



UNIVERSIDADE ESTADUAL DE MATO GROSSO DO SUL - UEMS

Curso de Ciência da Computação

Disciplina de Algoritmos Paralelos e Distribuídos

Professor: Dr. Rubens Barbosa Filho

Data: 06/05/2026

Ano: 4

Turma: U

Trabalho: Implementação de uma Árvore Geradora Mínima Paralela usando Memória Distribuída.

DESCRIÇÃO DO TRABALHO:

Encontrar uma Árvore Geradora Mínima (*Minimum Spanning Tree*) em um grafo é um problema bem conhecido em teoria dos grafos e suas aplicações como por exemplo, em redes de computadores.

Uma Árvore Geradora Mínima de um grafo ponderado $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ é um subconjunto de \mathbf{E} que forma uma árvore geradora mínima de \mathbf{G} com um peso total mínimo. Problemas que envolvam uma árvore geradora mínima possuem muitas aplicações em computação e no design de redes de comunicação, assim como aplicações indiretas nos campos de visão computacional e análise de *cluster*.

Neste trabalho, o objetivo será o de implementar um algoritmo paralelo para encontrar a árvore geradora mínima de um grafo utilizando a API MPI sobre a linguagem C no cluster do laboratório de Ciência da Computação da UEMS.

Parte I

Algoritmo Sequencial

Uma árvore estendida significa que todos os vértices devem ser conectados. Portanto, os dois subconjuntos disjuntos de vértices devem ser conectados para fazer uma árvore de expansão. Eles devem ser conectados com a borda de peso mínimo para torná-los uma árvore de abrangência mínima.

A seguir tem-se uma figura com o pseudo código do algoritmo:

algoritmo Árvore Geradora Mínima

entrada: Um grafo não direcionado ponderado $G = (V, E)$.

saída: F , uma floresta estendida mínima de G .

Inicialize uma floresta F para (V, E') onde $E' = \{\}$.

concluído := false

enquanto não **concluído** **faça**

 Encontre os componentes conectados de F e atribua a cada vértice seu componente

 Inicialize a aresta de menor peso para cada componente como "Nenhuma"

para cada aresta uv em E , onde uv estão em diferentes componentes de F :

 deixe wx ser a aresta de menor peso para a componente de u

se for-preferido(uv, wx) **então**

 Defina uv como a aresta de menor peso para a componente de u

 deixe yz ser a aresta de menor peso para a componente de v

se for-preferido-sobre(uv, yz) **então**

 Defina uv como a aresta de menor peso para o componente de v

se todos os componentes tiverem a aresta de menor peso definida como "Nenhum" **então**

// nenhuma outra árvore pode ser mesclada -- nós terminamos

complete := true

else

complete := false

para cada componente cuja aresta de menor peso não é "Nenhuma" **faça**

 Adicione sua aresta de menor peso em E'

função for-preferido-sobre($aresta1, aresta2$)

return ($aresta2$ is "None") **ou**

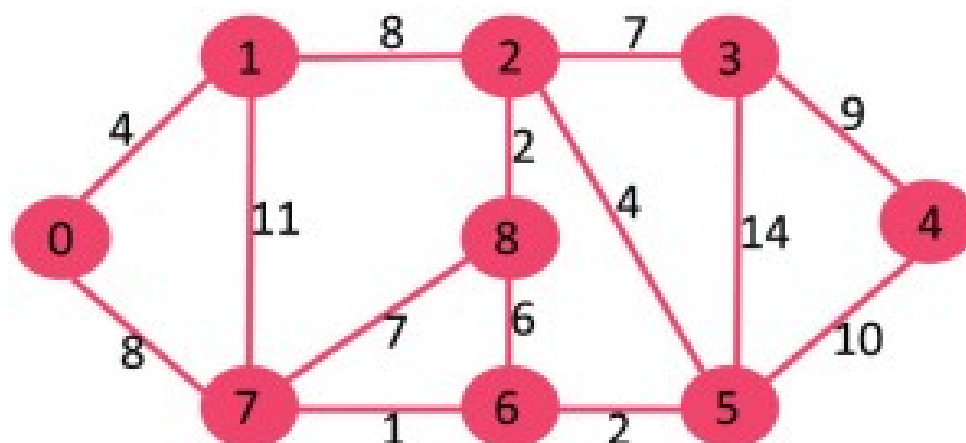
 ($\text{peso}(aresta1) < \text{peso}(aresta2)$) **ou**

 ($\text{peso}(aresta1) = \text{peso}(aresta2)$ e $\text{regra-desempate}(aresta1, aresta2)$)

