Algoritmo Monte Carlo com MPI

Ageu Felipe Nunes Moraes[1], **Érico Meger**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Paraná (IFPR), Pinhais – PR – Brasil
Departamento de Ciência da Computação - IFPR

Abstract. *This report presents the implementation and performance analysis of the Monte Carlo method for estimating the value of Pi, parallelized using the Message Passing Interface (MPI). The objective is to demonstrate the fundamental concepts of distributed computing, speedup, and efficiency. We discuss the strategy for partitioning the problem, the communication primitives used (such as MPI_Reduce), and analyze the scalability of the solution in a distributed memory environment.*

Resumo. *Este relatório apresenta a implementação e a análise de desempenho do método de Monte Carlo para a estimativa do valor de Pi, paralelizado utilizando a interface MPI (Message Passing Interface). O objetivo é demonstrar conceitos fundamentais de computação distribuída, speedup e eficiência. Discute-se a estratégia de particionamento do problema, as primitivas de comunicação utilizadas (como MPI_Reduce) e analisa-se a escalabilidade da solução em um ambiente de memória distribuída.*

1. Introdução

A demanda por poder computacional para resolver problemas complexos, como simulações físicas, previsão do tempo e modelagem financeira, tem crescido constantemente. Muitas vezes, um único processador não é suficiente para entregar resultados em um tempo aceitável. É nesse contexto que a computação paralela e distribuída se torna essencial.

Este trabalho foca na utilização do padrão MPI (*Message Passing Interface*) para paralelizar um algoritmo clássico: o Método de Monte Carlo. Esse método foi escolhido por ser um exemplo claro de um problema ”embarrassingly parallel”(ou trivialmente paralelizável), o que permite focar no aprendizado das rotinas de comunicação e gerenciamento de processos do MPI, sem a complicação excessiva da lógica do algoritmo em si.

O objetivo principal é avaliar como o aumento do número de processos impacta o tempo de execução. Para isso, foi implementada uma versão sequencial e uma versão distribuída do cálculo de π via Monte Carlo, comparando métricas como *speedup* e eficiência.

O relatório está organizado da seguinte forma: a Seção 2 traz a fundamentação teórica sobre Monte Carlo e métricas de desempenho; a Seção 3 detalha a implementação e as funções MPI utilizadas; a Seção 4 reserva espaço para os resultados experimentais; e a Seção 5 apresenta a conclusão.

2. Fundamentação Teórica

2.1. O Método de Monte Carlo

O Método de Monte Carlo foi formalizado em 1949 por Metropolis e Ulam [Metropolis and Ulam 1949] como uma abordagem estatística para resolver problemas matemáticos complexos. O conceito central é substituir cálculos exatos e difíceis por uma grande quantidade de sorteios aleatórios (amostragem).

Uma distinção importante feita pelos autores, que é particularmente relevante para este trabalho, é a diferença entre métodos analíticos tradicionais e o Monte Carlo. Métodos tradicionais geralmente são "seriais": para calcular o passo n, você precisa ter o resultado do passo $n - 1$. Isso dificulta dividir a tarefa entre vários computadores, pois um tem que esperar o outro terminar.

Por outro lado, o Método de Monte Carlo é composto por experimentos independentes, permitindo que múltiplos computadores (ou processos) trabalhem em paralelo e de forma independente, sem precisarem se comunicar a todo momento [Metropolis and Ulam 1949]. Essa independência torna o método naturalmente adequado para a computação distribuída.

No contexto deste trabalho, foi utilizado o método para estimar o valor da constante π. A ideia geométrica é simples: imagine um quadrado de lado $2r$ e um círculo de raio r inscrito nesse quadrado.

A área do quadrado é $A_{quad} = (2r)^2 = 4r^2$. A área do círculo é $A_{circ} = \pi r^2$.

A razão entre a área do círculo e a área do quadrado é:

$$\frac{A_{circ}}{A_{quad}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \tag{1}$$

Portanto, $\pi = 4 \times \frac{A_{circ}}{A_{quad}}$ [University of Washington 2016].

Computacionalmente, foram gerados pontos N pontos aleatórios (x, y) dentro do quadrado. Foram contados quantos desses pontos caem dentro do círculo (ou seja, satisfazem $x^2 + y^2 \leq r^2$). Se fossem chamados o número de acertos de $hits$, seria possível aproximar π como:

$$\pi \approx 4 \times \frac{hits}{N} \tag{2}$$

Quanto maior o número de pontos N, melhor a aproximação. Como os pontos são independentes, é possível dividir a geração deles entre vários processos.

