APRIMORANDO A MULTIPLICAÇÃO DE MATRIZES: UMA ANÁLISE COMPARATIVA ENTRE IMPLEMENTAÇÕES SEQUENCIAIS E PARALELAS UTILIZANDO MPI E OPENMP

Felipe Grosze Nipper de Oliveira <felipe.oliveira01@edu.pucrs.br>
Miguel Torres de Castro <miguel.c@edu.pucrs.br>
Roland Teodorowitsch2 <roland.teodorowitsch@pucrs.br> – Orientador

Pontifícia Universidade Católica do Rio Grande do Sul – Faculdade de Informática – Curso de Ciência da Computação
Av. Ipiranga, 6681 Prédio 32 Sala 505 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

23 de novembro de 2023

RESUMO

Este artigo tem como objetivo implementar e comparar uma versão paralela de um programa de multiplicação de matrizes usando MPI (Interface de Passagem de Mensagens) e OpenMP com sua contraparte sequencial. A versão sequencial, desenvolvida em C com MPI (processo único), forma a base para aprimoramento. A implementação paralela segue um modelo mestre-escravo em diferentes nós, cada um executando múltiplas threads. Medidas de desempenho são realizadas no cluster com configurações variadas de nós e threads, e tamanhos de matrizes de 100 a 2000. O estudo oferece uma análise abrangente do aumento de velocidade e eficiência, balanceamento de carga na execução paralela e percepções sobre a eficiência computacional da multiplicação de matrizes em ambientes de computação de alto desempenho.

Palavras-chave: OpenMP; Speedup; Eficiência; MPI; Multiplicação de Matrizes; Programação Paralela; Desempenho de Cluster; Modelo Mestre-Escravo; Computação de Alto Desempenho; Balanceamento de Carga.

ABSTRACT

Title: *"Rules for the elaboration of academic works in format of article"*

This paper investigates and compare a parallel version of a matrix multiplication program using MPI (Message Passing Interface) and OpenMP against its sequential counterpart. The sequential version, developed in C with MPI (single process), forms the basis for enhancement. The parallel implementation adheres to a master-slave model across different nodes, each running multiple threads. Performance measurements are conducted on the cluster with varying node and thread configurations, and matrix sizes ranging from 100 to 2000. The study provides a comprehensive analysis of speed-up and efficiency, load balancing in parallel execution, and insights into the computational efficiency of matrix multiplication in high-performance computing environments.

Key-words: *OpenMP; Speedup; Efficiency; MPI; Matrix Multiplication; Parallel Programming; Cluster Performance; Master-Slave Model; High-Performance Computing; Load Balancing.*

1. INTRODUÇÃO

A evolução da tecnologia de computação e a crescente necessidade de processamento de dados em larga escala têm impulsionado o desenvolvimento e a otimização de algoritmos em ambientes de computação de alto desempenho. Neste contexto, a multiplicação de matrizes, uma operação fundamental em diversas aplicações científicas e de engenharia, representa um desafio significativo quando se trata de matrizes de grandes dimensões. Tradicionalmente executada de maneira sequencial, essa operação pode resultar em tempos de processamento proibitivos, especialmente em cenários que exigem cálculos rápidos e eficientes. Portanto, surge a necessidade de explorar métodos alternativos que possam acelerar esse processo, mantendo ou até mesmo melhorando a precisão dos resultados.

Diante desse cenário, a paralelização emerge como uma abordagem promissora, oferecendo uma via para a aceleração do processamento de matrizes. Este artigo tem como objetivo implementar e comparar uma versão paralela de um programa de multiplicação de matrizes com sua contraparte sequencial. A paralelização é realizada utilizando a comunicação entre processos em um ambiente distribuído seguindo os padrões definidos pelo MPI Forum (MESSAGE PASSING INTERFACE FORUM, 2015). em conjunto com as funções

de biblioteca e variáveis de ambiente oferecidas pela especificação OpenMP (OPENMP ARCHITECTURE REVIEW BOARD, 2018), uma biblioteca para programação multi-thread. Este estudo detalha o processo de transformação de um programa sequencial em uma versão paralela, utilizando um modelo mestre-escravo em um ambiente de cluster.

O principal objetivo deste estudo é avaliar o impacto da paralelização no desempenho do programa em questão. Para isso, serão investigadas métricas como speedup e eficiência em diferentes configurações, variando a quantidade de nodos disponíveis. O speedup é uma medida que compara o tempo de execução de um programa sequencial com o de sua versão paralela, enquanto a eficiência avalia o quão eficazmente os recursos computacionais são utilizados.

