Avaliação de Desempenho dos Algoritmos de Números Primos e Monte Carlo em Ambientes HPC

Gabriella Osório Ribeiro[1], Lucas Freire Sêmeler[2], Wanderson Roger Azevedo Dias[2]

[1]Coordenadoria do Curso Técnico em Informática (CCTI)
[2]Coordenadoria do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas (CCSTADS)
Laboratório de Arquiteturas Computacionais e Computação Paralela (LACCP)
Instituto Federal de Rondônia (IFRO)
Ji-Paraná – RO – Brasil

{gabriellaosorioribeiro, lucasemeler, wradias}@gmail.com

Resumo. *A Computação de Alto Desempenho (HPC) é crucial para aplicações que requerem processamento intensivo, abrangendo as diversas áreas da ciência. Assim, este artigo apresenta a análise de desempenho dos algoritmos de Números Primos e Monte Carlo em versões serial e paralela, usando as bibliotecas OpenMP e MPI, e executando em Raspberry Pi e no ClusterPi. Os resultados mostraram que a paralelização reduziu significativamente o tempo de execução em comparação com a versão sequencial. O OpenMP superou o MPI para entradas menores, aproveitando melhor a memória compartilhada, enquanto MPI mostrou vantagens para grandes volumes de dados. O algoritmo de Monte Carlo, beneficiou-se da paralelização apenas com entradas maiores, devido ao overhead significativo em pequenas cargas de trabalho. A implementação em OpenMP apresentou os melhores speedups, sendo ≈2,9x e ≈1,5x para os algoritmos de Números Primos e Monte Carlo, respectivamente. Assim, a escolha entre OpenMP e MPI deve considerar a arquitetura do sistema e o tamanho do problema para otimizar o desempenho computacional.*

1. Introdução

A Computação de Alto Desempenho (HPC - *High Performance Computing*) tem se tornado essencial para aplicações que exigem alta capacidade de processamento de dados (Petersen e Arbenz, 2004), como nas áreas de saúde, biologia, engenharias, petróleo e gás, climatologia, física e química (Xavier *et al.*, 2007). Essas aplicações variam em termos de requisitos de *software* e *hardware*, com a execução dos processos dependendo de parâmetros de entrada e da leitura de dados ao longo do fluxo de execução. A evolução de sistemas computacionais no mercado reflete essa demanda, como demonstrado pela lista das 500 máquinas mais poderosas do mundo (https://www.top500.org/), composta por unidades de processamento interconectadas, voltadas para diversas aplicações (Top 500, 2024).

O aprimoramento do desempenho computacional é constantemente buscado pela indústria por meio da exploração de várias formas de paralelismo. Tecnologias como arquiteturas vetoriais, multiprocessadas, *multicore* e híbridas, que combinam CPU (*Central Processing Unit*) e GPU (*Graphics Processing Unit*), têm sido amplamente adotadas, além da formação de *clusters* e *grids*. Estações de trabalho, servidores, supercomputadores e até plataformas SoC (*System on Chip*) como a Raspberry Pi aproveitam esse paralelismo, integrando *hardwares* paralelos para aumentar o desempenho e atender às crescentes necessidades da computação moderna, conforme Souto *et al.* (2007).

O uso de *clusters* baseados na plataforma Raspberry Pi tem se destacado devido ao seu baixo custo de implementação e manutenção, consolidando-se como uma alternativa acessível para experimentos e pesquisas na área de Computação Paralela. Embora menos poderosa que *clusters* tradicionais compostos por servidores de alta *performance*, a Raspberry Pi oferece uma excelente relação custo-benefício, permitindo a construção de sistemas escaláveis com um orçamento modesto. Seu consumo reduzido de energia e a simplicidade de montagem tornam esse tipo de *cluster* ideal para ambientes educacionais, laboratórios de pesquisa e prototipagem de aplicações paralelas. Além disso, o formato modular das placas Raspberry Pi facilita a expansão do sistema, possibilitando o

desenvolvimento de soluções experimentais que podem ser adaptadas de acordo com a necessidade de cada projeto. Esse tipo de plataforma é amplamente utilizado em simulações de algoritmos paralelos, aprendizado sobre distribuição de tarefas e experimentos com balanceamento de carga, representando uma ferramenta poderosa para a democratização do ensino e pesquisa em Computação Paralela (Ignácio e Dias, 2023).

