

Paralelização de um Programa com OpenMPI: Avaliação de Desempenho em Computação Distribuída

Carlos Davi Gomes Pereira¹

¹Instituto de Engenharia e Geociências – Universidade Federal do Oeste do Pará (UFOPA) – Santarém – PA – Brasil

carlos.dgp@discente.ufopa.edu.br

Resumo. *Este artigo apresenta os resultados da paralelização do processo de compilação de um programa em C que realiza a contagem de números primos de forma paralela. Para isso, foi utilizada a plataforma Open MPI, uma implementação de código aberto do padrão MPI (Message Passing Interface), em um ambiente computacional distribuído.*

1. Introdução

A crescente demanda por processamento eficiente impulsionou o desenvolvimento de soluções computacionais distribuídas, capazes de dividir tarefas complexas entre múltiplos nós de processamento. Dentro desse contexto, a *Message Passing Interface* (MPI) se destaca como um dos principais paradigmas para a implementação de paralelismo em clusters de computadores. Além disso, o uso de técnicas de paralelismo e a distribuição de tarefas entre múltiplos processadores são fundamentais para otimizar o tempo de execução de programas científicos e aplicações de alto desempenho [Coelho 2018].

Este trabalho apresenta a implementação e avaliação de um programa paralelo para contagem de números primos utilizando a biblioteca *OpenMPI*, executado em um cluster *Beowulf* configurado num ambiente virtualizado. O objetivo é investigar o impacto da paralelização no desempenho computacional e avaliar os ganhos obtidos com a distribuição da carga de trabalho.

Ao longo do estudo, foi discutido a configuração do ambiente, a implementação do programa e os resultados obtidos a partir de testes práticos. A análise considera métricas como tempo de execução, permitindo uma compreensão mais profunda das vantagens e limitações do uso de *OpenMPI* em cenários de computação distribuída.

2. Ambiente Utilizado

O cluster foi configurado em uma máquina *host* que suporta a virtualização de múltiplos sistemas operacionais. Esse tipo de configuração é amplamente utilizado em clusters *Beowulf*, pois permite a criação de sistemas distribuídos de alto desempenho com *hardware* convencional e *software* livre, como distribuições GNU/Linux e bibliotecas de comunicação MPI e PVM [Tonidandel 2008]. A estrutura montada contém um nó mestre e quatro nós de processamento, assim como mostra a Figura 1.



Figura 1: Máquinas Virtuais

Fonte: O autor

A máquina mestre é uma máquina virtual com 2 núcleos de CPU, 2GB de RAM e 25GB de armazenamento, enquanto os nós de processamento são 4 máquinas virtuais, com as mesmas especificações de *hardware* da máquina mestre. Todas as máquinas estão conectadas via *Ethernet* em uma rede local.

3. Montagem do Cluster Beowulf

Para montar o cluster, seguiu-se algumas etapas básicas, começando pela instalação do sistema operacional e indo até a configuração do *OpenMPI*. O *Linux Mint* foi instalado em todas as máquinas virtuais, e cada uma recebeu um nome específico (master, node1, node2, node3 e node4). Foram atribuídos IPs fixos e ajustado o arquivo */etc/hosts* para que as máquinas se reconheçam pelo nome.

Foi criado um usuário *mpiuser* em todas as máquinas para rodar os processos MPI. O diretório */home/mpiuser* foi compartilhado pelo nó mestre e montado automaticamente nos nós de processamento. Foram geradas chaves SSH e distribuídas entre os nós para permitir login sem senha. O *OpenMPI* foi baixado, compilado e instalado manualmente em todas as máquinas. Durante a configuração de um cluster, é fundamental considerar aspectos como latência da comunicação, largura de banda e balanceamento de carga para garantir a eficiência da distribuição de tarefas [Rocha 2007].

Na Figura 2 é possível observar um fluxograma com as etapas de instalação e configuração do cluster, mostrando a sequência de ações.

