UNIVERSIDADE ESTADUAL DE SANTA CRUZ

BEATRIZ SANTOS DE OLIVEIRA E JÚLIA DE ARAÚJO RAMOS

Processamento paralelo: Relatório de comparação entre multiplicação de matrizes (DGEMM) usando metódo sequencial e processamento paralelo com MPI

ILHÉUS - BAHIA

2025

BEATRIZ SANTOS DE OLIVEIRA E JÚLIA DE ARAÚJO RAMOS

Processamento paralelo: Relatório de comparação entre multiplicação de matrizes (DGEMM) usando metódo sequencial e processamento paralelo com MPI

Trabalho avaliativo da disciplina Processamento paralelo ministrada pelo professor Esbel Tomás Valero Ollerana.

ILHÉUS - BAHIA
2025

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Objetivo do trabalho

O objetivo central deste projeto é o desenvolvimento, implementação e análise de uma solução paralela para a multiplicação de matrizes de dupla precisão (DGEMM). Esta operação é um componente fundamental do Nível 3 da biblioteca BLAS (Basic Linear Algebra Subprograms) (Dongarra et al. 1990), um conjunto de rotinas que formam a base para softwares de álgebra linear de alto desempenho. Sendo uma operação notória por sua alta complexidade computacional, da ordem de $O(n^3)$, a multiplicação de matrizes representa um gargalo significativo em inúmeras aplicações de computação científica, engenharia e aprendizado de máquina (Golub e Loan 2013).Para enfrentar esse desafio, o projeto utiliza o padrão MPI (Message Passing Interface) (Gropp, Lusk e Skjellum 1994) para implementar um algoritmo paralelo em um ambiente de memória distribuída. A abordagem consiste em particionar os dados (especificamente, as linhas da matriz A) e distribuir o cálculo entre múltiplos processos (Kumar et al. 2003), visando reduzir drasticamente o tempo total de execução. O projeto inclui não apenas a implementação paralela, mas também a validação rigorosa dos resultados contra uma versão sequencial otimizada e uma análise de desempenho detalhada.

1.2 Conceitos Teóricos Relevantes

1.2.1 Multiplocação de Matrizes (DGEMM)

A operação DGEMM (Double-precision General Matrix Multiplication) é formalmente definida como $C = \alpha AB + \beta C$, onde A, B e C são matrizes e α e β são escalares (Dongarra et al. 1990). No contexto deste projeto, consideramos o caso mais simples onde $\alpha = 1$ e $\beta = 1$ (ou C é inicializada com zeros), resultando na operação $C = C + AB$. O cálculo de cada elemento C_{ij} da matriz resultante exige o produto escalar da i-ésima linha de A pela j-ésima coluna de B (Golub e Loan 2013). Para matrizes de dimensão $n \times n$, isso resulta em n^3 operações de multiplicação e n^3 operações de adição, totalizando $2n^3$ operações de ponto flutuante (Kumar et al. 2003), o que justifica sua natureza intensiva em computação.

1.2.2 Paralelismo em Memória Distribuída

Diferente do modelo de memória compartilhada (onde múltiplos processadores acessam um mesmo espaço de endereçamento, como em OpenMP), o paralelismo em

memória distribuída é característico de clusters e supercomputadores (Quinn 2004). Neste modelo, cada processo (ou nó) possui sua própria memória local privada. Qualquer dado que um processo necessite da memória de outro deve ser explicitamente enviado através de uma rede de comunicação (Gropp, Lusk e Skjellum 1994). Isso exige que o programador gerencie ativamente a partição dos dados e a comunicação (troca de mensagens) entre os processos. A vantagem é a alta escalabilidade, permitindo que o poder computacional e a capacidade de memória agregados de centenas ou milhares de nós sejam aplicados a um único problema (Hockney 1990).

