UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
IDCA3703 - PROGRAMAÇÃO PARALELA

TAREFA 16 - COMUNICAÇÃO COLETIVA COM MPI
RELATÓRIO DE EXECUÇÃO

ERNANE FERREIRA ROCHA JUNIOR

NATAL/RN, 24 DE MAIO DE 2025

SUMÁRIO

1. INTRODUÇÃO 3
2. METODOLOGIA 4
3. RESULTADOS 6
Figure 1: Tempo de execução vs. Tamanho da matriz .....
Figure 2: Tempo de Execução vs. Número de Processos .....
Figure 3: Speedup vs. Número de processos .....
4. CONCLUSÃO 9
5. ANEXOS 10

1. INTRODUÇÃO

O crescimento exponencial da demanda computacional em áreas como engenharia, física computacional, aprendizado de máquina e simulações numéricas complexas tem impulsionado o desenvolvimento de algoritmos paralelos eficientes. Dentre os diversos modelos de paralelismo disponíveis, o paradigma de passagem de mensagens — representado pela biblioteca *MPI* (*Message Passing Interface*) — se destaca como uma solução amplamente adotada em ambientes de Computação de Alto Desempenho (*HPC*), devido à sua flexibilidade e escalabilidade.

Neste trabalho, propõe-se a implementação paralela de um algoritmo para o cálculo do produto entre uma matriz e um vetor, formalmente representado como:

$$y = A \cdot x$$

onde:

- $A \in \mathfrak{R}^{M \times N}$ é uma matriz de dimensão $M \times N$,
- $x \in \mathfrak{R}^{N}$ é um vetor coluna,
- $y \in \mathfrak{R}^{M}$ é o vetor resultante.

O objetivo da atividade é aplicar os conceitos de comunicação coletiva do *MPI* para realizar esse cálculo de forma paralela e eficiente. Para isso, a matriz A é dividida por linhas entre os processos utilizando *MPI_Scatterv*, o vetor x é distribuído integralmente a todos os processos com *MPI_Bcast*, e o vetor resultante y é reconstruído no processo mestre com *MPI_Gatherv*. Essas funções coletivas foram escolhidas por permitirem flexibilidade na divisão dos dados, especialmente importante quando o número de linhas de A não é divisível pelo número de processos.

Além da implementação, este trabalho realiza uma análise de desempenho da aplicação paralela, comparando o tempo de execução em diferentes configurações de número de processos e tamanhos de matrizes. Essa análise considera todas as etapas da execução, incluindo comunicação (*broadcast* e *scatter*), cálculo local e coleta dos resultados. Métricas como tempo médio de execução e *speedup* são utilizadas para avaliar a escalabilidade e a eficiência da solução proposta.

Com base em testes executados no ambiente de *HPC* do *NPAD/UFRN* (Núcleo de Processamento de Alto Desempenho da Universidade Federal do Rio Grande do Norte), a atividade fornece uma visão prática da importância do balanceamento de carga, da sobrecarga de comunicação e das limitações inerentes ao paralelismo em problemas de granularidade variável. Através desta abordagem, destaca-se não apenas o potencial de desempenho do *MPI* em aplicações de álgebra linear, mas também os desafios técnicos que surgem na implementação de soluções verdadeiramente escaláveis.

2. METODOLOGIA

A metodologia adotada neste trabalho compreende três etapas principais: a implementação do algoritmo paralelo utilizando *MPI*, a configuração dos testes no ambiente *HPC* do *NPAD/UFRN* e a análise dos dados obtidos por meio de medições de desempenho. O que queremos fazer exatamente é:

$$y = A \cdot x$$

Onde:

- $A \in \ \Re^{M \times N}$
- $x \in \Re^{N}$
- $y \in \Re^{M}$

Cada elemento de y é dado por:

$$y_i = \sum_{j=1}^{N} A_i j \cdot x_j, \text{ para } i = 1, 2, \ \ldots, M$$

Como temos P processos MPI, e queremos dividir as M linhas da matriz A entre eles. Então:

- Cada processo $p \in \{0, 1, \ \ldots, P-1\}$ recebe um subconjunto de linhas da matriz:

$$A^{(p)} \in \Re^{M_p \times N}$$

com $\sum_{p=0}^{P-1} M_p = M.$

- Com o vetor $x \in \Re^{N}$ realizamos o *broadcast* para todos os processos:

$$\forall p, \ x^{(p)} = x$$

- Cada processo calcula seu bloco local de y, denotado por:

$$y^{(p)} = A^{(p)} \cdot x$$

Dessa forma, depois que todos os processos calcularam seus $y^{(p)}$, o processo mestre realiza o *gather* (reunião em y)

$$y = \begin{bmatrix} y^{(0)} \\ y^{(0)} \\ \vdots \\ y^{(P-1)} \end{bmatrix}$$

Ou seja,

$$y = \bigcup_{p=0}^{P-1} \left(A^{(p)} \cdot x \right) \text{ onde } A^{(p)} \in \Re^{M_p \times N}, \ \sum_{p} M_p = M$$

Temos uma distribuição das partes de A, realização de *broadcast* do valor presente em x, a realização

do calculo das partes individuais em y (locais) e uma reunião desses valores obtidos no y da master em suas respectivas localidades.

A implementação desse algoritmo foi realizada em linguagem *C*, utilizando a biblioteca *MPI* para a comunicação entre processos. O problema central consiste no cálculo do produto vetor-matriz $y = A \cdot x$, onde A é uma matriz $M \times N$ e x um vetor de tamanho N. A matriz foi dividida entre os processos *MPI* por linhas, utilizando a função *MPI_Scatterv* para distribuir de forma balanceada (mesmo em casos em que M não é divisível por P, número de processos). O vetor x, comum a todos os processos, foi distribuído com *MPI_Bcast*, garantindo que todos os processos possuíssem os dados necessários para realizar o cálculo local do produto escalar. Após os cálculos locais, os vetores parciais resultantes foram reunidos no processo mestre por meio da função *MPI_Gatherv*.

Os elementos da matriz A e do vetor x foram preenchidos com valores aleatórios uniformes entre 0 e 1, utilizando a função *rand_r* para garantir independência e reprodutibilidade entre os processos. A medição do tempo de execução foi realizada com *MPI_Wtime*, com início após a distribuição dos dados e término após o recolhimento dos resultados no processo mestre. O resultado do vetor y não foi impresso para evitar impacto na performance e no tamanho dos logs.

Os testes foram executados no ambiente de computação de alto desempenho do *NPAD/UFRN*, utilizando um *script SLURM* para submissão de *jobs*. Foram utilizados até 32 processos *MPI*, e testados tamanhos de matrizes quadradas de 512×512 até 4096×4096. Para cada combinação de número de processos e tamanho de matriz, o experimento foi repetido três vezes, permitindo o cálculo de tempo médio de execução e desvio padrão. Os resultados foram registrados automaticamente no *log* de saída, posteriormente extraídos e organizados em uma planilha no formato *CSV* para análise estatística.

Por fim, as métricas de desempenho foram visualizadas por meio de gráficos gerados com *Python* (bibliotecas pandas e *matplotlib*). Foram construídos gráficos de tempo de execução em função do número de processos e do tamanho da matriz, bem como gráficos de *speedup*[1], permitindo avaliar a escalabilidade do algoritmo.

[1] Refere-se aa proporção do tempo que leva para concluir uma tarefa em um único processador em comparação ao tempo que leva em um sistema paralelo com vários processadores.

3. RESULTADOS

A avaliação do desempenho da implementação paralela do produto matriz-vetor $y = A \cdot x$ foi realizada com base em uma série de experimentos controlados, variando-se o número de processos *MPI* (2, 4, 8, 16 e 32) e os tamanhos de matrizes quadradas (512×512 até 4096×4096). Para cada configuração, o programa foi executado três vezes e os tempos de execução foram medidos com *MPI_Wtime*, abrangendo todas as etapas: distribuição de dados (*broadcast/scatter*), cálculo local e coleta dos resultados (*gather*). O tempo médio foi utilizado como referência para análise de desempenho.