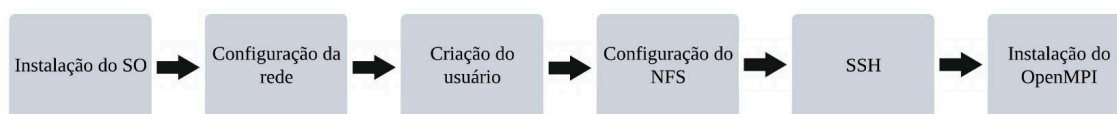


Figura 2: Fluxograma da Configuração do Cluster

Fonte: O autor

Os detalhes completos de cada uma dessas etapas, assim como o programa de teste, estão documentados em um repositório no GitHub, disponível em:

<https://github.com/CarlosDavi12/openmpi-cluster.git>

O diagrama de componentes da Figura 3 ilustra a estrutura física do cluster *Beowulf*, destacando a conexão entre o nó mestre (master) e os nós de processamento (node1, node2, node3 e node4) através de uma rede local *Ethernet*. Além disso, o diagrama mostra os serviços compartilhados, como o *NFS (Network File System)* para o diretório `/home/mpiuser` e a configuração de *SSH* para comunicação segura entre nós.

A integração do *OpenMPI* é representada, mostrando como ele gerencia a execução dos processos MPI e a comunicação entre os nós.

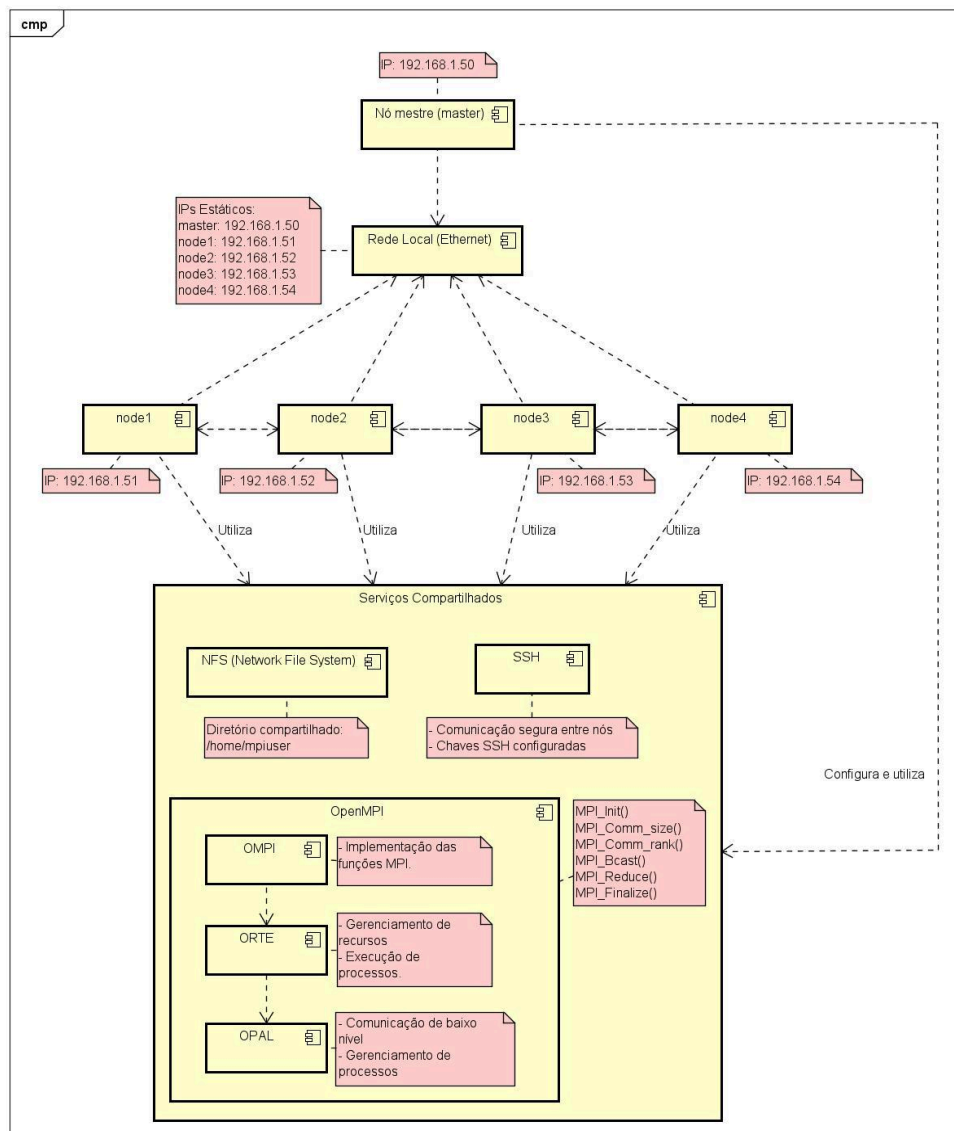


Figura 3: Diagrama de Componentes do Cluster *Beowulf*

Fonte: O autor

4. Fundamentos da Programação Paralela com MPI

O *OpenMPI* é composto por diferentes camadas, cada uma desempenhando um papel fundamental na execução de programas paralelos. A Figura 4 ilustra essa organização, destacando a relação entre os componentes principais.