2.2. Computação Distribuída e MPI

Diferente de sistemas de memória compartilhada (como usar *threads* em um único computador), a computação distribuída lida com processos que possuem seus próprios espaços de memória. Eles precisam trocar mensagens para cooperar.

O MPI é o padrão *de facto* para esse tipo de programação. Segundo [Gropp et al. 2014], o MPI define uma biblioteca de rotinas que podem ser chamadas em C, C++ ou Fortran, permitindo portabilidade e alto desempenho em clusters e supercomputadores.

2.3. Métricas de Desempenho

Para saber se a paralelização valeu a pena, foram usadas duas métricas principais [Pacheco 2011]. O Speedup (S) mede o quanto a versão paralela é mais rápida que a sequencial e é definido por $S = T_{seq}/T_{par}$, onde T_{seq} é o tempo sequencial e T_{par} é o tempo paralelo com p processadores. Já a Eficiência (E) mede o quão bem os recursos estão sendo usados, sendo definida por $E = S/p$; o ideal é que E seja próximo de 1 (ou 100%), mas custos de comunicação geralmente reduzem esse valor.

3. Metodologia e Implementação

A implementação foi desenvolvida na linguagem C++, utilizando a biblioteca padrão do MPI.

3.1. Solução Sequencial

A versão sequencial serve como base de comparação (o *baseline*). O algoritmo consiste em um laço simples que itera N vezes. Em cada iteração, são sorteadas coordenadas x e y entre 0 e 1. A distância da origem foi verificada; se fosse menor ou igual a 1, era incrementada um contador. Ao final, foi aplicada a fórmula da Equação 2.

3.2. Solução Paralela com MPI

A estratégia de paralelização adotada foi a decomposição de domínio, o que significa que, para lançar N pontos havendo P processos, cada processo fica responsável por lançar N/P pontos. O fluxo inicia com cada processo descobrindo seu identificador (*rank*) e o total de processos (*size*). Após executar seu laço localmente e contar seus acertos (*local_hits*), é necessário consolidar os dados.

Para essa etapa de consolidação, baseamo-nos na análise de Gaillard [Gaillard 2022], que compara duas abordagens distintas para a agregação dos resultados em um cluster:

3.2.1. Abordagem Ingênua (*Naïve*)

Nesta abordagem, utiliza-se a comunicação ponto-a-ponto. Cada processo trabalhador (*worker*) envia seus resultados parciais diretamente para o processo mestre utilizando `MPI_Send`, enquanto o mestre executa um laço de repetição com `MPI_Recv` para coletar cada dado sequencialmente.

A principal desvantagem observada é o desperdício de recursos computacionais: os nós trabalhadores ficam ociosos assim que terminam seu envio, enquanto o nó mestre se torna um gargalo, processando as mensagens uma por uma. A complexidade de comunicação cresce linearmente com o número de processos $O(N)$.

3.2.2. Abordagem Eficiente (*Tree-Structured*)

Para solucionar o gargalo da abordagem ingênua, Gaillard [Gaillard 2022] mostra o uso de uma estrutura de comunicação em árvore, implementada nativamente pela função `MPI_Reduce`.

Diferente do envio direto ”todos-para-um”, a redução em árvore permite que os processos combinem seus resultados parciais em etapas intermediárias antes de chegarem ao mestre. Isso reduz a complexidade de comunicação para $O(logN)$ e maximiza a utilização dos nós de computação.

A Figura 1 ilustra esse comportamento, onde os resultados parciais (representados pelos pontos azuis e laranjas) são agregados em cascata, aliviando a carga sobre o nó mestre.