A abordagem de baixo nível para programação paralela exige que os desenvolvedores utilizem técnicas de programação complexas e de baixa portabilidade, o que desestimula a maioria dos programadores e mesmo os mais experientes a evitar esse paradigma devido aos desafios associados, como problemas de depuração e sincronização (Souto *et al.*, 2007). Em termos gerais, os computadores modernos já oferecem suporte para *multithreading* e *multicore*, permitindo a decomposição de tarefas e dados para simplificar a exploração do paralelismo. O escalonamento dos *threads* ou processos, que envolve a distribuição da carga de trabalho entre os elementos de processamento, é gerenciado pelo sistema operacional, conforme Silberschatz *et al.*, (2001).

Paralelamente ao avanço no desenvolvimento de *hardware*, especialmente no que diz respeito a arquiteturas paralelas, surge à necessidade de disponibilizar recursos de programação que sejam compatíveis com os diversos ambientes computacionais. É igualmente crucial que existam mecanismos de programação capazes de integrar as diferentes arquiteturas paralelas, simplificando o processo de desenvolvimento. Para isso, uma camada de abstração é criada entre a aplicação e sua plataforma de execução, utilizando bibliotecas específicas para programação paralela, como OpenMP (*Open Multi-Processing*) (OpenMP, 2024) e MPI (*Message Passing Interface*) (MPI, 2024).

Portanto, este artigo apresenta uma análise do desempenho computacional na execução dos algoritmos de Números Primos e Monte Carlo, ambos implementados em versões sequenciais e paralelas. Para o paralelismo, foram utilizadas as bibliotecas OpenMP e MPI, e os testes foram realizados em Raspberry Pi, modelo 4B de 2 GB e em um *cluster* composto por Raspberry Pi, com 5 nós, denominado de *ClusterPi*. O restante do artigo está organizado da seguinte forma: a Seção 2 apresenta uma breve contextualização, a fim, de ambientar o estudo corrente; a Seção 3 apresenta as análises do desempenho computacional dos algoritmos implementados de forma sequencial e paralela; e a Seção 4 finaliza com as conclusões e ideias para trabalhos futuros.

2. Contextualização

O crescimento contínuo dos sistemas computacionais nos últimos anos tem sido impulsionado pelo avanço e pela exploração de arquiteturas paralelas, que utilizam múltiplas unidades de processamento para aprimorar o desempenho. Atualmente, diferentes níveis de paralelismo são explorados, desde núcleos de processamento em arquiteturas *multicore* (multiprocessadores) até *clusters* e *grids* (multicomputadores) em escala global. Essas abordagens oferecem um desempenho significativamente superior em comparação com arquiteturas mais simples. Além das inovações em *hardware*, é crucial que o *software* seja desenvolvido utilizando o paradigma de paralelismo para maximizar os benefícios de HPC e alcançar resultados superiores aos obtidos com arquiteturas menos complexas (Parhami, 2006). Dessa forma, a implementação e execução de algoritmos paralelos possibilitam a resolução de problemas complexos dentro de tempos de execução aceitáveis (Bell e Gray, 2002).

2.1. Programação Paralela

Na programação paralela é realizada a divisão de uma determinada aplicação em partes, de maneira que essas partes possam ser executadas simultaneamente, por várias unidades de processamento. Essas unidades devem cooperar entre si utilizando primitivas de comunicação e sincronização, realizando a quebra do paradigma de execução sequencial do fluxo de instruções (Gebali, 2011). Então, a programação paralela permite tirar proveito dos recursos disponibilizados pelas arquiteturas paralelas, assim, é necessário que os algoritmos das aplicações estejam preparados para operar neste tipo de arquitetura, a fim de melhorar a sua velocidade de processamento.

Para a programação em ambientes de execução paralela, foram crias as bibliotecas de comunicação paralela que possibilitam a implementação de programas paralelos em ambientes com memória compartilhada e distribuída (Jin *et al.*, 2011). As bibliotecas como OpenMP, PVM (*Parallel Virtual Machine*) e MPI possibilitam escrever programas paralelos usando as linguagens C, C++ e Fortran, para serem executados em arquiteturas paralelas (Lima *et al.*, 2016).

2.2. Biblioteca OpenMP

Conforme Chapman *et al.* (2007), OpenMP consiste em um padrão de programação paralela para arquiteturas de memória compartilhada. Sua primeira versão foi lançada em 1997 e tornou-se uma das principais APIs (*Application Program Interface*) para o desenvolvimento de aplicações paralelas. Assim, uma das formas básicas de explorar o paralelismo é fazer o uso da memória compartilhada. Nesse tipo de arquitetura, todos processadores podem acessar a memória e comunicar-se através de variáveis compartilhadas (Diaz *et al.*, 2012).