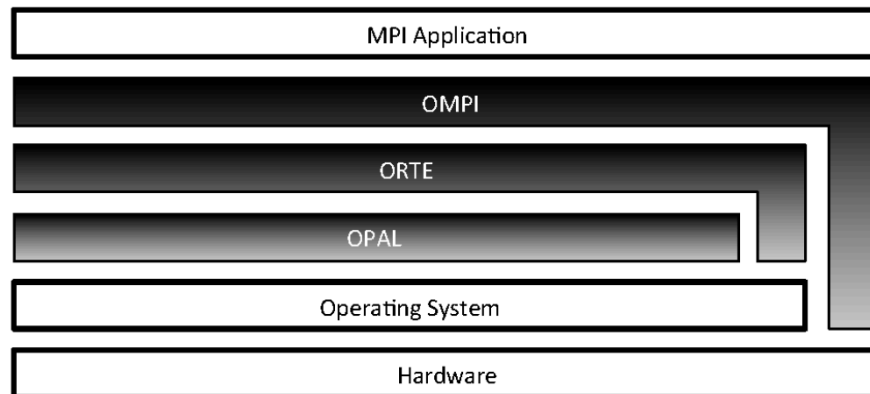


Figura 4: Arquitetura do *OpenMPI*

Fonte: SOJODI, 2021

No topo da hierarquia está a aplicação MPI, que interage diretamente com a camada *Open MPI* (OMPI). Essa é a camada mais alta do OpenMPI e a única visível para os aplicativos. Ela implementa a API do MPI, garantindo que a comunicação entre os processos ocorra conforme o padrão MPI.

Abaixo do OMPI está o *Open MPI Run-Time Environment* (ORTE), responsável por iniciar, monitorar e encerrar trabalhos paralelos. Ele gerencia a execução dos processos MPI distribuídos entre diferentes instâncias do sistema operacional, garantindo que funcionem como uma unidade coesa.

Na base das abstrações do *OpenMPI* está o *Open Portable Access Layer* (OPAL). Essa camada lida com operações mais baixas e específicas de cada sistema, como manipulação de memória compartilhada, descoberta de interfaces de rede, afinidade entre processadores e até temporizadores de alta precisão. O OPAL também fornece utilitários básicos, como estruturas de dados genéricas e ferramentas de depuração, tornando o *OpenMPI* compatível com diferentes sistemas operacionais.

Por fim, sustentando todas essas camadas, temos o sistema operacional e o *hardware*, que fornecem os recursos necessários para a execução das aplicações paralelas.

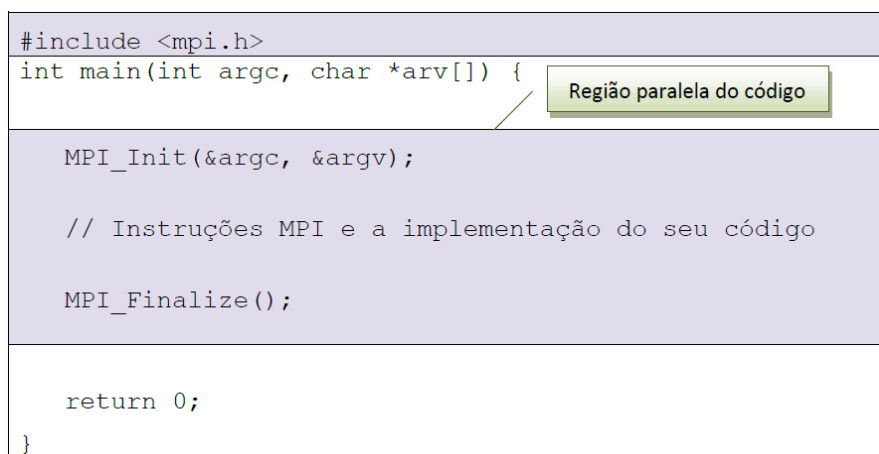
Alguns conceitos fundamentais do MPI são:

- **Processo:** programa que pode ser dividido para execução em múltiplos processadores.

