($> 80\%$ em 4 threads/processos), adequados a redes locais ou clusters acessíveis. Para grafos de grande escala ($> 8k$ vértices), CUDA em GPU é superior, com speedup $> 30\times$, recomendável em aplicações como, por exemplo, roteamento em telecomunicações ou navegação GPS em tempo real, desde que o overhead de setup seja amortizado.

5.1 Limitações do Trabalho

Em relação ao MPI, observou-se um problema de escalabilidade a partir de 4 processos, uma vez que o maior grafo, com 16.384 vértices, não pôde ser alocado na memória do ambiente de execução. Essa limitação torna-se ainda mais evidente quando o número de processos é elevado para 8, caso em que grafos com 8.192 vértices já causam falhas de alocação. No caso da implementação em CUDA, a quantidade de threads por bloco foi fixada em 256, o que pode ser sub-ótimo para arquiteturas de GPUs mais recentes; blocos maiores ou uma configuração mais adaptada ao hardware potencialmente permitiriam reduzir ainda mais o tempo de execução para grafos de grande escala, como o de 16.384 vértices.

Este estudo foi desenvolvido em um contexto acadêmico, com tempo e infraestrutura computacionais limitados, o que restringiu a exploração de configurações adicionais de hardware e de variações do algoritmo. Essas restrições podem ter limitado o alcance dos experimentos realizados e abrem espaço para investigações futuras em ambientes mais robustos.

5.2 Trabalhos Futuros

Considerando as limitações deste estudo, um primeiro desdobramento consiste em investigar estratégias para mitigar os problemas de escalabilidade observados na implementação em MPI, seja por meio de técnicas de particionamento mais eficientes, seja pela redução do custo de comunicação entre processos. Outro caminho promissor é realizar um estudo sistemático sobre o impacto do tamanho dos blocos na execução em GPU, buscando configurações mais adequadas às arquiteturas modernas, incluindo placas com unidades especializadas, como *Tensor Cores*. Adicionalmente, pode-se explorar formas de reestruturação e quantização das estruturas de dados para avaliar a viabilidade de implementar essas soluções em *Single Board Computers* (SBCs). Por fim, a repetição e ampliação dos experimentos em infraestruturas de hardware mais robustas, como servidores dedicados de GPU ou plataformas em nuvem, pode fornecer evidências adicionais sobre o comportamento das abordagens paralelas em cenários de maior escala.

Referências

Awari, R. (2017). Parallelization of shortest path algorithm using openmp and mpi. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 304–309. DOI: 10.1109/I-SMAC.2017.8058360.

Backes, A. R. (2016). *Estrutura de dados descomplicada: em linguagem C*. GEN LTC, Rio de Janeiro.

Calderon, S., Rucci, E., and Chichizola, F. (2024). Enhanced openmp algorithm to compute all-pairs shortest path on x86 architectures. In *arXiv preprint arXiv:2403.18619*. DOI: 10.48550/arXiv.2403.18619.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). *Introduction to Algorithms*. MIT Press.

Farber, R. (2011). *CUDA: Application Design and Development*. Morgan Kaufmann, Burlington, MA.

Hassoun, M. H. and Sanghvi, A. J. (1990). Fast computation of optimal paths in two- and higher-dimension maps. *Neural Networks*, 3(3):355–363. DOI: https://doi.org/10.1016/0893-6080(90)90078-Y.

Noto, M. and Sato, H. (2000). A method for the shortest path search by extended dijkstra algorithm. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0*, volume 3, pages 2316–2320. DOI: 10.1109/ICSMC.2000.886462.

NVIDIA Corporation (2023). *CUDA C++ Programming Guide, Version 12.2*. NVIDIA Corporation. Disponível em: `https://docs.nvidia.com/cuda/`.

OpenMP Architecture Review Board (1997). Openmp: A proposed industry standard API for shared memory programming. Technical report, OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.0.

Sardar, T. H. and Faizabadi, A. R. (2018). Parallelization and analysis of selected numerical algorithms using openmp and pluto on symmetric multiprocessing machine. *Data Technologies and Applications*, 53(1):20–32. DOI: https://doi.org/10.1108/DTA-05-2018-0040.

Silva, G. P., Bianchini, C. P., and Costa, E. B. (2022). *Programação paralela e distribuída: com MPI, OpenMP e OpenACC para computação de alto desempenho*. Casa do Código, São Paulo.

Solka, J. L., Perry, J. C., Poellinger, B. R., and Rogers, G. W. (1995). Fast computation of optimal paths using a parallel dijkstra algorithm with embedded constraints. *Neurocomputing*, 8(2):195–212. DOI: https://doi.org/10.1016/0925-2312(94)00018-N.

Song, B. (2025). High-performance parallelization of dijkstra's algorithm using mpi and cuda. *arXiv preprint arXiv:2504.03667*. DOI: 10.48550/arXiv.2504.03667.