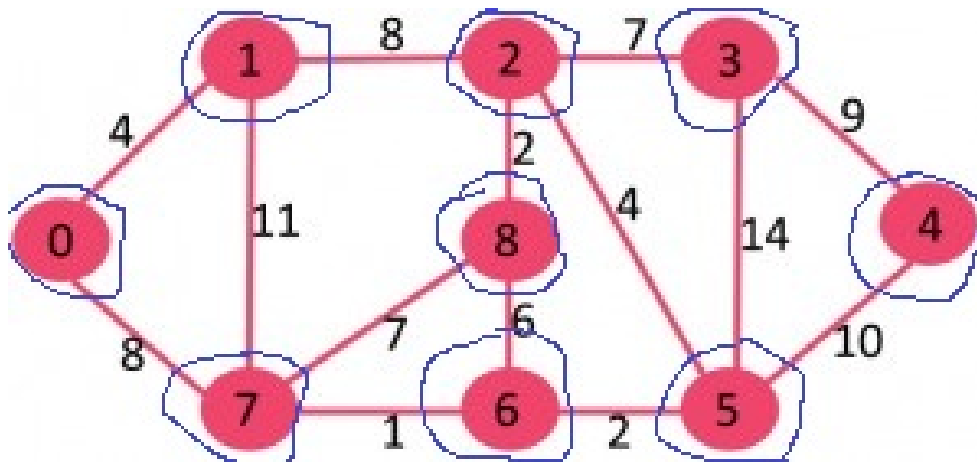
função regra-desempate($aresta1, aresta2$)

 a regra de desempate retorna **true** se e somente se $aresta1$

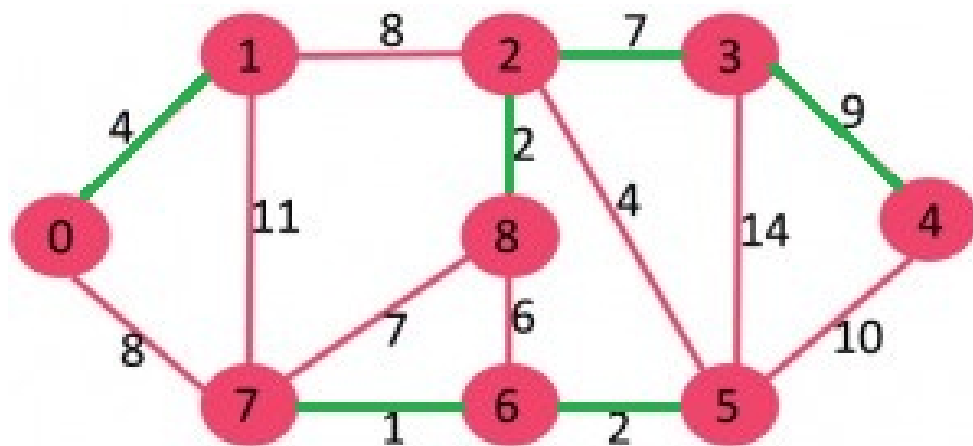
 for-preferido-sobre $aresta2$ no caso de um empate.



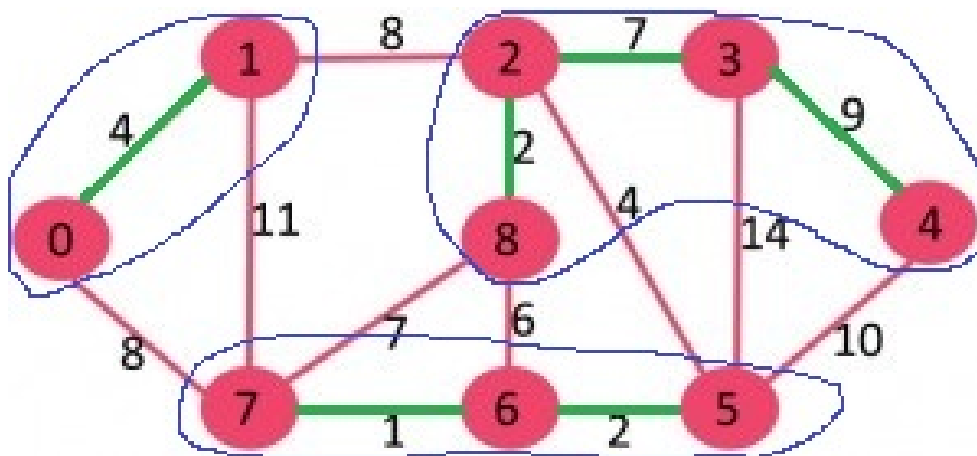
Inicialmente a árvore geradora mínima está vazia. Cada vértice é um único componente conforme destacado na figura a seguir por círculos em azul:



Para cada componente, encontre a borda com o menor peso que o conecta a algum outro componente. No exemplo a seguir, as bordas com menor peso são destacadas em verde.

