Relatório: Implementação de Produto Matriz-Vetor Usando MPI

Aluno: Cristovão Lacerda Cronje

1. Introdução

Nesta atividade, realizamos a implementação e análise de um código paralelo para multiplicação de matriz-vetor usando MPI. Os testes foram realizados em um ambiente de cluster com 8 nós e 1 tarefa por nó, variando o número de processos(1, 2, 4 e 8) e o tamanho da matriz MxN(500x500, 1000x1000 e 2000x2000), e vetor N. A partir dos dados de tempo, speedup e eficiência, discutimos o comportamento da aplicação e os efeitos da paralelização.

2. Objetivos

- Implementar a multiplicação paralela de matriz-vetor utilizando MPI.
- Avaliar o impacto do número de processos no tempo de execução.
- Analisar o speedup e eficiência obtidos para diferentes tamanhos de matriz.
- Compreender o efeito do balanceamento de carga e comunicação no desempenho.
- Estudar as cláusulas MPI utilizadas e sua influência na performance.

3. Metodologia

3.1 Estrutura do Código

O programa foi estruturado para dividir a matriz em blocos de linhas, distribuindo-os igualmente entre os processos. Cada processo calcula parte do produto matriz-vetor localmente, e os resultados são coletados no processo root para montagem do vetor resultado final.

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 sabe sua identidade para trabalhar na sua parte da matriz(**consulta local, não envolve comunicação entre processos).**
MPI_Comm_size	Obtém número total de processos	Para dividir a matriz proporcionalmente(**consulta local).**
MPI_Scatter	Distribui blocos da matriz entre processos	Garante que cada processo receba sua parte da matriz para computação local. Comunicação **bloqueante**: os processos esperam até o envio/recebimento dos dados concluírem antes de continuar.
MPI_Bcast	Difunde o vetor para todos os processos	Todos os processos precisam do vetor completo para multiplicação. Comunicação **bloqueante**: todos esperam até a transmissão terminar para prosseguir.
MPI_Gather	Coleta os resultados parciais no processo root	Junta os resultados locais no vetor final. Comunicação **bloqueante**: os processos aguardam até o processo root receber os dados.
MPI_Finalize	Finaliza o ambiente MPI	Libera os recursos MPI e encerra a execução paralela. É **bloqueante** para garantir que todos os processos terminem a execução MPI antes do encerramento do programa.

- **MPI_Isend / MPI_Irecv (comunicação não bloqueante)**:
  Permitem iniciar uma comunicação sem bloquear o processo, possibilitando

sobrepor a comunicação com computação. Isso pode reduzir o tempo total de execução, mas a implementação atual optou por simplicidade, usando comunicação bloqueante.

- **MPI_Barrier**:
  Sincroniza todos os processos, garantindo que todos alcancem um ponto antes de continuar. Não foi utilizada porque a sincronização explícita não era essencial para a lógica do programa atual.
- **MPI_Reduce / MPI_Allreduce**:
  Permitem combinar dados de todos os processos (exemplo: soma, máximo) e distribuir o resultado. Poderiam ser usados para agregar resultados intermediários sem comunicação manual, podendo melhorar a eficiência.
- **MPI_Scatterv**:
  Variante de MPI_Scatter que permite enviar blocos de dados de tamanhos diferentes para cada processo. Útil quando a matriz não é perfeitamente divisível pelo número de processos, permitindo balancear melhor a carga. I**sso evitaria falhas como a ocorrida para a matriz 500x500 com 8 processos.**

Impacto no desempenho:
O uso dessas cláusulas pode melhorar o desempenho do programa ao:

- Permitir melhor balanceamento de carga (MPI_Scatterv), evitando erros e sobrecarga em processos.
- Reduzir o tempo gasto esperando comunicação (MPI_Isend/MPI_Irecv).
- Simplificar agregações de resultados (MPI_Reduce/MPI_Allreduce).
- Controlar pontos de sincronização para evitar condições de corrida (MPI_Barrier).

Isso pode resultar em maior speedup e eficiência, principalmente em ambientes com custo de comunicação alto, como clusters com muitos nós.

