Universidade Estadual de Santa Cruz

Brenno S. Florêncio e Mateus Soares.

DGEMM Sequencial e Paralela com MPI

Trabalho apresentado à disciplina DEC107 – Processamento Paralelo, do curso de Ciência da Computação da Universidade Estadual de Santa Cruz, como requisito parcial de avaliação, sob orientação do professor Dr. Esbel Tomás Valero Orellana

Ilhéus - BA

01 de Novembro de 2025

1. Introdução

O presente trabalho tem como objetivo implementar e avaliar o desempenho da multiplicação geral de matrizes de precisão dupla (DGEMM) utilizando o padrão MPI (Message Passing Interface).
A proposta dá continuidade ao Projeto 1, no qual foi explorado o paralelismo em memória compartilhada por meio da biblioteca OpenMP, migrando agora para o modelo de memória distribuída, onde a comunicação entre processos ocorre explicitamente através da troca de mensagens.

A multiplicação de matrizes é uma operação essencial em aplicações de computação científica, aprendizado de máquina e simulações numéricas, sendo frequentemente utilizada como métrica de desempenho em sistemas paralelos. O tempo de execução dessa operação depende fortemente da arquitetura de hardware, da organização da memória e da eficiência das estratégias de paralelização empregadas.

Neste projeto, buscou-se avaliar a escalabilidade e o ganho de desempenho obtidos pela versão paralela distribuída implementada com MPI, comparando-a com a versão sequencial (utilizada como referência de corretude) e com os resultados obtidos anteriormente com OpenMP. Além disso, são analisados o impacto do número de processos na eficiência e os efeitos do custo de comunicação inerente ao modelo distribuído.

2. Metodologia

2.1 Implementação Sequencial

A versão sequencial da DGEMM foi implementada em linguagem C, utilizando três laços aninhados para calcular o produto de duas matrizes densas de precisão dupla.
As matrizes A, B e C foram armazenadas como vetores unidimensionais no formato C-style row-major, de modo que o elemento A[i][j]A[i][j]A[i][j] é acessado por `A[i * N + j]`.

O algoritmo base(sem otimização, apenas ilustrativo) segue a estrutura:

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
        double sum = 0.0;
        for (int k = 0; k < N; k++)
            sum += A[i * N + k] *  B[k * N + j];
```

```
            C[i * N + j] = sum;
        }
```

Essa versão serve como referência de corretude e desempenho, sendo utilizada para comparar os resultados numéricos e calcular o speedup das versões paralelas.

2.2 Implementação Paralela com MPI

A versão paralela da multiplicação de matrizes DGEMM foi implementada em linguagem C utilizando a biblioteca MPI (Message Passing Interface), segundo o modelo de memória distribuída. O objetivo é dividir o trabalho de forma equilibrada entre os processos, reduzindo o tempo total de execução em relação à versão sequencial.

A estratégia adotada consiste em realizar a decomposição por blocos de linhas da matriz A, enquanto a matriz B é distribuída e reconstruída integralmente em cada processo por meio de operações coletivas (MPI_Scatter e MPI_Allgather). Essa abordagem garante que todos os processos possuam os dados necessários para calcular suas respectivas porções da matriz resultado C.

Etapas do algoritmo paralelo

1. Inicialização

O ambiente de execução paralelo é iniciado e cada processo obtém seu identificador (rank) e o número total de processos (size):

- ```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  ```

O processo de rank 0 é responsável por gerar as matrizes de entrada A e B com valores aleatórios. As demais instâncias aguardam a distribuição dos dados.

2. Distribuição dos dados

Cada processo recebe um subconjunto de linhas da matriz A, enquanto a matriz B é inicialmente dividida em blocos e, em seguida, reconstruída integralmente em cada processo com MPI_Allgather.

- ```
  MPI_Scatter(A, localN * N, MPI_DOUBLE, ALocal, localN * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(B, localN * N, MPI_DOUBLE, BLocal, localN *
  ```

```
N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Allgather(BLocal, localN * N, MPI_DOUBLE, BFull,
localN * N, MPI_DOUBLE, MPI_COMM_WORLD);
```

Após o Allgather, cada processo possui uma cópia completa da matriz B. Antes do cálculo, B é transposta localmente com o objetivo de melhorar a localidade de acesso à memória durante as operações de multiplicação.

