Análise do Desempenho Computacional de Algoritmos Paralelizados com OpenMP e MPI Executados em Raspberry Pi

Augusto Linhares Junqueira Ignácio[1], 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

{augustol.j.ignacio, wradias}@gmail.com

Abstract. *Parallel computing has been growing rapidly in terms of computational performance, having as its advent the multiprocessor and multicomputer architectural models. However, the availability of such architectures (hardware) is not enough if they are not properly exploited, that is, the use of parallel programming (software). Thus, this article presents the computational performance of the execution of the Prime Numbers algorithm, implemented in a sequential and parallel way, using the OpenMP and MPI libraries for parallelism, and executed on the Raspberry Pi platform. For the input of 5 million numbers, the speedups obtained for the OpenMP algorithm were 3x and 1.7x, compared to the sequential and MPI implementations, respectively.*

Resumo. *A computação paralela vem crescendo rapidamente em termos de desempenho computacional, tendo como seus adventos os modelos arquiteturais multiprocessador e multicomputador. Entretanto, não bastam estarem disponíveis tais arquiteturas (hardware), se estas não são devidamente exploradas, ou seja, o uso da programação paralela (software). Assim, este artigo apresenta o desempenho computacional da execução do algoritmo de Números Primos, implementado de forma sequencial e paralela, usando as bibliotecas OpenMP e MPI para o paralelismo, e executado em plataforma Raspberry Pi. Para a entrada de 5 milhões de números, os speedups obtidos para o algoritmo OpenMP foi de 3x e 1.7x, comparado com as implementações sequencial e MPI, respectivamente.*

1. Introdução

O uso de ambientes de Processamento de Alto Desempenho (PAD) tem sido recorrente para a execução de aplicações que exijam uma significativa capacidade de processamento de dados [Petersen e Arbenz, 2004]. São inúmeras as aplicações encontradas no cenário atual que fazem uso de PAD. Além disso, muitas outras estão em desenvolvimento, sob forma de pesquisa, especialmente nas áreas da saúde, biologia, engenharias, petróleo e gás, climatologia, física, química, geologia, computação, aeroespacial, automobilística [Xavier *et al.*, 2007]. Tais aplicações, sob o ponto de vista computacional, são diferentes entre si em termos de implementação, execução, recursos de software e recursos computacionais. A todas essas observações adiciona-se ainda a forma como os processos da aplicação são executados, uma vez que estes são dependentes de uma série de fatores, tais como: parâmetros de entrada e leitura de dados durante o fluxo de execução, ou seja, um processo pode ser executado dinamicamente.

Atualmente, observa-se um grande número de sistemas computacionais à disposição no mercado, com uma considerável gama de recursos que satisfaça as necessidades dos desenvolvedores de software [Panetta *et al.*, 2007]. Isto pode ser claramente visto quando se analisa a lista das 500 máquinas com maior capacidade de processamento do mundo (https://www.top500.org/), as quais são compostas por inúmeras unidades de processamento (processadores) interligadas, seja através de um barramento comum, seja através de redes especiais, as quais são destinadas a uma infinidade de aplicações [Top 500, 2023]. Geralmente, tais soluções estão baseadas no desenvolvimento de arquiteturas paralelas [Andrews, 2001]. O uso de máquinas de arquiteturas vetoriais, multiprocessadas, *multicore* e, atualmente, arquiteturas híbridas constituídas por CPU (*Central Processing Unit*) e GPU

(*Graphics Processing Unit*), tem sido algumas das alternativas. Tais tecnologias podem ainda ser combinadas através da formação de *clusters* e *grids*. Chega-se, dessa forma, a composição de plataformas com múltiplos níveis de paralelismo.

É possível constatar que a indústria da computação tem como objetivo melhorar cada vez mais o desempenho computacional através do emprego de diversas formas de paralelismo. Para tanto, hardwares paralelos estão presentes nos principais equipamentos ou plataformas computacionais atuais. Estações de trabalho, servidores, supercomputadores, ou até mesmo plataformas SoC (*System on Chip*) como a Raspberry Pi e outras, fazem parte desta exploração de paralelismo para prover maior desempenho em suas respectivas plataformas [Souto *et al.*, 2007].