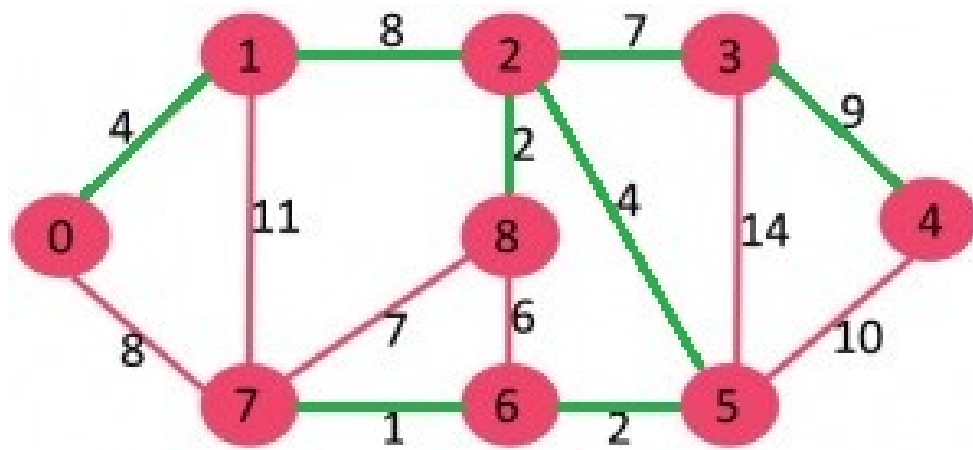


Agora, a árvore geradora mínima torna-se $\{0 - 1, 2 - 8, 2 - 3, 3 - 4, 5 - 6, 6 - 7\}$. Após cada etapa, os componentes passam a ser $\{\{0, 1\}, \{2, 3, 4, 8\}, \{5, 6, 7\}\}$. Os componentes estão circulados com a cor azul.



Repete-se novamente a etapa, isto é, para cada componente, encontra-se a aresta com o menor peso que a conecta a algum outro componente.

As bordas com o menor peso são destacadas em verde na figura a seguir. Agora a árvore geradora mínima torna-se $\{0 - 1, 2 - 8, 2 - 3, 3 - 4, 5 - 6, 6 - 7, 1 - 2, 2 - 5\}$



Neste estágio, há apenas um componente $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ que possui todas as arestas. Como há apenas um componente restante, para-se a execução do algoritmo e retorna a árvore geradora mínima.

Parte II

Algoritmo Paralelo

ALGORITMO PARALELO

Entrada: Grafo não direcionado ponderado $G = (V, E)$ representado por uma matriz de pesos $W(u,v)$.

Saída: Árvore Geradora de Custo Mínimo de G .

Início

para cada $u \in V$ faça em paralelo

$raiz(i) := i$

fim_paralelo

finalizado := Falso

enquanto not (finalizado) faça

 para cada $u \in V$ faça em paralelo

$vertice+proximo(u) := v$ tal que $raiz(u)$ seja diferente de $raiz(v)$ e $W(u,v) := \text{MIN}\{W(u,w) \text{ tal que } w \in V\}$

 fim_paralelo

 para cada componente k da floresta F_i faça em paralelo

 escolha um vertice u tal que $W(u, vertice+proximo(u))$ seja o menor vertice de k

 fim_paralelo

 combine os novos vertices e crie a nova floresta F_{i+1}

 para cada vertice $u \in V$ faça em paralelo

 procure pela $raiz(u)$

 se $raiz(u) == raiz(v)$ para todo $u, v \in V$ então

 finalizado := True

 fim_paralelo

fim_enquanto

Fim

1 Explicando o Algoritmo Paralelo

A floresta F_i contém várias árvores. Uma árvore é representada pela relação de seus pais. A estrutura de dados **vertice+proximo(u)** é usada para denotar o vértice de outra árvore o qual está mais próximo de **u**. **raiz** é outra estrutura de dados a qual fornece a raiz de uma árvore para a qual o nó pertence.

A saída deste algoritmo será em duas partes. Na primeira parte o algoritmo deverá exibir a distância da árvore mínima gerada, isto é, a somatória dos pesos das arestas dessa árvore. A segunda saída deverá ser o caminho dessa árvore. Essa saída poderá ser do tipo: nó x → aresta → nó y → aresta → nó z → aresta → nó w.....

Parte III

Objetivos

Este trabalho tem como objetivo implementar o algoritmo paralelo descrito na parte II com a função de encontrar a árvore geradora mínima de um grafo utilizando a API MPI sobre a linguagem C no cluster do laboratório de Ciência da Computação da UEMS.

O trabalho deve possuir uma versão sequencial implementada. A versão sequencial está descrita na parte I denominada **Algoritmo Sequencial**. Lembre-se que você precisa dos dados fornecidos pela parte sequencial para poder calcular o *speedup*. Assim sendo, a parte sequencial deverá ser executada em apenas uma máquina.

Junto ao arquivo de descrição do trabalho serão fornecidos os arquivos de entrada do algoritmo.