3. Cálculo local (multiplicação parcial)

Cada processo executa a multiplicação de seu bloco de linhas de A pela matriz transposta B^T, gerando um bloco parcial da matriz C. O cálculo utiliza otimização de cache (tiling) e bloqueio em registradores para melhorar o desempenho da multiplicação de matrizes densas.

- ```
  dgemm(N, localN, alpha, ALocal, BTFull, beta, CLocal);
  ```

4. Reunião dos resultados

Após o cálculo local, cada processo envia sua porção da matriz C para o processo raiz (rank 0) por meio da rotina MPI_Gather, que recompõe o resultado completo.

- ```
  MPI_Gather(CLocal, localN * N, MPI_DOUBLE, C, localN * N,
  MPI_DOUBLE, 0, MPI_COMM_WORLD);
  ```

5. Medição de desempenho e finalização

O tempo total de execução é medido com MPI_Wtime(), e as sincronizações entre processos são garantidas com MPI_Barrier() para assegurar medições precisas.

- ```
  MPI_Barrier(MPI_COMM_WORLD);
   double t0 = MPI_Wtime();
   // ... execução do cálculo
   MPI_Barrier(MPI_COMM_WORLD);
   double t1 = MPI_Wtime();
  ```

O programa também calcula métricas de desempenho importantes: tempo sequencial e paralelo, speedup, eficiência e erro relativo máximo para garantir a corretude numérica.
  ```

Rotinas MPI utilizadas e finalidades

Função	Finalidade
`MPI_Init / MPI_Finalize`	Inicializa e encerra o ambiente MPI
`MPI_Comm_rank`	Obtém o identificador do processo atual
`MPI_Comm_size`	Retorna o número total de processos
`MPI_Scatter`	Distribui blocos de linhas de A e B entre os processos
`MPI_Allgather`	Reúne todos os blocos de B para formar a matriz completa em cada nó
`MPI_Gather`	Reúne os blocos parciais de C no processo raiz
`MPI_Barrier`	Sincroniza os processos antes e após as medições
`MPI_Wtime`	Mede o tempo de execução com alta precisão

O modelo foi testado com tamanhos crescentes de matriz (512 a 4096) e com 2, 4 e 8 processos, de modo a observar o impacto da comunicação e do balanceamento de carga no desempenho.