Esta abordagem de baixo nível de paralelismo direciona o desenvolvedor para mecanismos de programação mais complexos e com pouca portabilidade. Somente um pequeno número de programadores encara este desafio, a grande maioria não se sujeita a trabalhar neste paradigma. Até mesmo programadores com mais experiência tentam ignorar a programação paralela, pelos problemas que esta abordagem necessita tratar, nos mais diversos ambientes de arquiteturas [Panetta *et al.*, 2007].

Os computadores em um contexto geral, já suportam *multithreading* e/ou *multicore*. A decomposição de tarefas e de dados é uma forma de abstrair a complexidade na exploração de paralelismo destes processadores. Sendo que o escalonamento dos *threads* ou processos é de responsabilidade do Sistema Operacional em realizar a distribuição da carga entre os elementos de processamento [Silberschatz *et al.*, 2001].

Ao mesmo tempo em que há um progresso no desenvolvimento de hardware, especialmente de arquiteturas paralelas, existe a necessidade de se oferecer recursos de programação compatíveis com os diferentes ambientes computacionais. Além disso, também é importante que haja mecanismos de programação capazes de integrar as diferentes arquiteturas paralelas existentes, simplificando assim o processo de programação. Desta forma, cria-se uma camada de abstração entre a aplicação em si e a sua plataforma de execução, podendo esta ser feita através de bibliotecas específicas para programação paralela, tais como: OpenMP (*Open Multi-Processing*) [OpenMP, 2023] e MPI (*Message Passing Interface*) [Mpi, 2023].

Portanto, neste artigo será apresentada a análise do desempenho computacional da execução do algoritmo de Números Primos, implementado de forma sequencial e paralela, usando as bibliotecas OpenMP e MPI para o paralelismo, e executado em um *cluster* formado por Raspberry Pi, modelo 4B, 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 do algoritmo implementado de forma sequencial e paralela e a Seção 4 finaliza com as conclusões e ideias para trabalhos futuros.

2. Contextualização

Um dos motivos pelos quais a evolução dos sistemas computacionais tem conseguido manter seu ritmo de crescimento nos últimos anos deve-se à exploração e a concepção de arquiteturas paralelas. Arquiteturas paralelas estão baseadas na utilização de múltiplas unidades de processamento. Atualmente estão sendo explorados diferentes níveis de paralelismo, que vão desde o núcleo de processamento das arquiteturas *multicore* e multiprocessadores, passando pelo uso de multicomputadores, através do uso de *clusters*, até a organização de um *grid* em escala mundial. Com isso, pode-se obter um desempenho mais elevado do que o estimado em arquiteturas mais simples.

No entanto, além das arquiteturas paralelas (hardware), também é necessário que as aplicações (*software*) sejam implementadas usando o paradigma de paralelismo, desta forma é possível explorar ainda mais os benefícios da Computação de Alto Desempenho ou HPC (*High Performance Computing*) e com isso, pode-se obter um desempenho computacional mais elevado do que o estimado em arquiteturas mais simples [Parhami, 2006]. Assim, através do desenvolvimento e execução de algoritmos paralelos é possível solucionar problemas com resoluções maiores em tempo de execução aceitável [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 vários elementos de processamento. Os elementos de processamento 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 de 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

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]. 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.

O OpenMP faz paralelismo explícito e é composto de 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* de computadores, a fim de otimizar ainda mais a utilização dos recursos disponíveis e diminuir o número de comunicações realizadas entre os diferentes nós [Pacheco, 2011]. 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.

2.2. Biblioteca MPI

MPI (*Message-Passing Interface*) [Mpi, 2022b] é a especificação de uma biblioteca de interface de troca de mensagens, criada a partir do consenso 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* (HPC) [Pacheco, 2011].

As implementações reais da biblioteca MPI diferem em qual versão e recursos do padrão MPI elas suportam. Atualmente, o MPI é executado em plataformas de memória distribuída, compartilhada e híbrido. Tradicionalmente, é implementado para as linguagens C, C++ e Fortran, porém há implementações menos comuns para as linguagens Java, Python e IDL. O modelo de programação permanece claramente um modelo de memória distribuída, independentemente da arquitetura física subjacente da máquina. Apesar de todas as facilidades do padrão de MPI, a identificação das zonas paralelas (áreas do código que podem ser executadas simultaneamente a outras tarefas) e o dimensionamento de carga entre os computadores participantes, ficam a cargo do desenvolvedor e devem ser explicitadas no programa. Assim, com o uso do MPI todo paralelismo é explícito, ou seja, o

desenvolvedor é exclusivamente o responsável por identificar corretamente o paralelismo e implementar algoritmos paralelos usando construtores da biblioteca no código [Trobec *et al.*, 2018]. Enfatizamos que o MPI é uma especificação, sendo assim, há disponíveis no mercado implementações livres e pagas. É possível relacionar no modelo pago o Intel MPI, HP MPI e o MATLab MPI. Já as implementações *Open Source* mais utilizadas são o OpenMPI e o MPICH [Lima *et at.*, 2020].