Além de oferecer informações valiosas sobre a eficácia da paralelização usando OpenMP e MPI, este trabalho também busca contribuir para uma compreensão mais profunda das questões e desafios associados à paralelização de algoritmos. Ao longo deste artigo, será apresentada a metodologia adotada, os resultados obtidos e a análise do desempenho do algoritmo paralelizado e refletiremos sobre as lições aprendidas.

2. DESENVOLVIMENTO

Nesta seção, será feita uma exploração detalhada da metodologia adotada no estudo, seguida de uma apresentação clara dos resultados alcançados e uma análise reflexiva deles. O objetivo é esclarecer as implicações práticas e teóricas dos dados obtidos e considerar o impacto e a eficácia da paralelização, utilizando o OpenMP e o MPI, no desempenho do programa.

2.1 Metodologia

A base do presente estudo é um programa sequencial que calcula a matriz resultante da multiplicação. As ferramentas utilizadas para a paralelização do algoritmo foram MPI e a biblioteca OpenMP. A MPI é uma ferramenta essencial para a criação de aplicações paralelas distribuídas, oferecendo um alto grau de portabilidade e eficiência em ambientes de computação de cluster. MPI proporciona uma ampla gama de funcionalidades para comunicação entre processos, incluindo envio e recebimento de mensagens, operações coletivas e gerenciamento de grupos de processos.

Durante o processo de teste da aplicação paralelizada com MPI, a ênfase foi colocada na variação do número de nodos, com as configurações variando de 1 a 4 nodos. Para cada configuração, foram medidos os tempos de execução com diversos tamanhos de matrizes para possibilitar o cálculo do *speedup* e da eficiência do programa, o que permitiu uma análise detalhada do desempenho em diferentes cenários de paralelização distribuída. Esta abordagem possibilitou a identificação de como a distribuição de carga e a comunicação entre os processos influenciam a eficiência geral da aplicação em um ambiente de cluster.

A OpenMP permite a criação de aplicações paralelas eficientes e portáveis através de um conjunto diversificado de diretivas de compilação, funções de biblioteca e variáveis de ambiente. O processo de teste envolveu a utilização de múltiplas threads pelo programa paralelizado. Ao utilizar essa biblioteca juntamente à MPI, foi possível aprimorar o desempenho do algoritmo. A seguir, será detalhado as principais decisões e abordagens implementadas no código.

Na figura 1, é possível visualizar a estratégia utilizada para enviar a segunda matriz da multiplicação (m2) do processo mestre para todos os processos escravos. Nesse caso, foi utilizado a função `MPI_Bcast` para realizar o envio da matriz para todos os processos.

```
MPI_Bcast(&m2, SIZE * SIZE, MPI_INT, MESTREID, MPI_COMM_WORLD);
```

Figura 1 – Código-fonte (programa em C)

Na Figura 2, mostra o cálculo realizado para enviar a parte da matriz 1 a ser calculada pelos outros processos. Foi utilizado a função `MPI_Send()` nela é passada o que será enviado, quantos elementos e para quem irá a mensagem, primeiro enviamos o índice da primeira linha a ser calculada e, em seguida, enviamos o pedaço da matriz a ser calculada.

```
int chunkSize = SIZE / (p - 1);
for (int i = 0; i < p - 1; ++i) {
        int offset = i * chunkSize;
        // Ajuste para enviar linhas faltantes int chunkSizeToSend = chunkSize;
        if (i == p - 2) {
                chunkSizeToSend = SIZE - offset;
        }
        // Envia dados MPI_Send(&offset, 1, MPI_INT, i + 1, 0, MPI_COMM_WORLD);
        MPI_Send(&chunkSizeToSend, 1, MPI_INT, i + 1, 0, MPI_COMM_WORLD);
        MPI_Send(&m1[offset][0], chunkSizeToSend * SIZE, MPI_INT, i + 1, 0, MPI_COMM_WORLD);
}
```

Figura 2 – Código-fonte (programa em C)

Na Figura 3, é possível perceber o processo de coleta da parte da matriz 1 que será calculada. Foi utilizado a função `MPI_Recv()` nela é passada o que será enviado, quantos elementos e quem enviou a mensagem.

```
int offset2, chunkSize2;
MPI_Recv(&offset2, 1, MPI_INT, MESTREID, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&chunkSize2, 1, MPI_INT, MESTREID, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&m1[offset2][0], chunkSize2 * SIZE, MPI_INT, MESTREID, 0, MPI_COMM_WORLD, &status);
```