O OpenMP faz paralelismo explícito e é composto por um conjunto de diretivas de compilador (#pragma) que descrevem o paralelismo no código-fonte, junto com uma biblioteca de suporte de sub-rotinas disponível para aplicativos e variáveis de ambiente. Pode-se combinar o OpenMP com o MPI para ser executado em um *cluster* e/ou *grid*, a fim de otimizar ainda mais a utilização dos recursos disponíveis e diminuir o número de comunicações realizadas entre os diferentes nós (Pacheco, 2011). O OpenMP utiliza a diretiva #pragma, definida no padrão da linguagem C/C++. O construtor paralelo #pragma omp parallel cria uma região paralela, mas isso não significa que o trabalho será executado em paralelo.

2.3. Biblioteca MPI

MPI (*Message-Passing Interface*) (MPI, 2024) é a especificação de uma biblioteca de interface de troca de mensagens, criada a partir do acordo do *MPI Forum*, que é composto por diversas organizações participantes, incluindo fornecedores, pesquisadores, desenvolvedores de bibliotecas de *software* e usuários. O MPI representa um paradigma de programação paralela no qual os dados são movidos de um espaço de endereçamento de um processo para o espaço de endereçamento de outro processo, através de operações de troca de mensagens.

O objetivo do MPI é estabelecer um padrão portátil, eficiente e flexível para a transmissão de mensagens, no qual é amplamente usado para escrever programas de transmissão de mensagens. Como tal, o MPI é a primeira biblioteca de passagem de mensagens padronizada, independente do fornecedor. As vantagens de desenvolver *software* de envio de mensagens usando o MPI correspondem às metas de *design* de portabilidade, eficiência e flexibilidade. MPI não é um padrão IEEE (*Institute of Electrical and Electronics Engineers*) ou ISO (*International Organization for Standardization*), mas tornou-se o atual "padrão da indústria" para escrever programas de transmissão de mensagens em plataformas *High Performance Computing* (Pacheco, 2011).

As implementações da biblioteca MPI variam conforme a versão e os recursos do padrão MPI que suportam. O MPI é utilizado em plataformas de memória distribuída, compartilhada e híbrida, e é tradicionalmente implementado em C, C++ e Fortran, embora existam versões para Java, Python e IDL. O modelo de programação é baseado em memória distribuída, independentemente da arquitetura física da máquina. Embora o padrão MPI ofereça várias facilidades, a identificação de zonas paralelas e o balanceamento de carga entre os nós são responsabilidades do desenvolvedor, que deve explicitar o paralelismo no código. Assim, todo o paralelismo é explícito, e o desenvolvedor deve implementar algoritmos paralelos utilizando os construtores da biblioteca (Trobec *et al.*, 2018). Existem implementações comerciais, como Intel MPI, HP MPI e MATLAB MPI, além de opções Open Source amplamente usadas, como OpenMPI e MPICH (Lima *et al.*, 2020).

2.4. Algoritmo do Cálculo de Números Primos

Os números primos são aqueles divisíveis apenas por 1 e por eles mesmos, e são fundamentais não só na matemática, mas também em outras áreas, como na criptografia contemporânea. A segurança de diversas transações, incluindo as financeiras, depende da fatoração de grandes números primos, o que torna desafiadora a tarefa de identificar esses números. Embora encontrar números primos grandes não seja extremamente difícil, a decomposição desses números em fatores primos é notavelmente complexa, especialmente para números com muitos dígitos. Por exemplo, decompor 2.244.354 em fatores primos é muito mais complicado do que reconhecer que 35 é igual a 7 x 5, e a dificuldade aumenta exponencialmente para números com cinquenta dígitos ou mais, a ponto de supercomputadores levarem milhões de anos para fatorar números de 256 *bits* (Akel, 2024).

Assim, neste artigo analisaremos algoritmos de números primos implementados de forma sequencial e paralela para determinar a quantidade de números primos em intervalos de 1×10^6, $1 \times$

10^7 e 1 x 10^8. Foram exploradas duas plataformas: a Raspberry Pi, com análises das versões sequencial e paralela utilizando a biblioteca OpenMP, e o *ClusterPi*, que executou a versão paralela com a biblioteca MPI. A comparação dessas abordagens oferece uma visão detalhada do desempenho de cada plataforma e da eficiência dos algoritmos.