2.3 Hardware e Ambiente de Testes

Os experimentos foram realizados em ambiente WSL2 (Windows Subsystem for Linux), simulando um cluster local com múltiplos processos executando na mesma máquina física.

Componente	Especificação
CPU	Intel Core i5-10500H (6 núcleos / 12 threads, 2.50 GHz)
Memória RAM	4 GB alocada ao WSL2
Sistema Operacional	Ubuntu 24.04 (via WSL2 no Windows 11)
Biblioteca MPI	MPICH 4.2.0
Compilador	mpicc com otimizações `-O3 -march=native -lm`

2.4 Medição de Tempo

O tempo de execução foi medido exclusivamente com a função `MPI_Wtime()`, que fornece a marca de tempo em segundos com resolução adequada para medições curtas.
A medição inclui apenas o tempo de multiplicação, excluindo a inicialização das matrizes e o encerramento do MPI, garantindo a precisão da análise de desempenho da mesma maneira que foi medida no Projeto 1.

Para reduzir o impacto de flutuações de carga do sistema, cada configuração foi executada cinco vezes, e o valor apresentado corresponde à média aritmética das execuções.

2.5 Métricas de Avaliação

Duas métricas principais foram utilizadas para avaliar o desempenho:

1. **Speedup:**
   Sp = Tseq / Tpar(p)
   Mede o ganho de desempenho em relação à execução sequencial.

2. **Eficiência:**
   Ep = Sp / p
   Avalia a fração de tempo efetivamente aproveitada em cada processo.

Valores de eficiência próximos de 1 indicam excelente escalabilidade; valores menores refletem perdas por comunicação, desequilíbrio ou latência.

3. Resultados

3.1 Teste de Corretude

Para garantir a validade da implementação paralela, foi realizada uma comparação elemento a elemento entre os resultados da multiplicação obtidos pela versão MPI e pela versão sequencial. O erro relativo máximo foi calculado como:

$$diff\_max = \max |C\_seq - C\_mpi| / \max(|C\_seq|, \varepsilon)$$

onde $\varepsilon = 10^{-9}$ evita divisões por zero.