2.3. Métrica de Desempenho em Computação Paralela

Com a computação paralela, pode-se deixar levar pela errônea ideia de que ao dividir o trabalho em *n* unidades de processamento, os ganhos de desempenho também seriam proporcionalmente em *n* vezes. É importante ressaltar que isto nem sempre é possível. De Rose e Navaux (2002) relaciona os grandes desafios na área de Processamento de Alto Desempenho, que explicam o motivo pelo qual nem sempre é possível obter tal resultado. Os mesmos estão pautados em 4 aspectos, sendo eles: arquiteturas paralelas, gerenciamento das máquinas, linguagens mais expressivas que consigam extrair o máximo de paralelismo e finalmente uma maior concorrência nos algoritmos.

Para auxiliar a mensurar os ganhos ao paralelizar algoritmos seriais, é comumente utilizada a métrica denominada de *Speedup*. Em computação paralela, o *Speedup* é utilizado para conhecer o quanto um algoritmo paralelo é mais rápido que seu correspondente sequencial. O *Speedup* pode ser obtido pela seguinte equação:

$$Speedup = \frac{Tempo\ Serial}{Tempo\ Paralelo}$$

2.4. Algoritmo do Cálculo de Números Primos

Os números primos assumem um papel crucial nos sistemas de criptografia contemporâneos, desempenhando um papel fundamental na garantia da segurança em uma gama diversificada de transações, incluindo as financeiras. Esta importância reside nas características particulares desses números, que proporcionam uma base sólida para a segurança dos dados através de complexos mecanismos de fatoração. Embora encontrar números primos relativamente grandes não seja uma tarefa excepcionalmente árdua, o processo inverso de desdobrar esses números em seus fatores primos é notavelmente desafiador. A distinção é clara ao comparar a simplicidade de reconhecer que 35 é igual a (7 x 5) com a complexidade de decompor 2.244.354 em fatores primos (2 x 3 x 7 x 53437). A dificuldade cresce exponencialmente ao considerar números com cinquenta dígitos, chegando ao ponto em que um supercomputador pode demandar milhões de anos para resolver um problema de fatoração de 256 *bits* [Akel, 2023].

Diante dessa complexidade, este artigo se concentra em apresentar implementações do algoritmo de Cálculo de Números Primos nas formas sequencial e paralela. Pois o objetivo é avaliar a quantidade de números primos existentes em um intervalo extenso, de 1 a 5 milhões. Para a execução destes testes, foram exploradas duas plataformas: a plataforma Raspberry Pi, onde avaliou-se as versões sequencial e paralela (implementadas com a biblioteca OpenMP), e o *ClusterPi*, onde executou-se a versão paralela implementada com a biblioteca MPI.

Ao adentrar no campo da programação paralela, esta pesquisa proporciona *insights* sobre como as implementações sequencial e paralela se comportam diante da tarefa desafiadora de calcular a quantidade de números primos em intervalos de grande magnitude. Ao explorar a execução destes algoritmos, é possível obter uma compreensão mais sólida da capacidade de processamento de apenas uma plataforma Raspberry Pi (modelo 4B) e do *ClusterPi*, além de oferecer um panorama de como diferentes abordagens de programação podem influenciar o desempenho.

Desta forma, a análise comparativa entre as implementações e plataformas fornecerá não apenas uma compreensão mais aprofundada da eficiência de cada abordagem, mas também abrirá portas para pesquisas futuras no campo da otimização de cálculos complexos e a adaptação de algoritmos para uma variedade de contextos computacionais.

3. Resultados e Discussão

Neste artigo, foi realizado uma análise comparativa dos tempos de execução de três distintas implementações do algoritmo de Cálculo de Números Primos. Estas implementações incluem uma

abordagem sequencial, bem como duas variantes executadas em paralelo, fazendo uso das bibliotecas OpenMP e MPI. As avaliações foram conduzidas dentro de um intervalo contendo até 5 milhões de números, proporcionando uma visão abrangente das eficiências relativas desses métodos quando executados paralelamente em arquiteturas multiprocessador (OpenMP) ou multicomputador (MPI).