A pesquisa proporciona *insights* sobre o desempenho dos algoritmos de cálculo de números primos em diferentes contextos computacionais, evidenciando a *performance* da Raspberry Pi (modelo 4B) e do *ClusterPi*. A análise comparativa entre as implementações sequencial e paralela não só revela a eficiência de cada abordagem, mas também abre possibilidades para futuras pesquisas na otimização de cálculos complexos e na adaptação de algoritmos a diferentes ambientes computacionais.

2.5. Algoritmo de Monte Carlo

Monte Carlo é um termo que se refere a métodos que utilizam amostragens aleatórias repetidas para resolver problemas, observando se a fração de amostras atende a uma propriedade estabelecida (Weisstein, 2024). Neste artigo, o algoritmo de Monte Carlo é utilizado para calcular o valor de π, e são apresentadas as execuções sequencial e paralela para avaliar os resultados obtidos.

Para calcular o valor de π, é usado a área da circunferência de raio *r*. Quando *r = 1*, a área da circunferência é igual a π. O método consiste em colocar a circunferência dentro de um quadrado e gerar pontos aleatórios dentro dessa figura. Se um ponto cai dentro da circunferência, é contado como parte da área da circunferência. A proporção de pontos dentro da circunferência em relação ao total de pontos no quadrado, multiplicada por 4, fornece uma estimativa de π, conforme a fórmula: $\pi = 4 * (P_{in} / P_{total})$, onde P_{in} é o número de pontos dentro da circunferência e P_{total} é o total de pontos gerados.

A paralelização do algoritmo de Monte Carlo, tanto em OpenMP quanto em MPI, foi realizada o particionamento do processo que verifica e contabiliza a quantidade de pontos gerados dentro do círculo, para assim a unidade mestre de processamento receber as quantidades de pontos e calcular o valor de π.

As execuções dos algoritmos sequencial e paralelos foram realizadas com diferentes quantidades de pontos aleatórios: 1 x 10^6, 5 x 10^6 e 1 x 10^7. Quanto maior o número de pontos gerados, mais próximo o resultado se aproxima do valor real de π, demonstrando a eficácia do método de Monte Carlo para estimar π com precisão crescente conforme aumenta o número de amostras.

3. Resultados e Discussão

Para o ambiente de testes, utilizou-se a plataforma Raspberry Pi 4, modelo B, com CPU *quad-core* ARM Cortex-A72 de 1.5 GHz, arquitetura ISA ARMv8-A de 64 *bits* e 2GB de memória RAM. Foi montado um *cluster*, denominado de *ClusterPi*, composto por cinco nós, sendo um nó mestre e quatro nós escravos, todos operando com o sistema operacional *Raspbian OS*, para a interconexão dos nós foi utilizado um *switch* gerenciável *Gigabit* de 8 portas TP-Link TL-SG108pe PoE. A infraestrutura de *software* incluiu o compilador *GCC* (*GNU Compiler Collection*) na versão 8.3 e as bibliotecas OpenMP (versão 4.5) e MPI (versão 3.4.1) para explorar o paralelismo. Foram implementados os algoritmos de cálculo de Números Primos e Monte Carlo, tanto na versão sequencial quanto na paralela, utilizando as bibliotecas OpenMP e MPI. O *setup* montado para os testes permitiu realizar uma avaliação do desempenho computacional dos algoritmos em um ambiente de computação serial e paralela, demonstrando as vantagens de tais abordagens em um contexto de *hardware* acessível e de baixo custo. A Figura 1 apresenta o *setup* montado para a realização dos testes.

As limitações da Raspberry Pi e do *ClusterPi* em tarefas de paralelização se devem a fatores de *hardware* e comunicação que afetam o desempenho. O processador ARM Cortex-A72, embora eficiente em consumo energético, opera em 1.5 GHz, o que limita seu desempenho em cálculos intensivos e execução de algoritmos paralelos complexos. Com apenas 2GB de RAM por nó, o sistema pode recorrer à memória *swap*, prejudicando a performance. Além disso, a arquitetura ARM tem menor largura de banda de memória e resfriamento limitado, resultando em *throttling* térmico em execuções longas.