Figura 3 – Código-fonte (programa em C)

Na Figura 4, é onde acontece a multiplicação das matrizes onde o escravo multiplica parte da matriz 1 recebida com a matriz 2 previamente recebida por broadcast. Nesse pedaço utilizamos a abordagem do OpenMP para deixar que cada uma das 16 *threads* calcule uma linha da matriz, para isso adicionamos um `#pragma omp parallel for` que irá criar um conjunto de threads que executarão o loop seguinte em paralelo. O `private(j, k)` especifica que as variáveis `j` e `k` são privadas para cada thread. Cada *thread* terá sua própria cópia dessas variáveis, garantindo que não haja conflito de valores entre as *threads* durante a execução do loop. A parte `shared(m1, m2, mres)` indica que as variáveis `m1`, `m2`, e `mres` são compartilhadas entre todas as threads. Isso significa que todas as *threads* podem acessar e modificar essas variáveis.

```
#pragma omp parallel for private(j, k) shared(m1, m2, mres)
for (i = offset2; i < chunkSize2 + offset2; i++)
{
        for (j = 0; j < SIZE; j++)
        {
                mres[i][j] = 0;
                for (k = 0; k < SIZE; k++)
                {
                        mres[i][j] += m1[i][k] * m2[k][j];
                }
        }
}
```

Figura 4 – Código-fonte (programa em C)

Na Figura 5, é possível ver o envio da matriz resultante da multiplicação de parte da matriz 1 com a matriz 2 para o mestre, onde irá realizar a reconstrução da matriz.

```
MPI_Send(&offset2, 1, MPI_INT, MESTREID, 0, MPI_COMM_WORLD);
MPI_Send(&chunkSize2, 1, MPI_INT, MESTREID, 0, MPI_COMM_WORLD);
MPI_Send(&mres[offset2][0], chunkSize2 * SIZE, MPI_INT, MESTREID, 0, MPI_COMM_WORLD);
```

Figura 5 – Código-fonte (programa em C)

Na Figura 6, é possível perceber o processo de coleta e consolidação de dados em uma aplicação de multiplicação de matrizes paralelizada. Situado dentro de um `for` que itera sobre o número total de processos no ambiente MPI, este bloco de código desempenha um papel crucial na arquitetura mestre-escravo do programa. Inicialmente, o código emprega a função `MPI_Recv()` para aguardar a recepção do valor da posição da primeira linha do bloco de qualquer processo escravo. Esta parte do código é fundamental, pois o valor recebido determina a posição específica na matriz resultante onde os dados processados serão inseridos. Após a recepção do valor (`offset`), o identificador do processo remetente é extraído, o que é essencial para rastrear qual nó escravo está enviando os dados.

Em sequência, o programa prossegue para receber o tamanho do bloco de dados (`chunkSize`) processado pelo nó escravo identificado. Este passo é crucial, pois define quantos elementos da matriz foram processados por esse nó específico. Finalmente, a matriz resultante, `mres`, é preenchida com os dados processados, recebidos através de outra chamada `MPI_Recv()`. Essa abordagem para coletar os resultados parciais de vários processos escravos e reuni-los no processo mestre exemplifica a eficácia da computação paralela distribuída.

```
for (int i = 0; i < p - 1; ++i)
{
        int offset, senderID;
        MPI_Recv(&offset, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        senderID = status.MPI_SOURCE;
        MPI_Recv(&chunkSize, 1, MPI_INT, senderID, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&mres[offset][0], chunkSize * SIZE, MPI_INT, senderID, 0, MPI_COMM_WORLD, &status);
}
```

Figura 6 –Código-fonte (programa em C)

2.2 Resultados

A execução do programa paralelizado sob diferentes configurações gerou dados valiosos, ilustrando a variação do desempenho do algoritmo. Na Figura 7 é possível visualizar as diferenças entre o tempo total de processamento de cada abordagem.

Ao analisarmos a tabela de desempenho do programa paralelizado, observamos uma clara demonstração do impacto da computação distribuída em diferentes configurações de nodos em comparação com a execução sequencial. Os dados revelam uma tendência de diminuição no tempo de execução à medida que o número de nodos aumenta de 1 para 4, corroborando a premissa de que a divisão de tarefas em um ambiente paralelo distribuído pode melhorar de maneira significativa a eficiência do processamento de operações computacionais intensivas. Este comportamento é consistente com os princípios fundamentais da computação paralela, onde a distribuição de carga e a execução simultânea de sub-processos visam a redução proporcional dos tempos de execução.

