Relatório tarefa 17: Tipos Derivados em MPI - Multiplicação Paralela de Matriz-Vetor com Distribuição por Colunas

Aluno: Cristovao Lacerda Cronje

1. Introdução

Nesta atividade, realizamos a implementação e análise de um código paralelo para multiplicação de matriz-vetor usando MPI, com a distribuição dos dados feita por colunas da matriz. Os testes foram realizados no cluster NPAD, com 8 nós e 1 tarefa por nó, variando o número de processos (1, 2, 4 e 8) e o tamanho da matriz M×N (1000×1000, 5000×5000 e 10000×10000). A partir dos dados de tempo, speedup e eficiência, discutimos o comportamento da aplicação, os efeitos da paralelização e as diferenças entre distribuição por linhas e por colunas.

2. Objetivos

- Implementar a multiplicação paralela de matriz-vetor distribuindo colunas entre os processos.
- Avaliar o impacto do número de processos no tempo de execução.
- Analisar o speedup e eficiência para diferentes tamanhos de matriz.
- Comparar o desempenho e a localidade de acesso à memória entre distribuição por linhas e por colunas.
- Estudar as cláusulas MPI utilizadas e sua influência no desempenho.

3. Metodologia

3.1 Estrutura do Código

Nesta abordagem, a matriz foi dividida em blocos de colunas e distribuída entre os processos. Cada processo recebeu as colunas correspondentes e um segmento do vetor x. Com essa distribuição, cada processo calcula uma contribuição parcial para todos os elementos do vetor resultado y_local. Para obter o vetor final y, utilizou-se a operação MPI_Reduce com MPI_SUM para somar os vetores parciais no processo root.

3.2 Cláusulas MPI utilizadas

Cláusula / Função	**Função**	**Efeito na Implementação**
MPI_Init	Inicializa o ambiente MPI	Permite a execução paralela e coordenação entre processos.
MPI_Comm_rank	Obtém o ID do processo	Cada processo identifica sua parte da matriz e do vetor para cálculo.
MPI_Type_vector	Cria um tipo derivado para colunas da matriz	Define a estrutura não contígua das colunas, com stride* para acesso correto.
MPI_Type_create_resized	Ajusta o tamanho do tipo derivado	Permite envio dos blocos de colunas com MPI_Scatter como se fossem contíguos.
MPI_Type_commit	Finaliza o tipo derivado criado	Torna o tipo utilizável em comunicação.
MPI_Scatter	Distribui os blocos de colunas e segmentos de vetor x	**Comunicação bloqueante**.Cada processo recebe suas colunas e parte de x para cálculo local.
MPI_Reduce com MPI_SUM	Soma os vetores parciais y_local no root	**Comunicação bloqueante** para agregar resultados; o root espera todos os processos.
MPI_Finalize	Finaliza o ambiente MPI	Encerra a execução paralela.

*stride: intervalo ou distância em memória entre elementos consecutivos de um conjunto de dados quando eles não estão armazenados de forma contígu).