As diferentes implementações foram codificadas em linguagem C, sendo posteriormente executadas em plataformas Raspberry Pi, mais especificamente no modelo 4B equipado com 2GB de memória RAM. As execuções foram realizadas no ambiente com o Sistema Operacional Linux, utilizando a distribuição *Raspbian*. A compilação do código foi realizada através do *GCC* (*Gnu Compiler Collection*) na versão 8.3.

É importante destacar que a Raspberry Pi 4B possui uma arquitetura *multicore*, composta por quatro núcleos, o que viabiliza a eficácia da implementação que faz uso da biblioteca OpenMP. Essa característica permite uma otimização notável no desempenho, explorando a capacidade de processamento paralelo proporcionada pelos núcleos.

Para o processo de execução do algoritmo implementado com a biblioteca MPI, foi utilizado um *ClusterPi* homogêneo formado por cinco nós (ver Figura 1). Nesse arranjo, um dos nós foi designado como o nó mestre, encarregado da distribuição das tarefas, enquanto os outros quatro atuaram como nós escravos, desempenhando a execução das tarefas específicas.

Figura 1. Estrutura do ClusterPi

Para assegurar uma comunicação eficiente entre os nós que compuseram o *ClusterPi*, foi usado o protocolo SSH (*Secure Socket Shell*). Além disso, para a interconexão dos nós, foi utilizado um *Switch Gigabit* composto por oito portas, da marca TPLink e modelo TL-SG108PE, contribuindo para a criação de uma rede ágil e robusta que facilitou a troca de mensagens entre os diferentes nós do *Cluster*, a Figura 1 apresenta a estrutura arquitetura montada para a realização dos testes.

Para os testes, foi estabelecido um intervalo numérico específico e conduzido cada implementação por um total de 10 execuções, permitindo derivar a média do tempo necessário para completar o algoritmo. Inicialmente, foram realizados testes com um intervalo compreendendo 1 até 1 milhão de números, visando analisar o Cálculo dos Números Primos contidos nesse intervalo.

Os resultados, mostrados na Tabela 1 (com 1 milhão de entradas), refletem que a média da execução sequencial do algoritmo foi concluída em 2 minutos e 36 segundos, equivalente há 156 segundos. Ao executar no *ClusterPi* (com os 4 nós de processamento, mais o nó mestre) o algoritmo implementado em MPI, a média do tempo de processamento obtida foi de 1 minuto e 14 segundos, equivalendo a 74 segundos. Obtendo um aumento de velocidade, *speedup*, de aproximadamente 2x em relação à execução da implementação sequencial.

No entanto, ao analisar a execução do algoritmo paralelo com a biblioteca OpenMP, foi observado uma melhoria ainda mais substancial no tempo de execução. Em média, computação foi realizada em meros 40 segundos, resultando em um *speedup* notável de quase 4x, quando comparado à execução sequencial e 1,8x em relação à implementação em MPI.

Esses resultados nos conduzem à inferência de que a implementação usando OpenMP demonstrou maior eficiência do que a alternativa em MPI, principalmente devido ao tamanho da entrada de dados (*input*) submetida ao processamento e consequentemente ao *overhead* gerado pela troca de mensagens entre os nós, característica intrínseca à implementação MPI.

Os mesmos experimentos foram repetidos, entretanto, desta vez foi focalizado aumentar o intervalo de entrada para o algoritmo, abrangendo 5 milhões de números. Os resultados desses testes proporcionaram uma análise mais ampla do desempenho das diferentes implementações em escalas mais substanciais.

Ao analisar os resultados apresentados na Tabela 1 (com 5 milhões de entradas), é possível verificar que a execução sequencial do algoritmo foi concluída em 58 minutos para o intervalo de 1 até 5 milhões de números. Ao utilizar a implementação em MPI, o tempo de execução foi consideravelmente reduzido, sendo concluído em 33 minutos, resultando um *speedup* de aproximadamente 1,7x em relação à execução da implementação sequencial.

Porém, a implementação utilizando a biblioteca OpenMP se destacou novamente, finalizando o cálculo dos Números Primos, no intervalo de 1 até 5 milhões de números, em apenas 19 minutos. Este resultado demonstra uma vantagem notável, superando em mais de 3x o desempenho da implementação sequencial e 1,7x a implementação usando a biblioteca MPI.