- **Grupo:** conjunto de processos. O comunicador padrão é o MPI_COMM_WORLD.
- **Rank:** identificação única de cada processo em um comunicador, determinado pela função MPI_Comm_rank.
- **Comunicador:** estrutura que define o domínio da comunicação.
- **Buffer de Aplicação e Sistema:** áreas de memória para armazenar dados transmitidos entre os processos.

Para iniciar a programação com MPI, é necessário incluir a biblioteca mpi.h, e o código deve ser envolvido entre os métodos MPI_Init e MPI_Finalize. O método MPI_Init recebe dois parâmetros: o número de argumentos e um vetor de caracteres com os argumentos, semelhantes aos parâmetros da função main em C. A execução do ambiente MPI é iniciada com a ferramenta mpirun, que também define o número de processos através da opção -np. A comunicação entre os processos pode ocorrer de maneira coletiva utilizando os métodos MPI_Bcast e MPI_Reduce. O método MPI_Bcast é utilizado para distribuir um dado de um processo para todos os outros dentro de um comunicador. Já MPI_Reduce permite combinar os dados de todos os processos em um único valor aplicando uma operação, como soma ou média, enviando o resultado para um processo específico. Essas funções facilitam a sincronização e agregação de dados no processamento paralelo.

A estrutura de código apresentada na Figura 5 é a base para qualquer aplicação que utilize MPI.



```
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    // Instruções MPI e a implementação do seu código

    MPI_Finalize();

    return 0;
}
```

Região paralela do código

Figura 5: Estrutura de um código MPI em C

Fonte: Pereira, 2014

5. Programa de Teste

Esta seção descreve o programa desenvolvido para avaliação da compilação paralela e os critérios a serem avaliados.

Para verificar se o cluster estava funcionando corretamente, foi utilizado um programa chamado primos_mpi_parallel.c, que distribui a contagem de números primos

entre os nós usando a biblioteca MPI. O código foi compilado com mpicc e rodado usando mpirun, permitindo a execução distribuída.

5.1 Descrição do Programa de Teste Desenvolvido

O programa `primos_mpi_parallel.c` utiliza *OpenMPI* para realizar a contagem de números primos de forma paralela, distribuindo a carga de trabalho entre múltiplos processos. A execução inicia com a configuração do ambiente MPI por meio da chamada `MPI_Init()`, que permite identificar o número total de processos disponíveis (`MPI_Comm_size()`) e atribuir um identificador único a cada um deles (`MPI_Comm_rank()`). O processo mestre (`id = 0`) exibe informações iniciais sobre a execução, incluindo a quantidade de processos envolvidos, e controla um *loop* onde o limite superior da busca por números primos cresce exponencialmente. Em cada iteração, o mestre transmite este limite aos demais processos via `MPI_Bcast()`, garantindo que todos trabalhem sobre o mesmo intervalo. Cada processo realiza a contagem de primos de forma distribuída, percorrendo o intervalo de busca de maneira intercalada, conforme seu identificador. A verificação da primalidade ocorre por meio de divisões sucessivas, utilizando um método direto. Os resultados parciais de cada processo são então combinados pelo mestre utilizando `MPI_Reduce()`, que soma os valores individuais e gera o total global de primos encontrados.

Ao final da execução, o mestre exibe os resultados, incluindo o tempo total de processamento, e finaliza a execução com `MPI_Finalize()`. Esse modelo de processamento paralelo permite uma execução mais eficiente da contagem de primos, distribuindo a carga computacional entre vários núcleos, reduzindo o tempo total de execução em comparação com abordagens sequenciais. Entenda a lógica do programa na Figura 6.

```
INICIAR MPI
OBTENHA num_processos, id_processo

SE id_processo == 0 ENTÃO EXIBIR informações iniciais

PARA limite ← 1 ATÉ 1048576 COM fator 2 FAÇA
    SE id_processo == 0 ENTÃO tempo_inicio ← tempo_atual

    MPI_Bcast(limite)
    primos_parciais ← contar_primos(limite, id_processo, num_processos)
    MPI_Reduce(primos_parciais → total_primos)

    SE id_processo == 0 ENTÃO EXIBIR limite, total_primos, tempo_total

FINALIZAR MPI
SE id_processo == 0 ENTÃO EXIBIR conclusão
```