Comparando os tempos de execução dos cenários paralelos com a versão sequencial, identifica-se uma superioridade expressiva daqueles, com destaque para o cenário de 4 nodos, que se apresenta como o mais veloz. Esse padrão se mantém, independentemente do aumento no número de iterações, sugerindo que a implementação paralela é não apenas mais eficiente, mas também escala de forma mais eficaz com o incremento da complexidade computacional. No entanto, anomalias pontuais nos dados indicam que em certos intervalos, como entre 1700 e 1800 iterações, o aumento no número de nodos não resulta em uma redução proporcional do tempo. Tais inconsistências podem ser atribuídas ao overhead inerente à comunicação e à sincronização entre os nodos, assim como a desafios associados ao balanceamento de carga.

Portanto, a análise dos dados coletados na tabela não apenas valida a eficácia da paralelização na otimização do tempo de execução, mas também ressalta a importância de uma implementação cuidadosa para minimizar custos adicionais de comunicação. A tabela subsequente ilustra esses resultados de forma empírica, oferecendo um subsídio valioso para a compreensão aprofundada da dinâmica de desempenho em sistemas paralelos.

Iterações	1 Nodo	2 Nodos	3 Nodos	4 Nodos	Sequencial
100	0,009085	0,02377	0,017513	0,016003	0,009452
200	0,069645	0,042812	0,040384	0,023509	0,071645
300	0,203078	0,051455	0,067216	0,033502	0,190393
400	0,585361	0,10619	0,105327	0,06453	0,612001
500	0,945084	0,156508	0,165624	0,094754	0,953859
600	1,664234	0,259001	0,245517	0,200781	1,624509
700	2,683317	0,402817	0,353028	0,287897	2,656104
800	5,381904	0,767086	0,59074	0,477692	5,26261
900	6,123692	0,874618	0,716563	0,576025	6,43342
1000	8,849154	1,242436	0,97911	0,780807	9,153989
1100	11,96528	1,66078	1,317762	1,021403	12,353932
1200	15,62003	2,157206	1,667937	1,311985	17,06923
1300	20,21895	2,770582	2,103578	1,615481	22,66506
1400	25,20204	3,525403	2,659991	2,068791	30,913583
1500	31,85112	4,35787	3,284681	2,541783	39,142963
1600	61,24671	8,182168	5,864116	4,479035	64,9637
1700	46,99446	6,677298	4,908989	3,754288	61,49029
1800	56,18535	7,853349	5,928007	4,420083	70,913161
1900	67,24216	9,524411	6,96795	5,333264	91,437073
2000	80,01466	10,85553	7,929763	6,06369	90,767193
Total	443,055314	61,49129	45,913796	35,165303	528,684167

Figura 7 – Tabela de Resultados do Tempo de Execução

A Figura 8 apresenta o *speedup* alcançado pelo programa em comparação com a versão sequencial para diferentes quantidades de nodos. Foi observado um incremento notável no *speedup* com o aumento do número de nodos, sugerindo uma melhoria significativa na velocidade de execução do programa paralelizado.

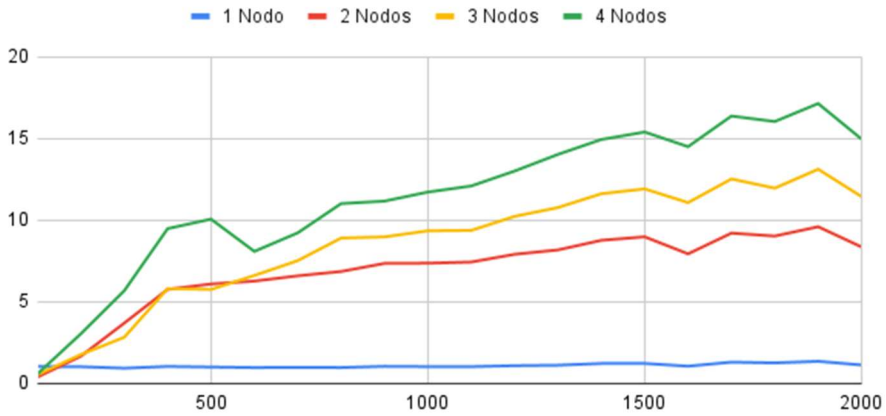

Figura 8 – Gráfico de Resultados de *Speedup*

Por outro lado, a eficiência do programa demonstrou uma tendência diversa. Como ilustrado na Figura 9, a eficiência tendia a decrescer à medida que mais nodos eram incorporados. Essa tendência sinaliza que, apesar do ganho em *speedup*, a utilização dos recursos computacionais não era proporcionalmente otimizada, levantando questões sobre a viabilidade de incrementar indefinidamente o número de nodos.