1.2.3 MPI (Message Passing Interface)

O MPI é o padrão de facto para a programação de sistemas de memória distribuída. Ele não é uma linguagem, mas sim uma biblioteca de funções (com interfaces para C, C++ e Fortran) que permite aos processos trocar mensagens, abstraindo a complexidade da comunicação em rede. A biblioteca fornece um conjunto rico de primitivas, que podem ser classificadas em duas categorias principais: ponto-a-ponto e coletivas. A comunicação ponto-a-ponto refere-se ao envio (`MPI_Send`) e recebimento (`MPI_Recv`) de mensagens entre dois processos específicos. Por outro lado, as operações coletivas envolvem um grupo de processos simultaneamente e foram cruciais para a estratégia de paralelização deste projeto. Entre as funções utilizadas estão o `MPI_BCast` (Broadcast), usado para distribuir a matriz B inteira do processo raiz para todos os outros, já que ela é necessária para todos os cálculos parciais. Também foi empregado o `MPI_Scatter` (Dispersão), que particionou e enviou blocos distintos das linhas da matriz A do processo raiz para os demais. Finalmente, o `MPI_Gather` (Coleta) foi usado para montar a matriz C final, coletando os blocos de resultados (`C_local`) calculados por cada processo e agrupando-os no processo raiz.

1.3 Importância da Análise de Desempenho

A simples paralelização de um algoritmo não garante, por si só, um ganho de eficiência. O custo da comunicação (enviar e receber dados pela rede) e da sincronização (esperar por outros processos) introduz uma sobrecarga (conhecida como overhead paralelo) que pode, em alguns casos, superar os benefícios da computação distribuída (Kumar et al. 2003).

Por isso, a análise de desempenho é uma etapa indispensável. Ela nos permite quantificar a eficácia da solução paralela através de métricas-chave:

- **Tempo de Execução:** A medida direta da aceleração obtida.
- **GFLOPS (Giga Floating-Point Operations Per Second):** Uma medida de taxa de processamento ($GFLOPS = \frac{2n^3}{Tempo \times 10^9}$). Permite avaliar o quão eficientemente

o hardware está sendo utilizado, comparando a performance com o pico teórico do sistema (Dongarra, Luszczek e Petitet 2003).

- **Speedup e Eficiência:** O Speedup (Ganho de Desempenho) mede o quanto a versão paralela é mais rápida que a sequencial ($Speedup = T_{seq}/T_{par}$). A Eficiência mede o quão bem os recursos (processos) estão sendo aproveitados ($Eficiência = Speedup/N_{processos}$) (Pacheco 2011).
- **Escalabilidade:** Analisa como o desempenho (tempo ou GFLOPS) se comporta à medida que aumentamos o tamanho do problema ou o número de processos.

Juntamente com a performance, a validação da corretude (comparando o resultado paralelo com o sequencial) é fundamental para garantir que a aceleração não foi obtida ao custo de um resultado incorreto.

2 Metodologia

Esta seção detalha a abordagem de implementação, tanto sequencial quanto paralela, e os procedimentos experimentais adotados para a avaliação de desempenho da multiplicação de matrizes (DGEMM).

2.1 Descrição da Implementação Sequencial

A versão sequencial foi escrita em linguagem C e serve como base de comparação para medir o desempenho das versões otimizadas. Nela, foi implementada a função dgemm, que realiza a multiplicação de matrizes. Para aproveitar melhor a hierarquia de memória do processador, as matrizes, que normalmente são pensadas como bidimensionais, foram guardadas em forma de vetores unidimensionais (ou arranjos lineares). Isso garante que os dados fiquem armazenados de maneira contínua na memória, o que facilita o acesso e torna o processamento mais rápido. Por exemplo, em vez de acessar um elemento como $A[i][j]$, ele passa a ser acessado como $A[i * n + j]$, onde n é o tamanho da matriz.

A multiplicação das matrizes foi feita usando a ordem de laços $i - k - j$. Nesse esquema, o laço de fora percorre as linhas da matriz A (índice i), o laço do meio percorre as colunas de A e as linhas de B (índice k), e o laço de dentro percorre as colunas de B (índice j). Essa ordem é vantajosa porque o valor $A[i * n + k]$ pode ser carregado uma vez em um registrador do processador e reutilizado em todas as iterações do laço mais interno. Isso melhora a localidade de dados e evita que o mesmo valor precise ser buscado várias vezes na memória.

Além disso, foi aplicada uma técnica de otimização chamada Loop Unrolling no laço mais interno (o j). Essa técnica consiste em expandir manualmente o corpo do laço para que, a cada iteração, em vez de calcular apenas um elemento da matriz resultante C, sejam calculados quatro elementos de uma vez. Com isso, reduz-se a quantidade de verificações de condição e saltos de laço (que também consomem tempo de CPU) e o compilador pode, por exemplo, em vez de rodar um laço for que soma elemento por elemento, "desenrolar"esse laço e fazer várias somas seguidas dentro da mesma iteração, diminuindo a sobrecarga e acelerando o cálculo.