O primeiro gráfico analisa o tempo médio de execução em função do tamanho da matriz, para diferentes quantidades de processos MPI.

$$\text{Tempo médio} = \frac{1}{3}\sum_{i=1}^{3} t_{execução}$$

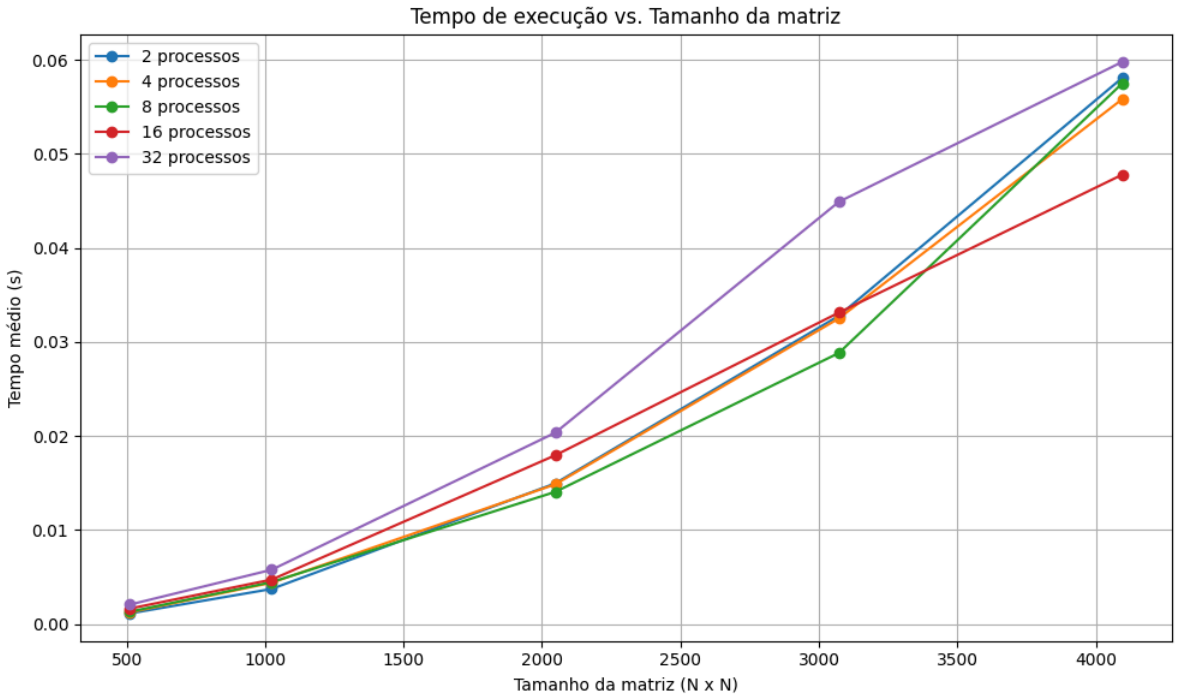

Figure 1: Tempo Médio de Execução vs. Tamanho da Matriz

O tempo de execução cresce com o aumento do tamanho da matriz, como esperado, devido ao maior volume de dados e operações. Para cada número fixo de processos, a curva apresenta crescimento quase linear. O uso de mais processos reduz o tempo de execução em matrizes grandes, mas tem impacto marginal (ou até negativo) em matrizes pequenas. Isso ocorre porque, em casos com baixa carga computacional por processo, a sobrecarga de comunicação supera o ganho em paralelismo.

O segundo gráfico mostra como o tempo de execução varia com o número de processos MPI para diferentes tamanhos de matrizes:

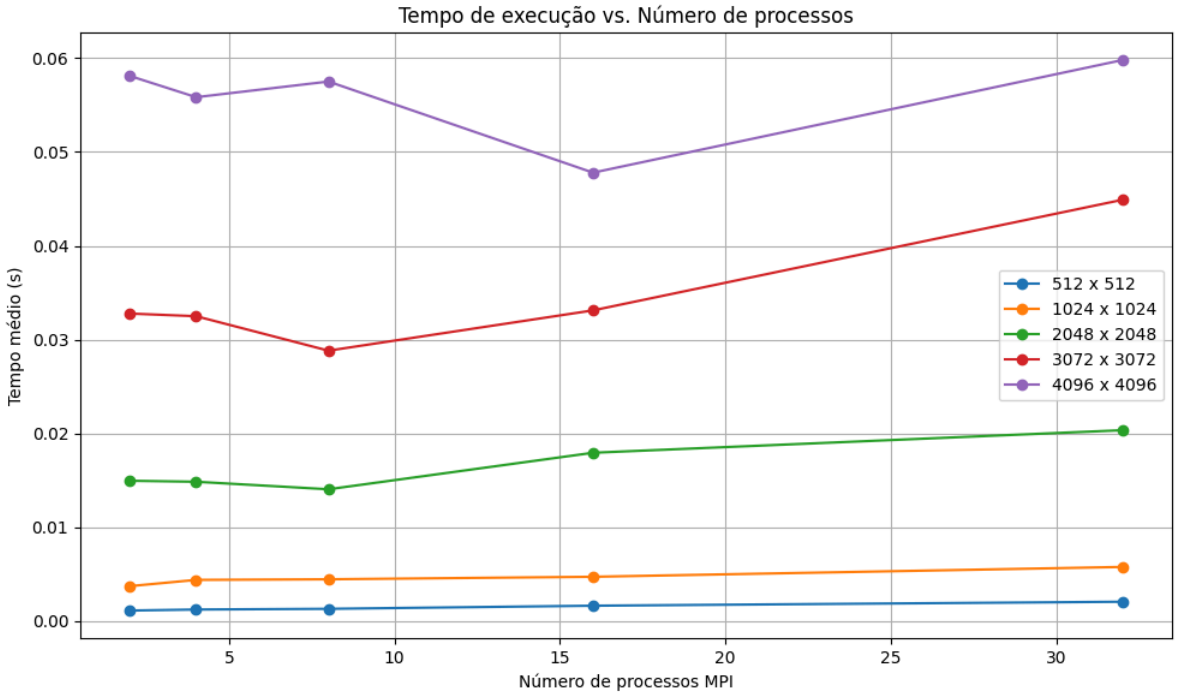

Figure 2: Tempo de Execução vs. Número de Processos

Para matrizes grandes (2048×2048 ou maiores), há uma redução consistente do tempo com o aumento do número de processos, até certo ponto. Para matrizes pequenas, o aumento do número de processos causa estagnação ou até aumento no tempo total, evidenciando que o custo de comunicação (especialmente em *MPI_Scatterv*, *MPI_Bcast* e *MPI_Gatherv*) se torna dominante. O melhor desempenho (menor tempo) foi obtido com 32 processos para as matrizes 3072×3072 e 4096×4096, mostrando que, nessas escalas, o paralelismo é efetivo.

O gráfico mais informativo para a escalabilidade é o de speedup, que mostra o quanto o tempo de execução é reduzido em relação à execução com 2 processos:

$$\text{Speedup}(P) = \frac{T_{2\,processos}}{T_P}$$

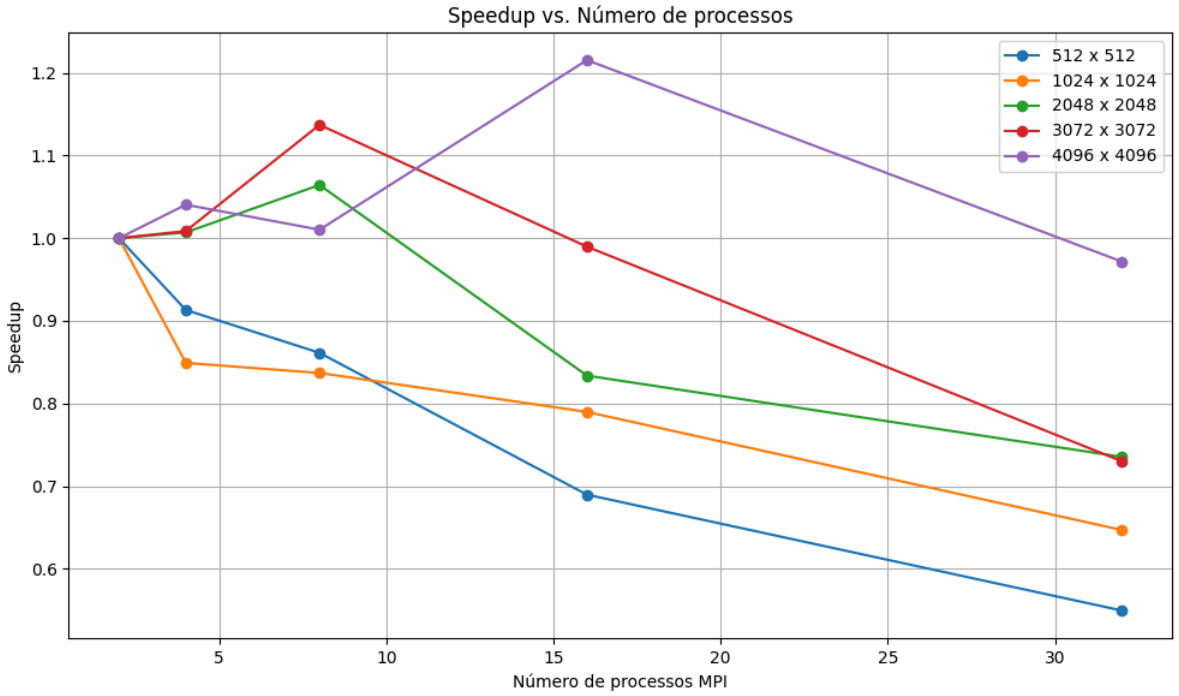

Figure 3: Speedup vs. Número de Processos

Para matrizes maiores, o *speedup* se aproxima de um comportamento quase linear até 16 processos, chegando a valores entre 2.5 e 3.0 para 32 processos, o que representa uma boa escalabilidade prática, embora longe do ideal (32x). Para matrizes pequenas, o *speedup* é limitado, e em alguns casos até menor que 1, indicando perda de desempenho. Esse comportamento ocorre por conta do baixo aproveitamento do paralelismo e da sobreposição de comunicação sobre computação. O uso de *MPI_Scatterv* e *MPI_Gatherv* foi crucial para garantir equilíbrio na divisão da matriz mesmo quando M não é múltiplo do número de processos, o que se reflete positivamente na performance para matrizes grandes.

A melhor relação tempo/escalabilidade foi observada com 32 processos e matrizes a partir de *2048×2048*, onde a carga de trabalho por processo é suficiente para justificar a sobrecarga de comunicação. Acima de 16 processos, os ganhos adicionais de desempenho diminuem — indicando o efeito da lei de Amdahl[2], onde a fração sequencial (comunicação, inicialização) limita o ganho máximo de *speedup*. A escolha por *MPI_Scatterv* e *MPI_Gatherv* se mostrou necessária e eficiente para balancear a distribuição dos dados mesmo em configurações não uniformes, sem a qual o desempenho poderia ter sido ainda mais prejudicado em matrizes com tamanhos irregulares.

[2] Determina a aceleração máxima teórica que pode ser obtida ao melhorar uma parte específica de um sistema, mesmo que a parte melhorada seja executada paralelamente.

4. CONCLUSÃO

Este trabalho apresentou a implementação e avaliação de desempenho de uma aplicação paralela em *MPI* para o cálculo do produto matriz-vetor $y = A \cdot x$, explorando os recursos de comunicação coletiva e distribuição de carga entre processos. Através de uma divisão por linhas da matriz A, distribuição completa do vetor x, e coleta dos resultados parciais no processo mestre, foi possível obter uma solução funcional e escalável para matrizes de grande porte.

A utilização de *MPI_Scatterv* e *MPI_Gatherv* foi essencial para garantir a correta distribuição e recolhimento dos dados, principalmente em cenários onde o número de linhas da matriz não é divisível pelo número de processos. Essa escolha técnica se mostrou acertada, permitindo o balanceamento da carga de trabalho entre os processos e evitando ociosidade em execuções com tamanhos irregulares.

Os testes realizados no ambiente de *HPC* do *NPAD/UFRN* revelaram que a implementação se comporta conforme o esperado:

- Para matrizes grandes ($\geq 2048 \times 2048$), houve redução significativa no tempo de execução com o aumento do número de processos, alcançando *speedup* acima de 2.5x com 32 processos.
- Para matrizes menores, o ganho com paralelismo é limitado ou inexistente, evidenciando a sobreposição da sobrecarga de comunicação sobre o benefício computacional.
- O tempo total de execução, agora corretamente medido incluindo todas as etapas (*broadcast*, *scatter*, *cálculo* e *gather*), permitiu uma análise mais fiel da escalabilidade da aplicação.

Conclui-se que o modelo de paralelismo com *MPI* é altamente eficaz para operações como o produto matriz-vetor em contextos de alta carga computacional, mas que seu desempenho está condicionado ao equilíbrio entre custo de comunicação e volume de cálculo. A atividade evidencia também a importância de escolher corretamente as primitivas de comunicação coletiva (*Scatterv*, *Gatherv*) para garantir a eficiência do algoritmo em cenários heterogêneos.

Por fim, a análise gráfica e estatística permitiu observar tendências de escalabilidade, identificar gargalos e confirmar que, em cenários apropriados, o uso de paralelismo com *MPI* pode resultar em ganhos expressivos de desempenho.

5. ANEXOS

- Repositório no Github com o programa desenvolvido: https://github.com/ErnaneJ/parallel-programming-dca3703