3.3 Distribuição de Colunas e Cálculo Paralelo com MPI

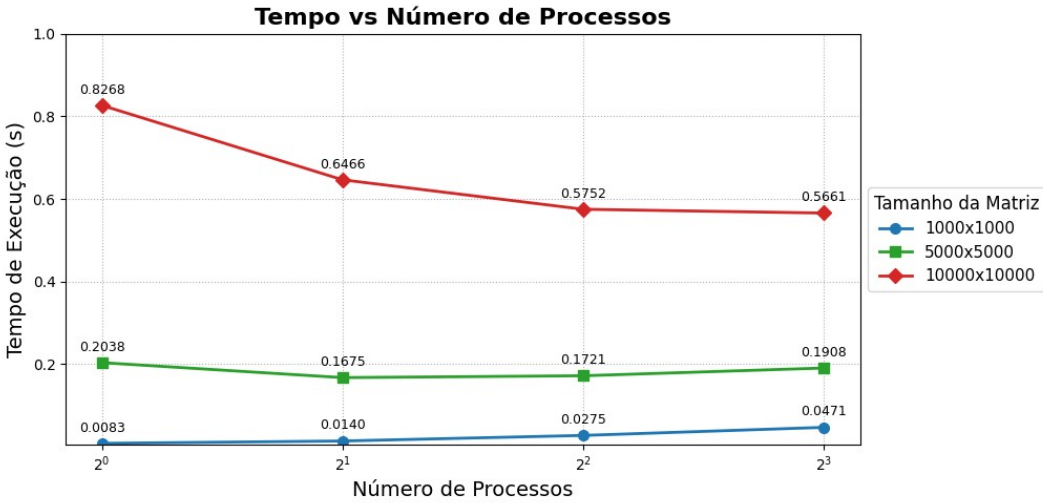
```
// Tipo derivado para uma coluna
MPI_Type_vector(M, 1, N, MPI_DOUBLE, &col_type);
MPI_Type_create_resized(col_type, 0, sizeof(double), &resized_col_type);
MPI_Type_commit(&resized_col_type);
// Scatterv das colunas
MPI_Scatterv(&(A[0][0]), counts, displs, resized_col_type,
             &(local_A[0][0]), local_N * M, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
// Scatterv do vetor x
MPI_Scatterv(x, counts, displs, MPI_DOUBLE,
             local_x, local_N, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
// Inicializa o vetor parcial
for (int i = 0; i < M; i++) {
    partial_y[i] = 0.0;
    for (int j = 0; j < local_N; j++) {
        partial_y[i] += local_A[i][j] * local_x[j];
    }
}
// Redução das contribuições para y
MPI_Reduce(partial_y, y, M, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

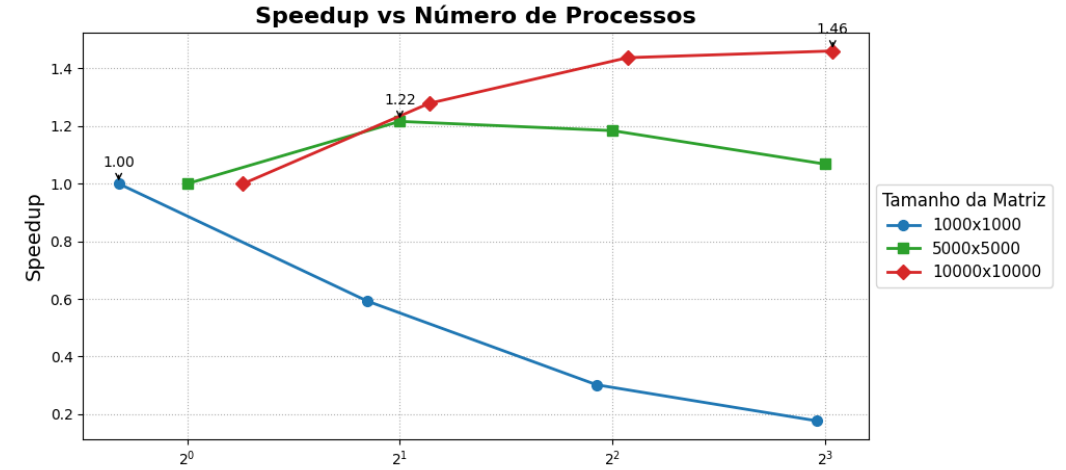
Nesta etapa do programa, implementamos a paralelização da multiplicação matriz-vetor distribuindo **colunas da matriz A** entre os processos. Como as colunas não são contíguas na memória, utilizamos **tipos derivados do MPI** para realizar a comunicação:

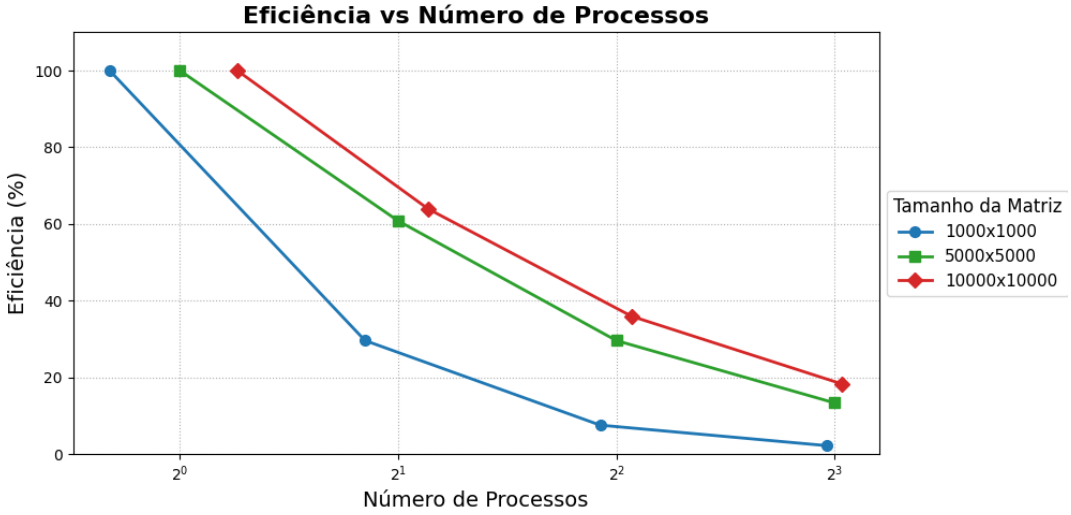
- Foi definido um tipo derivado com MPI_Type_vector, que descreve uma coluna da matriz (elementos espaçados por N posições), e ajustado com MPI_Type_create_resized para garantir o correto espaçamento entre colunas no buffer de envio.
- MPI_Scatterv (variante que permite enviar blocos de tamanhos diferentes a cada processo, mais eficiente para distribuições desiguais) foi usado para distribuir as colunas de A e as partes relevantes do vetor x, respeitando divisões desiguais quando N não é múltiplo do número de processos. Essa escolha reduz o volume de dados transferidos em comparação com MPI_Bcast(usado no vetor na tarefa 16), que enviaria o vetor x completo a todos, simplificando o código mas aumentando o tráfego.
- Cada processo localmente computou a multiplicação de suas colunas com o vetor x, acumulando os resultados parciais em partial_y.
- Por fim, os vetores parciais foram somados com MPI_Reduce, resultando no vetor y final, que representa o produto y=A·x, calculado de forma paralela.

Essa abordagem melhora o desempenho para matrizes com número de colunas elevado, garantindo balanceamento de carga e aproveitamento da comunicação não contígua via tipos derivados.

4. Resultados e Análise







4.1 Resultados obtidos

Matriz	Processos	Tempo (s)	Speedup	Eficiência (%)
1000×1000	1	0.008317	1.00	100.0
	2	0.014040	0.59	29.5
	4	0.027541	0.30	7.5
	8	0.047057	0.18	2.3
5000×5000	1	0.203765	1.00	100.0
	2	0.167535	1.22	61.1
	4	0.172065	1.18	29.5
	8	0.190767	1.06	13.2
10000×10000	1	0.826840	1.00	100.0
	2	0.646599	1.28	64.1
	4	0.575246	1.44	35.9
	8	0.566051	1.46	18.2

4.2 Explicação detalhada dos resultados

- **Matriz 1000×1000:**
  Com matrizes pequenas, a paralelização não trouxe ganhos. O tempo de execução **aumenta com o número de processos**, pois o overhead de comunicação (bloqueante) e sincronização supera o tempo economizado na computação

paralela. Além disso, o **acesso por colunas** tem padrão **não contíguo em memória**, prejudicando o cache. A eficiência despenca, chegando a **2,3% com 8 processos**.