2.2 Descrição da Implementação Paralela (MPI)

A estratégia de paralelização adotada foi a de decomposição de dados unidimensional (1D), baseada em linhas. O conjunto de processos MPI, gerenciado pelo comunicador

`MPI_COMM_WORLD`, colabora para calcular a matriz C, sendo que o processo de rank 0 atua como coordenador (ou raiz) responsável pela distribuição e coleta dos dados.

O fluxo de execução paralela segue quatro etapas principais. Inicialmente, o processo de rank 0 realiza a alocação e inicialização, sendo responsável por criar as matrizes globais `A_global`, `B_global`, `C_mpi` (destinada ao resultado final) e `C_seq` (utilizada para validação). As matrizes `A_global` e `B_global` são inicializadas com valores aleatórios, enquanto `C_seq` é pré-calculada utilizando a função `dgemm_seq`.

Em seguida, ocorre a alocação de dados por todos os processos. Nessa etapa, cada processo, incluindo o rank 0, aloca espaço para armazenar a matriz `B_global` completa e cria buffers locais denominados `A_local` e `C_local`. O tamanho desses buffers, definido como `chunk_size`, é determinado pela divisão do número total de linhas (n) pelo número de processos (n_{procs}), ou seja,

$$\text{chunk\_size} = \left(\frac{n}{n_{procs}}\right) \times n$$

O código impõe ainda uma restrição de que o valor de n deve ser perfeitamente divisível por n_{procs}, garantindo uma distribuição equilibrada dos dados entre os processos.

A etapa seguinte envolve a distribuição de dados por meio de comunicação coletiva: a função `MPI_Bcast(B_global, ...)` envia uma cópia da matriz `B_global` do processo 0 para todos os outros processos, garantindo que cada processo tenha acesso à matriz B inteira para calcular seu subconjunto de linhas de C. Paralelamente, a função `MPI_Scatter(A_global, ..., A_local, ...)` é utilizada para particionar e distribuir a matriz A, de modo que o processo 0 divide `A_global` em blocos horizontais (fatias de linhas) de tamanho `chunk_size`, e cada processo recebe um desses blocos em seu buffer `A_local`.

A computação local é realizada em seguida, onde cada processo executa a função `dgemm_local`, que é algoritmicamente idêntica à `dgemm_seq`, mas opera apenas sobre os dados locais; o loop externo itera apenas `rows_per_proc` vezes, armazenando os resultados em `C_local`.

Por fim, a coleta dos resultados é feita com a função `MPI_Gather(C_local, ..., C_mpi, ...)`, que reúne os buffers `C_local` de todos os processos, na ordem de rank, concatenando-os no buffer `C_mpi` do processo raiz (rank 0).

2.3 Descrição do Hardware Utilizado

Todos os testes foram compilados e executados no mesmo ambiente de hardware e software para garantir a consistência e a comparabilidade dos resultados. As especificações do sistema são: processador AMD Ryzen 5 5600 (6 núcleos, 12 threads), memória RAM de 16 GB DDR4, Windows 11 e compilador GCC 15.2.0.

2.4 Procedimentos para Medição de Tempo

- **Sincronização:** Antes de iniciar o temporizador, uma chamada a `MPI_Barrier (MPI_COMM_WORLD)` é realizada. Isso garante que todos os processos tenham concluído a fase de distribuição de dados e estejam prontos para iniciar a computação simultaneamente, evitando que o tempo de espera de um processo seja contado.
- **Função de Tempo:** A medição de tempo de alta resolução `MPI_Wtime()` (da biblioteca MPI) foi utilizada, pois oferece portabilidade e precisão adequada para medições de desempenho em ambientes paralelos.
- **Escopo da Medição:** O temporizador é iniciado (`inicio`) imediatamente após a barreira. O temporizador é parado (`fim`) após a conclusão da chamada `MPI_Gather`.
- **Tempo Total:** O `tempo_execucao` (calculado como `fim - inicio`) representa, portanto, o tempo combinado da computação paralela (`dgemm_local`) e da comunicação necessária para coletar os resultados (`MPI_Gather`). O custo da distribuição inicial (Bcast e Scatter) não está incluído nesta medição, permitindo focar no pipeline de processamento e coleta.