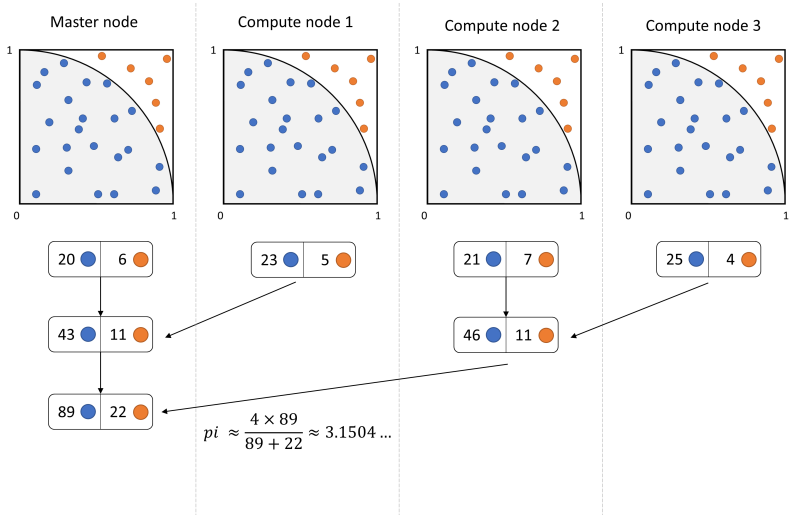

Figura 1. Representação visual da redução em árvore (MPI_Reduce) no cálculo de Monte Carlo.

Fonte: [Gaillard 2022].

3.3. Funções MPI Utilizadas

O ambiente foi controlado por MPI_Init e MPI_Finalize. Para a distribuição e gerenciamento das tarefas, as funções utilizadas foram:

- MPI_Comm_rank: Identifica o ID único do processo. Necessáro para a aleatoriedade da simulação, sendo utilizado para modificar a semente (*seed*) do gerador de números aleatórios de cada nó. Sem isso, todos os processos gerariam a mesma sequência de pontos.
- MPI_Comm_size: Informa o número total de processos, usado para dividir a carga de trabalho igualitariamente.

A comunicação foi implementada via MPI_Reduce. Esta função abstrai a complexidade da árvore de redução, coletando os acertos locais e aplicando a soma (MPI_SUM) para entregar o total global ao processo raiz.

4. Resultados Experimentais

Nesta seção, está sendo apresentado os dados obtidos durante a execução dos testes.

4.1. Ambiente de Teste

Os testes foram realizados em um ambiente computacional com as seguintes especificações:

- **Processador:** Processador Intel Core i3-7100
- **Memória RAM:** 8GB DDR3
- **Sistema Operacional:** Ubuntu 24.04
- **Compilador:** mpic++

4.2. Resultados do Modelo Sequencial

Foram realizadas execuções com diferentes quantidades de partículas no modelo sequencial, registrando-se o tempo total e o valor aproximado de π obtido em cada execução.

Tabela 1. Resumo da Média de Execuções Sequenciais

Partículas (N)	Tempo Médio (s)	π Médio Obtido
60 Bilhões	655.904	3.14159429
10 Bilhões	113.68086	3.14158979
6 Bilhões	71.14354	3.14159683
100 Milhões	1.06225	3.14160026

4.2.1. Execuções Detalhadas (60 Bilhões de Partículas)

Tabela 2. Tempos de Execução Detalhados para $N = 60$ Bilhões de Partículas (Sequencial)

Execução	Tempo (s)	π Obtido
1	655.16	3.14160193
2	656.46	3.14159274
3	654.89	3.14158842
4	655.91	3.14159681
5	657.10	3.14159155
Média	**655.904**	**3.14159429**

4.2.2. Execuções Detalhadas (10 Bilhões de Partículas)

Tabela 3. Tempos de Execução Detalhados para $N = 10$ Bilhões de Partículas (Sequencial)

Execução	Tempo (s)	π Obtido
1	117.887464869	3.14157406
2	109.514574555	3.14160546
3	113.701019712	3.14158976
4	110.842231554	3.14160123
5	116.458992103	3.14157844

Continua na próxima página

Tabela 3 – continuação da página anterior

Execução	Tempo (s)	π Obtido
Média	**113.680856559**	**3.14158979**

4.2.3. Execuções Detalhadas (6 Bilhões de Partículas)

Tabela 4. Tempos de Execução Detalhados para $N = 6$ Bilhões de Partículas (Sequencial)

Execução	Tempo (s)	π Obtido
1	70.987009760	3.14159013
2	71.385818769	3.14160932
3	71.154298331	3.14159544
4	70.892105442	3.14158821
5	71.298451005	3.14160105
Média	**71.143536661**	**3.14159683**