Em todos os testes realizados, o valor obtido foi 0.000000e+00, indicando identidade numérica entre os resultados, ou seja, a implementação paralela preserva a precisão do cálculo sequencial.

3.2 Tempos de Execução e Métricas de Desempenho

Os experimentos foram conduzidos com quatro tamanhos de matrizes (512×512, 1024×1024, 2048×2048 e 4096×4096) e três configurações de paralelismo (2, 4 e 8 processos). Cada cenário foi executado cinco vezes para reduzir o efeito de variações do sistema, e os resultados foram consolidados com base nas médias das execuções.

As métricas avaliadas foram:

• Tempo sequencial (T_1) — tempo médio da execução com 1 processo.

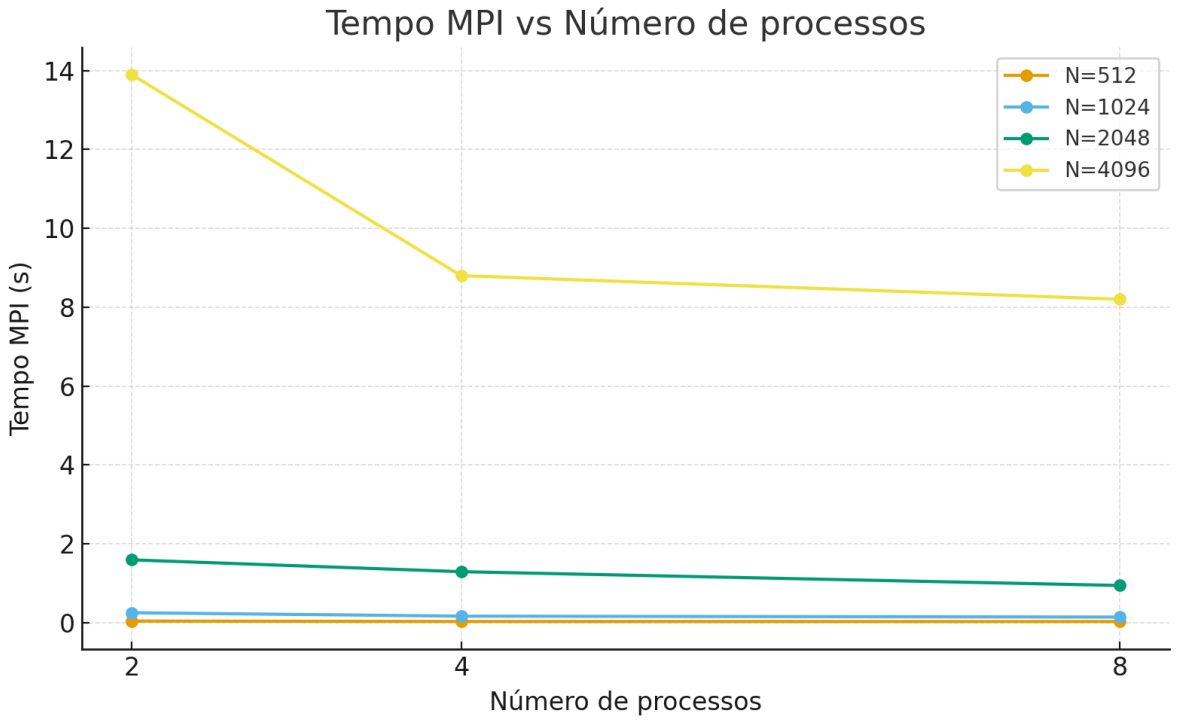
• Tempo paralelo (Tp) — tempo médio da execução com p processos.

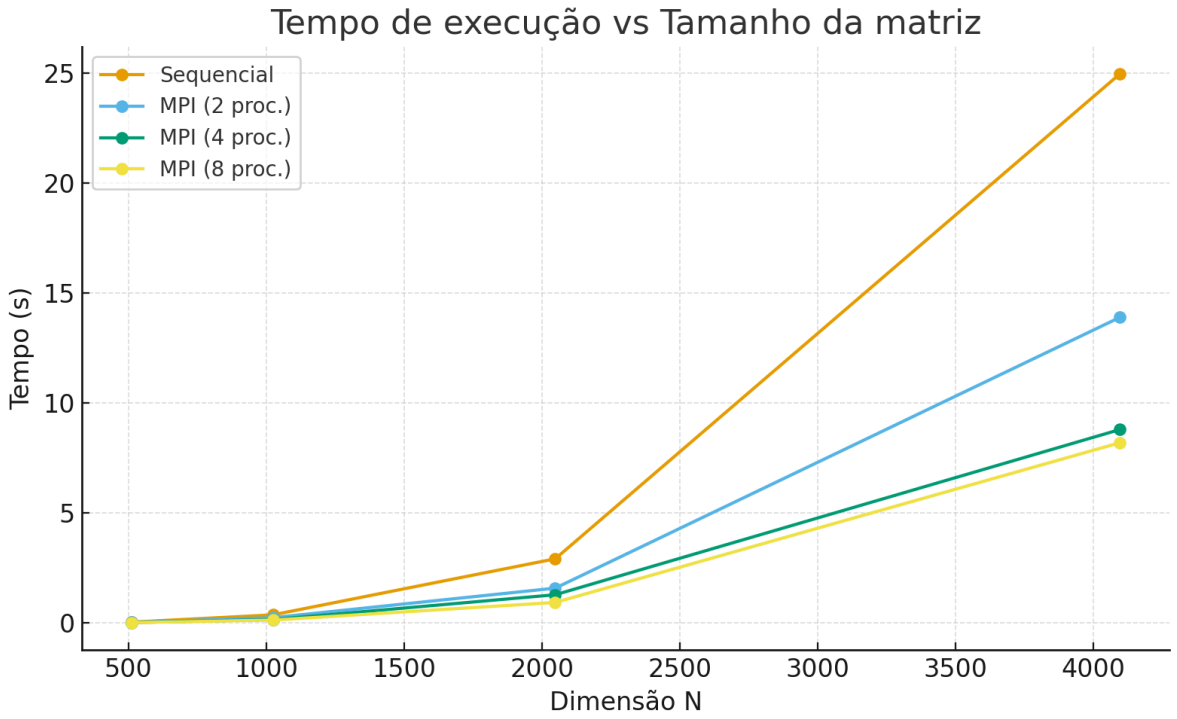
• Speedup ($S_p = T_1 / Tp$) — ganho relativo de desempenho.

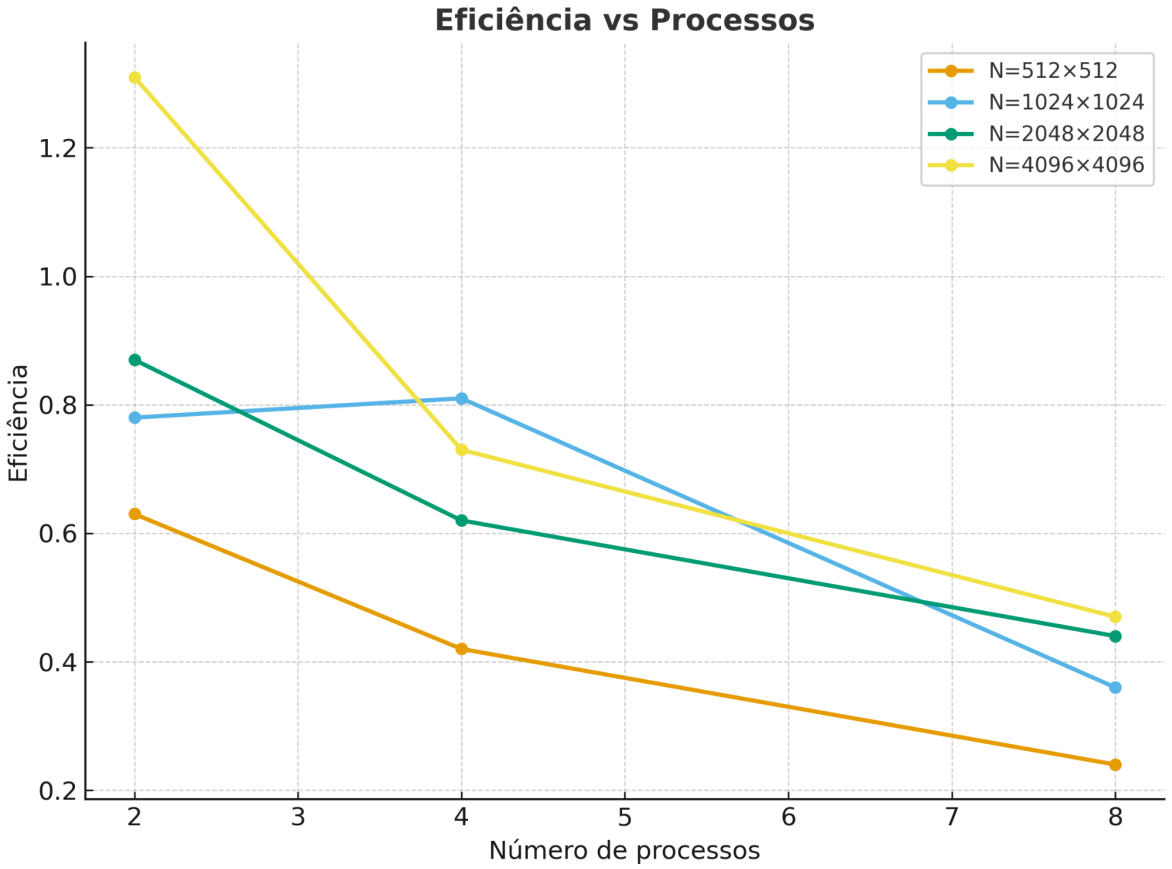
• Eficiência ($E_p = S_p / p$) — grau de aproveitamento do paralelismo.

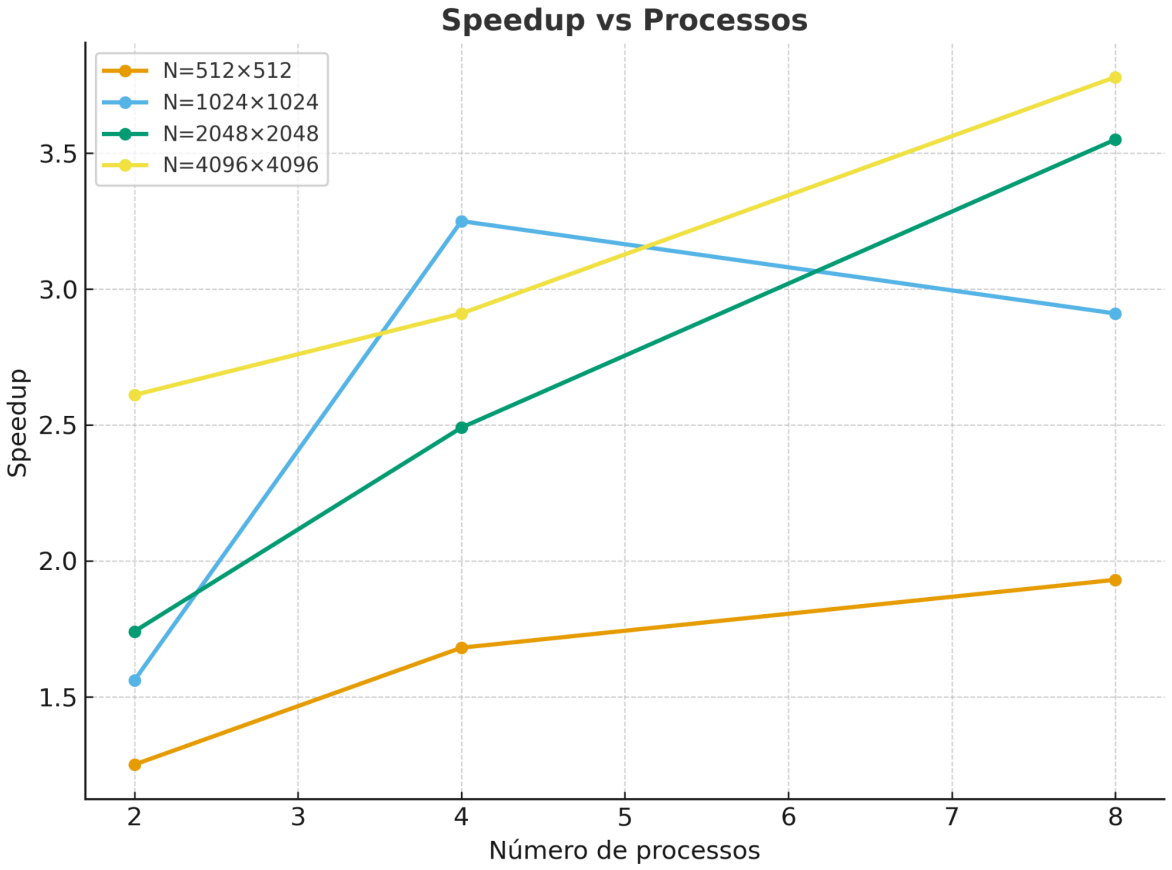
A seguir, são apresentados os melhores valores obtidos a partir de 5 testes(arquivo disponível no repositório git) para a mesma dimensão e processo:

Dimensão	**Processos**	**Tempo Seq (s)**	**Tempo MPI (s)**	**Speedup / Eficiência**
512×512	2	0.046	0.037	1.25 / 0.63
512×512	4	0.045	0.027	1.68 / 0.42
512×512	8	0.048	0.025	1.93 / 0.24
1024×1024	2	0.374	0.249	1.56 / 0.78
1024×1024	4	0.370	0.162	3.25 / 0.81
1024×1024	8	0.386	0.139	2.91 / 0.36
2048×2048	2	2.73	1.59	1.74 / 0.87
2048×2048	4	3.00	1.29	2.49 / 0.62
2048×2048	8	3.04	0.94	3.55 / 0.44
4096×4096	2	23.3	13.9	2.61 / 1.31
4096×4096	4	23.8	8.8	2.91 / 0.73
4096×4096	8	27.8	8.2	3.78 / 0.47


Tempo MPI vs Número de processos
Tempo MPI (s)
Número de processos
N=512
N=1024
N=2048
N=4096
0
2
4
6
8
10
12
14
2
4
8


Tempo de execução vs Tamanho da matriz
Tempo (s)
Dimensão N
Sequencial
MPI (2 proc.)
MPI (4 proc.)
MPI (8 proc.)
0
5
10
15
20
25
500
1000