- **Matrizes 5000×5000 e 10000×10000:**
  Há **redução de tempo** ao usar múltiplos processos, mas os ganhos são **modestos**. O MPI_Scatter de colunas e o MPI_Reduce de vetores completos introduzem alto overhead. O acesso colunar ainda limita o desempenho mesmo com maior carga computacional. A **eficiência continua caindo** conforme mais processos são usados.
- **Speedup e Eficiência:**
  O speedup ideal seria proporcional ao número de processos, com eficiência próxima de 100%. No entanto, o **overhead de comunicação**, o **acesso à memória não contíguo** e o custo da **redução global** impedem isso. A escalabilidade é prejudicada, especialmente em arquiteturas distribuídas (1 processo por nó).
- **Comparação com distribuição por linhas (Tarefa 16):**
  Na distribuição por linhas:
  - O acesso à memória é **contíguo**, favorecendo cache e reduzindo *cache misses*.
  - O vetor x é transmitido uma única vez com MPI_Bcast, e **não é necessário reduzir os resultados** com MPI_Reduce, pois cada processo calcula uma parte exclusiva de y.
  - A comunicação é mais eficiente, e a escalabilidade é significativamente melhor.

4.3 Impacto da Comunicação Bloqueante

- Operações como MPI_Scatter e MPI_Reduce são **bloqueantes**: todos os processos precisam aguardar o término da comunicação. Isso cria **pontos de espera** que se tornam gargalos, especialmente com **muitos processos** e **latência de rede entre nós diferentes**.
- A distribuição por colunas agrava esse impacto:
  1. O acesso tem **alto stride**, causando baixa eficiência de cache.
  2. O uso de **tipos derivados** (MPI_Type_vector e MPI_Type_create_resized) exige empacotamento/desempacotamento, aumentando o custo de comunicação.
  3. O MPI_Reduce precisa combinar **vetores inteiros** de cada processo, gerando tráfego elevado.
- Com **mais processos**, aumenta a concorrência na rede e a chance de desequilíbrio — processos mais rápidos esperam pelos mais lentos, aumentando o tempo total.

- Em sistemas com **1 processo por nó**, como no experimento, esse problema é mais visível, pois a comunicação entre nós é bem mais lenta que entre processos locais.
- Embora **comunicações não bloqueantes** (MPI_Isend, MPI_Irecv) pudessem permitir a **sobreposição entre comunicação e computação**, elas **não foram utilizadas** nesta implementação. Seu uso poderia reduzir o tempo ocioso e melhorar o desempenho.

---

5. Conclusão

A distribuição por colunas na multiplicação matriz-vetor com MPI mostrou-se **menos eficiente** devido a dois fatores principais:

- **Baixa localidade de memória**, com acesso não contíguo às colunas.
- **Comunicação bloqueante intensiva**, com MPI_Scatter e MPI_Reduce custosos.

Mesmo com matrizes maiores, os ganhos foram limitados e a **eficiência caiu rapidamente com o número de processos**. A comparação com a distribuição por linhas (Tarefa 16) reforça isso: lá, o **acesso é mais eficiente** e a **comunicação mais simples**, resultando em **melhor desempenho e escalabilidade**.

Recomendações para melhorias:

- Preferir **distribuição por linhas**, mais eficiente em memória e comunicação.
- Avaliar o uso de **comunicação não bloqueante** para reduzir gargalos.
- Reduzir **sincronizações globais** e otimizar o balanceamento de carga.