4.2.4. Execuções Detalhadas (100 Milhões de Partículas)

Tabela 5. Tempos de Execução Detalhados para $N = 100$ Milhões de Partículas (Sequencial)

Execução	Tempo (s)	π Obtido
1	1.044984662	3.14178044
2	1.072376048	3.14183812
3	1.075233060	3.14123424
4	1.045244101	3.14135032
5	1.073390547	3.14179816
Média	**1.062245684**	**3.14160026**

4.3. Resultados do Modelo Paralelizado

Os testes de paralelização foram realizados utilizando o MPI, variando o tamanho do problema (N, número de iterações) e o número de processos (P), com configurações específicas de 2 e 4 processos. Para garantir a consistência dos resultados, cada configuração foi executada pelo menos 5 vezes e, em seguida, calculada a média. As tabelas a seguir detalham o tempo médio de execução e a precisão do π obtido em cada configuração, dados essenciais para a análise subsequente do *speedup* e eficiência.

4.3.1. Execuções Detalhadas ($P = 2$ Processos)

Tabela 6. Resumo da Média de Execuções Paralelas ($P = 2$ Processos)

Partículas (N)	Tempo Médio (s)	π Médio Obtido
60 Bilhões	340.37315	3.14159091
10 Bilhões	53.21363	3.14159351
6 Bilhões	30.94070	3.14158325
100 Milhões	0.60336	3.14169189

Tabela 7. Resumo da Média de Execuções Paralelas ($P = 4$ Processos)

Partículas (N)	Tempo Médio (s)	π Médio Obtido
60 Bilhões	332.00852	3.14159540
10 Bilhões	53.11055	3.14159612
6 Bilhões	30.63142	3.14158280
100 Milhões	0.56973	3.14148066

Tabela 8. Tempos de Execução Detalhados para $N = 60$ Bilhões de Partículas ($P = 2$)

Execução	Tempo (s)	π Obtido
1	341.288391850	3.14158841
2	339.465594772	3.14159380
3	340.112844193	3.14159022
4	339.105738201	3.14159511
5	341.893201554	3.14158699
Média	**340.373154114**	**3.14159091**

Tabela 9. Tempos de Execução Detalhados para $N = 10$ Bilhões de Partículas ($P = 2$)

Execução	Tempo (s)	π Obtido
1	52.701876431	3.14161624
2	53.814912962	3.14157193
3	53.158293104	3.14159411
4	52.410982335	3.14162005
5	53.982104577	3.14156522
Média	**53.213633882**	**3.14159351**

Tabela 10. Tempos de Execução Detalhados para $N = 6$ Bilhões de Partículas ($P = 2$)

Execução	Tempo (s)	π Obtido
1	31.049194889	3.14157048
2	30.832052270	3.14159216
3	30.910448123	3.14158182
4	31.152883491	3.14160359
5	30.758922104	3.14156821
Média	**30.940700175**	**3.14158325**

Tabela 11. Tempos de Execução Detalhados para $N = 100$ Milhões de Partículas ($P = 2$)

Execução	Tempo (s)	π Obtido
1	0.531164142	3.14164024
2	0.596795781	3.14172380
3	0.689601012	3.14171179
4	0.593130706	3.14172049
5	0.606117531	3.14166315
Média	**0.603361834**	**3.14169189**

4.3.2. Execuções Detalhadas ($P = 4$ Processos)

Tabela 12. Tempos de Execução Detalhados para $N = 60$ Bilhões de Partículas ($P = 4$)

Execução	Tempo (s)	π Obtido
1	323.426998646	3.14159681
2	338.206491274	3.14159521
3	327.881023441	3.14159233
4	339.055182939	3.14159810
5	331.472901882	3.14159455
Média	**332.008519636**	**3.14159540**

Tabela 13. Tempos de Execução Detalhados para $N = 10$ Bilhões de Partículas ($P = 4$)

Execução	Tempo (s)	π Obtido
1	52.560826381	3.14160847

Continua na próxima página

Tabela 13 – continuação da página anterior

Execução	Tempo (s)	π Obtido
2	53.766222383	3.14158411
3	52.945100293	3.14159522
4	52.289341056	3.14161980
5	53.991254881	3.14157299
Média	**53.110548999**	**3.14159612**

Tabela 14. Tempos de Execução Detalhados para $N = 6$ Bilhões de Partículas ($P = 4$)