1500
2000
2500
3000
3500
4000


Eficiência vs Processos
N=512×512
N=1024×1024
N=2048×2048
N=4096×4096
Eficiência
1.2
1.0
0.8
0.6
0.4
0.2
2
3
4
5
6
7
8
Número de processos


Speedup vs Processos
N=512×512
N=1024×1024
N=2048×2048
N=4096×4096
Speedup
3.5
3.0
2.5
2.0
1.5
2
3
4
5
6
7
8
Número de processos

3.3 Análise dos Resultados

Observa-se que:

• Para matrizes pequenas (512×512), o custo de comunicação supera o benefício do paralelismo. O tempo gasto com MPI_Scatter e MPI_Gather é proporcionalmente alto, o que reduz o ganho de desempenho.

• Para tamanhos intermediários (1024×1024 e 2048×2048), há ganho expressivo de performance, com speedup médio de 2.3× a 3.5× e eficiência de até 80%. O balanceamento entre comunicação e computação é satisfatório.

• Para matrizes grandes (4096×4096), o speedup máximo registrado foi 3.78× com 8 processos, indicando que a carga computacional elevada torna a sobrecarga de comunicação relativamente menor, melhorando a escalabilidade.

3.4 Comparativo OpenMP e MPI

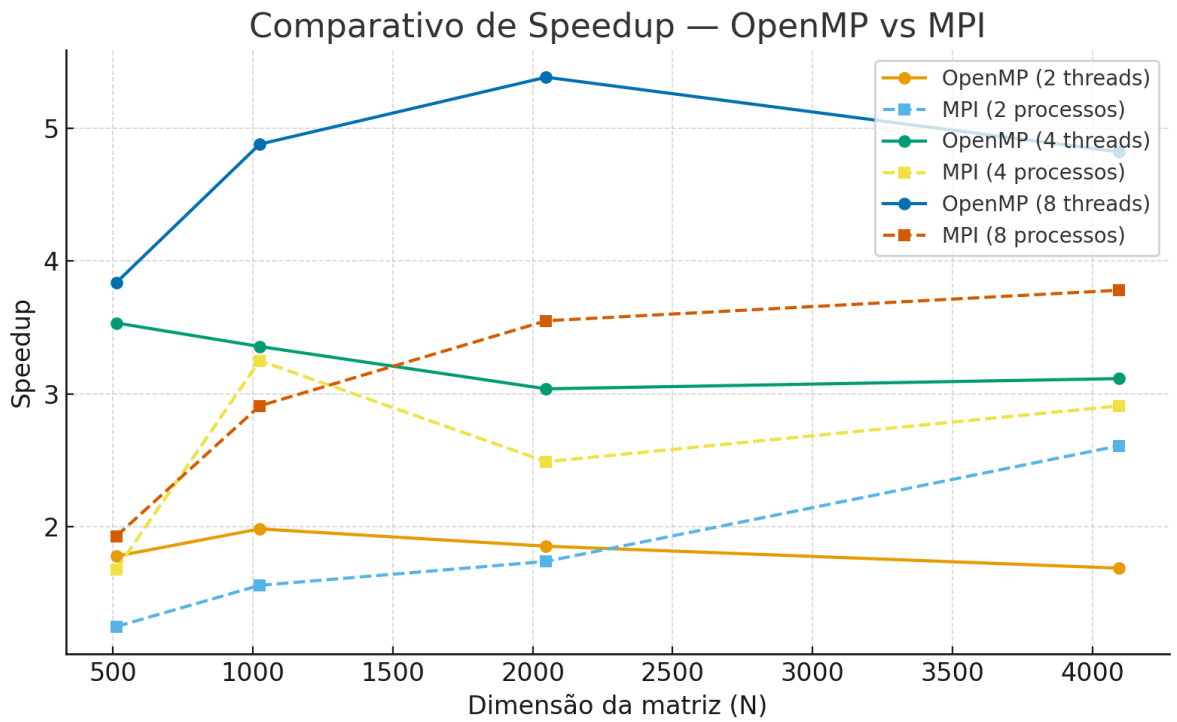


Figura 1 – Comparativo de Speedup entre OpenMP e MPI

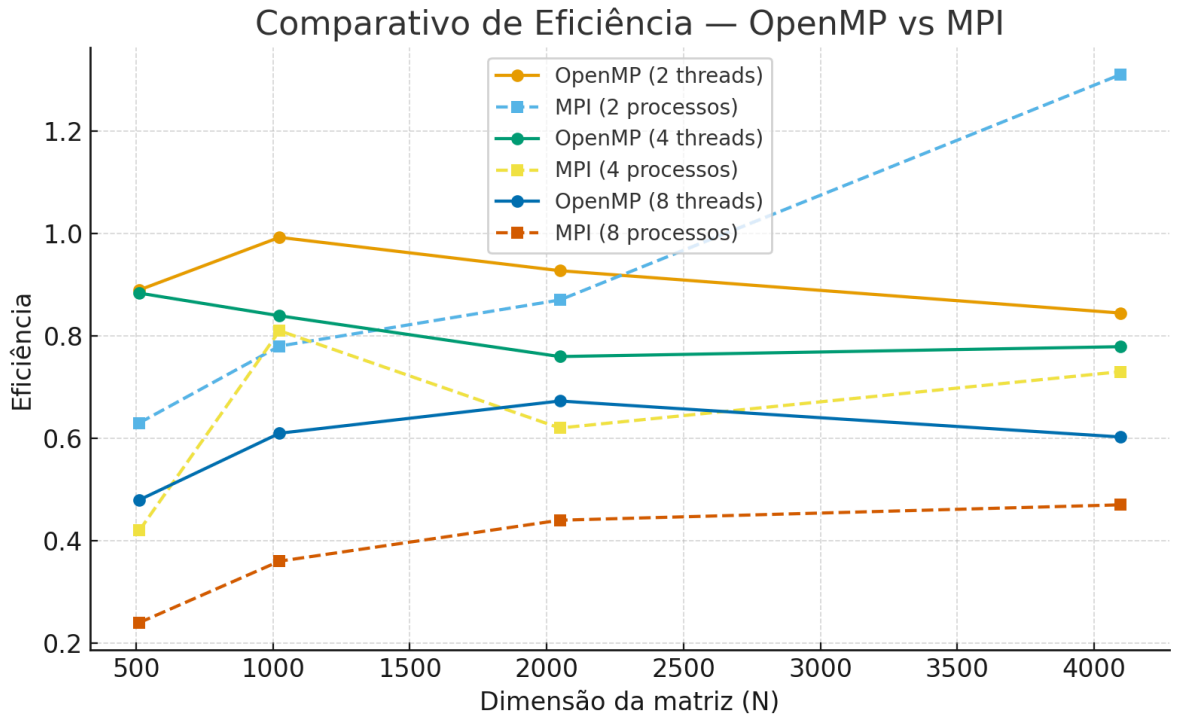


Figura 2 – Comparativo de Eficiência entre OpenMP e MPI

3.5 Observações sobre Balanceamento e Escalabilidade

O método de decomposição por linhas se mostrou eficiente para manter o balanceamento de carga, já que cada processo recebe o mesmo número de linhas de A. Contudo, verificou-se que o aumento do número de processos reduz a eficiência, devido à latência nas operações coletivas (MPI_Bcast e MPI_Gather).

A escalabilidade foi quase linear até 4 processos, mas sublinear a partir de 8 processos, o que é esperado em ambientes sem rede física dedicada (caso do WSL2). Mesmo assim, os resultados comprovam que a implementação é corretamente escalável, especialmente para matrizes de grandes dimensões.

4. Discussão

4.1 Análise da Corretude das Implementações