Figura 1. Setup para a realização dos testes

Após os testes, foram analisados os resultados das execuções dos algoritmos de Números Primos e Monte Carlo. Conforme apresentado anteriormente, as versões dos algoritmos sequencial e paralela (com OpenMP), utilizaram uma única plataforma Raspberry Pi. Na versão sequencial, o algoritmo executou em um único núcleo do processador da Raspberry Pi, enquanto que na versão paralela, usando a biblioteca OpenMP, operou com os quatro núcleos disponíveis na plataforma. Por outro lado, as versões paralelas dos algoritmos implementados com a biblioteca MPI foram executados no *ClusterPi*, utilizando apenas um núcleo de cada nó. Como esperado, o tempo de execução dos algoritmos paralelos foram significativamente menor comparado aos algoritmos sequenciais, demonstrando uma diferença notável e relevante para esta métrica analisada. As Tabelas 1 e 2 detalham os tempos de execução e o *speedup* dos algoritmos de Números Primos e Monte Carlo, respectivamente, junto com os valores de entrada utilizados nos experimentos, medidos em segundos. Os resultados ressaltam a eficácia dos métodos de paralelismo na redução do tempo de processamento, evidenciando as vantagens de utilizar bibliotecas como OpenMP e MPI para otimizar o desempenho computacional em plataformas de *hardware* acessíveis e de baixo custo. Além disso, a análise do tempo de execução permite identificar as melhores práticas e configurações para maximizar a eficiência e a utilização dos recursos computacionais disponíveis.

Tabela 1. Execução do algoritmo de Números Primos

Iteração	**Intervalo [1, 1.000.000]**			**Intervalo [1, 10.000.000]**			**Intervalo [1, 100.000.000]**		
	Serial	**OpenMP** (4 *cores*)	**MPI** (5 nós)	**Serial**	**OpenMP** (4 *cores*)	**MPI** (5 nós)	**Serial**	**OpenMP** (4 *cores*)	**MPI** (5 nós)
1	1,158	0,395	0,905	29,551	10,137	15,180	779,229	265,409	366,445
2	1,164	0,400	0,832	29,429	9,955	14,509	779,943	265,922	326,440
3	1,226	0,437	0,991	29,503	10,080	17,554	779,151	271,152	307,113
4	1,160	0,398	1,022	29,411	10,152	14,668	779,392	272,546	306,472
5	1,231	0,395	0,888	30,469	10,055	16,419	799,451	273,054	305,390
Média	**1,190**	**0,405**	**0,928**	**29,673**	**10,076**	**15,666**	**783,450**	**269,620**	**322,372**
Speedup	**--**	**2,93**	**1,28**	**--**	**2,94**	**1,89**	**--**	**2,91**	**2,43**

Conforme apresentado na Tabela 1, cada algoritmo foi executado cinco vezes e, com isso, obtivemos a média dos tempos de execução e o *speedup* das versões paralelas. Para os algoritmos de Números Primos, foram calculados os intervalos de 1, 10 e 100 milhões. As médias dos tempos de execução para a versão sequencial, alterando o intervalo de entradas, foram de 1,190s, 29,673s e 783,450s, respectivamente. Já na versão implementada em OpenMP, as médias dos tempos de execuções foram de 0,405s, 10,076s e 269,620s, respectivamente, atingindo um *speedup* máximo de 2,94x para o intervalo de 10 milhões. A versão implementada em MPI executou em 0,928s, 15,666s e 322,372s, respectivamente, com um *speedup* máximo de 2,43x para o intervalo de 100 milhões. Assim, observando a Tabela 1, podemos observar que os algoritmos que utilizam técnicas de execução em paralelo apresentaram ganhos substanciais em comparação com a implementação sequencial, demonstrando eficiência computacional através do uso da paralelização.

Na Tabela 1, também é possível observar que a execução do algoritmo na versão em OpenMP foi mais rápida que a do algoritmo implementado com MPI. Para entradas de 1 milhão, a versão em OpenMP executou 56% mais rápido do que a versão em MPI. Já com a entrada de 100 milhões, a implementação com OpenMP ainda continuou sendo mais rápida, mas a diferença se tornou menor, reduzindo para 16%. Um dos motivos para a implementação em OpenMP ser mais rápida é que os núcleos acessam a memória local e compartilhada, enquanto na versão em MPI, faz o uso de memória distribuída e é necessária a troca de mensagens, sincronização na execução e balanceamento de carga. No entanto, é importante destacar que a implementação usando a biblioteca MPI apresenta um aumento na *performance* e eficiência de processamento para entradas maiores, o que explica a diminuição da diferença entre os tempos de execução nas duas implementações.