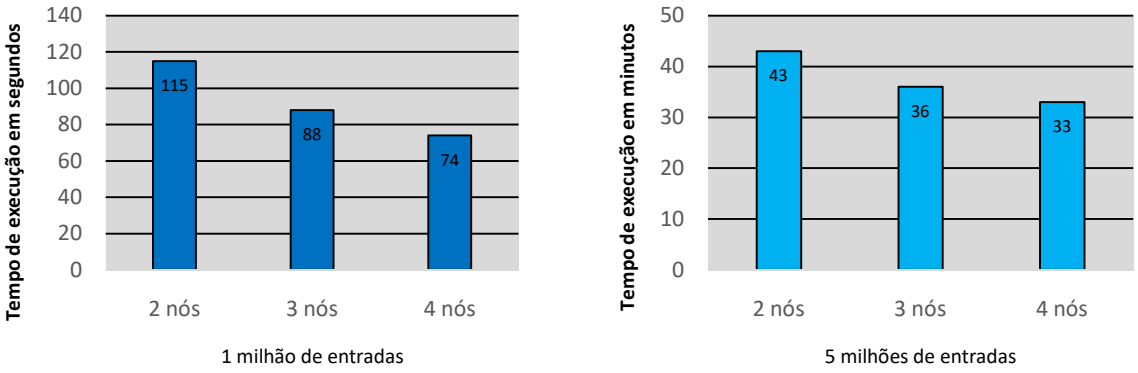
Tabela 1. Execução do cálculo de Números Primos com 1 e 5 milhões de entradas

	Entrada: 1.000.000 de números Tempo em segundos			**Entrada: 5.000.000 de números** Tempo em minutos		
Execuções	**Sequencial**	**OpenMP (4 *threads*)**	**MPI (5 nós)**	**Sequencial**	**OpenMP (4 *threads*)**	**MPI (5 nós)**
1	156,902	39,333	67,610	58,33	19,46	32,45
2	156,905	39,372	68,191	58,57	18,41	32,37
3	156,903	39,819	70,284	60,07	20,09	34,43
4	156,903	39,327	86,578	58,34	19,23	34,14
5	156,896	39,337	74,846	57,42	18,45	34,27
6	156,893	39,362	74,378	58,33	18,59	34,57
7	156,884	39,657	79,954	58,45	19,01	33,59
8	156,902	41,049	75,894	60,13	19,45	34,25
9	156,856	41,795	72,469	58,25	20,13	34,30
10	156,900	41,986	70,992	58,25	19,15	34,34
Média	**156,894**	**40,104**	**74,120**	**58,61**	**19,19**	**33,87**

Estes resultados reforçam a inferência anteriormente mencionada sobre o impacto do tamanho da entrada para o processamento e o *overhead* gerado pela troca de mensagens na implementação em MPI. A eficiência da abordagem OpenMP, que utiliza a arquitetura *multicore* da Raspberry Pi, mais uma vez se destaca quando confrontada com o desafio de lidar com intervalos de números ainda maiores. Isso sugere que, à medida que a escala do problema aumenta, a implementação em OpenMP continua a apresentar vantagens significativas em termos de desempenho, em comparação com as implementações sequencial e MPI. Portanto, escolher a abordagem de programação paralela apropriada depende não apenas das características do problema, mas também do tamanho da entrada e das peculiaridades da arquitetura de hardware.

Continuando com os testes, foi realizada análise da implementação em MPI, variando o número de nós do *Cluster*. A análise levou em consideração a escalabilidade dos nós escravos em termos de processamento. É possível observar na Figura 2(a), que o *Cluster* formado com 4 nós, executou o algoritmo de Números Primos (com 1 milhão de entradas) em 41 e 14 segundos mais

rápido, quando comparado com a formação do *Cluster* com 2 e 3 nós de processamento, respectivamente. Já com a entrada de 5 milhões de números, a diferença no tempo de execução foi de 10 e 3 minutos, quando comparado com o *Cluster* formado com 2 e 3 nós de processamento, respectivamente (ver Figura 2(b)).