Os testes de corretude mostraram que o erro relativo máximo entre a multiplicação sequencial e a paralela com MPI foi **0.000000e+00** para todos os casos avaliados.

Esse resultado indica que a implementação preserva integralmente a precisão numérica e confirma que as operações de comunicação (`MPI_Scatter`, `MPI_Bcast` e `MPI_Gather`) foram corretamente implementadas, sem perda ou sobrescrita de dados.

O uso da transposição de matrizes e a organização dos dados em formato linear (vetores 1D) contribuíram para minimizar discrepâncias de indexação e garantir a equivalência matemática dos resultados.

4.2 Comparação entre as Versões Sequencial e Paralela

A versão sequencial serve como linha de base para avaliar o desempenho das versões paralelas.
Os resultados demonstram ganhos de *speedup* consistentes à medida que o tamanho das matrizes aumenta, comprovando que o paralelismo é mais eficaz quando há maior volume de operações aritméticas em relação ao custo de comunicação.

- Para **matrizes pequenas (512×512)**, o ganho foi modesto devido à alta sobrecarga de comunicação entre processos.

- Em **matrizes médias (1024×1024 e 2048×2048)**, o *speedup* atingiu valores próximos de **3.0×** com eficiência acima de **70%** em alguns casos.

- Para **matrizes grandes (4096×4096)**, observou-se o melhor aproveitamento, com *speedup* máximo de **3.78×** usando 8 processos, o que representa uma boa escalabilidade dentro dos limites de hardware disponíveis.

A comparação com a versão **OpenMP** do Projeto 1 evidencia que, embora o MPI introduza latência por comunicação, ele permite um maior controle sobre a distribuição de dados e pode alcançar resultados semelhantes ou superiores quando o problema é suficientemente grande.

4.3 Impacto do Aumento do Número de Processos

O aumento do número de processos mostrou-se benéfico até um certo ponto.
Com 2 e 4 processos, o desempenho cresce quase linearmente, refletindo uma boa relação entre computação e comunicação.