Figura 6: Pseudocódigo do programa de Teste

Fonte: O autor

5.2 Descrição dos Critérios de Avaliação

Para avaliar o desempenho do programa paralelo de contagem de números primos, foram utilizadas três métricas principais: tempo de execução, *speedup* e eficiência. Essas métricas permitem quantificar os ganhos obtidos com a paralelização e avaliar a eficácia da distribuição da carga de trabalho entre os processos.

- **Tempo de Execução:**

O tempo de execução foi medido para diferentes configurações do programa, variando o número de processos utilizados. Essa métrica permite comparar o desempenho do programa em execução sequencial (um único processo) com o desempenho em execução paralela (múltiplos processos). A redução no tempo de execução indica uma melhoria no desempenho devido à paralelização.

- **Aceleração (Speedup):**

O *speedup* é uma medida que quantifica a aceleração obtida ao utilizar múltiplos processos em comparação com a execução sequencial [Rocha 2007]. Ele é calculado como a razão entre o tempo de execução sequencial ($T_{sequencial}$) e o tempo de execução paralela ($T_{paralelo}$).

$$Speedup = \frac{T_{sequencial}}{T_{paralelo}}$$

Um speedup próximo ao número de processos indica que a paralelização está sendo eficiente, com a carga de trabalho bem distribuída entre os nós.

- **Eficiência:**

A eficiência mede o quão bem os recursos de processamento estão sendo utilizados. Ela é calculada como a razão entre o *speedup* e o número de processos (N):

$$Eficiência = \frac{Speedup}{N}$$

Uma eficiência próxima de 1 (ou 100%) indica que os recursos estão sendo utilizados de forma ideal, enquanto valores mais baixos sugerem que há sobrecarga ou desbalanceamento na distribuição da carga de trabalho.

Com base nessas métricas, foram realizados testes com diferentes números de processos para avaliar o comportamento do cluster e a eficácia da paralelização. Os resultados obtidos permitiram identificar os ganhos de desempenho e os possíveis gargalos no processamento distribuído, fornecendo *insights* sobre a escalabilidade do sistema.

6. Resultados e Discussões

Nesta seção, são apresentados os resultados obtidos a partir da execução do programa de contagem de números primos paralelizado com *OpenMPI*. Foram realizados testes

variando o número de processos (1, 2, 4 e 8) para avaliar o desempenho do cluster em termos de tempo de execução, *speedup* e eficiência. Os resultados foram coletados em um ambiente virtualizado com um nó mestre e quatro nós de processamento, conforme descrito na Seção 2.

6.1 Tempo de Execução

O tempo de execução foi medido para diferentes limites de busca de números primos, variando de 1 até 1.048.576, com um fator de crescimento exponencial de 2. A Tabela 1 apresenta os tempos de execução (em segundos) para cada configuração de processos.

Tabela 1. Tempo de execução (s) para diferentes números de processos

Limite de Busca	1 Processo	2 Processos	4 Processos	8 Processos
1	3	1.138	79.023	140.600
2	0	7	1.692	4.070
4	0	1	1.047	4.096
8	0	1	1.626	5.543
16	0	1	2.333	4.309
32	1	2	1.054	7.038
64	2	3	4.811	6.304
128	8	7	1.759	8.290
256	26	23	1.354	6.172
512	81	80	1.241	4.100
1.024	294	281	1.757	7.246
2.048	1.410	1.060	2.602	4.177
4.096	4.605	3.798	5.469	7.479
8.192	17.169	23.715	16.422	11.095
16.384	56.683	63.009	55.976	25.491
32.768	212.119	226.847	191.716	93.841
65.536	773.614	816.997	716.689	422.085
131.072	2.712.731	2.859.768	2.412.751	1.468.707
262.144	10.041.833	10.632.510	8.714.025	4.237.955
524.288	38.296.716	40.364.199	33.812.465	15.125.868
1.048.576	143.641.019	153.351.611	129.300.189	55.906.143

Como esperado, o tempo de execução diminui significativamente à medida que o número de processos aumenta. Por exemplo, para o limite de busca de 1.048.576, o

tempo de execução com 1 processo foi de 143,64 segundos, enquanto com 8 processos, o tempo foi reduzido para 55,91 segundos. Isso demonstra a eficácia da paralelização na redução do tempo de processamento.