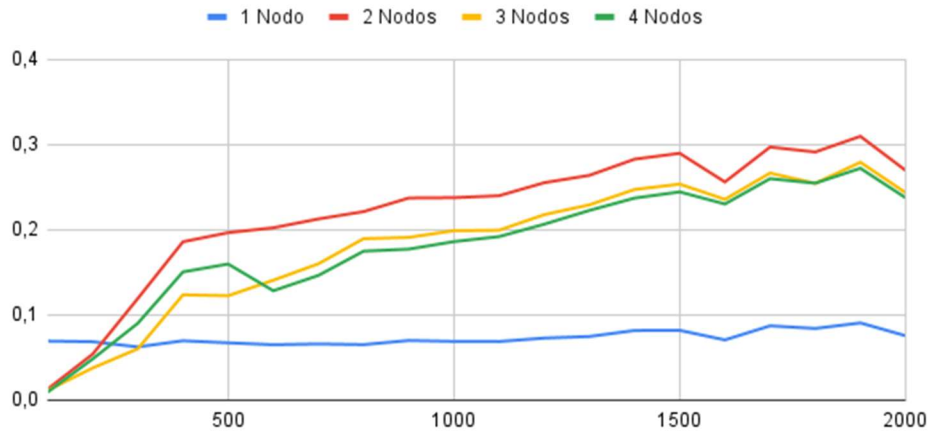

Figura 9 – Gráfico de Resultados de Eficiência

Os resultados indicam uma relação inversamente proporcional entre o *speedup* e a eficiência, ressaltando a necessidade de um balanceamento cuidadoso para garantir uma paralelização eficaz e uma utilização otimizada dos recursos disponíveis.

A análise detalhada dos resultados proporcionou revelações sobre o comportamento do algoritmo paralelizado e as implicações práticas da paralelização utilizando MPI e OpenMP. O aumento observado no *speedup* é um indicativo positivo do potencial da paralelização, demonstrando que o emprego de múltiplos nodos pode, de fato, acelerar significativamente o processo de cálculo de multiplicação de matrizes.

No entanto, a diminuição concomitante da eficiência traz à tona considerações cruciais sobre a alocação de recursos e a estratégia de paralelização. A eficiência reduzida sugere que, embora o programa execute mais rapidamente com um número maior de nodos, a utilização de recursos não é otimizada, resultando em uma sobrecarga que pode mitigar os benefícios do *speedup*.

Essas observações ressaltam a importância de uma abordagem ponderada na escolha do número de nodos e na estratégia de paralelização, visando não apenas a maximização do desempenho, mas também a otimização da utilização dos recursos computacionais.

3. CONCLUSÃO

Ao final deste estudo, foi possível observar o impacto significativo da paralelização no cálculo de multiplicações de matrizes utilizando as bibliotecas MPI e OpenMP. A análise detalhada dos resultados revelou um aumento no *speedup* com o incremento do número de nodos, evidenciando a eficácia da paralelização em melhorar o desempenho do programa. No entanto, a tendência de diminuição da eficiência ao aumentar o número de threads ressaltou a importância de um balanceamento cuidadoso entre ganho de desempenho e utilização ótima dos recursos.

As estratégias de paralelização empregadas no código demonstraram ser cruciais para maximizar o desempenho e garantir a corretude dos resultados. A utilização de diretivas do OpenMP e a troca de mensagens proporcionada pelo MPI, foram elementos-chave para evitar condições de corrida e assegurar a integridade dos dados.

Este estudo também proporcionou uma reflexão valiosa sobre os desafios e considerações inerentes à paralelização de algoritmos. A necessidade de equilibrar *speedup* e eficiência, bem como a importância de

uma estratégia de paralelização bem pensada, são aspectos cruciais para o sucesso da implementação de programas paralelos em contextos reais.

4. REFERÊNCIAS

OpenMP Architecture Review Board. **OpenMP Application Program Interface Version 5.0. [S.l.]: OpenMP, 2018**. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Acesso em: 22 nov. 2023.

Message Passing Interface Forum. MPI: **A Message-Passing Interface Standard Version 3.1. [S.l.]: MPI Forum, 2015**. Disponível em: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Acesso em: 22 nov. 2023.