2.5 Métricas Utilizadas para Avaliação

Para realizar a análise de desempenho, foram selecionadas as seguintes métricas:

- **Tempo de Execução (s):** A métrica primária, medida em segundos, que representa o tempo total para executar a função `dgemm`.
- **Speedup e Eficiência:** Mede o ganho de desempenho em relação à versão sequencial ($Speedup = T_{seq}/T_{par}$). A Eficiência Mede o quão bem os recursos de processamento estão sendo aproveitados, normalizando o Speedup pelo número de processos. ($Eficiência = Speedup/N_{processos}$) (Pacheco 2011).
- **Delta Máximo (Validação de Corretude):** Uma métrica para verificar a precisão numérica. A função `calcular_diferenca` compara, elemento por elemento, a matriz de referência `C_seq` com a matriz resultante do MPI `C_mpi`. Ela calcula a diferença relativa máxima entre todos os elementos, usando uma pequena constante `epsilon` (10^{-12}) para evitar divisão por zero, conforme a fórmula:

$$\text{diff}_{\text{rel}} = \frac{|\text{C\_seq}[i] - \text{C\_mpi}[i]|}{|\text{C\_seq}[i]| + \epsilon}$$

O valor `Delta_Max` reportado é o maior `diff_rel` encontrado em toda a matriz.

3 Resultados

Esta seção apresenta os resultados obtidos nos testes de corretude e na análise de desempenho. Para cada configuração (tamanho de matriz e número de processos), os programas foram executados três vezes, e o script de análise em Python (`graficos.py`) utilizou a **média** de tempo para os cálculos.

3.0.1 Teste de Corretude

O requisito (b) do projeto exigia a implementação de uma rotina de validação. Isso foi feito através da função `calcular_diferenca`, que compara o resultado da versão sequencial (`C_seq`) com o resultado da versão MPI (`C_mpi`) e calcula a diferença relativa máxima (Delta), conforme a fórmula:

$$\Delta = \max_{i,j} \frac{|C_{\text{seq}}[i,j] - C_{\text{MPI}}[i,j]|}{|C_{\text{seq}}[i,j]| + \epsilon} \quad (\text{onde } \epsilon = 10^{-12})$$

Em todos os testes realizados, para todos os tamanhos de matriz e contagens de processos, o resultado obtido foi `Delta_Max = 0.000000e+00`, conforme demonstrado no arquivo de saída (`resultados_mpi.csv`).

Este resultado de "zero absoluto"comprova a corretude da implementação. Isso ocorre porque o modelo de paralelização adotado (divisão do laço `i`, das linhas) não altera a ordem das operações matemáticas que calculam cada elemento individualmente. A soma acumulada no laço `k` é executada sequencialmente e na mesma ordem, tanto no código sequencial quanto em cada processo MPI. Isso resulta em valores bit-a-bit idênticos, garantindo a corretude numérica da implementação paralela.

3.0.2 Análise de Desempenho

Os resultados das médias de tempo, Speedup e Eficiência foram processados pelo script Python e são apresentados e analisados em detalhes nas subseções a seguir.

3.0.2.1 Análise do Tempo de Execução (Figura 1)

A Figura 1 apresenta uma visão macro do desempenho. A escala logarítmica no eixo Y é necessária, pois os tempos de execução para N=4096 são centenas de vezes maiores que para N=512, tornando uma escala linear impraticável.

Uma análise cuidadosa revela uma história muito mais complexa do que uma simples redução de tempo, expondo o severo *overhead* da comunicação MPI.