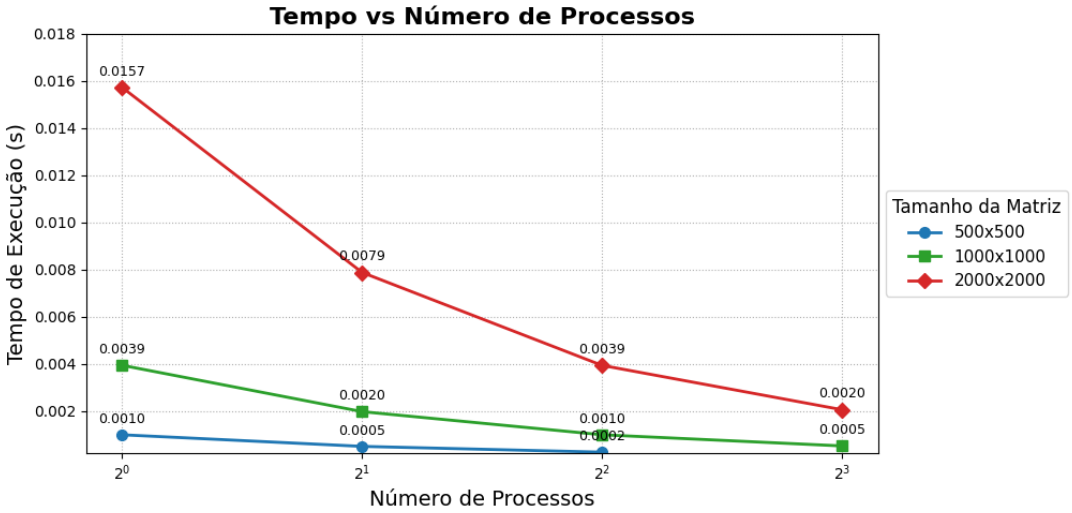
4. Resultados e Análise

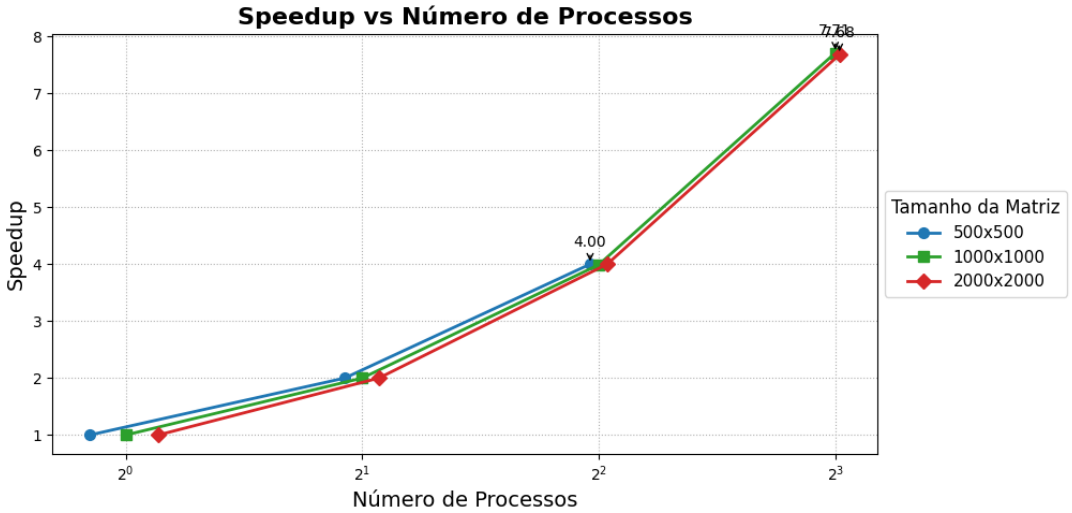
4.1 Tabela resumo de resultados (tempo em segundos)