6.2 Speedup

O speedup foi calculado para avaliar a aceleração obtida com a paralelização. A Figura 7 apresenta o speedup em função do número de processos para diferentes limites de busca.

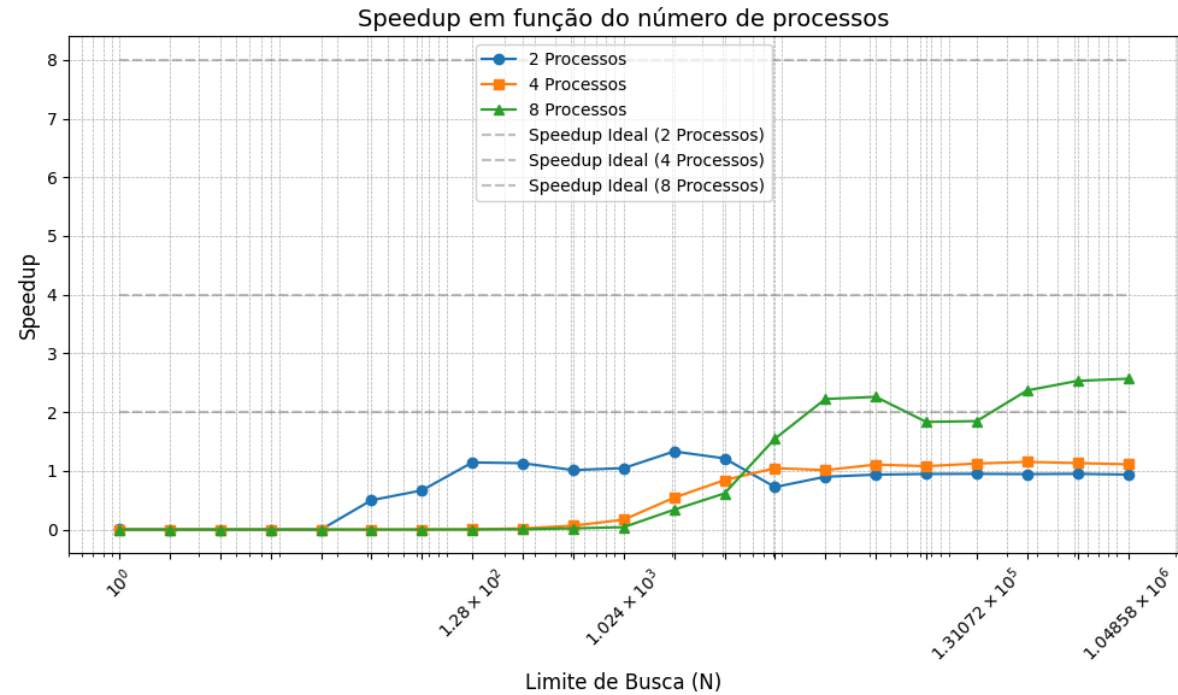


Figura 7: Speedup em função do número de processos

Fonte: O autor

O speedup ideal seria linear, ou seja, proporcional ao número de processos. No entanto, observa-se que o speedup real é inferior ao ideal, especialmente para um maior número de processos.

Por exemplo, para o limite de busca de 1.048.576, o speedup com 8 processos foi de aproximadamente 2,57, em vez de 8. Isso ocorre devido ao overhead de comunicação entre os processos e à possível desuniformidade na distribuição da carga de trabalho.

6.3 Eficiência

A eficiência foi calculada para avaliar o quão bem os recursos de processamento estão sendo utilizados. A Tabela 2 apresenta a eficiência para diferentes números de processos.