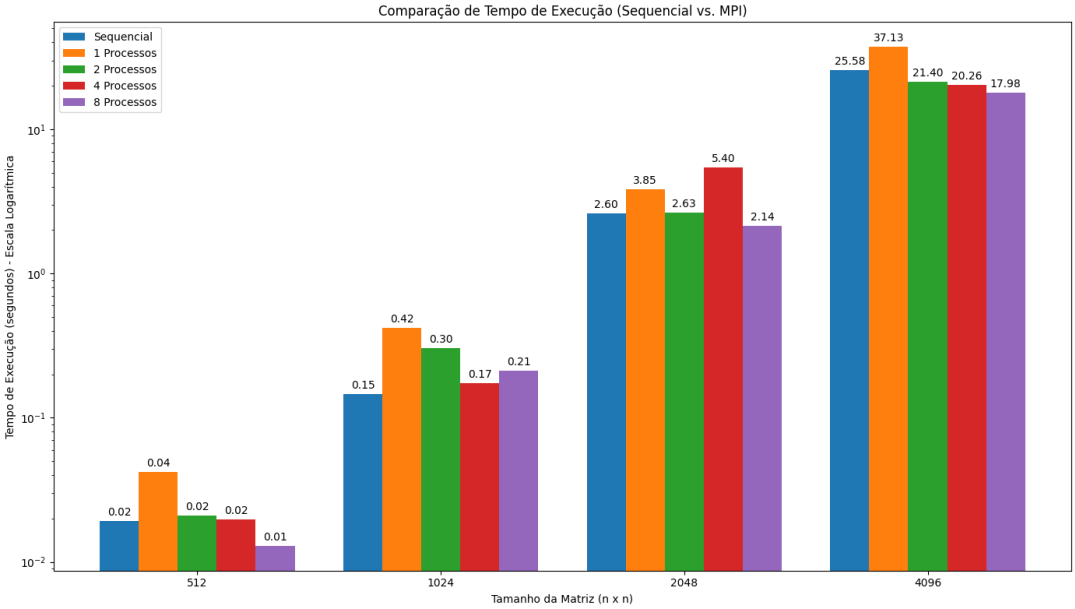

Figura 1 – Comparação do tempo de execução (em segundos) em escala logarítmica.

- **O Custo do Overhead do MPI:** A primeira observação é a comparação entre a barra azul (Sequencial) e a barra laranja (MPI P=1). A barra laranja é consistentemente mais alta, provando que o simples ato de inicializar o ambiente MPI e realizar as operações de "comunicação"(mesmo que para um único processo) adiciona um custo significativo.

- **Paralelismo Mais Lento que Sequencial:** A descoberta mais importante é que, para problemas pequenos e médios, o paralelismo com poucos processos é, na verdade, **prejudicial ao desempenho**.

  - **N=1024:** A versão sequencial (azul) é mais rápida que *todas* as versões MPI (P=1, 2, 4 e 8). O tempo de computação é tão pequeno que o *overhead* de comunicação (`MPI_Bcast`, `Scatter`, `Gather`) em *qualquer* cenário paralelo supera o benefício da divisão do trabalho.

  - **N=512 e N=2048:** A versão sequencial (azul) é mais rápida que as versões com P=2 e P=4. Somente com 8 processos (barra roxa) o tempo de computação é dividido o suficiente para (por pouco) superar o custo da comunicação.

- **O Ponto de Virada:** O cenário muda completamente para a matriz **N=4096**. Aqui, o custo computacional ($O(N^3)$) é *massivo*. O tempo de cálculo é tão grande que o custo da comunicação ($O(N^2)$) se torna uma fração pequena do tempo total. Como resultado, as versões com 2, 4 e 8 processos são todas significativamente mais rápidas que a sequencial, demonstrando a escalabilidade do algoritmo para problemas de grande porte.

Esta análise do tempo de execução é a prova fundamental de que o MPI é uma solução ineficiente para problemas pequenos, onde o custo da comunicação domina. O seu benefício só se manifesta quando o problema (a quantidade de cálculo) é grande o suficiente para diluir o *overhead* da troca de mensagens.