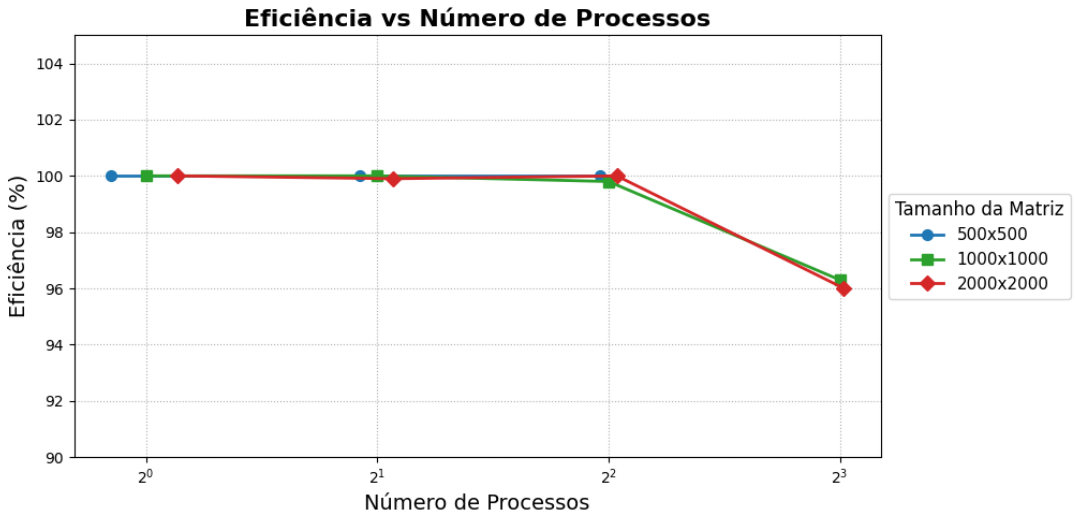
Matriz (MxN)	Processos	Tempo (s)	Speedup	Eficiência (%)
500x500	1	0.000983	1.00	100.0
500x500	2	0.000491	2.00	100.0
500x500	4	0.000246	4.00	100.0
500x500	8	-	-	-
1000x1000	1	0.003930	1.00	100.0
1000x1000	2	0.001965	2.00	100.0
1000x1000	4	0.000984	3.99	99.8
1000x1000	8	0.000510	7.71	96.3
2000x2000	1	0.015719	1.00	100.0
2000x2000	2	0.007870	2.00	99.9
2000x2000	4	0.003928	4.00	100.0
2000x2000	8	0.002047	7.68	96.0

Obs: A execução com 8 processos falhou porque 500 não é divisível por 8(matriz), impedindo divisão igual da matriz; usando **MPI_Scatterv**, seria possível distribuir blocos de tamanhos diferentes para cada processo, permitindo a execução mesmo com tamanhos não divisíveis.

4.2 Gráficos







- Tempo de execução decresce quase linearmente conforme o número de processos aumenta.

- Speedup e eficiência são excelentes (próximos de 100%) para matrizes maiores e até 8 processos.
- Para 8 processos, pequena queda na eficiência indica algum overhead de comunicação.

4.3 Análise Qualitativa

- **Efeito do número de processos**: Aumentar o número de processos reduz o tempo quase proporcionalmente, com speedups próximos do ideal (linear), principalmente para matrizes grandes, pois a computação domina o custo da comunicação.
- **Tamanho da matriz**: Matrizes maiores obtêm maior benefício da paralelização, já que o trabalho é maior e o overhead da comunicação fica menos significativo.
- **NPAD (1 processo por nó)**: A configuração com 1 task por nó facilita o balanceamento e reduz contenção por CPU e memória local, melhorando eficiência e estabilidade dos resultados.
- **Máquinas com menor capacidade**: Em computadores com menos núcleos e memória, a contenção por recursos e o custo da comunicação local aumentariam, tornando a escalabilidade menos eficiente. A diferença seria especialmente visível em matrizes pequenas, onde o tempo de comunicação passa a dominar.
- **Comunicação vs computação**: O speedup decrescente ao usar 8 processos mostra que, a partir de certo ponto, o custo de comunicação entre processos começa a impactar negativamente. Isso se torna mais evidente em matrizes menores ou sistemas com redes mais lentas.
- **Quando as diferenças se tornam significativas**: Os impactos negativos do aumento de processos aparecem especialmente em:
    - Matrizes pequenas (baixo custo computacional, comunicação entre processos se torna dominante.
    - Ambientes com alto custo de comunicação (como redes lentas).
    - Máquinas com poucos núcleos ou memória, onde há contenção por recursos.
    - Implementações que não balanceiam corretamente a carga entre processos.

5. Conclusão

A atividade mostrou que a paralelização com MPI para a multiplicação matriz-vetor é eficiente, especialmente para matrizes grandes. Observou-se speedup e eficiência próximos do ideal, com boa escalabilidade até 8 processos quando o problema é balanceado.

As **cláusulas coletivas do MPI** como MPI_Scatter, MPI_Bcast e MPI_Gather simplificaram a implementação e garantiram desempenho estável. No entanto, para **casos com muitos processos, matrizes menores ou máquinas heterogêneas**, o uso de **comunicação não bloqueante (MPI_Isend/Irecv)** ou **balanceamento dinâmico com MPI_Scatterv** pode melhorar ainda mais a performance, ao reduzir gargalos de comunicação e evitar subutilização de processos.