Tabela 2. Eficiência para diferentes números de processos

Limite de Busca	2 Processos	4 Processos	8 Processos
-----------------	-------------	-------------	-------------

1	0,13%	0,01%	0,00%
2	21,43%	0,21%	0,09%
4	50,00%	0,24%	0,12%
8	50,00%	0,15%	0,09%
16	50,00%	0,11%	0,14%
32	25,00%	0,47%	0,07%
64	33,33%	0,21%	0,16%
128	57,14%	0,23%	0,06%
256	56,52%	0,96%	0,21%
512	50,63%	1,63%	0,25%
1.024	52,31%	4,18%	0,51%
2.048	66,51%	13,55%	4,22%
4.096	60,61%	21,04%	7,70%
8.192	36,20%	26,14%	19,34%
16.384	44,99%	25,32%	27,80%
32.768	46,76%	27,66%	28,26%
65.536	47,34%	26,99%	22,91%
131.072	47,43%	28,11%	23,09%
262.144	47,22%	28,81%	29,62%
524.288	47,44%	28,32%	31,66%
1.048.576	46,83%	27,78%	32,12%

A eficiência tende a aumentar com o aumento do limite de busca, pois o overhead de comunicação se torna menos significativo em relação ao tempo total de execução. Para limites de busca menores, a eficiência é mais baixa, especialmente com 8 processos, devido ao maior impacto relativo da comunicação. Por exemplo, para o limite de 1.024, a eficiência com 8 processos foi de apenas 0,51%, enquanto para o limite de 1.048.576, a eficiência aumentou para 32,12%.

6.4 Discussões

Os resultados demonstram que a paralelização do programa de contagem de números primos utilizando OpenMPI é eficaz para reduzir o tempo de execução, especialmente para problemas de grande escala. No entanto, o speedup e a eficiência não atingem valores ideais, principalmente devido ao overhead de comunicação entre os processos e à possível desuniformidade na distribuição da carga de trabalho. Para melhorar o desempenho, algumas estratégias podem ser consideradas. A otimização da

comunicação, reduzindo o número de operações ou utilizando técnicas de agrupamento de mensagens, pode diminuir o overhead. O balanceamento de carga também é uma alternativa, pois um esquema mais eficiente garante que todos os processos sejam igualmente utilizados. Além disso, aumentar o tamanho do problema pode melhorar a eficiência, já que, em problemas maiores, o impacto relativo da comunicação se torna menos significativo.

Em resumo, a paralelização com OpenMPI mostrou-se uma abordagem viável para a contagem de números primos, com ganhos significativos de desempenho em cenários de grande escala. No entanto, é necessário considerar as limitações impostas pela comunicação e distribuição de carga para maximizar a eficiência do sistema.

7. Considerações Finais

A paralelização do programa de contagem de números primos com OpenMPI mostrou-se eficaz na redução do tempo de execução, especialmente para problemas de grande escala. Os testes evidenciaram ganhos de desempenho, mas também mostraram que o speedup e a eficiência não cresceram de forma linear, devido ao overhead de comunicação e ao desbalanceamento de carga. Para melhorar o desempenho, seria interessante otimizar a comunicação, equilibrar melhor a carga de trabalho e testar algoritmos mais eficientes. Além disso, a escolha do tamanho do problema é essencial para maximizar os benefícios da paralelização.

No geral, o estudo confirma que o OpenMPI é uma solução viável para computação distribuída, desde que suas limitações sejam consideradas. Trabalhos futuros podem explorar otimizações e comparar diferentes abordagens de paralelismo.

8. Referências

- SOJOODI, Amir. OpenMPI For Dummies. Disponível em: <https://amirsojoodi.github.io/posts/OpenMPI-For-Dummies/>. Acesso em: 25 fev. 2025.
- PEREIRA, F. J. Avaliação da plataforma Open MPI para paralelização do processo de compilação de software. 2014. Trabalho de Conclusão de Curso (Bacharelado em Tecnologias da Informação e Comunicação) – Universidade Federal de Santa Catarina, Campus Araranguá, Araranguá, 2014.
- COELHO, S. A. Introdução à Computação Paralela com o OpenMPI. Disponível em: <https://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/DSM/Introducao-a-Computacao-Paralela-com-o-OpenMPI.pdf>. Acesso em: 26 fev. 2025.
- ROCHA, R. Métricas de Desempenho - Programação Paralela e Distribuída. Universidade do Porto, 2007.
- TONIDANDEL, D. A. V. Manual de Montagem de um Cluster Beowulf sob a Plataforma GNU/Linux. Universidade Federal de Ouro Preto, 2008.