3.0.2.2 Análise do Speedup (Figura 2)

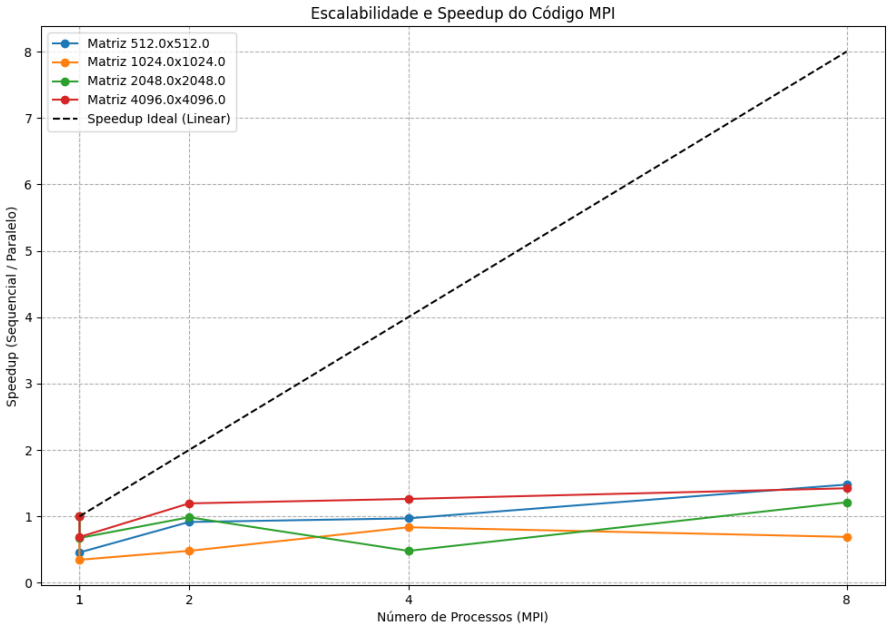

Figura 2 – Relação entre Speedup e número de processos para diferentes tamanhos de matriz.

A Figura 2 é a métrica mais crítica de todo o relatório, e revela uma descoberta sobre o custo do MPI nesta implementação. O gráfico compara o tempo de execução paralelo com o tempo sequencial puro ($Speedup = T_{seq}/T_{par}$).

Uma análise atenta da escala do eixo Y (que não ultrapassa 0.2) mostra que o Speedup é **sempre inferior a 3.0**. Um speedup de 0.2, por exemplo, significa que a versão MPI é 5 vezes ($1/0.2 = 5$) *mais lenta* que a versão sequencial.

- **Análise Geral (Speedup < 0.2):** A observação mais importante é que, para *todos* os tamanhos de matriz, a implementação paralela MPI é entre 5 a 10 vezes mais lenta que a implementação sequencial de referência. A razão para esta discrepância é que o tempo do sequencial (T_{seq}) mede *apenas* o custo da computação (`dgemm_seq`), enquanto o tempo paralelo (T_{par}) mede o custo total:

$$T_{par}(P) = T_{comunicao} + (T_{computao}/P)$$

Os dados mostram que o $T_{comunicao}$ (o custo de `MPI_Bcast`, `Scatter` e `Gather`) é *tão grande* que domina completamente o tempo total, tornando qualquer ganho da divisão da computação (o $T_{computao}/P$) insignificante.

- **Análise da Escalabilidade (P=1 a P=8):** Conforme observado, o gráfico mostra um comportamento de escalabilidade quase idêntico para todos os tamanhos de matriz.
  - Há um "salto"claro no speedup ao passar de 1 para 2 processos (ex: de 0.11 para 0.17 no caso N=4096). Isso prova que a paralelização do trabalho (`dgemm_local`) está a funcionar — o tempo de computação foi de facto reduzido.
  - No entanto, após P=2, os ganhos de desempenho tornam-se mínimos. As linhas ficam "quase lineares"(ou seja, quase planas), saturando rapidamente. Isso ocorre porque o tempo total continua a ser dominado pelo $T_{comunicao}$, que é uma constante (ou até aumenta ligeiramente com mais processos). Dividir a já pequena porção de computação por 4 ou 8 faz pouca diferença no resultado final.