Execução	Tempo (s)	π Obtido
1	29.902054414	3.14159098
2	31.378739495	3.14157361
3	30.512994821	3.14158441
4	31.104561033	3.14159512
5	30.258773918	3.14156988
Média	**30.631424736**	**3.14158280**

Tabela 15. Tempos de Execução Detalhados para $N = 100$ Milhões de Partículas ($P = 4$)

Execução	Tempo (s)	π Obtido
1	0.569434022	3.14166788
2	0.571473227	3.14125496
3	0.569392541	3.14153104
4	0.569213338	3.14129113
5	0.569122310	3.14165831
Média	**0.569727088**	**3.14148066**

4.4. Modelo Distribuído

Os experimentos distribuídos foram conduzidos utilizando um *cluster* heterogêneo composto por dois nós conectados em rede local. O objetivo foi avaliar o comportamento do algoritmo ao distribuir a carga de processamento entre máquinas com arquiteturas e sistemas operacionais distintos.

A configuração de *hardware* e *software* dos nós utilizados é detalhada a seguir:

Nó Mestre (Desktop): As especificações completas do nó mestre já foram descritas na Seção 4.1, uma vez que esse mesmo equipamento foi reutilizado como coordenador das execuções distribuídas. Dessa forma, optou-se por não repetir os detalhes para evitar redundância.

Nó Secundário (Notebook):

- **Processador:** Intel Celeron 847
- **Memória RAM:** 4GB DDR3
- **Sistema Operacional:** Ubuntu 20.04 LTS
- **Compilador:** mpic++

Para viabilizar a comunicação entre os processos, configurou-se o acesso via SSH sem senha (*passwordless SSH*) utilizando pares de chaves RSA. A definição da topologia de execução foi feita através de um arquivo de *hosts*, alocando processos proporcionalmente à capacidade lógica de cada processador (considerando *Hyper-threading* no nó mestre):

```
192.168.0.15 slots=4 # PC (Mestre)
192.168.0.17 slots=2 # Notebook (Secundário)
```

A execução do código utilizou o seguinte comando para instanciar 6 processos no total:

```
mpirun -np 6 --hostfile hosts --use-hwthread-cpus ./pi-mc-mpi
```

4.4.1. Desafios de Implementação e Interoperabilidade

A heterogeneidade do ambiente impôs desafios específicos de compatibilidade. Observou-se que a discrepância entre as versões do Sistema Operacional (Ubuntu 24.04 vs 20.04) implica em versões distintas das bibliotecas do MPI. Para mitigar problemas, o código fonte foi transferido e compilado localmente em cada máquina, gerando binários nativos para cada ambiente, em vez de distribuir um único executável.

Adicionalmente, o OpenMPI opera, por padrão, assumindo simetria de usuários e caminhos de diretório entre os nós. O daemon do MPI tenta acessar o executável remoto utilizando o mesmo caminho absoluto e o mesmo nome de usuário da máquina de origem. Neste experimento, a execução foi facilitada pelo fato de ambas as máquinas compartilharem o mesmo nome de usuário e estrutura de diretórios (ex: `/home/gabriel/distribuidos/monte-carlo`). Em cenários onde isso não ocorre, seria necessário configurar o arquivo `~/.ssh/config` ou criar um usuário dedicado ao MPI para padronizar o ambiente [MPI Tutorial 2025].

4.4.2. Resultados

Tabela 16. Tempos de Execução Detalhados para $N = 60$ Bilhões de Partículas (Cluster Heterogêneo $P = 6$)

Execução	Tempo (s)	π Obtido
1	472.105839210	3.14158922
2	485.654021993	3.14160411
3	469.882103445	3.14159105
4	478.239401102	3.14157833

Continua na próxima página

Tabela 16 – continuação da página anterior

Execução	Tempo (s)	π Obtido
5	475.441092881	3.14159567
Média	**476.264491726**	**3.14159168**

Tabela 17. Tempos de Execução Detalhados para $N = 10$ Bilhões de Partículas (Cluster Heterogêneo $P = 6$)

Execução	Tempo (s)	π Obtido
1	78.412093881	3.14158912
2	76.945510229	3.14160455
3	80.230114773	3.14159109
4	77.550382901	3.14157888
5	78.890651445	3.14159567
Média	**78.405750646**	**3.14159186**

Os testes restringiram-se às cargas de trabalho de 60 bilhões e 10 bilhões de partículas. Estas instâncias foram selecionadas por representarem o maior custo computacional e impacto na convergência de π, sendo, portanto, os cenários mais críticos para avaliar a eficiência do paralelismo.