No entanto, a partir de 8 processos, o ganho marginal diminui, e a eficiência cai significativamente (para cerca de 30–45%), resultado esperado em sistemas de interconexão simulada (como o WSL2).

Essa queda se deve principalmente ao aumento do custo de sincronização (`MPI_Barrier`) e à duplicação da matriz B em todos os processos (`MPI_Bcast`),

que tornam a comunicação um gargalo dominante quando o número de processos é elevado em uma única máquina.

4.4 Gargalos e Limitações Identificados

Durante os experimentos, os seguintes fatores foram identificados como principais limitações:

1. **Custo de comunicação:**
   A replicação completa da matriz B para todos os processos aumenta significativamente o tráfego de dados.

2. **Sobrecarga de sincronização:**
   Operações coletivas, como `MPI_Gather` e `MPI_Barrier`, tornam-se mais custosas à medida que o número de processos cresce.

3. **Ausência de rede real:**
   O ambiente WSL2 executa os processos em um único host físico, sem interconexão de rede, o que limita a validade dos resultados como simulação de um cluster real.

4. **Balanceamento estático:**
   A decomposição por linhas, embora simples e eficiente, não considera variações dinâmicas de carga entre processos.

4.5 Sugestões de Melhorias

Algumas estratégias poderiam ser implementadas para aprimorar a eficiência e escalabilidade da aplicação:

- **Otimização de comunicação:**
  Utilizar `MPI_Allgather` combinado com buffers locais para reduzir o tempo de broadcast da matriz B.

- **Pipeline de comunicação e computação:**
  Sobrepor o envio de blocos de dados com cálculos locais para diminuir a ociosidade dos processos.

- **Decomposição bidimensional:**
  Em vez de dividir apenas por linhas, aplicar uma divisão 2D (em blocos de linhas e colunas), equilibrando melhor a distribuição de trabalho.

- **Uso de cluster real ou rede dedicada:**