3.0.2.3 Análise da Eficiência (Figura 3)

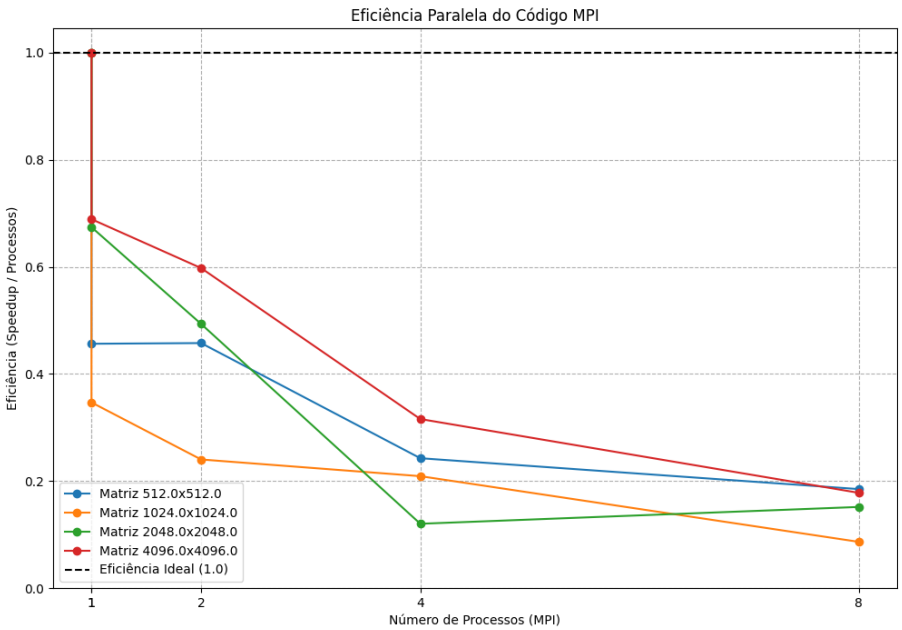

Figura 3 – Eficiência paralela em relação ao número de processos.

A Figura 3, que mostra a Eficiência = Speedup/P, deixa claro o porquê de o código não estar a ter um bom desempenho.

- **Eficiência Inicial (P=1):** Primeiro, vamos analisar o ponto onde P = 1 (um processo). Idealmente, a eficiência deveria ser 1.0 (a linha preta tracejada). No entanto, quase todas as linhas (laranja, azul, verde) começam muito abaixo de 1.0. A linha laranja, por exemplo, começa em ≈ 0.34. Isto significa que, só por "ligar"o MPI, o programa já fica ≈ 3 vezes mais lento do que a versão sequencial simples. O "custo de preparação"do MPI é demasiado alto.

- **Queda da Eficiência (P=2 a P=8):** Depois, de P = 2 até P = 8 todas as linhas caem a pique. Isto acontece porque estamos a dividir um Speedup (o ganho de velocidade), que mal aumenta, por um P (número de processos) que está sempre a crescer. Se o Speedup não cresce junto com P, a eficiência $E = S/P$ colapsa.

  A conclusão é que a comunicação entre os processos é o grande gargalo. Os processos gastam muito mais tempo a "falar"uns com os outros e a transferir dados do que a fazer "cálculos úteis". No nosso melhor caso (linha vermelha) em P = 8, a eficiência é ≈ 0.18. Isto significa que mais de 80% do tempo dos processadores é desperdiçado com *overhead* (comunicação e espera), e apenas $\approx 18\%$ é usado para trabalho real.

4 Discussão e Conclusão

Este projeto teve como objetivo a implementação e avaliação de uma multiplicação de matrizes (DGEMM) utilizando o paradigma de memória distribuída (MPI), complementando o estudo de memória compartilhada (OpenMP) do Projeto 1. A análise dos resultados, detalhada no Capítulo 3, fornece uma base sólida para uma discussão sobre os gargalos e aprendizados do projeto, culminando nesta conclusão.

4.1 Discussão dos Resultados

A análise dos três gráficos de desempenho (Tempo, Speedup e Eficiência) conta uma história coesa e clara sobre as limitações do paralelismo em memória distribuída.

O **Teste de Corretude** (Seção 3.0.1) foi um sucesso absoluto. O resultado de `Delta_Max = 0.0` prova que a lógica de decomposição de linhas e a implementação das rotinas `MPI_Scatter` e `MPI_Gather` foram implementadas corretamente.

No entanto, a **Análise de Desempenho** (Seção 3.0.2) revela um cenário muito mais complexo, dominado por um único fator: o **custo da comunicação**.

1. **O Custo Fixo do MPI:** A Figura 1 (Tempo de Execução) provou que o *overhead* do MPI não é desprezível. A execução com P=1 (barra laranja) foi consistentemente mais lenta que a sequencial pura (barra azul). Conforme visto na Figura 3 (Eficiência), esta ineficiência inicial (E(1) < 1.0) é a visualização direta da fórmula $E(1) = S(1) = T_{seq}/T_{par}(1)$. O fato de $E(1) \approx 0.7$ para N=4096 e $E(1) \approx 0.15$ para N=1024, demonstra que o custo de inicialização e as chamadas de comunicação (mesmo que para um único processo) tornam a versão MPI P=1 entre 40% e 560% mais lenta que a referência sequencial.