Ao analisarmos os valores dos tempos de execução é evidente que a paralelização com OpenMP traz maiores benefícios em cenários onde a comunicação entre núcleos pode ser minimizada e a memória compartilhada pode ser explorada de maneira mais eficiente. O *speedup* de 2,93x alcançado com OpenMP em comparação ao 1,28x obtido pelo MPI, para o intervalo de 1 milhão, indica uma superioridade clara em ambientes onde a latência de comunicação é crítica. Isso pode ser justificado pelo *overhead* associado à comunicação entre processos em MPI, que, apesar de ser eficiente para grandes volumes de dados, conforme aumenta a quantidade de entradas, não consegue superar as vantagens do acesso direto à memória compartilhada oferecido pelo OpenMP.

Entretanto, ao se aumentar a quantidade de dados processados, como no caso dos 100 milhões de números, a diferença de desempenho entre OpenMP e MPI diminui. Isso se deve à escalabilidade do MPI, que, embora tenha um *overhead* inicial maior, distribui a carga de trabalho de maneira mais uniforme e aproveita melhor a capacidade total de processamento em ambientes com múltiplos nós. Essa característica torna o MPI mais eficiente à medida que o tamanho do problema aumenta, evidenciando que para problemas extremamente grandes, a escolha entre OpenMP e MPI deve considerar não apenas o tempo de execução, mas também a arquitetura do sistema e o padrão de comunicação necessário para a aplicação específica.

Tabela 2. Execução do algoritmo Monte Carlo

Iteração	1.000.000 de pontos			5.000.000 de pontos			10.000.000 de pontos		
	Serial	**OpenMP** (4 *cores*)	**MPI** (5 nós)	**Serial**	**OpenMP** (4 *cores*)	**MPI** (5 nós)	**Serial**	**OpenMP** (4 *cores*)	**MPI** (5 nós)
1	0,320	0,175	1,140	1,315	0,900	3,420	2,537	1,650	3,771
2	0,267	0,185	2,000	1,252	0,829	3,360	2,550	1,774	3,034
3	0,267	0,179	1,600	1,371	0,855	3,360	2,498	1,718	3,370
4	0,253	0,194	2,050	1,316	0,874	3,560	2,516	1,764	3,185
5	0,254	0,174	2,240	1,377	0,901	3,810	2,544	1,779	4,632
Média	**0,272**	**0,181**	**1,806**	**1,326**	**0,872**	**3,503**	**2,529**	**1,737**	**3,598**
Speedup	**--**	**1,5**	**0,2**	**--**	**1,52**	**0,4**	**--**	**1,46**	**0,7**

Seguindo a mesma metodologia utilizada para os testes com os algoritmos de Números Primos, conforme apresentado na Tabela 2, cada algoritmo foi executado cinco vezes e, assim, obtivemos a média dos tempos de execução e o *speedup* para as versões paralelas. Usou-se como parâmetro de entrada para as execuções os valores de 1, 5 e 10 milhões de pontos. As médias dos tempos de execução para a versão sequencial, alterando a quantidade de pontos de entrada, foram de 0,272s, 1,326s e 2,529s, respectivamente. Já na versão implementada em OpenMP, as médias dos tempos de execução foram de 0,181s, 0,872s e 1,737s, respectivamente, atingindo *speedup* máximo de 1,52x para 5 milhões de pontos. A versão implementada em MPI executou em 1,806s, 3,503s e 3,598s, respectivamente, com *speedup* máximo de 0,7x para a entrada de 10 milhões de pontos. Assim, podemos observar que o algoritmo implementado com OpenMP mostrou-se o mais eficiente para a execução do algoritmo de Monte Carlo, processando os parâmetros de entrada já citados.

Analisando os valores na Tabela 2, a versão sequencial do algoritmo de Monte Carlo apresenta tempos de execução que aumentam linearmente com o número de pontos. A implementação em OpenMP demonstra uma redução significativa no tempo de execução em relação à versão sequencial, especialmente para 1 e 5 milhões de pontos, com *speedup* máximo de 1,52x. Este ganho de

performance pode ser atribuído à eficiência do OpenMP em gerenciar *threads* e distribuir a carga de trabalho de maneira equilibrada. Por outro lado, a implementação em MPI não apresentou resultados favoráveis em comparação com a versão sequencial. Os tempos de execução para a versão MPI foram consistentemente maiores, com um *speedup* inferior a 1 para todas as entradas testadas, devido ao *overhead* significativo associado à troca de mensagens e sincronização entre processos.