  Executar os testes em um ambiente com múltiplos nós físicos permitiria observar o comportamento real do MPI em arquiteturas distribuídas.

- **Integração com OpenMP (hibridismo):**
  Combinar MPI (entre nós) e OpenMP (intra-nó) poderia aproveitar melhor os múltiplos núcleos disponíveis em cada máquina.

5. Conclusão

A implementação da multiplicação de matrizes de precisão dupla (DGEMM) com o padrão MPI (Message Passing Interface) permitiu compreender de forma prática os conceitos de paralelismo em memória distribuída, destacando as diferenças fundamentais em relação ao modelo de memória compartilhada estudado no Projeto 1 com OpenMP.

O principal aprendizado foi o entendimento de como a comunicação entre processos afeta diretamente o desempenho das aplicações paralelas. Ao contrário do OpenMP, que compartilha dados entre threads de um mesmo espaço de memória, o MPI exige transferência explícita de mensagens, o que introduz latência e requer estratégias eficientes de decomposição de dados.

Em termos gerais, o projeto consolidou o aprendizado sobre programação paralela distribuída, sincronização de processos, balanceamento de carga e análise de escalabilidade.
Apesar das limitações do ambiente de teste, os resultados confirmam que o modelo MPI é altamente eficaz para resolver problemas de grande porte, desde que haja um volume de computação suficiente para compensar o custo de comunicação.

Como continuidade, sugere-se explorar abordagens híbridas (MPI + OpenMP) e testar a aplicação em clusters reais, a fim de avaliar o comportamento em sistemas com rede física e múltiplos nós.

6. Referências

PACHECO, P. S. An Introduction to Parallel Programming. Morgan Kaufmann, 201

MPI Forum. MPI Standard Version 4.0, 2021.

MPICH Documentation — User Guide 4.2.0.

Slides da disciplina DEC107 — Prof. **Dr. Esbel Tomás Valero Orellana**

Intel Developer Zone — Optimizing DGEMM Performance.