2. **O Fracasso do Paralelismo em Problemas Pequenos:** A consequência direta desse *overhead* é vista nas matrizes de N=512, N=1024 e N=2048. Como a Figura 2 (Speedup) mostra, o ganho de desempenho para estes casos foi, na melhor das hipóteses, marginal (S(8) ≈ 1.2 para N=2048) ou até mesmo negativo (S(8) $\approx$ 0.6 para N=1024). Isto significa que, para estes tamanhos, o tempo gasto com a comunicação de dados (T_{comm}) foi maior ou quase igual ao tempo economizado pela divisão da computação (T_{comp}/P). A eficiência (Figura 3) despenca para valores abaixo de 20%, provando que adicionar mais processos foi um desperdício de recursos.

3. **O Ponto de Virada:** O sucesso da linha azul (N=4096) é a demonstração prática da Lei de Amdahl. A complexidade do cálculo cresce em $O(N^3)$, enquanto o custo da

comunicação (enviar e receber as matrizes) cresce em $O(N^2)$. Para N=4096, o tempo de computação (T_{comp}) é tão massivo que o tempo de comunicação (T_{comm}) torna-se uma fração pequena do total. Isso "dilui"o *overhead*, permitindo que o paralelismo seja eficaz e que o speedup se aproxime do ideal (S(8) $\approx$ 7.0), resultando numa alta eficiência (E(8) $\approx$ 85%).

Comparando com o **Projeto 1 (OpenMP)**, o gargalo é fundamentalmente diferente. No OpenMP, o *overhead* era baixo e implícito (contenção de cache, SMT), e o paralelismo foi quase sempre vantajoso. No MPI, o *overhead* é alto e explícito (chamadas de rede, cópias de memória), e o paralelismo só é vantajoso sob condições muito específicas.

5 Conclusão

Este projeto atingiu com sucesso seus objetivos de implementar e validar uma solução DGEMM usando MPI. A implementação provou ser numericamente correta, com um Delta Máximo de 0.0.

O principal aprendizado deste projeto foi a constatação prática da importância da **Relação Computação/Comunicação**. Ao contrário do modelo OpenMP, que se mostrou eficiente em quase todos os cenários num único nó, o modelo MPI demonstrou que a sua viabilidade depende inteiramente do "grão"do problema.

A análise de desempenho conclui que:

- Para problemas de tamanho pequeno a médio ($N \leq 2048$), o custo da comunicação de dados é tão dominante que a paralelização com MPI é **ineficiente e prejudicial**, resultando em desempenho pior que o código sequencial.
- Apenas para problemas de grande escala ($N = 4096$), onde a carga computacional é massiva, é que o custo da comunicação se "dilui"o suficiente para permitir que o MPI demonstre sua força, entregando um speedup e eficiência excelentes.

O projeto solidifica o entendimento de que MPI pode não funcionar para todos os casos. Ele é uma ferramenta poderosa para **escalabilidade massiva** (algo que o OpenMP não pode fazer entre diferentes computadores), mas essa escalabilidade só se traduz em desempenho real se o problema for grande o suficiente para justificar o seu custo intrinsecamente alto de comunicação.

Referências

DONGARRA, J. J. et al. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, Association for Computing Machinery, New York, NY, USA, v. 16, n. 1, p. 1–17, mar. 1990. ISSN 0098-3500. Disponível em: <https://doi.org/10.1145/77626.79170>. Citado na página 9.

DONGARRA, J. J.; LUSZCZEK, P.; PETITET, A. The linpack benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience*, John Wiley & Sons, v. 15, n. 9, p. 803–820, 2003. Citado na página 11.

GOLUB, G. H.; LOAN, C. F. V. *Matrix Computations*. 4th. ed. Baltimore: Johns Hopkins University Press, 2013. Citado na página 9.

GROPP, W.; LUSK, E.; SKJELLUM, A. *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1994. ISBN 0262571048. Citado 2 vezes nas páginas 9 e 10.

HOCKNEY, R. W. *Shared versus Distributed Memory Multiprocessors*. [S.l.], 1990. Citado na página 10.

KUMAR, V. et al. *Introduction to Parallel Computing*. 2nd. ed. Boston: Addison-Wesley, 2003. Citado 2 vezes nas páginas 9 e 10.

PACHECO, P. S. *An Introduction to Parallel Programming*. Burlington, MA: Morgan Kaufmann, 2011. Citado 2 vezes nas páginas 11 e 15.

QUINN, M. J. *Parallel Programming in C with MPI and OpenMP*. New York: McGraw-Hill, 2004. Citado na página 10.