Como já expresso anteriormente, é importante destacar que a versão paralela do algoritmo implementada com a biblioteca MPI foi menos eficiente que a versão sequencial em todos os casos simulados. Essa situação pode ser justificada pela quantidade de entradas a serem processadas e pelo *overhead* imposto pelo MPI para a troca de mensagens. O *overhead* da comunicação e sincronização entre processos pode ser significativo, especialmente para menores volumes de dados, resultando em uma *performance* inferior. Em cenários com menores volumes de dados, o tempo gasto na comunicação entre os processos MPI pode superar os benefícios da paralelização, tornando a execução sequencial mais eficiente. Então, a escolha entre MPI e uma implementação sequencial deve considerar a relação entre o *overhead* de comunicação e o tamanho do problema a ser resolvido, a fim de maximizar a eficiência computacional.

Diferentemente do algoritmo de Números Primos, os *speedups* obtidos nas versões paralelas dos algoritmos de Monte Carlo (OpenMP e MPI) foram menores, conforme observado nas Tabelas 1 e 2, indicando uma maior sensibilidade desse algoritmo a cargas de entrada pequenas. A paralelização no algoritmo de Monte Carlo só demonstra benefícios significativos quando as entradas são extremamente grandes, exigindo computação massiva. Esse comportamento ocorre porque, com entradas menores, o *overhead* de gerenciamento de *threads* em OpenMP e de troca de mensagens em MPI pode anular os ganhos de paralelização. Desta forma, fazer paralelismo do algoritmo de Monte Carlo com entradas de unidade de milhões não é interessante no ponto de vista computacional.

Enquanto o algoritmo de Números Primos mostra melhorias claras com a paralelização mesmo para entradas menores, o algoritmo de Monte Carlo requer um volume de dados substancialmente maior para que a paralelização resulte em uma eficiência computacional perceptível. Outro ponto a ser destacado é que o algoritmo de Monte Carlo verifica valores entre 0 e 1, tornando-o inadequado para calcular entradas na ordem dos milhões. Portanto, a eficiência da paralelização para algoritmos como o de Monte Carlo depende fortemente do tamanho da carga de trabalho, evidenciando a necessidade de considerar a natureza e o volume dos dados ao escolher a abordagem paralela mais adequada.

4. Conclusões e Trabalhos Futuros

Os resultados obtidos demonstram a eficácia das abordagens paralelas em melhorar o desempenho computacional dos algoritmos de Números Primos e Monte Carlo, especialmente em plataformas de *hardware* acessíveis como a Raspberry Pi. A implementação dos algoritmos sequenciais e paralelos revelou que a paralelização pode oferecer ganhos significativos em termos de tempo de execução, com a biblioteca OpenMP apresentando superioridade em cenários onde a comunicação entre núcleos é minimizada e a memória compartilhada é bem explorada. Por outro lado, a biblioteca MPI mostrou um desempenho competitivo para problemas maiores, apesar de ter um *overhead* de comunicação maior que impactou negativamente em entradas menores.

As análises destacaram que o algoritmo de Números Primos se beneficiou consideravelmente da paralelização, com um *speedup* máximo de 2,94x para a versão OpenMP e 2,43x para a versão MPI. Já o algoritmo de Monte Carlo apresentou resultados menos favoráveis com MPI, especialmente para menores volumes de dados, evidenciando que o *overhead* de comunicação pode superar os ganhos de paralelização para casos menores. O uso de OpenMP foi mais eficiente para entradas menores, enquanto o MPI mostrou vantagens em cenários com grandes volumes de dados.

Para trabalhos futuros, sugerem-se as seguintes ideias: (i) explorar a integração de técnicas híbridas que combinem OpenMP e MPI para otimizar o desempenho em diferentes tamanhos de problemas; (ii) investigação sobre técnicas avançadas de balanceamento de carga e otimização de comunicação em MPI; (iii) desenvolvimento de algoritmos paralelos que aproveitem arquiteturas emergentes e *hardware* especializado, como GPUs; (iv) estudos comparativos com outras bibliotecas de paralelismo e ambientes de execução, incluindo nuvens e *clusters* heterogêneos; (v) análise de escalabilidade e eficiência em cenários práticos, para o avanço das técnicas de computação paralela.

Agradecimentos

Os autores agradecem ao Instituto Federal de Rondônia (IFRO), pelo apoio financeiro concedido através dos Editais Nº 13/2023/REIT - PROPESP/IFRO e Nº 14/2023/REIT - PROPESP/IFRO, ambos PIBITI Ciclo 2023-2024, que proporcionaram a execução desta pesquisa.