Observação: O algoritmo sequencial executa com uma complexidade $O(V)$ onde V é a quantidade de vértices. Desta forma, a saída será uma linha constante. O que se procura com o trabalho é verificar em que ponto o gráfico obtido com o paralelismo cruza essa linha.

A parte paralela deverá ser executada em 4, 8, 12 e 16 máquinas.

O seu algoritmo escalado para 4, 8, 12 e 16 máquinas deve ter pelo menos uma configuração com eficiência $\geq 70\%$. O trabalho deve ser feito em linguagem C (já que estamos buscando velocidade de execução) e, podem ser utilizadas todas as funções, sejam elas ponto-a-ponto ou coletivas.

Junto ao trabalho são fornecidos dois *setup* de execução. A leitura e distribuição desses dados deve ser feita utilizando MPI. Há um capítulo no livro só sobre I/O.

Para o código sequencial ha um setup menor identificado e, este **setup** menor, caso queira, pode ser usado para **TESTAR** o seu código paralelo. Uma vez que esse código menor passe no seu código paralelo, use o *setup* maior para gerar os dados de execução do seu trabalho.

Parte IV

O que entregar

1. O código e a documentação devem ser entregues em um arquivo Zip. Dentro deste arquivo Zip devem conter um readme.txt com o nome do autor e o comando para a execução do código e, um arquivo PDF da documentação. Inclua todos os arquivos fontes (.c, .h, makefile, não incluam executáveis ou arquivos objetos), em um único diretório. Um Makefile deve ser fornecido para a compilação do código.

2. O nome do arquivo deve seguir o seguinte padrão: *NomeAluno_RGM_NomeArquivo.c* para arquivo fonte.
3. Submissões onde os programas não sigam as especificações de parâmetro e nome, makefiles não funcionando ou arquivos necessários faltando **NÃO SERÃO CORRIGIDOS**. Programas que não compilem também não serão corrigidos.
4. Gráfico comparativo de tempo entre a execução sequencial e a execução paralela (para 4, 8, 12 e 16 máquinas);
5. Gráfico comparativo de tempo com somente a parte paralela (para 4, 8, 12 e 16 máquinas);
6. Gráfico com o cálculo do *Speedup*. Pode ser todas as execuções paralelas em um único gráfico.
7. Gráfico com o cálculo da **Eficiência**. Pode ser todas as execuções paralelas em um único gráfico.
8. Gráfico de Amdahl ou Gustafson Barsis. Escolha um dos dois.
9. *Log* de execução de cada máquina em separado com as informações que você usou para depurar seu código.

Parte V

Documentação

O texto da documentação deve ser breve, de forma que o professor possa entender o que foi feito no código sem ter que entender linha a linha dos arquivos. Implementações modularizadas deverão mencionar quais funções são implementadas em cada módulo. A documentação deverá conter os seguintes itens:

1. O sumário do problema a ser tratado;
2. Uma descrição sucinta dos algoritmos e dos tipos abstratos de dados, das principais funções, procedimentos e das decisões de implementação;
3. Decisões de implementação que porventura estejam omissos na especificação;
4. Como foi tratada a comunicação entre as máquinas;
5. As características da máquina usada na execução do código sequencial (clock, memória, entre outros).
6. Testes, mostrando que o programa está funcionando de acordo com as especificações, seguidos de sua análise;
7. Print screens mostrando o correto funcionamento do simulador e exemplos de testes executados;
8. Conclusão e referências bibliográficas.

Parte VI

Avaliação

A avaliação do trabalho será composta pela execução dos programas desenvolvidos e pela análise da documentação. Todos os alunos deverão apresentar e explicar em laboratório o trabalho.

Os seguintes itens serão avaliados:

1. A qualidade do código (código bem organizado, estruturado, com comentários explicativos, variáveis com nomes intuitivos, modularidade, etc);
2. Execução correta do código com entrada de testes a serem definidas no momento da avaliação. As entradas de testes irão exercitar a funcionalidade completa do código e, testar casos especiais ou de maior dificuldade de implementação, mas que devem ser tratados por um programa correto;
3. Conteúdo da documentação que deve conter os itens mencionados anteriormente;
4. Coerência e coesão da documentação (apresentação visual e organização, uso correto da língua portuguesa, qualidade textual e facilidade de compreensão);
5. CASOS DE CÓPIA NÃO SERÃO TOLERADOS;
6. Qualquer elemento que não esteja especificado neste documento, mas que tenha que ser inserido para que o simulador funcione, deve ser descrito na documentação de forma explícita.
7. Lembre-se que não estamos fazendo paralelismo de software, ou seja, nada de *threads*.

Parte VII

Data de entrega

Dia 01/07/2026 - quarta feira.

O trabalho deve ser feito individualmente.

Obs.: Este trabalho não será adiado.