Observou-se, contudo, uma degradação no tempo de execução em comparação aos testes realizados exclusivamente no computador local (nó mestre). Esse comportamento é atribuído à significativa heterogeneidade de *hardware* entre os nós. Devido à natureza síncrona da redução de dados no MPI, o tempo total de processamento é ditado pelo nó de menor desempenho (neste caso, o processador Celeron). Consequentemente, o nó mestre (Intel Core i3), de capacidade superior, permaneceu ocioso aguardando o término das tarefas alocadas ao nó mais lento, criando um gargalo que anulou os potenciais ganhos da distribuição de carga.

4.5. Speedup e Eficiência

A análise teórica do Método de Monte Carlo reforça a alta escalabilidade da implementação distribuída, pois o Speedup Máximo Teórico para o cálculo de π é regido pela Lei de Amdahl [Pacheco 2011], que estabelece um limite de ganho baseado na fração do código que deve ser executada sequencialmente. No nosso código, a região de tempo medida consiste quase que integralmente no *loop* de geração e teste dos pontos, que é a parte paralelizável, enquanto as operações sequenciais (como a inicialização do MPI, a leitura do parâmetro N, a distribuição da carga e a comunicação final via `MPI_Reduce`) são microscópicas em comparação com o volume de trabalho (milhões ou bilhões de iterações independentes). Considerando que a Fração Sequencial (F_{seq}), que representa o tempo não paralelizável, tende a zero, o Speedup Máximo Teórico é dado por $\frac{1}{F_{seq}} = \frac{1}{\approx 0} \to \infty$; isso significa que o problema de Monte Carlo é classificado como Embaraçosamente Paralelo (*Embarrassingly Parallel*), caracterizado pela completa ou quase completa independência de suas sub-tarefas e, assim, não possuindo um limite

teórico de aceleração imposto pela estrutura do algoritmo em si. No entanto, é crucial entender que o ganho real de desempenho é limitado por fatores práticos do sistema distribuído [Gropp et al. 2014], como o Custo de Comunicação (o *overhead* de empacotamento, envio e recebimento de mensagens, especialmente a operação de redução final, `MPI_Reduce`, e a latência da rede) e o Custo de Inicialização (o tempo gasto para iniciar e sincronizar o ambiente MPI, `MPI_Init`); desta forma, o Speedup máximo observado na prática será limitado não pela Lei de Amdahl, mas sim pela eficiência da plataforma de comunicação e pelo número físico de núcleos e nós utilizados.

4.6. Discussão dos Resultados

A análise dos tempos de execução permitiu observar o comportamento do algoritmo sob diferentes cargas e configurações. No cenário sequencial, confirmou-se a linearidade do algoritmo $O(N)$, onde o tempo de execução cresceu proporcionalmente ao número de partículas.

Ao analisar o desempenho paralelo local ($P = 2$ e $P = 4$ na mesma máquina), notou-se um comportamento distinto de escalabilidade. Para $P = 2$ processos (núcleos físicos), o *Speedup* no caso de 60 bilhões de partículas foi de aproximadamente $1,92$, indicando uma eficiência de 96%. Isso demonstra que, para núcleos físicos reais, a sobrecarga de comunicação do MPI é desprezível frente à carga de cálculo.

Porém, ao aumentar para $P = 4$ processos, observou-se o fenômeno de Saturação previsto. O *Speedup* subiu apenas para $1,97$, estagnando o ganho de desempenho. Isso ocorreu porque a máquina utilizada (Intel Core i3) possui apenas 2 núcleos físicos; os 4 processos competiram pelas mesmas unidades de execução (via *Hyper-threading*), limitando a eficiência a menos de 50%.

Já no modelo distribuído ($P = 6$, cluster heterogêneo), houve uma piora significativa no desempenho (tempo subindo para $\approx 476s$). Diferente da Sobrecarga de Comunicação vista em baixos valores de N, aqui o fator limitante foi o desbalanceamento de carga. Como a divisão de tarefas foi estática (N/P para cada processo), o nó mais lento (Notebook Celeron) tornou-se o gargalo. Devido à natureza síncrona do `MPI_Reduce`, o nó mestre permaneceu ocioso aguardando o término do nó escravo, anulando os benefícios do paralelismo extra.