(a) (b)

Figura 2. Tempo de Execução da Implementação em MPI no ClusterPi

4. Conclusões e Trabalhos Futuros

Neste estudo, foi realizado uma análise comparando o desempenho de três implementações diferentes do algoritmo de Cálculo de Números Primos. Estas variantes incluíram uma abordagem sequencial, bem como duas abordagens paralelas com as bibliotecas OpenMP e MPI. As avaliações foram conduzidas dentro de um intervalo de 1 até 5 milhões de números. Para a execução desses testes, foram escolhidas plataformas Raspberry Pi modelo 4B com 2GB de RAM, explorando sua arquitetura *multicore* de quatro núcleos. A implementação MPI foi testada em um *ClusterPi* homogêneo com cinco nós, enquanto a comunicação entre eles se deu através do protocolo SSH, apoiada por um Switch *Gigabit* de 8 portas.

Os resultados das avaliações evidenciaram que a implementação paralela com a biblioteca OpenMP superou as alternativas sequencial e MPI. Nos testes realizados com um intervalo de 1 milhão de números, a abordagem OpenMP apresentou um *speedup* de quase 4x em relação à execução sequencial e 1,8x em relação à implementação MPI. Ao expandir o intervalo para 5 milhões de números, a eficiência da implementação OpenMP permaneceu proeminente, culminando em um *speedup* de mais de 3x comparado à execução sequencial e 1,7x em relação à implementação MPI. Essa superioridade da abordagem OpenMP pode ser atribuída ao tamanho da entrada de processamento e ao overhead gerado pela troca de mensagens inerente à implementação MPI.

Como ideias para trabalhos futuros, sugerem-se: (i) investigar como as diferentes implementações se comportam em escalas ainda maiores, explorando intervalos numéricos de magnitude superior e analisando se as tendências de eficiência observadas se mantêm; (ii) explorar configurações alternativas de parâmetros na biblioteca MPI, como o número de nós e a alocação de tarefas (*threads*), para otimizar ainda mais o desempenho; (iii) estender os testes para incluir uma variedade de arquiteturas de hardware, como GPUs ou sistemas distribuídos de maior escala, para compreender como as implementações se comportam em contextos mais diversos; (iv) explorar a utilização de outras bibliotecas paralelas e/ou técnicas de programação, a fim de avaliar como diferentes abordagens impactam o desempenho do algoritmo; (v) aprofundar a investigação sobre o *overhead* gerado pela comunicação usando a biblioteca MPI, identificando possíveis estratégias de otimização no algoritmo para minimizar esse impacto; (vi) investigar o desempenho das implementações em algoritmos de Cálculo de Números Primos mais complexos, que possam fornecer *insights* adicionais sobre a escalabilidade e eficácia das abordagens paralelas e (vii) implementar as diferentes abordagens em outras linguagens de programação, como Python, a fim de avaliar o impacto na eficiência e desempenho, além de comparar com as implementações em C.

Agradecimentos

Os autores agradecem ao Instituto Federal de Rondônia (IFRO), pelo apoio financeiro concedido através dos Editais Nº 9/2022/REIT - PROPESP/IFRO e Nº 11/2022/REIT - PROPESP/IFRO, ambos PIBITI Ciclo 2022-2023, 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. (2023) "A Importância dos Números Primos – Unidades Imaginárias", disponível em https://medium.com/unidades-imaginárias/a-importância-dos-números-primos-1249a54cc57e. Acessado em 22 de julho de 2023.

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

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

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, 2016.

Mpi, (2023) "OpenMPI: Open Source High Performance Computing", disponível em http://www.open-mpi.org/. Acessado em 20 de julho de 2023.

Mpi, (2023) "A Message-Passing Interface Standard Version 2.1", disponível em www.mpi-forum.org/docs/mpi21-report.pdf. Acessado em 25 de junho de 2023.

OpenMP, (2023) "The OpenMP API Specification for Parallel Programming", disponível em http://openmp.org/. Acessado em 17 de julho de 2023.

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

Panetta, J., Filho, P. R. P. de S., Filho, C. A. da C., Motta, F. M. R. da, Pinheiro, S. S., Junior, I. P., Rosa, A. L. R., Monnerat, L. R., Carneiro, L. T., Albrecht, C. H. B. de. (2007) "Computational Characteristics of Production Seismic Migration an its Performance on Novel Processor Architectures". *In Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Gramado, RS, Brazil, pp. 11-18.

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.

Rose, C. D., Navaux, P. (2002) "Fundamentos de Processamento de Alto Desempenho". *Anais da II Escola Regional de Alto Desempenho (ERAD-RS)*, São Leopoldo, RS, pp. 3-29.

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. (2023) "Top 500 Supercomputing Site", disponível em https://top500.org/. Acessado em 01 de agosto de 2023.

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.