Referências

Andrews, G. R. (2001) "Foundations of Multithreaded, Parallel, and Distributed Programming". Boston, Massachusetts, EUA: Addison-Wesley, 2nd edition, 664p.

Akel, A. (2024) "A Importância dos Números Primos – Unidades Imaginárias", disponível em https://medium.com/unidades-imaginárias/a-importância-dos-números-primos-1249a54cc57e. Acessado em 22/06/2024.

Bell, G., Gray, J. (2002) "What's Next in High-Performance Computing?". *In Communications of the ACM*, 45(2):91-95, February.

Chapman, B.; Jost, G.; Pas, R. V. D. (2007) "Using OpenMP: Portable Shared Memory Parallel Programming". Cambridge, Massachusetts, EUA: MIT Press, 1st edition, 353p.

Diaz, J.; Munoz-Caro, C.; Nino, A. (2012) "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era". *In IEEE Trans. on Parallel and Distributed Systems*, 23(8):1369-1386, August.

Gebali, F. (2011) "Algorithms and Parallel Computing". Wiley, 1st edition, 364p.

Ignácio, A. L. J.; Dias, W. R. A. (2023) "Análise do Desempenho Computacional de Algoritmos Paralelizados com OpenMP e MPI Executados em Raspberry Pi". *In Workshop de Iniciação Científica - Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, Porto Alegre, RS, Brasil, pp. 41-48.

Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B. (2011) "High Performance Computing using MPI and OpenMP on Multi-core Parallel Systems". *In Parallel Computing,* 37(9):562-575, September.

Lima, F. A.; Dias, W. R. A.; Moreno, E. D. (2020) "Implementação de um Cluster Embarcado usando a Plataforma Raspberry Pi". *In Escola Regional de Alto Desempenho do Rio de Janeiro (ERAD-RJ)*, Nova Iguaçu, RJ, Brasil, pp. 11-15.

Lima, F. A.; Moreno, E. D.; Dias, W. R. A. (2016) "Performance Analysis of a Low Cost Cluster with Parallel Applications and ARM Processors". *In IEEE Latin America Transactions*, 14(11):4591-4596, December.

Mpi, (2024) "A Message-Passing Interface Standard Version 2.1", disponível em www.mpi-forum.org/docs/mpi21-report.pdf. Acessado em 12/06/2024.

OpenMP, (2024) "The OpenMP API Specification for Parallel Programming", disponível em http://openmp.org/. Acessado em 10/06/2024.

Pacheco, P. S. (2011) "An Introduction to Parallel Programming". Morgan Kaufmann Publishers, 1st Ed., 370p.

Parhami, B. (2006) "Introduction to Parallel Processing - Algorithms and Architectures". New York, USA: Kluwer Academic Publishers, 1st edition, 532p.

Petersen, W. P., Arbenz, P. (2004) "Introduction to Parallel Computing". New York, EUA: Oxford University Press, 1st edition, 259p.

Silberschatz, A., Peter, G., Gagne, G. (2001) "Sistemas Operacionais - Conceitos e Aplicações". Campus, 8ª edição, 618p.

Souto, R. P., Ávila, R. B., Navaux, P. A. O., Py, M., Maillard, N., Diverio, T. A., Velho, H. F. de C., Stephany, S., Preto, A., Panetta, J., Rodrigues, E., Almeida, E. (2007) "Processing Mesoscale Climatology in a Grid Environment". *In Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*, Rio de Janeiro, RJ, Brazil, pp. 363-370.

Top500. (2024) "Top 500 Supercomputing Site", disponível em https://top500.org/. Acessado em 01/07/2024.

Trobec, R., Slivnik, B., Bulić, P., Robič, B. (2018) "Introduction to Parallel Computing - From Algorithms to Programming on State-of-the-Art Platforms". Berlim, Alemanha: Springer, 1st edition, 255p.

Xavier, C., Sachetto, R., Vieira, V., Santos, R. W. dos, JR, W. M. (2007) "Multi level Parallelism in the Computational Modeling of the Heart". *In Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Gramado, RS, Brazil, pp. 3-10.

Weisstein, E. W. (2024) "Monte Carlo Method. Wolfram Research Inc.", disponível em http://mathworld.wolfram.com/MonteCarloMethod.html. Acessado em 23/05/2024.