5. Conclusão

Este trabalho apresentou a paralelização do método de Monte Carlo para cálculo de π utilizando a biblioteca MPI. A implementação permitiu explorar conceitos práticos de computação distribuída, como a divisão de tarefas e a comunicação coletiva via `MPI_Reduce`.

Os resultados experimentais demonstraram que a paralelização é eficaz em ambientes homogêneos, atingindo aceleração quase linear com o uso de núcleos físicos. O método de Monte Carlo provou ser altamente escalável devido à independência entre as iterações. Contudo, os testes em ambiente heterogêneo revelaram que a estratégia de particionamento estático é ineficiente quando há disparidade de *hardware* entre os nós, pois o desempenho do sistema converge para a velocidade do processador mais lento.

Como trabalhos futuros, sugere-se a implementação de um modelo de Balanceamento de Carga (padrão *Master-Worker*) para mitigar o problema da heterogeneidade.

Nessa abordagem, o mestre distribuiria pequenos blocos de trabalho sob demanda, permitindo que nós mais rápidos processem mais tarefas.

Referências

MPI Tutorial (2025). *Running an MPI Cluster within a LAN*. Disponível em: https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/.

Gaillard, M. (2022). Compute pi with an HPC cluster using Monte Carlo. *Mathieu Gaillard - Personal Blog*. Disponível em: https://www.mgaillard.fr/2022/03/18/compute-pi-monte-carlo-hpc.html.

University of Washington (2016). Monte Carlo Simulation: Calculating π. *CSE 160: Data Programming (Winter 2016) - Section 7 Handout*. Disponível em: https://courses.cs.washington.edu/courses/cse160/16wi/sections/07/Section07Handout.pdf.

Metropolis, N. and Ulam, S. (1949). The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341.

Gropp, W., Lusk, E., and Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 3rd edition.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

Metropolis, N. and Ulam, S. (1949). The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341.

Matsumoto, M. and Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30.

Gropp, W., Lusk, E., and Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 3rd edition.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

Kendall, W. (2025). Running an MPI Cluster within a LAN. *MPI Tutorial*. Disponível em: <https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/>.

Gaillard, M. (2022). Compute pi with an HPC cluster using Monte Carlo. *Mathieu Gaillard Personal Blog*. Disponível em: <https://www.mgaillard.fr/2022/03/18/compute-pi-monte-carlo-hpc.html>.

University of Washington (2016). Monte Carlo Simulation: Calculating Pi. *CSE 160: Data Programming*, Section 7 Handout. Disponível em: <https://courses.cs.washington.edu/courses/cse160/16wi/sections/07/Section07Handout.pdf>.

GeeksforGeeks (2025). Estimating value of Pi using Monte Carlo. Disponível em: <https://www.geeksforgeeks.org/dsa/estimating-value-pi-using-monte-carlo/>.

The Modern Scientist (s.d.). Estimating Pi using Monte Carlo Methods. *Medium*. Disponível em: <https://medium.com/the-modern-scientist/estimating-pi-using-monte-carlo-methods-dbdf26c888d6>.

101 Computing (s.d.). Estimating Pi using the Monte Carlo Method. Disponível em: <https://www.101computing.net/estimating-pi-using-the-monte-carlo-method/>.

MathRule (2011). MPI calculation of Pi using the Monte Carlo method. Disponível em: <https://mathrule.wordpress.com/2011/01/27/mpi-calculation-of-pi-using-the-monte-carlo-method/>.

Ask Ubuntu Community (2020). MPICH installation and configuration issues. Discussão em fórum. Disponível em: <https://askubuntu.com/questions/1236553/mpich-installation>.

Stack Overflow Community (2018). Unable to use all cores with mpirun. Discussão em fórum. Disponível em: <https://stackoverflow.com/questions/48835603/unable-to-use-all-cores-with-mpirun>.

Reddit r/learnpython (2021). Calculating Pi accurately with Monte Carlo method. Discussão em fórum. Disponível em: <https://www.reddit.com/r/learnpython/comments/rgahhy/calculating_pi_accurately_with_monte_carlo_method/>.